User's Manual, V1.1, Apr. 2002

TC11IB

System Units

32-Bit Single-Chip Microcontroller

Microcontrollers

Infineon technologies

Never stop thinking.

Edition 2002-04

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany

Attention please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TC11IB

System Units

32-Bit Single-Chip Microcontroller

Microcontrollers

Never stop thinking.

TC11IB System Units User's Manual
Revision History: 2002-04 V1.1

Previous Version: V1.0 2002-03

Page	Subjects (major changes since last revision)
1-8	Max.junction temperature is removed
1-8,1-27	Package name is corrected
1-43	Notes to pins V_{LMUREF} and V_{COMREF} are added
1-46	V_{ss} pins are corrected
2-17,18-6, 22-10	Reset values of register PC, FFI_CON are corrected
7-3,7-5, 7-6, 22-4	Address ranges of local memories are corrected
8-9,22-56	Reset value of PMU_CON2 register is corrected
9-6,22-55	Reset value of DMU_CON register is corrected
Chapter 14	Some language errors are corrected
14-2	Address alignment feature is added
14-10	ALE signal description is modified
14-20	AALIGN parameter description is added
14-23	Address alignment during bus access is added
14-86,14-93	AALIGN bit is added to register BUSCONx(x = 0~6) and register EMUBC
14-87,14-94	AALIGN bit function description is added
22-38	PCI_CS2_ID register is added
22-48	CPU_ID address is corrected

We Listen to Your Comments
Any information within this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

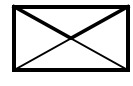


Table of Contents

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

Table of Contents **Page**

1 Introduction

This User's Manual describes the Infineon TC11IB, a 32-bit microcontroller DSP for industrial communication applications, based on the Infineon TriCore Architecture.

1.1 About this Document

This document is designed to be read primarily by design engineers and software engineers who need a detailed description of the interactions of the TC11IB functional units, registers, instructions, and exceptions.

This TC11IB User's Manual describes the features of the TC11IB with respect to the TriCore Architecture. Because TC11IB directly implements TriCore architectural functions, this manual simply refers to those functions as features of the TC11IB. In all cases where this manual describes a TC11IB feature without referring to the TriCore Architecture, this means that the TC11IB is a direct embodiment the TriCore Architecture.

Since TC11IB implements a subset of TriCore architectural features, this manual describes the TC11IB implementation, and then describes how it differs from the TriCore Architecture. For example, the TriCore Architecture specifies up-to four Memory Protection Register Sets, the TC11IB implements but two. Such differences between the TC11IB and the TriCore Architecture are documented in the text covering each such subject.

1.1.1 Related Documentations

A complete description of the TriCore architecture is found in the document titled "TriCore Architecture Manual". The architecture of the TC11IB is described separately this way because of the configurable nature of the TriCore specification: different versions of the architecture may contain a different mix of systems components. The TriCore architecture, however, remains constant across all derivative designs in order to preserve compatibility.

Additionally to this "TC11IB System Units User's Manual", a second document, the "TC11IB Peripheral Units User's Manual", is available. These two User's Manuals together with the "TriCore Architecture Manual" are required for the understanding the complete TC11IB microcontroller functionality.

Implementation-specific details such as electrical characteristics and timing parameters of the TC11IB can be found in the "TC11IB Data Sheet".

1.1.2 Textual Conventions

This document uses the following textual conventions for named components of the TC11IB:

- Functional units of the TC11IB are given in plain UPPER CASE. For example: “The EBU provides an interface to external peripherals”.
- Pins using negative logic are indicated by an overbar. For example: “The $\overline{\text{BYPASS}}$ pin is latched with the rising edge of the $\overline{\text{PORST}}$ pin”.
- Bit fields and bits in registers are in general referenced as “Register name.Bit field” or “Register name.Bit”. For example: “The Current CPU Priority Number bit field ICR.CCPN is cleared”. Most of the register names contain a module name prefix, separated by a underscore character “_” from the real register name (for example, “ASC_CON”, where “ASC” is the module name prefix, and “CON” is the real register name). In chapters describing peripheral modules the real register name is referenced also as kernel register name.
- Variables used to describe sets of processing units or registers appear in mixed-case font. For example, register name “MSGCFGn” refers to multiple “MSGCFG” registers with variable n. The bounds of the variables are always given where the register expression is first used (for example, “n = 31 - 0”), and is repeated as needed in the rest of the text.
- The default radix is decimal. Hexadecimal constants are suffixed with a subscript letter “H”, as in 100_H. Binary constants are suffixed with a subscript letter “B”, as in: 111_B.
- When the extent of register fields, groups of signals, or groups of pins are collectively named in the body of the document, they are given as “NAME[A:B]“, which defines a range for the named group from B to A. Individual bits, signals, or pins are given as “NAME[C]” where the range of the variable C is given in the text. For example: CLKSEL[2:0], and TOS[0].
- Units are abbreviated as follows:
  - **MHz** = Megahertz
  - **µs** = Microseconds
  - **kBaud, kBit** = 1000 characters/bits per second
  - **MBaud, MBit** = 1,000,000 characters/bits per second
  - **KByte** = 1024 bytes of memory
  - **MByte** = 1048576 bytes of memory

  In general, the *k* prefix scales a unit by 1000 whereas the K prefix scales a unit by 1024. Hence, the KByte unit scales the expression preceding it by 1024. The kBaud unit scales the expression preceding it by 1000. The M prefix scales by 1,000,000 or 1048576, and µ scales by .000001. For example, 1 KByte is 1024 bytes, 1 MByte is 1024×1024 bytes, 1 kBaud/kBit are 1000 characters/bits per second, 1 MBaud/MBit are 1000000 characters/bits per second, and 1 MHz is 1,000,000 Hz.
- Data format quantities are defined as follows:
  - **Byte** = 8-bit quantity
  - **Half-word** = 16-bit quantity
  - **Word** = 32-bit quantity
  - **Double-word** = 64-bit quantity

1.1.3 Reserved, Undefined, and Unimplemented Terminology

In tables where register bit fields are defined, the following conventions are used to indicate undefined and unimplemented function. Further, types of bits and bit fields are defined using the abbreviations as shown in **Table 1-1**.

Table 1-1 Bit Function Terminology

Function of Bits	Description
Unimplemented	Register bit fields named **0** indicate unimplemented functions with the following behavior. – Reading these bit fields returns 0. – Writing these bit fields has no effect. These bit fields are reserved. When writing, software should always set such bit fields to 0 in order to preserve compatibility with future products.
Undefined	Certain bit combinations in a bit field can be labeled “Reserved”, indicating that the behavior of the TC11IB is undefined for that combination of bits. Setting the register to undefined bit combinations may lead to unpredictable results. Such bit combinations are reserved. When writing, software must always set such bit fields to legal values as given in the tables.
rw	The bit or bit field can be read and written.
r	The bit or bit field can only be read (read-only).
w	The bit or bit field can only be written (write-only).
h	The bit or bit field can also be modified by hardware (such as a status bit). This symbol can be combined with ‘rw’ or ‘r’ bits to ‘rwh’ and ‘rh’ bits.

1.1.4 Register Access Modes

Read and write access to registers and memory locations are sometimes restricted. In memory and register access tables, the following terms are used.

Table 1-2 Access Terms

Symbol	Description
U	Access permitted in User Mode 0 or 1.
SV	Access permitted in Supervisor Mode.
R	Read-only register.
32	Only 32-bit word accesses are permitted to that register/address range.

Table 1-2 Access Terms (cont'd)

Symbol	Description
E	Endinit protected register/address.
PW	Password protected register/address.
NC	No change, indicated register is not changed.
BE	Indicates that an access to this address range generates a Bus Error.
nBE	Indicates that no Bus Error is generated when accessing this address range, even though it is either an access to an undefined address or the access does not follow the given rules.
nE	Indicates that no Error is generated when accessing this address or address range, even though the access is to an undefined address or address range. True for CPU accesses (MTCR/MFCR) to undefined addresses in the CSFR range.
X	Undefined value or bit.

1.1.5 Abbreviations

The following acronyms and termini are used within this document:

ADC	Analog-to-Digital Converter
AGPR	Address General Purpose Register
ALE	Address Latch Enable
ALU	Arithmetic and Logic Unit
ASC	Asynchronous/Synchronous Serial Controller
BCU	Bus Control Unit
CISC	Complex Instruction Set Computing
CPS	CPU Slave Interface Registers
CPU	Central Processing Unit
CSFR	Core Special Function Registers
DGPR	Data General Purpose Register
DMU	Data Memory Unit
DRAM	Dynamic Random Access Memory
EBU	External Bus Unit
FFI	FPI to FPI Interface
FPI	Flexible Peripheral Interconnect (Bus)
GPR	General Purpose Register
GPTU	General Purpose Timer Unit
ICACHE	Instruction Cache
I/O	Input / Output
LFI	LMB to FPI Interface
LMB	Local Memory Bus

LMU	Local Memory Unit
MMCI	MultiMediaCard Interface
MMU	Memory Management Unit
NMI	Non-Maskable Interrupt
OCDS	On-Chip Debug Support
OVRAM	Code Overlay Memory
PCP	Peripheral Control Processor
PMU	Program Memory Unit
PLL	Phase Locked Loop
PCODE	PCP Code Memory
PMU	Program Memory Unit
PRAM	PCP Parameter RAM
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
SCU	System Control Unit
SFR	Special Function Register
SPRAM	Scratch-Pad Code Memory
SRAM	Static Data Memory
SSC	Synchronous Serial Controller
STM	System Timer
WDT	Watchdog Timer

1.2 System Architecture Features of the TC11IB

The TC11IB combines three powerful technologies within one silicon die, achieving new levels of power, speed, and economy for embedded applications:

- Reduced Instruction Set Computing (RISC) processor architecture
- Digital signal processing (DSP) operations and addressing modes
- On-chip memories and peripherals

DSP operations and addressing modes provide the computational power necessary to efficiently analyze complex real-world signals. The RISC load/store architecture provides high computational bandwidth with low system cost. On-chip memory and peripherals are designed to support even the most demanding high-bandwidth real-time embedded control-systems tasks.

Additional high-level features of the TC11IB include:

- Program Memory Unit — instruction memory and instruction cache
- Data Memory Unit — data memory and data cache
- Embedded DRAM — 12Mbit code / data DRAM memory.
- Memory Management Unit — Windows CE compliant.
- Two FPI buses — flexible interconnection with peripherals with different performance.
- One LMB Bus — Optimized for memory access speed.
- Serial communication interfaces — flexible synchronous and asynchronous modes
- Multiple industrial communication interfaces — PCI v2.2, Fast Ethernet Interfaces.
- MultiMediaCard Interface
- Peripheral Control Processor — DMA operations and interrupt servicing
- General purpose timers
- On-chip debugging and emulation facilities
- Flexible interconnections to external components
- Flexible power-management

The TC11IB is a high performance microcontroller with TriCore CPU, program and data memories, buses, bus arbitration, an interrupt controller, a peripheral control processor, several on-chip peripherals, an external bus interface, and two industrial communication interfaces. Except meeting true real-time, deterministic operation features of Industrial controllers, TC11IB is designed as a platform for new industrial control systems that are migrating from specialized interface and communications architectures to standards pioneered in the computer industry, such as PCI and Ethernet communications, standard PC-type memories, and real-time operating systems such as Microsoft's Windows™ CE. To meet the needs of the most demanding embedded control systems applications, the competing issues of price/performance, real-time responsiveness, computational power, data bandwidth, and power consumption are key design elements.

The TC11IB offers several versatile on-chip peripheral units such as serial controllers, timer units, PCI and Fast Ethernet. Within the TC11IB, all these peripheral units are connected to the TriCore CPU/system via the Slow Flexible Peripheral Interconnect Bus (S_FPI) and/or Fast Flexible Peripheral Interconnect Bus (F_FPI) and Local Memory

Bus (LMB). Special I/O lines and Several I/O lines on the TC11IB ports are reserved for these peripheral units to communicate with the external world.

High Performance 32-Bit CPU

- 32-bit architecture with 4 GBytes unified data, program, and input/output address space
- Fast automatic context-switch
- Multiply-accumulate unit
- Saturating integer arithmetic
- Fast response local memory bus
- Two high performance on-chip peripheral buses (S_FPI Bus and F_FPI Bus)
- Register based design with multiple variable register banks
- Bit handling
- Packed data operations
- Zero overhead loop
- Precise exceptions
- Flexible power management

Instruction Set with High Efficiency

- 16/32-bit instructions for reduced code size
- Data types include: Boolean, array of bits, character, signed and unsigned integer, integer with saturation, signed fraction, double word integers, and IEEE-754 single precision floating-point
- Data formats include: Bit, 8-bit byte, 16-bit half word, 32-bit word, and 64-bit double word data formats
- Powerful instruction set
- Flexible and efficient addressing mode for high code density

External Bus Interface

- Programmable external bus interface for low cost system implementation (Intel-style and Motorola-style device/peripheral support)
- Glueless interface to a wide selection of external memories (ROM, EPROM, SRAM, SDRAM, Burst Flash and PC 100 SDRAM)
- 8/16/32 bit data transfer
- Support for big and little endian byte ordering at bus interface
- Flexible address generation and access timing

Integrated On-Chip Memory

- Main core code memory
  - 24 KBytes Scratch-pad RAM (SPRAM)
  - 8 KByte Instruction Cache (ICache) for external code memory accesses

- Main core data memory
  - 24 KBytes data memory (SRAM)
  - 8 KBytes Data Cache memory
- Local code/data memory
  - 512 KBytes LMU eDRAM
  - 1 Mbytes ComDRAM
- 16 KBytes boot ROM (BROM)
- Peripheral Control Processor memory
  - 16 KBytes volatile code memory (PCODE)
  - 4 KBytes volatile parameter memory (PRAM)

Interrupt System

- 110 Service Request Nodes (SRNs)
- Flexible interrupt prioritizing scheme with 256 interrupt priority levels
- Fast interrupt response
- Service requests are serviced either by CPU (= interrupt) or by PCP

Peripheral Control Processor (PCP)

- Data move between any two memory or I/O locations
- Data move until predefined limit reached supported
- Read - Modify - Write capabilities
- Full computation capabilities including basic MUL/DIV
  - Read/move data and accumulate it to previously read data
  - Read two data values and perform arithmetic or logically operation and store result
  - Bit handling capabilities (testing, setting, clearing)
  - Flow control instructions (conditional/unconditional jumps, breakpoint)

I/O Lines With Individual Bit Addressability

- Push/pull or selectable open drain output mode
- TTL input thresholds
- Fixed Pull-up/Pull-down devices

Plastic Ball Grid Array (P-BGA) Package

- The TC11IB is packaged in a P-BGA-388 package

Temperature Ranges

- Ambient temperature: -25 °C to +85 °C

System Clock Frequency

- Maximum System Clock Frequency: 96MHz

Complete Development Support

A variety of software and hardware development tools for the 32-bit microcontroller TC11IB is available from experienced international tool suppliers. The development environment for the Infineon 32-bit microcontroller includes the following tools:

- Embedded Development Environment for TriCore Products
- The TC11IB On-chip Debug Support (OCDS) provides a JTAG port for communication between external hardware and the system.
- The Flexible Peripheral Interconnect Buses (F_FPI Bus and S_FPI Bus) for on-chip interconnections and the FPI Bus control units (BCU0 and BCU1).
- The System Timer (STM) with high-precision, long-range timing capabilities.
- The TC11IB includes a power management system, a watchdog timer as well as a reset logic.

1.3 Block Diagram

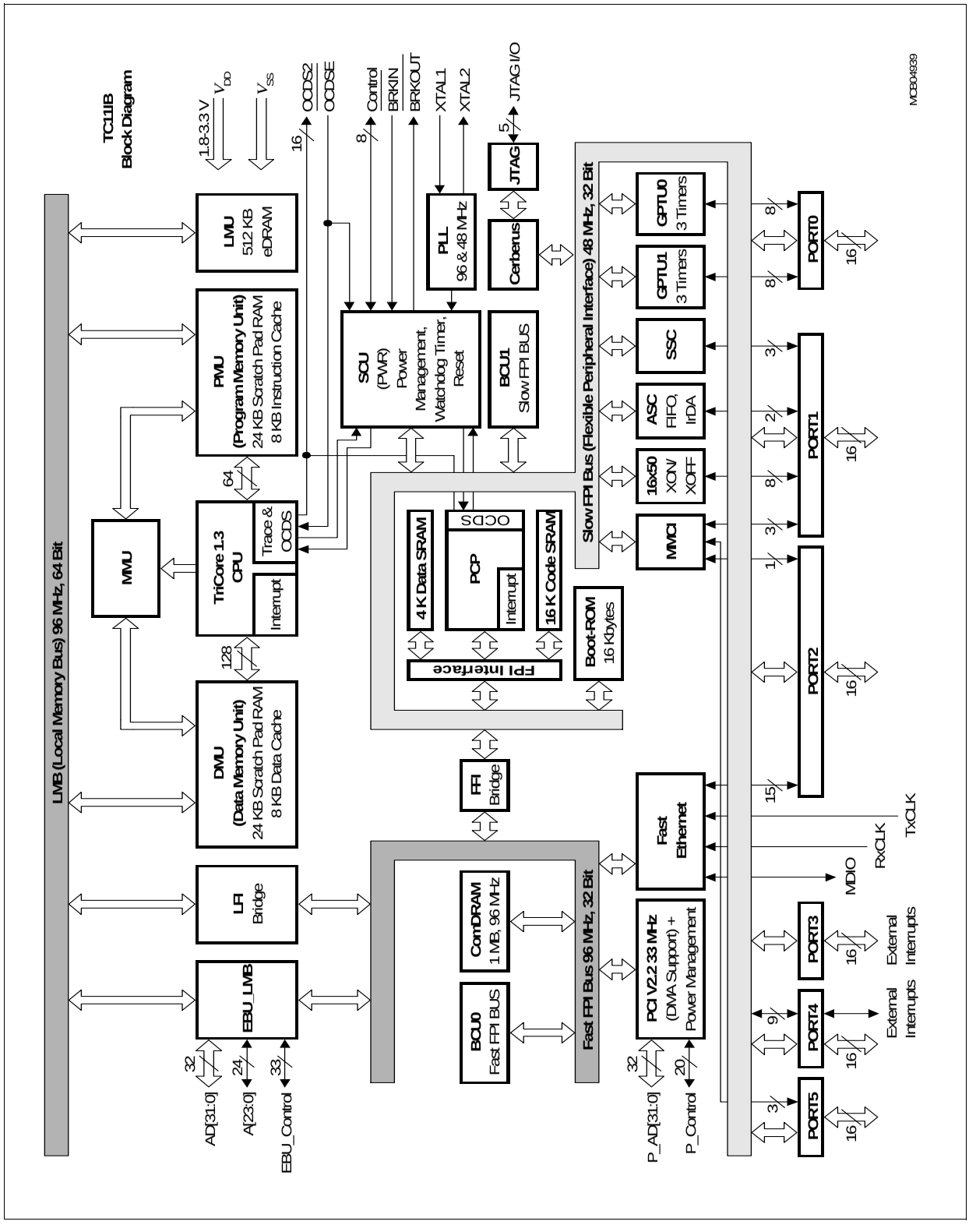


Figure 1-1 TC11IB Block Diagram

1.4 On-Chip Peripheral Units of the TC11IB

The following peripherals are all described in detail in the “TC11IB Peripheral Units User’s Manual”:

- One Asynchronous/Synchronous Serial Channels with baud rate generator, parity, framing, and overrun error detection
- One High Speed Synchronous Serial Channels with programmable data length and shift direction
- One Asynchronous Serial Interface with programmable XON/XOFF characters software flow control and hardware flow control
- Two Multifunctional General Purpose Timer Units with three 32-bit timer/counter each
- One MultiMediaCard Interface with pointer based data transfer and CRC protection
- One Ethernet Controller with Media Access Controller (MAC) fully compliant with IEEE 802.3 and Media Independent Interface (MII)
- One PCI Interface with PCI command support and PCI power management

The next sections within this chapter provide an overview of these peripheral units

Note: Additionally to the “TC11IB System Units User’s Manual”, a 2nd document, the “TC11IB Peripheral Units User’s Manual”, is available. These two User’s Manuals together with the “TriCore Architecture Manual” are required for the understanding the complete TC11IB microcontroller functionality.

1.4.1 Serial Interfaces

The TC11IB includes three serial peripheral interface units:

- Asynchronous/Synchronous Serial Interface (ASC)
- High-Speed Synchronous Serial Interface (SSC)
- Asynchronous Serial Interface (16X50)

1.4.1.1 Asynchronous/Synchronous Serial Interface

Figure 1-2 shows a global view of the functional blocks of the Asynchronous/ Synchronous Serial interface ASC.

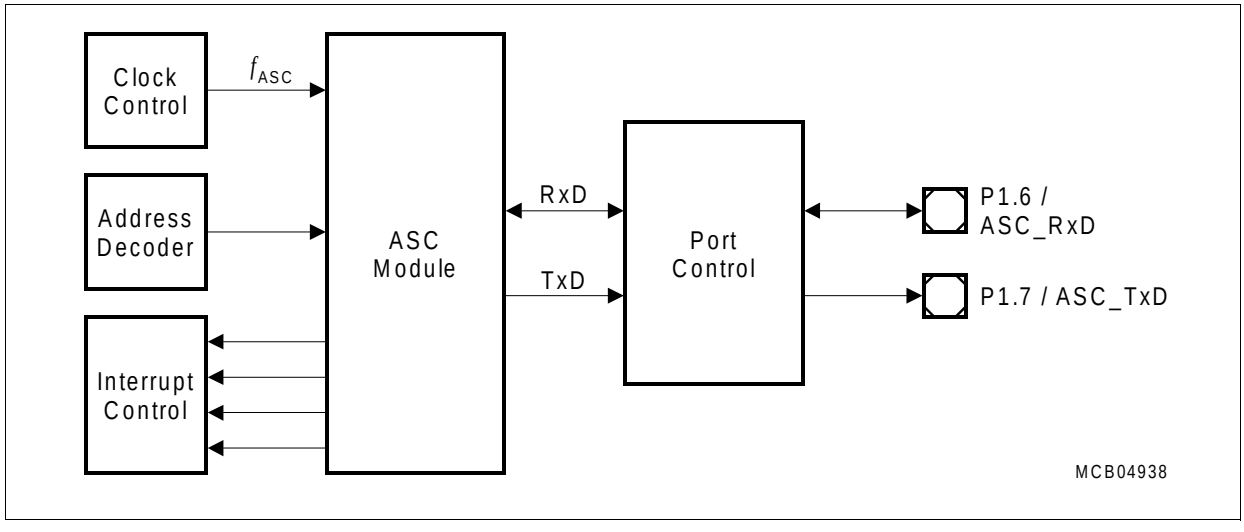


Figure 1-2 General Block Diagram of the ASC Interfaces

ASC Module communicates with the external world via one pair of I/O lines. The RXD line is the receive data input signal (in Synchronous Mode also output). TXD is the transmit output signal. Clock control, address decoding, and interrupt service request control are managed outside the ASC Module kernel.

The Asynchronous/Synchronous Serial Interface provides serial communication between the TC11IB and other microcontrollers, microprocessors or external peripherals.

The ASC supports full-duplex asynchronous communication and half-duplex synchronous communication. In Synchronous Mode, data is transmitted or received synchronous to a shift clock which is generated by the ASC internally. In Asynchronous Mode, 8-bit or 9-bit data transfer, parity generation, and the number of stop bits can be selected. Parity, framing, and overrun error detection are provided to increase the reliability of data transfers. Transmission and reception of data are double-buffered. For multiprocessor communication, a mechanism is included to distinguish address bytes from data bytes. Testing is supported by a loop-back option. A 13-bit baud rate generator

provides the ASC with a separate serial clock signal that can be very accurately adjusted by a prescaler implemented as a fractional divider.

Features:

- Full duplex asynchronous operating modes
  - 8- or 9-bit data frames, LSB first
  - Parity bit generation/checking
  - One or two stop bits
  - Baudrate from 3 MBaud to 0.71 Baud (@ 48 MHz clock)
- Multiprocessor mode for automatic address/data byte detection
- Loop-back capability
- Support for IrDA data transmission up to 115.2 KBaud maximum
- Half-duplex 8-bit synchronous operating mode
  - Baudrate from 6 MBaud to 488.3 Baud (@ 48 MHz clock)
- Double buffered transmitter/receiver
- Interrupt generation
  - On a transmitter buffer empty condition
  - On a transmit last bit of a frame condition
  - On a receiver buffer full condition
  - On an error condition (frame, parity, overrun error)
- FIFO
  - 8 bytes receive FIFO (RXFIFO)
  - 8 bytes transmit FIFO (TXFIFO)
  - Independent control of RXFIFO and TXFIFO
  - 9-bit FIFO data width
  - Programmable Receive/Transmit Interrupt Trigger Level
  - Receive and transmit FIFO filling level indication
  - Overrun error generation

1.4.1.2 High-Speed Synchronous Serial Interface

Figure 1-3 shows a global view of the functional blocks of the High-Speed Synchronous Serial interface SSC.

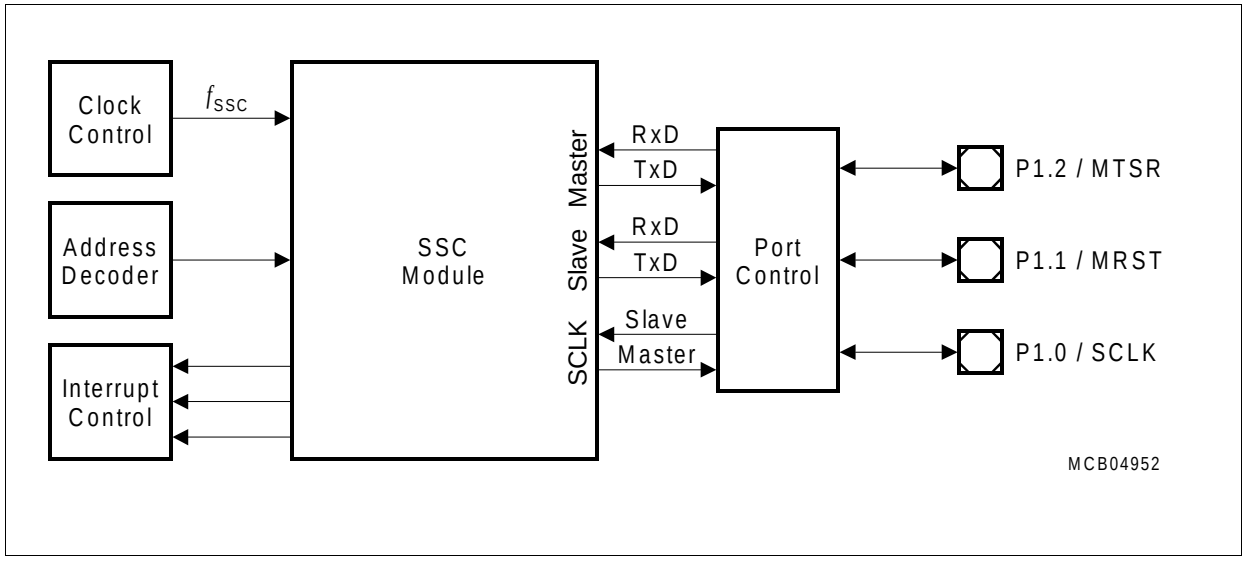


Figure 1-3 General Block Diagram of the SSC Interfaces

The SSC Module has three I/O lines, located at Port 1. The SSC Module is further supplied by separate clock control, interrupt control, address decoding, and port control logic.

The SSC supports full-duplex and half-duplex serial synchronous communication up to 24 MBaud (@ 48 MHz module clock). The serial clock signal can be generated by the SSC itself (master mode) or can be received from an external master (slave mode). Data width, shift direction, clock polarity, and phase are programmable. This allows communication with SPI-compatible devices. Transmission and reception of data are double-buffered. A 16-bit baud rate generator provides the SSC with a separate serial clock signal.

Features:

- Master and slave mode operation
  - Full-duplex or half-duplex operation
- Flexible data format
  - Programmable number of data bits: 2 to 16 bit
  - Programmable shift direction: LSB or MSB shift first
  - Programmable clock polarity: idle low or high state for the shift clock
  - Programmable clock/data phase: data shift with leading or trailing edge of the shift clock
- Baud rate generation from 24 MBaud to 366.2 Baud (@ 48 MHz module clock)
- Interrupt generation
  - On a transmitter empty condition
  - On a receiver full condition
  - On an error condition (receive, phase, baud rate, transmit error)
- Three-pin interface
  - Flexible SSC pin configuration

1.4.1.3 Asynchronous Serial Interface (16X50)

The 16X50 is a universal asynchronous receiver/transmitter (UART) which is fully prorammable.It supports word lengths from five to eight bits, an optional parity bit and one or two stop bits.If enabled, the parity can be odd, even or forced to a defined state. The 16X50 includes a 16-bit programmable baud rate generator and an 8-bit scratch register, together with two 16-byte FIFOs -one for transmit and one for receive. It has six modem control lines and supports a diagnostic loop-back mode. An interrupt can be generated from any one of 10 sources. **Figure 1-4** shows a global view of the functional blocks of the Asynchronous Serial Interface (16X50).

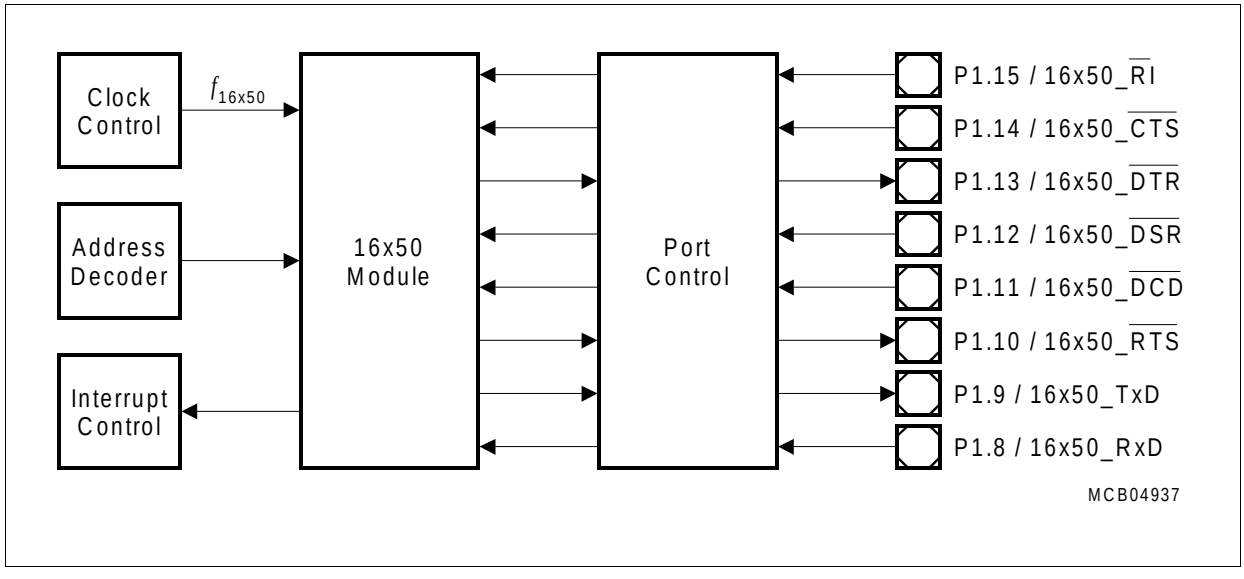


Figure 1-4 General Block Diagram of the 16X50 Interface

The 16X50 Module communicates with the external world via five input and three output lines located at Port 1.

The 16X50 provides serial asynchronous receive data synchronization, parallel-to-serial and serial-to-parallel data conversions for both the transmitter and receiver sections. These functions are necessary for converting the serial data stream into parallel data that is required with digital data systems. Synchronization for the serial data stream is accomplished by adding start and stops bits to the transmit data to form a data character (character orientated protocol). Data integrity is insured by attaching a parity bit to the data character. The parity bit is checked by the receiver for any transmission bit errors. The electronic circuitry to provide all these functions is fairly complex especially when manufactured on a single integrated silicon chip. The 16X50 represents such an integration with greatly enhanced features.

The 16X50 is an upward solution that provides 16 bytes of transmit and receive FIFO memory, instead of 1 byte provided in the 16C450. The 16X50 is designed to work with high speed modems and shared network environments, that require fast data processing

time. Increased performance is realized in the 16X50 by the larger transmit and receive FIFO's. This allows the external processor to handle more networking tasks within a given time. The 4 selectable levels of FIFO trigger provided for maximum data throughput performance especially when operating in a multi-channel environment. The combination of the above greatly reduces the bandwidth requirement of the external controlling CPU, increases performance, and reduces power consumption.

The 16X50 is capable of operation to 3 Mbps with a 48 MHz clock input (f_{16X50}).

Features:

- Software upward compatible with the NS16550A
- Standard modem interface
- Programmable word length, stop bits and parity
- Programmable baud rate generator
- Interrupt generation
- Diagnostic loop-back mode
- Scratch register
- Automatic hardware/software flow control
- Programmable XON/XOFF characters
- Independent transmit and receive control
- FIFO
  - 16 byte transmit FIFO
  - 16 byte receive FIFO with error flags
  - Four selectable receive FIFO interrupt trigger levels

1.4.2 Timer Units

The TC11IB includes two timer units:

– General Purpose Timer Units, GPTU0 and GPTU1.

1.4.2.1 General Purpose Timer Unit

Figure 1-5 shows a global view of all functional blocks of the two General Purpose Timer Unit (GPTU0 & GPTU1) Modules.

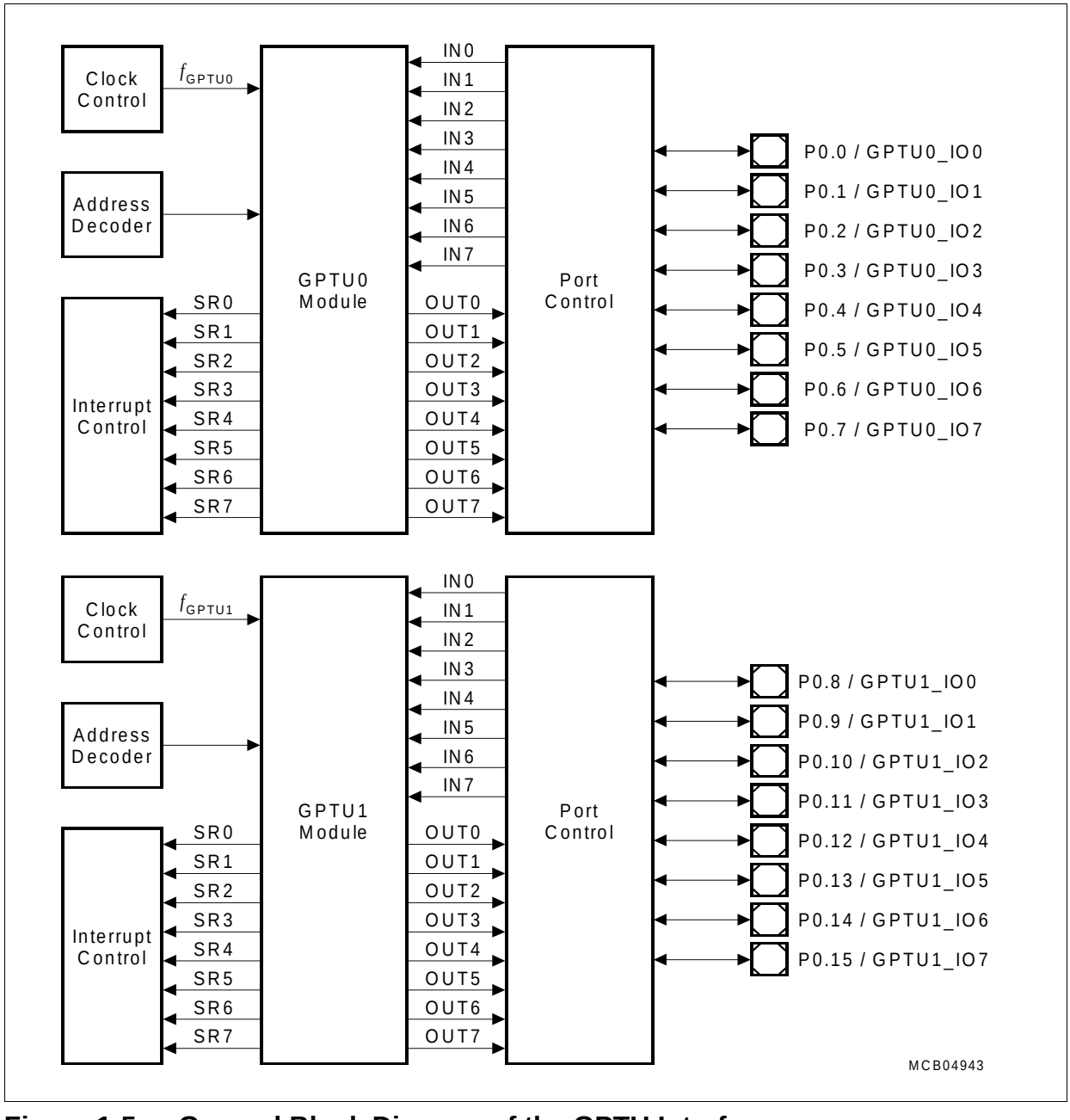


Figure 1-5 General Block Diagram of the GPTU Interface

Each GPTU module, GPTU0 and GPTU1, consists of three 32-bit timers designed to solve such application tasks as event timing, event counting, and event recording. And each GPTU module communicates with the external world via eight I/O lines located at Port 1.

The three timers in each GPTU Module T0, T1, and T2, can operate independently from each other or can be combined:

General Features:

- All timers are 32-bit precision timers with a maximum input frequency of f_{GPTU}.
- Events generated in T0 or T1 can be used to trigger actions in T2
- Timer overflow or underflow in T2 can be used to clock either T0 or T1
- T0 and T1 can be concatenated to form one 64-bit timer

Features of T0 and T1:

- Each timer has a dedicated 32-bit reload register with automatic reload on overflow
- Timers can be split into individual 8-, 16-, or 24-bit timers with individual reload registers
- Overflow signals can be selected to generate service requests, pin output signals, and T2 trigger events
- Two input pins can determine a count option

Features of T2:

- Count up or down is selectable
- Operating modes:
  - Timer
  - Counter
  - Quadrature counter (incremental/phase encoded counter interface)
- Options:
  - External start/stop, one-shot operation, timer clear on external event
  - Count direction control through software or an external event
  - Two 32-bit reload/capture registers
- Reload modes:
  - Reload on overflow or underflow
  - Reload on external event: positive transition, negative transition, or both transitions
- Capture modes:
  - Capture on external event: positive transition, negative transition, or both transitions
  - Capture and clear timer on external event: positive transition, negative transition, or both transitions
- Can be split into two 16-bit counter/timers

- Timer count, reload, capture, and trigger functions can be assigned to input pins. T0 and T1 overflow events can also be assigned to these functions.
- Overflow and underflow signals can be used to trigger T0 and/or T1 and to toggle output pins
- T2 events are freely assignable to the service request nodes.

1.4.3 MultiMediaCard Interface (MMCI)

The MultiMediaCard Interface module provides interface to MultiMediaCard bus. It supports the full MultiMediaCard bus protocol as defined in MultiMediaCard system specification version 1.3. **Figure 1-6** shows a global view of the MMCI module with the module specific interface connections.

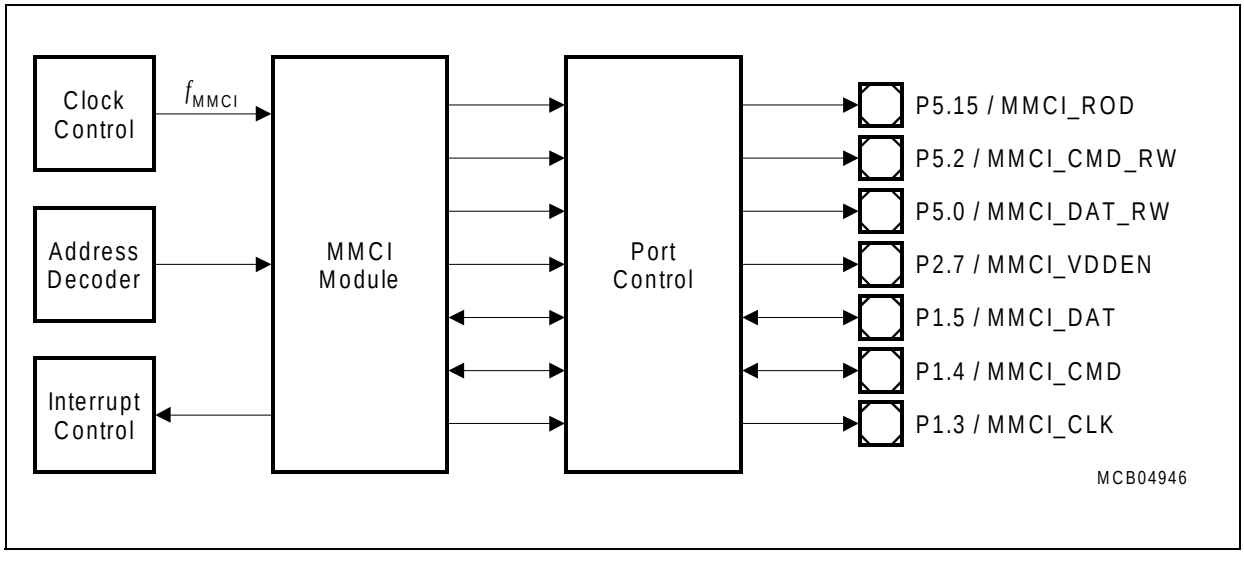


Figure 1-6 General Block Diagram of MMCI Interface

The MMCI module communicates with external world via two IO lines and five output lines which are located at Port 1, 2 and 5. Clock control, interrupt service and address decoding are managed outside the MMCI module Kernel.

MMCI handles the data transfer on CMD and DAT of the MMC Bus. It performs the transfer from bit serial to byte parallel or vice versa and sustains a 16 Mbps data rate. To fulfil the MMC Bus protocol, special bytes are modified via inserting start and stop bits or CRC bits. A clock controller is implemented to divide the clock to the necessary MMC Bus clock frequency.

Features

- 3 line serial interface --- Glueless interface to MultiMediaCard Bus
- Pointer based data transfer
- Block and sequential card access
- 16MHz MultiMediaCard bus clock generation
- CRC protection for the MultiMediaCard bus communication
- Optional programming voltage control
- Buffered data transfer
- Power management
- Data communication with a data rate up to 2 Mbyte/s

1.4.4 Ethernet Controller

The MAC controller implements the IEEE 802.3 and operates either at 100 Mbps or 10 Mbps. **Figure 1-7** shows a global view of the Ethernet Controller module with the module specific interface connections.

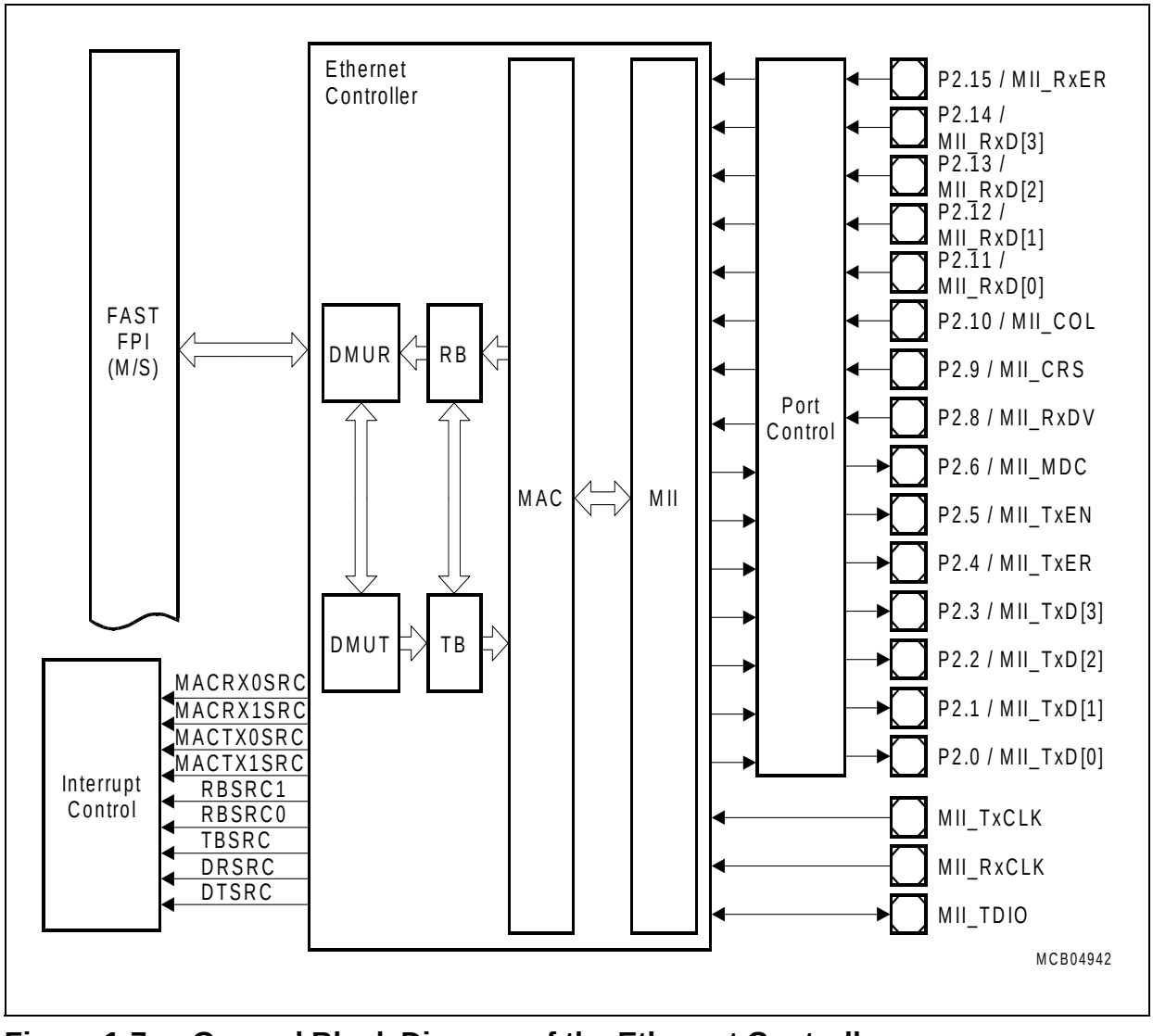


Figure 1-7 General Block Diagram of the Ethernet Controller

The Ethernet controller comprises the following functional blocks:

1. Media Access Controller (MAC)
2. Receive Buffer (RB)
3. Transmit Buffer (TB)
4. Data Management Unit in Receive Direction (DMUR)
5. Data Management Unit in Transmit Direction (DMUT)

RB as well as TB provides on-chip data buffering whereas DMUR and DMUT perform data transfer from/to the shared memory.

Two interfaces are provided by the Ethernet Controller Module:

1. MII interface for connection of Ethernet PHYs via eighteen Input / Output lines
2. Master/slave FPI bus interface for connection to the on-chip system bus for data transfer as well as configuration.

Features

- Media Independent Interface (MII) according to IEEE 802.3
- Support 10 or 100 Mbps MII-based Physical devices.
- Support Full Duplex Ethernet.
- Support data transfer between Ethernet Controller and COM-DRAM.
- Support data transfer between Ethernet Controller and SDRAM via EBU.
- 256 x 32 bit Receive buffer and Transmit buffer each.
- Support burst transfers up to 8 x 32 Byte.

Media Access Controller (MAC)

- 100/10-Mbps operations
- Full IEEE 802.3 compliance
- Station management signaling
- Large on-chip CAM (Content Addressable Memory)
- Full duplex mode
- 80-byte transmit FIFO
- 16-byte receive FIFO
- PAUSE Operation
- Flexible MAC Control Support
- Support Long Packet Mode and Short Packet Mode
- PAD generation

Media Independent Interface (MII)

- Media independence.
- Multi-vendor point of interoperability.
- Support connection of MAC layer and Physical (PHY) layer devices.
- Capable of supporting both 100 Mbps and 10 Mbps data rates.
- Data and delimiters are synchronous to clock references.
- Provides independent four bit wide transmit and receive data paths.
- Support connection of PHY layer and Station Management (STA) devices.
- Provides a simple management interface.
- Capable of driving a limited length of shielded cable.

1.4.5 PCI

The PCI Interface module of the TC11IB basically is a bus bridge between the on-chip FPI bus and the external PCI bus of the system. The PCI Interface is fully compliant to PCI Local Bus Specification Rev. 2.2. **Figure 1-8** shows a global view of the PCI module with the module specific pin connections.

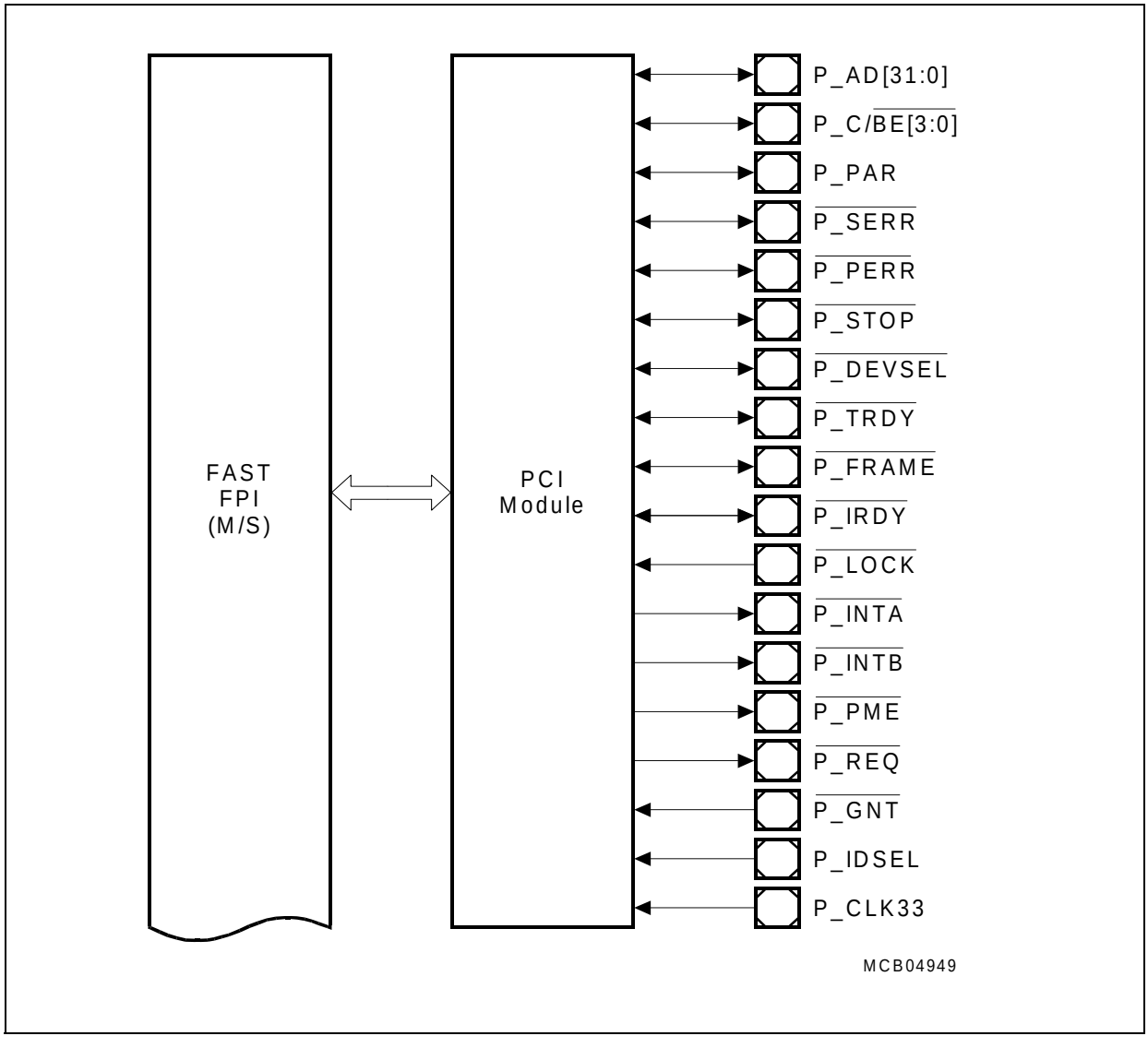


Figure 1-8 General Block Diagram of the PCI Interface

The PCI-FPI bridge is able to execute a number of various data transfers between the FPI bus and the PCI bus. Beside the standard PCI functions (configuration transactions), there are two main types of transfers which the bridge supports. Firstly, it will forward a transaction that any PCI initiator directs to the PCI interface of the TC11IB to the on-chip FPI bus. Secondly the bridge will forward certain transactions that a FPI master initiates on the FPI bus to the PCI bus. Depending on configuration, these transfers may be a

single data or burst transfers on both PCI and FPI bus. In addition, the bridge is able to handle a direct data transfer between PCI bus and FPI bus utilizing it's programmable DMA channel. The DMA channel can only be activated by a FPI master. In order to work as a PCI host bridge on the PCI bus, the variety of PCI transactions issued by the bridge includes configuration transactions of type 0 and type 1 when acting as a PCI master.

Features

- PCI V2.2 compliant, 32 bit, 33 MHz
- Multifunction Device, Support both PCI Master/Host functions. These functions can be activated by:
  - TriCore
  - Fast Ethernet
  - DMA Channel
- Support Burst Transfer from PCI to ComDRAM, SDRAM and Code DRAM.
- Support DMA Channel data transfers between PCI and FPI
- Loading of PCI Configuration Registers done by TriCore via FPI Bus access
- Support PCI Command
- Support Card-Bus.
- Power management
  - according to PCI Bus Power Management Interface Specification V1.1
  - Support Multiple PCI power management states D0, D1, D2, $D3_{cold}$
  - PME#-Signalling from Fast Ethernet in D1, D2.
- PCI Reset
  - All tristatable PCI outputs of the bridge are set to "Tristate" upon PCI Reset, compliant to PCI Local Bus Specification V2.2

1.5 Pin Definitions and Functions

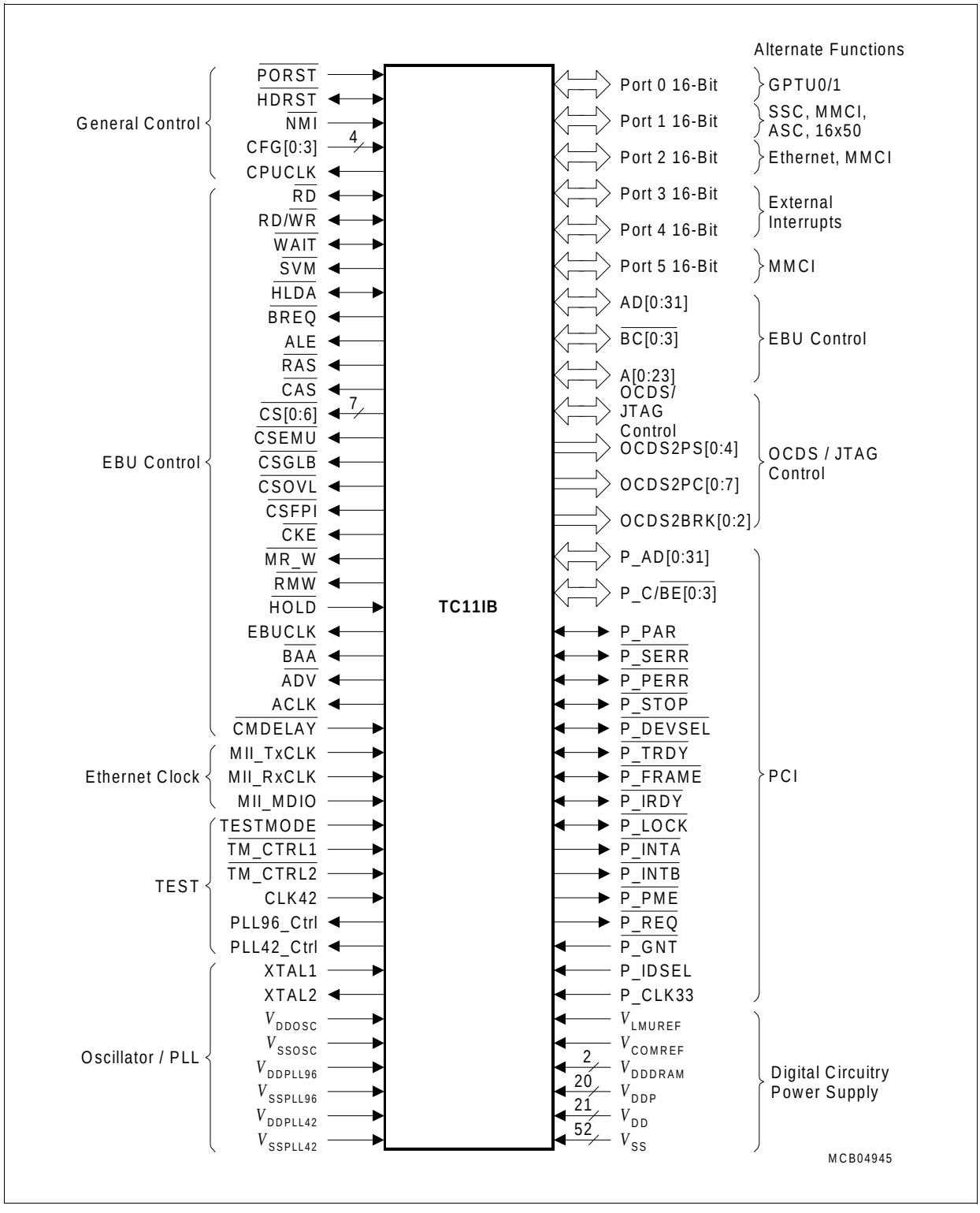


Figure 1-9 TC11IB Pin Configuration

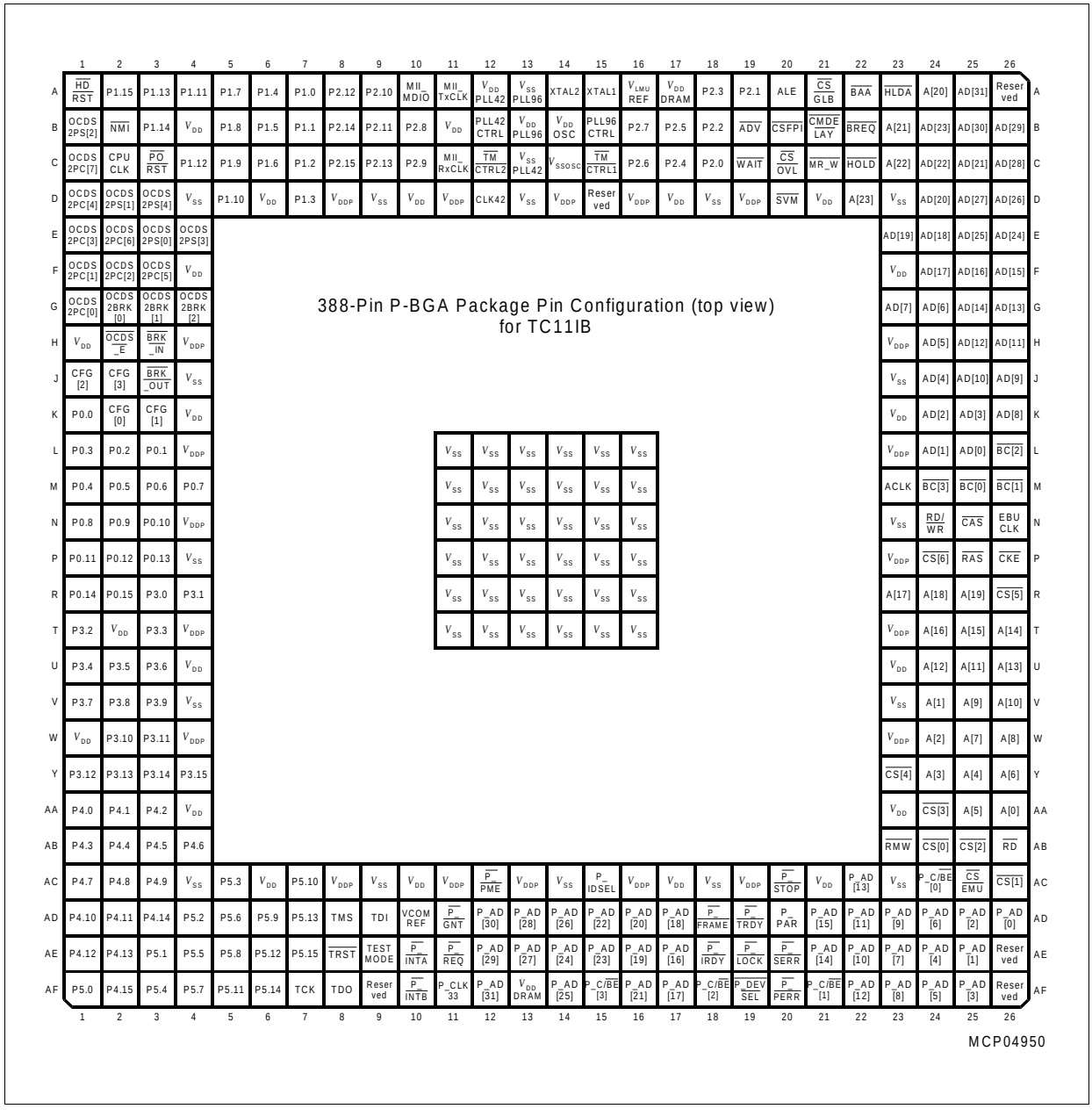


Figure 1-10 TC11IB Pinning: P-BGA-388 Package (top view)

Table 1-3 Pin Definitions and Functions

Symbol	Pin	In Out	PU/ PD[1)]	Functions
P0		I/O		**Port 0** Port 0 serves as 16-bit general purpose I/O port, which is also used as input/output for the General Purpose Timer Units (GPTU0 & GPTU1)
P0.0	K1	I/O	PUB	GPTU0_IO0 GPTU0 I/O line 0
P0.1	L3	I/O	PUB	GPTU0_IO1 GPTU0 I/O line 1
P0.2	L2	I/O	PUB	GPTU0_IO2 GPTU0 I/O line 2
P0.3	L1	I/O	PUB	GPTU0_IO3 GPTU0 I/O line 3
P0.4	M1	I/O	—	GPTU0_IO4 GPTU0 I/O line 4
P0.5	M2	I/O	—	GPTU0_IO5 GPTU0 I/O line 5
P0.6	M3	I/O	—	GPTU0_IO6 GPTU0 I/O line 6
P0.7	M4	I/O	—	GPTU0_IO7 GPTU0 I/O line 7
P0.8	N1	I/O	PUC	GPTU1_IO0 GPTU1 I/O line 0
P0.9	N2	I/O	PDC	GPTU1_IO1 GPTU1 I/O line 1
P0.10	N3	I/O	PDC	GPTU1_IO2 GPTU1 I/O line 2
P0.11	P1	I/O	PUC	GPTU1_IO3 GPTU1 I/O line 3
P0.12	P2	I/O	PUC	GPTU1_IO4 GPTU1 I/O line 4
P0.13	P3	I/O	PUC	GPTU1_IO5 GPTU1 I/O line 5
P0.14	R1	I/O	PUC	GPTU1_IO6 GPTU1 I/O line 6
P0.15	R2	I/O	PUC	GPTU1_IO7 GPTU1 I/O line 7

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions	
P1		I/O		**Port 1** Port 1 serves as 16-bit general purpose I/O port, which also is used as input/output for the serial interfaces (SSC,ASC,16X50) and MultiMediaCard Interface (MMCI)	
P1.0	A7	I/O	PUC	SCLK	SSC clock input/output line
P1.1	B7	I/O	PUC	MRST	SSC master receive / slave transmit input/output
P1.2	C7	I/O	PUC	MTSR	SSC master transmit / slave receive input/output
P1.3	D7	O	PUC	MMCI_CLK	MMCI clock output line
P1.4	A6	I/O	PUC	MMCI_CMD	MMCI command input/output line
P1.5	B6	I/O	PUC	MMCI_DAT	MMCI data input/output line
P1.6	C6	I/O	PUC	ASC_RXD	ASC receiver input/output line
P1.7	A5	O	PUC	ASC_TXD	ASC transmitter output line
P1.8	B5	I	PUC	16X50_RXD	16X50 receiver input line
P1.9	C5	O	PUC	16X50_TXD	16X50 transmitter output line
P1.10	D5	O	PUC	16X50_$\overline{\text{RTS}}$	16X50 request to send output line
P1.11	A4	I	PUC	16X50_$\overline{\text{DCD}}$	16X50 data carrier detection input line
P1.12	C4	I	PUC	16X50_$\overline{\text{DSR}}$	16X50 data set ready input line
P1.13	A3	O	PUC	16X50_$\overline{\text{DTR}}$	16X50 data terminal ready output line
P1.14	B3	I	PUC	16X50_$\overline{\text{CTS}}$	16X50 clear to send input line
P1.15	A2	I	PUC	16X50_$\overline{\text{RI}}$	16X50 ring indicator input line

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions	
P2		I/O		**Port 2** Port 2 serves as 16-bit general purpose I/O port, which is also used as input/output for Ethernet controller and MultiMediaCard (MMCI).	
P2.0	C18	O	—	MII_TXD0	Ethernet controller transmit data output line 0
P2.1	A19	O	—	MII_TXD1	Ethernet controller transmit data output line 1
P2.2	B18	O	—	MII_TXD2	Ethernet controller transmit data output line 2
P2.3	A18	O	—	MII_TXD3	Ethernet controller transmit data output line 3
P2.4	C17	O	—	MII_TXER	Ethernet controller transmit error output line
P2.5	B17	O	—	MII_TXEN	Ethernet controller transmit enable output line
P2.6	C16	O	—	MII_MDC	Ethernet controller management data clock output line
P2.7	B16	O	PUC	MMCI_VDDEN	MMCI power supply enable output line
P2.8	B10	I	PDC	MII_RXDV	Ethernet Controller receive data valid input line
P2.9	C10	I	PDC	MII_CRS	Ethernet Controller carrier input line
P2.10	A9	I	PUC	MII_COL	Ethernet Controller collision input line
P2.11	B9	I	PDC	MII_RXD0	Ethernet Controller receive data input line 0
P2.12	A8	I	PDC	MII_RXD1	Ethernet Controller receive data input line 1
P2.13	C9	I	PDC	MII_RXD2	Ethernet Controller receive data input line 2
P2.14	B8	I	PDC	MII_RXD3	Ethernet Controller receive data input line 3
P2.15	C8	I	PDC	MII_RXER	Ethernet Controller receive error input line

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions	
P3		I/O		**Port 3** Port 3 serves as 16-bit general purpose I/O port, which is also used as input for external interrupts.	
P3.0	R3	I	—	INT0	External interrupt input line 0
P3.1	R4	I	—	INT1	External interrupt input line 1
P3.2	T1	I	—	INT2	External interrupt input line 2
P3.3	T3	I	—	INT3	External interrupt input line 3
P3.4	U1	I	—	INT4	External interrupt input line 4
P3.5	U2	I	—	INT5	External interrupt input line 5
P3.6	U3	I	—	INT6	External interrupt input line 6
P3.7	V1	I	—	INT7	External interrupt input line 7
P3.8	V2	I	—	INT8	External interrupt input line 8
P3.9	V3	I	—	INT9	External interrupt input line 9
P3.10	W2	I	—	INT10	External interrupt input line 10
P3.11	W3	I	—	INT11	External interrupt input line 11
P3.12	Y1	I	—	INT12	External interrupt input line 12
P3.13	Y2	I	—	INT13	External interrupt input line 13
P3.14	Y3	I	—	INT14	External interrupt input line 14
P3.15	Y4	I	—	INT15	External interrupt input line 15

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions	
P4		I/O		**Port 4** Port 4 is used as general purpose I/O port, 8 pins of which (P4.0 ~ P4.7) also serve as inputs for external interrupts.	
P4.0	AA1	I	PDC	INT16	External interrupt input line 16
P4.1	AA2	I	PDC	INT17	External interrupt input line 17
P4.2	AA3	I	PDC	INT18	External interrupt input line 18
P4.3	AB1	I	PDC	INT19	External interrupt input line 19
P4.4	AB2	I	PUC	INT20	External interrupt input line 20
P4.5	AB3	I	PUC	INT21	External interrupt input line 21
P4.6	AB4	I	PUC	INT22	External interrupt input line 22
P4.7	AC1	I	PUC	INT23	External interrupt input line 23
P4.8	AC2	I/O	PUC		
P4.9	AC3	I/O	PUC		
P4.10	AD1	I/O	PUC		
P4.11	AD2	I/O	PUC		
P4.12	AE1	I/O	PUC		
P4.13	AE2	I/O	PUC		
P4.14	AD3	I/O	PUC		
P4.15	AF2	I/O	PUC		

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
P5		I/O		**Port 5** Port 5 serves as 16-bit general purpose I/O port, 3 pins of which (P5.0, P5.2 and P5.15) serve as output lines for MultiMediaCard Interface (MMCI) also.
P5.0	AF1	O	PUC	MMCI_DATRW MMCI data direction indicator output line
P5.1	AE3	I/O	PUC	
P5.2	AD4	O	PUC	MMCI_CMDRW MMCI command direction indicator output line
P5.3	AC5	I/O	PUC	
P5.4	AF3	I/O	PUC	
P5.5	AE4	I/O	PDC	
P5.6	AD5	I/O	PUC	
P5.7	AF4	I/O	PUC	
P5.8	AE5	I/O	PDC	
P5.9	AD6	I/O	PUC	
P5.10	AC7	I/O	—	
P5.11	AF5	I/O	—	
P5.12	AE6	I/O	—	
P5.13	AD7	I/O	—	
P5.14	AF6	I/O	—	
P5.15	AE7	O	PUC	MMCI_ROD MMCI command line mode indicator output line
$\overline{\text{HDRST}}$	A1	I/O	—	**Hardware Reset Input/Reset Indication Output** Assertion of this bidirectional open-drain pin causes a synchronous reset of the chip through external circuitry. This pin must be driven for a minimum duration. The internal reset circuitry drives this pin in response to a power-on, hardware, watchdog, power-down wake-up reset and eDRAM reset for a specific period of time. For a software reset, activation of this pin is programmable.
$\overline{\text{PORST}}$	C3	I	PUC	**Power-on Reset Input** A low level on $\overline{\text{PORST}}$ causes an asynchronous reset of the entire chip. $\overline{\text{PORST}}$ is a fully asynchronous level sensitive signal.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
$\overline{\text{NMI}}$	B2	I	PUB	**Non-Maskable Interrupt Input** A high-to-low transition on this pin causes a NMI-Trap request to the CPU.
CFG0 **CFG1** **CFG2** **CFG3**	K2 K3 J1 J2	I I I I	PDC PDC PUC PUC	**Operation Configuration Inputs** The configuration inputs define the boot options of the TC11IB after a hardware-invoked reset operation.
CPU CLK	C2	O	PUC	**Clock Output**
$\overline{\text{TRST}}$	AE8	I	PDC	**JTAG Module Reset/Enable Input** A low level at this pin resets and disables the JTAG module. A high level enables the JTAG module.
TCK	AF7	I	PUC	**JTAG Module Clock Input**
TDI	AD9	I	PUC	**JTAG Module Serial Data Input**
TDO	AF8	O	—	**JTAG Module Serial Data Output**
TMS	AD8	I	PUC	**JTAG Module State Machine Control Input**
$\overline{\text{OCDSE}}$	H2	I	PUC	**OCDS Enable Input** A low level on this pin during power-on reset ($\overline{\text{PORST}}$ = 0) enables the on-chip debug support (OCDS). In addition, the level of this pin during power-on reset determines the boot configuration.
$\overline{\text{BRKIN}}$	H3	I	PUC	**OCDS Break Input** A low level on this pin causes a break in the chip's execution when the OCDS is enabled. In addition, the level of this pin during power-on reset determines the boot configuration.
$\overline{\text{BRKOUT}}$	J3	O	—	**OCDS Break Output** A low level on this pin indicates that a programmable OCDS event has occurred.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
OCDS2 PS0	E3	O	PUC	**Pipeline Status Signal Outputs**
OCDS2 PS1	D2	O	PUC	
OCDS2 PS2	B1	O	PUC	
OCDS2 PS3	E4	O	PUC	
OCDS2 PS4	D3	O	PUC	
OCDS2 PC0	G1	O	PUC	**Indirect PC Address Outputs**
OCDS2 PC1	F1	O	PUC	
OCDS2 PC2	F2	O	PUC	
OCDS2 PC3	E1	O	PUC	
OCDS2 PC4	D1	O	PUC	
OCDS2 PC5	F3	O	PUC	
OCDS2 PC6	E2	O	PUC	
OCDS2 PC7	C1	O	PUC	
OCDS2 BRK0	G2	O	PUC	**Break Qualification Lines outputs**
OCDS2 BRK1	G3	O	PUC	
OCDS2 BRK2	G4	O	PUC	

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
				PCI Interface Address /Data Bus Input / Output Lines
P_AD0	AD26	I/O	—	PCI Interface Address / Data Bus Line 0
P_AD1	AE25	I/O	—	PCI Interface Address / Data Bus Line 1
P_AD2	AD25	I/O	—	PCI Interface Address / Data Bus Line 2
P_AD3	AF25	I/O	—	PCI Interface Address / Data Bus Line 3
P_AD4	AE24	I/O	—	PCI Interface Address / Data Bus Line 4
P_AD5	AF24	I/O	—	PCI Interface Address / Data Bus Line 5
P_AD6	AD24	I/O	—	PCI Interface Address / Data Bus Line 6
P_AD7	AE23	I/O	—	PCI Interface Address / Data Bus Line 7
P_AD8	AF23	I/O	—	PCI Interface Address / Data Bus Line 8
P_AD9	AD23	I/O	—	PCI Interface Address / Data Bus Line 9
P_AD10	AE22	I/O	—	PCI Interface Address / Data Bus Line 10
P_AD11	AD22	I/O	—	PCI Interface Address / Data Bus Line 11
P_AD12	AF22	I/O	—	PCI Interface Address / Data Bus Line 12
P_AD13	AC22	I/O	—	PCI Interface Address / Data Bus Line 13
P_AD14	AE21	I/O	—	PCI Interface Address / Data Bus Line 14
P_AD15	AD21	I/O	—	PCI Interface Address / Data Bus Line 15
P_AD16	AE17	I/O	—	PCI Interface Address / Data Bus Line 16
P_AD17	AF17	I/O	—	PCI Interface Address / Data Bus Line 17
P_AD18	AD17	I/O	—	PCI Interface Address / Data Bus Line 18
P_AD19	AE16	I/O	—	PCI Interface Address / Data Bus Line 19
P_AD20	AD16	I/O	—	PCI Interface Address / Data Bus Line 20
P_AD21	AF16	I/O	—	PCI Interface Address / Data Bus Line 21
P_AD22	AD15	I/O	—	PCI Interface Address / Data Bus Line 22
P_AD23	AE15	I/O	—	PCI Interface Address / Data Bus Line 23
P_AD24	AE14	I/O	—	PCI Interface Address / Data Bus Line 24
P_AD25	AF14	I/O	—	PCI Interface Address / Data Bus Line 25
P_AD26	AD14	I/O	—	PCI Interface Address / Data Bus Line 26
P_AD27	AE13	I/O	—	PCI Interface Address / Data Bus Line 27
P_AD28	AD13	I/O	—	PCI Interface Address / Data Bus Line 28
P_AD29	AE12	I/O	—	PCI Interface Address / Data Bus Line 29
P_AD30	AD12	I/O	—	PCI Interface Address / Data Bus Line 30
P_AD31	AF12	I/O	—	PCI Interface Address / Data Bus Line 31
P_PAR	AD20	I/O	—	**PCI Interface Parity Input / Output**
$\overline{\text{P\_SERR}}$	AE20	I/O	—	**PCI Interface System Error Input / Output**
$\overline{\text{P\_PERR}}$	AF20	I/O	—	**PCI Interface Parity Error Input / Output**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
P_STOP	AC20	I/O	—	**PCI Interface Stop Input / Output**
P_C/BE0 **P_C/BE1** **P_C/BE2** **P_C/BE3**	AC24 AF21 AF18 AF15	I/O I/O I/O I/O	— — — —	**PCI Interface Command / Byte Enable Inputs / Outputs**
P_IDSEL	AC15	I	—	**PCI Interface ID Select Input**
P_CLK33	AF11	I	—	**PCI Interface Clock Input**
P_REQ	AE11	O	—	**PCI Interface Bus Request Output**
P_GNT	AD11	I	—	**PCI Interface Bus Grant Input**
P_DEVSEL	AF19	I/O	—	**PCI Interface Device Select Input / Output**
P_TRDY	AD19	I/O	—	**PCI Interface Target Ready Input / Output**
P_FRAME	AD18	I/O	—	**PCI Interface Frame Input / Output**
P_IRDY	AE18	I/O	—	**PCI Interface Initiator Ready Input / Output**
P_LOCK	AE19	I	—	**PCI Interface Lock Input**
P_INTA	AE10	O	—	**PCI Interface Interrupt A Output**
P_INTB	AF10	O	—	**PCI Interface Interrupt B Output**
P_PME	AC12	O	—	**PCI Interface Power Management Event Output**
MII_ TXCLK	A11	I	PDC	**Ethernet Controller Transmit Clock** MII_TXD[3:0] and MII_TXEN are driven off the rising edge of the MII_TXCLK by the core and sampled by the PHY on the rising edge of the MII_TXCLK.
MII_ RXCLK	C11	I	PDC	**Ethernet Controller Receive Clock** MII_RXCLK is a continuous clock. Its frequency is 25 MHz for 100 Mbps operation, and 2.5 MHz for 10 Mbps. MII_RXD[3:0], MII_RXDV and MII_EXER are driven by the PHY off the falling edge of MII_RXCLK and sampled on the rising edge of MII_RXCLK.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
MII_ MDIO	A10	I/O	PDA	**Ethernet Controller Management Data Input / Output** When a read command is being executed, data which is clocked out of the PHY will be presented on the input line. When the Core is clocking control or data onto the MII_MDIO line, the signal will carry the information.
$\overline{CS0}$ **$\overline{CS1}$** **$\overline{CS2}$** **$\overline{CS3}$** **$\overline{CS4}$** **$\overline{CS5}$** **$\overline{CS6}$**	AB24 AC26 AB25 AA24 Y23 R26 P24	O O O O O O O	PUC PUC PUC PUC PUC PUC PUC	**EBU_LMB Chip Select Output Line 0** **EBU_LMB Chip Select Output Line 1** **EBU_LMB Chip Select Output Line 2** **EBU_LMB Chip Select Output Line 3** **EBU_LMB Chip Select Output Line 4** **EBU_LMB Chip Select Output Line 5** **EBU_LMB Chip Select Output Line 6** Each corresponds to a programmable region. Only one can be active at one time.
$\overline{CSEMU}$	AC25	O	PUC	**EBU_LMB Chip Select Output for Emulator Region**
$\overline{CSGLB}$	A21	O	PUC	**EBU_LMB Chip Select Global Output**
$\overline{CSOVL}$	C20	O	PUC	**EBU_LMB Chip Select Output for Overlay Memory**
$\overline{CSFPI}$	B20	I	PUC	**EBU_LMB Chip Select Input for Internal FPI Bus** For external master to select EBU_LMB as target in the slave mode
EBUCLK	N26	O	—	**EBU_LMB External Bus Clock Output** Derived from LMBCLK as equal, half or one-fourth of the frequency.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
				EBU_LMB Address / Data Bus Input / Output Lines
AD0	L25	I/O	PUC	EBU_LMB Address / Data Bus Line 0
AD1	L24	I/O	PUC	EBU_LMB Address / Data Bus Line 1
AD2	K24	I/O	PUC	EBU_LMB Address / Data Bus Line 2
AD3	K25	I/O	PUC	EBU_LMB Address / Data Bus Line 3
AD4	J24	I/O	PUC	EBU_LMB Address / Data Bus Line 4
AD5	H24	I/O	PUC	EBU_LMB Address / Data Bus Line 5
AD6	G24	I/O	PUC	EBU_LMB Address / Data Bus Line 6
AD7	G23	I/O	PUC	EBU_LMB Address / Data Bus Line 7
AD8	K26	I/O	PUC	EBU_LMB Address / Data Bus Line 8
AD9	J26	I/O	PUC	EBU_LMB Address / Data Bus Line 9
AD10	J25	I/O	PUC	EBU_LMB Address / Data Bus Line 10
AD11	H26	I/O	PUC	EBU_LMB Address / Data Bus Line 11
AD12	H25	I/O	PUC	EBU_LMB Address / Data Bus Line 12
AD13	G26	I/O	PUC	EBU_LMB Address / Data Bus Line 13
AD14	G25	I/O	PUC	EBU_LMB Address / Data Bus Line 14
AD15	F26	I/O	PUC	EBU_LMB Address / Data Bus Line 15
AD16	F25	I/O	PUC	EBU_LMB Address / Data Bus Line 16
AD17	F24	I/O	PUC	EBU_LMB Address / Data Bus Line 17
AD18	E24	I/O	PUC	EBU_LMB Address / Data Bus Line 18
AD19	E23	I/O	PUC	EBU_LMB Address / Data Bus Line 19
AD20	D24	I/O	PUC	EBU_LMB Address / Data Bus Line 20
AD21	C25	I/O	PUC	EBU_LMB Address / Data Bus Line 21
AD22	C24	I/O	PUC	EBU_LMB Address / Data Bus Line 22
AD23	B24	I/O	PUC	EBU_LMB Address / Data Bus Line 23
AD24	E26	I/O	PUC	EBU_LMB Address / Data Bus Line 24
AD25	E25	I/O	PUC	EBU_LMB Address / Data Bus Line 25
AD26	D26	I/O	PUC	EBU_LMB Address / Data Bus Line 26
AD27	D25	I/O	PUC	EBU_LMB Address / Data Bus Line 27
AD28	C26	I/O	PUC	EBU_LMB Address / Data Bus Line 28
AD29	B26	I/O	PUC	EBU_LMB Address / Data Bus Line 29
AD30	B25	I/O	PUC	EBU_LMB Address / Data Bus Line 30
AD31	A25	I/O	PUC	EBU_LMB Address / Data Bus Line 31
$\overline{\text{BC0}}$	M25	I/O	PUC	**EBU_LMB Byte Control Line 0**
$\overline{\text{BC1}}$	M26	I/O	PUC	**EBU_LMB Byte Control Line 1**
$\overline{\text{BC2}}$	L26	I/O	PUC	**EBU_LMB Byte Control Line 2**
$\overline{\text{BC3}}$	M24	I/O	PUC	**EBU_LMB Byte Control Line 3**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
				EBU_LMB Address Bus Input / Output Lines
A0	AA26	I/O	PUC	EBU_LMB Address Bus Line 0
A1	V24	I/O	PUC	EBU_LMB Address Bus Line 1
A2	W24	I/O	PUC	EBU_LMB Address Bus Line 2
A3	Y24	I/O	PUC	EBU_LMB Address Bus Line 3
A4	Y25	I/O	PUC	EBU_LMB Address Bus Line 4
A5	AA25	I/O	PUC	EBU_LMB Address Bus Line 5
A6	Y26	I/O	PUC	EBU_LMB Address Bus Line 6
A7	W25	I/O	PUC	EBU_LMB Address Bus Line 7
A8	W26	I/O	PUC	EBU_LMB Address Bus Line 8
A9	V25	I/O	PUC	EBU_LMB Address Bus Line 9
A10	V26	I/O	PUC	EBU_LMB Address Bus Line 10
A11	U25	I/O	PUC	EBU_LMB Address Bus Line 11
A12	U24	I/O	PUC	EBU_LMB Address Bus Line 12
A13	U26	I/O	PUC	EBU_LMB Address Bus Line 13
A14	T26	I/O	PUC	EBU_LMB Address Bus Line 14
A15	T25	I/O	PUC	EBU_LMB Address Bus Line 15
A16	T24	I/O	PUC	EBU_LMB Address Bus Line 16
A17	R23	I/O	PUC	EBU_LMB Address Bus Line 17
A18	R24	I/O	PUC	EBU_LMB Address Bus Line 18
A19	R25	I/O	PUC	EBU_LMB Address Bus Line 19
A20	A24	I/O	PUC	EBU_LMB Address Bus Line 20
A21	B23	I/O	PUC	EBU_LMB Address Bus Line 21
A22	C23	I/O	PUC	EBU_LMB Address Bus Line 22
A23	D22	I/O	PUC	EBU_LMB Address Bus Line 23
$\overline{RD}$	AB26	I/O	PUC	**EBU_LMB Read Control Line** Output in the master mode Input in the slave mode.
RD/$\overline{WR}$	N24	I/O	PUC	**EBU_LMB Write Control Line** Output in the master mode Input in the slave mode.
$\overline{WAIT}$	C19	I/O	PUC	**EBU_LMB Wait Control Line**
$\overline{SVM}$	D20	O	PUB	**EBU_LMB Supervisor Mode Output**
ALE	A20	O	PDC	**EBU_LMB Address Latch Enable Output**
$\overline{RAS}$	P25	O	PUC	**EBU_LMB SDRAM Row Address Strobe Output**
$\overline{CAS}$	N25	O	PUC	**EBU_LMB SDRAM Column Address Strobe Output**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
$\overline{\text{CKE}}$	P26	O	PUC	**EBU_LMB SDRAM Clock Enable Output**
MR/$\overline{\text{W}}$	C21	O	PUC	**EBU_LMB Motorola-style Read / Write Output**
$\overline{\text{HOLD}}$	C22	I	PUC	**EBU_LMB Hold Request Input** In External Master Mode: While $\overline{\text{HOLD}}$ is high, Tricore is operating in normal mode (is owner of the external bus). A high-to-low transition indicates a hold request from an external master.Tricore backs off the bus and activates $\overline{\text{HLDA}}$ and goes into hold mode. A low-to-high transitions causes an exit from hold mode.Tricore deactivates $\overline{\text{HLDA}}$ and takes over the bus and enters the normal operation again. In External Slave Mode: While both $\overline{\text{HOLD}}$ and $\overline{\text{HLDA}}$ are high, Tricore is in hold mode, the external bus interface signals are tristated. When Tricore is released out of hold mode ($\overline{\text{HLDA}}$ =0) and has completely taken over control of the external bus, a low level at this pin requests Tricore to go into hold mode again. But in any case Tricore will perform at least one external bus cycle before going into hold mode again.
$\overline{\text{HLDA}}$	A23	I/O	PUC	**EBU_LMB Hold Acknowledge Input / Output** In External Master Mode: OutPut. High during normal operation.When Tricore enters hold mode, it sets $\overline{\text{HLDA}}$ to low after releasing the bus. On exit of hold mode, Tricore first sets $\overline{\text{HLDA}}$ to high and then goes onto the bus again (to avoid collisions). In External Slave Mode: Input. A high-to-low transition at this pin releases Tricore from hold mode.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
$\overline{\text{BREQ}}$	B22	O	PUC	**EBU_LMB Bus Request Output** In External Master Mode: High during normal operation.Tricore activates $\overline{\text{BREQ}}$ earliest one clock cycle after activating $\overline{\text{HLDA}}$, if it has to perform an external bus access. If Tricore has regained the bus, $\overline{\text{BREQ}}$ is set to high one clock cycle after deactivation of $\overline{\text{HLDA}}$. In External Slave Mode: This signal is high as long as Tricore operates from internal memory. When it detects that an external access is required, it sets $\overline{\text{BREQ}}$ to low and waits for signal $\overline{\text{HLDA}}$ to become low. $\overline{\text{BREQ}}$ will go back to high when the slave has backed off the bus after it was requested to go into hold mode.
$\overline{\text{RMW}}$	AB23	I/O	PUC	**EBU_LMB Read-Modify-Write Signal Line**
$\overline{\text{BAA}}$	A22	O	PUC	**EBU_LMB Burst Address Advance Output** For advancing address in a burst flash access
$\overline{\text{ADV}}$	B19	O	PUC	**EBU_LMB Burst Flash Address Valid Output**
ACLK	M23	O	—	**EBU_LMB Additional Clock Output** Additional clock running equal, 1/2, 1/3 or 1/4 frequency of EBUCLK
$\overline{\text{CMDELAY}}$	B21	I	PUC	**EBU_LMB Command Delay Input** For inserting delays between address and command.
TEST MODE	AE9	I	PDC	**Test Mode Select Input** For normal operation of the TC11IB, this pin should be connected to V_{SS}.
$\overline{\text{TM}}$ CTRL1	C15	I	PUB	**Test Mode Control Input 1** For normal operation of the TC11IB, this pin should be connected to V_{DDP}.
$\overline{\text{TM}}$ CTRL2	C12	I	PUB	**Test Mode Control Input 2** For normal operation of the TC11IB, this pin should be connected to V_{DDP}.

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
CLK42	D12	I	PDC	**Test Clock 42 MHz Input** For normal operation of the TC11IB, this pin should be connected to V_{SS}.
PLL96 CTRL	B15	O	—	**Test PLL96 Analog Output** For normal operation of the TC11IB, this pin must not be connected.
PLL42 CTRL	B12	O	—	**Test PLL42 Analog Output** For normal operation of the TC11IB, this pin must not be connected.
XTAL1 **XTAL2**	A15 A14	I O	— —	**Oscillator/PLL/Clock Generator Input/Output Pins** XTAL1 is the input to the main oscillator amplifier and input to the internal clock generator. XTAL2 is the output of the main oscillator amplifier circuit. For clocking the device from an external source, XTAL1 is driven with the clock signal while XTAL2 is left unconnected. For crystal oscillator operation XTAL1 and XTAL2 are connected to the crystal with the appropriate recommended oscillator circuitry.
V_{DDOSC}	B14	—	—	**Main Oscillator Power Supply (1.8V)**
V_{SSOSC}	C14	—	—	**Main Oscillator Ground**
$V_{DDPLL96}$	B13	—	—	**PLL96 Power Supply (1.8V)**
$V_{SSPLL96}$	A13	—	—	**PLL96 Ground**
$V_{DDPLL42}$	A12	—	—	**Test PLL42 Power Supply (1.8V)** For normal operation of the TC11IB, this pin must not be connected.
$V_{SSPLL42}$	C13	—	—	**Test PLL42 Ground** For normal operation of the TC11IB, this pin must be connected to V_{SS}.
V_{LMUREF}	A16	—	—	**LMU Reference Voltage** This pin has to be connected to V_{SS}
V_{COMREF}	AD10	—	—	**ComDRAM Reference Voltage** This pin has to be connected to V_{SS}
V_{DDDRAM}	A17, AF13	—	—	**eDRAM Power Supply (1.8V)**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
V_{DD}	H1 W1 T2,B4 B11 D6,F4 D10 D17 D21 F23 K4 K23 U4 U23 AA4 AA23 AC6 AC10 AC17 AC21	—	—	**Core and Logic Power Supply (1.8V)**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
V_{DDP}	D8, D11, D14, D16, D19, H4, H23, L4, L23, N4, P23, T4, T23, W4, W23, AC8, AC11, AC13, AC16, AC19	—	—	**Ports Power Supply (3.3V)**

Table 1-3 Pin Definitions and Functions (cont'd)

Symbol	Pin	In Out	PU/ PD[1)]	Functions
V_{SS}	D4 D9 D13 D18 D23 J4 J23 N23 P4,V4 V23 AC4 AC9 AC14 AC18 AC23 L11 to L16, M11 to M16, N11 to N16, P11 to P16, R11 to R16, T11 to T16	—	—	**Ground**
N.C.	D15, A26, AE26, AF9, AF26	—	—	**Not Connected** These pins must not be connected.

1) Refers to internal pull-up or pull-down device connected and corresponding type. The notation '—' indicates that the internal pull-up or pull-down device is not enabled.

2 TC11IB Processor Architecture

The Central Processing Unit (CPU) of the TC11IB is based on the Infineon TriCore 32-bit microcontroller-DSP processor core architecture. It is optimized for real-time embedded systems, and combines:

- Reduced Instruction Set Computing (RISC) architecture
- Digital signal processing (DSP) operations and data structures
- Real-time responsiveness

The RISC load/store architecture provides high computational bandwidth with low system cost. Its superscalar design has three pipelines.

The TC11IB CPU is a Harvard-style architecture, with separate address and data buses for program and data memories. There are special instructions for common DSP operations and hardware-assisted data structure index generation for circular buffers (useful for filters) and bit-reversed indexing (useful for Fast Fourier Transforms). These features make it possible to efficiently analyze complex real-world signals. With the implementation of MMU (Memory Management Unit) in CPU, an embedded system based on Windows CE operating system is easier to build.

The CPU’s interrupt-processing architecture combines the quick responsiveness associated with microcontrollers with a high degree of interrupt-service flexibility. The architecture of the CPU minimizes interrupt latency by having few uninterruptable multi-cycle instructions, by supporting fast context switching, and supporting task-based memory protection. The combination of the interrupt-processing capabilities of the CPU and the Peripheral Control Processor (PCP) provide the system designer with tools to meet even the most demanding hard-deadline real-time scheduling requirements simply and efficiently.

While the TriCore architecture employs 32-bit Instruction formats, frequently-used instructions have an optional 16-bit instruction format. This results in smaller code size, and faster code bandwidth. Additional benefits of this approach include lowered program memory requirements, lower system cost, and less power consumption.

2.1 Central Processing Unit

This section provides an overview of the TC11IB Central Processing Unit (CPU) architecture. The basic features include the following.

- Data paths: 32 bits throughout
- Address space: 4 Gigabytes, unified, for data, program, and I/O
- Instruction formats: mixed 32-bit and 16-bit formats
- Low interrupt latency and flexible interrupt prioritization scheme
- Fast automatic context switching
- Separate multiply-accumulate unit
- Saturating integer arithmetic
- Bit-handling operations
- Packed-data operations
- Zero-overhead looping
- Flexible power management
- Byte and bit addressing
- Little-endian byte ordering
- Precise exceptions

Figure 2-1 illustrates the architecture of the TC11IB's Central Processing Unit (CPU). It is comprised of an Instruction Fetch Unit, an Execution Unit, a General Purpose Register File, and several peripheral interfaces.

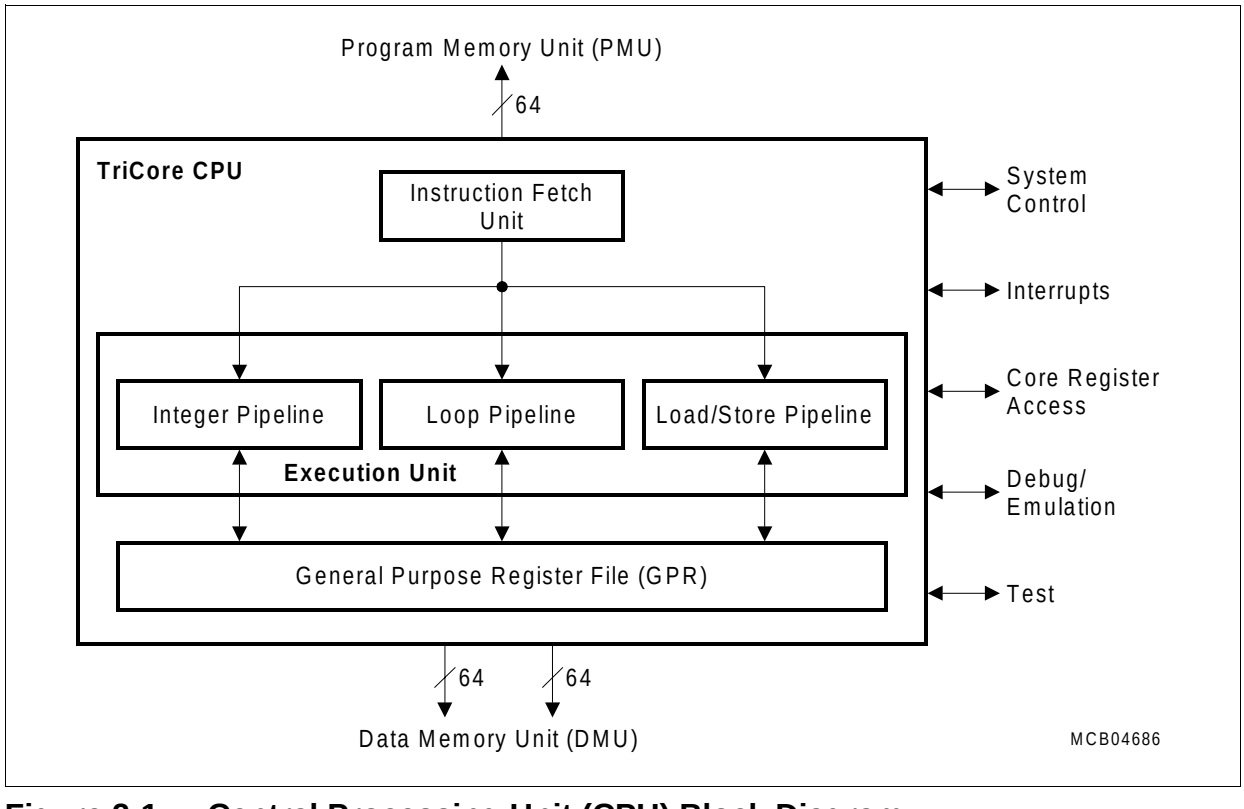


Figure 2-1 Central Processing Unit (CPU) Block Diagram

2.1.1 Instruction Fetch Unit

Figure 2-2 shows the Instruction Fetch Unit. It prefetches and aligns incoming instructions from the 64-bit wide Program Memory Unit (PMU). The Issue Unit directs the instruction to the appropriate pipeline. The Instruction Protection Unit checks the validity of accesses to the PMU and also checks for instruction breakpoint conditions. The PC Unit is responsible for updating the issue and prefetch program counters.

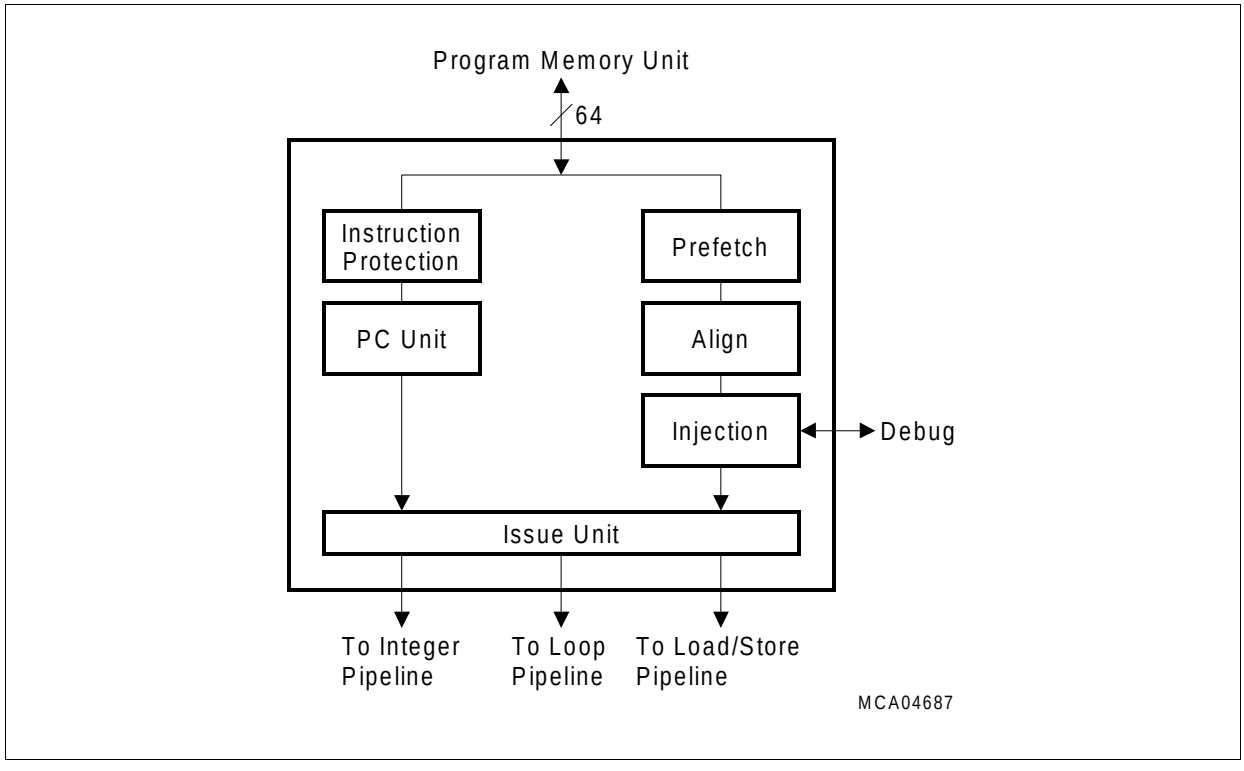


Figure 2-2 Instruction Fetch Unit

2.1.2 Execution Unit

As shown in **Figure 2-3**, the Execution Unit contains the Integer Pipeline, the Loop Pipeline, and the Load/Store Pipeline.

The Integer Pipeline and Load/Store Pipeline have four stages: Fetch, Decode, Execute, and Write-back. The Execute stage may extend beyond one cycle to accommodate multi-cycle operations such as load instructions.

The Loop Pipeline has two stages: Decode and Write-back.

All three pipelines operate in parallel, permitting up to three instructions to execute in one clock cycle.

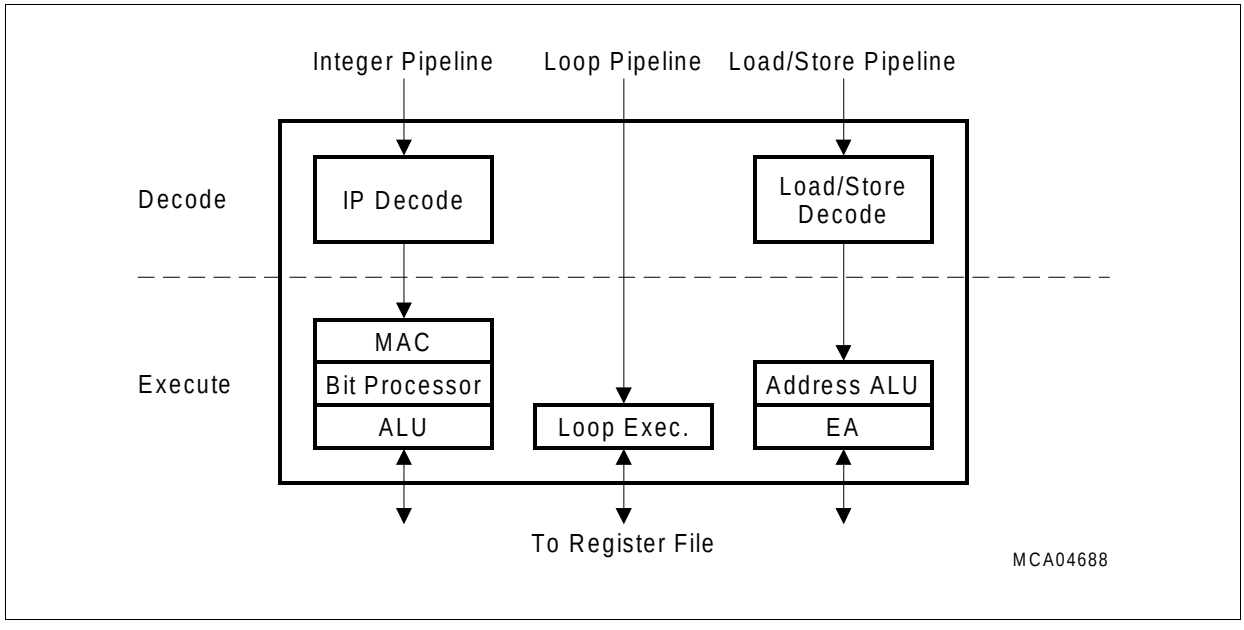


Figure 2-3 Execution Unit

Figure 2-3 introduces the following acronyms and abbreviations:

- IP Decode - Instruction Prefetch and Decode
- MAC - Multiply-Accumulate Unit
- ALU - Arithmetic/Logic Unit
- Loop Exec. - Loop Execution Unit
- EA - Effective Address

2.1.3 General Purpose Register File

The CPU has a General Purpose Register (GPR) file, divided into an Address Register File (registers A0 through A15) and a Data Register File (registers D0 through D15).

The data flow for instructions issued to/from the Load/Store Pipeline is steered through the Address Register File. The data flow for instructions issued to/from the Integer Pipeline and for data load/store instructions issued to/from the Load/Store Pipeline is steered through the Data Register File.

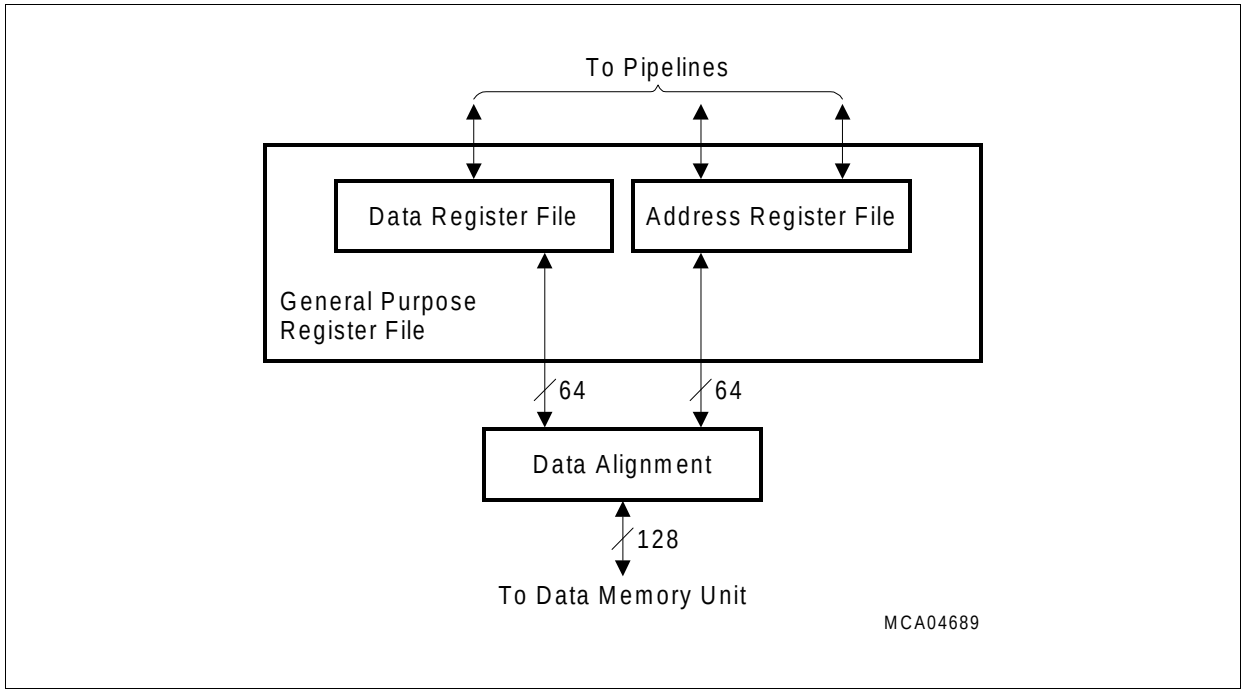


Figure 2-4 General Purpose Register File

2.1.4 Program State Registers

The program state registers consist of 32 General Purpose Registers (GPRs), two 32-bit registers with program status information (PCXI and PSW), and a Program Counter (PC). PCXI, PSW, and PC are Core Special Function Registers (CSFRs).

As shown in **Figure 2-5**, the 32 General Purpose Registers are divided into sixteen 32-bit data registers (D0 through D15) and sixteen 32-bit address registers (A0 through A15).

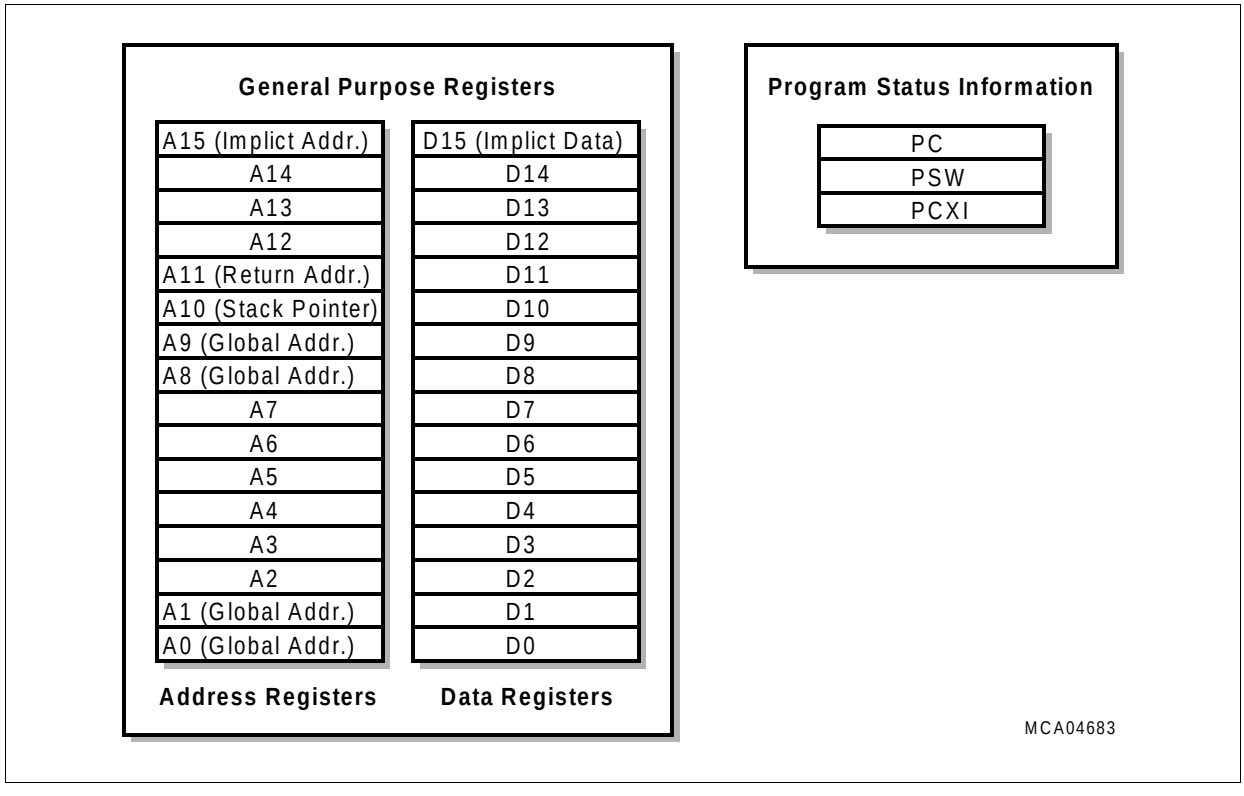


Figure 2-5 Program State Registers

Four GPRs have special functions: D15 is used as an implicit data register, A10 is the Stack Pointer (SP), A11 is the return address register, and A15 is the implicit address register.

Registers 0-7 are called the lower registers and 8-15 are called the upper registers.

Registers A0 and A1 in the lower address registers and A8 and A9 in the upper address registers are defined as system global registers. These registers are not included in either context partition, and are not saved and restored across calls or interrupts.The operating system normally uses them to reduce system overhead.

The PCXI and PSW registers contain status flags, previous execution information, and protection information.

2.1.5 Data Types

The TriCore instruction set supports operations on booleans, bit-strings, characters, signed fractions, addresses, signed and unsigned integers, integers with saturation and single-precision floating-point numbers. Most instructions work on a specific data type, while others are useful for manipulating several data types.

2.1.6 Addressing Modes

Addressing modes allow load and store instructions to efficiently access simple variables and data elements within data structures such as records, randomly and sequentially accessed arrays, stacks, and circular buffers. Simple variables and data elements are 1, 8, 16, 32 or 64 bits wide.

Addressing modes provide efficient compilation of programs written in the C programming language, easy access to peripheral registers, and efficient implementation of typical DSP data structures. Hardware-assisted DSP data structures include circular buffers for filters and bit-reversed indexing for FFTs. The following seven addressing modes are supported in the TriCore architecture:

- Absolute
- Base + Short Offset
- Base + Long Offset
- Pre-increment or pre-decrement
- Post-increment or post-decrement
- Circular (modulo)
- Bit-Reverse

2.1.7 Instruction Formats

The CPU architecture supports both 16-bit and 32-bit instruction formats. All instructions have a 32-bit format. The 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use and included to reduce code space.

2.1.8 Tasks and Contexts

Throughout this document, the term *task* refers to an independent thread of control: Software-Managed Tasks (SMTs) and Interrupt Service Routines (ISRs).

Software-Managed Tasks are created through the services of a real-time kernel or operating system and dispatched under the control of scheduling software. Interrupt Service Routines (ISRs) are dispatched by hardware in response to an interrupt. An ISR is the code that is invoked by the processor directly on receipt of an interrupt. Software-Managed Tasks are sometimes referred to as user tasks, assuming that they will execute in User Mode.

Each task is allocated its own permission level. The individual permissions are enabled or disabled primarily by I/O mode bits in the Program Status Word (PSW).

The processor state associated with a task is called the task's *context*. The context includes everything the processor needs in order to define the current state of the task. The system saves the current task's context when another task is about to run, and restores the task's context when the task is to be resumed. The context includes the Program State Registers. The CPU efficiently manages and maintains the contexts of the tasks through hardware.

2.1.8.1 Upper and Lower Contexts

The context is subdivided into the Upper Context and the Lower Context, as illustrated in **Figure 2-6**. The Upper Context consists of the upper address registers, A10 – A15, and the upper data registers, D8 – D15. These registers are designated as non-volatile, for purposes of function calling. The Upper Context also includes the PCXI and PSW registers. The Lower Context consists of the lower address registers, A2 through A7, the lower data registers, D0 through D7, and saved PC, and again the PCXI register.

Both Upper and Lower Contexts include a Link Word contained in register PCXI. Contexts are saved in fixed-size memory areas (see **Section 2.1.8.2**); they are linked together *via* the link word.

The Upper Context is saved automatically on Interrupts. It is also saved on CALL instructions and restored on RETURN instructions. The Lower Context must be saved and restored by the ISR if the ISR needs to use more registers than are available in the Upper Context.

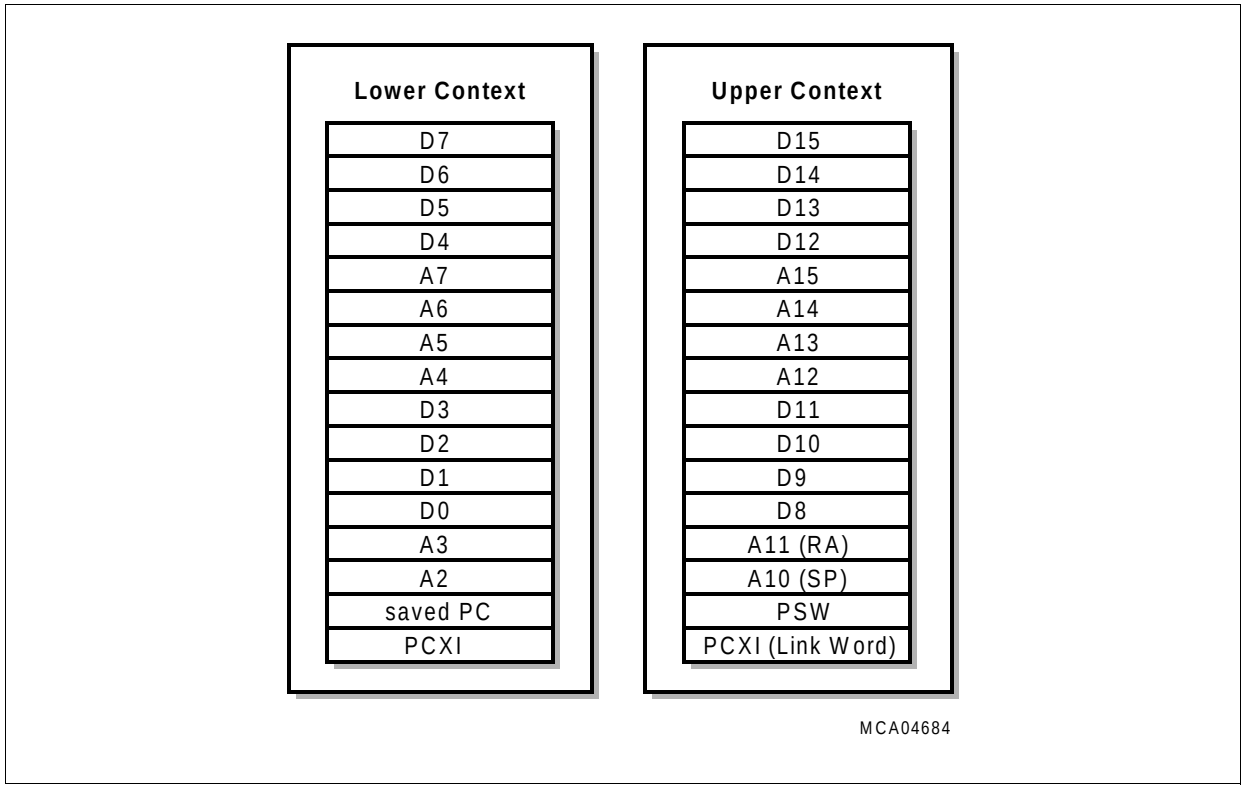


Figure 2-6 Upper and Lower Contexts

2.1.8.2 Context Save Areas

The architecture uses linked lists of fixed-size Context Save Areas (CSAs) which accommodate systems with multiple interacting threads of control. A CSA is sixteen words of memory storage, aligned on a 16-word boundary. A single CSA can hold exactly one Upper or one Lower Context. Unused CSAs are linked together on a free list. They are allocated from the free list as needed and returned to it when no longer needed. Allocation and freeing are handled transparently by the processor. They are transparent to the applications code. Only system initialization code and certain operating system exception-handling routines need to access the CSAs or their lists explicitly. The number of CSAs that can be used is limited only by the size of the available data memory.

2.1.8.3 Fast Context Switching

The TC11IB CPU uses a uniform context-switching method for function calls, interrupts, and traps. In all cases, Upper Context of the task is automatically saved and restored by hardware. Saving and restoring of the Lower Context is left as an option for the new task. An explanation of CPU management of the contexts can be found in **Section 2.2.2**.

Fast context switching is further enhanced by the TriCore's unique memory subsystem design, which allows a complete Upper or Lower Context to be saved in as little as two clock cycles.

2.1.9 Interrupt System

An interrupt request can be generated by the TC11IB on-chip peripheral units or it can be generated by external events. Requests can be targeted to either the CPU, or to the Peripheral Control Processor (PCP).

In order to better differentiate the programmable stages of interrupt processing available in the TC11IB, this document refers to an interrupt-triggering event as an Interrupt Service Request. The TC11IB interrupt system evaluates service requests for priority and to identify whether the CPU or PCP should receive the request. The highest-priority service request is then presented to the CPU (or PCP) by way of an interrupt.

In specific contexts where this level of formality is not required, the term Interrupt is used generally to mean an event directed to the CPU, while the term service request describes an event that can be directed to either the CPU or the PCP.

For a CPU interrupt, the entry code for the Interrupt Service Routine (ISR) is contained in an Interrupt Vector Table. Each entry in this table corresponds to a fixed-size code block. (If an ISR requires more code than fits in an entry, it must include a jump instruction to vector it to the rest of the ISR elsewhere in memory.) Each interrupt source is assigned an interrupt priority number. All priority numbers are programmable. The ISR uses the priority number to determine the location of the entry code block.

The prioritization of service routines enables nested interrupts and the use of interrupt priority groups. See **Chapter 15** for more information.

2.1.10 Trap System

Trap events break the normal execution of code much like interrupts. But traps are different from interrupts in these ways:

- Trap Service Routines (TSR) reside in the Trap Vector Table, separate from the Interrupt Vector Table.
- A trap does not change the CPU’s interrupt priority.
- Traps cannot be disabled by software, and are always active.

A trap occurs as a result of an exception within one of the following classes of events.

- Reset
- Internal protection
- Instruction errors
- Context management
- Internal bus and peripheral errors
- Assertion
- System call
- Non-maskable interrupt

Each entry in the Trap Vector Table corresponds to a fixed-size code block. (If a TSR requires more code than fits in an entry, it must include a jump instruction to vector it to

the rest of the TSR located elsewhere in memory.) When a trap is taken, its Trap Identification Number (TIN) is placed in data register D15. The trap handler uses the TIN to identify the cause of the trap. During trap arbitration, the pending trap with the lowest TIN will be chosen to execute. See **Chapter 16** for more information.

2.1.11 Protection System

There are two protection systems in the TC11IB. A memory-access protection system protects code and data memory regions, as described in **Section 2.1.11.1** and **Section 2.1.11.2**. Access to sensitive system registers is protected by hardware against system malfunctions, as described in **Section 2.1.11.3**.

2.1.11.1 Permission Levels

Each task can be assigned a specific permission level. Individual permissions are enabled through the I/O Mode bits in the Program Status Word (PSW). The three permission levels are listed here, in decreasing order of restrictiveness.

- **User-0 Mode**
  - Used for tasks that do not access peripheral devices.
  - Tasks at this level do not have permission to enable or disable interrupts.
- **User-1 Mode**
  - Used for tasks that access common, unprotected peripherals.
  - Accesses typically include read/write accesses to serial ports and read accesses to timers and most I/O status registers.
  - Tasks at this level may disable interrupts.
- **Supervisor Mode**
  - Permits read/write access to system registers and all peripheral devices.
  - Tasks at this level may disable interrupts.

2.1.11.2 Memory Protection Model

The Memory Protection Model of the CPU is based on address ranges, where each address range has an associated permission setting. Address ranges and their associated permissions are specified in identical sets of tables residing in the Core Special Function Register (CSFR) space. Each set is referred to as a Protection Register Set (PRS).

The TC11IB incorporates two sets of Protection Register Sets each for code and data memory. The number of sets is implementation-specific. Other TriCore products may have implemented a different number (up to four) of Protection Register Sets.

When the protection system is enabled, the CPU checks every load/store or instruction fetch address before performing the access. Legal addresses must fall within one of the ranges specified in the currently selected PRS, and permission for that type of access must be present in the matching range.

2.1.11.3 Watchdog Timer and ENDINIT Protection

Registers that control basic TC11IB configuration and operation can be protected via a special End-of-Initialization (ENDINIT) bit. The ENDINIT bit globally protects those TC11IB registers that control basic system configuration against unintentional modification. Write accesses to registers protected via this ENDINIT-bit are prohibited as long as this bit is set to 1. To clear the bit and to enable access to these registers again, a special password-protected access sequence to the Watchdog Timer registers must be performed. The bit must be set to 1 again within a defined time-out period, otherwise a system malfunction is assumed to have occurred, and the Watchdog Timer triggers a reset of the TC11IB. See **Chapter 20** for more details.

2.1.12 Reset System

Several events will cause the TC11IB system to be reset:

- **Power-On Reset**
  - Activated through an external pin when the power to the device is turned on (also called cold reset)
- **Hard Reset**
  - Activated through an external pin ($\overline{\text{HDRST}}$) during run time (also called warm reset)
- **Soft Reset**
  - Activated through a software write to a reset-request register, which has a special protection mechanism to prevent accidental access
- **Watchdog Timer Reset**
  - Activated through an error condition detected by the Watchdog Timer
- **Wake-up Reset**
  - Activated through an external pin to wake the device from a power saving mode

A status register allows the CPU to check which of the triggers caused the reset.

2.2 Processor Registers

The processor contains general purpose registers to store instruction operands. It has special purpose registers for managing the state of the processor itself.

The CPU's operations are controlled by a set of Core Special Function Registers (CSFRs). These registers also provide status information about its operation. The CSFRs are split into the following groups:

- Program State Information
- Context Management
- Stack Management
- Interrupt and Trap Control
- System Control
- Memory Protection
- Debug Control

The following sections summarize these registers. The CSFRs are complemented by a set of General Purpose Registers (GPRs). **Table 2-1** shows all CSFRs and GPRs.

Table 2-1 Core Register Map

Register Name	Description
D0 – D15	General Purpose Data Registers
A0 – A15	General Purpose Address Registers
PSW	Program Status Word
PCXI	Previous Context Information Register
PC	Program Counter
FCX	Free CSA List Head Pointer
LCX	Free CSA List Limit Pointer
ISP	Interrupt Stack Pointer
ICR	ICU Interrupt Control Register
BIV	Interrupt Vector Table Pointer
BTV	Trap Vector Table Pointer
SYSCON	System Configuration Register
DPRx_0 – DPRx_3	Data Segment Protection Registers for Set x (x = 0, 1)
CPRx_0 – CPRx_1	Code Segment Protection Registers for Set x (x = 0, 1)
DPMx_0 – DPMx_3	Data Protection Mode Register for Set x (x = 0, 1)
CPMx_0 – CPMx_1	Code Protection Mode Register for Set x (x = 0, 1)
DBGSR	Debug Status Register

Table 2-1 Core Register Map (cont'd)

Register Name	Description
EXEVT	External Break Input Event Specifier
SWEVT	Software Break Event Specifier
CREVT	Core SFR Access Event Specifier
TRnEVT	Trigger Event n Specifier (n = 0, 1)

The CPU accesses the CSFRs through two instructions: MFCR and MTCR. The MFCR instruction (Move From Core Register) moves the contents of the addressed CSFR into a data register. MFCR can be executed on any privilege level. The MTCR instruction (Move To Core Register) moves the contents of a data register to the addressed CSFR. To prevent unauthorized writes to the CSFRs, the MTCR instruction can be executed on Supervisor privilege level only.

The CSFRs are also mapped into the top of Segment 15 in the memory address space. This mapping makes the complete architectural state of the CPU visible in the address map. This feature provides efficient debug and emulator support.

*Note: The CPU is **not allowed** to access the CSFRs through this mechanism — it must use the MFCR and MTCR instructions. Trying to access the CSFRs through normal load and store instructions results in a MEM trap.*

The instruction set provides no single-bit, bit field, or load-modify-store accesses to the CSFRs. The only other instruction affecting a CSFR, is the RSTV instruction (Reset Overflow Flags), which resets only the overflow flags in the PSW, without modifying any of the other PSW bits. This instruction can be executed at any privilege level.

Note: Access to the Core SFRs through their mapped addresses in segment 15 is implemented primarily for debug purposes. Special attention needs to be paid when accessing these registers. It is strongly advised to not write to the CSFRs while the core is executing. Reading the registers while the core is running does not guarantee coherent status information.
A mid-range or high-range emulator can use the external bus as a fast route to the internal FPI Bus. However, certain restrictions are placed on this mode of operation regarding access to the CSFRs and GPRs: The external bus cannot be used to access state in the core (GPRs and CSFRs) while the core is running (not halted) and is configured to perform accesses to the external bus.

Figure 2-7 shows the General Purpose Registers (GPRs). The 32-bit wide GPRs are split evenly into sixteen data registers, or DGPRs, (D0 to D15) and sixteen address registers, or AGPRs, (A0 to A15). Separation of data and address registers facilitates efficient performance of arithmetic and memory operations in parallel. Several instructions interchange information between data and address registers in order, for example, to create or derive table indexes. 64-bit values can be represented by

concatenating two consecutive double-word-aligned data registers. Eight such extended-size registers (E0, E2, E4, E6, E8, E10, E12, and E14) are available.

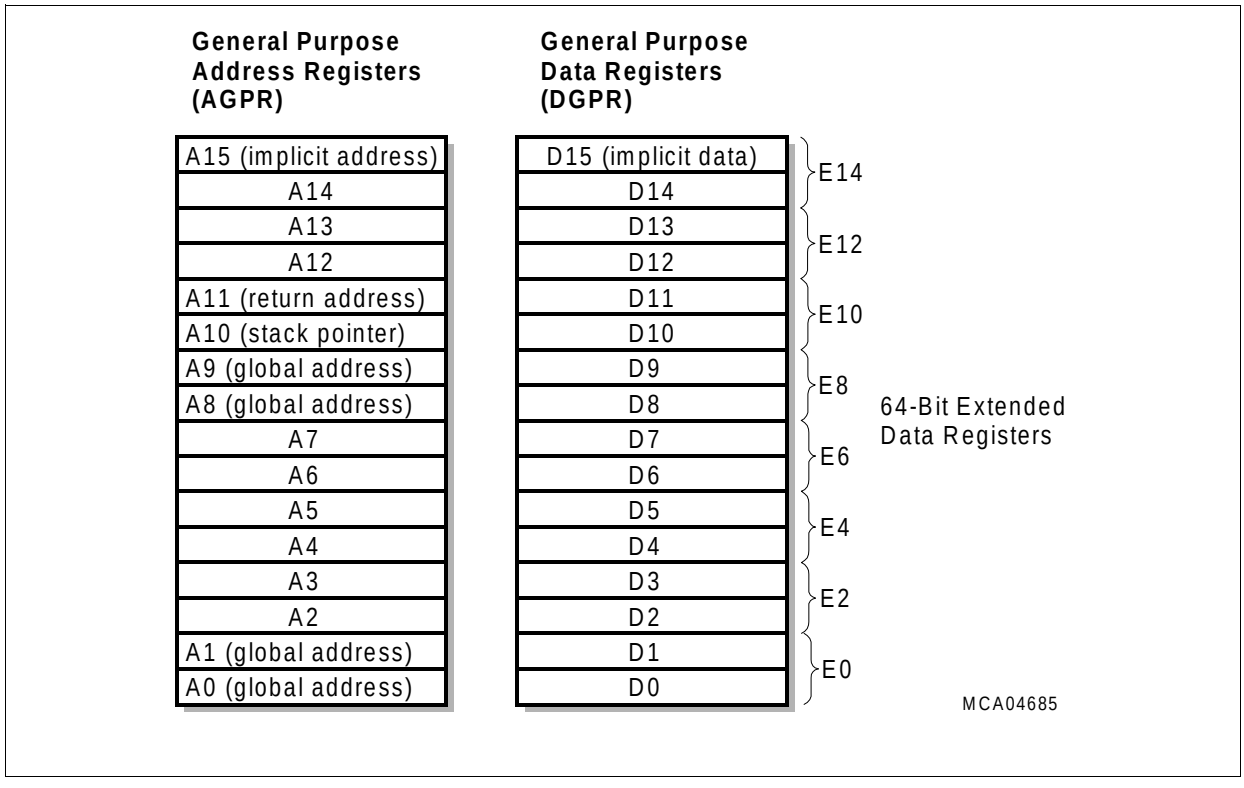


Figure 2-7 General Purpose Registers (GPRs)

As shown in **Figure 2-7**, registers A0, A1, A8, and A9 are defined as System Global Registers. Their contents are not saved and restored across calls, traps, or interrupts. Register A10 is used as the Stack Pointer (SP) register. A11 is used to store the return address (RA) for calls and linked jumps and to store the return program counter (PC) value for interrupts and traps as part of the Upper Context.

The 32-bit instructions have unlimited use of the GPRs. However, many 16-bit instructions implicitly use A15 as their address register and D15 as their data register to make the encoding of these instructions into 16 bits possible.

There are no separate floating-point registers — the data registers are used to perform floating-point operations. Floating-point data is saved and restored automatically using the fast context-switching capabilities of the TC11IB.

The GPRs are an essential part of a task's context. When saving or restoring a task's context to and from memory, the context is split into the Upper Context and Lower Context as shown in **Figure 2-6**. Registers A2 through A7 and D0 through D7 are part of the Lower Context. Registers A10 through A15 and D8 through D15 are part of the Upper Context.

2.2.1 Program State Information Registers

The PC, PSW, and PCXI registers hold and reflect Program State Information. When saving and restoring a task's context, the contents of these registers are saved and restored or modified during this process.

2.2.1.1 Program Counter (PC)

The Program Counter (PC) holds the address of the instruction which is currently fetched and forwarded to the CPU pipelines. The CPU handles updates of the PC automatically.

Software can use the current value of the PC for various tasks, such as performing code address calculations. Reading the PC through software executed by the CPU must only be done with an MFCR instruction. Explicit writes to the PC through an MTCR instruction must not be done due to possible unexpected behavior of the CPU.

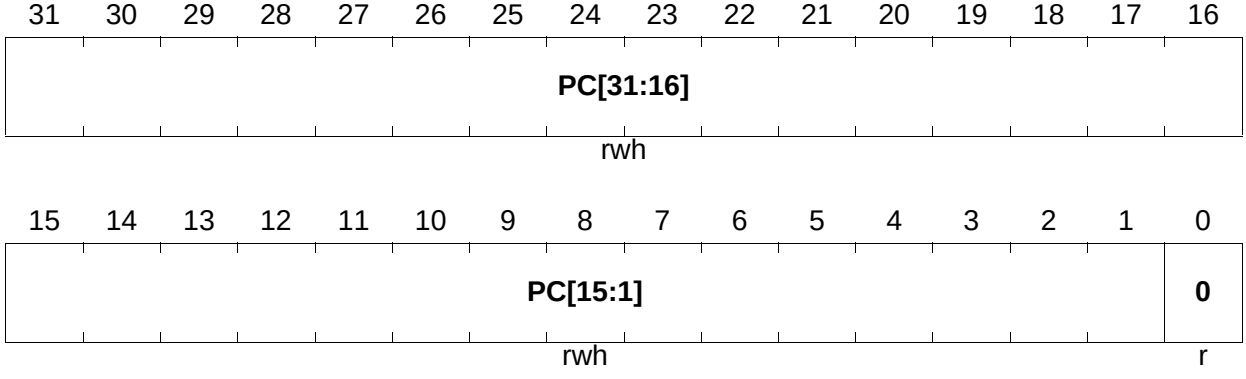
Note: The CPU must not perform Load/Store instructions to the mapped address of the PC in Segment 15. A MEM trap will be generated in such a case.

Note: Reading the PC while the Core is executing, either through an MFCR instruction or via its mapped address in Segment 15 (see below), will return a value which is representative of where the code is currently executed from, however, it is not guaranteed that the value returned will always correspond to an instruction that has been or will be executed. For example, it is possible for the PC to point to the target of a predicted branch which is subsequently resolved as mispredicted. Thus, the branch target instruction will not be executed; however, it should be possible to implement a statistical profile/coverage report with some degree of error by sampling the PC value while the CPU is running.

In Debug Mode, explicit read and write operations to the PC can be performed using its mapped address in Segment 15. This must only be done through an LMB Bus master other than the CPU itself (through the DMU). Several restrictions apply to this operation:

- *Writing to the PC while the Core is executing* is non-deterministic and the user is strongly advised not to do so. The correct sequence the user should adopt is: halt the Core, modify the PC, remove Core from Halt mode.
- *Reading the PC while the Core is halted* will return the PC of the first instruction to be executed once the Core is released from Halt mode. The only exception to this is if an interrupt or asynchronous trap is received by the Core immediately after it is removed from Halt mode prior to the first instruction being executed.
- *Writing to the PC while the Core is halted* will modify the PC in a deterministic way. the new value will be the PC of the first instruction to be executed once the Core is released from Halt mode. The only exception to this is if an interrupt or asynchronous trap is received by the Core immediately after it is removed from Halt mode prior to the first instruction being executed.

PC
Program Counter **Reset Values:** **Boot ROM Boot: DFFF FFFC$_H$**
External Memory Boot: A000 0000$_H$
Emulator Boot: DE00 0000$_H$



Field	Bits	Type	Description
PC	[31:1]	rwh	**Program Counter**
0	0	r	**Reserved**

Note: Bit 0 of the PC register is a read-only bit, hard-wired to 0. This ensures that only half-word aligned addresses can be placed into the PC (instructions can only be aligned to half-word addresses).

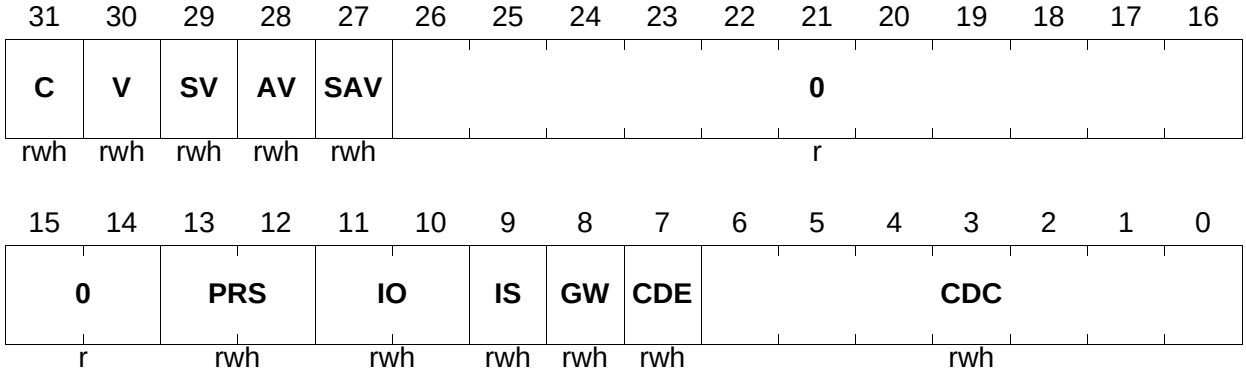
2.2.1.2 Program Status Word (PSW)

The Program Status Word (PSW) register holds the instruction flags and the control bits for a number of options of the overall protection system.

A special instruction is available that affects only the overflow flag bits in register PSW. The RSTV (Reset Overflow Flags) instruction clears bits V, SV, AV and SAV in PSW without modifying any other PSW bit.

PSW
Program Status Word

Reset Value: 0000 0B80$_H$



Field	Bits	Type	Description
CDC	[6:0]	rwh	**Call Depth Counter Field** The CDC field consists of two variable-width fields. The first is a mask field, consisting of a string of zero or more initial 1 bits, terminated by the first 0 bit. The remaining bits of the field are the call depth counter. 0ccccc$_B$ 6-bit counter; trap on overflow 10cccc$_B$ 5-bit counter; trap on overflow 110ccc$_B$ 4-bit counter; trap on overflow 1110cc$_B$ 3-bit counter; trap on overflow 11110c$_B$ 2-bit counter; trap on overflow 111110$_B$ 1-bit counter; trap on overflow 1111110$_B$ Trap every call (call trace mode) 1111111$_B$ Disable call depth counting When the call depth counter overflows, a trap is generated. Depending on the width of the mask field, the call depth counter can be set to overflow at any power of two boundary, from 1 to 64. Setting the mask field to 1111110$_B$ allows no bits for the counter, and causes every call to be trapped. This is used for call tracing. Setting the field to mask field to 1111111$_B$ disables call depth counting altogether.

Field	Bits	Type	Description
CDE	7	rwh	**Call Depth Count Enable** The CDE bit enables call-depth counting, provided that the CDC mask field is not all 1's. CDE is set to 1 by default, but should be cleared by the SYSCALL instruction Trap Service Routine to allow a trapped SYSCALL instruction to execute without producing another trap upon return from the trap handler. It is then set again when the next SYSCALL instruction is executed. 0 Call depth counter disabled 1 Call depth counter enabled
GW	8	rwh	**Global Register Write Permission** GW controls whether the current execution thread has permission to modify the global address registers. Most tasks and ISRs will use the global address registers as "read only" registers, pointing to the global literal pool and key data structures. However, a task or ISR can be designated as the "owner" of a particular global address register, and is allowed to modify it. The system designer must determine which global address variables are used with sufficient frequency and/or in sufficiently time-critical code to justify allocation to a global address register. By compiler convention, global address register A0 is reserved as the base register for short form loads and stores. Register A1 is also reserved for compiler use. Registers A8 and A9 are not used by the compiler, and are available for holding critical system address variables. 0 Write permission to global registers A0, A1, A8, and A9 is disabled 1 Write permission to global registers A0, A1, A8, and A9 is enabled

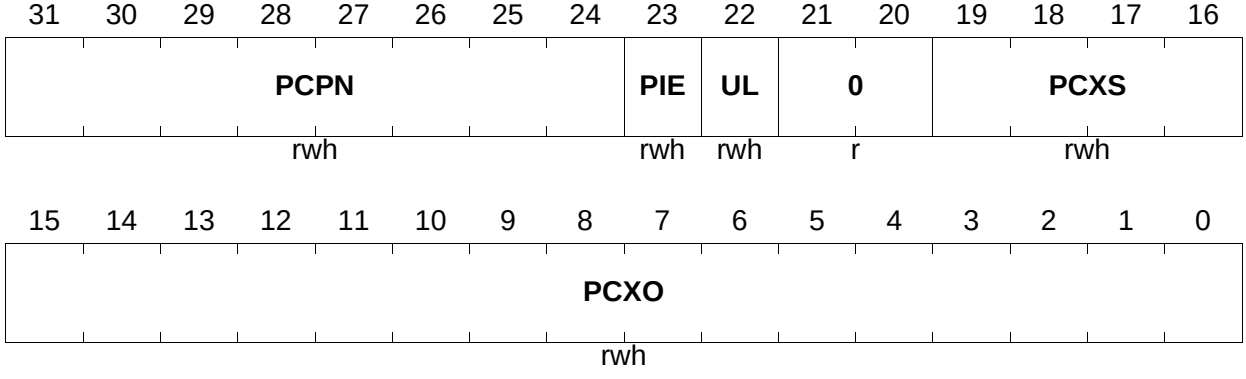
Field	Bits	Type	Description
IS	9	rwh	**Interrupt Stack Control** Determines whether the current execution thread is using the shared global (interrupt) stack or a user stack. 0 U**ser Stack**. If an interrupt is taken when the IS bit is 0, then the stack pointer register is loaded from the ISP register before execution starts at the first instruction of the Interrupt Service Routine. 1 S**hared Global Stack**. If an interrupt is taken when the IS bit is 1, then the current value of the stack pointer register is used by the Interrupt Service Routine.
IO	[11:10]	rwh	**Access Privilege Level Control** This 2-bit field selects determines the access level to special function registers and peripheral devices. 00_B **User-0 Mode**: No peripheral access. Access to segments 14 and 15 is prohibited and will result in a trap. This access level is given to tasks that need not directly access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts. 01_B **User-1 Mode**: regular peripheral access. This access level enables access to common peripheral devices that are not specially protected, including read/write access to serial I/O ports, read access to timers, and access to most I/O status registers. Tasks at this level may disable interrupts. 10_B **Supervisor Mode**. This access level enables access to all peripheral devices. It enables read/write access to core registers and protected peripheral devices. Tasks at this level may disable interrupts. 11_B **Reserved**; this encoding is reserved and is not defined.

Field	Bits	Type	Description
PRS	[13:12]	rwh	**Protection Register Set Selection** The PRS field selects one of two possible sets of memory protection register values controlling load and store operations and instruction fetches within the current process. This field indicates the current protection register set. 00 Protection register set 0 selected 01 Protection register set 1 selected 10 Reserved; don't use this combination 11 Reserved; don't use this combination
0	[26:14]	r	**Reserved**; read as 0; should be written with 0;
SAV	27	rwh	**Sticky Advance Overflow Flag** This flag is set whenever the advanced overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.
AV	28	rwh	**Advance Overflow Flag** This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the boolean exclusive of the two most significant bits of the result.
SV	29	rwh	**Sticky Overflow Flag** This flag is set when an overflow occurs. This flag remains set until it is explicitly reset by an RSTV (Reset Overflow bits) instruction.
V	30	rwh	**Overflow Flag** This flag is set when an overflow occurs.
C	31	rwh	**Carry Flag** This flag is set when a carry occurs.

2.2.1.3 Previous Context Information Register (PCXI)

This register holds information about the previous task's context, and is saved and restored together with both the Upper and the Lower Context. It also contains the Previous Context Pointer (PCX), which holds the address of the previous task's context save area (CSA).

PCXI
Previous Context Information Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
PCXO	[15:0]	rwh	**Previous Context Pointer Offset Field** The combined PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.
PCXS	[19:16]	rwh	**PCX Segment Address** This field contains the segment address portion of the PCX.
0	20, 21	r	**Reserved**; read as 0; should be written with 0;
UL	22	rwh	**Upper/Lower Context Tag** The UL context tag bit identifies the type of context saved. 0 Lower Context 1 Upper Context If the type does not match the type expected when a context restore operation is performed, a trap is generated.
PIE	23	rwh	**Previous Interrupt Enable** PIE indicates the state of the interrupt enable bit (ICR.IE) for the interrupted task.
PCPN	[31:24]	rwh	**Previous CPU Priority Number** This bit field contains the priority level number of the interrupted task.

2.2.2 Context Management Registers

The Context Management Registers (CMR) are comprised of three pointer registers, FCX, PCX, and LCX. These pointers handle context management and are used during context save/restore operations.

Each pointer register consists of two fields: a 16-bit offset and a 4-bit segment specifier. A Context Save Area (CSA) is an address range containing sixteen word locations (64 bytes). Each CSA can save one Upper Context or one Lower Context. Incrementing a CMR pointer offset value by 1 will point it at the CSA that is sixteen word locations above the previous one.

The FCX pointer register points to the head of the CSA free list. The previous context pointer (PCX) points to the CSA of the previous task. PCX is part of the previous context information register PCXI. The LCX pointer register is used to recognize impending CSA list underflows. If the value of FCX used on an interrupt or CALL instruction matches the limit value, the context-save operation will be completed, but the target address will be forced to the trap vector address that handles CSA list depletion.

2.2.2.1 Free Context List Head Pointer (FCX)

The FCX register points to the address of the next available context save area (CSA) in the linked list of CSAs. It is automatically updated on a context save operation to point to the next available CSA.

FCX
Free Context List Head Pointer **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0												FCXS			
R												rwh			

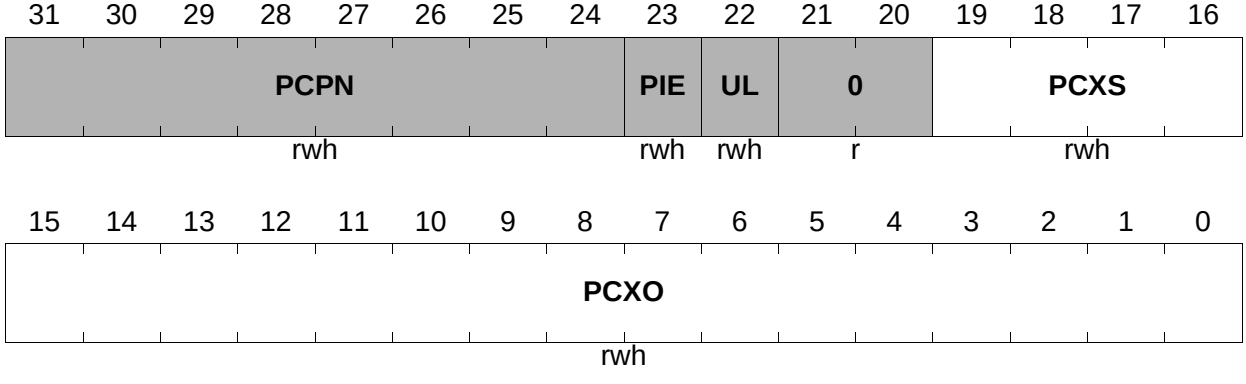
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCXO															
rwh															

Field	Bits	Type	Description
FCXO	[15:0]	rwh	**FCX Offset Address Field** The combined FCXO and FCXS fields form the FCX pointer, which points to the next available CSA.
FCXS	[19:16]	rwh	**FCX Segment Address Field** This bit field is used in conjunction with the FCXO field.
0	[31:20]	r	**Reserved**; read as 0; should be written with 0;

2.2.2.2 Previous Context Pointer (PCX)

The Previous Context Pointer (PCX) holds the address of the CSA of the previous task. PCX is part of PCXI. It is shown for easy reference. The bits not relevant to the pointer function are shaded.

PCX
Previous Context Pointer **Reset Value: 0000 0000$_H$**



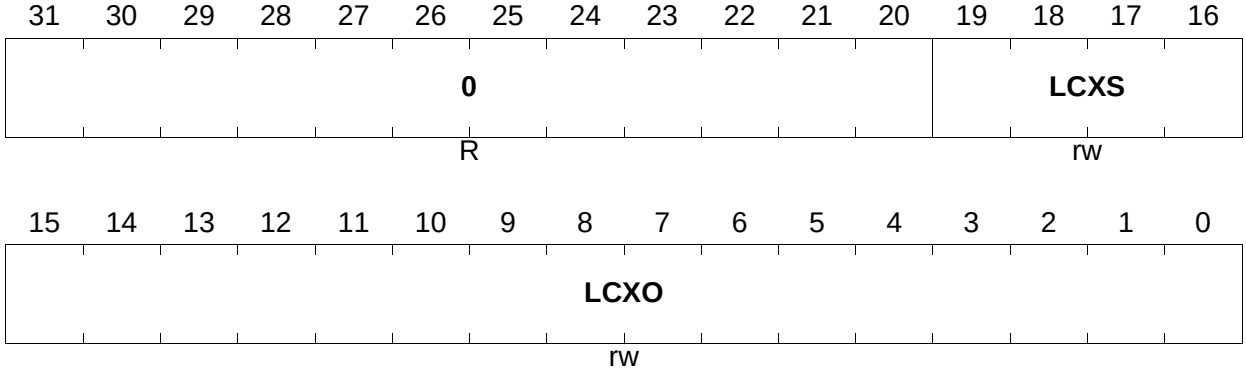
Field	Bits	Type	Description
PCXO	[15:0]	rwh	**Previous Context Pointer Offset Field** The combined PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.
PCXS	[19:16]	rwh	**PCX Segment Address** This field is used in conjunction with the PCXO field-
0	20, 21	r	**Reserved**; read as 0; should be written with 0;

Note: The shaded bit fields are described at register PCXI.

2.2.3 Free Context List Limit Pointer (LCX)

The LCX register points to the last context save area (CSA) in the linked list of free CSAs. The value is used on a context save operation to detect the usage of the last entry, and to trigger a trap to the CPU to allow proper software reaction.

LCX
Free Context List Limit Pointer **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
LCXO	[15:0]	rw	**Previous Context Pointer Offset Field** The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA.
LCXS	[19:16]	rw	**LCX Segment Address** This bit field is used in conjunction with the LCXO field.

2.2.4 Stack Management

General purpose address register A10 is designated as the Stack Pointer (SP). The initial contents of this register are usually set by an RTOS instruction when a task is created. This allows a private stack area to be assigned to individual tasks.

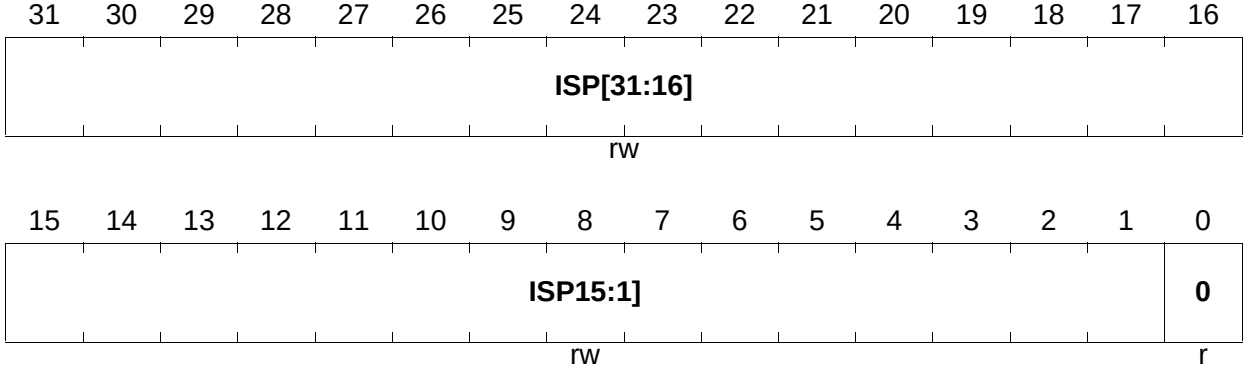
When entering Interrupt Service Routines (ISRs), the Stack Pointer is loaded with the contents of a separate register — the Interrupt Stack Pointer (ISP) — after saving its previous contents with the Upper Context. This helps to prevent interrupt service routines from accessing the private stack areas and possibly interfering with the context of software-managed tasks.

2.2.4.1 Interrupt Stack Pointer (ISP)

To separate the private stack of software managed tasks from the stack used for interrupt service routines (ISRs), an automatic switch is implemented in the TC11IB to use the Interrupt Stack Pointer (ISP) when entering ISRs. After saving the Upper Context, and with it register A10 (used as the stack pointer), register A10 is loaded with the contents of register ISP. When returning from the ISR, the previous value of the Stack Pointer is restored through the Upper Context restore operation.

Note: Register ISP is EndInit-protected!

ISP
Interrupt Stack Pointer **Reset Value: 0000 0100$_H$**



Field	Bits	Type	Description
ISP	[31:1]	rw	**Interrupt Stack Pointer**
0	0	r	**Reserved**; read as 0; should be written with 0;

2.2.5 Interrupt and Trap Control

Three CSFRs support interrupt and trap handling: the Interrupt Control Register (ICR), the Interrupt Vector Table Pointer (BIV), and the Trap Vector Table Pointer (BTV).

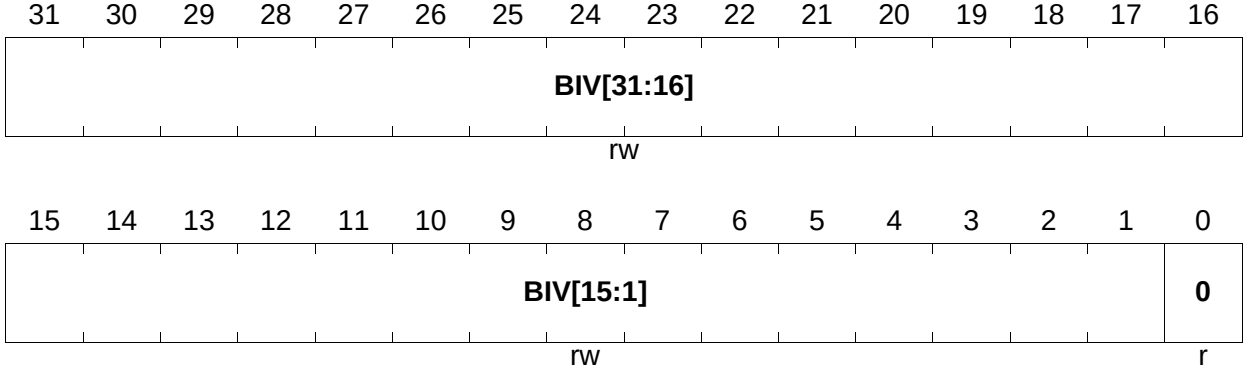
The ICR holds the current CPU priority number (CCPN), the enable/disable bit for the interrupt system, the pending interrupt priority number, and an implementation-specific control for the interrupt arbitration scheme. The other two registers hold the base addresses for the interrupt (BIV) and trap vector tables (BTV).

2.2.5.1 Interrupt Vector Table Pointer (BIV)

The BIV register points to the start address of the Interrupt Vector Table in code memory. More detailed information on the functions associated with this register and the Interrupt Vector Table can be found in **Chapter 15**.

Note: Register BIV is EndInit-protected!

BIV
Interrupt Vector Table Pointer **Reset Value: 0000 0000$_H$**



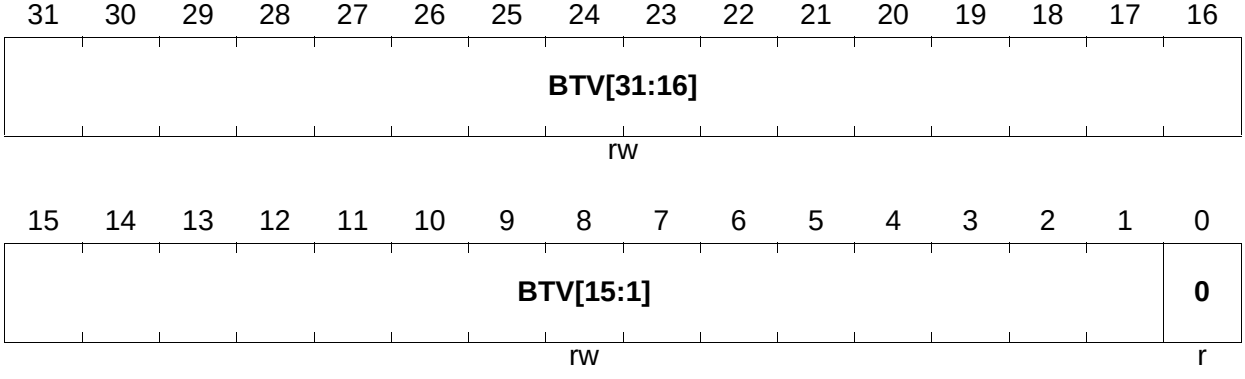
Field	Bits	Type	Description
BIV	[31:1]	rw	**Base Address of Interrupt Vector Table**
0	0	r	**Reserved**; read as 0; should be written with 0;

2.2.5.2 Trap Vector Table Pointer (BTV)

The BTV register points to the start address of the Trap Vector Table in code memory. More detailed information on the functions associated with this register and the Trap Vector Table can be found in **Chapter 16**.

Note: Register BTV is EndInit-protected,

BTV
Trap Vector Table Pointer **Reset Value: A000 0100$_H$**

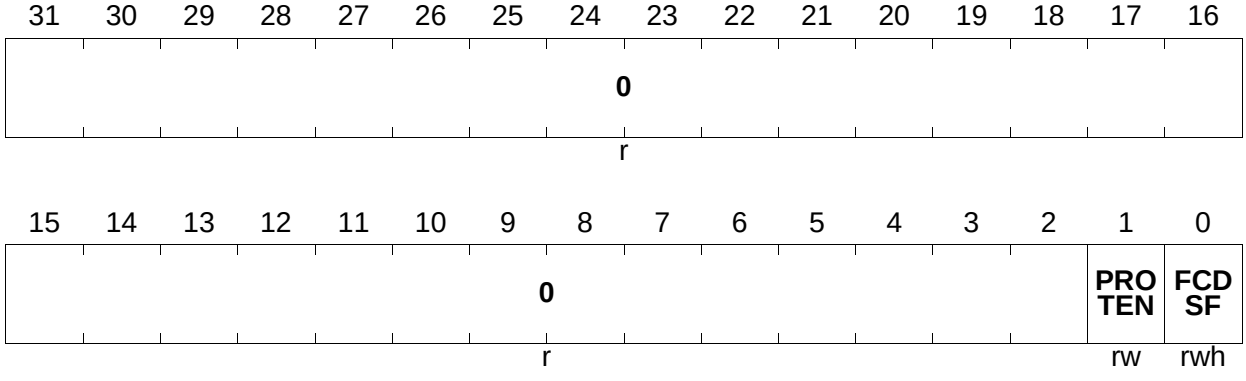


Field	Bits	Type	Description
BTV	[31:1]	rw	**Base Address of Trap Vector Table**
0	0	r	**Reserved**; read as 0; should be written with 0;

2.2.6 System Control Register

The System Configuration Control Register (SYSCON) provides the enable/disable bit for the memory protection system and a status flag for a Free Context List Depletion condition.

SYSCON
System Configuration Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
FCDSF	0	rwh	**Free Context List Depletion Sticky Flag** This sticky bit indicates that a FCD trap occurred since the bit was last cleared by software. 0 No FCD trap occurred since the last clear 1 An FCD trap occurred since the last clear
PROTEN	1	rw	**Memory Protection Enable** PROTEN enables the memory protection system. Memory protection is controlled through the memory protection register sets. Note that it is required to initialize the protection register sets prior to setting PROTEN to 1. 0 Memory Protection is disabled 1 Memory Protection is enabled
0	[31:2]	r	**Reserved**; read as 0; should be written with 0;

2.2.7 Memory Protection Registers

As described in **Section 2.1.11.2**, memory ranges are protected from unauthorized read-, write-, or instruction-fetch accesses. The TC11IB contains register sets (PRSs) that specify the addresses and the access permissions for a number of memory ranges. The TC11IB incorporates two sets each for data and code memory protection. See **Chapter 12** for detailed register descriptions.

2.2.8 Debug Registers

Six registers are implemented in the CPU to support debugging. These registers define the conditions under which a debug event is generated, the actions taken on the assertion of a debug event, and the status information supplied to the debug functions. See **Chapter 21** for detailed register descriptions.

2.2.9 CSFR Address Table

Table 2-2 lists all CSFRs of the TC11IB and their physical addresses. Except for the General Purpose Registers (GPRs), two addresses are given for each of the CSFRs. The 32-bit address represents the mapped address of the register in segment 15. Access to these mapped locations can be performed through the CPU's Slave Interface (CPS) by any LMB Bus master other than the CPU itself. The 16-bit address given for a register is the associated address when performing an access by the CPU through the MTCR and MFCR instructions.

Access modes to the CSFRs are described in the following notes and, therefore, are not contained in **Table 2-2**.

Note: The General Purpose Registers (GPRs) cannot be accessed by the CPU through MTCR and MFCR instructions. Therefore, they do not have a 16-bit address.

Note: Write accesses to CSFRs through the CPS interface by an LMB Bus master while the CPU is running might lead to unexpected behavior. It is strongly advised to write to these registers only when the CPU is halted.

Note: Read and write accesses from the LMB Bus must only be made with word-aligned word accesses. Any access not following this rule will be flagged with a bus error. The read or write operation will not be performed.

Note: Read accesses from the LMB Bus can be performed in User or Supervisor Mode. Write accesses from the LMB Bus must be performed in Supervisor Mode. A write attempt in User Mode will be flagged with a bus error. The write operation will not be performed.

*Note: Registers ISP, BIV, and BTV are EndInit-protected. To write successfully to these registers, the ENDINIT bit in register WDT_CON0 of the Watchdog Timer must be cleared. See **Chapter 20** for detailed information on the EndInit-protection.*

Table 2-2 CSFR Register Table

Register Short Name	Register Long Name	Address
Core Special Function Registers (CSFRs)		
PCXI	Previous Context Information Register	F7E1 $FE00_H$
PSW	Program Status Word	F7E1 $FE04_H$
PC	Program Counter	F7E1 $FE08_H$ / $FE08_H$
SYSCON	System Configuration Register	F7E1 $FE14_H$ / $FE14_H$
BIV	Interrupt Vector Table Pointer	F7E1 $FE20_H$ / $FE20_H$
BTV	Trap Vector Table Pointer	F7E1 $FE24_H$ / $FE24_H$
ISP	Interrupt Stack Pointer	F7E1 $FE28_H$ / $FE28_H$
ICR	ICU Interrupt Control Register	F7E1 $FE2C_H$ / $FE2C_H$
FCX	Free CSA List Head Pointer	F7E1 $FE38_H$ / $FE38_H$
LCX	Free CSA List Limit Pointer	F7E1 $FE3C_H$ / $FE3C_H$
General Purpose Registers (GPRs)		
D0	Data Register D0 (DGPR)	F7E1 $FF00_H$
D1	Data Register D1 (DGPR)	F7E1 $FF04_H$
D2	Data Register D2 (DGPR)	F7E1 $FF08_H$
D3	Data Register D3 (DGPR)	F7E1 $FF0C_H$
D4	Data Register D4 (DGPR)	F7E1 $FF10_H$
D5	Data Register D5 (DGPR)	F7E1 $FF14_H$
D6	Data Register D6 (DGPR)	F7E1 $FF18_H$
D7	Data Register D7 (DGPR)	F7E1 $FF1C_H$
D8	Data Register D8 (DGPR)	F7E1 $FF20_H$
D9	Data Register D9 (DGPR)	F7E1 $FF24_H$
D10	Data Register 10 (DGPR)	F7E1 $FF28_H$
D11	Data Register 11 (DGPR)	F7E1 $FF2C_H$
D12	Data Register 12 (DGPR)	F7E1 $FF30_H$
D13	Data Register 13 (DGPR)	F7E1 $FF34_H$
D14	Data Register 14 (DGPR)	F7E1 $FF38_H$
D15	Data Register 15 (DGPR)	F7E1 $FF3C_H$
A0	Address Register 0 (AGPR) Global Address Register	F7E1 $FF80_H$

Table 2-2 CSFR Register Table (cont'd)

Register Short Name	Register Long Name	Address
A1	Address Register 1 (AGPR) Global Address Register	F7E1 FF84$_H$
A2	Address Register 2 (AGPR)	F7E1 FF88$_H$
A3	Address Register 3 (AGPR)	F7E1 FF8C$_H$
A4	Address Register 4 (AGPR)	F7E1 FF90$_H$
A5	Address Register 5 (AGPR)	F7E1 FF94$_H$
A6	Address Register 6 (AGPR)	F7E1 FF98$_H$
A7	Address Register 7 (AGPR)	F7E1 FF9C$_H$
A8	Address Register 8 (AGPR) Global Address Register	F7E1 FFA0$_H$
A9	Address Register 9 (AGPR) Global Address Register	F7E1 FFA4$_H$
A10 (SP)	Address Register 10 (AGPR) Stack Pointer	F7E1 FFA8$_H$
A11 (RA)	Address Register 11 (AGPR) Return Address	F7E1 FFAC$_H$
A12	Address Register 12 (AGPR)	F7E1 FFB0$_H$
A13	Address Register 13 (AGPR)	F7E1 FFB4$_H$
A14	Address Register 14 (AGPR)	F7E1 FFB8$_H$
A15	Address Register 15 (AGPR)	F7E1 FFBC$_H$

2.3 Instruction Set Overview

This section provides an overview of the TriCore instruction set architecture. The basic properties and uses of each instruction type are described, as well as the selection and use of the 16-bit (short) instructions.

Note: The "TriCore Architecture Manual" describes each instruction more detailed.

2.3.1 Arithmetic Instructions

Arithmetic instructions operate on data and addresses in registers. Status information about the result of the arithmetic operations is recorded in the five status flags in the Program Status Word (PSW) register. The status flags are described in **Table 2-3**.

Table 2-3 PSW Status Flags

Status Flag	Description
C	**Carry Flag** This flag is set as the result of a carry out from an addition or subtraction instruction. Carry out can result from either signed or unsigned operations. It is also set by arithmetic shift.
V	**Overflow Flag** This flag is set when the signed result cannot be represented in the data size of the result; for example, when the result of a signed 32-bit operation is greater than 2^{31} - 1.
SV	**Sticky Overflow Flag** This flag is set when the overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.
AV	**Advance Overflow Flag** This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the Boolean exclusive-or of the two most-significant bits of the result.
SAV	**Sticky Advance Overflow Flag** This flag is set whenever the advanced overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.

The two signed overflow conditions (overflow and advance overflow) are calculated for all arithmetic instructions. In the case of packed instructions, the conditions are the OR of the conditions for each byte or half-word (parallel) operation. In the case of the multiply-accumulate instructions, the conditions are calculated after the accumulate operation. The unsigned overflow condition is carry for addition or borrow (no carry) for subtraction.

Numerically, overflow for signed 32-bit values occurs when a positive result is greater than $7FFFFFFF_H$ or a negative result is smaller than 80000000_H. Overflow for unsigned 32-bit values occurs when the result is greater than $FFFFFFFF_H$ or less than 00000000_H.

The status flags can be read by software using the Move From Core Register (MFCR) instruction and can be written using the Move to Core Register (MTCR) instruction. The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the V and SV bits, respectively, are set. The overflow bits can be cleared using the Reset Overflow Bits instruction (RSTV).

Individual arithmetic operations can be checked for overflow by reading and testing V. If it is necessary to know only if an overflow occurred somewhere in an entire block of computation, then the SV bit is reset before the block (using the RSTV instruction) and is tested after completion of the block (using MFCR). Jumping based on the overflow result can be done using a MFCR followed by a JZ.T or JNZ.T (conditional jump on the value of a bit).

The AV and SAV bits are set as a result of the exclusive OR of the two most-significant bits of the particular data type (byte, half-word, word, or double-word) of the result, which indicates that an overflow almost occurred.

Because most signal-processing applications can handle overflow by simply saturating the result, most of the arithmetic instructions have a saturating version for signed and unsigned overflow. Note that saturating versions of all instructions can be synthesized using short code sequences.

When saturation is used for 32-bit signed arithmetic overflow, if the true result of the computation is greater than $(2^{31} - 1)$ or less than -2^{31}, the result is set to $(2^{31} - 1)$ or -2^{31}, respectively. The bounds for 16-bit signed arithmetic are $(2^{15} - 1)$ and -2^{15}. The bounds for 8-bit signed arithmetic are $(2^7 - 1)$ and -2^7. When saturation is used for unsigned arithmetic, the lower bound is always zero and the upper bounds are $(2^{32} - 1)$, $(2^{16} - 1)$, and $(2^8 - 1)$. Saturation is indicated in the instruction mnemonic by an “S”, and unsigned is indicated by a “U” following the period (.). For example, the instruction mnemonic for a signed saturating addition is ADDS, and the mnemonic for an unsigned saturating addition is ADDS.U. Saturation is also used for signed fractions in DSP operations.

2.3.1.1 Integer Arithmetic

Move

Move instructions move a value in a data register or a constant value in the instruction to a destination data register. Move can be used to quickly load a large constant into a data register. A 16-bit constant is created using MOV (which sign-extends the value to 32 bits) or MOV.U (which zero-extends to 32 bits). The MOVH (Move Highword) instruction loads a 16-bit constant into the most-significant sixteen bits of the register and zero fills the least significant sixteen bits, which is useful for loading a left-justified

constant fraction. Loading a 32-bit constant can be done using a MOVH instruction followed by an ADDI (Add Immediate), or by a MOV.U followed by ADDIH (Add Immediate High Word).

Addition and Subtraction

There are three types of addition instructions: no saturation (ADD), signed saturation (ADDS), and unsigned saturation (ADDS.U). For extended precision addition, the ADDX (Add Extended) instruction sets the PSW carry bit to the value of the ALU carry out. The ADDC (Add with Carry) instruction uses the PSW carry bit as the carry in, and updates the PSW carry bit with the ALU carry out. For extended precision addition, the least significant word of the operands is added using the ADDX instruction, and the remaining words are added using the ADDC instruction. The ADDC and ADDX instructions do not support saturation.

Often it is necessary to add 16-bit or 32-bit constants to integers. The ADDI (Add Immediate) and ADDIH (Add Immediate High) instructions add a 16-bit, sign-extended constant or a 16-bit constant, left-shifted by 16. Addition of any 32-bit constant can be done using ADDI followed by an ADDIH.

All add instructions except those with constants have similar corresponding subtract instructions. Because the large immediate of ADDI is sign-extended, it may be used for both addition and subtraction.

The RSUB (Reverse Subtract) instruction subtracts a register from a constant. Using zero as the constant yields negation as a special case.

Multiply and Multiply-Add

Multiplication of two 32-bit integers that produce a 32-bit result can be handled using MUL (Multiply Signed), MULS (Multiply Signed with Saturation), and MULS.U (Multiply Unsigned with Saturation). The MULM (Multiply with Multiword Result) and MULM.U (Multiply with Multiword Result Unsigned) instructions produce the full 64-bit result, which is stored to a register pair; MULM is for signed integers, and MULM.U is for unsigned integers. Special multiply instructions are used for DSP operations.

The Multiply-Add instruction (MADD) multiplies two signed operands, adds the result to a third operand, and stores the result in a destination. Because the third operand and the destination do not use the same registers, the intermediate sums of a multi-term multiply-add instruction can be saved without requiring any additional register moves. The MADD, MADDS (Multiply-Add with Saturation), and MADDS.U (Multiply-Add with Saturation Unsigned) instructions operate on and produce 32-bit integers; MADDS and MADDS.U will saturate on signed and unsigned overflow, respectively. The instructions MADDM (Multiply-Add with Multiword Result), MADDM.U (Multiply-Add with Multiword Result Unsigned), MADDMS (Multiply-Add Multiword with Saturation), and MADDMS.U (Multiply-Add Multiword with Saturation Unsigned) can be used to add the 64-bit product to a 64-bit source and produce a 64-bit result.

The set of Multiply-Subtract (MSUB) instructions that supports the accumulation of products using subtraction instead of addition provides the same set of variations as the MADD instructions.

Division

Division of 32-bit by 32-bit integers is supported for both signed and unsigned integers. Because an atomic divide instruction would require an excessive number of cycles to execute, a divide-step sequence is used to reduce interrupt latency. The divide step sequence allows the divide time to be proportional to the number of significant quotient bits expected.

The sequence begins with a Divide-Initialize instruction (DVINIT(.U), DVINIT.H(U), or DVINIT.B(U), depending on the size of the quotient and whether the operands are to be treated as signed or unsigned). The divide initialization instruction extends the 32-bit dividend to 64 bits, then shifts it left by 0, 16, or 24 bits. Simultaneously it shifts in that many copies of the quotient sign bit to the low-order bit positions. Then follows 4, 2, or 1 Divide-Step instructions (DVSTEP or DVSTEP.U). Each divide step instruction develops eight bits of quotient.

At the end of the divide step sequence, the 32-bit quotient occupies the low-order word of the 64-bit dividend register pair and the remainder is held in the high-order word. If the divide operation was signed, the Divide-Adjust instruction (DVADJ) is required to perform a final adjustment of negative values. If the dividend and the divisor are both known to be positive, the DVADJ instruction can be omitted.

Absolute Value, Absolute Difference

A common operation on data is the computation of the absolute value of a signed number or the absolute value of the difference between two signed numbers. These operations are provided directly by the ABS and ABSDIF instructions and there is a version of each instruction which saturates when the result is too large to be represented as a signed number.

Min, Max, Saturate

Instructions are provided that directly calculate the minimum or maximum of two operands. The MIN and MAX instructions are used for signed integers, MIN.U and MAX.U are used for unsigned integers. The SAT instructions can be used to saturate the result of a 32-bit calculation before storing it in a byte or half-word in memory or a register.

Conditional Arithmetic Instructions

The conditional instructions — Conditional Add (CADD), Conditional Subtract (CSUB), and Select (SEL) — provide efficient alternatives to conditional jumps around very short

sequences of code. All of the conditional instructions use a condition operand that controls the execution of the instruction. The condition operand is a data register with any non-zero value interpreted as TRUE and a zero value interpreted as FALSE. For the CADD and CSUB instructions, the addition/subtraction is performed if the condition is TRUE. For the CADDN and CSUBN instructions it is performed if the condition is FALSE.

The SEL instruction copies one of its two source operands to its destination operand, with the selection of source operands determined by the value of the condition operand (This operation is the same as the C language “?” operation). A typical use might be to record the index value yielding the larger of two array elements:

```
index_max = (a[i] > a[j]) ? i : j;
```

If one of the two source operands in a Select instruction is the same as the destination operand, then the Select instruction implements a simple conditional move. This occurs fairly often in source statements of the general form:

```
if (<condition>) then <variable> = <expression>;
```

Provided that `<expression>` is simple, it is more efficient to evaluate it unconditionally into a source register, using a SEL instruction to perform the conditional assignment, rather than conditionally jumping around the assignment statement.

Logical

The TriCore architecture provides a complete set of 2-operand, bit-wise logic operations. In addition to the AND, OR, and XOR functions, there are the negations of the output — NAND, NOR, and XNOR — and negations of one of the inputs — ANDN and ORN (the negation of an input for XOR is the same as XNOR).

Count Leading Zeroes, Ones, and Signs

To provide efficient support for normalization of numerical results, prioritization, and certain graphics operations, three Count Leading instructions are provided: CLZ (Count Leading Zeros), CLO (Count Leading Ones), and CLS (Count Leading Signs). These instructions are used to determine the amount of left shifting necessary to remove redundant zeros, ones, or signs.

Note that the CLS instruction returns the number of leading redundant signs, which is the number of leading signs minus one. Furthermore, the following special cases are defined: CLZ(0) = 32, CLO(-1) = 32, and CLS(0) = CLS(-1) = 31.

For example, CLZ returns the number of consecutive zeros starting from the most-significant bit of the value in the source data register. In the example shown below (**Table 2-8**), there are seven zeros in the most-significant portion of the input register. If the most-significant bit of the input is a 1, CLZ returns 0.

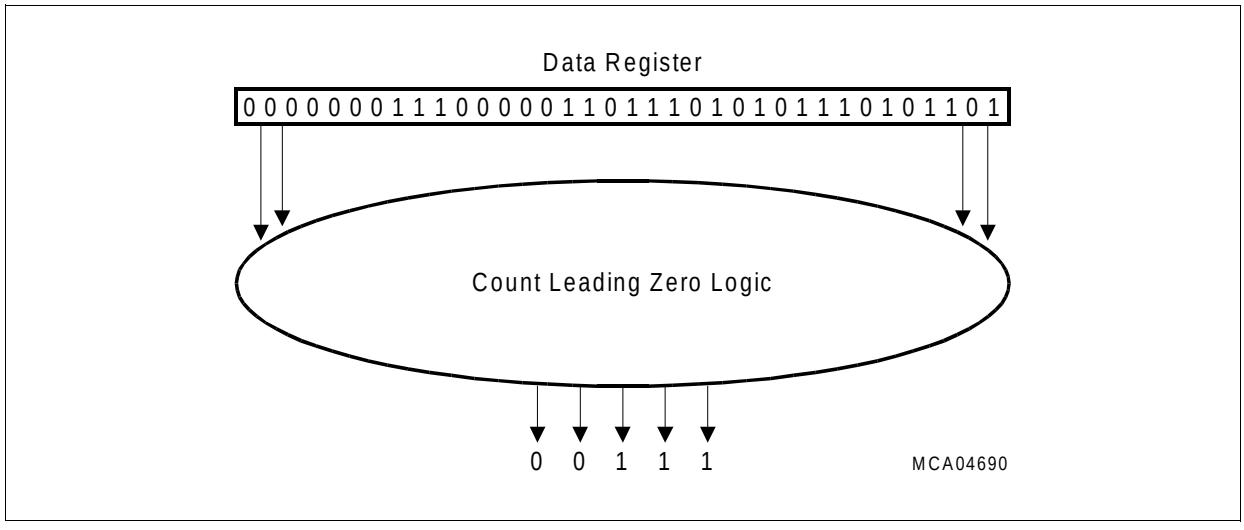


Figure 2-8 Operation of CLZ Instruction

The Count Leading instructions are useful for parsing certain Huffman codes and bit strings consisting of Boolean flags because the code or bit string can be quickly classified by determining the position of the first one (scanning from left to right).

Shift

The shift instructions support multi-bit shifts. The shift amount is specified by a signed integer (n), which may be the contents of a register or a sign-extended constant in the instruction. If $n \geq 0$, the data is shifted left by n[4:0]; otherwise, the data is shifted right by (-n)[4:0]. The (logical) shift instruction, SH, shifts in zeroes for both right and left shifts; the arithmetic shift instruction, SHA, shifts in sign bits for right shifts and zeroes for left shifts. The arithmetic shift with saturation instruction, SHAS, will saturate (on a left shift) if the sign bits that are shifted out are not identical to the sign bit of the result.

Bit Field Extract and Insert

The TriCore architecture supports three bit field extract instructions. The EXTR.U and EXTR instructions extract w (width) consecutive bits from the source, beginning with the bit number specified by the pos (position) operand. The width and position can be specified by two immediate values, by an immediate value and a data register, or by a data register pair. The EXTR.U instruction (**Figure 2-9**) zero-fills the most significant (32-w) bits of the result.

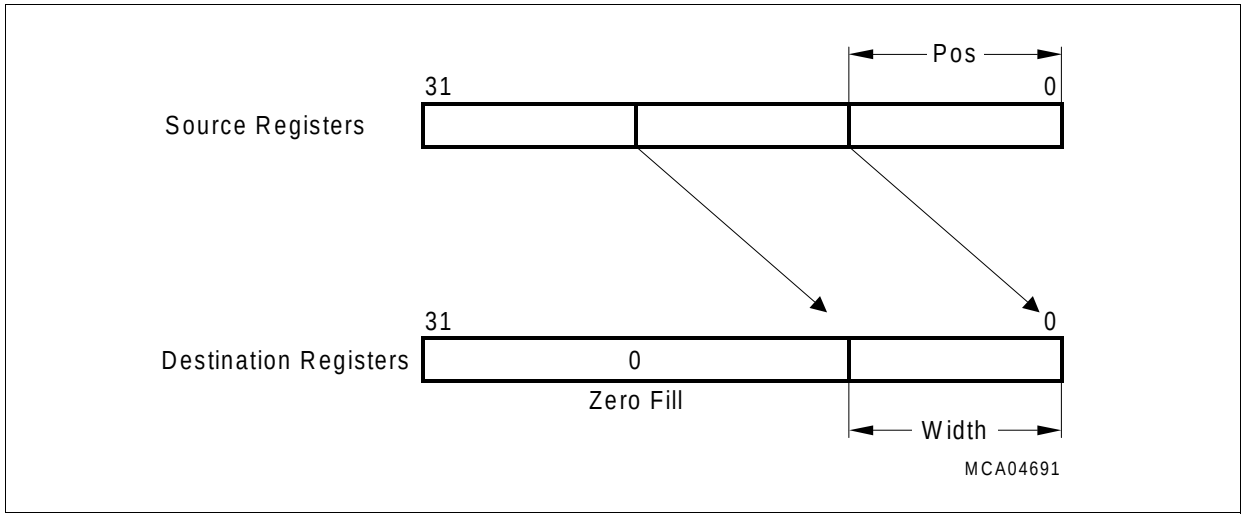


Figure 2-9 Operation of EXTR.U Instruction

The EXTR instruction (**Figure 2-10**) fills the most-significant bits of the result by sign-extending the bit field extracted (thus duplicating the most-significant bit of the bit field).

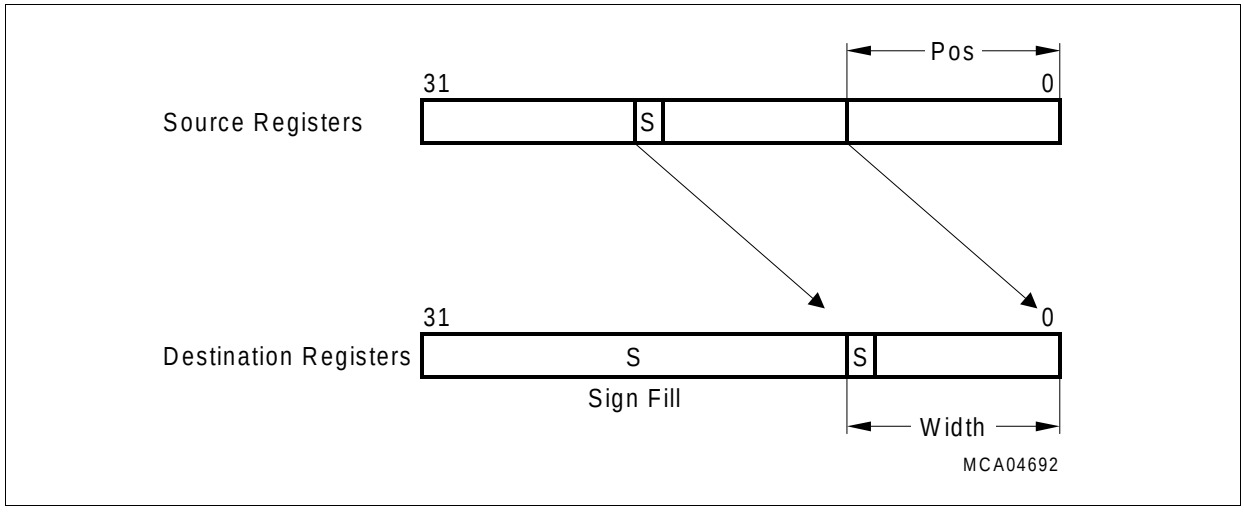


Figure 2-10 Operation of EXTR Instruction

The DEXTR instruction (**Figure 2-11**), concatenates two data register sources to form a 64-bit value from which 32 consecutive bits are extracted. The operation can be thought of as a left shift by pos bits, followed by the truncation of the least significant 32 bits of the result. The value of pos is contained in a data register or is an immediate value in the instruction.

The DEXTR instruction can be used to normalize the result of a DSP filter accumulation in which a 64-bit accumulator is used with several guard bits. The value of pos can be determined by using the CLS (Count Leading Signs) instruction. The DEXTR instruction can also be used to perform a multi-bit rotation by using the same source register for both of the sources that are concatenated.

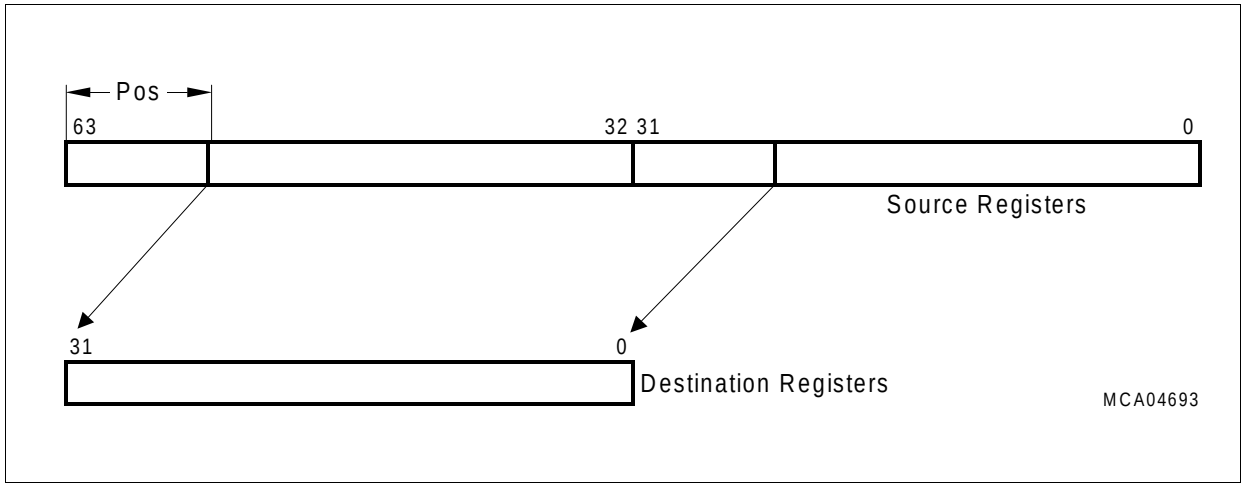


Figure 2-11 Operation of DEXTR Instruction

The INSERT instruction (**Figure 2-12**) takes the w least significant bits of a source data register, shifted left by pos bits and substitutes them into the value of another source register. All other (32-w) bits of the value of the second register are passed through. The values of width and pos are specified in the same way as for EXTR(.U). There is also an alternative form of INSERT that allows a zero-extended 4-bit constant to be the value which is inserted.

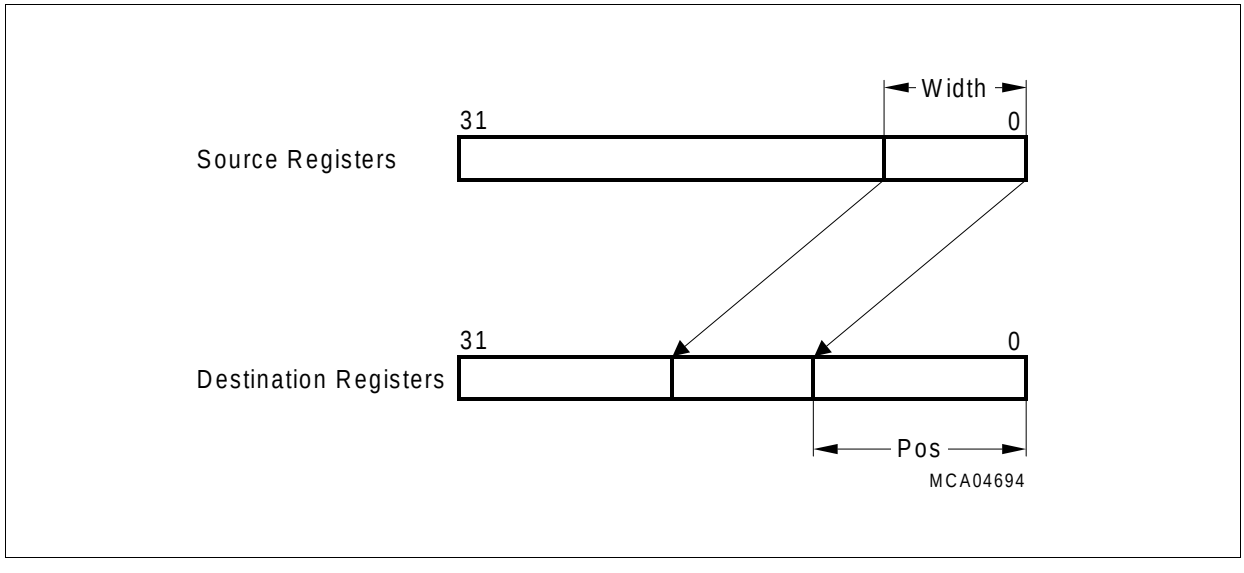


Figure 2-12 Operation of INSERT Instruction

2.3.1.2 DSP Arithmetic

DSP arithmetic instructions operate on 16-bit, signed fractional data in the 1.15 format (also known as Q15) and 32-bit signed fractional data in 1.31 format (also known as Q31). Data values in this format have a single high-order sign bit with a value of 0 or -1, followed by an implied binary point and fraction. Their values are in the range [-1, 1].

16-bit DSP data is loaded into the most significant half of a data register, with the 16 least significant bits set to zero. The left alignment of 16-bit data allows it to be added directly to 32-bit data in 1.31 format. All other fractional formats can be synthesized by explicitly shifting data as required.

Operations created for this format are multiplication, multiply-add, and multiply-subtract. The signed fractional formats 1.15 and 1.31 are supported with the MUL.Q and MULR.Q instructions. These instructions operate on two left-justified signed fractions and return a 32-bit signed fraction.

Scaling

The multiplier result can be shifted in two ways:

- Left shifted by 1
  - One sign bit is suppressed and the result is left-aligned, conserves the input format.
- Not shifted
  - The result retains its two sign bits (2.30 format).
  - This format can be used with IIR filters, in which some of the coefficients are between 1 and 2, and to have one guard bit for accumulation.

Special Case = -1 × -1 = +1

When multiplying the two maximum negative values (-1), the result should be the maximum positive number (+1). For example,

```
0x8000 * 0x8000 = 0x4000 0000
```

is correctly interpreted in Q format as:

```
-1(1.15 format) * -1(1.15 format) = +1 (2.30 format)
```

However, when the result is shifted left by one, the result is 0x8000 0000, which is incorrectly interpreted as:

```
-1(1.15 format) * -1(1.15 format) = -1 (1.31 format)
```

To avoid this problem, the result of a Q format operation (-1 * -1) that has been left-shifted by one (left-justified), is saturated to the maximum positive value. Thus,

```
0x8000 * 0x8000 = 0x7FFF FFFF
```

is correctly interpreted in Q format as:

```
-1(1.15 format) * -1(1.15 format) = (nearest representation of)+1 (1.31
format)
```

This operation is completely transparent to the user and does not set the overflow flags.

Guard Bits

When accumulating sums (for example, in filter calculations) guard bits are often required to prevent overflow. The instruction set directly supports the use of one guard bit when using a 32-bit accumulator. When more guard bits are required, a register pair (64 bits) can be used.

Rounding

Rounding is used to retain the 16-bit most-significant bits of a 32-bit result. Rounding is combined with the MUL, MADD, MSUB instructions, and is implemented by adding 1 to bit 15 of a 32-bit register.

Overflow and Saturation

Saturation on signed and unsigned overflow is implemented as part of the MUL, MADD, and MSUB instructions.

Sticky Advance Overflow and Block Scaling in FFT

The Sticky Advance Overflow (SAV) bit is set whenever an overflow “almost” occurs. It can be used in block scaling of intermediate results during an FFT calculation. Before each pass of applying a butterfly operation, the SAV bit is cleared, and after the pass the SAV bit is tested. If it is set, all of the data is scaled (using an arithmetic right shift) before starting the next pass. This procedure gives the greatest dynamic range for intermediate results without the risk of overflow.

Packed Arithmetic

The packed arithmetic instructions partition a 32-bit word into several identical objects, which can then be fetched, stored, and operated on in parallel. These instructions, in particular, allow the full exploitation of the 32-bit word of the TriCore architecture in signal and data processing applications.

The TriCore architecture supports two packed formats. The first format (**Figure 2-13**) divides the 32-bit word into two, 16-bit (half-word) values. Instructions which operate on data in this way are denoted in the instruction mnemonic by the “.H” and “.HU” data type modifiers.

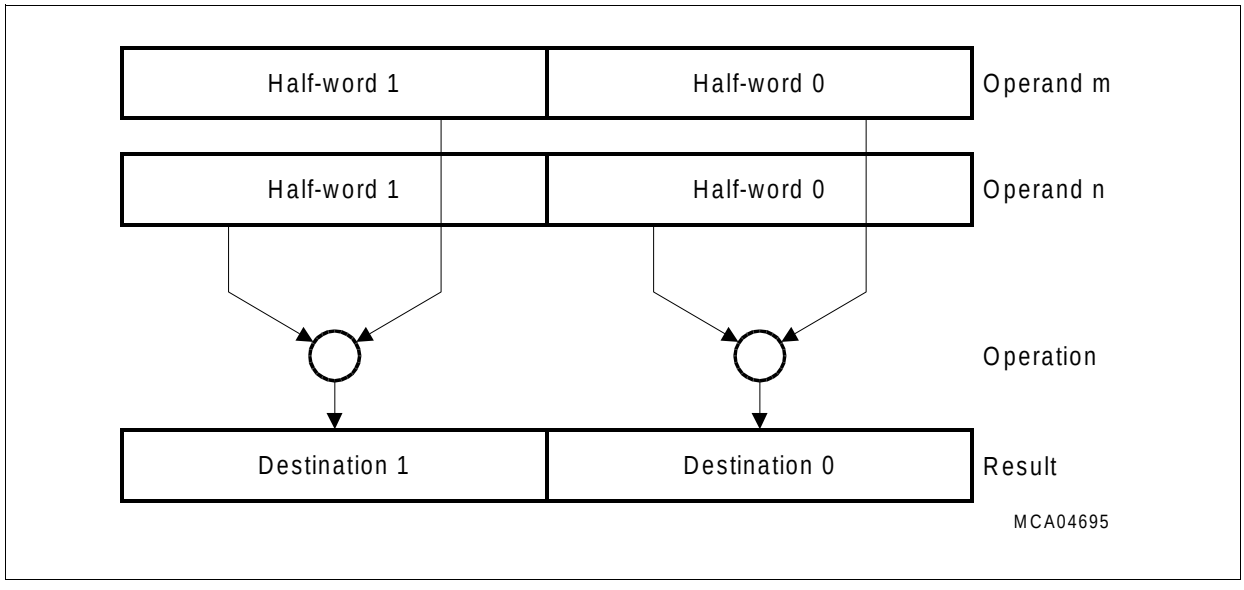


Figure 2-13 Packed Half-word Data Format

The second packed format (**Figure 2-14**) divides the 32-bit word into four, 8-bit values. Instructions that operate this way are denoted by the ".B" and ".BU" data type modifiers.

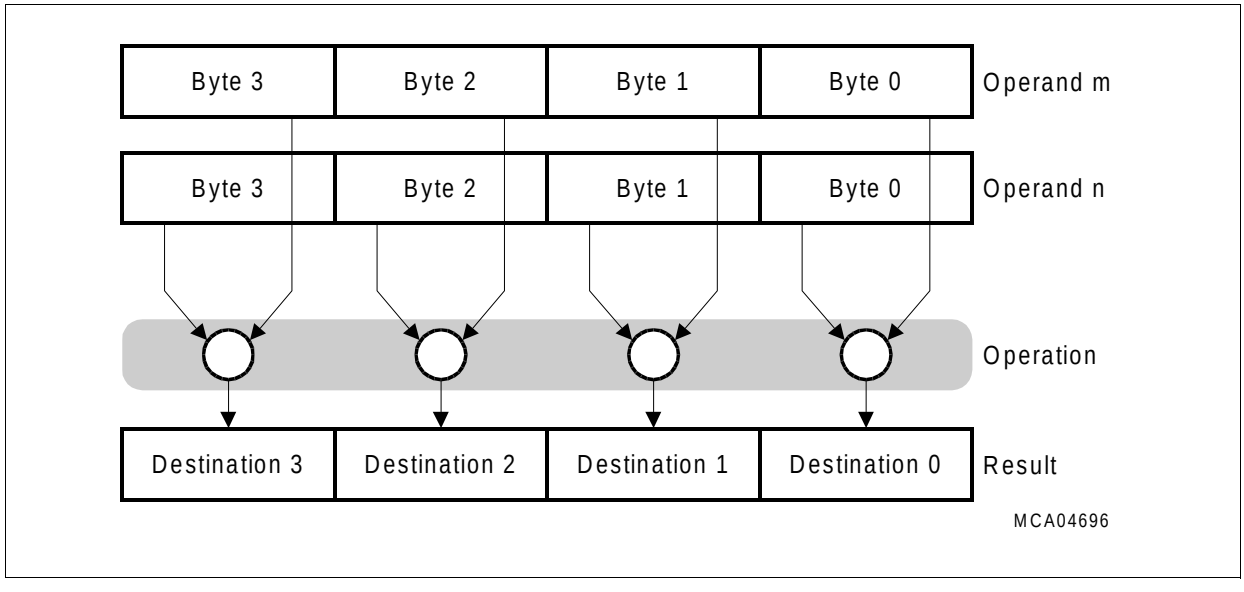


Figure 2-14 Packed Byte Data Format

The loading and storing of packed values into data registers is supported by the normal Load Word (LD.W) and Store Word (ST.W) instructions. The packed objects can then be manipulated in parallel by a set of special packed arithmetic instructions that perform such arithmetic operations as addition, subtraction, multiplication, etc.

Addition is performed on individual packed bytes or half-words using the ADD.B and ADD.H instructions and their saturating variations ADDS.B and ADDS.H. ADD.B ignores

overflow/underflow within individual bytes, while ADDS.B will saturate individual bytes to the most positive, 8-bit signed integer (127) on individual overflow, or to the most negative, 8-bit signed integer (-128) on individual underflow. Similarly, the ADD.H instruction ignores overflow/underflow within individual half-words, while the ADDS.H will saturate individual half-words to the most positive 16-bit signed integer (2^{15} - 1) on individual overflow, or to the most negative 16-bit signed integer (- 2^{15}) on individual underflow. Saturation for unsigned integers is also supported by the ADDS.BU and ADDS.HU instructions. Arithmetic on packed data also includes subtraction, multiplication, absolute value, and absolute difference.

2.3.2 Compare Instructions

The compare instructions use a perform operation on the contents of two registers. The Boolean result (1 = true and 0 = false) is stored in the least significant bit of a data register, and the remaining bits in the register are cleared to zero. **Figure 2-15** illustrates the operation of the LT (Less Than) compare instruction.

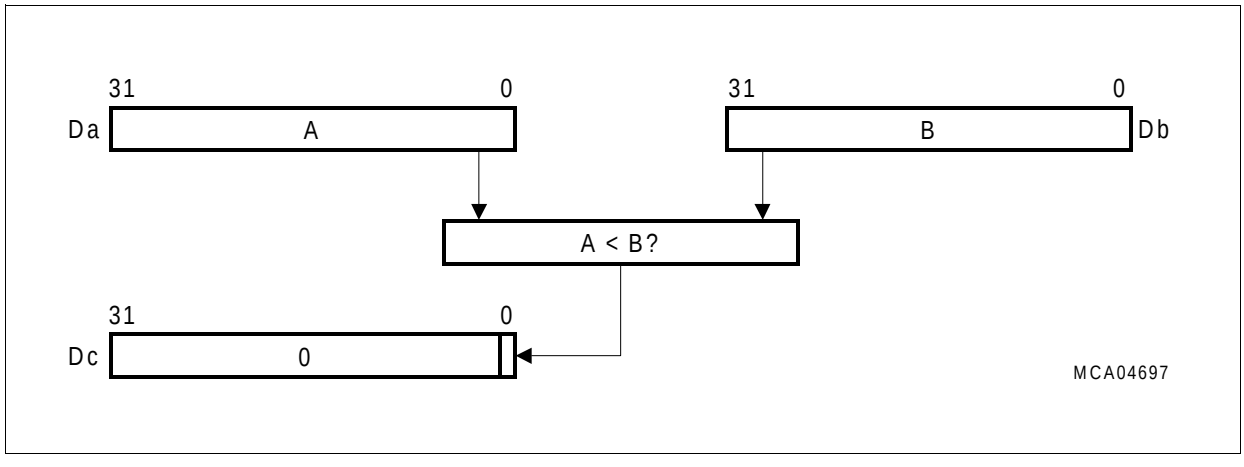


Figure 2-15 LT Comparison

The comparison instructions are: equal (EQ), not equal (NE), less than (LT), and greater than or equal to (GE), with versions for both signed and unsigned integers.

Comparison conditions not explicitly provided in the instruction set can be obtained by either swapping the operands when comparing two registers, or by incrementing the constant by one when comparing a register and a constant (**Table 2-4**).

Table 2-4 Equivalent Comparison Operations

"Missing" Comparison Operation	TriCore Equivalent Comparison Operation
LE Dc, Da, Db	GE Dc, Db, Da
LE Dc, Da, const	LT Dc, Da, (const + 1)

Table 2-4 Equivalent Comparison Operations

GT	Dc, Da, Db	LT	Dc, Db, Da
GT	Dc, Da, const	GE	Dc, Da, (const + 1)

To accelerate the computation of complex conditional expressions, the accumulation of versions of the comparison instructions are supported. These instructions — as indicated in the instruction mnemonic by “op” preceding the “.” (for example, op.LT) — combine the result of the comparison with a previous comparison result. The combination is a logic AND, OR, or XOR; for example, AND.LT, OR.LT, and XOR.LT. **Figure 2-16** illustrates combining the LT instruction with a Boolean operation.

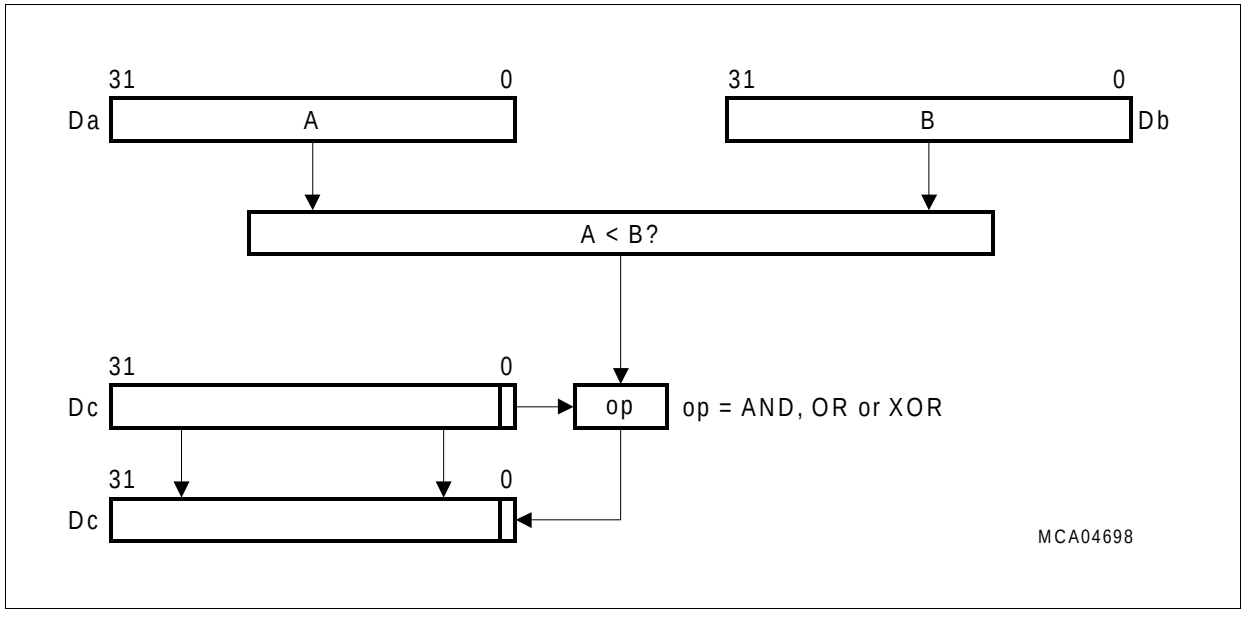


Figure 2-16 Combining LT Comparison with Boolean Operation

The evaluation of the following C expression can be optimized using the combined compare-Boolean operation:

```
d5 = (d1 < d2) || (d3 == d4);
```

Assuming all variables are in registers, two instructions will compute the value in d5:

```
lt     d5,d1,d2    ; compute (d1 < d2)
or.eq  d5,d3,d4    ; or with (d3 == d4)
```

Certain control applications require that several Booleans be packed into a single register. These packed bits can be used as an index into a table of constants or a jump table, which permits complex Boolean functions and/or state machines to be evaluated efficiently. To facilitate the packing of Boolean results into a register, compound Compare with Shift instructions (for example, SH.EQ) are supported. The result of the comparison is placed in the least significant bit of the result after the contents of the

destination register have been shifted left by one position. **Figure 2-17** illustrates the operation of the SH.LT (Shift Less Than) instruction.

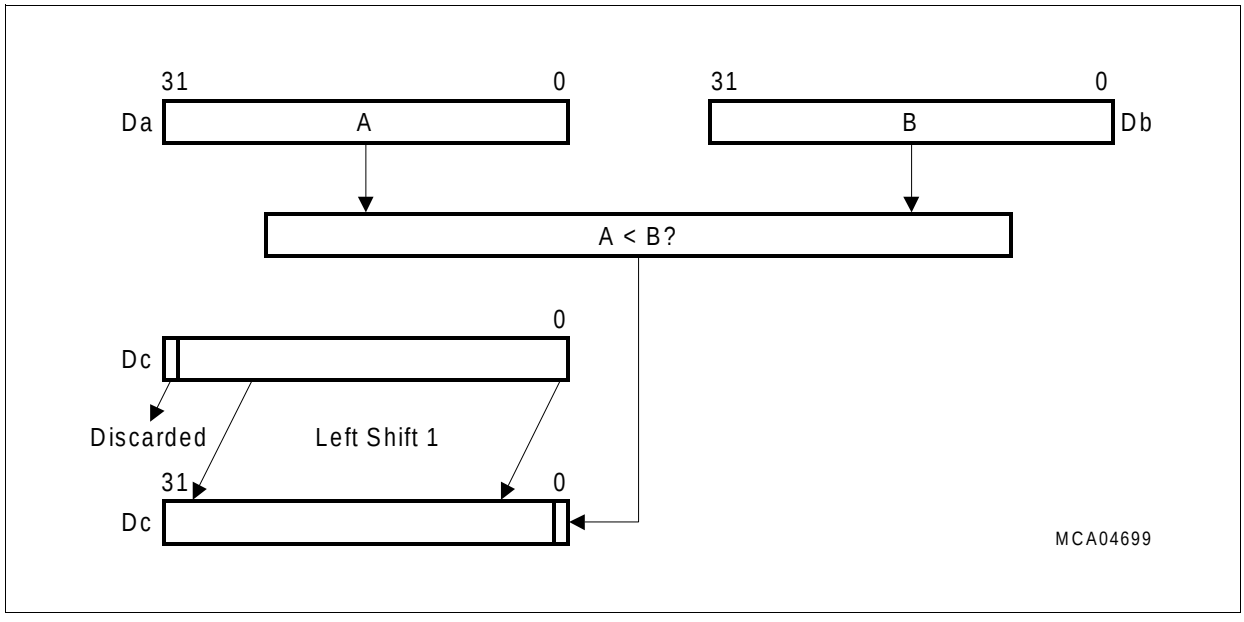


Figure 2-17 SH.LT Instruction

For packed bytes, there are special compare instructions that perform four individual byte comparisons and produce a 32-bit mask consisting of four “extended” Booleans. For example, EQ.B yields a result where individual bytes are FF_H for a match or 00_H for no match. Similarly, for packed half-words there are special compare instructions that perform two individual half-word comparisons and produce two extended Booleans. The EQ.H instruction results in two extended Booleans: $FFFF_H$ for a match and 0000_H for no match. There are even abnormal packed-word compare instructions that compare two words in the normal way but produce a single extended Boolean. The EQ.W instruction results in the extended Boolean $FFFFFFFF_H$ for match and 00000000_H for no match.

Extended Booleans are useful as masks, that can be used by subsequent bit-wise logic operations. Also, CLZ (count leading zeros) or CLO (count leading ones) can be used on the result to quickly find the position of the left-most match. **Figure 2-18** shows an example of the EQ.B instruction.

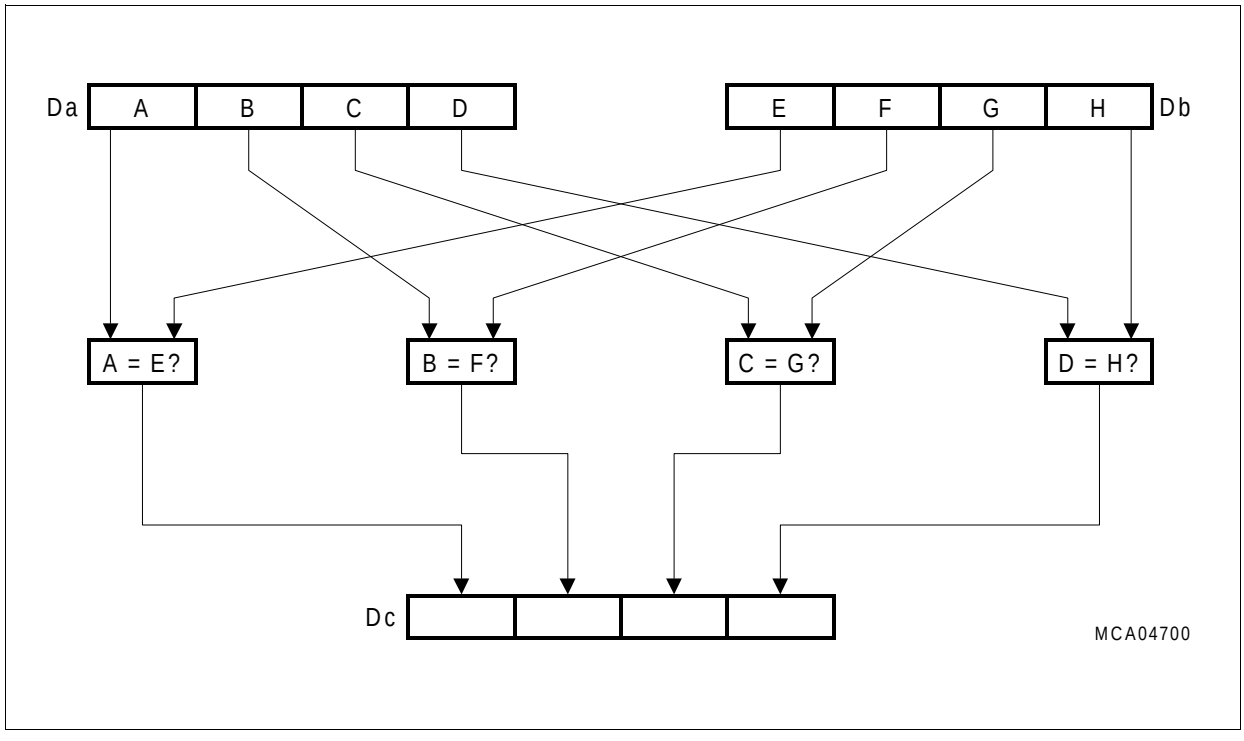


Figure 2-18 EQ.B Instruction Operation

2.3.3 Bit Operations

Instructions are provided that operate on single bits, denoted in the instruction mnemonic by the “T” data type modifier (for example, AND.T). There are eight instructions for combinatorial logic functions with two inputs, eight instructions with three inputs, and eight with two inputs and a shift. The one-bit result of a two-input function (for example, AND.T) is stored in the least significant bit of the destination data register, and the most-significant 31 bits are set to zero. The source bits can be any bit of any data register. This is illustrated in **Figure 2-19**. The available Boolean operations are: AND, NAND, OR, NOR, XOR, XNOR, ANDN, and ORN.

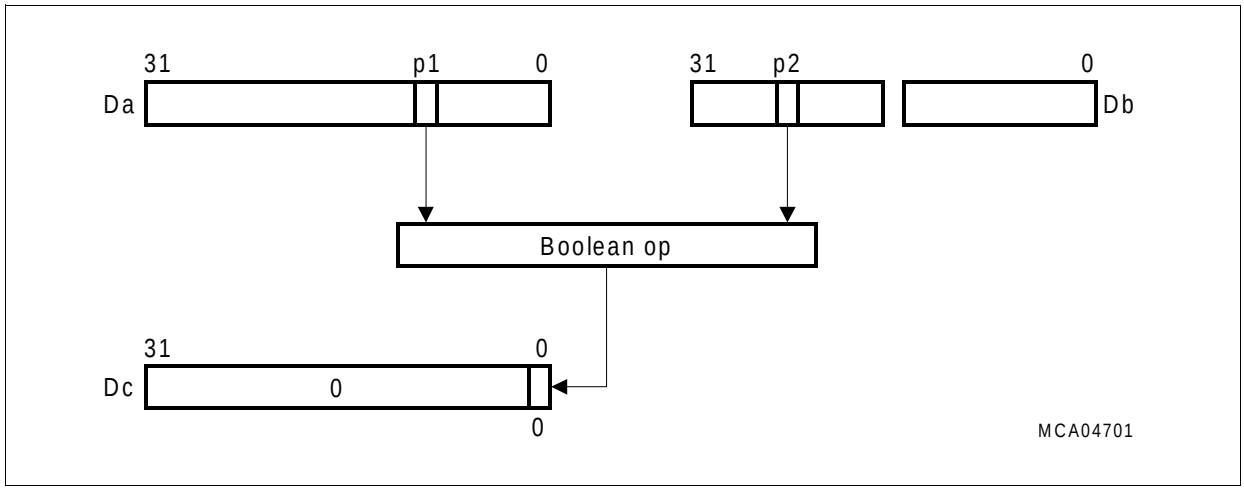


Figure 2-19 Boolean Operations

Evaluation of complex Boolean equations can use the 3-input Boolean operations in which the output of a two-input instruction is combined with the least significant bit of a third data register to form the input to a further operation. The result is written to bit 0 of the third data register, with the remaining bits unchanged (**Figure 2-20**).

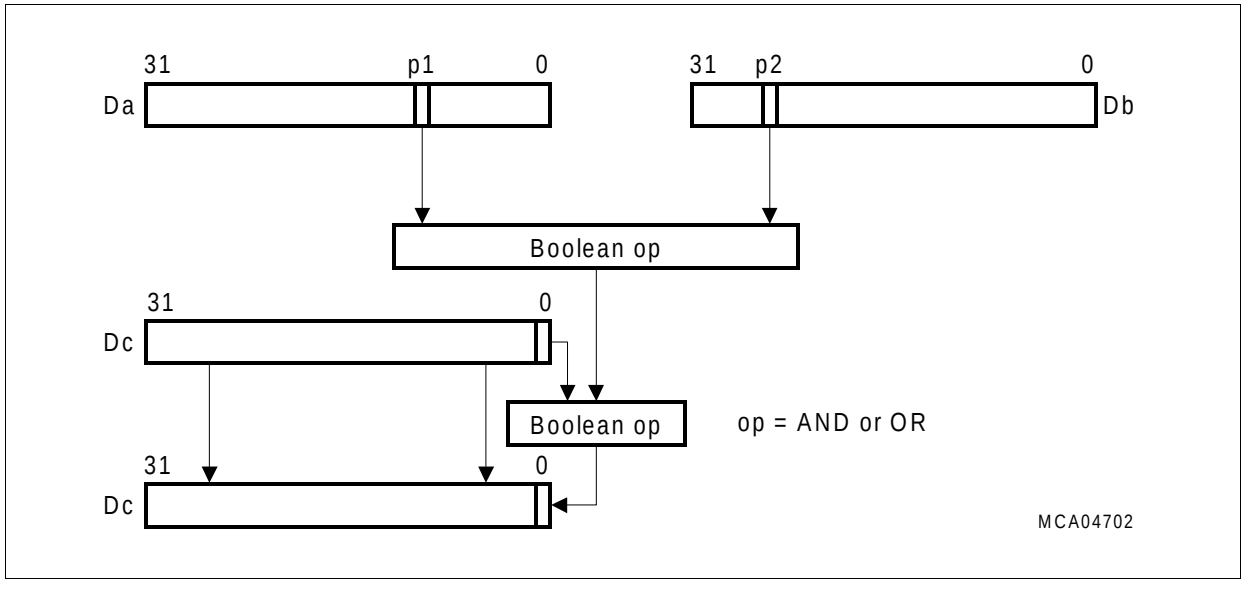


Figure 2-20 Three-Input Boolean Operation

Of the many possible three-input operations, eight have been singled out for the efficient evaluation of logical expressions. The eight instructions provided are: AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, and OR.OR.T.

Just as for the comparison instructions, the results of bit operations often need to be packed into a single register for controller applications. For this reason, the basic two-input instructions can be combined with a shift prefix (for example, SH.AND.T). These

operations first perform a single-bit left shift on the destination register and then store the result of the two-input logic function into its least significant bit (**Figure 2-21**).

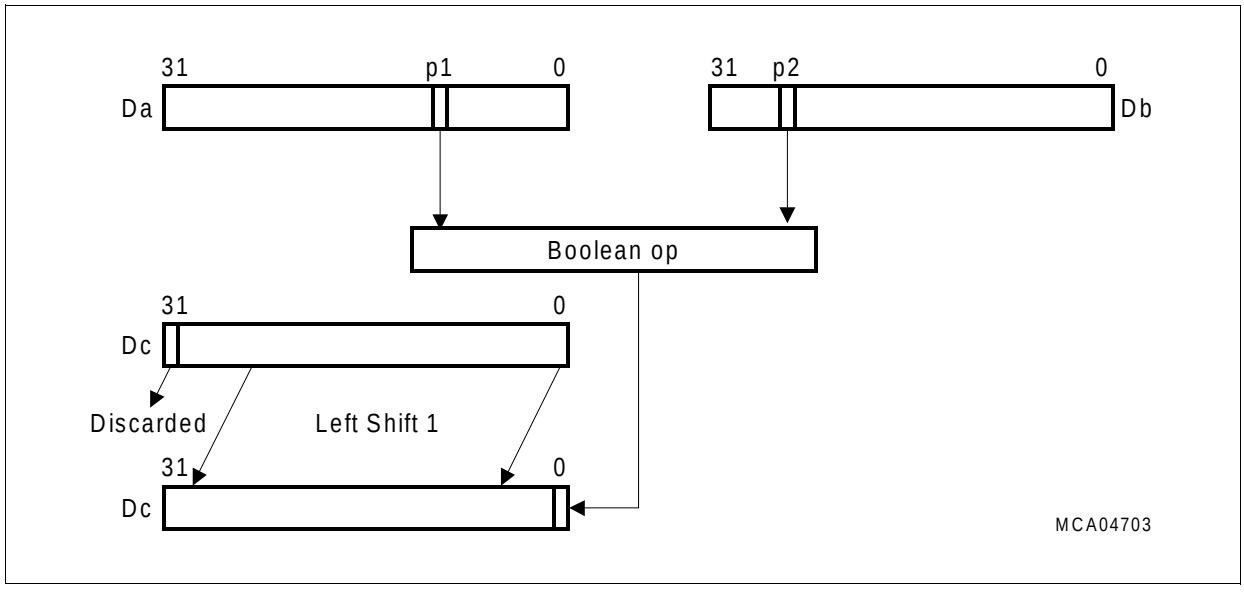


Figure 2-21 Shift Plus Boolean Operation

2.3.4 Address Arithmetic

The TriCore architecture provides selected arithmetic operations on the address registers. These operations supplement the address calculations inherent in the addressing modes used by the load and store instructions.

Initialization of base pointers requires loading a constant into an address register. When the base pointer is in the first 16 KBytes of each segment, this can be done using the Load Effective Address (LEA) instruction, using the absolute addressing mode. Loading a 32-bit constant into an address register can be accomplished using MOVH.A followed by an LEA that uses the base plus 16-bit offset addressing mode. For example,

```
movh.a   a5, ((ADDRESS+0x8000)>>16) & 0xffff
lea      a5, [a5](ADDRESS & 0xffff)
```

The MOVH.A instruction loads a 16-bit immediate into the most-significant 16-bits of an address register and zero-fills the least significant 16-bits. Adding a 16-bit constant to an address register can be done using the LEA instruction with the base plus offset addressing mode. Adding a 32-bit constant to an address register can be done in two instructions: an Add Immediate High Word (ADDIH.A), which adds a 16-bit immediate to the most-significant 16 bits of an address register, followed by an LEA using the base plus offset addressing mode. For example,

```
addih.a  a8, ((OFFSET+0x8000)>>16) & 0xffff
lea      a8, [a8](OFFSET & 0xffff)
```

The Add Scaled (ADDSC.A) instruction directly supports the use of a data variable as an index into an array of bytes, half-words, words, or double-words.

2.3.5 Address Comparison

As with the comparison instructions that use the data registers (see **Section 2.3.2**), the comparison instructions using the address registers put the result of the comparison in the least significant bit of the destination data register and clear the remaining register bits to zeros. An example of the Less Than (LT.A) instruction is shown in **Figure 2-22**.

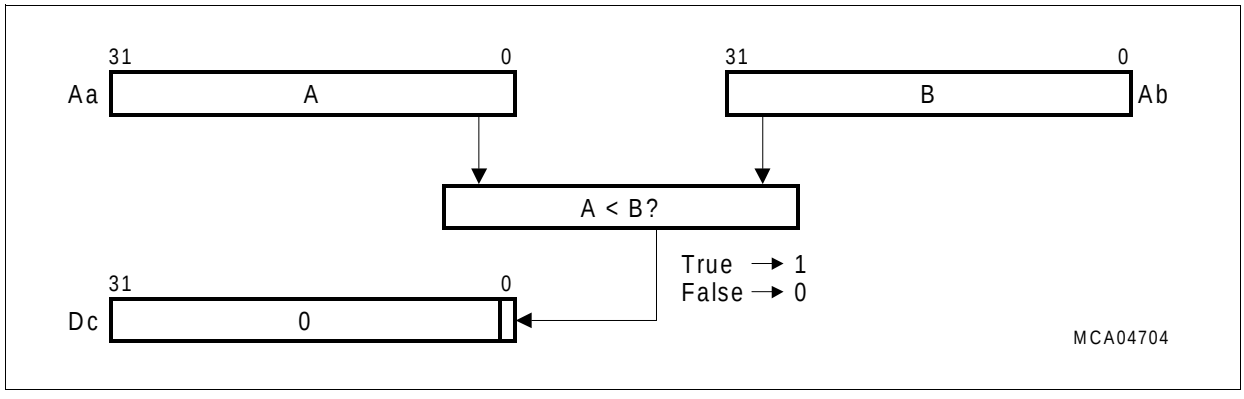


Figure 2-22 LT.A Comparison Operation

There are comparison instructions for equal (EQ.A), not equal (NE.A), less than (LT.A), and greater than or equal to (GE.A). As with the comparison instructions using the data registers, comparison conditions not explicitly provided in the instruction set can be obtained by swapping the two operand registers (**Table 2-5**).

Table 2-5 Comparison Operations

"Missing" Comparison Operation	TriCore Equivalent Comparison Operation
LE.A Dc, Aa, Ab	GE.A Dc, Ab, Aa
GT.A Dc, Aa, Ab	LT.A Dc, Ab, Aa

In addition to these instructions, instructions that test whether an address register is equal to zero (EQZ.A), or not equal to zero (NEZ.A) are supported. These instructions are useful to test for null pointers — a frequent operation when dealing with linked lists and complex data structures.

2.3.6 Branch Instructions

Branch instructions change the flow of program control by modifying the value in the PC register. There are two types of branch instructions: conditional and unconditional. Whether or not a conditional branch is taken depends on the result of a Boolean compare operation (see **Section 2.3.2**) rather than on the state of condition codes.

2.3.6.1 Unconditional Branch

There are three groups of unconditional branch instructions: Jump instructions, Jump and Link instructions, and Call and Return instructions.

A Jump instruction simply loads the Program Counter with the address specified in the instruction. A Jump and Link instruction does the same, and also stores the address of the next instruction in the “return address register” A11/RA. A Jump and Link instruction can be used to implement a subroutine call when the called routine does not modify any of the caller’s non-volatile registers. The Call instructions differ from a Jump and Link in that they save the caller’s non-volatile registers in a dynamically-allocated save area. The Return instruction, in addition to performing the return jump, restores the non-volatile registers.

Each group of unconditional Jump instructions contains separate instructions that differ in how the target address is specified. There are instructions using a relative 24-bit signed displacement (J, JL, and CALL), instructions using 24 bits of displacement as an absolute address (JA, JLA, and CALLA), and instructions using the address contained in an address register (JI, JLI, CALLI, RET, and RFE).

There are additional 16-bit instructions for a relative jump using an 8-bit displacement (J), an instruction for an indirect jump (JI), and an instruction for a return (RET).

Both the 24-bit and 8-bit relative displacements are scaled by two before they are used, because all instructions must be aligned on an even address. The use of a 24-bit displacement is shown in **Figure 2-23**.

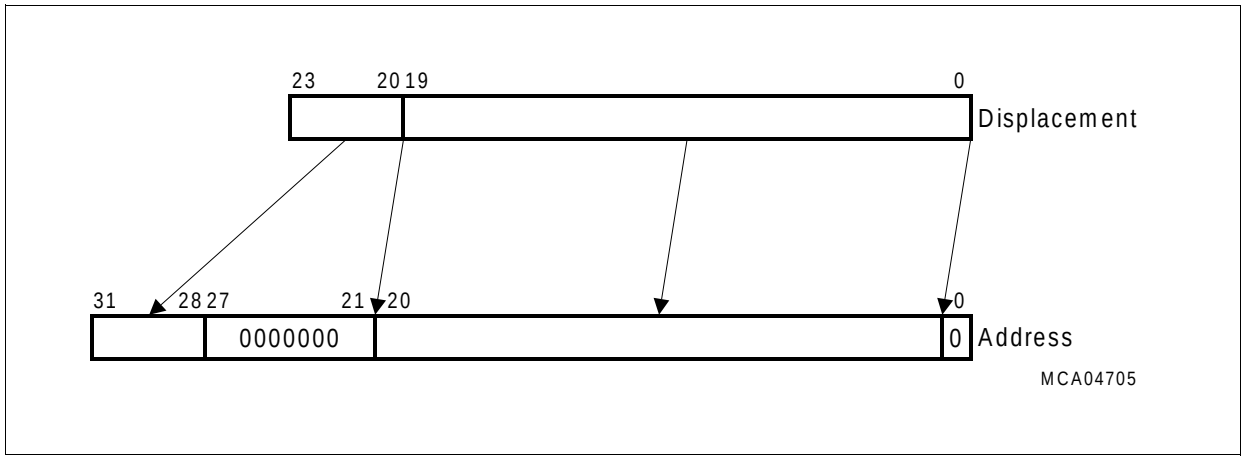


Figure 2-23 Displacement as Absolute Address

2.3.6.2 Conditional Branch

The conditional branch instructions use the relative addressing mode, with a displacement value encoded in 4, 8, or 15 bits. The displacement is scaled by 2 before it is used, because all instructions must be aligned on an even (half-word) address. The

scaled displacement is sign-extended to 32 bits before it is added to the program counter, unless otherwise noted.

The Boolean test uses the contents of data registers, address registers, or individual bits in data registers.

Conditional Jumps on Data Registers

Six of the Conditional Jump instructions use a 15-bit signed displacement field: comparison for equality (JEQ), non-equality (JNE), less than (JLT), less than unsigned (JLT.U), greater than or equal (JGE), and greater than or equal unsigned (JGE.U). The second operand to be compared may be an 8-bit sign- or zero-extended constant. There are two 16-bit instructions that test whether the implicit D15 register is equal to zero (JZ) or not equal to zero (JNZ). The displacement is 8-bit in this case. Another two 16-bit instructions compare the implicit D15 register with a 4-bit, sign-extended constant (JEQ, JNE). The jump displacement field is limited to 4 zero-extended bits in this case.

There is a full set of 16-bit instructions that compare a data register to zero: JZ, JNZ, JLTZ, JLEZ, JGTZ, and JGEZ. Because any data register may be specified, the jump displacement is limited to 4-bit zero-extended constant in this case.

Conditional Jumps on Address Registers

The Conditional Jump instructions that use address registers are a subset of the data register Conditional Jump instructions. Four Conditional Jump instructions use a 15-bit signed displacement field: comparison for equality (JEQ.A), non-equality (JNE.A), equal to zero (JZ.A), and non-equal to zero (JNZ.A).

Because testing pointers for equality to zero is so frequent, two 16-bit instructions are provided (JZ.A and JNZ.A) with a displacement field limited to four zero-extended bits.

Conditional Jumps on Bits

Conditional jumps can be performed based on the value of any bit in any data register. The JZ.T instruction jumps when the bit is clear, and the JNZ.T instruction jumps when the bit is set. For these instructions, the jump displacement field is 15 bits.

There are two 16-bit instructions that test any of the lower 16 bits in the implicit register D15 and have a displacement field of four zero-extended bits.

2.3.6.3 Loop Instructions

Four special versions of Conditional Jump instructions are intended for efficient implementation of loops. The JNEI and JNED instructions are like a normal JNE instruction, but with an additional increment or decrement operation of the first register operand. The increment or decrement operation is performed unconditionally after the comparison. The jump displacement field is 15 bits. For example, a loop that should be executed for D3 = 3, …, 10 can be implemented as follows:

```
  lea   d3,3
loop1:
  ...
  jnei  d3,10,loop1
```

The LOOP instruction is a special kind of jump that utilizes the special TriCore hardware that implements “zero overhead” loops. The LOOP instruction only requires execution time in the pipeline the first and last time it is executed (for a given loop). For all other iterations of the loop, the LOOP instruction has zero execution time. For example, a loop that should be executed 100 times may be implemented as:

```
  mova   a2,99
loop2:
  ...
  loop   a2,loop2
```

The LOOP instruction above requires execution cycles the first and 100th time it is executed, but the other 98 executions require no cycles.

Note that the LOOP instruction differs from the other Conditional Jump instructions in that it uses an address register for the iteration count, rather than a data register. This allows it to be used in filter calculations in which a large number of data register reads and writes occur each cycle. Using an address register for the LOOP instruction reduces the need for an extra data register read port.

The LOOP instruction has a 32-bit version using a 15-bit displacement field (left-shifted by one bit and sign-extended), and a 16-bit version that uses a 4-bit displacement field. Unlike other 16-bit relative jumps, the 4-bit value is one-extended rather than zero-extended, because this instruction is specifically intended for loops.

An unconditional variant of the LOOP instruction is provided (LOOPU) which utilizes the zero overhead LOOP hardware. Such an instruction is used at the end of a while LOOP body to optimize the jump back to the start of the while construct.

2.3.7 Load and Store Instructions

The Load and Store instructions use seven addressing modes to move data between registers and memory (**Table 2-6**). The addressing mode determines the effective byte address for the Load or Store instruction and any update of the base pointer address register.

Table 2-6 Addressing Modes

Addressing Mode	Syntax	Effective Address	Instruction Format
Absolute	constant	{offset18[17:14], 14’bo, offset 18[13:0]}	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BO

Table 2-6 Addressing Modes

Base + Long Offset	[An]offset	A[a]+sign_ext(offset16)	BOL
Pre-increment	[+An]offset	A[a]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[a]	BO
Circular	[An+c]offset	A[b]+A[b+1][15:0] (b is even)	BO
Bit-reverse	[An+r]	A[b]+A[b+1][15:0] (b is even)	BO

2.3.7.1 Load/Store Basic Data Types

The TriCore architecture defines loads and stores for the basic data types — corresponding to bytes, half-words, words and double-words — as well as for signed fractions and addresses. The movement of data between registers and memory for the basic data types is illustrated in **Figure 2-24**. Note that when the data loaded from memory is smaller than the destination register (that is, 8- and 16-bit quantities), the data is loaded into the least significant bits of the register (except for fractions which are loaded into the most significant bits of a register), and the remaining register bits are sign- or zero-extended to 32 bits, depending on the particular instruction.

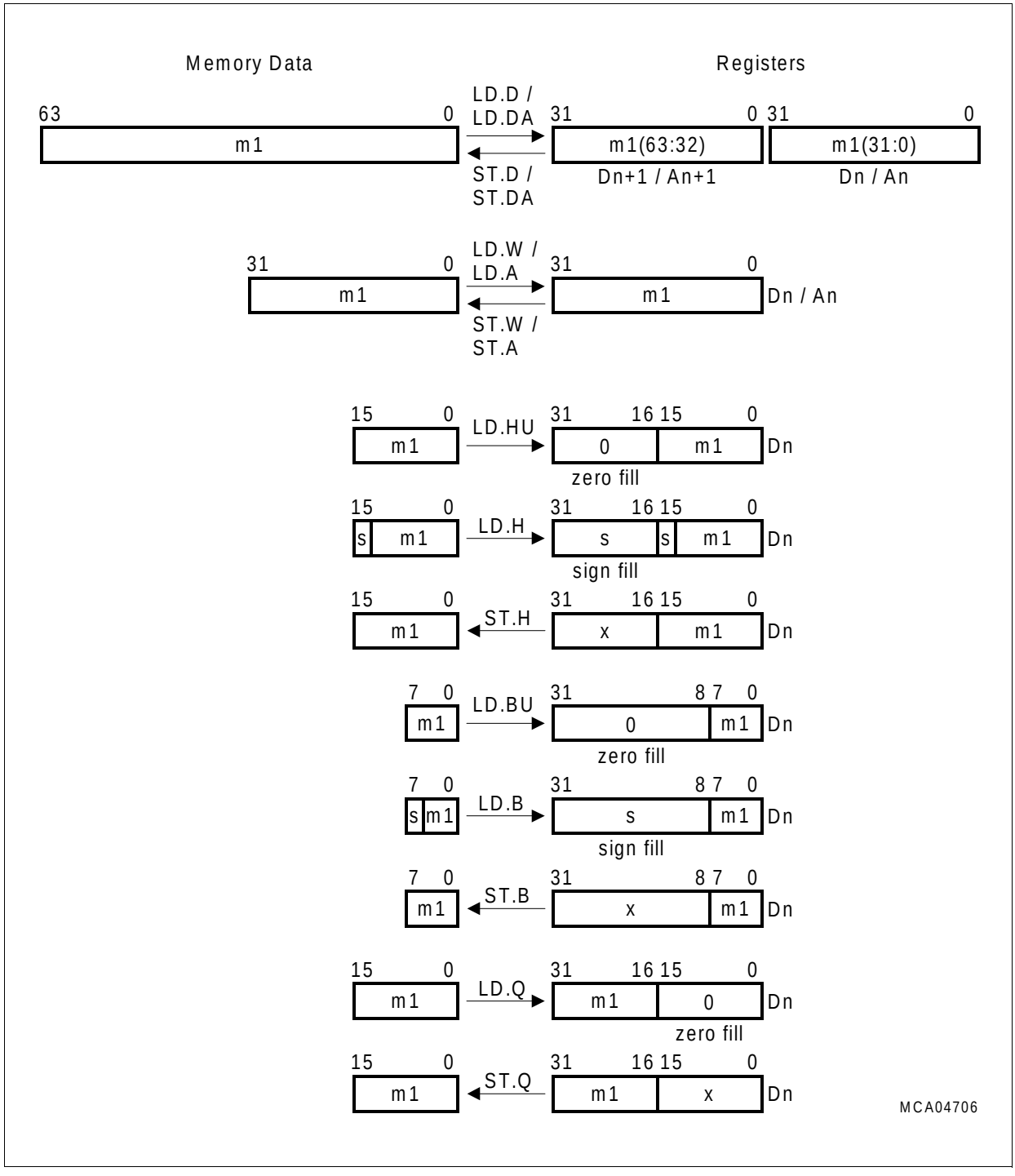


Figure 2-24 Load/Store Basic Data Types

2.3.7.2 Load Bit

The approach used to load individual bits depends on whether the bit within the word (or byte) is given statically or dynamically.

Loading a single bit with a fixed bit offset from a byte pointer is accomplished with an ordinary load instruction. One then can extract, logically operate on, or jump on any bit in a register.

Loading a single bit with a variable bit offset from a word-aligned byte pointer is done with a special scaled offset instruction. This offset instruction shifts the bit offset to the right by three positions (producing a byte offset), adds this result to the byte pointer above, and finally zeroes out the two lower bits, thus, aligning the access on a word boundary. A word load can then access the word that contains the bit which can be extracted with an extract instruction. The extract instruction uses only the lower five bits of the bit pointer, that is, the bits that were either shifted out or masked out above. An example is:

```
ADDSC.AT    A8,A9,D8       ; A9 = byte pointer. D8 = bit offset.
LD.W        D9,[A8]
EXTR.U      D10,D9,D8,1    ; D10[0] = loaded bit.
```

2.3.7.3 Store Bit and Bit Field

The ST.T instruction can clear or set single memory or peripheral bits, resulting in reduced code size. ST.T statically specifies a byte address and a bit number within that byte, and indicates whether the bit should be set or cleared. The addressable range for this instruction is the first 16 KBytes of each of the 16 memory segments.

Using any of the addressing modes, the Insert Mask (IMASK) instruction can be used in conjunction with the Load-Modify-Store (LDMST instruction) to store a single bit or a bit field to a location in memory. This operation is especially useful for reading and writing memory-mapped peripherals. The IMASK instruction is very similar to the INSERT instruction, but IMASK generates a data register pair that contains a mask and a value. The LDMST instruction uses the mask to indicate which portion of the word to modify. An example of a typical instruction sequence is:

```
imask   E8,3,4,2       ; insert value = 3, position = 4, width = 2
ldmst   _IOREG,E8      ; at absolute address "_IOREG"
```

To clarify the operation of the IMASK instruction, consider the following example. The binary value 1011_B is to be inserted starting at bit position 7 (the width is four). The IMASK instruction would result in the following two values:

```
0000 0000 0000 0000 0000 0111 1000 0000     MASK
0000 0000 0000 0000 0000 0101 1000 0000     VALUE
```

To store a single bit with a variable bit offset from a word-aligned byte pointer, first the word address is determined in the same way as for the load above. Next the special scaled offset instruction shifts the bit offset to the right by three positions — which

produces a byte offset — then adds this offset to the byte pointer above, and finally zeroes out the two lower bits, thus aligning the access on a word boundary. An IMASK and LDMST instruction can store the bit into the proper position in the word. An example is:

```
ADDSC.AT    A8,A9,D8        ; A9 = byte pointer. D8 = bit offset.
IMASK       E10,D9,D8,1     ; D9[0] = data bit.
LDMST       [A8],E10
```

2.3.8 Context Related Instructions

As well as the instructions that implicitly save and restore contexts (such as Calls and Returns), the TriCore instruction set includes instructions that allow a task's contexts to be explicitly saved, restored, loaded, and stored. These instructions are detailed in the following sections.

2.3.8.1 Context Saving and Restoring

The Upper Context of a task is always automatically saved on a call, interrupt, or trap. It is automatically restored on a return. However, the Lower Context of a task must be saved/restored explicitly.

The SVLCX instruction (Save Lower Context) saves registers A2 through A7 and D0 through D7 together with the return address in register A11/RA and the PCXI. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The RSLCX instruction (Restore Lower Context) restores the Lower Context. It loads registers A2 through A7 and D0 through D7 from the CSA. It also loads A11/RA from the saved PC field. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The BISR instruction (Begin Interrupt Service Routine) enables the interrupt system (ICR.IE is set to one), allows the modification of the CPU priority number (CCPN), and saves the Lower Context in the same manner as the SVLCX instruction.

2.3.8.2 Context Loading and Storing

The effective address of the memory area in which the context is stored to or loaded from is part of the Load or Store instruction. The effective address must resolve to a memory location aligned on a 16-word boundary; otherwise a data address alignment trap (ALN) is generated.

The STUCX instruction (Store Upper Context) stores the same context information that is saved with an implicit Upper Context save operation: Registers A10 – A15 and D8 – D15, and the current PSW and PCXI.

The LDUCX instruction (Load Upper Context) loads registers A10 – A15 and D8 – D15. The PSW and link word fields in the saved context in memory are ignored. The PSW, FCX, and PCXI are unaffected.

The STLCX instruction (Store Lower Context) stores the same context information that is saved with an explicit Lower Context save operation: Registers A2 – A7 and D0 – D7, together with the return address (RA) in A11 and the PCXI. The LDLCX instruction (Load Lower Context) loads registers A2 through A7 and D0 through D7. The saved return address and the link word fields in the context stored in memory are ignored. Registers A11/RA, FCX, and PCXI are not affected.

2.3.9 System Instructions

The system instructions allow user-mode and supervisor-mode programs to access and control various system services, including interrupts, and the TriCore's debugging facilities. There are also instructions that read and write the core registers, for both user and supervisor-only mode programs.

2.3.9.1 System Call

The SYSCALL instruction generates a system call trap, providing a secure mechanism for user-mode application code to request supervisor services. The system call trap — like other traps — vectors to the trap handler table, using the three-bit hardware-furnished trap class ID as an index. The trap class ID for system call traps is six. The trap identification number (TIN) is specified by an immediate constant in the SYSCALL instruction, and serves to identify the specific supervisor service that is being requested.

2.3.9.2 Synchronization Primitives

The TriCore architecture provides two synchronization primitives. These primitives provide a mechanism to software through which it can guarantee the ordering of various events within the machine.

DSYNC

The first primitive, DSYNC, provides a mechanism through which a data memory barrier can be implemented. The DSYNC instruction guarantees that all data accesses associated with instructions semantically prior to the DSYNC instruction are completed before any data memory accesses associated with an instruction semantically after DSYNC are initiated. This includes all accesses to the system bus and local data memory.

ISYNC

The second primitive, ISYNC, provides a mechanism through which the following can be guaranteed:

- If an instruction semantically prior to ISYNC make a software visible change to a piece of architectural state, then the effects of this change are seen by all instructions semantically after ISYNC. For example, if an instruction changes a code range in the

protection table, the use of an ISYNC will guarantee that all instructions after the ISYNC are fetched and matched against the new protection table entry.
- All cached states in the pipeline, such as loop cache buffers, are invalidated.

Operation of the ISYNC instruction is thus described as follows:

1. Wait until all instructions semantically prior to the ISYNC have completed.
2. Flush the CPU pipeline and cancel all instructions semantically after the ISYNC.
3. Invalidate all cached state in the pipeline.
4. Prefetch the next instruction after the ISYNC.

2.3.9.3 Access to the Core Special Function Registers

The TriCore accesses the CSFRs through two instructions: MFCR and MTCR. The MFCR instruction (Move From Core Register) moves the contents of the addressed CSFR into a data register. MFCR can be executed at any privilege level. The MTCR instruction (Move To Core Register) moves the contents of a data register to the addressed CSFR. To prevent unauthorized writes to the CSFRs, the MTCR instruction can only be executed at the supervisor privilege level.

The CSFRs are also mapped into the top of segment 15 in the memory address space. This mapping makes the complete architectural state of the core visible in the address map, which allows efficient debug and emulator support.

Note: It is not permitted for the core to access the CSFRs through this mechanism; it must use MFCR and MTCR.

There are no instructions allowing bit, bit field, or load-modify store accesses to the CSFRs. The RSTV instruction (Reset Overflow Flags) resets the overflow flags in the PSW, without modifying any of the other bits in the PSW. This instruction can be executed at any privilege level.

2.3.9.4 Enabling/Disabling the Interrupt System

For non-interruptible operations, the ENABLE and DISABLE instructions allow the explicit enabling and disabling of interrupts in user and supervisor modes. While disabled, an interrupt will not be taken by the CPU regardless of the relative priorities of the CPU and the highest interrupt pending. The only “interrupt” that will be serviced while interrupts are disabled is the NMI (non-maskable interrupt) since it is a trap.

If a user process accidentally disables interrupts for longer than a specified time, the Watchdog Timer can be used to recover.

Programs executing in supervisor mode can use the 16-bit Begin ISR (BISR) instruction to save the Lower Context of the current task, set the current CPU priority number, and re-enable interrupts (which are disabled by the processor when an interrupt is taken).

2.3.9.5 RET and RFE

The function return instruction (RET) is used to return from a function that was invoked via a CALL instruction. The return from exception instruction (RFE) is used to return from an interrupt or trap handler. The two instructions perform very similar operations; they restore the Upper Context of the calling function or interrupted task, and branch to the return address contained in register A11 (prior to the context restore operation). The instructions differ in the error checking they perform for call depth management. Issuing an RFE instruction when the current call depth (as tracked in the PSW) is nonzero generates a context nesting error trap. Conversely, a context call depth underflow trap is generated when an RET instruction is issued when the current call depth is zero.

2.3.9.6 Trap Instructions

The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the PSW's V and SV bits, respectively, are set (**Section 2.3.1**).

2.3.9.7 No Operation

Although there are many ways to represent a no-operation (for example, adding zero to a register), an explicit NOP instruction is included so that it can be easily recognized, allowing the CPU to minimize power consumption during its execution. For example, a sequence of NOP instructions in a loop could be used as a low-power state that has a very fast interrupt response time.

2.3.10 16-Bit Instructions

The 16-bit instructions are a subset of the 32-bit instruction set, chosen because of their frequency of static use. They significantly reduce static code size and thus provide a reduction in the cost of code memory and a higher effective instruction bandwidth. Because the 16-bit and 32-bit instructions all differ in the primary opcode, the two instruction sizes can be freely intermixed.

The 16-bit instructions are formed by imposing one or more of the following format constraints: smaller constants, smaller displacements, smaller offsets, implicit source, destination, or base address registers, and combined source and destination registers (the 2-operand format). In addition, the 16-bit load and store instructions support only a limited set of addressing modes.

The registers D15 and A15 are used as implicit registers in many 16-bit instructions. For example, there is a 16-bit compare instruction (EQ) that puts a Boolean result in D15, and a 16-bit conditional move instruction (CMOV) which is controlled by the Boolean in D15.

The 16-bit load and store instructions are limited to the register indirect (base plus zero offset), base plus offset (with implicit base or source/destination register), and post-increment (with default offset) addressing modes.

2.4 CPU Pipelines

This section describes the TC11IB CPU pipelines including the integer and load/store pipelines, and the loop pipeline.

2.4.1 CPU Pipeline Overview

As specified by the TriCore architecture, the TC11IB implements a pipelined, superscalar processor architecture that allows the execution of up to three instructions in parallel. The processor pipeline design reduces branch latency, data dependencies, and overall system complexity.

Two major pipelines perform integer operations and load/store operations. Each of these has four stages: Fetch (common to both), Decode, Execute, and Write-back. A third minor pipeline optimizes DSP loops. The three pipelines are illustrated in **Figure 2-25**.

2.4.2 Integer and Load/Store Pipelines

The Integer Pipeline executes the following operation types.

- Integer arithmetic and logical operations
- Bit-wise logical operations
- Multiply-accumulate (MAC) operations
- Integer division
- Conditional data jumps

The Load/Store Pipeline executes the following operation types.

- Load and Store operations
- Context-switch operations
- System operations
- Address arithmetic calculations
- Unconditional and conditional branch target calculations

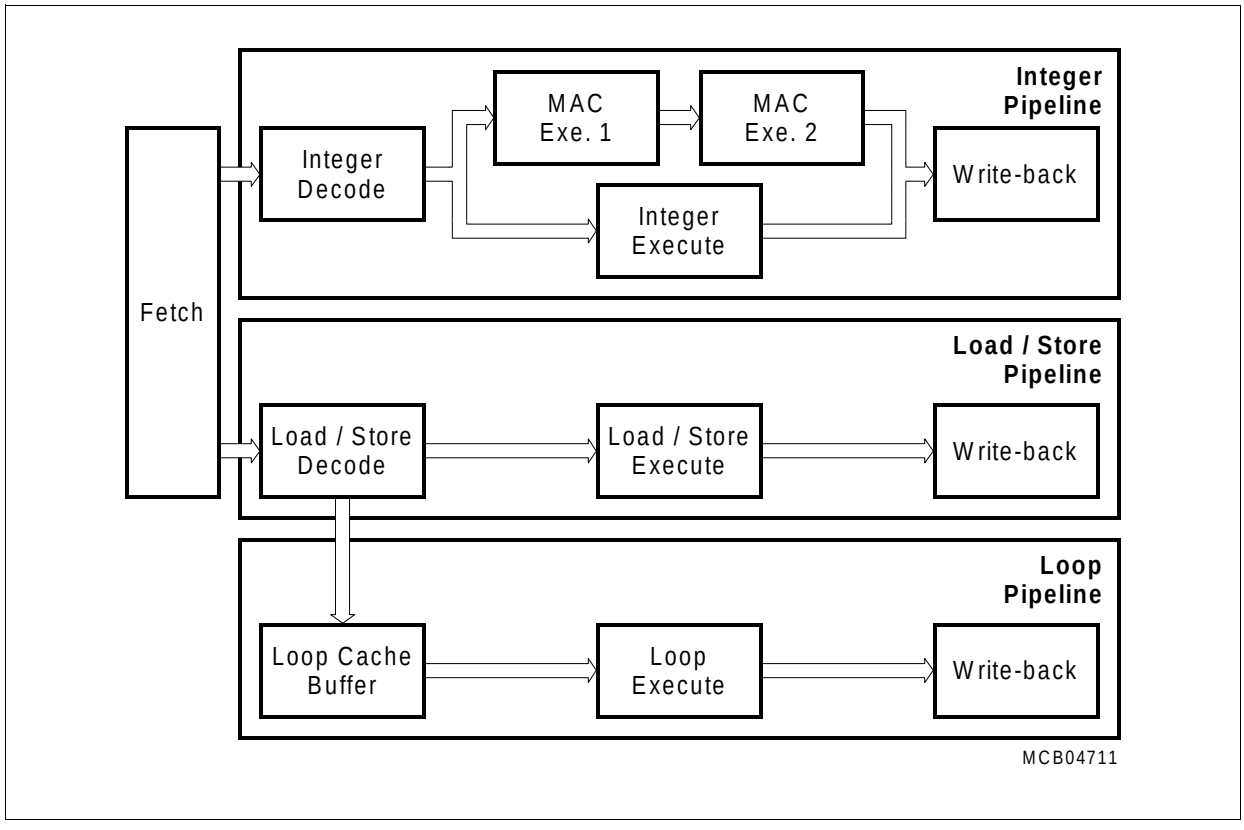


Figure 2-25 Pipeline Architecture

The pipelines share a common fetch stage that can issue one instruction to each pipeline per cycle. Certain issue constraints apply. For instance, when two instructions are issued in parallel, the first instruction must be an integer pipeline instruction. An integer ADD followed by a load instruction can be issued in parallel, but a load followed in the pair by an integer ADD cannot.

For example, the following code sequence takes four cycles.

```
add     d0, d1, d2
sub     d0, d0, d3
ld.w    d1, [a0]0
xor     d2, d1, d0
st.w    [a1]0, d2
ld.a    a0, [a5]4
```

Cycle	Integer	Load/Store
1	add	–
2	sub	ld.w
3	xor	st.w
4	–	ld.a

Note that on the third cycle, the XOR instruction and dependent store are dual-issued. The result from the XOR can be forwarded to the store instruction without any stall penalty. In general, all required forwarding paths are implemented so that dependent instructions can be executed without stall penalties.

All simple integer operations, bit operations, and address arithmetic instructions execute in a single cycle. Divide instructions, such as DVSTEP, require eight uninterruptable cycles to execute.

The multiply-accumulate (MAC) instructions are executed in a special two-stage MAC pipeline. The first stage contains two 16 × 16-bit multipliers. The second stage contains the accumulation, rounding and saturation logic. The MAC pipeline can perform a 32 × 32-bit multiply every two cycles with a latency of three cycles, or two 16 × 16-bit multiplies every cycle with a latency of two cycles.

2.4.3 Loop Pipeline

The Loop Pipeline optimizes the execution of loops, such as those typically found in DSP applications. This pipeline is driven by the Loop Cache Buffer (LCB), which stores the location, target, and other required information. The loop instruction is executed in the Load/Store Pipeline on its first iteration, and in the loop pipeline thereafter. If the loop is single-issued, the LCB is updated when it is detected in the decode stage of the pipeline. On subsequent iterations of the loop, when the LCB detects the end of the loop, it automatically fetches the start of the loop body. Unlike a normal Branch Target Buffer hit, the loop instruction itself is not fetched. It is injected from the LCB into the Loop Pipeline during the last execute cycle.

For example, the following code will execute as shown below:

```
        mov.a   a0, number_of_iterations - 1
loop_start:
        add     d0, d0, d1
        ld.w    d1, [a0+]4
        loop    d0, loop_start
```

Cycle	Integer	Load/Store	Loop
1	–	mov.a	–
2	add	ld.w	–
3	–	loop	–
4	–	–	–
5	add	ld.w	loop
6	add	ld.w	loop
7	add	ld.w	loop
8	add	ld.w	loop

As can be seen, after the first pass through the loop, each subsequent iteration will take only one clock cycle, thereby providing zero overhead loop capability.

2.4.4 Context Operations

Context save and context restore operations associated with calls, returns, interrupts, and so on, use the 128-bit data bus between the register file and the local on-chip data memory. The CPU contains dedicated hardware to optimize context switching, resulting in a context-save time to the on-chip local memory of between two and four cycles.

3 Clock System

This chapter describes the TC11IB's clock system. Topics covered include clock gating, clock domains, clock generation, the operation of clock circuitry, boot-time operation, fail-safe operation, clock control registers, and power management.

The TC11IB clock system performs the following functions:

- Acquires and buffers incoming clock signals to create master clock frequencies
- Distributes in-phase synchronized clock signals throughout the TC11IB's entire clock tree
- Divides system master clock frequencies into lower frequencies required by the different modules for operation.
- Dynamically reduces power consumption during operation in some functional units
- Statically reduces power consumption through programmable power-saving modes
- Reduces electromagnetic interference (EMI)

The clock system must be operational before the TC11IB can function, so it contains special logic to handle power-up and reset operations. Its services are fundamental to the operation of the entire system, so it contains special fail-safe logic.

Figure 3-1 shows the structure of the TC11IB clock system. The system clocks f_{SYS} are generated by the oscillator circuit and the phase-locked loop (PLL) unit. The module clocks are all derived from the system clocks. Each peripheral module can define a specific operation frequency of its module clock f_{MOD}.

The functionality of the control blocks shown in **Figure 3-1** varies depending on the functional unit being controlled. Some functional units, such as the FPI Bus or the watchdog timer, are directly driven by the system clocks. Detailed descriptions on the clock control register options for each unit are described in **Section 3.2**.

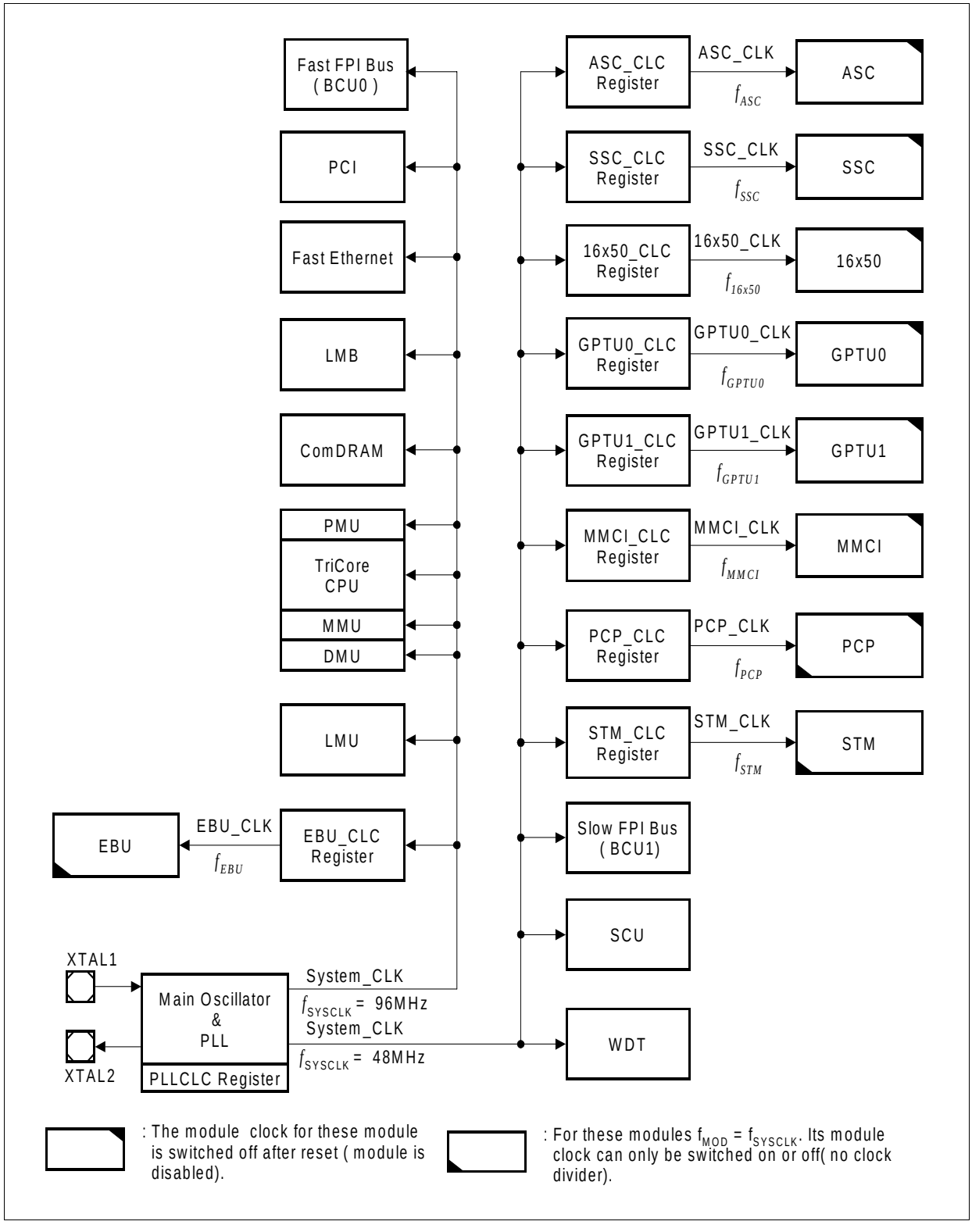


Figure 3-1 TC11IB Clocking System

3.1 Clock Generation Unit

The Clock Generation Unit in the TC11IB, shown in **Figure 3-2**, consists of an oscillator circuit and a Phase-Locked Loop (PLL). The PLL can convert a low-frequency external clock signal to a high-speed internal clock for maximum performance. The PLL also has fail-safe logic that detects to degenerate external clock behavior such as abnormal frequency deviations or a total loss of the external clock. It can execute emergency actions if it looses its lock on the external clock. PLL can provide the 96MHz and 48MHz clocks.

In general, the clock generation unit is controlled through the System Control Unit (SCU) module of the TC11IB.

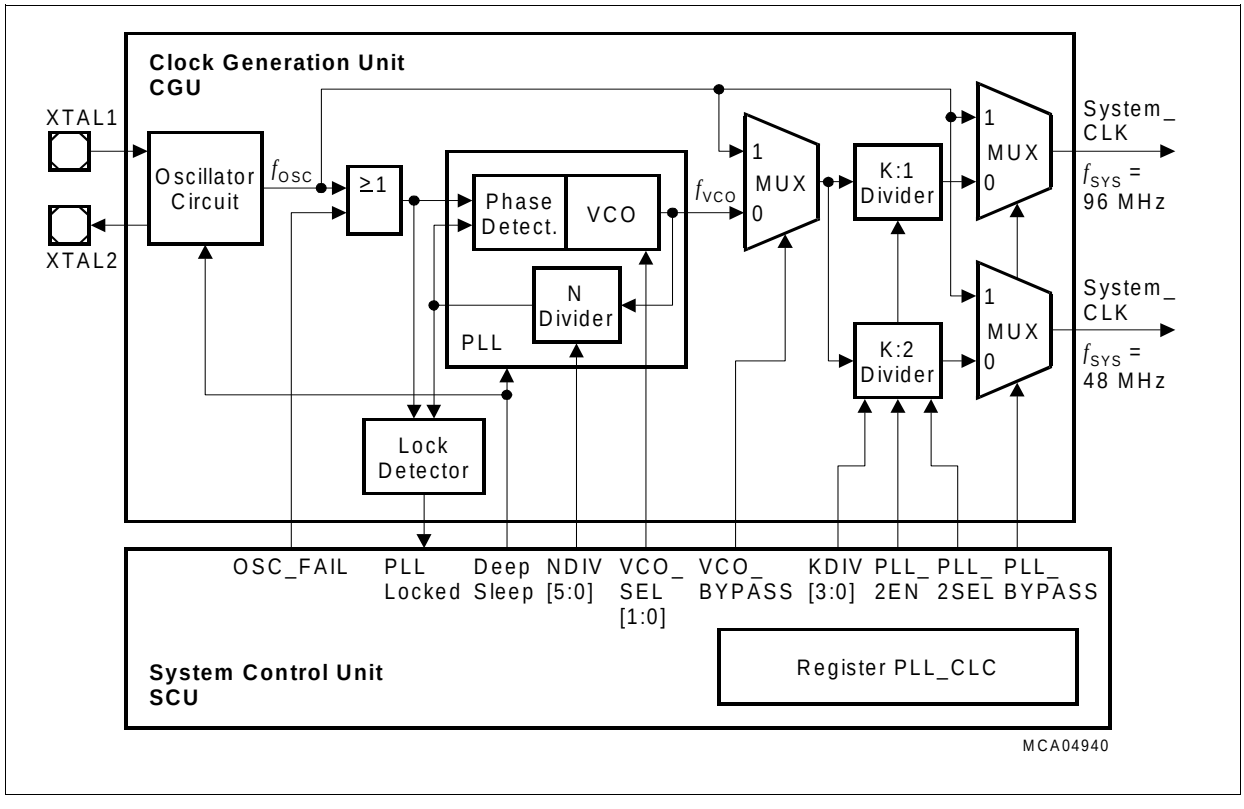


Figure 3-2 Clock Generation Unit Block Diagram

The following sections give descriptions of the various blocks of the clock generation unit.

3.1.1 Oscillator Circuit

The oscillator circuit, designed to work with both, an external crystal oscillator or an external stable clock source, basically consists of an inverting amplifier with XTAL1 as input, and XTAL2 as output.

When using a crystal, a proper external oscillator circuitry must be used, connected to both pins, XTAL1 and XTAL2. The on-chip oscillator frequency can be 12 MHz. When using an external clock signal it must be connected to XTAL1. XTAL2 is left open (unconnected).

Further specifications on the frequency limits of the clock circuitry are given in the TC11IB device specifications (Data Sheet).

Figure 3-3 shows the recommended external oscillator circuitries for both operating modes, external crystal mode and external input clock mode.

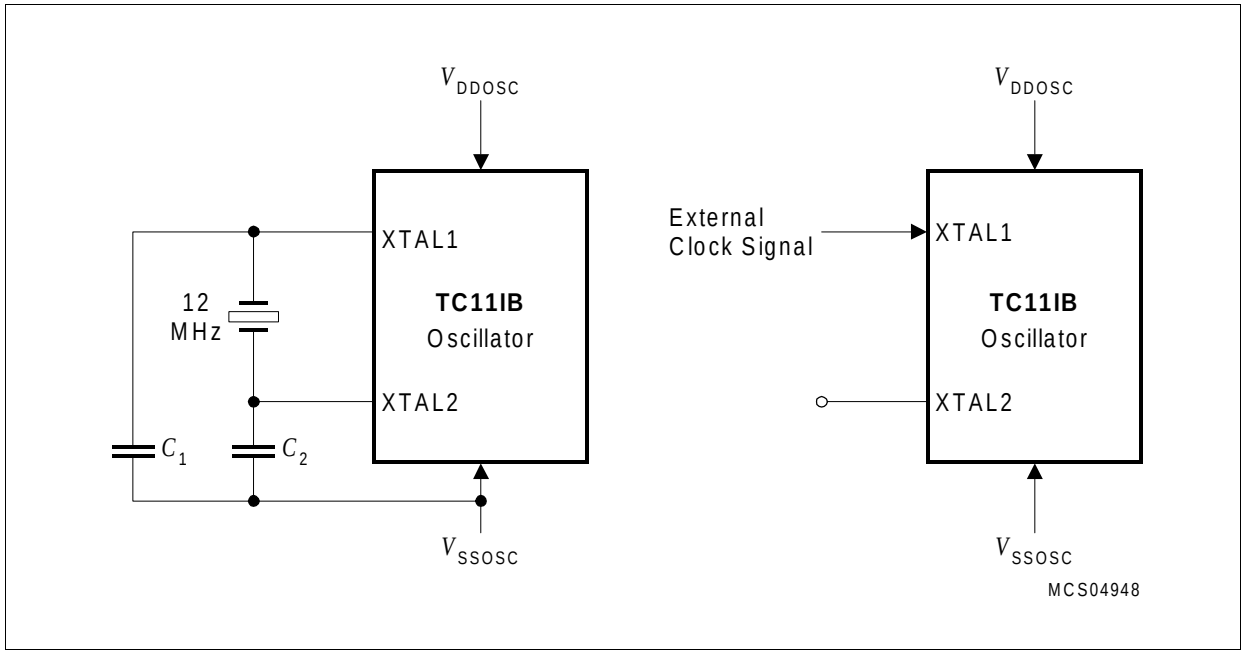


Figure 3-3 TC11IB Main Oscillator Circuitries

3.1.2 Phase-Locked Loop (PLL)

The PLL consists of a voltage controlled oscillator (VCO) with a feedback path. A divider in the feedback path divides the VCO frequency down. The resulting frequency is then compared to the externally applied frequency. The phase detection logic determines the difference between the two clock signals and accordingly controls the frequency of the VCO. During start-up, the VCO increases its frequency until the divided feedback clock matches the external clock frequency. A lock detection logic monitors and signals this condition. The phase detection logic continues to monitor the two clock signals and adjusts the VCO clock if required.

Due to this operation, the VCO clock of the PLL has a frequency which is a multiple of the externally applied clock. The factor for this operation is controlled through the value applied to the divider in the feedback path. That is why this factor is often called a multiplier, although it actually controls a divider.

3.1.2.1 N-Divider

The feedback N-divider is fixed at 16 by hardware in TC11IB.

3.1.2.2 VCO Frequency Ranges

Stable and reliable operation of the VCO, and minimization of the jitter (the frequency variations of the VCO output between adjustment points), is critical for precise clock generation. To provide optimum behavior, the following frequency range for f_{VCO} must be selected:

$$150\ \text{MHz} \leq f_{VCO} \leq 200\ \text{MHz} \quad [3.1]$$

3.1.2.3 Lock Detection

A lock detector circuit determines whether the PLL is locked appropriately to the external clock signal, and indicates the PLL lock state to the SCU. If the PLL looses synchronization to the external clock due to a failure of the external clock, the SCU detects this case and shuts off the oscillator input to the VCO via activation of the OSC_FAIL signal.

3.1.2.4 K-Divider

The K-Divider is a software controlled divider. The bit field KDIV is provided in register PLL_CLC. Software can write to this field in order to change the system frequency f_{SYS}.

The divider is designed such that a synchronous switching of the clock is performed without spurious or shortened clock pulses when software changes the divider factor KDIV. However, special attention has to be paid concerning the effect of such a clock change to the various modules in the system. For instance, changing the clock frequency while an external memory access is performed by the EBU could result in a failure of the access. It is strongly recommended to perform clock frequency changes only when no critical system operations are in progress to avoid hazardous effects.

3.1.2.5 Enable/Disable Control

If Deep Sleep Mode is selected, the PLL is shut off by the SCU via the DEEP_SLEEP signal. In Deep Sleep Mode, also the main oscillator circuit is disabled.

3.1.3 Determining the System Clock Frequency

The system clocks, the 96MHz and 48MHz clock, are obtained from the first and second outputs respectively via using the oscillator clock of 12MHz multiplied by the PLL and optionally divided by the K-divider. The system clock frequency f_{SYS} can be made proportional to the ratio N / K, where the clock scale factor N = 16, and bit field PLL_CLC.KDIV determines the clock scale factor K.

The VCO output frequency is determined by

$$f_{VCO} = 16 \times f_{OSC} \quad [3.2]$$

and the resulting system clock is determined by

$$f_{SYS-FIRST} = f_{VCO}/K = \frac{16}{K} \times f_{OSC} \quad [3.3]$$

$$f_{SYS-SECOND} = f_{VCO}/K = \frac{16}{2K} \times f_{OSC} \quad [3.4]$$

Both, VCO_BYPASS and PLL_BYPASS, must be inactive for this PLL operation.

3.1.4 PLL Clock Control and Status Register

The PLL Clock Control and Status Register PLL_CLC is located in the address range reserved for the System Control Unit (SCU). It holds the hardware configuration bits of the PLL, latched at the end of power-on reset, and provides the control for the K-Factor as well as the PLL Lock status bit.

Note that register PLL_CLC is specially protected. In order to write to PLL_CLC, the WDT_CON0.ENDINIT bit must be set to 0 through a password-protected access mechanism to register WDT_CON0.

The indicator “U” in the reset value of PLL_CLC indicates that the reset values for these bits are user-defined through the value applied to the PLL configuration pins.

PLL_CLC
PLL Clock Control Register **Reset Value: 000F 01UU$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0										NDIV					
r										r					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0				KDIV				VCOSEL		VCO BYP	PLL BYP	0	2SEL	2EN	LOCK
r				rw				r		r	r	r	r	r	r

Field	Bits	Type	Description
LOCK	0	r	**PLL Lock Status Flag** 0 PLL is not locked 1 PLL is locked

Field	Bits	Type	Description
2EN	1	r	**PLL 2nd Output Clock Enable** Fixed at ‘1b’, always enabled.
2SEL	2	r	**PLL 2nd Output Clock Ratio** Fixed at ‘0b’. Ratio is 1: 2.
PLLBYP	4	r	**PLL Bypass Status Flag** Indicates the state of the BYPASS control input line latched with the signal decoded from the JTAG input pins (for test only). 0 Normal operation 1 PLL bypass mode
VCOBYP	5	r	**VCO Bypass Status Flag** Indicates the state of the VCO_BYPASS control input line latched with the signal decoded from the JTAG input pins. (for test only) 0 Normal operation 1 VCO Bypass mode (PLL output clock is derived from input clock divided by K-Divider)
VCOSEL	[7:6]	r	**VCO Range Select** Fixed at ‘01b’. (150 - 200MHz)
KDIV	[11:8]	rw	**PLL K-Factor Selection** The reset value is ‘0001b’. (K = 2)
NDIV	[21:16]	r	**PLL N-Factor Selection** Fixed at ‘001111b’. (N = 16)
0	3, [15:12], [31:22]	r	**Reserved**; Returns 0 if read; should be written with 0.

3.1.5 Startup Operation

When power is switched on to the TC11IB, a low level has to be applied to the power-on reset pin, $\overline{\text{PORST}}$. PLL begins operation at this time. The voltage controlled oscillator (VCO) of the PLL will start up very quickly and generate an internal clock with the PLL base frequency. As soon as the external crystal provides a stable clock frequency, PLL will lock to this frequency according to the various PLL factors which are fixed by hardware internally. This state is signalled by setting bit PLL_CLC.LOCK.

The low level at pin $\overline{\text{PORST}}$ has to be held long enough to make sure that a stable clock is provided to the TC11IB. In case of an external crystal oscillator, it can take several ms until the oscillator has started up and is stable. If the clock input is provided by another

clock source with faster startup characteristics, the requirements for the $\overline{\text{PORST}}$ low level can be relaxed accordingly.

The TC11IB remains in power-on reset state until PLL is locked and the $\overline{\text{PORST}}$ pin has been de-asserted. Two situations are possible when $\overline{\text{PORST}}$ becomes inactive (low-to-high transition):

1. PLL is not locked (PLL_CLC.LOCK = 0):
   The PLL provides an emergency clock at PLL base frequency. The system clock will be at this PLL base frequency as long as the LOCK bit is not set. In this case, a program should wait for bit LOCK to be set before it proceeds time critical initialization procedures and operations.
2. PLL is locked (PLL_CLC.LOCK = 1):
   The PLL is already at its nominal frequency.

3.1.6 PLL Loss of Lock Operation

The PLL provides mechanisms to detect a failure of the external clock and to bring the TC11IB into a safe state in such a case. If the PLL loses the lock to the external clock, either due to a break of the crystal or an external line, it resets its lock line PLL_LOCK. The clock control circuitry then sets the PLL Loss of Clock NMI flag (PLLNMI) in register NMISR and activates a NMI trap request to the CPU. In addition, it disables the oscillator input clock f_{OSC} to the PLL to avoid unstable operation due to noise or sporadic clock pulses coming from the oscillator circuit and the PLL still trying to lock onto this invalid clocks. Without having an input clock, the PLL gradually slows down to its base frequency and remains there. While this frequency is defined within a certain frequency range, emergency actions can be taken by the CPU since the TC11IB is still clocked.

The TC11IB remains in this state until the next power-on reset through pin $\overline{\text{PORST}}$, where then the PLL tries to restart and lock to the external clock again. No other reset cause can terminate this loss-of-clock state to avoid unstable operation due to the PLL trying to lock again.

Note that the loss of lock state does not refer to the start-up process during power-on or wake-up from deep sleep mode. The oscillator input clock to PLL thus is not disabled at such situations.

3.2 Power Management and Clock Gating

Because power dissipation is related to the frequency of gate transitions, the TC11IB performs power management principally by clock gating - that is, controlling whether the clock is supplied to its various functional units. Gating off the clock to unused functional modules also reduces electromagnetic interference (EMI) since EMI is related to both the frequency and the number of gate transitions.

Clock gating is done either dynamically or statically. Dynamic clock gating in this context means that the TC11IB itself enables or disables clock signals within some functional

modules to conserve power. Static gating means that software must enable or disable clock signals to functional modules. Clock gating is performed differently at different levels of system scope: dynamic gating is generally performed at the lowest levels, either within a small region of logic, or at functional-unit boundaries for uncomplicated functions where hardware can dynamically determine whether that functionality is required, and can enable or disable it appropriately without software intervention. Static gating - which requires software intervention - is used to enable or disable clock delivery to individual high-level functional units, or to disable clock delivery globally at the clock's source. When the clock to individual functional units is gated off, they are said to be in Sleep Mode. When the TC11IB's clock is gated off at its source, the TC11IB as a whole is said to be in Deep Sleep Mode.

The TC11IB implements four levels of clock gating:

1. Gated dynamically at the register
   The clock is shut off to a particular local resource in a functional module when this resource is not being used in that clock cycle. This operation is done primarily in the CPU and the PCP data paths, where unused resources are easily identified and controlled in each clock cycle.
2. Gated dynamically at the functional unit (Idle Mode)
   The clock is shut off at the functional unit boundary when the unit has nothing useful to do. This operation is done primarily in the CPU and the PCP. For the CPU, idle mode is controlled via software. The PCP disables its own clock when no program is running.
3. Gated statically at each functional unit (Sleep Mode)
   Software can send a global sleep request to individual functional units requesting that they enter Sleep Mode. Software must determine when conditions are such that entering Sleep Mode is appropriate. The individual units can be programmed to ignore or respond to this signal. If programmed to respond, units will first complete pending operations, then will shut off their own clocks according to their own criteria.
4. Gated at the clock source (Deep Sleep Mode)
   The PLL and oscillator are shut off, thereby gating the clock to all functional units. The system can only be restored to operation by receiving a power-on reset signal from the $\overline{\text{PORST}}$ pin or a non-maskable interrupt signal from the $\overline{\text{NMI}}$ pin. Entering Deep Sleep Mode is under software control. Software must determine when conditions are such that entering Deep Sleep Mode is appropriate.

3.2.1 Clock Control

The functionality of the clock control registers varies depending on the functional unit being controlled. The clock for the CPU is controlled by the CPU hardware itself. The clock is switched off to the CPU automatically during Idle and Sleep Mode.

The PCP also controls its own clock automatically. Whenever the PCP is idle - that is when no channel program is running - the PCP shuts off its clock. It will automatically re-

enable the clock again when a PCP interrupt is detected. The PCP also controls the clocking of the PICU and PCP interrupt arbitration logic. The Peripheral Interrupt Control Unit (PICU) arbitrates service requests for the PCP and administers the PCP Interrupt Arbitration Bus.

The Slow FPI Bus, Fast FPI Bus and LMB Bus clocks have no clock control feature. They always run at their own system clock frequencies f_{SYS}. Slow FPI Bus runs at 48MHz. Fast FPI Bus and LMB Bus run at 96MHz. However, these Buses are so designed that no signal lines are switching when there is no activity on the bus. Hence, power consumption of these buses is minimized by design.

Similar operation applies to the CPU interrupt system. It runs with the system clock frequency f_{SYS}, however, signal lines do only change when activity, such as an arbitration round, is required.

The on-chip peripheral units of the TC11IB, including the GPTU0, GPTU1, ASC, SSC, 16X50, MMCI, and the System Timer (STM) each have dedicated clock-control registers. The generic name of these registers is given in this chapter as CLC. All clock control registers have the same bit field layout, however not all peripheral units implement all functions of these registers. In general, these registers control on/off state, clock frequency for Run Mode, operation in Sleep Mode, and operation during Debug Suspend Mode.

PCI interface has its own power management.

3.2.2 Module Clock Generation

As shown in **Figure 3-1** most of the of on-chip peripheral modules of the TC11IB have clock control registers implemented. The generic name of these registers is “CLC”. This section describes the general functionality of these CLC registers.

All CLC registers have basically the same bit and bit field layout. However, not all CLC register functions are implemented for each peripheral unit. **Table 3-1** defines in detail which bits and bit fields of the CLC registers are implemented for each peripheral module.

The CLC register basically controls the generation of the peripheral module clock which is derived from the system clock. The following functions for the module are associated with the CLC register:

- Peripheral clock static on/off control
- Peripheral clock frequency in Run Mode
- Peripheral clock frequency/behavior in Sleep Mode
- Operation during Debug Suspend Mode

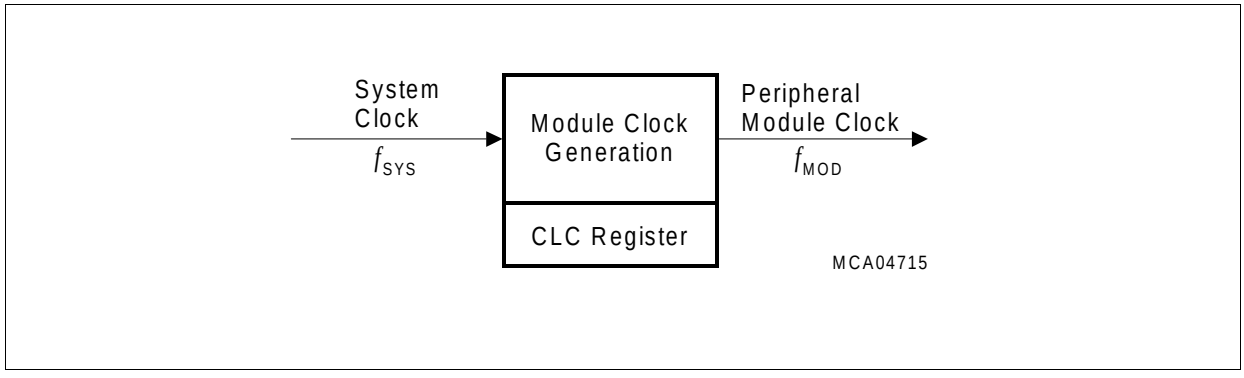


Figure 3-4 Module Clock Generation

3.2.3 Clock Control Registers

MOD_CLC
Clock Control Register **Reset Value: Module Specific**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RMC								0	0	FS OE	SB WE	E DIS	SP EN	DIS S	DIS R
rw								r	r	rw	w	rw	rw	r	rw

Field	Bits	Type	Description
DISR	0	rw	**Module Disable Request Bit** Used for enable/disable control of the module. 0 Module disable is not requested 1 Module disable is requested
DISS	1	r	**Module Disable Status Bit** Bit indicates the current status of the module 0 Module is enabled 1 Module is disabled

Field	Bits	Type	Description
SPEN	2	rw	**Module Suspend Enable** Used for enabling the suspend mode. 0 Module cannot be suspended (suspend is disabled). 1 Module can be suspended (suspend is enabled). This bit is writable only if SBWE is set to 1 during the same write operation.
EDIS	3	rw	**Sleep Mode Enable Control** Used for module sleep mode control. 0 Sleep mode request is regarded. Module is enabled to go into sleep mode. 1 Sleep mode request is disregarded: Sleep mode cannot be entered on a request.
SBWE	4	w	**Module Suspend Bit Write Enable for OCDS** Defines whether SPEN and FSOE are write protected. 0 Bits SPEN and FSOE are write protected 1 Bits SPEN and FSOE are overwritten by respective value of SPEN or FSOE This bit is a write only bit. The value written to this bit is not stored. Reading this bit returns always 0.
FSOE	5	rw	**Fast Switch Off Enable** Used for fast clock switch off in OCDS suspend mode. 0 Clock switch off in OCDS suspend mode via Disable Control Feature (Secure Clock Switch Off) 1 Fast clock switch off in OCDS suspend mode This is writable only if SBWE is set to 1 during the same write operation.
RMC	[15:8]	rw	**8-Bit Clock Divider Value in Run Mode** Max. 8-bit divider value If RMC is set to 0 the module is disabled.
0	7, 6, [31:16]	r	**Reserved**; returns 0 if read; should be written with 0;

Module Enable/Disable Control

If a module is not used at all by an application, it can be completely shut off by setting bit DISR in its clock control register. For peripheral modules with a run mode clock divider field RMC, a second option to completely switch off the module is to set bit field RMC to 00_H. This also disables the module's operation.

The status bit DISS always indicates whether a module is currently switched off (DISS = 1) or switched on (DISS = 0). With a few exceptions (STM_CLC, EBU_CLC), the default state of a peripheral module after reset is “module disabled” with DISS set.

Write operations to the registers of disabled modules are not allowed. However, the CLC of a disabled module can be written. An attempt to write to any of the other writable registers of a disabled module except CLC will cause the Bus Control Unit (BCU) to generate a bus error.

A read operation of registers of a disabled module is allowed and does not generate a bus error.

Note: A destructive read access occurring while a module is disabled is treated as a normal read access. This means, if a module register or a bit of it is cleared as a side-effect of a read access of an enabled module, it will not be cleared by this read access while the module is disabled.

Sleep Mode Control

The EDIS bit in the CLC register controls whether a module is stopped during sleep mode or not. If EDIS is 0 (default after reset), a sleep mode request can be recognized by the module and, when received, its clock is shut off.

If EDIS is set to 1, a sleep mode request is disregarded by the module and the module continues its operation.

Debug Suspend Mode Control

During emulation and debugging of TC11IB applications, the execution of an application program can be suspended. When an application is suspended, normal operation of the application’s program is halted, and the TC11IB begins (or resumes) executing a special debug monitor program. When the application is suspended, a suspend signal is generated by the TC11IB and sent to all modules. If bit SPEN is set to 1, the operation of the peripheral module is stopped when the suspend signal is asserted. If SPEN is set to 0, the module does not react to the suspend signal but continues its normal operation. This feature allows each peripheral module to be adapted to the unique requirements of the application being debugged. Setting SPEN bits is usually performed by a debugger.

This feature is necessary because application requirements typically determine whether on-chip modules should be stopped or left running when an application is suspended for debugging. For example, a peripheral unit that is controlling the motion of an external device through motors in most cases must not be stopped so as to prevent damage of the external device due to the loss of control through the peripheral. On the other hand, it makes sense to stop the system timer while the debugger is actively controlling the chip because it should only count the time when the user’s application is running.

Note that it is never appropriate for application software to set the SPEN bit. The debug suspend mode should only be set by a debug software. To guard against application

software accidently setting SPEN, bit SPEN is specially protected by the mask bit SBWE. The SPEN bit can only be written if, during the same write operation, SBWE is set, too. Application software should never set SBWE to 1. In this way, user software can not accidentally alter the value of the SPEN bit that has been set by a debugger.

Note: The operation of the Watchdog Timer is always automatically stopped in debug suspend mode.

Entering Disabled Mode

Software can request that a peripheral unit shall be put into Disabled Mode by setting DISR. A module will also be put into Disabled Mode if the sleep mode is requested and the module is configured to allow Sleep Mode.

In Secure Shut-off Mode, a module first finishes any operation in progress, then proceeds with an orderly shut down. When all sub-components of the module are ready to be shut down, the module signals its clock control unit, which turns off the clock to this peripheral unit, that it is now ready for shut down. The status bit DISS is updated by the peripheral unit accordingly.

The kernel logic of the peripheral unit and its FPI Bus interface must both perform shut-down operations before the clock can be shut off in Secure Shut-off Mode. This is performed as follows. The peripheral module's FPI Bus interface provides an internal acknowledge signal as soon as any current bus interface operation is finished. For example, if there is a PCP write access to a peripheral in progress when a disable request is detected, the access will be terminated correctly. Similarly, the peripheral's kernel provides an internal acknowledge signal when it has entered a stable state. The clock control unit for that peripheral module shuts off the module's clock when it receives both acknowledge signals.

During emulation and debugging, it may be necessary to monitor the instantaneous state of the machine - including all or most of its modules - at the moment a software breakpoint is reached. In such cases, it may not be desired that the kernel of a module finish whatever transaction is in progress before stopping, because that might cause important states in this module to be lost. Fast Shut-off Mode, controlled by bit FSOE, is available for this situation.

If FSOE = 0, modules are stopped as described above. This is called Secure Shut-off Mode. The module kernel is allowed to finish whatever operation is in progress. The clock to the unit is then shut off if both the bus interface and the module kernel have finished their current activity. If Fast Shut-off Mode is selected (FSOE = 1), clock generation to the unit is stopped as soon as any outstanding bus interface operation is finished. The clock control unit does not wait until the kernel has finished its transaction. This option stops the unit's clock as fast as possible, and the state of the unit will be the closest possible to the time of the occurrence of the software breakpoint.

Note: The Fast Shut-off Mode is the only shut down operating mode available in the TC11IB, regardless of the state of the FSOE bit.

Whether Secure Shut-off Mode or Fast Shut-off Mode is required depends on the application, the needs of the debugger, and the type of unit. For example, the analog-to-digital converter might allow the converter to finish a running analog conversion before it can be suspended. Otherwise the conversion might be corrupted and a wrong value could be produced when Debug Suspend Mode is exited and the unit is enabled again. This would affect further emulation and debugging of the application's program.

On the other hand, if a problem is observed to relate to the operation of the external analog-to-digital converter itself, it might be necessary to stop the unit as fast as possible in order to monitor its current instantaneous state. To do this, the Fast Shut-off Mode option would be selected. Although proper continuation of the application's program might not be possible after such a step, this would most likely not matter in such a case.

Note that it is never appropriate for application software to set the FSOE bit. Fast Shut-off Mode should only be set by debug software. To guard against application software accidently setting FSOE, bit FSOE is specially protected by the mask bit SBWE. The SPEN bit can only be written if, during the same write operation, SBWE is set, too. Application software should never set SBWE to 1. In this way, user software can not accidentally alter the value of the FSOE bit. Note that this is the same guard mechanism used for the SPEN bit. In this way, user software can not accidentally alter the value of the FSOE bit.

Module Clock Divider Control

Most of the TC11IB peripheral modules have an 8-bit or 2-bit control field in their CLC registers for Run Mode clock control (RMC). The clock divider circuit is located in the bus interface of these peripheral modules.

A value of 00_H in RMC disables the clock signals to these modules (module clock is switched off). If RMC is not equal to 00_H, the module clock for a unit is generated

$$f_{MOD} = f_{SYS} / RMC_{MOD} \quad [3.5]$$

where "MOD" stands for the module name and "RMC_{MOD}" is the content of its CLC register RMC field with a range of 1..255.

Note: The number of module clock cycles (wait states) which are required for a "destructive read" access (means: flags/bits are set/reset by a read access) to a module register of a peripheral unit depends on the selected module clock frequency.
Therefore, a slower module clock (selected via bit field RMC in the CLC register) results in a longer read access time for peripheral units with "destructive read" access (e.g. ASC, SSC).

3.2.4 CLC Register Implementations

Table 3-1 shows which of the CLC register bits/bit fields is implemented for each peripheral module in the TC11IB.

Table 3-1 CLC Registers in the TC11IB

Register	Module		DISS, DISR, Bit [1:0]	SPEN Bit 2	EDIS Bit 3	SBWE Bit 4	FSOE Bit 5	RMC Bit [15:8]
	Name	State after Reset						
SSC_CLC	SSC	disabled	■	■	■	■	■	■
ASC_CLC	ASC	disabled	■	■	■	■	■	■
16x50_CLC	16x50	disabled	■	■	■	■	■	■
GPTU0_CLC	GPTU0	disabled	■	■	■	■	■	■
GPTU1_CLC	GPTU1	disabled	■	■	■	■	■	■
EBU_CLC	EBU	enabled	■	–	–	–	–	–
STM_CLC	STM	enabled	■	■	■	■	■	–
MMCI_CLC	MMCI	enabled	■	■	■	■	■	–
PLL_CLC	PLL	enabled	completely different bit definitions (see **Section 3.1.4**)					

Note: The ports of the TC11IB don't provide CLC registers.

4 System Control Unit

4.1 Overview

The System Control Unit (SCU) of the TC11IB handles the system control tasks. All these system functions are tightly coupled, thus, they are conveniently handled by one unit, the SCU. The system tasks of the SCU are:

- Reset Control (described in **Chapter 5**)
  - Generation of all internal reset signals
  - Generation of external $\overline{\text{HDRST}}$ reset signal
- PLL Control (described in **Chapter 3**)
  - PLL_CLC Clock Control Register
- Power Management Control (described in **Chapter 6**)
  - Enabling of several power-down modes
  - Control of the PLL in power-down modes
- Watchdog Timer (described in **Chapter 20**)
- Trace Control
- Device Identification

This chapter describes the last two tasks in this feature list. The other tasks are described in other chapters of this document, as indicated.

4.2 Registers Overview

The basic SCU registers can be divided into three types, as shown in **Figure 4-1**. **Table 4-1** provides the long name, offset address, and location details for each of the basic registers.

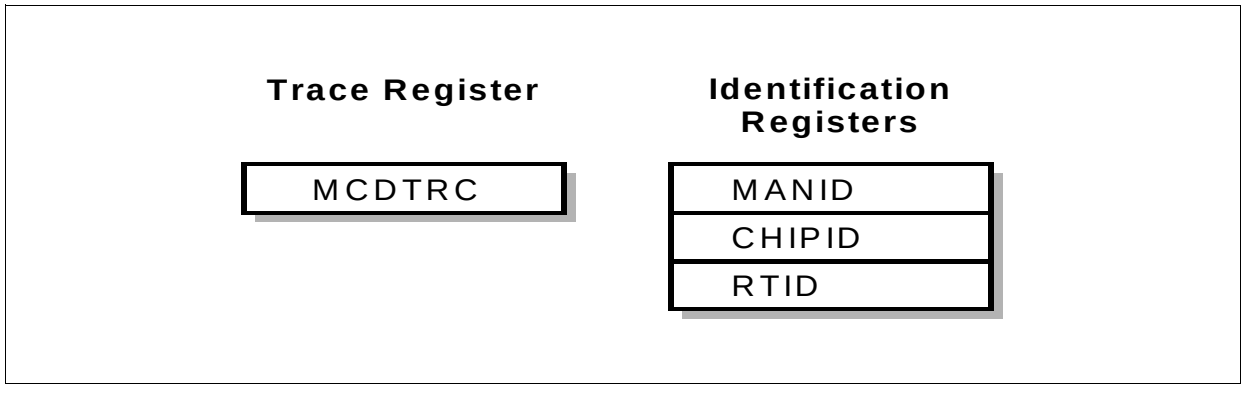


Figure 4-1 SCU Registers

Table 4-1 SCU Registers

Register Short Name	Register Long Name	Offset Address	Description see
SCU_MCDTRC	Trace Control Register	0054_H	**Page 4-4**
MANID	Manufacturer Identification Register	0070_H	**Page 4-5**
CHIPID	Chip Identification Register	0074_H	**Page 4-6**
RTID	Redesign Tracing Identification Register	0078_H	**Page 4-7**

In the TC11IB, the registers of the SCU are located in the following address range:

– Module Base Address: $F000\ 0000_H$
  Module End Address: $F000\ 00FF_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 4-1**)

4.3 Trace Control

This part of the SCU controls the interconnections of Trace Port with the trace interfaces of the Trace Control Unit (TCU) and the Peripheral Control Processor (PCP) and CPU.

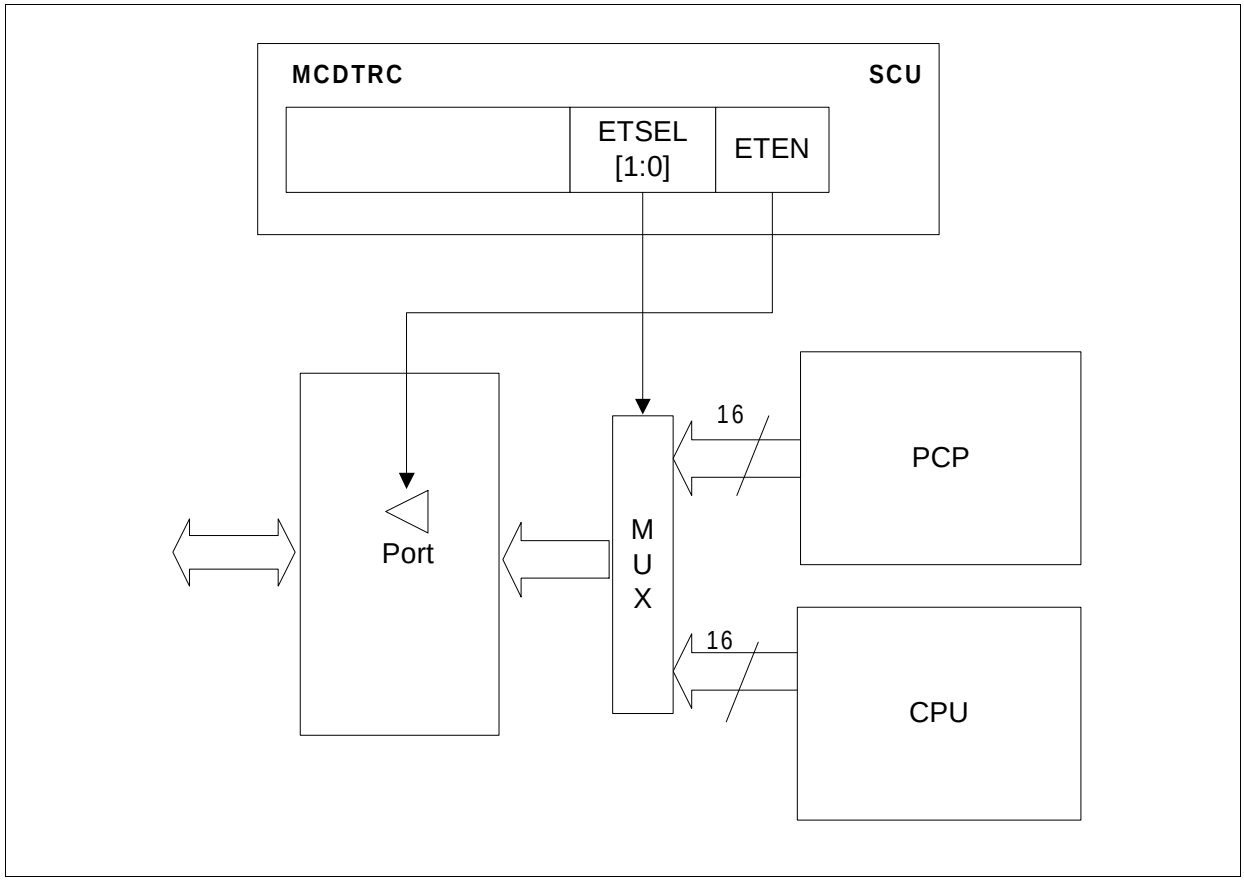
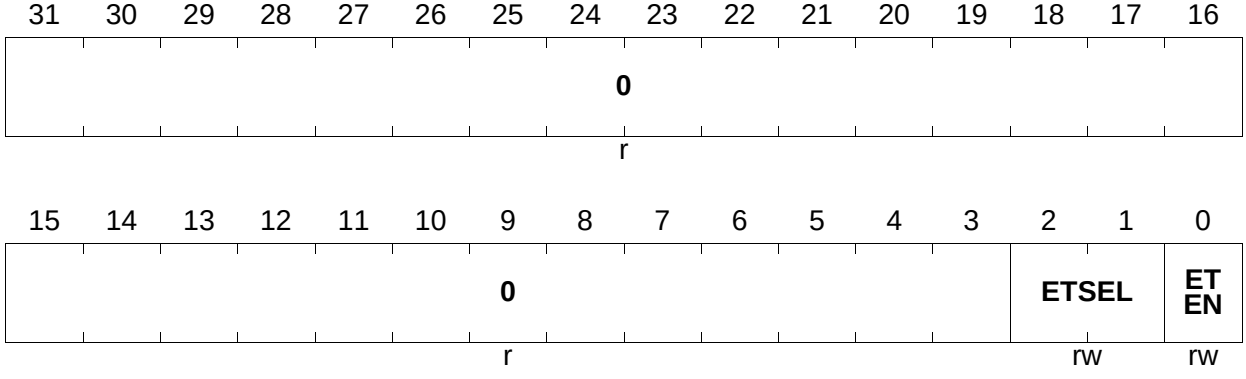


Figure 4-2 Trace Control within the SCU

*Note: The trace features of the TC11IB are described in detail in **Chapter 21** of this User's Manual.*

SCU_MCDTRC
SCU Trace Control Register

Reset Value: 0000 0000$_H$

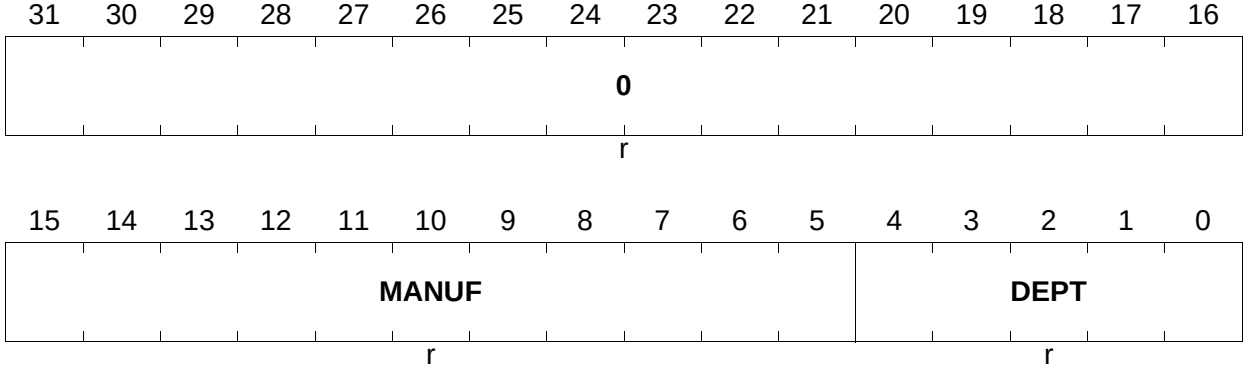


Field	Bits	Type	Description
ETEN	0	rw	**Emulation Trace Enable** 0 Emulation trace disabled 1 Emulation trace enabled
ETSEL	[2:1]	rw	**Emulation Trace Select** 00 CPU trace selected 01 PCP trace selected 10 Reserved 11 Reserved
0	[31:3]	r	**Reserved**; read as 0; should be written with 0.

4.4 Identification Registers

The SCU includes four identification register: one for the SCU module identification and three for device identification.

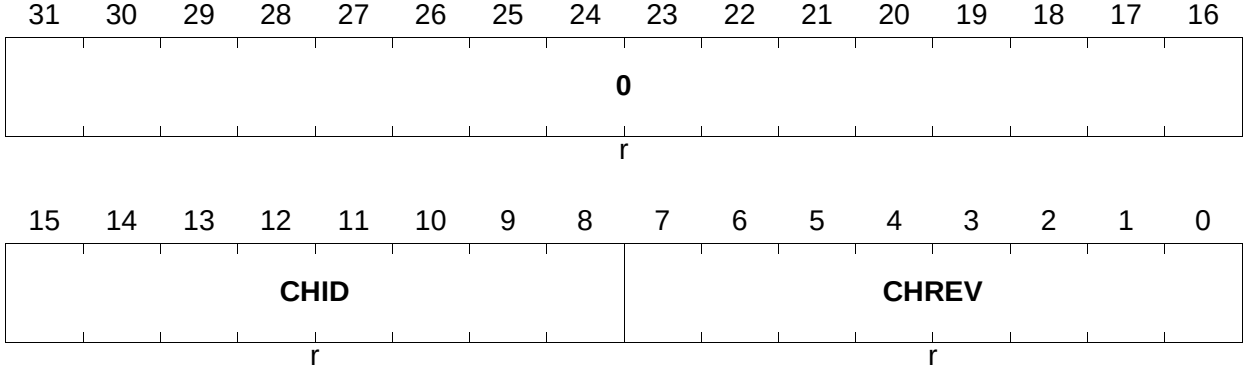
MANID
Manufacturer Identification Register **Reset Value: 0000 1820$_H$**



Field	Bits	Type	Description
DEPT	[4:0]	r	**Department Identification Number** = 00_H: indicates the department AI MC within Infineon Technologies.
MANUF	[15:5]	r	**Manufacturer Identification Number** This is a JEDEC normalized manufacturer code. MANUF = $C1_H$ for Infineon Technologies.
0	[31:16]	r	**Reserved**; read as 0.

CHIPID
Chip Identification Register

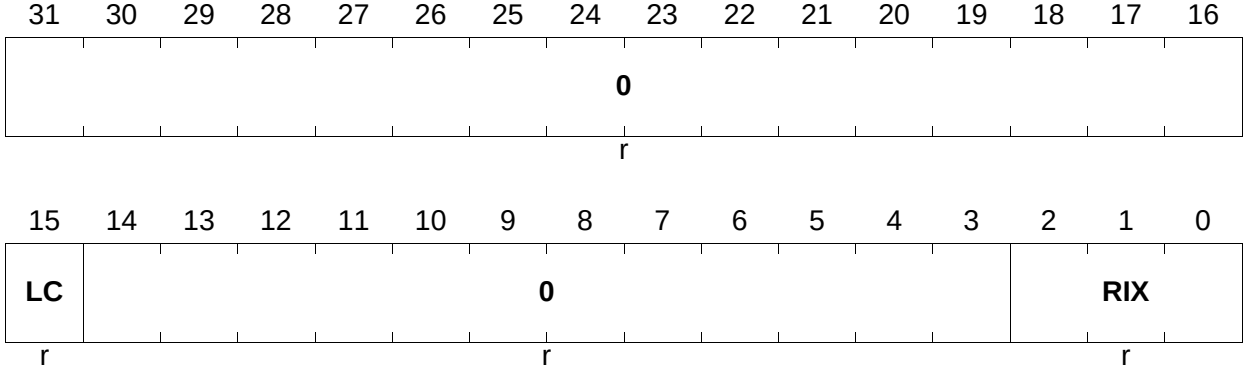
Reset Value: 0000 85XX$_H$



Field	Bits	Type	Description
CHREV	[7:0]	r	**Chip Revision Number** 01_H = first revision
CHID	[15:8]	r	**Chip Identification Number** 85_H = TC11IB
0	[31:16]	r	**Reserved**; read as 0.

RTID
Redesign Tracing Identification Register

Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
RIX	[2:0]	r	**Redesign Index** 0_H Original revision 1_H-7_H Modified revisions
LC	15	r	**Laser Correction Flag** 0 No laser correction 1 Laser correction
0	[14:3], [31:16]	r	**Reserved**; read as 0.

5 Reset and Boot Operation

This chapter describes the conditions under which the TC11IB will be reset, the reset and boot operations, and the available boot options.

5.1 Overview

When the TC11IB device is first powered up, several boot parameters must be defined to enable proper start operation of the device, such as the start location of the code. To accomplish parameter definition, the device has a separate Power-On Reset ($\overline{\text{PORST}}$) pin and a number of configuration pins that are sampled during the power-on reset sequence. At the end of this sequence, the sampled values are latched, and cannot be modified until the next power-on reset. This guarantees stable conditions during the normal operation of the device.

There are two ways to reset the device while it is operating: a hardware reset or a software reset. For reset causes coming from the external world, a reset input pin, $\overline{\text{HDRST}}$, is provided. If software detects conditions which require the device to be reset, a software reset can be performed by writing to a special register, the Reset Request (RST_REQ) register.

The Watchdog Timer (WDT) module is also capable of resetting the device if it detects a malfunction in the system. If the WDT is not serviced correctly and/or in time, it first generates an NMI request to the CPU (this allows the CPU to gather debug information), and then resets the device after a predefined time-out period.

Another type of reset which needs to be detected in many applications is a reset while the device is in Deep Sleep mode (Wake-Up reset). This makes it possible to distinguish a wake-up reset from a power-on reset. For a power-on reset, the contents of the memories are undefined; but, the memory contents are well defined after a wake-up reset from deep sleep.

After a reset has been executed, the Reset Status (RST_SR) register provides information on the type of the last reset and the selected boot configuration.

The external reset pin, $\overline{\text{HDRST}}$, has a double-function. It serves as a reset input from the external world to reset the device, and it serves as a reset output to the external world to indicate that the device has executed a reset. For this purpose, pin $\overline{\text{HDRST}}$ is implemented as a bidirectional open-drain pin with an internal weak pull-up device.

The boot configuration information required by the device to perform the desired start operation after a power-up reset includes the start location for the code execution, and the activation of special modes. Some of the special modes include: enabling the on-chip debugging features or placing the pins of the chip into a high-impedance mode. This information is supplied to the chip via a number of dedicated input pins which are sampled and latched with a power-on reset. However, the software reset provides the special option to alter these parameters to allow a different start configuration after the software reset has finished.

5.2 Reset Registers

The two reset registers are shown in **Figure 5-1**. The long name, offset address, and location of detailed information are provided in **Table 5-1**.

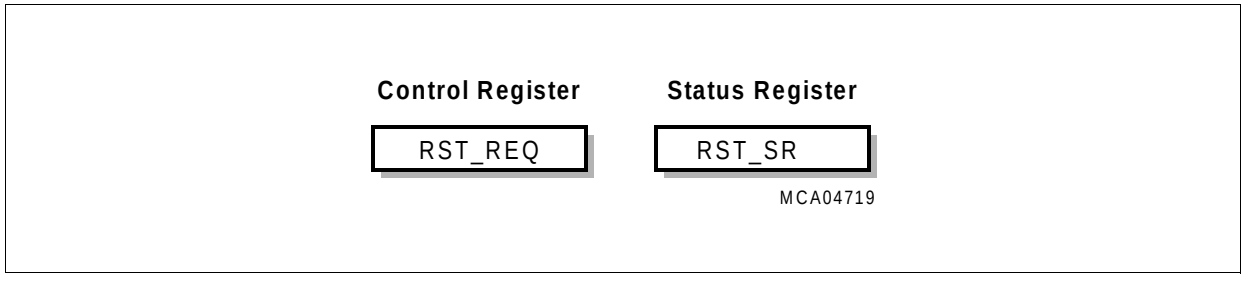


Figure 5-1 Reset Registers

Table 5-1 Reset Registers

Register Short Name	Register Long Name	Offset Address	Description see
RST_REQ	Reset Request Register	0010_H	**Page 5-5**
RST_SR	Reset Status Register	0014_H	**Page 5-3**

In the TC11IB, the reset registers are located in the address range of the SCU.

- Module Base Address. F000 0000_H
  Module End Address. F000 $00FF_H$
- Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 5-1**)

5.2.1 Reset Status Register (RST_SR)

After a reset, the Reset Status Register RST_SR indicates the type of reset occurred and indicates which parts of the TC11IB were affected by the reset. It also holds the state of the boot configuration pins that are latched at power-on reset. Register RST_SR is a read-only register.

RST_SR
Reset Status Register

Power-On Reset Value: 0000 1000 000U UUUU UUUU 0000 0000 0111_B
Hardware Reset Value: 0001 0000 000U UUUU 0000 0000 0010_B
Software Reset Value: 0010 0000 000U UUUU UUUU 0000 0000 $0UUU_B$
Watchdog Timer Reset Value: 0100 0000 000U UUUU UUUU 0000 0000 0101_B
Power-Down Wake-up Reset Value: 1000 0000 000U UUUU UUUU 0000 0000 0011_B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PWD RST	WDT RST	SFT RST	HD RST	PWO RST	0						HW BRK IN	HW OCD SE	HWCFG		
rh	rh	rh	rh	rh	r						rh	rh	rh		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HWC FG	CFGP0			0									RS EXT	X	RS STM
rh	rh			r									rh	rh	rh

Field	Bits	Type	Description
RSSTM	0	rh	**System Timer Reset Status** 0 System Timer was not reset 1 System Timer was reset
X	1	rh	**Reserved**; bit with no function; default after reset is 0.
RSEXT	2	rh	**$\overline{\text{HDRST}}$ Line State during Last Reset** 0 $\overline{\text{HDRST}}$ was not activated as output by TC11IB 1 $\overline{\text{HDRST}}$ was activated as output by TC11IB
CFGP0	[14:12]	rh	**Boot Software Configuration Bits Status** Status of general purpose IO pins P0.13 - P0.11 latched with the rising edge of $\overline{\text{PORST}}$. There is no automatic evaluation of this information by hardware. It is for software use only.
HWCFG	[18:15]	rh	**Boot Configuration Selection Status** Status of the configuration pins CFG[2:0], CFG[3] latched with power-on reset.
HWOCDSE	19	rh	**State of $\overline{\text{OCDSE}}$ Pin** Value of the OCDS enable pin latched at the end of power-on reset.

Field	Bits	Type	Description
HWBRKIN	20	rh	**State of $\overline{\text{BRKIN}}$ Pin** Value of the break input pin latched at the end of power-on reset.
PWORST	27	rh	**Power-On Reset Status Flag** 0 The last reset was not a power-on reset 1 The last reset was a power-on reset
HDRST	28	rh	**Hardware Reset Status Flag** 0 The last reset was not a hardware reset 1 The last reset was a hardware reset
SFTRST	29	rh	**Software Reset Status Flag** 0 The last reset was not a software reset 1 The last reset was a software reset
WDTRST	30	rh	**Watchdog Reset Status Flag** 0 The last reset was not a watchdog reset 1 The last reset was a watchdog reset
PWDRST	31	rh	**Power-Down/Wake-Up Reset Status Flag** 0 The last reset was not a wake-up from power-down reset 1 The last reset was a wake-up from power-down reset
0	[11:3], [26:21]	r	**Reserved**; returns 0 if read.

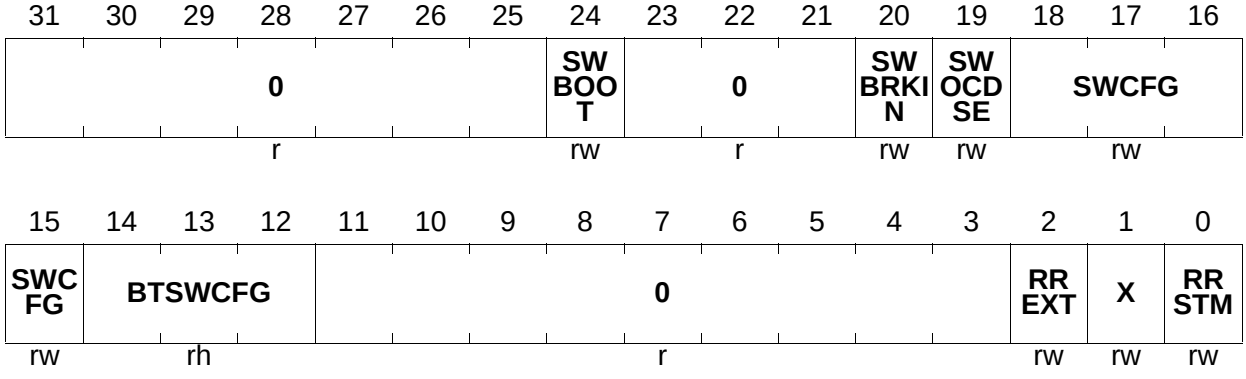
5.2.2 Reset Request Register (RST_REQ)

The Reset Request Register RST_REQ is used to generate a software reset. Unlike the other reset types, the software reset can exclude two functions from the reset. These are the System Timer and the external reset output $\overline{\text{HDRST}}$. In addition, it can change the boot configuration.

A software reset is invoked by any write to register RST_REQ. This register is EndInit-protected, meaning that bit WDT_CON0.ENDINIT must be set to 0 first through the password-protected access scheme for WDT_CON0. Once access is gained through the Endinit protection scheme, RST_REQ can be written, causing a software reset.

RST_REQ
Reset Request Register

Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
RRSTM	0	rw	**Reset Request for the System Timer** 0 Do not reset the System Timer 1 Reset the System Timer
X	1	rw	**Reserved**; bit with no function; writing to this bit stores the value which is written; default after reset is 0.
RREXT	2	rw	**Reset Request for External Devices** 0 Do not activate reset output $\overline{\text{HDRST}}$ and do not reset the PCI interface. 1 Activate reset output $\overline{\text{HDRST}}$ and reset the PCI interface.
BTSWCFG	[14:12]	rh	**Boot Software Configuration Value** Different boot options can be distinguished via the value here. If bit SWBOOT is set, the value here is used instead of the latched information from P0.13 - P0.11.
SWCFG	[18:15]	rw	**Software Boot Configuration** A software boot configuration different from the external applied hardware configuration can be specified with these bits. The configuration encoding is equal to the CFG[2:0], CFG[3]encoding.
SWOCDSE	19	rw	**Software OCDS Enable Signal Boot Value** Determines the desired value for the OCDS enable input signal to be used for software boot.
SWBRKIN	20	rw	**Software Break Signal Boot Value** Determines the desired value for the break input signal to be used for software boot.

Field	Bits	Type	Description
SWBOOT	24	rw	**Software Boot Configuration Selection** 0 Use the previously latched hardware configuration 1 Use the programmed software configuration
0	[11:3], [23:21] [31:25]	r	**Reserved**; read as 0; should be written with 0.

5.3 Reset Operations

A detailed description of each of the reset options is given in the following sections.

5.3.1 Power-On Reset

The $\overline{\text{PORST}}$ pin performs a power-on reset, also called cold reset. Driving the $\overline{\text{PORST}}$ pin low causes an asynchronous reset of the entire device. The device then enters its power-on reset sequence.

The external configuration input pins for the PLL are sampled in order to select the proper operating mode of the PLL. The PLL itself has its own power-on reset circuitry, and is not affected by any other reset condition other than a low signal transition on the $\overline{\text{PORST}}$ pin.

Simultaneously, the reset circuitry drives the $\overline{\text{HDRST}}$ pin low, and then waits for the following two conditions to occur:

1. The PLL is locked and the system clock is active
2. Pin $\overline{\text{PORST}}$ is negated (driven high)

When both of these conditions are met, the power-on reset sequence is terminated synchronously with the next system clock transition. The power-on reset indication flag PWORST in the Reset Status Register RST_SR is set, while all other reset cause indication flags are cleared. (Fields in this register that are set include the power-on reset indication flag (PWORST), as well as the reset status flags for the System Timer (RSSTM) and the reset output pin (RSEXT)). The rising edge of the signal at pin $\overline{\text{PORST}}$ causes the state of the configuration pins for the boot options to be latched into the appropriate registers.

5.3.2 External Hardware Reset

The external hardware reset pin $\overline{\text{HDRST}}$ serves as an external reset input as well as a reset output. It is an active-low, bidirectional open-drain pin with an internal weak pull-up. An active-low signal at this pin causes the chip to enter its hard-reset sequence synchronously with the next system clock transition. The $\overline{\text{HDRST}}$ pin is held low by the reset circuitry until its internal reset sequence is terminated.

When the sequence is terminated, the reset circuitry then releases $\overline{\text{HDRST}}$ (that is, it does not actively drive this pin anymore, so the weak pull-up can try to drive the pin high). It then begins monitoring the level of the pin. If the pin is still low (indicating that it is still being driven low externally), the reset circuitry holds the chip in hardware reset until a high level is detected on $\overline{\text{HDRST}}$. The hardware reset sequence is then terminated. The following flags in the Reset Status Register are then set: HDRST, RSSTM and RSEXT. Other reset cause indication flags are cleared.

Note: A hardware reset does not cause the configuration pins for the boot options to be latched. The configuration state that was latched at the end of the last power-on reset still controls these functions. Also, the PLL is not affected by an external hardware reset, but continues to operate according to its selected mode.

5.3.3 Software Reset

A software reset is invoked by writing the appropriate bits in the Reset Request Register (RST_REQ). Unlike the other forms of reset, the software reset can exclude two system functions from being reset. These are the System Timer and the external reset output $\overline{\text{HDRST}}$. Also, a software reset can change the boot configuration as a side-effect.

Excluding some system functions from a software reset offers these potential advantages:

- The System Timer can continue to clock accumulated elapsed time.
- The external components of a system can continue to operate while only the TC11IB is reset.

To perform a software reset, the Reset Request Register RST_REQ must be written. However, RST_REQ is EndInit-protected to avoid an unintentional software reset. The ENDINIT bit in the Watchdog Timer control register WDT_CON0 must be cleared via the password-protected access scheme. When this is done, a write access to RST_REQ can then be performed.

To exclude system functions from software reset, the appropriate bits in RST_REQ must be set to 0:

- Set RREXT to 0 to avoid activating the reset output $\overline{\text{HDRST}}$ and the PCI interface.
- Set RRSTM to 0 to avoid resetting the System Timer.

To change the boot configuration latched at the end of power-on reset, the software boot selection bit SWBOOT must be set, and the desired boot configuration must be written to bits SWBRKIN, SWOCDSE, and SWCFG. In addition, a configuration selection for the Boot Software can be placed in the bit field BTSWCFG.

When the software reset is terminated, bit RST_SR.SFTRST is set, indicating that the last reset was a software reset. All other reset cause indication flags are cleared. The reset status of the System Timer (RSSTM) and $\overline{\text{HDRST}}$ pin (RSEXT) are set according to the bits in RST_REQ at the time the software reset was initiated.

The PLL is not affected by a software reset; it continues to operate according to its previous mode.

Note: The boot configuration bits in the Reset Request Register RST_REQ are only used on software reset. In particular, the SWCFG bits that can be set to cause the TC11IB to boot using internal memory (if SWCFG is set to 0) are not effective on hardware boot. Regardless of the state of RST_REQ, any reset other than a software reset always uses the hardware configuration.

5.3.4 Watchdog Timer Reset

A Watchdog Timer overflow or access error occurs only in response to severe and/or unknown malfunctions of the TC11IB, caused by software or hardware errors. Therefore, the entire TC11IB is given a Watchdog Timer reset whenever the Watchdog Timer overflows.

Before the Watchdog Timer generates its reset, it first signals a non-maskable interrupt (NMI) and enters a time-out mode. The NMI invokes a Trap Service Routine (NMI is really a trap, not an interrupt). The trap handler can save critical state of the machine for subsequent examination of the cause of the Watchdog Timer failure. However, it is not possible to stop or terminate the Watchdog Timer's time-out mode or prevent the pending watchdog reset.

However, software can preempt the Watchdog Timer by issuing a software reset on its own. Because the cause of the system failure is presumably unknown at that time, and it is presumably uncertain which functions of the TC11IB are operating properly, it is recommended that the software reset be configured to reset all system functions including the System Timer and external reset output $\overline{\text{HDRST}}$, and to use the hardware boot configuration.

Eventually, if the NMI trap handler does not perform a software reset, or if the system is so compromised that the trap handler cannot be executed, the Watchdog Timer will cause a Watchdog Timer reset to occur at the end of its time-out mode period. The actions performed on a Watchdog Timer reset sequence are the same as are performed for an external hardware reset. At the end of the Watchdog Timer reset sequence, bits WDTRST, RSSTM and RSEXT are set in register RST_SR. All other reset cause indication flags are cleared.

Bit RST_REQ.RREXT can also be used to exclude some system functions from a Watchdog Timer reset. If bit RST_REQ.RREXT is cleared to 0, then the $\overline{\text{HDRST}}$ output and the reset to the PCI interface will not be asserted by Watchdog Timer reset.

5.3.4.1 Watchdog Timer Reset Lock

When the system emerges from any reset condition, the Watchdog Timer becomes active, and, — unless prevented by initialization software — will eventually time out. Ordinarily, initialization software will configure the Watchdog Timer and commence

servicing it on a regular basis to indicate that it is functioning appropriately. Should the system be malfunctioning so that initialization and service are not performed in a timely fashion, the Watchdog Timer will time out, causing a Watchdog Timer reset.

If the TC11IB system is so corrupted that it is chronically unable to service the Watchdog Timer, the danger could arise that the system would be continuously reset every time the Watchdog Timer times out. This could lead to serious system instability, and to the loss of information about the original cause of the failure. However, the reset circuitry of the TC11IB is designed to detect this condition. If a Watchdog Timer error occurs while one or both of the Watchdog Timer error flags (WDT_SR.WDTAE and WDT_SR.WDTOE) are already set to 1, the reset circuitry locks the TC11IB permanently in reset (Reset Lock) until the next power-on reset occurs by activation of the $\overline{\text{PORST}}$ pin.

This situation could arise, for example, if the connection to external code memory is lost or memory becomes corrupt, such that no valid code can be executed, including the initialization code. In this case, the initial time-out period of the Watchdog Timer cannot be properly terminated by software. The Watchdog Timer error flag WDTOE will be set when the Watchdog Timer overflows, and a Watchdog Timer reset will be triggered (after the watchdog reset pre-warning phase). The error flag WDTOE is not cleared by the Watchdog Timer reset which subsequently occurs. After finishing the Watchdog Timer reset sequence, the TC11IB will again attempt to execute the initialization code. If the code still cannot be executed because of connection problems, the WDTOE bit will not have been cleared by software. Again, the Watchdog Timer will time out and generate a Watchdog Timer reset. However, this time the reset circuitry detects that WDTOE is still set while a Watchdog Timer error has occurred, indicating danger of cyclic resets. The reset circuitry then puts the TC11IB in Reset Lock. This state can only be deactivated again through a power-on reset.

5.3.4.2 Deep-Sleep Wake-Up Reset

Power is still applied to the TC11IB during Deep Sleep power-management mode, which preserves the contents of the TC11IB's static RAM. If Deep Sleep mode is entered appropriately, all important system state information will have been preserved in static RAM by software. The only way to terminate Deep Sleep mode is for the TC11IB to be externally reset. However, while external reset will cause the TC11IB's registers to return to their default reset values, the contents of the static RAM is not affected. This can be important to the application software because initialization of the static RAM can be skipped, and data written to it before Deep Sleep mode was entered will still be valid.

If the TC11IB is in Deep Sleep mode, there are three options to awaken it:

1. A power-on reset $\overline{\text{PORST}}$
2. An external $\overline{\text{NMI}}$ event with a reset sequence
3. An external $\overline{\text{NMI}}$ event without a reset sequence

Selection between the two types of external $\overline{\text{NMI}}$ event is made via the control bit PMG_CON.DSRE. The advantage of using an external $\overline{\text{NMI}}$ event without a reset sequence is that the system can be more quickly awakened.

5.3.5 LMU eDRAM Reset

In TC11IB, the following three signals are provided to the LMU eDRAM for reset control.

Table 5-2 LMU Reset Control Signals

Signals	Description
eDRAM_PORST	This signal is activated by power-on reset ($\overline{\text{PORST}}$) or hardware reset ($\overline{\text{HDRST}}$).
ASYNC_RST	**Asynchronous Reset** This signal is activated by power-on reset ($\overline{\text{PORST}}$) or hardware reset ($\overline{\text{HDRST}}$)
SYNC_RST	**Synchronous Reset** This signal is activated by watchdog timer reset or software reset.

The interface between the SCU and the LMU is shown in **Figure 5-2**.

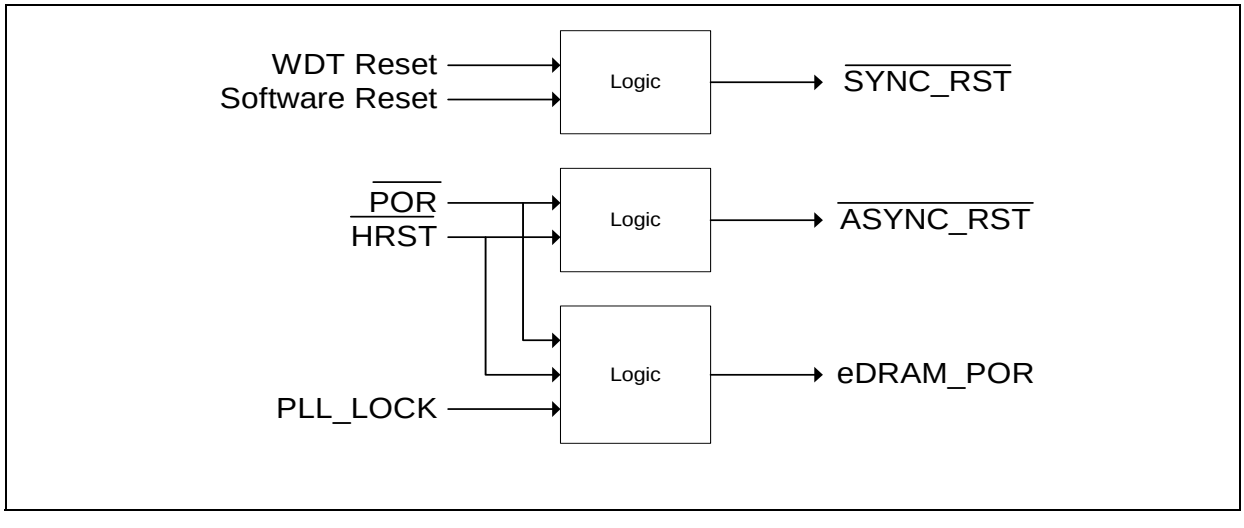


Figure 5-2 SCU to LMU Interface

The timing for eDRAM power-on reset cycle is shown as **Figure 5-3**

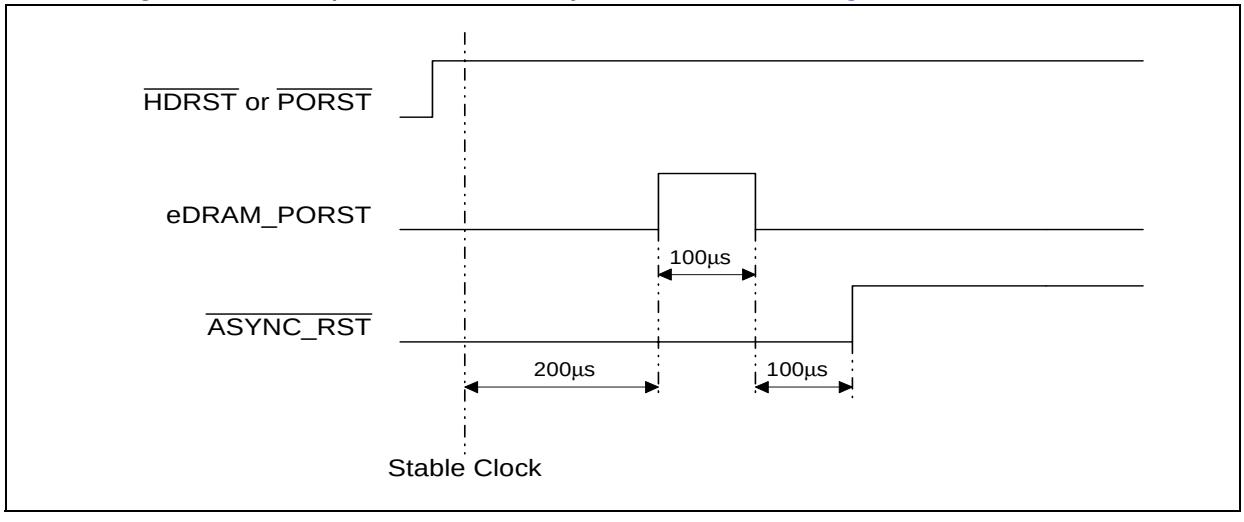


Figure 5-3 Timing for eDRAM Power-On Reset Cycle

When either PORST or HDRST is activated, the output signals eDRAM_PORST and ASYNC_RST are both forced to low. A counter is also enabled for timing the signal generation. After 200µs, eDRAM_POR is asserted high and maintained for 100µs. A further 100µs later, ASYNC_RST is de-asserted.

Since PORST is an asynchronous input, counting can only proceed after the PLL clock has locked. On the other hand, a stable PLL clock is already available when HDRST is asserted. Thus counting can begin after the trailing edge of HDRST.

To reduce the size of counter, the input clock, slow FPI clock of 48MHz, can be divided down (to say 1MHz, for example).

A detailed description of the behavior of the eDRAM module on the different reset conditions see at LMU module description.

5.3.6 State of the TC11IB After Reset

Table 5-3 lists the modules/functions and types of reset and indicates whether and how the various functions of the TC11IB are affected. A ■ indicates that a function is reset to its default state.

Table 5-3 Effect of Reset Types on TC11IB Modules/Functions

Module / Function	Wake-up Reset	Watchdog Reset	Software Reset	Hardware Reset	Power-On Reset
CPU Core	■	■	■	■	■
Peripherals (except System Timer)	■	■	■	■	■

Table 5-3 Effect of Reset Types on TC11IB Modules/Functions (cont'd)

Module / Function	Wake-up Reset	Watchdog Reset	Software Reset	Hardware Reset	Power-On Reset
On-Chip Static RAM (code or data)	Not affected	Not affected; contents may be unreliable	Not affected	Not affected	Not affected; contents are invalid
On-Chip Cache (see note 1)	■	■	■	■	■
LMU eDRAM	content undefined	Content preserved (see note 2)	Content preserved (see note 2)	Content undefined	content undefined
Com eDRAM	content undefined	content undefined	content undefined	content undefined	content undefined
PCI Interface (see note 3)	■	Optional	Optional	■	■
System Timer (see note 4)	■	Not affected	Optional	Not affected	■
Debug Unit	■	■	Optional	■	■
Oscillator / PLL	■	Not affected	Not affected	Not affected	■
External Bus Control Unit	■	■	■	■	■
External Bus Pins	Depending on Reset Config.	Depending on Reset Config.	Depending on Reset Config.	Depending on Reset Config.	Depending on Reset Config.
Port Pins	Tri-stated	Tri-stated	Tri-stated	Tri-stated	Tri-stated
Reset output pin $\overline{\text{HDRST}}$	■	Optional	Optional	■	■
Boot Configuration taken from	Latched hardware configur-ation	Latched hardware configur-ation	Optional latched hardware or software configur-ation	Latched hardware configur-ation	External pins

Note: 1) The actual data contents of the cache are not affected through a reset; however the cache tag information is cleared, resulting in an 'empty' cache.

2) The data integrity of a location being written on software reset or watchdog timer reset activation can not be guaranteed. Spurious data might be written to the location where the aborted write pointed to.
3) All tristatable PCI outputs are set to 'Tristate' when reset is active. Please refer to Chapter "PCI Interface" for more details.
4) The counting of the System Timer is not affected by Watchdog Timer Reset and Hardware Reset.

5.4 Booting Scheme

When the TC11IB is reset, it needs to know the type of configuration required to start in after the reset sequence is finished. Internal state is usually cleared through a reset. This is especially true in the case of a power-up reset. Thus, boot configuration information needs to be applied by the external world through input pins.

Boot configuration information is required:

- for the start location of the code execution,
- and activation of special modes and conditions

For the start of code execution and activation of special mode, the TC11IB implements two basic booting schemes: a hardware scheme which is invoked through external pins, and a software scheme in which software can determine the boot options, overriding the externally-applied options.

5.4.1 Hardware Booting Scheme

The hardware booting scheme uses the state of a number of external input pins — sampled and latched with a power-on reset — to determine the start configuration of the chip. The state of these pins is latched into the Reset Status Register RST_SR when the power-on reset signal (pin $\overline{\text{PORST}}$) is released. This hardware configuration determined through the bits HWOCDSE, HWBRKIN, and HWCFG[3:0] is used for all hardware-invoked reset options (power-on, hard, watchdog and wake-up reset). In order to have several boot Software options available, pins P0.11 - P0.13 are latched with pin $\overline{\text{PORST}}$ activation into CFGP0.[13-11] bit field of the reset status register RST_SR.

5.4.2 Software Booting Scheme

The Reset Request Register RST_REQ, used for generating a software reset, contains seven bits that have the same meaning as the corresponding seven bits in the RST_SR register. On a software reset, software can choose to set a different boot configuration from the one latched with power-on reset. This option is selected through bit SWBOOT in register RST_REQ. When writing to this register, the desired values for bits SWOCDSE, SWBRKIN, and SWCFG[3:0] are written along with bit SWBOOT set to 1. In addition, bit field BTSWCFG can be used to provide further option selection information to the boot Software. This causes the device to start in the configuration

selected through the software boot configuration bits in register RST_REQ instead of starting with the hardware boot configuration stored in register RST_SR.

5.4.3 Boot Options

The architecture of the TriCore booting schemes provides a number of different boot options for the start of code execution. Several of them have sub-options even. Especially option 2 supports a wide variety of sub-options by the boot ROM Software evaluating the latched configuration bits and Port 0 general Purpose IO bits latched during reset at CFGP0[13:11].

Table 5-4 shows the boot options available in the TC11IB. Note that the signals $\overline{\text{OCDSE}}$, $\overline{\text{BRKIN}}$, and CFG[3:0] can be either the corresponding bits HWOCDSE, HWBRKIN, and HWCFG[3:0] in register RST_SR, or the software configuration bits SWOCDSE, SWBRKIN and SWCFG[3:0] in register RST_REQ.

Table 5-4 Boot Selections

$\overline{\text{OCDSE}}$	$\overline{\text{BRKIN}}$	CFG [3]	CFG [2:0]	Type of Boot	Boot Source	Initial PC Value
1	1	X	000_B	Start directly in core scratchpad memory	SRAM (Only via SW Reset)	D400 0000_H
			Not (000 or 100)	Start from Boot ROM	Boot ROM, SSC BSL mode[1)] (BootStrap Loader) or ASC BSL mode[1)]	DFFF $FFFC_H$
		0	100_B	External memory as slave directly via EBU	External Memory (non-cached, CS0)	A000 0000_H
		1	100_B	External memory as master directly via EBU		
1	0	don't care		Tri-state chip (deep sleep)	–	–

Table 5-4 Boot Selections (cont'd)

OCDSE	BRKIN	CFG [3]	CFG [2:0]	Type of Boot	Boot Source	Initial PC Value
0	1	0	100_B	Go to halt with EBU enabled as slave	–	–
		1		Go to halt with EBU enabled as master		
		all other combina-tions		Go to halt with EBU disabled		
0	0	don't care		Go to external emulator space	–	DE00 0000_H

1) SSC/ASC BootStrap Loader is built in BOOT ROM which provides a mechanism to load the startup program, which is executed after reset, via the SSC/ASC interface. After successfully loaded, the startup program will be executed from the address at 0xC000 0004H.

5.4.4 Boot Configuration Handling

- The inputs CFG[3:0] are latched internally with the rising edge of $\overline{\text{PORST}}$ to guarantee a stable value during normal operation (during $\overline{\text{PORST}}$ active the latches are transparent). The latched values can only be changed by another power-on reset.
- The CFG[3:0] pins determine the hardware boot configuration after power-on reset / hardware reset. This configuration can be changed by software in conjunction with a software reset (software boot configuration).
- The boot software must read the actual software configuration (register RST_REQ) to determine how to proceed (for example: entering boot-strap loader). It is also possible to read the latched value of the configuration pins.

5.4.5 Normal Boot Options

The normal boot options are invoked when both, $\overline{\text{OCDSE}}$ and $\overline{\text{BRKIN}}$ are set to 1.

TC11IB has three options for booting: External, Internal Boot ROM and Internal SRAM (Scratchpad). The SRAM Boot Option can be selected only through a software reset when first valid code has been written to the on-chip code memory.

In order to access external memory, the External Bus Unit (EBU) must have information about the type and access mechanism of the external boot code memory. This information is not available through the boot configuration pins. Special actions must be taken first by the EBU in order to determine the configuration settings.

The EBU initiates a special external bus access in order to retrieve information about the external code memory. This access is performed to address A000 0004_H such that

regardless of the type and characteristics of the external memory, configuration information can be read from the memory into the EBU. By examining this information, the EBU determines the exact requirements for accesses to the external memory. It then configures the control registers accordingly, and performs the first instruction fetch from address A000 0000_H.

5.4.6 Debug Boot Options

Debug boot options are selected if the states of the bits $\overline{\text{OCDSE}}$ and $\overline{\text{BRKIN}}$ are not both activated.

Two of the options enable emulators to take control over the TC11IB. If only $\overline{\text{OCDSE}}$ is activated ($\overline{\text{OCDSE}}$ = 0), the TC11IB goes into the HALT state. External hardware emulators can then configure the TC11IB via the JTAG interface. If $\overline{\text{BRKIN}}$ is activated ($\overline{\text{BRKIN}}$ = 0), the TC11IB starts execution out of a special external memory region reserved for debugging.

After configuring the TC11IB via either of these boot options, the regular application configuration can be invoked by executing a software reset with a software boot option. By setting the software configuration bits in register RST_REQ such that the debug boot options are deactivated, a normal boot of the TC11IB is accomplished after the software reset terminates.

Note: The state of the external $\overline{\text{OCDSE}}$ pin is also latched by other circuitry in the TC11IB, enabling special debugging features if a low signal level is latched at this pin when the power-on reset ($\overline{\text{PORST}}$) signal is raised. A software boot with a normal boot configuration (that is, bit SWOCDSE = 1) does not affect this operation.

The third debug boot option places the TC11IB into a tri-state mode. All pins are deactivated, including the oscillator, and internal circuitry is held in a low-power mode. This mode can be used to connect emulator probes to a TC11IB soldered onto a board to perform testing.

6 Power Management

This chapter describes the power management system for the TC11IB. Topics include the internal system interfaces, external interfaces, state diagrams, and the operations of the CPU and peripherals. The Power Management State Machine (PMSM) is also described.

6.1 Power Management Overview

The TC11IB power management system allows software to configure the various processing units so that they automatically adjust to draw the minimum necessary power for the application.

As shown in **Table 6-1**, there are four power management modes:

- Run Mode
- Idle Mode
- Sleep Mode
- Deep Sleep Mode

Table 6-1 Power Management Mode Summary

Mode	Description
Run	The system is fully operational. All clocks and peripherals are enabled, as determined by software.
Idle	The CPU clock is disabled, waiting for a condition to return it to Run Mode. Idle Mode can be entered by software when the processor has no active tasks to perform. All peripherals remain powered and clocked. Processor memory is accessible to peripherals. A reset, Watchdog Timer event, a falling edge on the $\overline{\text{NMI}}$ pin, or any enabled interrupt event will return the system to Run Mode.
Sleep	The system clock continues to be distributed only to those peripherals programmed to operate in Sleep Mode. Interrupts from operating peripherals, the Watchdog Timer, a falling edge on the $\overline{\text{NMI}}$ pin, or a reset event will return the system to Run Mode. Entering this state requires an orderly shut-down controlled by the Power Management State Machine.
Deep Sleep	The system clock is shut off; only an external signal will restart the system. Entering this state requires an orderly shut-down controlled by the Power Management State Machine (PMSM).

The operation of each system component in each of these states can be configured by software. The power management modes provide flexible reduction of power consumption through a combination of techniques, including:

– Stopping the CPU clock
– Stopping the clocks of other system components individually
– Clock-speed reduction of some peripheral components individually
– Power-down of the entire system with fast restart capability

The Power Management State Machine (PMSM) controls the power management mode of all system components during Run Mode, Idle Mode, and Sleep Mode. The PMSM continues to operate in Idle Mode and Sleep Mode, even if all other system components have been disabled, so that it can re-awaken the system as needed. In Deep Sleep Mode, even the PMSM is disabled and the system must be re-awakened from an external source. This flexibility in power management provides minimum power consumption for any application.

The Power Management State Machine is implemented in the System Control Unit (SCU) module of the TC11IB. Thus, it is accessible through the FPI Bus interface by any FPI Bus master.

As well as these explicit software-controlled power-saving modes, special attention has been paid in the TC11IB to provide automatic power-saving in those operating units that are currently not required or idle. To save power, these are shut off automatically until their operation is required again.

In typical operation, Idle Mode and Sleep Mode will be entered and exited frequently during the runtime of an application. For example, system software will typically cause the CPU to enter Idle Mode each time it must wait for an interrupt before continuing its tasks. In Sleep Mode and Idle Mode, wake-up is performed automatically when any enabled interrupt signal is detected or if the Watchdog Timer signals the CPU with an NMI trap.

No clock is running in a system in Deep Sleep Mode, so it cannot be awakened by an interrupt or the Watchdog Timer. It will be awakened only when it receives an external non-maskable interrupt (NMI) or reset signal, as described **Section 6.3.3**. Software must prepare the external environment of the TC11IB to cause one of these signals under the appropriate conditions before entering Deep Sleep Mode. If Deep Sleep Mode were entered unintentionally without an event of this nature first being prepared, the TC11IB might never emerge from Deep Sleep Mode. For this reason, the register used to set up Deep Sleep Mode can be changed only by way of a password-protected access mechanism (see **Section 6.3.3**).

6.2 Power Management Control Registers

The set of registers used for power management is divided between central TC11IB components and peripheral components. The PMG_CSR and the PMG_CON registers provide software control and status information for the Power Management State Machine (PMSM). There are individual clock control registers for peripheral components because the Sleep Mode behavior of each peripheral component is programmable. When entering Idle Mode and Sleep Mode, the PMSM directly controls TC11IB

components such as the CPU, but indirectly controls peripheral components through their clock control registers.

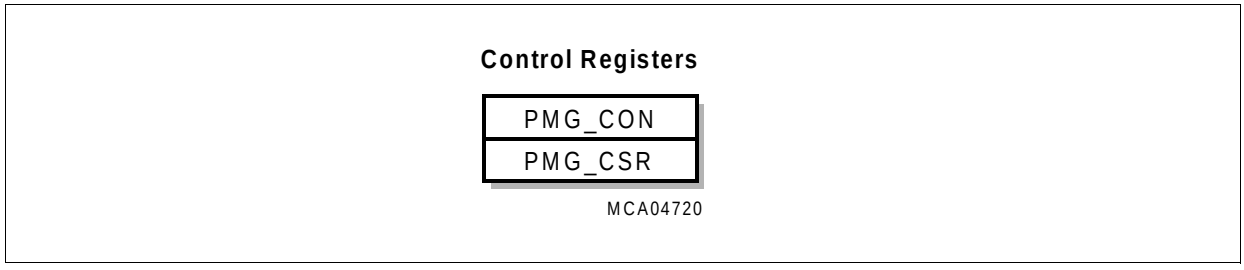


Figure 6-1 Power Management Registers

Table 6-2 Power Management Registers

Register Short Name	Register Long Name	Offset Address	Description see
PMG_CON	Power Management Control Register	0030_H	Page 6-4
PMG_CSR	Power Management Control and Status Register	0034_H	Page 6-5

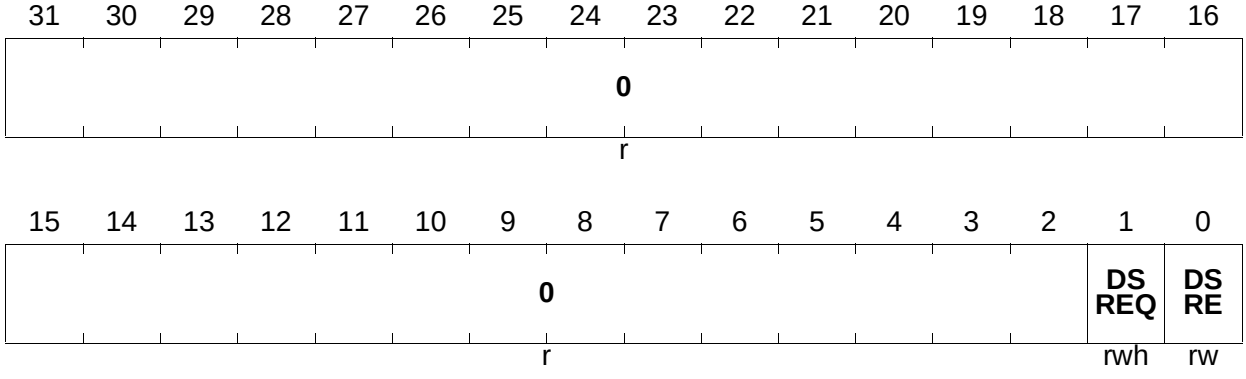
In the TC11IB, the reset registers are located in the address range of the SCU:

- Module Base Address. F000 0000_H
  Module End Address. F000 $00FF_H$
- Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 6-2**)

6.2.1 Power Management Control Register PMG_CON

The Power Management Control Register PMG_CON is used to request Deep Sleep Mode. This register is specially protected to avoid unintentional invocation of Deep Sleep Mode.

PMG_CON
Power Management Control Register **Reset Value: 0000 0001$_H$**



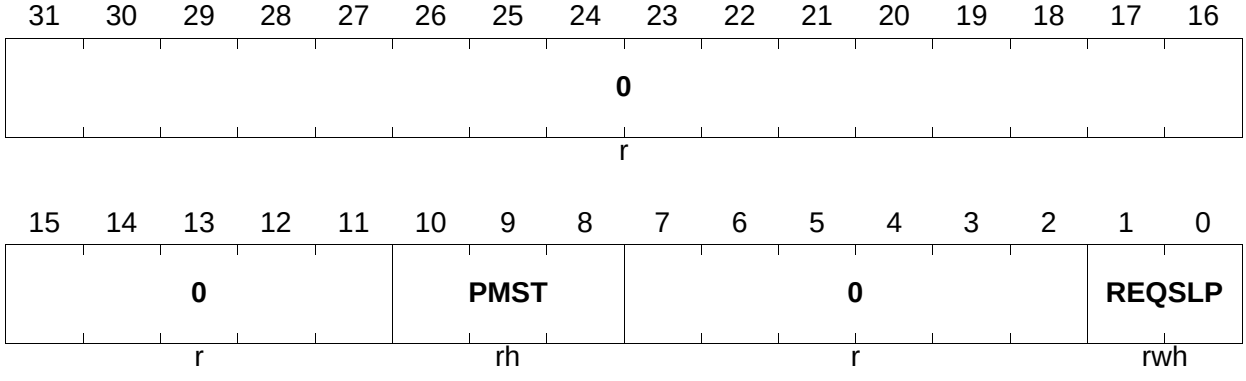
Field	Bit	Type	Function
DSRE	0	rw	**Reset On Wake-Up From Deep Sleep** Wake-up from deep sleep can be caused by either a power-on reset or through a low level at the $\overline{\text{NMI}}$ pin. The state of DSRE determines whether a full internal hardware reset should be performed on exit from deep sleep. 0 No internal reset will be performed on exit from deep sleep 1 An internal hardware reset will be performed on exit from deep sleep
DSREQ	1	rwh	**Deep Sleep Request Bit** 0 Normal Mode 1 Deep Sleep Mode requested Bit is reset by hardware on wake-up from deep sleep mode.
0	[31:2]	r	**Reserved**; read as 0; should be written with 0.

Note: The PMG_CON register is specially protected to avoid unintentional invocation of Deep Sleep Mode. In order to write to PMG_CON.DSREQ, the WDT_CON0.ENDINIT bit must be set to 0 through a password-protected access mechanism. WDT_CON0.ENDINIT must then be set to 1 to make the changed value of DSREQ become effective.

6.2.2 Power Management Control and Status Register PMG_CSR

The Power Management Control and Status Register PMG_CSR stores Idle Mode and Sleep Mode request bits. It also shows the status of the Power Management State Machine. Its fields are described below.

PMG_CSR
Power Management Control and Status Register **Reset Value: 0000 0100$_H$**



Field	Bit	Type	Function
REQSLP	[1:0]	rwh	**Idle Mode and Sleep Mode Request Bits** 00 Normal Run Mode 01 Request Idle Mode 10 Request Sleep Mode 11 Reserved; do not use this combination; In Idle Mode, Sleep Mode, or Deep Sleep Mode, these bits are cleared in response to an enabled interrupt, a wake-up from Deep Sleep Mode via the $\overline{\text{NMI}}$ pin or $\overline{\text{PORST}}$ pin, or when bit 15 of the Watchdog Timer count register (the WDT_SR.TIM[15] bit) changes from 0 to 1.
PMST	[10:8]	rh	**Power Management State Machine Status** 000 Waiting for PLL Lock condition 001 Normal Run Mode 010 Idle Mode requested 011 Idle Mode acknowledged 100 Sleep Mode 101 Deep Sleep Mode 110 Undefined, reserved 111 Undefined, reserved
0	[7:2], [31:11]	r	**Reserved**; read as 0; should be written with 0.

6.3 Power Management Modes

This section describes power management modes, their operations, and how power management modes are entered and exited. It also describes the behavior of TC11IB system components in all power management modes.

6.3.1 Idle Mode

Software requests the Idle Mode by setting the PMG_CSR.REQSLP bit field to 01_B.

The Power Management State Machine (PMSM) posts an idle request signal to the CPU. The CPU finishes its current operation, sends an acknowledge signal back to the PMSM, and then enters an inactive state in which the CPU clocks and the DMU and PMU memory units are shut off. Other system components also can request the TC11IB to enter idle mode. For example, FPI-Bus agents, the PCP, can request idle Mode by writing to the PMG_CSR register.

In Idle Mode, memory accesses to the DMU and PMU via the FPI Bus cause these units to awaken automatically to handle the transactions. When memory transactions are complete, the DMU and PMU return to Idle Mode again.

The system will be returned to Run Mode through occurrence of any of the following conditions:

- An interrupt signal is received from an enabled interrupt source.
- An NMI request is received either from an external source via the $\overline{\text{NMI}}$ pin or from the Watchdog Timer. The Watchdog Timer triggers an NMI trap request in Idle mode when its count value (WDT_SR.TIM) transitions from $7FFF_H$ to 8000_H.
- An external power-on signal $\overline{\text{PORST}}$ or hardware reset signal $\overline{\text{HDRST}}$ is received.
- A software reset is requested by another FPI Bus agent (such as the PCP) by writing to the reset request register RST_REQ.

If any of these conditions arise, the TC11IB immediately awakens and returns to Run Mode. If it is awakened by a hardware or software reset signal, the TC11IB system begins its reset sequence. If it is awakened by a Watchdog Timer overflow event, it executes the instruction following the one which was last executed before Idle Mode was entered. If it is awakened by an NMI signal or interrupt signal, the CPU will immediately vector to the appropriate handler.

6.3.2 Sleep Mode

Software can request the Sleep Mode by setting PMG_CSR.REQSLP = 10_B.

6.3.2.1 Entering Sleep Mode

Sleep Mode is entered in two steps. In the first step, the CPU is put into Idle Mode in the same manner as described in **Section 6.3.1**. When the PMSM receives the Idle acknowledge signal back from the CPU, it goes on to the second step.

In the second step, a sleep signal is then broadcast on the FPI Bus. Each FPI Bus interface unit receives this signal. The response of each FPI Bus unit to the sleep signal is determined by its own clock control register (CLC). These registers must have been previously configured by software.

6.3.2.2 TC11IB State During Sleep Mode

Sleep Mode is disabled for a unit if its CLC_EDIS bit field is 1. The sleep signal is ignored by this unit and it continues normal operation.

If the unit's clock control register bit CLC_EDIS is 0, Sleep Mode is enabled for this unit. In this case, the sleep signal will cause this unit to enter Sleep Mode. Two actions then occur:

1. The unit's bus interface finishes whatever transaction was in progress when the signal was received.
2. The unit's functions are suspended.

The TriCore architecture qualifies the actions in step 2 as follows. Depending on the module's Fast Shut-Off Enable bit CLC.FSOE in the clock control registers, the module's clocks are either immediately stopped (CLC.FSOE = 1), or the unit is allowed time to finish ongoing operations (CLC.FSOE = 0) before the clocks are stopped. For example, setting CLC.FSOE to 1 for a serial port will stop all actions in the serial port immediately when the sleep signal is received. Ongoing transmissions or receptions will be aborted. If CLC.FSOE is 0, ongoing transmissions or receptions will be completed, and then the clock will be shut off. The purpose of setting CLC.FSOE = 1 is to allow a debugger to observe the internal state of a peripheral unit immediately.

Please refer to the respective peripheral unit chapters for discussions of the exact implementation of Sleep Mode (Clock Control Register) for a specific peripheral unit.

6.3.2.3 Exiting Sleep Mode

The system will be returned to Run Mode by the same events that exit Idle Mode, as described in **Section 6.3.1**. The response of the CPU to being awakened is also the same as for Idle Mode. Peripheral units which have entered Sleep Mode will switch back to their selected Run Mode operation.

6.3.3 Deep Sleep Mode

In Deep Sleep Mode, the PMSM shuts off all clocks, the PLL, and the oscillator. Therefore, Deep Sleep Mode consumes the least power of all TC11IB states.

Deep Sleep Mode is requested through software by setting the PMG_CON.DSREQ bit to 1. The request bits for Deep Sleep Mode have been separated intentionally from the Idle Mode and Sleep Mode request bits to minimize the chance of inadvertently invoking Deep Sleep Mode.

Because no clock is running in a system in Deep Sleep Mode, it can not be awakened by any interrupt source, including the Watchdog Timer. It can only be awakened when it receives an external reset or $\overline{\text{NMI}}$ signal, as described in this section. Software must prepare the external environment of the TC11IB to cause one of these signals under the appropriate conditions before entering Deep Sleep Mode. If Deep Sleep Mode were entered unintentionally without an event of this nature first being prepared, the TC11IB might never emerge from Deep Sleep Mode. For this reason, the PMG_CON register which sets up Deep Sleep Mode is specially protected. In order to write to PMG_CON, the WDT_CON0.ENDINIT bit must be set to 0 through a password-protected access mechanism to register WDT_CON. In order for the request to be activated, WDT_CON0.ENDINIT must first be set to 1 after the write to PMG_CON.

6.3.3.1 Entering Deep Sleep Mode

Deep Sleep Mode is entered in three steps. In the first step, the CPU is put into Idle Mode in the same way as described in **Section 6.3.1**. When the PMSM receives the Idle acknowledge signal back from the CPU, it goes on to the second step in which the PMSM activates the sleep signal, as described in **Section 6.3.2**. In the third step, the PMSM shuts off all clocks, the PLL, and the oscillator.

Note: The Power-On Reset Pin $\overline{\text{PORST}}$ should be kept stable when powering the TC11IB down.

Note: The software which turns on deep sleep mode must reside in the internal code scratch pad RAM to ensure that no external code accesses via the EBU are running when the PLL clock is shut down.

6.3.3.2 TC11IB State During Deep Sleep Mode

In Deep Sleep Mode, all port pins hold their state when Deep Sleep Mode is entered. The Deep Sleep Reset Enable Bit PMG_CON.DSRE controls whether the TC11IB is reset when Deep Sleep Mode is left.

- PMG_CON.DSRE = 0: TC11IB is not reset when Deep Sleep Mode is left.
- PMG_CON.DSRE = 1: TC11IB is reset when Deep Sleep Mode is left. Port pins are put into the reset state.

6.3.3.3 Exiting Deep Sleep Mode

Deep Sleep Mode can be exited in two ways:

- A power-on reset signal is detected ($\overline{\text{PORST}}$)
- The $\overline{\text{NMI}}$ pin detects a falling edge

When returning to full-power operation, the first step is to restart the oscillator and PLL, and re-enable the system clocks. This generally requires external hardware to wait until

the PLL has had time to lock to its external clock source before the system can return to reliable operation.

Exactly how the TC11IB system returns from Deep Sleep Mode depends upon which signal re-awakens it. If awakened by a falling edge on the $\overline{\text{NMI}}$ pin, it further depends upon the state of the PMG_CON.DSRE bit.

6.3.3.4 Exiting Deep Sleep Mode With A Power-On Reset Signal

When awakened through a power-on reset signal ($\overline{\text{PORST}}$), the system initiates the same reset sequence as is used when power is first applied. The TC11IB automatically initiates its clock-acquisition sequence. This provides the time needed for the PLL to lock to the oscillator. The TC11IB will remain in the reset state until both the PLL is locked and the $\overline{\text{PORST}}$ signal is deactivated.

6.3.3.5 Exiting Deep Sleep Mode With an $\overline{\text{NMI}}$ Signal

The state of the Deep Sleep Reset Enable Bit, PMG_CON.DSRE, determines what happens when the TC11IB is awakened through a falling edge on the $\overline{\text{NMI}}$ pin.

If DSRE was set to 1 before entering Deep Sleep Mode, the TC11IB will execute a reset sequence similar to the power-on reset sequence. Therefore, all port pins are put into their reset state and stay in this state until they are affected in some way by program execution.

If DSRE was set to 0 before entering Deep Sleep Mode, a fast wake-up sequence is used. In this case, the TC11IB does not wait for the PLL to stabilize and lock to the external clock. Instead, it resumes operation as soon as the PLL provides clock signals. The port pins continue to hold their state which was valid during Deep Sleep Mode until they are affected by program execution.

Special attention must be paid when using this type of wake-up. As soon as the device is woken up from Deep Sleep mode, the PLL begins generating clocks starting with the PLL's base frequency. When the external oscillator begins to generate clock signals, the PLL will begin to increase its frequency in order to achieve the programmed frequency ($f_{OSC} \times N$). Note that the start-up time of an external crystal oscillator can be in the range of some ms. This will continue until the PLL is locked to the external clock. Thus, since the TC11IB does not wait until the PLL has locked, its operation is based on a clock which will increase in frequency until the PLL is locked to the programmed frequency. Software can poll the PLL Lock Status bit (PLL_CLC.LOCK) for the lock status of the PLL.

Note: For wake-up through NMI, the NMI signal must held active until the clock system starts. Otherwise, the TC11IB will not enter the NMI trap handler routine.

6.3.4 Summary of TC11IB Power Management States

Table 6-3 summarizes the state of the various units of the TC11IB during Run Mode, Idle Mode, Sleep Mode, and Deep Sleep Mode.

Table 6-3 State of TC11IB Units During Power Management Modes

Unit	Run Mode	Idle Mode	Sleep Mode	Deep Sleep Mode
Oscillator & PLL	On	On	On	Off
CPU	Executing	Idle	Idle	Off (no clock)
DMU & PMU	Active	Idle, but accessible	Idle, but accessible	Off (no clock). Memory units hold their contents
Watchdog Timer	Functioning as programmed	Functioning as programmed	Functioning as programmed	Off (no clock)
Slow FPI Bus Peripherals	Functioning as programmed	Functioning as programmed	Functioning as programmed	Off (no clock)
Debug Unit	Functioning	Functioning	Functioning	Off (no clock)
LMU (Code/ Data DRAM) & ComDRAM	Refresh running	Refresh running	Refresh running	No Refresh
External Bus Controller (EBU)	Functioning as programmed	Functioning as programmed	Functioning as programmed	Off (no clock); The EBU pins hold the last value.
Ports	Functioning as programmed	Functioning as programmed	Functioning as programmed	Off (no clock). The port pins hold the last value.

Table 6-3 State of TC11IB Units During Power Management Modes

Unit	Run Mode	Idle Mode	Sleep Mode	Deep Sleep Mode
Ethernet Controller	Functioning as programmed	Functioning as programmed	Functioning as programmed	Input clocks are disabled. MDIO hold the last value if it is output. Or disabled if it is input and PMG_CON.DSRE is set to ‘1’, otherwise it remains enabled.
PCI	Functioning as programmed	Functioning as programmed	Functioning as programmed	Output pins hold the last value. Input pins are disabled if PMG_CON.DSRE is set to ‘1’, otherwise input pins remain enabled.

7 Memory Map of On-Chip Local Memories

The memory system of the TC11IB provides the following memories:

- Program Memory Unit (PMU) with
  - 24 KBytes Code Scratch-Pad RAM (SRAM)
  - 8 KBytes Instruction Cache (ICache)
- Data Memory Unit (DMU) with
  - 24 KBytes Data Scratch-Pad RAM (SRAM)
  - 8 KBytes Data Cache (DCache)
- 16 KBytes Boot ROM (BROM)
- eDRAM Local Memory Unit (LMU) with
  - 512 KBytes Code/Data Memory
- ComDRAM with
  - 1MBytes Code/Data Memory
- Peripheral Control Processor (PCP) with
  - 16 KBytes Data Memory (PCODE)
  - 4 KBytes Parameter RAM (PRAM)

This chapter gives an overview on the TC11IB memory map. Details on the specific features of the memories in the PMU, DMU, LMU and PCP modules are described in the specific **Chapter 8**, **Chapter 9**, **Chapter 11** and **Chapter 17** in this User's Manual.

7.1 TC11IB Address Map

Table 7-1 defines the specific segment oriented address blocks of the TC11IB with its corresponding address range, size, and PMU/DMU access view.

Table 7-1 TC11IB Block Address Map

Segment	Address Range	Size	Description	DMU Acc.	PMU Acc.	
0 – 7	0000 0000$_H$ – 7FFF FFFF$_H$	2 GB	MMU/ FPI Space	via F_FPI	via F_ FPI	cached
8	8000 0000$_H$ – 8FFF FFFF$_H$	256 MB	External Memory Space mapped from Segment 10	via LMB	via LMB	
9	9000 0000$_H$ – 9FDF FFFF$_H$	254 MB	PCI Space mapped from Segment 11	via F_FPI	via F_FPI	
	9FE0 0000$_H$ – 9FEF FFFF$_H$	1 MB	Com-DRAM Space mapped from Segment 11			
	9FF0 0000$_H$ – 9FFF FFFF$_H$	1 MB	Reserved	–	–	
10	A000 0000$_H$ – AFBF FFFF$_H$	252 MB	External Memory Space	via LMB	via LMB	non-cached
	AFC0 0000$_H$ – AFC7 FFFF$_H$	512 KB	LMU Space	via LMB	via LMB	
	AFC8 0000$_H$ – AFFF FFFF$_H$	3.5 MB	Reserved	–	–	
11	B000 0000$_H$ – BFDF FFFF$_H$	254 MB	PCI Space mappable into segment 9	via F_FPI	via F_FPI	
	BFE0 0000$_H$ – BFEF FFFF$_H$	1 MB	ComDRAM Space			
	BFF0 0000$_H$ – BFFF FFFF$_H$	1 MB	Reserved	–	–	
12	C000 0000$_H$ – C007 FFFF$_H$	512 KB	Local Memory Unit eDRAM Space	via LMB	via LMB	cached
	C008 0000$_H$ – CFFF FFFF$_H$	255.5 MB	Reserved	–	–	

Table 7-1 TC11IB Block Address Map (cont'd)

Seg-ment	Address Range	Size	Description	DMU Acc.	PMU Acc.	
13	D000 0000$_H$ – D000 5FFF$_H$	24 KB	Local Data Scratchpad Memory (SRAM)	DMU local	via LMB	non-cached
	D000 6000$_H$ – D3FF FFFF$_H$	~ 64 MB	Reserved	–	–	
	D400 0000$_H$ – D400 5FFF$_H$	24 KB	Local Code Scratchpad Memory (SRAM)	via LMB	PMU local	
	D400 6000$_H$ – D7FF FFFF$_H$	~64 MB	Reserved	–	–	
	D800 0000$_H$ – DDFF FFFF$_H$	96 MB	External Memory Space	via LMB	via LMB	
	DE00 0000$_H$ – DEFF FFFF$_H$	16 MB	Emulator Memory Space			
	DF00 0000$_H$ – DFFF BFFF$_H$	~16 MB	Reserved	–	–	
	DFFF C000$_H$ – DFFF FFFF$_H$	16 KB	Boot ROM Space	via S_FPI	via S_FPI	
14	E000 0000$_H$ – E7FF FFFF$_H$	128 MB	External Memory Space	via LMB	–	
	E800 0000$_H$ – E807 FFFF$_H$	512 KB	Local Memory Space mapped to LMB Segment 12	–		
	E808 0000$_H$ – E83F FFFF$_H$	3.5 MB	Reserved			
	E840 0000$_H$ – E840 5FFF$_H$	24 KB	Local Data Memory (SRAM) mapped to LMB Segment 13			
	E840 6000$_H$ – E84F FFFF$_H$	~1 MB	Reserved			
	E850 0000$_H$ – E850 5FFF$_H$	24 KB	Local Code Memory (SRAM) mapped to LMB Segment 13			
	E850 6000$_H$ – EFFF FFFF$_H$	~123 MB	Reserved			

Table 7-1 TC11IB Block Address Map (cont'd)

Seg-ment	Address Range	Size	Description	DMU Acc.	PMU Acc.	
15	F000 0000$_H$ – F00F FFFF$_H$	1 MB	On-Chip Peripherals & Ports	via S_FPI	via S_FPI	non-cached
	F010 0000$_H$ – F017 FFFF$_H$ [1)]	512 KB	Reserved	–	–	
	F018 0000$_H$ – F018 FFFF$_H$	64 KB	ComDRAM Control Registers	via S_FPI	via S_FPI	
	F019 0000$_H$ – F03F FFFF$_H$ [1)]	2.4375 MB	Reserved	–	–	
	F040 0000$_H$ – F04F FFFF$_H$	1 MB	PCI/FPI-Bridge Registers	via F_FPI	–	
	F050 0000$_H$ – F0FF FFFF$_H$	~11 MB	Reserved	–		
	F100 0000$_H$ – F1FF FFFF$_H$	16 MB	PCI Configuration Space	via F_FPI		
	F200 0000$_H$ – F200 05FF$_H$	6 x 256 B	BCU0 and Fast Ethernet Registers			
	F200 0600$_H$ – F7E0 FEFF$_H$	~94 MB	Reserved	–		
	F7E0 FF00$_H$ – F7E0 FFFF$_H$	256 B	CPU Slave Interface Registers (CPS)	via F_FPI		
	F7E1 0000$_H$ – F7E1 FFFF$_H$	64 KB	Core SFRs			
	F7E2 0000$_H$ – F7FF FFFF$_H$	15 x 128 KB	Reserved	–		
	F800 0000$_H$ – F87F FFFF$_H$	8 MB	LMB Peripheral Space (EBU_LMB and local memory eDRAM control registers)	via LMB		
	F880 0000$_H$ – FFFF FFFF$_H$	120 MB	Reserved	–		

1) Any access to this area will result in unpredicted behaviors of PORTs.

*Note: Accesses to address regions defined as "Reserved" in **Table 7-1** lead to a bus error. The exceptions are marked with 1).*

Segments 0-7

This memory range is assigned to be MMU Space if the processor is operating in Virtual mode. But if the processor is operating in Physical mode, this memory range is assigned to be FPI Space.

Segment 8

This memory segment is mirrored from Segment 10 for external EBU Space. But this segment have to be accessed via LMB Bus.

Segment 9

This memory segment contains 255 MBytes mapped from Segment 11. In which, 254MBytes is assigned to PCI and 1 MBytes is assigned to Com-DRAM. Both spaces can be accessed via fast FPI bus. Another 1 Mbytes is reserved in the TC11IB. It is assigned to cached access purposes in future product derivatives.

Segment 10

This memory segment contains 252 MBytes reserved area for external code/data memory and 4 Mbytes reserved for Local Memory Unit. Both spaces can be accessed via LMB Bus. But 4 Mbytes LMU space can not be accessed by any fast FPI master. If so, an error acknowledge will be given.

Segment 11

This memory segment is mapped to Segment 9. 254 Mbytes is assigned to PCI, 1 MBytes is assigned to ComDRAM memory and 1 MBytes is reserved.

Segment 12

This memory segment contains the 512 KBytes Local Code/Data DRAM memory operating in cached mode.

Segment 13

This memory segment contains the 24 KBytes Local Data Scratchpad Memory (DMU SRAM), the 24 KBytes Local Code Scratchpad Memory (PMU SRAM), the 16 KBytes Boot ROM, the 96 MBytes External Code/Data Memory and the 16 MBytes Emulator Memory operating in non-cached mode.

Segment 14

This memory segment contains a non-cached 128 MByte segment reserved for external code/data memory. The 512 KBytes Local Data Memory is mapped to LMB Segment 12. The 24 KBytes Local Data Memory (SRAM) is mapped to LMB Segment 13 and the 24 KBytes Local Code Memory (SRAM) is mapped to LMB Segment 13. Such internal mapping is done by LFI bridge that makes the FPI translations appear at LMB bus at different addresses. The **Table 7-2** summaries the mapping from FPI to LMB.

Table 7-2 Segment 14 Mapping from FPI to LMB

FPI Address	Name	LMB Address	Size
0xE850 0000-0xE850 5FFF	Code SRAM	0xD400 0000-0xD400 5FFF	24KByte
0xE840 0000-0xE840 5FFF	Data SRAM	0xD000 0000-0xD000 5FFF	24KByte
0xE800 0000-0xE807 FFFF	LMU	0xC000 0000-0xC007 FFFF	512KByte

Segment 15

This memory segment is dedicated for CPU, PCP, on-chip peripheral units, and ports (see **Table 7-3**).

7.2 Memory Segment 15 - Peripheral Units

Table 7-3 shows the block address map of Segment 15.

Table 7-3 Block Address Map of Segment 15

Symbol	Description	Address Range	Size
SCU	System Control Unit	F000 0000$_H$ – F000 00FF$_H$	256 Bytes
PCISIR	PCI Software Interrupt Request	F000 0100$_H$ – F000 01FF$_H$	256 Bytes
BCU1	Slow FPI Bus Control Unit 1	F000 0200$_H$ – F000 02FF$_H$	256 Bytes
STM	System Timer	F000 0300$_H$ – F000 03FF$_H$	256 Bytes
OCDS	On-Chip Debug Support	F000 0400$_H$ – F000 04FF$_H$	256 Bytes
–	Reserved	F000 0500$_H$ – F000 05FF$_H$	–
GPTU0	General Purpose Timer Unit 0	F000 0600$_H$ – F000 06FF$_H$	256 Bytes
GPTU1	General Purpose Timer Unit 1	F000 0700$_H$ – F000 07FF$_H$	256 Bytes
ASC	Async./Sync. Serial Interface	F000 0800$_H$ – F000 08FF$_H$	256 Bytes
16X50	Asynchronous Serial Interface	F000 0900$_H$ – F000 09FF$_H$	256 Bytes
SSC	High-Speed Synchronous Serial Interface	F000 0A00$_H$ – F000 0AFF$_H$	256 Bytes
MMCI	MultiMediaCard Interface	F000 0B00$_H$ – F000 0BFF$_H$	256 Bytes
SRU	Service Request Unit	F000 0C00$_H$ – F000 0DFF$_H$	512 Bytes
–	Reserved	F000 0E00$_H$ – F000 27FF$_H$	–
P0	Port 0	F000 2800$_H$ – F000 28FF$_H$	256 Bytes
P1	Port 1	F000 2900$_H$ – F000 29FF$_H$	256 Bytes
P2	Port 2	F000 2A00$_H$ – F000 2AFF$_H$	256 Bytes

Table 7-3 Block Address Map of Segment 15 (cont'd)

Symbol	Description	Address Range	Size
P3	Port 3	F000 2B00$_H$ – F000 2BFF$_H$	256 Bytes
P4	Port 4	F000 2C00$_H$ – F000 2CFF$_H$	256 Bytes
P5	Port 5	F000 2D00$_H$ – F000 2DFF$_H$	256 Bytes
–	Reserved	F000 2E00$_H$ – F000 3EFF$_H$	–
PCP	PCP Registers	F000 3F00$_H$ – F000 3FFF$_H$	256 Bytes
	Reserved	F000 4000$_H$ – F000 FFFF$_H$	–
	PCP Data Memory (PRAM)	F001 0000$_H$ – F001 0FFF$_H$	4 KBytes
	Reserved	F001 1000$_H$ – F001 FFFF$_H$	–
	PCP Code Memory (PCODE)	F002 0000$_H$ – F002 3FFF$_H$	16 KBytes
–	Reserved	F002 4000$_H$ – F017 FFFF$_H$	–[1)]
ComDRAM	ComDRAM Control Registers	F018 0000$_H$ – F018 FFFF$_H$	64 KBytes
–	Reserved	F019 0000$_H$ – F03F FFFF$_H$	–[1)]
PCIBCR	PCI Bridge Configuration Registers	F040 0000$_H$ – F04F FFFF$_H$	1 MBytes
–	Reserved	F050 0000$_H$ – F0FF FFFF$_H$	–
PCICS	PCI Configuration Space Registers	F100 0000$_H$ – F1FF FFFF$_H$	16 MBytes
BCU0	Fast FPI Bus Control Unit 0	F200 0000$_H$ – F200 00FF$_H$	256 Bytes
ECU	Ethernet Controller Unit	F200 0100$_H$ – F200 05FF$_H$	1280 Bytes
–	Reserved	F200 0600$_H$ – F7E0 FEFF$_H$	–

Table 7-3 Block Address Map of Segment 15 (cont'd)

Symbol	Description	Address Range	Size
CPU	Slave Interface Registers (CPS)	F7E0 FF00$_H$ – F7E0 FFFF$_H$	256 Bytes
	Reserved	F7E1 0000$_H$ – F7E1 7FFF$_H$	–
	MMU	F7E1 8000$_H$ – F7E1 80FF$_H$	256 Bytes
	Reserved	F7E1 8100$_H$ – F7E1 BFFF$_H$	–
	Memory Protection Registers	F7E1 C000$_H$ – F7E1 EFFF$_H$	12 KBytes
	Reserved	F7E1 F000$_H$ – F7E1 FCFF$_H$	–
	Core Debug Register (OCDS)	F7E1 FD00$_H$ – F7E1 FDFF$_H$	256 Bytes
	Core Special Function Registers (CSFRs)	F7E1 FE00$_H$ – F7E1 FEFF$_H$	256 Bytes
	General Purpose Register (GPRs)	F7E1 FF00$_H$ – F7E1 FFFF$_H$	256 Bytes
–	Reserved	F7E2 0000$_H$ – F7FF FFFF$_H$	–
EBU	EBU_LMB External Bus Unit	F800 0000$_H$ – F800 01FF$_H$	512 Bytes
–	Reserved	F800 0200$_H$ – F800 03FF$_H$	–
LMU	Local Memory Unit	F800 0400$_H$ – F800 04FF$_H$	256 Bytes
–	Reserved	F800 0500$_H$ – F87F FBFF$_H$	–
DMU	Local Data Memory Unit	F87F FC00$_H$ – F87F FCFF$_H$	256 Bytes
PMU	Local Program Memory Unit	F87F FD00$_H$ – F87F FDFF$_H$	256 Bytes
LCU	LMB Bus Control Unit	F87F FE00$_H$ – F87F FEFF$_H$	256 Bytes
LFI	LMB to FPI Bus Bridge (LFI)	F87F FF00$_H$ – F87F FFFF$_H$	256 Bytes
–	Reserved	F880 0000$_H$ – FFFF FFFF$_H$	–

1) Any access to this area will result in unpredicted behaviors of PORTs.

*Note: Accesses to address defined as "Reserved" in **Table 7-3** lead to a bus error. The exceptions are marked with 1).*

8 Program Memory Unit

The Program Memory Unit PMU controls the CPU code fetches from internal, local and external code memory. The functionality of the PMU is implemented with the combination of the Program Memory Interface (PMI), External Program Memory (PMEM) and the Program Cache TAGRAM Memory (PTAG). The PMU of the TC11IB consists of the functional blocks as shown in **Figure 8-1**:

- PMEM
  - 24 KByte scratch-pad code RAM (SRAM)
  - 8 KByte instruction cache (ICACHE)
- PMI
  - PMU control block
  - Interface to the CPU Instruction Fetch Unit
  - Interface to the MMU
  - LMB Bus interface

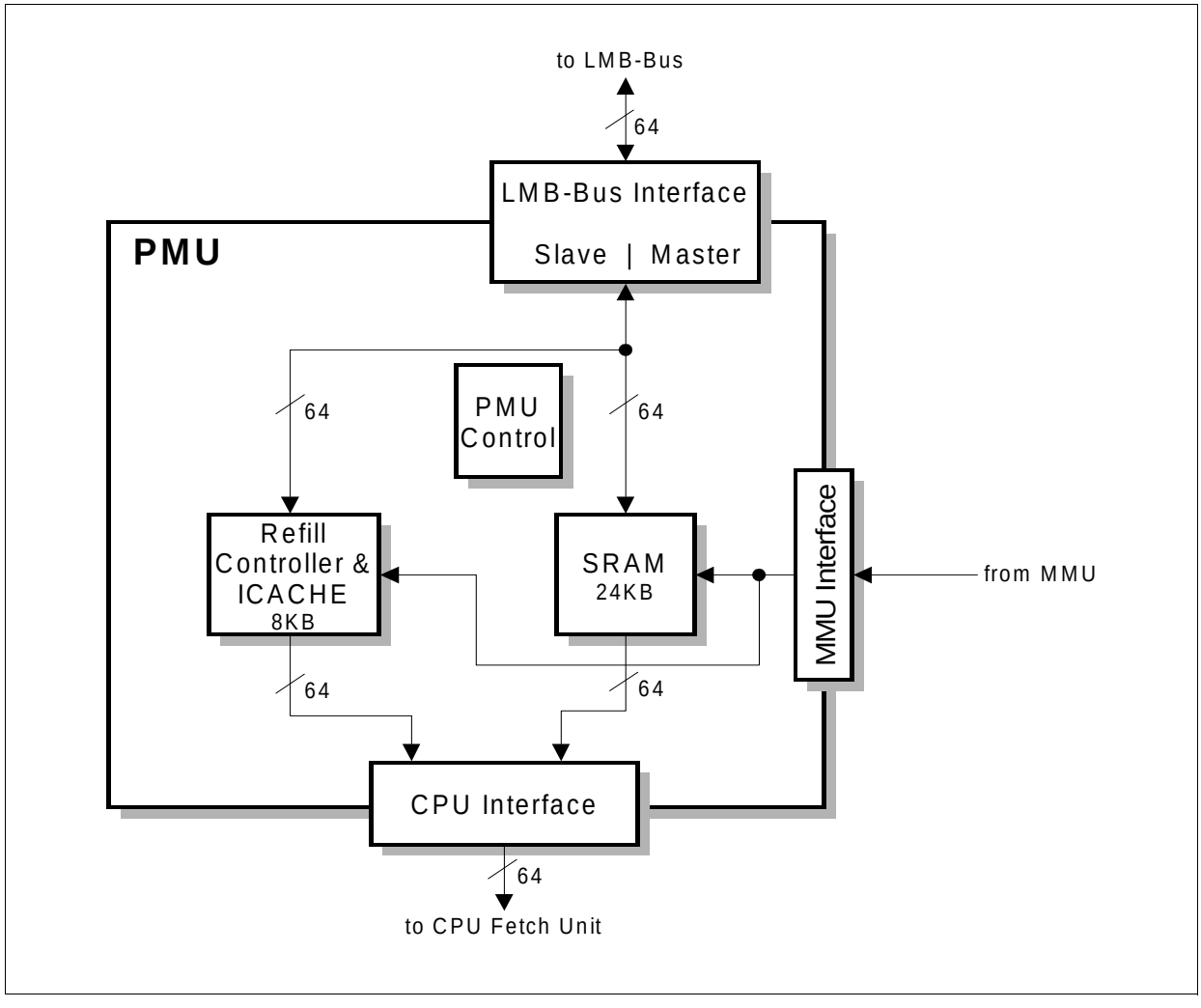


Figure 8-1 PMU Block Diagram with Data Paths

The LMB Bus interface is a master/slave interface which handles all transactions between LMB Bus and the PMU code memories. The master part of the interface is used when the PMU needs to access resources which are located on the LMB Bus. The slave part of the interface is used when another LMB Bus master needs to access PMU resources such as the CSRAM.

The MMU interface provides instruction access address together with CPU interface.

The Instruction Cache contains the cache RAM with the tag RAM and the Refill controller.

8.1 Memories Controlled by PMU

Table 8-1 gives the sections of TC11IB internal code/program memories that are controlled by the PMU.

Table 8-1 Address Map of PMU Related Memories

Segment	Address	Name	Description
On-Chip Memory			
13	D400 0000_H - D400 $5FFF_H$	SPRAM	24K Local Scratch-Pad Code RAM

8.2 Functions

The PMU functional flow is presented in the **Figure 8-2**. There are four functional flow paths.

- Path 1: Fetch Streaming or ICache Bypass from LMB.
- Path 2: ICache Hit (Read from ICache Way 0 or Way 1) or SPR access (from upper or lower SPR bank).
- Path 3: ICache Refill/Miss from LMB (Write to ICache).
- Path 4: SPR accesses from any LMB Master (Read/Write from/to SPR).

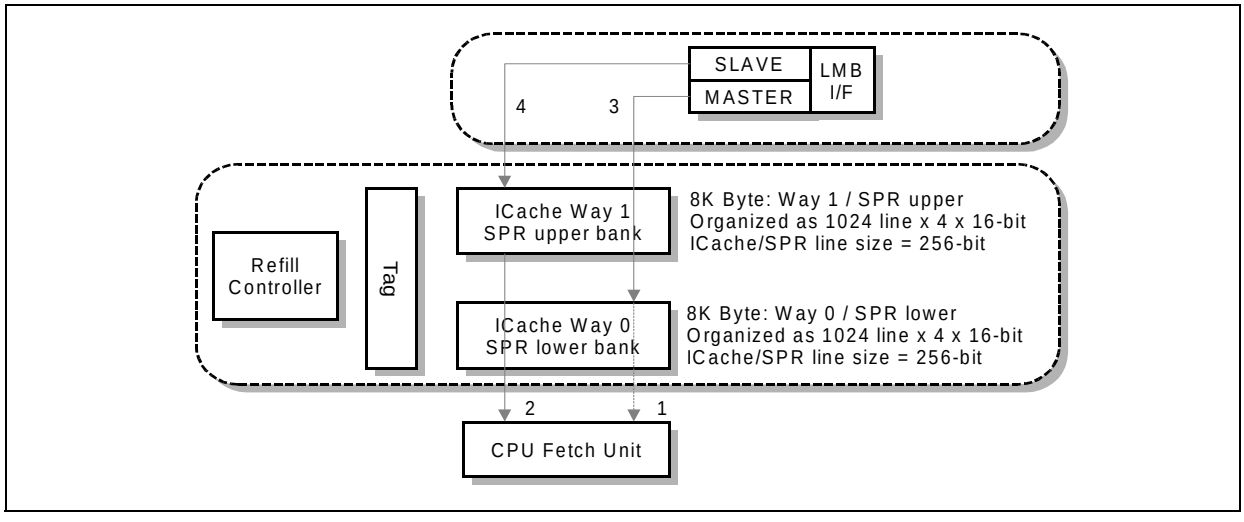


Figure 8-2 PMU Functional Flow

8.3 Scratch-Pad RAM, SPRAM

The Scratch-Pad RAM (SPRAM) is a 24-KByte static RAM. As a code memory, it is assigned especially to hold code that must be executed very fast (e.g. interrupt routines).

The SPRAM can be accessed from the LMB Bus side by another bus master, such as the Data Memory Unit, DMU. On a read access from the LMB Bus (possible in supervisor mode as well as in user mode), the data width can be only 64 bits (double-word) wide. The natural alignment of the accessed data must be obeyed, that is, bytes can be aligned on any byte boundary, half-words must be aligned to half-word (even byte) boundaries, and word accesses must be aligned to word boundaries. Accesses not following this rule will be flagged with an LMB Bus error by the PMU.

On a write access from the LMB Bus (only possible in supervisor mode!), the data width can only be 64 bits wide and must be aligned to double-word boundaries. Byte and half-word accesses are not allowed.

CPU fetch accesses to the address range of the SPRAM are never cached in the ICACHE. They are always directly targeted to the SPRAM. A code fetch access from the CPU to the SPRAM can be performed in one clock cycle, the data width of such an access is 64 bits. The fetch logic also supports unaligned accesses (16-bit aligned), with a minimum penalty of one cycle for unaligned accesses. Note that the CPU Fetch Unit can only read from the SPRAM and never write to it.

8.4 Instruction Cache, ICACHE

The ICACHE of the PMU is a two-way set-associative cache with a Least-Recently-Used (LRU) replacement algorithm.

8.4.1 Cache Organization

The organization of the ICACHE is 256 cache lines with 32 bytes per line. Each cache line is divided into four double-words (64 bits) with a valid bit in the tag line for each word. Alignment of a cache line results in an 4-double-word address line border (address bits A[4:0] = 0). With the 32/16-bit mixed instruction set formats of the TriCore, a full cache line can hold a minimum of eight 32-bit instructions and a maximum of sixteen 16-bit instructions.

The address of a CPU instruction fetch is first decoded to determine the access target (for example: Scratch Pad RAM, address range accessible via LMB Bus, cachable area). All CPU instruction fetch accesses in the address ranges of the cachable area (segments 8-9 and 12) are targeted to the Refill Buffer. If the ICACHE is enabled and ICACHE bypass disabled, the ICACHE is also targeted. If the address and its associated instruction are found in the cache (Cache Hit), the instruction is passed to the CPU's Fetch Unit. If the address is not found in the cache (Cache Miss), the PMU's cache controller issues a cache refill sequence.

8.4.2 Cache Bypass Control

The ICACHE can be bypassed as controlled by bit PMU_CON0.CCBYP to provide a direct fetch access from the CPU to on-chip and off-chip resources. The default value for bit CCBYP after reset is 1, thus bypassing of the ICACHE is enabled. To enable the ICACHE, CCBYP has to be set to 0 during initialization.

Note: PMU_CON0 register is an ENDINIT-protected register.

8.4.3 Refill Sequence for Cache

Cache refill is performed with a Critical Double Word First strategy until the end of the ICache line, without wrapping around, i.e. the refill size being 1, 2, 3 or 4 double words. This means that the refill sequence starts with the instruction actually requested (the critical double word) by the CPU Fetch Unit and continues to the end of the cache line. A refill will always be done in 64-bit quantities. If the critical word maps onto the first 64-bit entry in the cache line, a refill of the entire cache line, four double words, will be performed. If the critical word maps onto the last 64-bit entry of a cache line, only this double-word will be refilled. In any case all valid bits of the refilled cache line are cleared. Thus, depending on the location of the critical word, the refill sequence will always be from one up to four double-words without wrap-around (the instructions mapping to the refilled cache line which are on addresses lower than that of the critical word are not fetched, except for instructions located within the double-word containing the critical

word). A refill sequence will always only affect one cache line and is fully pipelined by the PMU. There is no prefetching of the next cache line (no crossing of lines). Except this, mode, the refill mechanism also allows Burst Refill (2W, 4W and 8W) the ICache line.

8.4.4 Instruction Streaming

The ICACHE supports instruction streaming, meaning that during a refill sequence, it can already deliver the critical word to the CPU's Fetch Unit (after having it assembled to a double-word) before the sequence is completed. If the ICACHE is bypassed, an access to a cachable address space is performed such that the cache controller issues a refill sequence without updating the cache contents (cache data and valid bits)

8.4.5 Cache Coherency, Cache Invalidation

The PMU does not have automatic cache coherence support. Changes in the contents of memory areas external to the PMU which have already been cached in the ICACHE are not detected. Software has to provide the cache coherency in such a case. The PMU supports this via the cache invalidation function. The ICACHE contents can be invalidated through setting the invalidate control bit PMU_CON1.CCINV. While this bit is set to 1, all cache accesses will be treated as Cache Miss Operations and a cache refill is performed.

8.5 PMU Registers

As shown in **Figure 8-3**, the following control register are implemented in the PMU. These registers and their bits are described in this section.

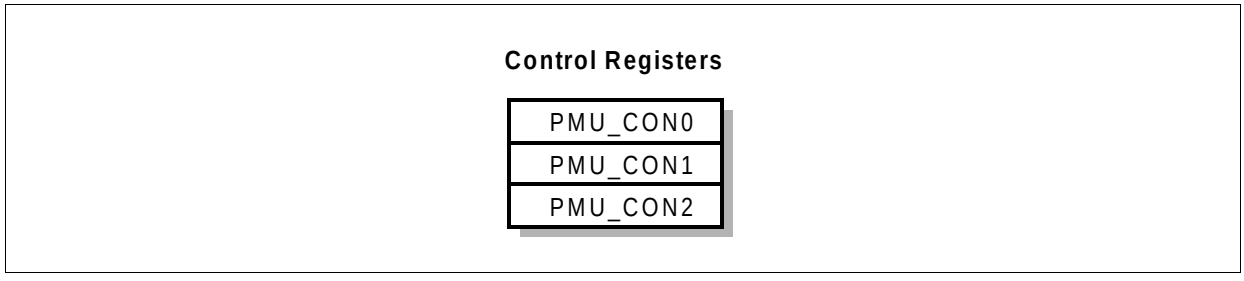


Figure 8-3 PMU Registers

Table 8-2 PMU Registers

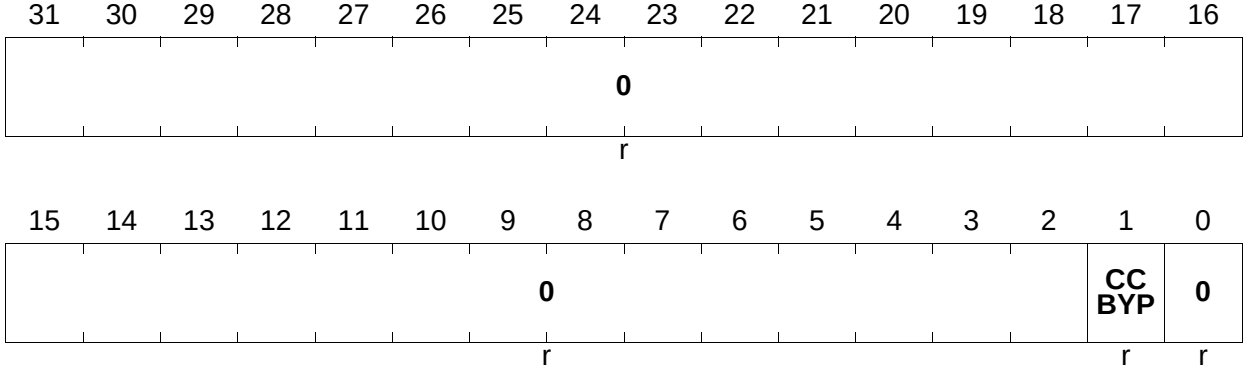
Register Short Name	Register Long Name	Offset Address	Description see
PMU_CON0	PMU Control Register 0	0010_H	**Page 8-7**
PMU_CON1	PMU Control Register 1	0014_H	**Page 8-8**
PMU_CON2	PMU Control Register 2	0018_H	**Page 8-9**

In the TC11IB, the registers of the PMU are located in the following address range:

– Module Base Address: F87F $FD00_H$
  Module End Address: F87F $FDFF_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 8-2**)

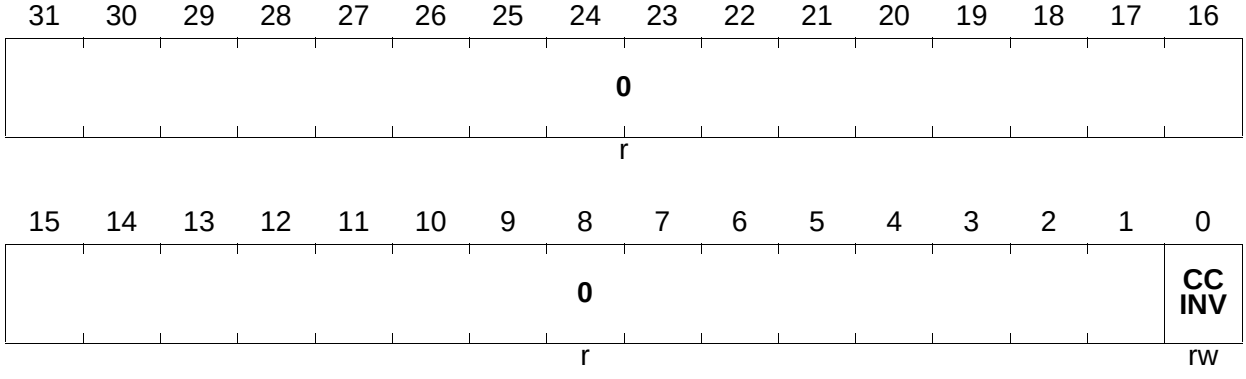
8.5.1 PMU Control Registers

PMU_CON0
PMU Control Register 0 **Reset Value: 0000 0002$_H$**



Field	Bits	Type	Description
CCBYP	1	rw	**Code Cache Bypass Control** 0 Cache is enabled 1 Cache is disabled (default after reset)
0	[31:2], 0	r	**Reserved**; read as 0; should be written with 0.

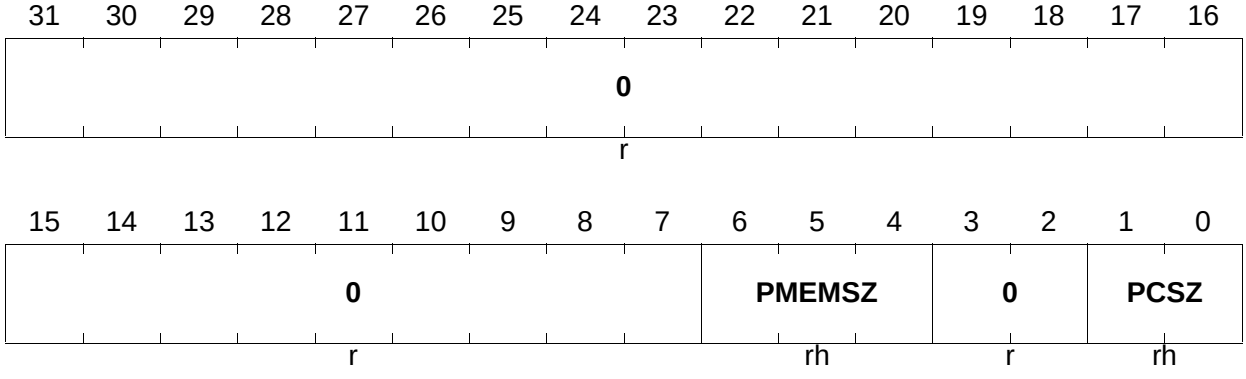
PMU_CON1
PMU Control Register 1 **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
CCINV	0	rw	**Code Cache Invalidate Control** 0 Normal Operation 1 Cache code is flushed As long as CCINV is set, all instruction fetch accesses generate a cache refill. It is advised to keep CCINV set until ICache coherency is guaranteed.
0	[31:1]	r	**Reserved**; read as 0; should be written with 0.

PMU_CON2
PMU Control Register 2

Reset Value: 0000 0032$_H$



Field	Bits	Type	Description
PCSZ	[1:0]	rh	**Program Cache Size** 10 8 Kbyte cache, 128x55 TAGRAM othersReserved. This field shows the configuration of the TAGRAM.
PMEMSZ	[6:4]	rh	**Program Memory Size** 011 32 Kbyte PMEM othersReserved This field shows the configuration of the Program Memory[1)].
0	[31:7] [3:2]	r	**Reserved**; read as 0; should be written with 0.

1) The Program Memory PMEM itself is not part of the TriCore system.
SPR size = total size (PMEM) - cache size.

9 Data Memory Unit

The functionality of the Data Memory Unit (DMU) is implemented with the combination of the Data Memory Interface (DMI), External Data Memory (DMEM) and two Data Cache TAGRAM Memories (DTAG). The block of DMU is shown in **Figure 9-1**, contains:

- DMEM
  - 24 KBytes data memory (SRAM)
  - 8 KBytes data cache memory (DCACHE)
- DMI
  - DMU control block
  - Interface to the CPU Load/Store Unit
  - Interface to the MMU Unit
  - Interface to the LMB Bus

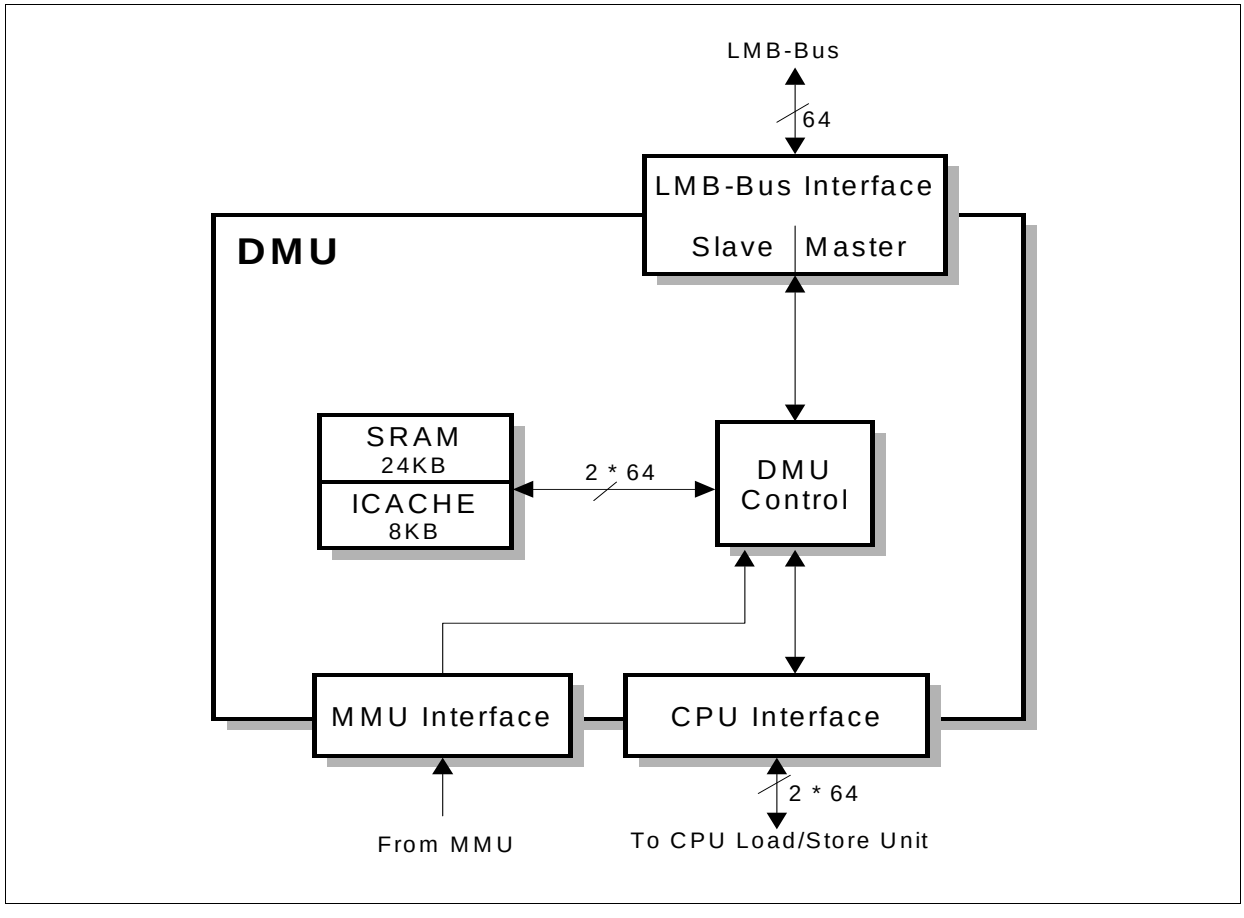


Figure 9-1 Block Diagram of the Data Memory Unit (DMU)

The LMB Bus interface of the DMU can operate in either master or slave mode. The master part of the interface is used when the CPU Load/Store Unit requests a data access to a data resource that is outside the DMU on the LMB Bus (for example, a module connected to the LMB Bus, such as Local Memory Unit (LMU)). The slave part

of the interface is required when another LMB Bus master (such as the External Bus Control Unit (EBU)) needs to access the DMU data memory.

The data width for read and write accesses to/from the data memory within the DMU via the LMB Bus can be 8, 16, 32 or 64 bits (byte, half-word, word or double-word). Not only the natural alignment of the accessed data is supported, that is, byte must be aligned to byte boundaries, half-word must be aligned to half-word (even byte) boundaries, word accesses must be aligned to word boundaries and double-word must be aligned to double-word boundaries, but also unalignment accesses are supported. These include word access can be half-word aligned, double-word access can be half-word aligned also.

The MMU interface provides data access address together with CPU interface.

The data memory is located at the beginning of the non-cacheable Segment 13. DMU Registers are located at Segment 15.

Table 9-1 DMU Address Map

Segment	Address	Name	Description	CPU Access		Bus Access	
				Load	Store	Read	Write
13	D000 0000$_H$ - D000 5FFF$_H$	SRAM	Data Memory	1)		2) 3)	
13	D000 6000$_H$ - D3FF FFFF$_H$	Reserved DMU Space		–		BE	BE
15	F87F FC00$_H$ - F87F FCFF$_H$	–	DMU Registers	1)		2)	

1) CPU Load/Store accesses to this range can be performed in User or Supervisor Mode. Access width can be 8, 16, 32 or 64 bit, with 8-bit data aligned on byte boundaries and all others aligned on half-word (16 bit) boundaries. Misaligned accesses to the data memory by the CPU's Load/Store Unit will not occur since such conditions will already be handled inside the CPU (Unalignment trap, ALN).

2) This address range only can be accessed via LMB Bus. The read/write accesses from the LMB Bus can be performed in User or Supervisor Mode.

3) Address range D000 0000$_H$ - D000 5FFF$_H$ can only be used by TriCore for Context-Save area and load/store. It means that this area can not be used for storing instructions, including interrupt as well as data for other masters.

The placement of the SRAM into the lower half of Segment 13 facilitates the use of the absolute addressing mode for load and store operations, supporting fast access to data stored in the lower 16 KBytes of the data memory (in absolute addressing mode, an address in the lower 16 KBytes of each of the segments can be specified as an immediate address of a load/store instruction; such an address does not have to first be loaded into an address register).

Note: Read-modify-write instructions from LMB Bus to DMU memory are locked.

9.1 DMU Trap Generation

Several error conditions can lead to a trap being reported by the DMU back to the CPU. These include range errors, DMU control register access errors, and LMB Bus errors.

To facilitate a detailed analysis of an error/trap, the DMU provides two read-only status registers that hold information about the type of the error. The Synchronous Trap Flag Register (DMU_STR) contains the flags indicating the cause of a synchronous trap, while the Asynchronous Trap Flag Register (DMU_ATR) holds the flags for the cause of an asynchronous trap.

In general, whether an operation in the DMU can result in a synchronous or asynchronous error trap depends on the actual condition and sequence of operation in the DMU. Thus, for each of the possible DMU error scenarios, an error flag is provided in both registers DMU_STR and DMU_ATR. When an error is detected in the DMU, the respective trap signal is generated to the CPU and the appropriate bit in the associated trap flag registers is set.

The Trap Service Routine (TSR) invoked through the trap then needs to read the appropriate DMU Trap Flag Register to further determine the root cause of the trap. Reading a DMU Trap Flag Register in Supervisor Mode returns the contents of the register, and then clears the register to 0. Reading a trap flag register in User Mode only returns the contents of the register, but leaves it unaltered. The latter operation is implemented to allow debuggers/emulators to examine the status of the trap flag register without modifying it. The TSRs of user application code should always read these registers in Supervisor Mode in order to clear their contents.

9.1.1 LMB Bus Error

Two kinds of status flags are implemented to indicate an LMB Bus error. One kind of flag indicates errors resulting from a LMB Bus store operation, the other one kind indicates errors resulting from a LMB Bus load operation. The appropriate flags (DMU_STR.LFESTF, DMU_STR.SFESTF, DMU_ATR.LFEATF or DMU_ATR. SFEATF) are set if a DMU operation to or from the LMB Bus is performed, and an error occurs on the LMB Bus.

9.1.2 Range Error

Range errors are caused by accesses to reserved address ranges in the DMU. Accesses to address ranges in Segment 13 (DMU), Segment 14 and Segment 15 which are not covered by the data memory or the DMU control register ranges will lead to a range trap.

In each of the DMU trap flag registers, two kinds of status flags are implemented to indicate a range error. One kind of flag indicates errors resulting from a store operation

(DMU_ATR. SREATF and DMU_STR. SRESTF), the other one kind indicates errors resulting from a load operation (DMU_ATR. LREATF and DMU_STR. LRESTF). The appropriate flag is set if an access to the reserved address ranges is performed.

9.1.3 DMU Register Access Error

DMU register access errors are caused if an improper access to a DMU register is performed.

CPU load/store access to the DMU registers must only be made with word-aligned word accesses. An access not conforming to this rule, or an access that does not follow the specified privilege mode (supervisor mode, EndInit-protection), or a write access to a read-only register, will lead to a DMU Control Register Error trap. An access to reserved locations within the DMU register address area will not be flagged with an error. A read will return all zeros, a write will have no effect.

In each of the DMU trap flag registers, two kind of status flags are implemented to indicate a register access error. One kind of flag indicates errors resulting from a store operation (DMU_ATR.SCEATF and DMU_STR. SCESTF), the other one kind indicates errors resulting from a load operation (DMU_STR.LCESTF and DMU_ATR. LCEATF).The appropriate flag is set if an improper access to the DMU registers is performed.

9.1.4 Cache Management Error

Cache management errors are generated when one of the special cache instructions, DFLUSH, DINV and DFLINV, specify a non-cacheable address.

9.2 DMU Registers

As shown in **Figure 9-2** and **Table 9-2**, one control register and two trap status registers are implemented in the DMU. The registers and their bits are described in the following sections.

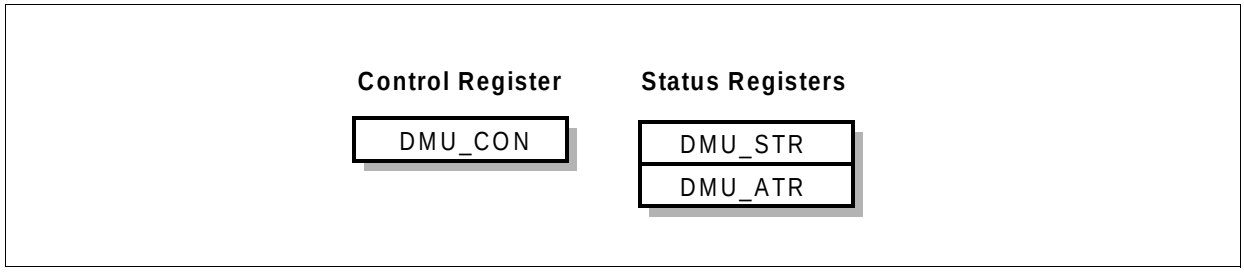


Figure 9-2 DMU Registers

Table 9-2 DMU Registers

Register Short Name	Register Long Name	Offset Address	Description see
DMU_CON	DMU Control Register	0010_H	**Page 9-6**
DMU_STR	DMU Synchronous Trap Flag Register	0018_H	**Page 9-7**
DMU_ATR	DMU Asynchronous Trap Flag Register	0020_H	**Page 9-9**

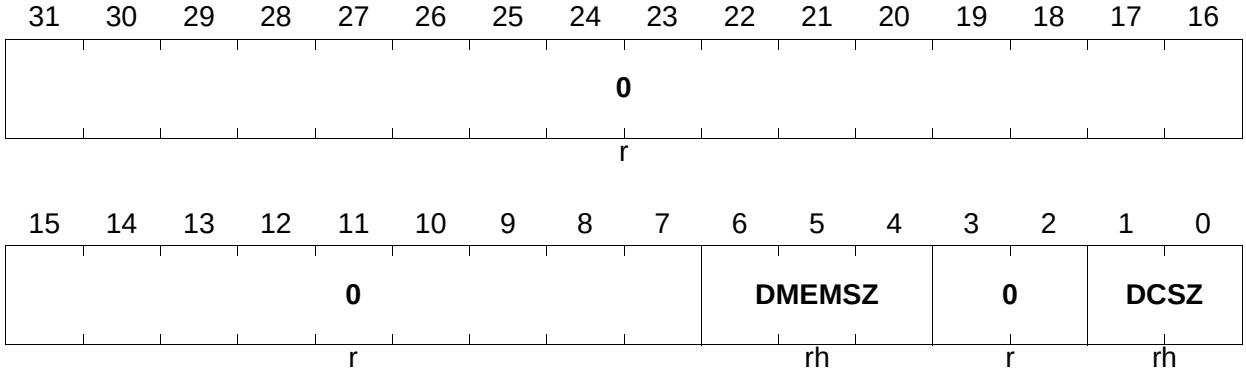
Note: Accesses to DMU registers must be made with word-aligned word accesses. An access not conforming to this rule will cause a bus error if the access was from the LMB Bus, or a trap in case of a CPU load/store access.

In the TC11IB, the registers of the DMU are located in the following address range:

– Module Base Address. F87F $FC00_H$
  Module End Address. F87F $FCFF_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 9-2**)

9.2.1 Control Register

DMU_CON
DMU Control Register **Reset Value: 0000 0032$_H$**



Field	Bits	Type	Description
DCSZ	[1:0]	rh	**Data Cache Size** 10 8 Kbyte cache, 128x55 TAGRAM othersReserved. This field shows the configuration of the TAGRAM.
DMEMSZ	[6:4]	rh	**Data Total Memory Size** 011 32 Kbyte DMEM othersReserved This field shows the configuration of the Data Memory[1)].
0	[31:7] [3:2]	r	**Reserved**; read as 0; should be written with 0.

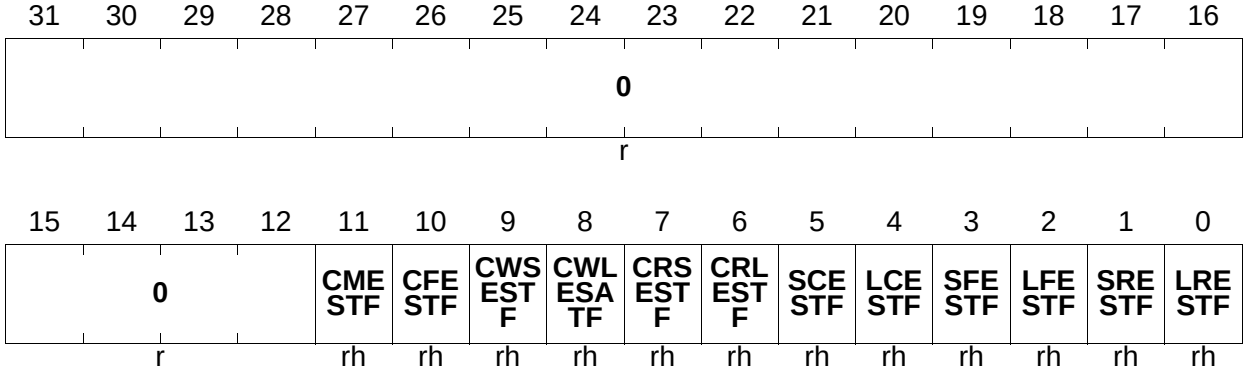
1) Cache size must be less than or equal to the SPR
SPR(Scratch Pad RAM) size = total size of DMEM - cache size

9.2.2 Synchronous Trap Flag Register

The Synchronous Trap Flag Register, DMU_STR, holds the flags that inform about the

root cause of a DMU Synchronous Trap (DSE) event.

DMU_STR
DMU Synchronous Trap Flag Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
LRESTF	0	rh	**Load Range Synchronous Error Flag** 0 No error 1 Synchronous load range error has occurred
SRESTF	1	rh	**Store Range Synchronous Error Flag** 0 No error 1 Synchronous store range error has occurred
LFESTF	2	rh	**LMB Bus Load Synchronous Error Flag** 0 No error 1 Synchronous LMB Bus load error has occurred
SFESTF	3	rh	**LMB Bus Store Synchronous Error Flag** 0 No error 1 Synchronous LMB Bus store error has occurred
LCESTF	4	rh	**DMU Register Load Synchronous Error Flag** 0 No error 1 Synchronous DMU register load error has occurred
SCESTF	5	rh	**DMU Register Store Synchronous Error Flag** 0 No error 1 Synchronous DMU register store error has occurred

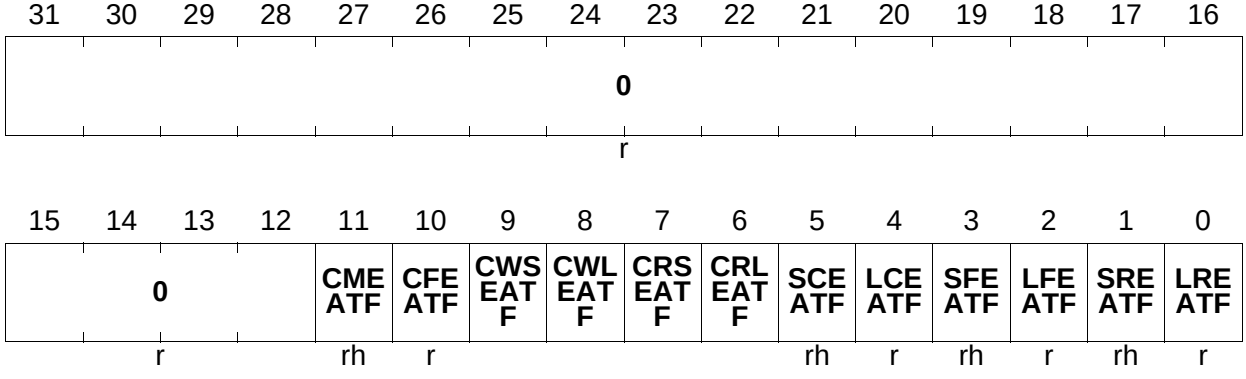
Field	Bits	Type	Description
CRLESTF	6	rh	**Cache Refill Load Synchronous Error Flag** 0 No error 1 Synchronous cache refill load error has occurred
CRSESTF	7	rh	**Cache Refill Store Synchronous Error Flag** 0 No error 1 Synchronous cache refill store error has occurred
CWLESTF	8	rh	**Cache Writeback Load Synchronous Error Flag** 0 No error 1 Synchronous cache writeback load error has occurred
CWSESTF	9	rh	**Cache Writeback Store Synchronous Error Flag** 0 No error 1 Synchronous cache writeback store error has occurred
CFESTF	10	rh	**Cache Flush Synchronous Error Flag** 0 No error 1 Synchronous cache flush error has occurred
CMESTF	11	rh	**Cache Management Synchronous Error Flag** 0 No error 1 Synchronous cache management error has occurred
0	[31:12]	r	**Reserved**; read as 0.

Note: When reading DMU_STR in Supervisor Mode, the contents of the register are returned and the bits of the register are then automatically cleared. Reading DMU_STR in User Mode returns the contents only, the register is not cleared.

9.2.3 Asynchronous Trap Flag Register

The Asynchronous Trap Flag Register, DMU_ATR, holds the flags that inform about the root cause of a DMU Asynchronous Trap (DAE) event.

DMU_ATR
DMU Asynchronous Trap Flag Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
LREATF	0	rh	**Load Range Asynchronous Error Flag** 0 No error 1 Asynchronous load range error has occurred
SREATF	1	rh	**Store Range Asynchronous Error Flag** 0 No error 1 Asynchronous store range error has occurred
LFEATF	2	rh	**LMB Bus Load Asynchronous Error Flag** 0 No error 1 Asynchronous LMB Bus load error has occurred
SFEATF	3	rh	**LMB Bus Store Asynchronous Error Flag** 0 No error 1 Asynchronous LMB Bus store error has occurred
LCEATF	4	rh	**DMU Register Load Asynchronous Error Flag** 0 No error 1 Asynchronous DMU register load error has occurred
SCEATF	5	rh	**DMU Register Store Asynchronous Error Flag** 0 No error 1 Asynchronous DMU register store error has occurred

Field	Bits	Type	Description
CRLEATF	6	rh	**Cache Refill Load Asynchronous Error Flag** 0 No error 1 Asynchronous cache refill load error has occurred
CRSEATF	7	rh	**Cache Refill Store Asynchronous Error Flag** 0 No error 1 Asynchronous cache refill store error has occurred
CWLEATF	8	rh	**Cache Writeback Load Asynchronous Error Flag** 0 No error 1 Asynchronous cache writeback load error has occurred
CWSEATF	9	rh	**Cache Writeback Store Asynchronous Error Flag** 0 No error 1 Asynchronous cache writeback store error has occurred
CFEATF	10	rh	**Cache Flush Asynchronous Error Flag** 0 No error 1 Asynchronous cache flush error has occurred
CMEATF	11	rh	**Cache Management Asynchronous Error Flag** 0 No error 1 Asynchronous cache management error has occurred
0	[31:12]	r	**Reserved**; read as 0.

Note: When reading DMU_ATR in Supervisor Mode, the contents of the register are returned and the bits of the register are automatically cleared. Reading DMU_ATR in User Mode returns the contents only, the register is not cleared.

10 Memory Management Unit

The TC11IB Memory Management Unit (MMU) handles all memory operations as well as arbitration between data stores and memory. The MMU implements virtual memory and translates virtual addresses of each running process to physical addresses in memory. The load/store unit addresses are translated and fed into the Data Memory Unit (DMU), while the instruction fetch unit addresses are translated and fed into the Program Memory Unit (PMU). Virtual memory is a method by which applications are written assuming a full 64-bit address space is available. This abstraction requires partitioning of the logical (virtual) address space into pages that are mapped into physical (real) memory. The operating system in turn translates a 64-bit address into a 32-bit address space supported by the processor. The MMU provides the translation of a 32-bit virtual address to a 32-bit physical address through the use of Translation Lookaside Buffers (TLB).

The MMU also provides memory protection so that a process can be prohibited from reading or writing the address space of another process. This guarantees memory integrity between processes. Access protection is also supported to ensure that any given process does not gain unauthorized access to memory. For example, a process will not be allowed to modify areas that are marked as read-only or reserved for supervisory software.

Finally, the MMU performs the arbitration function among I/O, Data cache, Instruction cache, and TLB references to memory. In essence, the MMU controls and prioritizes access to main memory. At any given time, a contention for memory access may arise between an I/O access involving the bus as well as internal accesses requested by Instruction Cache, Data Cache, and TLB references. The MMU is connected to the system as shown in **Figure 10-1**:

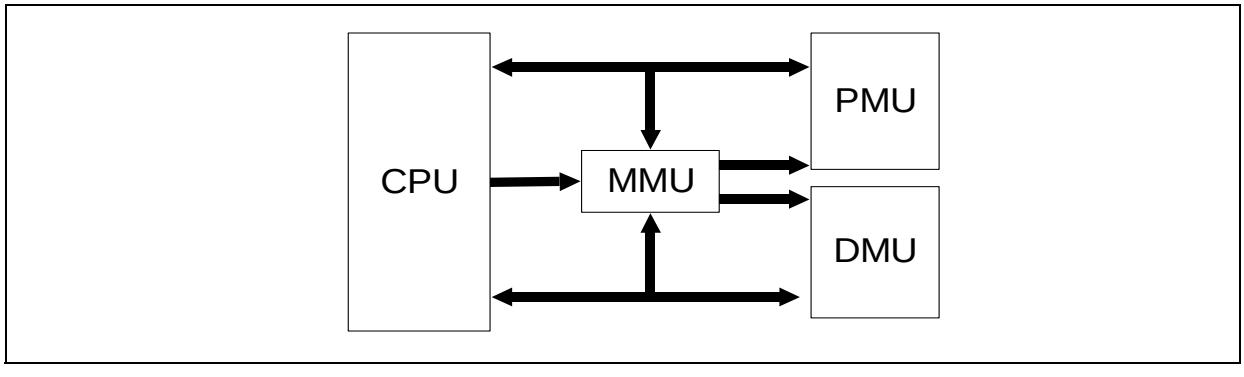


Figure 10-1 MMU System Integration Diagram

Features

- Implements virtual memory
- Provides virtual address to physical address memory translation
- Provides memory protection
- Performs arbitration among Input/Output, Data Cache, Instruction Cache, and Translation Lookaside Buffer references to memory.

10.1 Address Spaces

The size of the virtual address space is 4 GB, divided into 16 segments of 256 MB. The upper 4 bits of the 32-bit virtual address identify the segment. Virtual segments are numbered 0 – 15. A virtual address is always translated into a physical address before accessing memory. The size of the physical address space is 4 GB, divided into 16 segments of 256 MB. The upper 4 bits of the 32-bit physical address identify the segment. Physical segments are numbered 0 – 15. The physical and virtual address space maps are shown in **Figure 10-2**. A 32-bit virtual address is comprised of a Virtual Page Number (VPN) concatenated with a Page Offset. A 32-bit physical address is comprised of a Physical Page Number (PPN) concatenated with a Page Offset.

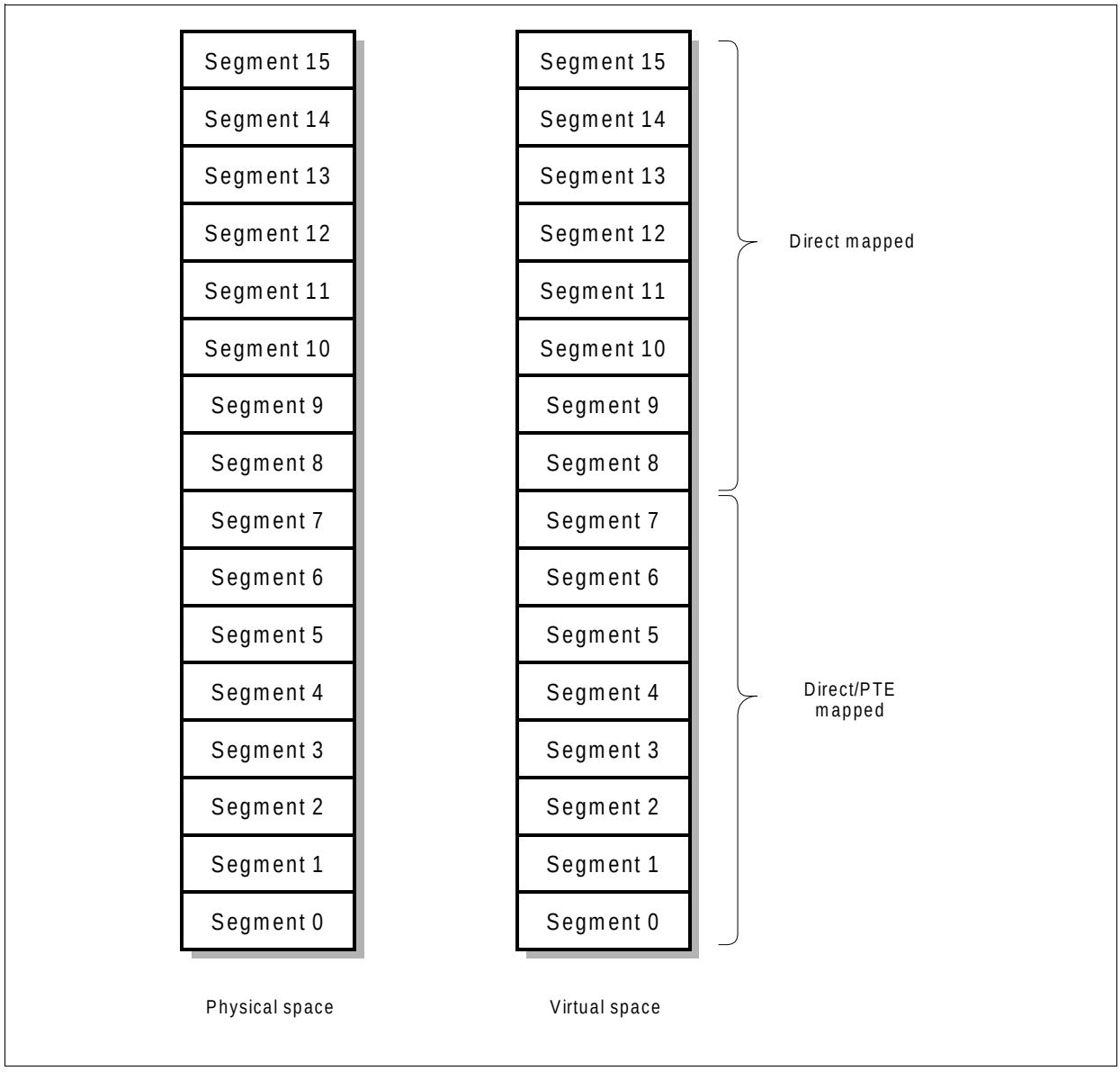


Figure 10-2 Physical and Virtual Address Spaces

10.1.1 Address Translation

The virtual address is translated into a physical address by one of two mechanisms: (a) direct translation or (b) Page Table Entry (PTE)-based translation, as shown in **Figure 10-3**. If the virtual address belongs to the upper half of the virtual address space, the virtual address is used directly as the physical address (direct translation). If the virtual address belongs to the lower half of the address space, the virtual address is used directly as the physical address if the processor is operating in Physical mode (direct translation) or is translated using a Page Table Entry if the processor is operating in Virtual mode (PTE translation). The MMU_CON.V bit controls the Physical/Virtual operating mode of the processor, as outlined in the section on the MMU_CON register. Translation using the PTE is achieved by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN) to obtain a physical address.

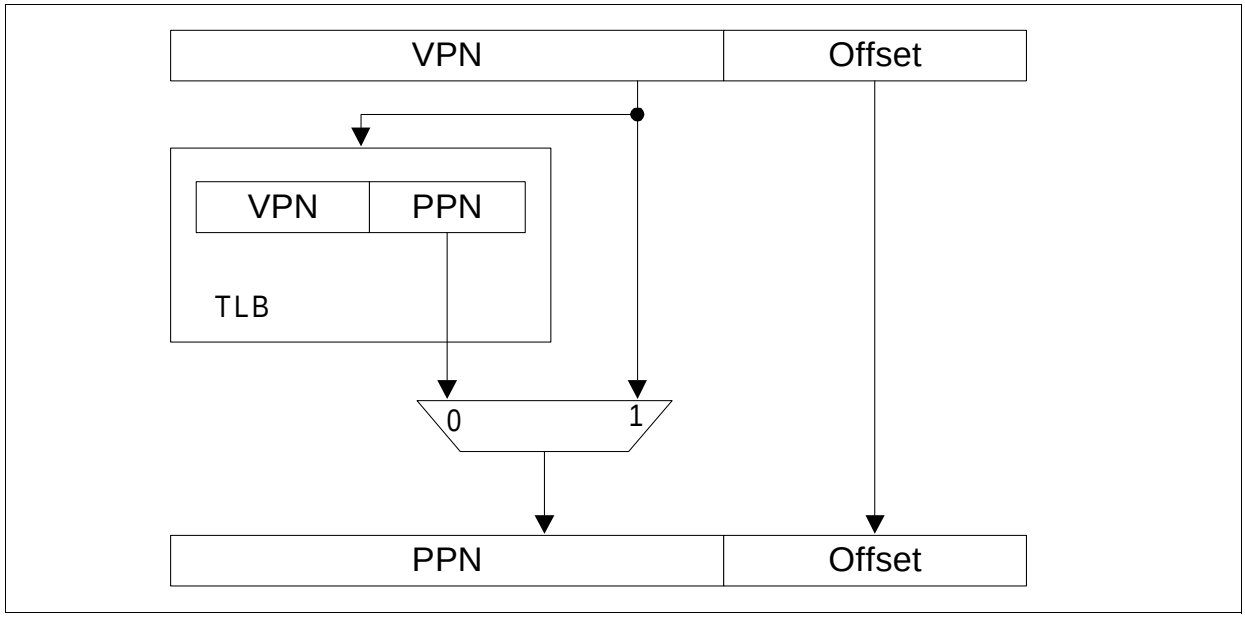


Figure 10-3 Virtual Address Translation

For Context Pointers (PCX, FCX, LCX), the address translations are constrained to use the direct translation path.

10.1.2 Translation Lookaside Buffers

The MMU provides PTE-based virtual address translation through two Translation Lookaside Buffers (TLBs): TLB-A and TLB-B. The MMU supports up to four page sizes, including: 1 KB, 4 KB, 16 KB, and 64 KB. However, at any given time, each TLB provides translations for only one particular page size. The page size setting of each TLB is determined by the MMU_CON.SZA and MMU_CON.SZB fields, as outlined in the section on the MMU_CON register.

Each TLB contains N TLB Table Entries (TTEs) where 4 <= N <= 128. The MMU_CON.TSZ field determines the size of each TLB, as outlined in the section on the MMU_CON register. Each TTE has an 8-bit index associated with it. Index numbers 0, ..., MMU_CON.TSZ are used for the entries in TLB-A. Index numbers 128, ..., 128+MMU_CON.TSZ are used for the entries in TLB-B. Each TTE contains a Page Table Entry (PTE).

A Translation Lookaside Buffers Table Entry (TTE) contains the following fields:

- ASI (Address Space Identifier): This is a 5-bit field that specifies the address space corresponding to the virtual address. ASIs allow mappings of up to 32 virtual address spaces to coexist in the TLB.
- VPN (Virtual Page Number): This field contains the Virtual Page Number. This field stores 32 – log2 (PageSize) bits, where PageSize is the size of the page in bytes.
- PPN (Physical Page Number): This field contains the Physical Page Number. This field stores 32 – log2 (PageSize) bits, where PageSize is the size of the page in bytes.
- XE (Execute Enable) enables instruction fetch to the page.
- WE (Write Enable) enables data writes to the page.
- RE (Read Enable) enables data reads from the page
- C (Cacheability bit) indicates that the page is cacheable.
- G (Global bit) indicates that the page is globally mapped, thus making it visible in all address spaces.
- V (Valid bit) indicates that the TTE contains a valid mapping.

10.1.3 Cacheability

The cacheability of a virtual address is determined using separate mechanisms for the two translation paths.

10.1.3.1 Cacheability for Direct Translation

The cacheability status of a virtual address that undergoes direct translation is controlled by a specific cacheability attribute associated with the segment, shown as **Figure 10-4**. The segment cacheability attributes are not part of the MMU specification. These cacheability attributes are provided by the system memory map for the specific CPU core.

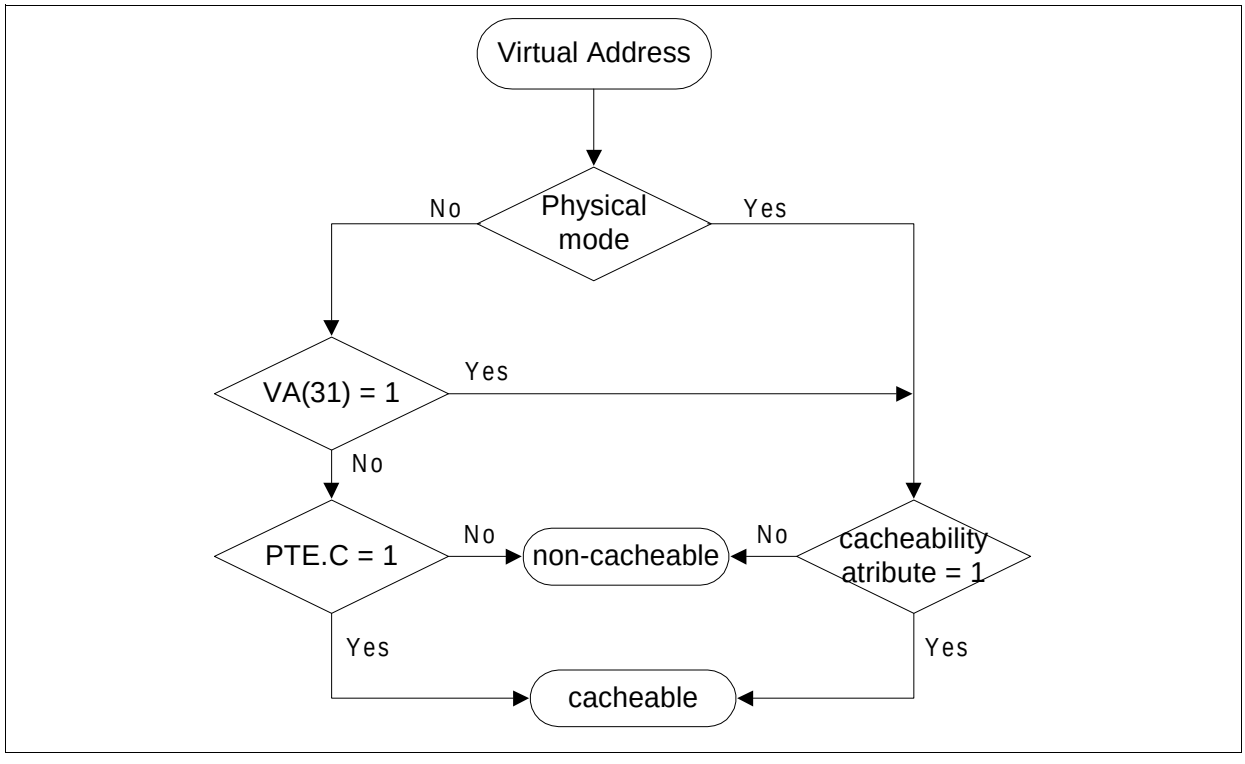


Figure 10-4 Cacheability of a Virtual Address

10.1.3.2 Cacheability for PTE-Based Translation

The cacheability status of a virtual address that undergoes PTE-based translation is determined using the cacheability attribute of the PTE used for the address translation. Each PTE has a Cacheability bit (C bit) that controls the cacheability status of the page.

10.1.4 Memory Protection

Memory protection is enforced using separate mechanisms for the two translation paths.

10.1.4.1 Protection for Direct Translation

Memory protection for addresses that undergo direct translation is enforced using the range-based protection used in the previous generation of the TriCore architecture. This mechanism protects memory ranges from unauthorized read, write, or instruction fetch accesses. The TriCore architecture provides up to four protection register sets. The PSW.PRS field controls the selection of the protection register set. Refer to the TriCore Architecture Manual for details of this protection model. Also, User-0 accesses to virtual addresses in the upper half of the virtual address space are disallowed in Virtual mode. In Physical mode, User-0 accesses are disallowed only to segments 14 and 15. Any User-0 access to a virtual address that is restricted to User-1 or Supervisor mode will cause a Virtual Address Protection (VAP) Trap in both Physical and Virtual modes.

10.1.4.2 Protection for PTE-Based Translation

Memory protection for addresses that undergo PTE-based translation is enforced using the PTE used for the address translation. The PTE provides support for protecting a process from unauthorized read, write, or instruction fetches by other processes. The PTE uses the following bits for the purpose of protection:

- XE (Execute Enable) enables instruction fetch to the page.
- WE (Write Enable) enables data writes to the page.
- RE (Read Enable) enables data reads from the page.

10.1.5 Multiple Address Spaces

The MMU provides efficient support for multiple virtual address spaces.

Each TTE contains an Address Space Identifier (ASI) that identifies the address space corresponding to the particular virtual address. Ambiguities in virtual address mappings are avoided by the use of the Address Space Identifier. A special Address Space Identifier (ASI) register is also provided to enable multiple address spaces.

The virtual address translation is performed by a TTE if: (a) it is a valid non-global TTE that matches the incoming VPN of the virtual address and the address space identifier contained in the ASI register, or (b) it is a valid global TTE that matches the incoming VPN. Note that global TTEs are indicated by the Global bit (G bit) and such mappings are visible to all virtual address spaces.

10.1.6 MMU Traps

MMU traps belong to Trap Class Number (TCN) 0 in the TriCore architecture. The MMU can generate the following traps:

- VAF (Virtual Address Fill)
- VAP (Virtual Address Protection)

The Virtual Address Fill (VAF) trap is generated if PTE-based translation is required for a virtual address and the PTE corresponding to the translation is missing in the MMU. The Virtual Address Protection (VAP) trap is generated if the access is disallowed. The VAF trap is assigned a TIN (Trap Identification Number) of 0 while the VAP trap is assigned a TIN of 1. Both VAF and VAP are synchronous traps.

The events that happen on an MMU trap are identical to the events that happen on any other trap. The virtual address is right shifted by 10 + 2*min(SZA, SZB) and loaded into the Translation Fault Address (TFA) register. The upper context is saved before loading the TIN into the D15 register. The address of the trap handler is computed by left shifting the TCN by 5 bits and ORing it with the Base Trap Vector (BTV) register. An implicit transfer of control then occurs by loading the program counter with the address of the trap handler.

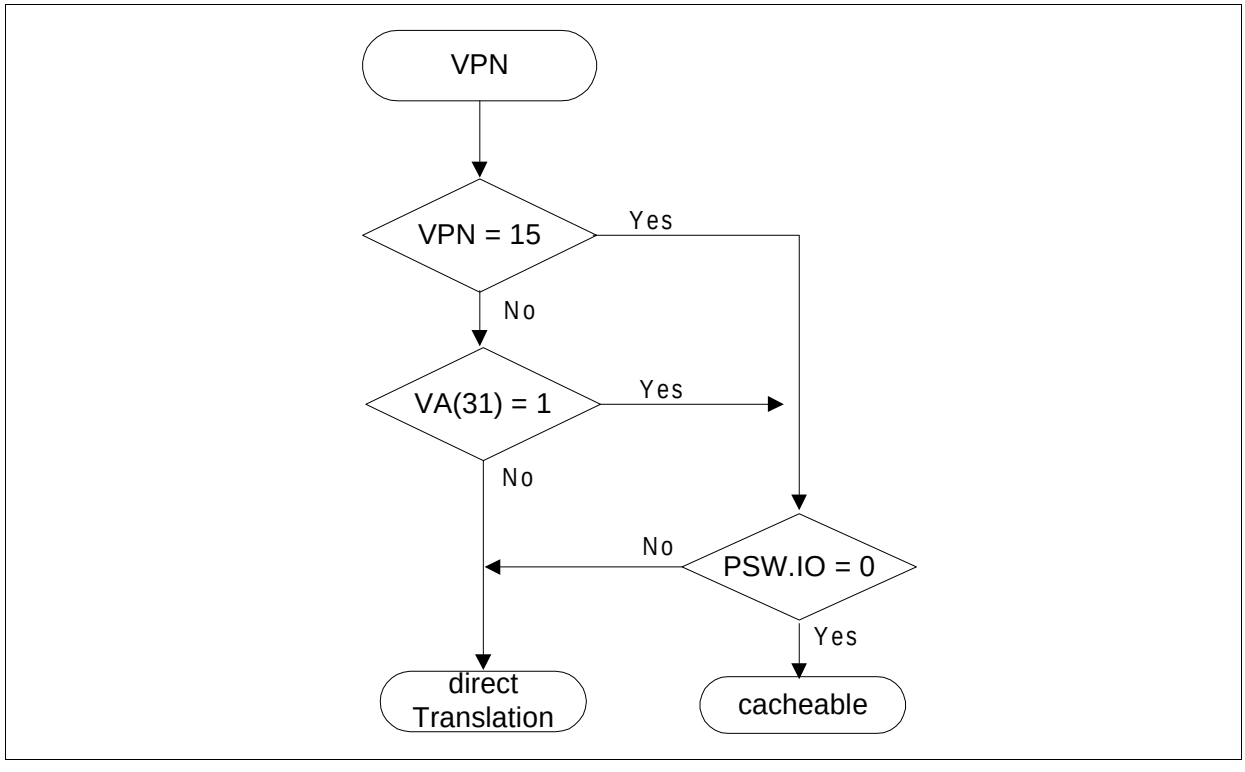


Figure 10-5 MMU Traps in Physical Mode

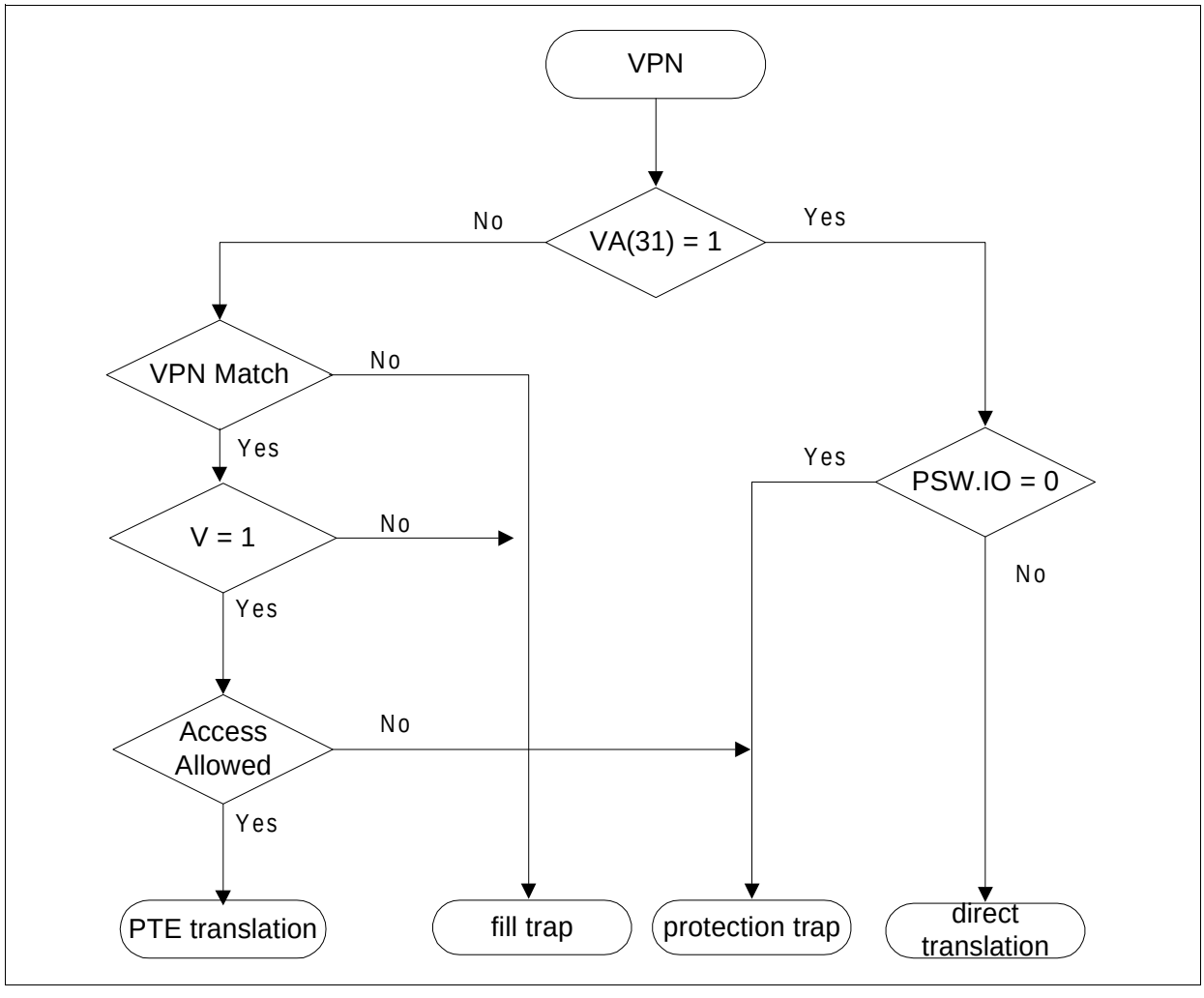


Figure 10-6 MMU Traps in Virtual Mode

10.1.7 MMU Instructions

All MMU instructions are privileged non-faulting instructions that require PSW.IO = 2 (Supervisor mode) for execution.

10.1.7.1 MAP Command (TLB Map)

The tlbmap instruction is used to install a mapping in the MMU. The tlbmap instruction takes an extended data register (Ea) as a parameter. The even Ea register contains the virtual address for the translation; the odd Ea register contains the page attributes and PPN. The least significant log2(PageSize) bits of the virtual address are ignored by the tlbmap instruction. The page attributes are contained in the Most Significant Byte of the odd register with the format as shown. The 24 Least Significant Bits of the odd register contain the PPN which is padded on the left with two zeros and on the right with log2(PageSize) – 10 zeros. The ASI for the translation is obtained from the ASI register.

31	30	29	28	27	26	25	24
V	**XE**	**WE**	**RE**	**G**	**C**	**PSZ**	

Installing a mapping for a virtual address for which a mapping already exists in the MMU results in an undefined operation. Installing a mapping for a page size which is not one of the two valid page sizes for either of the two TLBs results in a NOP. Installing a mapping when the two TLBs have identical page size settings results in the mapping being installed in one of the two TLBs.

The hardware replacement algorithm for the tlbmap instruction allows any set of four VPNs specified by VPN_a, VPN_{a+1}, VPN_b, and VPN_c where VPN_a and VPN_b are arbitrary and VPN_c belongs to the set $\{VPN_{b+j}\ j = 1, ..., 63\}$ to be co-resident in the MMU at any time. This guarantees that all the VPNs required for executing any single instruction can be co-resident in the MMU.

10.1.7.2 DEMAP Command (TLB Demap)

The tlbdemap instruction is used to uninstall a mapping in the MMU. The tlbdemap instruction takes a data register (Da) as a parameter. The Da register contains a virtual address for the demap operation as shown. The address space identifier for the demap operation is obtained from the ASI register. Demapping a translation that does not exist in the MMU results in a NOP.

31	0
Virtual Address	

10.1.7.3 FLUSH Command (TLB Flush)

The tlbflush instruction is used to flush mappings from the MMU. There are two variants of the tlb-flush instruction. The tlbflush.a instruction flushes all the mappings from TLB-A while the tlb-flush.b instruction flushes all the mappings from TLB-B.

10.1.7.4 PROBE Command (TLB Probe)

The tlbprobe instruction has two variants: tlbprobe.a and tlbprobe.i.

The tlbprobe.a (TLB Probe Address) instruction takes a data register (Da) as a parameter and is used to probe the MMU for a virtual address. The Da register contains the virtual address for the probe. The address space identifier for the probe is obtained from the ASI register.

The tlbprobe.i (TLB Probe Index) instruction takes a data register (Da) as a parameter and is used to probe the TLB at a given index. The Da register contains the index for the probe.

The tlbprobe instruction returns the ASI and VPN of the translation in the Translation Virtual Ad-dress register (TVA), the PPN and attributes in the Translation Physical Address register (TPA), and

the TLB index of the translation in the Translation Page Index register (TPX). The TPA.V bit is set to 0 if the TTE contained an invalid translation or an invalid index was used for the probe.

10.2 MMU Registers

All MMU Special Function Registers are memory-mapped. All registers can be read using the MFCR instruction. The MMU_CON and MMU_ASI registers are the only software-writable registers. The MMU_CON and MMU_ASI registers are written using the MTCR instruction. The registers implemented in the MMU are shown in **Figure 10-7** and **Table 10-1**. The registers and their bits are described in the following sections.

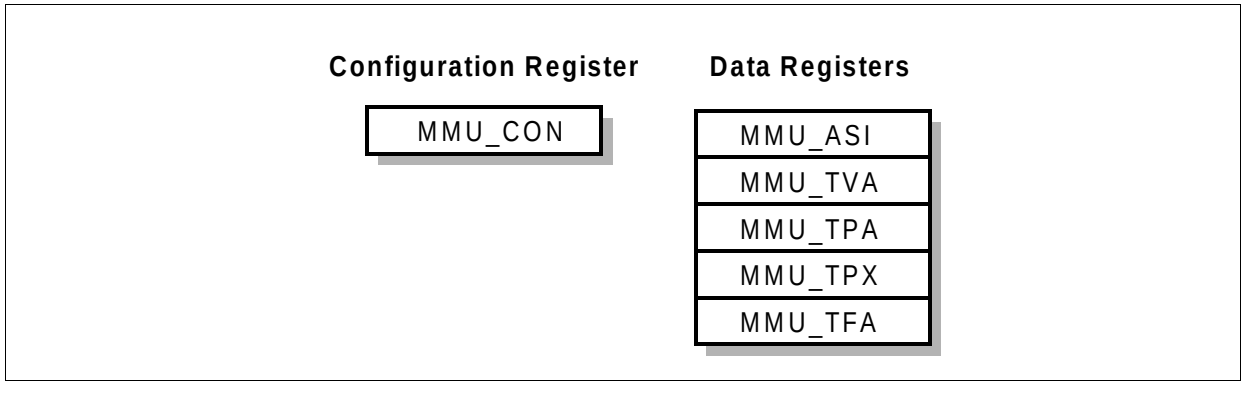


Figure 10-7 MMU Registers

Table 10-1 MMU Registers

Register Short Name	Register Long Name	Offset Address	Description see
MMU_CON	MMU Configuration Register	0000_H	Page 10-12
MMU_ASI	MMU Address Space Identifier Register	0004_H	Page 10-13
MMU_TVA	MMU Translation Virtual Address Register	$000C_H$	Page 10-14
MMU_TPA	MMU Translation Physical Address Register	0010_H	Page 10-15
MMU_TPX	MMU Translation Page Index Register	0014_H	Page 10-16
MMU_TFA	MMU Translation Fault Address Register	0018_H	Page 10-17

In the TC11IB, the registers of the MMU are located in the following address range:

– Module Base Address. F7E1 8000_H
  Module End Address. F7E1 $80FF_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 10-1**)

10.2.1 Configuration Register

MMU_CON
MMU Configuration Register **Reset Value: 0000 $07E0_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

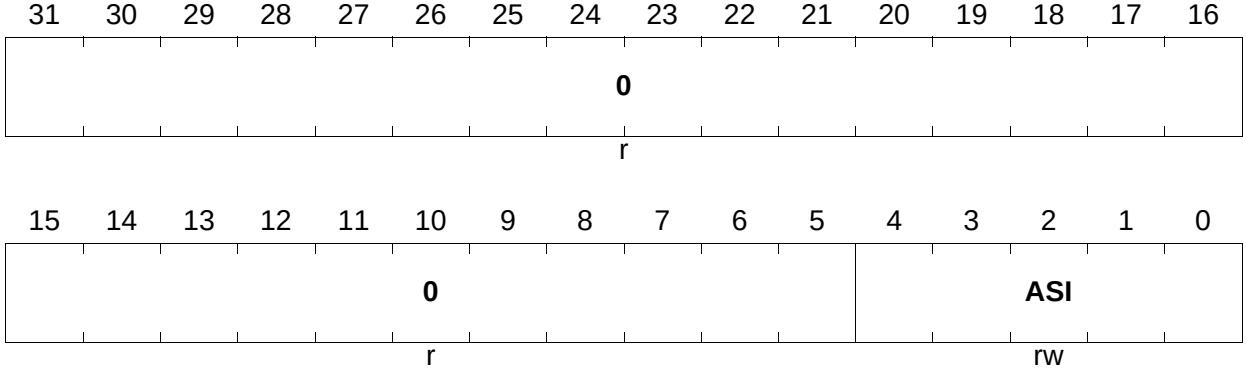
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0				TSZ							SZB		SZA		V
r				r							rw		rw		rw

Field	Bits	Type	Description
V	0	rw	**Virtual Mode Control** 0 The processor is in Physical Mode 1 The Processor is in Virtual Mode
SZA	[2:1]	rw	**Page Size A Control** The page size of the mappings in TLB-A: 00 1 KB 01 4 KB 10 16 KB 11 64 KB
SZB	[4:3]	rw	**Page Size B Control** The page size of the mappings in TLB-B: 00 1 KB 01 4 KB 10 16 KB 11 64 KB
TSZ	[11:5]	r	**TLB Size Control** The entries of TLB-A are indexed 0 through TSZ The entries of TLB-B are indexed 128 through 128+TSZ
0	[31:12]	r	**reserved**

10.2.2 Address Space Identifier Register

The Address Space Identifier (ASI) register contains the address space identifier of the current process.

MMU_ASI
MMU Address Space Identifier Register **Reset Value: 0000 001F$_H$**

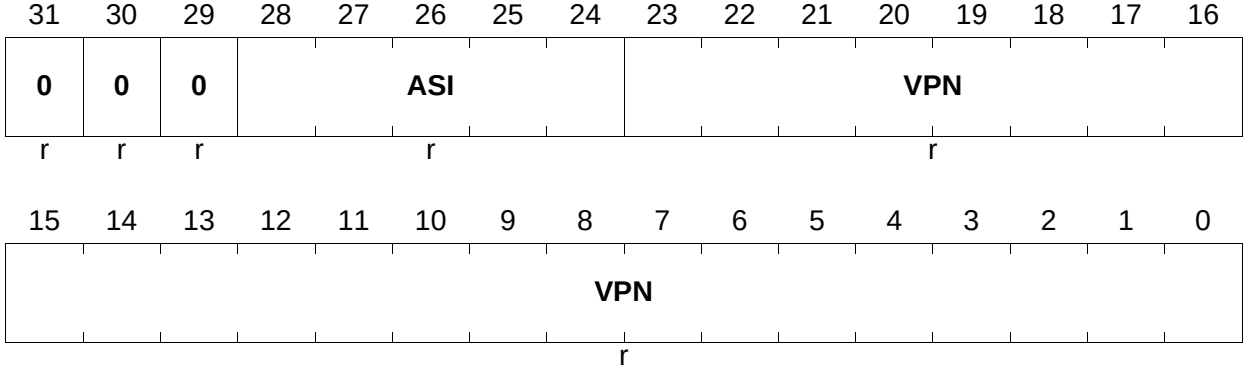


Field	Bits	Type	Description
ASI	[4:0]	rw	**Address Space Identifier** The Address Space Identifier of the current process.
0	[31:5]	r	**reserved**

10.2.3 Translation Virtual Address Register

Translation Virtual Address register (TVA) is used to return the ASI and VPN of a translation by a tlbprobe instruction.

MMU_TVA
MMU Translation Virtual Address Register **Reset Value: 0000 0000$_H$**

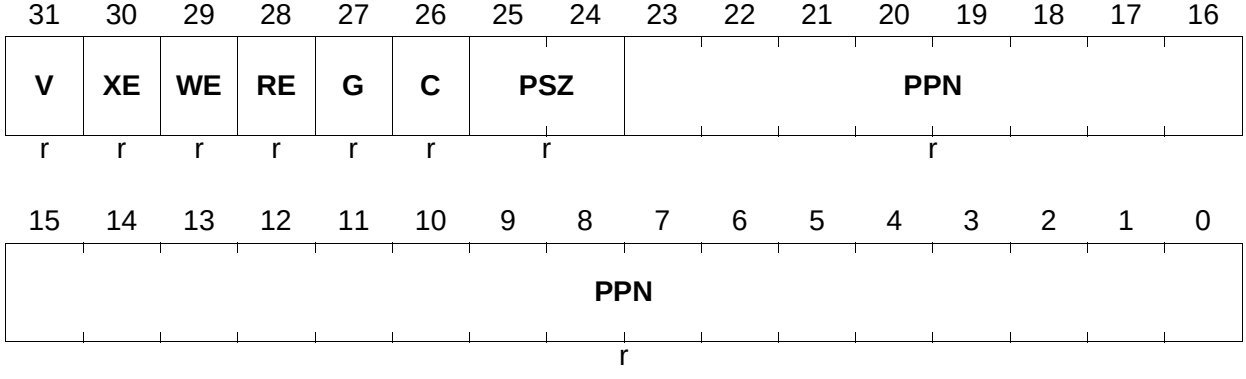


Field	Bits	Type	Description
VPN	[23:0]	r	**Virtual Page Number**
ASI	[28:24]	r	**Address Space Identifier**
0	[31:12]	r	**reserved**

10.2.4 Translation Physical Address Register

The Translation Physical Address register (TPA) is used to return the PPN and attributes of a translation by a tlbprobe instruction.

MMU_TPA
MMU Translation Physical Address Register **Reset Value: 0000 0000$_H$**



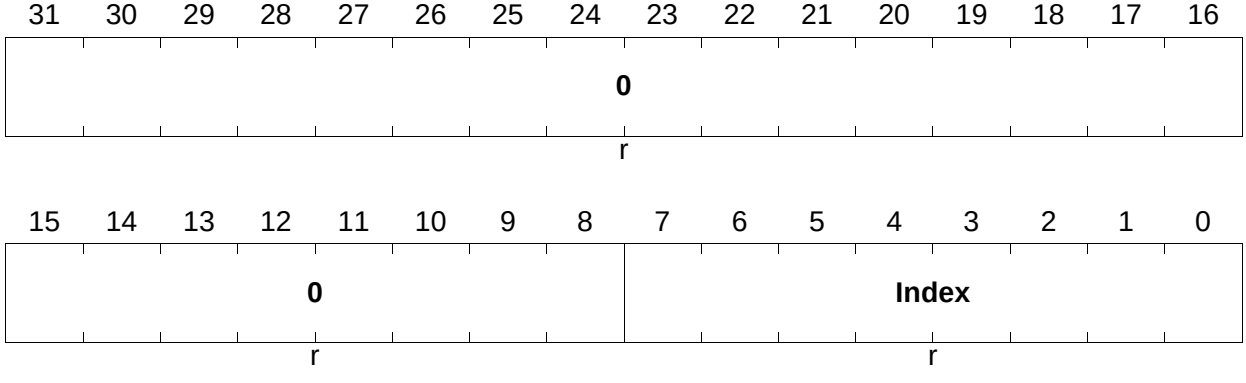
Field	Bits	Type	Description
PPN	[23:0]	r	**Physical Page Number**
PSZ	[25:24]	r	**Page Size Control** 00 1 KB 01 4 KB 10 16 KB 11 64 KB
C	26	r	**Cacheability Control** 0 The page is non-cacheable 1 The page is cacheable
G	27	r	**Global Control** 0 The page is not globally mapped 1 The page is globally mapped, making it visible in all address spaces
RE	28	r	**Read Enable Control** 0 Disable data read from the page 1 Enable data read from the page

Field	Bits	Type	Description
WE	29	r	**Write Enable Control** 0 Disable data written to the page 1 Enable data written to the page
XE	30	r	**Execute Enable Control** 0 Disable instruction fetch to the page 1 Enable instruction fetch to the page
V	31	r	**Valid Control** 0 Invalid mapping 1 Valid mapping

10.2.5 Translation Page Index Register

The Translation Page Index register (TPX) is used to return the TLB index of a translation by a tlbprobe instruction.

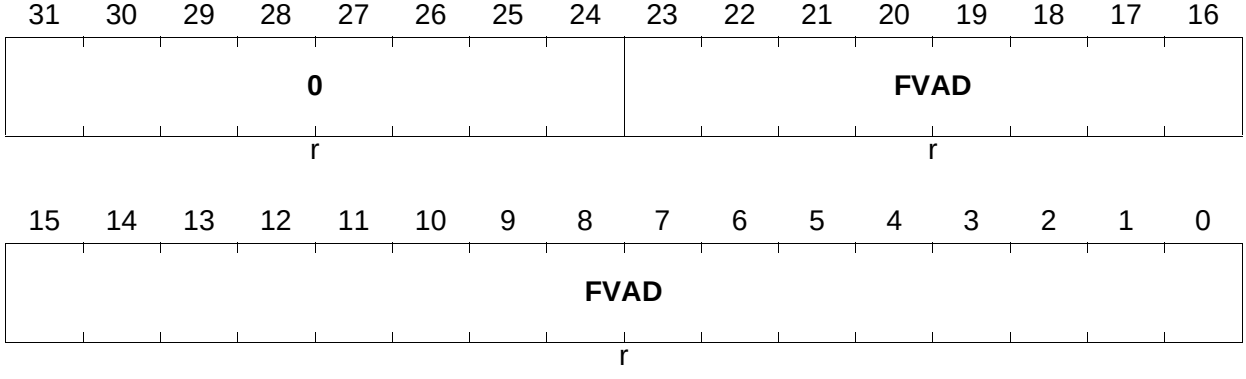
MMU_TPX
MMU Translation Page Index Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
Index	[7:0]	r	**TLB Index**
0	[31:8]	r	**reserved**

10.2.6 Translation Fault Address Register

MMU_TFA
MMU Translation Fault Address Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
FVAD	[23:0]	r	**Faulting Virtual Address** Faulting virtual Address shifted by 10 + 2 * min (SZA, SZB) bits.
0	[31:24]	r	**reserved**

11 On-Chip Local Memories

The TC11IB contains three on-chip local memories.

- 512-KB eDRAM Local Memory Unit connected to Local Memory Bus (LMB).
- 1-MB ComDRAM connected to Fast Flexible Peripheral Bus Interface (F_FPI).
- 16-KB Boot ROM connected to Slow Flexible Peripheral Bus Interface (S_FPI).

11.1 Local Memory Unit

The Local Memory Unit (LMU) contains a 512-KB eDRAM, the interface to the LMB bus and corresponding control logic. It provides read, write, and refresh cycles to eDRAM and some data management facilities, for example, a program memory prefetch facility. It supports many kinds of refresh, the rates of which are programmable to suit the statistics of the accesses, clock speed, and temperature. For safety and robustness, the LMU goes directly into a refresh cycle immediately after a reset. Additionally, the prefetch facilities may be set to paged only or paged and random mode using the control registers. The paged accesses are performed as this requires fewer LMB data cycles and uses less power to complete the access. Generally, the LMU is flexible due to its programmability and configuration. The LMU consists of the functional blocks as shown in **Figure 11-1**:

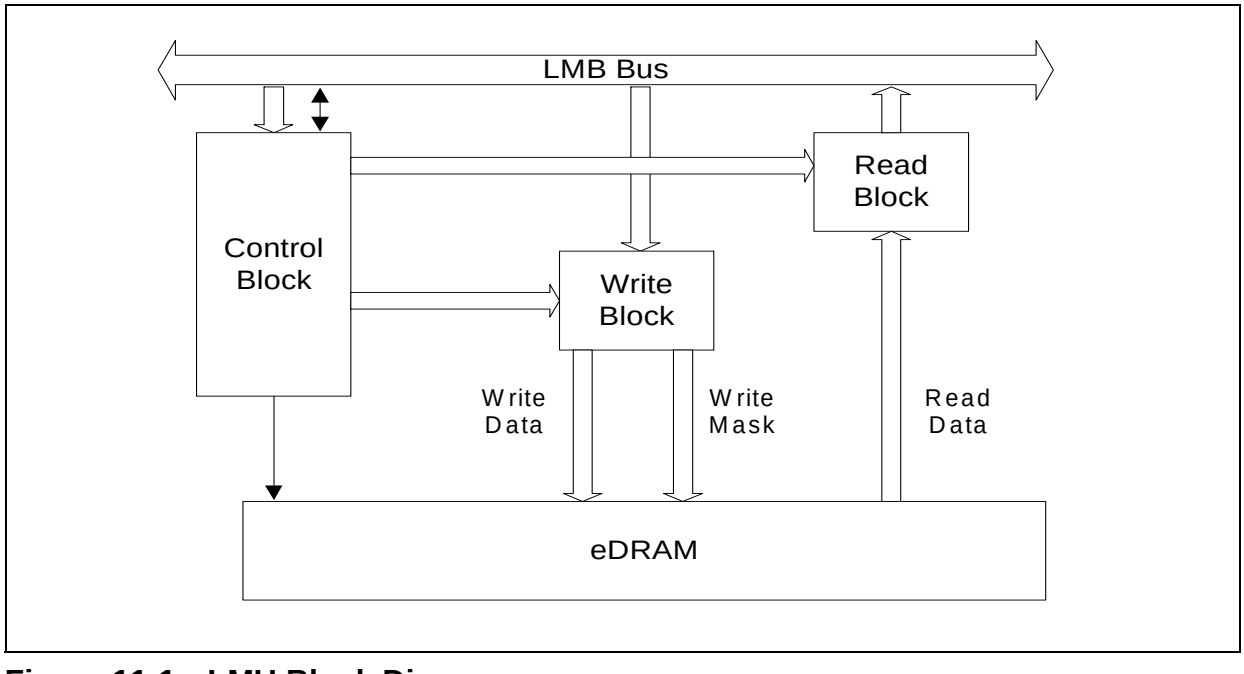


Figure 11-1 LMU Block Diagram

Features

- 512-KB eDRAM
- 256-bit wide Read/Write Data Bus
- Supports all Burst Mode Accesses
- Access Prefetch Functionality
- Supports Random and Paged Accesses
- Provides eDRAM Refresh Cycle
- Forced and Refresh Request Modes
- Programmable/Configurable Forced Refresh and Request Times
- Synchronous WatchDog Timer Reset

11.1.1 eDRAM Overview

The embedded eDRAM has the following features:

- 256 bit wide write data bus and 256 bit wide write mask
- 256 bit wide read data bus
- Random Access Time is 13 ns
- Paged Access Time is 6.6 ns
- Page Width is 8*256 bits

11.1.2 eDRAM Address Map

Table 11-1 shows the memory map of LMU eDRAM.

Table 11-1 LMU eDRAM Address Map

Address	Name	Size
0xF800 0400-0xF800 07FF	LMU eDRAM control register space	1 KB
0xE800 0000-0xE83F FFFF	Mapped LMU eDRAM space seen from FPI	4 MB
0xC000 0000-0xC007 FFFF	Regular LMU eDRAM space	512 KB
0xAFC0 0000-0xAFC7 FFFF	Non-Cached LMU eDRAM space	512 KB

11.1.3 LMU Operation Overview

The LMU supports both Local Memory Bus (LMB) access and non-LMB access. Besides control registers, access and error transaction, there are three main types of access starts initiated by the LMB:

- Access from scratch register or prefetch register (read only)
- Random access
- Paged access

There are two kinds of non-LMB access flows:

- Refresh flow

- Prefetch flow

For LMB access, because LMB is pipelined, which includes three pipeline phases: Arbitration Phase (one cycle), Address Phase (one cycle) and Data Phase (minimum one cycle), longer bursts have proportionally less overhead, more efficient. Therefore, it is important that the LMU supports all burst modes, especially the larger one.

Because the LMU address is a byte address, the LMB address has the following relevance:

- The LMB address (32 down to 0) are relevant to SDTB
- The LMB address (32 down to 1) are relevant to SDTH
- The LMB address (32 down to 2) are relevant to SDTW
- The LMB address (32 down to 3) are relevant to SDTD, BTR2, BTR4 and BTR8

The LMB address (4 down to 3) corresponds to the SDTD, BTR2, BTR4 and BTR8 start point position in the eDRAM 256-bit wide bus. When these bits are “00”, the access starts at the beginning of the bus. When these bits are “01”,”10” and “11”, the access starts in positions 2, 3 and 4 respectively, as shown in **Figure 11-2**.

In the case of BTR2 or BTR4, when the access comes to the end of the memory data bus, subsequent blocks of 64 start again at the beginning of the same bus, i.e. only one memory access occurs. For a BTR8, the same is true except that two bus widths are accessed at the same time, i.e. only two memory accesses occur.

Note: A burst access never crosses a page boundary, i.e. for a BTR8 access, cycle 2 is always paged. The LMU returns an error transaction if the address is inconsistent with the transaction size. If the two accesses of a BTR8 do not fit inside a single page, an error is issued also.

- The LMB address (7 down to 5) corresponds to the eDRAM address within current page.

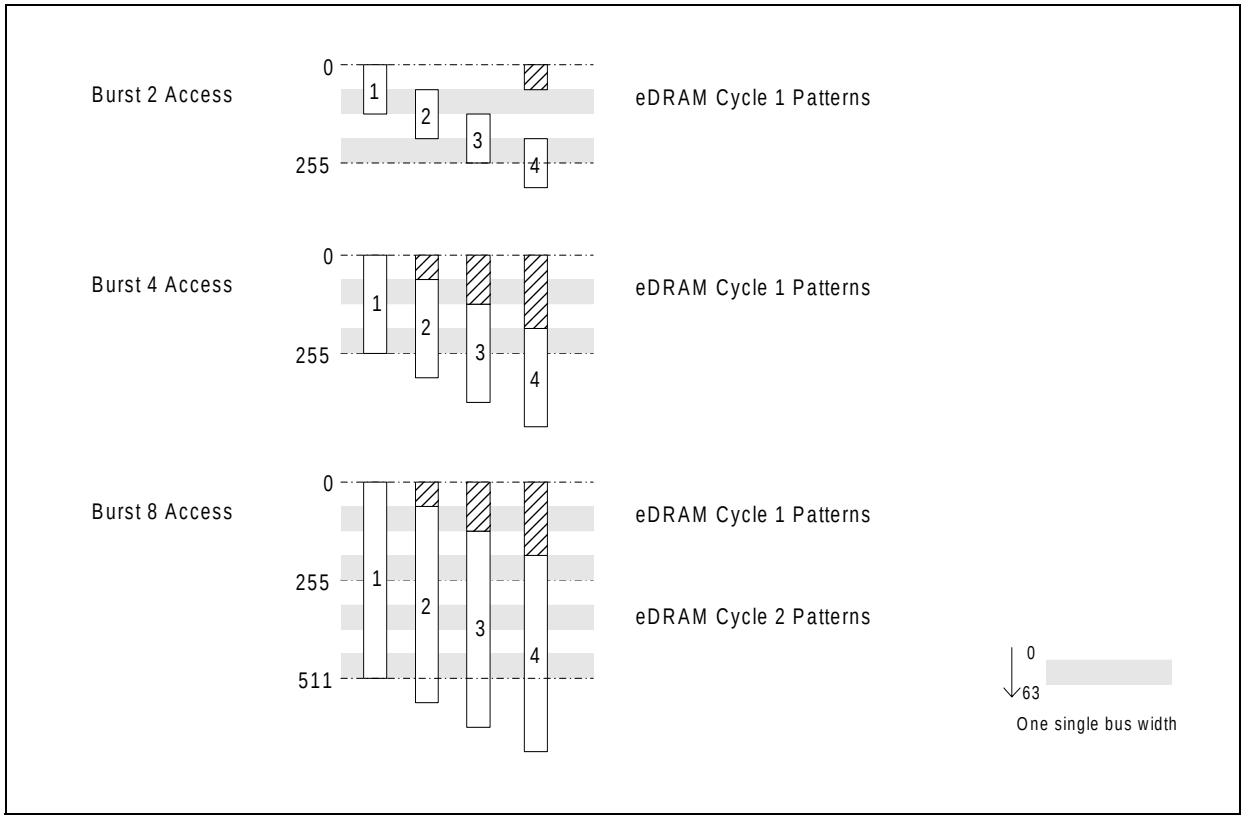


Figure 11-2 Access Patterns And Cycles

The bytes, words and double words are aligned to the LMB read bus boundary in the same way as they reside in the memory address space. The LMU uses the same flow for SDTW,SDTH,SDTB and SDTS read access. **Table 11-2** lists the data access alignment.

Table 11-2 Access Alignment vs. LMB Data Bus (64-bit wide)

LMB address (2-0)	111	110	101	100	011	010	001	000
SDTB	Byte 7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0
SDTH	Half Word 3		Half Word 2		Half Word 1		Half Word 0	
SDTW	Word 1				Word0			
SDTD	Double Word 0							

Table 11-3 lists the LMU functions and their required data phase cycles.

Table 11-3 LMU Functions

Function	Function Description	Cycles	Remark
Error	Indicate an illegal access request has been made	2	
Refresh	Perform a refresh cycle	3	After the refresh, the next cycle must be 7 ns away.
Control Register Access	Read/Write to a control register	2	
Prefetch	Load prefetch register from random/paged access	3	
Register Read1	Read SDTD, SDTW, SDTH, SDTB access from scratch or prefetch registers	2	
Register Read2	Read BTR2 from scratch or prefetch registers	3	
Register Read4	Read BTR4 from scratch or prefetch registers	5	
Write1	Write starting random/paged SDTD, SDTW, SDTH, SDTB	3	
Write2	Write starting random/paged BTR2	4	
Write4	Write starting random BTR4	6	This access contains an integral refresh.
Write8	Write starting random BTR8	10	This access contains an integral refresh.
Wrapped Write8	Write starting random BTR8 with data wrapping	12	This access contains an integral refresh.
Random Read1	Read starting random SDTD, SDTW, SDTH, SDTB	4	
Paged Read1	Read starting paged SDTD, SDTW, SDTH, SDTB	3	
Random Read2	Read starting random BTR2	5	
Paged Read2	Read starting paged BTR2	4	

Table 11-3 LMU Functions

Function	Function Description	Cycles	Remark
Random Read4	Read starting random BTR4	7	
Paged Read4	Read starting paged BTR4	6	
Random Read8	Read starting random BTR8	11	
Paged Read8	Read starting paged BTR8	10	
Wrapped Read8	Read starting random/paged BTR8 with data wrap	12	

Note: A wrapped BTR8 access data from the second eDRAM access is always the first to be placed on the LMB data, therefore wrapped BTR8 accesses are longer than non-wrapped BTR8 access.

11.1.3.1 LMB Slot Condition

The LMB Slot Condition (only one condition exists for all operations) may be set for:

- Refreshes
- Random prefetch requests
- Paged prefetch requests

The normal request mechanism for either the refresh or the prefetch detects that there is no current access in the address phase for the LMU, before commencing the flow. This means that should flow be commenced, and an access appears on the bus, the access will have to be detained until the current flow is completed.

For the slot condition, the LMU is set to recognize accesses of a specified size to another ‘slave’. The LMU then performs the required operation (i.e. refresh or prefetch) while the bus is otherwise occupied. That is, if the LMU is set to detect that another slave is doing a BRT4, BRT8 access, it is known that even with zero wait state transactions, it has sufficient time to do either a refresh of prefetch random or paged flow. This ensures that these operations will then be completely hidden, and that no access collision will occur.

Note: The slotted condition may be set to a size less than the number of cycles required to perform operation(s) merely to gain a statistical advantage.

11.1.3.2 Read Data Scratch Registers

The LMU has two scratch registers, one for the PMU read data and one for non PMU read data. Each time, there is a read access to the eDRAM, the LMU registers the eDRAM read data into scratch register for possible reuse. The scratch registers are the registered value of the entire last read access, i.e. the registered data is 256-bit wide. If

a read address coincides with a scratch register address, then no eDRAM access is performed and the data is selected from the active scratch register. This reduces the number of cycles required to complete the access and save power.

The scratch registers and prefetch register are read by means of one of three Register Read functions.

11.1.3.3 Prefetch Mechanism

Program data can be loaded in advance during a free controller timeslot. It is assumed that the next access follows sequentially from the previous access. As well as registering the data from the last access, the LMU can speculatively load data from the next address into the PMU prefetch register for possible use. The prefetch operates by means of a prefetch request mechanism.

Prefetch Request

The LMU generates a prefetch request if the PMU (read only) makes a SDTD, BTR2, BTR4 last access in the eDRAM data bus.

The prefetch request is killed when the access that follows does not have the above PMU condition, if the prefetch has been serviced, or if an abort occurs.

The prefetch mode of operation may be set to random or paged for both a slotted bus and normal bus condition, by means of the mode register. Requests are serviced when there is a single free timeslot detected on the bus, or when a slotted bus condition is detected.

Note: Care must be taken when setting the prefetch mode for the optimum performance. If another master is frequently accessing while the PMU is making accesses, constant use of a random PMU prefetch in between may hinder the other master by constantly closing the page.

11.1.3.4 Refresh Modes of Operation

There are four LMU refresh types:

- Forced refresh
- Request refresh
- Slotted refresh request
- Integral refresh

The refresh request rates may be set by means of the REFRATE register. The timing of the refreshes is generated in the LMU Control Block. All LMU refreshes have the same effect on the eDRAM when performed. A refresh may collide with an LMB access request. The LMU will handle such a collision by sampling the access in the address phase and inserting wait states to the access’s first data phase. The LMU commences the sampled access after the refresh or prefetch has been completed.

Forced Refresh

The forced refresh can maintain the integrity of the memory if another kind of refresh has not been preformed within the maximum time between refreshes.

The forced refresh takes precedence over any other actions, i.e. memory accesses.

Request Refresh

This type of refresh is a request, and may be fit into the next available timeslot; thus, not filling the timeslot for an access. This can help to hide the refresh action.

Note: A free timeslot detected may be only one cycle long, and the refresh takes 3 cycles. i.e. this refresh mode may block accesses to the LMU for up to two cycles.

Note: Setting this value low (i.e. generate many requests) will tend to increase the number of refreshes, and hence increase the power consumption of the memory, and maybe cause detrimental blocking of the LMU.

Slotted Refresh

This refresh aims to avoid blocking of the LMU operation. The slotted refresh looks for non-LMU accesses on the LMB and attempts to make refreshes while these are in progress.

Note: If correctly configured, this may be set high or permanently on without blocking the LMU.

Note: If setting permanently and refresh has priority over prefetch, the prefetch request will not be serviced. It is suggested that setting this permanently and giving prefetch priority may give good results.

Integral Refresh

Some flows contain sufficient timeslots to contain refreshes. These flows include:

- Write starting random BTR2
- Write starting random BTR8
- Write starting random BTR8 with data wrapping

When any kind of refresh is performed, the refresh counter is reset.

Note: The refresh happens as early as possible in the flow in order to reduce the ‘refresh margin’.

Note: All flows that contain a refresh should reopen the current page

Note: Read BTR8 and above (smaller reads do not have sufficiently long timeslots) do not contain refreshes as this requires extra registering of the read data, and this slows the LMU’s operation.

11.1.3.5 Error

An error indicates an illegal access request has been made. The following reasons may result in the LMU error:

- Access is in the control, the eDRAM or eDRAM map empty regions
- Address is not consistent with the access size
- Address is such that the second access of a BTR8 access crosses a page boundary
- Non supervisor control write
- Non 32-bit control write
- Write to a read only control register
- Control read BRT2, BRT4 or BRT8
- BRT2, BRT4, or BRT8 read modify write, (read phase only)

No access occurs to any source if an error sequence occurs.

11.1.3.6 LMU Reset

The LMU has two types of reset available: asynchronous and synchronous. Both types reset the refresh counter to its maximum count; i.e., the LMU always goes into a refresh immediately after a reset.

Asynchronous Reset

The LMU has an asynchronous reset signal that resets all the synchronous cells in the LMU. During reset, the LMU is completely inoperable. It cannot accept or respond to an access or refresh.

*Note: The details about eDRAM Power On Reset (POR) are described in **Section 5.3.5**. The reset signal must be driven such that the required timings are satisfied.*

Synchronous Reset

The LMU has a synchronous reset in order that it my be issued a reset at any time and still perform a predetermined and safe action.

The synchronous reset is taken into account by the LMU either when the eDRAM is in the idle state, in the last refresh state, or in the last data phase of the current transaction. This ensures that the eDRAM control signal timings are respected, and that the LMU may be interrupted if no idle condition occurs on the bus; i.e., there are continuous accesses.

When the synchronous reset coincides with an idle, end of a refresh, or last data phase, the next state is the start of a refresh cycle. If, after a refresh, the synchronous reset signal is still asserted, the eDRAM will again be reset to the start of a refresh cycle and another refresh will occur; i.e., refreshes happen continuously while the synchronous reset signal is asserted.

The value in the refresh counter during a synchronous reset is irrelevant as the LMU is refreshing irrespective of its value. After the reset is deasserted, the LMU will issue a refresh done signal that resets the counter to zero, and the counter will begin to count on the following LMB clock pulse. This has the effect of synchronizing the refresh counter with the time between refreshes.

This synchronous reset signal must be asserted for a period equal to the longest transaction that the LMU may be asked to perform, for example, BTR8 access.

Note: If it could be assumed that the LMU will be in idle when a synchronous reset is issued, then the reset signal need only be active for one clock cycle. The problem with this assumption is that a refresh may occur, thus a minimum of 4 clocks would be required in this case.

Note: Worst Case Minimum Synchronous Reset Time (for BTR8 system) = 12 cycles.

11.1.4 LMU Registers

As shown in **Figure 11-3**, the following control registers are implemented in the LMU. These registers and their bits are described in this section

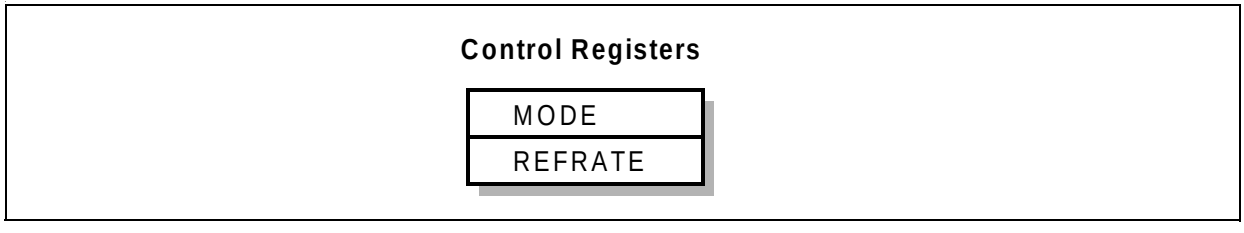


Figure 11-3 LMU Registers

Table 11-4 LMU Registers

Register Short Name	Register Long Name	Offset Address	Description see
LMU_MODE	LMU Mode Register	0000_H	**Page 11-11**
LMU_REFRATE	LMU Refresh Rate Register	0008_H	**Page 11-12**

Note: Write access to these registers must be performed in supervisor mode and SDTW (32-bit) write only.

Note: Each of these control registers has a unique LMB address for a 64-bit access. The registers are aligned so that the lower 32 bits of the LMB read data contain the read information. The upper 32 bits will always read as 0. The Read SDTD,SDTW,SDTH, and SDTB are allowed.

Note: Read Modify Write Access should be SDTW access. The LMB address [1:0] = 00_B for a read modify write access.

In the TC11IB, the registers of the LMU are located in the following address range:

– Module Base Address. F800 0400$_H$
  Module End Address. F800 04FF$_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 11-4**)

11.1.4.1 LMU MODE Register

LMU_MODE
LMU Mode Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0									BTR8	BTR4	BTR2	SDTD	SDTW	SDTH	SDTB
r									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								CLKGAT	PRESLP	PRESL	PREONP	PREON	REFPRI	0	
r								rw	rw	rw	rw	rw	rw	r	

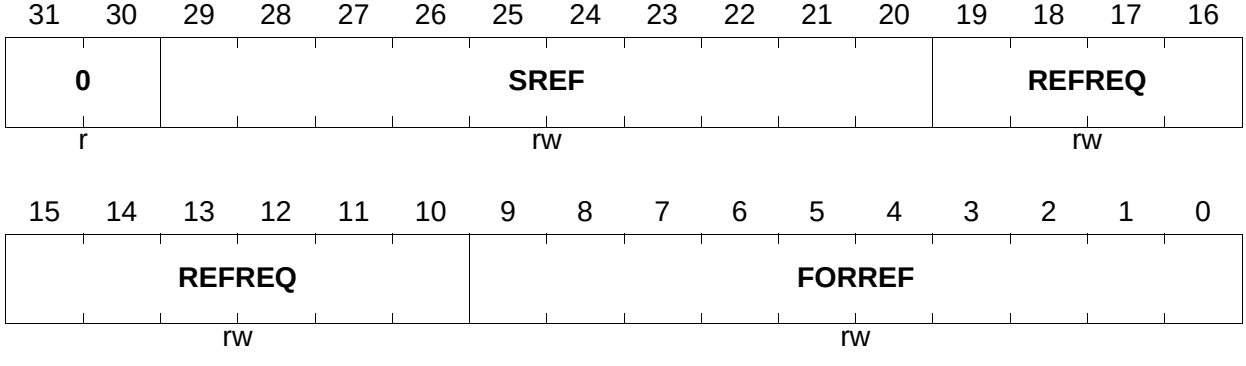
Field	Bits	Type	Description
REFPRI	2	rw	If 1,refreshes have priority over prefetch mechanism. If 0, prefetch has priority.
PREON	3	rw	If 1, LMU is able to service random prefetch requests.
PREONP	4	rw	If 1, LMU is able to service paged prefetch requests.
PRESL	5	rw	If 1, LMU is enabled to service random prefetch when slotted condition is satisfied.
PRESLP	6	rw	If 1, LMU is enabled to service paged prefetch when slotted condition is satisfied.
CLKGAT	7	rw	Gate for clock to Prefetch & Scratch registers. When 0, no register read possible.
SDTB	16	rw	If 1, LMU is enabled to make slotted refresh/prefetch for STDB transfers
SDTH	17	rw	If 1, LMU is enabled to make slotted refresh/prefetch for STDH transfers
SDTW	18	rw	If 1, LMU is enabled to make slotted refresh/prefetch for STDW transfers

Field	Bits	Type	Description
SDTD	19	rw	If 1, LMU is enabled to make slotted refresh/prefetch for STDD transfers
BTR2	20	rw	If 1, LMU is enabled to make slotted refresh/prefetch for BTR2 transfers
BTR4	21	rw	If 1, LMU is enabled to make slotted refresh/prefetch for BTR4 transfers
BTR8	22	rw	If 1, LMU is enabled to make slotted refresh/prefetch for BTR8 transfers
0	[1:0] [15:8] [31:23]	r	**Reserved**; read as 0; should be written with 0.

11.1.4.2 REFRATE Register

The LMU has an 8-bit address field for the refresh request and an 8-bit address for the forced refresh. When the divide value on the refresh request register is larger, only forced refreshes occur.

LMU_REFRATE
LMU Refresh Rate Register **Reset Value: 0140 5014$_H$**



Field	Bits	Type	Description
FORREF	[9:0]	rw	**Forced Refresh** The number of clocks of forced refresh
REFREQ	[19:10]	rw	**Refresh Request** The number of clocks of refresh request
SREF	[29:20]	rw	**Slotted Refresh** The number of clocks of slotted refresh
0	[31:30]	r	**reserved**

Note: The refresh cycle takes 3 clocks. Therefore, a forced refresh of 3 clocks or fewer will completely block out any subsequent accesses. In order to protect against bad values being written to the REFRATE forced refresh field, LMB metastability, noise etc., if any of {3,2,1,0} is detected in the REFRATE register, the forced refresh reset value will be loaded on the subsequent clock.

11.2 ComDRAM

Another 1-MB eDRAM of ComDRAM is implemented in TC11IB and connected on Fast FPI Bus. The embedded eDRAM has the similar features as the one in LMU:

- 256 bit wide write data bus and 256 bit wide write mask
- 256 bit wide read data bus

By default, ComDRAM is disabled. Before it is used, the clock to ComDRAM must first be initialized. On the other hand, ComDRAM registers are affected by OCDS Suspend Mode (refer to **Section 21.3.1**). If OCDS Suspend Mode is active, all writes to ComDRAM registers are ignored. The S_FPI acknowledges the write access, but the write is not executed internally. Read access are not affected. ComDRAM can be reset via software.

The address range of ComDRAM is: BFE0 0000_H - BFEF $FFFF_H$

11.2.1 ComDRAM Registers

As shown in **Figure 11-4**, the following control registers are implemented in the ComDRAM. These registers and their bits are described in this section

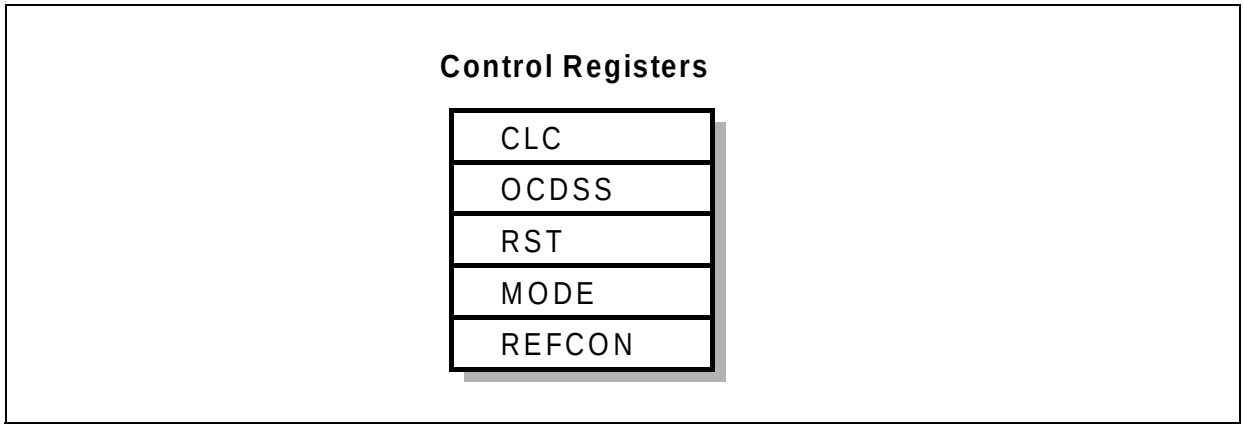


Figure 11-4 ComDRAM Registers

Table 11-5 ComDRAM Registers

Register Short Name	Register Long Name	Offset Address	Description see
ComDRAM_CLC	ComDRAM Clock Register	0000_H	Page 11-14
ComDRAM_OCDSS	ComDRAm OCDS Suspend Register	0004_H	Page 11-15
ComDRAM_RST	ComDRAM Reset Register	0040_H	Page 11-16
ComDRAM_REFCON	ComDRAM Refresh Control Register	$01A0_H$	Page 11-17
ComDRAM_MODE	ComDRAM Mode Register	$01A4_H$	Page 11-17

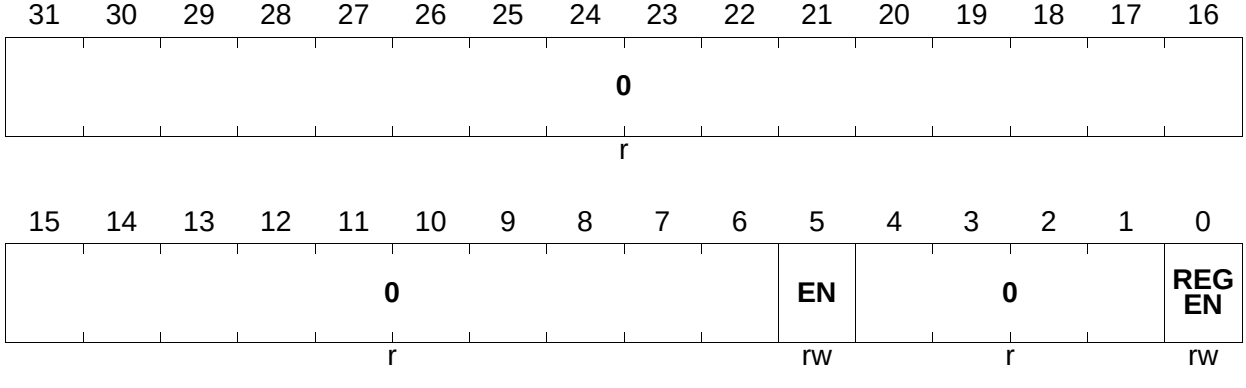
Note: Write access to these registers must be performed in supervisor mode and ComDRAM_CLC is EndInit protected.

In the TC11IB, the registers of the ComDRAM are located in the following address range:

– Module Base Address. F018 0000_H
  Module End Address. F018 $FFFF_H$
– Absolute Register Address = Module Base Address + Offset Address (offset addresses see **Table 11-5**)

11.2.1.1 ComDRAM Clock Register

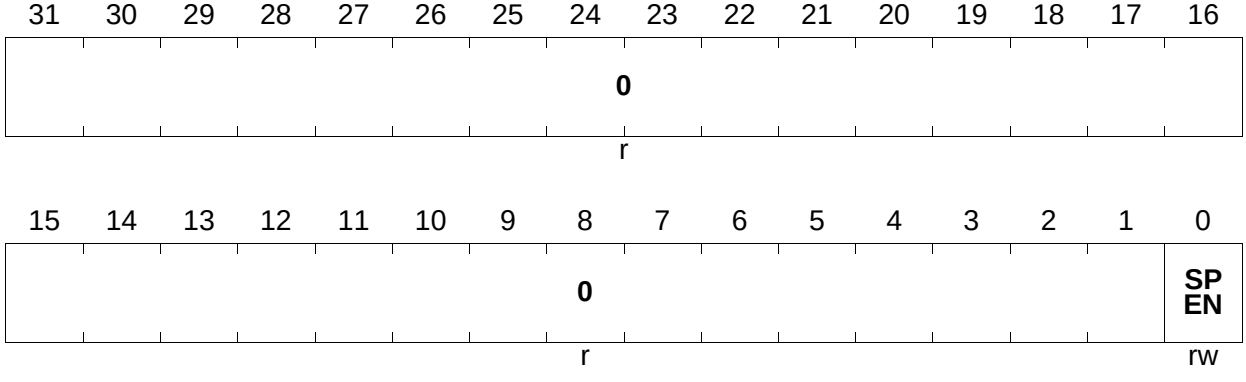
ComDRAM_CLC
ComDRAM Clock Register **Reset Value: 0000 0000_H**



Field	Bits	Type	Description
REGEN	0	rw	**Switch the clock of the ComDRAM register on S_FPI Bus** 0 Clock off 1 Clock on
EN	5	rw	**Switch the clock of the ComDRAM** 0 Clock off 1 Clock on
0	[4:1] [31:6]	r	**Reserved**; read as 0; should be written with 0.

11.2.1.2 ComDRAM OCDS Suspend Register

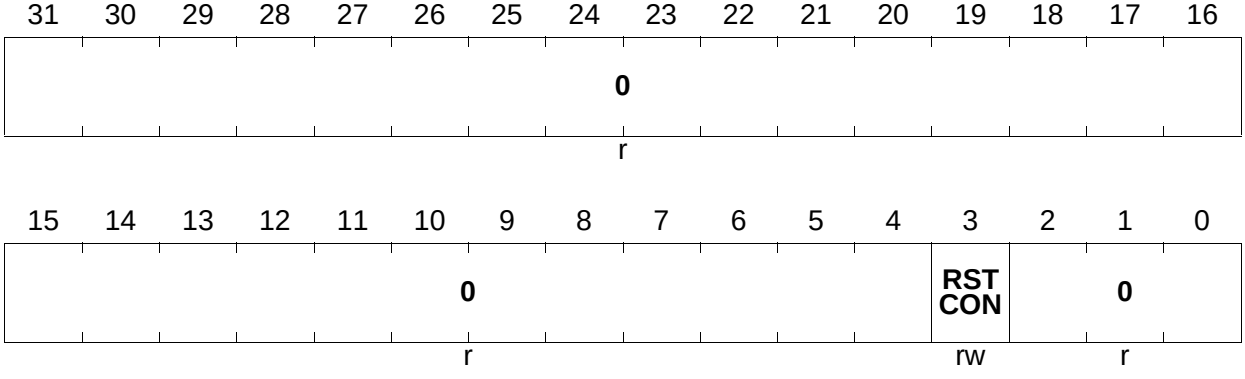
ComDRAM_OCDSS
ComDRAM OCDS Suspend Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
SPEN	0	rw	**Suspend Enable Bit for OCDS** 0 Suspend disabled 1 Suspend enabled
0	[31:1]	r	**Reserved**; read as 0; should be written with 0.

11.2.1.3 ComDRAM Reset Register

ComDRAM_RST
ComDRAM Reset Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
RSTCON	3	rw	**ComDRAM Reset Control** 0 No reset 1 Reset ComDRAM
0	[2:0] [31:4]	r	**Reserved**; read as 0; should be written with 0.

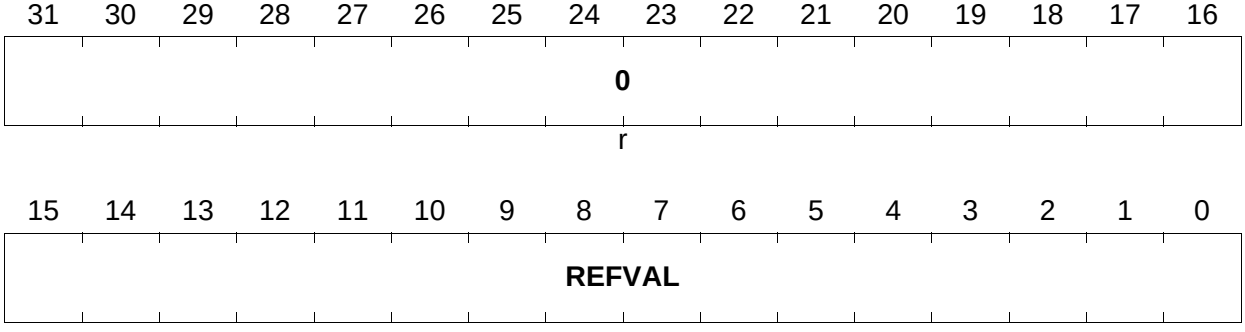
11.2.1.4 ComDRAM Refresh Register

The delay of the ComDRAM refresh cycles is controlled by the ComDRAM Refresh Register. The refresh rate can be gotten via the following formula:

REFVAL = (Refresh Time/Clock Period) - 1 [11.1]

For example, after reset the refresh rate is 400ns and clock frequency is 96MHz, the REFVAL is 25_H.

ComDRAM_REFCON
ComDRAM Refresh Control Register **Reset Value: 0000 0025$_H$**

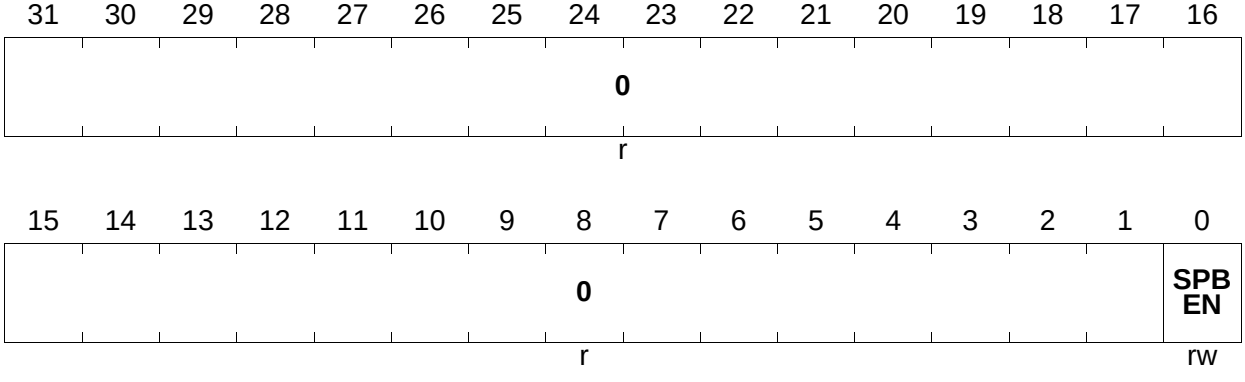


Field	Bits	Type	Description
REFVAL	[15:0]	rw	**Refresh Rate** The number of clocks of refresh.
0	[31:16]	r	**reserved**

11.2.1.5 ComDRAM MODE Register

MODE register enables the Scratchpad buffer of ComDRAM interface. This buffer saves the data of the last FPI access to ComDRAM. The access could be SDTB, SDTH, SDTW, BTR2 and BTR4. The next access to the ComDRAM will check whether the new data are already in Scratchpad buffer. If there are, then the access goes only to the buffer and does not go to the eDRAM. When a Write the address of data in Scratchpad buffer is executed or a FPI abort occurs, the Scratchpad buffer is flushed.

ComDRAM_MODE
COmDRAM Mode Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
SPBEN	0	rw	**ComDRAM Scratchpad Buffer Enable Control** 0 The Scratchpad Buffer is disabled 1 The Scratchpad Buffer is enabled
0	[31:1]	r	**Reserved**; read as 0; should be written with 0.

11.3 Boot ROM

The TC11IB contains 16-KB of Boot ROM memory, which can be used for:

- Device operating mode initialization routines
- Bootstrap loader support
- Test functions

11.3.1 Bootstrap Loader Support

An integrated bootstrap mechanism is provided to support a system start with boot operation after reset. If the boot mode is selected during reset, program execution is started from the Boot ROM. The functionality of the boot routine will be similar to the one implemented in other 16-Bit microcontrollers from Infineon.

Note: The bootstrap loader and the functionality of the boot routines will be described in detail in a separate document.

12 Memory Protection System

This chapter describes memory protection for the TC11IB. Topics covered include the architecture of the memory protection system and the memory protection registers.

12.1 Memory Protection Overview

The TC11IB memory protection system specifies the addressable range and read/write permissions of memory segments available to the currently executing task. The memory protection system controls the position and range of addressable segments in memory. It also controls the kinds of read and write operations allowed within addressable memory segments. Any illegal memory access is detected by the memory protection hardware, which then invokes the appropriate Trap Service Routine (TSR) to handle the error. Thus, the memory protection system protects critical system functions against both software and hardware errors. The memory protection hardware can also generate signals to the Debug Unit to facilitate tracing illegal memory accesses.

As shown in **Figure 12-1**, there are two Memory Protection Register Sets in the TC11IB, numbered 0 and 1, which specify memory protection ranges and permissions for code and data. The PSW.PRS bit field determines which of these is the set currently in use by the CPU. Because the TC11IB uses a Harvard-style memory architecture, each Memory Protection Register Set is broken down into a Data Protection Register Set and a Code Protection Register Set. Each Data Protection Register Set can specify up to four address ranges to receive particular protection modes. Each Code Protection Register Set can specify up to two address ranges to receive particular protection modes.

Each of the Data Protection Register Sets and Code Protection Register Sets determines the range and protection modes for a separate memory area. Each contains register pairs which determine the address range (the Data Segment Protection Registers and Code Segment Protection Registers) and one register (Data Protection Mode Register) which determines the memory access modes which apply to the specified range.

The pairs of memory range registers determine the lower address boundary and the upper address boundary of each memory range. The Data Protection Mode Registers and Code Protection Mode Registers determine the access permissions for the ranges specified in their corresponding address range registers.

The memory protection system can also be used to generate signals to the Debug Unit when the processor attempts to access certain memory addresses. When used this way, values in the memory range registers are regarded as individual addresses, instead of defining an address range. An equality comparison with the contents of the address register pairs is performed instead of the normal address range calculation. If enabled for this function, signals are generated to the Debug Unit if the address of a memory access equals any of the address range registers.

Note that while the TriCore architecture allows as many as four Memory Protection Register Sets, the TC11IB implements two; and while the TriCore architecture allows as many as four Code Segment Protection Register Sets, the TC11IB implements two.

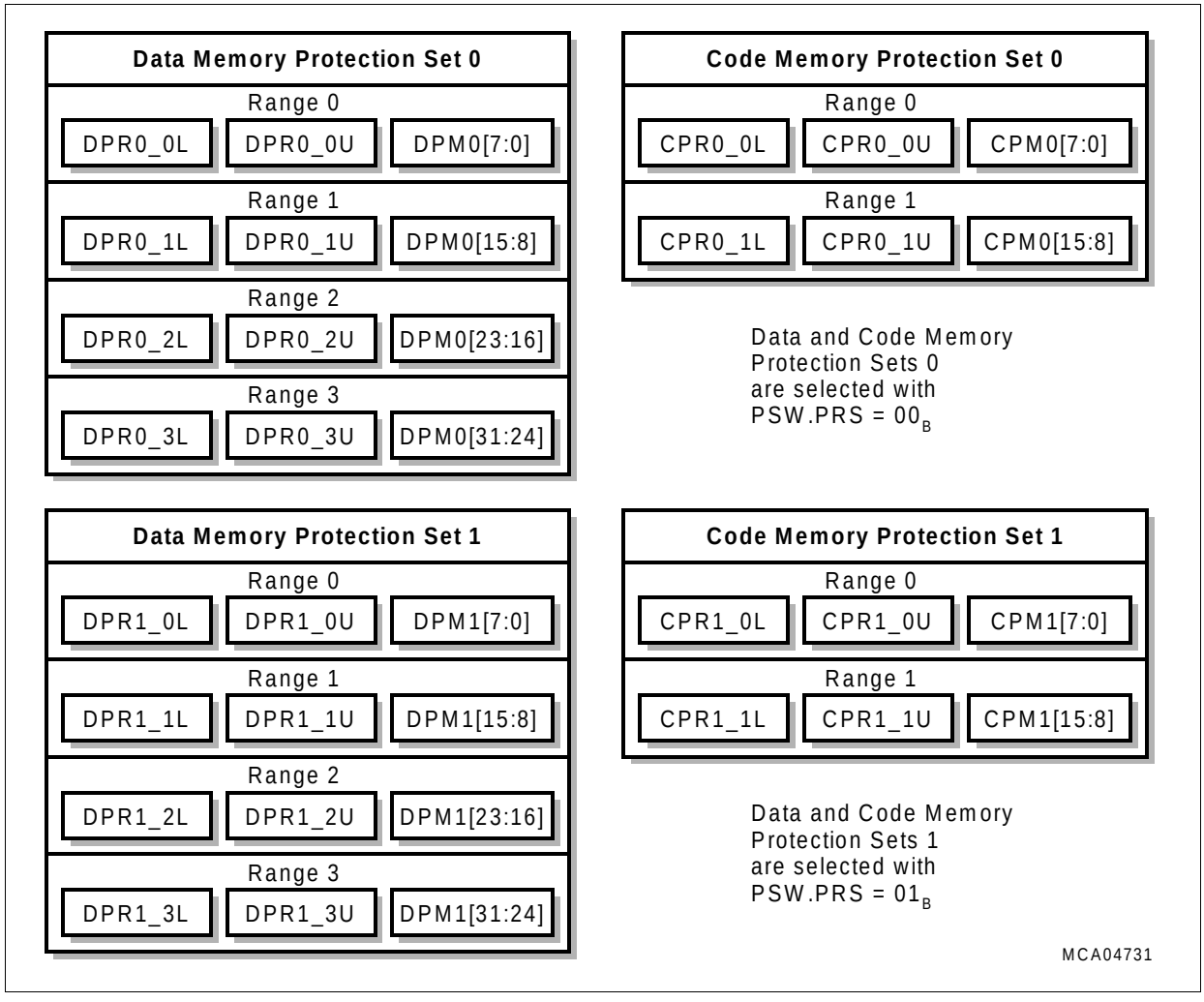


Figure 12-1 Memory Protection Register Sets

12.2 Memory Protection Registers

The TC11IB memory protection architecture is based on memory segments which are specified by address ranges and their associated access permissions or modes. Specific access permissions are associated with each addressable range. Ranges and their associated permissions are specified in two Memory Protection Register Sets (PRS) residing in the Core Special Function Registers (CSFR). A PRS consists of Data Segment Protection Registers, Data Protection Mode Registers, Code Segment Protection Registers, and Code Protection Mode Registers. The organization of these registers is shown in **Figure 12-1**. The PSW_PRS bit field indexes the current PRS. The current PRS determines what accesses can be performed by the processor for each memory segment.

Because of the Harvard-style architecture of the TC11IB, each PRS contains separate registers for checking data accesses and code accesses. Memory ranges are specified by pairs of registers which give lower and upper boundary for the associated ranges.

Data and code memory range registers are collectively named DPR*x*_*n*{L,U} and CPR*x*_*n*{L,U}, respectively. In all cases, *x* refers to the specific Memory Protection Register Set that the register is in, *n* refers to the range within the set, and *L* and *U* refer to the lower and upper boundary, respectively. For some lower boundary *L*, upper boundary *U*, and address *a*, the range defined by each address-range register pair is the interval: $L \leq a < U$.

The memory protection system can also be used to generate signals to the Debug Unit when the processor attempts to access particular memory addresses. When used this way, values in the DPR*x*_*n*{L,U} and CPR*x*_*n*{L,U} registers are regarded as individual addresses, instead of defining an address range. An equality comparison with the contents of the address register pairs is performed instead of the normal address range calculation. If enabled for this function, signals are generated to the Debug Unit if the address of a memory access equals any of the DPR*x*_*n*{L,U} and CPR*x*_*n*{L,U} registers.

When used for normal memory protection (not for debugging), the memory protection system performs as outlined in the following paragraphs. When the CPU performs load and store operations, data addresses are checked against the memory ranges given by the current data protection registers. Likewise, when the CPU fetches instructions, the address of the instruction is checked against the memory ranges given by the current code protection registers.

Range checking is disabled if the lower address is greater than the upper address. If the lower address is equal to the upper address, the segment is regarded as empty. If the address does not correspond to an allowable address range in any segment of the current PRS, a trap signal is generated by the memory protection hardware. Note that range checking is also disabled if the mode of a segment indicates that it is to signal the Debug Unit.)

If the address being examined is found to fall within an enabled, non-empty, and allowable range, the associated mode register is checked for access permissions. If the access mode is not allowed, a trap signal is generated by the memory protection hardware.

Table 12-1 shows all registers of the TC11IB Memory Protection Unit.

Table 12-1 Memory Protection Registers

Register Short Name	Register Long Name	Offset Address	Description see
DPR0_0L	Data Segment Protection Register Set 0, Range 0, Lower	0000_H	Page 12-11
DPR0_0U	Data Segment Protection Register Set 0, Range 0, Upper	0004_H	
DPR0_1L	Data Segment Protection Register Set 0, Range 1, Lower	0008_H	
DPR0_1U	Data Segment Protection Register Set 0, Range 1, Upper	$000C_H$	
DPR0_2L	Data Segment Protection Register Set 0, Range 2, Lower	0010_H	
DPR0_2U	Data Segment Protection Register Set 0, Range 2, Upper	0014_H	
DPR0_3L	Data Segment Protection Register Set 0, Range 3, Lower	0018_H	
DPR0_3U	Data Segment Protection Register Set 0, Range 3, Upper	$001C_H$	

Table 12-1 Memory Protection Registers (cont'd)

Register Short Name	Register Long Name	Offset Address	Description see
DPR1_0L	Data Segment Protection Register Set 1, Range 0, Lower	0400_H	Page 12-11
DPR1_0U	Data Segment Protection Register Set 1, Range 0, Upper	0404_H	
DPR1_1L	Data Segment Protection Register Set 1, Range 1, Lower	0408_H	
DPR1_1U	Data Segment Protection Register Set 1, Range 1, Upper	$040C_H$	
DPR1_2L	Data Segment Protection Register Set 1, Range 2, Lower	0410_H	
DPR1_2U	Data Segment Protection Register Set 1, Range 2, Upper	0414_H	
DPR1_3L	Data Segment Protection Register Set 1, Range 3, Lower	0418_H	
DPR1_3U	Data Segment Protection Register Set 1, Range 3, Upper	$041C_H$	Page 12-11
CPR0_0L	Code Segment Protection Register Set 0, Range 0, Lower	1000_H	Page 12-14
CPR0_0U	Code Segment Protection Register Set 0, Range 0, Upper	1004_H	
CPR0_1L	Code Segment Protection Register Set 0, Range 1, Lower	1008_H	
CPR0_1U	Code Segment Protection Register Set 0, Range 1, Upper	$100C_H$	
CPR1_0L	Code Segment Protection Register Set 1, Range 0, Lower	1400_H	Page 12-14
CPR1_0U	Code Segment Protection Register Set 1, Range 0, Upper	1404_H	
CPR1_1L	Code Segment Protection Register Set 1, Range 1, Lower	1408_H	
CPR1_1U	Code Segment Protection Register Set 1, Range 1, Upper	$140C_H$	
DPM0	Set 0 Data Protection Mode Register, Set 0	2000_H	Page 12-12

Table 12-1 Memory Protection Registers (cont'd)

Register Short Name	Register Long Name	Offset Address	Description see
DPM1	Data Protection Mode Register, Set 1	2080_H	Page 12-12
CPM0	Code Protection Mode Register, Set 0	2200_H	Page 12-15
CPM1	Code Protection Mode Register, Set 1	2280_H	Page 12-15

In the TC11IB, the memory protection registers are located in the following address range:

– Module Base Address. F7E1 $C000_H$
  Module End Address. F7E1 $EFFF_H$
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 12-1**)

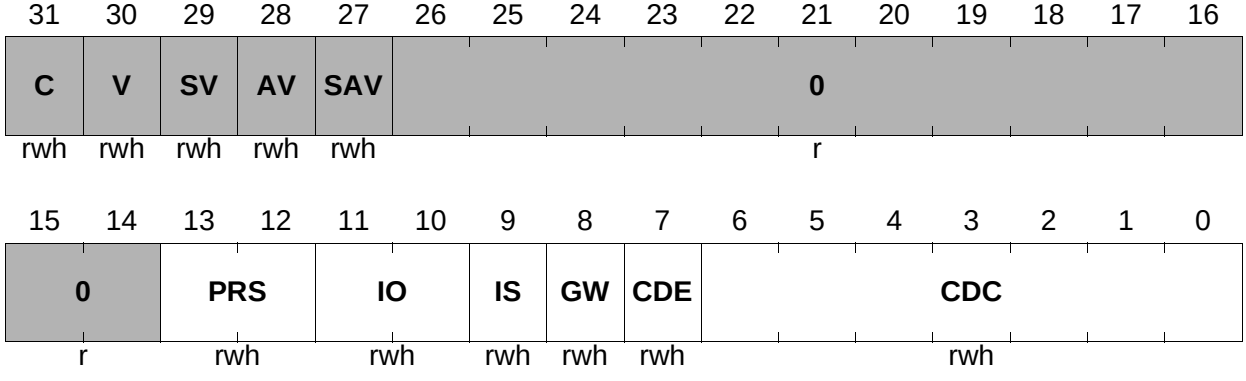
There are two major components within the memory protection system:

– The control bits and bit fields in the PSW.
– The memory protection registers which control program execution and memory access.

12.2.1 PSW Protection Fields

The control fields in the PSW that do not deal with the protection system are shaded in the PSW register table below.

PSW
Program Status Word **Reset Value: 0000 0B80$_H$**



Field	Bits	Type	Description
CDC	[6:0]	rwh	**Call Depth Counter** The CDC field consists of two variable-width fields. The first is a mask field, consisting of a string of zero or more initial 1 bits, terminated by the first 0 bit. The remaining bits of the field are the call depth counter. 0ccccc$_B$ 6-bit counter; trap on overflow 10cccc$_B$ 5-bit counter; trap on overflow 110ccc$_B$ 4-bit counter; trap on overflow 1110cc$_B$ 3-bit counter; trap on overflow 11110c$_B$ 2-bit counter; trap on overflow 111110$_B$ 1-bit counter; trap on overflow 1111110$_B$ Trap every call (call trace mode) 1111111$_B$ Disable call depth counting When the call depth counter overflows, a trap is generated. Depending on the width of the mask field, the call depth counter can be set to overflow at any power of two boundary, from 1 to 64. Setting the mask field to 1111110$_B$ allows no bits for the counter, and causes every call to be trapped. This is used for call tracing. Setting the field to mask field to 1111111$_B$ disables call depth counting altogether.

Field	Bits	Type	Description
CDE	7	rwh	**Call Depth Count Enable** The CDE bit enables call-depth counting, provided that the CDC mask field is not all 1’s. CDE is set to 1 by default, but should be cleared by the SYSCALL instruction Trap Service Routine to allow a trapped SYSCALL instruction to execute without producing another trap upon return from the trap handler. It is then set again when the next SYSCALL instruction is executed. 0 Call depth counter disabled 1 Call depth counter enabled
GW	8	rwh	**Global Register Write Permission** GW controls whether the current execution thread has permission to modify the global address registers. Most tasks and ISRs will use the global address registers as “read only” registers, pointing to the global literal pool and key data structures. However, a task or ISR can be designated as the “owner” of a particular global address register, and is allowed to modify it. The system designer must determine which global address variables are used with sufficient frequency and/or in sufficiently time-critical code to justify allocation to a global address register. By compiler convention, global address register A0 is reserved as the base register for short form loads and stores. Register A1 is also reserved for compiler use. Registers A8 and A9 are not used by the compiler, and are available for holding critical system address variables. 0 Write permission to global registers A0, A1, A8, and A9 is disabled 1 Write permission to global registers A0, A1, A8, and A9 is enabled

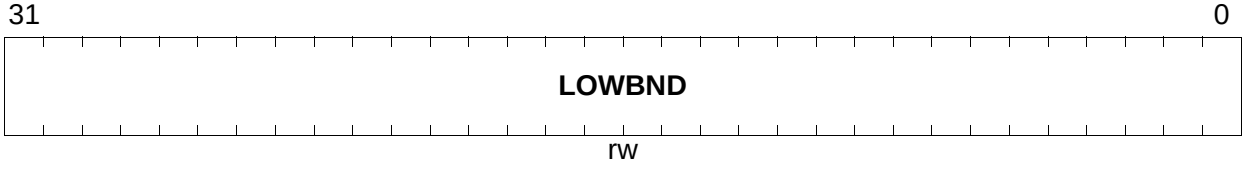
Field	Bits	Type	Description
IS	9	rwh	**Interrupt Stack Control** Determines whether the current execution thread is using the shared global (interrupt) stack or a user stack. 0 U**ser Stack**. If an interrupt is taken when the IS bit is 0, then the stack pointer register is loaded from the ISP register before execution starts at the first instruction of the Interrupt Service Routine. 1 S**hared Global Stack**. If an interrupt is taken when the IS bit is 1, then the current value of the stack pointer register is used by the Interrupt Service Routine.
IO	[11:10]	rwh	**Access Privilege Level Control** This 2-bit field selects determines the access level to special function registers and peripheral devices. 00_B **User-0 Mode**: No peripheral access. Access to segments 14 and 15 is prohibited and will result in a trap. This access level is given to tasks that need not directly access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts. 01_B **User-1 Mode**: regular peripheral access. This access level enables access to common peripheral devices that are not specially protected, including read/write access to serial I/O ports, read access to timers, and access to most I/O status registers. Tasks at this level may disable interrupts. 10_B **Supervisor Mode**. This access level enables access to all peripheral devices. It enables read/write access to core registers and protected peripheral devices. Tasks at this level may disable interrupts. 11_B **Reserved**; this encoding is reserved and is not defined.

Field	Bits	Type	Description
PRS	[13:12]	rwh	**Protection Register Set Selection** The PRS field selects one of two possible sets of memory protection register values controlling load and store operations and instruction fetches within the current process. This field indicates the current protection register set. 00 Protection register set 0 selected 01 Protection register set 1 selected 10 Reserved; don't use this combination 11 Reserved; don't use this combination
0	[26:14]	r	**Reserved**; read as 0; should be written with 0.
–	[31:27]	rwh	Not used for memory protection purposes.

12.2.2 Data Memory Protection Register

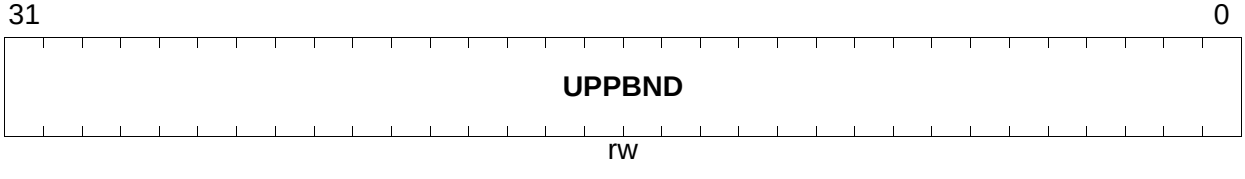
The lower and upper boundaries of a data memory segment are specified by word-length register pairs DPR*x*_*n*L and DPR*x*_*n*U respectively, where *x* is the Memory Protection Register Set number (0..1) and *n* is the range number (0..3).

DPR0_0L DPR0_1L DPR0_2L DPR0_3L
DPR1_0L DPR1_1L DPR1_2L DPR1_3L
Data Segment Protection Register n, Set x, Lower Bound DPRx_nL (x = 0, 1, n = 0-3)
Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
LOWBND	[31:0]	rw	**Lower Boundary Address**

DPR0_0U DPR0_1U DPR0_2U DPR0_3U
DPR1_0U DPR1_1U DPR1_2U DPR1_3U
Data Segment Protection Register n, Set x, Upper Bound DPRx_nU (x = 0, 1, n = 0-3)
Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
UPPBND	[31:0]	rw	**Upper Boundary Address**

The access permissions of the two data memory ranges are specified by the registers DPM*x*, where *x* is the Memory Protection Register Set number (x = 0, 1). Four byte fields within each DPM*x* register are assigned to the range number (0..3). Note that in one set the mode register with the four ranges is located in a single word register. Byte field DPMx[7:0] is assigned to Range 0, byte field DPMx[15:8] is assigned to Range 1, byte field DPM[23:16] is assigned to Range 2, and byte field DPMx[31:24] is assigned to Range 3.

DPM0 DPM1

Data Protection Mode Registers DPMx (x = 0, 1) **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WE 3	**RE 3**	**WS 3**	**RS 3**	**WBL 3**	**RBL 3**	**WBU 3**	**RBU 3**	**WE 2**	**RE 2**	**WS 2**	**RS 2**	**WBL 2**	**RBL 2**	**WBU 2**	**RBU 2**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WE 1	**RE 1**	**WS 1**	**RS 1**	**WBL 1**	**RBL 1**	**WBU 1**	**RBU 1**	**WE 0**	**RE 0**	**WS 0**	**RS 0**	**WBL 0**	**RBL 0**	**WBU 0**	**RBU 0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

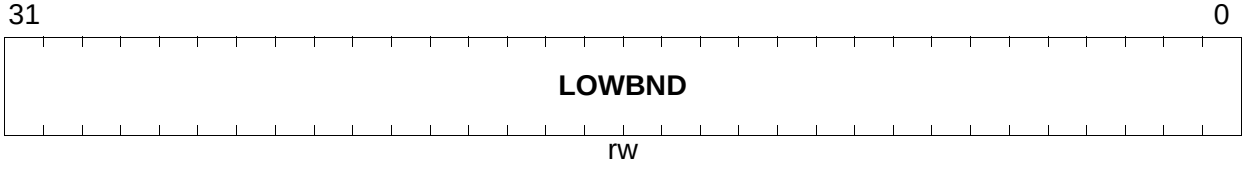
Field	Bits	Type	Description
RBUn (n = 0-3)	0, 8, 16, 24	rw	**Data Read Signal on Upper Bound Access Range n** 0 Data read signal is disabled 1 A signal is asserted to the debug unit on a data read access to an address that matches the upper boundary address of the associated address range.
WBUn (n = 0-3)	1, 9, 17, 25	rw	**Write Signal on Upper Bound Access Range n** 0 Write signal is disabled 1 A signal is asserted to the debug unit on a data write access to an address that matches the upper boundary address of the associated address range.
RBLn (n = 0-3)	2, 10, 18, 26	rw	**Data Read Signal on Lower Bound Access Range n** 0 Data read signal is disabled 1 A signal is asserted to the debug unit on a data read access to an address that matches the lower boundary address of the associated address range.

Field	Bits	Type	Description
WBLn (n = 0-3)	3, 11, 19, 27	rw	**Data Write Signal on Lower Bound Access Range n** 0 Data write signal is disabled 1 A signal is asserted to the debug unit on a data write access to an address that matches the lower boundary address of the associated address range
RSn (n = 0-3)	4, 12, 20, 28	rw	**Address Range Data Read Signal Range n** 0 Data read signal is disabled 1 A signal is asserted to the debug unit on data read accesses to the associated address range
WSn (n = 0-3)	5, 13, 21, 29	rw	**Address Range Data Write Signal Range n** 0 Data write signal is disabled 1 A signal is asserted to the debug unit on data write accesses to the associated address range
REn (n = 0-3)	6, 14, 22, 30	rw	**Address Range Data Read Enable Range n** RE controls reads to the addresses in the associated range. 0 Data read accesses to the associated address range are not permitted 1 Data read accesses to the associated address range are permitted
WEn (n = 0-3)	7, 15, 23, 31	rw	**Address Range Data Write Enable Range n** WE controls writes to the addresses in the associated range. 0 Data write accesses to the associated address range are not permitted 1 Data write accesses to the associated address range are permitted

12.2.3 Code Memory Protection Register

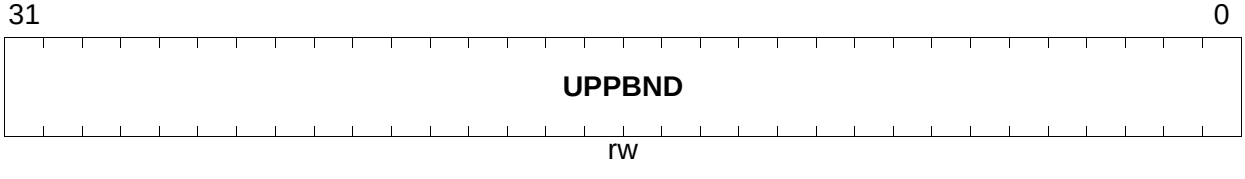
The lower and upper boundaries of a code memory segment are specified by word length register pairs CPR*x*_*n*L and CPR*x*_*n*U respectively, where *x* is the Memory Protection Register Set number (0..1) and *n* is the range number (0..1).

CPR0_0L CPR0_1L
CPR1_0L CPR1_1L
Code Segment Protection Register n, Set x, Lower Bound CPRx_nL (x = 0, 1, n = 0, 1)
Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
LOWBND	[31:0]	rw	**Lower Boundary Address**

CPR0_0U CPR0_1U
CPR1_0U CPR1_1U
Code Segment Protection Register n, Set x, Upper Bound CPRx_nU (x = 0, 1, n = 0, 1)
Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
UPPBND	[31:0]	rw	**Upper Boundary Address**

The access permissions of the two code memory ranges are specified by the registers CPM*x*, where *x* is the Memory Protection Register Set number (x = 0, 1). Two byte fields within each CPM*x* register are assigned to the range number (0, 1). Note that in one set, the mode register with the two ranges is located in a single word register. Byte field CPMx[7:0] is assigned to Range 0, and byte field CPMx[15:8] is assigned to Range 1.

CPM0 CPM1

Code Protection Mode Registers CPMx (x = 0, 1) **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XE 1	**0**	**XS 1**	**0**	**BL 1**	**0**	**0**	**BU 1**	**XE 0**	**0**	**XS 0**	**0**	**BL 0**	**0**	**0**	**BU 0**
rw	r	rw	r	rw	r	r	rw	rw	r	rw	r	rw	r	r	rw

Field	Bits	Type	Description
BUn (n = 0, 1)	0, 8	rw	**Execute Signal on Upper Bound Access Range n** 0 Upper bound execute signal is disabled 1 A signal is asserted to the debug unit on an instruction fetch to an address that matches the upper bound address of the associated address range
BLn (n = 0, 1)	3, 11	rw	**Execute Signal on Lower Bound Access Range n** 0 Lower bound execute signal is disabled 1 A signal is asserted to the debug unit on an instruction fetch to an address that matches the lower bound address of the associated address range
XSn (n = 0, 1)	5, 13	rw	**Address Range Execute Signal Range n** 0 Execute signal is disabled 1 A signal is asserted to the debug unit on instruction fetches to the associated address range
XEn (n = 0, 1)	7, 15	rw	**Address Range Execute Enable Range n** 0 Instruction fetches to the associated address range are not permitted 1 Instruction fetches to the associated address range are permitted
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

At any given time, one of the sets is the current protection register set that determines the legality of memory accesses by the current task or ISR. The PRS field in the PSW indicates the current protection register set number. Each protection register set contains separate address range tables for checking data accesses and code accesses. The range table entry is a pair of words specifying a lower and an upper boundary for the associated range. The range defined by one range table entry is the address interval:

- lower bound ≤ address < upper bound

Each range table entry has an associated mode table entry in which access permissions and debug signal conditions for that range are specified. For load and store operations, data address values are checked against the entries in the data range table. For instruction fetches, the PC value for the fetch is checked against the entries in the code range table. When an address is found to fall within a range defined in the appropriate range table, the associated mode table entry is checked for access permissions and debug signal generation.

Modes of Use for Range Table Entries

An individual range table entry can be used for memory protection or for debugging; it is rarely used for both purposes. If the upper and lower bound values have been set for debug breakpoints, they probably are not meaningful for defining protection ranges, and vice versa.However, it is possible — and reasonable — to have some entries in the table for memory protection and others for debugging.

To disable an entry for memory protection, clear both the RE and WE bits in a data range table entry or clear the XE bit in a code range table entry. The entry can be disabled for use in debugging by clearing any debug signal bits. If a range entry is being used for debugging, the debug signal bits that are set determine whether it is used as a single range comparator (giving an in-range/not in-range signal) or as a pair of equal comparators. The two uses are not mutually exclusive.

Using Protection Register Sets

If there were only one protection register set, then either the mappings would need to be general enough to apply to all tasks and ISRs — thus, not terribly useful for isolating software errors in individual tasks — or there would need to be substantial overhead paid on interrupts and task context switches for updating the tables to match the currently executing task or ISR. Those drawbacks are avoided by providing for multiple sets of tables, with two bits in the PSW to select the currently active set.

Note that supervisor mode does not automatically disable memory protection. The protection register set selected for supervisor tasks will normally be set up to allow write access to regions of memory protected from user mode access. In addition, of course, supervisor tasks can execute instructions to change the protection maps, or to disable the protection system entirely. But supervisor mode does not implicitly override memory protection, and it is possible for a supervisor task to take a memory protection trap.

12.3 Sample Protection Register Set

Figure 12-2 illustrates Data Protection Register Set *n*, where *n* is one of the two sets selected by the PSW.PRS field. Each register set in this example consists of four range table entries. The defined ranges can potentially overlap or be nested. Nesting of ranges can be used, for example, to allow write access to a subrange of a larger range in which the current task is allowed read access. The four Data Segment Protection Registers and four Data Protection Mode Registers are set up as follows:

- Data Segment Protection Register 3 (DPRn_3) defines the upper and lower boundaries for Data Range 4. Data Protection Mode Register 3 (DPMn_3) defines the permissions and debug conditions for Data Range 4.
- Data Segment Protection Register 2 (DPRn_2) defines the upper and lower boundaries for Data Range 3. Data Protection Mode Register 2 (DPMn_2) defines the permissions and debug conditions for Data Range 3. Note that Data Range 3 is nested within Data Range 4.
- Data Segment Protection Register 1 (DPRn_1) defines the upper and lower boundaries for Data Range 2. Data Protection Mode Register 1 (DPMn_1) defines the permissions and debug conditions for Data Range 2.
- Data Segment Protection Register 0 (DPRn_0) defines the upper and lower boundaries for Data Range 1. Data Protection Mode Register 0 (DPMn_0) defines the permissions and debug conditions for Data Range 1.

This same configuration can be used to illustrate Code Protection Register Set n.

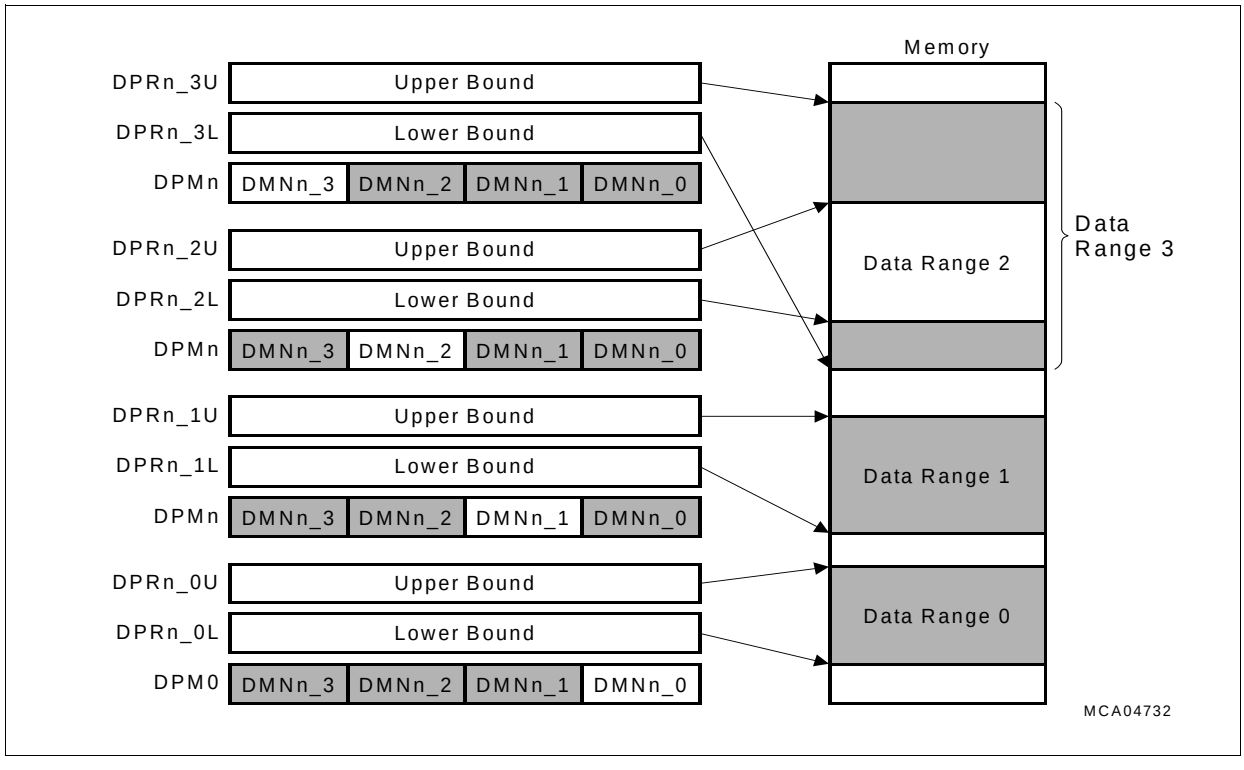


Figure 12-2 Example Configuration of a Data Protection Register Set (Set *n*)

12.4 Memory Access Checking

If the protection system is enabled, before any memory access (read, write, execute) is performed, it is checked for legality as determined by all of the following:

- The protection enable bits in the SYSCON Register,
- The current I/O privilege level (0 = User-0 Mode; 1 = User-1 Mode; 2 = Supervisor Mode), and
- The ranges defined in the currently selected protection register set.

Data addresses (read and write accesses) are checked against the currently selected data address range table, while instruction fetch addresses are checked against the code address range tables. The mode entries for the data range table entries enable only read and write accesses, while the mode entries for the code range table entries enable only execute access. In order for data to be read from program space, there must be an entry in the data address range table that covers the address being read. Conversely, there must be an entry in the code address range table for the instruction being read.

Access to the internal and external peripherals is through the two upper segments of the TC11IB address space (high-order address bits equal to 1110_B and 1111_B). Access checking for addresses in the peripheral segments is independent of access checking in the remainder of the address space. Access to peripheral segments is not allowed for tasks at I/O privilege Level 0 (User-0 tasks). Tasks at I/O privilege Level 1 and higher have access rights to the peripheral segment space. However, the validity of any access attempt depends on the presence of a peripheral at the accessed address, and any restrictions it may impose on its own access. Protected peripherals, for example, require I/O privilege Level 2, as reflected by the supervisor line value on the system bus.

If the memory protection system is disabled, any access to any memory address outside of the peripheral segments is permitted, regardless of the I/O privilege level. There are no memory regions reserved for supervisor access only, when the memory protection system is disabled.

When the memory protection system is enabled, for an access to be permitted, the address for the access must fall within one or more of the ranges specified in the currently selected protection register set. Furthermore, the mode entry for at least one of the matching ranges must enable the requested type of access.

12.4.1 Permitted versus Valid Accesses

A memory access can be permitted within the ranges specified in the data and code range tables without necessarily being valid. A range specified in a range table entry could cover one or more address regions where no physical memory was implemented. Although that would normally reflect an error in the system code that set up the address range, the memory protection system only uses the range table entries when determining whether an access is permitted. In addition, if the memory protection system

is disabled, all accesses must be taken as permitted, although individual accesses may or may not be valid.

An access that is not permitted under the memory protection system results in a memory protection trap. When permitted, an access to an unimplemented memory address results in a bus error trap, provided that the memory address is in one of the segments reserved for local memory. If the address is an external memory address, the result depends on the memory implementation, and is not architecturally defined. An access can also be permitted but invalid due to a misaligned address. Misaligned accesses result in an alignment trap, rather than a protection trap.

12.4.2 Crossing Protection Boundaries

An access can straddle two regions. For example, **Figure 12-3** illustrates the condition where Instruction A lies in an execute region of memory, Instruction C lies in a no-execute region of memory, and Instruction B straddles the execute/no execute boundary.

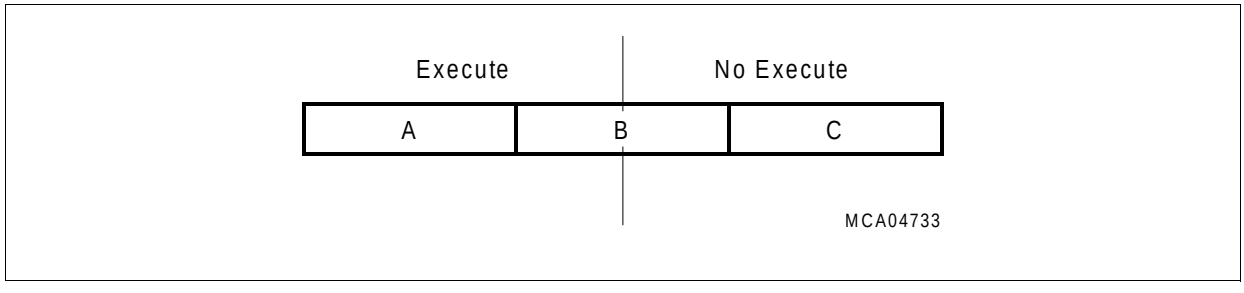


Figure 12-3 Protection Boundaries

Because the PC is used in the comparison with the range registers, the program error exception is not signaled until Instruction C is fetched. The same is true for all comparisons — the address of the first accessed byte is compared against the memory protection range registers. Hence, an access assumes the memory protection properties of the first byte in the access regardless of the number of bytes involved in the access.

For normal accesses, this assumption is not a problem because the regions are set up according to the natural access boundaries for the code or data that the region contains. For wild accesses due to software or hardware errors, stores are the main concern. In the worst case, a double-word store that is aligned on a half-word boundary can extend three half-words beyond the end of the region in which its address lies.

One way to prevent boundary crossings is to leave at least three half-words of buffer space between regions. This configuration prevents wild stores from destroying data in adjacent read-only regions, for example.

13 Parallel Ports

The TC11IB has 96 digital input/output port lines organized into six parallel 16-bit ports, Port P0 to Port P5. The parallel ports can be all used as general purpose I/O lines or they can perform input/output functions for the on-chip peripheral units, even 24 of them can be used as interrupt inputs. Port P0 to Port P5 are assigned to the on-chip peripheral units for their specific I/O operations. An overview on the port-to-peripheral unit assignment is shown in **Figure 13-1**.

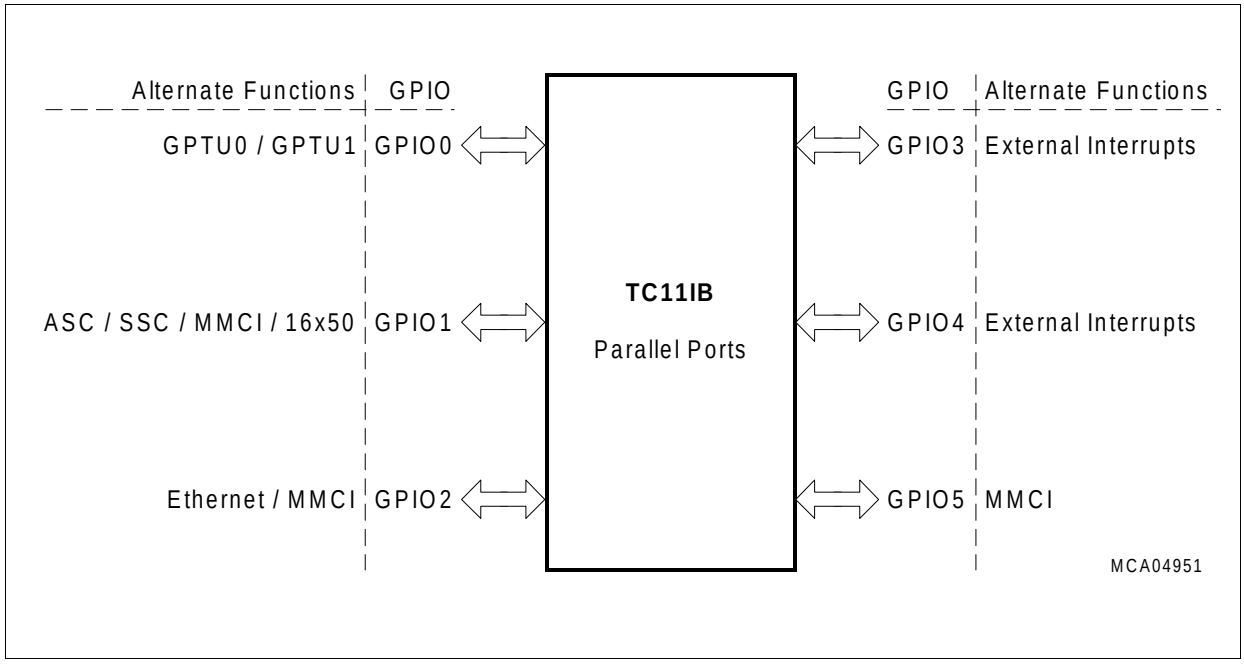


Figure 13-1 Parallel Ports of the TC11IB

13.1 General Port Operation

Figure 13-2 shows a general block diagram of an TC11IB port line. Each port line is equipped with a number of control and data bits, enabling very flexible usage of the line.

Each port pin can be configured for input or output operation. In input mode (default after reset), the output driver is switched off (high-impedance). The actual voltage level present at the port pin is translated into a logic 0 or 1 via a Schmitt-Trigger device and can be read via the read only register Px_IN. In output mode, the output driver is activated and drives the value supplied through the multiplexer to the port pin. Switching between input and output mode is accomplished through the Px_DIR register, which enables or disables the output driver.

The output multiplexer in front of the output driver enables the port output function to be used for different purposes. If the pin is used as general purpose output, the multiplexer is switched by software to the Output Data Register Px. Software can set or clear the bit in Px, and therefore it can directly influence the state of the port pin.

Latch Px_IN is provided for input functions of the on-chip peripheral units. Its input is connected to the output of the input Schmitt-Trigger. Further, an input signal can be connected directly to the various inputs of the peripheral units (AltDataIn). The function of the input line from the pin to the input latch Px_IN and to AltDataIn is independent of the port pin operates as input or output. This means that when the port is in output mode, the level of the pin can be read by software via latch Px_IN or a peripheral can use the pin level as an input. This offers additional advantages in an application.

- The data written to the output register Px by software can be used as input data to an on-chip peripheral. This enables, for example, peripheral tests via software without external circuitry. Examples for this can be the triggering of a timer count input, generating an external interrupt, or simulating the incoming serial data stream to a serial port receive input via software.
- When the pin is used as an output, the actual logic level at the pin can be examined through reading latch Px_IN and compared against the applied output level (either applied through software via the output register Px, or via an alternate output function of a peripheral). This can be used to detect some electrical failures at the pin caused through external circuitry. In addition, software supported arbitration schemes can be implemented in this way using the open-drain configuration and an external wired-And circuitry. Collisions on the external communication lines can be detected when a logic 1 is output, but a logic 0 is seen when reading the pin value via the input latch Px_IN.
- The output data from a peripheral applied to the pin via an alternate output function can be read through software or can be used by the same or another peripheral as input data. This enables testing of peripheral functions or provides additional connections between on-chip peripherals via the same pin without external wires.

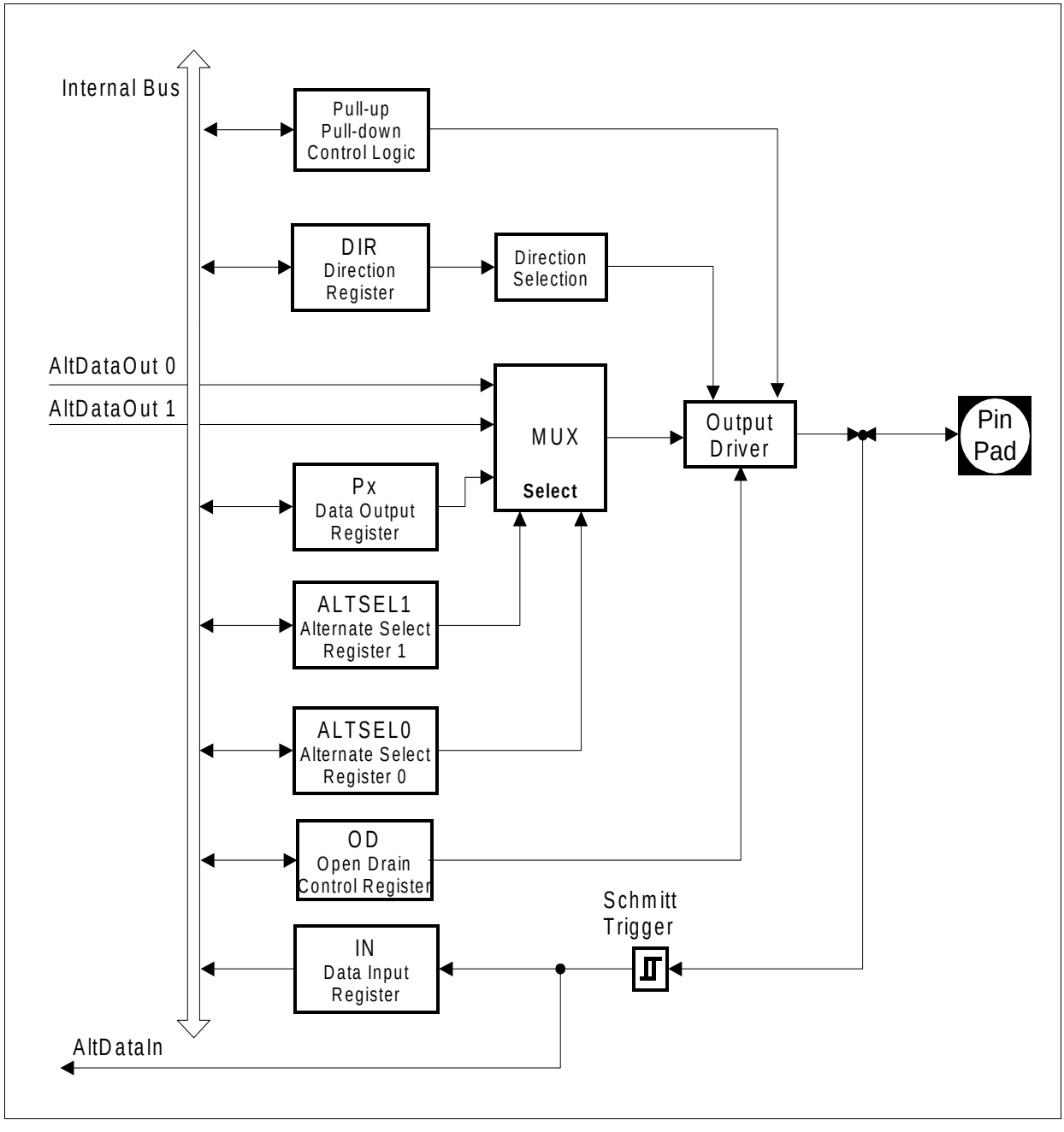


Figure 13-2 General Port Structure

13.2 Port Kernel Registers

The individual control and data bits of each digital parallel port are implemented in a number of registers. Bits with the same meaning and function are assembled together in the same register. Each parallel port consists of a set of registers. The registers are used to configure and use the port as general purpose I/O or alternate function input/output. For most ports not all registers are implemented. The availability of the kernel registers in the specific ports is defined in **Section 13.3** to **Section 13.8**.

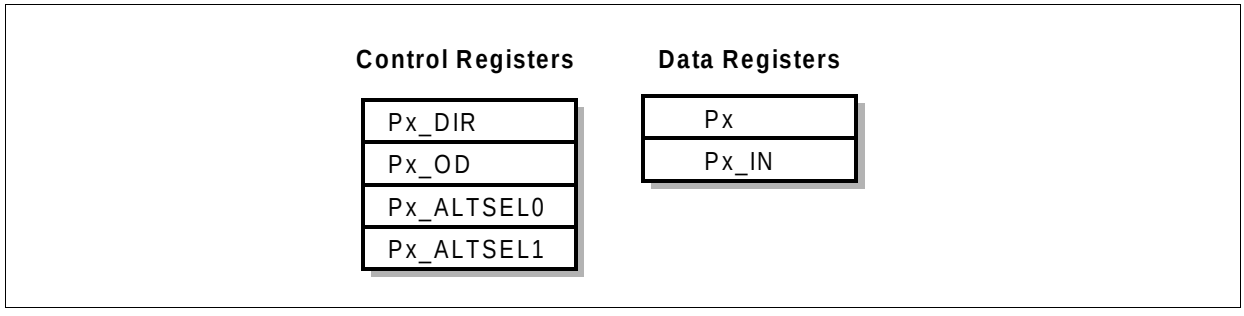


Figure 13-3 Port Kernel Registers

Table 13-1 Port Kernel Registers

Register Short Name	Register Long Name	Offset Address	Description see
Px	Port x Data Output Register	0010_H	**Page 13-5**
Px_IN	Port x Data Input Register	0014_H	**Page 13-6**
Px_DIR	Port x Direction Register	0018_H	**Page 13-7**
Px_OD	Port x Open Drain Control Register	$001C_H$	**Page 13-8**
Px_ALTSEL0	Port x Alternate Select Register 0	0044_H	**Page 13-9**
Px_ALTSEL1	Port x Alternate Select Register 1	0048_H	**Page 13-9**

In the TC11IB, the registers of the digital ports are located in the address ranges as shown in **Table 13-2**.

Table 13-2 Port Registers Address Ranges

Port No.	Address Range	Port No.	Address Range
Port 0	F000 2800_H - F000 $28FF_H$	Port 3	F000 $2B00_H$ - F000 $2BFF_H$
Port 1	F000 2900_H - F000 $29FF_H$	Port 4	F000 $2C00_H$ - F000 $2CFF_H$
Port 2	F000 $2A00_H$ - F000 $2AFF_H$	Port 5	F000 $2D00_H$ - F000 $2DFF_H$

– Absolute Register Address = Module Base Address (**Table 13-2**) + Offset Address (**Table 13-1**)

13.2.1 Data Output Register

If a port pin is used as general purpose output (GPIO), output data is written into register Px of port x.

Px
Port x Data Output Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P15	**P14**	**P13**	**P12**	**P11**	**P10**	**P9**	**P8**	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

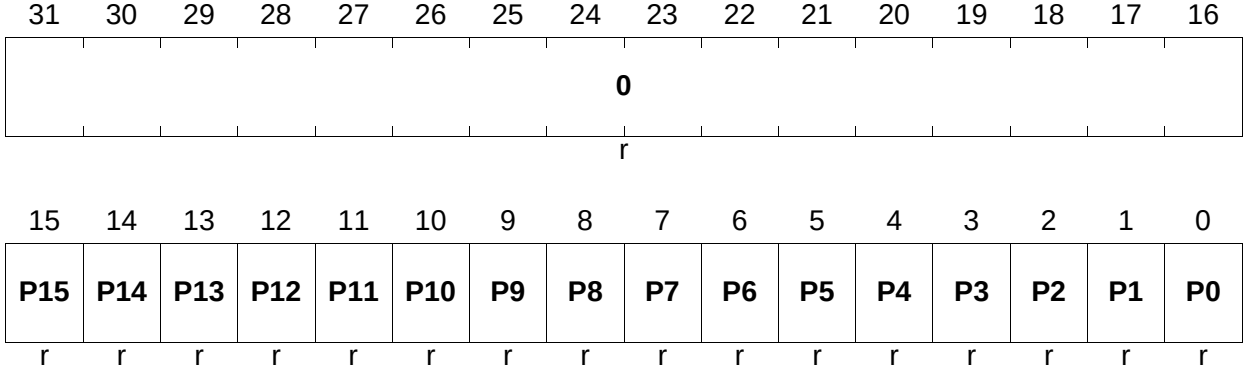
Field	Bits	Type	Description
Pn **(n = 15-0)**	n	rw	**Port x Pin n Output Value** 0 Port x pin n output value = 0 (default after reset) 1 Port x pin n output value = 1
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

The contents of Px.n are output on the assigned pin if the pin is assigned as GPIO pin and the direction is switched/set to output (Px_DIR.n = 1). A read operation of Px returns the register value and not the state of the Px pins.

13.2.2 Data Input Register

The value at a port pin can be read through the read-only register Px_IN. The data input register Px_IN always contains a latched value of the assigned port pin.

Px_IN
Port x Data Input Register **Reset Value: 0000 XXXX$_H$**



Field	Bits	Type	Description
Pn **(n = 15-0)**	n	rw	**Port x Pin n Latched Input Value** 0 Port x input pin n latched value = 0 1 Port x input pin n latched value = 1
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

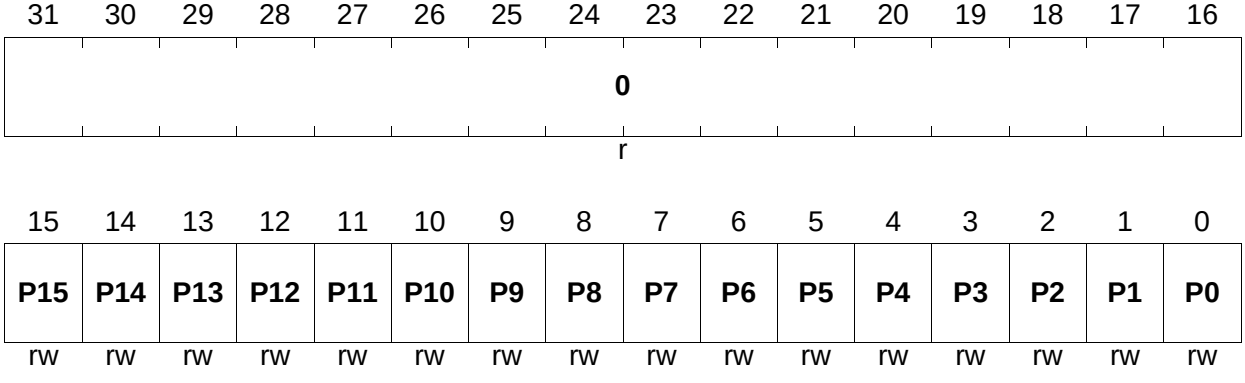
13.2.3 Direction Register

The direction of port pins can be controlled in the following ways:

- Always controlled by Px_DIR register
- Controlled by Px_DIR register if used for GPIO and controlled by the peripheral if used for alternate function
- Controlled by Px_DIR register if used as GPIO and fixed direction if used for alternate function
- Always fixed if used for GPIO and alternate function

If the port direction is controlled by the respective direction register Px_DIR, the following encoding is defined:

Px_DIR
Port x Direction Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
Pn **(n = 15-0)**	n	rw	**Port x Pin n Direction Control** 0 Direction is set to input (default after reset) 1 Direction is set to output
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

13.2.4 Open Drain Control Register

For ports P1 to P3, each pin in output mode can be switched to Open Drain Mode. If driven with 1, no driver will be activated; if driven with 0, the pull-down transistor will be activated.

The open drain mode is controlled by the register **Px_OD**.

Px_OD
Port x Open Drain Control Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P15	**P14**	**P13**	**P12**	**P11**	**P10**	**P9**	**P8**	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
Pn **(n = 15-0)**	n	rw	**Port x Pin n Open Drain Mode** 0 Normal Mode, output is actively driven for 0 and 1 state 1 Open Drain Mode, output is actively driven only for 0 state
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

13.2.5 Pull-Up/Pull-Down Device Control

In the TC11IB, internal pull-up/pull-down devices is fixed assignment to apply to a port pin. It is unnecessary to configure the input and output characteristics. This means that normal timing and high current outputs are always enabled due to the fixed output driver characteristics of the GPIO ports. During Power Down, the output driver is switched off. The last driven value can be kept valid by an internal weak pull-up/pull-down device.

13.2.6 Alternate Input Functions

The number of alternate functions that uses a pin for input is not limited. Each port control logic of an I/O pin provides several input paths:

- Digital input value via register
- Direct digital input value

13.2.6.1 Alternate Output Functions

Alternate functions are selected via an output muliplexer which can select up to four output lines. This muliplexer can be controlled by the following signals:

- Register Px_ALTSEL0
- Register Px_ALTSEL1

Selection of alternate functions are defined in registers Px_ALTSEL0 and Px_ALTSEL1. The tables in the port chapters **Section 13.3** to **Section 13.8** define which type of select signal is used for the alternate function selection for each port pin.

Px_ALTSELn (n = 1, 0)
Port x Alternate Select Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

13.3 Port 0

Port 0 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as I/O for GPTU0 and GPTU1 modules.The port lines can be used as standard GPIO pins if its alternate function is not required.

13.3.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices

13.3.2 Registers

The following port kernel registers are available at Port 0:

Table 13-3 Port 0 Kernel Registers

Register Short Name	Register Long Name
P0	Port 0 Data Output Register
P0_IN	Port 0 Data Input Register
P0_DIR	Port 0 Direction Register
P0_ALTSEL0	Port 0 Alternate Function Select Register 0
P0_ALTSEL1	Port 0 Alternate Function Select Register 1

13.3.3 Port Configuration and Function

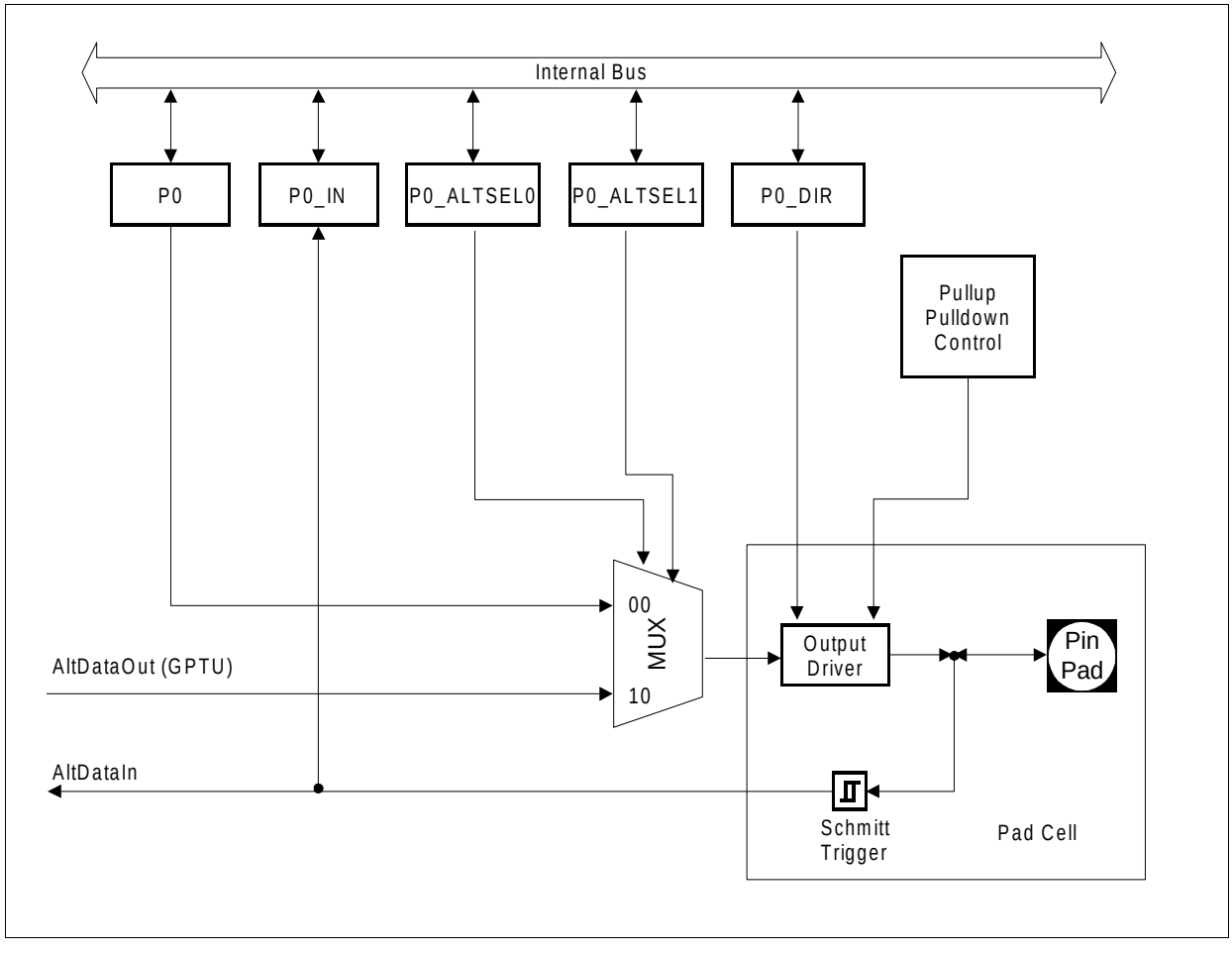


Figure 13-4 Port 0 Configuration

Table 13-4 Port 0 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P0.x (x = 7-0)	General purpose input	P0_IN.x		P0_DIR.x = 0
	General purpose output	P0.x	P0_ALTSEL0.x = 0 P0_ALTSEL1.x = 0	P0_DIR.x = 1
	GPTU0 I/O lines used as input	GPTU0		P0_DIR.x = 0
	GPTU0 I/O lines used as output		P0_ALTSEL0.x = 1 P0_ALTSEL1.x = 0	P0_DIR.x = 1
P0.x (x = 15-8)	General purpose input	P0_IN.x		P0_DIR.x = 0
	General purpose output	P0.x	P0_ALTSEL0.x = 0 P0_ALTSEL1.x = 0	P0_DIR.x = 1
	GPTU1 I/O lines used as input	GPTU1		P0_DIR.x = 0
	GPTU1 I/O lines used as output		P0_ALTSEL0.x = 1 P0_ALTSEL1.x = 0	P0_DIR.x = 1

13.4 Port 1

Port 1 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as I/O for SSC, MMCI, ASC and 16X50 modules. The port lines can be used as standard GPIO pins if its alternate function is not required.

13.4.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices
- Selectable open-drain functions for the following:

– P1.4/MMCI.CMD_OUT/MMCI.CMD_IN
– P1.5/MMCI.DAT_OUT/MMCI.DAT_IN

13.4.2 Registers

The following port kernel registers are available at Port 1:

Table 13-5 Port 1 Kernel Registers

Register Short Name	Register Long Name
P1	Port 1 Data Output Register
P1_IN	Port 1 Data Input Register
P1_DIR	Port 1 Direction Register
P1_OD	Port 1 Open Drain Control Register
P1_ALTSEL0	Port 1 Alternate Function Select Register 0

13.4.3 Port Configuration and Function

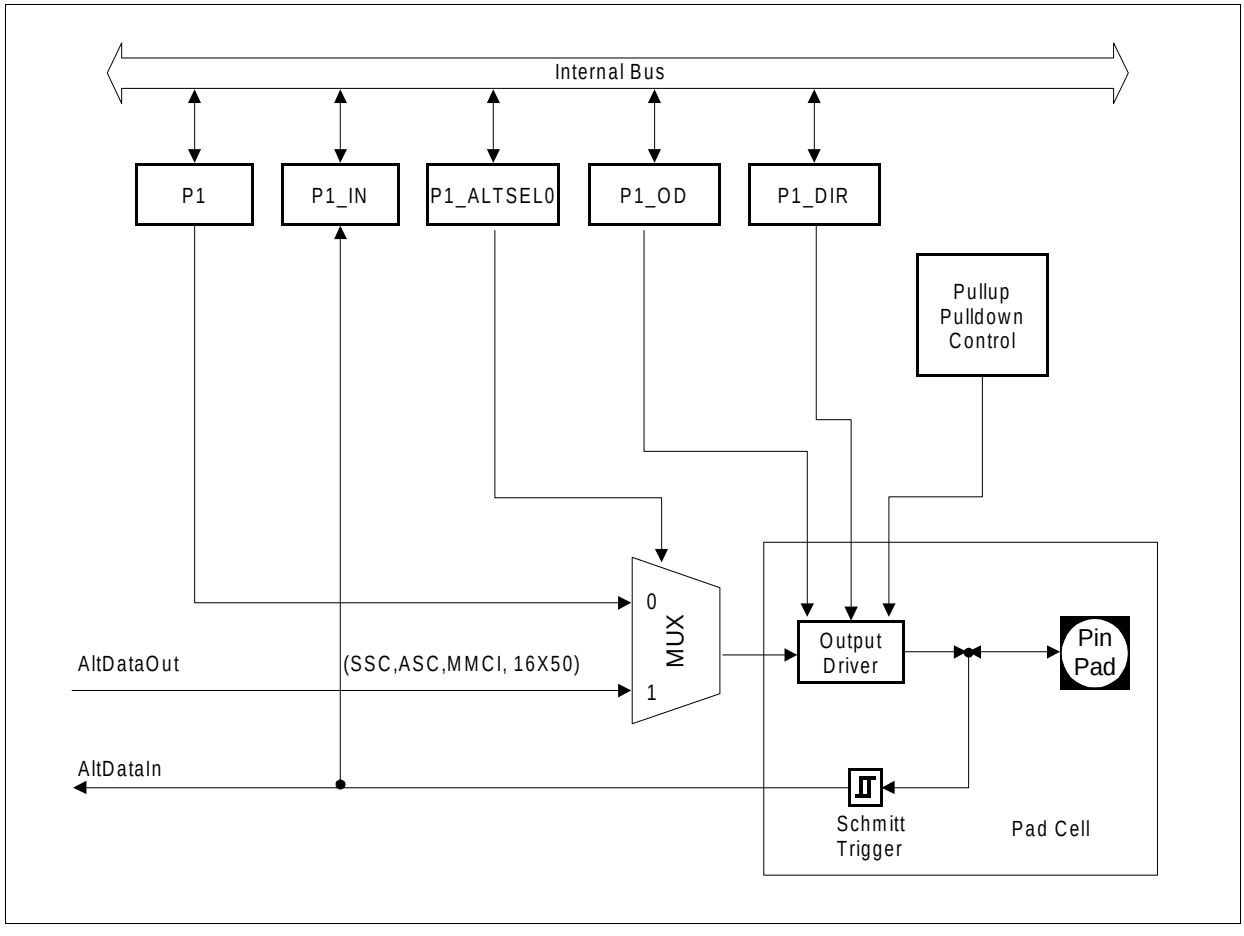


Figure 13-5 Port 1 Configuration

Table 13-6 Port 1 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P1.0	General purpose input	P1_IN.0		P1_DIR.0 = 0
	General purpose output	P1_OUT.0	P1_ALTSEL0.0 = 0	P1_DIR.0 = 1
	SSC clock input SCLK	SSC		P1_DIR.0 = 0
	SSC clock output SCLK		P1_ALTSEL0.0 = 1	P1_DIR.0 = 1

Table 13-6 Port 1 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P1.1	General purpose input	P1_IN.1		P1_DIR.1 = 0
	General purpose output	P1_OUT.1	P1_ALTSEL0.1 = 0	P1_DIR.1 = 1
	SSC master receive input MRST	SSC		P1_DIR.1 = 0
	SSC slave transmit output MRST		P1_ALTSEL0.1 = 1	P1_DIR.1 = 1
P1.2	General purpose input	P1_IN.2		P1_DIR.2 = 0
	General purpose output	P1_OUT.2	P1_ALTSEL0.2 = 0	P1_DIR.2 = 1
	SSC slave receive input MTSR	SSC		P1_DIR.2 = 0
	SSC master transmit output MTSR		P1_ALTSEL0.2 = 1	P1_DIR.2 = 1
P1.3	General purpose input	P1_IN.3		P1_DIR.3 = 0
	General purpose output	P1_OUT.3	P1_ALTSEL0.3 = 0	P1_DIR.3 = 1
	MMCI clock output CLK	MMCI	P1_ALTSEL0.3 = 1	P1_DIR.3 = 1
P1.4	General purpose input	P1_IN.4		P1_DIR.4 = 0
	General purpose output	P1_OUT.4	P1_ALTSEL0.4 = 0	P1_DIR.4 = 1
	MMCI command input CMD	MMCI		P1_DIR.4 = 0
	MMCI command output CMD		P1_ALTSEL0.4 = 1	P1_DIR.4 = 1
P1.5	General purpose input	P1_IN.5		P1_DIR.5 = 0
	General purpose output	P1_OUT.5	P1_ALTSEL0.5 = 0	P1_DIR.5 = 1
	MMCI data input DAT	MMCI		P1_DIR.5 = 0
	MMCI data output DAT		P1_ALTSEL0.5 = 1	P1_DIR.5 = 1

Table 13-6 Port 1 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P1.6	General purpose input	P1_IN.6		P1_DIR.6 = 0
	General purpose output	P1_OUT.6	P1_ALTSEL0.6 = 0	P1_DIR.6 = 1
	ASC receiver input RXD used as input	ASC		P1_DIR.6 = 0
	ASC receiver input RXD used as output		P1_ALTSEL0.6 = 1	P1_DIR.6 = 1
P1.7	General purpose input	P1_IN.7		P1_DIR.7 = 0
	General purpose output	P1_OUT.7	P1_ALTSEL0.7 = 0	P1_DIR.7 = 1
	ASC transmitter output TXD	ASC	P1_ALTSEL0.7 = 1	P1_DIR.7 = 1
P1.8	General purpose input	P1_IN.8		P1_DIR.8 = 0
	General purpose output	P1_OUT.8	P1_ALTSEL0.8 = 0	P1_DIR.8 = 1
	16X50 receiver input RXD	16X50		P1_DIR.8 = 0
P1.9	General purpose input	P1_IN.9		P1_DIR.9 = 0
	General purpose output	P1_OUT.9	P1_ALTSEL0.9 = 0	P1_DIR.9 = 1
	16X50 transmitter output RXD	16X50	P1_ALTSEL0.9 = 1	P1_DIR.9 = 1
P1.10	General purpose input	P1_IN.10		P1_DIR.10 = 0
	General purpose output	P1_OUT.10	P1_ALTSEL0.10 = 0	P1_DIR.10 = 1
	16X50 request to send output RTS	16X50	P1_ALTSEL0.10 = 1	P1_DIR.10 = 1
P1.11	General purpose input	P1_IN.11		P1_DIR.11 = 0
	General purpose output	P1_OUT.11	P1_ALTSEL0.11 = 0	P1_DIR.11 = 1
	16X50 data carrier detection input DCD	16X50		P1_DIR.11 = 0

Table 13-6 Port 1 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P1.12	General purpose input	P1_IN.12		P1_DIR.12 = 0
	General purpose output	P1_OUT.12	P1_ALTSEL0.12 = 0	P1_DIR.12 = 1
	16X50 data set ready input DSR	16X50		P1_DIR.12 = 0
P1.13	General purpose input	P1_IN.13		P1_DIR.13 = 0
	General purpose output	P1_OUT.13	P1_ALTSEL0.13 = 0	P1_DIR.13 = 1
	16X50 data terminal ready output DTR	16X50	P1_ALTSEL0.13 = 1	P1_DIR.13 = 1
P1.14	General purpose input	P1_IN.14		P1_DIR.14 = 0
	General purpose output	P1_OUT.14	P1_ALTSEL0.14 = 0	P1_DIR.14 = 1
	16X50 clear to send input CTS	16X50		P1_DIR.14 = 0
P1.15	General purpose input	P1_IN.15		P1_DIR.15 = 0
	General purpose output	P1_OUT.15	P1_ALTSEL0.15 = 0	P1_DIR.15 = 1
	16X50 ring indicator input RI	16X50		P1_DIR.15 = 0

13.5 Port 2

Port 2 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as I/O for the Ethernet controller and MMCI modules. The port lines can be used as standard GPIO pins if its alternate function is not required.

13.5.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices

13.5.2 Registers

The following port kernel registers are available at Port 2:

Table 13-7 Port 2 Kernel Registers

Register Short Name	Register Long Name
P2	Port 2 Data Output Register
P2_IN	Port 2 Data Input Register
P2_DIR	Port 2 Direction Register
P2_ALTSEL0	Port 2 Alternate Function Select Register

13.5.3 Port Configuration and Function

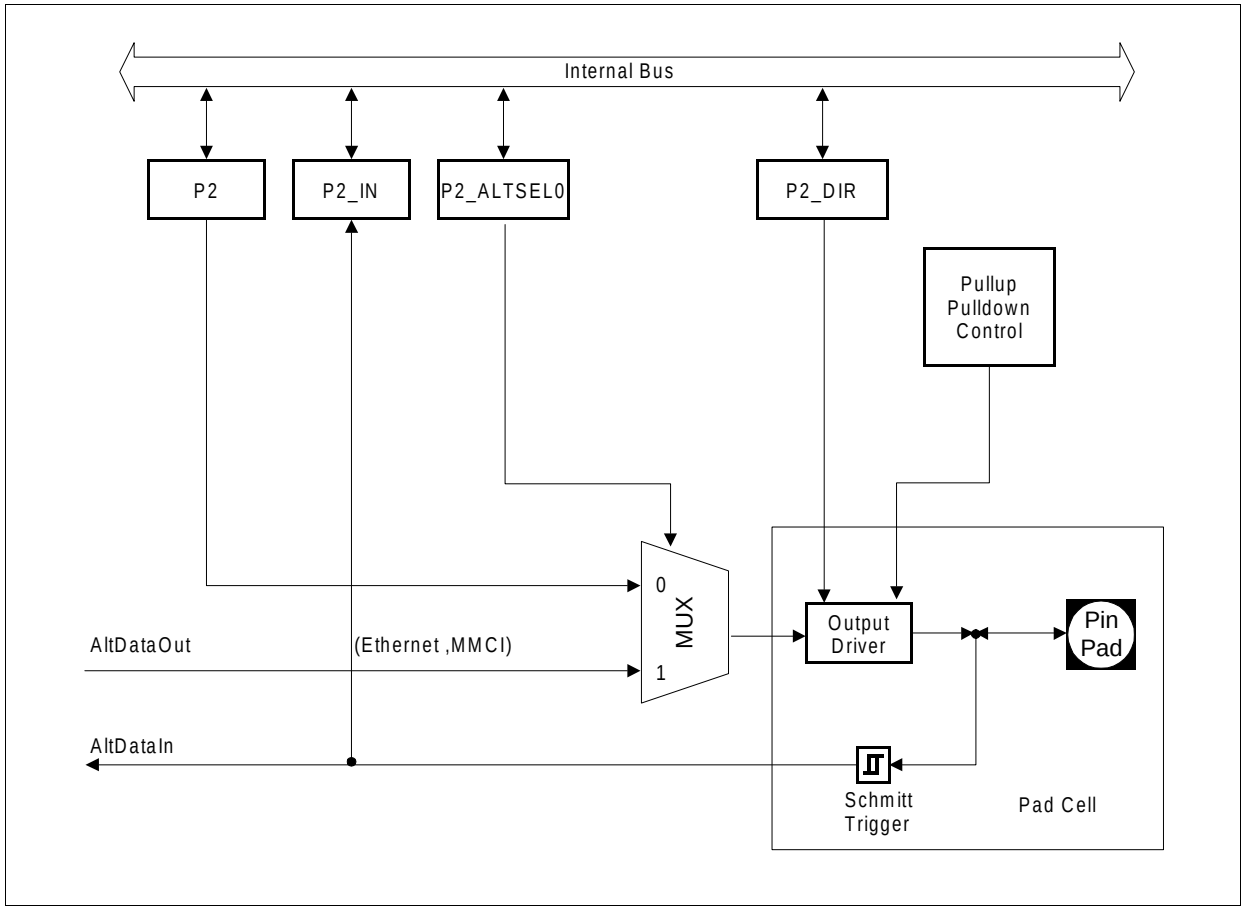


Figure 13-6 Port 2 Configuration

Table 13-8 Port 2 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P2.0	General purpose input	P2_IN.0		P2_DIR.0 = 0
	General purpose output	P2_OUT.0	P2_ALTSEL0.0 = 0	P2_DIR.0 = 1
	Ethernet transmit data output 0 MIITXD0	Ethernet	P2_ALTSEL0.0 = 1	P2_DIR.0 = 1
P2.1	General purpose input	P2_IN.1		P2_DIR.1 = 0
	General purpose output	P2_OUT.1	P2_ALTSEL0.1 = 0	P2_DIR.1 = 1
	Ethernet transmit data output 1MIITXD1	Ethernet	P2_ALTSEL0.1 = 1	P2_DIR.1 = 1
P2.2	General purpose input	P2_IN.2		P2_DIR.2 = 0
	General purpose output	P2_OUT.2	P2_ALTSEL0.2 = 0	P2_DIR.2 = 1
	Ethernet transmit data output 2 MIITXD2	Ethernet	P2_ALTSEL0.2 = 1	P2_DIR.2 = 1
P2.3	General purpose input	P2_IN.3		P2_DIR.3 = 0
	General purpose output	P2_OUT.3	P2_ALTSEL0.3 = 0	P2_DIR.3 = 1
	Ethernet transmit data output 3 MIITXD3	Ethernet	P2_ALTSEL0.3 = 1	P2_DIR.3 = 1
P2.4	General purpose input	P2_IN.4		P2_DIR.4 = 0
	General purpose output	P2_OUT.4	P2_ALTSEL0.4 = 0	P2_DIR.4 = 1
	Ethernet transmit error output MIITXER	Ethernet	P2_ALTSEL0.4 = 1	P2_DIR.4 = 1
P2.5	General purpose input	P2_IN.5		P2_DIR.5 = 0
	General purpose output	P2_OUT.5	P2_ALTSEL0.5 = 0	P2_DIR.5 = 1
	Ethernet transmit enable output MIITXEN	Ethernet	P2_ALTSEL0.5 = 1	P2_DIR.5 = 1

Table 13-8 Port 2 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P2.6	General purpose input	P2_IN.6		P2_DIR.6 = 0
	General purpose output	P2_OUT.6	P2_ALTSEL0.6 = 0	P2_DIR.6 = 1
	Ethernet management data clock output MIIMDC	Ethernet	P2_ALTSEL0.6 = 1	P2_DIR.6 = 1
P2.7	General purpose input	P2_IN.7		P2_DIR.7 = 0
	General purpose output	P2_OUT.7	P2_ALTSEL0.7 = 0	P2_DIR.7 = 1
	MMCI power supply enable output VDDEN	MMCI	P2_ALTSEL0.7 = 1	P2_DIR.7 = 1
P2.8	General purpose input	P2_IN.8		P2_DIR.8 = 0
	General purpose output	P2_OUT.8	P2_ALTSEL0.8 = 0	P2_DIR.8 = 1
	Ethernet receive data valid input MIIRXDV	Ethernet	P2_ALTSEL0.8 = 1	P2_DIR.8 = 0
P2.9	General purpose input	P2_IN.9		P2_DIR.9 = 0
	General purpose output	P2_OUT.9	P2_ALTSEL0.9 = 0	P2_DIR.9 = 1
	Ethernet carrier input MIICRS	Ethernet		P2_DIR.9 = 0
P2.10	General purpose input	P2_IN.10		P2_DIR.10 = 0
	General purpose output	P2_OUT.10	P2_ALTSEL0.10 = 0	P2_DIR.10 = 1
	Ethernet collision input MIICOL	Ethernet		P2_DIR.10 = 0
P2.11	General purpose input	P2_IN.11		P2_DIR.11 = 0
	General purpose output	P2_OUT.11	P2_ALTSEL0.11 = 0	P2_DIR.11 = 1
	Ethernet receive data input 0 MIIRXD0	Ethernet		P2_DIR.11 = 0

Table 13-8 Port 2 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P2.12	General purpose input	P2_IN.12		P2_DIR.12 = 0
	General purpose output	P2_OUT.12	P2_ALTSEL0.12 = 0	P2_DIR.12 = 1
	Ethernet receive data input 1 MIIRXD1	Ethernet		P2_DIR.12 = 0
P2.13	General purpose input	P2_IN.13		P2_DIR.13 = 0
	General purpose output	P2_OUT.13	P2_ALTSEL0.13 = 0	P2_DIR.13 = 1
	Ethernet receive data input 2 MIIRXD2	Ethernet		P2_DIR.13 = 0
P2.14	General purpose input	P2_IN.14		P2_DIR.14 = 0
	General purpose output	P2_OUT.14	P2_ALTSEL0.14 = 0	P2_DIR.14 = 1
	Ethernet receive data input 3 MIIRXD3	Ethernet		P2_DIR.14 = 0
P2.15	General purpose input	P2_IN.15		P2_DIR.15 = 0
	General purpose output	P2_OUT.15	P2_ALTSEL0.15 = 0	P2_DIR.15 = 1
	Ethernet receive error input MIIRXER	Ethernet		P2_DIR.15 = 0

13.6 Port 3

Port 3 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as input for the external interrupts. For further details regards the interrupt control, please refer to Chapter “Interrupt System”.

13.6.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices
- Selectable open-drain function

13.6.2 Registers

The following port kernel registers are available at Port 3:

Table 13-9 Port 3 Kernel Registers

Register Short Name	Register Long Name
P3	Port 3 Data Output Register
P3_IN	Port 3 Data Input Register
P3_DIR	Port 3 Direction Register
P3_OD	Port 3 Open Drain Control Register

13.6.3 Port Configuration and Function

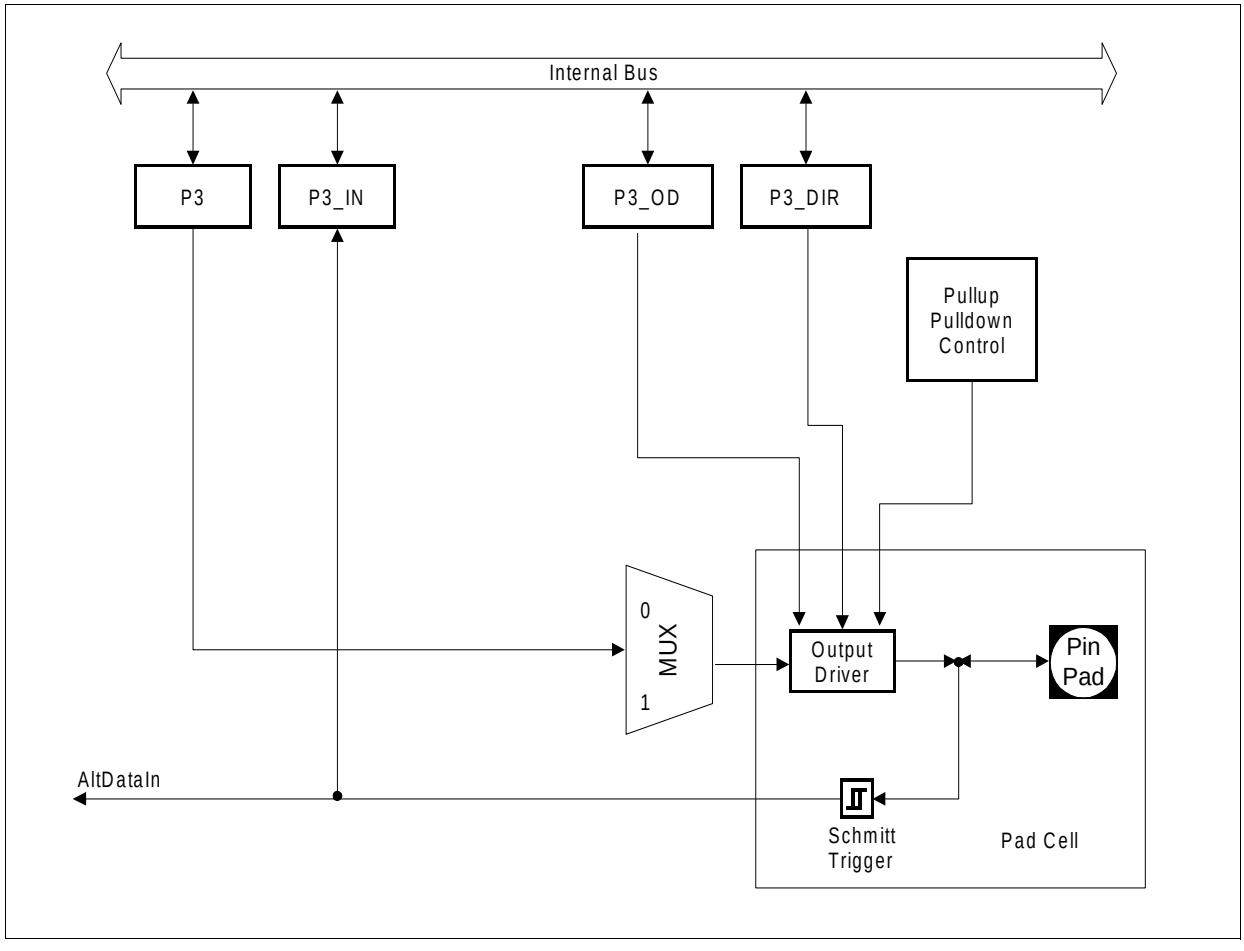


Figure 13-7 Port 3 Configuration

Table 13-10 Port 3 Functions

Port Pin	Pin Functionality	Associated Register	Direction Control
P3.x (x = 15-0)	General purpose input	P3_IN.x	P3_DIR.x = 0
	General purpose output	P3.x	P3_DIR.x = 1

13.7 Port 4

Port 4 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as input for the external interrupts. For further details regards the interrupt control, please refer to Chapter “Interrupt System”.

13.7.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices

13.7.2 Registers

The following port kernel registers are available at Port 4:

Table 13-11 Port 4 Kernel Registers

Register Short Name	Register Long Name
P4	Port 4 Data Output Register
P4_IN	Port 4 Data Input Register
P4_DIR	Port 4 Direction Register

13.7.3 Port Configuration and Function

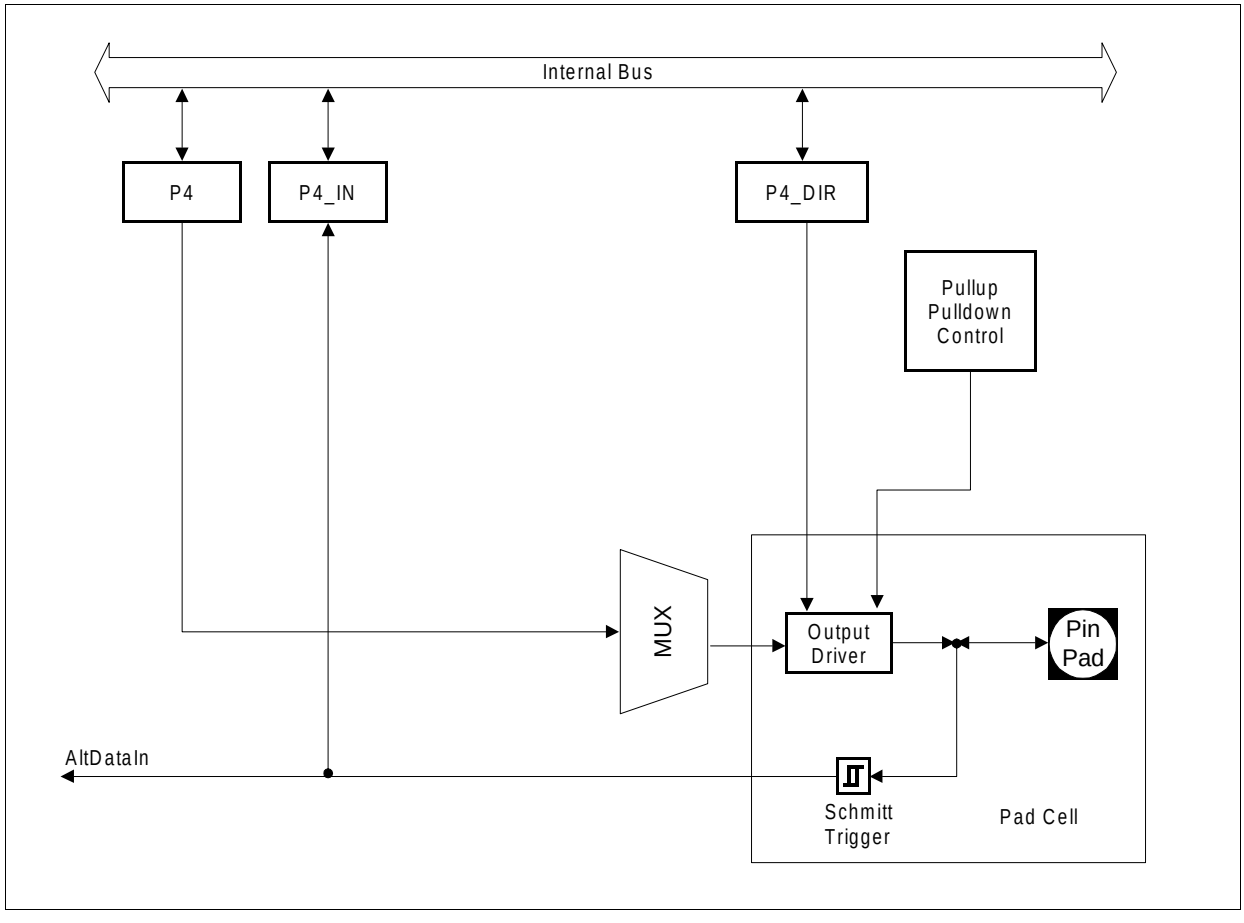


Figure 13-8 Port 4 Configuration

Table 13-12 Port 4 Functions

Port Pin	Pin Functionality	Associated Register	Direction Control
P4.x (x = 15-0)	General purpose input	P4_IN.x	P4_DIR.x = 0
	General purpose output	P4.x	P4_DIR.x = 1

13.8 Port 5

Port 5 is a 16-bit bidirectional I/O port. It serves as GPIO lines or as output of MMCI module. The port lines can be used as standard GPIO pins if its alternate function is not required.

13.8.1 Features

- Push/pull output drivers
- 3.3 Volt operation for GPIO
- Fixed port hold logic
- Fixed slew-rate and output driver strength
- Fixed pull-up/pull-down devices

13.8.2 Registers

The following port kernel registers are available at Port 5:

Table 13-13 Port 5 Kernel Registers

Register Short Name	Register Long Name
P5	Port 5 Data Output Register
P5_IN	Port 5 Data Input Register
P5_DIR	Port 5 Direction Register
P5_ALTSEL0	Port 5 Alternate Function Select Register 0
P5_ALTSEL1	Port 5 Alternate Function Select Register 1

13.8.3 Port Configuration and Function

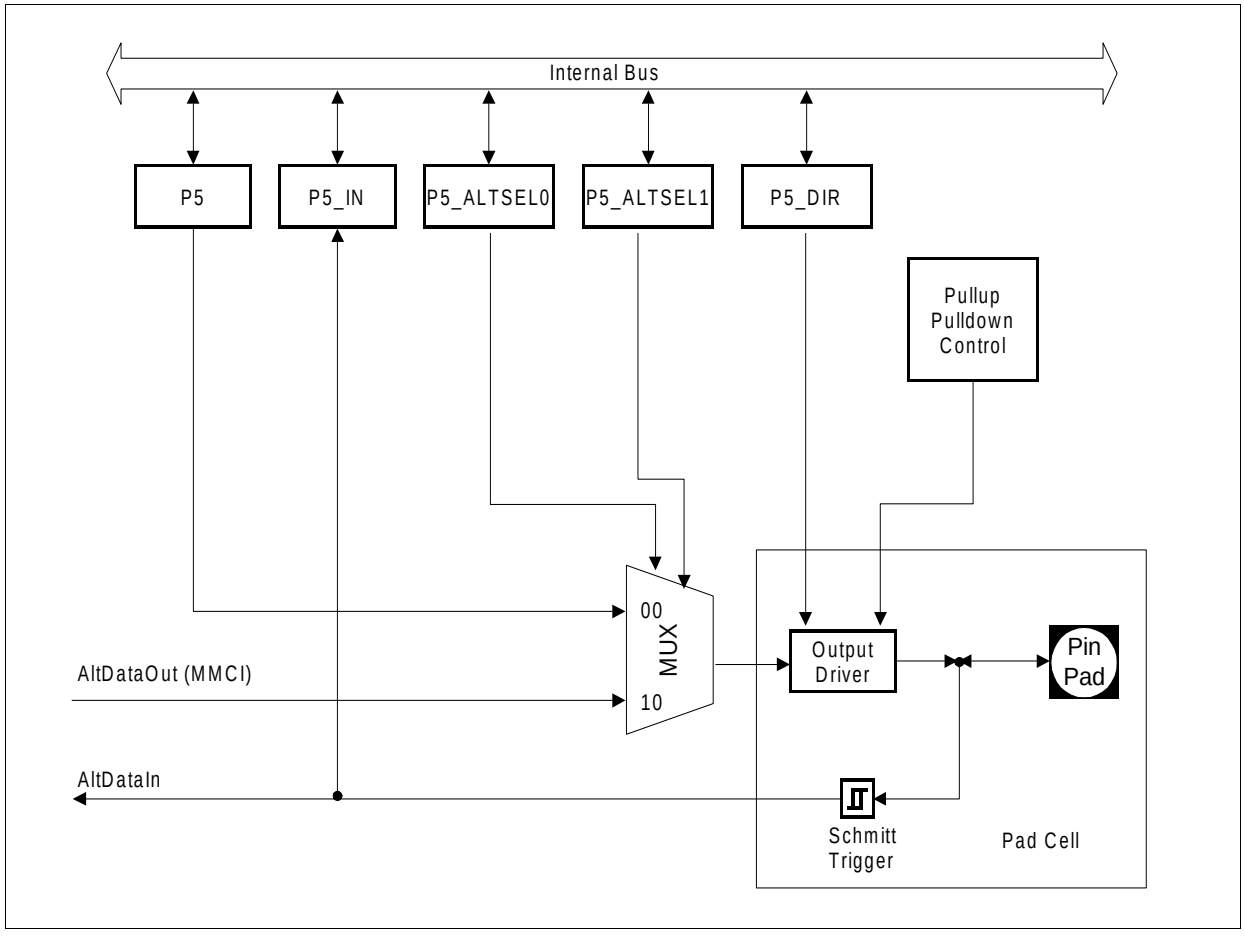


Figure 13-9 Port 5 Configuration

Table 13-14 Port 5 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P5.0	General purpose input	P5_IN.0		P5_DIR.0 = 0
	General purpose output	P5_OUT.0	P5_ALTSEL0.0 = 0 P5_ALTSEL1.0 = 0	P5_DIR.0 = 1
	MMCI data direction indicator output DATRW	MMCI	P5_ALTSEL0.0 = 0 P5_ALTSEL1.0 = 1	P5_DIR.0 = 1

Table 13-14 Port 5 Functions

Port Pin	Pin Functionality	Associated Register/Modul	Alternate Function	Direction Control
P5.1	General purpose input	P5_IN.1		P5_DIR.1 = 0
	General purpose output	P5_OUT.1	P5_ALTSEL0.1 = 0 P5_ALTSEL1.1 = 0	P5_DIR.1 = 1
P5.2	General purpose input	P5_IN.2		P5_DIR.2 = 0
	General purpose output	P5_OUT.2	P5_ALTSEL0.2 = 0 P5_ALTSEL1.2 = 0	P5_DIR.2 = 1
	MMCI command direction indicator output CMDRW	MMCI	P5_ALTSEL0.2 = 0 P5_ALTSEL1.2 = 1	P5_DIR.2 = 1
P5.x (x = 14-3)	General purpose input	P5_IN.x		P5_DIR.x = 0
	General purpose output	P5_OUT.x	P5_ALTSEL0.x = 0 P5_ALTSEL1.x = 0	P5_DIR.x = 1
P5.15	General purpose input	P5_IN.15		P5_DIR.15 = 0
	General purpose output	P5_OUT.15	P5_ALTSEL0.15 = 0 P5_ALTSEL1.15 = 0	P5_DIR.15 = 1
	MMCI command line mode indicator output ROD	MMCI	P2_ALTSEL0.15 = 1 P5_ALTSEL1.15 = 1	P5_DIR.15 = 1

14 External Bus Unit

The External Bus Control Unit (EBU) of the TC11IB is the interface between external memories and peripheral units and the internal memories and peripheral units. The basic structure of the EBU is shown in **Figure 14-1**.

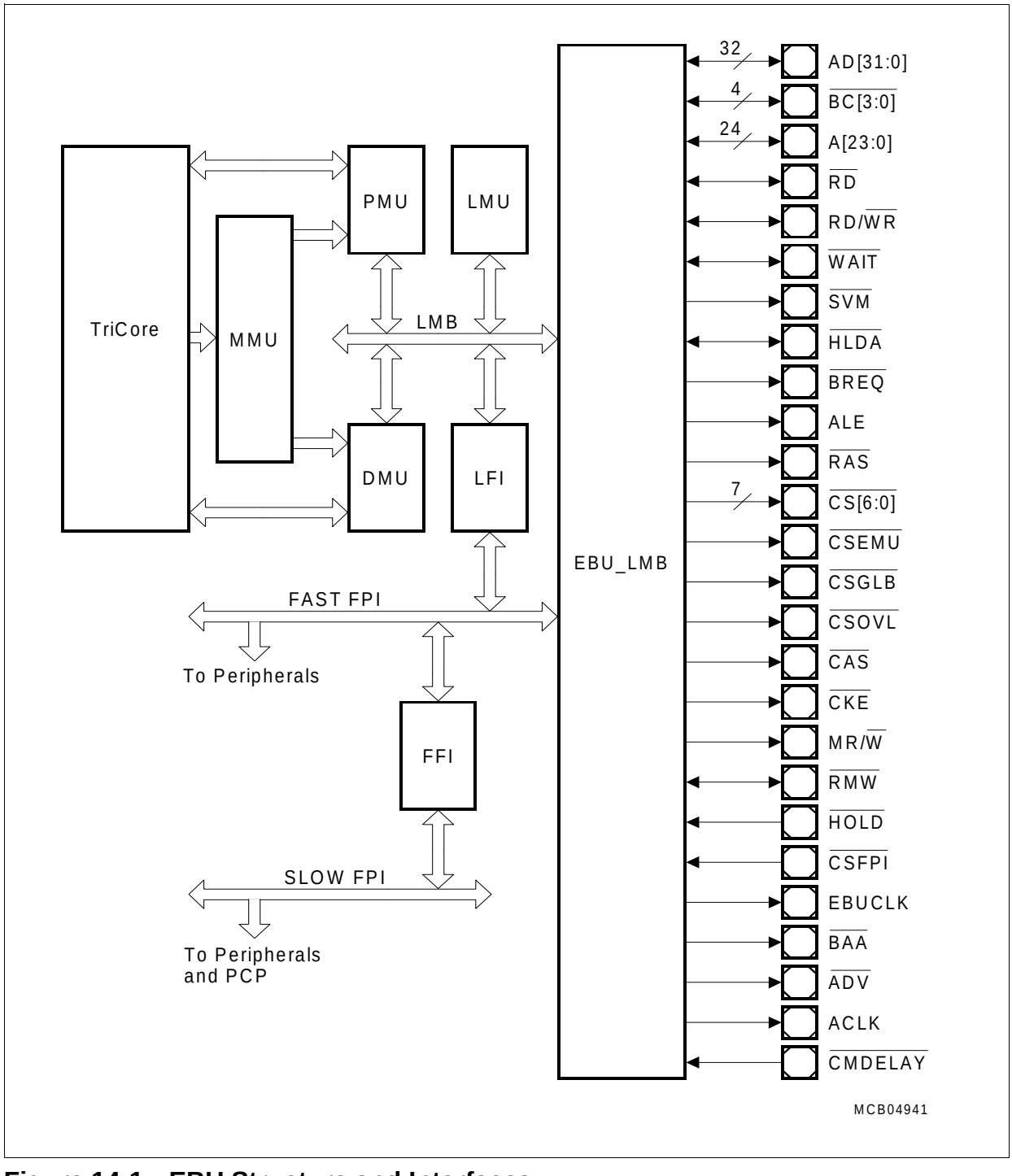


Figure 14-1 EBU Structure and Interfaces

The EBU is primarily used for the following two operations:

- Any LMB or FPI masters can access external memories through the EBU.
- An external master can only access internal devices through the FPI Bus

The EBU controls all transactions required for these two operations and in particular handles the arbitration between these two operations.

The types of external devices/Bus modes controlled by the EBU are:

  – Intel style peripherals (separate $\overline{RD}$ and $\overline{WR}$ signals)
  – Motorola style peripherals (MR/ $\overline{W}$ signals)
  – ROMs, EPROMs
  – Static RAMs
  – PC 100 SDRAMs (Burst Read/Write Capacity/Multi-Bank/Page support)
  – Specific types of Burst Mode Flashes (Intel 28F800F3/28F160F3, AMD 29BL162)
  – Special support for external emulator/debug hardware

14.1 Overview

The External Bus Controller (EBU) connects the internal LMB bus and FPI bus and the external bus. The EBU is always a slave on the LMB Bus and either a master or slave on the FPI Bus. Any LMB of FPI masters thus can access external memories or devices through the EBU of TC11IB. The maximum length of the bursts is dependent upon the size of program and data cache lines, i.e. 8 x32-bit words. The EBU of TC11IB also supports shorter bursts of 4 and 2 of 32-bit words. Single transfers (non-burst) are supported for 8-bit, 16-bit and 32-bit wide access. The EBU of TC11IB also allows external master to access internal devices once it grabs the ownership of the external bus, but access is limited through the FPI bus. The EBU of the TC11IB

- Supports Local Memory Bus (LMB 64-bit)
- Supports External bus frequency up to 96 MHz. External bus frequency: internal LMB frequency = 1:1 or 1:2 or 1:4
- Provides highly programmable access parameters
- Supports Intel-and Motorola-style peripherals/devices
- Supports PC 100 SDRAM (burst access, multibanking, precharge, refresh)
- Supports16-and 32-bit SDRAM data bus and 64,128 and 256MBit devices
- Supports Burst Flash (Intel 28F800F3/160F3,AMD 29BL162)
- Supports Multiplexed access (address &data on the same bus) when PC 100 SDRAM is not presented on the External Bus
- Supports Address Alignment, external address space up to 64 MBytes.
- Supports Data Buffering: Code Prefetch Buffer, Read/Write Buffer.
- Is external master arbitration compatible to C166 and other Tricore devices
- Provides 8 programmable address regions (1 dedicated for emulator)
- Provides an $\overline{CSGLB}$ signal, bit programmable to combine one or more $\overline{CS}$ lines for buffer control
- Provides $\overline{RMW}$ signal reflecting read-modify-write action

- Supports Little- and Big-endian
- Provides signal for controlling data flow of slow-memory buffer
- Provides slave unit for external (off-chip) master to access devices on FPI bus

If PC100 compatibility is used, only SDRAM devices can be connected directly to the EBU pins. Other devices must be connected through buffers. But when PC 100 SDRAM devices are not connected, these buffers are not needed for other devices. Additionally, the EBU provides special support for external emulator and debugging hardware.

The external bus established by the EBU consists of a 32-bit wide data bus, a 24-bit wide address bus, and a number of control signals. With eight user Chip Select lines, eight external address ranges can be accessed, each with a size of up to 64 MBytes (besides the special emulator range). Each of these ranges can be programmed individually in terms of location, size and access parameters (such as data size, address mode, wait states, etc.), making it possible to connect and access different device types in one system. The EBU dynamically adjusts the access sequence according to the programmed parameters for each selectable device.

14.2 EBU Features

- 32-bit wide data bus (D[31:0])
  - Data width of external device can be 8, 16 or 32 bits
  - Automatic data assembly/disassembly operation
  - Demultiplexed or multiplexed (address and data on the same bus) operation
- 24-bit wide address bus (A[23:0])
- Bus control signals
  - EBU clock output (EBUCLK)
  - Additional clock output (ACLK)
  - Address latch enable (ALE)
  - Read ($\overline{RD}$) and read/write (RD/$\overline{WR}$)
  - Read/Modify/Write signal ($\overline{RMW}$)
  - Four byte control signals ($\overline{BC}$[3:0])
  - Seven user Chip Selects ($\overline{CS}$[6:0])
  - Buffer control signal ($\overline{CSGLB}$)
  - External synchronous/asynchronous wait state control ($\overline{WAIT}$)
  - Motorola-style device control (MR/$\overline{W}$)
  - Non-standard devices control signal ($\overline{CMDELAY}$)
  - Protection signal ($\overline{SVM}$)
- SDRAM control signals
  - SDRAM Devices clock signal ($\overline{CKE}$)
  - SDRAM row address strobe signal ($\overline{RAS}$)
  - SDRAM column address strobe signal ($\overline{CAS}$)
- Burst Flash control signals
  - Burst Flash address valid signal ($\overline{ADV}$)

  - Burst address advance signal ($\overline{BAA}$)
- 7 user address ranges
  - Programmable location and size
  - Individual Chip Select for each range
  - Programmable mirror function: the same physical device can be accessed in two different address ranges
  - Enable/disable control
- Programmable access parameters for each address range
  - Address mode (multiplexed/demultiplexed)
  - Data width
  - Byte control signal operation
  - Address setup and hold timing
  - Data hold wait states
  - Read/write wait states
  - Recovery cycle wait states
  - External $\overline{WAIT}$ input enable and active level control, asynchronous or synchronous operation
  - Write protection for region
- Programmable wait state insertion to meet recovery/tristate time needs of external devices between
  - Read and write accesses
  - Accesses to different address ranges
- External bus arbitration
  - Simple three-line interface: bus hold ($\overline{HOLD}$), hold acknowledge ($\overline{HLDA}$) and bus request ($\overline{BREQ}$) signals
  - External bus master/slave arbitration operation
  - External master can access EBU with special Chip Select ($\overline{CSFPI}$) to access on-chip devices connected to the FPI Bus
- Automatic self-configuration on boot from external memory
  - Reads configuration data from external memory
- Dedicated emulation support
  - Emulator address range
  - Emulator memory Chip Select ($\overline{CSEMU}$)
  - Overlay Chip Select for emulator memory ($\overline{CSOVL}$)
  - Special boot from emulation memory

14.3 Basic EBU Operation

The EBU is the interface or gateway from the internal on-chip system to the external on-board system. But the EBU can operate in both directions, it is also a gateway from the external world to the internal on-chip system. **Figure 14-2** shows an example for the connection of an external system, including an external bus master, to the EBU. Note:

not all signals are shown in this diagram. For example, the connections from the external master to the Chip Select ($\overline{CSn}$) lines are not shown.

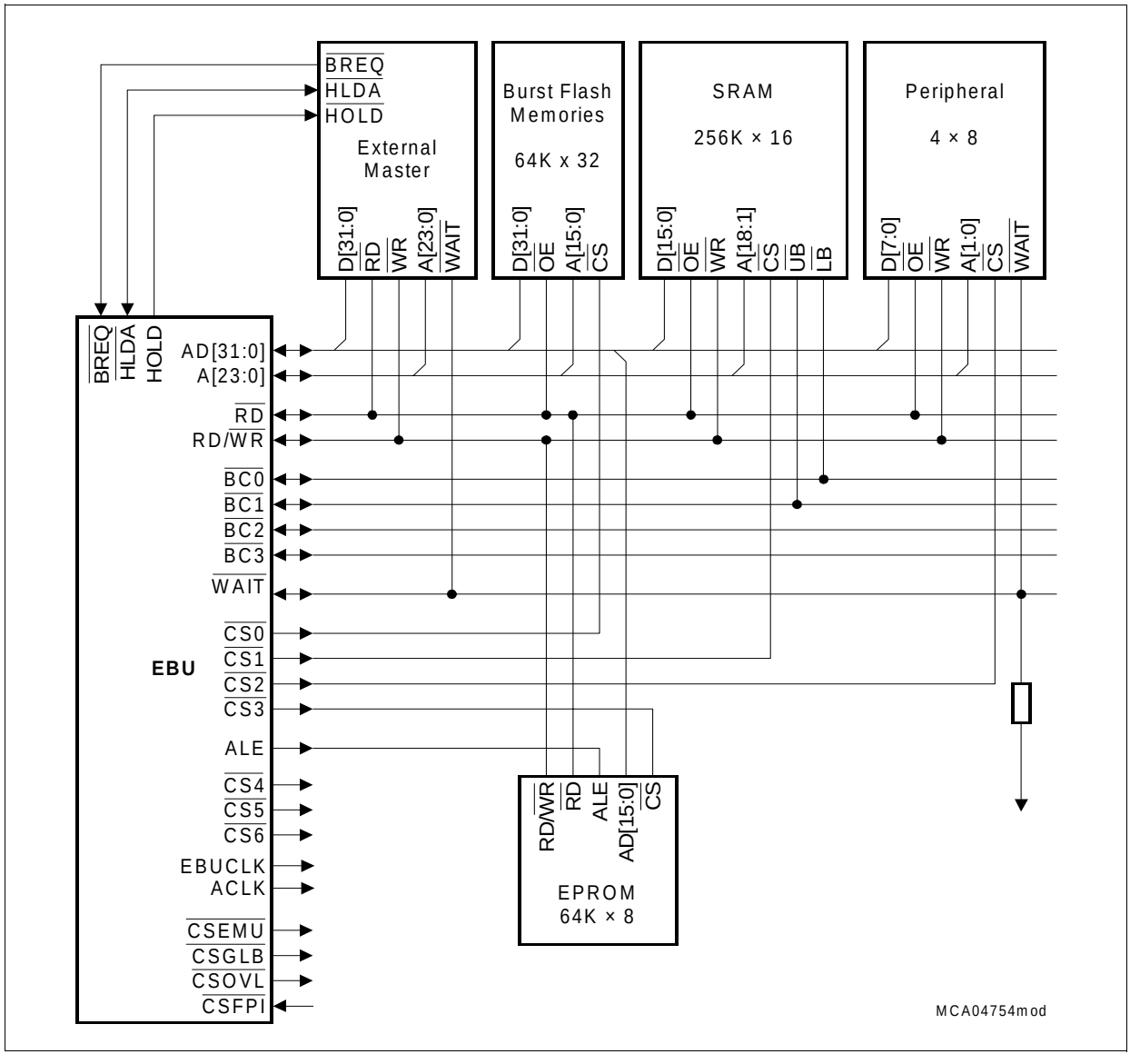


Figure 14-2 Example Configuration for Connection of External Devices

*Note: The example given in **Figure 14-2** is valid for small systems with a low capacity (ca. 30 pF max.). For larger systems and high frequency applications, the external bus must be separated by additional buffers into a fast section (that is, low capacity, maximum capacitance of 30 pF, and 0 wait states) and a slow section (that is, high capacity, with wait states).*

The basic operation of the EBU in these two fundamental modes is described in the following sections. It is assumed for these descriptions that there are no blocking conditions for the operations (such as the EBU is busy, arbitration conflict, etc.), and that

no bus arbitration is required. In later sections of this chapter, operation of the EBU is described in more detail, and all special conditions and prerequisites as well as bus arbitration procedures are considered.

14.3.1 Internal to External Operation

When an internal LMB Bus master wants to perform a read or write transaction from/to a device connected to the external bus, it sends out the address onto the LMB Bus. The address needs to be in the address ranges defined as external, as shown in **Table 14-1**.

Table 14-1 EBU External Address Ranges

Segment	Address Range	Description
8	$8000\ 0000_H$ - $8FFF\ FFFF_H$	External memory space (cached area)
10	$A000\ 0000_H$ - $AFBF\ FFFF_H$	External memory space (non-cached area)
13	$D800\ 0000_H$ - $DDFF\ FFFF_H$	External peripheral and data memory space (non-cached area)
	$DE00\ 0000_H$ - $DEFF\ FFFF_H$	External emulator memory (non-cached area)
14	$E000\ 0000_H$ - $E7FF\ FFFF_H$	External peripheral and data memory space (non-cached area)
15	$F800\ 0000_H$ - $F800\ 03FF_H$	EBU special control registers (non-cached area)

The EBU reacts to addresses in these ranges only. It compares the address sent from the LMB Bus master against the address ranges pre-programmed in its address select registers, EBU_ADDSELx. If it finds a match in one (or more) of the address regions, it selects the associated bus control register, EBU_BUSCONx and the associated bus access parameter register EBU_BUSAPx, for that region, and starts to perform the external access according to the parameters programmed in the EBU_BUSCONx and EBU_BUSAPx registers.

On a write operation, the write data from the LMB Bus master is stored inside the EBU, and the master can continue with its other tasks. The EBU will take care of properly storing the data to the external device.

On a read operation, the LMB Bus master must wait until the EBU has retrieved the data from the external device and has sent it to the master via the LMB Bus. The internal LMB Bus is blocked for that time, no other transaction can take place.

14.3.2 External to Internal Operation

If an external bus master wants to access a device connected to the external bus or to an internal module on the FPI Bus, it first needs to receive ownership of the external bus

from the EBU via the bus arbitration procedure. The EBU releases all signals of the external bus to the other bus master. Internal pull-up or pull-down devices connected to the EBU signals guarantee stable signal conditions during the transition phase, until the other bus master has taken over and drives the bus signals.

The external master then performs its transaction over the external bus. If the access is to the internal FPI Bus, the EBU is activated as a slave via a special Chip Select signal. It then acts as an bus master on the internal FPI Bus, performing the required access on behalf of the external bus master.

Three reprogrammable address extension registers are provided to extend the external 24-bit address to the full 32-bit FPI Bus address. the EBU offers the following memory map to the external master:

Table 14-2 EBU Memory Map in Slave Mode

Address A(23:0)	Resource
$C00000_H$ - $FFFFFF_H$	**FPI** The 32-bit FPI Target Address is generated via the EXTCON.AEXT3 field[1]. Byte, Half-Word or Word access is allowed (subject to support by the corresponding FPI resource).
800000_H - $BFFFFF_H$	**FPI** The 32-bit FPI Target Address is generated via the EXTCON.AEXT2 field[1].Byte, Half-Word or Word access is allowed (subject to support by the corresponding FPI resource).
400004_H - $7FFFFF_H$	**reserved**; should not be read or written.
400000_H	**EXTCON Register** 32-bit read/write access only (see **Section 14.12.8**).
000000_H - $3FFFFF_H$	**FPI** The 32-bit FPI Target Address is generated via the EXTCON.AEXT0 field[1]. Byte, Half-Word or Word access is allowed (subject to support by the corresponding FPI resource).

[1] According to the mapping scheme defined in **Section 14.6.2**

14.4 EBU Signal Description

The external signals of the EBU are listed in **Table 14-3** and described in the following sections.

Table 14-3 EBU Signals available on the TC11IB Ports

Signal	Type	Pull	Function
AD[31:0]	I/O	Up	Address/Data bus lines 31-0
A[23:0]	I/O	Up	Address bus lines 23-0
$\overline{\text{CS}}$[6:0]	O	Up	Chip Select n (n = 6-0)
$\overline{\text{CSEMU}}$	O	Up	Chip Select for emulation region selects external emulator memory region
$\overline{\text{CSOVL}}$	O	Up	Chip Select for overlay memory selects external overlay memory region
$\overline{\text{CSGLB}}$	O	Up	Chip Select Global
$\overline{\text{CSFPI}}$	I	Up	Chip Select FPI for external master selects the EBU as target in the slave mode
EBUCLK	O	–	External bus clock
ACLK	O	–	Additional clock
$\overline{\text{RD}}$	I/O	Up	Read control line; active during read operation
RD/$\overline{\text{WR}}$	I/O	Up	Write control line; active during write operation
ALE	O	Down	Address latch enable
$\overline{\text{ADV}}$	O	up	Address valid strobe
MR/$\overline{\text{W}}$	O	Up	Motorola-style read/write control line
$\overline{\text{BC0}}$	I/O	Up	Byte control line n (n = 3-0) controls the byte access to corresponding byte location
$\overline{\text{BC1}}$	I/O	Up	
$\overline{\text{BC2}}$	I/O	Up	
$\overline{\text{BC3}}$	I/O	Up	
$\overline{\text{WAIT}}$	I/O	Up	Wait for inserting wait states
$\overline{\text{CMDELAY}}$	I	Up	Command delay for inserting delays input
$\overline{\text{BAA}}$	O	Up	Burst address advance output
$\overline{\text{RMW}}$	O	Up	Read/Modify/Write signal output
$\overline{\text{HOLD}}$	I	Up	Hold request input
$\overline{\text{HLDA}}$	I/O	Up	Hold acknowledge input/output
$\overline{\text{BREQ}}$	O	Up	Bus request output
CKE	O	Up	Clock enable for SDRAM output
$\overline{\text{RAS}}$	O	Up	Row address strobe for SDRAM output

Table 14-3 EBU Signals available on the TC11IB Ports (cont'd)

Signal	Type	Pull	Function
$\overline{\text{CAS}}$	O	Up	Column address strobe for SDRAM output
$\overline{\text{SVM}}$	O	Up	Supervisor mode output

14.4.1 Address Bus, A[23:0]

The address bus of the EBU consists of 24 address lines, giving a directly addressable range of 16 MBytes. Directly addressable means that these address lines can be used to access any location within one external device, such as a memory. This external device is selected via one of the Chip Select lines. As there are seven Chip Selects, seven such devices with up to 16 MBytes of address range can be used in the external system.

Note: The address bus outputs the address in multiplexed mode.

If an external bus master is used in the system, and this master performs an access to the internal bus via the EBU, the address bus is switched to input. A special mechanism, described in **Section 14.6.2**, is provided to extend the external 24 address lines to the full 32-bit internal address.

14.4.2 Address/Data Bus, AD[31:0]

The Address/Data bus transfers data information in demultiplexed mode, and transfers address and data information in multiplexed mode. The width of this bus is 32 bits. External devices with 8, 16, or 32 bits of data width can be connected to the data bus. The EBU adjusts the data on the data bus to the width of the external device, according to the programmed parameters in its control registers. See **Section 14.5.4** for more information. The byte control signals, $\overline{\text{BC}}x$, specify which part of the data bus carries valid data. See also **Section 14.4.5**.

In multiplexed mode, the 32-bit address is first output on the bus. The bus is then set to input on a read access, or the data is output on a write access. Signal ALE captures the address from the bus either by the external device itself or into an external address latch.

Note: In multiplexed mode, only the lower 26 lines of this 32-bit bus are used to transfer the address. The upper 6 lines are valid but irrelevant.

14.4.3 Read/Write Strobes, $\overline{\text{RD}}$ and RD/$\overline{\text{WR}}$

Two lines are provided to trigger the read ($\overline{\text{RD}}$) and write (RD/$\overline{\text{WR}}$) operations of external devices. While some read/write devices require both signals, there are devices with only one control input. The RD/$\overline{\text{WR}}$ line is then used for these devices. This line will go to an active low level on a write, and will stay inactive high on a read. The external device should only evaluate this signal in conjunction with an active Chip Select. Thus, an active

Chip Select in combination with a high level on the RD/$\overline{WR}$ line indicates a read access to this device.

14.4.4 Address Latch Enable, ALE

This signal is used to indicate a valid address on the address bus A[23:0] (demultiplexed mode) or address/data bus AD[31:0] (multiplexed mode). The high-to-low transition of this signal is used to capture the address in an external address latch (transparent latch) or the external device. The length of ALE is programmable to accommodate timing requirements of the external device.

14.4.5 Byte Control Signals, $\overline{BCx}$

The byte control signals $\overline{BC[3:0]}$ select the appropriate byte lanes of the data bus for both read and write accesses. **Table 14-4** shows the activation on access to a 32-bit, 16-bit or 8-bit external device. Please note that this scheme supports little-endian devices.

Table 14-4 Byte Control Pin Usage

Width of External Device	BC3	BC2	BC1	BC0
32-bit device with byte write capability	AD[31:24]	AD[23:16]	AD[15:8]	AD[7:0]
16-bit device with byte write capability	inactive (high)	inactive (high)	AD[15:8]	AD[7:0]
8-bit device	inactive (high)	inactive (high)	inactive (high)	AD[7:0]

Signals $\overline{BCx}$ can be programmed for different timing. The available modes cover a wide range of external devices, such as RAM with separate byte write-enable signals, and RAM with separate byte Chip Select signals. This allows external devices to connect without any external “glue” logic. Refer to **Table 14-5** for byte-control timing.

Table 14-5 Byte Control Signal Timing Options

Programmed Mode	$\overline{BCx}$ Signal Timing
Chip Select Mode	$\overline{BCx}$ signals have the same timing as the generated Chip Select $\overline{CS}$.
Control Mode	$\overline{BCx}$ signals have the same timing as the generated control signals $\overline{RD}$ or RD/$\overline{WR}$.
Write Enable Mode	$\overline{BCx}$ signals have the same timing as the generated control signal RD/$\overline{WR}$.

14.4.6 External Extension of the Command Delay, $\overline{\text{CMDELAY}}$

The $\overline{\text{CMDELAY}}$ input can be used to extend the Command Delay Phase, i.e. to cause the EBU to insert additional cycles prior to the activation of the $\overline{\text{RD}}$, RD/$\overline{\text{WR}}$ control lines. This signal can programmed on a region to region basis to be ignored or sampled either synchronously or asynchronously (selected via the BUSCON[6:0].XCMDDELAY and EMUBC.XCMDDELAY parameters). The signal will only take effect after the programmed number of Command Delay Phase cycles has passed (i.e. the signal can only be used to extend the phase, not to shorten it).

When programmed for synchronous operation, $\overline{\text{CMDELAY}}$ is sampled on every rising edge of LMB Clock during the Command Delay phase. This sampled value is then used on the next rising edge of LMB Clock to decide whether to prolong the Command Delay Phase or to start the next phase.

Note: Due to the one-cycle delay in Synchronous Mode between the sampling of the $\overline{\text{CMDELAY}}$ input and its evaluation by the EBU, the Command Delay Phase must always be programmed to be at least one LMB Clock cycle (via the BUSAP[6:0].CMDDELAY and EMUBAP.CMDDELAY bit fields) in this mode.

When programmed for asynchronous operation, $\overline{\text{CMDELAY}}$ is also sampled at each rising edge of LMB Clock during the Command Phase. However an extra cycle is inserted for synchronization prior to the use of the sampled value (i.e. the sampled value is not used until the second following rising edge of LMB Clock). This minimizes the chance of the propagation of metastable signals into the module due to the $\overline{\text{CMDELAY}}$ signal changing at or around the same time as the rising edge of LMB Clock. This, in turn, allows the use of asynchronous input signals.

Note: Due to the two-cycle delay in Asynchronous Mode between the sampling of the $\overline{\text{CMDELAY}}$ input and its evaluation by the EBU, the Command Delay Phase must always be programmed to be at least two LMB Clock cycles (via the BUSAP[6:0].CMDDELAY and EMUBAP.CMDDELAY bit fields) in this mode.

14.4.7 Variable Wait State Control, $\overline{\text{WAIT}}$

This is an input signal to the EBU allowing the external device to force the EBU to insert additional wait states prior to deactivation of the $\overline{\text{RD}}$, RD/$\overline{\text{WR}}$ lines. $\overline{\text{WAIT}}$ can be enabled/disabled on a region to region basis by software and programmed to be active low or active high (the active level forces additional wait states). The BUSCON[6:0].WAITINV and EMUBC.WAITINV bits are used to select the desired polarity. Its sampling by the EBU can be selected to be synchronous or asynchronous (selected via the BUSCON[6:0].WAIT and EMUBC.WAIT parameters).

A fixed number of initial wait states should be programmed for the access because the external device usually requires time to react to an access and properly set $\overline{\text{WAIT}}$ to the appropriate level, and because the EBU requires time to sample and react to the $\overline{\text{WAIT}}$ signal.

If synchronous mode is selected, $\overline{WAIT}$ is sampled on the rising edge of LMB Clock so that it can be evaluated at the next rising edge of the clock. Due to the one cycle delay in Synchronous Mode between the sampling of the $\overline{WAIT}$ input and its evaluation by the EBU, the wait states must always be programmed to be at least one LMB Clock cycle via the BUSAP[6:0].WAITRDC, BUSAP[6:0].WAITWRC, EMUBAP.WAITRDC and EMUBAP.WAITWRC bit fields. In asynchronous mode, the $\overline{WAIT}$ signal is also sampled at each rising edge of LMB Clock. However, an extra cycle is inserted for synchronization prior to the use of the sampled value, so that it minimizes the chance of the propagation of metastable signals into the module due to the $\overline{WAIT}$ signal changing at or around the same time as the rising edge of LMB Clock. Thus, asynchronous operation of $\overline{WAIT}$ may result in one additional wait state compared to synchronous operation. Due to the two-cycle delay in Asynchronous Mode between the sampling of the $\overline{WAIT}$ input and its evaluation by the EBU, the wait states must always be at least two LMB Clock cycles. This is programmed via the BUSAP[6:0].WAITRDC, BUSAP[6:0].WAITWRC, EMUBAP.WAITRDC and EMUBAP.WAITWRC bit fields.

14.4.8 Chip Select Lines, $\overline{CSx}$, $\overline{CSGLB}$

The EBU provides seven user Chip Selects, $\overline{CS0}$, $\overline{CS1}$, $\overline{CS2}$, $\overline{CS3}$, $\overline{CS4}$, $\overline{CS5}$, and $\overline{CS6}$. The address ranges for which these Chip Selects are generated are programmed via the address select registers, EBU_ADDSELx, in a very flexible way (see **Section 14.5.1**).

Chip Select line $\overline{CSGLB}$ can be programmed to combine one or more the above $\overline{CSx}$ lines. This signal can be used to control a buffer located between the EBU and slow memory/peripheral devices when using PC100 SDRAM.

If overlapping address regions are programmed in the EBU_ADDSELx registers, only one Chip Select—the one with the lower number (higher priority)—will be activated on an access within the overlapping address range.

If the number of Chip Select lines is not sufficient, additional Chip Select signals can be generated by combining one Chip Select output with some address bits. In this case, all generated Chip Selects must share the same EBU timing and data width parameters. **Figure 14-3** shows how $\overline{CS3}$ can be divided into four smaller regions. Using this solution, the regions must be of equal size.

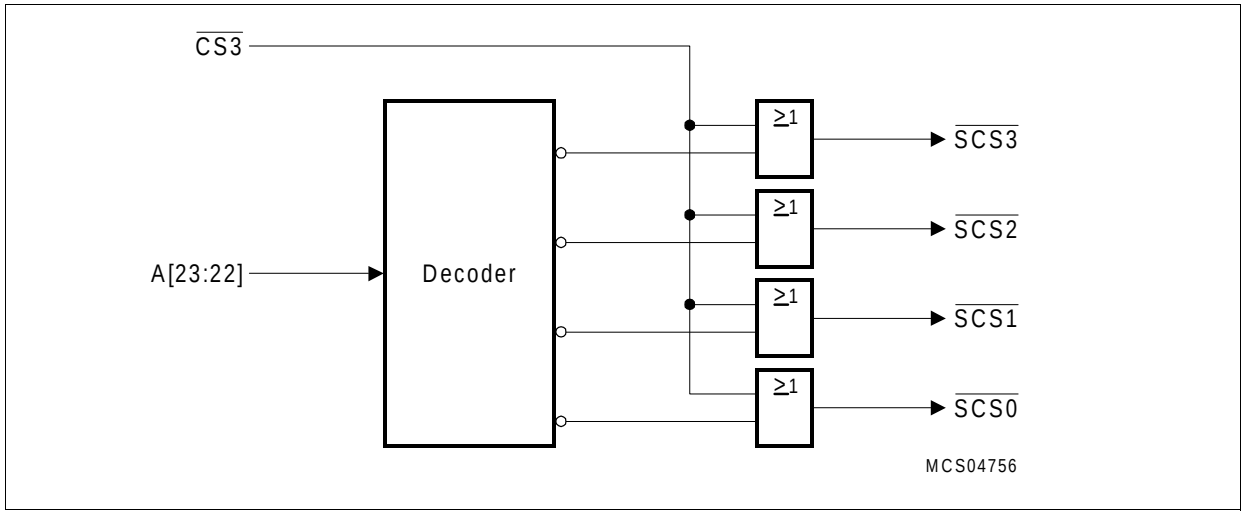


Figure 14-3 Simple Chip Select Expansion

14.4.9 EBU Arbitration Signals, HOLD, HLDA and BREQ

These signals are used by the EBU to negotiate ownership of the external bus with another external bus master. The HOLD signal (Hold Request) is used to request release of the bus from the EBU. If done so, the EBU acknowledges it with signal HLDA (Hold Acknowledge). Signal BREQ (Bus Request) is used by the EBU to signal its desire to get bus ownership to the external bus master.

More detailed descriptions of these signals and the bus arbitration modes of the EBU can be found in **Section 14.7**.

14.4.10 EBU Chip Select, CSFPI

An external bus master has the option to access modules connected to the internal FPI Bus of the TC11IB. To do so, it first must arbitrate for ownership of the external bus. Then, it accesses the external bus with an appropriate address and activates CSFPI to inform the EBU that the access is to be performed from the external bus onto the internal FPI Bus. In this case, the EBU acts as a slave on the external bus, but as a master on the internal FPI Bus. It performs the FPI Bus transaction on behalf of the external bus master. Refer to **Section 14.6.3** and its subsections for more information on this signal.

14.4.11 Emulation Support Signals, CSEMU and CSOVL

To support emulation and debugging, the EBU provides a special emulator memory Chip Select, CSEMU, and an overlay memory Chip Select, CSOVL. A detailed description of these signals can be found in **Section 14.5.1**.

Note: These signals are intended solely for the purpose of emulation and debugging. Using these signals for normal application purposes may result in conflicts when

using emulators/debuggers, and may severely hinder proper debugging. It is strongly recommended to exclude these signals from normal application usage.

14.5 Detailed Internal to External EBU Operation (Master Mode)

The following subsections provide more insight into the operation of the EBU for internal to external transactions. In the master mode, the EBU supports interconnection to a wide variety of devices with flexible programming of the access parameters. The types of external access cycle provided by the EBU include:

- Asynchronous devices—demuxed and muxed accesses like ROMs, E²PROMs, SRAMs, peripherals, etc.
- Burst mode Flash devices
- SDRAM devices

Examples of external memories/peripherals include:-

- INTEL style peripherals (separate RD and WR signals).
- Motorola style peripherals (MR/W signals).
- ROMs, EPROMs.
- Static RAMs.
- SDRAMs (burst read/write capability/multi bank/page support).
- Specific types of Burst Mode Flashes.
- Multiplexed address/data bus peripherals.

Each (internal) LMB/FPI master can access external devices via the EBU.The EBU provides a number of user programmable external memory regions each with an associated individual Chip Select signal (see **Section 14.5.1**). An LMB/FPI transaction that matches one of these user programmable external memory regions will be translated by the EBU to the appropriate external access(es). The type of transfer and the parameters of the external bus transaction are flexible and programmable on a region by region basis.

If PC100 compatibility is required, then only the PC100 SDRAM devices can be connected directly to the EBU pins. In this case, other memory/peripheral devices must be connected through buffers to minimize EBU pin loading. Buffers are not needed when PC100 SDRAM devices are not used.

14.5.1 EBU Address Regions

The EBU provides eight programmable address regions (including the emulator range), each with its own Chip Select. The access parameters for each of the region can be programmed individually to accommodate different types of external devices. Seven of these regions are provided for normal user application purposes, while the eighth one is reserved for emulator usage.

Three EBU registers and one Chip Select line are dedicated to each of the regions. The address range of the region is programmed through the address select register,

EBU_ADDSELx (x = 0..6). The access parameters for the external device in that region are programmed through the respective bus control register, EBU_BUSCONx, and bus access parameter register, EBU_BUSAPx. Each region can be defined as either normal asynchronous/demultiplexed, multiplexed, burst Flash, or SDRAM access. The access to the external device is performed using the associated Chip Select line, CSx. Additionally $\overline{\text{CSGLB}}$ can be programmed to combine one or more the above $\overline{\text{CS}}$ lines. This signal can be used to control the buffer located between the EBU of TC11IB and slow devices.

The EBU also provides an overlay memory Chip Select $\overline{\text{CSOVL}}$ to redirect accesses to the target system to another external memory. Both external devices must have the same access parameters (data width, address range, timing). $\overline{\text{CSOVL}}$ can be enabled for accesses to each defined region and must be used in the following way:

- $\overline{\text{CSOVL}}$ gates the $\overline{\text{RD}}$ and RD/$\overline{\text{WR}}$ signal to the target system. These signals can pass through only if $\overline{\text{CSOVL}}$ is inactive (‘1’), i.e. a read/write access to the target system is possible.
- $\overline{\text{CSOVL}}$ enables the external overlay memory (wired to the Chip Select of the memory), i.e. if $\overline{\text{CSOVL}}$ is active, a read/write access to the overlay memory will be performed (instead of the target system).

Identification of the region to be redirected is programmable through the **OVL** parameter in the EMUOVL register.

Table 14-6 summarizes the registers and Chip Selects associated with the eight regions.

Table 14-6 EBU Address Regions, Registers and Chip Selects

Address Region	Address Select Register	Bus Control Register	Bus Access Parameter Register	Chip Select
User region 0	EBU_ADDRSEL0	EBU_BUSCON0	EBU_BUSAP0	$\overline{\text{CS0}}$
User region 1	EBU_ADDRSEL1	EBU_BUSCON1	EBU_BUSAP1	$\overline{\text{CS1}}$
User region 2	EBU_ADDRSEL2	EBU_BUSCON2	EBU_BUSAP2	$\overline{\text{CS2}}$
User region 3	EBU_ADDRSEL3	EBU_BUSCON3	EBU_BUSAP3	$\overline{\text{CS3}}$
User region 4	EBU_ADDRSEL4	EBU_BUSCON4	EBU_BUSAP4	$\overline{\text{CS4}}$
User region 5	EBU_ADDRSEL5	EBU_BUSCON5	EBU_BUSAP5	$\overline{\text{CS5}}$
User region 6	EBU_ADDRSEL6	EBU_BUSCON6	EBU_BUSAP6	$\overline{\text{CS6}}$
Emulator region	EBU_EMUAS	EBU_EMUBC	EBU_EMUBAP	$\overline{\text{CSEMU}}$

14.5.1.1 Address Region Selection

Any LMB Bus address belonging to one of the external ranges shown in **Table 14-1** can activate the EBU (provided the EBU is idle). It picks up the address and compares it to

the eight address regions programmed through its address select registers (including the emulator range). Each address select register (EBU_ADDSELx, EBU_EMUAS) contains five bit fields. The bit field are:

- Bit REGEN is the enable control of that region. If the region is disabled (REGEN = 0), any address in that region presented to the EBU will result in a bus error reported back to the master requesting the access. Also, the Chip Select associated with that range is disabled.
- Bit field BASE specifies address bits A[31:12] of region x, where A[31:28] must only point to segments 8,10,13,14 and 15 which are covered by the EBU (see **Table 14-1**).
- Bit field MASK specifies how many bits of an LMB Bus address must match the contents of the BASE(x) bit field (to a maximum of 15, starting with A[26]). (Note that address bits A[31:27] must always match.) This parameter defines the length of a region.
- The ALTSEG bit field specifies an alternate segment for comparison with A(31:28). This means that A(31:28) is compared to **BASE**(31:28) and also to **ALTSEG**(11:8). For both case, A(27) must match **BASE**(27).
- Bit ALTEN indicates whether **ALTSEG** is valid or invalid.

Figure 14-4 illustrates how the comparison of the LMB Bus address to the address region setup in register EBU_ADDSELx/EBU_EMUAS is performed to determine whether or not a region is selected.

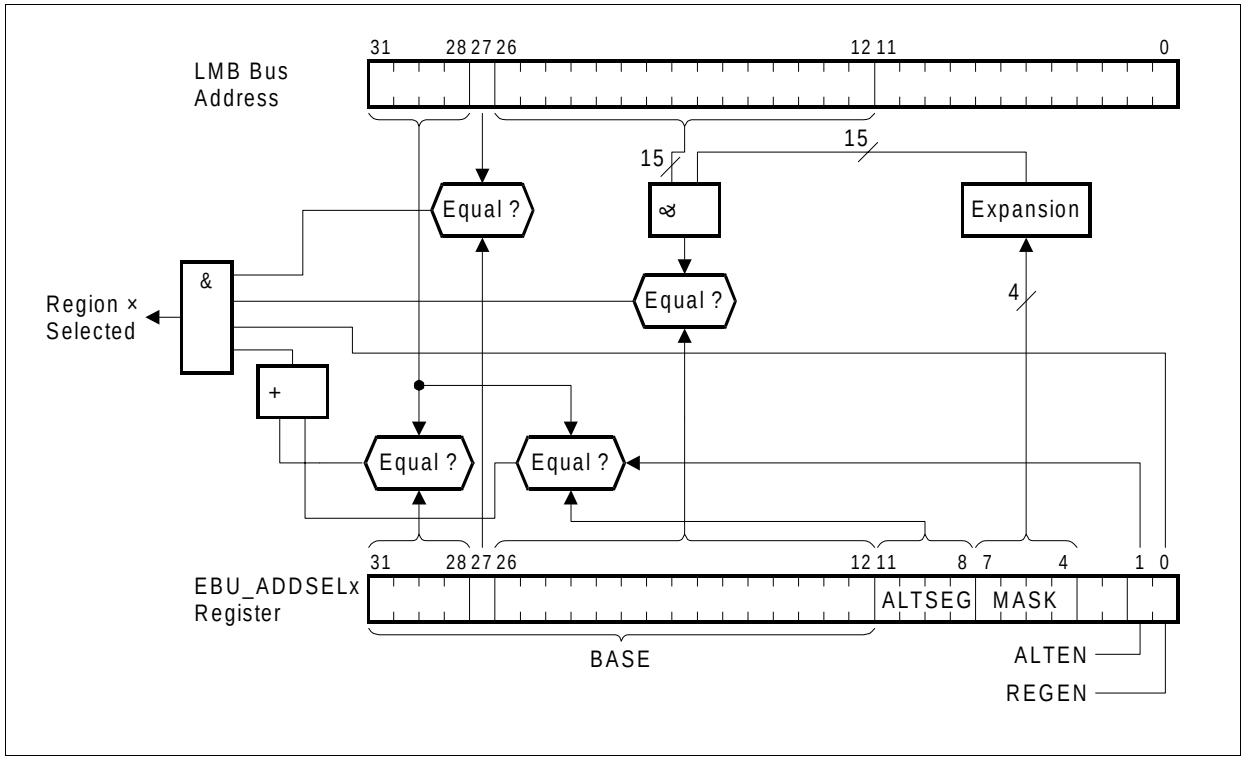


Figure 14-4 Address Region Selection

This address region scheme described above implies the following:

- The smallest possible address region is 2^{12} bytes (4 KBytes)
- The largest possible address region is 2^{27} bytes (128 MBytes)
- The start address of a region depends on the size of the region. It must be at an address which is a multiple of the size of a region; for example, the smallest region can be placed on any 4 KBytes boundary, while the largest region can be placed on 128-MByte boundaries only.

Table 14-7 shows the possible region sizes and start granularity, as determined by the programming of the MASK bit field. The range of the offset address within such a region is also given. Note that in demultiplexed mode, only addresses A[23:0] are actually output to the external system. In multiplexed mode, a 32-bit address is output on AD[31:0].

Table 14-7 EBU Address Regions Size and Start Address Relations

MASK	No. of Address Bits compared to BASE[26:12]	Range of Address Bits compared to BASE[26:12]	Region Size and Start Address Granularity	Range of Offset Address Bits within Region
1111_B	15	A[26:12]	4 KBytes	A[11:0]
1110_B	14	A[26:13]	8 KBytes	A[12:0]
1101_B	13	A[26:14]	16 KBytes	A[13:0]
1100_B	12	A[26:15]	32 KBytes	A[14:0]
1011_B	11	A[26:16]	64 KBytes	A[15:0]
1010_B	10	A[26:17]	128 KBytes	A[16:0]
1001_B	9	A[26:18]	256 KBytes	A[17:0]
1000_B	8	A[26:19]	512 KBytes	A[18:0]
0111_B	7	A[26:20]	1 MByte	A[19:0]
0110_B	6	A[26:21]	2 MBytes	A[20:0]
0101_B	5	A[26:22]	4 MBytes	A[21:0]
0100_B	4	A[26:23]	8 MBytes	A[22:0]
0011_B	3	A[26:24]	16 MBytes	A[23:0]
0010_B	2	A[26:25]	32 MBytes	A[24:0]
0001_B	1	A[26]	64 MBytes	A[25:0]
0000_B	0	–	128 MBytes	A[26:0]

Because of the scheme shown in **Table 14-7**, memory regions can overlap and there can be gaps between regions. In such cases, the EBU actions will be determined by the specific case, as follows.

- An address lies in EBU register region (LMB only):
  – The EBU will perform desired EBU register access.
- An address lies in exactly one defined region:
  – The EBU will perform the requested access to external memory.
- An address lies in more than one region (overlapping regions):
  – The access is performed to the region with higher priority where region 0 has the highest priority, region 7 has the lowest.
- The address does not lie in any region, or lies in a disabled region:
  – In case of an unknown external address or disabled region, the EBU will return an error-acknowledge code on the LMB/FPI Bus.

*Note: When mirrored segments are being defined, one must take care that there is no collision. TC11IB does not have a checking mechanism to ensure that each segment defined (either in **BASE**(31:28) or **ALTSEG**(11:8) or both) is exclusive. Therefore, one must ensure that each mapping from region 0 to 7 does not interfere with any other, otherwise only the mapping with the highest priority will take effect. Region 0 is the highest, while region 7 is the lowest.*

14.5.1.2 Address Region Parameters

When a LMB Bus address presented to the EBU belongs to one of its programmed active (enabled) address regions, the EBU performs the external bus access according to the global EBU parameters stored in register EBU_CON and according to the individual parameters stored in the bus control register, EBU_BUSCONx, and bus access

parameter register, EBU_BUSAPx, associated with that address region. The selection of the appropriate access parameters is shown in **Figure 14-5**.

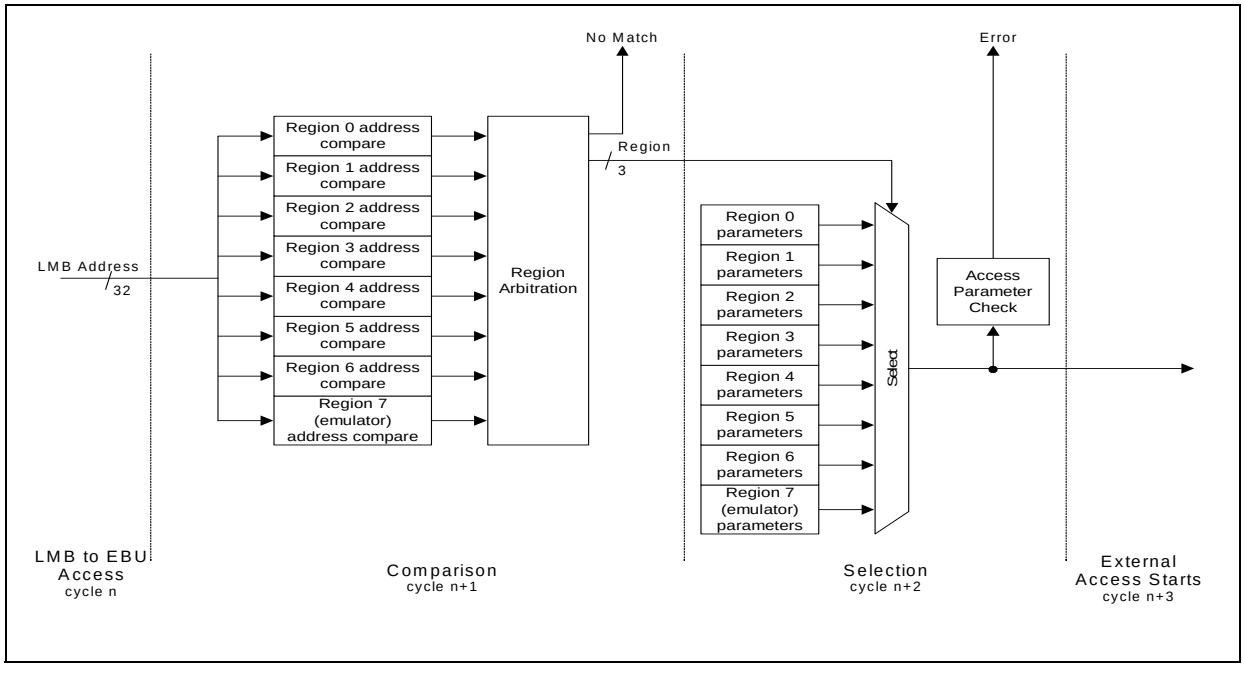


Figure 14-5 Access Parameter Selection Phases

At the end of the LMB/FPI address phase (cycle ‘n’ above), the 32-bit LMB address is made available to the region comparison logic. In the second phase (cycle ‘n + 1’ above), the region comparison logic determines whether the LMB/FPI address lies within an active region (or regions). If the address is not matched (i.e. is not within an active region) then the “No Match” signal causes the EBU to return an LMB/FPI Error Acknowledge and discard the access. Otherwise, the “Region” signal passes the region index number (in the case of multiple region matches, this is the index number of the lowest matching region) to the parameter selection logic. During the third phase (cycle ‘n + 2’ above), the parameter selection logic selects the appropriate access cycle parameters. At the end of this cycle, the access parameters associated with the highest priority region are checked. If the access is invalid, then the EBU returns an LMB/FPI Error Acknowledge and discards the access. Otherwise, the parameters are made available to the external bus driver logic. These parameters are then used to select/initialize the appropriate external access cycle state machine and also to make the selection of the external bus pins to be used for the access cycle. The next cycle (cycle ‘n + 3’ above) should see the actual start of the external bus access cycle on the external bus pins.

The part of programmable parameters for the support of eight external regions are listed in **Table 14-8**.

Table 14-8 LMB Master Mode Programmable Parameters

Parameter	Function	Register
AGEN	Access type for each external region: DEMULTIPLEXED, MULTIPLEXED, BURST_FLASH, SDRAM_TYPE0, SDRAM_TYPE1	**EBU_BUSCONx (x=0-6) EBU_EMUBC**
ALTSEG	Alternate segment defined for each external region	**EBU_ADDRSELx (x=0-6) EBU_EMUAS**
BASE	Base address for each external region which is used in conjunction with the mask parameter	**EBU_ADDRSELx (x=0-6) EBU_EMUAS**
MASK	Address mask for each external region. Specifies the number of right-most bits in the base address starting from bit 26.	**EBU_ADDRSELx (x=0-6) EBU_EMUAS**
ALTEN	Enable bit for alternate segment for each external region.	**EBU_ADDRSELx (x=0-6) EBU_EMUAS**
REGEN	Enable bit for each external region. A disabled region will always generate a miss during address comparison.	**EBU_ADDRSELx (x=0-6) EBU_EMUAS**
ENDIAN	The endian mode for each external region: LITTLE_ENDIAN or BIG_ENDIAN	**EBU_BUSCONx (x=0-6) EBU_EMUBC**
PORTW	The data width for each external region: 16_BIT or 32_BIT	**EBU_BUSCONx (x=0-6) EBU_EMUBC**
WR	To specify the write protection for each memory region: ON or OFF	**EBU_BUSCONx (x=0-6) EBU_EMUBC**
AALIGN	To enable address alignment for each memory region: ON or OFF	**EBU_BUSCONx (x=0-6) EBU_EMUBC**
GLBCS	To select one or more Chip Select lines to generate $\overline{\text{CSglb}}$	**EBU_CON**

Table 14-8 LMB Master Mode Programmable Parameters

Parameter	Function	Register
OVL	To select one or more Chip Select lines to generate $\overline{\text{CSovl}}$	EBU_EMUOVL
BUSCLK	To select the prescaler factor for EBUCLK, equal, half or one-fourth of LMB Clock	EBU_CON

Write Protection

Each address region has an associated bit to provide write protection (programmable on a region by region basis). Write protection is controlled by the “WR” bit in the **EBU_BUSCONx (x=0-6)** and **EBU_EMUBC** registers.

Little/Big Endian Access Modes

The native mode of the EBU is little endian. In addition, the EBU provides limited support for big endian access modes. Big endian accesses are supported by modifying the least significant two address bits. This feature is programmable (on a region by region basis) through the “ENDIAN” bit in the **EBU_BUSCONx (x=0-6)** and **EBU_EMUBC** registers.

Big endian access mode is available only for single accesses (8, 16 or 32 bit) to external regions programmed for 16-bit wide operation. There is no support for burst accesses to regions that have been programmed for big endian operation. There is no concept of little and big endian for accesses to 32-bit wide external regions.

Table 14-9 shows how the lower two bits of the LMB/FPI address is modified prior to being issued as an address on the external bus:

Table 14-9 Little and Big Endian Address Translation

Internal (LMB/FPI) Address (binary)	External Address (binary)			
	Little Endian		Big Endian	
	16-bit Access	8-bit Access	16-bit Access	8-bit Access
...xxx00_B	...xxx00_B	...xxx00_B	...xxx10_B	...xxx11_B
...xxx01_B	-[1)]	...xxx01_B	-[1)]	...xxx10_B
...xxx10_B	...xxx10_B	...xxx10_B	...xxx00_B	...xxx01_B
...xxx11_B	-[1)]	...xxx11_B	-[1)]	...xxx00_B

1) This represents a non-aligned 16-bit access which can never be generated on the LMB.

LMB Bus Width Translation

If the internal access width is greater than the external bus width specified for the selected external region (programmed via the **EBU_BUSCONx (x=0-6).PORTW** bit field), then the internal access is split into several external accesses to complete the required access. For example, if the LMB request is to read a 64-bit word and the external device is specified to be 16-bit wide, then the EBU will perform four external accesses (i.e. to 4 x 16-bit external addresses). When multiple accesses are generated in this way, external bus arbitration is suspended until the access is complete (i.e. the EBU remains the owner of the bus for the duration of the access sequence). The external accesses are performed in ascending LMB/FPI address order.

To allow proper bus width translation, the EBU has the capability to re-align data between the external bus and the LMB as shown in **Figure 14-6**.

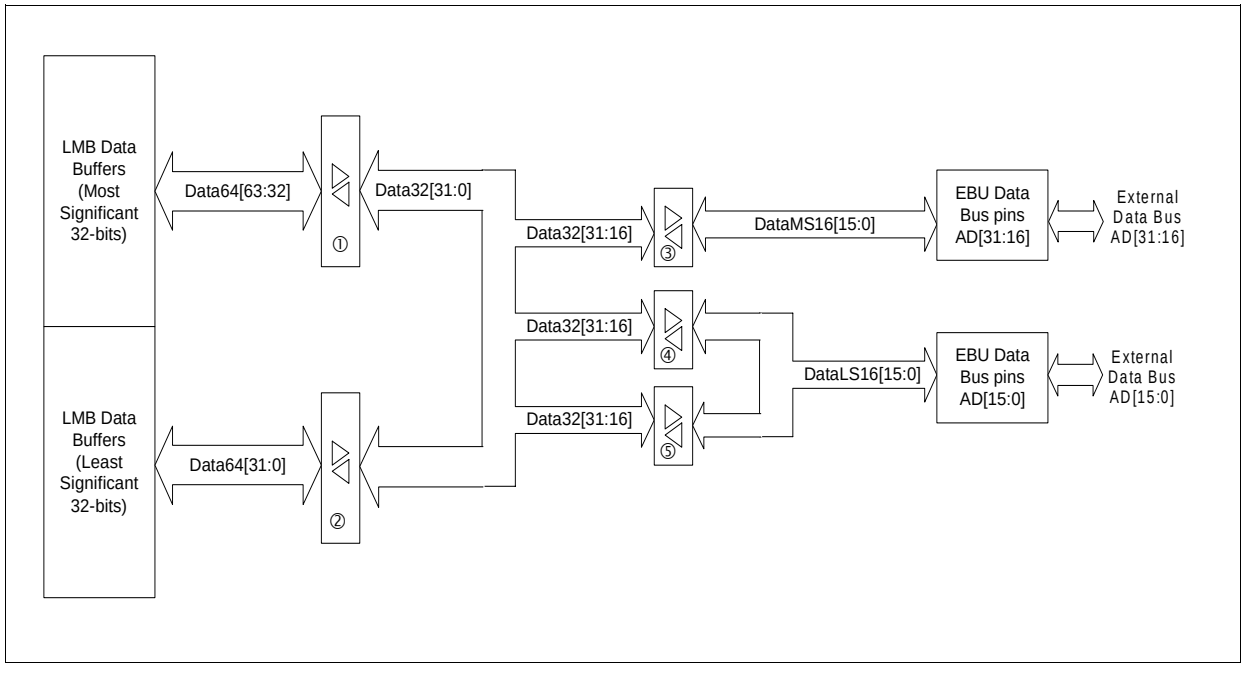


Figure 14-6 LMB to External Bus Data Re-alignment

- During an access to a 32-bit wide external region either “buffer 1” or “buffer 2” is enabled (according to bit 2 of the LMB address being accessed) to perform the required 64 bit (LMB) data to 32-bit bus alignment (signified by “Data32[31:0]” above). To generate a 32-bit access to the external data bus (AD[31:0]), “buffer 3” and “buffer 5” are enabled together.
- During an access to a 16-bit wide external region, either “buffer 1” or “buffer 2” is enabled (according to bit 2 of the LMB address being accessed) and either “buffer 4” or “buffer 5” is enabled (according to bit 1 of the LMB address being accessed). This allows any LMB channel byte pair (i.e. any properly aligned 16-bit data) to be re-aligned to the lower 16 bits of the external data bus (AD[15:0]).

*Note: When accessing a 16-bit wide external region programmed for big endian operation, the LMB address is modified prior to being driven to the external bus (see **Table 14-9**). For a 64-bit access, this will generate external bus accesses in the address sequence $x010_B$, $x000_B$, $x110_B$, $x100_B$. For a 32-bit access, the external bus address sequence will be $xx10_B$, $xx00_B$.*

FPI Bus Width Translation

If the internal access width is greater than the external bus width specified for the selected external region (programmed via the **EBU_BUSCONx (x=0-6).PORTW** bit field), then the internal access is split into several external accesses to complete the required access. For example, if the FPI request is to read a 32-bit word and the external device is specified to be 16-bit wide, then the EBU will perform two external accesses (i.e. to 2 x 16-bit external addresses). When multiple accesses are generated in this way, external bus arbitration is suspended until the access is complete (i.e. the EBU remains the owner of the bus for the duration of the access sequence). The external accesses are performed in ascending FPI address order.

To allow proper bus width translation, the EBU has the capability to re-align data between the external bus and the FPI as shown in **Figure 14-7**.

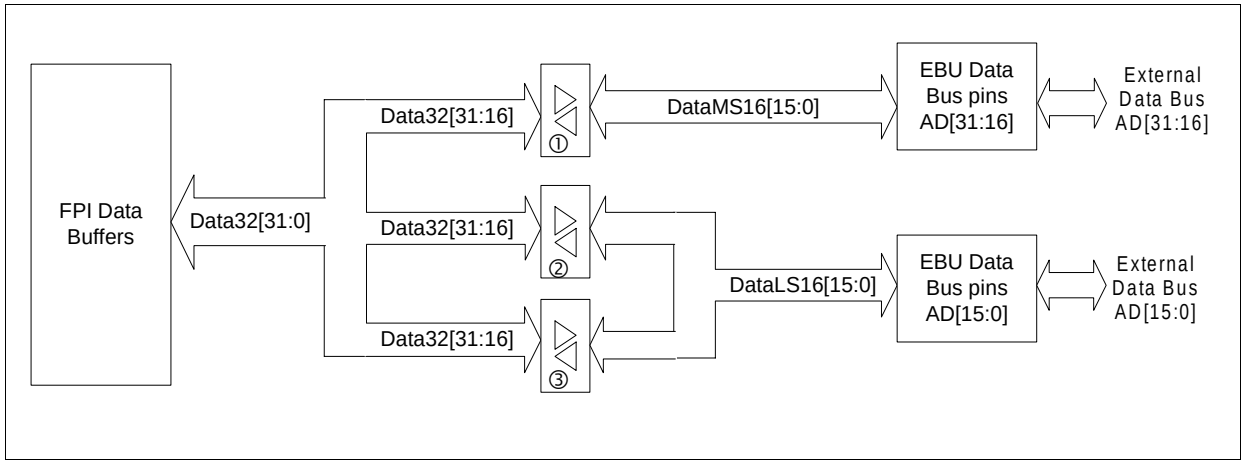


Figure 14-7 FPI to External Bus Data Re-alignment

- During an access to a 32-bit wide external region, “buffer 1” and “buffer 3” are enabled.
- During an access to a 16-bit wide external region, either “buffer 2” or “buffer 3” is enabled (according to bit 1 of the FPI address being accessed). This allows any FPI channel byte pair (i.e. any properly aligned 16-bit data) to be re-aligned to the lower 16 bits of the external data bus (AD[15:0]).

*Note: When accessing a 16-bit wide external region programmed for big endian operation, the FPI address is modified prior to being driven to the external bus (see **Table 14-9**). For a 32-bit access, the external bus address sequence will be $xx10_B$, $xx00_B$.*

External Bus Clock Generation

The EBU uses the LMB clock to generate all external bus access sequences. Two additional clocks are generated by the EBU for accesses to synchronous memory devices:

- EBUCLK is used to interface to SDRAM devices. The parameter **BUSCLK** in the **EBU_BUSCONx (x=0-6)** register is used to control the frequency of this signal.
- ACLK is used to interface to Burst Flash device. The parameter extclock in the **EBU_BFCON** register is used to control the frequency of this signal.

Address Alignment During Bus Accesses

During an external bus access, the EBU will optionally align the internal byte address to generate the appropriate external word or half-word address aligned to the external address pins A[23:0] and also on AD[31:0] in the case of an access to a multiplexed device. This alignment is enabled via the **EBU_BUSCONx (x=0-6).AALIGN** control bits to allow selection of address alignment mode on a region by region basis. When enabled, alignment will be performed according to the bus width selected (i.e. the value of the appropriate **EBU_BUSCONx (x=0-6).PORTW** field) for each Chip Select. This alignment will be as follows:

Table 14-10 EBU External Bus Address Alignment

PORTW	Bus Width	Address Alignment
0	reserved setting	-
1	16-bit	$A_{external}[23:0] = A_{LMB}[24:1]$
2[1)]	32-bit	$A_{external}[23:0] = A_{LMB}[25:2]$
3	reserved setting	-

1) When PORTW = 2 and the Chip Select is configured for WinCE Multiplexed Mode accesses, then a 32-bit address is issued. In this case (when alignment is enabled), the address alignment is $A_{external}[29:0] = A_{byte}[31:2]$, $A_{external}[31:30] = 0$.

When alignment is not enabled (default after reset) for a Chip Select, then the EBU will issue a byte address to the external bus.

Read/Modify/Write Accesses

During a Read/Modify/Write access from either the LMB or FPI bus, the EBU uses the $\overline{RMW}$ pin to signal that a Read/Modify/Write access is underway. During the read access cycle (of a Read/Modify/Write access), the RMW pin is driven low with the appropriate $\overline{CSx}$ (Chip Select) pin. The RMW pin remains high throughout the subsequent write access (and all subsequent accesses that are not the start of a Read/Modify/Write access).

FPI Read/Modify/Write transactions can only be generated for accesses up to and including Word accesses i.e. Byte (8-bit), Half-Word (16-bit) and Word (32-bit) accesses. Generation of a Read/Modify/Write access with data widths greater than 32-bits will cause invalid EBU operation.

Note: The Read/Modify/Write feature is intended for use with fully compatible external devices (e.g. devices containing an additional instance of the EBU). External bus slave devices that are not fully compatible with the Read/Modify/Write feature must not use the $\overline{RMW}$ signal.

14.5.2 Driver Turn-Around Wait States

Besides the wait states that can be inserted into an external access, the EBU supports the insertion of wait states in between consecutive accesses. This may be necessary if the current access is to a different address region than the previous one, or if a read access is followed by a write access, or vice versa. The insertion of wait states between the accesses allows the timing of accesses to external devices to be fine-tuned to gain higher performance.

When, for instance, a number of read accesses to an external memory are performed with a demultiplexed bus configuration, the memory is the only driver on the data bus, providing its data onto the bus. The memory is constantly selected via its Chip Select. If an access to a different device is performed (different address region, different Chip Select), the memory is deselected, and the next device is selected. However, many memory devices need a specific time to fully release the bus, to tristate their output drivers. Recovery wait states would need to be inserted at the end of the last access to the memory to ensure enough time to get off the bus before the next access occurs.

A similar situation is true if a read access is followed by a write access. The data bus driver role must change from the memory to the EBU. Again, the memory needs time to release the data bus, and recovery wait states need to be inserted.

If this recovery wait state insertion were to be programmed via the address region parameters (see **Section 14.5.1.2**), the wait states would apply to every access to the device, thus, slowing down the access performance. Instead, the EBU offers the option to insert such wait states either between accesses to different address regions (different Chip Selects) or between read and write accesses.

Because the internal clock runs much faster than the external devices, some devices may need to have long cycles in each phase. A general multiplier for all delays is not necessary and can be replaced with a better scheme where some configurations may have one or more of the phases relatively much longer than another. This will optimize the access and tune individual phase better to the device access characteristics.

Parameter **CMULT** (in BUSAPx and EMUBAP registers) provides multiplication factors of 1, 4, 8, 16 and 32. Delay parameters **WAITRDC**, **WAITWRC**, **DTARDWR** and **DTACS** are hardwired to always use the multiplier, as these delays tend to be larger than other

access parameters. Other individual seven access parameters (**ADDRC**, **AHOLDC**, **CMDDELAY**, **BURSTC**, **DATAC**, **RDRECOVC** and **WRRECOVC**) are programmable as to whether they are multiplied by **CMULT**. Each bit in **MULTMAP** is used to indicate whether each of the seven parameters mentioned above is multiplied by **CMULT**.

Programming of these wait states is done through register EBU_BUSCONx and EBU_BUSAPx. Between 0 and 480 idle cycles can be inserted between accesses to different address regions via bit field DTACS, while DTARDWR provides the option to insert between 0 and 15 idle cycles multiplied by bit field CMULT between a read and a write access, or vice versa. With these options, access performance to external devices is significantly improved, especially when a number of consecutive access of the same type (read or write) is performed.

14.5.3 Data Buffering

There are three data buffers used to adapt accesses taken place between the 16-bit or 32-bit external bus (accesses to external devices, SDRAMs, Flashes, or other memories) and the internal 64-bit Local Memory Bus (LMB, running at the multiplication of the power of two of the external bus speed, 1x, 2x, 4x) transactions and vice versa. A single 64-bit word Data Read Buffer, four 64-bit words Code Prefetch Buffer and four 64-bit word Data Write Buffer. The ratio of LMB to external bus frequency is programmable via the **EBUCON.BUSCLK**. The LMB transactions can not be split, i.e. the master will own the bus until the transactions are completed.

- **Code Prefetch Buffer,**
  For LMB code fetch transactions, every two 32-bit word data in the buffer will make up for a LMB response. This buffer is enough to service an LMB four word burst read request (BTR4) and support the code cache line size of TriCore. When the code is highly sequential in nature, the prefetch buffer can help speeding up the instruction fetching process. This feature is programmable through bit **pre** in each of BUSCONx and EMUBC registers. On an instruction fetch cycle, a transaction originated from the PMU takes place (most likely a cache line refill) on the LMB. At the end of this request, a prefetch activity is triggered to fill up the Code Prefetch Buffer by reading the next four consecutive 64-bit words. When using synchronous memories (i.e. SDRAM or Burst Flash), the prefetch is performed by extending the access by the appropriate number of cycles.
  For example, SDRAM burst (maximum up to the page boundary, code prefetching across SDRAM page boundaries is not supported). This prefetch will follow on immediately from the triggering code access, extending the access by typically 8 clock cycles for SDRAM. For asynchronous memories, the prefetch is performed by generating the appropriate number of device read accesses.The result of prefetch will be that, when the next request of sequential instructions arrive, the buffer is ready with the instructions. The only exception will be if a data access starts during the code access. In this case the pending prefetch will be cancelled. This feature will be user selectable and controlled by bit **wpre** in the BUSCONx and EMUBC registers.

Note: A code prefetch is only triggered by a PMU read access that is handled via the Data Read Buffer. A PMU read access that is serviced by the Code Prefetch Buffer will not generate a subsequent code prefetch.

- **Data Read Buffer**
  This buffer is used when a LMB master performs a read from a device on the external bus. The buffer helps composing the appropriate data width (e.g. 32 or 64-bit data) to be delivered on LMB from the appropriate number of 16-bit or 32-bit data fetched from external bus. When a LMB data read transaction occurs, this can be translated to a external read access or a bypass from Data Write Buffer. If the requested data is not in the Data Write Buffer, an external read access is generated. User must disable/ enable the bypass feature through **DLOAD** bit in BUSCONx or EMUBC registers, but this feature is always disabled for segment 14 (A[31:28] is equal to ’1110’). Software must ensure data coherency through DLOAD bit. When the EBU receives an external bus read access which represents a TriCore code fetch (i.e. generated by PMU) to a memory region with prefetch enabled, this will cause the EBU to perform a Code Prefetch into the Code Prefetch Buffer.
- **Data Write Buffer**
  This buffer is used when a cache line is being updated to the memory. When a LMB data write transaction occurs, this will be translated to external write accesses. The buffer holds the 64-bit data to be written in two 32-bit external write accesses. The size of this buffer matches cache line size of the data cache. A bypass is used from this buffer to the Data Read Buffer, when the buffer holds the requested read data as explained above.

14.5.4 Data Width of External Devices

The EBU supports external devices with a data width of 8, 16 or 32 bits. If the data width of an access is less than the width of the external device, the internal access is split into several external accesses to fetch the complete data requested.

14.5.5 Basic Access Timing

This section describes the basic access sequences of the EBU to external devices. Refer to the TC11IB Data Sheet for detailed timing diagrams and timing values.

Note: All timings described in this section are specified relative to the EBUCLK signal.

14.5.5.1 Standard Access Phases

Accesses to asynchronous and Burst Flash devices are composed of a number of Standard Access Phases. There are seven Standard Access Phases:

- **Address Phase (AP)**
  The phase in which the valid address is being put on the address bus. This phase is compulsory and can be repeated for slower devices. The falling edge of ALE signal

can be used by devices to sample the valid address. The number of additional cycles (the number of LMB Clock cycles) is programmed as **ADDRC** parameter in BUSAPx and EMUBAP registers.
When an access is performed to a Burst Flash device, the start of the Address Phase is always synchronized to a rising edge of the appropriate ACLK signal.

- **Address Hold (AH)**
  This phase is only for multiplexed devices where additional hold cycles are required to continue to drive the address lines, but having ALE deasserted. The number of cycles (the number of LMB Clock cycles) in this phase is programmable through **AHOLDC** parameter in BUSAPx and EMUBAP registers and can optionally be multiplied by the **CMULT** parameter, which is enabled via **MULTMAP** parameter (in BUSCONx and EMUBC registers).
- **Command Delay (CD)**
  This optional phase is the delay between address and command phase. Some devices are not fast enough to receive commands immediately after taking in the address, delay are needed in between. The number of delay cycles can be programmed as **CMDDELAY** (in BUSAPx and EMUBAP registers) and optionally be multiplied by the **CMULT** parameter. This Command Delay phase can also be prolonged externally by asserting the $\overline{\text{CMDELAY}}$ signal when the region being accessed is programmed for external command delay control via the **XCMDDELAY** parameters (in BUSAPx and EMUBAP registers). To distinguish between the two, in this document, **CDi** means 'internally-programmed' Command Delay and **CDe** means 'externally-prolonged' Command Delay by the assertion of $\overline{\text{CMDELAY}}$ signal.
- **Command Phase (CP)**
  The phase in which the action read/write is explicitly being put out. This phase is compulsory and can be repeated for inserting waitstates. There are separate parameters to reflect the number of wait state cycles in the read and write access. The **WAITRDC** parameter indicates how many additional cycles are required in a read access, while **WAITWRC** for write accesses. External device can also prolong the wait states by asserting the $\overline{\text{WAIT}}$ signal. For read accesses, data are latched at the end of this phase. In write accesses, this is the phase where the data is being put on the AD[31:0] bus and latched by external devices at the rising edge of $\overline{\text{WR}}$. **WAITRDC** and **WAITWRC** are in the BUSAPx and EMUBAP registers. Just like CD, CP is also distinguished into **CPi** and **CPe**. CPi also means 'internally-programmed' CP and CPe means 'externally-prolonged' Command Phase by the assertion of $\overline{\text{WAIT}}$ signal.
- **Data Hold (DH)**
  Hold cycles are provided for write data when the $\overline{\text{WR}}$ signal has gone inactive. This phase is optional and can be repeated, for example to have a prolonged hold bus for a write access to a slow device. The parameter **DATAC** (in BUSAPx and EMUBAP registers) is the number of cycles in the data hold phase.
- **Burst Phase (BP)**
  This phase is compulsory during accesses to Burst Flash devices. In order to read the required amount of data from the Burst Flash device, this phase is repeated as many

times as required. The length of each Burst Phase can be programmed via the **BURSTC** parameter (in BUSAPx and EMUBAP registers) and can optionally be multiplied by the **CMULT** parameter.

- **Recovery Phase (RP)**
  In this optional phase, idle cycles are inserted between transfers on the data bus. When accessing the same memory region, the number of inserted recovery cycles are programmable. In the case of read and write accesses the number of cycles in the recovery phase can also be different (programmable through **RDRECOVC** in BUSAPx and EMUBAP registers and **WRRECOVC** in BUSAPx and EMUBAP registers), to adapt the various behavior of devices. When going across a different memory region, the recovery cycles can also be inserted (**DTACS** in BUSAPx and EMUBAP registers), as well as switching between read and write (**DTARDWR** in BUSAPx and EMUBAP registers).

Table 14-11 Parameters for Recovery Phase

Case			Parameter
Switching to different $\overline{\text{CSn}}$	**Current access**	**Next access**	
same $\overline{\text{CSn}}$	read	read	RDRECOVC
same $\overline{\text{CSn}}$	write	write	WRRECOVC
same $\overline{\text{CSn}}$	read	write	max(RDRECOVC, DTARDWR)
same $\overline{\text{CSn}}$	write	read	max(WRRECOVC, DTARDWR)
different $\overline{\text{CSn}}$	read	read	max(DTACS, RDRECOVC)
different $\overline{\text{CSn}}$	write	write	max(DTACS, WRRECOVC)
different $\overline{\text{CSn}}$	read	write	max(DTACS,RDRECOVC, DTARDWR)
different $\overline{\text{CSn}}$	write	read	max(DTACS,WRRECOVC, DTARDWR)

The parameters in **Table 14-11** imply the maximum number of recovery cycles under each particular circumstances. For example, a read access to a region associated with $\overline{\text{CS1}}$ is followed by a write to a region associated with $\overline{\text{CS2}}$. In this case, if **DTARDWR** is greater than **DTACS** and **RDRECOVC**, then the number of recovery cycles between the two accesses is **DTARDWR**.

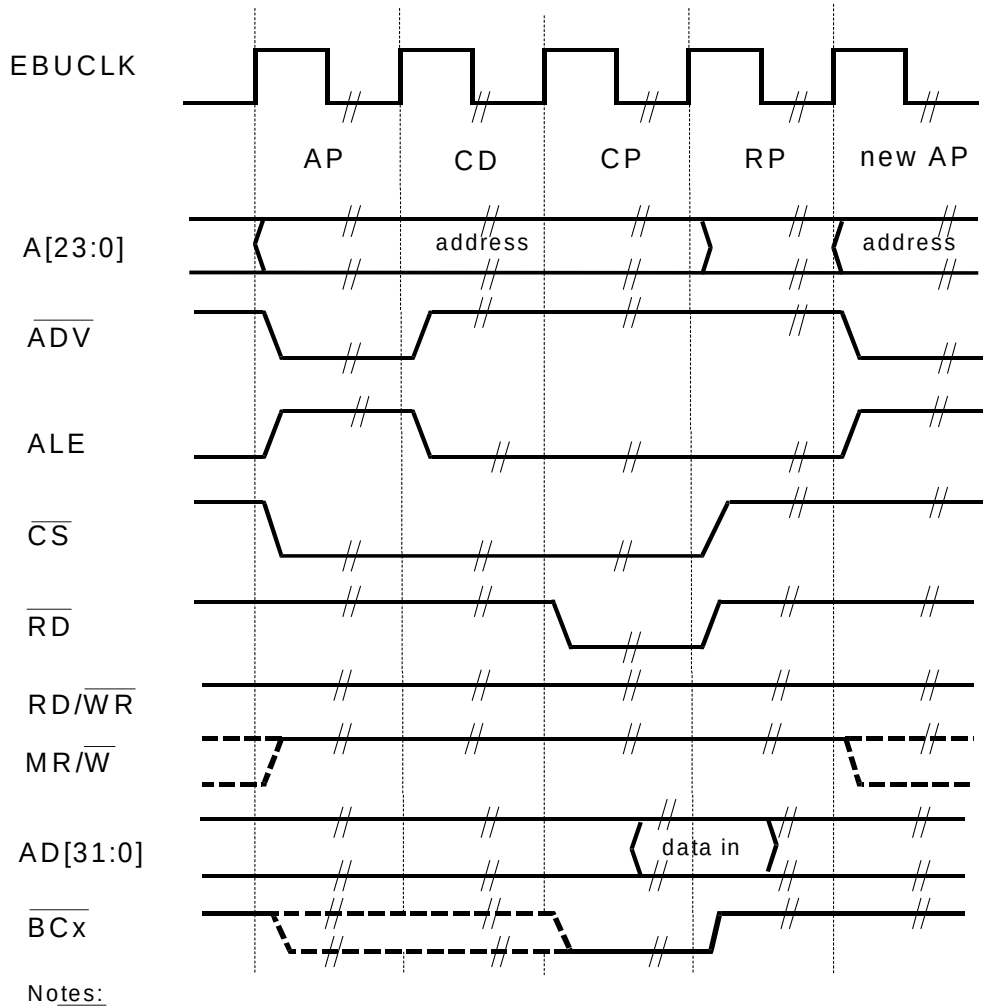
Note: Throughout the document the number following the phase name means the number of clock cycle has been executed so far for the respective phase. For example, AP2 points to the second clock in the Address Phase. CPe3 means the third clock in the Command Phase and is being extended by external wait states.

14.5.5.2 Access to Demultiplexed Devices

The EBU supports different configurations of Demultiplexed memory/peripheral devices. Selection of the appropriate Demultiplexed Memory configuration is performed by programming the **PORTW** parameter in registers BUSCONx.

*Note: These settings only apply when the "**AGEN**" field specifies that the device connected to the appropriate Chip Select is a Demultiplexed device.*

Devices with demultiplexed access (demultiplexed access) can be controlled by separate $\overline{RD}$ and RD/$\overline{WR}$ signals. **Figure 14-8** shows the basic sequence of a read access in demultiplexed mode.



Notes:
1. BCx (byte Control) signals can be programmed to be asserted either with read/write active (RD or RD/WR) or with Chip Select active (CSx).
2. CD and RP are optional phases.

Figure 14-8 Basic Read Access Timing in Demultiplexed Devices

This type of access cycle consists of from two to four phases as follows:

- Address Phase (compulsory).
- Command Delay Phase (optional).
- Command Phase (compulsory).
- Recovery Phase (optional).

Figure 14-9 shows an example of a write access to a demultiplexed device. This type of access cycle consists of from two to five phases as follows:

- Address Phase (compulsory).
- Command Delay Phase (optional).
- Command Phase (compulsory).
- Data Hold Phase (optional).
- Recovery Phase (optional).

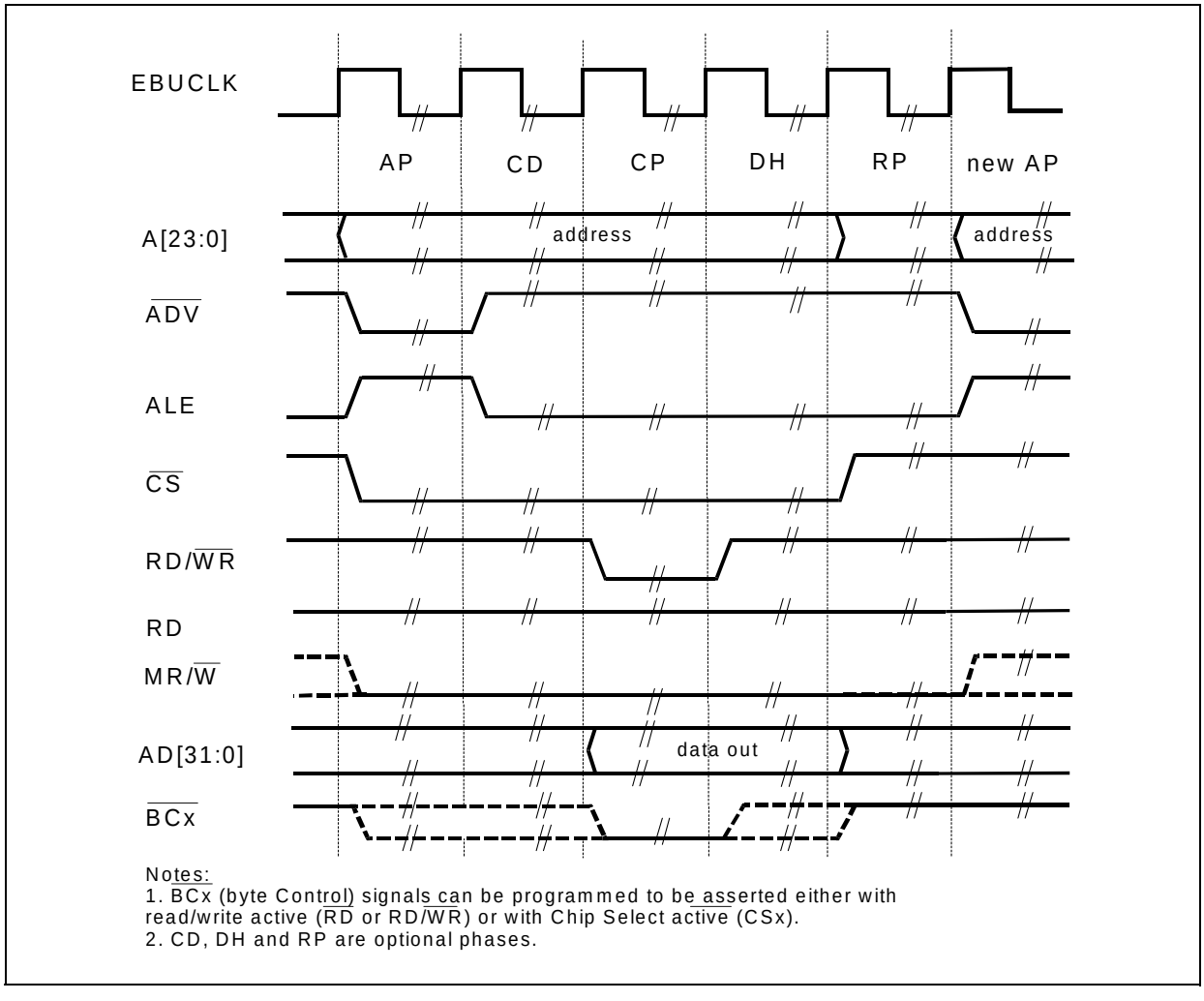


Figure 14-9 Basic Write Access Timing in Demultiplexed Devices

14.5.5.3 Access to Multiplexed Devices

The EBU supports different configurations of Multiplexed memory/peripheral devices. Selection of the appropriate Multiplexed Memory configuration will be performed by programming the **PORTW** parameter in registers BUSCONx as.

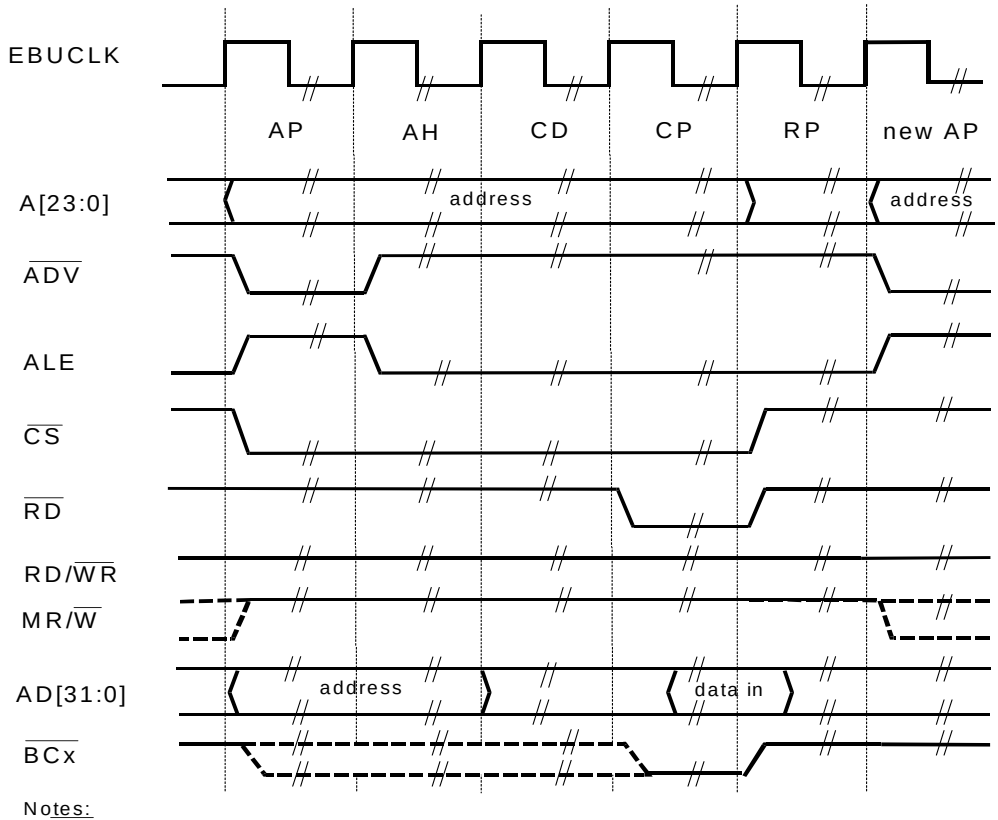
*Note: These settings only apply when the "**AGEN**" field specifies that the device connected to the appropriate Chip Select is a Multiplexed device.*

Devices using multiplexed address and data lines can be supported by the EBU according to the features and requirements. In multiplexed mode, the address/data bus AD[31:0] is shared between address output and data input/output. In the first part of access, the address is driven onto AD[31:0].

The address latch enable signal (ALE) is used to capture the address either into the external device (supporting multiplexed address/data) or into an external address latch. Then, the bus is switched to either input for a read access, or the write data is driven onto the bus on a write access. **Figure 14-10** shows the basic sequence of a read access in multiplexed mode. Under Multiplexed access mode, 128 MB memory space externally can be addressed to support Windows CE applications.

This type of access cycle consists of from two to five phases as follows:

- Address Phase (compulsory).
- Address Hold Phase (optional)
- Command Delay Phase (optional).
- Command Phase (compulsory).
- Recovery Phase (optional).



Notes:
1. BCx (byte Control) signals can be programmed to be asserted either with read/write active (RD or RD/WR) or with Chip Select active (CSx).
2. AH, CD and RP are optional phases.

Figure 14-10 Basic Read Access Timing in Multiplexed Devices

Figure 14-11 shows an example of a write access to a multiplexed device. This type of access cycle consists of from two to six phases as follows:

- Address Phase (compulsory).
- Address Hold Phase (optional).
- Command Delay Phase (optional).
- Command Phase (compulsory).
- Data Hold Phase (optional).
- Recovery Phase (optional).

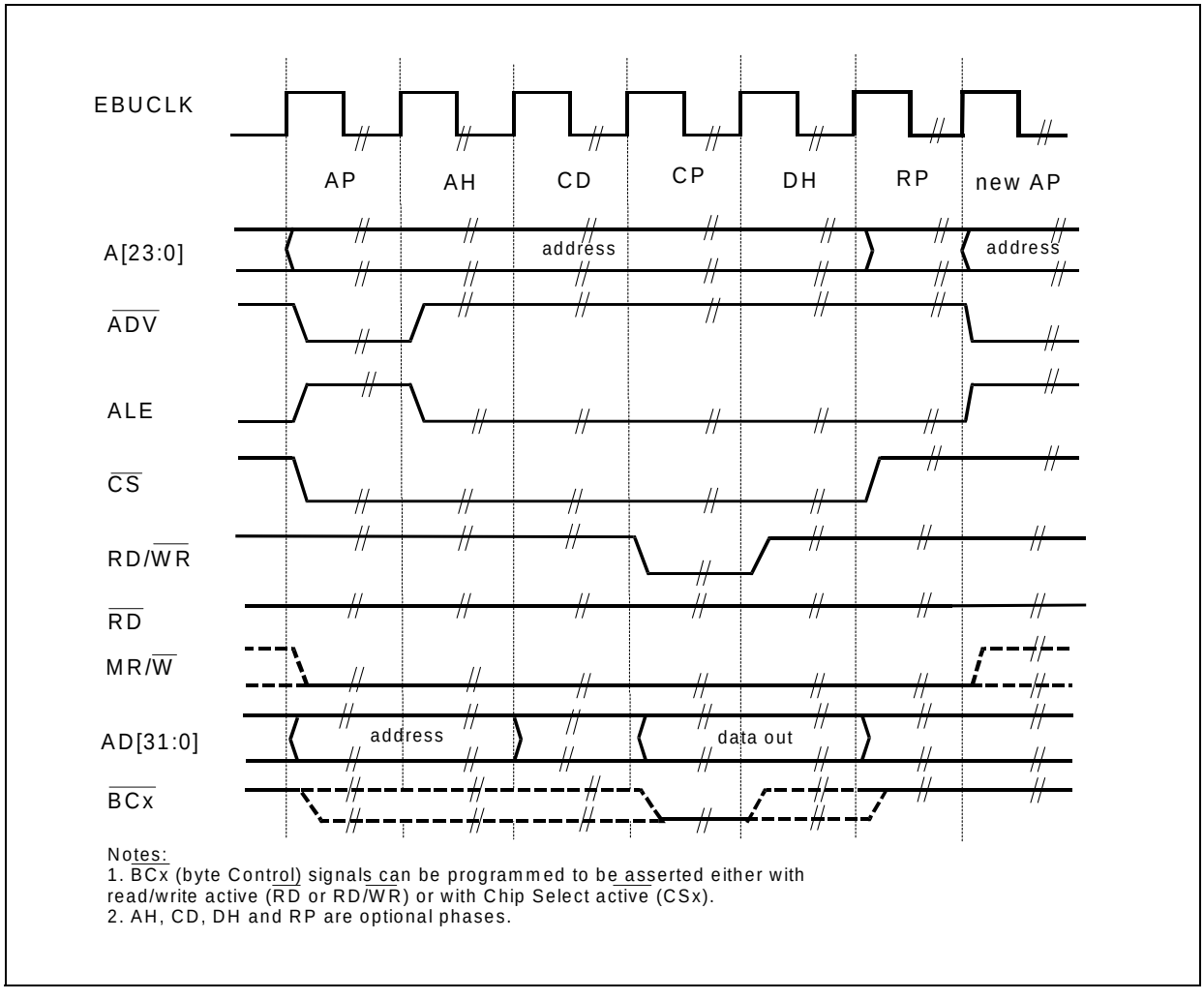


Figure 14-11 Basic Write Access Timing in Multiplexed Devices

14.5.6 Interfacing to Asynchronous Devices

Figure 14-12 illustrates a typical connection for Intel-style and Motorola-style peripherals. This also illustrates the use of a buffer to maintain pin loading requirements when using PC100 SDRAM devices. In this configuration all external devices (except the SDRAM) are connected through a buffer. The MR/$\overline{W}$ signal indicates the data direction for the current transfer and can be used to control the data direction through the buffer for the AD[31:0] bus (as well as controlling whether an access to a Motorola-Style device is read or write). The $\overline{CSGLB}$ signal is used to enable the outputs of the buffer during any access to a device other than SDRAM.

Note: When operating the EBU in this configuration, one must ensure that the $\overline{CSGLB}$ output is asserted low for all accesses to devices which are connected via the buffer by programming the appropriate value into the EBUCON.GLBCS bit field.

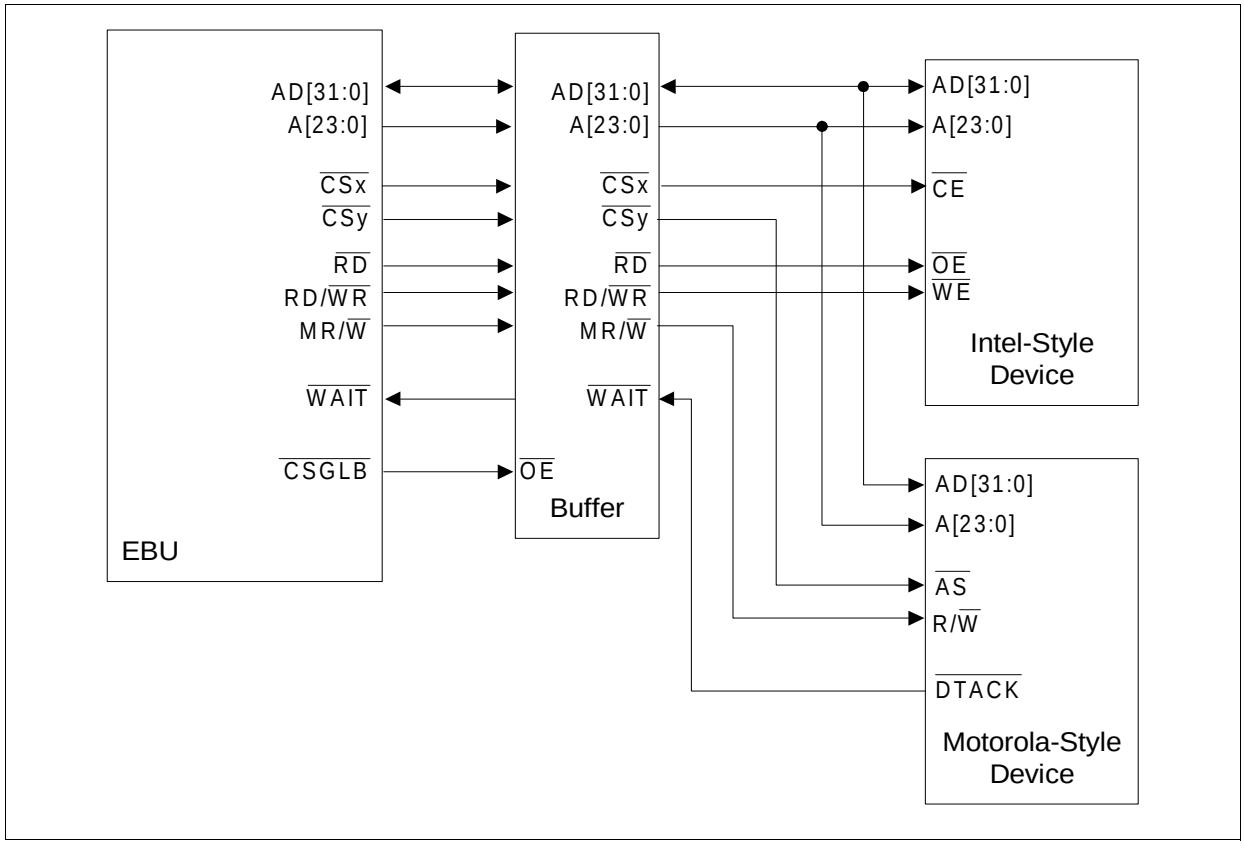


Figure 14-12 Typical Connection of Asynchronous Devices

Referring to **Figure 14-11** (multiplexed write access cycle), it can be seen that it is impossible to support multiplexed devices when a buffer is used in the configuration shown in **Figure 14-12** above. From these two figures it can be seen that during the Address Phase the data direction of the buffer (controlled by the MR/$\overline{W}$ signal) would cause the buffer to attempt to drive AD[31:0] at the same time that the EBU is attempting to drive the multiplexed address to AD[31:0]. This would lead to bus contention and also the issuing of an invalid address to a multiplexed device located on the right hand side of the buffer (as shown in **Figure 14-12** above).

In general, there are two critical phases during asynchronous device accesses. These phases are:

- Command Delay Phase (see **Section 14.5.5.1**).
- Command Phase (see **Section 14.5.5.1**).

For maximum flexibility, in addition to internal length programming for these two phases, the EBU also provides optional length control by use of external control lines:

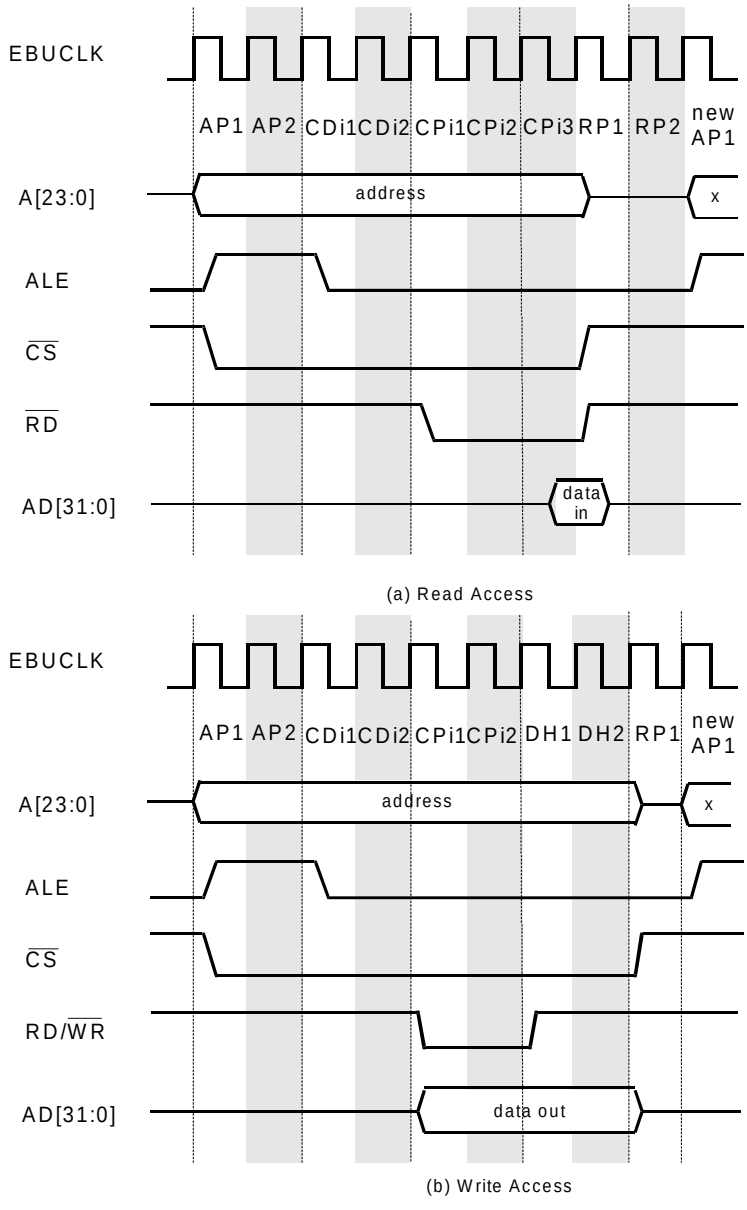
14.5.6.1 Interfacing to Intel-style Devices

Figure 14-13 shows an example of accessing an Intel-Style demultiplexed device for both read and write accesses. This shows the insertion of delay cycles (shown shaded) to adjust the access cycle to the device's timing requirements.

Both read and write accesses begin with a two cycle Address Phase followed by a two cycle Command Delay Phase.

For the read access, the Command Delay Phase is followed by a three cycle Command Phase. At the end of the Command Phase, the data is read (latched) by the EBU. A one cycle Recovery Phase is inserted at the end of the cycle. At the start of this Recovery Phase all control signals return to their non-active levels.

For the write access, the Command Delay Phase is followed by a two-cycle Command Phase. During a write access it is possible to insert a Data Hold Phase to satisfy the data hold time requirements of the device. In the example the Data Hold Phase consists of two cycles. During the Data Hold Phase, the RD/$\overline{WR}$ control signal is driven to the non-active state but the data and address are still driven on the bus.



Notes: ALE is not used for interfacing to Intel-style devices and is shown for reference only.

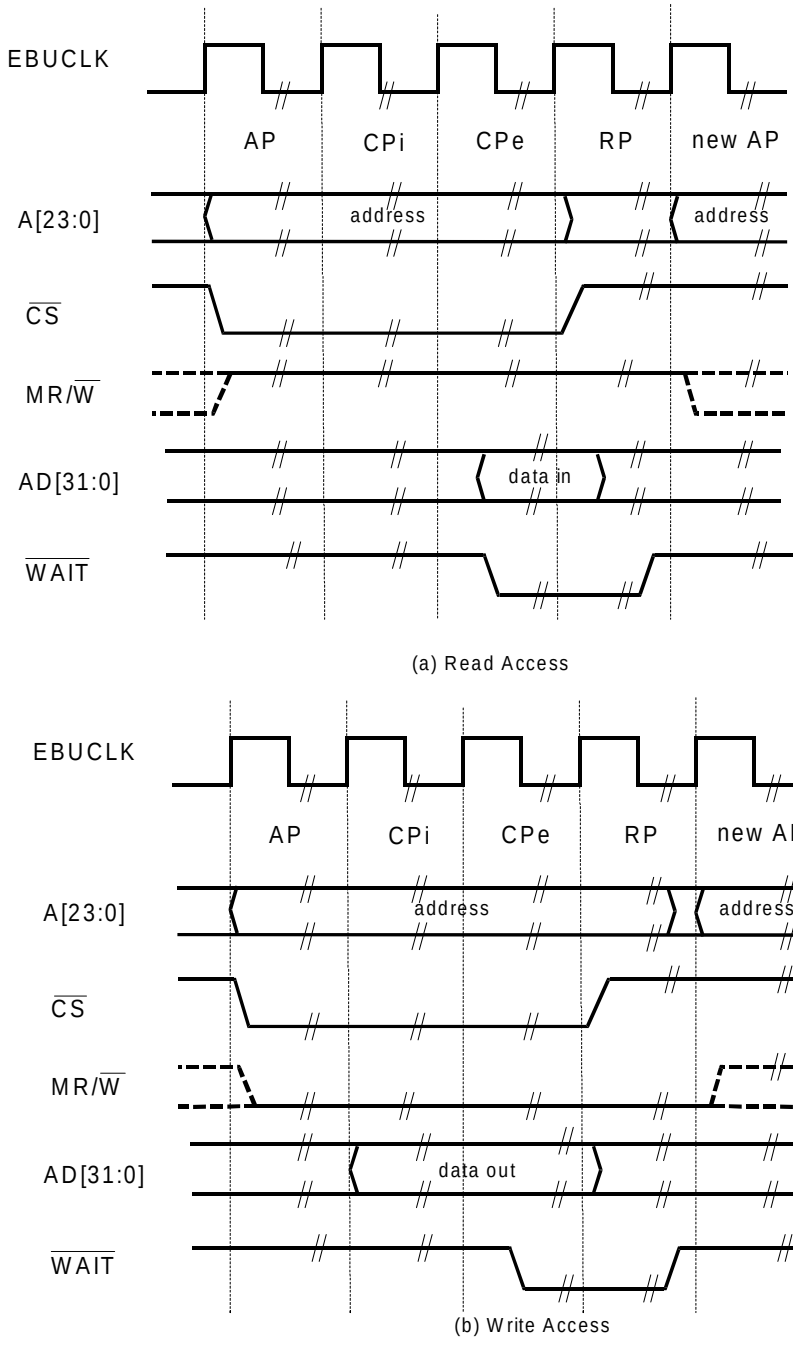
Figure 14-13 Example of an Intel-style Demultiplexed Device Access

14.5.6.2 Interfacing to Motorola-Style Devices

Figure 14-14 gives an example of Motorola-style access. The Chip Select signal ($\overline{CS}$) is used to generate the $\overline{AS}$ input for the device. The MR/$\overline{W}$ signal maintains its level (according to whether the cycle is read or write) from the Address Phase to the Recovery Phase during each access.

Externally controlled wait states are used to synchronize with the DTACK output from the device. A device asserts DTACK during a read access to signal that the data is available on the data bus. During a write access the falling edge of DTACK signals that the device has completed the write access and the data can be removed from the data bus. This can be accommodated by use of the WAIT signal with inverse polarity (selected via the **EBU_BUSCONx (x=0-6).WAITINV** or **EBU_EMUBC.WAITINV** bits) with the following limitations:

- During read accesses a minimum command phase (three LMB Clock cycles) must be used to ensure correct recognition of the WAIT input.
- Only the falling edge of DTACK is recognized by the WAIT pin and the user must insert sufficient recovery cycles to ensure that the DTACK pin has returned high before the next external bus cycle starts (i.e. before the next falling edge of any CSx signal). Typically, devices de-assert DTACK after the rising edge of their AS input (which is connected to an EBU CSx output).



Notes: CS is connected as AS to Motorola devices, while WAIT (with inverse polarity) is connected to the DTACK output.

Figure 14-14 Example of an Motorola-style Demultiplexed Device Access

14.6 Detailed External to Internal EBU Operation (Slave Mode)

The following subsections provide more detailed insight into the operation of the EBU for external to internal transactions. Such transactions can be performed by an external bus master that wants to perform a read or write access to an on-chip FPI Bus device. The master needs to have ownership of the external bus, thus, bus arbitration will be required before such an access (see **Section 14.7.1**). This master can access the EBU by activating the EBU Chip Select input $\overline{\text{CSFPI}}$ and presenting a proper address on the address bus. The features and functions of this operation and its basic timing are described in the following.

Note: In the TC11IB, if an external bus master is used to access internal peripherals, then PCP must never access the external Bus. When the external master is not used, PCP may access the external bus.

14.6.1 EBU Signal Direction

When the EBU is accessed by an external master that wants to read or write an internal module, the direction of some of the EBU signals need to be reversed. The address bus and control signals are now inputs to the EBU, driven by the external master. The data bus is input for writes, and output for reads. The $\overline{\text{WAIT}}$ signal is now driven by the EBU to indicate to the master the necessity for additional wait states. **Table 14-12** lists the EBU signals for external to internal operation and indicates their direction and relevance for the access.

Table 14-12 EBU Signals for External to Internal Operation

Signal	Direction	Pull	Driven by
AD[31:0]	Input for write access, output for read access	Up	External Master for writes, the EBU for reads
A[23:2]	Input	Up	External Master
$\overline{\text{RD}}$	Input	Up	External Master
RD/$\overline{\text{WR}}$	Input	Up	External Master
$\overline{\text{CSFPI}}$	Input	Up	External Master
$\overline{\text{BC0}}$	Input	Up	External Master
$\overline{\text{BC1}}$	Input	Up	External Master
$\overline{\text{BC2}}$	Input	Up	External Master
$\overline{\text{BC3}}$	Input	Up	External Master
$\overline{\text{WAIT}}$	Output (Open Drain)	Up	EBU
$\overline{\text{RMW}}$	Input/Output	Up	EBU

14.6.2 Address Translation

While the EBU is operating in Slave mode, it offers the memory map to the external master, as shown in **Table 14-2**.

Because the external address bus is only 24 bits wide, any external address presented by an external bus master must be extended to the full 32-bit FPI Bus address width via an address translation mechanism. It gives access up to three 4 MByte regions at a time. The external 24-bit address is extended to a full 32-bit FPI address by concatenation of A[21:2] with the contents of one of three 10-bit registers. An address extension register will be selected depending on bits A[23:22]. The default values of the address extension registers give an external master the possibility to access the 4 Mbytes of segment 14: $E8000000_H$ - $E83FFFFF_H$ (CPU memories), segment 11: $BFC00000_H$ - $BFFFFFFF_H$ (ComDRAM) and segment 15: $F0000000_H$ - $F03FFFFF_H$ (internal peripheral space). These registers can be reprogrammed by the external master in EXTCON register to access also other areas (if enabled by software through bit **EXTRECON** in EBUCON register). To write correct address extension values into register EXTCON is necessary. This register is only accessible from the external bus (by external master). The extended address must never point to an external region because this could cause a deadlock situation on the LMB bus. The EBU of TC11IB would try to access an external region while being occupied by the access of the external master.

In order to keep security, it is possible to inhibit all accesses from an external master to FPI. The bit **EXTACC** of register EBUCON is internally combined with the $\overline{\text{CSFPI}}$ signal and can prevent the device from ever being selected. All external accesses to FPI will be performed in user or supervisor mode depending on the value of bit **EXTSVM** in register EBUCON.

The external master can set bit **RMWEN** in register EXTCON so that the EBU of TC11IB locks the FPI bus to speed up a sequence of accesses (e.g. for read-modify-write accesses). Like other parameters in EXTCON, this can only be changed by external master if bit **EXTRECON** in EBUCON register is set.

Table 14-13 Default Values for the Address Extension

A[23:22]	AEXT(n)[31:22]	Location
00_B	1110 1000 00	Segment 14: CPU memories
01_B	others	Used for reprogramming EXTCON register
10_B	1011 1111 11	Segment 11: ComDRAM
11_B	1111 0000 00	Segment 15: Internal Peripheral

Figure 14-15 gives an overview on the address extension operation. The byte-control mapping function generates A[1:0] and specifies the access type from the byte-control

signals $\overline{BC}[3:0]$, which must be generated by the external master. The EBU always behaves as a little-endian 32-bit device that supports aligned accesses only.

Table 14-14 Byte Control Mapping Function

$\overline{BC3}$	$\overline{BC2}$	$\overline{BC1}$	$\overline{BC0}$	FPI Bus Access Width	FPI Bus Address A[1:0]
			0	Byte	00_B
		0	0	Half-word	
0	0	0	0	Word	
		0		Byte	01_B
	0			Byte	10_B
0	0			Half-word	
0				Byte	11_B
Other combinations				Undefined	

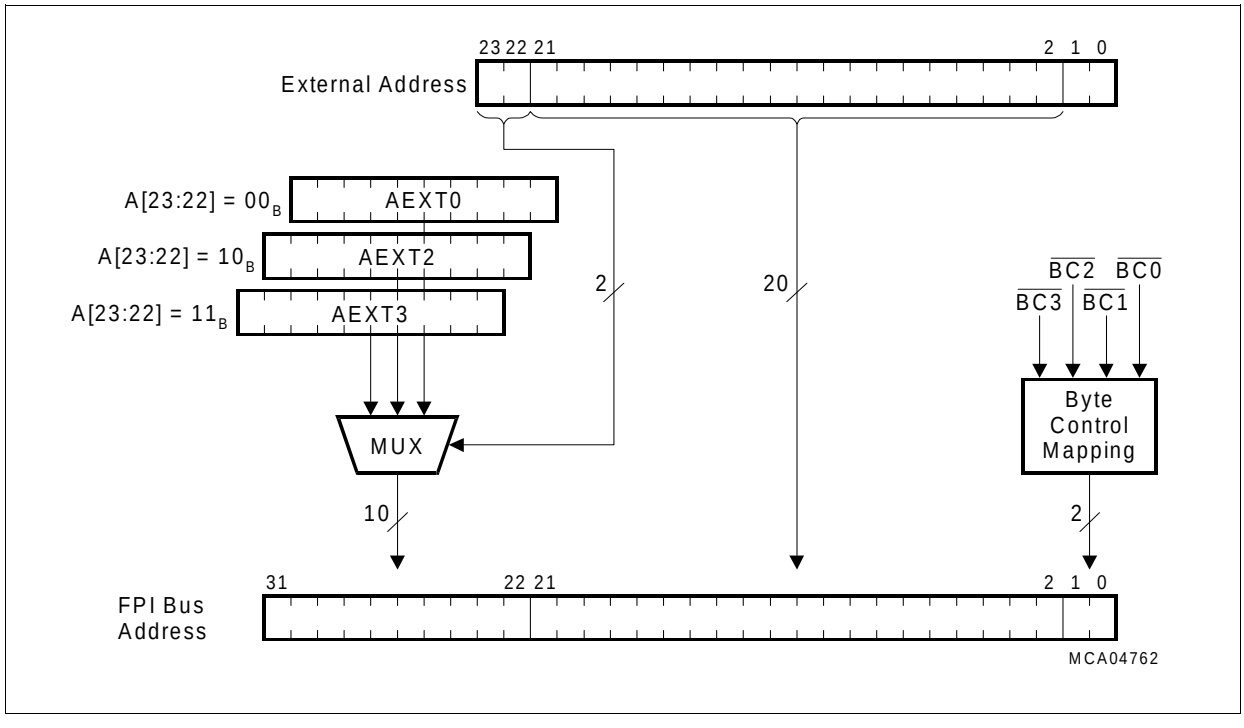


Figure 14-15 External to Internal Address Extension

14.6.3 External to Internal Access Control

Three bits in register EBU_CON and one bit in register EBU_EXTCON help to control external accesses to the EBU. Access from an external master to the internal resources via the EBU can be enabled or disabled via bit EXTACC in register EBU_CON. Accesses can be performed in User mode or in Supervisor mode as determined by the level at pin

SVM (defined by.bit EBU_CON.EXTSVM). The option to reprogram the address extension registers by the external master can also be enabled or disabled through bit EXTRECON. These controls are useful to prevent hostile or faulty accesses from the external world onto the FPI Bus.

If the external master needs to perform read-modify-write accesses to the FPI Bus, it can lock the FPI Bus such that no other transaction can take place between the read and the write operation. This is done through bit RMWEN in register EBU_EXTCON. See also **Section 14.12.8**.

14.6.4 Basic Access Timing

When accessed by an external master, the EBU behaves like a 32-bit wide, little-endian device with byte write capability. Accesses must be naturally aligned. Thus, an external master must drive the $\overline{BC}$n signals and align byte writes to the appropriate data bus byte lane. Usually, the external master derives its clock from a source other than the EBU. Thus, timing synchronization is specified in relation to the $\overline{RD}$ and RD/$\overline{WR}$ signals.

Access time is mainly determined by the time needed for synchronization (two internal clock cycles required, worst case) and the time consumed by the FPI Bus transfer (at least three internal clock cycles required). The earliest start point of an FPI Bus transfer for an external master access is given in **Table 14-15**.

Table 14-15 Earliest Start Point of FPI Bus Transfer for External Master Accesses

Type of Access	Start Point of FPI Transfer (earliest)
External master reads from FPI device	synchronized address bus, $\overline{RD}$, RD/$\overline{WR}$ and $\overline{BC}$
External master writes to FPI device	synchronized address bus, $\overline{RD}$, RD/$\overline{WR}$, $\overline{BC}$ and data bus

A write access will request additional waitstates only if the former write access has not finished yet (because write accesses are buffered).

A Read/Modify/Write operation (on the external bus) consists of a read cycle followed by a write cycle. The write access must be of the same data width and to the same address as the proceeding read. At the start of the read cycle, the external master drives the $\overline{RMW}$ signal low to show that the read access is part of a Read/Modify/Write access. When the EBU samples the $\overline{RMW}$ signal low at the start of a read cycle, this causes the EBU FPI Master interface to generate an FPI RMW access (rather than a discrete Read access). This will automatically lock the FPI bus until the completion of the associated Write access.

Note: Read/Modify/Write accesses must only be generated by a fully compatible external master.

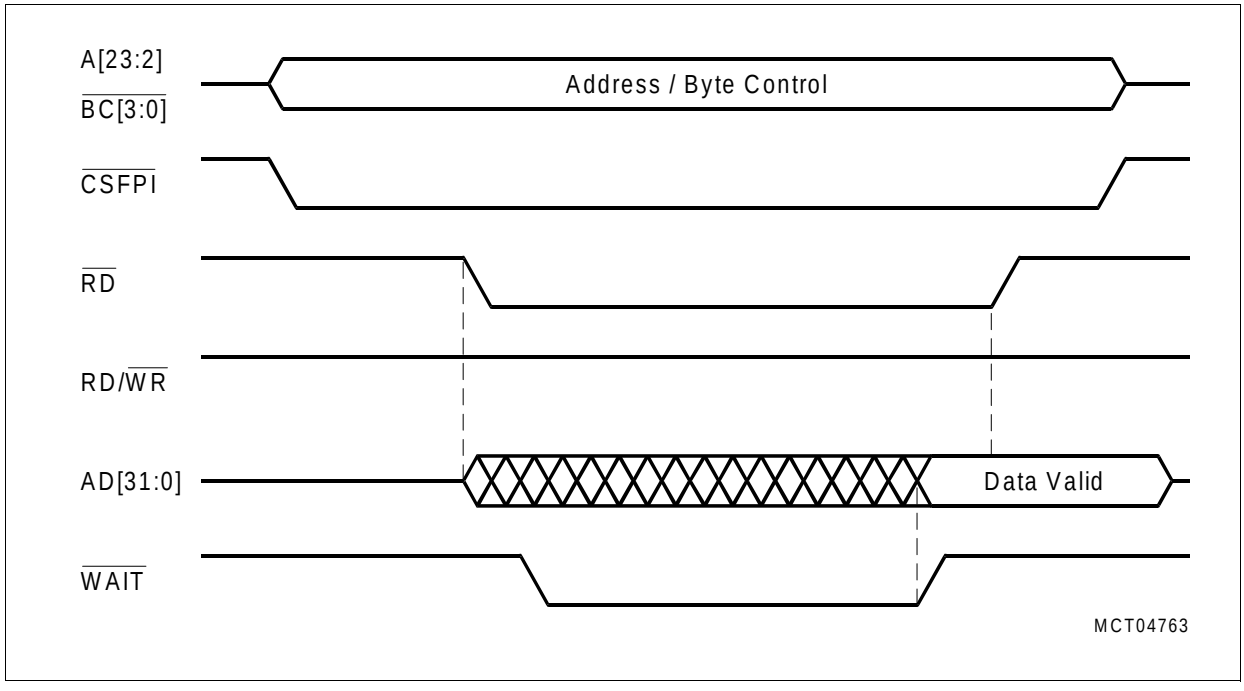


Figure 14-16 Basic External to Internal Read Access Timing

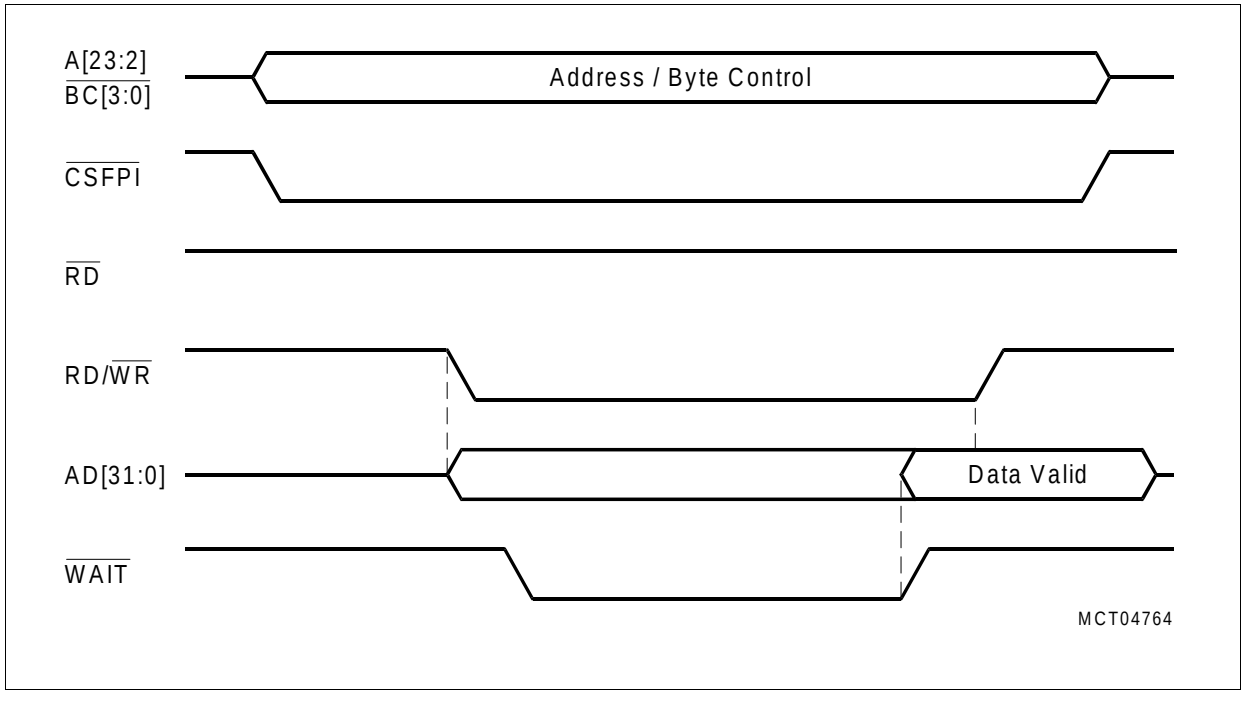


Figure 14-17 Basic External to Internal Write Access Timing

14.7 Arbitration

There are multiple functions for the External Bus Controller. It establishes and controls the external bus and acts as a bidirectional interface between the internal and the external system. Additionally, it provides bus arbitration support to allow up to two masters to use the external bus. Other than the EBU, an external master can grab the external bus and access external devices (leaving the EBU off the external bus) or accessing internal devices (through FPI bus). Every time a master requires the external bus, it will signal to the current master and wait for it to release the bus. This scheme is compatible to other Tricore and C166 devices, therefore, such devices can be used as external master.

14.7.1 Arbitration Modes

The arbitration logic of the EBU can be configured to one of four modes through configuration pins during reset or setting ARBMODE after reset:

- No Bus:
  All accesses from internal LMB/FPI Bus masters to the external bus via the EBU are disabled and will cause an LMB/FPI Bus error.
- Arbiter Mode:
  The EBU is normally the owner of the external bus. This mode is the default mode after reset if external boot is configured. Bus ownership must be requested from the EBU if another external bus master (configured to Participant Mode) needs to get onto the external bus.
- Participant Mode:
  Another external bus master is the default owner of the external bus. The EBU will request bus ownership from this master only in case of a pending transfer issued by an internal LMB/FPI Bus master.
- Sole Master Mode:
  This mode is used when the EBU is the only master on the external bus. No external bus arbitration is necessary in this configuration, and accesses to the external bus are always possible. LMB accesses to the external bus take priority over FPI accesses to the external bus. If an FPI master submits a request to access the external bus which cannot be serviced due to a pending or on-going LMB transaction then the EBU will issue an FPI-Retry response. If an LMB master submits a request to access the external bus which cannot be serviced due to a pending or on-going FPI transaction then the EBU will issue an LMB-Retry response.The arbitration signals are not evaluated in this mode.

14.7.1.1 Arbitration Signals

The EBU has three signals dedicated to external bus arbitration, $\overline{\text{HOLD}}$ (hold input), $\overline{\text{HLDA}}$ (hold acknowledge) and $\overline{\text{BREQ}}$ (bus request). As described in this section, these signals are used differently depending on the arbitration mode. **Figure 14-18** shows the

interconnection of arbitration signals for an arbiter and a participant. **Table 14-16** shows the function of the arbitration pins in arbiter mode, and **Table 14-17** shows the function of these pins in participant mode. The term “participant mode” in this context relates to the mode of bus arbitration; the EBU is still considered as one of the external bus masters in this mode.

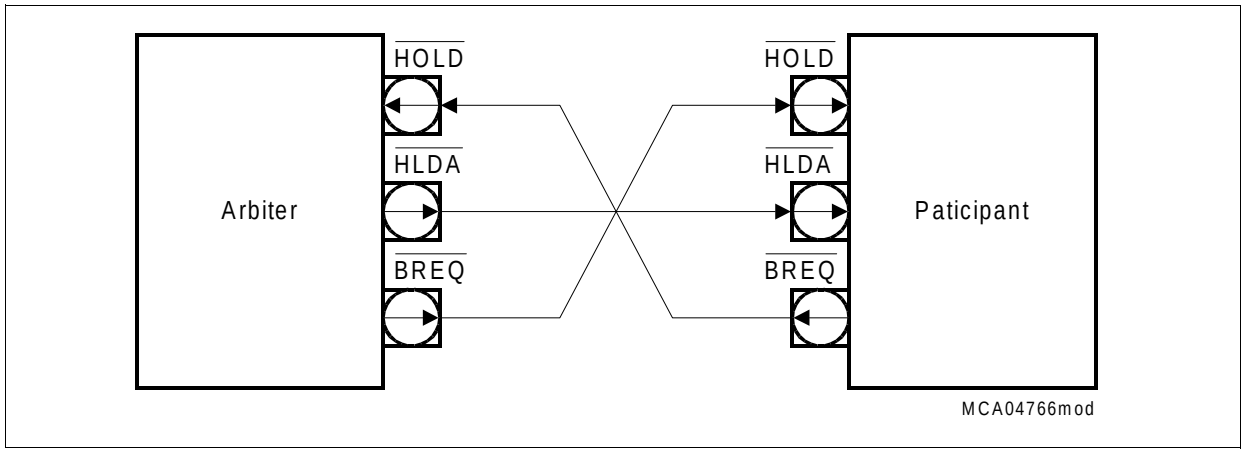


Figure 14-18 Connection of the Bus Arbitration Signals

Note: This simple connection allows two devices with the same (such as two TC11IBs) or very similar arbitration schemes to perform bus arbitration. If more than two external bus masters are used, external circuitry is required for proper arbitration.

Table 14-16 Arbitration Signals in Arbiter Mode

Pin	Type	Function in Arbitration External Master Mode
$\overline{\text{HOLD}}$	In	While $\overline{\text{HOLD}}$ is high, the EBU is operating in normal mode and is the owner of the external bus. A high-to-low transition indicates a hold request from an external device. The EBU finishes ongoing transactions, then backs off the bus, activates $\overline{\text{HLDA}}$ and goes into hold mode. A low-to-high transition causes it to exit from hold mode. The EBU deactivates $\overline{\text{HLDA}}$, takes over the bus, and resumes normal operation.
$\overline{\text{HLDA}}$	Out	This signal is high during normal operation. When the EBU enters hold mode, it sets $\overline{\text{HLDA}}$ low after releasing the bus. On exit of hold mode, the EBU first sets $\overline{\text{HLDA}}$ high and then goes onto the bus again. It does this to avoid collisions.
$\overline{\text{BREQ}}$	Out	This signal is high during normal operation. The EBU activates $\overline{\text{BREQ}}$ at the earliest one clock cycle after activating $\overline{\text{HLDA}}$ if it must perform an external bus access. If the EBU has regained the bus, $\overline{\text{BREQ}}$ is set to high one clock cycle after deactivation of $\overline{\text{HLDA}}$.

Table 14-17 Arbitration Signals in Participant Mode

Pin	Type	Function in Arbitration External Slave Mode
$\overline{\text{HOLD}}$	In	While both $\overline{\text{HOLD}}$ and $\overline{\text{HLDA}}$ are high, the EBU is in hold mode, and the external bus interface signals are tristated. When the EBU is released out of hold mode ($\overline{\text{HLDA}}$ = 0) and has completely taken over control of the external bus, a low level at this pin requests the EBU to go into hold mode again. But in any case the EBU will perform at least one external bus cycle before going into hold mode again.
$\overline{\text{HLDA}}$	In	A high-to-low transition at this pin releases the EBU from hold mode.
$\overline{\text{BREQ}}$	Out	This signal is high as long as the EBU operates from internal memory. When it detects that an external access is required, it sets $\overline{\text{BREQ}}$ to low and waits for signal $\overline{\text{HLDA}}$ to become low. $\overline{\text{BREQ}}$ will go back to high when the slave has backed off the bus after it was requested to go into hold mode.

When synchronous arbitration signal sampling is selected (parameter **ARBSYNC** in EBUCON), the arbitration input signals are sampled and evaluated in the same clock cycle. This mode provides the least overhead during arbitration (i.e. when changing bus ownership). The disadvantage is that the input signals must adhere to set-up and hold times with LMB Clock to prevent the propagation of meta-stable signals into the EBU.

When asynchronous arbitration signal sampling is selected (parameter **ARBSYNC** in EBUCON), the arbitration signals are sampled and then fed to an additional latch to be evaluated in the cycle following that in which they were sampled (i.e. the signals pass through a cascade of two latches before being evaluated). This provides the EBU with good immunity to signals changing state at or around the time at which they are sampled. The disadvantage is the introduction of additional latency during arbitration (i.e. when changing bus ownership).

14.7.1.2 Arbitration Sequence

Figure 14-19 shows the sequence of bus arbitration signals in a arbiter/participant system. In this context, the terms “arbiter” and “participant” refer to the arbitration modes. The participant is one of the bus masters on the external bus, capable of initiating bus transactions and driving the external bus, but it is not the default owner of the bus. It must request the bus from the arbiter (default owner of the external bus) to perform a transaction. At start-up, the arbiter is in normal mode and operating on the external bus; the participant is in hold mode, operating from internal memory; the participant’s bus interface is tristated.

- **Stage 1)** The participant detects that it must perform an external bus access. It activates $\overline{\text{BREQ}}$, which issues a hold request to the arbiter.

- **Stage 2)** The arbiter activates $\overline{\text{HLDA}}$ after having released the bus. This initiates the participant's exit from hold sequence.
- **Stage 3a)** When the arbiter detects that it also must perform external bus accesses, it activates his $\overline{\text{BREQ}}$. The earliest time for the master to activate $\overline{\text{BREQ}}$ is two LMB clock cycles after the activation of the arbiter's $\overline{\text{HLDA}}$ signal. However, the participant will ignore this signal until it has finished the current access. In this way it is assured that the participant will at least perform one complete external bus access.
- **Stage 3b)** If the arbiter can operate from internal memory while in hold mode, it leaves the $\overline{\text{BREQ}}$ signal high until it detects that an external bus access must be performed. Therefore, the participant can stay on the bus until the arbiter does not request the bus.
- **Stage 4)** When the arbiter has requested the bus again through activation of his $\overline{\text{BREQ}}$ signal, the participant will complete the current access and go into hold mode again. After having tristated its bus interface, the participant deactivates its $\overline{\text{BREQ}}$ signal, thus releasing the arbiter from hold mode.
- **Stage 5)** The arbiter has terminated its hold mode and deactivates its $\overline{\text{HLDA}}$ signal. Now, the arbiter again controls the external bus.
- **Stage 6)** The arbiter deactivates its $\overline{\text{BREQ}}$ signal one cycle after deactivation of $\overline{\text{HLDA}}$. From now on (and not earlier), the participant can generate a new hold request to the master. With this procedure, it is assured that the arbiter can perform at least one complete bus cycle before it goes into hold mode again if requested by the participant.

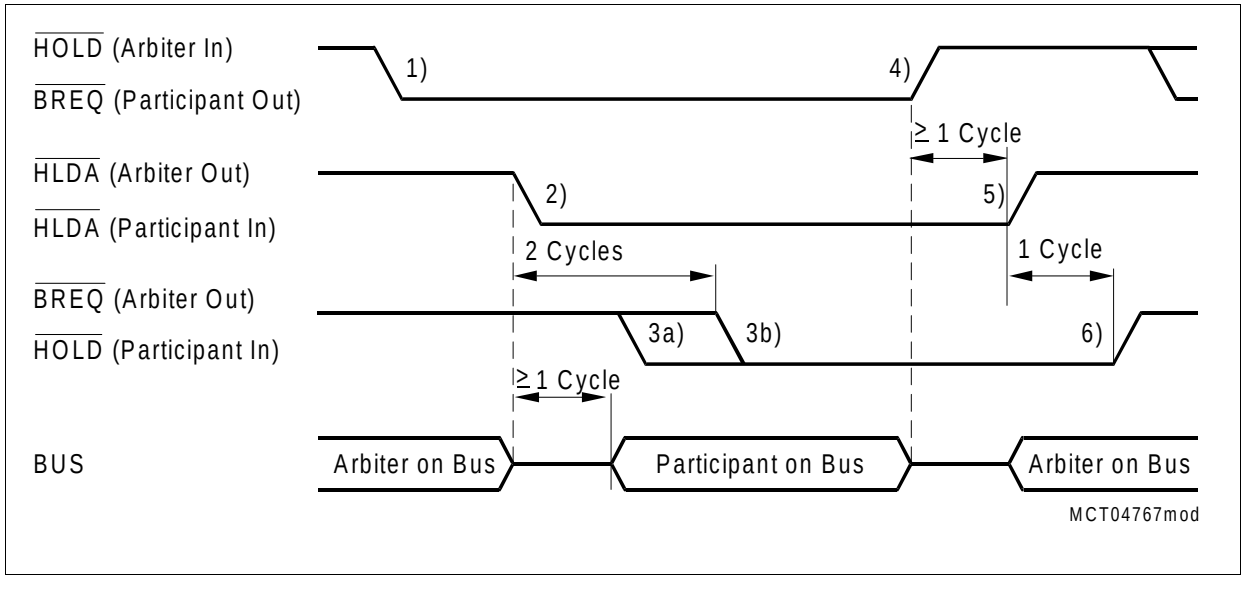


Figure 14-19 Bus Arbitration Sequence

The result of this arbitration scheme is such that if both devices would constantly request the bus (for example, if both are executing code out of external memory), they alternately gain bus access. However, it is typical for this configuration that the participant would

execute out of internal memory, only eventually requesting an external bus transaction to load/store data.

The behavior of the external bus controller depends in general on the status of the external bus. It can be summarized as follows:

Table 14-18 External Bus Controller Actions Depending on Bus Ownership

Owner of external bus	Access	Actions of external bus controller
External master	External master to FPI device	- Release external bus (tristate all drivers, some signals need pull ups) - Perform requested FPI access - Return result to external master (read access only)
	LMB master to external device	- Reject LMB access with RETRY - Request ownership of external bus
Ext. bus controller	External master to FPI device	Not possible
	LMB master to external device	- Perform requested external access or accesses (burst) - Return result to requesting LMB master (read access only)

Switching of external bus ownership occurs in the following cases:

Table 14-19 Conditions for Changing Bus Ownership

Owner of external bus	Prerequisites	Trigger to change ownership in external master mode	Trigger to change ownership in external slave mode
Ext. bus controller	Previous access has finished or time-out after gaining ownership detected	Request of ext. master and no access pending	Request of ext. master
External master	Previous access has finished	LMB request or no request of ext. master any more	LMB request

14.7.2 Locking the External Bus

The EBU allows the external bus to be locked to perform any arbitrary uninterrupted sequence of external bus accesses. Two methods are allowed to lock the external bus:

- **Locked LMB/FPI Accesses**
  When the EBU has ownership of the external bus and is performing external bus accesses in response to a locked LMB or FPI access sequence, then ownership of the external bus will not be relinquished until the locked LMB access sequence has completed.
- **EBUCON.EXTLOCK bit**
  When bit **EXTLOCK** of register EBUCON is set, then the EBU will retain ownership of the external bus until “EXTLOCK” is subsequently cleared. If **EBUCON.EXTLOCK** is written to 1 when the EBU is the owner of the external bus, then this has immediate effect (i.e. the external master is immediately prevented from gaining ownership of the bus until **EBU_CON.EXTLOCK** is cleared). If **EBUCON.EXTLOCK** is written to 1 when the EBU is not the owner of the external bus, then this has no immediate effect. When the EBU subsequently gains ownership of the bus, then the external master is prevented from regaining ownership of the bus until **EBUCON.EXTLOCK** is cleared.

Note: There is no time-out mechanism associated with the **EBUCON.EXTLOCK** *bit. When the EBU is owner of the external bus with the* ***EXTLOCK*** *bit set, then the external master will remain locked off the bus until the* ***EXTLOCK*** *bit is cleared.*

14.7.3 EBU Reaction to an LMB Access to the External Bus

The reaction of the EBU to a request from an LMB master to access the external bus is as follows:

- If the EBU is operating in “no bus” mode, then it is impossible for an LMB or FPI master to access the external bus. For this reason, the EBU generates an LMB/FPI (as appropriate) error whenever an attempt is made to access the external bus while in “no bus” mode.
- If the EBU is operating in “sole master” mode, then it has access to the external bus at all times and as a result it is possible for the EBU to immediately perform the required external bus access.
- If the EBU is operating in “Arbiter” or “Participant” modes and receives a request for an external access from an LMB or FPI master when it is not the owner of the external bus (or is not able to retain ownership of the bus), then the request is rejected with a “retry”. This event also triggers the EBU to arbitrate with the external master in order to attempt to gain ownership of the external bus so that the request can be serviced when it is resubmitted by the master. This strategy ensures that the FPI Bus/LMB remain available while the EBU arbitrates for the external bus.

The strategy of issuing a retry (when the EBU is not the owner of the external bus) detailed above results in the potential for a locked bus condition. Consider the case of an LMB master issuing a request for an external bus access. The EBU rejects this

access with a retry (in order to retain LMB availability) but at the same time starts arbitration for ownership of the bus. Once ownership of the bus has been obtained, the EBU retains ownership until the next LMB to external bus access occurs. If the LMB master (or any other LMB masters) subsequently performs no external bus accesses (e.g. fails to re-submit the original access request), then the EBU would retain indefinite ownership of the bus and it would become impossible for the external master to access the external bus.

The EBU contains a time-out mechanism to avoid this lock condition. Once the EBU has gained ownership of the external bus, it will retain ownership only until one of these two occurs:

- An LMB or FPI to External Bus access occurs (as appropriate), or
- A (programmable) number of LMB cycles have elapsed (without an LMB to external bus access).

Once either of these conditions has occurred, the pending access is cancelled and the EBU will continue to arbitrate the external bus in the normal fashion. The desired time-out (number of LMB Clock cycles) is programmed by use of the **EBUCON.TOUTC** field (see **Section 14.12.5**)

14.8 EBU Boot Process

If external boot is selected—meaning that after reset, initial code execution begins from external memory—the EBU needs to access the external default or boot memory. However, because no application software has been executed yet, there is no information inside the EBU concerning the type of external memory and the proper access parameters to it.

To get around this problem, the EBU first starts a “blind” boot access to the external memory using a set of default values. This boot access is designed such that the EBU can access the external boot memory without knowing the exact parameters of it. In this way, the EBU retrieves additional detailed access information from a predefined location in the boot memory. It configures itself according to these parameters, and then performs the first true code fetch from location 0 of the boot memory, now with the proper access parameters.

Naturally, this boot access can handle only a limited variety of external boot memory types and access schemes. The boot memory must be either a ROM, EPROM, Flash-EPROM memory compatible to these types. The access is in demultiplexed mode only. The memory can be 16 or 32 bits wide. The boot memory must be connected to $\overline{CS0}$.

If external boot takes place, a configuration pin selects between external master or slave mode. The EBU will perform exactly one read access to a specific address (0x000004) in the external memory to read configuration data. During this fetch operation, any LMB request will be acknowledges with RETRY code. **Figure 14-20** gives an overview of this boot access.

Between reset becoming inactive and the access described in **Figure 14-20**, a gap of 256 clock cycles will be inserted to satisfy the recovery needs of external synchronous devices like Flash ROMs. During the read access the maximum number of programmable waitstates will be inserted (waitrdc * cmult = 7 * 32 = 224) and the evaluation of the $\overline{\text{WAIT}}$ signal will be inhibited.

It is assumed, that the EBU is in external master mode, when a boot from external memory is performed. That means, that during reset phase the $\overline{\text{HLDA}}$ signal will be pulled inactive (pull up) and that the EBU is owner of the external bus immediately after reset. When external boot is disabled, the EBU will come up with external bus arbitration turned off after reset, i.e. no access from LMB to external memory possible without EBU of TC11IB reconfiguration.

The configuration data must be coded in the following way:

BOOTCFG
EBU External Boot Memory Configuration Word

Boot Memory Offset Address + 04_H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CFG32	CMULT			BCGEN		WAIT		WAITINV	ADDRC		WAITRDC			0	

EBU of TC11IB can only support boot ROM 16-bit or 32-bit wide. A description of the bits, except for “CFG32” bit, can be found in the register description. If “CFG32”bit is set to 1, then the configuration data consists of a full 32-bit word, otherwise 16-bit. However for 32-bit word, the upper 16-bit is used for future extension. The data width of the boot ROM is determined as shown in **Table 14-20**:

Table 14-20 Boot Memory Data Width Encoding

CFG32	Boot Memory Data Width
0	01_B: 16-bit
1	10_B: 32-bit

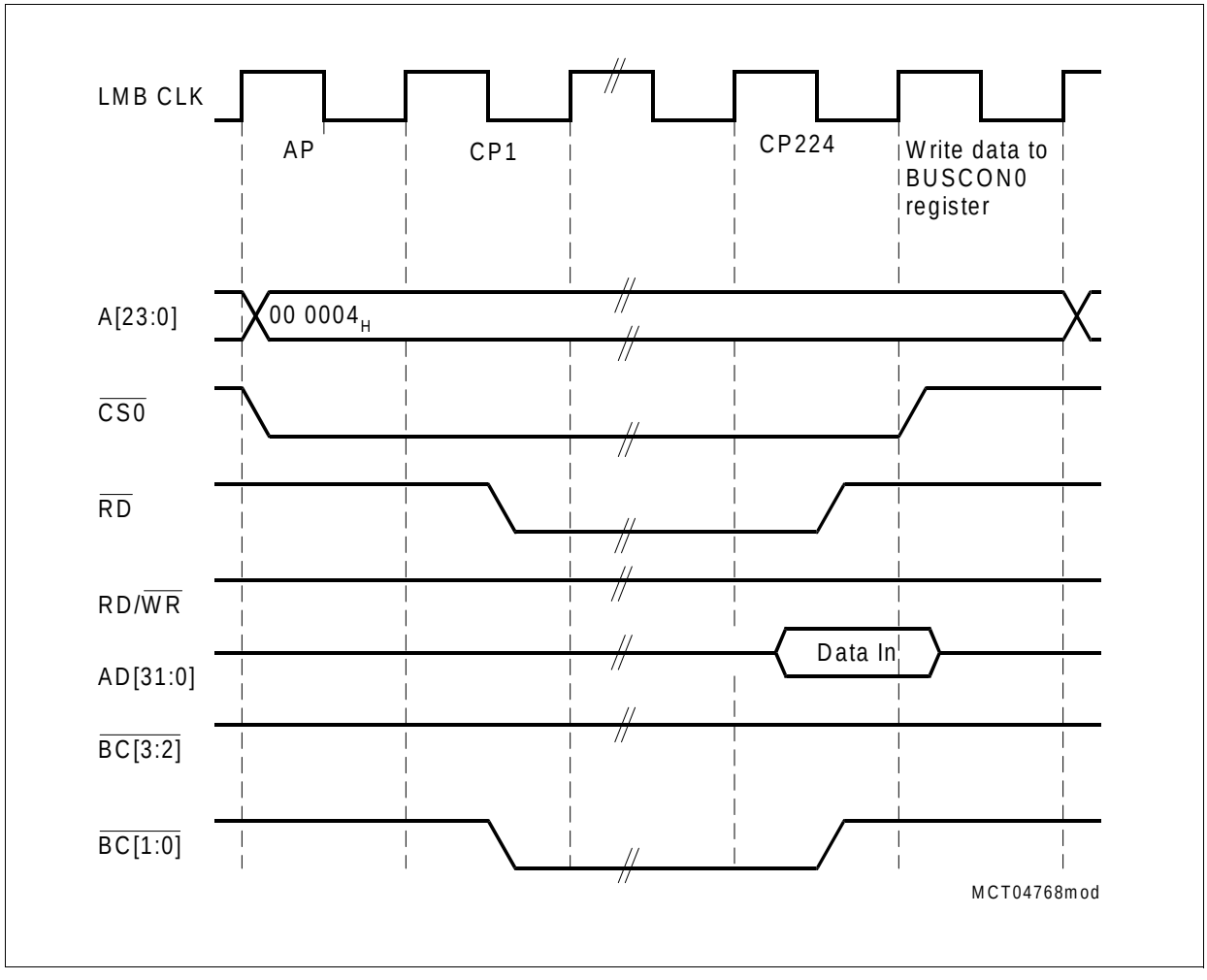


Figure 14-20 EBU Boot Process after Reset

14.9 Emulation Support

The TC11IB supports emulation and debugging via a number of measures. Some of the features are provided through the EBU via the external bus. A special emulation boot is provided after reset which activates the EBU to direct code and data accesses from the CPU to a dedicated emulator memory region. Additionally, accesses to application memory can be redirected to emulation memory during debugging and emulation to allow replacement of application memory contents with special emulation memory.

Note: The EBU uses special registers and signals for this emulation support. These resources are dedicated for these purposes and must not be used in normal operation. Proper emulation and debugging are not guaranteed and supported if these restrictions are not obeyed.

The following subsections describe the EBU emulation support in more detail.

14.9.1 Emulation Boot

One of the boot options of the TC11IB, selectable during reset, is to start execution out of a special external emulation memory. This memory is connected to the external bus of the EBU in a standard way, however, a special Chip Select, $\overline{\text{CSEMU}}$, is provided for this memory. The address range for this emulation memory is predefined to Segment 13, starting at address DE00 0000_H with a size of 16 MByte.

If emulation boot is selected during reset, then after the end of the reset sequence the Program Counter (PC) of the CPU is set to DE00 0000_H (pointing to the emulation memory) and the EBU is enabled. The EBU has an address select register, EBU_EMUAS, a bus control register, EBU_EMUBC, and a bus access parameter register, EBU_EMUBAP, dedicated to the emulation memory. The address area and access parameters in these registers are set to predefined default values for a certain type of emulation memory. Thus, the EBU does not need to perform a boot access to the memory to retrieve further configuration data, as required for a normal boot. The code fetch requests from the CPU activate the EBU, which in turn performs a respective access to the emulation memory.

In this way, emulation software instead of application software is executed directly after reset. After having performed necessary initialization and programming, the emulation software usually executes a soft reset with the proper boot configuration to perform a normal boot and returning to the application software.

14.9.2 Overlay Memory

During emulation and debugging, it is often necessary to modify or replace the application code. While this is not very difficult to do with easily writable memories, such as RAMs, it can be awkward or even not possible without removing the memory or adding special provisions for on-board reprogramming when the code is stored in non-volatile memory such as a ROM or an EPROM.

The solution to this problem provided by the EBU is an overlay memory Chip Select, $\overline{\text{CSOVL}}$. This Chip Select line can be programmed to be active in addition to the normal Chip Select connected to the application memory. An additional overlay memory can then be connected to the external bus, using this overlay Chip Select to activate it. Additionally, the $\overline{\text{CSOVL}}$ line is used to gate the read and write signals to the application memory. **Figure 14-21** gives an overview for such a configuration. Only the signals relevant for this feature are shown.

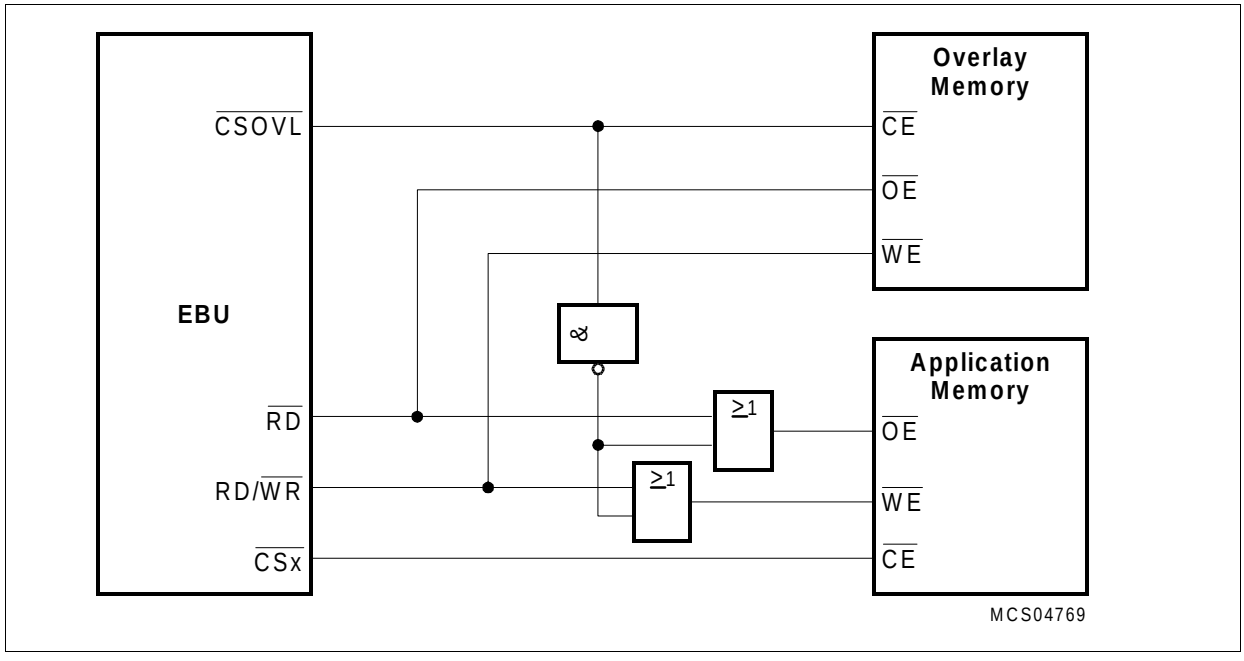


Figure 14-21 Use of the Overlay Chip Select

If the overlay Chip Select option is selected for an address range, then it is activated for all accesses to this address range in addition to the activation of the regular Chip Select, $\overline{CSx}$. Thus, the overlay memory is activated for these accesses. To ensure that the regular application memory is not driving the bus on a read or storing the data on a write, the inverted overlay Chip Select controls two OR-gates to disable the read and write signals to the memory. In this way, the overlay memory is accessed instead of the application memory.

The selection of the overlay Chip Select is performed through register EBU_EMUCON. For each of the seven regular Chip Selects $\overline{CS}[6:0]$, an enable bit for $\overline{CSOVL}$ is provided. It is possible to activate $\overline{CSOVL}$ for one or more of the regular Chip Selects.

Note: To guarantee proper access, the overlay memory must meet the same access requirements as the application memory. The access to it is performed according to the parameters programmed for the application memory via the EBU_BUSCONx and EBU_BUSAPx registers associated with the regular Chip Select.

*Note: Use of the overlay Chip Select feature is intended for emulation support. The circuitry shown in **Figure 14-21** is usually provided on the emulator probe. It does not need to be included in the application circuitry.*

14.10 External Instruction Fetches

This section describes the synchronous burst Flash memory accesses that are initiated and controlled by the PMU and which use the EBU lines for external access. The EBU of the TC11IB supports two types of burst mode Flash memories:

– AMD 29BL162
– Intel 18F800F3 and 28F160F3

14.10.1 Signal List

The signals shown in **Table 14-21** are used for synchronous burst Flash memory accesses and are part of the EBU interface:

Table 14-21 EBU Signals used for Burst Flash Memory Accesses

Signal	Type	Function
AD[31:0]	I/O	Data bus
$\overline{\text{RD}}$	O	Read control
A[23:1]	O	Address bus
$\overline{\text{ADV}}$	O	Address valid strobe
$\overline{\text{BAA}}$	O	Burst address advance
$\overline{\text{WAIT}}$	I	Wait/terminate burst control
$\overline{\text{CS}}$[7:0]	O	Chip select
EBUCLK	O	External EBU clock output
ACLK	O	Slower EBU clock, equal 1/2, 1/3 or 1/4 of the frequency of the EBUCLK

14.10.2 Basic Functions

The EBU is designed to perform burst mode cycles for an external code Flash memory. These burst mode cycles are executed via a separate instruction fetch bus as shown in **Figure 14-1**. In general, the burst mode cycle capability provides the following features:

- Fully synchronous timing with flexible programmable timing parameters (address cycles, read wait cycles and data cycles)
- Programmable WAIT function
- Programmable burst (mode and length)
- 16-bit or 32-bit data bus width
- Support of INTEL 28F800F3 and 28F160F3 Fast Boot Block Flash Memory
- Support of AMD 29BL162 Burst Mode Flash Memory

Figure 14-22 shows the basic configuration of external burst Flash memory connections.

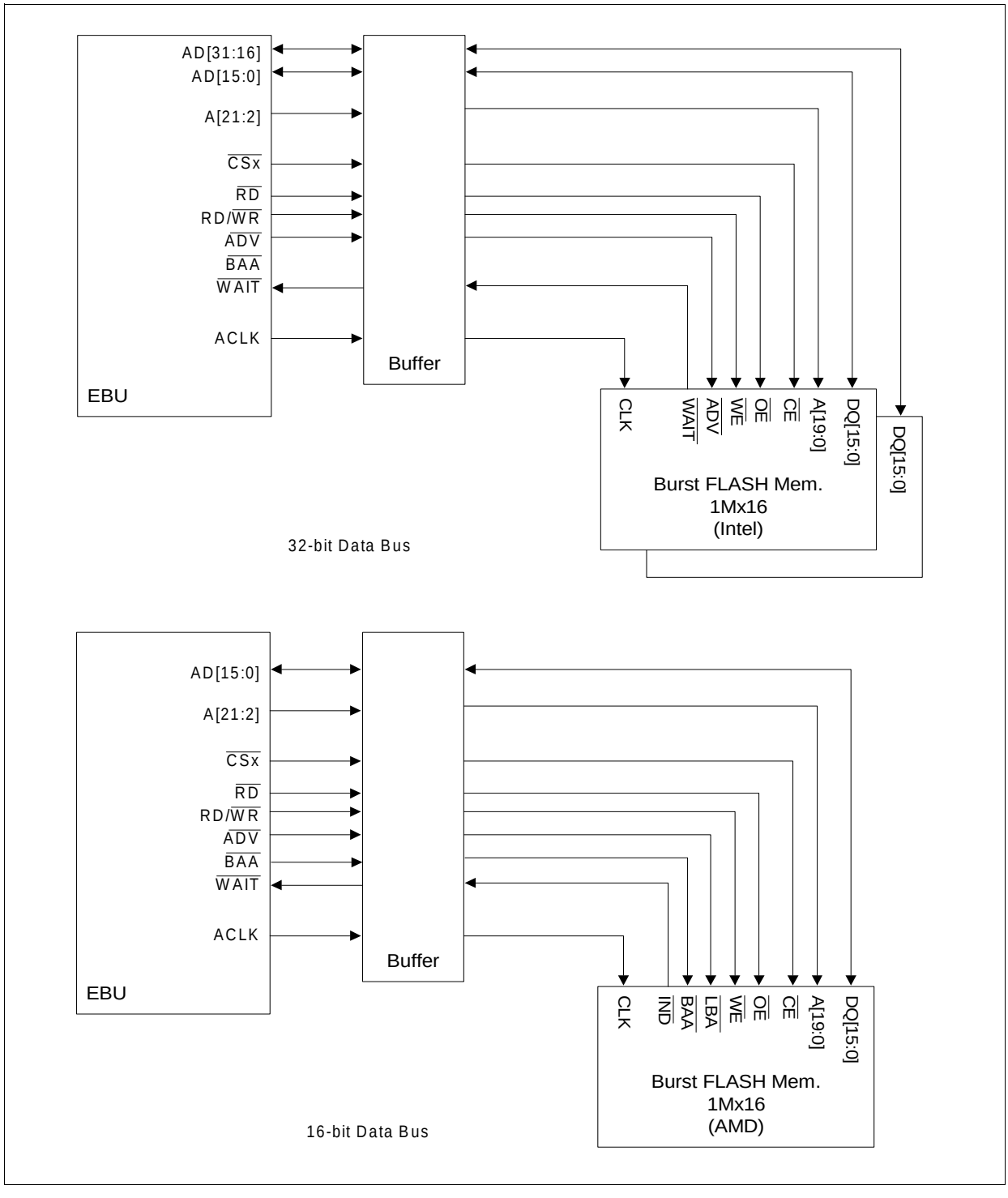


Figure 14-22 Example of Connection with Intel/AMD Flash Devices

Flash devices are started as asynchronous devices. Software must set the suitable timing parameters to write to an internal device register (Intel's Burst Flash) or sequence of write access (AMD's Burst Flash) in order to switch to synchronous mode.

14.10.3 Cycle Definitions of Burst Mode Timing

Accesses to Burst Flash devices are composed of a number of Standard Access Phases (details in **Section 14.5.5.1**).The timing diagrams on the following pages use abbreviations for the clock cycles:

- Fully synchronous timing
  - AP Address Phase (1 to n cycles), must start at the rising edge of ACLK
  - CD Command Delay (0 to n cycles)
  - CP Command Phase (1 to n cycles)
  - BP Burst Phase (1 to n cycles), repeatable to complete the burst length
  - RP Recovery Phase (0 to n cycles)
- Programmable number of cycles for each phase: Address Phase, Command Delay, Command Phase, Burst Phase and Recovery Phase
- Programmable fetching burst length

Figure 14-23 shows the basic timing of a synchronous burst mode operation.

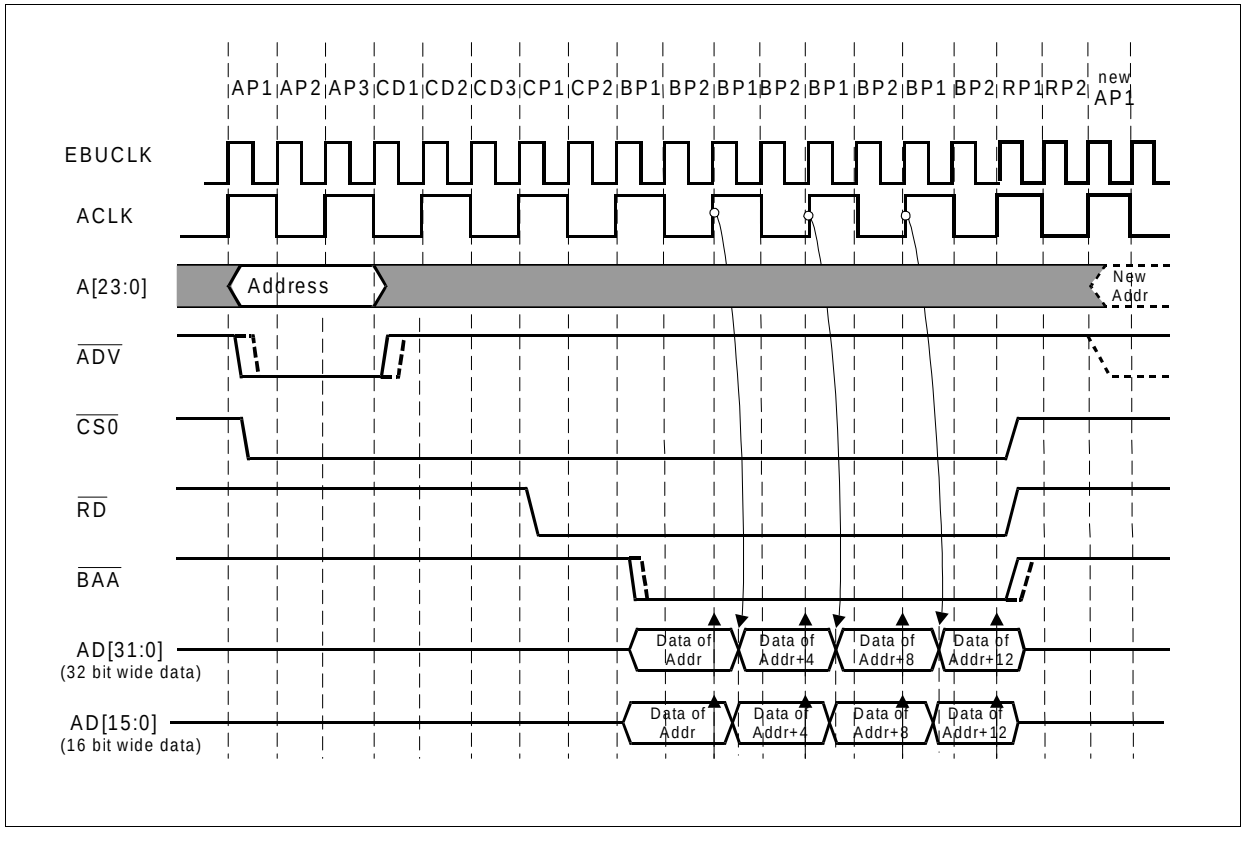


Figure 14-23 Synchronous Burst Read Operation (4 Word Burst)

Because the EBUCLK runs too fast for external devices, there is a second clock source running at equal, 1/2, 1/3, or 1/4 of the EBUCLK frequency. This clock (ACLK) can be used by external devices for synchronous access. Accesses for devices using this clock

must be programmed so that critical phases can be guaranteed valid sampling on ACLK rising edge. However other signals are relative to EBUCLK.

The $\overline{ADV}$ and $\overline{BAA}$ signals go low on the negative edge of EBUCLK due to delay by default. The EBU allows these delays (1/2 an EBU clock cycle) to be removed via **EBASE** parameter in BFCON register. Resultant signal timing will be as **Table 14-22**.

Table 14-22 $\overline{ADV}$/$\overline{BAA}$ Signal Timings

Signal	Delay Disabled	Delay Enabled
ADV	Start of AP1	Middle of AP1
BAA	Start of BP1	Middle of BP1

The Burst Phase must be repeated for the rest of fetching burst length (programmable through **FETBLEN** = 1, 2, 4 or 8), which select the maximum number of Burst Phases in a single access. When an LMB request exceeds the amount of data that can be fetched by the programmed number of Burst Phases, the EBU will automatically generate the appropriate number of burst access to supply the required amount of data. Selection of Continuous Burst Mode (by use of **FBBMSEL**) overrides the maximum burst setting.

The programmability of the length of the Address, Command Delay and Command Phases allows flexible configuration to meet the initial read access time of a Burst Flash device.

Figure 14-23 shows an example of a burst length of 4 and the following parameters:

- ADDRC = 3 (BUSAPx register),
- CMDDELAY =3 (BUSAPx register),
- WAITRDC = 2 (BUSAPx register),
- BURSTC = 2 (BUSAPx register),
- RDRECOVC = 2(BUSAPx register),
- FETBLEN = 4 (BFCON register),
- PORTW = 16 & 32 (BUSCONx register),
- EXTLOCK = 1/2 of EBUCLK frequency

.

Table 14-23 Programmable Access Phase

Access Phase	# Cycles	Parameter
Address Phase	addrc	addrc = 1 to n

Table 14-23 Programmable Access Phase

Access Phase	# Cycles	Parameter
Command Delay	CMDDELAY	CMDDELAY = 0 to n
Command Phase	read access: **WAITRDC**	WAITRDC = 1 to n
Burst Phase	BURSTC	BURSTC = 1 to n
Recovery Phase	switch to different $\overline{CS}$: **DTACS** after every read: **RDRECOVC**	DTACS = 0 to n RDRECOVC = 0 to n

14.10.4 External Cycle Control via the $\overline{WAIT}$ Input

The EBU provides control of the Burst Flash device via the $\overline{WAIT}$ input. This allows the EBU to support operation of Burst Flash while crossing Burst Flash page boundaries. During a Burst Flash access, the $\overline{WAIT}$ input operates in one of three modes in TC11IB:

- Disabled
- Asynchronous Wait for Page Load (Intel devices).
- Synchronous Terminate and Start New Burst (AMD device).

Selection of the mode in which the $\overline{WAIT}$ input operates during Burst Flash accesses is selected via the **BUSCON[6:0].WAIT** bit field (see **Section 14.12.3**) and the **BFCON.WAITFUNC** bit (see **Section 14.12.6**). The **WAITFUNC** bit selects either Wait for Page Load or Terminate and Start New Burst mode. The **WAIT** bit field selects one of three sampling modes:

- Synchronous Sampling
- Asynchronous Sampling
- Disabled

*Note: Selection of "Disabled" via the wait bit field prevents the $\overline{WAIT}$ input having any effect on a Burst Flash access cycle regardless of the setting of the **WAITFUNC** bit.*

14.10.4.1 Asynchronous Wait for Page Load Mode (Intel)

In this mode, the $\overline{WAIT}$ input being asserted low during a burst access causes the EBU to complete the current Burst Phase and then to perform an additional Burst Phase (if the previous BP was not the last of the current access). Following this, the next Burst Phase (again if this BP required) will not be started until the $\overline{WAIT}$ input is de-asserted. In addition, the start of the next Burst Phase is also synchronized to the next rising edge of the ACLK signal to ensure that the EBU continues to sample data from the device at the correct time. This mode allows an Intel Burst Flash device to temporarily suspend the

burst sequence in order to perform a page load when a page boundary is crossed during the access. This mode supports the use of Intel Burst Flash devices configured for Early Wait Generation Mode.

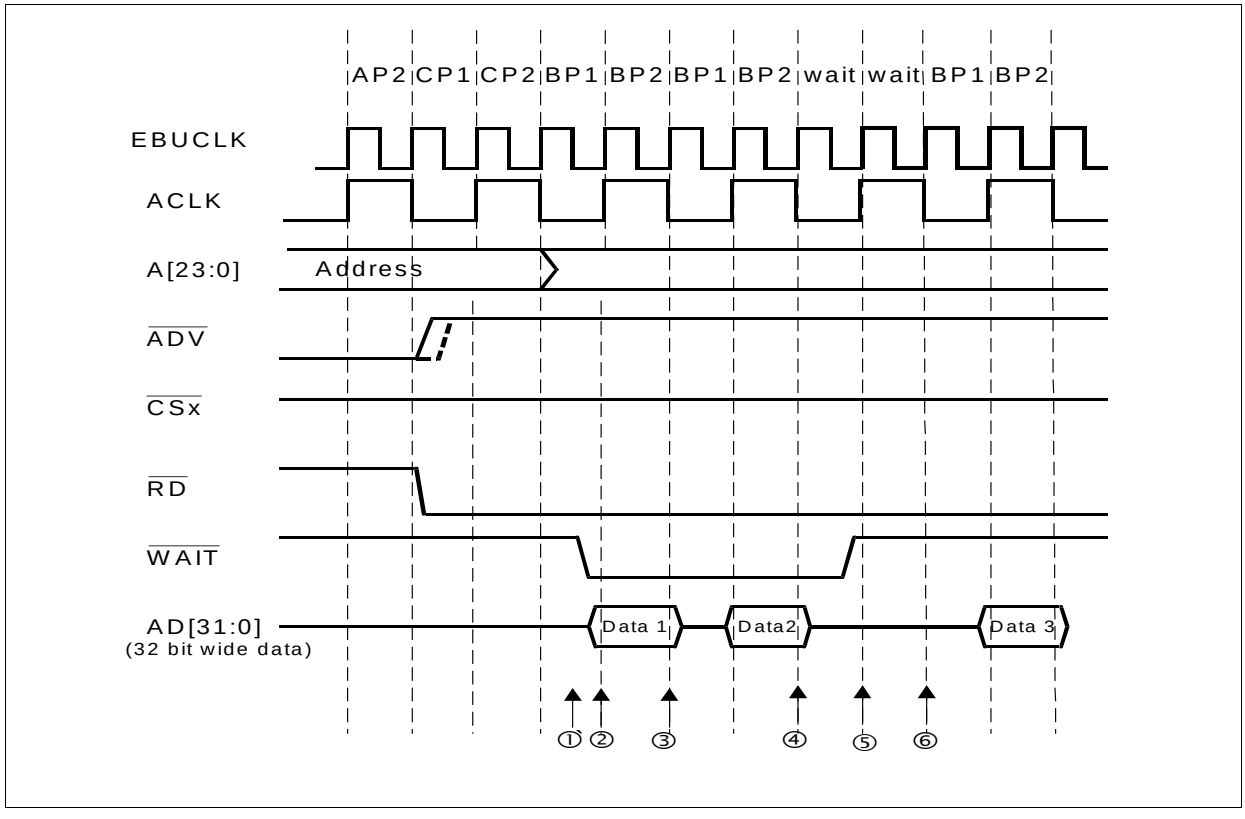


Figure 14-24 $\overline{\text{Wait}}$ Input in Asynchronous Wait for Page Load Mode (waitfunc = 0)

Figure 14-24 shows an example of the insertion of wait cycles in Asynchronous Wait for Page Load Mode:

1. The Burst Flash drives $\overline{\text{WAIT}}$ low to signal that after two data values have been read. a page boundary will be crossed (the Flash is operating in Early Wait Generation Mode).
2. The $\overline{\text{WAIT}}$ input is sampled low on the rising edge of EBUCLK.
3. The sample taken at ,1 is re-synchronized to the rising edge of ACLK.
4. The re-synchronized sample (from “2”) is evaluated and causes the insertion of wait cycles.
5. The $\overline{\text{WAIT}}$ input is sampled high on the rising edge of EBUCLK prior to an ACLK rising edge.
6. The sample taken at “4” in conjunction with the rising edge of ACLK allows the EBU to start the next Burst Phase.

14.10.4.2 Synchronous Terminate and Start New Burst Mode (AMD)

In this mode, the $\overline{\text{WAIT}}$ input being asserted low during a burst access causes the EBU to complete the current Burst Phase and then to terminate the current access. If additional data is required, the EBU will start a new access cycle (i.e. Address Phase, Command Delay Phase, Command Phase, Burst Phase etc.). The EBU calculates the address to be issued during the new Address Phase to ensure correct sequential reading of the Burst Flash device(s). This mode allows an AMD Burst Flash device to cause the EBU terminate the current access when a page boundary is crossed and to issue a complete new burst read access. This allows the Burst Flash device to perform a page load at the start of the new access. This mode supports AMD Burst Flash Devices in Terminate mode.

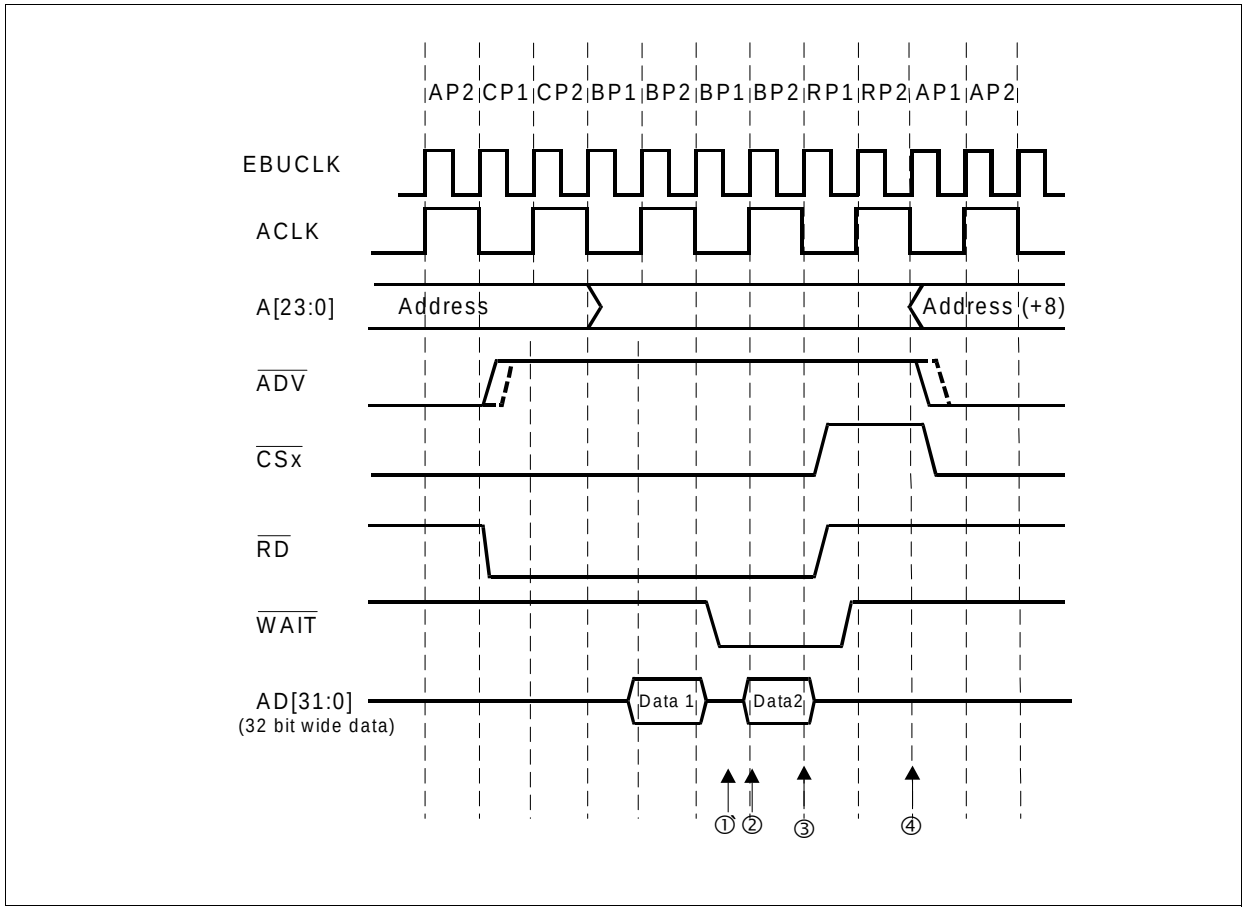


Figure 14-25 $\overline{\text{WAIT}}$ Input in Synchronous Terminate and Start New Burst Mode

Figure 14-25 shows an example of the termination of a burst access and the start of a new access cycle in response to the $\overline{\text{WAIT}}$ input asserted low in Synchronous Terminate and Start New Burst Mode:-

1. The Burst Flash drives $\overline{\text{WAIT}}$ low to signal that after the current data value has been read a page boundary will be crossed.

2. The WAIT input is sampled low on the rising edge of EBUCLK.
3. The sample (from “2”) is evaluated and causes the termination of the current access.
4. After the appropriate Recovery Phase, the EBU issues a new Burst Flash access (synchronized to an ACLK rising edge). The burst start address is automatically updated by the EBU to read the appropriate data (the example shows a 32-bit wide bus).

14.10.5 Termination of a Burst Access

A burst read operation is terminated by de-asserting $\overline{CSx}$ signal followed by the appropriate length Recovery Phase. **Figure 14-26** shows termination of a burst access following the read of two locations (i.e. two Burst Phases) from the Burst Flash device(s).

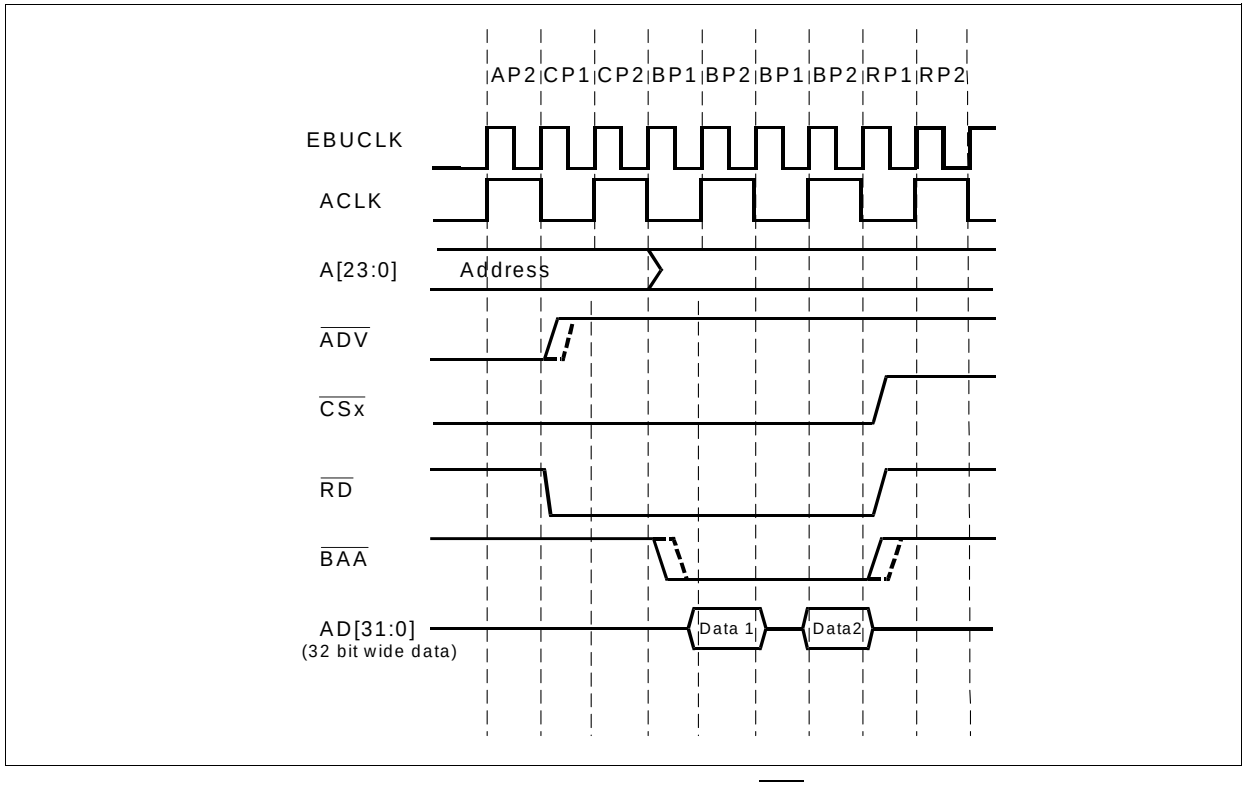


Figure 14-26 Termination a Burst by Deasserting $\overline{CS}$

Terminating a burst is important, if the LMB’s burst request is shorter than the fetch burst length. For example when LMB request is 2 beats of 64-bit data (i.e. 4 x32-bit data) and fetch burst length of the device is 8, then termination comes after the fourth data being latched. Also when dealing with continuous burst, deasserting $\overline{CSx}$ will terminate the burst.

14.11 SDRAM Interface

The TC11IB SDRAM interface supports 64 MBits (organized as 4 banks x 1M x 16), 128 MBits (as 4 banks x 2M x16), and 256 MBits (as 4 banks x 4M x16) SDRAMs. The EBU can simultaneously support two SDRAM memories, Type 0 and Type 1. Each has different access/refresh parameters and corresponds to the SDRAM region with a Chip Select signal CSx.

SDRAMs are synchronous DRAMs with burst read/write capability which are controlled by a set of commands at the pins $\overline{CS}$, $\overline{RAS}$, $\overline{CAS}$, $\overline{WE}$, DQM, A10. A SDRAM contains multiple DRAM banks which are addressed by bank select(s) and multiplexed addresses (row / column address). A periodic refresh must be performed like in standard DRAMs.

Features

- PC100 compatible (if multiplexed devices are not connected to the external bus)
- Multibank support
- Interleaved access support
- Supports 64, 128 and 256 MBits SDRAM devices
- Maximum address space of 128 MBytes, support of 16- and 32-bit wide
- Very high bandwidth > 200 MByte/s for 32-bit wide access
- Individual SDRAM parameters for each $\overline{CS}$ strobe (up to two different SDRAM types)
- Autorefresh mode support for power-down mode
- Data types: halfword and word for single reads and word for burst reads
- Power-on/mode-set sequence triggered by LMB write to SDRAM configuration register
- Programmable refresh rate
- Programmable timing parameters (row-to-column delay, row-precharge time, mode-register setup time, initialization refresh cycles, refresh periods)

The supported SDRAM devices include (but not limited to) the following:

- Infineon, HYB39S16160, HYB39S256160
- IBM, IBM0325164
- Samsung, KM416S1020
- Micron, MT48LC2M32, MT48LCM4M16, MT48LC16M16
- Hyundai, HY57V161610, HY57V651620
- Hitachi HM5225165

14.11.1 Signal List

The following signals are used for the SDRAM interface:

Table 14-24 SDRAM Signal List

Signal	Type	Function
AD(31:0)	I/O	Data bus
A(23:0)	O	Address bus
RD/$\overline{WR}$	O	Read and write control
$\overline{BC}$(3:0)	O	DQM output control
CKE	O	Clock enable
$\overline{CS}$(7:0)	O	Chip select
EBUCLK	O	External EBU of TC11IB Clock
$\overline{CAS}$	O	Column address strobe
$\overline{RAS}$	O	Row address strobe

14.11.2 External Interface

The external interface is PC100 compatible and can be directly connected to two sets of DRAM chips without any glue-logic. To maintain pin loading requirements, all other devices can be connected only at a buffered bus extension (i.e. only two components may be directly connected to the EBU of TC11IB address and data buses and certain control signals). Additionally, special board layout constraints apply.

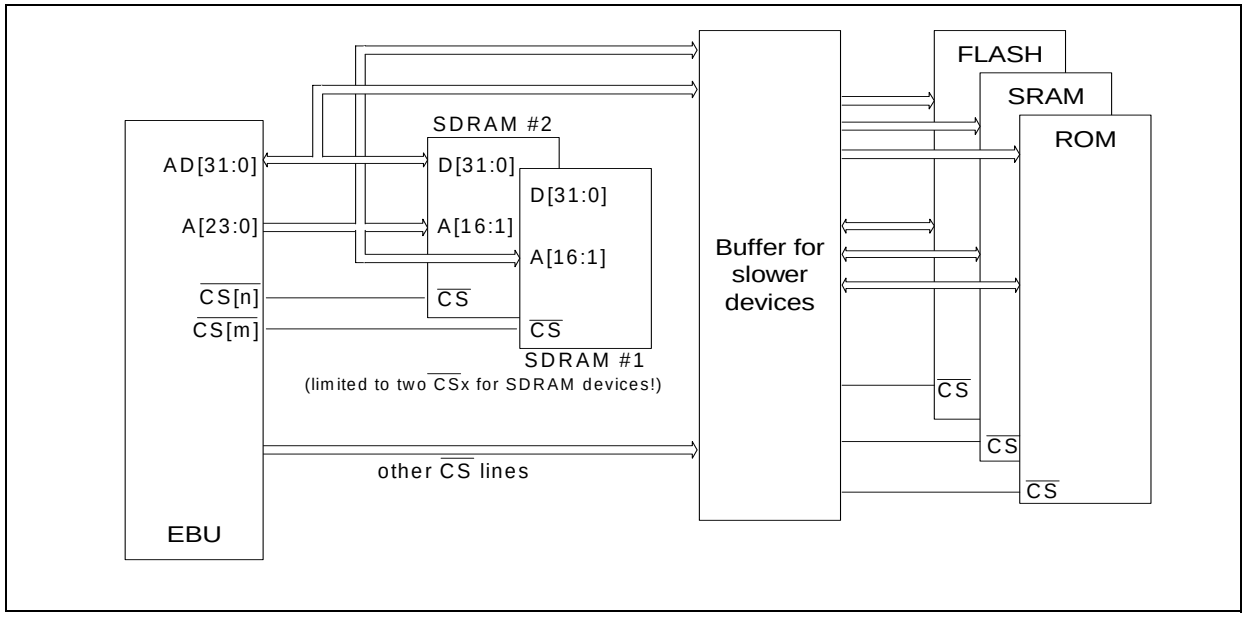


Figure 14-27 Connectivity for SDRAM

Multibanking is supported in order to allow interleaved banks accesses. When a bank is already opened, it will be registered so that access to the same row within the bank can be sped up. Comparison of banks are done prior to initiating external memory accesses.

14.11.3 Supported SDRAM Commands

Table 14-25 lists the supported commands, how they are being triggered and which signals are activated.

Table 14-25 Supported SDRAM Commands

Command	Event	CKE (n-1)	CKE (n)	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	RD/$\overline{WR}$	See Table 14-30 and Table 14-31 for EBU of TC11IB pins			
								A11	A10	BA (1:0)	A (9:0)
Device deselect	region not selected	H	-	H	-	-	-	-	-	-	-
Nop	idle	H	-	L	H	H	H	-	-	-	-
Bank activate	open a closed bank	H	-	L	L	H	H	valid address			
Read	read access	H	-	L	H	L	H	valid addr	L	valid address	
Write	write access	H	-	L	H	L	L	valid addr	L	valid address	
Precharge selective	bank or page miss	H	-	L	L	H	L	-	L	bank	-
Precharge all	refresh is due or going into power down	H	-	L	L	H	L	-	H	-	-
Auto-refresh	refresh is due, after precharge all is done	H	H	L	L	L	H	-	-	-	-
Self refresh entry	going into power down after precharge all is done	H	L	L	L	L	H	-	-	-	-
Self refresh exit	coming out of power down	L	H	H	-	-	-	-	-	-	-
Mode register set	during initialization	H	-	L	L	L	L	valid mode (see register SDRMOD)			

14.11.4 Power Up Sequence

During power-up the SDRAM should be initialized with the proper sequence. This includes the requirement of bringing up the V_{DD}, V_{DDQ} and the stable clock (minimum 100 µs before any accesses to SDRAM) and $\overline{CS}$ remains inactive.

14.11.5 Initialization Sequence

SDRAMs must be initialized before being used. At least one NOP cycle must be issued after 1 ms of $\overline{CS}$ inactive (device deselect). This must be followed by 200 µs pause by software and then a Precharge All Banks command. Following this, the device must go through Auto Refresh Cycles (the number of refresh commands is programmable through **CRFSH** in SDRMCON[1:0] registers and the number of NOP cycles in between is programmable through **CRC**). At the end of it, the Mode Register must be programmed through the address lines. Following that some number of NOP cycles programmable through **CRSC** in SDRMCON[1:0] registers. This sequence must be carried out for each of SDRAM type

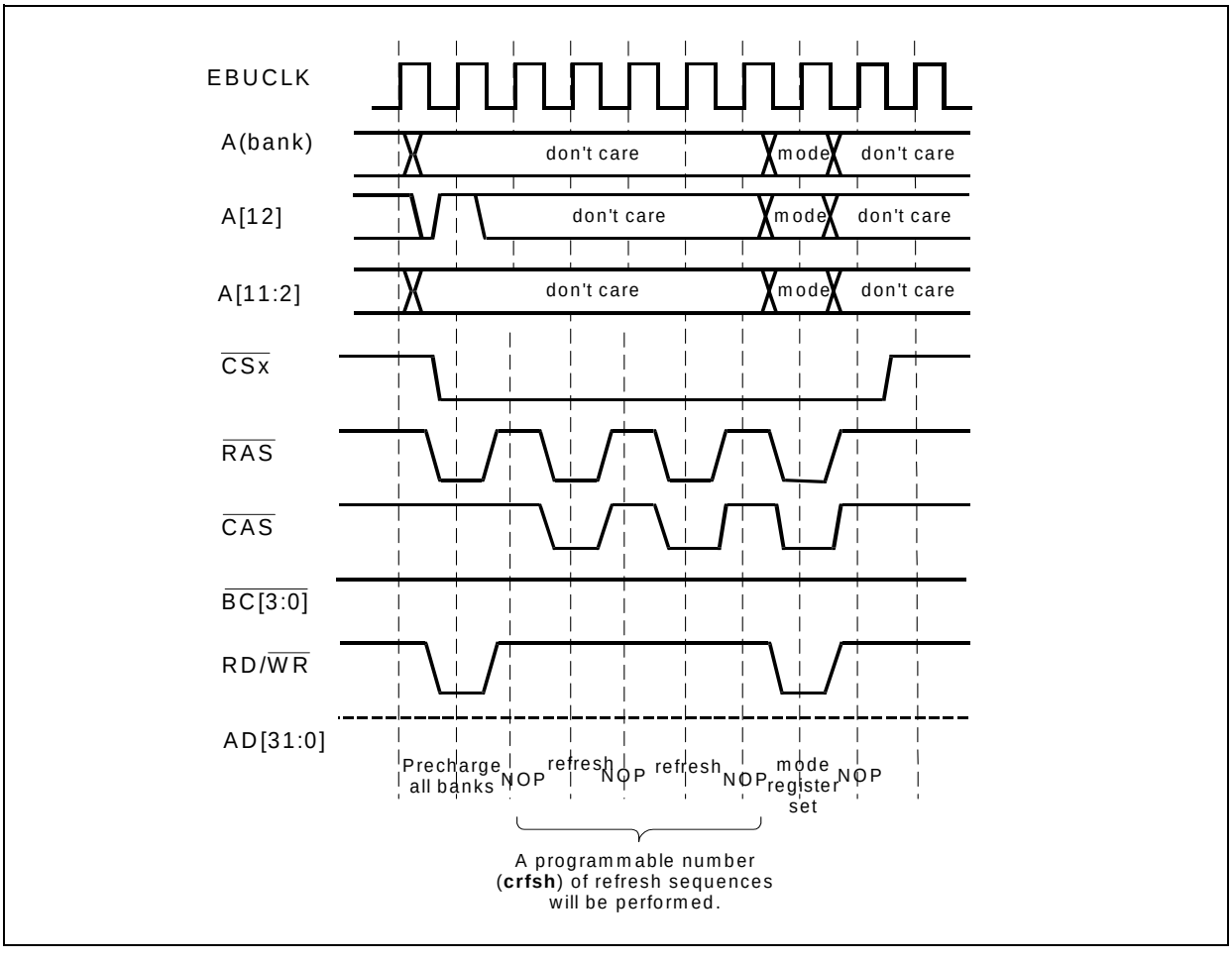


Figure 14-28 SDRAM Initialization

The sequence is triggered by a write to the SDRAM mode register SDRMOD0 or SDRMOD1. All the regions having **AGEN** in BUSCONx set to ’011’ will be configured with the mode from SDRMOD0, while regions with **AGEN** equals ’100’ configured from SDRMOD1. While this sequence is being executed, **SDRMBUSY** flag in the SDRMSTAT0,1 status register will be set accordingly.

The setting sequence of corresponding registers is critical. Especially the SDRMODx registers, they must be the last one to set because this write will trigger the initialization sequence. The recommended the sequence of setting registers is as follow:

1. ADDRSELx
2. BUSCONx
3. BUSAPx (not relevant to SDRAM)
4. SDRMCONx
5. SDRMREFx
6. SDRMODx (must be the last one)

The contents of register SDRMOD0,1 will be written to the SDRAM mode register in each device at the end of this sequence via the address pins A[15:2], if the port width is set to 32-bit or A[14:1] if the port is 16-bit. The user must make sure that the SDRAM is programmed in the following way:

Table 14-26 SDRAM Mode Register Setting

Field	Value	Meaning	SDRMOD0, 1 Position	Corresponding Address Pins	
				32-bit	16-bit
Burst length	“011” “010” “001” "000"	bursts of length 8 reserved reserved bursts of length 1	**BURSTL**[2:0]	A[4:2]	A[3:1]
Burst type	‘0’	sequential bursts	**BTYP**[3]	A[5]	A[4]
$\overline{\text{CAS}}$ latency	“001” “010” “011” “100”	latency 1 (for test only) latency 2 latency 3 latency 4	**CASLAT**[6:4]	A[8:6]	A[7:5]
Operation Mode	all ‘0’	burst read and burst write	**OPMODE** [13:7]	A[15:9]	A[14:8]

The EBU of TC11IB uses the $\overline{\text{CAS}}$ latency value and burst length to adjust the burst read timing. All other fields have no influence on the EBU of TC11IB, which means only single value is accepted for those fields.

The complete initialization sequence described will only be issued on the first write (since reset) to the relevant SDRMOD register. On subsequent writes, the SDRAM device does not need to be initialized, so a simple mode register set command can be issued. A precharge-all command needs to be issued to the SDRAM before this can happen.

14.11.6 SDRAM Burst Accesses

In order to deliver good performance, the EBU of TC11IB only supports burst length of 1 and 8. Other burst lengths are supported but they are implemented together with the data masking operation. Code prefetching is at most done with burst length of 8, while data load/store can be of variable length.

Figure 14-29 shows the read performance of the SDRAM interface. The following assumption is made:

- LMB clock is 1:1 to the external bus clock (EBUCLK)
- Program Memory Unit (PMU) requests for instructions.
- SDRAM commands (e.g. Bank-Activate to open a bank) is generated once it is confirmed that the address requested maps to an SDRAM region.
- Two clocks (due to the $\overline{\text{CAS}}$ latency, caslat = 2) after read command being issued, the first word of the burst start appears on the external data bus.
- To satisfy the current LMB request, two 32-bit words are needed to compose a 64-bit instruction double word.

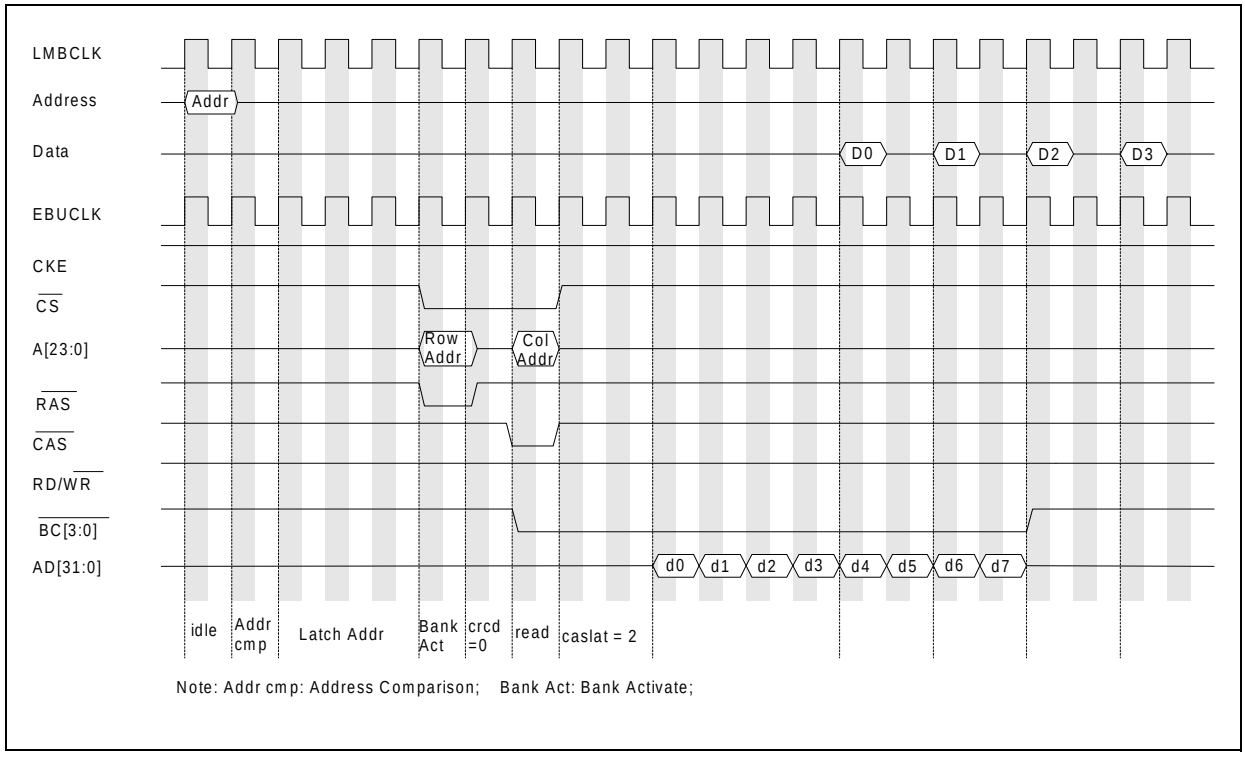


Figure 14-29 SDRAM Read Access To a Closed Bank (CAS latency of two)

As the FIFO buffer being emptied (through LMB data bus), prefetch unit (if enabled) can initiate the next burst so that there is no gaps or bubbles in the external data bus. This burst will fill up the FIFO prefetch buffer. Basically it is possible to have continuous burst on the external data bus without any gaps, provided that the PMU is also able to empty the FIFO buffer in time.

When single 16-bit SDRAM is being used (instead of 32-bit), the FIFO prefetch buffer needs to collect four 16-bit half-words to make up a 64-bit instruction double-word. Therefore, the timing needs to be adjusted accordingly.

The timing for instruction fetching is same as the data fetching shown above. **Figure 14-30** shows a data write happens to an open bank after an instruction fetching. Comparing with an access to a close bank, an access to an open bank skips the row access.

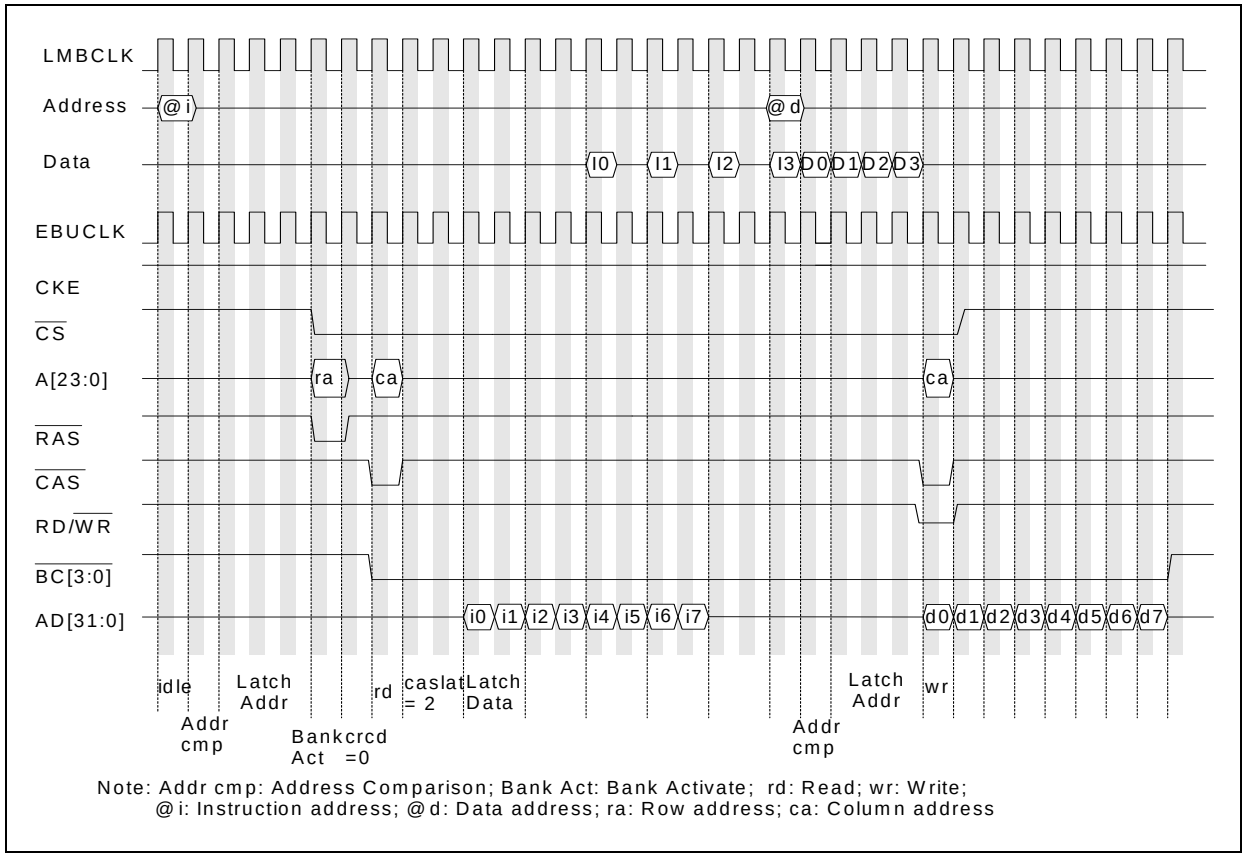


Figure 14-30 SDRAM Write Access To an Open Bank After Instruction Fetching

To support burst length shorter than 8, the interface will simply use data masking. However burst length of 1 is supported without data masking. When the data masking is activated (through DQM) during a read cycle, the data output are disabled and become high impedance after a two clock delay, independent of $\overline{CAS}$ latency. If the data masking is activated during a write cycle, the write operation is prohibited immediately (zero clock latency). Data masking could be activated through the $\overline{BCx}$ outputs which are connected to DQM during the SDRAM access.

14.11.7 Multibanking Operation

The design supports up to 4 banks (2 for instruction and 2 for data) being opened at the same time for each SDRAM type. To speed up execution, the comparison of bank identifiers must happen at the same time with region comparison (i.e. after the LMB address phase is recognized). In the next cycle then, it can be decided whether it is a page hit or a page miss access.

If the next access is to an open bank (comparing to all four banks just to cater for the possibility of mixed code and data in one region), it will then proceed with one of the access commands without having to close one of the bank. If nothing matches, then it must close one of the banks. If the access is for instruction fetching, then it must close one of the 'instruction' banks. If the access is for data access, then it must close one of the 'data' banks.

A random retirement strategy will close one of the open banks to make way for a new open bank.

14.11.7.1 Bank-Page Tag Structure

The EBU of TC11IB can open up to four banks at one time for each of SDRAM type. The opened banks are stored as address bits associated with the banks. **Table 14-30** and **Table 14-31** show how banks and pages/rows are recognizable from the address bits. For example, if one region of SDRAM is configured as: 32-bit wide, having 4 banks in the device, 8192 of rows and 512 of columns, then tag for each of the bank is bits 24 to 31 of the address (Address[31:24]). Each open bank has an associated open page, and for example the page tag is Address[23:11].

Each pair of bank-page tag has a validity bit. Upon reset all of these bits are zero. Out of four bank-page tag pairs, the first two pairs are allocated for instructions and the other two for data. All requests from PMU are recognized as instructions requests, while others are data requests.

Each time there is a new LMB request, the address is compared against all valid bank-page tag pairs. After one clock cycle, there will be two decisions to make. If the current access is targeted to recognized SDRAM region(s), then the EBU of TC11IB must recognize whether the requested address is a page-hit and whether it's a bank hit. **Figure 14-31** illustrates the decision process.

14.11.7.2 Bank Mask and Page Mask

The EBU of TC11IB needs to generate a bank hit and a page hit signal. To generate a bank hit, the LMB address must be applied with a bank mask. Also to generate a page hit, it must be applied with a page mask. The bank mask and the page mask depends on the following configuration of SDRAMs being used:

- Number of banks in the device: 2 or 4
- Data-width: 16-bit or 32-bit

– Number of columns: 256, 512, 1024 or 2048
– Number of rows/pages: 2048, 4096, 8192

Based on the above options, the page mask can be one of the followings (address bits 31-27 are surely compared in the address comparison, see **Section 14.5.1**):

– bit-26 to bit-9
– bit-26 to bit-10
– bit-26 to bit-11
– bit-26 to bit-12
– bit-26 to bit-13

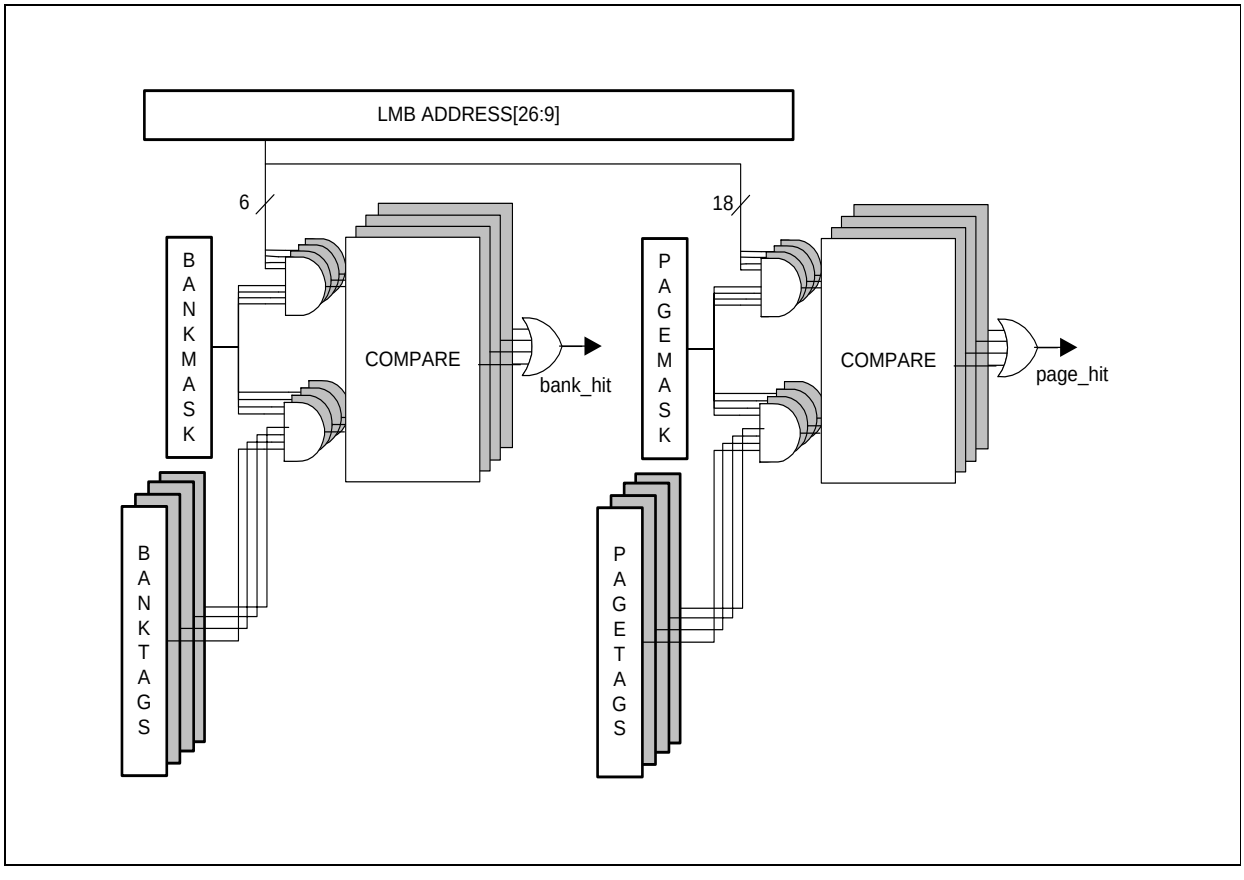


Figure 14-31 Generating Bank_hit And Page_hit

While the bank mask is one of the following:

– bit-26 to bit-20
– bit-26 to bit-21
– bit-26 to bit-22
– bit-26 to bit-23
– bit-26 to bit-24
– bit-26 to bit-25
– bit-26 only

In order to avoid complexities, user must choose which one of the bank mask and page matches the configuration of the SDRAM being used. The parameters that need to be programmed are **BANKM** and **PAGEM** in SDRMCON0,1 registers.

14.11.7.3 Decisions over Page_hit and Bank_hit

Generally, a page hit also means a bank hit, but a page miss does not necessarily mean a bank miss. When a page hit occurs, EBU of TC11IB can continue the access operation without updating the stored tag pairs of bank-page. Unfortunately, a page miss can result in several other activities.

When a page miss and a bank hit occur, it means the EBU of TC11IB must close current bank, i.e. do a precharge. This is then followed by (re-)activating the bank and continue with the access operation. The current bank need not to be invalidated, but the new page tag must be stored.

For a page miss and bank miss event, look first for an instruction request (from PMU). If there are already two valid pairs of instruction bank-page tags being stored, one of the pairs must be randomly invalidated. This means that the EBU of TC11IB must issue a precharge operation to close particular bank. Following this, the new pair of bank and page tag is stored at the invalidated/vacant slot and the bank activate command is issued.

If the page miss and bank miss event happen while there is at least one invalid/vacant slot (e.g. when the system first being started up), the EBU of TC11IB does not need to issue a precharge operation but rather continues with activating the new bank and storing the new pair of bank-page tags.

For a data request, the activity will be similar but only affected pairs of data bank-pages. **Table 14-27** lists the activities on a cycle-by-cycle basis.

Table 14-27 Cycle By Cycle Activities Of Multibanking Operation

Cycle n	Cycle n+1	Cycle n+2	Cycle n+3
Comparing the LMB address with the stored bank-page tags, if any.	page_hit :		
	Continue with *read* or *write* command.	not relevant	not relevant
	page_miss and bank_hit:		
	Precharge particular bank and change the page tag with the new one.	Insert *idle* cycles (repeatable) to satisfy $t_{RP.}$	Continue with *bank activate* command, etc.
	page_miss and bank_miss:		
	if no vacant slots are available: Randomly pick one pair of instruction or data bank tags (depends on the originator), close/ *precharge* the selected bank. Store new bank-page tags.	Insert *idle* cycles (repeatable) to satisfy $t_{RP.}$	Continue with *bank activate* command, etc.
	if there is a vacant slot available : Store new bank-page tag and validate the slot. Continue with *bank activate* command, etc.	not relevant	not relevant

14.11.8 Banks Precharge

The system is required to precharge a bank under one of these conditions:

1. A bank needs to be selectively precharged when the next access goes within the same bank, but to a different page/row.
2. When an LMB request cannot be completed before the row active time $t_{RAS\ max}$ is due, then the bank must explicitly be closed and opened again for the current request. Since $t_{RAS\ max}$ is usually much greater (in order of 100 µs) compared to the refresh period (distributed refresh is in order of 15 µs for 4096 rows), generally this is fulfilled by obediently carrying out a refresh to the SDRAMs.

3. Accompanying a page miss is also naturally a selective bank precharge operation, as mentioned previously.
4. All banks must also be precharged, when a refresh cycle is due, as explained next. See **Chapter 14.11.9**.
5. All banks must also be precharged, prior to issuing Self Refresh Entry command. See **Chapter 14.11.10**.

Activities in (1) and (3) are carried out as a result of the address comparison explained in **Chapter 14.11.7**. Activity (2) and (4) are covered by the refresh timer.

14.11.9 Refresh Cycles

It is required that the devices must be refreshed within certain time limit. Prior to being refreshed, the devices must be precharged as mentioned above.

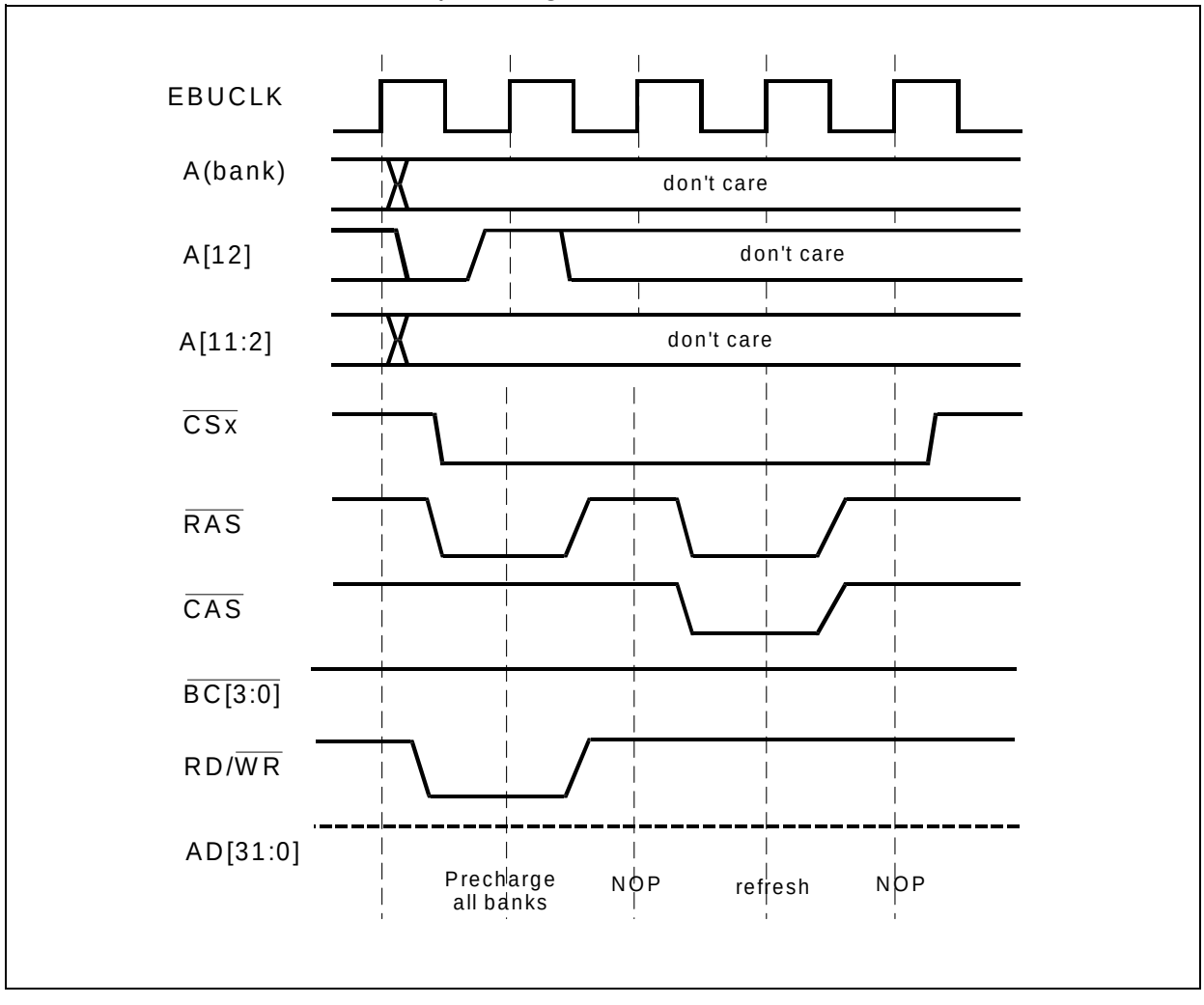


Figure 14-32 SDRAM Refresh

This sequence is periodically triggered by an internal refresh counter with programmable rate (**REFRESHC** in SDRMREF[1:0] registers). All SDRAM banks will be precharged

before the refresh sequence can be started and each SDRAM type is refreshed separately. The specific refresh command being issued is the Auto Refresh (CBR) command, in which the device keeps track of the row addresses to be refreshed. The number of this command being issued for each refresh operation is programmable through **REFRESHR** in SDRMREF[1:0].

A refresh request has precedence over a LMB access to SDRAM, i.e. if both occur at the same time the refresh sequence is entered and the LMB access is rejected with “RETRY”. A refresh error occurs when a previous refresh request has not been satisfied yet and another refresh request occurs and an error flag (**REFERR**) in the SDRMSTAT[1:0] status register will be set accordingly. This error flag can be cleared by writing to SDRMCON[1:0] respective to the appropriate address region.

14.11.10 Power Down Support

Before entering the power-down mode, software must write 1 to bit **SELFEN** in SDRMREF0 register. The EBU of TC11IB will then:

- Precharge all the banks, and
- Issue a self refresh command (see **Table 14-25**) to all SDRAM devices (regardless whether the device belongs to access type 0 or type 1).

On completion of this command, all SDRAM devices would ignore all inputs except the CKE signal. The read-only bit **SELFRENST** reflects the status of issuing this command. When the command is completed, power-down can be safely entered. The devices would perform low-current self refresh during the power down mode. When exiting from power-down and before doing any accesses to SDRAM, software must write 1 to bit **SELFREX** in the SDRMREF0 registers. The EBU of TC11IB will then assert the CKE signal for all the SDRAM devices to exit the self-refresh mode. The read-only bit **SELFREXST** reflects the completion of this command, upon which an access to SDRAMs can be performed.

14.11.11 SDRAM Addressing Scheme

The EBU of TC11IB requires the SDRAMs to be configured to read/write bursts of length 1 or 8. A burst shorter than 8 (e.g. a single access) can be generated by stopping the burst with another command or simply masking it with DQM. Due to the wrap around feature of the SDRAMs, a burst must start at certain addresses to prevent the wrap around (a burst must not cross an address modulo 8*4). This guarantees also that the internal page boundaries of the SDRAMs will not be crossed by any burst access. Bursts

are 16- or 32-bit wide transfers, therefore LMB address A[0] or A[1:0] respective of any burst address must be “0” or "00"!

Table 14-28 16-bit Burst Address Restrictions, A[0] = "0"

Burst Length	LMB Address A[3:1]	SDRAM Burst Address Generation
1	any	single access
8	“000” (0)	0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

Table 14-29 32-bit Burst Address Restrictions, A[1:0] = "00"

Burst Length	LMB Address A[4:2]	SDRAM Burst Address Generation
1	any	single access
8	“000” (0)	0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

The following SDRAM types can be connected to the EBU of TC11IB:

Table 14-30 Supported SDRAM Configurations for 32-bit Wide Data Bus

SDRAM PORTW = 10 (32-bit)			EBU of TC11IB Pins						multiplexed LMB Address	setting for AWIDTH
			A[16]	A[15]	A[14]	A[13]	A[12]	A[11:2]		
256M Bit	Pins		BA[1]	BA[0]	A[12]	A[11]	A[10]	A[9:0]		
	16Mx16	row	RA[14] / BA[1]	RA[13] / BA[0]	RA[12]	RA[11]	RA[10]	RA[9:0]	A[25:11]	10
		col					CMD	CA[8:0]	A[25:24], A[10:2]	
128M Bit	Pins			BA[1]	BA[0]	A[11]	A[10]	A[9:0]		
	8Mx16	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[24:11]	10
		col					CMD	CA[8:0]	A[24:23], A[10:2]	
64MBit	Pins			BA[1]	BA[0]	A[11]	A[10]	A[9:0]		
	16Mx4	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[25:12]	11
		col					CMD	CA[9:0]	A[25:24], A[11:2]	
	8Mx8	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[24:11]	10
		col					CMD	CA[8:0]	A[24:23], A[10:2]	
	4Mx16	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[23:10]	01
		col					CMD	CA[7:0]	A[23:22], A[9:2]	

Table 14-30 Supported SDRAM Configurations for 32-bit Wide Data Bus

SDRAM PORTW = 10 (32-bit)			EBU of TC11IB Pins						multiplexed LMB Address	setting for AWIDTH
			A[16]	A[15]	A[14]	A[13]	A[12]	A[11:2]		
16MBit	Pins					BS	A[10]	A[9:0]		
	4Mx4	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[23:12]	11
		col					CMD	CA[9:0]	A[23], A[11:2]	
	2Mx8	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[22:11]	10
		col					CMD	CA[8:0]	A[22], A[10:2]	
	1Mx16	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[21:10]	01
		col					CMD	CA[7:0]	A[21], A[9:2]	
64MBit	Pins				BA	A[11]	A[10]	A[9:0]		
	2Mx32	row			RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[22:10]	01
		col					CMD	CA[7:0]	A[22], A[9:2]	

RA: row address
BA: bank select (MSB of row address)
CA: column address
CMD: autoprecharge command is currently not supported
Area in shades are not recommended when having PC100 SDRAM configurations, in order to minimize loads on the pads.

Table 14-31 Supported SDRAM Configurations for 16-bit Wide Data Bus

SDRAM PORTW = 01 (16-bit)			EBU of TC11IB Pins						Multiplexed LMB Address	setting for AWIDTH
			A[15]	A[14]	A[13]	A[12]	A[11]	A[10:1]		
256M Bit	Pins		BA[1]	BA[0]	A[12]	A[11]	A[10]	A[9:0]		
	16Mx16	row	RA[14] / BA[1]	RA[13] / BA[0]	RA[12]	RA[11]	RA[10]	RA[9:0]	A[24:10]	10
		col					CMD	CA[8:0]	A[24:23], A[9:1]	
128M Bit	Pins			BA[1]	BA[0]	A[11]	A[10]	A[9:0]		
	8Mx16	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[23:10]	10
		col					CMD	CA[8:0]	A[23:22], A[9:1]	
64MBit	Pins			BA[1]	BA[0]	A[11]	A[10]	A[9:0]		
	16Mx4	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[24:11]	11
		col					CMD	CA[9:0]	A[24:23], A[10:1]	
	8Mx8	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[23:10]	10
		col					CMD	CA[8:0]	A[23:22], A[9:1]	
	4Mx16	row		RA[13] / BA[1]	RA[12] / BA[0]	RA[11]	RA[10]	RA[9:0]	A[22:9]	01
		col					CMD	CA[7:0]	A[22:21], A[8:1]	
16MBit	Pins					BS	A[10]	A[9:0]		
	4Mx4	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[22:11]	11
		col					CMD	CA[9:0]	A[22], A[10:1]	
	2Mx8	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[21:10]	10
		col					CMD	CA[8:0]	A[21], A[9:1]	
	1Mx16	row				RA[11] / BA[0]	RA[10]	RA[9:0]	A[20:9]	01
		col					CMD	CA[7:0]	A[20], A[8:1]	

RA: row address
BA: bank select (MSB of row address)
CA: column address
CMD: autoprecharge command is currently not supported
Area in shades are not recommended when having PC100 SDRAM configurations, in order to minimize loads on the pads.

Each SDRAM bank must be 16 or 32-bit wide (programmable through **PORTW** in BUSCONx or EMUBC registers). The byte selection, e.g. for performing a byte write, is handled via the $\overline{BC}$(3:0) signals. Since there will be two address regions supported for SDRAMs, up to two types of SDRAM can be used. Each region can only have single configuration type. First region is selected if parameter **AGEN** (in BUSCONx or EMUBC registers) is equal to ’011’ and the associated set of registers for parameter setting is SDRMREF0, SDRMCON0, SDRMOD0 and SDRMSTAT0. The second region is associated with SDRMREF1, SDRMCON1, SDRMOD1 and SDRMSTAT1 and selected when **AGEN** is equal to ’100’.

The following addressing scheme supports all configurations mentioned above:

Table 14-32 SDRAM Address Multiplexing Scheme

Port Width	Address Type	Pin Usage	Mode
32-bit (PORTW = 10)	column address	EBU of TC11IB Pins A[11:2]:= LMB A[11:2] EBU of TC11IB Pins A[12]:= CMD	all modes
		EBU of TC11IB Pin A[16:13]:= LMB A[26:23]	AWIDTH = “11”
		EBU of TC11IB Pin A[16:13]:= LMB A[25:22]	AWIDTH = “10”
		EBU of TC11IB Pin A[16:13]:= LMB A[24:21]	AWIDTH = “01”
	row address	EBU of TC11IB Pins A[16:2]:= LMB A[26:12]	AWIDTH = “11”
		EBU of TC11IB Pins A[16:2]:= LMB A[25:11]	AWIDTH = “10”
		EBU of TC11IB Pins A[16:2]:= LMB A[24:10]	AWIDTH = “01”
16-bit (PORTW = 01)	column address	EBU of TC11IB Pins A[10:1]:= LMB A[10:1] EBU of TC11IB Pins A[11]:= CMD	all modes
		EBU of TC11IB Pin A[15:12]:= LMB A[25:22]	AWIDTH = “11”
		EBU of TC11IB Pin A[15:12]:= LMB A[24:21]	AWIDTH = “10”
		EBU of TC11IB Pin A[15:12]:= LMB A[23:20]	AWIDTH = “01”
	row address	EBU of TC11IB Pins A[15:1]:= LMB A[25:11]	AWIDTH = “11”
		EBU of TC11IB Pins A[15:1]:= LMB A[24:10]	AWIDTH = “10”
		EBU of TC11IB Pins A[15:1]:= LMB A[23:9]	AWIDTH = “01”

14.12 EBU Registers

This section describes the control registers and programmable parameters of the EBU. **Figure 14-33** shows all LMB Bus accessible registers associated with the EBU.

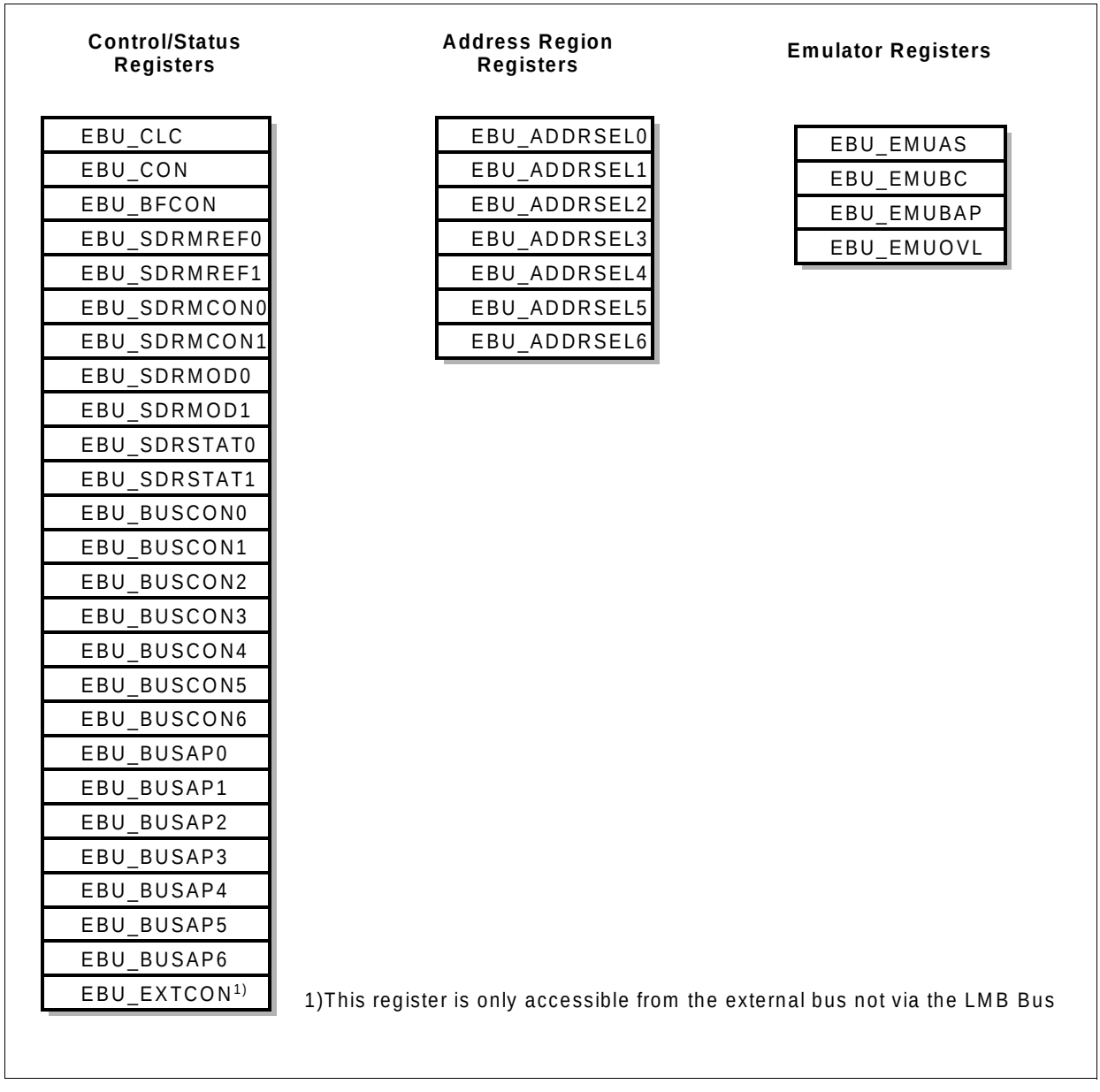


Figure 14-33 EBU Registers

Table 14-33 EBU Registers

Register Short Name	Register Long Name	Offset Address	Description see
EBU_CLC	EBU Clock Control Register	0000_H	Page 14-86
EBU_CON	EBU External Bus Configuration Register	0010_H	Page 14-100
EBU_BFCON	EBU Burst Flash Access Control Register	0020_H	Page 14-102
EBU_SDRMREF0	EBU SDRAM Type 0 Refresh Control Register	0040_H	Page 14-103
EBU_SDRMREF1	EBU SDRAM Type 1 Refresh Control Register	0048_H	
EBU_SDRMCON0	EBU SDRAM Type 0 Configuration Register	0050_H	Page 14-105
EBU_SDRMCON1	EBU SDRAM Type 1 Configuration Register	0058_H	
EBU_SDRMOD0	EBU SDRAM Type 0 Mode Register	0060_H	Page 14-107
EBU_SDRMOD1	EBU SDRAM Type 1 Mode Register	0068_H	
EBU_SDRSTAT0	EBU SDRAM Type 0 Status Register	0070_H	Page 14-108
EBU_SDRSTAT1	EBU SDRAM Type 1Status Register	0078_H	
EBU_ADDRSEL0	EBU Memory Region 0 Base Address Select Register	0080_H	Page 14-87
EBU_ADDRSEL1	EBU Memory Region 1 Base Address Select Register	0088_H	
EBU_ADDRSEL2	EBU Memory Region 2 Base Address Select Register	0090_H	
EBU_ADDRSEL3	EBU Memory Region 3 Base Address Select Register	0098_H	
EBU_ADDRSEL4	EBU Memory Region 4 Base Address Select Register	$00A0_H$	
EBU_ADDRSEL5	EBU Memory Region 5 Base Address Select Register	$00A8_H$	
EBU_ADDRSEL6	EBU Memory Region 6 Base Address Select Register	$00B0_H$	

Table 14-33 EBU Registers (cont'd)

Register Short Name	Register Long Name	Offset Address	Description see
EBU_BUSCON0	EBU Memory Region 0 External Bus Configuration Register	$00C0_H$	Page 14-88
EBU_BUSCON1	EBU Memory Region 1External Bus Configuration Register	$00C8_H$	
EBU_BUSCON2	EBU Memory Region 2 External Bus Configuration Register	$00D0_H$	
EBU_BUSCON3	EBU Memory Region 3 External Bus Configuration Register	$00D8_H$	
EBU_BUSCON4	EBU Memory Region 4 External Bus Configuration Register	$00E0_H$	
EBU_BUSCON5	EBU Memory Region 5 External Bus Configuration Register	$00E8_H$	
EBU_BUSCON6	EBU Memory Region 6 External Bus Configuration Register	$00F0_H$	
EBU_BUSAP0	EBU Memory Region 0 External Bus Access Parameter Register	0100_H	Page 14-91
EBU_BUSAP1	EBU Memory Region 1External Bus Access Parameter Register	0108_H	
EBU_BUSAP2	EBU Memory Region 2 External Bus Access Parameter Register	0110_H	
EBU_BUSAP3	EBU Memory Region 3 External Bus Access Parameter Register	0118_H	
EBU_BUSAP4	EBU Memory Region 4 External Bus Access Parameter Register	0120_H	
EBU_BUSAP5	EBU Memory Region 5 External Bus Access Parameter Register	0128_H	
EBU_BUSAP6	EBU Memory Region 6 External Bus Access Parameter Register	0130_H	
EBU_EMUAS	EBU Emulator Region Base Address Select Register	0160_H	Page 14-94
EBU_EMUBC	EBU Emulator Region External Bus Configuration Register	0168_H	Page 14-95
EBU_EMUBAP	EBU Emulator Region External Bus Access Parameter Register	0170_H	Page 14-98

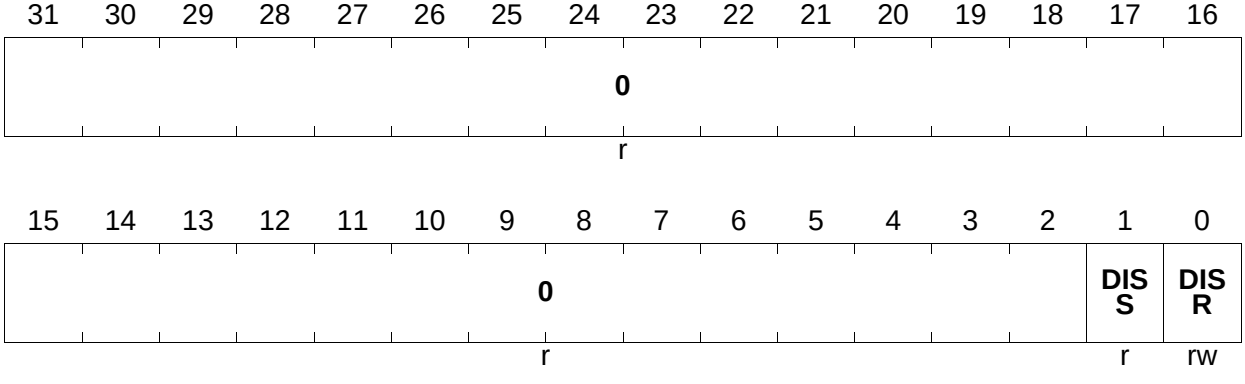
Table 14-33 EBU Registers (cont'd)

Register Short Name	Register Long Name	Offset Address	Description see
EBU_EMUOVL	EBU Overlay Memory Chip Select Generation Register	0178_H	Page 14-100
EBU_EXTCON	EBU External Access Configuration Register	Only from external 400000_H	Page 14-109

14.12.1 Clock Control Register

The EBU clock control register CLC allows the EBU to be generally enabled/disabled. After reset the EBU is enabled.

EBU_CLC
EBU Clock Control Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
DISR	0	rw	**EBU Disable Request Bit** Used for enable/disable control of the EBU. 0 EBU disable is not requested 1 EBU disable is requested
DISS	1	r	**EBU Disable Status Bit** Bit indicates the current status of the EBU. 0 EBU is enabled (default after reset) 1 EBU is disabled
0	[31:2]	r	**Reserved**; read as 0; should be written with 0.

14.12.2 Address Select Registers

The EBU Address Select Registers EBU_ADDSELx (x=0-6) establish and control memory regions for external accesses.

EBU_ADDRSELx (x=0-6)
EBU Address Select Register x **ADDRSEL[6:1] Reset Value: 0000 0000$_H$**
ADDRSEL0 Reset Value (internal boot): 0000 0000$_H$
ADDRSEL0 Reset Value (external boot): A000 0001$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BASE															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BASE				ALTSEG				MASK				0		ALT EN	REG EN
rw				rw				rw				r		rw	rw

Field	Bits	Type	Description
REGEN	0	rw	**Memory Region Enable** 0: memory region is disabled (default after reset except for ADDRSEL0) 1: memory region is enabled *Note: In the case of ADDRSEL0, when the EBU is in external boot mode, the default value of this bit after reset is 1.*
ALTEN	1	rw	**Alternate Segment Comparison Enable** 0: altseg is never compared with LMB address (default after reset) 1: altseg is always compared with LMB address
MASK	[7:4]	rw	**Memory Region Address Mask** Specifies the number of rightmost bits in the base address starting at bit 26, which should be included in the address comparison. Bits (31:27) will always be part of the comparison.
ALTSEG	[11:8]	rw	**Memory Region Alternate Segment** Alternate segment to be compared with LMB address bit (31:28).

Field	Bits	Type	Description
BASE	[31:12]	rw	**Memory Region Base Address** Base address to be compared with LMB address in conjunction with the mask control.
0	[3:2]	r	**Reserved bits** These bits are reserved; read as ‘0’, must be written with ‘0’.

14.12.3 Bus Configuration Registers

The EBU Bus Configuration Registers EBU_BUSCONx (x = 0-6) and Bus Access Parameter Registers EBU_BUSAPx configure access modes and access timing to the external memory regions defined through the EBU_ADDRSELx registers.

EBU_BUSCONx (x=0-6)
EBU Bus Configuration Register x **Reset Value (internal boot): 8092 8000$_H$**
Reset Value (external boot): 8092 807F$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WR	**AGEN**			**XCMDDELAY**		**WAIT**		**PORTW**		**BCGEN**		**WAITINV**	**PRE**	**DLOAD**	**ENDIAN**
rw	rw			rw		rw		rw		rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMULT			**0**			**AALIGN**	**WPRE**	**0**	**MULTMAP**						
rw			r			rw	rw	r	rw						

Field	Bits	Type	Description
MULTMAP	[6:0]	rw	**Multiplier map** A mask value specifies which of the programmable delay cycles are set to use the multiplier defined in cmult. XXXXXX0: addrc does not use the multiplier XXXXXX1: addrc uses the multiplier XXXXX0X: aholdc does not use the multiplier XXXXX1X: aholdc uses the multiplier XXXX0XX: cmddelay does not use the multiplier XXXX1XX: cmddelay uses the multiplier XXX0XXX: burstc does not use the multiplier XXX1XXX: burstc uses the multiplier XX0XXXX: datac does not use the multiplier XX1XXXX: datac uses the multiplier X0XXXXX: rdrecovc does not use the multiplier X1XXXXX: rdrecovc uses the multiplier 0XXXXXX: wrrecovc does not use the multiplier 1XXXXXX: wrrecovc uses the multiplier *Note: “waitrdc”,”waitwrc”.”dtardwr” and “dtacs” are not programmable and always use the “cmult” multiplier*
WPRE	8	rw	**Weak Prefetch** 0 Running prefetch access cannot be aborted by an interrupting data access. 1: Running prefetch access can be aborted by an interrupting data access.
AALIGN	9	rw	**Address Alignment** 0 EBU always issues a byte address when performing an external bus access via this Chip Select. 1: EBU aligns the address according to the setting of the PORTW field.

Field	Bits	Type	Description
CMULT	[15:13]	rw	**Cycle multiplier control** Specifies a multiplier for the cycles specified via **multmap** (see below). **waitrdc**, **waitwrc**, **dtardwr** and **dtacs** are always using the multiplier. 000 : multiplier is 1 001 : multiplier is 4 010 : multiplier is 8 011 : multiplier is 16 100 : multiplier is 32 (default after reset) 101, 110, 111 : reserved
ENDIAN	16	rw	**Endian mode** 0 little endian access mode (default after reset) 1 big endian access mode
DLOAD	17	rw	**Enforce data upload from external bus** 0 Data access is fed from data write buffer if it's available 1 Data access is always fed from the external bus access
PRE	18	rw	**Prefetch mechanism for each code access** 0 Code access never uses prefetch buffer mechanism 1 Code access always uses prefetch buffer mechanism
WAITINV	19	rw	**Reversed Polarity at $\overline{\text{WAIT}}$** 0 OFF, input at $\overline{\text{WAIT}}$ pin is active low (default after reset) 1 polarity reversed, input at $\overline{\text{WAIT}}$ pin is active high
BCGEN	[21:20]	rw	**Byte Control Signal Control** To select the timing mode of the byte control signals. 00 to follow Chip Select 01 to follow control signal ($\overline{\text{RD}}$, RD/$\overline{\text{WR}}$) (default after reset) 10 to follow write enable signal (RD/$\overline{\text{WR}}$ only) 11 to be used as DQM for SDRAM access
PORTW	[23:22]	rw	**Port width** 00 reserved 01 16-bit 10 32-bit (default after reset) 11 reserved

Field	Bits	Type	Description
WAIT	[25:24]	rw	**External Wait State Control** 00 OFF (default after reset) 01 Asynchronous input at $\overline{\text{WAIT}}$ 10 Synchronous input at $\overline{\text{WAIT}}$ 11 reserved
XCMDDELAY	[27:26]	rw	**External Command Delay Control** 00 OFF (default after reset) 01 Asynchronous input at $\overline{\text{CMDELAY}}$ 10 Synchronous input at $\overline{\text{CMDELAY}}$ 11 reserved
AGEN	[30:28]	rw	**Address Generation Control** 000 Demultiplexed access (default after reset) 001 Multiplexed access 010 Burst Flash access 011 SDRAM access type 0 100 SDRAM access type 1 other values : reserved
WR	31	rw	**Memory Region Write Protection** 0: writes to the memory region are enabled 1: writes to the memory region are disabled (default after reset)
0	7, [12:10]	r	**Reserved bits** Read as ‘0’, must be written with ‘0’.

EBU_BUSAPx (x=0-6)

EBU Bus Access Parameter Register x **Reset Value: FFFF FFFF$_H$**

31 30	29 28	27 26 25	24 23 22	21 20 19	18 17 16
ADDRC	**AHOLDC**	**CMDDELAY**	**WAITRDC**	**WAITWRC**	**BURSTC**
rw	rw	rw	rw	rw	rw

15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
DATAC	**RDRECOVC**	**WRRECOVC**	**DTARDWR**	**DTACS**
rw	rw	rw	rw	rw

Field	Bits	Type	Description
DTACS	[3:0]	rw	**Recovery cycles between different regions** Minimum number of cycles following an access, if the next access is to a different region. This number is multiplied by **cmult** in BUSCONx. 0 - 15 :number of idle cycles
DTARDWR	[7:4]	rw	**Recovery cycles between read and write accesses** Minimum number of cycles between a read and write access, and vice versa. This number is multiplied by **cmult** in BUSCONx. 0 - 15 :number of idle cycles
WRRECOVC	[10:8]	rw	**Recovery cycles after write accesses** Number of idle cycles after write accesses. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[6]** in BUSCONx is set to 1. 0 -7 : number of idle cycles
RDRECOVC	[13:11]	rw	**Recovery cycles after read accesses** Number of idle cycles after read accesses. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[5]** in BUSCONx is set to 1. 0 -7 : number of idle cycles
DATAC	[15:14]	rw	**Data hold cycles for write accesses** Number of data hold cycles during write accesses. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[4]** in BUSCONx is set to 1. 0 - 3 : number of hold cycles
BURSTC	[18:16]	rw	**Data cycles during burst accesses** Number of data cycles during burst accesses. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[3]** in BUSCONx is set to 1. 0 - 7 : number of data cycles
WAITWRC	[21:19]	rw	**Programmed Wait States for wait accesses** Number of programmed wait states for write accesses.This number is always multiplied by **cmult** in BUSCONx. 0 - 7 : number of wait states

Field	Bits	Type	Description
WAITRDC	[24:22]	rw	**Programmed Wait States for Read Accesses** Number of programmed wait states for read accesses. This number is always multiplied by **cmult** in BUSCONx. 0 - 7 : number of wait states
CMDDELAY	[27:25]	rw	**Programmed Command Delay Cycles** Number of delay cycles during command delay phase. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[2]** in BUSCONx is set to 1. Writing value 0 to this field is ignored and minimum value of 1 will be set. 1 - 7: number of delay cycles
AHOLDC	[29:28]	rw	**Address Hold Cycles for Multiplexed Accesses** Number of address hold cycles during multiplexed accesses. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[1]** in BUSCONx is set to 1. 0 - 3 : number of hold cycles
ADDRC	[31:30]	rw	**Address Cycles** Number of cycles for address phase. This number is multiplied by **cmult** in BUSCONx, if bit **multmap[0]** in BUSCONx is set to 1. Writing value 0 to this field is ignored and minimum value of 1 will be set. 1 - 3: number of cycles

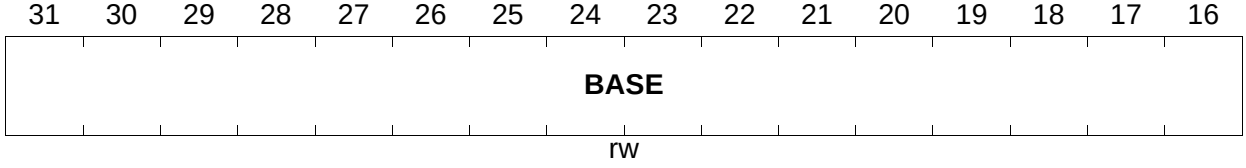
14.12.4 Emulator Configuration Registers

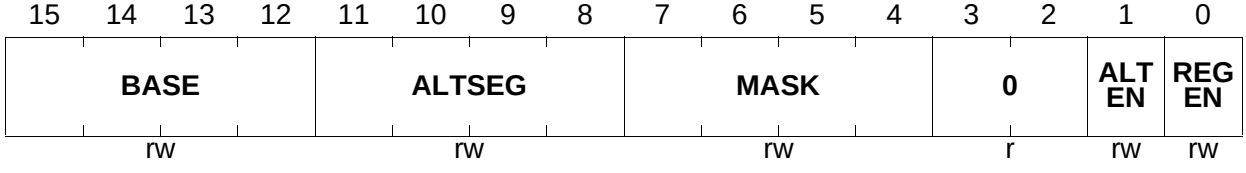
The EBU Emulator Address Select Register EBU_EMUAS defines the address region for the emulator memory. This register has the same layout and semantics as the EBU_ADDRSELx registers.The EBU Emulator Bus Configuration Register EBU_EMUBC and the EBU Bus Access Parameter Register EBU_EMUBAP define the access parameters for the emulator memory region determined through register EBU_EMUAS. These two registers have the same layout and semantics as the

EBU_BUSCONx and EBU_BUSAPx. The EBU Emulator Overlay Register EBU_EMUOVL provides overlay memory control to the emulator.

EBU_EMUAS

EBU Emulator Address Select Register **Reset Value: DE00 0031$_H$**





Field	Bits	Type	Description
REGEN	0	rw	**Memory Region Enable** 0 memory region is disabled 1 memory region is enabled (default after reset)
ALTEN	1	rw	**Alternate Segment Comparison Enable** 0 altseg is never compared with LMB address (default after reset) 1 altseg is always compared with LMB address
MASK	[7:4]	rw	**Memory Region Address Mask** Specifies the number of rightmost bits in the base address starting at bit 26, which should be included in the address comparison. Bits (31:27) will always be part of the comparison.
ALTSEG	[11:8]	rw	**Memory Region Alternate Segment** Alternate segment to be compared with LMB address bit (31:28).
BASE	[31:12]	rw	**Memory Region Base Address** Base address to be compared with LMB address in conjunction with the mask control.
0	[3:2]	rw	**Reserved bits** These bits are reserved and read as ‘0’, must be written with ‘0’.

EBU_EMUBC
EBU Emulator Bus Configuration Register

Reset Value: 0190 2077$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WR	AGEN			XCMDDELAY		WAIT		PORTW		BCGEN		WAITINV	PRE	DLOAD	ENDIAN
rw	rw			rw		rw		rw		rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMULT			0			AALIGN	WPRE	0	MULTMAP						
rw			r			rw	rw	r	rw						

Field	Bits	Type	Description
MULTMAP	[6:0]	rw	**Multiplier map** A mask value specifies which of the programmable delay cycles are set to use the multiplier defined in cmult. XXXXXX0: addrc does not use the multiplier XXXXXX1: addrc uses the multiplier XXXXX0X: aholdc does not use the multiplier XXXXX1X: aholdc uses the multiplier XXXX0XX: cmddelay does not use the multiplier XXXX1XX: cmddelay uses the multiplier XXX0XXX: burstc does not use the multiplier XXX1XXX: burstc uses the multiplier XX0XXXX: datac does not use the multiplier XX1XXXX: datac uses the multiplier X0XXXXX: rdrecovc does not use the multiplier X1XXXXX: rdrecovc uses the multiplier 0XXXXXX: wrrecovc does not use the multiplier 1XXXXXX: wrrecovc uses the multiplier *Note: “waitrdc”,”waitwrc”.”dtardwr” and “dtacs” are not programmable and always use the “cmult” multiplier*
WPRE	8	rw	**Weak Prefetch** 0 Code prefetch cannot be aborted by an interrupting data access. 1: Code prefetch can be aborted by an interrupting data access.

Field	Bits	Type	Description
AALIGN	9	rw	**Address Alignment** 0 EBU always issues a byte address when performing an external bus access via this Chip Select. 1: EBU aligns the address according to the setting of the PORTW field.
CMULT	[15:13]	rw	**Cycle multiplier control** Specifies a multiplier for the cycles specified via **multmap** (see below). **waitrdc**, **waitwrc**, **dtardwr** and **dtacs** are always using the multiplier. 000 : multiplier is 1 001 : multiplier is 4 010 : multiplier is 8 011 : multiplier is 16 100 : multiplier is 32 101, 110,111 : reserved.
ENDIAN	16	rw	**Endian mode** 0 little endian access mode 1 big endian access mode
DLOAD	17	rw	**Enforce data upload from external bus** 0 Data access is fed from data write buffer if it's available 1 Data access is always fed from the external bus access
PRE	18	rw	**Prefetch mechanism for each code access** 0 Code access never uses prefetch buffer mechanism 1 Code access always uses prefetch buffer mechanism
WAITINV	19	rw	**Reversed polarity at $\overline{\text{WAIT}}$** 0 OFF, input at $\overline{\text{WAIT}}$ pin is active low 1 polarity reversed, input at $\overline{\text{WAIT}}$ pin is active high
BCGEN	[21:20]	rw	**Byte Control Signal Control** To select the timing mode of the byte control signals. 00 to follow Chip Select 01 to follow control signal ($\overline{\text{RD}}$, RD/$\overline{\text{WR}}$) 10 to follow write enable signal (RD/$\overline{\text{WR}}$ only) 11 to be used as DQM for SDRAM access

Field	Bits	Type	Description
PORTW	[23:22]	rw	**Port Width** 00 reserved 01 16-bit 10 32-bit 11 reserved
WAIT	[25:24]	rw	**External Wait State Control** 00 OFF 01 Asynchronous input at $\overline{\text{WAIT}}$ 10 Synchronous input at $\overline{\text{WAIT}}$ 11 reserved
XCMDDELAY	[27:26]	rw	**External Command Delay Control** 00 OFF 01 Asynchronous input at $\overline{\text{CMDELAY}}$ 10 Synchronous input at $\overline{\text{CMDELAY}}$ 11 reserved
AGEN	[30:28]	rw	**Address Generation Control** 000 Demultiplexed access 001 Multiplexed access 010 Burst Flash access 011 SDRAM access type 0 100 SDRAM access type 1 other values : reserved
WR	31	rw	**Memory Region Write Protection** 0 writes to the memory region are enabled 1 writes to the memory region are disabled
0	7, [12:10]	r	**Reserved bits** These bits are reserved and read as ‘0’, must be written with ‘0’.

EBU_EMUBAP

EBU Emulator Bus Access Parameter Register **Reset Value: 5248 4911$_H$**

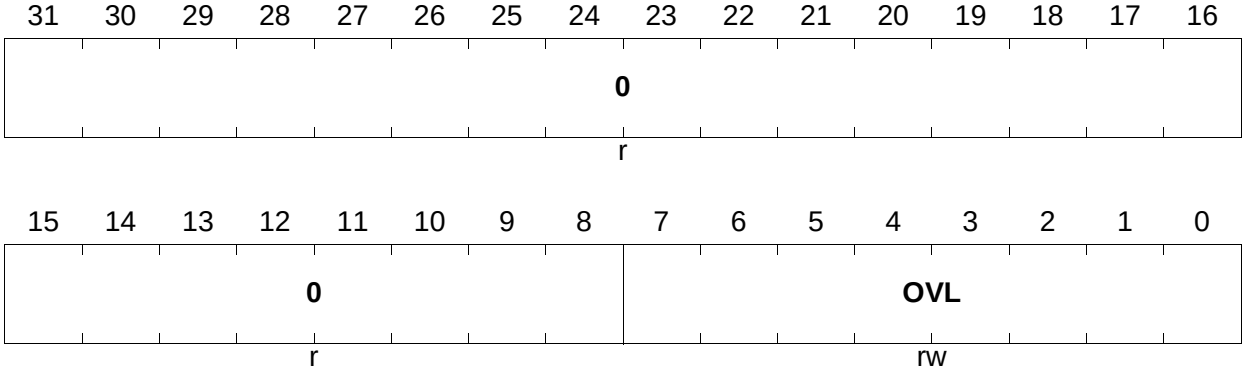
31 30	29 28	27 26 25	24 23 22	21 20 19	18 17 16
ADDRC	AHOLDC	CMDDELAY	WAITRDC	WAITWRC	BURSTC
rw	rw	rw	rw	rw	rw

15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
DATAC	RDRECOVC	WRRECOVC	DTARDWR	DTACS
rw	rw	rw	rw	rw

Field	Bits	Type	Description
DTACS	[3:0]	rw	**Recovery cycles between different regions** Minimum number of cycles following an access, if the next access is to a different region. This number is multiplied by **cmult** in EMUBC. 0 - 15 :number of idle cycles
DTARDWR	[7:4]	rw	**Recovery cycles between read and write accesses** Minimum number of cycles between a read and write access, and vice versa. This number is multiplied by **cmult** in EMUBC. 0 - 15 :number of idle cycles
WRRECOVC	[10:8]	rw	**Recovery cycles after write accesses** Number of idle cycles after write accesses. This number is multiplied by **cmult** in EMUBC, if bit **multmap[6]** in EMUBC is set to 1. 0 -7 : number of idle cycles
RDRECOVC	[13:11]	rw	**Recovery cycles after read accesses** Number of idle cycles after read accesses. This number is multiplied by **cmult** in EMUBC, if bit **multmap[5]** in EMUBC is set to 1. 0 -7 : number of idle cycles
DATAC	[15:14]	rw	**Data hold cycles for write accesses** Number of data hold cycles during write accesses. This number is multiplied by **cmult** in EMUBC, if bit **multmap[4]** in EMUBC is set to 1. 0 - 3 : number of hold cycles

Field	Bits	Type	Description
BURSTC	[18:16]	rw	**Data cycles during burst accesses** Number of data cycles during burst accesses. This number is multiplied by **cmult** in EMUBC, if bit **multmap[3]** in EMUBC is set to 1. 0 - 7 : number of data cycles
WAITWRC	[21:19]	rw	**Programmed Wait States for wait accesses** Number of programmed wait states for write accesses.This number is always multiplied by **cmult** in EMUBC. 0 - 7: number of wait states
WAITRDC	[24:22]	rw	**Programmed Wait States for read accesses** Number of programmed wait states for read accesses. This number is always multiplied by **cmult** in EMUBC. 0 - 7: number of wait states
CMDDELAY	[27:25]	rw	**Programmed command delay cycles** Number of delay cycles during command delay phase. This number is multiplied by **cmult** in EMUBC, if bit **multmap[2]** in EMUBC is set to 1. 1 - 7: number of delay cycles
AHOLDC	[29:28]	rw	**Address hold cycles for multiplexed accesses** Number of address hold cycles during multiplexed accesses. This number is multiplied by **cmult** in EMUBC, if bit **multmap[1]** in EMUBC is set to 1. 0 - 3: number of hold cycles
ADDRC	[31:30]	rw	**Address Cycles** Number of cycles for address phase. This number is multiplied by **cmult** in EMUBC, if bit **multmap[0]** in EMUBC is set to 1. Writing value 0 to this field will be ignored and minimum value of 1 is set. 1 - 3: number of cycles

EBU_EMUOVL
EBU Emulator Overlay Register **Reset Value: 0000 0000$_H$**

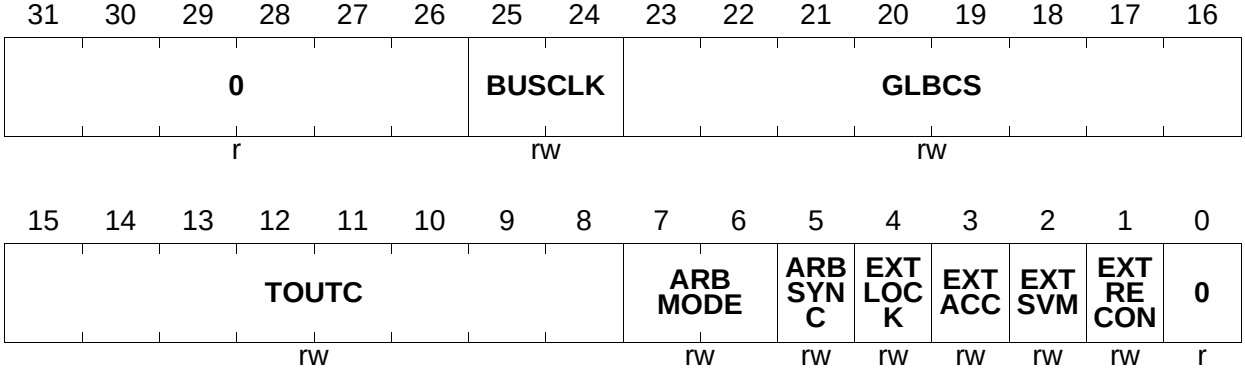


Field	Bits	Type	Description
OVL	[7:0]	rw	**Overlay Chip Select signal** To select one or more of the Chip Select to generate $\overline{\text{CSovl}}$ 0: $\overline{\text{CSovl}}$ is always inactive other: if ovl[n] is set, it means $\overline{\text{CSovl}}$ will always be asserted when $\overline{\text{CS[n]}}$ is asserted
0	[31:8]	r	**Reserved bits** These bits are reserved and read as ‘0’, must be written with ‘0’.

14.12.5 EBU Configuration Register

The EBU Configuration Register EBU_CON provides control of the EBU bus.

EBU_CON
EBU Configuration Register **Reset Value (internal boot): 0000 0028$_H$**
Reset Value (external boot): 0001 0068$_H$

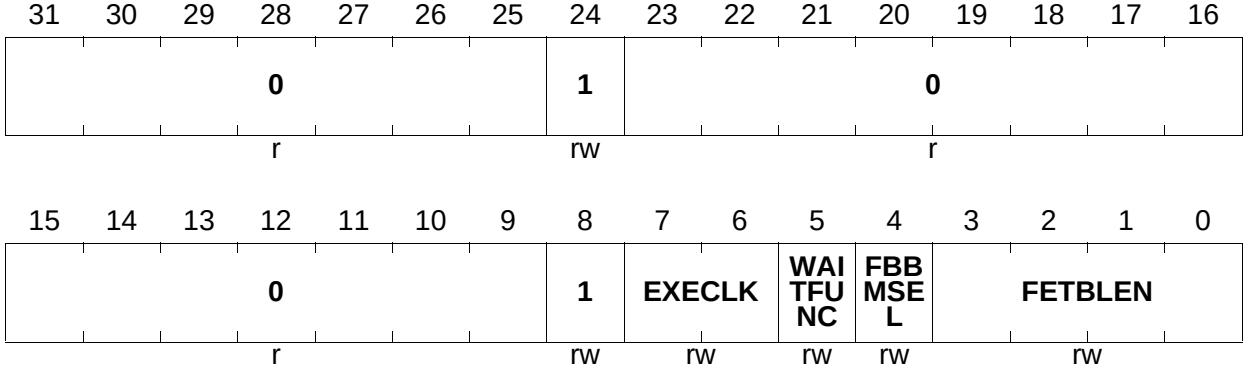


Field	Bits	Type	Description
EXTRECON	1	rw	**Extension Reconfiguration** The ability of external master to change address extension register EXTCON. 0 no 1 yes
EXTSVM	2	rw	**External Access in Supervisor Mode** 0 external device access to FPI bus is performed in user mode 1 external device access to FPI bus is performed in supervisor mode
EXTACC	3	rw	**External Device Access to FPI Bus** 0 external device access to FPI bus is disabled 1 external device access to FPI bus is enabled
EXTLOCK	4	rw	**External Bus Lock Control** 0 external bus is not locked after EBU of TC11IB gain ownership 1 external bus is locked after EBU of TC11IB gain ownership
ARBSYNC	5	rw	**Arbitration Signal Synchronization** 0 arbitration inputs are synchronous 1 arbitration inputs are asynchronous
ARBMODE	[7:6]	rw	**EBU Arbitration Strategy** 0 EBU is disabled 1 EBU is in external master mode 2 EBU is in external slave mode 3 No arbitration, EBU is the only bus master
TOUTC	[15:8]	rw	**Bus Time-out Control** Number of inactive cycles leads to a bus time-out after EBU gain ownership 0 time-out is disabled 1-255 time-out is after timeoutc*8 clock cycles
GLBCS	[23:16]	rw	**Global Chip Select Signal** To select one or more of the Chip Select to generate $\overline{\text{CSglb}}$ 0 $\overline{\text{CSglb}}$ is always inactive other if globalCS[n] is set, it means $\overline{\text{CSglb}}$ will always be asserted when $\overline{\text{CS[n]}}$ is asserted

Field	Bits	Type	Description
BUSCLK	[25:24]	rw	**EBU clock generation** 0 Frequency of EBU of TC11IB clock is equal to LMB clock 1 Frequency of EBU of TC11IB clock is half of LMB clock 2 Frequency of EBU of TC11IB clock is one-fourth of LMB clock 3 reserved
0	0, [31:26]	r	**Reserved bits** These bits are reserved and read as ‘0’, must be written with ‘0’.

14.12.6 Burst Flash Control Register

EBU_BFCON
EBU Burst Flash Control Register **Reset Value: 0100 01D0$_H$**



Field	Bits	Type	Description
FETBLEN	[3:0]	rw	**Fetch Burst Length for Burst Flash** Defines maximum number of burst data cycles which are executed by the EBU during access to Burst Flash. 000: 1 data access (default after reset) 001: 2 data accesses 010: 4 data accesses 011: 8 data accesses 1xx : reserved

Field	Bits	Type	Description
FBBMSEL	4	rw	**Flash Burst Buffer Mode Select for Burst Flash** 0: continuous mode 1: Flash burst buffer length defined by value in **FETBLEN** (default after reset)
WAITFUNC	5	rw	**Function of $\overline{\text{WAIT}}$ Input for Burst Flash** 0: $\overline{\text{WAIT}}$ input operates as a wait data bus function on bursts.(default after reset) 1: The $\overline{\text{WAIT}}$ input operates as a terminate burst function.
EXTCLK	[7:6]	rw	**Frequency of External Clock at Pin ACLK** 00: equal LMBCLK frequency 01: 1/2 of LMBCLK frequency 10: 1/3 of LMBCLK frequency 11: 1/4 of LMBCLK frequency (default after reset)
1	8,24	rw	**These bits should be clear to 0 after reset**
0	[31:25] [24:9]	r	**Reserved Bits.** Read as ‘0’ always.

14.12.7 SDRAM Configuration Registers

EBU_SDRMREFx (x=0,1)
EBU SDRAM Refresh Register x **Reset Value: 0000 0000$_H$**

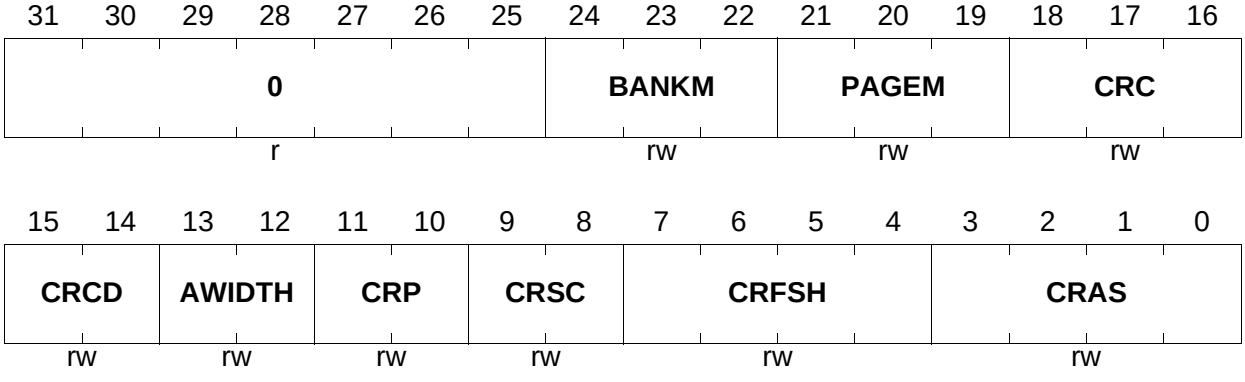
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0			SELFREN	SELFRENST	SELFREX	SELFREXST	REFRESHR			REFRESHC					
r			rw	r	rw	r	rw			rw					

Field	Bits	Type	Description
REFRESHC	[5:0]	rw	**Refresh Counter Period** Number of clock cycles between refresh operations. 0 no refresh needed (default after reset) 1-63 refresh period is refreshc x 64 clock cycles

Field	Bits	Type	Description
REFRESHR	[8:6]	rw	**Number of Refresh Commands** The number of additional refresh commands issued to SDRAM each time a refresh is due. 0 only one refresh command is issued (default after reset) 1-7 additional one to seven refresh commands are issued
SELFREXST	9	r	**Self Refresh Exit Status (only in SDRMREF0 register)** If this bit is set to 1, it means the Self Refresh Entry command has been successfully issued. This bit is reset when bit selfren is set to 1 or a reset takes place.
SELFREX	10	rw	**Self Refresh Exit (only in SDRMREF0 register)** When this bit is written with 1 the Self Refresh Exit command is issued to all SDRAM devices, regardless whether they are attached to type 0 or type 1.
SELFRENST	11	r	**Self Refresh Entry Status (only in SDRMREF0 register)** If this bit is set to 1, it means the **Self Refresh Entry** command has been successfully issued. This bit is reset when bit selfrex is set to 1 or a reset takes place.
SELFREN	12	rw	**Self Refresh Entry (only in SDRMREF0 register)** When this bit is written with 1 the **Self Refresh Entry** command is issued to all SDRAM devices, regardless whether they are attached to type 0 or type 1.
0	[31:13]	r	**Reserved bits** read as ‘0’, must be written with ‘0’.

EBU_SDRMCONx (x = 0,1)
EBU SDRAM Control Register x **Reset Value: 0000 0000$_H$**

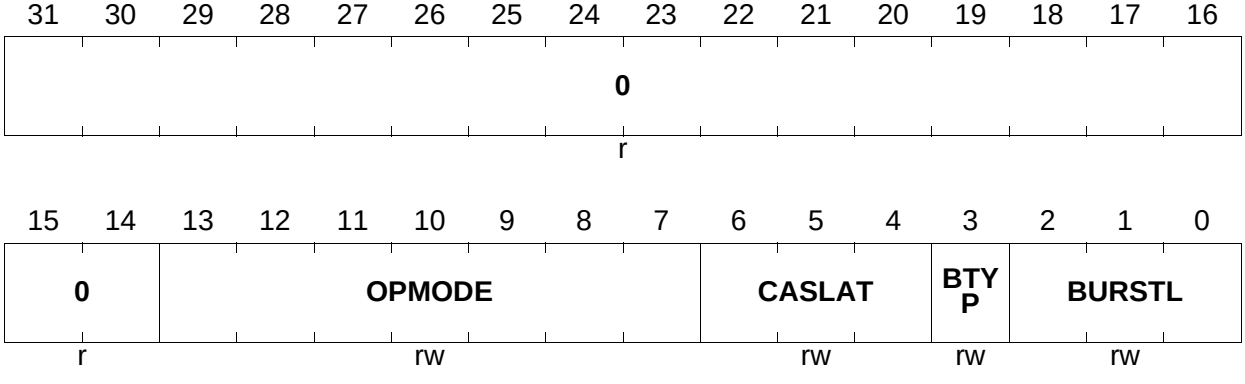


Field	Bits	Type	Description
CRAS	[3:0]	rw	**Row to Precharge Delay Counter** Number of clock cycles between row activate command and a precharge command. 0-15 minimum Cras + 1 clock cycles (default after reset Cras is 0)
CRFSH	[7:4]	rw	**Initialization Refresh Commands Counter** Number of refresh commands issued during power-up initialization sequence. 0-15 perform Crfsh + 1 NOP cycles (default after reset Crfsh is 0)
CRSC	[9:8]	rw	**Mode Register Set-up Time** Number of NOP cycles after a mode register set command. 0-3 insert Crsc + 1 NOP cycles (default after reset Crsc is 0)
CRP	[11:10]	rw	**Row Precharge Time Counter** Number of NOP cycles inserted after a precharge command. The actual number performed can be greater due to $\overline{CAS}$ latency and burst length. 0-3 insert Crsc + 1 NOP cycles (default after reset Crp is 0)

Field	Bits	Type	Description
AWIDTH	[13:12]	rw	**Width of Column Address** Number of address bits from bit 0 to be used for column address depending on the port width. If port width = 32-bit 00 Address(8:0), otherwise: Address(7:0) 01 Address(9:0), otherwise: Address(8:0) 10 Address(10:0), otherwise: Address(9:0) 11 Address(11:0), otherwise: Address(10:0) If port width = 16-bit 00 Address(7:0), otherwise: Address(6:0) 01 Address(8:0), otherwise: Address(7:0) 10 Address(9:0), otherwise: Address(8:0) 11 Address(10:0), otherwise: Address(9:0)
CRCD	[15:14]	rw	**Row to Column Delay Counter** Number of NOP cycles between a row address and a column address. 0-3 insert Crcd + 1 NOP cycles (default after reset Crcd is 0)
CRC	[18:16]	rw	**Row Cycle Time Counter** Number of NOP cycles between refresh commands in the power-up initialization sequence. 0-7 insert Crc + 1 NOP cycles (default after reset Crc is 0)
PAGEM	[21:19]	rw	**Mask for Page Tag** Number of address bits from bit 26 to be used for comparing page tags. 0 always generates page miss (default after reset) 1 address bit 26 to 9 2 address bit 26 to 10 3 address bit 26 to 11 4 address bit 26 to 12 5 address bit 26 to 13 6 reserved 7 reserved

Field	Bits	Type	Description
BANKM	[24:22]	rw	**Mask for Bank Tag** Number of address bits from bit 26 to be used for comparing bank tags. 0 always generates bank miss (default after reset) 1 address bit 26 to 20 2 address bit 26 to 21 3 address bit 26 to 22 4 address bit 26 to 23 5 address bit 26 to 24 6 address bit 26 to 25 7 address bit 26
0	[31:25]	r	**Reserved bits** Read as ‘0’, must be written with ‘0’.

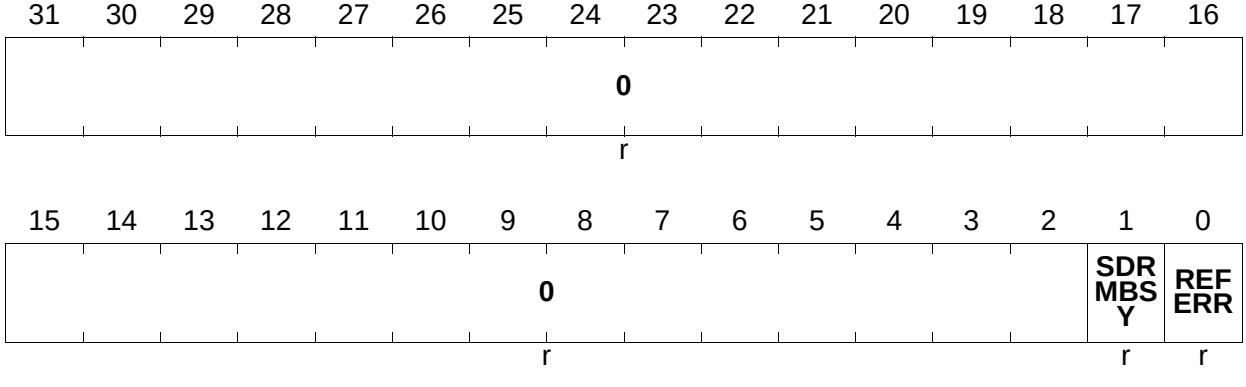
EBU_SDRMODx (x = 0,1)
EBU SDRAM Mode Register x **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
BURSTL	[2:0]	rw	**Burst length** Number of locations can be accessed with a single command. 0 1 (default after reset) 1 reserved 2 reserved 3 8 other values:reserved

Field	Bits	Type	Description
BTYP	3	rw	**Burst type** EBU only supports sequential burst. 0 only this value should be written (default after reset) other valuesreserved
CASLAT	[6:4]	rw	**$\overline{\text{CAS}}$ latency** Number of clocks between a READ command and the availability of data. 2 two clocks (default after reset) 3 three clocks other valuesreserved
OPMODE	[13:7]	rw	**Operation Mode** EBU only supports burst write standard operation. 0 only this value should be written (default after reset) other values:reserved
0	[31:14]	r	**Reserved bits** Read as ‘0’, must be written with ‘0’.

EBU_SDRSTATx (x=0,1)
EBU SDRAM Status Register x **Reset Value: 0000 0000$_H$**

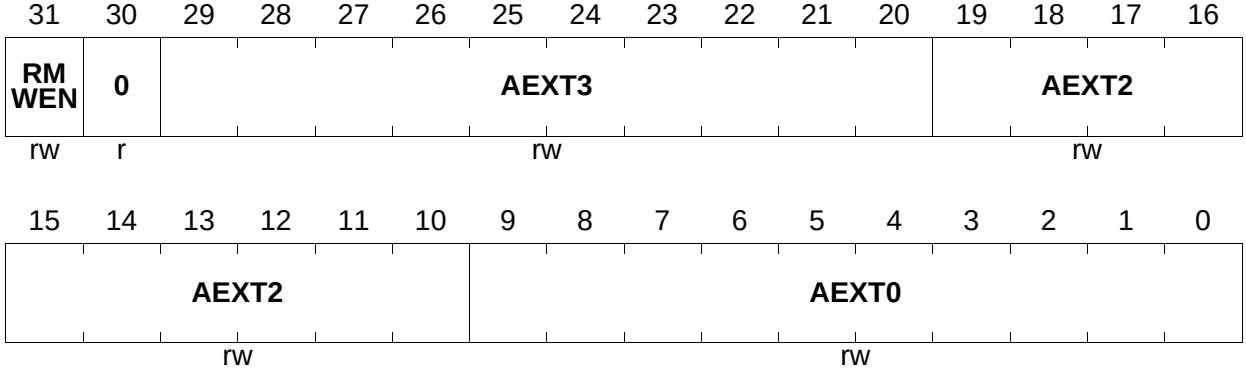


14.12.8 External Access Configuration Register

The External Access Configuration Register EBU_EXTCON is only accessible from the external bus, not via the LMB Bus. In addition, this register can be write-protected by use of the EBU_EBUCON.EXTRECON bit.

Field	Bits	Type	Description
REFERR	0	r	**SDRAM Refresh Error** Unsuccessful previous refresh request collides with a new request. This bit is reset by a write access to SDRMCON[1:0] respectively. 0 no refresh error 1 refresh error occurred
SDRMBSY	1	r	**SDRAM Busy** The status of power-up initialization sequence. 0 power-up initialization sequence is not running 1 power-up initialization sequence is running
0	[31:2]	r	**Reserved bits** Read as ‘0’, must be written with ‘0’.

EBU_EXTCON
EBU External Access Configuration Register **Reset Value: 3C0B FFA0$_H$**



Field	Bits	Type	Description
AEXT0	[9:0]	rw	**Address Extension 0** Address extension bits A[31:22] for external accesses with A[23:22] = ’00’, default: 1110100000_B (E800 0000_H - E83F $FFFF_H$)
AEXT2	[19:10]	rw	**Address Extension 2** Address extension bits A[31:22] for external accesses with A[23:22] = ’10’, default: 1011111111_B (BFC0 0000_H - BFFF $FFFF_H$)

Field	Bits	Type	Description
AEXT3	[29:20]	rw	**Address Extension 3** Address extension bits A[31:22] for external accesses with A[23:22] = ’11’, default: 1111000000_B ($F0000\ 0000_H$ - $F03F\ FFFF_H$)
RMWEN	31	rw	**Read Modify Write Enable** Disables/enables read/modify/write accesses to internal resources (via the FPI bus): 0: Read/Modify/Write accesses are disabled 1: Read/Modify/Write accesses are enabled *Note: This bit must only be written with ‘1’ when used with a compatible external master (e.g. another instance of EBU)*
0	30	r	**Reserved bits** These bits are reserved and read as ‘0’, must be written with ‘0’.

14.12.9 EBU Register Address Range

In the TC11IB, the registers of the EBU are located in the following address range:

- Module Base Address. $F800\ 0000_H$
  Module End Address. $F800\ 02FF_H$
- Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 14-33**)

15 Interrupt System

The TC11IB interrupt system provides a flexible and time-efficient means for processing interrupts. This chapter describes the interrupt system for the TC11IB. Topics covered include the architecture of the interrupt system, interrupt system configuration, and the interrupt operations of the TC11IB peripherals and Central Processing Unit (CPU). General information is also given about the Peripheral Control Processor (PCP). For details about that unit, see **Chapter 17**.

15.1 Overview

An interrupt request can be serviced either by the CPU or by the Peripheral Control Processor (PCP). These units are called “Service Providers”. Interrupt requests are called “Service Requests” rather than “Interrupt Requests” in this document because they can be serviced by either of the Service Providers.

Each peripheral in the TC11IB can generate service requests. Additionally, the Bus Control Unit, the Debug Unit, the PCP, and even the CPU itself can generate service requests to either of the two Service Providers.

As shown in **Figure 15-1**, each TC11IB unit that can generate service requests is connected to one or multiple Service Request Nodes (SRN). Each SRN contains a Service Request Control Register mod_SRCx, where “mod” is the identifier of the service requesting unit and “x” an optional index. Two buses connect the SRNs with two Interrupt Control Units, which handle interrupt arbitration among competing interrupt service requests, as follows:

- The Interrupt Control Unit (ICU) arbitrates service requests for the CPU and administers the CPU Interrupt Arbitration Bus.
- The Peripheral Interrupt Control Unit (PICU) arbitrates service requests for the PCP and administers the PCP Interrupt Arbitration Bus.

Units which can generate service requests are:

- General Purpose Timer Units (GPTU 0 and GPTU 1) with 8 SRNs each
- High-Speed Synchronous Serial Interface (SSC) with 3 SRNs
- Asynchronous/Synchronous Serial Interfaces (ASC) with 4 SRNs
- Asynchronous Serial Interface (16x50) with 1 SRN
- MultiMediaCard Interface (MMCI) with 1 SRN
- Ethernet Controller with 9 SRNs
- PCI Interface with 1 SRN
- PCI Software Interrupts with 32 SRNs
- External Interrupts with 24 SRNs
- FPI Bus Control Units (BCU0 and BCU1) with 1 SRN each
- LMB Bus Control Units (LCU) with 1 SRN
- Peripheral Control Processor (PCP) with 12 SRNs
- Central Processing Unit (CPU) with 4 SRNs

- Debug Unit (OCDS) with 1 SRN

The PCP can make service requests directly to itself (via the PICU), or it can make service requests to the CPU. The Debug Unit can generate service requests to the PCP or the CPU. The CPU can make service requests directly to itself (via the ICU), or it can make service requests to the PCP. The CPU Service Request Nodes are activated through software.

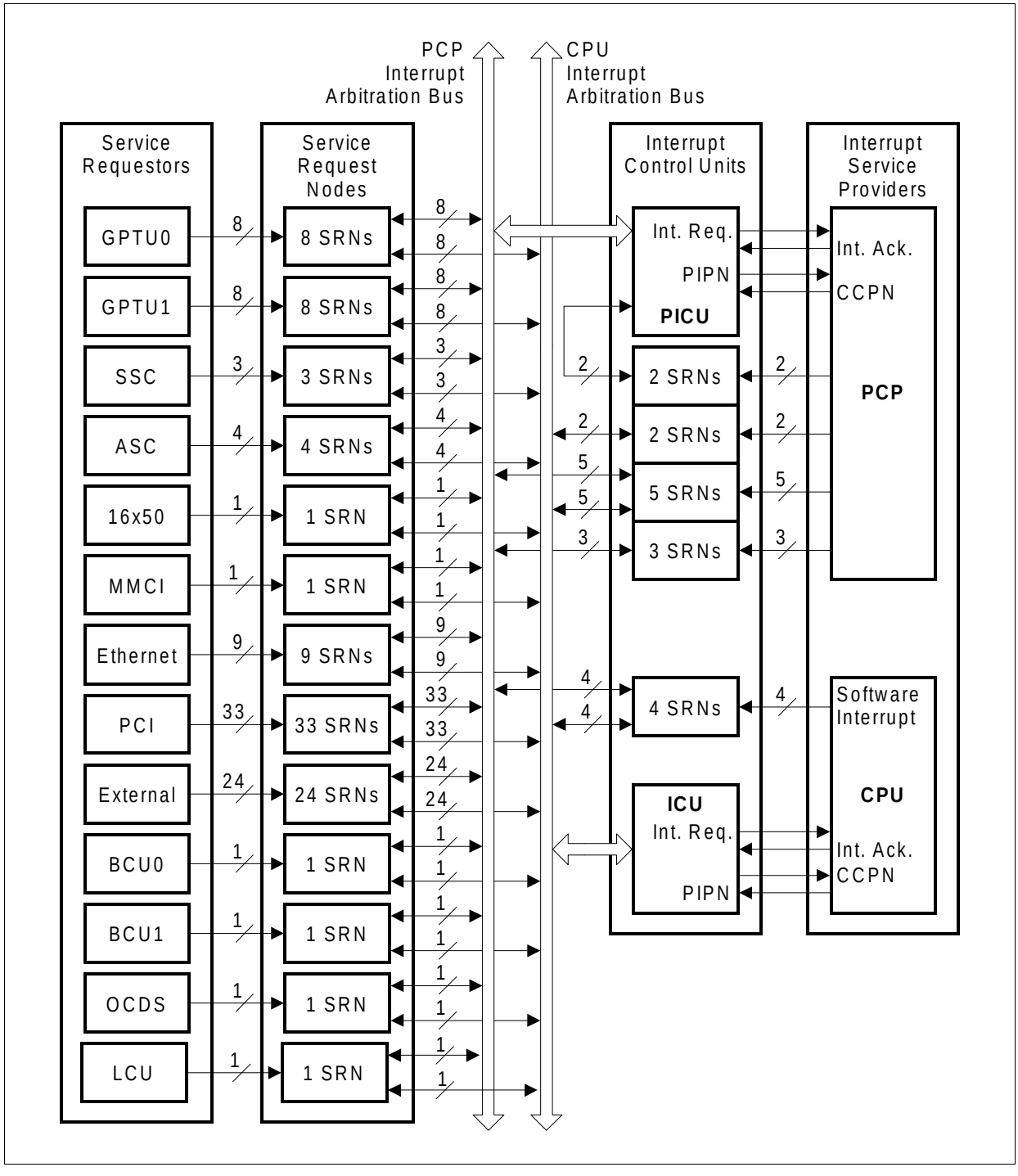


Figure 15-1 Block Diagram of the TC11IB Interrupt System

15.2 Service Request Nodes

In total, there are 111 Service Request Nodes available in the TC11IB. Note that the four CPU Service Request Nodes can be activated only by software (either CPU instructions or PCP instructions).

Each Service Request Node contains a Service Request Control Register (SRC) and interface logic that connects it to the triggering unit on one side and to the two interrupt arbitration buses on the other side. Some peripheral units of the TC11IB have multiple Service Request Nodes.

15.2.1 Service Request Control Registers

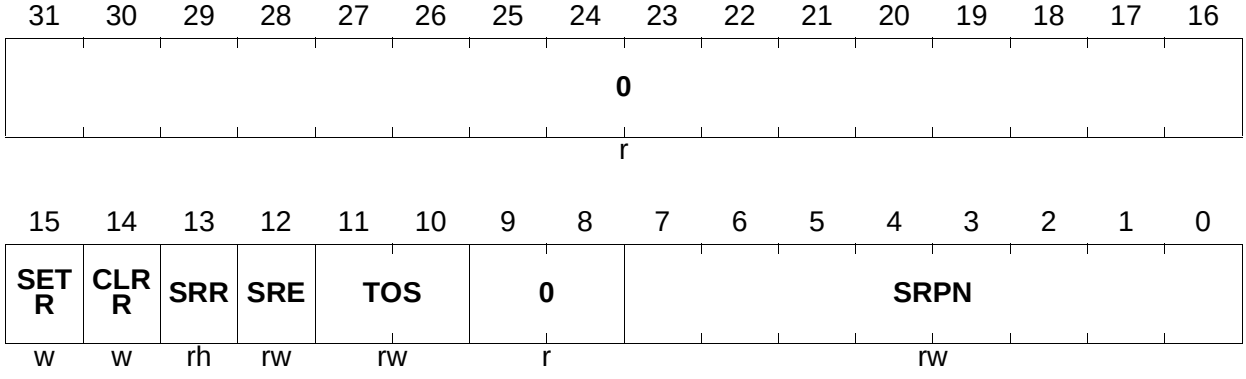
All Service Request Control Registers in the TC11IB have the same format. In general, these registers contain:

- Enable/disable information
- Priority information
- Service Provider destination
- Service request active status bit
- Software-initiated service request set and reset bits

Besides being activated by the associated triggering unit through hardware, each SRN can also be set or reset by software via two software-initiated service request control bits.

Note: The description given in this chapter characterizes all Service Request Control Registers of the TC11IB. Informations on further peripheral module interrupt functions, such as enable or request flags, are described in the corresponding chapters of the peripheral modules.

mod_SRC
Service Request Control Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number** 00_H Service request is never serviced 01_H Service request is on lowest priority .. .. FF_H Service request is on highest priority
TOS	[11:10]	rw	**Type of Service Control** 00_B CPU service is initiated 01_B PCP request is initiated $1X_B$ Reserved
SRE	12	rw	**Service Request Enable** 0 Service request is disabled 1 Service request is enabled
SRR	13	rh	**Service Request Flag** 0 No service request is pending 1 A service request is pending
CLRR	14	w	**Request Clear Bit** CLRR is required to reset SRR. 0 No action 1 Clear SRR; bit value is not stored; read always returns 0; no action if SETR is set also.
SETR	15	w	**Request Set Bit** SETR is required to set SRR. 0 No action 1 Set SRR; bit value is not stored; read always returns 0; no action if CLRR is set also.
0	[9:8], [31:16]	r	**Reserved**; read as 0; should be written with 0.

15.2.2 Service Request Flag (SRR)

A trigger event in a peripheral associated with this register causes SRR to be set to 1. Service requests can be acknowledged automatically by hardware or can be polled by software. If the corresponding enable bit SRE is set, a service request will be forwarded for arbitration to the Service Provider indicated by the TOS bit. When the service request is acknowledged by the Service Provider (either the CPU or the PCP), this bit is reset by hardware to 0.

When set, the SRR flag indicates that a service request is pending. It can be set or reset directly by associated hardware or indirectly through software using the SETR and CLRR bits. For instance, in the General Purpose Timer Unit, an associated timer event can cause this bit to be set to 1. The details of how hardware events can cause the SRR bit to be set are defined in the individual peripheral chapters.

The acknowledgment of the service request by either the Interrupt Control Unit (ICU) or the PCP Interrupt Control Unit (PICU) causes the SRR bit to be cleared.

SRR can be set or cleared either by hardware or by software regardless of the state of the enable bit SRE. However, the request is only forwarded for service if the enable bit is set. If SRE = 1, a pending service request takes part in the interrupt arbitration of the service provider selected by the device's TOS field. If SRE = 0, a pending service request is excluded from interrupt arbitrations.

SRR is automatically reset by hardware when the service request is acknowledged and serviced. The SRR bit can also be monitored, set, and reset by software via the SETR or CLRR bits respectively. This allows software to poll for events in peripheral devices. SRR must be reset by software in this case by writing a 1 to CLRR. Writing directly to this bit via software has no effect.

15.2.2.1 Request Set and Clear Bits (SETR, CLRR)

The SETR and CLRR bits allow software to set or clear the service request bit SRR. Writing a 1 to SETR causes bit SRR to be set to 1. Writing a 1 to CLRR causes bit SRR to be cleared to 0. If hardware attempts to modify SRR during an atomic read-modify-write software operation (such as the bit-set or bit-clear instructions) the software operation will succeed and the hardware operation will have no effect.

The value written to SETR or CLRR is not stored. Writing a 0 to these bits has no effect. These bits always return 0 when read. If both, SETR and CLRR are set to 1 at the same time, SRR is not changed.

15.2.2.2 Enable Bit (SRE)

The SRE bit enables an interrupt to take part in the arbitration for the selected Service Provider. It does not enable or disable the setting of the request flag SRR; the request flag can be set by hardware or by software (via SETR) independent of the state of the

SRE bit. This allows service requests to be handled automatically by hardware or through software polling.

If SRE = 1, pending service requests are passed on to the designated Service Provider for interrupt arbitration. The SRR bit is automatically set to 0 by hardware when the service request is acknowledged and serviced. It is recommended that in this case, software should not modify the SRR bit to avoid unexpected behavior due to the hardware controlling this bit.

If SRE = 0, pending service requests are not passed on to Service Providers. Software can poll the SRR bit to check whether a service request is pending. To acknowledge the service request, the SRR bit must then be reset by software by writing a 1 to CLRR.

Note: In this document, 'active source' means a Service Request Node whose Service Request Control Register has its request enable bit SRE set to 1 to allow its service requests to participate in interrupt arbitration.

15.2.3 Type-of-Service Control (TOS)

There are two Service Providers for service requests in the TC11IB, the CPU and the PCP. The TOS bit field is used to select whether a service request generates an interrupt to the CPU (TOS[0] = 0) or to the PCP (TOS[0] = 1). Bit TOS[1] is read-only, returning 0 when read. Writing to this bit position has no effect. However, to ensure compatibility with future extensions, it should always be written with 0.

15.2.4 Service Request Priority Number (SRPN)

The 8-bit Service Request Priority Number (SRPN) indicates the priority of a service request with respect to other sources requesting service from the same Service Provider, and with respect to the priority of the Service Provider itself.

Each active source selecting the same Service Provider must have a unique SRPN value to differentiate its priority. The special SRPN value of 00_H excludes an SRN from taking part in arbitration, regardless of the state of its SRE bit. The SRPN values for active sources selecting different Service Providers (CPU vs. PCP) may overlap. If a source is not active — meaning its SRE bit is 0 — no restrictions are applied to the service request priority number.

The SRPN is used by Service Providers to select an Interrupt Service Routine (ISR) or Channel Program (in case of the PCP) to service the request. ISRs are associated with Service Request Priority Numbers by an Interrupt Vector Table located in each Service Provider. This means that the TC11IB Interrupt Vector Table is ordered by priority number. This is unlike traditional interrupt architectures in which their interrupt vector tables are ordered by the source of the interrupt. The TC11IB Interrupt Vector Table allows a single peripheral can have multiple priorities for different purposes.

The range of values for SRPNs used in a system depends on the number of possible active service requests and the user-definable organization of the Interrupt Vector Table.

The 8-bit SRPNs permit up to 255 sources to be active at one time (remembering that the special SRPN value of 00_H excludes an SRN from taking part in arbitration).

15.3 Interrupt Control Units

The Interrupt Control Units manage the interrupt system, arbitrate incoming service requests, and determine whether and when to interrupt the Service Provider. The TC11IB contains two interrupt control units, one for the CPU (called ICU), and one for the PCP (called PICU). Each one controls its associated interrupt arbitration bus and manages the communication with its Service Provider (see **Figure 15-1**).

15.3.1 Interrupt Control Unit (ICU)

15.3.1.1 ICU Interrupt Control Register (ICR)

The ICU Interrupt Control Register ICR holds the current CPU priority number (CCPN), the global interrupt enable/disable bit (IE), the pending interrupt priority number (PIPN), and bit fields which control the interrupt arbitration process.

ICR
ICU Interrupt Control Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					C ONE CYC	CARBCYC		PIPN							
r					rw	rw		rh							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0							IE	CCPN							
r							rwh	rwh							

Field	Bits	Type	Description
CCPN	[7:0]	rwh	**Current CPU Priority Number** The Current CPU Priority Number (CCPN) bit field indicates the current priority level of the CPU. It is automatically updated by hardware on entry and exit of interrupt service routines, and through the execution of a BISR instruction. CCPN can also be updated through an MTCR instruction.

Field	Bits	Type	Description
IE	8	rwh	**Global Interrupt Enable Bit** The interrupt enable bit globally enables the CPU service request system. Whether a service request is delivered to the CPU depends on the individual Service Request Enable Bits (SRE) in the SRNs, and the current state of the CPU. IE is automatically updated by hardware on entry and exit of an Interrupt Service Routine (ISR). IE is cleared to 0 when an interrupt is taken, and is restored to the previous value when the ISR executes an RFE instruction to terminate itself. IE can also be updated through the execution of the ENABLE, DISABLE, MTCR, and BISR instructions. 0 Interrupt system is globally disabled 1 Interrupt system is globally enabled
PIPN	[23:16]	rh	**Pending Interrupt Priority Number** PIPN is a read-only bit field that is updated by the ICU at the end of each interrupt arbitration process. It indicates the priority number of the pending service request. PIPN is set to 0 when no request is pending, and at the beginning of each new arbitration process. 00_H No valid pending request YY_H A request with priority YY_H is pending
CARBCYC	[25:24]	rw	**Number of Arbitration Cycles** CARBCYC controls the number of arbitration cycles used to determine the request with the highest priority. 00_B 4 arbitration cycles (default) 01_B 3 arbitration cycles 10_B 2 arbitration cycles 11_B 1 arbitration cycle
CONECYC	26	rw	**Number of Clocks per Arbitration Cycle Control** The CONECYC bit determines the number of system clocks per arbitration cycle. This bit should be set to 1 only for system designs utilizing low system clock frequencies. 0 2 clocks per arbitration cycle (default) 1 1 clock per arbitration cycle
0	[15:9], [31:27]	r	**Reserved**; read as 0; should be written with 0.

15.3.1.2 Operation of the Interrupt Control Unit (ICU)

Service-request arbitration is performed in the ICU in parallel with normal CPU operation. When a triggering event occurs in one or more interrupt sources, the associated SRNs, if enabled, send service requests to the ICU via the arbitration bus. The ICU determines which service request has the highest priority. The ICU will then forward the service request to the CPU. The service request will be acknowledged by the CPU and serviced, depending upon the state of the CPU.

The ICU arbitration process takes place in one or more arbitration cycles over the CPU Interrupt Arbitration Bus. The ICU begins a new arbitration process when a new service request is detected. At the end of the arbitration process, the ICU will have determined the service request with the highest priority number. This number is stored in the ICR.PIPN bit field and becomes the pending service request.

After the arbitration process, the ICU forwards the pending service request to the CPU by attempting to interrupt it. The CPU can be interrupted only if interrupts are enabled globally (that is, ICR.IE = 1) and if the priority of the service request is higher than the current processor priority (ICR.PIPN > ICR.CCPN). Also, the CPU may be temporarily blocked from taking interrupts, for example, if it is executing a multi-cycle instruction such as an atomic read-modify-write operation. The full list of conditions which could block the CPU from immediately responding to an interrupt request generated by the ICU is:

- Current CPU priority, ICR.CCPN, is equal to or higher than the pending interrupt priority, ICR.PIPN
- Interrupt system is globally disabled (ICR.IE = 0)
- CPU is in the process of entering an interrupt- or trap-service routine
- CPU is executing non-interruptible trap services
- CPU is executing a multi-cycle instruction
- CPU is executing an instruction which modifies the conditions of the global interrupt system, such as modifying the ICR
- CPU detects a trap condition (such as context depletion) when trying to enter a service routine

When the CPU is not otherwise prevented from taking an interrupt, the CPU’s program counter will be directed to the Interrupt Service Routine entry point associated with the priority of the service request. Now, the CPU saves the value of ICR.PIPN internally, and acknowledges the ICU. The ICU then forwards the acknowledge signal back to the SRN that is requesting service, to inform it that it will be serviced by the CPU. The SRR bit in this SRN is then reset to 0.

After sending the acknowledgement, the ICU resets ICR.PIPN to 0 and immediately starts a new arbitration process to determine if there is another pending interrupt request. If not, ICR.PIPN remains at 0 and the ICU enters an idle state, waiting for the next interrupt request to awaken it. If there is a new service request waiting, the priority number of the new request will be written to ICR.PIPN at the end of the new arbitration

process and the ICU will deliver the pending interrupt to the CPU according to the rules described in this section.

If a new service request is received by the ICU before the CPU has acknowledged the pending interrupt request, the ICU deactivates the pending request and starts a new arbitration process. This reduces the latency of service requests posted before the current request is acknowledged. The ICU deactivates the current pending interrupt request by setting the ICR.PIPN bit field to 0, indicating that the ICU has not yet found a new valid pending request. It then executes its arbitration process again. If the new service request has a higher priority than the previous one, its priority will be written to ICR.PIPN. If the new interrupt has a lower priority, the priority of the previous interrupt request will again be written to ICR.PIPN. In any case, the ICU will deliver a new interrupt request to the CPU according to the rules described in this section.

Once the CPU has acknowledged the current pending interrupt request, any new service request generated by an SRN must wait at least until the end of the next service request arbitration process to be serviced.

Essentially, arbitration in the ICU is performed whenever a new service request is detected, regardless of whether or not the CPU is servicing interrupts. Because of this, the ICR.PIPN bit field always reflects the pending service request with the highest priority. This can, for example, be used by software polling techniques to determine high-priority requests while leaving the interrupt system disabled.

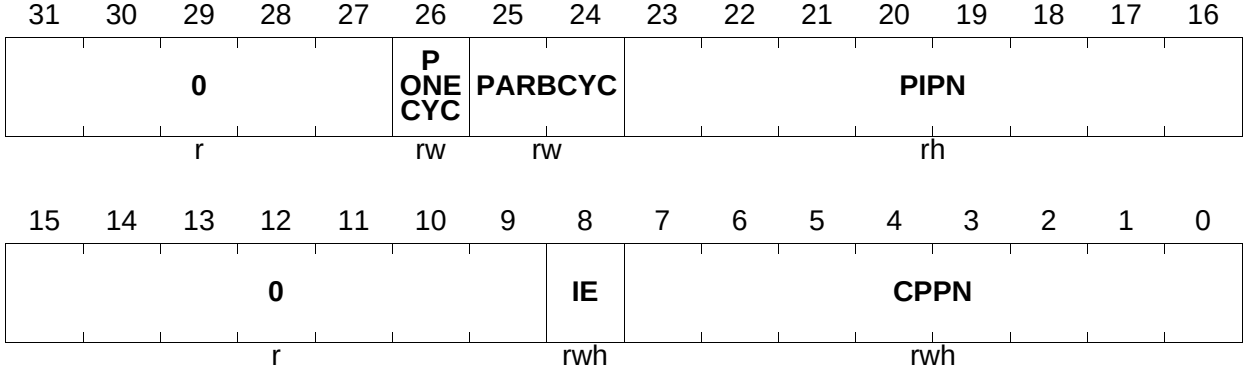
15.3.2 PCP Interrupt Control Unit (PICU)

The PCP Interrupt Control Unit (PICU) is closely coupled with the PCP and its Interrupt Control Register (PCP_PICR). The operation of the PICU is very similar to the ICU of the CPU with respect to the overall scheme. However, the PCP cannot handle nested interrupts, that is, an interrupt request to the PCP cannot interrupt the service of another interrupt request. Thus, schemes such as interrupt priority grouping, are not feasible in the PCP.

15.3.2.1 PICU Interrupt control Register

The PCP_ICR register is nearly identical to the ICR register of the CPU. It holds the current PCP priority number (PCP_ICR_CPPN), the global interrupt enable/disable bit (PCP_ICR_IE), the pending interrupt priority number (PCP_ICR_PIPN), as well as bits to control interrupt arbitration cycles.

PCP_ICR
PCP Interrupt Control Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
CPPN	[7:0]	rwh	**Current PCP Priority Number** The Current PCP Priority Number (CPPN) bit field indicates the current priority level of the PCP. It is automatically updated by hardware on entry and exit of a PCP Channel Program.
IE	8	rwh	**Global Interrupt Enable Bit** The Global Interrupt Enable status bit is updated by hardware according to the state of the INT bit in the register R7. 0 Interrupt system is globally disabled 1 Interrupt system is globally enabled
PIPN	[23:16]	rh	**Pending Interrupt Priority Number** PIPN is a read-only bit field that is updated by the PICU at the end of each interrupt arbitration process. It indicates the priority number of the pending service request. PIPN is set to 0 when no request is pending, and at the beginning of each new arbitration process. 00_H No valid pending request YY_H A request with priority YY_H is pending
PARBCYC	[25:24]	rw	**Number of Arbitration Cycles** PARBCYC controls the number of arbitration cycles used to determine the request with the highest priority. 00_B 4 arbitration cycles (default) 01_B 3 arbitration cycles 10_B 2 arbitration cycles 11_B 1 arbitration cycle

Field	Bits	Type	Description
PONECYC	26	rw	**Number of Clocks per Arbitration Cycle Control** The CONECYC bit determines the number of system clocks per arbitration cycle. This bit should be set to 1 only for system designs utilizing low system clock frequencies. 0 2 clocks per arbitration cycle (default) 1 1 clock per arbitration cycle
0	[15:9], [31:27]	r	**Reserved**; read as 0; should be written with 0.

15.4 Arbitration Process

The arbitration process implemented in the TC11IB uses a number of arbitration cycles to determine the pending interrupt request with the highest priority number, SRPN. In each of these cycles, two bits of the SRPNs of all pending service requests are compared against each other. The sequence starts with the high-order bits of the SRPNs and works downwards, such that in the last cycle, bits[1:0] of the SRPNs are compared. Thus, to perform an arbitration through all 8 bits of an SRPN, four arbitration cycles are required. There are two factors determining the duration of the arbitration process:

- Number of arbitration cycles, and
- Duration of arbitration cycles.

Both of these can be controlled by the user, as described in the following sections.

15.4.1 Controlling the Number of Arbitration Cycles

In a real-time system where responsiveness is critical, arbitration must be as fast as possible. Yet to maintain flexibility, the TC11IB system is designed to have a large range of service priorities. If not all priorities are needed in a system, arbitration can be speeded up by not examining all the bits used to identify all 255 unique priorities.

For instance, if a 6-bit number is enough to identify all priority numbers used in a system, (meaning that bits [7:6] of all SRPNs are always 0), it is not necessary to perform arbitration on these two bits. Three arbitration cycles will be enough to find the highest number in bits [5:0] of the SRPNs of all pending requests. Similarly, the number of arbitration cycles can be reduced to two if only bits [3:0] are used in all SRPNs, and the number of arbitration cycles can be reduced to one cycle if only bits [1:0] are used.

The ICR.CARBCYC bit field controls the number of cycles in the arbitration process. Its default value is 0, which selects four arbitration cycles. **Table 15-1** gives the options for arbitration cycle control.

Table 15-1 Arbitration Cycle Control

Number of Arbitration Cycles	4	3	2	1
ICR.CARBCYC	00_B	01_B	10_B	11_B
Relevant bits of the SRPNs	[7:0]	[5:0]	[3:0]	[1:0]
Range of priority numbers covered	1..255	1..63	1..15	1..3

Note: If less than four arbitration cycles are selected, the corresponding upper bits of the SRPNs are not examined, even if they do not contain zeros.

15.4.2 Controlling the Duration of Arbitration Cycles

During each arbitration cycle, the rate of information flow between the SRNs and the ICU can become limited by propagation delays within the TC11IB when it is executing at high system clock frequencies. At high frequencies, arbitration cycles may require two system clocks to execute properly. In order to optimize the arbitration scheme at lower system frequencies, an additional control bit, ICR.CONECYC is implemented. The default value of 0 of this bit selects two clock cycles per arbitration cycle. Setting this bit to 1 selects one clock cycle per arbitration cycle. This bit should only be set to 1 for lower system frequencies. Setting this bit for system frequencies above the specified limit leads to unpredictable behavior of the interrupt system. Correct operation is not then guaranteed.

15.5 Entering an Interrupt Service Routine

When an interrupt request from the ICU is pending and all conditions are met such that the CPU can now service the interrupt request, the CPU performs the following actions in preparation for entering the designated Interrupt Service Routine (ISR):

1. Upper context of the current task is saved[1)]. The current CPU priority number, ICR.CCPN, and the state of the global interrupt enable bit, ICR.IE, are automatically saved with the PCXI register (bit field PCPN and bit PIE).
2. Interrupt system is globally disabled (ICR.IE is set to 0).
3. Current CPU priority number (ICR.CCPN) is set to the value of ICR.PIPN.
4. PSW is set to a default value:
   - All permissions are enabled, that is, PSW.IO = 10_B
   - Memory protection is switched to PRS0, that is, PSW.PRS = 0.
   - The stack pointer bit is set to the interrupt stack, that is, PSW.IS = 1.
   - The call depth counter is cleared, the call depth limit is set to 63, that is, PSW.CDC = 0.

[1)] Note that, if a context-switch trap occurs while the CPU is in the process of saving the upper context of the current task, the pending ISR will not be entered, the interrupt request will be left pending, and the CPU will enter the appropriate trap handling routine instead.

5. Stack pointer, A10, is reloaded with the contents of the Interrupt Stack Pointer, ISP, if the PSW.IS bit of the interrupted routine was set to 0 (using the user stack), otherwise it is left unaltered.
6. CPU program counter is assigned an effective address consisting of the contents of the BIV register ORed with the ICR.PIPN number left-shifted by 5. This indexes the Interrupt Vector Table entry corresponding to the interrupt priority.
7. Contents at the effective address of the program counter in the Interrupt Vector Table is fetched as the first instruction of the Interrupt Service Routine (ISR). Execution continues linearly from there until the ISR branches or exits.

As explained, receipt of further interrupts is disabled (ICR.IE = 0) when an Interrupt Service Routine is entered. At the same time, the current CPU priority ICR.CCPN is set by hardware to the priority of the interrupting source (ICR.PIPN).

Clearly, before the processor can receive any more interrupts, the ISR must eventually re-enable the interrupt system again by setting ICR.IE = 1. Furthermore, the ISR can also modify the priority number ICR.CCPN to allow effective interrupt priority levels. It is up to the user to enable the interrupt system again and optionally modify the priority number CCPN to implement interrupt priority levels or handle special cases (see next sections).

To simply enable the interrupt system again, the ENABLE instruction can be used, which sets ICR.IE bit to 1. The BISR instruction offers a convenient way to re-enable the interrupt system, to set ICR.CCPN to a new value, and to save the lower context of the interrupted task. It is also possible to use an MTCR instruction to modify ICR.IE and ICR.CCPN. However, this should be performed together with an ISYNC instruction (which synchronizes the instruction stream) to ensure completion of this operation before the execution of following instructions.

Note: The lower context can also be saved through execution of a SVLCX (Save Lower Context) instruction.

15.6 Exiting an Interrupt Service Routine

When an ISR exits with an RFE (Return From Exception) instruction, the hardware automatically restores the upper context. Register PCXI, which holds the Previous CPU Priority Number (PCPN) and the Previous Global Interrupt Enable Bit (PIE), is a part of this upper context. The value saved in PCPN is written to ICR.CCPN to set the CPU priority number to the value before the interruption, and bit PIE is written to ICR.IE to restore the state of this bit. The interrupted routine then continues.

Note: There is no automatic restoring of the lower context on an exit from an Interrupt Service Routine. If the lower context was saved during the execution of the ISR, either through execution of the BISR instruction or a SVLCX instruction, the ISR must restore the lower context again via the RSLCX (Restore Lower Context) instruction before it exits through RFI execution.

15.7 Interrupt Vector Table

Interrupt Service Routines are associated with interrupts at a particular priority by way of the Interrupt Vector Table. The Interrupt Vector Table is an array of Interrupt Service Routine entry points.

When the CPU takes an interrupt, it calculates an address in the Interrupt Vector Table that corresponds with the priority of the interrupt (the ICR.PIPN bit field). This address is loaded in the program counter. The CPU begins executing instructions at this address in the Interrupt Vector Table. The code at this address is the start of the selected Interrupt Service Routine (ISR). Depending on the code size of the ISR, the Interrupt Vector Table may only store the initial portion of the ISR, such as a jump instruction that vectors the CPU to the rest of the ISR elsewhere in memory.

The Interrupt Vector Table is stored in code memory. The BIV register specifies the base address of the Interrupt Vector Table. Interrupt vectors are ordered in the table by increasing priority.

The Base of Interrupt Vector Table register (BIV) stores the base address of the Interrupt Vector Table. It can be assigned to any available code memory. Its default on power-up is fixed at $0000\ 0000_H$. However, the BIV register can be modified using the MTCR instruction during the initialization phase of the system, before interrupts are enabled. With this arrangement, it is possible to have multiple Interrupt Vector Tables and switch between them by changing the contents of the BIV register.

*Note: The BIV register is protected by the ENDINIT bit (see **Chapter 20**). Modifications should only be done while the interrupt system is globally disabled (ICR.IE = 0). Also, an ISYNC instruction should be issued after modifying BIV to ensure completion of this operation before execution of following instructions.*

When interrupted, the CPU calculates the entry point of the appropriate Interrupt Service Routine from the PIPN and the contents of the BIV register. The PIPN is left-shifted by five bits and ORed with the address in the BIV register to generate a pointer into the Interrupt Vector Table. Execution of the ISR begins at this address. Due to this operation, it is recommended that bits [9:7] of register BIV are set to 0 (see **Figure 15-2**). Note that bit 0 of the BIV register is always 0 and cannot be written to (instructions have to be aligned on even byte boundaries).

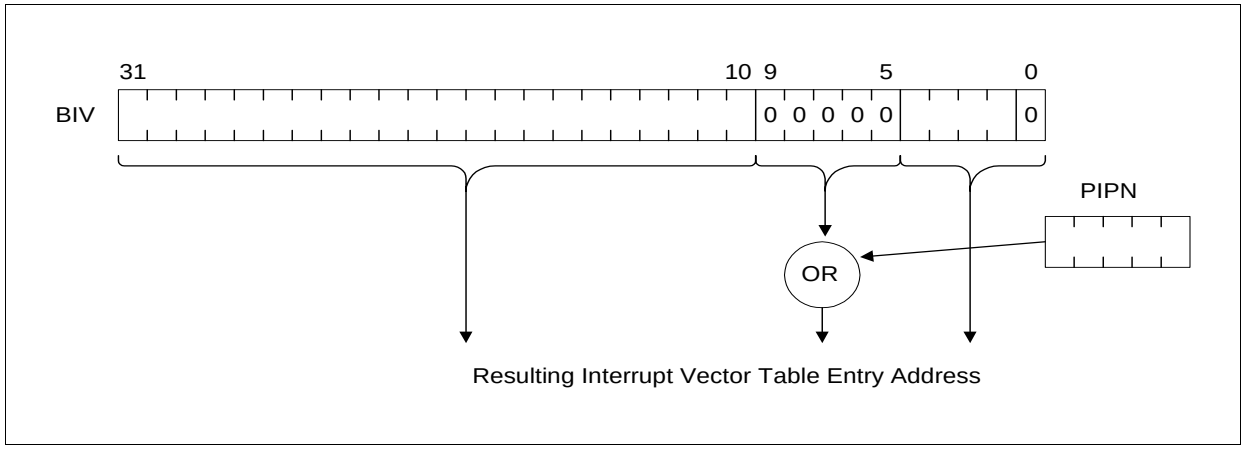


Figure 15-2 Interrupt Vector Table Entry Address Calculation

Left-shifting the PIPN by 5 bits creates entries into the Interrupt Vector Table which are evenly spaced 8 words apart. If an ISR is very short, it may fit entirely within the eight words available in the vector table entry. Otherwise, the code at the entry point must ultimately cause a jump to the rest of the ISR residing elsewhere in memory. Due to the way the vector table is organized according to the interrupt priorities, the TC11IB offers an additional option by allowing to span several Interrupt Vector Table entries so long as those entries are otherwise unused. **Figure 15-3** illustrates this.

The required size of the Interrupt Vector Table depends only on the range of priority numbers actually used in a system. Of the 256 vector entries, 255 may be used. Vector entry 0 is never used, because if ICR.PIPN is 0, the CPU is not interrupted. Distinct interrupt handlers are supported, but systems requiring fewer entries need not dedicate the full memory area required by the largest configurations.

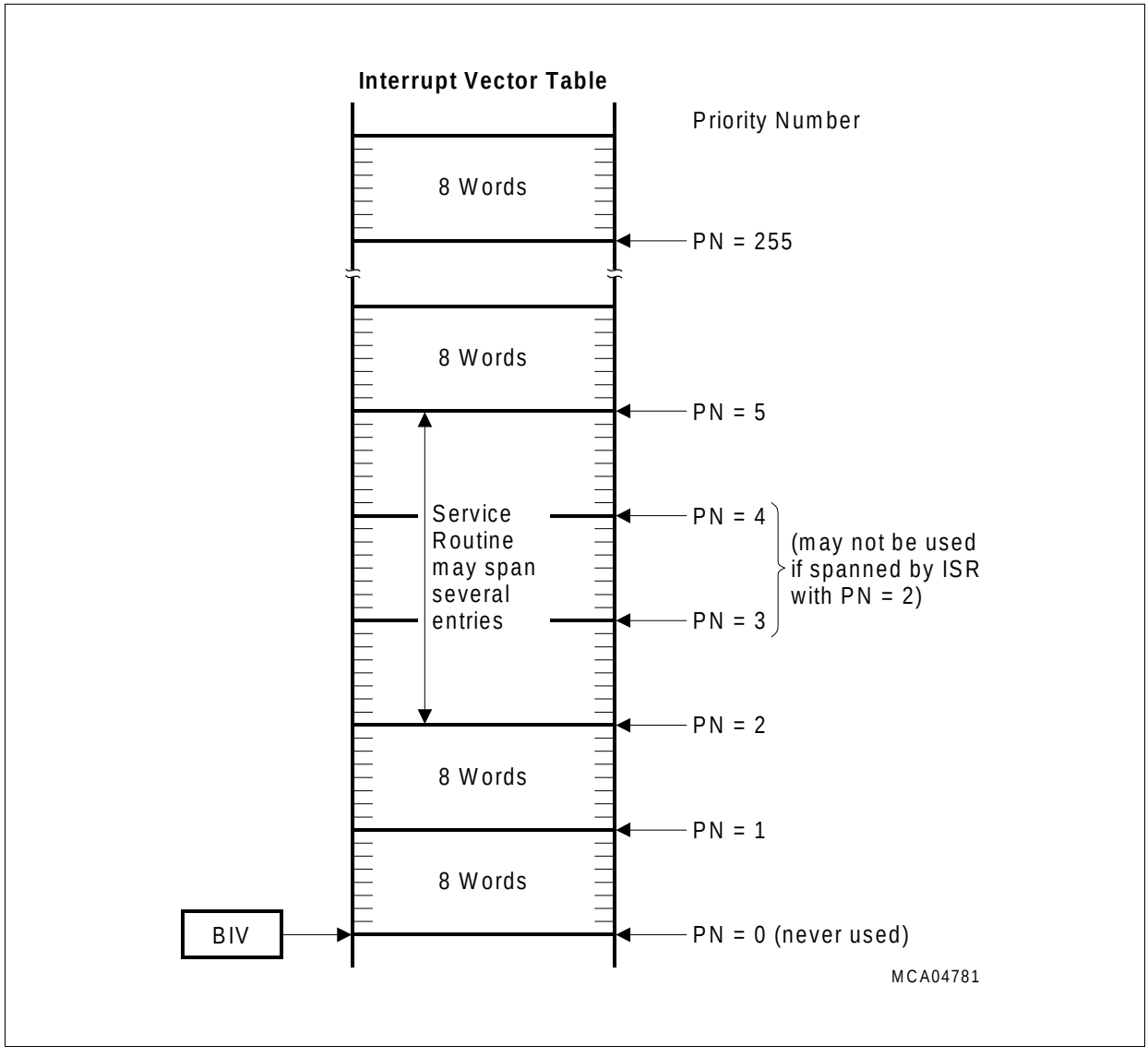


Figure 15-3 Interrupt Vector Table

15.8 Usage of the TC11IB Interrupt System

The following sections give some examples of using the TC11IB interrupt system to solve both typical and special application requirements.

15.8.1 Spanning Interrupt Service Routines Across Vector Entries

Each Interrupt Vector Table entry consists of eight words of memory. If an ISR can be made to fit directly in the Interrupt Vector Table there is no need for a jump instruction to vector to the rest of the interrupt handler elsewhere in memory. However, only the simplest ISRs can fit in the eight words available to a single entry in the table. But it is easy to arrange for ISRs to span across multiple entries, since the Interrupt Vector Table

is ordered not by the interrupt source but by interrupt priority. This technique is explained in this section.

In the example of **Figure 15-3**, entry locations 3 and 4 are occupied by the ISR for entry 2. In **Figure 15-3**, the next available entry after entry 2 is entry 5. Of course, if this technique is used, it would be improper to allow any SRN to request service at any of the spanned vector priorities. Thus, priority levels 3 and 4 must not be assigned to SRNs requesting CPU service. They can, however, be used to request PCP service.

There is a performance trade-off which may arise when using this technique because the range of priority numbers used increases. This may have an impact on the number of arbitration cycles required to perform arbitration. Consider the case in which a system uses only three active interrupt sources, that is, where there are only three SRNs enabled to request service. If these three active sources are assigned to priority numbers 1, 2, and 3, it would be sufficient to perform the arbitration in just one cycle. However, if the ISR for interrupt priority 2 is spanned across three Interrupt Vector Table entries as shown in **Figure 15-3**, the priority numbers 1, 2 and 5 would have to be assigned. Thus, two arbitration cycles would have to be used to perform the full arbitration process.

The trade-off between the performance impact of the number of arbitration cycles and the performance gain through spanning service routines can be made by the system designer depending on system needs. Reducing the number of arbitration cycles reduces the service request arbitration latency - spanning service routines reduces the run time of service routines (and therefore also the latency for further interrupts at that priority level or below). For example, if there are multiple fleeting measurements to be made by a system, reducing arbitration latency may be most important. But if keeping total interrupt response time to a minimum is most urgent, spanning Interrupt Vector Table entries may be a solution.

15.8.2 Configuring Ordinary Interrupt Service Routines

When the CPU starts to service an interrupt, the interrupt system is globally disabled and the CPU priority ICR.CCPN is set to the priority of the interrupt now being serviced. This blocks all further interrupts from being serviced until the interrupt system is enabled again.

After an ordinary ISR begins execution, it is usually desirable for the ISR to re-enable global interrupts so that higher-priority interrupts (that is, interrupts that are greater than the current value of ICR.CCPN) can be serviced even during the current ISR's execution. Thus, such an ISR may set ICR.IE = 1 again with, for instance, the ENABLE instruction.

If the ISR enables the interrupt system again by setting ICR.IE = 1 but does not change ICR.CCPN, the effect is that from that point on the hardware can be interrupted by higher-priority interrupts but will be blocked from servicing interrupt requests with the same or lower priority than the current value of ISR.CCPN. Since the current ISR is clearly also at this priority level, the hardware is also blocked from delivering further

interrupts to it as well. (This condition is clearly necessary so that the ISR can service the interrupt request atomically.)

When the ISR is finished, it exits with an RFE instruction. Hardware then restores the values of ICR.CCPN and ICR.IE to the values of the interrupted program.

15.8.3 Interrupt Priority Groups

It is sometimes useful to create groups of interrupts at the same or different interrupt priorities that cannot interrupt each other's ISRs. For instance, devices which can generate multiple interrupts, such as the General Purpose Timer, may need to have interrupts at different priorities interlocked in this way. The TC11IB interrupt architecture can be used to create such interrupt priority groups. It is effected by managing the current CPU priority level ICR.CCPN in a way described in this section.

If it is wished, for example, to make an interrupt priority group out of priority numbers 11 and 12, one would not want an ISR executing at priority 11 to be interrupted by a service request at priority 12, since this would be in the same priority group. One would wish that only interrupts above 12 should be allowed to interrupt the ISRs in this interrupt priority group. However, under ordinary ISR usage the ISR at priority 11 would be interrupted by any request with a higher priority number, including priority 12.

If, however, all ISRs in the interrupt priority group set the value of ICR.CCPN to the highest priority level within their group before they re-enable interrupts, then the desired interlocking will be effected.

Figure 15-4 shows an example for this. The interrupt requests with the priority numbers 11 and 12 form one group, while the requests with priority numbers 14 through 17 form another group. Each ISR in group 1 sets the value of ICR.CCPN to 12, the highest number in that group, before re-enabling the interrupt system. Each ISR in group 2 sets the value of ICR.CCPN to 17 before re-enabling the interrupt system. If, for example, interrupt 14 is serviced, it can only be interrupted by requests with a priority number higher than 17; therefore it will not be interrupted by requests from its own priority group or requests with lower priority.

In **Figure 15-4**, the interrupt request with priority number 13 can be said to form an interrupt priority group with just itself as a member.

Setting ICR.CCPN to the maximum value 255 in each service routine has the same effect as not re-enabling the interrupt system; all interrupt requests can then be considered to be in the same group.

Interrupt priority groups are an example of the power of the TC11IB priority-based interrupt-ordering system. Thus the flexibility of interrupt priority levels ranges from all interrupts in one group to each interrupt request building its own group, and to all possible combinations in between.

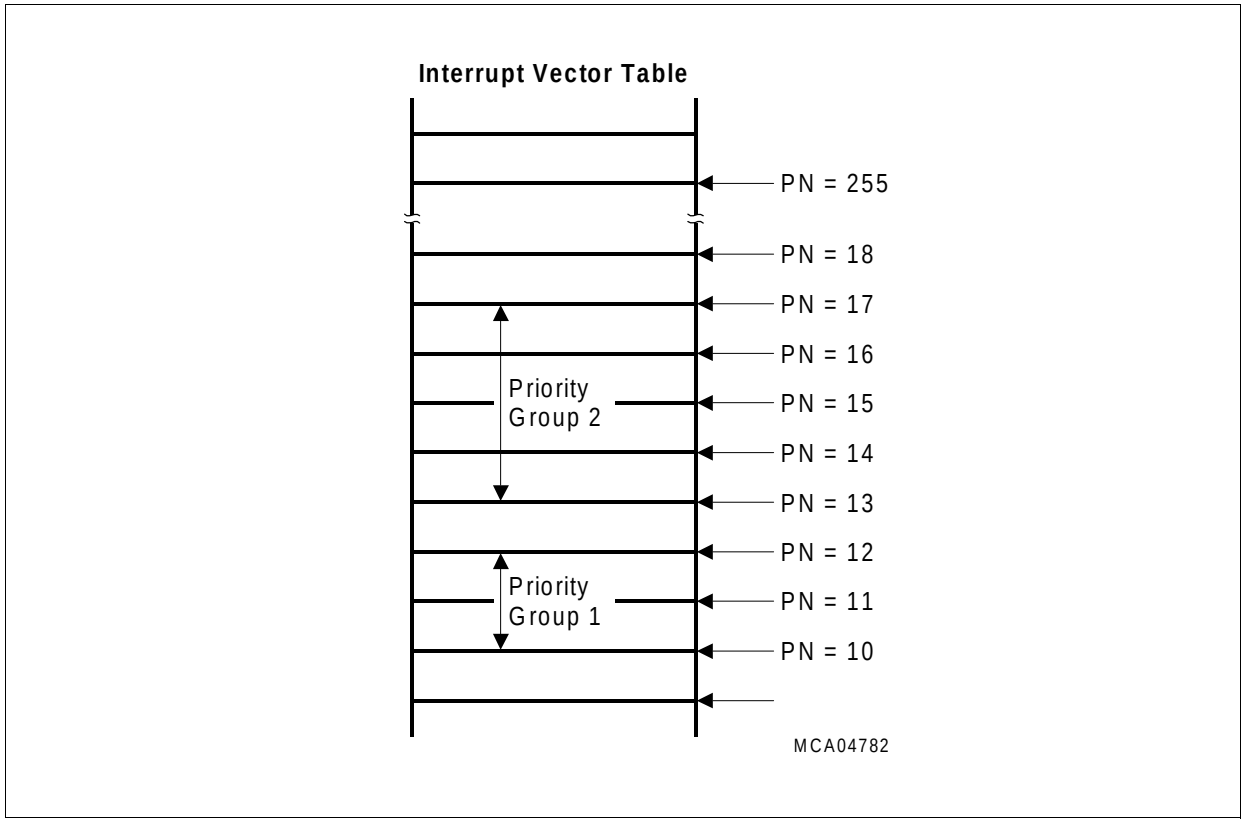


Figure 15-4 Interrupt Priority Groups

15.8.4 Splitting Interrupt Service Across Different Priority Levels

Interrupt service can be divided into multiple ISRs that execute at different priority levels. For example, the beginning stage of interrupt service may be very time-critical, such as to read a data value within a limited time window after the interrupt request activation. However, once the time-critical phase is past, there may still be more to do — for instance, to process the observation. During this second phase, it might be acceptable for this ISR to be interrupted by lower-level interrupts. This can be performed as follows.

Say for example, the initial interrupt priority is fixed very high because response time is critical. The necessary actions are carried out immediately by the ISR at that high-priority level. Then the ISR prepares to invoke another ISR at a lower priority level through software to perform the lower-priority actions.

To invoke an ISR through software, the high-priority ISR directly sets an interrupt request bit in a SRN that will invoke the appropriate low-priority ISR. Then the high-priority ISR exits.

When the high-priority ISR exits, the pending low-priority interrupt will eventually be serviced (depending on the priority of other pending interrupts). When the low-priority ISR eventually executes, the low-priority actions of the interrupt will be performed.

The inverse of this method can also be employed, where a low-priority ISR raises its own priority level, or leaves interrupts turned off while it executes. For instance, the priority of a service request might be low because the time to respond to the event is not critical, but once it has been granted service, this service should not be interrupted. In this case, the ISR could raise the value of ICR.CCPN to a priority that would exclude some or all other interrupts, or simply leave interrupts disabled.

15.8.5 Using different Priorities for the same Interrupt Source

For some applications, the urgency of a service request may vary, depending on the current state of the system. To handle this, different priority numbers (SRPNs) can be assigned at different times to a service request depending on the application needs.

Of course, Interrupt Service Routines must be placed in the Interrupt Vector Table at all addresses corresponding to the range of priorities used. If service remains the same at different priorities, copies of the ISR can be placed at the possible different entries, or the entries can all vector to a common ISR. If the ISR should execute different code depending on its priority, one need merely put the appropriate ISR in the appropriate entry of the Interrupt Vector Table.

This flexibility is another advantage of the TC11IB interrupt architecture. In traditional interrupt systems where the interrupt vectors are ordered by interrupting source, the ISR would have to check the current priority of the interrupt request and perform a branch to the appropriate code section, causing a delay in the response to the request. In the TC11IB, however, the extra check and branch in the ISR are not necessary, which reduces the interrupt latency.

Because this approach may necessitate an increase in the range of interrupt priorities, the system designer must trade off this advantage against any possible increase in the number of arbitration cycles.

15.8.6 Software Initiated Interrupts

Software can set the service request bit in a SRN by writing to its Service Request Control Register. Thus, software can initiate interrupts which are handled by the same mechanism as hardware interrupts.

After the service request bit is set in an active SRN, there is no way to distinguish between a software initiated interrupt request and a hardware interrupt request. For this reason, software should only use SRNs and interrupt priority numbers that are not being used for hardware interrupts.

The TC11IB architecture includes four Service Request Nodes which are intended solely for the purpose of generating software interrupts. These SRNs are not connected to any hardware that could generate a service request, and so are only able to be used by software. Additionally, any otherwise unused SRN can be employed to generate software interrupts.

15.8.7 Interrupt Priority 1

Interrupt Priority 1 is the first and lowest-priority entry in the Interrupt Vector Table. It is generally reserved for ISRs which perform task management. ISRs whose actions cause software-managed tasks to be created post a software interrupt request at priority level 1 to signal the event.

The ISR that triggers this event can then execute a normal return from interrupt. There is no need for it to check whether the ISR is returning to the background-task priority level (priority 0) or is returning to a lower-priority ISR that it interrupted. When there is a pending interrupt at a priority higher than the return context for the current interrupt, this interrupt will then be serviced. When a return to the background-task priority level (level 0) is performed, the software-posted interrupt at priority level 1 will be serviced automatically.

15.9 CPU Service Request Nodes

To support software initiated interrupts, the TC11IB contains four Service Request Nodes which are not attached to triggering peripherals. These SRNs can only cause interrupts when software sets the service request bit in one of their Service Request Control Registers. These SRNs are called the CPU Service Request Nodes.

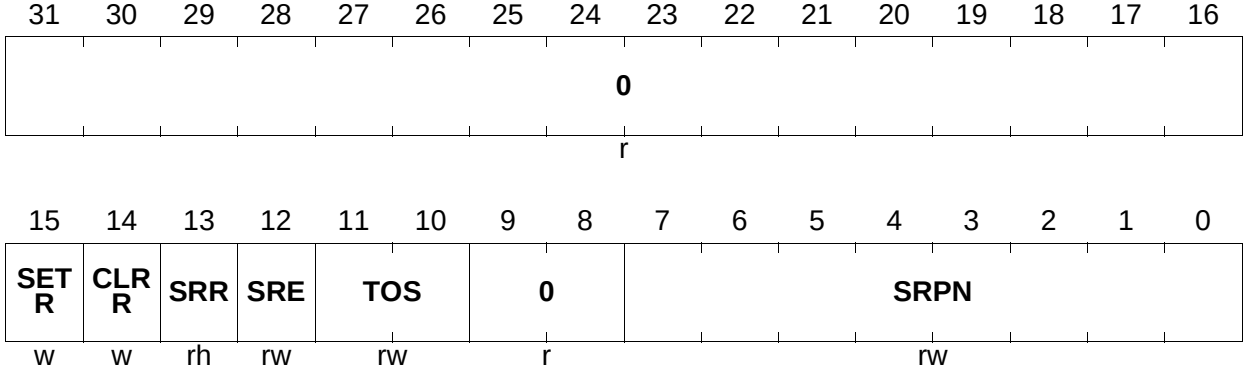
The PCP can also cause these SRNs to generate service requests. An external bus master can also generate service requests this way.

Additionally, any otherwise unused SRN can be employed to generate software interrupts.

Note: The CPU Service Request Control Registers are not bit-addressable.

CPU_SRC0
CPU Service Request Control Register 0
CPU_SRC1
CPU Service Request Control Register 1
CPU_SRC2
CPU Service Request Control Register 2
CPU_SRC3
CPU Service Request Control Register 3

Reset Values: 0000 0000$_H$



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**

Field	Bits	Type	Description
0	[9:8], [31:16]	r	**Reserved**

*Note: See **Section 15.2.1** for detailed register description.*

15.10 External Interrupts

The TC11IB contains 24 Service Request Nodes (SRN) for generating external interrupts which locate at GPIO port 3 and port 4. Each SRN contains a Service Request Control Register and interface logic that connects it to the triggering unit on one side and to the interrupt arbitration bus on the other side.

A rising or falling edge (or both) on the pin sets the Service Request flag (SRR) located in the Service Request Control Register (SRC). The Trigger Edge Select Registers (TES0 and TES1) provide 2 bits per interrupt for selecting the active edge. In addition, a filter can be enabled so that spikes which may exist in a noisy environment will be ignored. The Filter Enable Registers (FEN0 and FEN1) provide 2 bits per interrupt to select the on/off function of the filter and its filter time. The duration of the filter time can be either 1, 2 or 3 clock cycles. Bit SRR is cleared by hardware when the service request is acknowledged.

With the filter enabled, a service request will be flagged only if the interrupt input signal remains stable for at least *n* clocks after an active edge has been detected. The value of *n* is as specified in the register FEN0/1. As an example, at 48MHz (slow FPI clock) and with *n*=3, a filter time of about 60ns will be achieved. In applications where little or no

noise is expected from the interrupt pins, the filters can be disabled. **Figure 15-5** illustrates the block diagram.

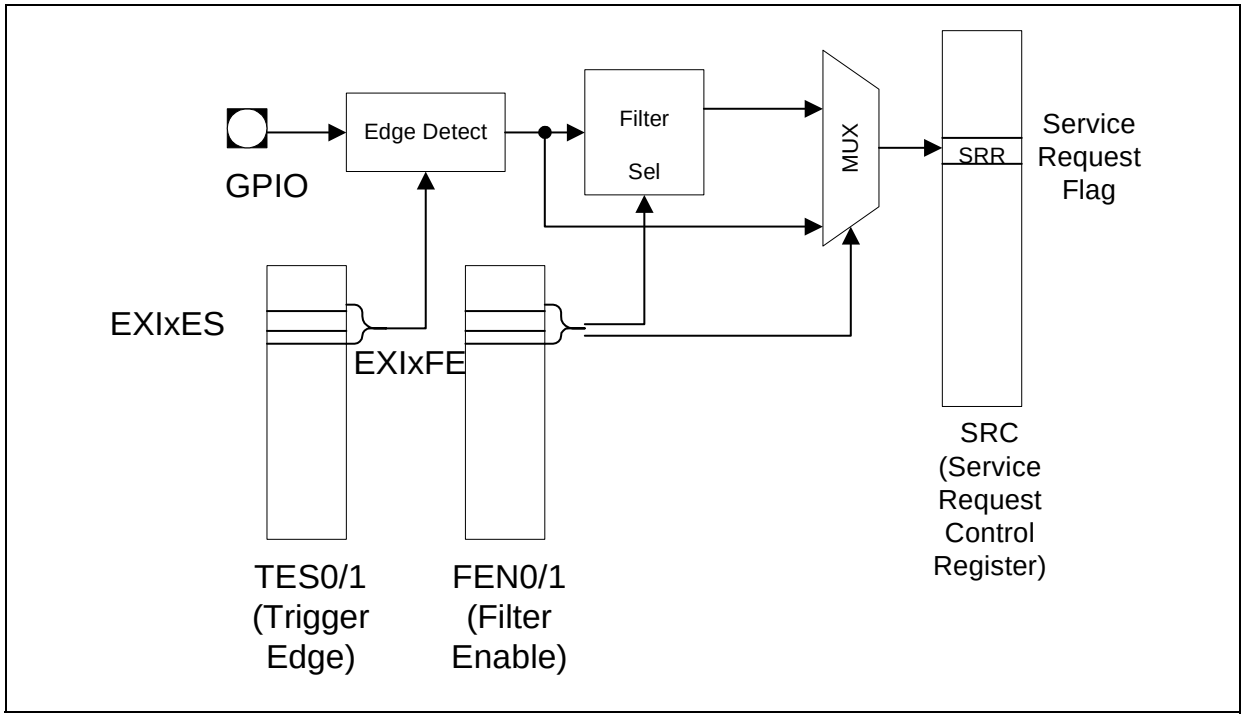


Figure 15-5 Overview of External Interrupt Handling

15.10.1 Register Description

The 24 Service Request Control Registers for the external interrupts have the same format. Each SRN can be set or reset by software via two software-initiated service request control bits. The address map for registers dedicated to external interrupt handling is listed in **Table 15-2**.

Table 15-2 Address Map for External Interrupt Registers

Register Short Name	Register Long Name	Address	Description
EINT_SRC23	Service Request Control Reg. for Ext. Interrupt 23	F000 0C7C$_H$	**Page 15-29**
EINT_SRC22	Service Request Control Reg. for Ext. Interrupt 22	F000 0C78$_H$	**Page 15-29**
EINT_SRC21	Service Request Control Reg. for Ext. Interrupt 21	F000 0C74$_H$	**Page 15-29**
EINT_SRC20	Service Request Control Reg. for Ext. Interrupt 20	F000 0C70$_H$	**Page 15-29**

Table 15-2 Address Map for External Interrupt Registers (cont'd)

Register Short Name	Register Long Name	Address	Description
EINT_SRC19	Service Request Control Reg. for Ext. Interrupt 19	F000 0C6C$_H$	Page 15-29
EINT_SRC18	Service Request Control Reg. for Ext. Interrupt 18	F000 0C68$_H$	Page 15-29
EINT_SRC17	Service Request Control Reg. for Ext. Interrupt 17	F000 0C64$_H$	Page 15-29
EINT_SRC16	Service Request Control Reg. for Ext. Interrupt 16	F000 0C60$_H$	Page 15-29
EINT_SRC15	Service Request Control Reg. for Ext. Interrupt 15	F000 0C5C$_H$	Page 15-29
EINT_SRC14	Service Request Control Reg. for Ext. Interrupt 14	F000 0C58$_H$	Page 15-29
EINT_SRC13	Service Request Control Reg. for Ext. Interrupt 13	F000 0C54$_H$	Page 15-29
EINT_SRC12	Service Request Control Reg. for Ext. Interrupt 12	F000 0C50$_H$	Page 15-29
EINT_SRC11	Service Request Control Reg. for Ext. Interrupt 11	F000 0C4C$_H$	Page 15-29
EINT_SRC10	Service Request Control Reg. for Ext. Interrupt 10	F000 0C48$_H$	Page 15-29
EINT_SRC9	Service Request Control Reg. for Ext. Interrupt 9	F000 0C44$_H$	Page 15-29
EINT_SRC8	Service Request Control Reg. for Ext. Interrupt 8	F000 0C40$_H$	Page 15-29
EINT_SRC7	Service Request Control Reg. for Ext. Interrupt 7	F000 0C3C$_H$	Page 15-29
EINT_SRC6	Service Request Control Reg. for Ext. Interrupt 6	F000 0C38$_H$	Page 15-29
EINT_SRC5	Service Request Control Reg. for Ext. Interrupt 5	F000 0C34$_H$	Page 15-29
EINT_SRC4	Service Request Control Reg. for Ext. Interrupt 4	F000 0C30$_H$	Page 15-29

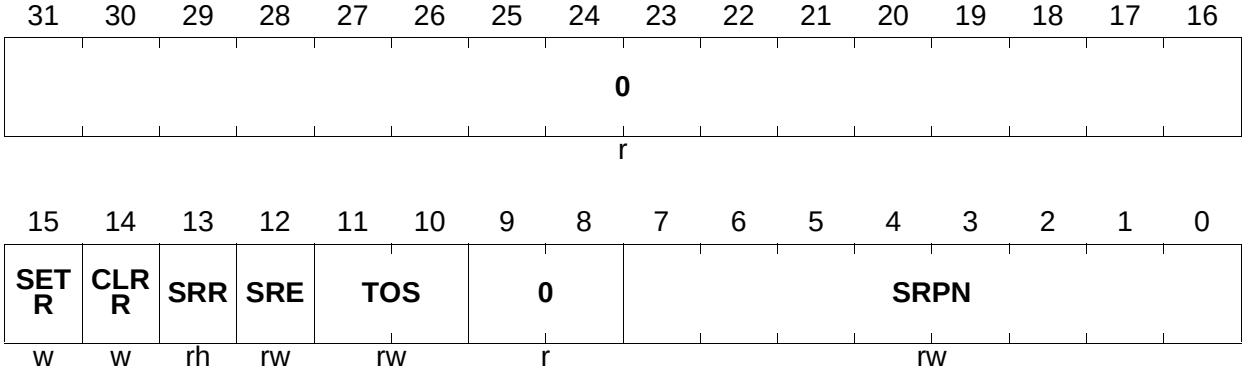
Table 15-2 Address Map for External Interrupt Registers (cont'd)

Register Short Name	Register Long Name	Address	Description
EINT_SRC3	Service Request Control Reg. for Ext. Interrupt 3	F000 0C2C$_H$	**Page 15-29**
EINT_SRC2	Service Request Control Reg. for Ext. Interrupt 2	F000 0C28$_H$	**Page 15-29**
EINT_SRC1	Service Request Control Reg. for Ext. Interrupt 1	F000 0C24$_H$	**Page 15-29**
EINT_SRC0	Service Request Control Reg. for Ext. Interrupt 0	F000 0C20$_H$	**Page 15-29**
FEN1	Filter Enable Register 1	F000 0C1C$_H$	**Page 15-32**
FEN0	Filter Enable Register 0	F000 0C18$_H$	**Page 15-31**
TES1	Trigger Edge Select Register 1	F000 0C14$_H$	**Page 15-30**
TES0	Trigger Edge Select Register 0	F000 0C10$_H$	**Page 15-30**

The general form of the Service Request Control Register is shown below.

EINT_SRC0-23
Service Request Control Register for External Interrupt 0-23

Reset Values: 0000 0000$_H$



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**
0	[9:8], [31:16]	r	**Reserved**

External interrupts are triggered only on transitions of the interrupt request inputs. Depending on the settings of the TES0/1 registers, rising or falling edges or both can trigger an interrupt request. The registers TES0 and TES1 are defined as follows.

TES0
Trigger Edge Select Register 0 **Reset Values: 0000 0000$_H$**

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
EXI15ES	**EXI14ES**	**EXI13ES**	**EXI12ES**	**EXI11ES**	**EXI10ES**	**EXI9ES**	**EXI8ES**
rw	rw	rw	rw	rw	rw	rw	rw

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
EXI7ES	**EXI6ES**	**EXI5ES**	**EXI4ES**	**EXI3ES**	**EXI2ES**	**EXI1ES**	**EXI0ES**
rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
EXIxES (x=0,1...15)	[1:0] ... [31:30]	rw	**External Interrupt Edge Selection Field** 00_H No interrupt is triggered (default) 01_H Interrupt is triggered on positive edge (rising) 10_H Interrupt is triggered on negative edge (falling) 11_H Interrupt is triggered on both edges (rising and falling)

TES1
Trigger Edge Select Register 1 **Reset Values: 0000 0000$_H$**

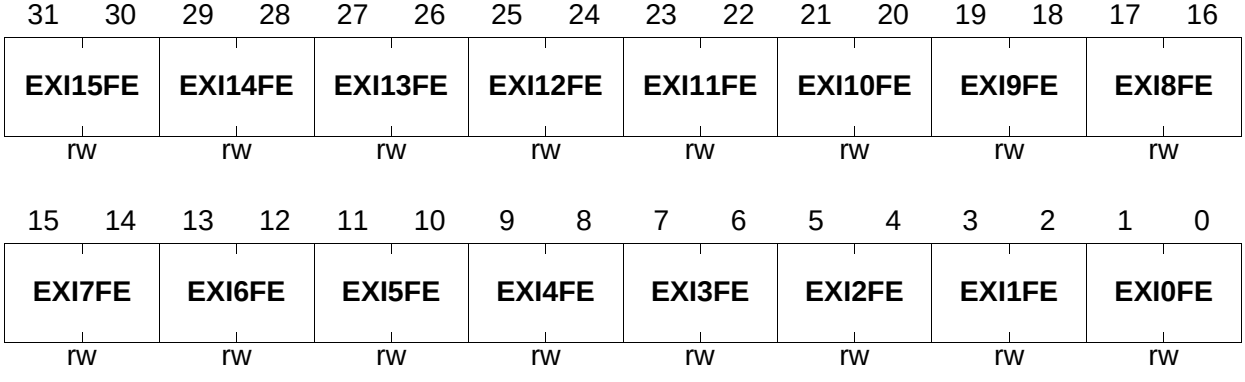
31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
0							
r							

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
EXI23ES	**EXI22ES**	**EXI21ES**	**EXI20ES**	**EXI19ES**	**EXI18ES**	**EXI17ES**	**EXI16ES**
rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
EXIxES (x=16,17...23)	[1:0] ... [15:14]	rw	**External Interrupt Edge Selection Field** 00_H No interrupt is triggered (default) 01_H Interrupt is triggered on positive edge (rising) 10_H Interrupt is triggered on negative edge (falling) 11_H Interrupt is triggered on both edges (rising and falling)

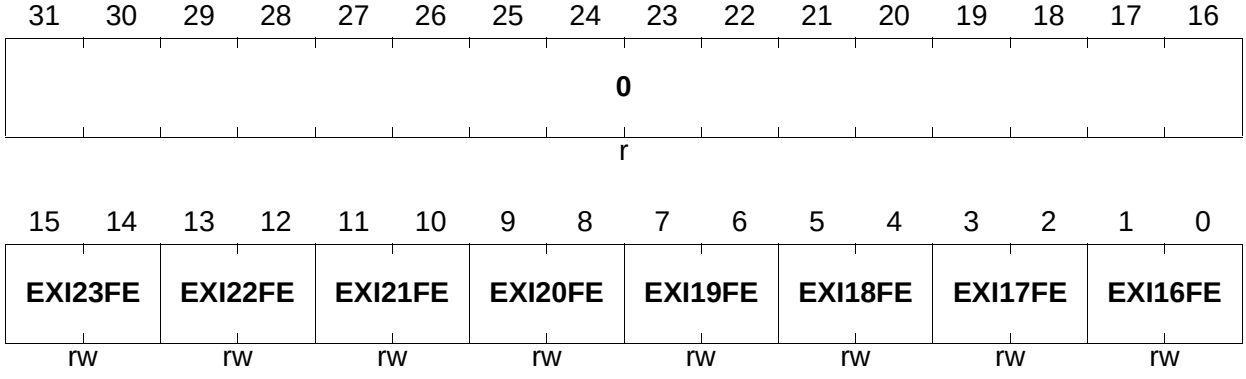
The effect of noise on the external interrupt request inputs can be minimized if a filter is enabled. By setting the registers FEN0/1, the duration of the filter time – from 1 clock to 3 clocks – can be selected. The registers FEN0 and FEN1 are defined as follows.

FEN0
Filter Enable Register 0 **Reset Values: 0000 0000$_H$**



Field	Bits	Type	Description
EXIxFE (x=0,1...15)	[1:0] ... [31:30]	rw	**External Interrupt Filter Enable Field** 00_H Filter is disabled (default) 01_H Filter is enabled, Filter time is 1 clock cycle. 10_H Filter is enabled, Filter time is 2 clock cycle. 11_H Filter is enabled, Filter time is 3 clock cycle.

FEN1
Filter Enable Register 1 **Reset Values: 0000 0000$_H$**



Field	Bits	Type	Description
EXIxES (x=16,17...23)	[1:0] ... [15:14]	rw	**External Interrupt Filter Enable Field** 00_H Filter is disabled (default) 01_H Filter is enabled, Filter time is 1 clock cycle. 10_H Filter is enabled, Filter time is 2 clock cycle. 11_H Filter is enabled, Filter time is 3 clock cycle.

15.11 PCI Interrupts

15.11.1 PCI Interrupt

There is a Service Request Node (SRN) dedicated for supporting PCI interrupts in the TC11IB. Initiating of each individual PCI interrupt is done in the PCI block itself. When any of these interrupts is active, an interrupt request signal is asserted. This signal, generated with a 96 MHz clock, must be asserted for at least 2 clocks. It must be inactive for a minimum of 2 clocks before it can be asserted again for the next interrupt request.

If an active interrupt request is detected, then the service request flag (bit SRR in register PCI_SRC) is set. This bit is cleared by hardware when the interrupt request is serviced. An overview of the PCI interrupt handling is shown in **Figure 15-6**.

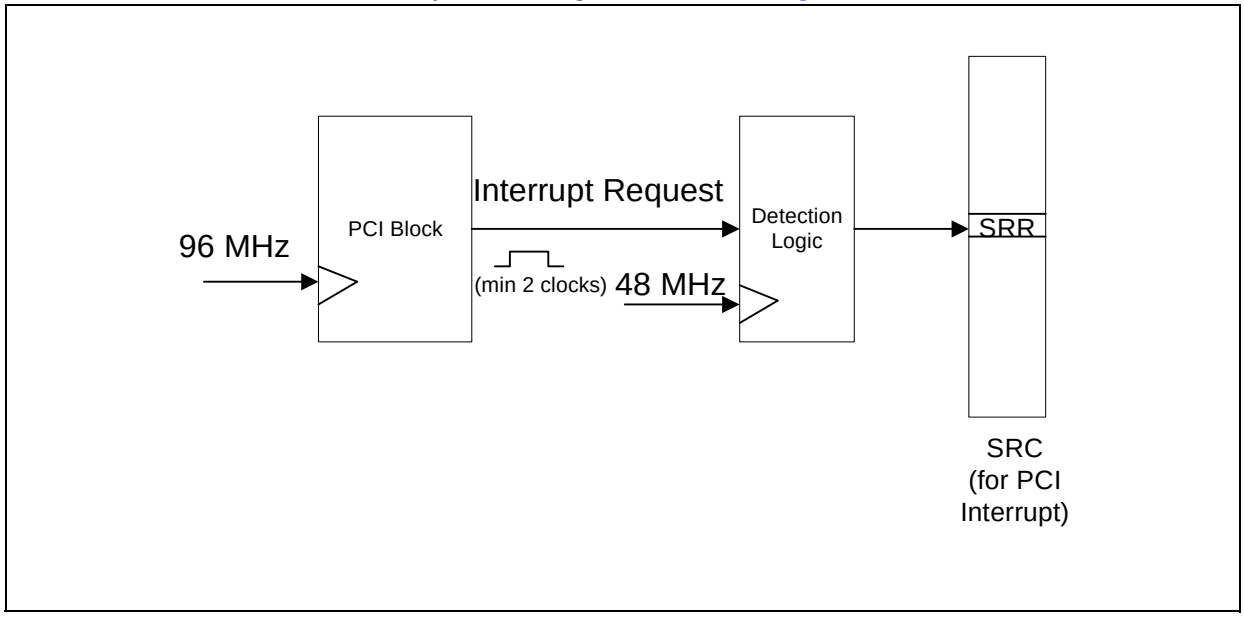


Figure 15-6 Overview of PCI Interrupt Handling

15.11.2 PCI Software Interrupt

In addition, the TC11IB has 32 Service Request Nodes (SRNs) to support the PCI software initiated interrupts. Each of these interrupt is assigned a unique FPI-address. A write instruction to such an address is decoded to set the corresponding service request flag (bit SRR) in the SRN. The data in the write instruction is irrelevant. This scheme is illustrated in **Figure 15-7**.

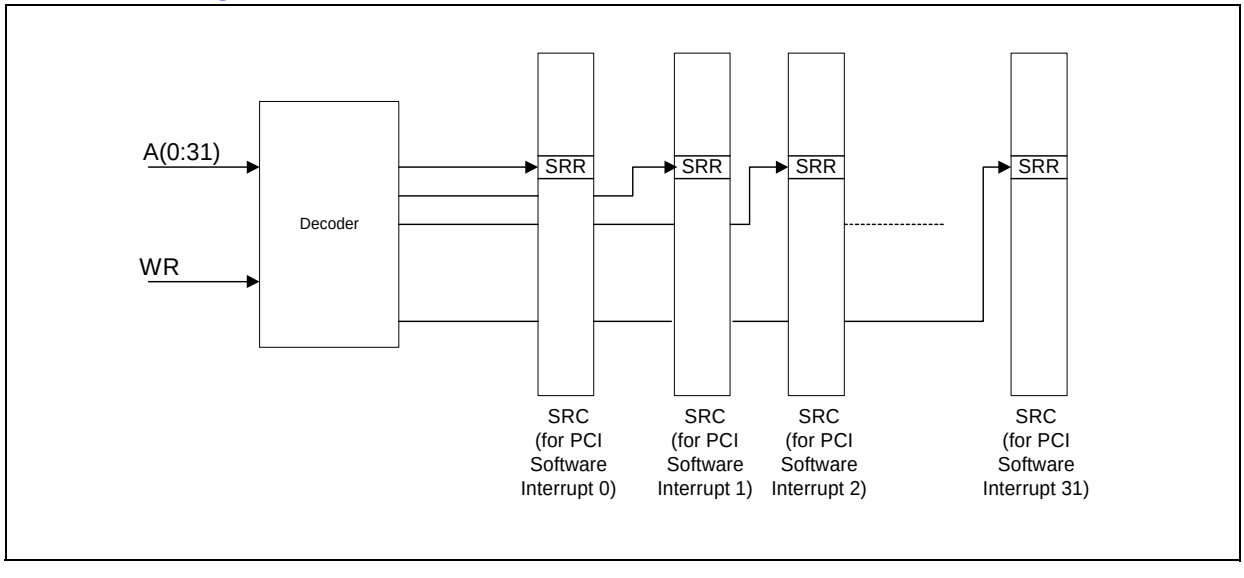


Figure 15-7 PCI Software Initiated Interrupts

The assignment of the PCI software interrupts is listed in **Table 15-3**. These addresses reside within the SCU address space.

Table 15-3 Address Assignment for PCI Software Interrupt Request Registers

Short Name	Description	Address
PCI_SW_IRQ31	PCI Software Interrupt Request 31	F000 01FC$_H$
PCI_SW_IRQ30	PCI Software Interrupt Request 30	F000 01F8$_H$
PCI_SW_IRQ29	PCI Software Interrupt Request 29	F000 01F4$_H$
PCI_SW_IRQ28	PCI Software Interrupt Request 28	F000 01F0$_H$
PCI_SW_IRQ27	PCI Software Interrupt Request 27	F000 01EC$_H$
PCI_SW_IRQ26	PCI Software Interrupt Request 26	F000 01E8$_H$
PCI_SW_IRQ25	PCI Software Interrupt Request 25	F000 01E4$_H$
PCI_SW_IRQ24	PCI Software Interrupt Request 24	F000 01E0$_H$
PCI_SW_IRQ23	PCI Software Interrupt Request 23	F000 01DC$_H$
PCI_SW_IRQ22	PCI Software Interrupt Request 22	F000 01D8$_H$
PCI_SW_IRQ21	PCI Software Interrupt Request 21	F000 01D4$_H$
PCI_SW_IRQ20	PCI Software Interrupt Request 20	F000 01D0$_H$
PCI_SW_IRQ19	PCI Software Interrupt Request 19	F000 01CC$_H$
PCI_SW_IRQ18	PCI Software Interrupt Request 18	F000 01C8$_H$
PCI_SW_IRQ17	PCI Software Interrupt Request 17	F000 01C4$_H$
PCI_SW_IRQ16	PCI Software Interrupt Request 16	F000 01C0$_H$
PCI_SW_IRQ15	PCI Software Interrupt Request 15	F000 01BC$_H$
PCI_SW_IRQ14	PCI Software Interrupt Request 14	F000 01B8$_H$
PCI_SW_IRQ13	PCI Software Interrupt Request 13	F000 01B4$_H$
PCI_SW_IRQ12	PCI Software Interrupt Request 12	F000 01B0$_H$
PCI_SW_IRQ11	PCI Software Interrupt Request 11	F000 01AC$_H$
PCI_SW_IRQ10	PCI Software Interrupt Request 10	F000 01A8$_H$
PCI_SW_IRQ9	PCI Software Interrupt Request 9	F000 01A4$_H$
PCI_SW_IRQ8	PCI Software Interrupt Request 8	F000 01A0$_H$
PCI_SW_IRQ7	PCI Software Interrupt Request 7	F000 019C$_H$
PCI_SW_IRQ6	PCI Software Interrupt Request 6	F000 0198$_H$
PCI_SW_IRQ5	PCI Software Interrupt Request 5	F000 0194$_H$
PCI_SW_IRQ4	PCI Software Interrupt Request 4	F000 0190$_H$

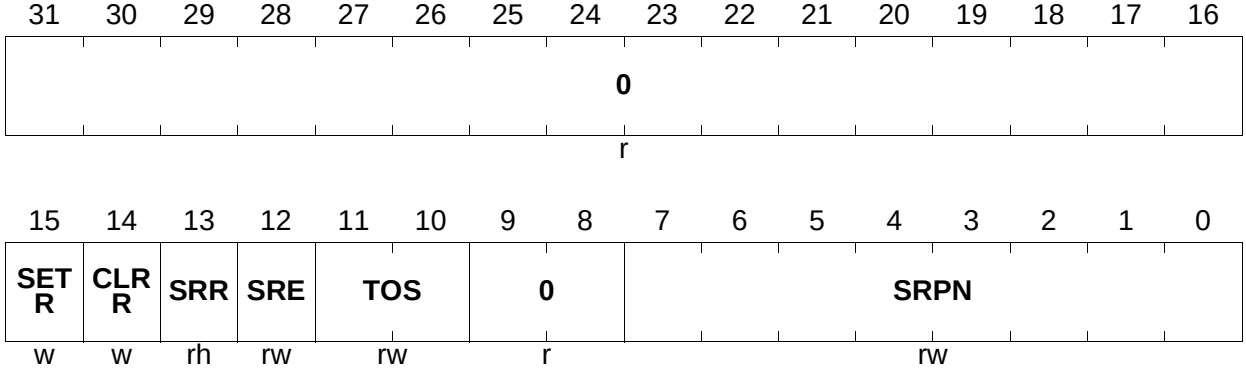
Table 15-3 Address Assignment for PCI Software Interrupt Request Registers

Short Name	Description	Address
PCI_SW_IRQ3	PCI Software Interrupt Request 3	F000 018C$_H$
PCI_SW_IRQ2	PCI Software Interrupt Request 2	F000 0188$_H$
PCI_SW_IRQ1	PCI Software Interrupt Request 1	F000 0184$_H$
PCI_SW_IRQ0	PCI Software Interrupt Request 0	F000 0180$_H$

15.11.3 PCI SRC Registers

The general form of the service Request Control Registers is shown below.

PCI_SW_SRC0-31
Service Request Control Register for PCI Software Interrupt 0-31
PCI_SRC
Service Request Control Register for PCI Interrupt **Reset Values: 0000 0000$_H$**



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**
0	[9:8], [31:16]	r	**Reserved**

15.12 Fast FPI Bus Control Unit Interrupt (BCU0)

The TC11IB specially implemented a pulse detection block which samples the interrupt request input of the Bus Control Unit for Fast FPI (BCU0). This pulse detection logic is clocked at 48 MHz (slow FPI clock) while the interrupt request input from BCU0 is generated at 96 MHz (fast FPI clock). To ensure that no interrupt request is missed, the interrupt request input signal must be asserted for at least 2 cycles of the 96MHz clock. This signal must remain inactive for at least 2 clocks before it can be asserted again.

Figure 15-8 illustrates the handling of the BCU0 interrupt request. **Figure 15-9** shows the waveform of the interrupt request detection. If the request is maintained for at least 2 cycles of the fast FPI clock, then it can be detected on the rising edge of the slow FPI clock.

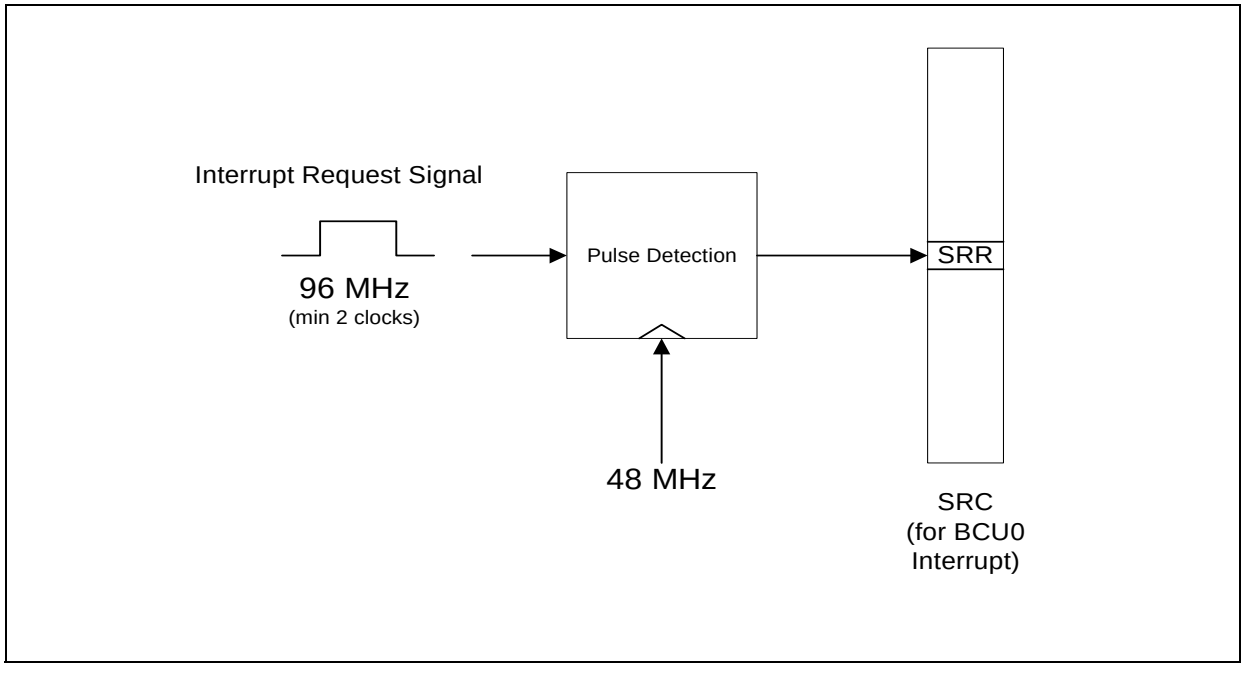


Figure 15-8 Overview of BCU0 Interrupt Handling

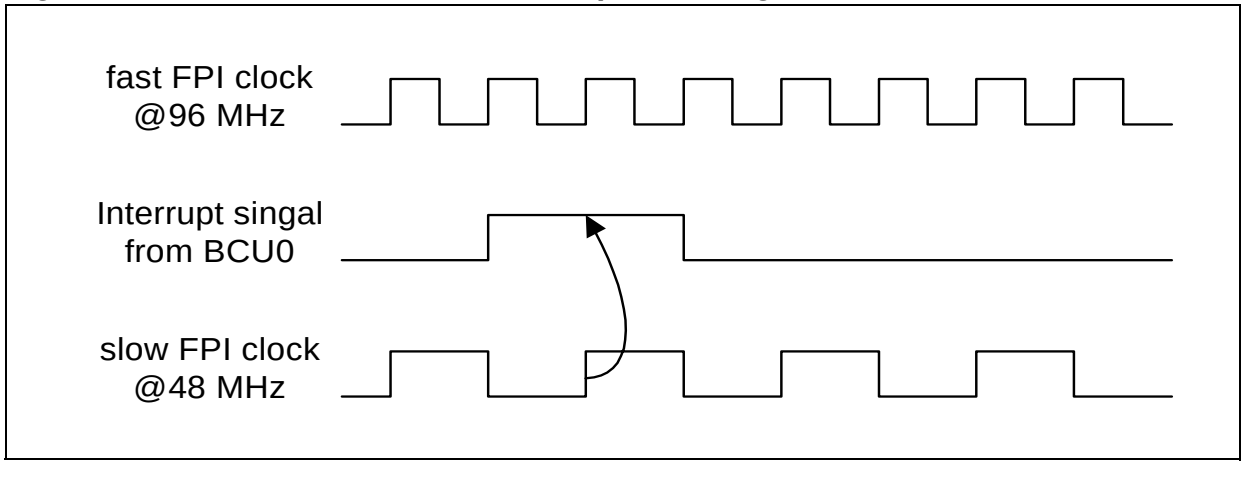
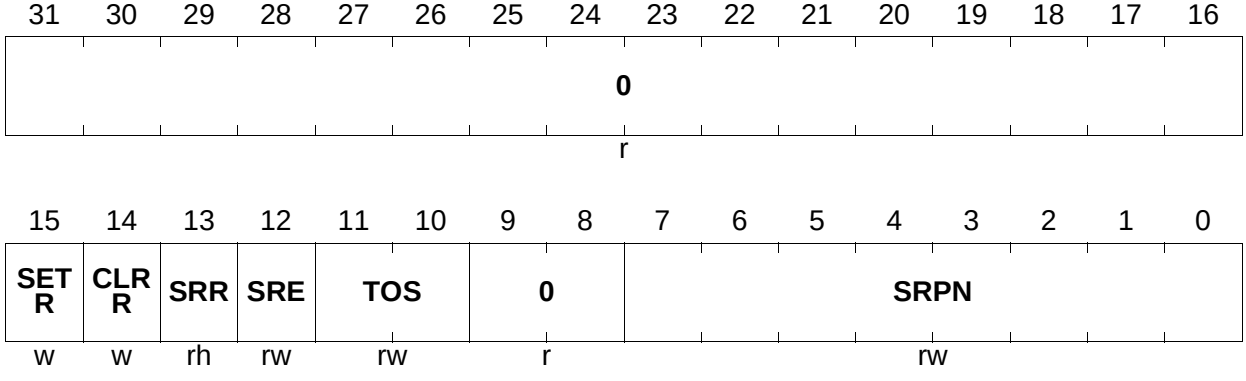


Figure 15-9 BCU0 Interrupt Request Detection

15.12.1 BCU0 SRC Register

The general form of the service Request Control Registers is shown below.

BCU0_SRC
Service Request Control Register for BCU0 Interrupt Reset Values: 0000 0000$_H$



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**
0	[9:8], [31:16]	r	**Reserved**

15.13 Ethernet Controller Interrupts

A similar pulse detection logic to the ones used for PCI and BCU0 interrupts, is implemented in the TC11IB, to support the Ethernet Controller interrupts. There are altogether nine interrupt signals from the Ethernet block.

Figure 15-10 illustrates the handling of Ethernet interrupt requests. The request signal is generated based on the 96MHz clock; while the pulse detection logic is clocked by the 48MHz clock. To ensure that no interrupt request is missed, the request signal must be asserted for at least 2 cycles of the 96MHz clock. The request signal must remain inactive for at least 2 clocks before it can be asserted again.

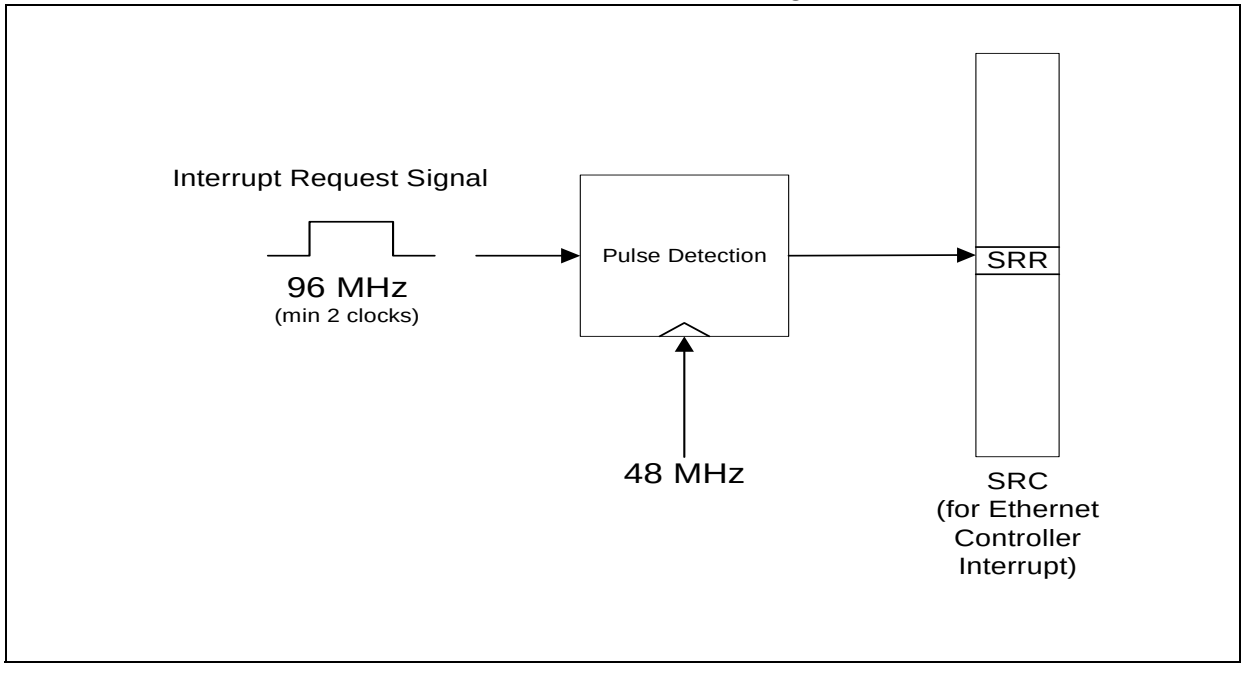


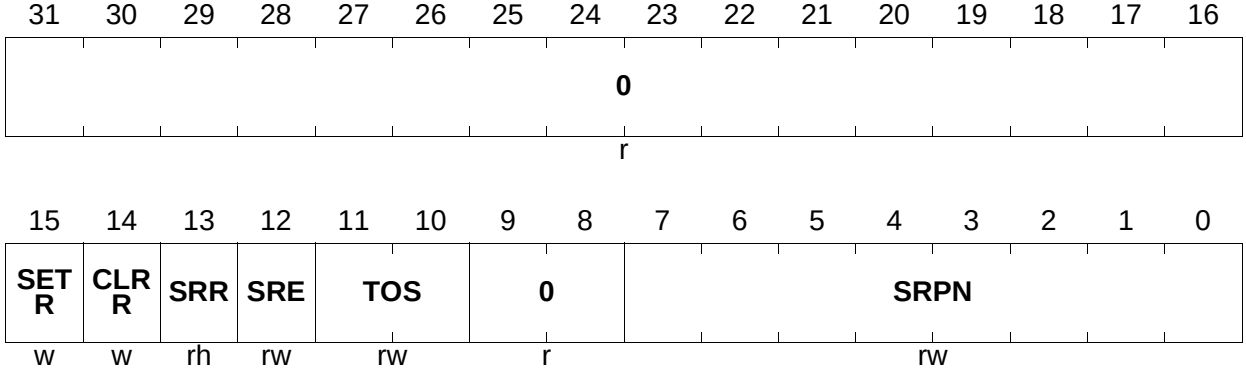
Figure 15-10 Overview of Ethernet Interrupt Handling

15.13.1 Ethernet Controller SRC Registers

The general form of the service Request Control Registers is shown below.

Ethernet_MACTX0SRC
MAC TX 0 Service Request Control Register
Ethernet_MACRX0SRC
MAC RX 0 Service Request Control Register
Ethernet_MACTX1SRC
MAC TX 1Service Request Control Register
Ethernet_MACRX1SRC
MAC RX 1 Service Request Control Register
Ethernet_RBSRC0
RB Service Request Control 0 Register
Ethernet_RBSRC1
RB Service Request Control 1 Register
Ethernet_TBSRC
TB Service Request Control Register
Ethernet_DRSRC
DR Service Request Control Register
Ethernet_DTSRC
DT Service Request Control Register

Reset Values: 0000 0000$_H$



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**

Field	Bits	Type	Description
0	[9:8], [31:16]	r	**Reserved**

15.14 Service Request Register Table

Table 15-4 shows all SRN registers with their short and long name.

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
Bus Control Unit 0 (BCU0)		
BCU0_SRC	Service Request Control Register for BCU0 Interrupt	F000 0C04$_H$
Bus Control Unit 1 (BCU1)		
S_BCU_SRC	BCU1 Service Request Control Register	F000 02FC$_H$
General Purpose Timer Unit 0 (GPTU0)		
GPTU0_SRC7	GPTU0 Service Request Control Register 7	F000 06E0$_H$
GPTU0_SRC6	GPTU0 Service Request Control Register 6	F000 06E4$_H$
GPTU0_SRC5	GPTU0 Service Request Control Register 5	F000 06E8$_H$
GPTU0_SRC4	GPTU0 Service Request Control Register 4	F000 06EC$_H$
GPTU0_SRC3	GPTU0 Service Request Control Register 3	F000 06F0$_H$
GPTU0_SRC2	GPTU0 Service Request Control Register 2	F000 06F4$_H$
GPTU0_SRC1	GPTU0 Service Request Control Register 1	F000 06F8$_H$
GPTU0_SRC0	GPTU0 Service Request Control Register 0	F000 06FC$_H$
General Purpose Timer Unit 1(GPTU1)		
GPTU1_SRC7	GPTU1 Service Request Control Register 7	F000 07E0$_H$
GPTU1_SRC6	GPTU1 Service Request Control Register 6	F000 07E4$_H$
GPTU1_SRC5	GPTU1 Service Request Control Register 5	F000 07E8$_H$
GPTU1_SRC4	GPTU1 Service Request Control Register 4	F000 07EC$_H$
GPTU1_SRC3	GPTU1 Service Request Control Register 3	F000 07F0$_H$
GPTU1_SRC2	GPTU1 Service Request Control Register 2	F000 07F4$_H$
GPTU1_SRC1	GPTU1 Service Request Control Register 1	F000 07F8$_H$
GPTU1_SRC0	GPTU1 Service Request Control Register 0	F000 07FC$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
Asynchronous Serial Channels (ASC)		
ASC_TSRC	ASC Transmit Interrupt Service Request Control Register	F000 08F0$_H$
ASC_RSRC	ASC Receive Interrupt Service Request Control Register	F000 08F4$_H$
ASC_ESRC	ASC Error Interrupt Service Request Control Register	F000 08F8$_H$
ASC_TBSRC	ASC Transmit Buffer Interrupt Service Req. Control Register	F000 08FC$_H$
Asynchronous Serial Interface (16X50)		
16X50_SRC	16X50 Interrupt Service Request Control Register	F000 09FC$_H$
Synchronous Serial Channels (SSC)		
SSC_TSRC	SSC Transmit Interrupt Service Request Control Register	F000 0AF4$_H$
SSC_RSRC	SSC Receive Interrupt Service Request Control Register	F000 0AF8$_H$
SSC_ESRC	SSC Error Interrupt Service Request Control Register	F000 0AFC$_H$
MultiMediaCard Interface (MMCI)		
MMCI_SRC	MMCI Interface Interrupt Service Request Control Register	F000 0BFC$_H$
External Interrupts		
EINT_SRC0	Service Request Control Register for External Interrupt 0	F000 0C20$_H$
EINT_SRC1	Service Request Control Register for External Interrupt 1	F000 0C24$_H$
EINT_SRC2	Service Request Control Register for External Interrupt 2	F000 0C28$_H$
EINT_SRC3	Service Request Control Register for External Interrupt 3	F000 0C2C$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
EINT_SRC4	Service Request Control Register for External Interrupt 4	F000 0C30$_H$
EINT_SRC5	Service Request Control Register for External Interrupt 5	F000 0C34$_H$
EINT_SRC6	Service Request Control Register for External Interrupt 6	F000 0C38$_H$
EINT_SRC7	Service Request Control Register for External Interrupt 7	F000 0C3C$_H$
EINT_SRC8	Service Request Control Register for External Interrupt 8	F000 0C40$_H$
EINT_SRC9	Service Request Control Register for External Interrupt 9	F000 0C44$_H$
EINT_SRC10	Service Request Control Register for External Interrupt 10	F000 0C48$_H$
EINT_SRC11	Service Request Control Register for External Interrupt 11	F000 0C4C$_H$
EINT_SRC12	Service Request Control Register for External Interrupt 12	F000 0C50$_H$
EINT_SRC13	Service Request Control Register for External Interrupt 13	F000 0C54$_H$
EINT_SRC14	Service Request Control Register for External Interrupt 14	F000 0C58$_H$
EINT_SRC15	Service Request Control Register for External Interrupt 15	F000 0C5C$_H$
EINT_SRC16	Service Request Control Register for External Interrupt 16	F000 0C60$_H$
EINT_SRC17	Service Request Control Register for External Interrupt 17	F000 0C64$_H$
EINT_SRC18	Service Request Control Register for External Interrupt 18	F000 0C68$_H$
EINT_SRC19	Service Request Control Register for External Interrupt 19	F000 0C6C$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
EINT_SRC20	Service Request Control Register for External Interrupt 20	F000 0C70$_H$
EINT_SRC21	Service Request Control Register for External Interrupt 21	F000 0C74$_H$
EINT_SRC22	Service Request Control Register for External Interrupt 22	F000 0C78$_H$
EINT_SRC23	Service Request Control Register for External Interrupt 23	F000 0C7C$_H$
PCI Interface (PCI)		
PCI_SRC	Service Request Control Register for PCI Interrupt	F000 0C0C$_H$
PCI_SW_SRC0	Service Request Control Register for PCI Software Interrupt 0	F000 0C80$_H$
PCI_SW_SRC1	Service Request Control Register for PCI Software Interrupt 1	F000 0C84$_H$
PCI_SW_SRC2	Service Request Control Register for PCI Software Interrupt 2	F000 0C88$_H$
PCI_SW_SRC3	Service Request Control Register for PCI Software Interrupt 3	F000 0C8C$_H$
PCI_SW_SRC4	Service Request Control Register for PCI Software Interrupt 4	F000 0C90$_H$
PCI_SW_SRC5	Service Request Control Register for PCI Software Interrupt 5	F000 0C94$_H$
PCI_SW_SRC6	Service Request Control Register for PCI Software Interrupt 6	F000 0C98$_H$
PCI_SW_SRC7	Service Request Control Register for PCI Software Interrupt 7	F000 0C9C$_H$
PCI_SW_SRC8	Service Request Control Register for PCI Software Interrupt 8	F000 0CA0$_H$
PCI_SW_SRC9	Service Request Control Register for PCI Software Interrupt 9	F000 0CA4$_H$
PCI_SW_SRC10	Service Request Control Register for PCI Software Interrupt 10	F000 0CA8$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
PCI_SW_SRC11	Service Request Control Register for PCI Software Interrupt 11	F000 0CAC$_H$
PCI_SW_SRC12	Service Request Control Register for PCI Software Interrupt 12	F000 0CB0$_H$
PCI_SW_SRC13	Service Request Control Register for PCI Software Interrupt 13	F000 0CB4$_H$
PCI_SW_SRC14	Service Request Control Register for PCI Software Interrupt 14	F000 0CB8$_H$
PCI_SW_SRC15	Service Request Control Register for PCI Software Interrupt 15	F000 0CBC$_H$
PCI_SW_SRC16	Service Request Control Register for PCI Software Interrupt 16	F000 0CC0$_H$
PCI_SW_SRC17	Service Request Control Register for PCI Software Interrupt 17	F000 0CC4$_H$
PCI_SW_SRC18	Service Request Control Register for PCI Software Interrupt 18	F000 0CC8$_H$
PCI_SW_SRC19	Service Request Control Register for PCI Software Interrupt 19	F000 0CCC$_H$
PCI_SW_SRC20	Service Request Control Register for PCI Software Interrupt 20	F000 0CD0$_H$
PCI_SW_SRC21	Service Request Control Register for PCI Software Interrupt 21	F000 0CD4$_H$
PCI_SW_SRC22	Service Request Control Register for PCI Software Interrupt 22	F000 0CD8$_H$
PCI_SW_SRC23	Service Request Control Register for PCI Software Interrupt 23	F000 0CDC$_H$
PCI_SW_SRC24	Service Request Control Register for PCI Software Interrupt 24	F000 0CE0$_H$
PCI_SW_SRC25	Service Request Control Register for PCI Software Interrupt 25	F000 0CE4$_H$
PCI_SW_SRC26	Service Request Control Register for PCI Software Interrupt 26	F000 0CE8$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
PCI_SW_SRC27	Service Request Control Register for PCI Software Interrupt 27	F000 0CEC$_H$
PCI_SW_SRC28	Service Request Control Register for PCI Software Interrupt 28	F000 0CF0$_H$
PCI_SW_SRC29	Service Request Control Register for PCI Software Interrupt 29	F000 0CF4$_H$
PCI_SW_SRC30	Service Request Control Register for PCI Software Interrupt 30	F000 0CF8$_H$
PCI_SW_SRC31	Service Request Control Register for PCI Software Interrupt 31	F000 0CFC$_H$
Ethernet Controller (Ethernet)		
Ethernet_MACTX0 SRC	MAC TX 0 Service Request Control Register	F000 0D10$_H$
Ethernet_MACRX0 SRC	MAC RX 0 Service Request Control Register	F000 0D14$_H$
Ethernet_MACTX1 SRC	MAC TX 1 Service Request Control Register	F000 0D18$_H$
Ethernet_MACRX1 SRC	MAC RX 1 Service Request Control Register	F000 0D1C$_H$
Ethernet_RBSRC0	RB Service Request Control 0 Register	F000 0D20$_H$
Ethernet_RBSRC1	RB Service Request Control 1 Register	F000 0D24$_H$
Ethernet_TBSRC	TB Service Request Control Register	F000 0D28$_H$
Ethernet_DRSRC	DR Service Request Control Register	F000 0D2C$_H$
Ethernet_DTSRC	DT Service Request Control Register	F000 0D30$_H$
Peripheral Control Processor (PCP)		
PCP_SRC11	PCP Service Request Control Register 11	F000 3FD0$_H$
PCP_SRC10	PCP Service Request Control Register 10	F000 3FD4$_H$
PCP_SRC9	PCP Service Request Control Register 9	F000 3FD8$_H$
PCP_SRC8	PCP Service Request Control Register 8	F000 3FDC$_H$
PCP_SRC7	PCP Service Request Control Register 7	F000 3FE0$_H$
PCP_SRC6	PCP Service Request Control Register 6	F000 3FE4$_H$

Table 15-4 Special Function Register Table of Service Request Control Registers

Register Short Name	Register Long Name	Absolute Address
PCP_SRC5	PCP Service Request Control Register 5	F000 3FE8$_H$
PCP_SRC4	PCP Service Request Control Register 4	F000 3FEC$_H$
PCP_SRC3	PCP Service Request Control Register 3	F000 3FF0$_H$
PCP_SRC2	PCP Service Request Control Register 2	F000 3FF4$_H$
PCP_SRC1	PCP Service Request Control Register 1	F000 3FF8$_H$
PCP_SRC0	PCP Service Request Control Register 0	F000 3FFC$_H$
OCDS		
CPU_SBSRC	Software Break Service Request Control Register	F7E0 FFBC$_H$
CPU		
CPU_SRC3	CPU Service Request Control Register 3	F7E0 FFF0$_H$
CPU_SRC2	CPU Service Request Control Register 2	F7E0 FFF4$_H$
CPU_SRC1	CPU Service Request Control Register 1	F7E0 FFF8$_H$
CPU_SRC0	CPU Service Request Control Register 0	F7E0 FFFC$_H$
LCU		
LCU_SRC	LCU Service Request Control Register	F87F FEFC$_H$

16 Trap System

The TC11IB trap system provides a means for the CPU to service conditions that are so critical that they must not be postponed. Such conditions include both catastrophic developments, such as an attempt by the CPU to execute an illegal instruction, as well as routine developments such as system calls. This chapter describes the trap system for the TC11IB. Topics covered include trap types, trap handling, and non-maskable interrupts (NMIs). Traps direct the processor to execute Trap Service Routines (TSR) stored in a Trap Vector Table.

16.1 Trap System Overview

Traps break the normal execution of code, much like interrupts, but traps are different from interrupts in these ways:

- Trap Service Routines (TSR) reside in the Trap Vector Table, which is separate from the Interrupt Vector Table.
- A trap does not change the CPU's interrupt priority, so the ICR.CCPN field is not changed.
- Traps cannot be disabled by software. Traps are always active.
- The return address, saved when a Trap Service Routine is invoked, is the address of the instruction in progress at the moment the trap was raised, whereas the return address of an interrupt is the address of the instruction that would have been executed next if the interrupt had not occurred.

The CPU aborts the instruction in progress when a trap occurs, and forces execution to the appropriate TSR. The TSR decides whether the situation is correctable or not. If not, the TSR takes appropriate action, which may involve aborting the current task, or even resetting the TC11IB. If the situation is routine or correctable, the TSR performs whatever action is necessary, then exits, whereupon the CPU re-executes the previously aborted instruction.

Traps may arise within the CPU, for instance, as a side-effect of the execution of instructions. These traps are typically synchronous with the processor instruction clock. They may also be generated by events external to the CPU, such as a peripheral or external NMI signal. Hardware-generated traps are typically asynchronous with the processor instruction clock.

Traps can signal a variety of routine or serious events. For instance, traps can be used to

- Implement memory protection and virtual memory
- Provide unprivileged applications access to privileged system services
- Manage task-based context-switching
- Respond to urgent external conditions, such as an NMI
- Respond to urgent internal conditions, such as signals from the Watchdog Timer, the LMB Bus, the Fast FPI Bus, the Slow FPI Bus or the PLL
- Detect access to memory by other system components

- Signal events from task to task
- Administer overflow and underflow of hardware tables and lists
- Recover from catastrophic software errors

Many traps arise as a consequence of the execution of instructions:

- The SYSCALL instruction generates a trap that is usually intended to signal a request for system services by an unprivileged application.
- An attempt to execute an illegal instruction opcode produces a trap as a side-effect. The instruction is aborted, and a trap is invoked. This protects a system from poorly-written or damaged programs.
- When an application attempts to execute an unimplemented instruction opcode, the trap that results can invoke a TSR to emulate the operation of that instruction in software, thereby extending the instruction set.
- If an application attempts to access protected memory, the resulting trap may be used by the system to read in pages from memory that the application needs.
- If an arithmetic operation produces an invalid result, a trap is generated. In some cases, the TSR may attempt to correct the result through software, or it may cause the application to terminate.

Other uses of traps include:

- Context management
- Recovery from FPI Bus error signals
- Access to memory by a peripheral
- Handling the Non-Maskable “Interrupt” (actually trap) signal from the external NMI input, from the Watchdog Timer, or from the PLL if it loses stable clock signals

When a hardware trap condition is detected, the processor’s trap control system supplies a two-part number that identifies the cause of the trap. The first part of the number is a three-bit Trap Class Number (TCN); the second part is an eight-bit Trap Identification Number (TIN). The TCN is used to index the Trap Vector Table to identify the proper TSR to handle the trap. The TIN is loaded into register D15 of the TSR’s context to further identify the precise cause of the trap. The TSR must examine the TIN in software.

16.2 Trap Classes

The TC11IB has eight trap classes, as shown in **Table 16-1**. Each trap has a Trap Identification Number (TIN), that identifies the number of the trap within its class. When the CPU hardware goes to service a trap, the TIN is loaded into register D15 before the first instruction of the trap handler is executed. A trap is completely identified by its Trap Class Number (TCN) and its TIN.

Table 16-1 summarizes and classifies all TC11IB-supported traps. In the column “Type”, an “S” stands for a Synchronous trap, while “A” indicates an Asynchronous trap. “SW” and “HW” indicate a Software trap or a Hardware trap, respectively. The column “Saved PC” states which Program Counter value is saved during the trap entry. “ThisPC” indicates that the PC value of the instruction causing the trap is saved, while “NextPC” is the PC value pointing to the instruction which would have been executed next.

Table 16-1 TC11IB Supported Traps

Trap ID (TIN)	Trap Name	Trap Type	Saved PC	Description
Class 0 - MMU (TCN = 0)				
0	MMUVAF	S, HW	ThisPC	MMU: Virtual Address Fill
1	MMUVAP	S, HW	ThisPC	MMU: Virtual Address Protection
Class 1 - Internal Protection Traps (TCN = 1)				
1	PRIV	S, HW	ThisPC	Privileged Instruction
2	MPR	S, HW	ThisPC	Memory Protection: Read Access
3	MPW	S, HW	ThisPC	Memory Protection: Write Access
4	MPX	S, HW	ThisPC	Memory Protection: Execution Access
5	MPP	S, HW	ThisPC	Memory Protection: Peripheral Access
6	MPN	S, HW	ThisPC	Memory Protection: Null Address
7	GRWP	S, HW	ThisPC	Global Register Write Protection
Class 2 - Instruction Errors (TCN = 2)				
1	IOPC	S, HW	ThisPC	Illegal Opcode
4	ALN	S, HW	ThisPC	Data Address Alignment Error
5	MEM	S, HW	ThisPC	Invalid Memory Address
Class 3 - Context Management (TCN = 3)				
1	FCD	S, HW	ThisPC	Free Context List Depleted (FCX==LCX)
2	CDO	S, HW	ThisPC	Call Depth Overflow
3	CDU	S, HW	ThisPC	Call Depth Underflow

Table 16-1 TC11IB Supported Traps (cont'd)

Trap ID (TIN)	Trap Name	Trap Type	Saved PC	Description
4	FCU	S, HW	see Note	Free Context List Underflow (FCX==0)
5	CSU	S, HW	ThisPC	Context List Underflow (PCX==0)
6	CTYP	S, HW	ThisPC	Context Type Error (PCXI.UL is wrong)
7	NEST	S, HW	ThisPC	Nesting Error: RFE with non-zero call depth
Class 4 - System Bus Errors (TCN = 4)				
1	PSE	S, HW	ThisPC	Bus Error on Program Fetch Operation
2	DSE	S, HW	ThisPC	Bus Error on Data Load Operation
3	DAE	A, HW	ThisPC	Bus Error on Data Store Operation
Class 5 - Assertion Traps (TCN = 5)				
1	OVF	S, SW	ThisPC	Arithmetic Overflow
2	SOVF	S, SW	ThisPC	Sticky Arithmetic Overflow
Class 6 - System Call (TCN = 6)				
1)	SYS	S, SW	NextPC	System Call
Class 7 - Non-Maskable Interrupt (TCN = 7)				
0	NMI	A, HW	NextPC	Non-Maskable Interrupt

1) For the system call trap via the SYSCALL instruction, the TIN is created from an immediate constant in the SYSCALL instruction supplied by the calling software. The range of values for this constant is 0 to 255, inclusive.

Note: The normal trap entry mechanism is not used, instead, a jump to the FCU trap handler is performed.

16.2.1 Synchronous Traps

Synchronous traps are associated with the execution, or attempted execution, of processor instructions. The trap is taken immediately and serviced before execution can proceed beyond that instruction (except for the SYSCALL instruction).

16.2.2 Asynchronous Traps

Asynchronous traps are similar to interrupts, in that they are associated with hardware conditions detected externally and signaled back to the processor. Some asynchronous traps result indirectly from instructions that have been executed previously, but the direct association with those instructions has been lost. Others such as the NMI arise strictly from external events.

Note: Due to a missing trap queue in the TriCore architecture, it is possible to lose asynchronous traps (e.g. caused by an FPI Bus write operation) if several traps are generated within a very short time frame.

16.2.3 Hardware Traps

Hardware traps are generated as a result of problems encountered while executing processor instructions. Examples include attempting to execute an illegal instruction opcode, attempting to access protected memory, and attempting to access data memory at a misaligned address.

16.2.4 Software Traps

Software traps are used to make system calls and test assertions in software. For example, a client application can call a privileged system function by executing the SYSCALL instruction, which invokes a TSR to begin executing in privileged mode.

There is a single entry in the Trap Vector Table for the SYSCALL trap. An application executing the SYSCALL instruction must embed a system-defined eight-bit immediate constant in the SYSCALL instruction, which becomes the TIN for the SYSCALL trap. Thus the application can signal its need for specific privileged services.

16.2.5 Trap Descriptions

In this section, each of the traps listed in **Table 16-1** is described in more detail.

MMU Trap

The MMU can generate the following traps:

- VAF (Virtual Address Fill)
- VAP (Virtual Address Protection)

The Virtual Address Fill trap is generated if PTE based translation is required for a virtual address and the PTE corresponding to the translation is missing in the MMU. The Virtual Address Protection trap is generated if the access is disallowed.

PRIV Trap

The PRIV trap is detected in the decode stage of the load/store pipeline. The PRIV trap is generated whenever an attempt is made to execute a protected system instruction in User Mode. The protected system instructions are:

- MTCR
- BISR

A PRIV trap is also taken whenever an attempt is made to execute one of the following instructions in User Mode 0:

- ENABLE
- DISABLE

MPR Trap

Read memory protection violations are detected in the execute stage of the load/store pipeline. The MPR trap is generated for LD/LDMST and SWAP instructions when the memory protection system is enabled and the effective address does not lie within any range with read permissions enabled.

MPW Trap

Write memory protection violations are detected in the execute stage of the load/store pipeline. The MPW trap is generated for ST/LDMST and SWAP instructions when the memory protection system is enabled and the effective address does not lie within any range with read permissions enabled.

MPX Trap

The Execution Access Memory Protection Trap is detected in the fetch stage. The MPX trap is generated when the memory protection system is enabled and the PC does not lie within any range with execute permissions enabled.

MPP Trap

The Peripheral Access Memory Protection Trap is detected in the execute stage of the load/store pipeline. It is generated when either segment 14 or 15 is targeted by any memory operation while the machine is in User Mode 0.

MPN Trap

The Null Address Memory Protection Trap is detected in the execute stage of the load/store pipeline. It is generated when any memory operation targets address 0.

GRWP Trap

The GRWP trap is detected in the decode stage of the load/store pipeline. The GRWP trap is generated whenever an attempt is made to execute an instruction that modifies one of the four global registers, A0, A1, A8 and A9, while the Global Write Enable (PSW_GW) is 0.

IOPC Trap

The IOPC trap can be detected in either the integer or load/store decode stages. The IOPC trap is raised when an invalid instruction is decoded, that is, the instruction in the decode stage does not map onto a known opcode.

ALN Trap

The ALN trap is detected in the execute stage of the load/store pipeline. The trap is raised whenever a memory operation does not conform to the expected memory alignment constraints.

MEM Trap

The MEM trap is detected in the execute stage of the load/store pipeline. The trap is raised whenever an attempt is made to access an invalid memory address such as:

- An effective address that lies in a different segment to the base address
- An address that crosses a segment boundary
- An address range in the DMU or PMU that does not map onto an implemented area of memory
- An address in the Core SFRs (CSFRs)

FCD Trap

The FCD trap is detected in the decode stage of the load/store unit. An FCD trap is raised whenever a save context operation is performed and the Free Context List Pointer (FCX) equals the contents of the Free Context Limit Pointer (LCX).

CDO Trap

The CDO trap is detected in the decode stage of the load/store pipeline. The trap results when a call is attempted and the call-depth limit has been reached (call-depth counter overflow).

CDU Trap

The CDU trap is detected in the decode stage of the load/store pipeline. The trap results when a RET instruction is attempted and the call-depth counter equals 0.

FCU Trap

The Free Context List Underflow Trap is one of the most serious error conditions in the machine. The trap results when a save context operation is performed and the FCX equals 0. This trap is also raised if any error occurs during a context save operation.

The normal trap entry mechanism is not taken, instead a jump to the FCU trap handler is performed.

CSU Trap

The CSU trap is detected in the decode stage of the load/store pipeline. The trap results when a restore-context operation is performed and Previous Context Pointer (PCXI.PCX) equals 0.

CTYP Trap

The CTYP trap is detected in the decode stage of the load/store pipeline. The trap is raised when a context-restore operation is performed on an incorrect context type. That is if a restore lower context is performed when the PCXI.UL = 1, or a restore upper context is performed when the PCXI.UL = 0.

NEST Trap

The Nesting Error Trap (is detected in the decode stage of the load/store pipeline. The NEST trap results when an RFE instruction is attempted and the call depth counter does not equal 0.

PSE Trap

The Program Fetch Synchronous Error Trap is detected in the integer or load/store decode stage. The PSE trap is raised when the fetch of an instruction from the Program Memory Unit (PMU) results in an error (e.g. fetch from a reserved address).

DSE Trap

The Data Load/Store Synchronous Error Trap is detected in the execute stage of the load/store unit. The DSE trap is generated by the DMU on a cache management error, DMU control register access error, FPI Bus access error, or a DMU memory range error. The exact cause of the error can be read in the DMU Synchronous Trap Flag Register, DMU_STR. DSE traps occur in general on load accesses to the DMU.

DAE Trap

The Data Load/Store Asynchronous Error Trap is an asynchronous trap. The DSE trap is generated by the DMU on either a cache management error, DMU control register access error, FPI Bus access error, or a DMU memory range error. The exact cause of the error can be read via the DMU Asynchronous Trap Flag Register, DMU_ATR. DAE traps occur in general on store accesses to the DMU.

OVF Trap

The OVF trap is detected in the execute stage of the load/store pipeline. The trap is raised by the TRAPV instruction when the instruction is executed and the Overflow Flag, PSW.V, is set.

SOVF Trap

The SOVF trap is detected in the execute stage of the load/store pipeline. The trap is raised by the TRAPSV instruction when the instruction is executed and the Sticky Overflow Flag, PSW.SV, is set.

SYS Trap

The SYS trap is detected in the decode stage of the load/store pipeline. The trap is raised implicitly by the SYSCALL instruction. For the system call trap via the SYSCALL instruction, the TIN is created from an immediate constant in the SYSCALL instruction supplied by the calling software. The range of values for this constant is 0 through 255.

NMI Trap

The NMI is an asynchronous trap. The generation of the NMI is handled by the Power-Watchdog-Reset (PWR) block in the system. The source can be the external $\overline{\text{NMI}}$ input, a Watchdog Timer error condition, or a loss of stable clock signal in the PLL.

16.3 Trap Vector Table

The entry-points of all Trap Service Routines are stored in code memory in the Trap Vector Table. The BTV register specifies the base address of the Trap Vector Table in code memory. It can be assigned to any available code memory. Its default on power-up is fixed at A000 0100_H. However, the BTV register can be modified using the MTCR instruction during the initialization phase of the system. With this arrangement, it is possible to have multiple Trap Vector Tables and switch between them by changing the contents of the BTV register.

Note: The BTV register is protected by the ENDINIT bit. An ISYNC instruction should be issued after modifying BTV so as to avoid untoward pipeline behavior.

When a trap event occurs, a trap identifier is generated by the hardware detecting the event. The trap identifier is made up of a Trap Class Number (TCN) and a Trap Identification Number (TIN).

The TCN is left-shifted by five bits and ORed with the address in the BTV register to form the entry address of the TSR. Due to this operation, it is recommended that bits [7:5] of register BTV are set to 0 (see **Figure 16-1**). Note that bit 0 of the BTV register is always 0 and can not be written to (instructions have to be aligned on even byte boundaries).

Left-shifting the TCN by 5 bits creates entries into the Trap Vector Table which are evenly spaced 8 words apart. If a trap handler (TSR) is very short, it may fit entirely within the eight words available in the Trap Vector Table entry. Otherwise, the code at the entry point must ultimately cause a jump to the rest of the TSR residing elsewhere in memory.

Unlike the Interrupt Vector Table, entries in the Trap Vector Table cannot be spanned.

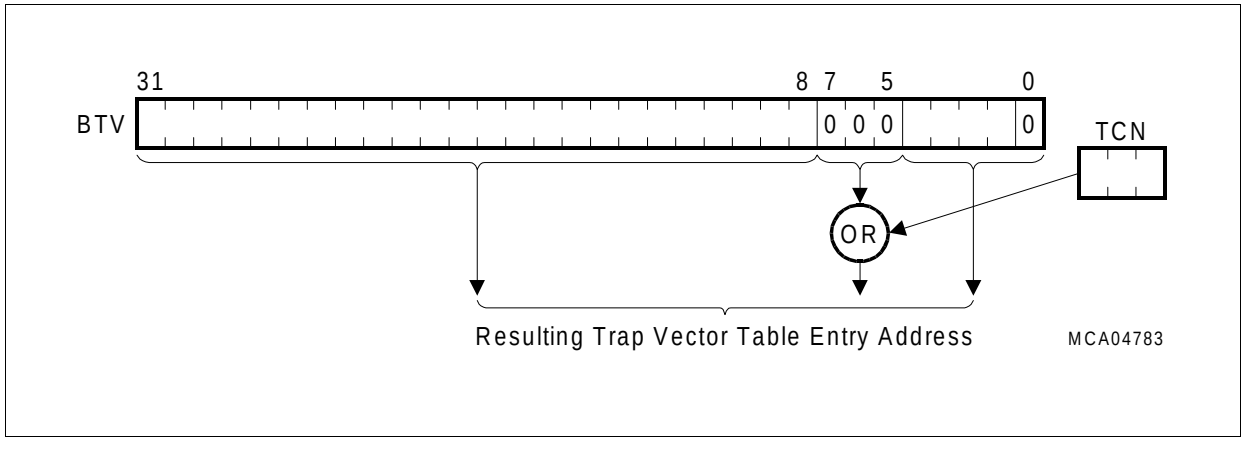


Figure 16-1 Trap Vector Table Entry Address Calculation

16.3.1 Entering a Trap Service Routine

The following actions are performed to enter a TSR when a trap event is detected by the hardware:

1. The upper context of the current task is saved[1)].
2. The interrupt system is globally disabled (ICR.IE = 0).
3. The current CPU priority number (CCPN) is not changed.
4. The PSW is set to a default value:
   – All permissions are enabled: PSW.IO = 10_B
   – Memory protection is switched to PRS 0: PSW.PRS = 00_B.
   – The stack pointer bit is set for using the interrupt stack: PSW.IS = 1.
   – The call-depth counter is cleared, the call depth limit is set for 64: PSW.CDC = 0.
5. The stack pointer, A10, is reloaded with the contents of the Interrupt Stack Pointer, ISP, if the PSW.IS bit of the interrupted routine was set to 0 (using the user stack), otherwise it is left unaltered.
6. The Trap Vector Table is accessed to fetch the first instruction of the TSR. The effective address is the contents of the BTV register ORed with the Trap Class Number (TCN) left-shifted by 5.

Although traps leave the ICR.CCPN unchanged, TSRs still begin execution with interrupts disabled. They can therefore perform critical initial operations without interruption, until they specifically re-enable interrupts.

Since entry into a trap handler is only determined by the TCN, software in the TSR must determine the exact cause of the trap by evaluation of the TIN stored in register D15.

1) If a context-switch trap occurs while the CPU is in the process of saving the upper context of the current task, the pending ISR will not be entered, the interrupt request will be left pending, and the CPU will enter the appropriate trap handling routine instead.

16.4 Non-Maskable Interrupt

Although called an interrupt, the non-maskable interrupt (NMI) is actually serviced as a trap, since it is not interruptible and does not follow the standards for regular interrupts.

In the TC11IB, three different events can generate a NMI trap:

- A transition on the $\overline{\text{NMI}}$ input pin
- An error or wake-up signal from the Watchdog Timer
- The PLL upon loss of external clock stability

The type of an NMI trap is indicates in the NMI Status Register (NMISR).

16.4.1 NMI Status Register

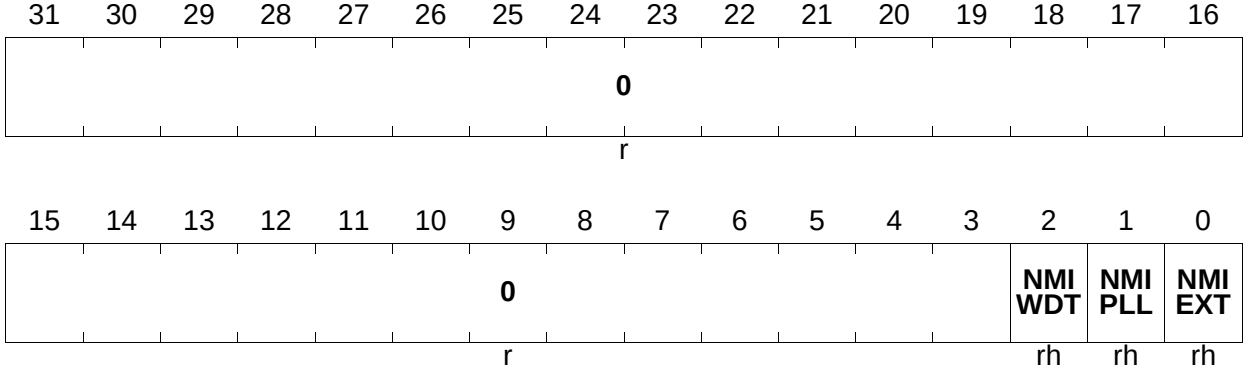
The source of a NMI trap can be identified through three status bits in NMISR. The bits in NMISR are read-only; writing to them has no effect.

The CPU detects a zero-to-one transition of the NMI input signal as indicating an NMI trap event. It then sets NMISR.NMIEXT. If the Watchdog Timer times out, it sets NMISR.NMIWDT. If the PLL loses its clock signal, it sets NMISR.PLL1.

The bits in NMISR are ORed together to generate an NMI trap request to the CPU. If one of the NMISR bits is newly asserted while another bit is set, no new NMI trap request is generated. All flags are cleared automatically after a read of NMISR. Therefore, after reading NMISR, the NMI TSR must check all bits in NMISR to determine whether there have been multiple causes of an NMI trap.

Note: The NMISR register is located in the address range reserved for the System Control Unit (SCU).

NMISR
NMI Status Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
NMIEXT	0	rh	**External NMI Flag** 0 No external NMI request has occurred 1 An external NMI request has been detected
NMIPLL	1	rh	**PLL NMI Flag** 0 No PLL NMI has occurred 1 The PLL has lost the lock to the external crystal
NMIWDT	2	rh	**Watchdog Timer NMI Flag** 0 No watchdog NMI occurred 1 The Watchdog Timer has entered the pre-warning phase due to a watchdog error.
0	[31:3]	r	**Reserved**; read as 0.

16.4.2 External $\overline{\text{NMI}}$ Input

An external NMI event is generated when a one-to-zero transition is detected at the external $\overline{\text{NMI}}$ input pin. NMISR.NMIEXT is set in this case. The $\overline{\text{NMI}}$ pin is sampled at the system clock frequency. A transition is recognized when one sample shows a 1 and the next sample shows a 0. Subsequent 0-samples or a 0-to-1 transition do not trigger any action.

16.4.3 Phase-Locked Loop NMI

The PLL clock generation unit sets the NMIPLL flag when it detects a loss in the synchronization with the external oscillator clock input. This condition means that the PLL clock frequency is no longer stable, and that the PLL will now decrease to its base frequency.

16.4.4 Watchdog Timer NMI

The Watchdog Timer sets the NMIWDT flag for two conditions:

- A Watchdog Timer error has occurred
- Bit 15 of the Watchdog Timer is set while the CPU is in idle mode

A Watchdog Timer error can produce an NMI event because

- Access to register WDT_CON0 was attempted improperly, or
- The Watchdog Timer overflowed either in Time-Out Mode or in Normal Watchdog Timer Mode.

When the CPU is in Idle Mode and the Watchdog Timer is not disabled, an increment of the Watchdog Timer counter from $7FFF_H$ to 8000_H (that is, when bit 15 of the timer is set to 1) sets the NMIWDT bit to wake up the CPU.

17 Peripheral Control Processor

This chapter describes the Peripheral Control Processor (PCP), its architecture, programming model, registers, and instructions.

17.1 Peripheral Control Processor Overview

The Peripheral Control Processor (PCP) performs tasks that would normally be performed by the combination of a DMA controller and its supporting CPU interrupt service routines in a traditional computer system. It could easily be considered as the host processor's first line of defense as an interrupt-handling engine. The PCP can off-load the CPU from having to service time-critical interrupts. This provides many benefits, including:

- Avoiding large interrupt-driven task context-switching latencies in the host processor
- Reducing the cost of interrupts in terms of processor register and memory overhead
- Improving the responsiveness of interrupt service routines to data-capture and data-transfer operations
- Easing the implementation of multitasking operating systems.

The PCP has an architecture that efficiently supports DMA type transactions to and from arbitrary devices and memory addresses within the TC11IB and also has reasonable stand alone computational capabilities.

17.2 PCP Architecture

The PCP is made up of several modular blocks as follows. Please refer to **Figure 17-1**.

- PCP Processor Core
- Code Memory (PCODE)
- Parameter Memory (PRAM)
- PCP Interrupt Control Unit (PICU)
- PCP Service Request Nodes (PSRN)
- System bus interface to the Slow Flexible Peripheral Interface (FPI Bus)

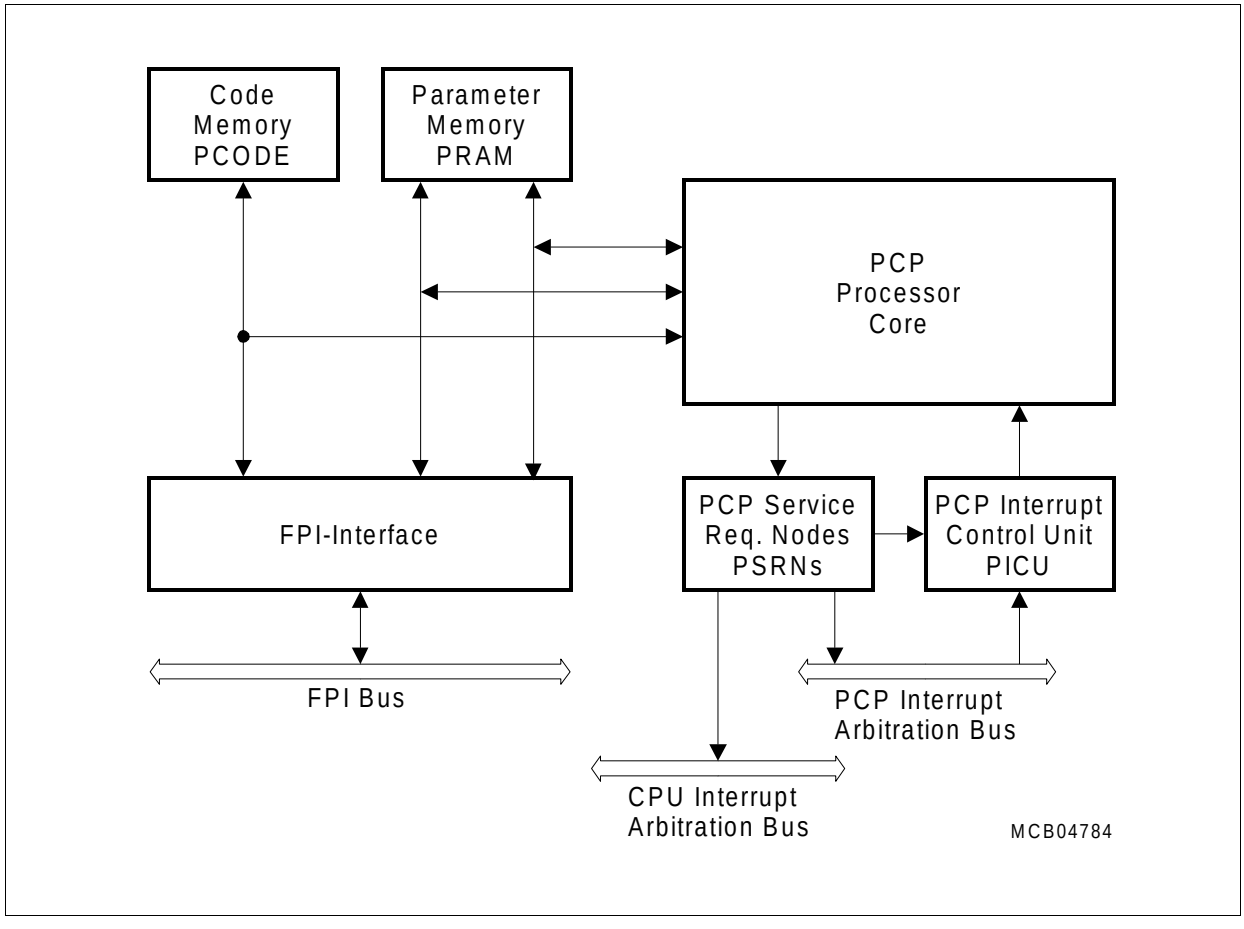


Figure 17-1 PCP Block Diagram

17.2.1 PCP Processor

The PCP Processor is the main engine of the PCP. It contains an instruction pipeline, a set of general purpose registers, an arithmetic/logic unit (ALU), as well as control and status registers and logic. Its instruction set is optimized especially for the tasks it has to perform. **Table 17-1** provides an overview of the PCP instruction set.

The PCP processor core receives service requests from peripherals or other modules in the system via its Interrupt Control Unit (PICU) and executes a Channel Program (see **Section 17.3**) selected via the priority number of each service request. It first restores the channel program's context from the PRAM and then starts to execute the channel program's instructions stored in the code memory (PCODE). Upon an exit condition, it terminates the channel program and saves its context into PRAM. It is then ready to receive the next service request.

The PCP processor core is capable of suspending execution of a Channel Program on receipt of a service request with a higher priority than the channel currently being executed. The core will automatically resume processing of the original Channel

Program once the higher priority request (or requests) has been processed. A Channel the has been suspended in this way is termed as a Suspended Channel.

The PCP is fully interrupt-driven, meaning it is only activated through service requests; there is no main program running in the background as with a conventional processor.

Table 17-1 PCP Instruction Set Overview

Instruction Group	Description
DMA primitives	Efficient DMA channel implementation
Load/Store	Transfer data between PRAM or FPI memory and the general purpose registers, as well as move or exchange values between registers
Arithmetic	Add, subtract, compare and complement
Divide/Multiply	Divide and multiply
Logical	And, Or, Exclusive Or, Negate
Shift	Shift right or left, rotate right or left, prioritize
Bit Manipulation	Set, clear, insert and test bits and multiple bits set/clear
Flow Control	Jump conditionally, jump long, exit
Miscellaneous	No operation, Debug

17.2.2 PCP Code Memory

The Code Memory (PCODE) of the PCP holds the channel programs, consisting of PCP instructions. All instructions of the PCP are 16 bits long; thus, the PCP accesses its code memory in 16-bit (half-word) quantities. With the 16-bit Program Counter (PC) of the PCP, a maximum of 64 K instructions can be addressed. This results in a maximum size of the PCP code memory of 128 KBytes. The actual type (Flash, ROM, SRAM, etc.) and size of the code memory is implementation specific; see **Section 17.15** for the implemented type and size of the code memory in this derivative.

The PCP code memory is viewed from the FPI Bus as a 32-bit wide memory, that must be accessed with 32-bit (word) accesses, and is addressed with byte addresses. Thus, care has to be taken when calculating PCP instruction FPI addresses. See **Section 17.9** for details.

Note: The PCP has a “Harvard” architecture and therefore cannot directly access code memory other than reading instructions from it. It is recommended that the PCP should not access PCODE via the FPI Bus.

17.2.3 PCP Parameter RAM

The PCP Parameter RAM (PRAM) is the local holding place for each Channel Program's context, and for general data storage, for example, constants, variables and semaphores. It is also an area that the PCP and the host processor or other FPI Bus masters can use to communicate and share data.

While a portion of the PRAM is always implicitly used for the context save areas of the channel programs, the remaining area can be used for channel-specific or general data storage. A programmable 8-bit data pointer (DPTR), concatenated with a 6-bit offset, is provided for arbitrary access to the PRAM. The effective address is a 14-bit word address, allowing a PRAM size of up to 64 KBytes. The actual type (SRAM, DRAM, etc.) and size of the parameter RAM is implementation specific; see **Section 17.15** for the implemented size of the PRAM in this derivative.

Both the PCP and FPI Bus masters address the PRAM as 32-bit words. There is no concept of half-word or byte accesses to PRAM. FPI Bus masters must, however, use byte addresses in order to access PRAM memory. As for the code memory, care has to be taken when calculating PRAM FPI addresses. See **Section 17.9** for details.

17.2.4 Slow FPI Bus Interface

The PCP can access all peripheral units on the FPI Bus and other resources through the FPI Bus interface. The PCP can become an FPI Bus slave, so that other FPI Bus master may access code and PRAM memory and the control and status registers in the PCP.

The Code Memory and PRAM Memory blocks are visible to FPI Bus masters as a block of memory on the FPI Bus. If an FPI Bus master accesses PCP Code or PRAM memory concurrently with the PCP, the external FPI Bus master is given precedence over the PCP to avoid deadlocks. The PCP access is stalled for several cycles until the FPI Bus master has completed its access. If an FPI Bus master performs an atomic read-modify-write access to a PCP memory block, any concurrent PCP access to that block is stalled for the duration of the atomic operation.

17.2.5 PCP Interrupt Control Unit and Service Request Nodes

The PCP is activated in response to an interrupt request programmed for PCP service in one of the service request nodes of the system (nodes associated with a peripheral, the CPU, external interrupts, etc.). The PCP Interrupt Control Unit (PICU) determines the request with the currently highest priority and routes the request together with its priority number to the PCP processor core. It also acknowledges the requesting source when the PCP starts the service of this interrupt.

The PCP itself can generate service requests to either the CPU or itself through a number of PCP Service Request Nodes (PSRNs). The PSRNs are also used to store all information required by the PCP core to allow the later restart of a Channel program

when it is suspended in favour of a higher priority Service Request. Please refer to **Section 17.5.3** for more detailed information on the operation of these nodes.

17.3 PCP Programming Model

The PCP programming model can be viewed as a set of autonomous programs, or tasks, called Channel Programs, that share the processing resources of the PCP. Channel Programs may be short and simple, or very complex, but, they can coexist persistently within the PCP.

From a programming point of view, the individual parts of a channel program are its instruction sequence in the code memory and its context in the parameter RAM. It uses the instruction set and the general purpose registers (R0 - R7) of the PCP processor core to perform the necessary operations, and to communicate with the various resources of the on-chip and off-chip system depending on its task in the application.

These parts of the programming model are discussed in the following sections (with the obvious exception of the system environment outside of the scope of the PCP).

17.3.1 General Purpose Register Set of the PCP

The program-accessible register file of the PCP is composed of eight 32-bit General Purpose Registers (GPRs). These registers are all accessible by PCP programs directly as part of the PCP instruction set. Source and destination registers must be specified in most instructions. These registers are referenced to in this document as R*n* or R[*n*], where *n* is in the range 0..7.

Table 17-2 Directly Accessible Registers

Register	Implicit Use	Description
R0	Accumulator	Implicit target for some arithmetic and logical instructions
R1	–	32-bit general-use register
R2	Return Address	32-bit general-use register
R3	–	32-bit general-use register
R4	SRC (Source)	Source Pointer for BCOPY/COPY instructions
R5	DST (Destination)	Destination Pointer for BCOPY/COPY instructions

Table 17-2 Directly Accessible Registers

Register	Implicit Use	Description
R6	CPPN/SRPN/ TOS/CNT1	CNT1: Transfer Count for BCOPY/COPY. TOS: Type of Service. SRPN: 8-bit field used for posting interrupt on EXIT instruction. CPPN: Current PCP Priority Number
R7	DPTR/Flags	PRAM Data Pointer (DPTR) and Status Flags

R7 is the only one of the eight registers that may not be used as a full GPR. The most significant 16 bits of R7 may not be written, and will always read back as 0. However, no error will occur when writing to the most significant 16 bits.

Note: The general purpose registers of the PCP are not memory-mapped into the overall address space. They can only be directly accessed through PCP instructions. The contents of all or some of the registers are part of a channel program's context stored in the PRAM between executions of the channel program. This context is then accessible from outside the PCP.

17.3.1.1 Register R0

R0 is used as an implicit operand destination for some instructions. These are detailed in the individual instruction descriptions.

17.3.1.2 Registers R1, R2, and R3

R1, R2, and R3 are general-use registers. It is recommended that, by convention, R2 should be used as a return address register when call and return program structures are used.

17.3.1.3 Registers R4 and R5

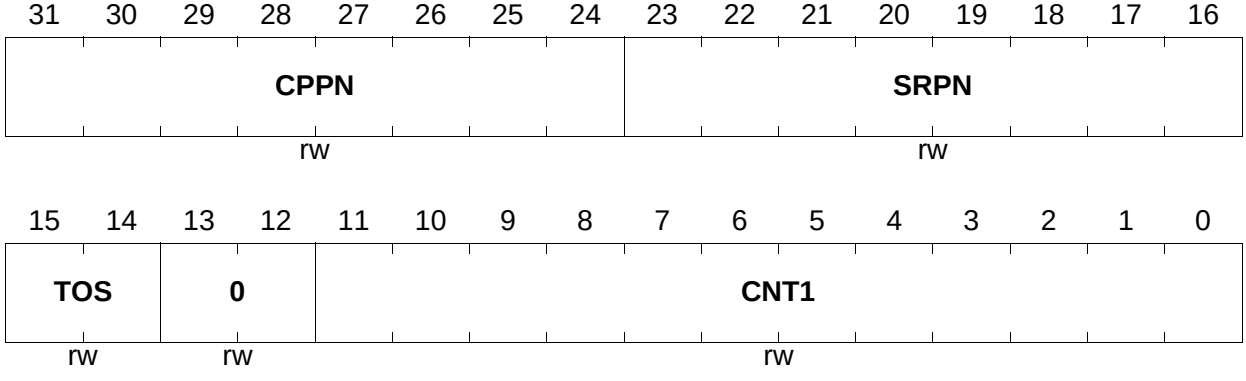
Registers R4 and R5 are also general-use registers. However, the BCOPY/COPY instructions implicitly use R5 and R6 as full 32-bit address pointers (R4 is used as the source address and R5 as the destination address). As the BCOPY/COPY instruction use these registers to maintain the address pointers, either or both R5 and R6 values may or may not be modified by the BCOPY/COPY instruction depending on the options used in the instruction.

17.3.1.4 Register R6

Register R6 may also be used as a general-use register. Again however, there are some instructions that use fields within R6. If the BCOPY, COPY or EXIT instructions are used, then the field R6.CNT1 can be optionally implicitly used as a counter. In the case of the

BCOPY and COPY instructions, R6.CNT1 can be used as an "outer loop counter" to streamline the movement of large amounts of data. R6.CNT1 can also (optionally) be decremented by an exit instruction which allows a channel to maintain a count of the number of times that it has been invoked. If an EXIT instruction is used that causes an interrupt, R6.SRPN and R6.TOS must be configured properly prior to execution of the EXIT. The TOS field selects the destination for the request (e.g. TriCore CPU or PCP), the SRPN selects the priority of the requested interrupt. If interrupt priority management is used, then R6.CPPN must be set to the priority level at which the channel shall run at its next invocation, before the EXIT is executed. The fields for R6 are shown below.

PCP Register R6 **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
CNT1	[11:0]	rw	**General-use/Outer Loop Count for BCOPY, COPY or EXIT Instruction**
TOS	[15:14]	rw	**General-use/Type Of Service for EXIT Interrupt**
SRPN	[23:16]	rw	**General-use/Service Request Priority Number for EXIT Interrupt**
CPPN	[31:24]	rw	**General-use/PCP Priority Number Posted to PICU**
0	[13:12]	rw	**General-use**

17.3.1.5 Register R7

Register R7 is an exception with respect to the other registers in that not all bits within the register can be written, and the implicit use of the remaining bits virtually excludes the use of R7 as a general purpose register. R7 serves purposes similar to those of a Program Status Word found in traditional processors.

R7 holds the flag bits, a channel enable/disable control bit, and the PRAM data pointer (DPTR). The upper 16 bits of R7 are reserved.

Most instructions of the PCP update the flags (CN1Z, V, C, N, Z) in R7 according to the result of their operation. See **Table 17-16** for details on the flag updates of the individual instructions. The values of the flag bits in R7 maintain their state until another instruction that updates their state is executed.

Note: Implicit updates to the flags caused by instruction take precedence over any bits that are explicitly moved to R7. For example, if a MOV instruction is used to place $0000FF03_H$ in R7, then the bit positions for the Z (zero) and N (negative) flags are being written with 1. The MOV instruction, however, implicitly updates the Z and N flag bits in R7 as a result of its operation. Because the number is not negative, and not zero, it will update the Z and N flags to 0. As a result, the value left in R7 after the MOV is complete will be $0000FF00_H$ (i.e C = 1, Z = 0, N = 0).
It is recommended that only SET and CLR instructions should be used to explicitly modify flags in R7.

The data pointer, R7.DPTR, is the means of accessing PRAM variables programmatically. It points to a 64-word PRAM segment that may be addressed by instructions which can use the PRAM for source or destination operands (xx.P and xx.PI instructions). The 8 bits of the DPTR are concatenated with a 6-bit offset value (either specified in the instruction as an immediate value or contained in one of the registers) to give a 14 bit (word) address. A program is able to update the DPTR value dynamically, in order to index more than 64 words of PRAM.

The channel enable control bit, R7.CEN, allows the enabling or disabling of specific channel programs. If an interrupt request is received for a channel which is disabled, an error is forced and an error interrupt to the CPU is activated.

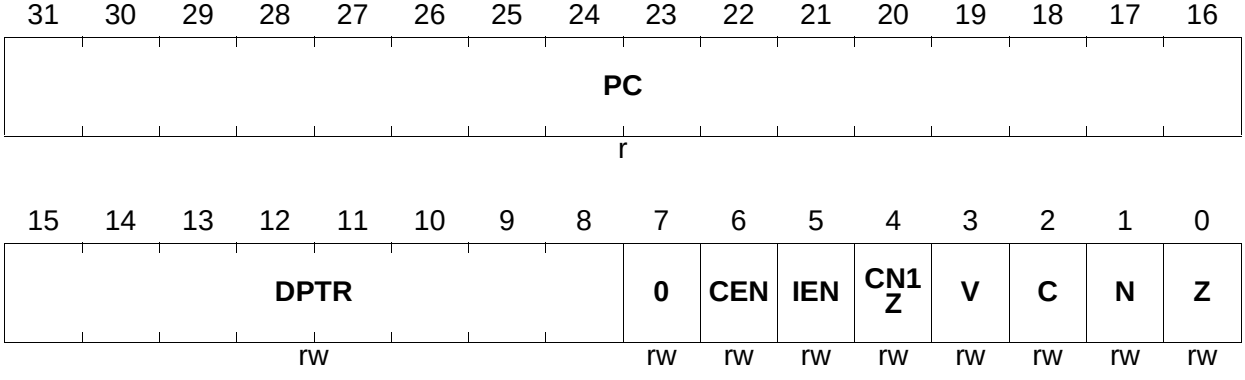
The interrupt of channel enable control bit, R7.IEN, allows the enabling or disabling of channel interruption on a channel to channel basis. When R7.IEN is 0, the channel will continue execution regardless of the priority of any new Service Requests. When R7.IEN is 1 and conditions allow, the channel will be suspended on receipt of a higher priority Service Request.

The 16-bit Program Counter, R7.PC, points to the next instruction location. This allows an address range of up to 64K instructions. But this field is neither read (Read as 0) nor directly written via PCP instructions. The value of PC is saved in the context save area as part of a context save.

17.3.2 Contexts and Context Models

After initialization, the instruction sequence of a PCP channel program is, permanently stored (i.e. usually at least as long as the application is running) in the code memory, and data parameters are held in the parameter RAM. These will remain stored regardless of whether a particular channel program is currently idle or is executing (although, of course, the value of data variables in the PRAM might be modified by the PCP or other FPI Bus masters). The contents of the general purpose registers of the PCP (used as

PCP Register R7 **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
Z	0	rw	**Zero**
N	1	rw	**Negative**
C	2	rw	**Carry**
V	3	rw	**Overflow**
CNZ	4	rw	**Outer Loop Counter 1 Zero Flag**
IEN	5	rw	**Interrupt of Channel Enable Control Bit** 0 Channel is not interruptible 1 Channel can be suspended in favour of a higher priority Service Request
CEN	6	rw	**Channel Enable Control Bit**
DPTR	[15:8]	rw	**Data Pointer Segment Address for PRAM accesses**
0	7	rw	**Reserved;** should always be written with 0.
PC	[31:16]	r	**Reserved**; read as 0; should be written with 0.

address pointers, data variables, intermediate results, etc.) however, are usually only valid for a given channel program as long as it is executing. If another channel program is executed, it will re-use the general purpose register according to its needs.

Thus, the state of the general purpose registers of a channel program (termed the “Context” of the channel) needs to be preserved while a channel program is not being executed. The content of the registers needs to be saved when execution of a channel program finishes, and needs to be restored before execution starts again.

The PCP implements automatic handling of these context save and restore operations. On termination of a channel program, the state of all or some of the general purpose registers is automatically copied to a defined area in the PRAM (Context Save). If the same channel program is re-activated, the contents of the registers are restored by

copying the values from the same defined PRAM area into the appropriate registers (Context Restore).

The defined area in the PRAM for the context save and restore operations is called the Context Save Area (CSA). Each channel program has its own individual, predefined region in the CSA. When a service request is accepted by the PCP, the service request priority number (SRPN) associated with the request is used to select the channel program and its respective CSA region.

17.3.2.1 Context Models

A Context Model is a means of selecting whether some or all of the registers are saved and restored when a context switch occurs. In order to serve different application needs in terms of PRAM space usage, the PCP offers a choice between three different context models:

- Full Context: Eight Registers (8 × 32-bit words) are saved/restored per channel.
- Small Context: Four Registers (4 × 32-bit words) are saved/restored per channel.
- Minimum Context Model: Two Registers (2 × 32-bit words) are saved/restored.

As illustrated in **Figure 17-2**, the contents of R0 through R7 constitute the Full Context of a channel program. A Small Context consists of R4 through R7. Use of the small context model allows for correct operation of DMA channels, as well as channels which are not required to save large amounts of data in their contexts between invocations. A Minimum Context saves and restores only R6 and R7.

To distinguish the actual register contents from the copies stored in the PRAM context regions, the term CRx is used throughout the rest of this document to refer to the register values in the context regions. Registers R6 and R7 are always handled in a special way during context save and restore operations, this is described in detail in **Section 17.3.2.3**.

The context model is selected via a bit field (PCP_CS.CS) in the global PCP control register PCP_CS, this is a global setting (i.e. the selected context model is used for all channels). Once a context model has been selected (during PCP configuration) and the PCP has been started the PCP must continue to use that context model. Attempting to change the context model in use during PCP operation will lead to invalid context restore operations which will in turn lead to invalid PCP operation.

In the case of small and minimum context models, the unsaved and unrestored registers (shaded in **Figure 17-2**) can be thought of as global registers that any Channel Program can use or change, or reference as constants — for example as base address pointers (see **Section 17.13.2** for more detail).

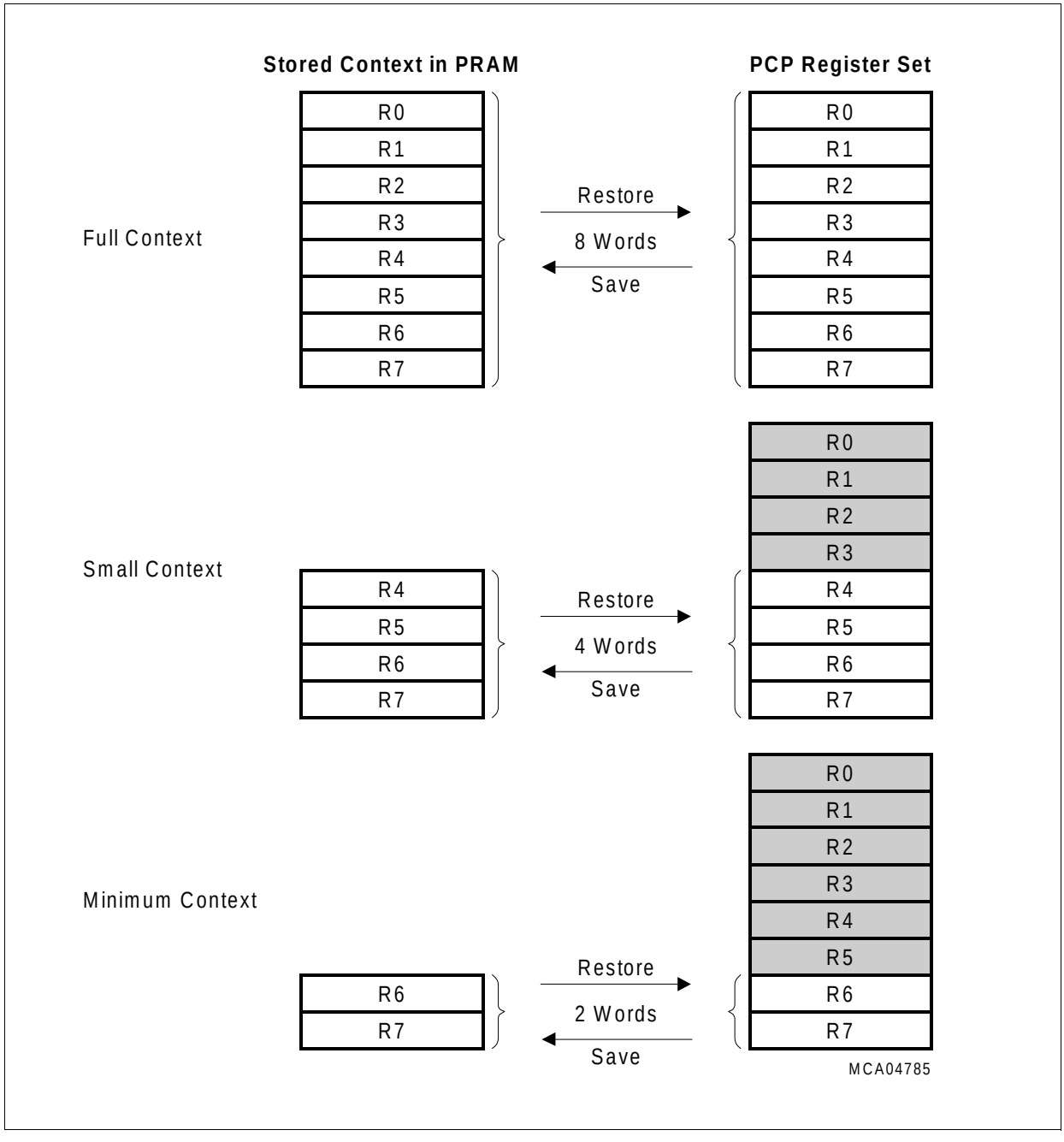


Figure 17-2 PCP Context Models

17.3.2.2 Context Save Area

The Context Save Area (CSA) is a region in PRAM reserved for storing the contexts for all channel programs (while any particular channel is not executing). Each channel's context is stored in a region of the CSA based on the channel number. The channel number is equal to the priority number (SRPN) of the service request. The PCP uses this number to calculate the start address of the context of the associated channel program. The size of a context is determined by the context model that the PCP has been initialized to use. As all channels use the same context size, the PRAM address (word address) of the context for a particular channel is simply calculated by multiplying the channel number by the number of registers in the context (8 for full context, 4 for small context and 2 for minimum context). **Figure 17-3** shows the resulting PRAM layout, and from this it can be seen that changing the context model also changes the base address for all regions within the CSA. Thus, the chosen context model may only be set when the PCP is initialized, and may not be changed during operation.

The context save area in the PRAM starts at address 00_H and grows upward. It is partitioned into equally sized regions, where the size of these regions is determined by the selected context model. The priority number (SRPN) of a service request is used to access the appropriate context region for the associated channel program. Since a request with an SRPN of 00_H is not considered as valid request in the TriCore Architecture, the bottom region (context region 0) of the CSA is never used for an actual context.

The total size of the CSA depends on the context model and the number of service request numbers used in a given system. Each priority number used in a service request node which can activate interrupts to the PCP must be represented through a dedicated context region in the PRAM. The highest address range in the PRAM used for a context region is determined by the highest priority number presented to the PCP with a service request.

The range of usable priority numbers is further determined by the size of the implemented PRAM and by the space required for other variables and global data located in the PRAM. See **Section 17.15** for the implemented size of the PRAM in this derivative. As an example, a PRAM memory of 2 KBytes, solely used for the CSA, can store up to 255 minimum contexts, allowing the highest SRPN used for a PCP service request to be 255 (remember, an SRPN of 0 and an associated context region is never used; the valid SRPNs and the context and channel numbers range from 1 to 255). With a small context model, 127 contexts can be stored, resulting in 127 being the highest usable SRPN in this configuration. Finally, a full context model allows 63 context areas, with 63 being the highest usable SRPN. Interrupt requests to the PCP with priority numbers that would cause loading of a context from outside the available PRAM area must not be generated. Invalid PCP operation will result should this situation be allowed to occur. The PCP can be optionally configured such that if an interrupt request is received that would cause loading of a context from outside the available PRAM area

then an error exit is forced, and an error interrupt to the CPU is activated (see **Section 17.6.1**).

If portions of the PRAM are used for other variables and global data, the space available for the CSA and the range of valid SRPNs is reduced by the memory space required for this data. For best utilization of PRAM it is advisable to have the CSA grow upwards as a contiguous area without any ‘holes’, meaning that all SRPNs in the range 1..max. are actually used to place interrupt requests on the PCP. Unused regions within the CSA (that is, the unused region at the base of the CSA and any context regions associated with unused channels) cannot be used for general variable storage.

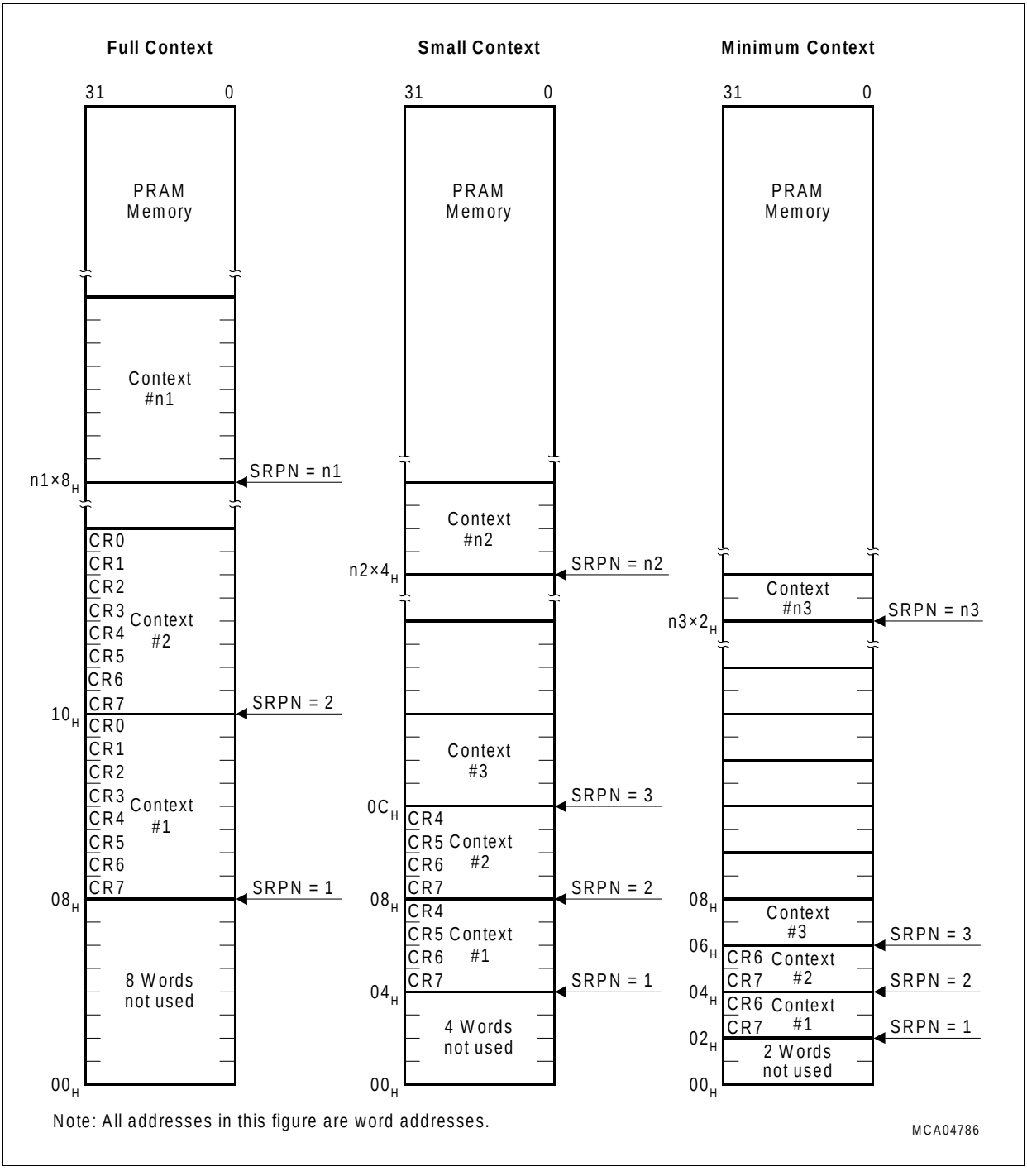


Figure 17-3 Context Storage in PRAM

When choosing the context model for a given application, the following considerations can be helpful. When choosing the small or the minimum context models, save and restore operations for registers not handled in the automatic context operations can still be handled through explicit load and store instructions under control of the user. This may be advantageous for applications where the majority of the channels don't need the

full context, and only some would require more context to be saved. In this case, a smaller context model can be used, and the channels which would require more register to be saved/restored would do this via explicit load and store instructions. This is especially advantageous if the channel program can be designed such that the initial real time response operations can be executed using only the registers which have been automatically restored. Then, as the timing requirements of the service are relaxed, further register contents can be restored from PRAM through regular load instructions. Of course, the contents of these registers needs to be explicitly saved, through regular store instructions, before the exit of the channel program.

The criteria for choosing the context model are listed in the following:

- Size of PRAM memory implemented in a given derivative
- Amount of channels (= SRPNs) which need to be used in a system
- Amount of PRAM used for general variables and globals
- Amount of context (register content) which need to be saved and restored quickly by most of the most important channels

While registers R0 through R5 are always restored in a normal manner (according to the context size), the registers R6 and R7 merit discussion regarding context restore operations. The memory location CR7 in a context region is used to hold two different pieces of information: namely the lower part of register R7, and the PC value of the channel. Similarly, the memory location CR6 in a context region can also be used to hold two different pieces of information: namely the value to be restored to register R6, and the Operating Priority (CPPN) value of the channel. This leads to the Restore/Save operations described in the following two sections.

17.3.2.3 Context Restore Operation for CR6 and CR7

The operation of R6 and R7 context restore varies according to whether the channel program that is starting is a new channel program (i.e. a channel program that is starting in response to the receipt of a new Service Request) or a suspended channel program (i.e. a channel program that is re-starting after being suspended in favour of a higher priority channel program). In addition, when a new channel program is starting, the context restore operation depends on the channel start mode that has been selected. (see **Section 17.3.3**).

Channel Resume Mode

Figure 17-4 illustrates the operation of a context restore for a new channel program when Channel Resume Mode has been selected. The PC is loaded from CR7[31:16] and the lower half of R7 is loaded from CR7[15:0]. The operating priority of the channel is taken from CR6[31:24] and all of R6 is loaded from CR6.

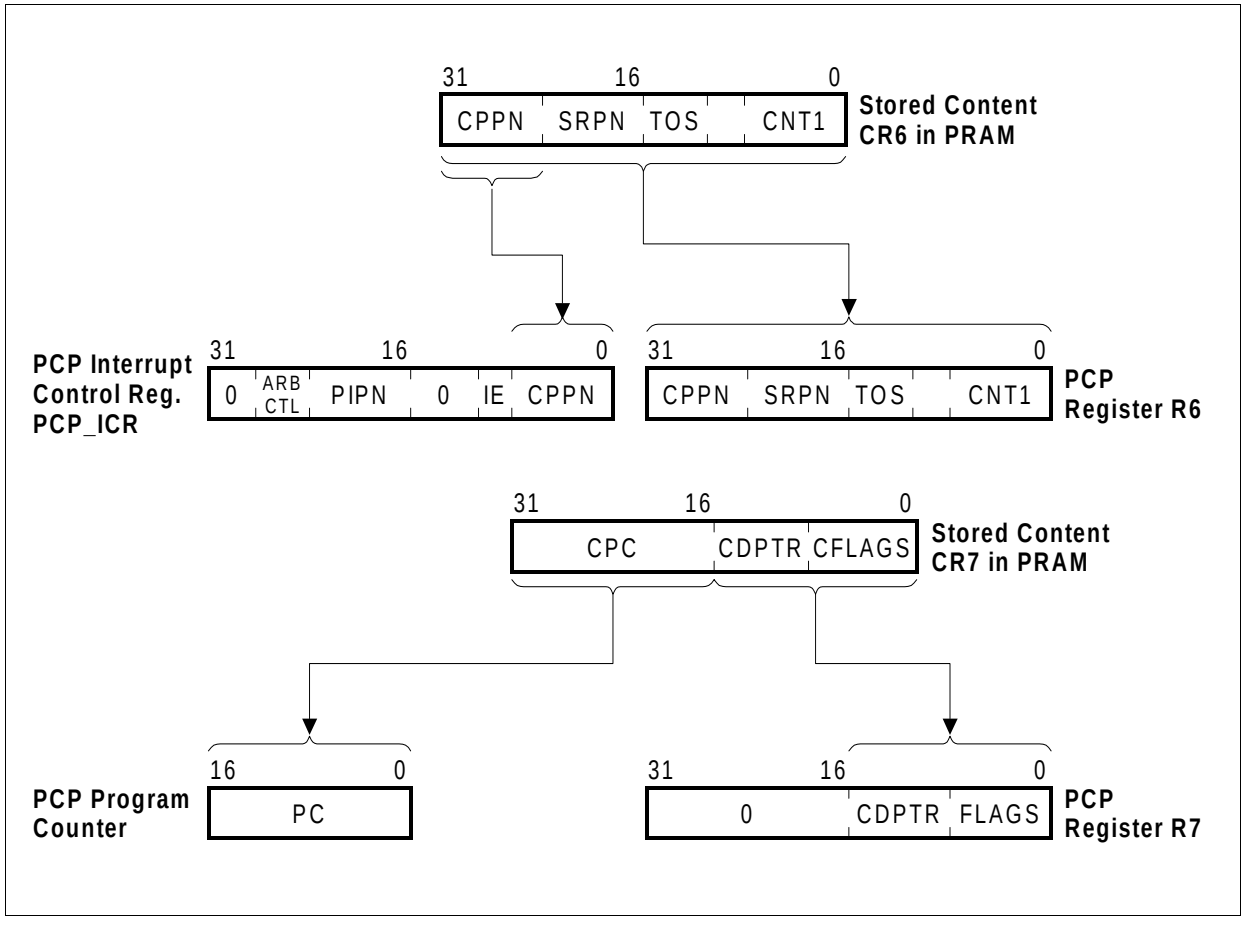


Figure 17-4 Context Restore for CR6 and CR7 in Channel Resume Mode

Channel Restart Mode

Figure 17-5 illustrates the operation of a context restore for a new channel program when Channel Restart Mode has been selected. The PC is loaded with the channel entry entry table address and the lower half of R7 is loaded from CR7[15:0]. The operating priority of the channel is taken from CR6[31:24] and all of R6 is loaded from CR6.

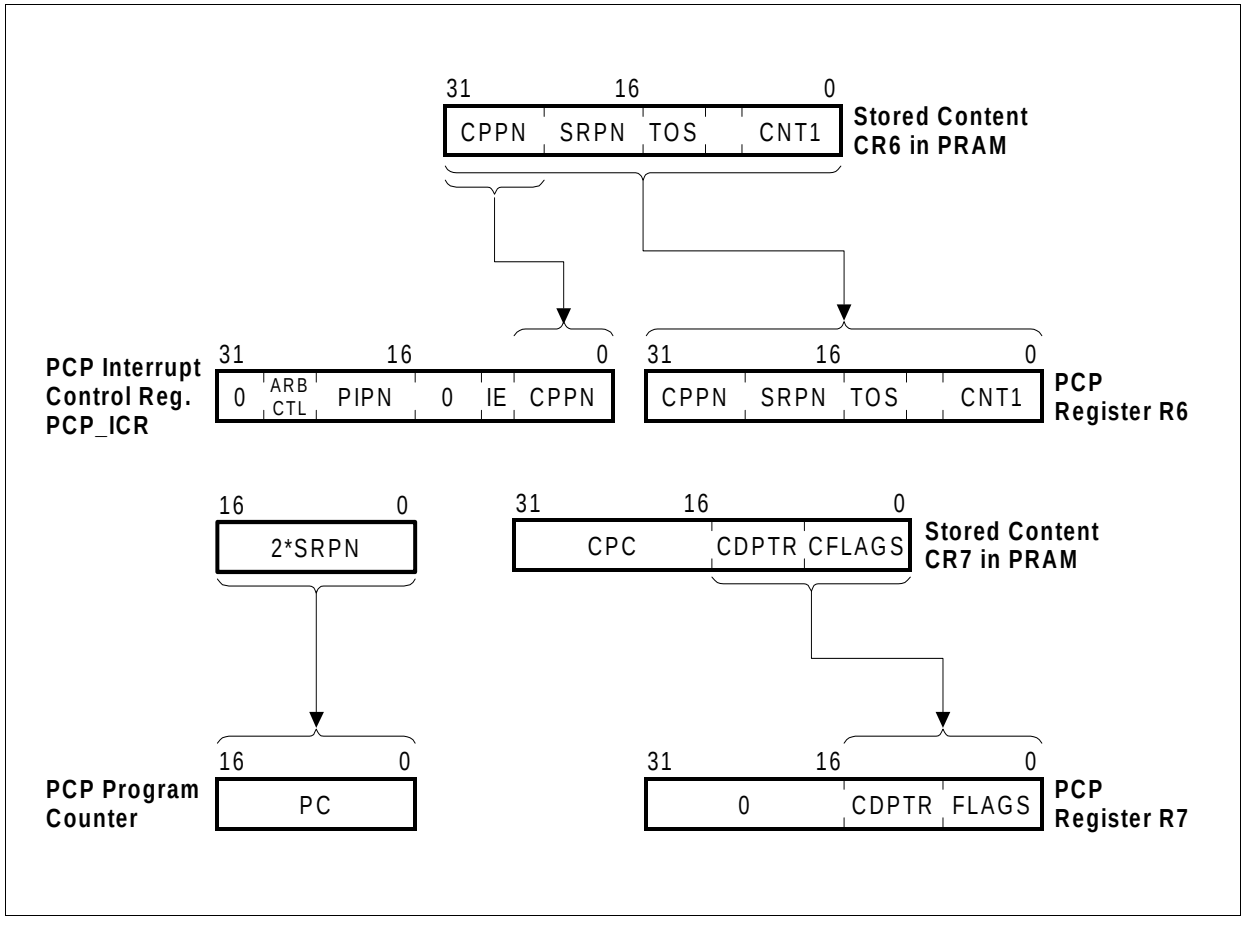
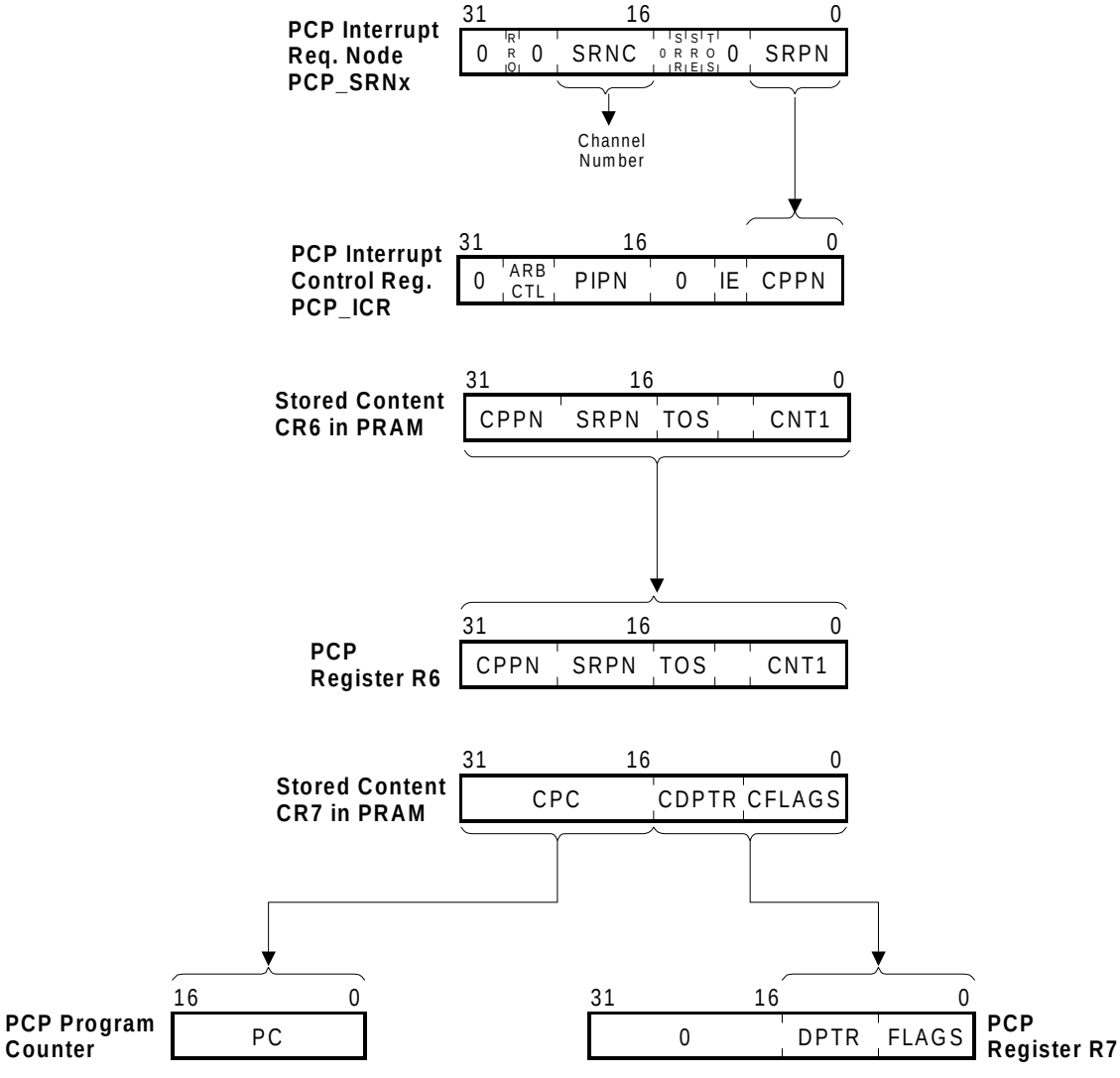


Figure 17-5 Context Restore for CR6 and CR7 in Channel Restart Mode

Suspended Channel Restart

Figure 17-6 illustrates the operation of a context restore for a suspended channel program. The PC is loaded from CR7[31:16] (regardless of the Channel Start Mode) and the lower half of R7 is loaded from CR7[15:0]. All of R6 is loaded from CR6. The figure also shows how the operating priority of the channel (PCP_IR.CPPN) is restored from the Service Request Node that was used to store the Suspended Interrupt Request.



Note: During a context restore for a suspended channel, the PCP_SRCNx.SRNC field (n=9,10,11) is used to determine the channel number.

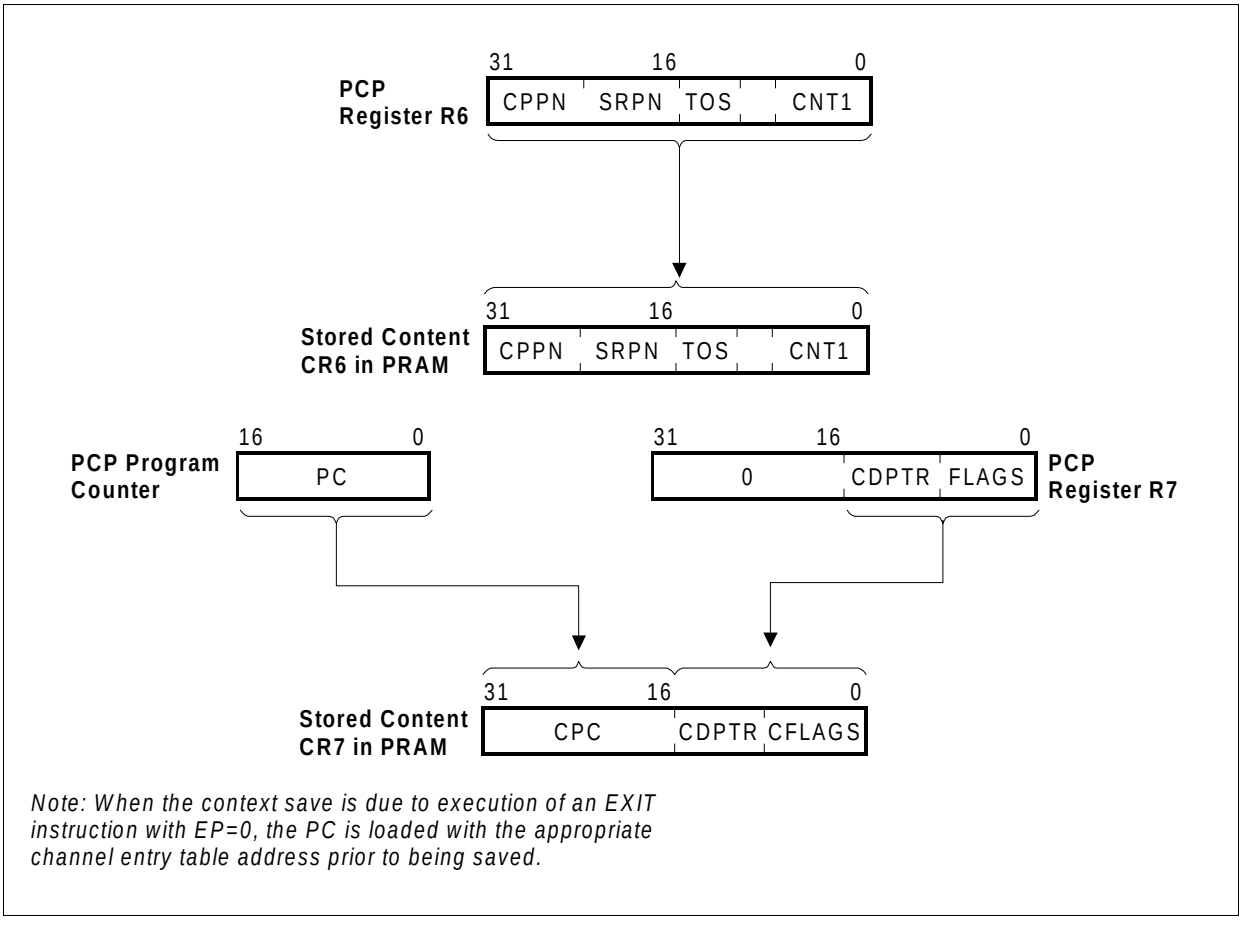
Figure 17-6 Context Restore for CR6 and CR7 Of Suspended Channel Restart

17.3.2.4 Context Save Operation for CR6 and CR7

The operation of R6 and R7 context save varies according to whether the save operation is the result of a channel exit condition or whether the channel is being suspended in favour of a higher priority channel program.

Channel Resume Mode

Figure 17-7 illustrates the operation of a context save for a channel exit when Channel Resume Mode has been selected. The value written to CR7 is created by concatenating the 16-bit PC value with the lower 16-bit of R7. CR6 is written with the value taken from R6.



Note: When the context save is due to execution of an EXIT instruction with EP=0, the PC is loaded with the appropriate channel entry table address prior to being saved.

Figure 17-7 Context Save for CR6 and CR7 in Channel Resume Mode

Channel Restart Mode

Figure 17-8 illustrates the operation of a context save for a channel exit when Channel Restart Mode has been selected. This is same as for Channel Resume mode except that the PC value is discarded and the appropriate channel entry table address is written to CR7[31:16].

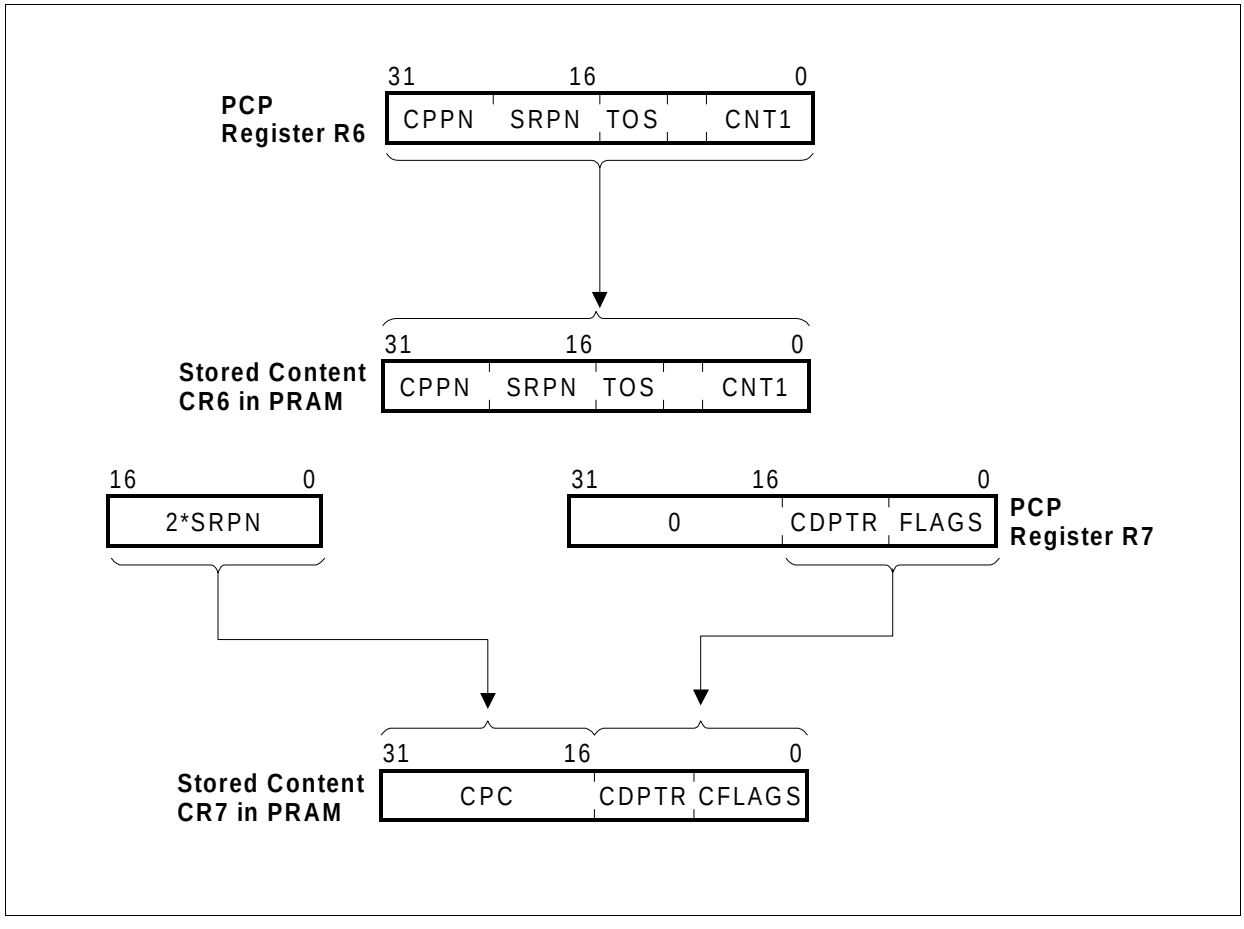


Figure 17-8 Context Save for CR6 and CR7 in Channel Restart Mode

Channel Suspended

Figure 17-9 illustrates the operation of a context save for a channel which is being suspended. This is same as for Channel Program Resume mode except that an interrupt request is created to allow the channel to be restarted at a later time. This restore operation utilizes one of the three specially extended Service Request Nodes to store the interrupt request. The information stored as part of the interrupt request is channel number (SRPN) and the operating priority (CPPN) with which the channel was operating prior to being suspended. This operation in conjunction with the suspended channel restore operation shown in **Figure 17-6** allows the temporary suspension of a channel in favour of higher priority channel.

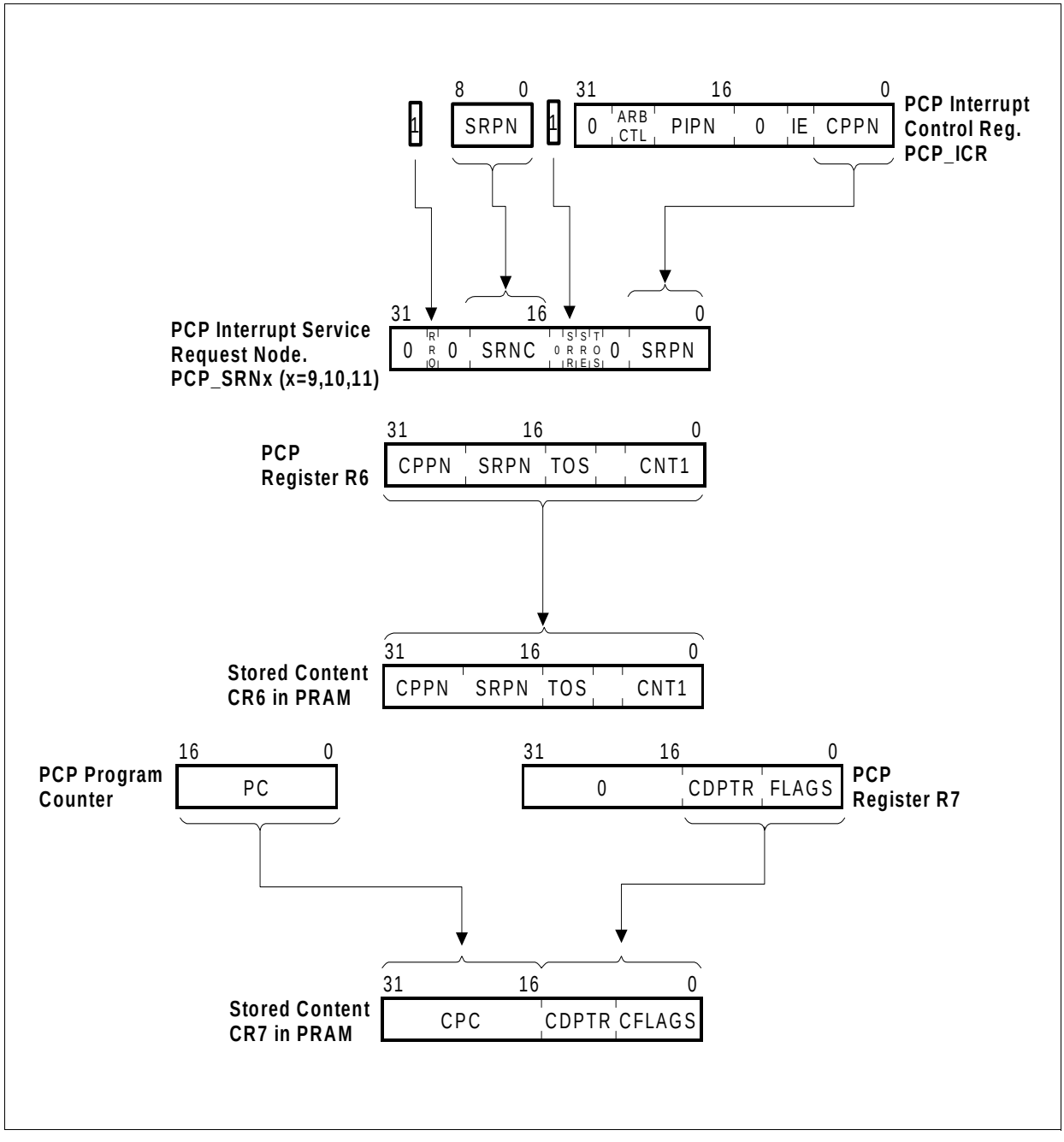


Figure 17-9 Context Save for CR6 and CR7 Of Channel Suspend

17.3.2.5 Initialization of the Contexts

The programmer is responsible for configuring each Channel Program's context before commencing operation. Because this must be done by writing to the PCP across the FPI Bus, it is important to understand exactly where each Channel Program's context is from the FPI Bus perspective (see **Page 17-45** for details).

17.3.2.6 Context Save Optimization

The PCP has an optimized context switching strategy consisting of optimization of both context load and store. During a context load where the channel that is starting is also the last channel that the PCP was running, then the PCP general purpose registers already contain the values appropriate to the channel. In this case there is no need to reload the context (i.e. the PCP can immediately continue operation at the appropriate point in the code without having to perform a context load). During a Context Store (i.e. the PCP exits a channel as a result of EXIT or DEBUG instructions or exits in response to a higher priority channel interrupt) only those registers that have ben updated (i.e. was written to) since the context was loaded are saved to the context save area.

17.3.3 Channel Programs

The PCP code memory (PCODE) is used to store the instruction sequences, the Channel Programs, for each of the PCP channels. The individual channel programs for the individual PCP service requests can be usually viewed as independent and separate programs. There is no background program defined and running for the PCP in TC11IB as there would be with traditional processors.

When the PCP receives a service request for a specific channel program, it needs to exactly determine which channel program to activate and where to start its execution. To accommodate different application needs, the PCP architecture allows the selection of two different entry methods into the channel programs:

- Channel ReStart Mode
- Channel Resume Mode

Channel Restart Mode forces the PCP to begin each Channel Program from a known fixed point in code memory that is related to the interrupt number. At the entry point related to the interrupt number in question there will typically be a jump instruction which vectors the PCP to the main body of the channel program. This is identical to the traditional interrupt vector jump table. In Channel Restart Mode, channel code execution will always start at the same address in the interrupt entry table each time the channel is requested.

Channel Resume Mode allows the PCP to begin execution at the PC address restored as part of the channel program context. This mode allows for code to be contiguous and start at any arbitrary address. It also allows for the implementation of interrupt-driven state machines, and even the sharing of code across multiple programs with different context.

The selection of one of the two modes is a global PCP setting, that is, it applies to all channels. Selection is made via the PCP_CS.RCB bit in the PCP configuration register PCP_CS (see **Section 17.11.2**).

17.3.3.1 Channel Restart Mode

Channel Restart Mode is selected with PCP_CS.RCB = 1. In this mode, the PCP views the code memory as being partitioned into an interrupt entry table at the beginning of the code memory, and a general code storage area above this table.

The interrupt entry table consists of two instruction slots (2 × 16-bit) for each channel. When a PCP service request is received, the PCP calculates the start PC for the requested channel by a simple equation based on the service request priority number (SRPN) of that request (PC = 2 × SRPN). It then executes the instruction found on that address. If more than two instructions are required for the operation of the channel program, then one of the instructions within the interrupt entry table must be a jump to the remainder of the channel's code. The PCP executes the channel's code until an exit condition or higher priority interrupt is detected.

It is recommended that all EXIT instructions for all channels should use the EP = 0 setting when the PCP is operated in Channel Restart Mode (see **Section 17.12.15**).

Note that when Channel Restart Mode is in use, a Channel Entry Table must be provided with a valid entry for every channel being used. **Figure 17-10** shows an example of Code Memory organization when Channel Restart Mode has been selected. Failure to provide a valid entry for all channels that are in use will lead to invalid PCP operation.

17.3.3.2 Channel Resume Mode

Channel Resume Mode is selected with PCP_CS.RCB = 0. In this mode, the user can arbitrarily determine the address at which the channel program will be started the next time it is invoked. For this purpose, the PC is saved and restored as part of the context of a PCP channel.

Additional flexibility is available when Channel Resume Mode is globally selected by configuring each EXIT instruction to determine the channel start address to be used on the next invocation of a channel (see **Section 17.12.15**). When the EP = 0 setting is used the PC value saved in the channel's context (saved in CPC) is the address of the appropriate location in the channel entry table. This forces the channel to start at the appropriate location in the interrupt entry table at next invocation. When the EP = 1 setting is used the PC value saved in the channel's context is the address of the instruction immediately following the EXIT instruction. The use of the EP = x setting with the EXIT instruction allows the mixture of channels that use a Channel Restart strategy with others using a Channel Resume strategy, and also allows individual channels to use either strategy as appropriate on different invocations.

Note: A valid entry within a Channel Entry Table must be provided for every channel that uses an EXIT instruction with the EP = 0 setting when Channel Resume Mode has been selected. Failure to provide a valid entry for such channels will lead to invalid PCP operation.

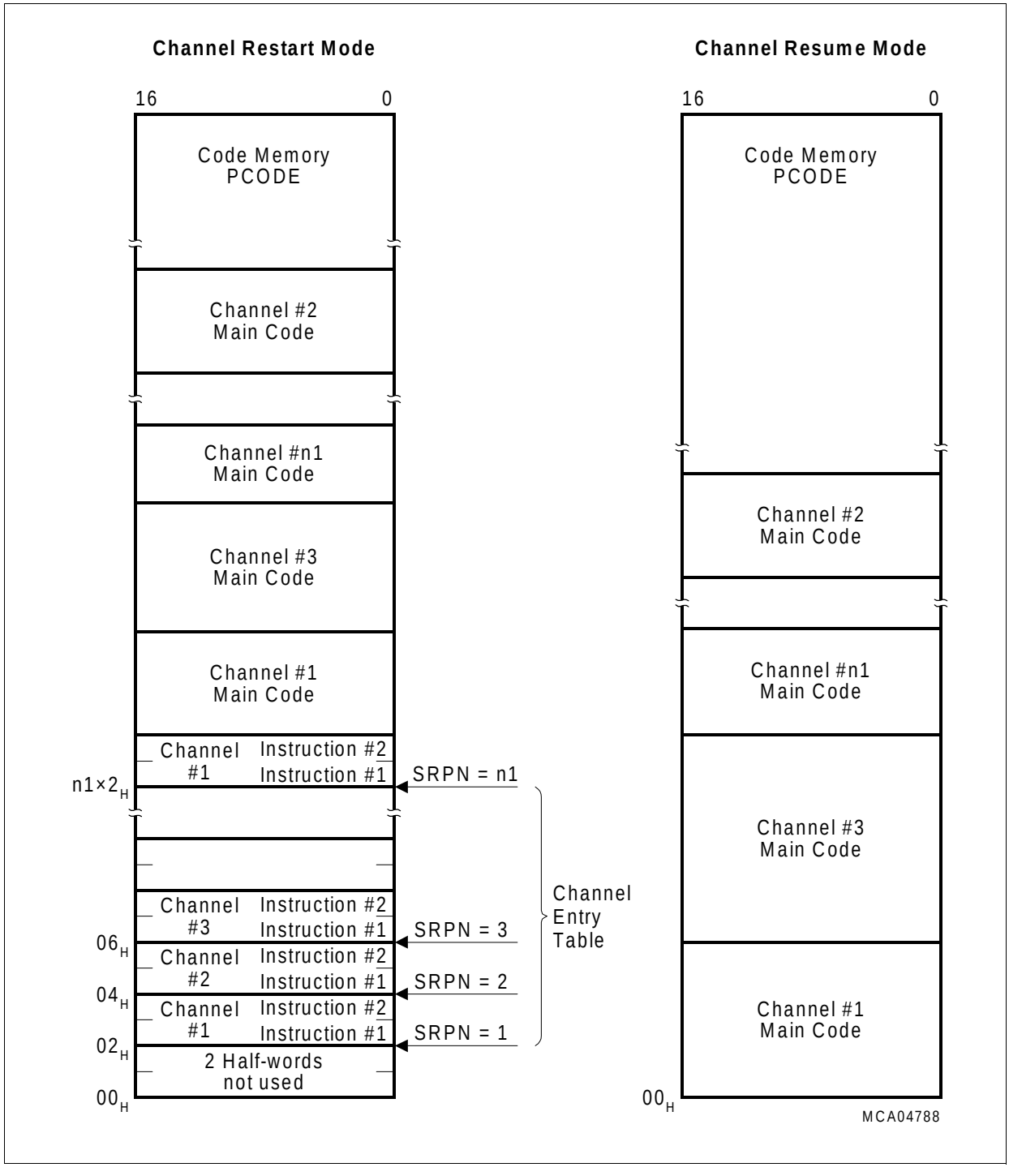


Figure 17-10 Examples of Code Memory Organization for Channel Restart and Channel Resume Modes

Note: The Code Memory address offsets in the above figure are shown as PCP instruction (half-word) offsets. To obtain FPI address offsets (byte offset) multiply each offset by two.

17.4 PCP Operation

This section describes how to initialize the PCP, how to invoke a Channel Program, and the general operation of the PCP.

17.4.1 PCP Initialization

The PCP is placed in a quiescent state when the TC11IB is first powered-on or reset. Before a Channel Program can be enabled, the PCP as a whole must be initialized by some other FPI Bus master, typically the CPU. Initialization steps include:

- Configure global PCP registers
  - Initialize PCP Control and Status Register (with PCP_CS.EN = 0)
  - Configure interrupt system via PCP_ICR.
- Load Channel Programs into the code memory PCODE.
- Load initial context (if/as required) of Channel Programs in PRAM (R0–R7 for Maximum context, R4–R7 for Small context, R6–R7 for Minimum context). Only those registers in each channel whose initial content is required on first invocation of the channel need to be loaded. This may need to include the initial PC, depending on the value of PCP_CS.RCB.
- Clear R7 in the context for unused channels.
- Enable PCP operation PCP_CS.EN = 1.

Now, the PCP is able to begin accepting interrupts and executing Channel Programs.

17.4.2 Channel Invocation and Context Restore Operation

A Channel Program is started when any one of the following conditions occurs:

- The current round of PCP interrupt arbitration results in a winning interrupt number (SRPN) and the PCP is currently quiescent (has exited the previous channel and stored the context for that channel)
- The current round of PCP interrupt arbitration results in a winning interrupt number (SRPN) which is greater than the current channel priority, R7.IEN ==1 and suitable Service Request Node space is available in the PSRN

When this happens, the winning SRPN becomes the current interrupt and a context restore operation occurs before the Channel Program can begin operation as follows.

- The context of the Channel (= winning SRPN) is restored from PRAM into the general purpose registers from the appropriate address within the CSA. Depending on the value of PCP_CS.CS, a Full, Small, or Minimum Context restore is performed.
- The new priority level of the PCP is taken from R6.CPPN field and is written to PCP_ICR.CPPN. This value can be useful during debugging, as the CPPN of the currently executing or last-executed Channel Program can be read from PCP_ICR. After the Channel Program starts, the value of R6 may be changed without altering the value of the effective CPPN, because updates to the value of R6.CPPN have no effect until the next invocation of the Channel Program.

- If the R7.CEN bit is clear (0), then an error has occurred because a disabled Channel Program has been invoked, the PCP_ES.DCR bit is set to flag the error, and the Channel Program performs an error exit.
- If the R7.CEN bit is set (1), then code execution begins at the value of the restored PC or at the address of the interrupt routine in the Channel Entry Table, depending on the value of PCP_CS.RCB.

17.4.3 Channel Exit and Context Save Operation

The context of a channel program must be saved when it terminates. Four events can cause the termination of a channel program:

- Execution of the EXIT instruction (normal termination)
- A higher priority interrupt causes suspension of the Channel and the start of a new Channel Program
- Occurrence of an error
- Execution of the DEBUG instruction (channel termination is optional). The DEBUG instruction must be only used in DEBUG mode, otherwise an Illegal Operation (IOP) error will be generated.

These channel termination possibilities are described in the next sections.

17.4.3.1 Normal Exit

Under normal circumstances, a channel program finishes by executing an EXIT instruction. This instruction has several setting fields which allow the user to specify a number of optional actions to be performed during the channel exit sequence. These optional actions are:

- Decrement counter CNT1.
- Set the start PC for the next channel invocation to the next instruction address (Channel Resume) or to the channel entry address (Channel Restart)
- Disable further invocations of this channel.
- Generate an interrupt request to the CPU or to the PCP itself.

When the EXIT instruction is executed, the following sequence occurs:

- If EC = 1 is specified Counter R6.CNT1 is decremented and the CN1Z flag is updated.
- If ST = 1 is specified bit R7.CEN (Channel Enable) is cleared (i.e. the channel is disabled).
- If EP = 0 is specified ***or*** PCP_CS.RCB = 1 (Channel Restart Mode has been selected) the PCP program counter to be saved to context location CR7.PC is set to the appropriate channel entry table address. If EP = 1 is specified ***and*** PCP_CS.RCB = 1 (Channel Resume Mode has been selected) the PCP program counter to be saved to context location CR7.PC is set to the address of the instruction immediately following the EXIT instruction.

- If INT = 1 is specified ***and*** the specified condition cc_B is True, then an interrupt request is raised according to the SRPN value held in R6.SRPN. The interrupt is asserted via one of the PCP_SRC*x* registers, where *x* is determined by the combination of the value of R6.TOS and the list of free entries. This allows the conditional creation of a service request to the CPU or PCP with the SRPN value indicated in register R6.SRPN.
- The channel program's context (including all register modifications caused within this EXIT sequence) is saved to the appropriate region in the PRAM Context Save Area. Depending on the value of PCP_CS.CS, either a Full, Small, or Minimum Context save is used.

Note: Particular attention must be paid to the values of R6 and R7 prior to execution of the EXIT instruction. When posting an interrupt request, one must ensure that R6.SRPN and R6.TOS contain the correct values to generate the required interrupt request. When using the Outer Loop Counter (CNT1), one must ensure the value in R6.CNT1 will provide the required function. When using interrupt priority management, one must ensure the R6.CPPN contains the interrupt priority with which the channel is to run on next invocation. If the channel is to be subsequently re-invoked, one must ensure that the Channel Enable Bit (R7.CEN) is set.

17.4.3.2 Exit as a Result of an Interrupt

If all requirements are in place for a Channel to be interrupted and an SRPN which is of higher value than the currently posted CPPN wins arbitration then an interrupt occurs. On interrupt the following sequence of events occurs:

- The PC (R7.PC) is stored such that code execution will continue from the correct address on return (note that this is always the case regardless of the value of CS.RCB). This "return address" is the next instruction address ("NextPC") unless the interrupt is serviced during execution of a BCOPY or COPY instruction configured (via CNC) to use an outer loop counter (all other instructions complete their operation before an interrupt can be serviced), in which case the PC remains at the current instruction address.
- The channel number (SRPN) of the currently executing channel is recorded to allow the correct context to be loaded on channel resumption.
- The priority (CPPN) of the currently executing channel is recorded to allow proper interrupt arbitration to be performed to decide when the channel should be resumed.
- A "Suspended Interrupt" request is raised in the PSRN to allow operation of the channel to be resumed at the appropriate time.
- The context is saved back to the PRAM Context Save Area. Depending on the chosen context size (PCP_ES.CS) a Full, Small, or Minimum Context save is performed.

The PCP Core may now begin the Channel Start Sequence for the new channel.

17.4.3.3 Error Condition Channel Exit

PCP error conditions can occur for a variety of reasons (e.g. an invalid operation code was executed by a Channel Program, or an FPI Bus error occurred). When an error condition occurs, the PCP Error Status register (PCP_ES) is updated to reflect the error and the Channel Program is aborted. The error exit sequence is as follows:

- The channel enable bit R7.CEN is cleared. This means the channel program will be unable to restart until another FPI Bus master has re-configured the channel program's stored context to set CR7.CEN to 1 again.
- The PC of the instruction which was executing when the error occurred is stored in PCP_ES.EPC.
- The number of the channel program which was executing when the error occurred is stored in PCP_ES.EPN.
- The error type is set in the appropriate field of register PCP_ES.
- The context is saved back to the PRAM Context Save Area. Depending on the chosen context size (PCP_ES.CS) a Full, Small, or Minimum Context save is performed.
- If the error condition was not due to an FPI Bus error, then an interrupt request to the CPU is generated with the priority number stored in register PCP_CS.ESR.

The repetitive posting of PCP Error Interrupts will not cause an overwhelming number of interrupts to the CPU. In this situation, the PCP's CPU service request queue will quickly fill and force the PCP to stall until the CPU can resolve the situation.

Note: An error condition (other than an FPI Bus error) will result in an interrupt being sent to the CPU. This interrupt routine which responds to this interrupt must be capable of dealing with the cause as recorded in PCP_ES, and it must be able to restore the channel program to operation. This minimum required to restart the channel; [program is to set the context value of CR7.CEN = 1.

17.4.4 Debug Exit

If the DEBUG instruction is programmed to stop the channel program execution (SDB = 1 has been specified), the PCP performs an exit sequence which is very similar to the error exit sequence, with the exception that no interrupt request to the CPU is generated. This sequence is:

- If RTA = 0, then the channel enable bit R7.CEN is cleared optionally according to the SDB setting. This means the channel program will be unable to restart until another FPI Bus master has reconfigured the channel program's stored context to set CR7.CEN to 1 again. Otherwise, the R7.CEN bit remains unchanged and the PC is decremented (such that it points to the DEBUG instruction)
- If EDA = 1, a break-point event is generated
- If DAC = 1, then the PCP_CS.EN bit is cleared. This means that the PCP will not execute any further channel programs until the PCP_CS.EN bit is set by another FPI Bus master.

- The address of the DEBUG instruction (i.e. the current PC) is stored in register PCP_ES.PC.
- The current channel number is stored in register PCP_ES.PN.

The execution of the current channel program is stopped at the point of the DEBUG instruction. This instruction only disables the current channel, the PCP will continue to operate, accepting service requests for other channels as they arise.

Note: The DEBUG instruction must be only used in DEBUG mode. Otherwise, an “Illegal Operation” (IOP) error will be generated.

17.5 PCP Interrupt Operation

The PCP Interrupt Control Unit (PICU) and the PCP’s Service Request Nodes (PCP_SRC0..11) are similar to the CPU’s ICU and all other SRNs in the system. They do, however, have some special characteristics, which are described in the following sections. **Figure 17-11** shows an overview of the PCP interrupt scheme.

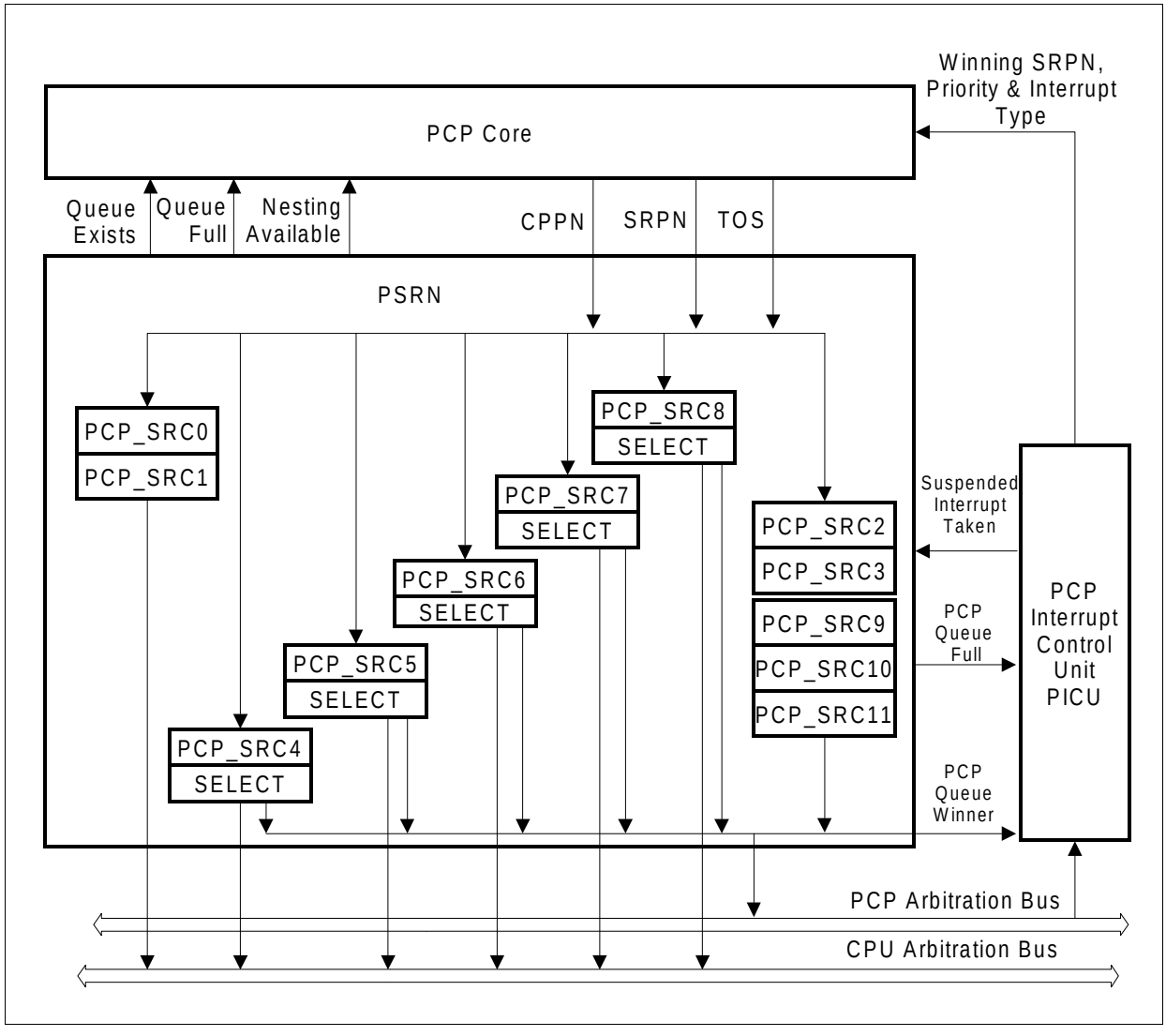


Figure 17-11 PCP Interrupt Block Diagram

17.5.1 Issuing Service Requests to CPU or PCP

The PCP may use one of three mechanisms to raise an interrupt request to the CPU or itself. The first, and most inefficient, method is where a PCP channel program issues service requests by performing an FPI Bus write operation to an external service request node (SRN). Alternatively the PCP can raise a Service Request using one of its own internal SRNs. An interrupt can only be generated by the PCP via an internal SRN when executing an EXIT instruction or when an error condition occurs. In the following descriptions, PCP service requests triggered through an EXIT instruction or the suspended operation are called “implicit” PCP service requests to distinguish them from the “explicit” way of generating a service request through an FPI Bus write to a service request node external to the PCP.

17.5.2 PCP Interrupt Control Unit

The Interrupt Control Unit of the PCP, PICU, operates in a similar manner to the Interrupt Control Unit, ICU, of the CPU. The PICU manages the PCP service request arbitration bus and handles the communication of service requests and priority numbers to and from the PCP kernel. The PCP_ICR register is provided to control and monitor the arbitration process.

When one or more service requests to the PCP are activated, the PICU performs an arbitration round to determine the request with the highest priority. It then places the priority number of this “winning” service request into the PIPN field of register PCP_ICR and generates a service request to the PCP kernel.

If the PCP kernel is currently busy processing a channel program, the new request is left pending until the current channel program has finished.

When the PCP kernel is ready to accept a new service request, it calculates the context start address from the Pending Interrupt Priority Number, PIPN, stored in register ICR and begins with the context restore. It notifies the PICU of the acceptance of this request, and in turn the PICU acknowledges the winner of the last arbitration round. This service request node then resets its Service Request Flag, SRR.

There is one special condition where the PICU operates differently to the CPU Interrupt Control Unit. This special operation is described in **Section 17.5.5.1**.

The PCP interrupt arbitration can be adapted to the application’s needs and characteristics through controls in register PCP_ICR. Bit field PCP_ICR.ARBCYC controls the number of arbitration cycles per arbitration round (one through four cycles), while bit PCP_ICR.ONECYC controls whether one arbitration cycle equals one or two system (FPI) clock cycles.

17.5.3 PCP Service Request Nodes (PSRN)

The PCP contains twelve service request nodes including twelve service request control registers, PCP_SRC0..11, which are provided for implicit PCP service requests. The service request control registers differ from standard SRC registers in that they are fully controlled by the PCP kernel. They are read-only registers during PCP operation. One can not generate interrupts by writing to them.

The twelve service request nodes are split into four groups.

The first group, containing registers PCP_SRC0 and PCP_SRC1, handles implicit PCP service requests targeted to the CPU. The Type of Service control fields, TOS, of these registers hard-wired to 00_B, directing the requests to the CPU.

The second group, registers PCP_SRC2 and PCP_SRC3, handles the service requests targeted to the PCP itself. The respective TOS field of these registers are hard-wired to 01_B, directing the requests to the PCP.

The third group, registers PCP_SRC4, PCP_SRC5, PCP_SRC6, PCP_SRC7 and PCP SRC8, has programmable TOS (Type Of Service) field which allow these registers to be assigned (at configuration time) to any of the available Interrupt Buses.

The last group, registers PCP_SRC9, PCP_SRC10 and PCP_SRC11, is an extended version of a standard Service Request Node. They handle the service requests targeted to the PCP itself which also includes service requests representing a suspended interrupt. The respective TOS field of these registers are hard-wired to 01_B, directing the requests to the PCP.

The service request enable bits, SRE, of the PCP_SRCx registers are hard-wired to 1, meaning these service requests are always enabled.

Note: programming a PCP_SRCx register (x= 4 to 8) with TOS value representing a non-available interrupt bus (10_B or 11_B in the TC11IB)will disable Service Request Node x.

The actual service request flag, SRR, and the service request priority number, SRPN, of the PCP_SRCx registers is updated by the PCP when it generates an implicit service request. The way this is performed is described in the following section.

The twelve service request nodes in the four types described above are implemented as a queue with two entries. When the PCP generates an implicit service request, it places the request into the next available free entry of the appropriate queue rather than writing it into a specific register. Queue management logic automatically ensures proper handling of the queue. In the case where both entries of a queue are filled with pending service requests, the queue management reports this condition to the PCP kernel via a ‘queue full’ signal.

In the following descriptions, the terms “CPU Queue” and “PCP Queue” are used to refer to the queues in the four groups of PCP service request nodes.

17.5.4 Issuing PCP Service Requests

The PCP can issue implicit service requests on the execution of an EXIT instruction, when suspending a channel or when an error occurs during a channel program execution. While the service request generation for the EXIT instruction is optional, a service request is always generated when a channel is suspended or an error occurs. Further differences between these three mechanisms are detailed in the following sections.

17.5.4.1 Service Request on EXIT Instruction

An implicit PCP service request is issued when the INT field of the EXIT instruction is set to 1 and the specified condition code, cc_B, of this instruction is true. Such a service request can be issued to either the CPU or to the PCP itself, depending on the programmed value in the TOS field of register R6. The PCP examines the TOS field in register R6 and issues a service request to the appropriate queue of the service request

nodes. Along with this request, it passes the service request priority number stored in the SRPN field of register R6 to the queue. If the queue has a free entry left, the service request flag, SRR, of the associated service request register, PCP_SRCx, will be set, and the service request priority number will be written to the SRPN field of the SRC register. Please see **Section 17.5.5.1** for the case there is no free entry in the queue.

Because the desired service request is programmed through the TOS and SRPN fields in register R6, each channel program can issue its individual service request. Note that this register needs to be programmed properly if a service request is to be generated by the EXIT instruction.

17.5.4.2 Service Request on Suspension of Interrupt

An implicit PCP service request is issued when the PCP suspends execution of the ongoing channel program in favour of a service request with a higher priority. Such a service request is always issued to the PCP's own interrupt bus and is stored in one of the three extended Service Request Nodes (PCP_SRC9, PCP_SRC10 and PCP_SRC11). Along with this request, it passes the current channel operating priority (CPPN) as a Service Request Priority Number and also the channel number (the original SRPN). The service request flag, SRR, and the Restart Request flag, RRQ, of the associated service request register, PCP_SRCx, will be set. The Operating Priority will be written to the SRPN field and the Channel Number will be written to the SRNC field of the SRC register.

Use of the Operating Priority as the SRPN for resumption of the channel program ensures that during following arbitration rounds the PCP will resume execution of the suspended channel program at the appropriate time.

The PCP treats an interrupt request with the RRQ bit set in a special fashion. In this case, the PCP clears the interrupt request bit in the appropriate internal Service Request Node but does not issue an interrupt acknowledge to any external nodes. This prevents the unwanted clearing of external service requests with an SRPN that matches the priority of a suspended channel.

Note: The PCP will only suspend channel operation when there are two or more free service Request Nodes with the appropriate TOS value for the PCP and one of the free Service Request Nodes is an Extended Service Request Node. This allows for the posting of an interrupt request to the PCP on exit from the new channel program.

17.5.4.3 Service Request on Error

While a service request triggered through an EXIT instruction is optional and can be issued either to the CPU or to the PCP itself, a service request due to an error condition will always be automatically issued and will always be directed to the CPU. The PCP issues a service request to the CPU queue of the service request nodes. Along with this

request, it passes the service request priority number stored in the ESR field of register PCP_CS to the queue. If the queue has a free entry left, the service request flag, SRR, of the associated service request register, PCP_SRCx, will be set, and the service request priority number will be written to the SRPN field of the SRC register. See **Section 17.5.5.1** for the case when there is no free entry in the queue.

Due to the fact that the priority number is stored in the global control register PCP_CS, all channel programs share the same service request routine in case of an error. The exact cause of the error and the channel number of the program which was executed when the error occurred can be determined through examination of the contents of the Error/Debug Status Register, PCP_ES.

17.5.5 Queue Management Control and Status Logic

The PSRN Queue Management Control and Status Logic allocates Interrupt Requests posted by the PCP Core to the appropriate SRNs and also generates Status Signals to ensure the correct operation of the PCP. It should be noted that the PCP Core operates from the viewpoint of TOS (Type Of Service) while the PSRN has fixed interrupt ports. The Queue Management Logic is responsible for handling the transfer of any interrupt request raised by the PCP, with its associated TOS value, to an SRN connected to the appropriate interrupt bus (as defined by the TOS Mapping Input).

17.5.5.1 Queue Full Operation

Queuing the implicit service requests typically allows the PCP to continue with the next service request without stalling. The depth of the queue and the number of channel programs using them determines the stall rate. Depending on the selected service provider (via R6.TOS in case of an EXIT interrupt or always to the CPU in case of an error interrupt) the request is routed to a free entry in the appropriate queue.

If no free entries are available in a queue at the time the PCP wants to post a request to that queue, the PCP is forced to stall until an entry becomes clear. This ensures that the PCP does not lose any interrupts. An entry in a queue becomes free when its service request flag, SRR, is cleared through an acknowledge from the PICU (that is, the CPU or PCP, as appropriate, has started to service this request).

One special case needs to be resolved for the PCP related queue through special operation of the PICU. Consider the case where the PCP queue is full, meaning registers PCP_SRC2, PCP_SRC3 and PCP_SRC9 to PCP_SRC11 are already loaded with pending service requests to the PCP. If the PCP kernel now needed to post an additional service request into that queue, a deadlock situation would be generated: The PCP would stall, since there is not a free entry in the PCP queue in which to place the request. In turn, as the PCP is stalled, it cannot accept new service requests and so the PCP service request queue cannot be emptied. This would result in a deadlock of the PCP.

To avoid such a deadlock, the PICU performs a special arbitration round as soon as the PCP queue becomes full. In this arbitration round, only the service request nodes assigned to the PCP queue are allowed to participate; all service requests from nodes external to the PCP are excluded, regardless of whether their priorities are higher or lower than those of the PCP queue. In this way, it is guaranteed that one entry in the PCP queue gets serviced, freeing one slot in the queue.

The PCP programmer needs to carefully consider this special operation. It ensures that deadlocks are avoided, but it implies that if too many PCP channel programs post service requests to the PCP (self-interrupt), the PCP will have to service these rather than outside interrupt sources. Depending on the priority given to these requests, this could undermine an otherwise appropriate use of the interrupt priority scheme. It is recommended to design the system such that in most cases, high priority numbers can be assigned to these self-interrupts, such that they can win normal arbitration rounds, avoiding the situation where the PCP queue becomes full.

Note: If the CPU queue is full, the PCP can continue to operate until it needs to post another service request to the CPU queue.

17.6 PCP Error Handling

The PCP contains a number of fail-safe mechanisms to ensure that error conditions are handled gracefully and predictably. In addition to providing an extra level of system robustness suitable for high integrity and safety critical systems these mechanisms can often ease the task of finding programming errors during the development process. Whenever an error is detected the channel program that was executing exits and the PCP_ES register is updated with information to allow determination of the error that occurred, the instruction address and the channel program that was executing when the error occurred (see **Section 17.4.3.3**).

17.6.1 Enforced PRAM Partitioning

As previously discussed PRAM can be considered as being split into two distinct areas. The lower of these two areas is the Context Save Area (see **Section 17.3.2.2**) used for storing context information for each active channel while the channel program is not actually executing. The remainder of PRAM is available for general use and is typically used to hold variables and global data.

The default configuration of the PCP allows the PCP to use PRAM as a single area. While this default configuration allows complete flexibility regarding the use of PRAM, this flexibility also introduces the possibility of invalid PCP operation as a result of the following issues:

- Any channel program is allowed to write to any PRAM location (including any location in the CSA). This means that a channel program may be inadvertently programmed to corrupt the context save region belonging to another channel causing invalid operation of the corrupted channel when it next executes.

- Generation of an interrupt request to the PCP with a priority number that would cause loading of a context from outside the CSA will cause the spurious execution of a channel program with an invalid context loaded from outside the CSA.

To avoid spurious PCP operation as a result of either of these programming errors, the PCP can be optionally configured via the global PCP control register (PCP_CSA) to enforce strict partitioning of PRAM. PRAM partitioning is selected by programming PCP_CS.PPE = 1 and the size of the CSA in use is selected via the PCP_CS.PPS bit field (see **Section 17.11.2**). When PRAM partitioning has been enabled a PCP Error will be generated on either of the following events:

- A channel program executes a PRAM write instruction with a target area within the CSA. This prevents a channel corrupting the context save region of any other channel.
- An incoming interrupt request causes the PCP to attempt to load a context from outside the CSA. This prevents the PCP from running an invalid channel program as a result of an invalid interrupt request.

Note: Enabling PRAM partitioning (PCP_CS.PPE = 1) with a CSA size of zero (PCP_CS.PPS = 0) is an invalid setting and will cause a PCP Error Event whenever any interrupt request is received by the PCP.

17.6.2 Channel Watchdog

The Channel Watchdog is a PCP internal watchdog which optionally allows the user to ensure that the PCP will not become locked into executing a single channel due to an endless loop or unexpected software sequence. As each channel executes the PCP maintains an internal count of the number of instructions that have been read from CMEM since the channel started. If the watchdog function is enabled (by programming PCP_CS.CWE = 1) and the internal instruction fetch counter reaches the threshold programmed by the user (programmed via PCP_CS.CWT) then a PCP Error is generated. The threshold setting (PCP_CS.CWT) is global to all channels. From this it follows that the threshold must be selected to be greater than the maximum number of instructions that can be fetched by any channel program, taking all channels into consideration. It should be noted that the instruction width of the PCP is 16 bits and that therefore execution of an instruction that is encoded into 32 bits (e.g. LDL.IL) will generate two CMEM instruction reads, and therefore will cause the internal watchdog counter to be incremented twice.

Note: Enabling the Channel Watchdog function (PCP_CS.CWE = 1) with a threshold of zero (PCP_CS.CWT = 0) is an invalid setting and will cause a PCP Error Event whenever any interrupt request is received by the PCP.

17.6.3 Invalid Opcode

The PCP includes the Invalid Opcode mechanism to check that each instruction fetched from CMEM represents a legal instruction. If the PCP attempts to execute an illegal instruction, then a PCP Error is generated.

17.6.4 Instruction Address Error

An Instruction Address Error is generated if the PCP attempts to execute an instruction from an illegal address. An address is considered to be illegal if:

- The address is outside the available CMEM area (see **Section 17.15** for the CMEM size implemented in this derivative)

and/or

- The specified address could not be contained in the 16 bit PC (i.e. an address calculation yielded a 16 bit unsigned overflow).

The second of these cases can result from an address calculation either from the execution of a PC relative jump instruction (either a JC, JC.I, or JL instruction) or the PC being incremented following execution of the previous instruction.

17.7 PCP Reset

The PCP can be reset in two ways:

17.7.1 Hard Reset

Hard Reset is forced when the reset signal is asserted. This forces a reset of all PCP blocks. The effect of Hard Reset is to:

- Halt any operating channel.
- Reset all control registers to their default values.
- Reset the PCP Processing Core to the default state.
- Reset the FPI Bus Interface.

17.7.2 Soft Reset

Soft Reset is generated by setting bit RST of PCP_CS. Soft reset differs from hard reset in that the interrupt system is not reset. Specifically all interrupt nodes are not completely reset, and the PCP Interrupt Control Unit (PICU) is also not affected. This strategy prevents loss of synchronization between any interrupt node and the Arbitration Unit (ICU) to which it is connected due to one, but not the other being reset. The effect of Soft Reset is to:

- Halt any operating channel.
- Reset all control registers except PCP_ICR and PCP_SRCx registers to their default values.

- Clear all SRR and RRQ bits in all PCP_SRCx registers (all other bits are unchanged).
- Reset the PCP Processing Core to the default state.
- Reset the FPI Bus Interface.

Note: As the PCP is closely coupled to the FPI bus, it is impossible to clearly partition the domains that are affected by reset. Generation of a Soft Reset is likely to cause an FPI bus error as the PCP's interface to the FPI is also reset. Based on this, it is recommended that Soft Reset should not be used, but that a System Wide Hard Reset should be generated instead. Soft Reset may still however prove useful in a debugging environment. In case of Soft Reset to be used, the PCP must be in idle state, i.e. PCP is not active on FPI bus as a slave or a master.

17.8 Instruction Set Overview

The following subsections present an overview of the instruction set and the available addressing modes of the PCP in the TC11IB.

17.8.1 DMA Primitives

Table 17-3 DMA Transfer Instructions

DMA Transfer	COPY	Move value from FPI Bus source address location to FPI Bus destination address location. Optionally increment or decrement source and destination pointer registers. Optionally repeat instruction until counter CNT1 reaches 0.
	BCOPY	Move a block of data from FPI Bus source address location to FPI Bus destination address location. Optionally increment or decrement source and destination pointer registers. Counter CNT0 controls the block size and CNT1 implements automatic multiple block transfers.

17.8.2 Load and Store

Note: If a conditional instruction's condition code is false, the operation will be treated as a "No Operation". Register values will not be changed and the flags will not be updated.

Note: The use of R7 as a PRAM offset with the LD.P and ST.P instructions is not allowed. The PCP will generate an illegal Opcode Error Exit if this instruction is encountered using R7 as a PRAM offset.

Table 17-4 Load and Store Instructions

Load	LD.F	Load value from FPI Bus address location into register (FPI Bus address = register content)
	LD.I	Load immediate value into register
	LD.IF	Load value from FPI Bus address location into register (FPI Bus address = register content + immediate offset)
	LD.P	Load value from PRAM address location into register (PRAM address = DPTR + register offset)
	LD.PI	Load value from PRAM address location into register (PRAM address = DPTR + immediate offset)
	LDL.IL	Load 16-bit immediate value into lower bits [15:0] of register
	LDL.IU	Load 16-bit immediate value into upper bits [31:16] of register
Store	ST.F	Store register value to FPI Bus address location (FPI Bus address = register content)
	ST.IF	Store register value to FPI Bus address location (FPI Bus address = register content + immediate offset)
	ST.P	Store register value to PRAM address location (PRAM address = DPTR + register offset)
	ST.PI	Store register value to PRAM address location (PRAM address = DPTR + immediate offset)
Move	MOV	Conditionally move register value to register

17.8.3 Exchange Instructions

Table 17-5 Exchange Instructions

Exchange	XCH.F	Exchange contents between FPI Bus address location and register
	XCH.PI	Exchange contents between PRAM address location and register

17.8.4 Arithmetic and Logical Instructions

Arithmetic instructions that are fully register-based execute conditionally depending on the specified Condition Code A. All other arithmetic instructions such as PRAM (.PI), indirect (.I), and FPI (.F and .IF) execute unconditionally.

Note: If a conditional instruction's condition code is false, the operation will be treated as a "No Operation". Register values will not be changed and the flags will not be updated.

Table 17-6 Arithmetic Instructions

Add	ADD	Add register to register (conditionally)
	ADD.I	Add immediate value to register
	ADD.F	Add content of FPI Bus address location to register (Byte, Half-word or Word)
	ADD.PI	Add content of PRAM address location to register
Subtract	SUB	Subtract register from register (conditionally)
	SUB.I	Subtract immediate value from register
	SUB.F	Subtract content of FPI Bus address location from register (Byte, Half-word or Word)
	SUB.PI	Subtract content of PRAM address location from register
Compare	COMP	Compare register to register (conditionally)
	COMP.I	Compare immediate value to register
	COMP.F	Compare content of FPI Bus address location to register (Byte, Half-word or Word)
	COMP.PI	Compare content of PRAM address location to register
Negate	NEG	Negate register (2's complement, conditionally)

Logical instructions that are fully register-based execute conditionally as determined by the specified Condition Code A. All other logical instructions, such as PRAM (.PI), indirect (.I), and FPI (.F and .IF) execute unconditionally.

Table 17-7 Logical Instructions

Logical And	AND	Register AND register (conditionally)
	AND.F	Content of FPI Bus address location AND register (Byte, Half-word or Word)
	AND.PI	Content of PRAM address location AND register
Logical Or	OR	Register OR register (conditionally)
	OR.F	Content of FPI Bus address location OR register (Byte, Half-word or Word)
	OR.PI	Content of PRAM address location OR register
Logical Exclusive-Or	XOR	Register XOR register (conditionally)
	XOR.F	Content of FPI Bus address location XOR register (Byte, Half-word or Word)
	XOR.PI	Content of PRAM address location XOR register
Logical Not	NOT	Invert register (1’s complement, conditionally)
Shift	SHL	Shift left register
	SHR	Shift right register
Rotate	RL	Rotate left register
	RR	Rotate right register
Prioritize	PRI	Calculate position of first set bit (1-bit) in register, from left
Memory Read/ Modify/ Write	MCLR.PI	Clear specified bits within a PRAM address location
	MSET.PI	Set specified bits within a PRAM address location

Note: If a conditional instruction’s condition code is false, the operation will be treated as a “No Operation”. Register values will not be changed and the flags will not be updated.

17.8.5 Bit Manipulation

All bit manipulation instructions except INB are executed unconditionally. If conditional bit handling is required, INB should be used.

Table 17-8 Bit Manipulation Instructions

Set Bit	SET	Set bit in register
	SET.F	Set bit in FPI Bus address location
Clear Bit	CLR	Clear bit in register
	CLR.F	Clear bit in FPI Bus address location
Insert Bit	INB	Insert carry flag into register (position given by content of a register)
	INB.I	Insert carry flag into register (position given by immediate value)
Check Bit	CHKB	Set carry flag depending on value of specified register bit

17.8.6 Flow Control

Table 17-9 Flow Control Instructions

Jump	JC	Jump conditionally to PC + short immediate offset address
	JC.A	Jump conditionally to immediate absolute address
	JC.I	Jump conditionally to PC + register offset address
	JC.IA	Jump conditionally to register absolute address
	JL	Jump unconditionally to PC + long immediate offset address
Exit Channel	EXIT	Unconditionally exit channel program execution (optionally generate interrupt request and/or inhibit channel)
No Operation	NOP	Low-power No-Operation
Debug	DEBUG	Conditionally generate debug event (optionally stop channel program execution)

Note: If a conditional instruction's condition code is false, the operation will be treated as a "No Operation". Register values will not be changed and the flags will not be updated.

17.8.7 Addressing Modes

The PCP needs to address locations in memory in different ways, as determined by the type of memory being accessed and the type of action being performed on that location.

17.8.7.1 FPI Bus Addressing

The PCP performs all FPI system bus accesses in Supervisor mode. All FPI Bus accesses from the PCP are indirect to some extent. The main indirect addressing on the FPI Bus uses a 32-bit absolute address located in the general purpose register indicated in the instruction. This address must be properly aligned for the type of data access — byte, half-word or word. If it is not aligned, the results are undefined.

– Effective Target Address [31:0] = <R[*a*]>

where *a* is the number of the register, for instance, R2. Instructions using this address mode are indicated through the ".F" suffix.

For indirect-plus-immediate addressing on the FPI Bus, the 32-bit absolute address located in the general purpose register indicated in the instruction is added to the immediate 5-bit offset value encoded in the instruction. This address must be properly aligned for the type of data access (byte, half-word or word). If it is not aligned, the results are undefined.

– Effective Target Address [31:0] = <R[*a*]> + #offset5

where *a* is the number of the register and #offset5 is a 5-bit immediate offset value. Instructions using this addressing mode are indicated through the ".IF" suffix (only available for load and store, LD.IF and ST.IF).

This addressing mode is particularly useful for managing peripherals, where the peripheral base address is held in R[*a*], and the offset can index directly into a specific control register.

The BCOPY and COPY instructions use the indirect absolute addressing with predefined PCP registers. Register R4 is used as the source address pointer, while R5 represents the destination address pointer.

– Effective Source Address [31:0] = <R4>
– Effective Destination Address [31:0] = <R5>

Note: All FPI Bus accesses by the PCP are performed in Supervisor mode.

Note: The PCP is not allowed to access its own registers via instructions executed in the PCP.

17.8.7.2 PRAM Addressing

The PRAM is always addressed indirectly by the PCP. The normal address used is the value of the R7.DPTR field (8 bits) concatenated with an immediate 6-bit offset value encoded in the instruction, yielding a 14-bit word address. This enables access to 16 KWords (64 KBytes). Because R7.DPTR is part of a channel program's context, a channel program may alter the DPTR value at any time.

– Effective PRAM Address[13:0] = <R7.DPTR> << 6 + #offset6

Instructions using this addressing mode are indicated through the ".PI" suffix.

To provide effective indexing into large tables or stores of data, an alternate form of indirect addressing can also be used on load and store operations to PRAM. The value of the DPTR field (8 bits) is concatenated with the least significant 6 bits of R[*a*], again yielding a 14-bit word address. The most significant bits [31..6] of R[*a*] are ignored.

– Effective PRAM Address[13:0] = <R7.DPTR> << 6 + <R[*a*][5:0]>

Instructions using this addressing mode are indicated through the ".P" suffix (load and store only, LD.P and ST.P).

17.8.7.3 Bit Addressing

Single bits can be addressed in the PCP general purpose registers or in FPI Bus address locations. A 5-bit value indicates the location of a bit in the register specified in the instruction. This bit location is either given through an immediate value in the instruction or through the lower five bits of a second register (indirect addressing).

– Effective Bit Position[0..31] = #imm5
– Effective Bit Position[0..31] = <R[a][4:0]>

The immediate bit addressing is used by instructions SET and CLR and their variants as well as by INB.I and CHKB. Indirect bit addressing is used by the INB instruction only.

17.8.7.4 Flow Control Destination Addressing

The Jump instructions are split into two groups: PC-relative jumps and jumps to an absolute address.

For PC-relative jumps, the destination address is a positive or negative offset from the PC of the next instruction. The offset is either contained in the lower 16 bits of a register (the upper 16 bits are ignored), or is given as immediate value of the instruction. The immediate values are sign-extended to 16 bits. If the effective jump address is outside the available CMEM area (or the jump address calculation caused an overflow), then a PCP Error Condition has occurred.

– Effective JUMP Address[15:0] = NextPC + Signed(R[*a*][15:0]); +/- 32K instructions
– Effective JUMP Address[15:0] = NextPC + Sign-Extend(#offset10);
  +/- 512 instructions

– Effective JUMP Address[15:0] = NextPC + Sign-Extend(#offset6);
  +/- 32 instructions

The function NextPC indicates the instruction that would be fetched next by the program counter. Instructions using this addressing are JL, JC and JC.I.

For absolute addressing, the actual address in code memory where program flow is to resume is either an immediate value #imm16 in the code memory location immediately following the Jump instruction, or it is contained in the lower 16 bits of a register. If the value is greater than the PC size implemented, an error condition has occurred.

– Effective JUMP Address[15:0] = #imm16
– Effective JUMP Address[15:0] = <R[a]>

Instructions using these addressing modes are JC.A (immediate absolute address) and JC.IA (indirect absolute address).

17.9 Accessing PCP Resources from the FPI Bus

Any FPI Bus master can access the three distinct PCP address ranges from the FPI Bus side. Accesses to the PCP control register, the parameter RAM (PRAM), and the code memory (PCODE) are detailed in the following sections. Note that the PCP itself is not allowed to access its control registers through PCP instructions.

17.9.1 Access to the PCP Control Registers

FPI Bus accesses to the PCP control registers must always be performed in Supervisor Mode with word accesses; byte or half-word accesses will result in a bus error.

All PCP control registers can be read at any time. Write operations are only possible to the PCP_CS register, all other register are read-only. Register PCP_CS can be optionally ENDINIT protected via bit PCP_CS.EIE (see **Section 17.11.2**).

17.9.2 Access to the PRAM

FPI Bus accesses to the PRAM must always be performed with word accesses; byte or half-word accesses will result in a bus error.

Attention needs to be paid when accessing the context save areas and data sections of the PCP channel programs. The location of a specific channel's context save area is dependent on the chosen context model, full, small or minimum context. **Table 17-10** shows these addresses.

Table 17-10 FPI Bus Access to Context Save Areas

Channel #	Full Context	Small Context	Minimum Context
0 (see note)	PRAM Base Address + 00_H	PRAM Base Address + 00_H	PRAM Base Address + 00_H
1	PRAM Base Address + 20_H	PRAM Base Address + 10_H	PRAM Base Address + 08_H
2	PRAM Base Address + 40_H	PRAM Base Address + 20_H	PRAM Base Address + 10_H
3	PRAM Base Address + 60_H	PRAM Base Address + 30_H	PRAM Base Address + 18_H
n	PRAM Base Address + $n \times 20_H$	PRAM Base Address + $n \times 10_H$	PRAM Base Address + $n \times 08_H$

Note: Since channel #0 is not defined (no service request with SRPN = 0), the first area is not an actual context save area. It is recommended that this area should not be used by PCP channel programs.

The FPI Bus address of a word location pointed to by the data pointer R7_DPTR is calculated by the following formula:

– Effective FPI Bus address[31:0] = (PRAM Base Address) + (<DPTR> << 6)

17.9.3 Access to the PCODE

FPI Bus accesses to the code memory PCODE must always be performed with word accesses; byte or half-word accesses will result in a bus error.

When using a channel entry table, the FPI Bus address of a specific channel's entry location is given by the following formula:

– Effective FPI Bus address[31:0] = (PCODE Base Address) + $04_H \times$ Channel Numb.

The FPI Bus address of an instruction pointed to by the PCP program counter, PC, is calculated by the following formula:

– Effective FPI Bus address[31:0] = (PCODE Base Address) + <PC> << 1

17.10 Debugging the PCP

For debugging the PCP a special instruction, DEBUG, is provided. This instruction can only be used when the PCP is in Debug Mode. It can be placed at important locations inside the code to track and trace program execution. The execution of the instruction depends on a condition code specified with the instruction. The actions programmed for this instruction will only take place if the specified condition is true.

The following actions are performed when the DEBUG instruction is executed and the condition code is true:

- store the current PC, i.e. the address of the DEBUG instruction, in register PCP_ES.EPC
- store the current channel number in register PCP_ES.EPN

In addition, the following operations can be programmed through fields in the DEBUG instruction:

- optionally stop the channel program execution (instruction field SDB)
- optionally generate an external debug event at pin $\overline{\text{BRKOUT}}$ (instruction field EDA)
- optionally prevent the PCP from executing any further channel programs (instruction field DAC)
- optionally cause the PCP to decrement the PC prior to saving the channel context (instruction field RTA)

If the DEBUG instruction is programmed to stop the channel program execution, the action taken by the PCP depends on the value of the RTA instruction field:

- If RTA == 0, the PCP disables further invocations of the current channel through clearing bit R7.CEN, and then performs a context save. The execution of this channel is stopped at the point of the DEBUG instruction. If the DAC instruction field == 0, the PCP will continue to operate to accept service requests for other channels as they arise. Since the stopped channel was disabled before saving its context, service requests for this channel will result in an error exit. When re-enabling the channel, its enable bit CEN in the saved context location CR7 must be set.
- If RTA == 1, the PCP does not modify bit R7.CEN (i.e. the channel remains enabled), decrements the PC (so that it again points to the DEBUG instruction), and then performs a context save. The execution of this channel is stopped at the point of the DEBUG instruction. If the DAC instruction field == 0, The PCP will continue to operate, accepting service requests for other channels as they arise.Since the stopped channel was not disabled before saving its context, service requests for this channel will not result in an error exit but will simply cause re-execution of the DEBUG instruction and hence a repeat of the channel exit.

Note: When a channel is stopped by DEBUG the context of the stopped channel will be saved to the appropriate region of the CSA before the channel terminates. Where a Small or Minimum Context model is being used the values of the general purpose registers not included in the context will not be saved, and indeed these

register values may be changed by the operation of another active channel. In this case, the required registers should be explicitly saved to PRAM by store instructions prior to execution of the DEBUG instruction.

If the DEBUG instruction is programmed to stop all channel program execution, the PCP disables further invocations of any channel through clearing bit PCP_CS.EN. The execution of this channel is only stopped according to the SDB instruction field value. The PCP will only start to re-accept service requests when PCP_CS.EN is written to 1.

Note: If PCP_CS.RCB = 0 (Channel Resume Mode), then the channel program will begin executing at whatever PC is restored from the context location CR7.PC which will allow a channel program to resume from the instruction following the DEBUG instruction. If PCP_CS.RCB = 1 (Channel Restart Mode), then the channel program is forced to always start at its channel entry table location no matter what the restored context value is for the PC. This means that in Channel Restart Mode it is not possible to restart the channel program from where it was halted by the debug event. It is recommended that when using Channel Restart Mode the user should also program all EXIT instructions with the “EP = 0” setting to allow selection of Channel Resume Mode for debugging without changing operation of the channel programs.

17.11 PCP Registers

The PCP can be viewed as being a peripheral on the FPI Bus. As with any other peripheral, there are control registers, normally set by the CPU acting as an external FPI Bus master to the PCP during initialization. Control registers select the operating modes of the PCP, and status registers provide information about the current state of the PCP to the external Bus master. **Figure 17-12** gives an overview of the PCP registers.

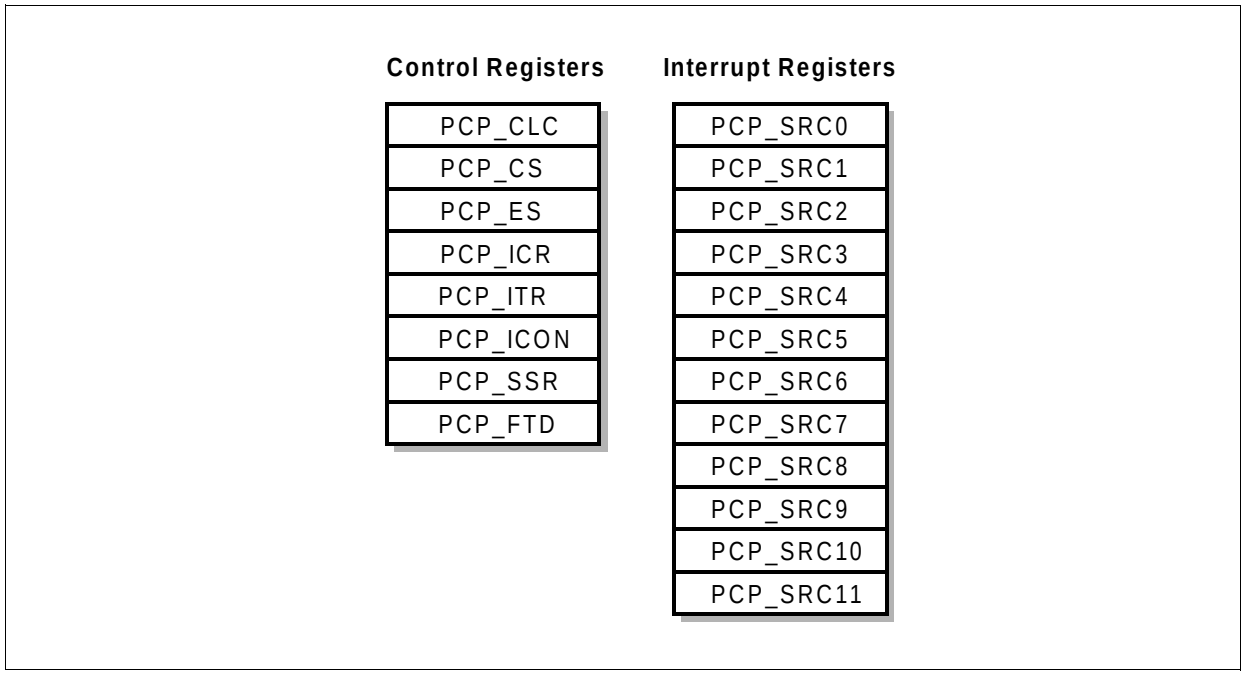


Figure 17-12 PCP Registers

Table 17-11 PCP Registers

Register Short Name	Register Long Name	Offset Address	Description see
PCP_CLC	PCP Clock Control Register	0000_H	**Page 17-51**
PCP_CS	PCP Control/Status Register	0010_H	**Page 17-51**
PCP_ES	PCP Error/Debug Status Register	0014_H	**Page 17-54**
PCP_ICR	PCP Interrupt Control Register	0020_H	**Page 17-56**
PCP_ITR	PCP Interrupt Threshold Control Register	0024_H	**Page 17-57**
PCP_ICON	PCP Interrupt Configuration Register	0028_H	**Page 17-59**
PCP_SSR	PCP Stall Status Register	$002C_H$	**Page 17-60**
PCP_FTD	PCP Feature Disable/Test Register	0030_H	**Page 17-62**
PCP_SRC11	PCP Service Request Control Register 11	$00D0_H$	**Page 17-76**

Table 17-11 PCP Registers

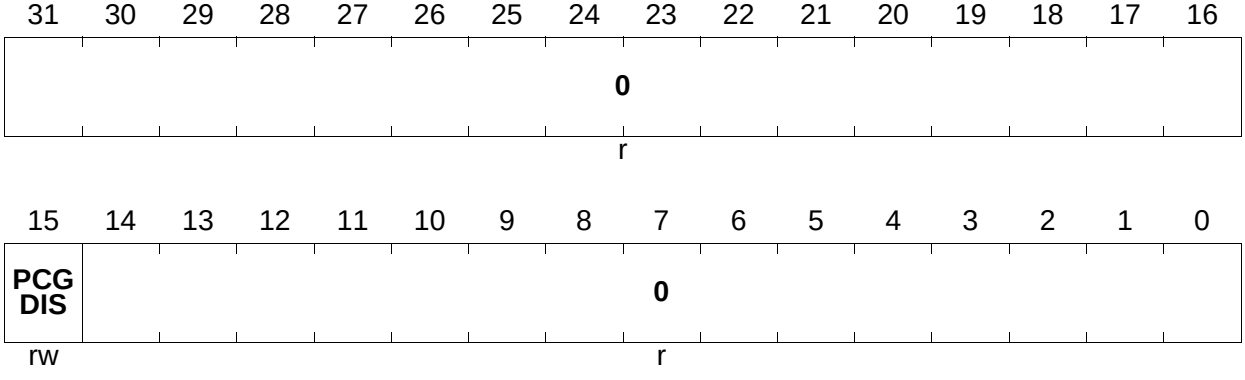
Register Short Name	Register Long Name	Offset Address	Description see
PCP_SRC10	PCP Service Request Control Register 10	$00D4_H$	Page 17-74
PCP_SRC9	PCP Service Request Control Register 9	$00D8_H$	Page 17-72
PCP_SRC8	PCP Service Request Control Register 8	$00DC_H$	Page 17-71
PCP_SRC7	PCP Service Request Control Register 7	$00E0_H$	Page 17-70
PCP_SRC6	PCP Service Request Control Register 6	$00E4_H$	Page 17-69
PCP_SRC5	PCP Service Request Control Register 5	$00E8_H$	Page 17-68
PCP_SRC4	PCP Service Request Control Register 4	$00EC_H$	Page 17-67
PCP_SRC3	PCP Service Request Control Register 3	$00F0_H$	Page 17-66
PCP_SRC2	PCP Service Request Control Register 2	$00F4_H$	Page 17-65
PCP_SRC1	PCP Service Request Control Register 1	$00F8_H$	Page 17-64
PCP_SRC0	PCP Service Request Control Register 0	$00FC_H$	Page 17-63

The control registers are accessible by any master via the FPI Bus. The control registers must be configured at initialization and then left unaltered. This is typically done by the CPU. The only setting that can be dynamically modified is the PCP_CS.EN bit. All other bits must only be modified when PCP_CS.EN == 0 and PCP_CS.RS == 0.

The PCP control and status registers are accessible only to the CPU when it is operating in Supervisor mode. PCP control and status registers must be accessed with 32-bit read and write operations only.

17.11.1 PCP Clock Control Register, PCP_CLC

PCP_CLC
PCP Clock Control Register **Reset Value: 0000 0000$_H$**

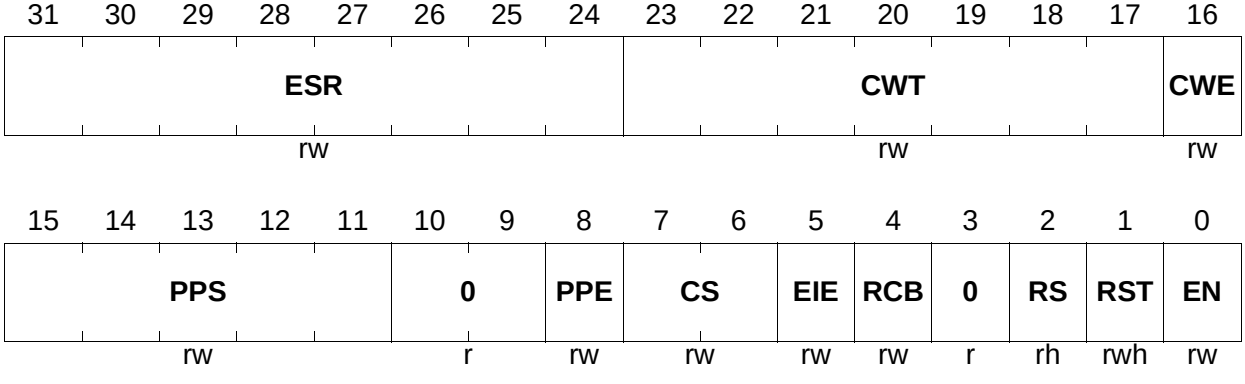


Field	Bits	Type	Description
PCGDIS	0	rw	**PCP Clock Gating Disable** 0 PCP Internal Clock stops when PCP is idle. (default) 1 PCP Internal Clock always runs
0	[31:16] [14:0]	r	**Reserved**; read as 0; should be written with 0.

17.11.2 PCP Control and Status Register, PCP_CS

This register can be ENDINIT-protected via bit EIE.

PCP_CS
PCP Control/Status Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
EN	0	rw	**PCP Enable** 0 PCP is disabled for operation (default) 1 PCP is enabled for operation *Note: This bit does not enable/disable clocks for power saving. It stops the PCP from accepting new service requests.*
RST	1	rwh	**PCP Reset Request** 0 No PCP reset operation is requested 1 PCP reset is requested. Halt any operating channel. Reset all control registers to default values. Reset PCP state to default value. This bit is always read as 0, but is written with 1 to initiate a Software reset.
RS	2	rh	**PCP Run/Stop Status Flag** 0 PCP is stopped or idle (default) 1 PCP is currently running
RCB	4	rw	**Restart Channel at Base** 0 Channel resume operation mode selected; channel start PC is taken from restored context 1 Channel restart operation mode selected; channel start PC is derived from the requested channel number (= priority number of service request) *Note: This is a global control bit and applies to all channels.*
EIE	5	rw	**ENDINIT Enable** 0 Writes to PCP_CS are enabled if ENDINIT-protection is generally enabled (see note) 1 Writes to PCP_CS are disabled if ENDINIT-protection is generally enabled (see note)
CS	[7:6]	rw	**Context Size Selection** 00 Use Full Context for all channels 01 Use Small Context for all channels 10 Use Minimum Context for all channels 11 Reserved

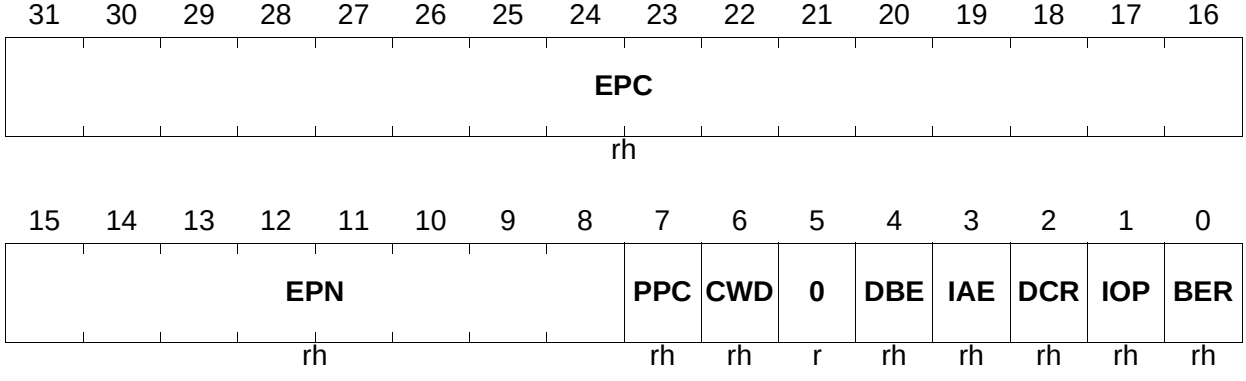
Field	Bits	Type	Description
PPE	8	rw	**PRAM Partitioning Enable** 0 PRAM is not partitioned (Default) 1 PRAM is partitioned *Note: When partitioned, the PRAM is divided into two areas (CSA and reminder). A PCP error will be generated on an inappropriate action in either region (PCP write operation with a target address in the CSA or context restore from outside the CSA).*
PPS	[15:11]	rw	**PRAM Partition Size** 0_H Reserved 1_H CSA contains 9 context save regions ... n_H CAS contains 1 + 8 x n context save regions *Note: The actual size of the CSA (in words) is given by the formula (8 x 'n' + 1) x 'x', where 'n' is the PPS value and 'x' is the number of registers in the selected context model. This setting also controls the number of channels that can be invoked, e.g. setting this field to 1 will give a CSA containing 9 context save regions, channel 0 can not be used so this setting allows the use of 8 channels (Channels 1 to 8). Do not set this field to 0 when PPE = 1 as this will disable all channels.*
CWE	16	rw	**Channel Watchdog Enable** 0 Disable Channel Watchdog 1 Enable Channel Watchdog Checking *Note: When enabled the Channel Watchdog counts the number of instructions executed since the channel started. If this number exceeds the Channel Watchdog Threshold then a PCP error is generated.*
CWT	[23:17]	rw	**Channel Watchdog Threshold** 0_H Reserved 1_H Threshold = 16 instructions ... n_H Threshold = 16 x n instructions

Field	Bits	Type	Description
ESR	[31:24]	rw	**Error Service Request Number** SRPN for interrupt to CPU on an error condition. 00_H No interrupt request posted (default) value n Post n as SRNP interrupt to CPU on an error condition (n not equal 00_H)
0	3, [10:9]	r	**Reserved**; read as 0; should be written with 0.

17.11.3 PCP Error/Debug Status Register, PCP_ES

This is a read-only register, providing state information about error and debug conditions.

PCP_ES
PCP Error/Debug Status Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
BER	0	rh	**Bus Error Flag** Set if the last error/debug event was an error generated by an FPI Bus error or an invalid address access, otherwise clear. *Note: An FPI Bus error event does not cause the PCP to post an error interrupt to the CPU. An FPI Bus error interrupt is however generated by the FPI control logic.*
IOP	1	rh	**Invalid Opcode** Set if the last error/debug event was an error generated by the PCP attempting to execute an Invalid Opcode (i.e. the value fetched from CMEM for execution by the PCP did not represent a valid instruction), otherwise clear.

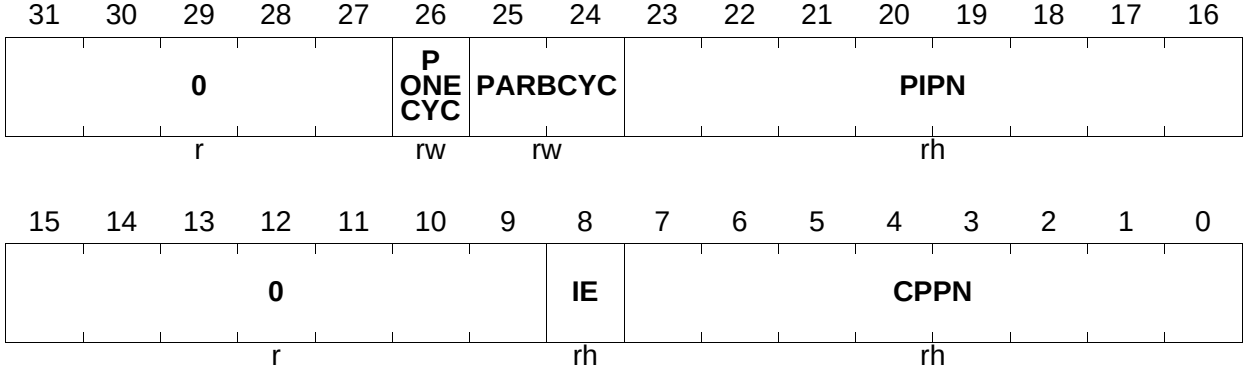
Field	Bits	Type	Description
DCR	2	rh	**Disabled Channel Request Flag** Set if the last error/debug event was an error generated by receipt of an interrupt request with an SRPN that attempted to start a disabled PCP channel, otherwise clear.
IAE	3	rh	**Instruction Address Error** Set if the last error/debug event was an error generated by the PCP attempting to fetch an instruction from an address outside the implemented CMEM range as a result of a jump or branch instruction, otherwise clear.
DBE	4	rh	**Debug Event Flag** Set if the last error/debug event was a debug event. *Note: A debug event does not cause the posting of an interrupt to the CPU.*
CWD	6	rh	**Channel Watchdog Triggered** Set if the last error/debug event was an error generated by a channel program attempting to execute more instructions than allowed by PCP_CS.CWN.
PPC	7	rh	**PRAM Partitioning Check** Set if the last error/debug event was an error generated by a channel program attempting to perform a write to a PRAM address within the Context Save Area or receipt of an interrupt request that would have caused a context restore from outside the CSA.
EPN	[15:8]	rh	**Error Service Request Priority Number** Channel number of the channel that was operating when the last error/debug event occurred. The value stored is the service request priority number which invoked this channel (= channel number), NOT the current PCP priority number stored in field CPPN in register PICR. Default = 00_H
EPC	[31:16]	rh	**Error PC** PC value of the instruction that was executing when an error or debug event occurred. Default = 0000_H.
0	5	r	**Reserved;** read as 0.

Note: An interrupt request with the SRPN held in PCP_CS.ESR is posted to the CPU whenever a PCP error event, other than an FPI Bus error, occurs. FPI Bus error interrupt generation is automatically handled by the FPI Bus control logic rather than by the PCP. The execution of a DEBUG instruction is not classed as an error event and does not therefore generate an interrupt request to the CPU. The entire contents of the register are updated whenever there is a debug or an error event detected (i.e. all status/error bits, other than the bit representing the last PCP error/debug event, are cleared). This register therefore only provides a record of the last error/debug event encountered. The only way to clear this register is to reset the PCP.

17.11.4 PCP Interrupt Control Register, PCP_ICR

This register controls the operation of the PCP Interrupt Control Unit (PICU).

PCP_ICR
PCP Interrupt Control Register **Reset Value: 0000 0000$_H$**



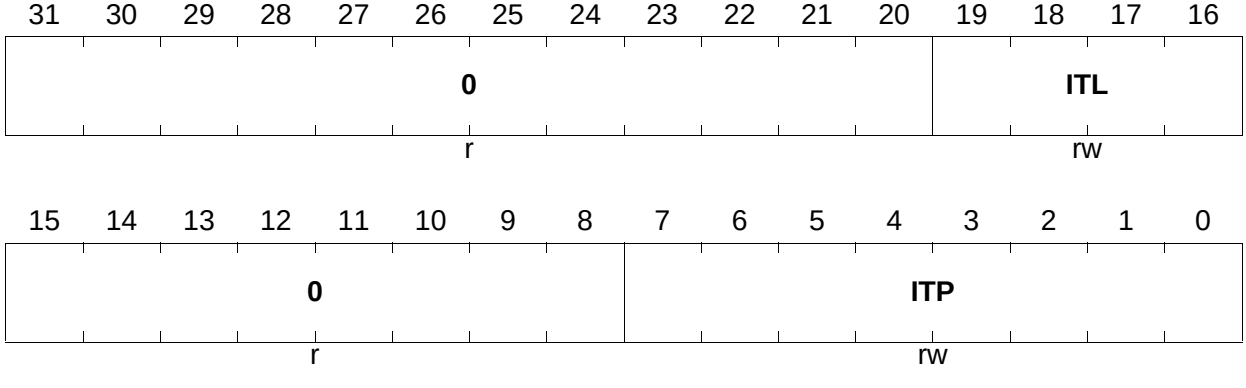
Field	Bits	Type	Description
CPPN	[7:0]	rh	**Current PCP Priority Number** This field indicates the current priority level of the PCP and is automatically updated by hardware on entry into an Interrupt Service Routine.
IE	8	rh	**Interrupt Enable** This status bit is updated by hardware according to the state of the PCP register bit R7.IEN.
PIPN	[23:16]	rh	**Pending Interrupt Priority Number** This read-only field is updated by the PICU at the end of each arbitration process and indicates the priority number of a pending request. PIPN is set to 00_H when no request is pending and at the beginning of a new arbitration process.

Field	Bits	Type	Description
PARBCYC	[25:24]	rw	**Number of Arbitration Cycles Control** This bit field controls the number of arbitration cycles used to determine the request with the highest priority. It follows the same coding scheme as described for the CPU interrupt arbitration. 00 Four arbitration cycles (default) 01 Three arbitration cycles 10 Two arbitration cycles 11 One arbitration cycle
PONECYC	26	rw	**Clocks per Arbitration Cycle Control** This bit determines the number of clocks per arbitration cycle. 0 Two clocks per arbitration cycle (default) 1 One clock per arbitration cycle
0	[15:9], [31:27]	r	**Reserved**; read as 0; should be written with 0.

17.11.5 PCP Interrupt Threshold Register, PCP_ITR

This register specifies the number of active interrupt entries and priority control.

PCP_ITR
PCP Interrupt Thrilled Register **Reset Value: 0000 0000$_H$**



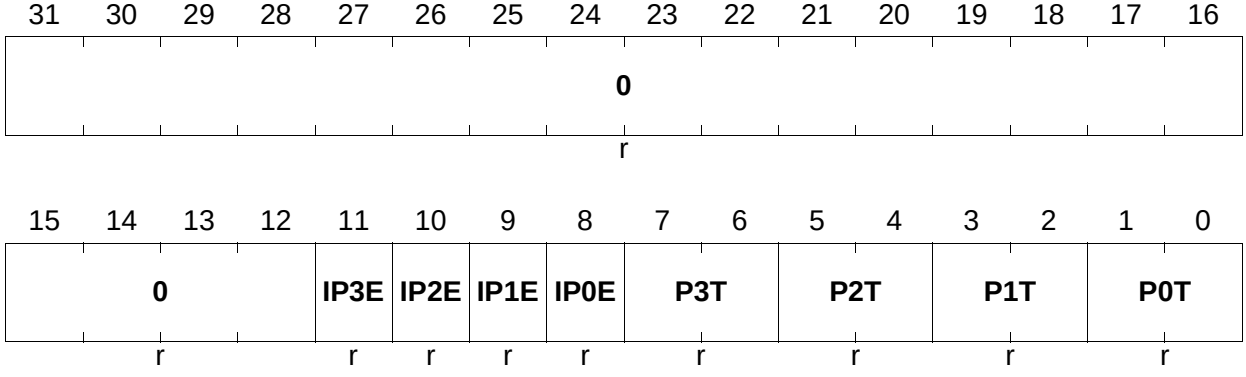
Field	Bits	Type	Description
ITP	[7:0]	rw	**PCP Interrupt Threshold Service Request Priority Number** This field contains the interrupt priority that is to be posted to the interrupt queue associated with Interrupt Port 0 when the threshold condition is reached. (setting this value to 0 or disables the threshold detection mechanism).
ITL	[19:16]	rw	**Interrupt Threshold Level** This bit field specifies the number of active interrupt entries at which an warning interrupt should be issued to the interrupt queue associated with interrupt port 0 (i.e. when the number of active port 0 interrupt requests stored in all SRCx registers reaches this value then an interrupt is posted to port 0 with the priority programmed into the ITP field). When ITL is programmed to 0 or is >= the number of SRCx registers that can contain port 0 interrupt requests then the threshold warning mechanism is disabled).
0	[15:8], [31:20]	r	**Reserved**; read as 0; should be written with 0.

This bit field specifies the number of active interrupt entries at which an warning interrupt should be issued to the interrupt queue associated with interrupt port 0 (i.e. when the number of active port 0 interrupt requests stored in all SRCx registers reaches this value then an interrupt is posted to port 0 with the priority programmed into the ITP field). When ITL is programmed to 0 or is >= the number of SRCx registers that can contain port 0 interrupt requests then the threshold warning mechanism is disabled).

17.11.6 PCP Interrupt Configuration Register, PCP_ICON

This is a read-only register, providing state information about interrupt configuration.

PCP_ICON
PCP Interrupt Configuration Register **Reset Value: 0000 03E4$_H$**



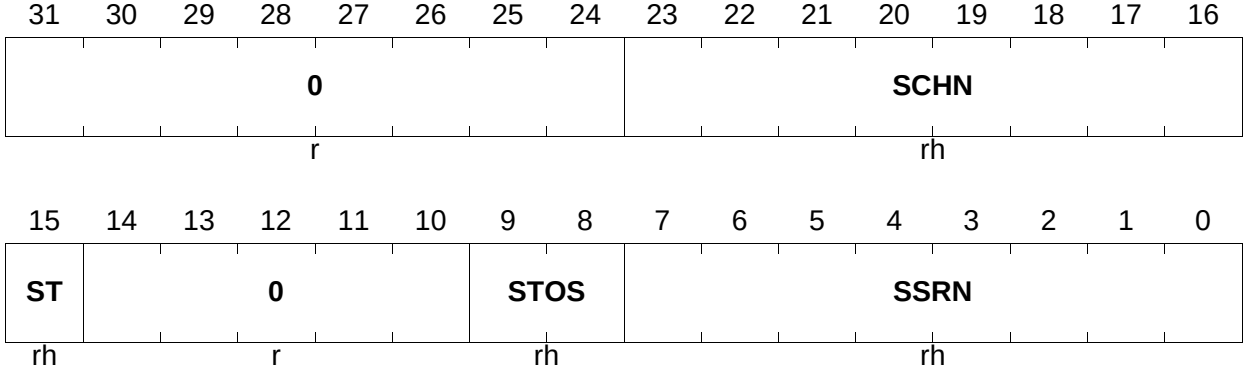
Field	Bits	Type	Description
P0T	[1:0]	r	**PCP Interrupt Bus 0 TOS Mapping** This field reflects the TOS associated with Interrupt Bus 0 (CPU interrupt arbitration bus). The PCP should use this value in R6.TOS when it wishes to raise an interrupt request via interrupt bus 0.
P1T	[3:2]	r	**PCP Interrupt Bus 1 TOS Mapping** This field reflects the TOS associated with Interrupt Bus 1(PCP interrupt arbitration bus). The PCP should use this value in R6.TOS when it wishes to raise an interrupt request to itself (the PCP is always connected to Interrupt Port 1).
P2T	[5:4]	r	**PCP Interrupt Bus 2 TOS Mapping** This field reflects the TOS associated with Interrupt Bus 2. *Note: Interrupt bus 2 is not available in the TC11IB.*
P3T	[7:6]	r	**PCP Interrupt Bus 3 TOS mapping** This field reflects the TOS associated with Interrupt Bus 3. *Note: Interrupt bus 3 is not available in the TC11IB.*
IP0E	8	r	**PCP Interrupt Bus 0 Enable** This bit reflects the status of interrupt bus 0.(CPU interrupt arbitration bus). Interrupt bus 0 is always enabled.

Field	Bits	Type	Description
IP1E	9	r	**PCP Interrupt Bus 1 Enable** This bit reflects the status of interrupt bus 1 (PCP interrupt arbitration bus). Interrupt bus 1 is always enabled.
IP2E	10	r	**PCP Interrupt Bus 2 Enable** This bit reflects the status of interrupt bus 2. Interrupt bus 2 is always disabled (not implemented in the TC11IB).
IP3E	11	r	**PCP Interrupt Bus 3 Enable** This bit reflects the status of interrupt port 3. Interrupt bus 3 is always disabled (not implemented in the TC11IB).
0	[31:12]	r	**Reserved**; read as 0; should be written with 0.

17.11.7 PCP Stall Status Register, PCP_SSR

This register contains the stall status information.

PCP_SSR
PCP Stall Status Register **Reset Value: 0000 0000$_H$**

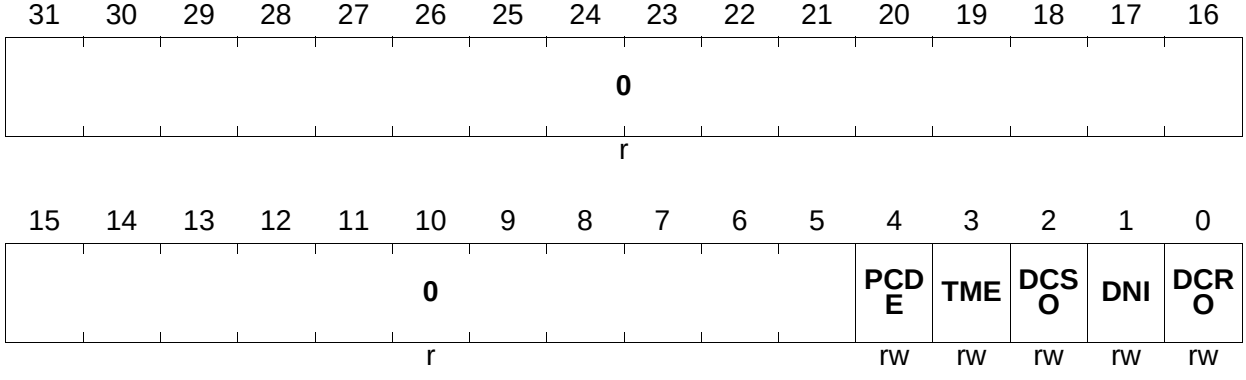


Field	Bits	Type	Description
SSRN	[7:0]	rh	**PCP Stalled Service Request Number** This field shows the Service Request Number which was being posted when the last (or present) stall condition occurred This field can only be cleared by a reset.

Field	Bits	Type	Description
STOS	[9:8]	rh	**PCP Stalled Type of Service** This field shows the Type-Of-Service to which an interrupt was being posted which caused the last (or present) stall condition (i.e. the service request queue that was full when the PCP attempted to post a request to it). This field can only be cleared by a reset.
ST	15	rh	**PCP Stalled Status** This bit shows the stalled status of the PCP. 0 PCP is not stalled 1 PCP is stalled
SCHN	[23:16]	rh	**PCP Stalled Channel Number** This field shows the channel number of the channel that was executing when the last (or present) stall condition occurred. This field can only be cleared by a reset.
0	[14:10] [31:24]	r	**Reserved**; read as 0; should be written with 0.

17.11.8 PCP Feature Test/Disable Register, PCP_FTD

PCP_FTD
PCP Feature Test/Disable Register **Reset Value: 0000 0000$_H$**

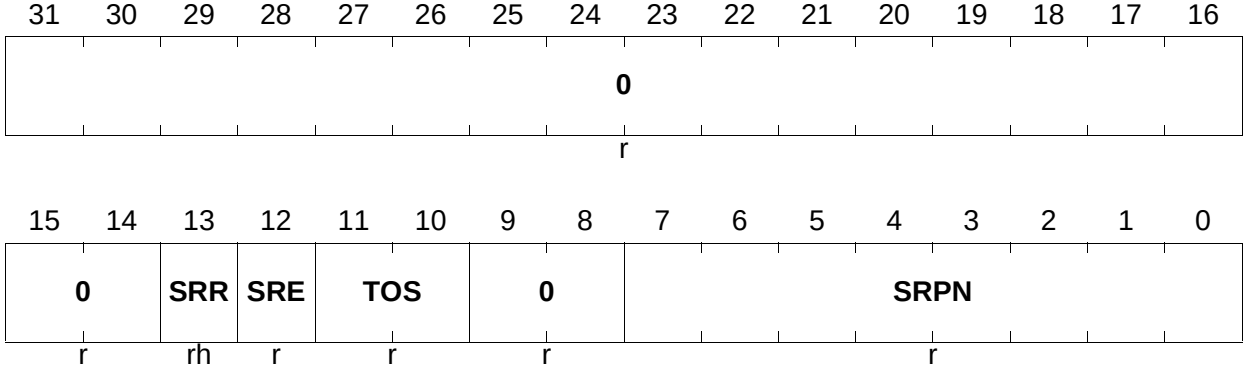


Field	Bits	Type	Description
DCRO	0	rw	**PCP Disable Context Restore Optimization** 0 Context Restores are optimized. 1 Context Restored are not optimized.
DNI	1	rw	**PCP Disable Nested Interrupts** 0 Nested Interrupts are enabled 1 Nested Interrupts are disabled
DCSO	2	rw	**PCP Disable Context Store Optimization** 0 Context Stores are optimized 1 Context Stores are not optimized
TME	3	rw	**Test Memory Error** 0 No memory error 1 Generate memory error *Note: This bit simulates the error signal that would be generated by a memory (CMEM or PRAM) that has an error status output. When the bit is written to 1, any channel that is executing and any subsequent channels will perform an Error Exit associated with a memory error. The bit must be re-written to 0 to allow normal PCP operation to resume.*

Field	Bits	Type	Description
PCDE	5	rw	**Prevent Channel Disable on Illegal Exit Instruction** 0 Disable the channel if an illegal TOS/SRPN value is posted via an EXIT instruction. 1 Do not disable the channel if an illegal TOS/ SRPN value is posted via an EXIT instruction.
0	[31:5]	r	**Reserved**; read as 0; should be written with 0.

17.11.9 PCP Service Request Control Register 0

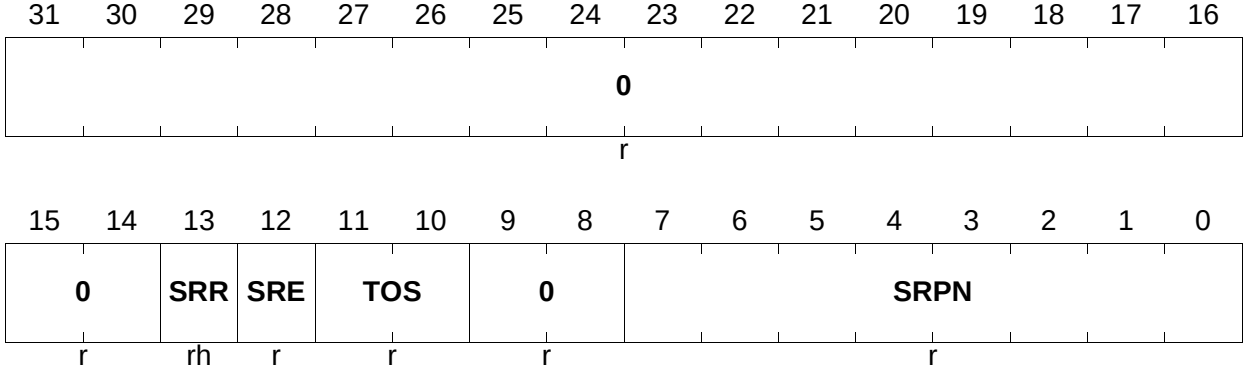
PCP_SRC0
PCP Service Request Control Register 0 **Reset Value: 0000 1000$_H$**
(Service Request Node for Interrupt Bus 0)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 0 Service Request Priority Number** SRPN Entry in SRC0 (Default = 0)
TOS	[11:10]	r	**PCP Node 0 Type of Service Control** Always read as 00_B. This means TOS is associated with interrupt bus 0 (CPU interrupt arbitration bus).
SRE	12	r	**PCP Node 0 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 0 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0; should be written with 0.

17.11.10 PCP Service Request Control Register 1

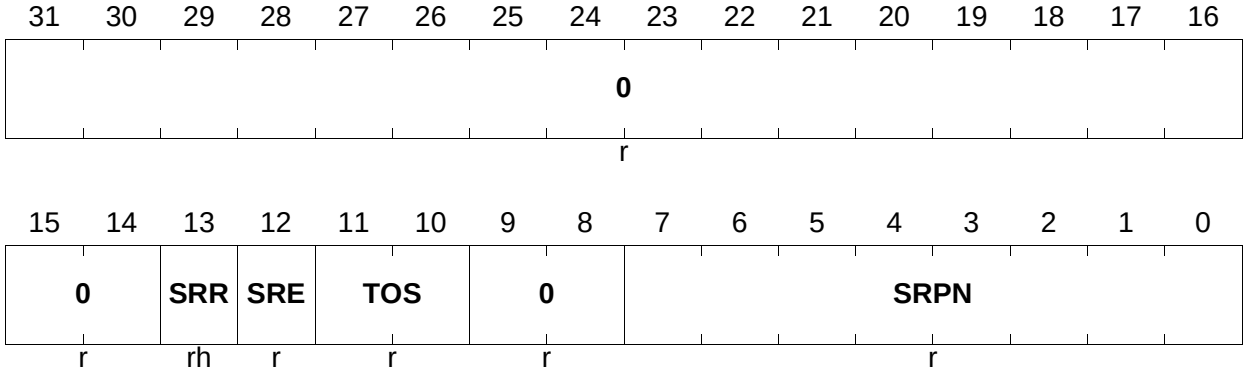
PCP_SRC1
PCP Service Request Control Register 1 **Reset Value: 0000 1000$_H$**
(Service Request Node for Interrupt Bus 0)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 1 Service Request Priority Number** SRPN entry in SRC1 (Default = 0)
TOS	[11:10]	r	**PCP Node 1 Type of Service Control** Always read as 00_B. This means TOS is associated with interrupt bus 0 (CPU interrupt arbitration bus).
SRE	12	r	**PCP Node 1 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 1 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.11 PCP Service Request Control Register 2

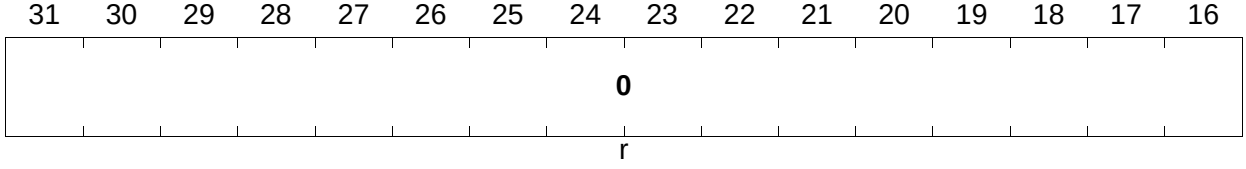
PCP_SRC2
PCP Service Request Control Register 2 **Reset Value: 0000 1400$_H$**
(Service Request Node for Interrupt Bus 1)

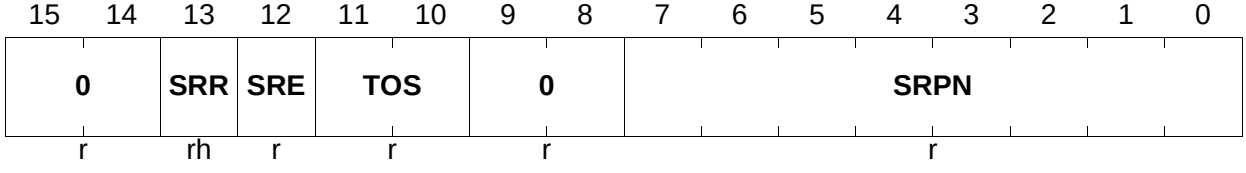


Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 2 Service Request Priority Number** SRPN entry in SRC2 (Default = 0)
TOS	[11:10]	r	**PCP Node 2 Type of Service Control** Always read as o1B. This means TOS is associated with interrupt bus 1(PCP interrupt arbitration bus)
SRE	12	r	**PCP Node 2 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 2 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.12 PCP Service Request Control Register 3

PCP_SRC3
PCP Service Request Control Register 3 **Reset Value: 0000 1400$_H$**
(Service Request Node for Interrupt Bus 1)

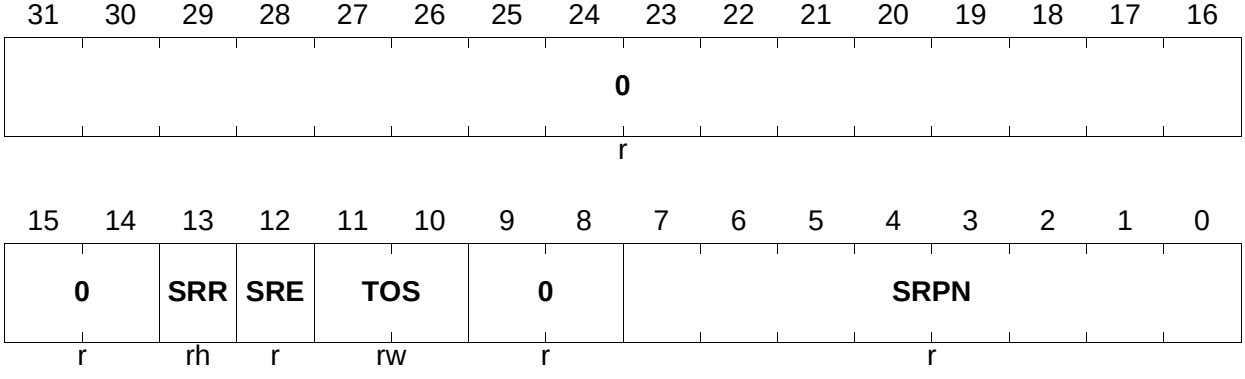




Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 3 Service Request Priority Number** SRPN entry in SRC3 (Default = 0)
TOS	[11:10]	r	**PCP Node 3 Type of Service Control** Always read as 01_B. This means TOS is associated with interrupt bus 1 (PCP interrupt arbitration bus)
SRE	12	r	**PCP Node 3 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 3 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.13 PCP Service Request Control Register 4

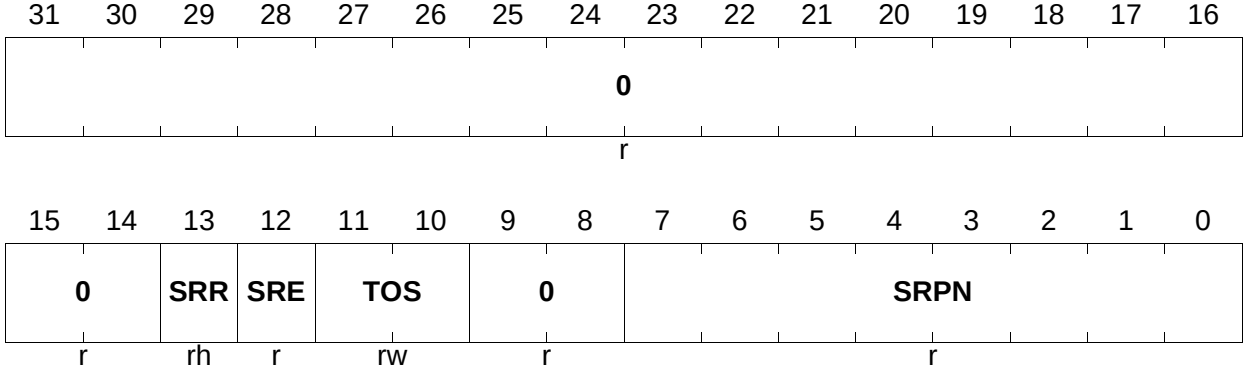
PCP_SRC4
PCP Service Request Control Register 4 **Reset Value: 0000 1000$_H$**
(Service Request Node programmable for any Interrupt Bus)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 4 Service Request Priority Number** SRPN entry in SRC4 (Default = 0)
TOS	[11:10]	rw	**PCP Node 4 Type of Service Control** PCP Service Request Node 4 Type-of-Service Control. Value depends on the interrupt mapping that has been selected. This bit field must be written at PCP configuration time and not subsequently modified while the PCP is operating.
SRE	12	r	**PCP Node 4 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 4 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.14 PCP Service Request Control Register 5

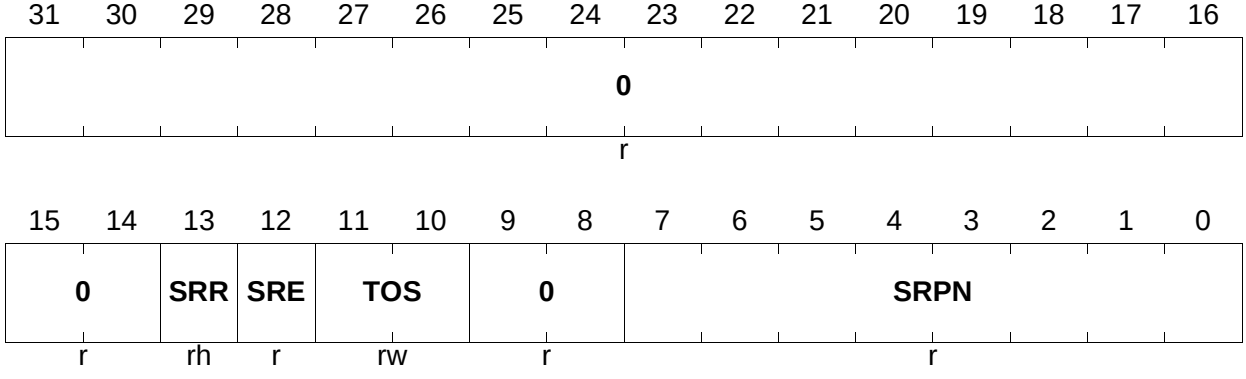
PCP_SRC5
PCP Service Request Control Register 5 **Reset Value: 0000 1000$_H$**
(Service Request Node programmable for any Interrupt Bus)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 5 Service Request Priority Number** SRPN entry in SRC5 (Default = 0)
TOS	[11:10]	rw	**PCP Node 5 Type of Service Control** PCP Service Request Node 5 Type-of-Service Control. Value depends on the interrupt mapping that has been selected. This bit field must be written at PCP configuration time and not subsequently modified while the PCP is operating.
SRE	12	r	**PCP Node 5 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 5 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.15 PCP Service Request Control Register 6

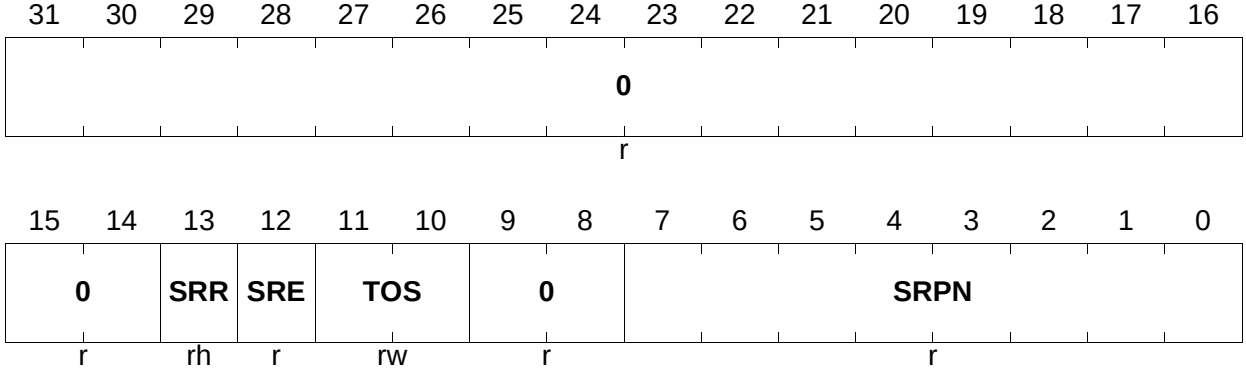
PCP_SRC6
PCP Service Request Control Register 6 **Reset Value: 0000 1000$_H$**
(Service Request Node programmable for any Interrupt Bus)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 6 Service Request Priority Number** SRPN entry in SRC6 (Default = 0)
TOS	[11:10]	rw	**PCP Node 6 Type of Service Control** PCP Service Request Node 6 Type-of-Service Control. Value depends on the interrupt mapping that has been selected. This bit field must be written at PCP configuration time and not subsequently modified while the PCP is operating.
SRE	12	r	**PCP Node 6 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 6 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.16 PCP Service Request Control Register 7

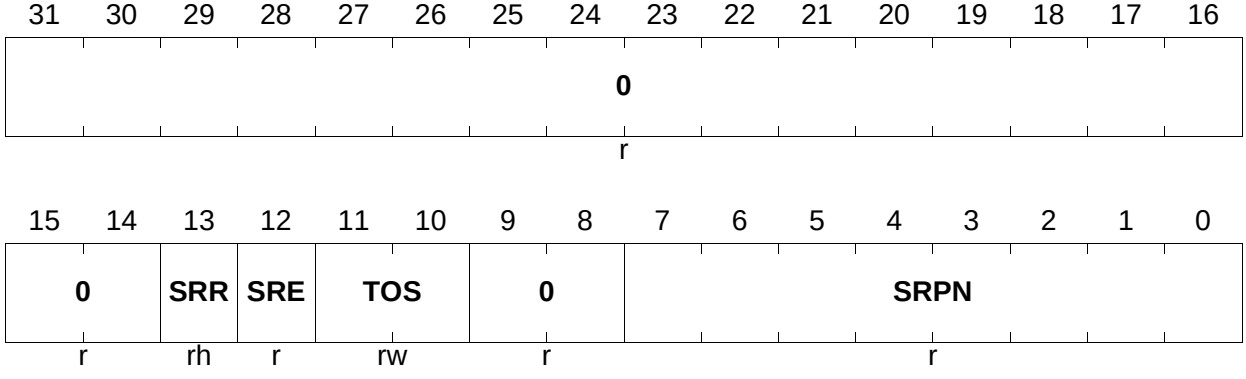
PCP_SRC7
PCP Service Request Control Register 7 **Reset Value: 0000 1000$_H$**
(Service Request Node programmable for any Interrupt Bus)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 7 Service Request Priority Number** SRPN entry in SRC7 (Default = 0)
TOS	[11:10]	rw	**PCP Node 7 Type of Service Control** PCP Service Request Node 7 Type-of-Service Control. Value depends on the interrupt mapping that has been selected. This bit field must be written at PCP configuration time and not subsequently modified while the PCP is operating.
SRE	12	r	**PCP Node 7 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 7 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.17 PCP Service Request Control Register 8

PCP_SRC8
PCP Service Request Control Register 8 **Reset Value: 0000 1000$_H$**
(Service Request Node programmable for any Interrupt Bus)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 8 Service Request Priority Number** SRPN entry in SRC8 (Default = 0)
TOS	[11:10]	rw	**PCP Node 8 Type of Service Control** PCP Service Request Node 8 Type-of-Service Control. Value depends on the interrupt mapping that has been selected. This bit field must be written at PCP configuration time and not subsequently modified while the PCP is operating.
SRE	12	r	**PCP Node 8 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 8 Service Request Flag** 0 No service request (Default) 1 Valid active service request
0	[9:8], [15:14], [31:16]	r	**Reserved**; read as 0.

17.11.18 PCP Service Request Control Register 9

PCP_SRC9
PCP Service Request Control Register 9 **Reset Value: 0000 1400$_H$**
(Service Request Node with Suspended Interrupt Capability)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0			**RRQ**	**0**				**SRCN**							
r			rh	r				rh							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		**SRR**	**SRE**	**TOS**		**0**		**SRPN**							
r		rh	r	rw		r		r							

Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 9 Service Request Priority Number** SRPN entry in SRC9 (Default = 0) When the PCP interrupt request contained within SRC9 was raised by the PCP core when executing an EXIT instruction then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC9 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt then this field contains the CPPN value of the PCP when the interrupt was suspended.
TOS	[11:10]	rw	**PCP Node 9 Type of Service Control** Always reads as the TOS associated with interrupt bus 1 (PCP interrupt arbitration bus).
SRE	12	r	**PCP Node 9 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 9 Service Request Flag** 0 No service request (Default) 1 Valid active service request

Field	Bits	Type	Description
SRCN	[23:16]	rh	**PCP Node 9 Service Request Channel Number** Channel Number Entry in SRC9 (Default = 0). When the PCP interrupt request contained within SRC9 was raised by the PCP core when executing an EXIT instruction, then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC9 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt, then this field contains the channel number of the channel that was suspended.
RRQ	28	rh	**PCP Node 9 Channel Restart Request** Set when the SRC9 register contains an active service request that is associated with a suspended interrupt (i.e. a channel that has been interrupted by a higher priority channel). 0 The interrupt is not suspended 1 The interrupt is suspended RRQ is always 0 when SRR is 0.
0	[9:8], [15:14], [27:24] [31:29]	r	**Reserved**; read as 0.

17.11.19 PCP Service Request Control Register 10

PCP_SRC10
PCP Service Request Control Register 10 **Reset Value: 0000 1400$_H$**
(Service Request Node with Suspended Interrupt Capability)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0			**RRQ**	**0**				**SRCN**							
r			rh	r				rh							

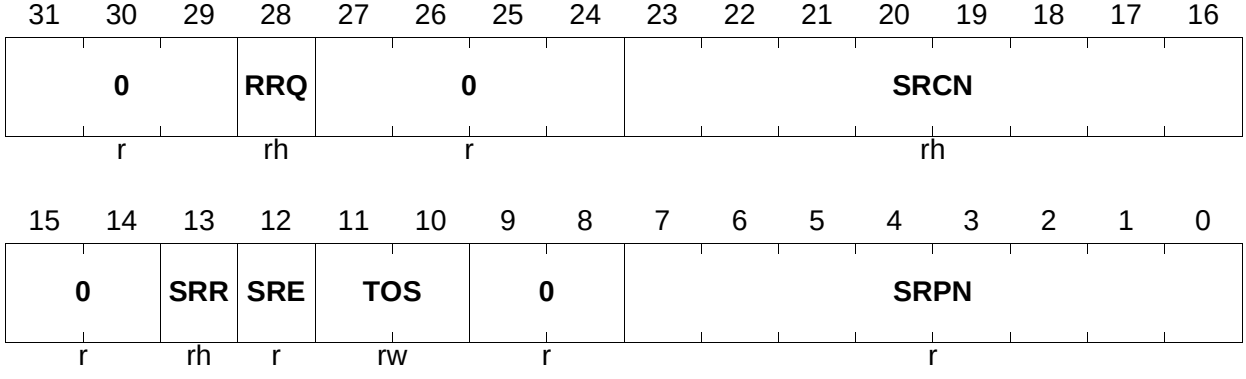
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		**SRR**	**SRE**	**TOS**		**0**		**SRPN**							
r		rh	r	rw		r		r							

Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 10 Service Request Priority Number** SRPN entry in SRC10 (Default = 0) When the PCP interrupt request contained within SRC10 was raised by the PCP core when executing an EXIT instruction then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC10 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt then this field contains the CPPN value of the PCP when the interrupt was suspended.
TOS	[11:10]	rw	**PCP Node 10 Type of Service Control** Always reads as the TOS associated with interrupt bus 1 (PCP arbitration bus).
SRE	12	r	**PCP Node 10 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 10 Service Request Flag** 0 No service request (Default) 1 Valid active service request

Field	Bits	Type	Description
SRCN	[23:16]	rh	**PCP Node 10 Service Request Channel Number** Channel Number Entry in SRC10 (Default = 0). When the PCP interrupt request contained within SRC10 was raised by the PCP core when executing an EXIT instruction, then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC10 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt, then this field contains the channel number of the channel that was suspended.
RRQ	28	rh	**PCP Node 10 Channel Restart Request** Set when the SRC10 register contains an active service request that is associated with a suspended interrupt (i.e. a channel that has been interrupted by a higher priority channel). 0 The interrupt is not suspended 1 The interrupt is suspended RRQ is always 0 when SRR is 0.
0	[9:8], [15:14], [27:24] [31:29]	r	**Reserved**; read as 0.

17.11.20 PCP Service Request Control Register 11

PCP_SRC11
PCP Service Request Control Register 11 **Reset Value: 0000 1400$_H$**
(Service Request Node with Suspended Interrupt Capability)



Field	Bits	Type	Description
SRPN	[7:0]	r	**PCP Node 11 Service Request Priority Number** SRPN entry in SRC11 (Default = 0) When the PCP interrupt request contained within SRC11 was raised by the PCP core when executing an EXIT instruction then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC11 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt then this field contains the CPPN value of the PCP when the interrupt was suspended.
TOS	[11:10]	rw	**PCP Node 11 Type of Service Control** Always reads as the TOS associated with interrupt bus 1 (PCP arbitration bus).
SRE	12	r	**PCP Node 11 Service Request Enable** Always read as 1 (enabled)
SRR	13	rh	**PCP Node 11 Service Request Flag** 0 No service request (Default) 1 Valid active service request

Field	Bits	Type	Description
SRCN	[23:16]	rh	**PCP Node 11 Service Request Channel Number** Channel Number Entry in SRC11 (Default = 0). When the PCP interrupt request contained within SRC11 was raised by the PCP core when executing an EXIT instruction, then this field contains the SRPN value taken from R6 when the exit instruction was executed. When the PCP interrupt request contained within SRC11 was raised by the PCP core when suspending execution of a channel program in order to service a higher priority interrupt, then this field contains the channel number of the channel that was suspended.
RRQ	28	rh	**PCP Node 11 Channel Restart Request** Set when the SRC11 register contains an active service request that is associated with a suspended interrupt (i.e. a channel that has been interrupted by a higher priority channel). 0 The interrupt is not suspended 1 The interrupt is suspended RRQ is always 0 when SRR is 0.
0	[9:8], [15:14], [27:24] [31:29]	r	**Reserved**; read as 0.

17.12 PCP Instruction Set Details

This section describes the instruction set architecture of the PCP in detail.

17.12.1 Instruction Codes and Fields

All PCP instructions use a common set of fields to describe such things as the source register, and the state of flags. Additionally, many instructions (including arithmetic and many flow control instructions), are conditionally executed.

The descriptions of the PCP instructions are based on the following conventions.

< >	Implicit source/destination are contents of brackets
>>, <<	Shift left or right, respectively.
[ ]	Indirect access based on contents of brackets (de-reference).
#imm*NN*	Immediate value encoded into an instruction with width *NN.*
#offset*NN*	Address offset immediate value with width *NN*.
NextPC	The current executing instruction’s address + 1. (The next instruction to be fetched.)
cc_*A, cc_B*	Condition Code CONDCA/CONDCB.

17.12.1.1 Conditional Codes

Many PCP instructions have the option of being executed conditionally. The condition code of an instruction is the field that specifies the condition to be tested before the instruction is executed. Depending on the type of instruction there are 8 or 16 condition codes available. The set of 8-condition codes is referred to as CONDCA, while the set of 16-condition codes is referred to as CONDCB. The condition codes are based on an equation of the Flags held in R7. See **Table 17-12**.

Table 17-12 Condition Codes

Description	CONDCA/B	Test (Flag Bits)	Code	Mnemonic
Unconditional	A / B	–	0_H	cc_UC
Zero/Equal	A / B	Z = 1	1_H	cc_Z
Not Zero/Not Equal	A / B	Z = 0	2_H	cc_NZ
Overflow	A / B	V = 1	3_H	cc_V
Carry/Unsigned Less Than/ Check Bit True	A / B	C = 1	4_H	cc_C, cc_ULT
Unsigned Greater Than	A / B	C OR Z = 0	5_H	cc_UGT
Signed Less Than	A / B	N XOR V = 1	6_H	cc_SLT
Signed Greater Than	A / B	(N XOR V) OR Z = 0	7_H	cc_SGT
Negative	B	N = 1	8_H	cc_N
Not Negative	B	N = 0	9_H	cc_NN
Not Overflow	B	V = 0	A_H	cc_NV
No Carry/Unsigned Greater than or Equal	B	C = 0	B_H	cc_NC, cc_UGE
Signed Greater Than or Equal	B	N XOR V = 0	C_H	cc_SGE
Signed Less than or Equal	B	(N XOR V) OR Z = 1	D_H	cc_SLE
CNT1 Equal Zero	B	CN1Z = 1	E_H	cc_CNZ
CNT1 Not Equal Zero	B	CN1Z = 0	F_H	cc_CNN

17.12.1.2 Instruction Encoding

Most instructions are encoded in 16 bits. This allows two instruction to be fetched out of 32 bit instruction memory per access. For example a COPY and an EXIT instruction can be fetched simultaneously, performing a simple DMA transaction.**Table 17-13** lists the instruction field definitions of the PCP instruction set architecture.

Note: The exact syntax for these fields may be different depending on which tool (e.g. assembler) is used. Please refer to the respective tool descriptions.

Table 17-13 Instruction Field Definitions

Symbol	Syntax	Description
CNC		**Counter Control** This field is used by the BCOPY/COPY instruction to control the number of repetitions of the data transfer. See also **Figure 17-14**.
	CNC = 00	Decrement CNT0 after each transfer. Continue until CNT0 = 0 and proceed to next instruction
	CNC = 01	Post Decrement CNT0 after each instruction transfer. Continue until CNT0 = 0, then decrement CNT1 and proceed to next instruction.
	CNC = 10	Post Decrement CNT0 after each instruction transfer. Continue until CNT0 = 0, then Decrement CNT1. Reload CNT0 value and continue. Continue until CNT1 = 0, then proceed to next instruction.
	CNC = 11	Reserved.
RC0		**Counter RC0 Reload Value** Counter RC0 is an implicit counter used by the COPY instruction. The reload value given in the instruction specifies how many data transfers are to be performed by the instruction. See also **Figure 17-13**.
	RC0=000	Perform 8 data transfers
	RC0=001	Perform 1 data transfers
	...	...
	RC0=111	Perform 7data transfers
		Block Size Control This field is used by the BCOPY instruction to control the Block size
	RC0 = 0	8 words
	RC0 = 1	Reserved (error)
	RC0 = 2	2 words
	RC0 = 3	4 words
	Others	Reserved (error)

Table 17-13 Instruction Field Definitions (cont'd)

Symbol	Syntax	Description
DAC		**Disable All Channels Control.**
	DAC = 0	Allow the PCP to continue to execute Channel Programs in response to Service Requests.
	DAC = 1	Clear CS.EN which stops the PCP executing and further channel programs
cc_*A*, *cc_B*	see **Table 17-12**	**Condition Code** Specifies conditional execution of instruction according to CONDCA or CONDCB.
EC		**Exit Count Control**
	EC = 0	No action
	EC = 1	Decrement CNT1
EDA		**External Debug Action**
	EDA = 0	No External Debug Action caused
	EDA = 1	Cause an External Debug Action (breakpoint pin etc.)
EP		**Entry Point Control**
	EP = 0	Set the PC to Channel Program Start. EP = 0 assumes that a Channel Entry Table exists in the base of Code Memory. Failure to provide one will cause improper operation.
	EP = 1	Set the PC to the address contained in NextPC (next instruction) address.
INT		**Interrupt Control**
	INT = 0	No Interrupt
	INT = 1	INT = 1 AND (cc_B = True) means Issue Interrupt
RTA		**Return to This Address**
	RTA = 0	The channel is disabled (R7.CEN = 0) and the PC value stored in the context is NextPC
	RTA = 1	Allow further Channel Program execution and decrement PC so that DEVUG instruction is re-executed on next invocation. *Note: This field has no effect id SDB==0.*
SIZE		**Data Size Control**
	SIZE = 00	Byte (8-bit)
	SIZE = 01	Half-word (16-bit)
	SIZE = 10	Word (32-bit)
	SIZE = 11	Reserved

Table 17-13 Instruction Field Definitions (cont'd)

Symbol	Syntax	Description
SRC+-		**Source Address Pointer Control**
	SRC (00)	No Change (SRC)
	SRC+ (01)	Post Increment by Size (SRC+)
	SRC- (10)	Post Decrement by Size (SRC-)
	(11)	Reserved
DST+-		**Destination Address Pointer Control**
	DST (00)	No Change (DST)
	DST+ (01)	Post Increment by Size (DST+)
	DST- (10)	Post Decrement by Size (DST-)
	(11)	Reserved.
S/C		**Test Bit Control**
	S/C = 0	Check for Clear (0)
	S/C = 1	Check for Set (1)
SDB		**Stop on Debug**
	SDB = 0	Continue running if debug event triggered
	SDB = 1	Stop PCP if debug event triggered
ST		**Stop Channel**
	ST = 0	Continue channel execution, Leave Channel Program enabled
	ST = 1	Stop Channel Program execution, Perform actions according to RTA setting

17.12.2 Counter Operation for COPY Instruction

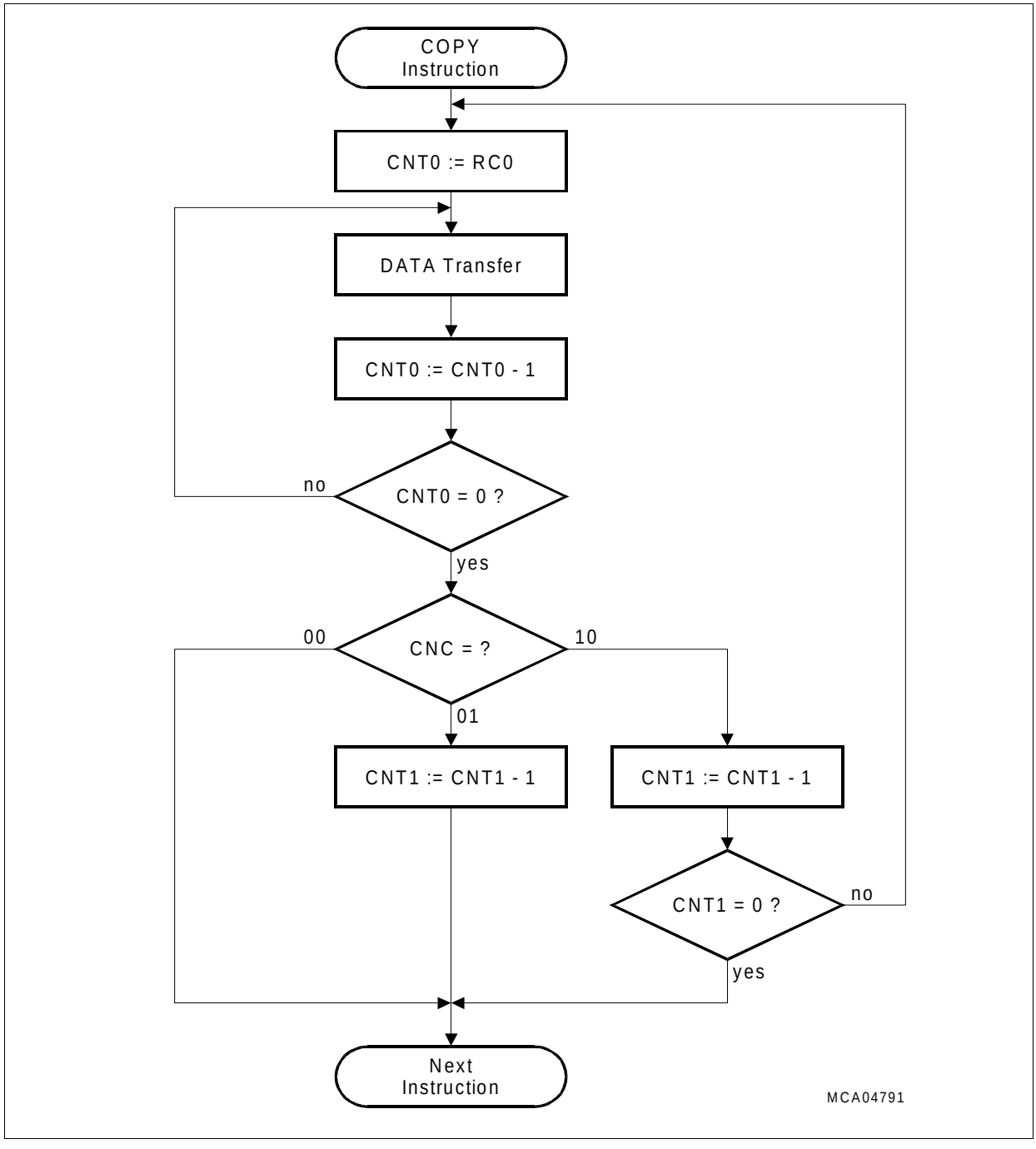


Figure 17-13 Counter Operation for COPY Instruction

17.12.3 Counter Operation for BCOPY Instruction

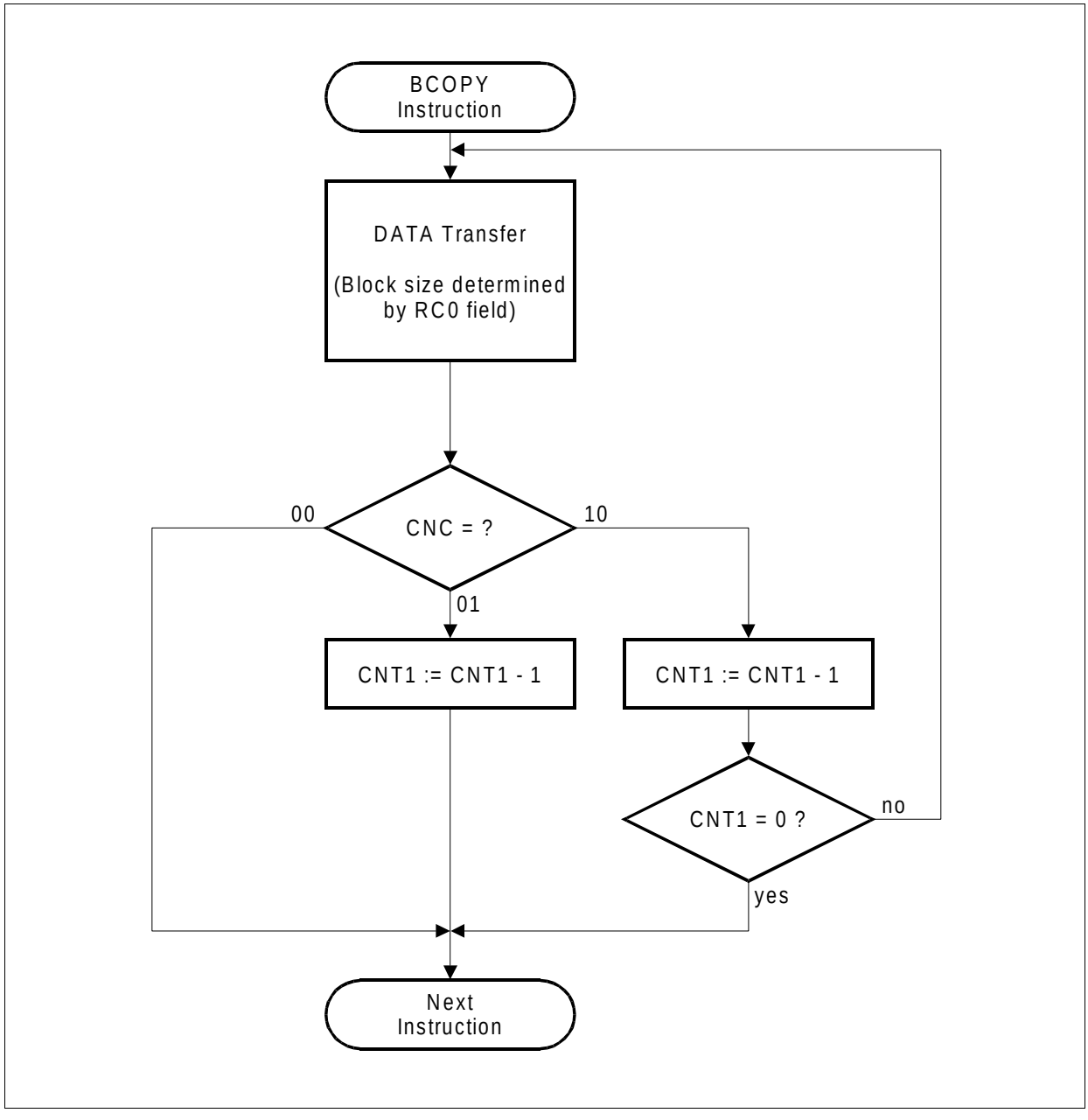


Figure 17-14 Counter Operation for BCOPY Instruction

17.12.4 Divide and Multiply Instructions

The PCP provides Multiply and Divide capabilities (unsigned values only). All Multiply and Divide instructions operate on 8 bits of data (taken from the dividend for divide, from the multiplicand for multiply). This strategy allows the user to implement the appropriate number of instructions (“steps”) as required for his data format.

Each execution of a divide instruction (DSTEP) performs a division which generates 8 bits of result, and also manipulates the registers being used to allow the execution of consecutive divide (DSTEP) instructions to build divide algorithms in multiples of 8 bits.

Each execution of a multiply instruction (MSTEP.L and MSTEP.U) performs a multiplication on 8 bits of data (taken from the multiplicand) and also manipulates the registers to allow execution of consecutive multiply instructions to build multiply algorithms in multiples of 8 bits.

The following restrictions apply to the use of Divide and Multiply instructions:

– The first instruction of any divide sequence must be the DINIT (initialization) instruction. Any additional instructions, other than MINIT, MSTEP.L or MSTEP.U, may also be used within the sequence as long as they do not modify any of the registers used for division (R0, Ra and Rb). All subsequent divide instructions within the sequence (DSTEP) must use the same register for dividend and the same register for divisor as was used in the preceding DINIT instruction.
– The first instruction of any multiply sequence must be the MINIT (initialization) instruction. Any additional instructions, other than DINIT or DSTEP, may also be used within the sequence as long as they do not modify any of the registers used for multiplication (R0, Ra and Rb). All subsequent multiply instructions within the sequence (MSTEP.L and MSTEP.U) must use the same register for multiplicand and the same register for multiplier as was used in the preceding MINIT instruction.
– Neither of the operand registers (Ra or Rb) may be R0 (which is used implicitly within all the instructions) or R7 (flag register). In addition, the same register may not be supplied as both operand registers of an instruction (e.g. DSTEP R3, R3 is invalid).

Note: Failure to adhere to these restrictions will yield undefined results.

Note: Special care must be taken when using Multiply and Divide sequences when a Channel Program is interruptible. In this case it must be ensured that a sequence can not be corrupted by the execution of Multiply or Divide instructions executed by a higher priority channel. The R7.IEN bit can be used to ensure that a sequence is not interruptible.

In the following descriptions attached to each Multiply/Divide instruction a Pseudo Code Model is supplied to provide an unambiguous definition of the function of each instruction. The Model provided for the DSTEP and MSTEP32 instructions uses 32 bit unsigned integer arithmetic, ignoring any possible overflows. The model supplied for the MSTEP64 uses a 40-bit unsigned multiply and then shifts this result right by 8 bits (discards the least significant 8 bits of the 40-bit result).

The DSTEP instruction also has some conditions stipulated regarding input values to the instruction. Use of the Model for the DSTEP instruction with invalid input values will yield an invalid result.

17.12.4.1 Divide Instructions.

Table 17-14 Divide Instructions

DINIT	Syntax	**DINIT Rb, Ra**
	Description	Initialize Divide/Multiply logic ready for divide sequence.
	Operation	Clear R0 If value of Ra is 0 then set V (to flag divide by 0 error) otherwise clear V If value of Rb is 0 and the value of Ra is not 0 then set Z (to flag a zero result)
	Flags	Z - set if Rb (dividend) = 0 and Ra (divisor) does not = 0, otherwise cleared V - set if Ra (divisor) = 0, otherwise cleared. All other flags unaffected
DSTEP	Syntax	**DSTEP Rb, Ra**
	Description	Perform an unsigned divide step, generating 8 bits of result.
	Operation	Shift R0 left by 8 bits, copy the MS byte of Rb into LS byte of R0 Shift Rb left by 8 bits and add (R0 divided by Ra) Load R0 with (the remainder of R0 divided by Ra) (see note below regarding the size of the “R0 divided by Ra” result)
	Flags	All flags unaffected

17.12.4.2 Multiply Instructions.

Table 17-15 Multiply Instructions

MINIT	Syntax	**MINIT Rb, Ra**
	Description	Initialize Divide/Multiply logic ready for multiply sequence.
	Operation	Clear R0 If value of R[a] is zero or value of Rb is zero then set Z (to flag zero result) else clear Z.
	Flags	Z - set if result will be 0 (Ra = 0 or Rb = 0), cleared otherwise All other flags unaffected

Table 17-15 Multiply Instructions

MSTEP.L	Syntax	**MSTEP.L Rb, Ra**
	Description	Perform an unsigned multiply step, using eight bits of data taken from Rb, keeping the LS 32 bits of a potential 64 bit result
	Operation	Left rotate Rb by 8 bits Left shift R0 by 8 bits Add (Ra multiplied by the LS 8 bits of Rb) to R0
	Flags	All flags unaffected
MSTEP.U	Syntax	**MSTEP.U Rb, Ra**
	Description	Perform an unsigned multiply step, using eight bits of data taken from Rb, keeping a 40 bit result (when executed 4 times this compiles a full 32x32 bit multiply 64 bit result).
	Operation	Add (Ra multiplied by the LS 8 bits of Rb) to R0 (1) Shift Rb right by 8 bits Copy the LS 8 bits of the result of (1) above to the MS 8 bits of Rb Copy the MS 32 bits of the result of (1) above to R0
	Flags	All flags unaffected

17.12.5 ADD, 32-Bit Addition

ADD	Syntax	**ADD Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then add the contents of register Ra to the contents of register Rb; place the result in Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[b] + R[a] else NOP
	Flags	N, Z,
ADD.I	Syntax	**ADD.I Ra, #imm6**
	Description	Add the zero-extended immediate value imm6 to the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] + zero_ext(imm6)
	Flags	N, Z

ADD.F	Syntax	**ADD.F Rb, [Ra], Size**
	Description	Add the contents of the address location specified by the contents of register Ra to the contents of register Rb; place the result in Rb. *Note: Byte and Half-word values are zero-extended.*
	Operation	R[b] = R[b] + zero_ext(FPI[R[a]])
	Flags	N, Z
ADD.PI	Syntax	**ADD.PI Ra, [#offset6]**
	Description	Add the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6 to the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] + PRAM[(DPTR<<6) + zero_ext(#offset6)]
	Flags	N, Z

17.12.6 AND, 32-Bit Logical AND

AND	Syntax	**AND Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then perform a bitwise logical AND of the contents of register Ra and the contents of register Rb; place the result in Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[b] AND R[a] else NOP
	Flags	N, Z
AND.F	Syntax	**AND.F Rb, [Ra], Size**
	Description	Perform a bitwise logical AND of the contents of the address location, specified by the contents of register Ra, and the contents of register Rb; place the result in Rb.
	Operation	R[b] = R[b] AND zero_ext(FPI[R[a]])
	Flags	N, Z

AND.PI	Syntax	**AND.PI Ra, [#offset6]**
	Description	Perform a bitwise logical AND of the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6, and the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] AND PRAM[(DPTR<<6) + zero_ext(#offset6)]
	Flags	N, Z

17.12.7 BCOPY, DMA Instruction

BCOPY	Syntax	**BCOPY DST+-, SRC+-, CNC, RC0**
	Description	Allows the PCP to perform DMA type transfers using FPI block transfers. Moves a block of data of FPI Bus source location to FPI Bus destination location using FPI burst mode. Source location is pointed to by the contents of register R4; destination location is pointed to by the contents of register R5. Options (see also**Table 17-13**): Source pointer (SRC+-): Increment, decrement or unchanged; Destination pointer (SRC+-): Increment, decrement or unchanged; Counter control (CNC): Optionally decrement the R6.CNT1 field and use it as an outer loop counter to implement automatic multiple block transfers. Counter 0 reload value (RC0): Control the block size.
	Operation	temp = zero_ext(FPI[R[4]]); value loaded and extended depending on SIZE FPI(R[5]) = temp R4 = R4 +/- n; n depending on SRC+- and CNT0 R5 = R5 +/- n; n depending on DST+- and CNT0 for counter operation see **Figure 17-14** and **Table 17-13**.
	Flags	CN1Z

17.12.8 CHKB, Check Bit

CHKB	Syntax	**CHKB Ra, #imm5, S/C**
	Description	If bit imm5 of register Ra is equal to the specified test value S/C then set the carry flag R7.C, else clear the carry flag.
	Operation	if (R[a][imm5] = S/C) then R7_C = 1 else R7_C = 0
	Flags	C

17.12.9 CLR, Clear Bit

CLR	Syntax	**CLR Ra, #imm5**
	Description	Clear bit imm5 of register Ra to 0.
	Operation	R[a][imm5] = 0
	Flags	None
CLR.F	Syntax	**CLR.F [Ra], #imm5, Size**
	Description	Clear bit imm5 of the address location specified through the contents of register Ra to 0. This instruction is executed using a locked read-modify-write FPI Bus transaction.
	Operation	FPI[(R[a])][imm5] = 0
	Flags	None

17.12.10 COMP, 32-Bit Compare

COMP	Syntax	**COMP Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then subtract the contents of register Ra from the contents of register Rb; set the flags in register R7 according to the result of the subtraction; discard the subtraction result. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R7_FLAGS = Flags(R[b] - R[a])
	Flags	N, Z, V, C

COMP.I	Syntax	**COMP.I Ra, #imm6**
	Description	Subtract the sign-extended immediate value imm6 from the contents of register Ra; set the flags in register R7 according to the result of the subtraction; discard the subtraction result.
	Operation	R7_FLAGS = Flags(R[a] - sign_ext(imm6))
	Flags	N, Z, V, C
COMP.F	Syntax	**COMP.F Rb, [Ra], Size**
	Description	Subtract the contents of the address location specified by the contents of register Ra from the contents of register Rb; set the flags in register R7 according to the result of the subtraction; discard the subtraction result.
	Operation	R7_FLAGS = Flags(R[b] - sign_ext(FPI[R[a]]))
	Flags	N, Z, V, C
COMP.PI	Syntax	**COMP.PI Ra, [#offset6]**
	Description	Subtract the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6, from the contents of register Ra; set the flags in register R7 according to the result of the subtraction; discard the subtraction result.
	Operation	R7_FLAGS = Flags(R[a] - PRAM[(DPTR<<6) + zero_ext(#offset6)])
	Flags	N, Z, V, C

17.12.11 COPY, DMA Instruction

COPY	Syntax	**COPY DST+-, SRC+-, CNC, RC0, SIZE**
	Description	Moves the contents of FPI Bus source location to FPI Bus destination location. Source location is pointed to by the contents of register R4; destination location is pointed to by the contents of register R5. Options (see also**Table 17-13**): Source pointer (SRC+-): Increment, decrement or unchanged; Destination pointer (SRC+-): Increment, decrement or unchanged; Counter control (CNC): Optionally decrement the R6.CNT1 field and use it as an outer loop counter to implement automatic multiple block transfers. Counter 0 reload value (RC0): After every transfer, CNT0 is decremented. The COPY instruction is repeated until CNT0 reaches zero. Data transfer width (SIZE): byte, half-word, word (pointers are incremented/decremented based upon SIZE).
	Operation	temp = zero_ext(FPI[R[4]]); value loaded and extended depending on SIZE FPI(R[5]) = temp R4 = R4 +/- n; n depending on SRC+- and SIZE R5 = R5 +/- n; n depending on DST+- and SIZE for counter operation see **Figure 17-14** and **Table 17-13**.
	Flags	CN1Z

17.12.12 DEBUG, Debug Instruction

DEBUG	Syntax	**DEBUG EDA, SDB, DAC, RTA, cc_B**
	Description	Conditionally cause a debug event if condition CONDCB is true. Optionally stop channel execution (SDB = 1) and/or generate an external debug event (EDA = 1).

	Operation	if (CONDCB = True) then
		if (EDA = 1) then activate BRK_OUT pin
		if (SDB = 1) then
		if(RTA = 0) then
		R7_CEN = 0; disable further channel invocation
		else
		PC = PC-1
		endif
		save_context
		idle
		endif
		if (DAC = 1)
		PCP_CS.EN = 0
		endif
		set ES.DBE; indicate debug event
		ES.PC = NextPC
		ES.PN = channel_number
		endif

	Flags	none

Note: If the condition code is false, the instruction will be treated as a NOP. i.e. no registers will be modified and flags will not be updated and no external debug action will be asserted and the channel will not exit.

17.12.13 DINIT, Divide Initialization Instruction

DINIT	Syntax	**DINIT <R0>, Rb, Ra**
	Description	Initialize Divide logic ready for divide sequence (Rb / Ra) and Clear R0. If value of Ra is 0 then set V (to flag divide by 0 error) otherwise clear V. If value of Rb is 0 and value of Ra is not 0 then set Z (to flag a zero result) otherwise clear Z.
	Operation	R0 = 0
	Flags	Z, V

17.12.14 DSTEP, Divide Instruction

DSTEP	Syntax	**DSTEP <R0>, Rb, Ra**
	Description	Perform 1 step (eight bits) of an unsigned 32- by 32-bit divide (Rb / Ra). Shift R0 left by 8 bits, copy the most significant byte of Rb into LS byte of R0. Shift Rb left by 8 bits and add (R0 divided by Ra). Load R0 with (the remainder of R0 divided by Ra).
	Operation	R0 = (R0 << 8) + (Rb >> 24) Rb = (Rb << 8) + R0 / Ra R0 = R0 % Ra
	Flags	Z

*Note: The value in Ra must always be greater than the value in R0 prior to execution of the DSTEP instruction. If the rules specified in **Section 17.12.4** are followed then the above description and operation are correct. Failure to adhere to these rules will yield undefined results.*

17.12.15 EXIT, Exit Instruction

EXIT	Syntax	**EXIT EC, ST, INT, EP, cc_B**
	Description	Unconditionally exit channel program execution. Optionally decrement counter CNT1 (EC = 1), disable further channel invocation (ST = 1), generate an interrupt request (INT = 1) if condition CONDCB is true. Field EP is used to set the channel code entry point in Channel Resume Mode to either the address of the next instruction (EP = 1) or to the start address of the channel (EP = 0). The EXIT instruction is finished with a context save operation. *Note: The EP option is only in effect when Channel Resume operation is globally selected through PCP_CS.RCB = 0. If PCP_CS.RCB = 1, Channel restart mode is selected for all channels, and the EP field of the EXIT instruction is disregarded.*

	Operation	if (EC = 1) then CNT1 = CNT1 - 1
		if (ST = 1) then R7_CEN = 0
		if ((INT = 1) AND (cc_B = True)) then activate_interrupt_request
		if (EP = 1) then R7_PC = NextPC else R7_PC = channel_entry_point
		save_context

	Flags	CN1Z

17.12.16 INB, Insert Bit

INB	Syntax	**INB Rb, Ra, cc_A**
	Description	If CONDCA is true, then insert the carry flag R7.C into register Rb at the bit position specified through bits [4..0] of register Ra. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b][R[a][4:0]] = R7_C else NOP
	Flags	None

INB.I	Syntax	**INB.I Ra, #imm5**
	Description	Insert the carry flag R7.C into register Ra at the bit position specified through the immediate value imm5.
	Operation	R[a][imm5] = R7_C
	Flags	None

17.12.17 JC, Jump Conditionally

JC	Syntax	**JC offset6, cc_B**
	Description	If CONDCB is true, then add the sign-extended value specified by offset6 to the contents of the PC, and jump to that address. If CONDCB is false, no operation is performed.
	Operation	if (CONDCB = True) then (PC = PC + sign_ext(offset6)) else NOP
	Flags	None
JC.A	Syntax	**JC.A #address16, cc_B**
	Description	If CONDCB is true, then load the value specified by address16 into the PC, and jump to that address. If CONDCB is false, no operation is performed.
	Operation	if (CONDCB = True) then (PC = address16) else NOP
	Flags	None
JC.I	Syntax	**JC.I Ra, cc_B**
	Description	If CONDCB is true, then add the value specified by Ra[15:0] to the contents of the PC, and jump to that address. Value Ra[15:0] is treated as a signed 16-bit number. If CONDCB is false, no operation is performed.
	Operation	if (CONDCB = True) then (PC = PC + (R[a][15:0])) else NOP
	Flags	None
JC.IA	Syntax	**JC.IA Ra, cc_B**
	Description	If CONDCB is true, then load the value specified by Ra[15:0] into the PC, and jump to that address. Value Ra[15:0] is treated as an unsigned 16-bit number. If CONDCB is false, no operation is performed.
	Operation	if (CONDCB = True) then (PC = (R[a][15:0])) else NOP
	Flags	None

17.12.18 JL, Jump Long Unconditional

JL	Syntax	**JL offset10**
	Description	Add the sign-extended value specified by offset10 to the contents of the PC, and jump to that address.
	Operation	PC = PC + sign_ext(offset10)
	Flags	None

17.12.19 LD, Load

LD.F	Syntax	**LD.F Rb, [Ra], Size**
	Description	Load the zero-extended contents of the address location specified by the contents of register Ra into register Rb.
	Operation	R[b] = zero_ext(FPI[R[a]])
	Flags	N, Z
LD.I	Syntax	**LD.I Ra, #imm6**
	Description	Load the zero-extended value specified by imm6 into register Ra.
	Operation	R[a] = zero_ext(imm6)
	Flags	N, Z
LD.IF	Syntax	**LD.IF [Ra], #offset5, Size**
	Description	Load the zero-extended contents of the address location, specified by the addition of the contents of register Ra and the value specified by imm5, into register R0.
	Operation	R[0] = zero_ext(FPI[R[a] + zero_ext(imm5)])
	Flags	N, Z
LD.P	Syntax	**LD.P Rb, [Ra], cc_A**
	Description	If condition CONDCA is true, then load the contents of the PRAM address location, specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value Ra[5:0] into register Rb. If condition CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = PRAM[(DPTR<<6) + zero_ext(R[a][5:0])] else NOP
	Flags	N, Z

LD.PI	Syntax	**LD.PI Ra, [#offset6]**
	Description	Load the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6 into register Ra.
	Operation	R[a] = PRAM[(DPTR<<6) + zero_ext(offset6)]
	Flags	N, Z

17.12.20 LDL, Load 16-bit Value

LDL.IL	Syntax	**LDL.IL Ra, #imm16**
	Description	Load the immediate value imm16 into the lower bits of register Ra (bits [15:0]). Bits [31:16] of register Ra are unaffected. Value imm16 is treated as an unsigned 16-bit number.
	Operation	R[a][15:0] = imm16
	Flags	N, Z
LDL.IU	Syntax	**LDL.IU Ra, #imm16**
	Description	Load the immediate value imm16 into the upper bits of register Ra (bits [31:16]). Bits [15:0] of register Ra are unaffected.
	Operation	R[a][31:16] = imm16
	Flags	N, Z

17.12.21 PRAM Bit Instructions

MCLR.PI	Syntax	**MCLR.PI Ra, #offset6**
	Description	Perform an AND of the contents of the specified register with the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset. Write the result back to the PRAM location
	Operation	Ra = PRAM[DPTR<<6 + #offset6]= Ra .AND. PRAM[DPTR<<6 + #offset6]
	Flags	N,Z

MSET.PI	Syntax	**MSET.PI Ra, #offset6**
	Description	Perform an OR of the contents of the specified register with the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset. Write the result back to the PRAM location
	Operation	Ra = PRAM[DPTR<<6 + #offset6]= Ra .OR. PRAM[DPTR<<6 + #offset6]
	Flags	N,Z

Note: MCLR and MSET are read/modify/write operations that can not be interrupted by an access from another FPI master. They can be used to implement semaphore systems.

17.12.22 Multiply Initialization Instruction

MINIT	Syntax	**MINIT <R0>, Rb, Ra**
	Description	Initialize Multiply logic ready for multiply sequence. Clear R0. If value of Ra is zero or value of Rb is zero then set Z (to flag zero result) else clear Z.
	Operation	R0 = 0
	Flags	Z

17.12.23 MOV, Move Register to Register

MOV	Syntax	**MOV Rb, Ra, cc_A**
	Description	If condition CONDCA is true, then move the contents of register Ra into register Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[a] else NOP
	Flags	N, Z

17.12.24 Multiply Instructions

MSTEP.L	Syntax	**MSTEP.L <R0>, Rb, Ra**
	Description	Perform an unsigned multiply step, using eight bits of data taken from Rb, keeping the least significant 32 bits of a potential 64 bit result. Left rotate Rb by 8 bits. Shift R0 left by 8 bits. Add (Ra multiplied by the least significant 8 bits of Rb) to R0. If value of R0 is zero then set Z (to signal zero result) else clear Z.
	Operation	Rb = (Rb << 8) + (Rb >> 24) R0 = (R0 << 8) + (Rb & 0xff) × Ra
	Flags	Z
MSTEP.U	Syntax	**MSTEP.U <R0>, Rb, Ra**
	Description	Perform an unsigned multiply step, using eight bits of data taken from Rb, keeping 40 bits of a potential 64 bit result. Add (Ra multiplied by the least significant 8 bits of Rb) to R0 and retain the 40 bit result (shown as temp below). Store the most significant 32 bits of the result (temp) in R0. Shift Rb right by 8 bits. Store the least significant 8 bits of the first result (temp) in the most significant 8 bits of Rb. If value of R0 is zero then set Z (to signal zero result) else clear Z.
	Operation	temp = R0 + Ra × (Rb & 0xff) R0 = temp >> 8 Rb = (Rb >> 8) + ((temp & 0xff) << 24)
	Flags	Z

Note: In the case of the MSTEP64 instruction above the “temp” variable is a 40 bit variable and all calculations are performed using 40 bit unsigned arithmetic. All other calculations use 32 bit unsigned arithmetic.

17.12.25 NEG, Negate

NEG	Syntax	**NEG Rb, Ra, cc_A**
	Description	If condition CONDCA is true, then move the 2's complement of the contents of register Ra into register Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = (- R[a]) else NOP
	Flags	N, Z

17.12.26 NOP, No Operation

NOP	Syntax	**NOP**
	Description	No operation. The NOP instruction puts the PCP in low-power operation.
	Operation	no operation
	Flags	None

17.12.27 NOT, Logical NOT

NOT	Syntax	**NOT Rb, Ra, cc_A**
	Description	If condition CONDCA is true, then move the 1's complement of the contents of register Ra into register Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = NOT(R[a]) else NOP
	Flags	N, Z

17.12.28 OR, Logical OR

OR	Syntax	**OR Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then perform a bitwise logical OR of the contents of register Ra and the contents of register Rb; place the result in Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[b] OR R[a] else NOP
	Flags	N, Z
OR.F	Syntax	**OR.F Rb, [Ra], Size**
	Description	Perform a bitwise logical OR of the contents of the address location, specified by the contents of register Ra, and the contents of register Rb; place the result in Rb.
	Operation	R[b] = R[b] OR zero_ext(FPI[R[a]])
	Flags	N, Z
OR.PI	Syntax	**OR.PI Ra, [#offset6]**
	Description	Perform a bitwise logical OR of the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6, and the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] OR PRAM[(DPTR<<6) + zero_ext(#offset6)]
	Flags	N, Z

17.12.29 PRI, Prioritize

PRI	Syntax	**PRI Rb, Ra, cc_A**
	Description	If condition CONDCA is true, then find the bit position of the most significant 1 in register Ra and put the number into register Rb. The bit location, 31..0, is encoded as a 5-bit number stored in Rb[4:0]. If the contents of Ra is zero, bit Rb[5] is set, while all other bits in Rb are cleared. If CONDCA is false, no operation is performed.

	Operation	if (CONDCA = False) then
		  NOP
		else
		  if (R[a] = 0) then
		  R[b] = 0x20
		  else
		  R[b] = bit_pos(most_significant_1(R[a]))

	Flags	N, Z

17.12.30 RL, Rotate Left

RL	Syntax	**RL Ra, #imm5**
	Description	Rotate the contents of register Ra to the left by the number of bit positions specified through the 5-bit value imm5. The values defined for imm5 are 1, 2, 4 and 8. The carry flag, R7.C, is set to the last bit shifted out of bit 31 of register Ra.
	Operation	tmp = R[a] R[a] = R[a] << imm5; imm5 = 1, 2, 4, 8 R7_C = last bit shifted out of R[a] tmp = tmp >> 32 - imm5 R[a] = tmp OR R[a]
	Flags	N, Z

17.12.31 RR, Rotate Right

RR	Syntax	**RR Ra, #imm5**
	Description	Rotate the contents of register Ra to the right by the number of bit positions specified through the 5-bit value imm5. The values allowed for imm5 are 1, 2, 4 and 8.
	Operation	tmp = R[a] R[a] = R[a] >> imm5; imm5 = 1, 2, 4, 8 tmp = tmp << 32 - imm5 R[a] = tmp OR R[a]
	Flags	N, Z

17.12.32 SET, Set Bit

SET	Syntax	**SET Ra, #imm5**
	Description	Set bit imm5 of register Ra to 1.
	Operation	R[a][imm5] = 1
	Flags	None
SET.F	Syntax	**SET.F [Ra], #imm5, Size**
	Description	Set bit imm5 of the address location specified through the contents of register Ra to 1. This instruction is executed using a locked read-modify-write FPI Bus transaction.
	Operation	FPI[(R[a])][imm5] = 1
	Flags	None

17.12.33 SHL, Shift Left

SHL	Syntax	**SHL Ra, #imm5**
	Description	Shift the contents of register Ra to the left by the number of bit positions specified through the 5-bit value imm5. The values allowed for imm5 are 1, 2, 4 and 8. The carry flag, R7.C, is set to the last bit shifted out of bit 31 of register Ra. Zeros are shifted in from right.
	Operation	R[a] = R[a] << imm5; imm5 = 1, 2, 4, 8 R7_C = last bit shifted out of R[a]
	Flags	N, Z, C

17.12.34 SHR, Shift Right

SHR	Syntax	**SHR Ra, #imm5**
	Description	Shift the contents of register Ra to the right by the number of bit positions specified through the 5-bit value imm5. The values allowed for imm5 are 1, 2, 4 and 8. Zeros are shifted in from left.
	Operation	R[a] = R[a] >> imm5; imm5 = 1, 2, 4, 8
	Flags	N, Z

17.12.35 ST, Store

ST.F	Syntax	**ST.F Rb, [Ra], Size**
	Description	Store the contents of register Rb to the address location specified by the contents of register Ra. When the Size is byte or half-word, the data is stored with the internal LSB (bit 0) properly aligned to the correct FPI Bus byte or half-word lane.
	Operation	FPI[R[a]] = R[b]
	Flags	None
ST.IF	Syntax	**ST.IF [Ra], #offset5, Size**
	Description	Store the contents of R0 to the address location specified by the addition of the contents of register Ra and the value specified by imm5. When the Size is byte or half-word, the data is stored with the internal LSB (bit 0) properly aligned to the correct FPI Bus byte or half-word lane.
	Operation	FPI[R[a] + zero_ext(imm5)] = R[0]
	Flags	None
ST.P	Syntax	**ST.P Rb, [Ra], cc_A**
	Description	If condition CONDCA is true, then store the contents of Rb to the PRAM address location specified by the addition of the contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value Ra[5:0]. If condition CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then PRAM[(DPTR<<6) + zero_ext(R[a][5:0])] = R[b] else NOP
	Flags	None
ST.PI	Syntax	**ST.PI Ra, [#offset6]**
	Description	Store the contents of register Rb to the PRAM location specified by the addition of the contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6.
	Operation	PRAM[(DPTR<<6) + zero_ext(offset6)] = R[b]
	Flags	None

17.12.36 SUB, 32-Bit Subtract

SUB	Syntax	**SUB Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then subtract the contents of register Ra from the contents of register Rb; place the result in Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[b] - R[a] else NOP
	Flags	N, Z, V, C
SUB.I	Syntax	**SUB.I Ra, #imm6**
	Description	Subtract the zero-extended immediate value imm6 from the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] - zero_ext(imm6)
	Flags	N, Z, V, C
SUB.F	Syntax	**SUB.F Rb, [Ra], Size**
	Description	Subtract the zero-extended contents of the address location specified by the contents of register Ra from the contents of register Rb; place the result in Rb.
	Operation	R[b] = R[b] - zero_ext(FPI[R[a]])
	Flags	N, Z, V, C
SUB.PI	Syntax	**SUB.PI Ra, [#offset6]**
	Description	Subtract the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6 from the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] - PRAM[(DPTR<<6) + zero_ext(#offset6)]
	Flags	N, Z, V, C

17.12.37 XCH, Exchange Instructions

XCH.F	Syntax	**XCH.F Rb, [Ra], Size**
	Description	Exchange contents of Rb and FPI address specified by the contents of register Ra. When Size is byte or half word, the value is stored with the internal LSB (bit 0) properly aligned to the correct FPI byte or half-word lane.
	Operation	Rb = FPI[Ra] and FPI[Ra] = Rb
	Flags	N, Z
XCH.PI	Syntax	**XCH.PI Ra, [#offset6]**
	Description	Perform exchange of the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6, and the contents of register Ra;
	Operation	R[a] = PRAM[(DPTR<<6) + zero_ext(#offset6)] and PRAM[(DPTR<<6) + zero_ext(#offset6)] = R[a]
	Flags	N, Z

17.12.38 XOR, 32-Bit Logical Exclusive OR

XOR	Syntax	**XOR Rb, Ra, cc_A**
	Description	If the condition CONDCA is true, then perform a bitwise logical Exclusive-OR of the contents of register Ra and the contents of register Rb; place the result in Rb. If CONDCA is false, no operation is performed.
	Operation	if (CONDCA = True) then R[b] = R[b] XOR R[a] else NOP
	Flags	N, Z
XOR.F	Syntax	**XOR.F Rb, [Ra], Size**
	Description	Perform a bitwise logical Exclusive-OR of the contents of the address location, specified by the contents of register Ra, and the contents of register Rb; place the result in Rb.
	Operation	R[b] = R[b] XOR zero_ext(FPI[R[a]])
	Flags	N, Z
XOR.PI	Syntax	**XOR.PI Ra, [#offset6]**
	Description	Perform a bitwise logical Exclusive-OR of the contents of the PRAM location specified by the addition of contents of the PRAM Data Pointer, shifted left by six bits, and the zero-extended 6-bit value offset6, and the contents of register Ra; place the result in Ra.
	Operation	R[a] = R[a] XOR PRAM[(DPTR<<6) + zero_ext(#offset6)]
	Flags	N, Z

17.12.39 Flag Updates of Instructions

Most instructions update the state flags in R7. In **Table 17-16**, each instruction is shown with the flags that it updates.

Table 17-16 Flag Updates

Instruction	CN1Z	V	C	N	Z
ADD	–	yes	yes	yes	yes
AND	–	–	–	yes	yes
BCOPY	yes[1)]	–	–	–	–
CHKB	–	–	yes	–	–
CLR	–	–	–	–	–
COMP	–	yes	yes	yes	yes
COPY	yes	–	–	–	–
DEBUG	–	–	–	–	–
DINIT	–	yes	–	–	yes
DSTEP	–	–	–	–	–
EXIT	yes	–	–	–	–
INB	–	–	–	–	–
JC	–	–	–	–	–
JL	–	–	–	–	–
LD	–	–	–	yes[2)]	yes
LDL	–	–	–	yes	yes
MCLR	–	–	–	yes	yes
MINIT	–	–	–	–	yes
MOV	–	–	–	yes	yes
MSET	–	–	–	yes	yes
MSTEP.L	–	–	–	–	–
MSTEP.U	–	–	–	–	–
NEG	–	yes	yes	yes	yes
NOP	–	–	–	–	–
NOT	–	–	–	yes	yes
OR	–	–	–	yes	yes
PRI	–	–	–	yes[3)]	yes

Table 17-16 Flag Updates (cont'd)

Instruction	CN1Z	V	C	N	Z
RR	–	–	–	yes	yes
RL	–	–	yes	yes	yes
SET	–	–	–	–	–
SHR	–	–	–	yes	yes
SHL	–	–	yes	yes	yes
ST	–	–	–	–	–
SUB	–	yes	yes	yes	yes
XCH	–	–	–	yes	yes
XOR	–	–	–	yes	yes

1) CN1Z is only modified by the BCOPY, COPY or EXIT instructions and then only when the instruction has been configured to decrement R6.CNT1 (for BCOPY/COPY CNC = 1 or CNC = 2, for EXIT, EC = 1). All other instructions have no effect on the CN1Z flag.

2) When the instruction is LD.I, this flag is always cleared as bit 31 of the result is always 0.

3) This flag is always cleared as bit 31 of the result is always 0.

17.13 Programming of the PCP

In this section, several techniques are outlined to help design Channel Programs. There are also examples on configuring a Channel Program's context.

17.13.1 Initial PC of a Channel Program

A Channel Program can begin operation at the Channel Entry Table location corresponding to the priority of the interrupt. This is much like an interrupt vector location for that channel in a traditional processor architecture. When the Channel Program is started, the PC is set to 2 times the Channel Number (SRPN). Since the base of the Channel Entry Table is the bottom of the code memory (PCODE) address range, and since each entry in the table is two instructions long, this address computation results in the first instruction of the Channel Program for that SRPN being fetched from memory for execution.

Alternately, the Channel Program can be made to begin executing at whatever address its restored context holds in R7.PC.

If PCP_CS.RCB = 1, then the Channel Program is forced to always start at its Channel Entry Table location regardless of the PC value stored in the CSA. If PCP_CS.RCB = 0, then the Channel Program will simply begin executing at whatever PC value is restored in the context R7.PC.

It is important to be aware of the implications of these two approaches on how code memory should be configured, and what the initial value of the PC should be in the Channel Program's context that is loaded in the PRAM Context Save Area at boot time.

17.13.1.1 Channel Entry Table

When PCP_CS.RCB = 1, the program counter of the PCP is vectored to the appropriate channel entry table each time a channel program is invoked by the receipt of an interrupt. The PCP is forced to start executing from its channel entry table location regardless of its previous context or PC state.

If the EXIT instruction is executed with EP = 0, the PC saved during the context save operation will be the channel entry table location for that channel. That means that the next time the Channel Program is started, it will begin operation at the appropriate location in the Channel Entry Table.

Note: If EP = 0 is set in any Channel Program, or if PCP_CS.RCB = 1, a Channel Entry Table must be provided at the base of Code Memory. Otherwise this table is not needed.

17.13.1.2 Channel Resume

When PCP_CS.RCB = 0, the program counter of the PCP is vectored to the address that is restored from the Channel Program's context. This means that before exiting, a Channel Program must itself arrange for where it will resume execution by configuring the value of its PC in its saved context so that it restarts at the desired location.

In this way, arbitrarily complex interrupt-driven state machines can be created as individual Channel Programs. Channel Programs that always start at their beginning, that pick up where they left off, or pick up elsewhere, or that have a mix of these approaches can be constructed.

An example of a restarting Channel Program is shown below. Before exiting, the channel branches back to the address of the START label minus 1 (note that START – 1 = CH16) and then exits. This will leave the next value of the PC in the Channel Program's context as the address of the START label.

```
CH16:                                       ;Channel Program 16
       EXIT EC=1 ST=0 INT=0 EP=1 cc_UC      ;exit, no intr., leave PC @ next
START:                                      ;nominal channel start address
       ST.IF base #0x8 SIZE=32              ;output note from R0
       JC    CH16, cc_UC                    ;loop back before exit
```

Note that when the Channel Program is originally configured by the programmer, the PC field in the R7 context of this Channel Program should also be set to the address of the START label.

Similarly, an interrupt-driven state machine can be created by exiting with the next PC value pointing to the start of the next state in a state machine implemented by the

Channel Program. The next example (see below) shows a program starting at the address to the STATE0 label. It proceeds after the first interrupt to STATE1 – 1, where the Channel Program is left ready for the next state, STATE1 in the state machine. After the next interrupt it executes to address STATE2 – 1 and the Channel Program is left ready for the next state, STATE2. After another interrupt, it proceeds through STATE2. The Channel Program jumps back to START, which is STATE0 – 1. The state machine has gone through one cycle and it is ready to restart in STATE0.

```
;This program is intended to test the sequence of exit/operate just
;as if you were implementing an interrupt driven state machine.
;It requires a periodic sequence of interrupts.

START:
        EXIT EC=1,ST=0,INT=0,EP=1,cc_UC       ;begin exit
STATE0:
        COMP.I   R5,#0x0        ;compare to interrupt number it should be
        JC       ERROR,cc_NZ    ;jump to error routine if not correct
        ADD.I    R5,#0x1        ;increment state number
        EXIT EC=1,ST=0,INT=0,EP=1,cc_UC       ;begin exit
STATE1:
        COMP.I   R5,#0x1        ;compare to interrupt number it should be
        JC       ERROR,cc_NZ    ;jump to error routine if not correct
        ADD.I    R5,#0x1        ;increment state number
        EXIT EC=1,ST=0,INT=0,EP=1,cc_UC       ;begin exit
STATE2:
        COMP.I   R5,#0x2        ;compare to interrupt number it should be
        JC       ERROR,cc_NZ    ;jump to error routine if not correct
        LD.I     R5,#0x0        ;reset state number
        JC       START,cc_UC    ;jump back to start of state machine
```

The last state could just as easily have ended with an EXIT that resets the PC to the Channel Entry Table (EP = 0) rather than jumping back to START.

17.13.2 Channel Management for Small and Minimum Contexts

If Small or Minimum Contexts are being used, only some of the registers are saved and restored. The integrity of the general purpose registers that are *not* included in the context must be handled explicitly by Channel Programs, since these are not saved and restored with the context of the interrupted Channel Program.

Channel Programs may still use all registers reliably. Channel Programs can be so designed that they either ignore the values in unsaved registers, or those registers are used to store constants that no Channel Program changes. Hence they never need to be saved and restored. Alternately, Channel Programs can use these unused general purpose registers as temporary variables as long as the values of such registers can not be corrupted by the interrupt of the channel program by a higher priority channel.

17.13.3 Unused Registers as Globals or Constants

Registers R0 through R3 (for the Small Context model), or R0 through R5 (for the Minimum Context Model) can be used to store constants such as addresses that are available to all Channel Programs. Hence, these registers hold global data, and no Channel Program is allowed to change them.

Since the general purpose registers of the PCP are not directly accessible from the FPI Bus, there does need to be an initial Channel Program that sets these values at or near boot time. There are two choices here. A boot-time interrupt Channel Program can be invoked once to perform initialization, or there can be a program that routinely loads these values as a matter of course, and is invoked at boot time or as upon receipt of the very first interrupt.

17.13.4 Dispatch of Low Priority Tasks

A higher-priority Channel Program may wish to start a low-priority background task, or periodically pause and re-start itself later when there is no other action required at the moment. This can be accomplished in several ways, as follows.

- Post an SRPN to a free SRN on the FPI Bus, then EXIT.
- Perform an EXIT, posting the interrupt to the PCP, and indicating the Channel Number to be started.
- Use a single Channel Program as a list-driven or state-driven task dispatcher.

The first approach is straightforward to program, but uses a system SRN resource. It's advantage is that it allows continuous channel operation without using the interrupt queue or risk blocking other uses of the PCP.

The second approach can be implemented by having a looping Channel Program continue operation in the background. It will also always be superseded by any higher priority tasks.

The third approach uses a Channel Program to dispatch other non-interrupt-driven Channel Programs in an arbitrary order determined by the Channel Program dispatcher. In this way, multiple tasks could be continuously operated without over-using the PCP service-request queue. This approach would be useful when the aim is to poll for Service Requests in the peripheral SRN's rather than having them started by PCP hardware.

17.13.5 Code Reuse Across Channels (Call and Return)

A special Jump instruction is included in the PCP instruction set to allow subroutines to be called from multiple Channel Programs. A routine may be jumped to directly, and then returned from using the JC.IA instruction. JC.IA allows a calling Channel Program to set aside a register for its return address, which will typically be the value of the next PC. The called subprogram can then execute a JC.IA, to the address stored in the register specified, causing a return-from-subroutine operation. The programmer must adopt and

enforce a calling convention to determine which register holds the return address. Register R2 is conventionally used for this purpose.

For example:

```
Main Routine:                          Subroutine:
        LD.IL   R2,#RETURN               SUB:       MOV...
        JC.A    #SUB                                ADD...
RETURN: MOV     ...                                 ...
        ...                                         JC.IA  R2
```

17.13.6 Case-like Code Switches (Computed Go-To)

The JC.I instruction can be used to implement a multi-way branch for branch-on-bit or branch-on-state conditional branches. This instruction allows a conditional relative jump based on an index held in a register. If this instruction is combined with a table of jump addresses, a switch-type statement can be implemented. The default case, that is when the condition code = False, is the next instruction, as is the jump with register index = 0. The table can be any arbitrary length. The index register should be checked for range before the jump into the table is performed.

For Example:

```
         COMP    R3,#5       ;compare R3 to #5 - the number of entries
                             ;in the table
         JC.I    R3,cc_ULE
DEFAULT: JL      #case_0     ;destination if R3 = 0 or condition = false
         JL      #case_1     ;destination if R3 = 1
         JL      #case_2     ;destination if R3 = 2
         JL      #case_3     ;destination if R3 = 3
         JL      #case_4     ;destination if R3 = 4
         JL      #case_5     ;destination if R3 = 5
```

17.13.7 Simple DMA Operation

A simple interrupt driven DMA requires at least the Small Context model to operate properly. Its operation is comprised of three stages, as follows:

- The device interrupts the PCP to indicate it is ready to receive or provide data.
- The PCP moves the amount of data that it is programmed to move
- The PCP finishes and (optionally) interrupts the CPU to notify it that the DMA is complete.

There are two options for implementing this simple DMA Operation.

17.13.7.1 COPY Instruction

A simple DMA Channel Program can consist of just two instructions (COPY and EXIT). In the example below, a device interrupts the PCP to notify it that it has data in its output

buffer, which is four words deep. The COPY instruction copies 4 words to memory at a time. It decrements CNT1 (which is initialized by the CPU in CR6_CNT1 context) after each 4 word transfer. The EXIT command then executes, and if CNT1 was decremented to zero (0), the condition code cause it to issue an interrupt with the value held R6.SRPN.

```
COPY    DST+, SRC, CNC=1, RC0=4, SIZE=32   ; Do the peripheral to
                                            ;  memory DMA
EXIT    EC=0, ST=0, INT=1, EP=0, cc_CNZ     ; Done burst so exit
```

In the example above, the COPY instruction increments the destination held in R5 (DST+), and the source address is left constant in R4 (SRC). All permutations of decrement, increment or do not modify can be applied to either pointer register (R4 and R5) by use of the SRC and DST fields (SRC-, SRC+, or SRC and DST-, DST+, or DST).

Building on this basic DMA method, Scatter and Gather DMA channels can be created.

17.13.7.2 BCOPY Instruction (Burst Copy)

The BCOPY instruction is in principle similar to the COPY instruction except that it uses the FPI Burst mode to perform the transfers rather than performing individual reads/ writes. As for the COPY instruction, the FPI bus is locked between the burst read and burst write to ensure that a valid set of data is transferred. The BCOPY instruction allows support of all burst sizes supported by FPI Burst Mode except a burst size of 1 (i.e. 2, 4 or 8 words). The CNT0 field is used to control the burst size. Both the Source and Destination addresses (R4 and R5) must be correctly aligned for the burst size being used. If either address is incorrectly aligned the PCP will generate an “Illegal Operation” Error Exit.

17.14 PCP Programming Notes and Tips

This section discusses constraints on the use of the PCP and points out some non-obvious issues.

17.14.1 Notes on PCP Configuration

- Only one context model may be used at a time for all channels, and the PCP must remain in that context model once started and configured.
- In order for a specific Channel Program to be enabled, its context must have R7.CEN = 1. If R7.CEN = 0 then the Channel Program will terminate when invoked, and cause a Disabled Channel Request error.
- The Channel Context Address from the FPI Bus as viewed during channel configuration is as follows:
  – Full Context Model: PRAM Base + $20_H \times n$
  – Small Context Model: PRAM Base + $10_H \times n$
  – Minimum Context Model: PRAM Base + $08_H \times n$
  where n is the Channel Number.
- PCP_CS.RCB and context must be consistent. If RCB is configured to 0, then each Channel Program will start at the PC restored from its context. If the wrong address is pre-configured in the context, the Channel Program will not operate properly.
- The programmer of the PCP may lock PCP_CS by setting PCP_CS.EIE = 1. When the global ENDINIT bit is set, the PCP_CS register will no longer be writable, and attempting to do so will cause an FPI Bus error.
- An error condition will result in an interrupt being sent to the local FPI Bus master. The targeted interrupt service routine must be capable of dealing with the cause as recorded in PCP_ES, and, if required, it must be able to return the halted Channel Program to operation. The minimum required to do that is to set the context value of R7.CEN = 1.
- The only PCP register bit that can be dynamically modified during PCP operation is the PCP_CS.EN bit. When writing to any other PCP register bits, one must ensure that the PCP is disabled (PCP_CS.EN = 0) and that the PCP is quiescent (PCP_CS.RS = 0).

17.14.2 General Purpose Register Use

- The most significant 16 bits of R7 may not be written, and will always read back as 0. However, no error will occur if a write to the most significant 16 bits occurs.
- Care must be taken with the use of R6 as a general-use register to ensure that R6 contains the correct value prior to execution of the EXIT command. As R6 contains the CNT1 (counter used in COPY and optionally in EXIT instructions), SRPN and TOS (Service Request number to use during optional interrupt at Channel Program EXIT) fields it is recommended that R6 should not be used to pass values from one invocation of a channel program to the next invocation.

- If PRAM is to be accessed programmatically, then R7.DPTR must be configured properly as a pointer into the PRAM. This points to the 64-word segment that may be addressed by the *xx*.P instructions and the *xx*.PI instructions. It is not recommended to set R7.DTPR to point into the Context Save Area. Special care must be taken that the Context PRAM is not overwritten.
- The programmer must be careful, when updating R7.DTPR (or any other field in R7), not to inadvertently clear R7.CEN. This would cause the Channel Program to generate a disabled channel interrupt to the CPU when the next interrupt request to the channel occurs.
- Any update to the Flags that is caused by an instruction (e.g. MOV R7, R0 which updates Z and N) takes precedence over any explicit bits that are moved to R7. See **Section 17.3.1.5**.
- The interrupt system assumes SRPN 0 is not a request. Full Context packing leaves the least significant 8 × 32-bit entries where channel 0 would normally be un-used. That is, PRAM Base -> PRAM Base + 1 channel. In addition, for Small Context, the least significant 4 × 32-bit entries are un-used, and for Minimum Context the least significant 2 × 32-bit entries are un-used. These “un-used” entries should not be used by channel programs.
- If EP = 0 is used, or if PCP_CS.RCB = 1, a Channel Entry Table must be provided at the base of Code Memory.
- If there is a plan to use the Small or Minimum Context model, and the lower registers are to hold global values, then there needs to be an initial Channel Program that sets these values at or near boot time. There are at least two choices for how to implement this. For instance, a boot interrupt Channel Program can be invoked once to perform initialization, or there can be a program that routinely loads these values as a matter of course, and it is invoked at boot time, or at the very first interrupt. See **Section 17.13.3**.
- When using Small or Minimum Context models and allowing a channel to be interrupted, care must be taken to ensure that the value of any registers that are not included in the context but are being used by a channel are not corrupted by interruption of the channel and subsequent operation of a higher priority channel. Particular care must be taken when using instructions that use R0 implicitly. If necessary critical instruction sequences should be protected by use of the R7.IEN bit.

17.14.3 Use of Channel Interruption

When a channel program consists of only a few instructions, it is best to configure the channel to be non-interruptible. This increases overall efficiency by removing the context save/restore overhead that would be incurred if the channel is interruptible.

17.14.3.1 Dynamic Interrupt Masking

A channel program can dynamically control whether it can be interrupted by use of the R7.IEN bit. When masking interrupts (by clearing R7.IEN), it must be noted that there is a delay of one instruction before the mask becomes effective. As a result the instruction that clears R7.IEN must be placed at least one instruction before the instruction sequence that is to be uninterruptible. As an example consider the following sequence:

```
Formatting
CLR R7, IEN; Clear the R7.IEN bit
          ; Interrupt can occur here
NOP
         ; Interrupt can occur here
         ; First instruction of non-interruptible code sequence
```

17.14.3.2 Control of Channel Priority (CPPN)

The PCP provides three extended Service Request Nodes (PCP_SRC9, PCP_SRC10 and PCP_SRC11) which allow storage of suspended channel interrupt requests. This allows interrupt nesting to a depth of four. This limit on the nesting depth carries the danger that a high priority service request will not be serviced because the PCP's interrupt nesting depth has been exceeded.

It is recommended that a four level grouping scheme should be adopted to avoid this problem. All PCP interrupt sources should be listed in order of their SRPNs(service Request Priority Numbers). This list should then be subdivided into four contiguous groups, Group 0 being the lowest priority and Group 3 being the highest. The context save area for each channel program should be configured such that CR6.CPPN contains the SRPN value of the highest channel program within the group to which the channel belongs. As each channel starts the Operating Priority (CPPN) of the channel is loaded from the context. AS a result, with the scheme recommended above, any channel program will run with the priority of the highest SRPN within the group. As a result the channel can only be interrupted by a service request from a higher priority group (e.g. a Group 0 channel program can be interrupted by a new service request for a channel in any group from 1 to 3, a group 2 channel program can only be interrupted by a new service request for a channel in group 3).

Note: When using this scheme, each channel program must ensure that, prior to channel exit, the R6.CPPN field contains the appropriate value to ensure that when the channel is next invoked it will run at the correct priority.

17.14.4 Implementing Divide Algorithms

As discussed in **Section 17.12.4**, a divide algorithm ***must*** always start with a DINIT instruction followed by a number of DSTEP instructions (up to four depending on the data width that is required). Prior to execution of any DSTEP instruction R0 always contains either 0 (if this is the first DSTEP instruction in a divide sequence R0 contains

0 due to the preceding DINIT instruction), or the remainder from the previous DSTEP instruction). The dividend to be used in this step is generated in R0 by taking 256 × the remainder of the last DSTEP instruction (R0 << 8) and adding the most significant byte of Rb (Rb >> 24) as the least significant byte of the new dividend.

Since the remainder of the last DSTEP instruction is by definition always less than the divisor (Ra) it can be guaranteed that the result of the division of the dividend (calculated as above) by the divisor (Ra) can always be contained within an 8 bit result. The description given in **Section 17.12.14** only holds true under this condition. If the restrictions on the use of the DSTEP instruction (specified within **Section 17.12.4**) are adhered to then the above condition is always met and this description of the instruction is correct. Failure to adhere to these conditions will lead to invalid results which are outside the scope of this document.

During execution of a divide sequence Rb is used both to compile the final divide result and to hold the remnants of the original dividend. For example in a 32-/32-bit divide sequence (which consists of 4 DSTEP instructions - see below) Rb will have the following content:

- After the 1st DSTEP instruction:
  The least significant 3 bytes (24 bits) of the original 32-bit dividend (held in the most significant 3 bytes of Rb) and the most significant byte of the final result (held in the least significant byte of Rb).
- After the 2nd DSTEP instruction:
  The least significant 2 bytes (16 bits) of the original 32-bit dividend (held in the most significant 2 bytes of Rb) and the most significant 2 bytes of the final result (held in the least significant 2 bytes of Rb).
- After the 3rd DSTEP instruction:
  The least significant byte of the original 32-bit dividend (held in the most significant byte of Rb) and the most significant 3 bytes of the final result (held in the least significant 3 bytes of Rb).
- After the final DSTEP instruction:
  The 32 bit final result.

Note that the DSTEP instruction ***always*** uses the divisor as a 32 bit value. In any divide sequence the dividend can be 8, 16, 24 or 32 bits (according to the number of DSTEP instructions in the sequence) but the divisor is ***always*** 32 bits. Prior to the DINIT instruction the dividend must always occupy the appropriate most significant bits within the 32 bit dividend register (Rb).

Divide Examples

Example of a 32/32 bit divide (R5 / R3):

```
DINIT   R5, R3                          ;Initialize ready for the divide
JC      HANDLE_DIVIDE_BY_ZERO, cc_V     ;V flag was set so jump to divide
                                        ;by zero error handler
```

```
DSTEP   R5, R3                      ;4 DSTEP instructions
                                    ;(4 * 8 = 32 bit
DSTEP   R5, R3                      ; divide)
DSTEP   R5, R3
DSTEP   R5, R3
```

After this sequence R5 holds the result, R0 the remainder and R3 is unchanged.

Example of a 8/32 bit divide (R4 / R2):

```
RR      R4, 8                       ;Rotate R4 right by 8 to move
                                    ;least significant byte into
                                    ;most significant byte
DINIT   R4, R2                      ;Initialize ready for the divide
JC      HANDLE_DIVIDE_BY_ZERO, cc_V ;V flag was set so jump to divide
                                    ;by zero error handler
DSTEP   R4, R2                      ;DSTEP instruction
                                    ;(1 * 8 = 8 bit divide)
```

After this sequence R4 holds the result, R0 the remainder and R2 is unchanged.

Note that the above example is specified as being a 8/32 bit divide rather than an 8/8 bit divide (see comments above).

17.14.5 Implementing Multiply Algorithms

As discussed in **Section 17.12.4**, a multiply algorithm ***must*** always start with a MINIT instruction followed by a number of MSTEP32 or MSTEP64 instructions. The MSTEP32 instruction is used to compile a multiplication result contained in 32 bits, discarding any overflows. The MSTEP64 instruction is used to compile a 64-bit multiplication result with the least significant 32 bits of the result contained in Rb and the most significant 32 bits of the result contained in R0.

Multiply Examples

Example of a 32 × 8 bit multiply (R4 × R1) yielding a 32 bit result (R4 = 32 bit, R1 = 8 bit):

```
RR       R1, 8              ;Rotate least significant byte of R1 to most
                            ;significant byte
MINIT    R1, R4             ;Initialize ready for multiply
MSTEP32  R1, R4             ;Perform one MSTEP32 instruction
                            ;(8 bit multiply)
```

After this sequence, R0 holds the result, R1 is left unchanged (right rotated by RR instruction then left rotated by MSTEP32 instruction), R4 is unchanged. The result is only valid if there is no overflow (i.e. the product of the 8-bit number in R1 multiplied by the 32-bit number in R4 can be contained within 32 bits). It is the users responsibility to ensure that this is the case. The overflow condition cannot be detected after execution of the multiply sequence.

Example of a 32 × 16 bit multiply (R3 × R2) yielding a 32 bit result
(R3 = 32 bit, R2 = 16 bit):

```
RR       R2, 8         ;Perform two 8 bit rotations (RR instructions)
                       ;to get original least significant 16 bits into
                       ;most significant 16 bits
RR       R2, 8
MINIT    R2, R3        ;Initialize ready for multiply
MSTEP32  R2, R3        ;Perform two MSTEP32 instructions
                       ;(16 bit multiply)
MSTEP32  R2, R3
```

After this sequence R0 holds the result, R2 is left unchanged (right rotated by two RR instructions then left rotated by two MSTEP32 instructions), R3 is unchanged. The comment above regarding overflow also applies to this sequence.

Example of a 32 × 32 bit multiply (R5 × R2) yielding a 64 bit result
(R5 = 32 bit, R2 = 32 bit):

```
MINIT    R2, R5        ;Initialize ready for multiply
MSTEP64  R2, R5        ;Perform 4 MSTEP64 instructions(64 bit multiply)
MSTEP64  R2, R5
MSTEP64  R2, R5
MSTEP64  R2, R5
```

After this sequence R0 and R2 hold the result (most significant word in R0, least significant word in R2), R5 is unchanged. There is no possibility of overflow as the result of 32 × 32 bits can always be contained in 64 bits.

17.15 PCP Implementation in TC11IB

The addresses of the PCP registers and memories in the TC11IB are given in the following subsections:

17.15.1 PCP Memories

In the TC11IB, the location of the registers and the memories sizes of the PRAM and the PCODE are given in **Table 17-17**.

Table 17-17 General Block Address Map

Unit		Address Range	Access Mode		Size
			Read	Write	
PCP	PCP Registers	$F000\ 3F00_H$ - $F000\ 3FFF_H$	see **Table 17-11**		256 Bytes
	Reserved	$F000\ 4000_H$ - $F000\ FFFF_H$	BE	BE	–
	PCP Data Memory (PRAM) (static RAM)	$F001\ 0000_H$ - $F001\ 0FFF_H$	nE, 32	nE, 32	4 KBytes
	Reserved	$F001\ 1000_H$ - $F001\ FFFF_H$	BE	BE	–
	PCP Code Memory (PCODE) (static RAM)	$F002\ 0000_H$ - $F002\ 3FFF_H$	nE, 32	nE, 32	16 KBytes

Note: “BE” means that in case of an access to this address region a bus error is generated.

17.15.2 PCP Register Address Range

In the TC11IB the registers of the PCP are located in the following address range:

– Module Base Address: $F000\ 3F00_H$
  Module End Address: $F000\ 3FFF_H$.
– Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 17-11**)

18 LMB Bus, FPI Buses, and Bus Control

This chapter provides an overview of the internal Local Memory Bus (LMB), the Fast Flexible Peripheral Interconnect (F_FPI) Bus, the Slow Flexible Peripheral Interconnect (S_FPI) Bus, the corresponding Bus Control Units (LCU, BCU0 and BCU1), the LMB-FPI (LFI) Bridge, and the FPI-FPI (FFI) Bridge for the TC11IB. Topics covered in this chapter also include the bus characteristics, bus arbitration, scheduling, prioritizing, error conditions, and debugging support.

18.1 Fast FPI Bus and Slow FPI Bus Overview

In the TC11IB, the TriCore system is located on the LMB Bus, while the peripherals are divided between two FPI Buses based on their performance. The use of two FPI Buses provides better support for the large number of peripherals requiring an FPI Bus. The FPI-FPI Bridge is a bidirectional bridge that splits the 32-bit FPI-Bus into two branches with different FPI-clock speeds (96MHz/48MHz). The Fast FPI Bus (F_FPI) runs at a speed of 96 MHz. This is the FPI Bus to which most of the high-performance peripherals, such as ComDRAM, PCI-FPI etc., are connected. The Slow FPI Bus (S_FPI) runs at half the speed of its fast counterpart. The S_FPI Bus is used to connect other standard peripherals.

These two FPI Buses interconnect most of the functional units of the TC11IB, such as the on-chip peripheral components. The Fast FPI Bus also interconnects the TC11IB to external components by way of the External Bus Controller Unit (EBU). **Figure 18-1** illustrates the FPI Buses and the modules connected with them.

The FPI Buses are designed for quick acquisition by on-chip functional units and quick data transfer. The low setup overhead of the FPI Bus access protocol guarantees fast FPI Bus acquisition, required for time-critical applications.

The FPI Buses are designed to sustain high transfer rates. For example, a peak transfer rate of up to 800 Mbps can be achieved with a 100 MHz bus clock and 32-bit data bus. Multiple data transfers per bus arbitration cycle allow the FPI Bus to operate at close to its peak bandwidth.

Additional features of the FPI Bus include:

- Multiple bus masters supported
- Demultiplexed address/data operation supported
- Clock synchronization supported
- Scalable address and data buses; address bus up to 32-bits wide and data bus up to 64-bits wide
- Data transfer types include 8-, 16-, 32- and 64-bit sizes
- Single- and multiple-data transfers per bus acquisition cycle
- Split read transfers supported
- Burst transfer capability
- Minimized EMI and power consumption by design

Functional units of the TC11IB are connected to the FPI Buses via the FPI Bus interfaces. The FPI Bus interfaces act as bus agents, requesting bus transactions on behalf of their functional unit, or responding to transaction requests.

There are three types of bus agents:

- Master agents can initiate and control FPI Bus transactions
- Slave agents respond to FPI Bus transaction requests to read or write internal registers and memories
- Master-Slave agents support advanced features, such as split read transfer support and error acknowledgement generation. Depending on the type of transaction, these agents may act as master or slave or both.

When an FPI Bus master seeks to initiate a transfer on an FPI Bus, it first signals a request for bus ownership. When bus ownership is granted, it initiates an FPI Bus read or write transaction. The unit targeted by the transaction becomes the FPI Bus slave, and responds with the requested action.

Some functional units operate only as slaves, while others can operate as either masters or slaves. The FFI Bridge typically operates as Slow FPI Bus master. The LFI bridge typically operates as Fast FPI Bus master. Standard on-chip peripheral units are typically FPI Bus slaves. The interface type (M/S = Master/Slave interface) of the various modules in the TC11IB can be seen in **Figure 18-1**.

To increase availability of the bus, read transaction may be split into two independent transfers: first, the transfer request with some additional information on data type and block size, and, later, the transfer of the requested data from the slave to the master. For the second part, the slave becomes the master on the bus and the requesting master becomes the slave. Agents supporting this type of transfer are called master-slave agents.

FPI Bus arbitration is performed by the on-chip FPI Bus Control Unit (BCU). In case of bus errors, the BCU generates an interrupt request to the CPU, and can provide debugging information about the error to the CPU.

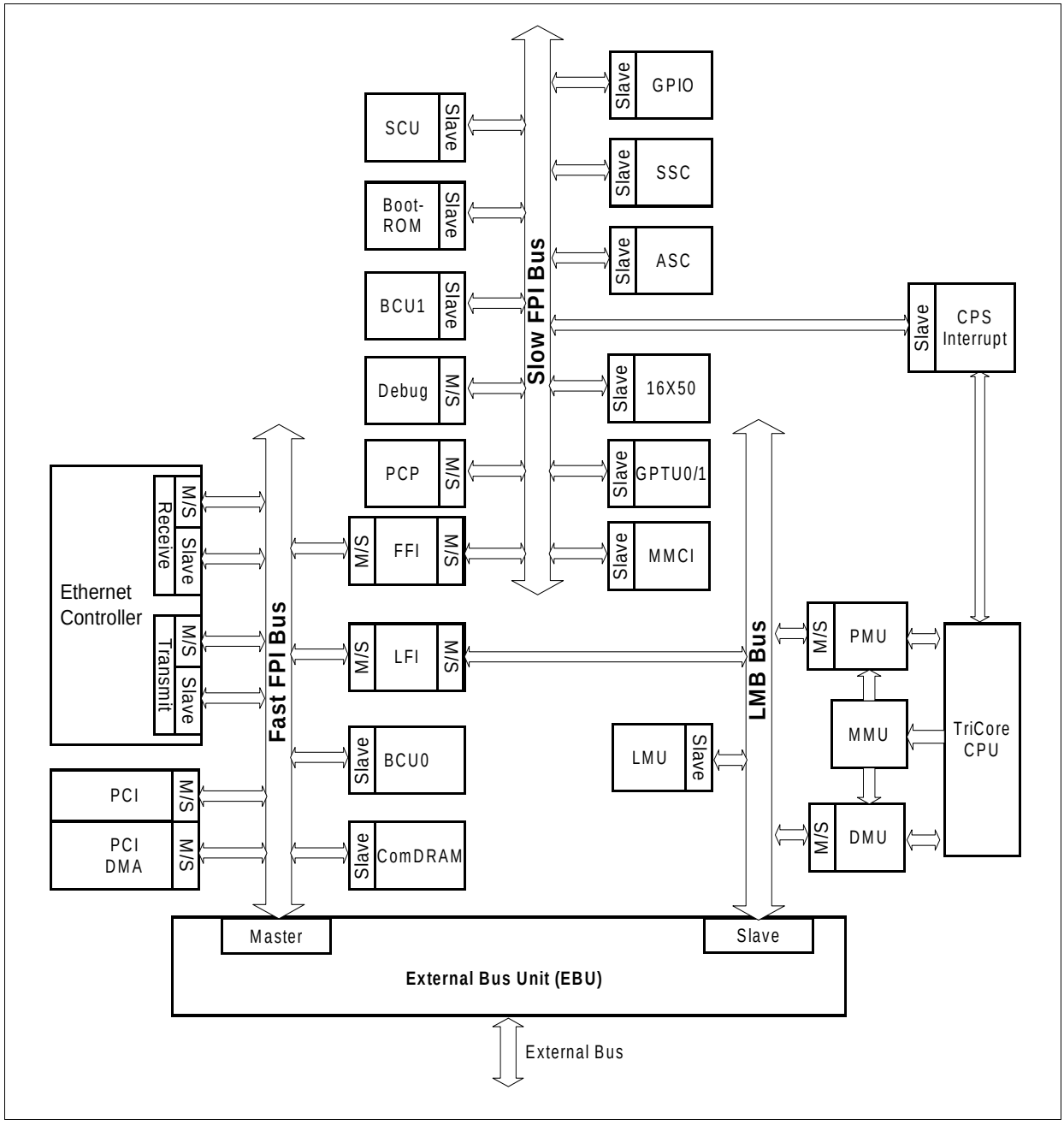


Figure 18-1 TC11IB Bus System Block Diagram

18.2 Local Memory Bus Overview (LMB)

The local memory bus is a synchronous, pipelined, split bus with support for variable block size transfer. This 96 MHz, 64-bit wide data bus supports TriCore system with fast response time and is optimized for speed. It allows the Data memory Unit (DMU) and Program memory Unit (PMU) to have fast access to local memory and reduces the load on the FPI Bus.

All signals on the bus relate to the positive clock edge. The LMB supports 8-,16-, 32-, and 64-bit single beat transactions and variable length 64-bit block transfers. The bus is also designed to support multiple master/slaves and slaves, as shown in **Figure 18-1**.

Features

- Optimized for high-speed and high-performance
- 32-bit address, 64-bit data busses
- Slave controlled wait state insertion
- Address pipelining (max depth - 2)
- Split transactions
- Variable block length: 2, 4, or 8 beats of 64-bit data

The LMB is functionally similar to the FPI Bus. A complete bus transaction includes four phases: Arbitration phase, Address phase, Data phase, and Termination phase. LMB Bus arbitration is performed by the on-chip LMB Bus Control Unit (LCU). In case of bus errors, the LCU generates an interrupt request to the CPU, and can provide debugging information about the error to the CPU.

As shown in **Figure 18-1**, if any master of the LMB Bus—such as the CPU—wants to access any peripheral on the Slow FPI Bus, it must go from the LMB to the Fast FPI Bus via the LFI bridge, then to the Slow FPI Bus via the FFI Bridge. The Data Memory Unit (DMU) typically operates as Fast FPI Bus master.

18.3 FPI-FPI Bridge

Because of the large number of peripherals located on the FPI Bus in the TC11IB, the FPI-FPI Bridge will be used there as a bidirectional bridge to split the 32-Bit FPI- Bus into two branches with different FPI-clock speeds (96MHz/48MHz). Unidirectional FPI-FPI bridge circuitry can be used to interface the two FPI buses at a frequency ratio of 2:1.

The FPI-FPI bridge is transparent. It is a unidirectional bridge that forwards transactions from a primary FPI Bus to a secondary FPI Bus. The forwarding of the read and write transactions is done via an internal pipeline. Hence, it is ensured that the ordering of the transactions will not be changed. The FPI Slave interface of the bridge is connected to the primary FPI Bus and the FPI Master Interface is connected to the secondary FPI Bus. Master and Slave are connected via an internal interface. This internal interface consists of some sideband signals and three buffers: command buffer (cbuf), read-data buffer (rbuf), and write-data buffer (wbuf). The depths of the FIFOs must be adapted to the system needs. For single transaction support, only FIFOs with a depth of one will be needed; but, for single transaction block support data, FIFOs with a depth of eight would be required. For multi-transaction and block transaction support, the command FIFO must be increased as well. The FIFO controllers and FIFO memory modules are placed in the bridge. The bridge is also prepared to support any FIFO depth configuration for the internal buffers, so it can be adapted to different requirements of supported transaction types (single, block) and pipeline depth.

The interface of the bridge to the Fast FPI Bus is defined as primary FPI Bus (p_) and the interface to the Slow FPI Bus is defined as secondary FPI bus (s_). In the TC11IB, the bi-directional FPI-FPI Bridge contains two unidirectional FPI-FPI bridge circuits (shown in **Figure 18-2**). The bridge supports block transactions in both directions, therefore both unidirectional FPI bridges have the same internal FIFO configuration:

- cbuf (command buffer) depth: 1
- wbuf (write buffer) depth: 8
- rbuf (read buffer) depth: 8

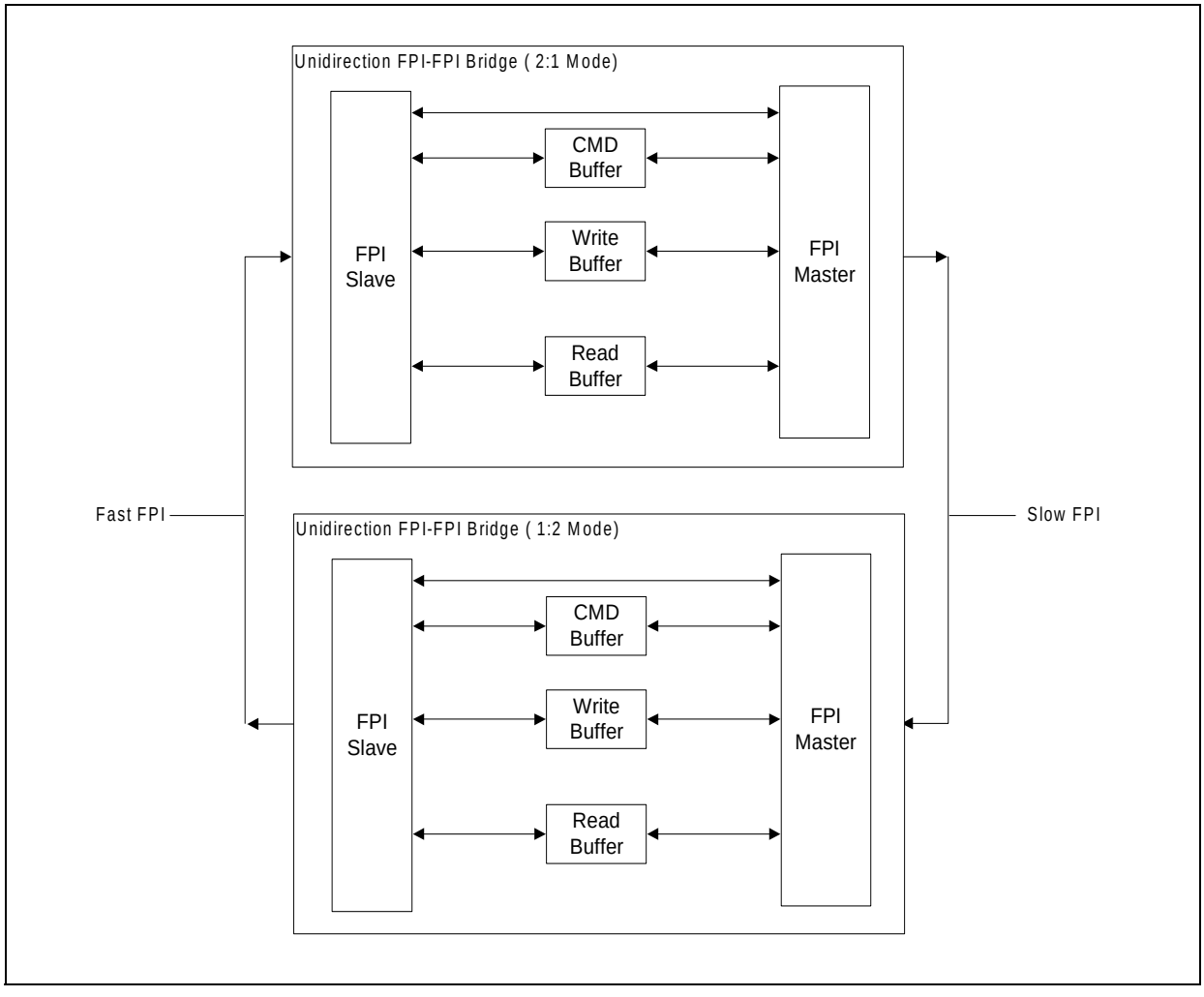


Figure 18-2 FPI-FPI Bridge

Features

- Single/Block Data Read/Write Transactions (8/16/32-Bit)
- FPI-RMW Support
- Optimized for FPI-Bus frequency ratios 2:1
- FPI Master ID (TAG) visible to addressed slaves

- Special Retry/Abort functionality

Note: Block Transaction support depends on generic settings and the depth of the bridge's internal read- and write data buffer.

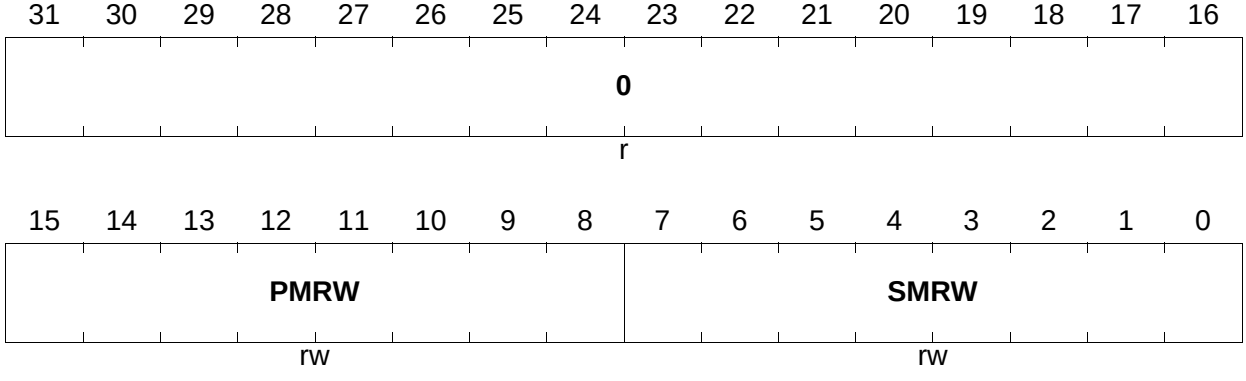
18.3.1 FFI Bridge Control Register

The Register FFI_CON provides flexibility in the configuration of the maximum read wait states used by the slow-to-fast FFI Bridge and the fast-to-slow FFI Bridge.

Table 18-1 FFI Register

Register Short Name	Register Long Name	Absolute Address	Description see
FFI_CON	FFI Control Register	F000 0058$_H$	Page 18-6

FFI_CON
FFI Bridge Control Register **Reset Values: 0000 0404$_H$**



Field	Bits	Type	Description
SMRW	[7:0]	rw	**Secondary Maximum Read Wait State** This value is to be used by the slow-to-fast FFI Bridge to determine maximum read wait state. Access longer than this will be treated as delayed-read access.
PMRW	[15:8]	rw	**Primary Maximum Read Wait State** This value is to be used by the slow-to-fast FFI Bridge to determine maximum read wait state. Access longer than this will be treated as delayed-read access.
0	[31:16]	r	**Reserved**; read as 0; should be written with 0.

18.4 LMB-FPI Bridge

The LMB-FPI(LFI) block provides circuitry to interface (bridge) the Fast FPI Bus to the Local Memory Bus (LMB), as well as to the LMB's Local Bus Control Unit (LBCU). FIFOs are implemented to decouple the transfer of data between the FPI Bus and the LMB Bus. This compensates for mismatches in relative bus performance and allows each bus to operate at its own optimal rate. The block diagram is shown in **Figure 18-3**

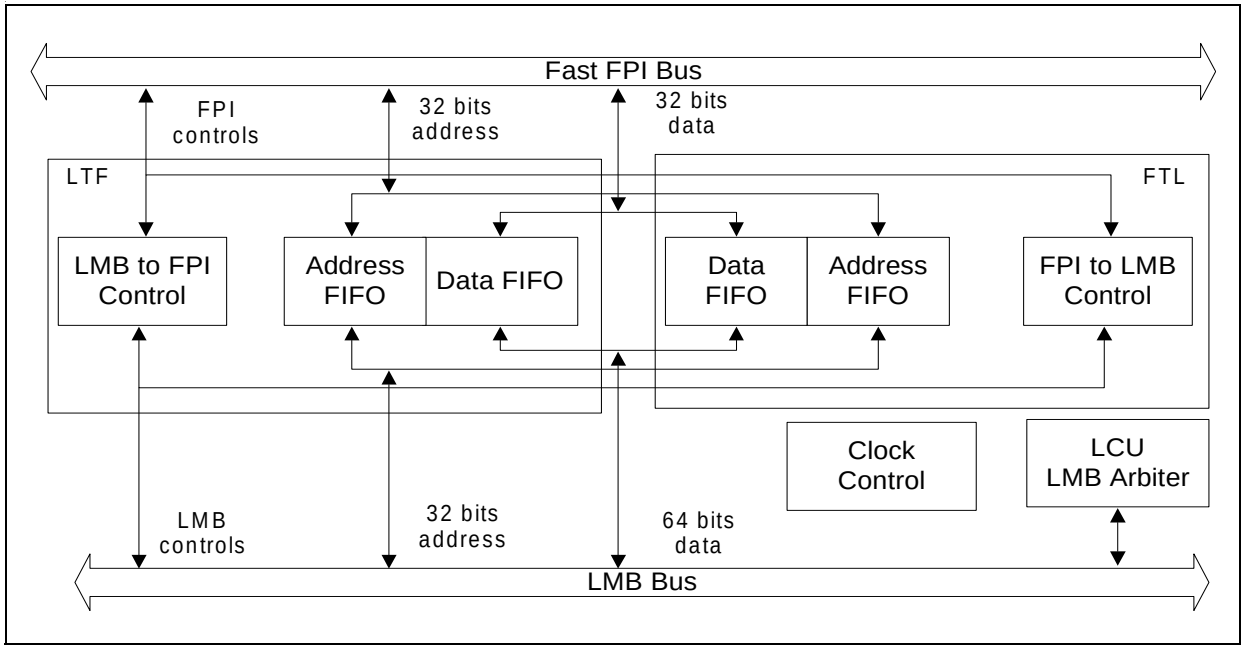


Figure 18-3 LFI Block Diagram

Features

- Burst/Single transactions, from the FPI to the LMB and from the LMB to the FPI.
- Pipelined transactions on both sides of the bridge (pipeline with two stages), with the the exception that FPI transactions to LMB slaves are not pipelined on the LMB.
- Split LMB to FPI read transactions (freeing the LMB during long FPI read transactions). This feature is programmable.
- High efficiency and performance:
  - Fastest access across the bridge takes three cycles, using a bypass.
  - There are no dead cycles on arbitration.
- Supports all FPI data sizes: singles (byte/hword/word) or bursts (BTR2/BTR4/ BTR8).
- Supports most LMB data sizes, singles (byte, hword, word and dword) and bursts (BTR2 and BTR4).
- Can act as the default master on the FPI side.
- Can handle abort, error and retry conditions on both sides of the bridge.
- The LMB clock is off when no transactions are issued to the LFI from both buses and none is in process in the LFI. The clock to LCU is never off.

18.4.1 LFI Configuration Register

The LFI has one configuration register that can be addressed as an LMB slave.

Table 18-2 LFI Register

Register Short Name	Register Long Name	Absolute Address	Description see
LFI_CON	LFI Control Register	F87F FF10$_H$	Page 18-8

LFI_CON
LFI Configuration Register **Reset Value: 0000 0306$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0						1	1	0					1	1	SPLT
r						r	r	r					r	r	rw

Field	Bits	Type	Description
SPLT	0	rw	**Split Mode Functionality** 0 All transactions are taken as non-split transactions. 1 All transaction are taken as split transactions.
1	[9:8] [2:1]	r	Reserved.
0	[31:10] ,[7:3]	r	Reserved. Read as ‘0’ always.

Note: Due to the current implementation of the LFI, it is recommended that bit SPLT should be set to 1 after the CPU has booted code from BootROM. If the CPU is booted from an external memory, then this bit must be set to 1 by the bootcode. Failing to do so may result in loss of read accesses to registers with a destructive effect as well as CPU access to FPI being blocked by any heavy traffic of FPI peripherals accessing the LMB.

18.5 Bus Control Units

Each of the three buses—Local Memory Bus (LMB), Fast FPI Bus (F_FPI), and Slow FPI Bus (S_FPI)—in the TC11IB has its own Bus Control Unit. They are functionally equivalent and differ only in the number of masters they have to arbitrate and the different address maps they have to take care of for error processing.

This section provides a description of the FPI-Bus Control Units. The description applies as well to the LMB-Bus Control Unit, with some exceptions.

The on-chip FPI Bus Control Unit provides bus arbitration, bus error handling, and debug information for error cases. Its design optimizes the speed of bus arbitration. Additionally, it is designed for low power consumption and low EMI.

The BCU arbitrates among the FPI Bus agents to determine the next FPI Bus master. It drives the bus if no other FPI Bus agent is assigned bus ownership to prevent the FPI Bus from electrically floating. It acts as a bus slave when its registers are targeted by an FPI Bus transaction.

Figure 18-4 shows the block diagram of the BCU.

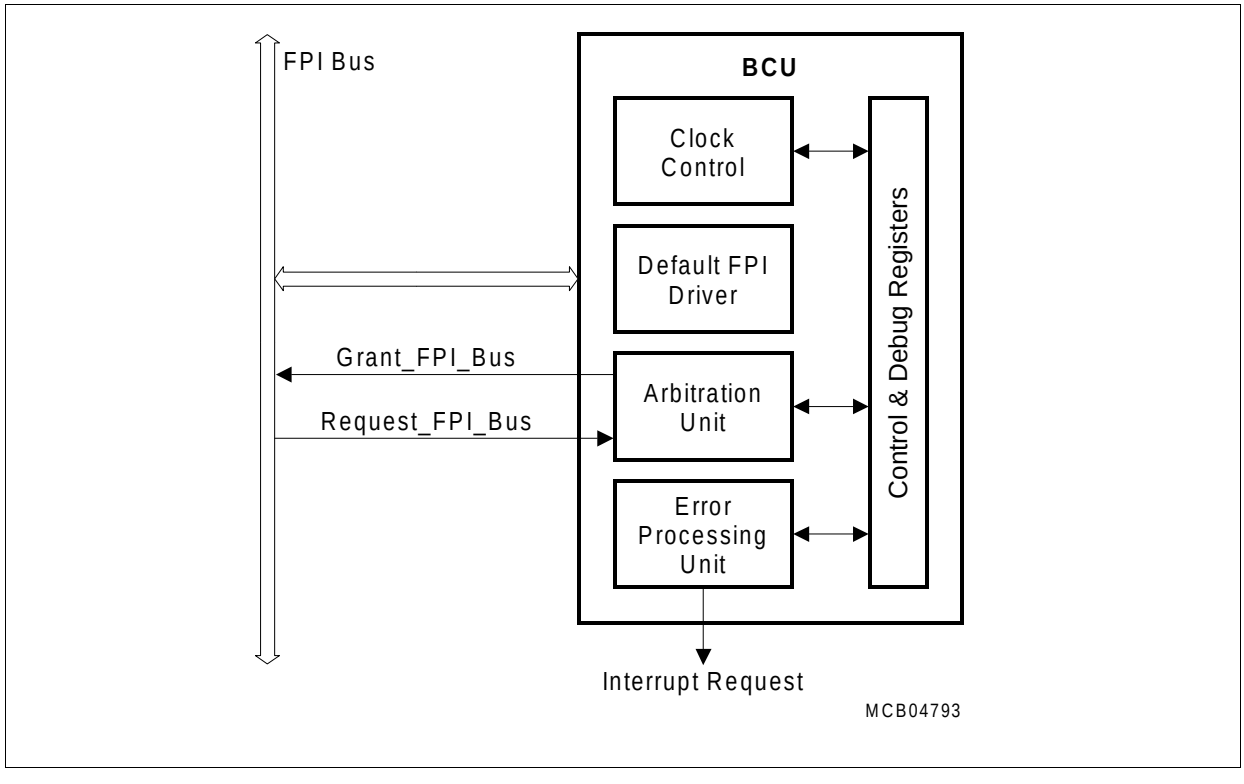


Figure 18-4 FPI Bus Control Unit Block Diagram

The Error Processing Unit is responsible for gathering information and loading the debug registers in the event of a bus error. The default FPI driver becomes active only when no other bus master is able to drive the bus. The Clock Control Unit, if enabled, awakens the BCU only as needed. The “Request_FPI_Bus” lines signal a request to the BCU from

a bus master and the “Grant_FPI_Bus” lines are used to grant bus ownership. The control registers control the general operation of the BCU.

18.5.1 FPI Bus Arbitration

The Arbitration Unit (AB) of the BCU determines whether it is necessary to arbitrate for FPI Bus ownership, and, if so, determines which available bus requestor gets the FPI Bus for the next data transfer. During arbitration, the bus is granted to the requesting agent with the highest priority. If no request is pending, the bus is granted to a default master. If no bus master takes the bus, the BCU itself will drive the FPI Bus to prevent it from floating electrically.

18.5.1.1 Arbitration Priority

Different arbitration priorities apply to the different buses, as shown in **Table 18-3**, **Table 18-4**, **Table 18-5**, and **Table 18-6** (lower numbers are higher priority).

Table 18-3 Priority of LMB-Bus Agents

Priority	Agent	Comment
0 (high)	LFI Bridge	
1	Data Memory Unit (DMU)	Default master
2 (low)	Program Memory Unit (PMU)	

Table 18-4 Priority of F_ FPI-Bus Agents (F_BCU_CON.PMS=0)

Note: F_BCU_CON.PMS=0) is default after Reset

Priority	Agent	Comment
0 (high)	Any bus requestor meeting the starvation protection criteria is assigned this priority	Highest priority; used only for starvation protection
1	PCI Bridge	
2	Ethernet Unit Receiver	
3	Ethernet Unit Transmitter	
4	PCI DMA Contr.	
5	EBU (XMI)	
6	F_FPI-Bus Bridge	
7 (low)	LMB-Bus Bridge	Lowest priority; Default master

Table 18-5 Priority of F_ FPI-Bus Agents (F_BCU_CON.PMS=1)

Priority	Agent	Comment
0 (high)	Any bus requestor meeting the starvation protection criteria is assigned this priority	Highest priority; used only for starvation protection
1	F_FPI-Bus Bridge	
2	PCI Bridge	
3	Ethernet Unit Receiver	
4	Ethernet Unit Transmitter	
5	PCI DMA Contr.	
6	EBU (XMI)	
7 (low)	LMB-Bus Bridge	Lowest priority; Default master

Table 18-6 Priority of S_ FPI-Bus Agents

Priority	Agent	Comment
0 (high)	Any bus requestor meeting the starvation protection criteria is assigned this priority	Highest priority; used only for starvation protection
1	TCU (Test and Debug Unit)	
2	PCP	Default master
3 (low)	FPI-Bus Bridge	Lowest priority

In normal operation, one of the bus masters automatically serves as default master. The bus is granted to this default master whenever there is no request from any other bus master. In this way, the bus is always driven by one of the masters. In some exceptional circumstances, however, the BCU must drive the FPI Bus. These conditions include:

- After reset
- A non-existing module is accessed (error)
- A time-out condition occurs (error)
- No other master can be granted to the FPI Bus

18.5.1.2 Bus Starvation Protection (not LMB Bus)

Because assignment of priorities to these six bus agents is fixed, it is possible that a lower-priority requestor may never be granted the bus if a higher-priority requestor continuously asks for, and receives, bus ownership. To protect against bus starvation of lower-priority masters, an optional feature of the TC11IB can detect such cases and momentarily raise the priority of the lower-priority requestor to the highest priority (above all other priorities), thereby guaranteeing it access.

Starvation protection employs a counter that is incremented each time an arbitration is performed by the BCU. When this counter reaches a user-programmable threshold value, all bus request lines are sampled, and, for each active bus request, a request flag is set in an internal BCU register. This flag is reset automatically when a master is granted the bus.

When the counter reaches the threshold value, it is automatically reset to 0 and starts counting up again. When the next period is finished, the request lines are sampled again. If an active request is detected, for which the request flag set during the last sample is still set, it means that this master was not granted the bus during the previous period. This master will now be set to the highest priority and will be granted service. If there are several masters for which this starvation condition applies, they are served in the order of their hardwired priority ranking.

Starvation protection can be enabled or disabled through the CON.SPE bit. The sample period of the counter is programmed through the CON.SPC bit field. This bit field should be set to a value at least equal to or greater than the number of masters. Its reset value is 40_H.

18.5.2 Error Handling

Two classes of error condition can arise on the FPI Bus:

- A slave indicates a severe problem—such as an unaligned data access request, by returning an error code instead of an acknowledge
- A time-out is detected for the current bus operation (indicates a non-responding slave)

Three classes of error condition can arise on the Local Memory Bus:

- A master accesses the un-implemented slave address
- A slave indicates a severe problem with an error acknowledge code
- Split Response fails

A bus error condition causes the BCU to issue an interrupt request to the selected service provider (usually the CPU), and, if enabled, causes the BCU to capture information about the bus error condition for debugging.

Bus error information gathering is enabled by default. It can be disabled by setting the CON.DBG bit to 0. If a bus error occurs when enabled, the status of the bus, including address, data, and the control information, is captured into registers EADD, EDAT, and

ECON, respectively. Kernel software must read the debug information in response to the interrupt to examine and resolve the problem.

Note: If the CPU itself caused the bus error either through a load/store operation via the DMU or an instruction fetch operation via the PMU, a bus trap is issued to the CPU in addition to the interrupt issued by the BCU. To handle this condition, the trap routine in the kernel software must read the BCU error status registers and then clear the interrupt request from the BCU.

Interpreting the BCU Error Information

Some knowledge about the operation of the internal FPI Bus is required in order to interpret the captured information in case of a bus error. Although the address and data values captured in registers EADD and EDAT, respectively, are self-explanatory, the captured FPI Bus control information needs some more explanation.

Register ECON captures the state of the read (RDN), write (WRN), Supervisor Mode (SVM), acknowledge (ACK), ready (RDY), abort (ABT), time-out (TOUT), identification (TAG), and operation code (OPC) lines of the FPI Bus.

The read and write signals are active-low. For regular read or write accesses, only one of these lines is activated (set to 0). There is one special case defined for the FPI Bus. If a master performs a read-modify-write transaction (for example, to modify a bit in a peripheral register), this transaction is indicated by both lines, read and write, being activated in the first access (read access).

The supervisor mode signal is set to 1 for an access in Supervisor Mode, and set to 0 for an access in User Mode. The ready signal indicates the end of a transfer. It is normally driven to 1 in the (last) data cycle. During wait state insertion, ready is driven to 0.

Under certain conditions, a master can abort a transfer that has already started. This is indicated with the abort signal set to 0.

The time-out signal indicates if there was no response on the bus to an access, and the programmed time (via CON.TOUT) has elapsed. TOUT is set to one in this case. An acknowledge code must be driven by the selected slave during each data cycle of an access.These codes are listed in **Table 18-7** and **Table 18-8**.

Table 18-7 FPI Bus Acknowledge Codes

Code (ACK)	Description
00_B	NSC: No Special Condition.
01_B	ERR: Bus Error, last data cycle is aborted.
10_B	SPT: Split Transaction (not used in the TC11IB)
11_B	RTY: Retry. Slave can currently not respond to the access. Master needs to repeat the access later.

Table 18-8 LMB Bus Acknowledge Codes

Code (ACK)	Description
000_B	NSC: No Special Condition.
001_B	SPT: Split Transaction
010_B	RTY: Retry. Slave can currently not respond to the access. Master needs to repeat the access later.
011_B	ERR: Error Transaction.
111_B	BTNS: Block Transfer Not Supported.
others	Reserved

Each master on the FPI Bus is assigned a 4-bit identification number, the TAG (see **Table 18-9**). This makes it possible to distinguish which master has performed the current transaction.

Table 18-9 FPI Bus TAG Assignments in the TC11IB

TAG-Number	Module	Description
0	TCU	Test Control Unit
1	EBU	Master part of External Bus Controller
2	PCP	Peripheral Control Processor
3	DMU	Data Memory Unit
4	PMU	Program Memory Unit
5	PCI	PCI DMA
6	ERU	Ethernet Receive Unit
7	ETU	Ethernet Transmit Unit
8	PCI	PCI Bridge
9..15	---	Reserved

Transactions on the FPI Buses and the LMB Bus are classified via a 4-bit operation code, listed in **Table 18-10** and **Table 18-11**. Note that the split transactions (OPC = 1000_B to 1110_B) are not used for the FPI Buses in the TC11IB.

Table 18-10 FPI Bus Operation Codes (OPC)

OPC	Description	OPC	Description
0000_B	Single Byte Transfer (8-bit)	1000_B	Split Block Transfer Request (1 transfer)
0001_B	Single Half-Word Transfer (16-bit)	1001_B	Split Block Transfer Request (2 transfers)
0010_B	Single Word Transfer (32-bit)	1010_B	Split Block Transfer Request (4 transfers)
0011_B	Single Double-Word Transfer (64-bit)	1011_B	Split Block Transfer Request (8 transfers)
0100_B	2-Word Block Transfer	1100_B	Split Block Response
0101_B	4-Word Block Transfer	1101_B	Split Block Failure
0110_B	8-Word Block Transfer	1110_B	Split Block End
0111_B	Reserved	1111	No operation

Table 18-11 LMB Bus Operation Codes (OPC)

OPC	Description	OPC	Description
0000_B	Single Byte Transfer (8-bit)	1000_B	Split Block Transfer Request (2 transfers)
0001_B	Single Half-Word Transfer (16-bit)	1001_B	Split Block Transfer Request (4 transfers)
0010_B	Single Word Transfer (32-bit)	1010_B	Split Block Transfer Request (8 transfers)
0011_B	Single Double-Word Transfer (64-bit)	1011_B	Reserved
0100_B	Reserved	1100_B	Split Block Response (2 transfers)
0101_B	Split Response Failure	1101_B	Split Block Response (4 transfers)
0110_B	Reserved	1110_B	Split Block Response (8 transfers)
0111_B	Reserved	1111	Split Single Data Transfer Response

Block Transfer (BTRx)

These opcodes are used to initiate block data read or write transfers of various block lengths: 2 to 4 double words.

Split Response Single Data Transfer (SSDTx)

This opcode is used by slaves that support split transaction. The slave drives this opcode during a split transaction response along with the ID of the former requestor on the Tag bus and the prior sent target address on the Address bus. The response data size (byte, half, word or double), is identical to the original request with a single data beat. This response can also be used for multiple beat responses on a split block transfer request.

Split Response Block Transfer (BSTRx)

This opcode is used by slaves that support split transaction. The slave drives this opcode during a split transaction response along with the ID of the former requestor on the Tag bus and the prior sent target address on the Address bus. The response block lengths: 2 to 8.

Split Response Failure (SRF)

This opcode is used by slaves that support split transaction. The slave drives this opcode together with the prior sent target address on the Address bus and the ID of the former requestor on the Tag bus to inform the former requestor that it can not return a valid data response due to an error, such as a parity error.

18.5.3 BCU Power Saving Mode

The BCU can be configured so that it shuts down automatically when not needed by disabling its internal clock. When it is needed again—for instance, when a bus request signal is received from a master—the BCU will enable its clock and perform the arbitration. If no further bus activity is required after the transfer has completed, the BCU will automatically shut off its clock and return to idle mode.

Automatic power management is controlled through the CON.PSE bit. When set to 1, power management is disabled, and the BCU clock is always active. This might be required, for instance, to debug both the active and idle FPI Bus states of an application via an external emulator or other debugging hardware.

18.5.4 F_FPI Bus Address Map

The BCU on the F_FPI bus must react on valid addresses being sent out by the master agents.

Table 18-12 F_FPI Bus Address Map

Seg	Address	Size	Target	Action
15	F880 0000$_H$ - FFFF FFFF$_H$	120 MB	Reserved	Generate bus error
	F800 0000$_H$ - F87F FFFF$_H$	8 MB	LMB area	LFI Bridge
	F7E2 0000$_H$ - F7FF FFFF$_H$	15 x 128 KB	-	Generate bus error
	F7E1 0000$_H$ - F7E1 FFFF$_H$	64 KB	Core SFR	Select the module
	F7E0 FF00$_H$ - F7E0 FFFF$_H$	256 B	CPS	Select the module
	F200 0600$_H$ - F7E0 FEFF$_H$	~ 94 MB	-	Generate bus error
	F200 0000$_H$ - F200 05FF$_H$	6x256 byte	F_FPI area	Select the module
	F100 0000$_H$ - F1FF FFFF$_H$	16 MB	PCI Configuration Space	Select the module
	F050 0000$_H$ - F0FF FFFF$_H$	11 MB	-	Generate bus error
	F040 0000$_H$ - F04F FFFF$_H$	1 MB	PCI-FPI Bridge	Select the module
	F000 0000$_H$ - F03F FFFF$_H$	4 MB	S_FPI area	FPI Bridge
14	E880 0000$_H$ - EFFF FFFF$_H$	120 MB	-	Generate bus error
	E000 0000$_H$ - E87F FFFF$_H$	136 MB	LMB area	LFI Bridge
13	DF00 0000$_H$ - DFFF FFFF$_H$	16 MB	S_FPI area	FPI Bridge
	D800 0000$_H$ - DEFF FFFF$_H$	112 MB	LMB area	LFI Bridge
	D000 0000$_H$ - D7FF FFFF$_H$	128 MB	-	Generate bus error
12	C000 0000$_H$ - CFFF FFFF$_H$	256 MB	-	Generate bus error
11	BFFF 0000$_H$ - BFFF FFFF$_H$	1 MB	-	Generate bus error
	BFE0 0000$_H$ - BFEF FFFF$_H$	1 MB	Com-DRAM	Select the module
	B000 0000$_H$ - BFDF FFFF$_H$	254 MB	PCI Space	Select the module

Table 18-12 F_FPI Bus Address Map (cont'd)

10	A000 0000$_H$ - AFFF FFFF$_H$	256 MB	LMB area	LFI Bridge
9	9FF0 0000$_H$ - 9FFF FFFF$_H$	1 MB	-	Generate bus error
	9FE0 0000$_H$ - 9FEF FFFF$_H$	1 MB	ComDRAM (mirrored)	Select the module
	9000 0000$_H$ - 9FDF FFFF$_H$	254 MB	PCI Space (mirrored)	Select the module
8	8000 0000$_H$ - 8FFF FFFF$_H$	256 MB	LMB area	LFI Bridge
0..7	0000 0000$_H$ - 7FFF FFFF$_H$	2 GB	PCI	Select the module

Note: If 'empty' addresses are accessed within module areas, the module will generate an access error if required. Otherwise, the access will be terminated by a bus timeout.

18.5.5 S_FPI Bus Address Map

The BCU on the S_FPI bus must react on valid addresses being sent out by the master agents.

Table 18-13 S_FPI Bus Address Map

Seg	Address	Size	Target	Action
15	F040 0000$_H$ - FFFF FFFF$_H$	252 MB	LMB/F_FPI area	FPI Bridge
	F010 0000$_H$ - F03F FFFF$_H$	3 MB	Reserved	
	F000 0000$_H$ - F00F FFFF$_H$	1 MB	Infineon Peripherals	Select the module
14	E000 0000$_H$ - EFFF FFFF$_H$	256 MB	LMB/F_FPI area	FPI Bridge
13	DFFF C000$_H$ - DFFFFFFF$_H$	~16 KB	Boot ROM	Select the module
	DF00 0000$_H$ - DFFF BFFF$_H$	~16 MB	Reserved BootROM space	Generate bus error
	D000 0000$_H$ - DEFF FFFF$_H$	240 MB	LMB area	FPI Bridge
12	C000 0000$_H$ - CFFF FFFF$_H$	256 MB	LMB area	FPI Bridge
11	B000 0000$_H$ - BFFF FFFF$_H$	256 MB	F_FPI area	FPI Bridge
10	A000 0000$_H$ - AFFF FFFF$_H$	256 MB	LMB area	FPI Bridge
9	9000 0000$_H$ - 9FFF FFFF$_H$	256MB	F_FPI area	FPI Bridge
8	8000 0000$_H$ - 8FFF FFFF$_H$	256 MB	LMB area	FPI Bridge
0..7	0000 0000$_H$ - 7FFF FFFF$_H$	2 GB	F_FPI area	FPI Bridge

Note: If 'Reserved' addresses are accessed within module areas, the module will result in unexpected behavior in some pins or generate an access error if required. Otherwise, the access will be terminated by a bus timeout.

18.5.6 BCU Registers

The BCU registers can be divided into three types, as shown in **Figure 18-5**.

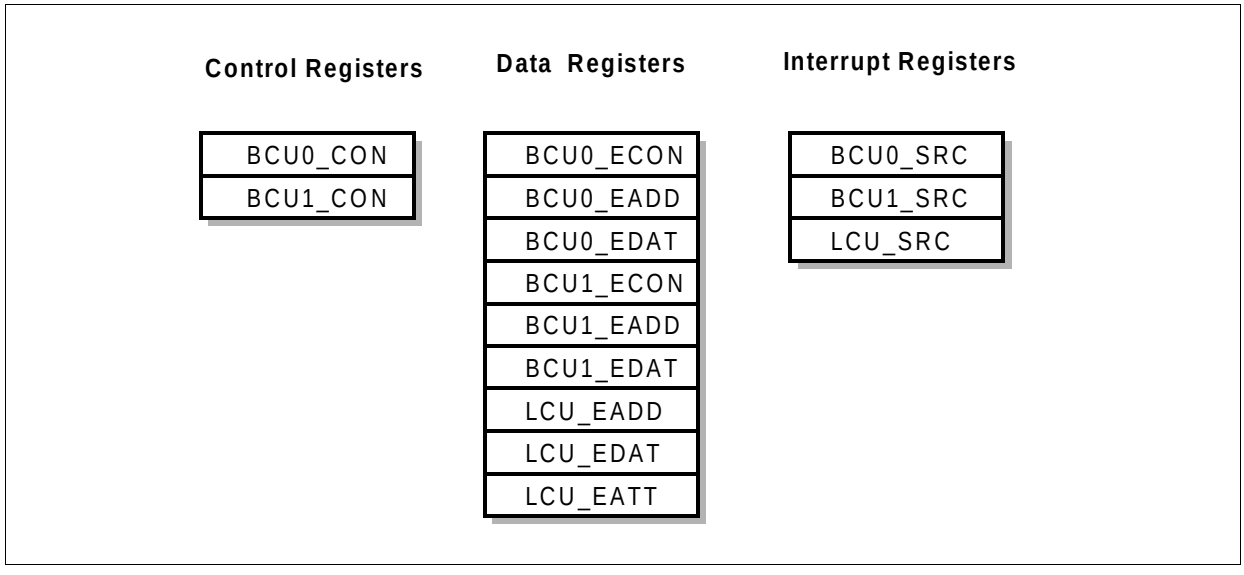


Figure 18-5 BCU Registers

Table 18-14 BCU Registers

Register Short Name	Register Long Name	Offset Address	Description see
BCU0_CON	BCU0 Control Register	0010_H	**Page 18-20**
BCU0_ECON	BCU0 Error Control Capture Register	0020_H	**Page 18-24**
BCU0_EADD	BCU0 Error Address Capture Register	0024_H	**Page 18-25**
BCU0_EDAT	BCU0 Error Data Capture Register	0028_H	**Page 18-25**
BCU0_SRC	BCU0 Service Request Control Register	F000 $0C04_H$	**Page 18-28**
BCU1_CON	BCU1 Control Register	0010_H	**Page 18-22**
BCU1_ECON	BCU1 Error Control Capture Register	0020_H	**Page 18-24**
BCU1_EADD	BCU1 Error Address Capture Register	0024_H	**Page 18-25**
BCU1_EDAT	BCU1 Error Data Capture Register	0028_H	**Page 18-25**
BCU1_SRC	BCU1 Service Request Control Register	$00FC_H$	**Page 18-28**
LCU_EATT	LCU1 Error Attribute Capture Register	0020_H	**Page 18-27**

Table 18-14 BCU Registers

Register Short Name	Register Long Name	Offset Address	Description see
LCU_EADD	LCU1 Error Address Capture Register	0024_H	Page 18-26
LCU_EDAT	LCU Error Data Capture Register	0028_H	Page 18-26
LCU_SRC	LCU Service Request Control Register	$00FC_H$	Page 18-28

In the TC11IB, the registers of the BCU are located in the address range:

- BCU0 Module Base Address: F200 0000_H
  BCU0 Module End Address: F200 $00FF_H$
- BCU1 Module Base Address: F000 0200_H
  BCU1 Module End Address: F000 $02FF_H$
- LCU Module Base Address: F87F $FE00_H$
  LCU Module End Address: F87F $FEFF_H$
- Absolute Register Address = Module Base Address + Offset Address (see Table 18-14)

Note: The BCU allows word accesses only (32-bit) to its control and data registers. Byte and half-word accesses will result in a bus error.

18.5.6.1 BCU Control Register

The BCU Control Register controls the overall operation of the BCU, including setting the starvation sample period, the bus timeout period, enabling starvation-protection mode, and error handling. There is a slight difference in the CON registers for the slow and fast FPI-Bus: the Fast FPI-Bus BCU allows the arbitration priorities to be modified for better debugging support. The LMB-Bus BCU has no control register.

BCU0_CON
BCU0 Control Register **Reset Value: 4009 $FFFF_H$**

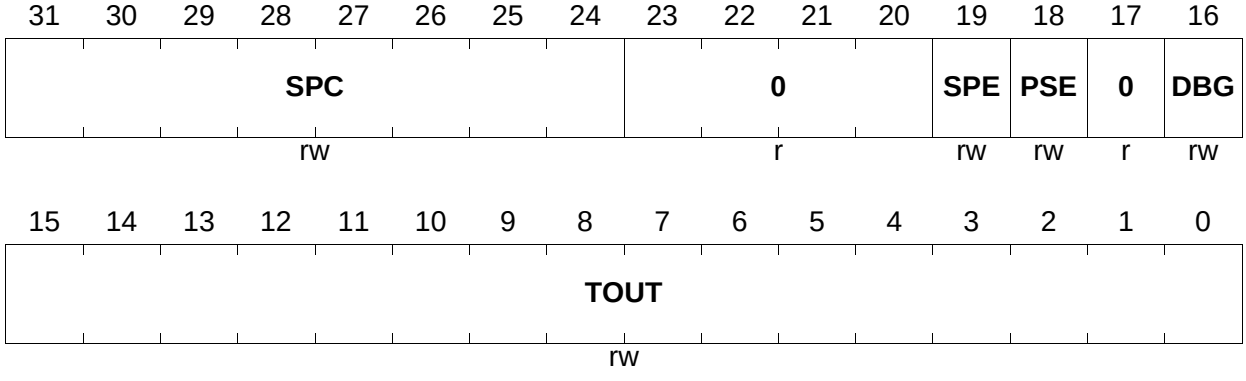
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SPC								0			PMS	SPE	PSE	0	DBG
rw								r			rw	rw	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TOUT															
rw															

Field	Bits	Type	Description
TOUT	[15:0]	rw	**BCU0 Bus Time-Out Value** The bit field defines the number of FPI Bus timeout cycles. Default after reset is $FFFF_H$ (= 65536 bus cycles).
DBG	16	rw	**BCU0 Debug Trace Enable** 0 BCU0 debug trace disabled. No error information captured. 1 BCU0 debug trace enabled (default after reset). Error information is captured in registers BCU0_EADD, BCU0_EDAT, and BCU0_ECON.
PSE	18	rw	**BCU0 Power Saving (Automatic Clock Control) Enable** 0 BCU0 power saving disabled (default after reset) 1 BCU0 power saving enabled
SPE	19	rw	**BCU0 Starvation Protection Enable** 0 BCU0 protection disabled 1 BCU0 protection enabled (default after reset)
PMS	20	rw	**BCU0 Priority Mode Selection** 0 BCU0 standard priority mode (default after reset) 1 BCU0 debug priority mode
SPC	[31:24]	rw	**BCU0 Sample Period Control** Defines the sample period for the starvation counter. Must be larger than the number of masters. The reset value is 40_H.
0	17, [23:21]	r	**Reserved**; read as 0; should be written with 0.

BCU1_CON
BCU1 Control Register

Reset Value: 4009 FFFF$_H$



Field	Bits	Type	Description
TOUT	[15:0]	rw	**BCU1 Bus Timeout Value** The bit field defines the number of FPI Bus time-out cycles. Default after reset is FFFF$_H$ (= 65536 bus cycles).
DBG	16	rw	**BCU1 Debug Trace Enable** 0 BCU1 debug trace disabled. No error information captured. 1 BCU1 debug trace enabled (default after reset). Error information is captured in registers BCU0_EADD, BCU0_EDAT, and BCU0_ECON.
PSE	18	rw	**BCU1 Power Saving (Automatic Clock Control) Enable** 0 BCU1 power saving disabled (default after reset) 1 BCU1 power saving enabled
SPE	19	rw	**BCU1 Starvation Protection Enable** 0 BCU1 protection disabled 1 BCU1 protection enabled (default after reset)
SPC	[31:24]	rw	**BCU1 Sample Period Control** Defines the sample period for the starvation counter. Must be larger than the number of masters. The reset value is 40$_H$.
0	17, [23:21]	r	**Reserved**; read as 0; should be written with 0.

18.5.6.2 BCU Debug Registers

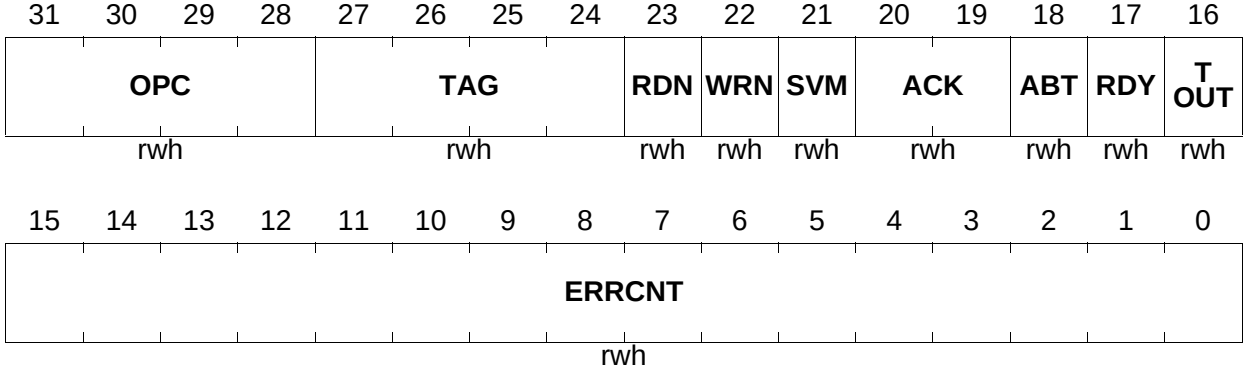
The capture of bus error conditions is enabled by setting CON.DBG to 1. In case of a bus error, information about the condition will then be stored in the BCU debug registers. The BCU debug registers can then be examined by software to help determine the cause of the error.

If enabled, and a bus error occurs, the BCU Error Control Capture Register, ECON, will hold the captured FPI Bus control information, and a count of the number of bus errors. The BCU Error Address Capture Register, EADD, will store the captured FPI Bus address, and the BCU Error Data Capture Register, EDAT, will store the captured FPI Bus data.

If the capture of bus error conditions is disabled (BCU_CON.DBG = 0), these registers remain untouched.

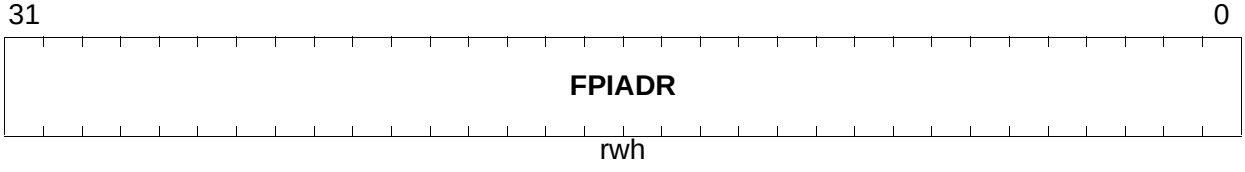
Note: These registers store only the first error. In case of multiple bus errors, an error counter ECON[15:0] shows the number of bus errors since the first error occurred. A hardware reset clears this 16-bit counter to 0, but the counter can be set to any value through software. This counter is prevented from overflowing, so a value of 2^{16} - 1 indicates that at least many errors have occurred, but there may have been more. After ECON has been read, the ECON, EADD and EDAT registers are re-enabled to trace FPI Bus activity.

BCU0_ECON
BCU0 Error Control Capture Register **Reset Value: 0000 0000$_H$**
BCU1_ECON
BCU1 Error Control Capture Register **Reset Value: 0000 0000$_H$**



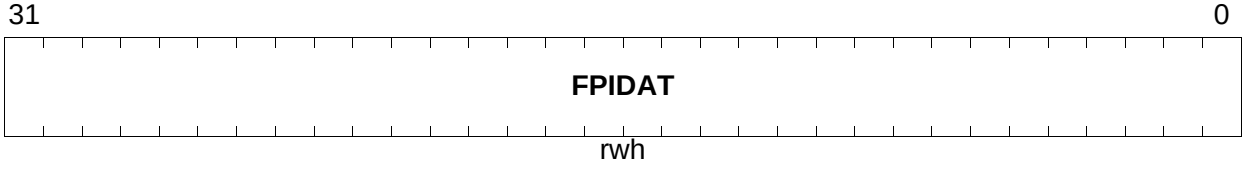
Field	Bits	Type	Description
ERRCNT	[15:0]	rwh	**Number of FPI Bus Error Counter** ERRCNT is incremented on each occurrence of an FPI Bus error. ERRCNT is reset to 0000_H after the BCUx_ECON register is read (x=0,1).
TOUT	16	rwh	**State of FPI Bus Timeout Signal (active high)**
RDY	17	rwh	**State of FPI Bus Ready Signal** (active high)
ABT	18	rwh	**State of FPI Bus Abort Signal** (active low)
ACK	[20:19]	rwh	**State of FPI Bus Acknowledge Signal**
SVM	21	rwh	**State of FP Bus Supervisor Mode Signal** (active high)
WRN	22	rwh	**State of FPI Bus Write Signal** (active low).
RDN	23	rwh	**State of FPI Bus Read Signal** (active low).
TAG	[27:24]	rwh	**FPI Bus Tag Number** see **Table 18-9**
OPC	[31:28]	rwh	**FPI Bus Operation Code** see **Table 18-10**

BCU0_EADD
BCU0 Error Address Capture Register **Reset Value: 0000 0000$_H$**
BCU1_EADD
BCU1 Error Address Capture Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
FPIADR	[31:0]	rwh	**Captured FPI Bus Address (in case of a bus error)** *Note: If there are multiple errors, only the address of the first error is captured.*

BCU0_EDAT
BCU0 Error Data Capture Register **Reset Value: 0000 0000$_H$**
BCU1_EDAT
BCU1 Error Data Capture Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
FPIDAT	[31:0]	rwh	**Captured FPI Bus Data (in case of a bus error)** *Note: If there are multiple errors, only the data for the first error are captured.*

18.5.6.3 LCU Debug Registers

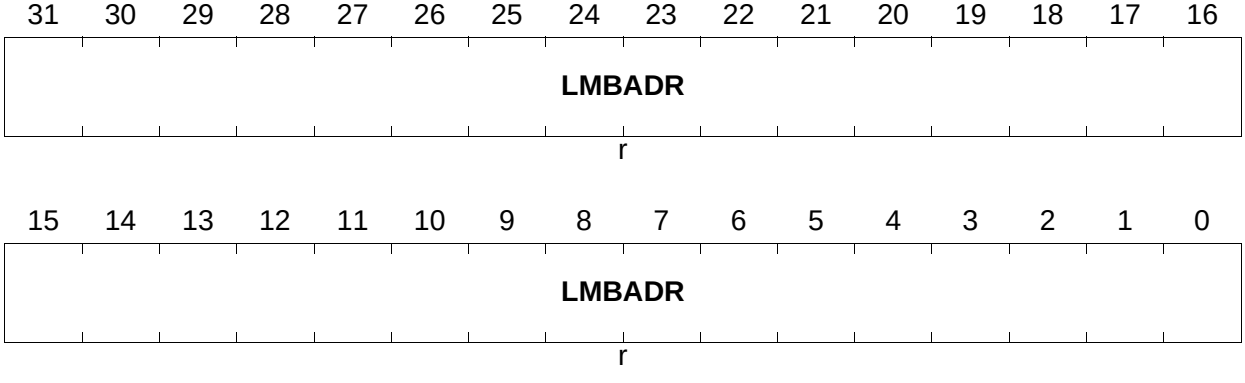
The LCU follows the LMB states. If any error occurs on LMB bus, LCU will issues an LMB error interrupt, and captures the status of the LMB bus, including the address, attributes and data, into registers EADD, EATT and EDAT, respectively. If there are multiple errors, only the information of the first error is captured. Software should read the debug information in response to the interrupt to examine and resolve the problem.

The error capture registers are written on the first error detected on the LMB bus. Once an error transaction is captured, the LEATT lock bit is set. No further error transaction

information will be captured by the error capture registers until the LEATT Lock bit is cleared, by writing 1 into it. After reset, the values of these registers are undefined.

The EADD register captures the Local bus Address of the transaction that got an error indication. If there are multiple errors, only the address of the first error is captured.

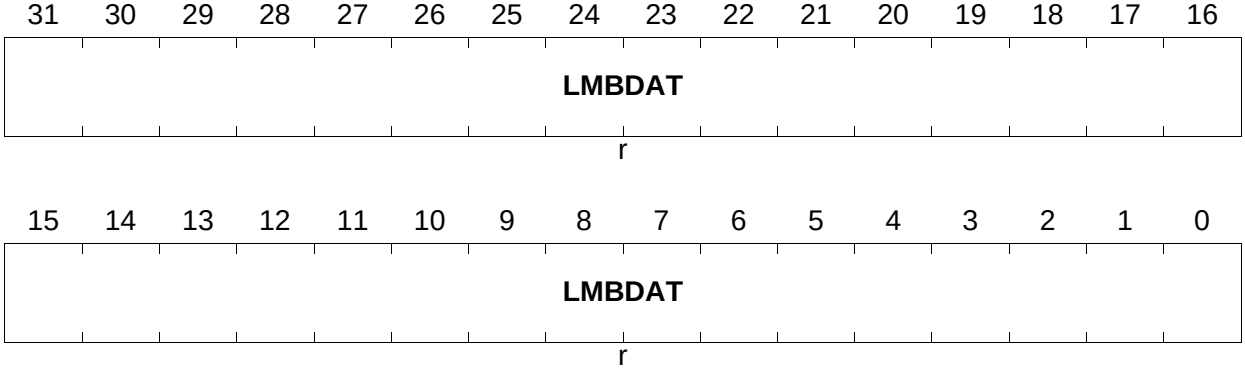
LCU_EADD
Error Address Capture Register **Reset Value: 0000 0000$_H$**



Field	Bit	Type	Description
LMBADR	[31:0]	r	**Captured LMB-Bus Address (in case of a bus error).** *Note: If there are multiple errors, only the address of the first error is captured.*

The EDAT register captures the Local Bus Data of the write transaction that got an error indication. If there are multiple errors, only the data of the first error is captured.

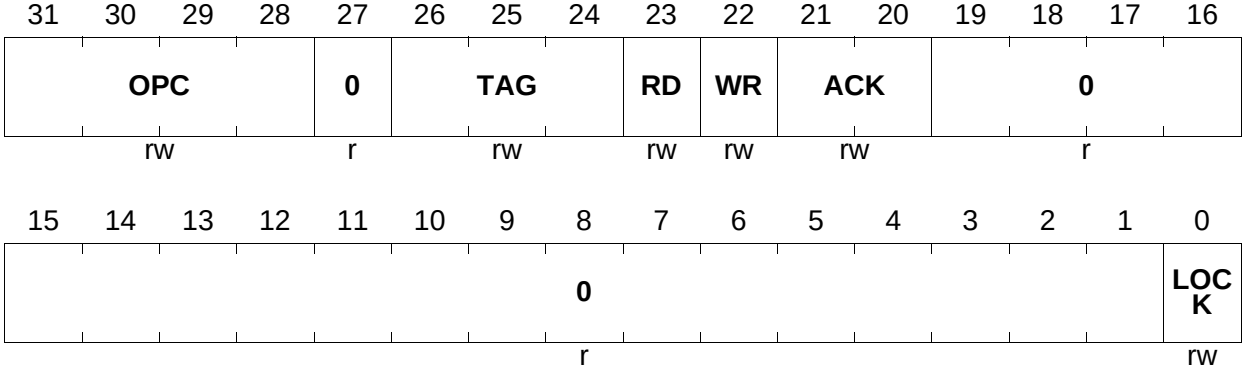
LCU_EDAT
Error Data Capture Register **Reset Value: 0000 0000$_H$**



Field	Bit	Type	Function
LMBDAT	[31:0]	r	**Captured LMB-Bus Data (in case of a bus error).** *Note: If there are multiple errors, only the data for the first error are captured.*

The EATT register captures the Local bus Attributes of the transaction that got an error indication. If there are multiple errors, only the data of the first error is captured.

LCU_EATT
Error Attribute Capture Register **Reset Value: 0000 0000$_H$**

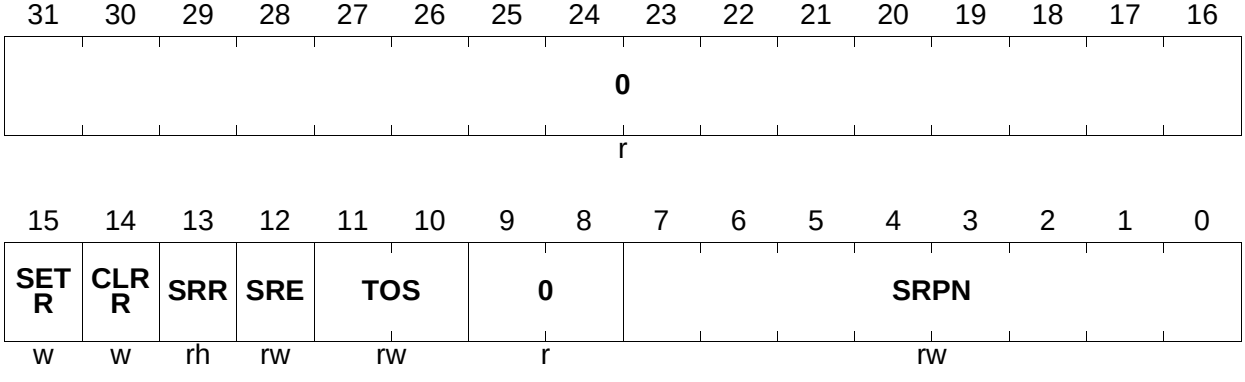


Field	Bit	Type	Description
LOCK	[0]	rw	**Lock Bit**
ACK	[21:20]	rw	**State of LMB-Bus Acknowledge Signal.** see **Table 18-8**
WR	22	rw	**State of LMB-Bus Write Signal (active low).**
RD	23	rw	**State of LMB-Bus Read Signal (active low).**
TAG	[26:24]	rwh	**LMB-Bus Tag Number.** see **Table 18-9**
OPC	[31:28]	rwh	**LMB-Bus Operation Code.** see **Table 18-11**
0	[19:1] 27	r	**Reserved**; read as 0; should be written with 0.

18.5.6.4 BCU Service Request Control Register

In case of a bus error, the BCU generates an interrupt request to the selected service provider (usually the CPU). This interrupt request is controlled through a standard service request control register.

BCU0_SRC
BCU0 Service Request Control Register **Reset Values: 0000 0000$_H$**
BCU1_SRC
BCU1 Service Request Control Register **Reset Values: 0000 0000$_H$**
LCU_SRC
LCU Service Request Control Register **Reset Values: 0000 0000$_H$**



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number**
TOS	[11:10]	rw	**Type of Service Control**
SRE	12	rw	**Service Request Enable**
SRR	13	rh	**Service Request Flag**
CLRR	14	w	**Request Clear Bit**
SETR	15	w	**Request Set Bit**
0	[9:8], [31:16]	r	**Reserved**; read as 0; should be written with 0.

*Note: Further details of interrupt handling and processing are described in **Chapter 15** in this User's Manual.*

19 System Timer

19.1 Overview

This chapter describes the System Timer (STM). The TC11IB's STM is designed for global system timing applications requiring both high precision and long range. The STM has the following features:

- Free-running 56-bit counter
- All 56 bits can be read synchronously
- Different 32-bit portions of the 56-bit counter can be read synchronously
- Driven by clock, f_{STM} = 48 MHz.
- Counting begins at power-on reset
- Continuous operation is not affected by any reset condition except power-on reset

Special STM register semantics provide synchronous views of the entire 56-bit counter, or 32-bit subsets at different levels of resolution.

The maximum clock period is $2^{56} \times 1 / f_{STM}$. At f_{STM} = 48 MHz, for example, the STM counts 47.6 years before overflowing. Thus, it is capable of continuously timing the entire expected product life-time of a system without overflowing.

19.2 Kernel Functions

The STM is an upward counter, running with the system clock frequency ($f_{STM} = f_{SYSCLK}$ = 48MHz). It is enabled per default after reset, and immediately starts counting up. Other than via reset, it is impossible to affect the contents of the timer during normal operation of the application, it can only be read, but not written to. Depending on the implementation of the clock control of the STM, the timer can optionally be disabled or suspended for power-saving and debugging purposes via a clock control register.

Due to the 56-bit width of the STM, it is impossible to read its entire contents with one instruction. It needs to be read with two load instructions. Since the timer would continue to count between the two load operations, there is a chance that the two values read are not be consistent (due to possible overflow from the low part of the timer to the high part between the two read operations). To enable a synchronous and consistent reading of the STM contents, a capture register (CAP), is implemented. It latches the contents of the high part of the STM each time the low part, TIM0, is read. Thus, it holds the upper value of the timer at exactly the same time when the lower part is read. The second read operation would then read the contents of the CAP to get the complete timer value.

The System Timer can also be read in sections from seven registers, TIM0 through TIM6, which select increasingly higher-order 32-bit ranges of the System Timer. These can be viewed as individual 32-bit timers, each with a different resolution and timing range.

Figure 19-1 is an overview on the System Timer module. It shows the options for reading parts of STM contents.

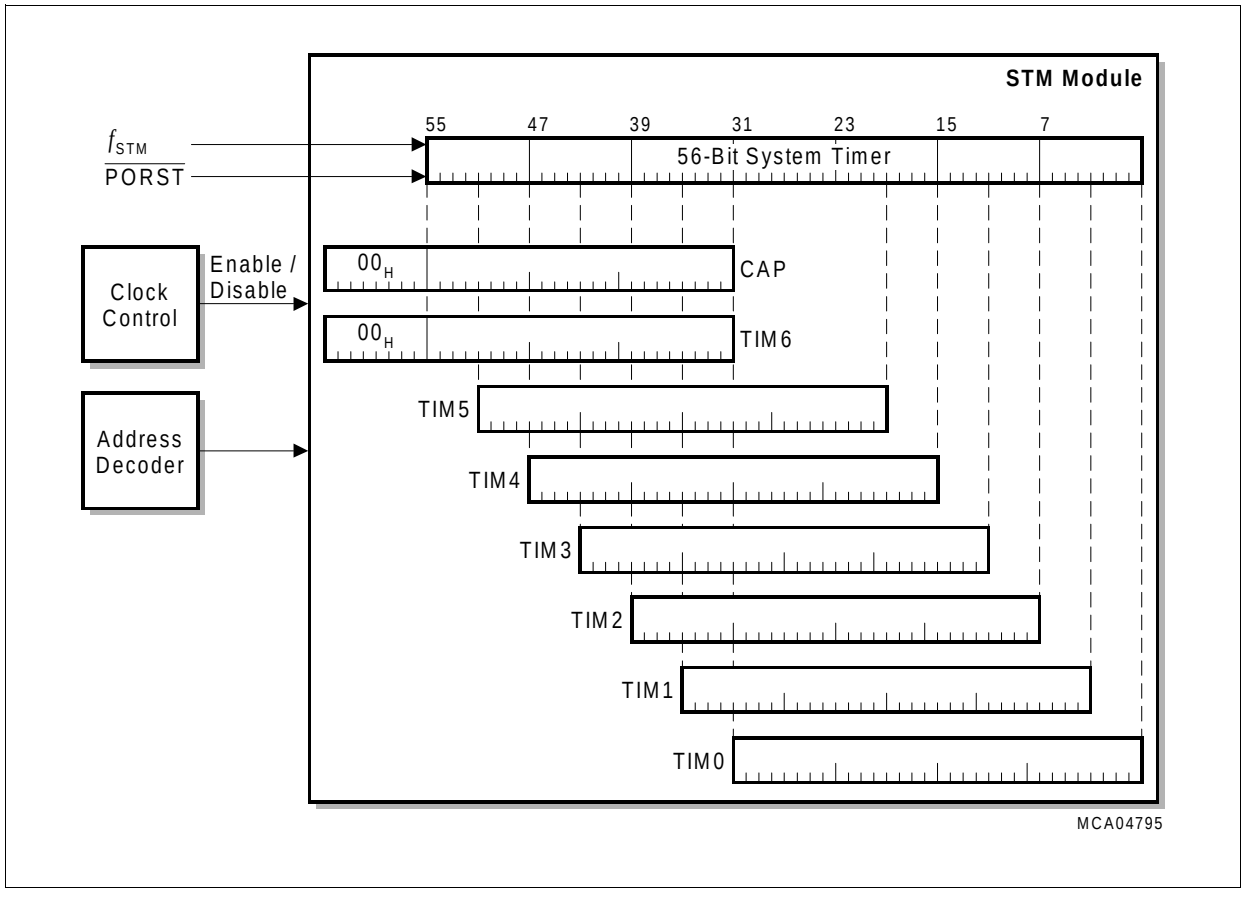


Figure 19-1 General Block Diagram of the STM Module

Table 19-1 is an overview on the individual timer registers with their resolution and timing range. As an example, the values for a 48 MHz system frequency are given.

Table 19-1 System Timer Resolutions and Ranges

Register	STM Bits	Resolution [s]	Range [s]	Example Frequency: 48 MHz $f_{STM} = f_{SYSCLK}$	
				Resolution	Range
TIM0	[31:0]	f_{STM}	$2^{32} / f_{STM}$	20.83 ns	89.48 s
TIM1	[35:4]	$16 / f_{STM}$	$2^{36} / f_{STM}$	333.33 ns	1431.66 s
TIM2	[39:8]	$256 / f_{STM}$	$2^{40} / f_{STM}$	5.33 µs	381.77 min
TIM3	[43:12]	$4096 / f_{STM}$	$2^{44} / f_{STM}$	85.33 µs	101.8 h
TIM4	[47:16]	$65536 / f_{STM}$	$2^{48} / f_{STM}$	1.36 ms	67.87 days
TIM5	[51:20]	$2^{20} / f_{STM}$	$2^{52} / f_{STM}$	21.85 ms	2.98 yr
TIM6	[55:32]	$2^{32} / f_{STM}$	$2^{56} / f_{STM}$	89.48 s	47.6 yr
CAP	[55:32]	$2^{32} / f_{STM}$	$2^{56} / f_{STM}$	89.48 s	47.6 yr

19.3 Kernel Registers

The STM registers can be divided into two types, as shown in **Figure 19-2**.

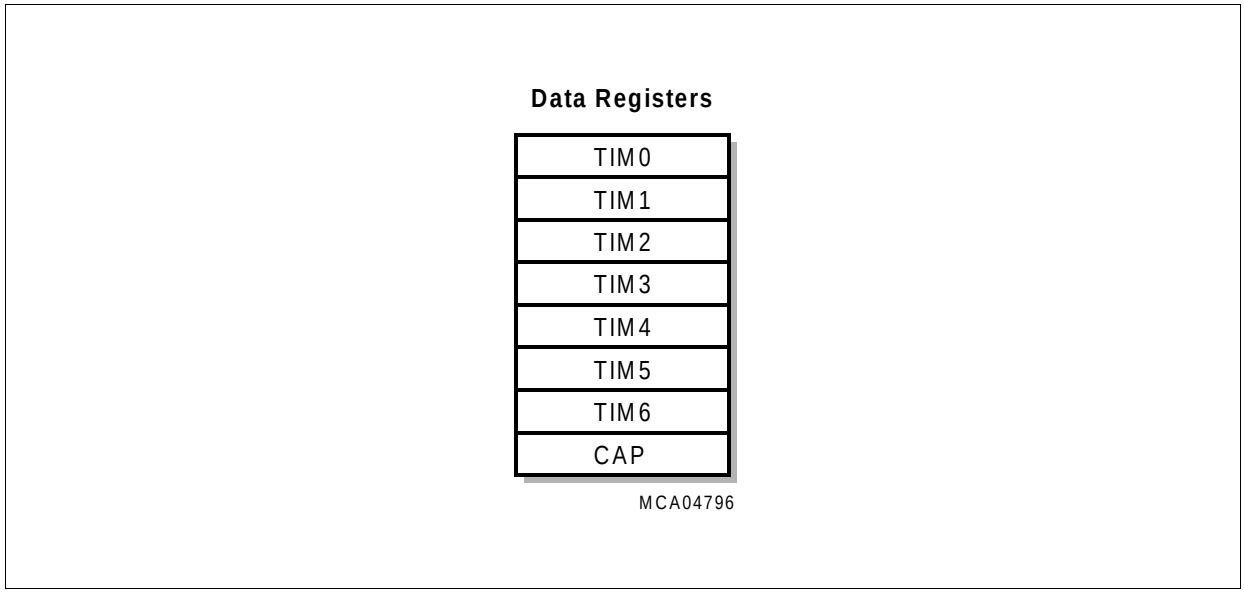


Figure 19-2 SFRs of the STM Module

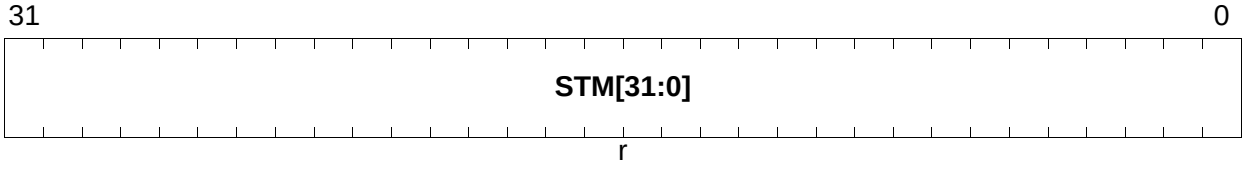
Table 19-2 STM Kernel Registers

Register Short Name	Register Long Name	Offset Address	Description see
TIM0	Timer Register 0	0010_H	**Page 19-5**
TIM1	Timer Register 1	0014_H	**Page 19-5**
TIM2	Timer Register 2	0018_H	**Page 19-5**
TIM3	Timer Register 3	$001C_H$	**Page 19-5**
TIM4	Timer Register 4	0020_H	**Page 19-6**
TIM5	Timer Register 5	0024_H	**Page 19-6**
TIM6	Timer Register 6	0028_H	**Page 19-6**
CAP	Timer Capture Register	$002C_H$	**Page 19-6**

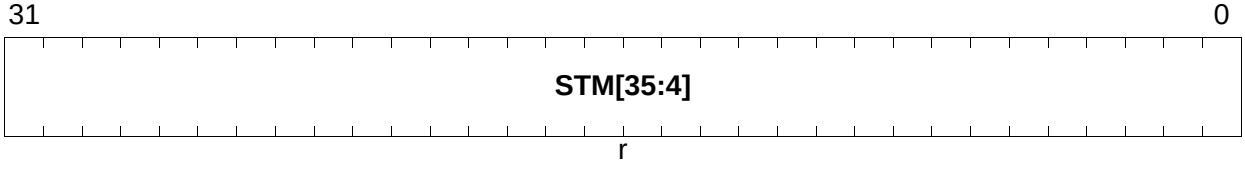
Note: All STM kernel register names described in this section will be referenced in other parts of this TC11IB User's Manual with the module name prefix "STM_".

TIM1 to TIM6 provide 32-bit views at varying resolutions of the underlying STM counter.

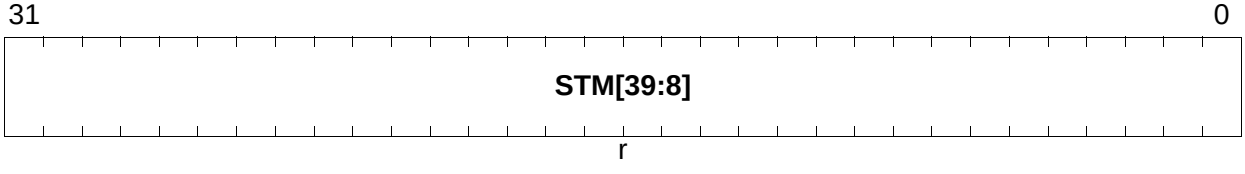
TIM0
Timer Register 0 **Reset Value: 0000 0000$_H$**



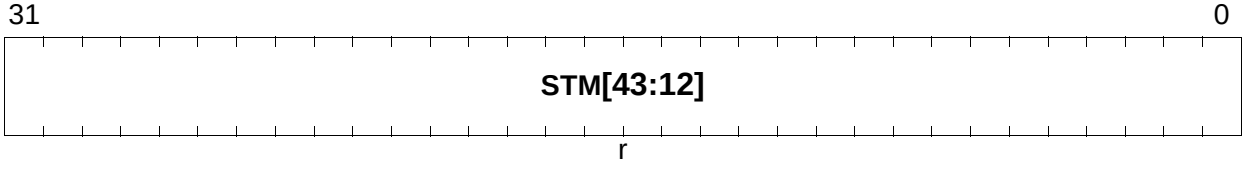
TIM1
Timer Register 1 **Reset Value: 0000 0000$_H$**



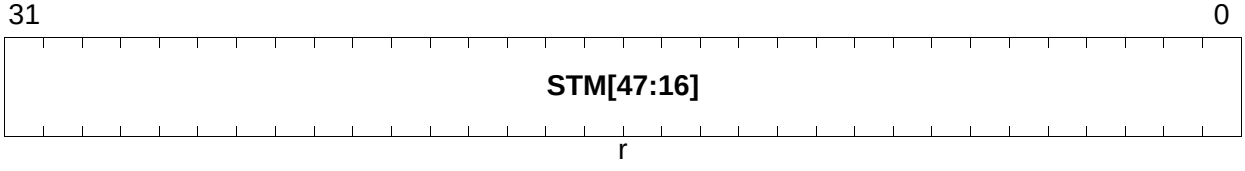
TIM2
Timer Register 2 **Reset Value: 0000 0000$_H$**



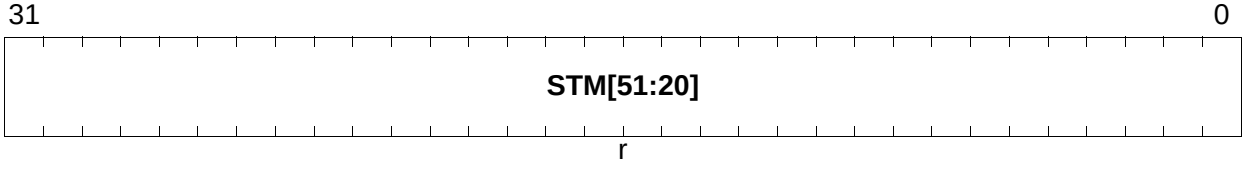
TIM3
Timer Register 3 **Reset Value: 0000 0000$_H$**



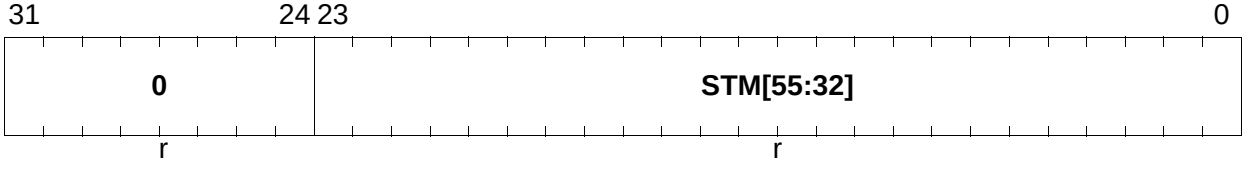
TIM4
Timer Register 4 **Reset Value: 0000 0000$_H$**



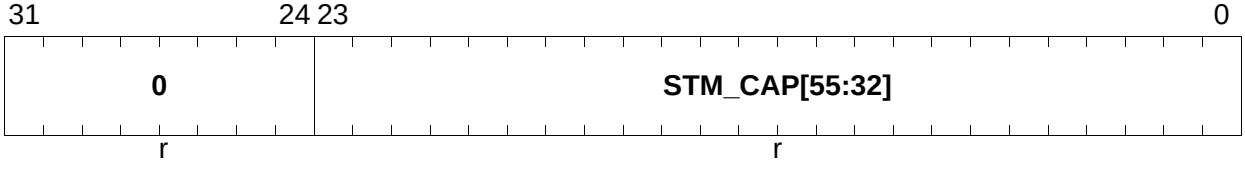
TIM5
Timer Register 5 **Reset Value: 0000 0000$_H$**



TIM6
Timer Register 6 **Reset Value: 0000 0000$_H$**



CAP
Timer Capture Register **Reset Value: 0000 0000$_H$**



Note: CAP captures the system timer bits [55:32] when a read of TIM0 (contains the system timer bits [31:0]) is performed in order to enable software to operate with a coherent value of all the 56 bits of the system timer.

Note: The bits in registers CAP - TIM0 are all read only.

19.4 External Register

The clock control register allows to switch the System Timer on or off. After power-on reset, the System Timer is always enabled and starts counting. The System Timer can be disabled by setting bit DISR to 1.

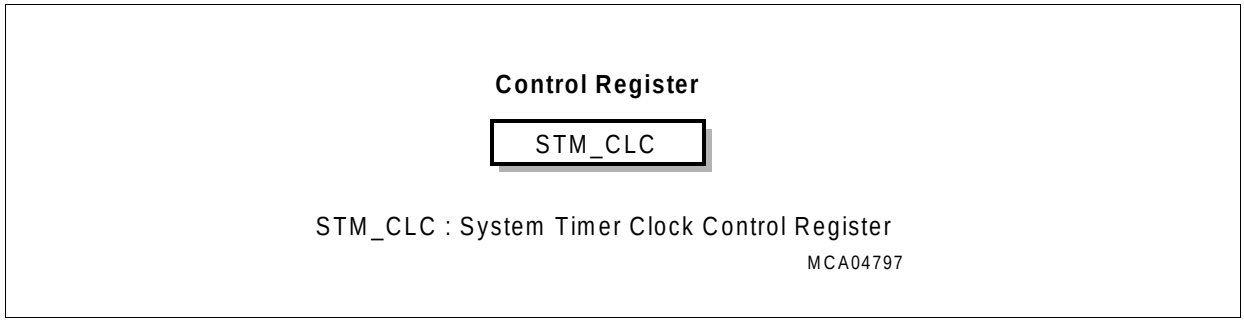


Figure 19-3 STM External Register

STM_CLC
System Timer Clock Control Register **Reset Value: 0000 0000$_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0														DISS	DISR
r														r	rw

Field	Bits	Type	Description
DISR	0	rw	**Module Disable Request Bit** Used for enable/disable control of the module. 0 No disable requested 1 Disable requested
DISS	1	r	**Module Disable Status Bit** Bit indicates the current status of the module 0 Module is enabled 1 Module is disabled
0	[31:2]	r	**Reserved**; read as 0; should be written with 0;

19.5 STM Register Address Ranges

In the TC11IB, the registers of the STM module are located in the following address range:

- Module Base Address: F000 0300_H
  Module End Address: F000 $03FF_H$
- Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 19-2**)

20 Watchdog Timer

This chapter describes the TC11IB Watchdog Timer (WDT). Topics include an overview of the Watchdog Timer function and descriptions of the registers, the password protection scheme, accessing registers, modes, and initialization.

20.1 Watchdog Timer Overview

The Watchdog Timer (WDT) provides a highly reliable and secure way to detect and recover from software or hardware failure. The WDT helps to abort an accidental malfunction of the TC11IB in a user-specified time period. When enabled, the WDT will cause the TC11IB system to be reset if the WDT is not serviced within a user-programmable time period. The CPU must service the WDT within this time interval to prevent the WDT from causing a TC11IB system reset. Hence, routine service of the WDT confirms that the system is functioning properly.

In addition to this standard “Watchdog” function, the WDT incorporates the EndInit feature and monitors its modifications. A system-wide line is connected to the ENDINIT bit implemented in a WDT control register, serving as an additional write-protection for critical registers (besides Supervisor Mode protection). Registers protected via this line can only be modified when Supervisor Mode is active and bit ENDINIT = 0.

Because servicing the Watchdog and modifications of the ENDINIT bit are critical functions that must not be allowed in case of a system malfunction, a sophisticated scheme is implemented which requires a password and guard bits during accesses to the WDT control register. Any write access that does not deliver the correct password or the correct value for the guard bits is regarded as a malfunction of the system, and a Watchdog reset is triggered. In addition, even after a valid access has been performed and the ENDINIT bit has been cleared to provide access to the critical registers, the Watchdog imposes a time-limit for this access window. If ENDINIT has not been properly set again before this limit expires, the system is assumed to malfunction, and a Watchdog reset is triggered. These stringent requirements, although not a guarantee, nevertheless provide a high degree of assurance of the robustness of system operation.

A further enhancement in the TC11IB’s Watchdog Timer is its reset prewarning operation. Instead of immediately resetting the device on the detection of an error, as known from standard Watchdogs, the WDT first issues an Non-maskable Interrupt (NMI) to the CPU before finally resetting the device at a specified time period later. This gives the CPU a chance to save system state to memory for later examination of the cause of the malfunction, an important aid in debugging.

20.2 Features of the Watchdog Timer

The major features of the WDT are summarized here. The Watchdog Timer is implemented in the System Control Unit (SCU) module of the TC11IB. **Figure 20-1** gives an overview of its interface signals.

- 16-bit Watchdog counter.
- Selectable input frequency: $f_{SYSCLK}/256$ or $f_{SYSCLK}/16384$ (f_{SYSCLK} = 48 MHz).
- 16-bit user-definable reload value for normal Watchdog operation, fixed reload value for Time-Out and Prewarning Modes.
- Incorporation of the ENDINIT bit and monitoring of its modifications.
- Sophisticated password access mechanism with fixed and user-definable password fields.
- Proper access always requires two write accesses. The time between the two accesses is monitored by the WDT and limited.
- Access Error Detection: Invalid password (during first access) or invalid guard bits (during second access) trigger the Watchdog reset generation.
- Overflow Error Detection: An overflow of the counter triggers the Watchdog reset generation.
- Watchdog function can be disabled; access protection and ENDINIT monitor function remain enabled.
- Double Reset Detection: If a Watchdog induced reset occurs twice without a proper access to its control register in between, a severe system malfunction is assumed and the TC11IB is held in reset until a power-on reset. This prevents the device from being periodically reset if, for instance, connection to the external memory has been lost such that even system initialization could not be performed.
- Important debugging support is provided through the reset prewarning operation by first issuing an NMI to the CPU before finally resetting the device after a certain period of time.

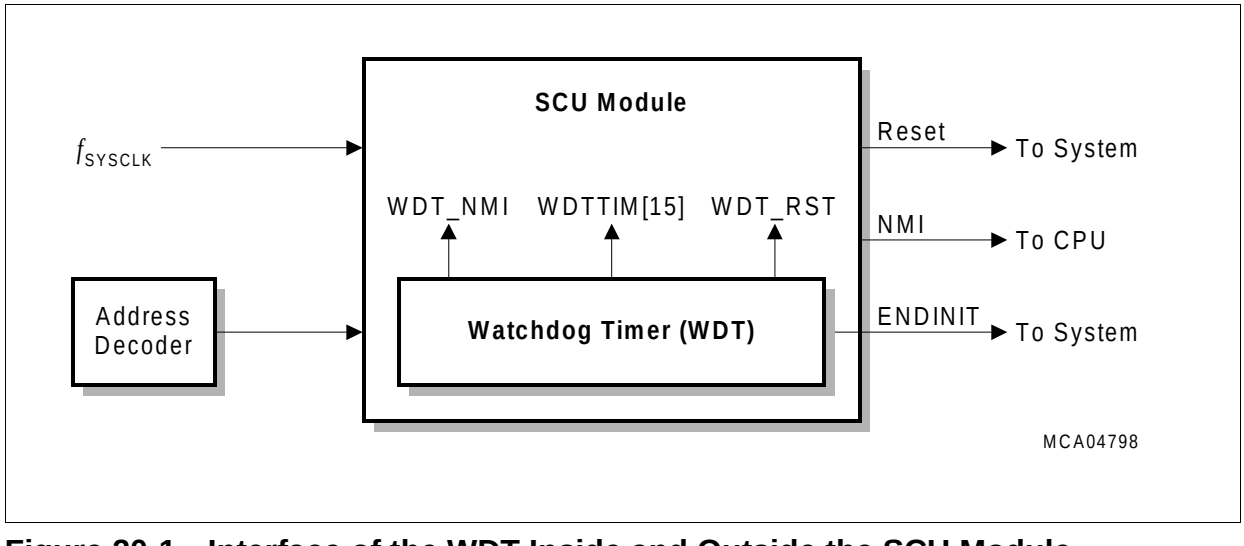


Figure 20-1 Interface of the WDT Inside and Outside the SCU Module

20.3 The EndInit Function

Because understanding of the ENDINIT bit and its function is an important prerequisite for the descriptions in the following sections, its function is explained first.

There are a number of registers in the TC11IB that are usually programmed only once during the initialization sequence of the application. Modification of such registers during normal application run can have a severe impact on the overall operation of modules or the entire system.

While the Supervisor Mode, which allows writes to registers only when it is active, provides a certain level of protection against unintentional modifications, this might not provide enough security for system critical registers.

The TC11IB provides one more level of protection for such registers via the EndInit feature. This is a highly secure write protection scheme that makes unintentional modifications of registers protected by this feature nearly impossible.

The EndInit feature consists of an ENDINIT bit incorporated in the Watchdog Timer control register, WDT_CON0. A system-wide line is connected to this bit. Registers protected via EndInit use the state of this line to determine whether or not writes are enabled. Writes are only enabled if ENDINIT = 0 and Supervisor Mode is active. Write attempts if this condition is not true will cause a bus error, the register contents will not be modified in this case.

An additional line, controlled through a separate bit, to protect against unintentional writes does provide an extra level of security. However, to get the highest level of security, this bit is incorporated in the highly secure access protection scheme implemented in the Watchdog Timer. This is a complex procedure, that makes it nearly impossible for the ENDINIT bit to be modified unintentionally. It is explained in the following sections. In addition, the WDT monitors ENDINIT modifications by starting a time-out sequence each time software opens access to the critical registers through clearing ENDINIT to 0. If the Time-out period ends before ENDINIT is set to 1 again, a malfunction of the software and/or the hardware is assumed and the device is reset.

The access protection scheme and the EndInit time-out operation of the WDT is described in the following sections. **Table 20-1** lists the registers that are protected via the EndInit feature in the TC11IB.

Table 20-1 TC11IB Registers Protected via the EndInit Feature

Normal Mode	Description
mod_CLC	All clock control registers of the individual peripheral modules are EndInit-protected.
BTV, BIV, ISP	Trap and interrupt vector table pointer as well as the interrupt stack pointer are EndInit-protected.
WDT_CON1	The Watchdog Timer Control Register 1, which controls the disabling and the input frequency of the Watchdog Timer, is EndInit-protected. In addition, its bits will only have an effect on the WDT when ENDINIT is properly set to 1 again.

20.4 Watchdog Timer Operation

The following sections describe the registers, the operation, and different modes of the WDT, as well as the password access mechanism. **Figure 20-2** gives an example for the operation of the Watchdog Timer. A rough description of the sequence of events in this figure is provided here. Refer to the following sections for a detailed explanation.

1. Time-Out Mode is automatically entered after reset. Timer counts with slowest input clock.
2. Time-Out Mode terminated and Normal Mode is entered by setting ENDINIT to 1.
3. Normal Mode is terminated and Time-Out Mode is entered through a password access to WDT_CON0. The reload value was set to REL_1.
4. Time-Out Mode is terminated and Normal Mode entered again by setting ENDINIT to 1. The reload value WDTREL has been changed to REL_2 and the timer input clock was set to the fast clock.
   Events 3) and 4) constitute a Watchdog Timer service sequence.
5. The Watchdog Timer was not serviced and continued to count until overflow. Reset Prewarning Mode is entered. Timer counts with selected fast input clock. Watchdog operation cannot be altered or stopped in this mode.
6. Timer continued to count until overflow, generating a Watchdog Timer reset.
7. Time-Out Mode is automatically entered after reset. Timer counts with slowest input clock.
8. Time-Out Mode is terminated and Normal Mode is entered again.

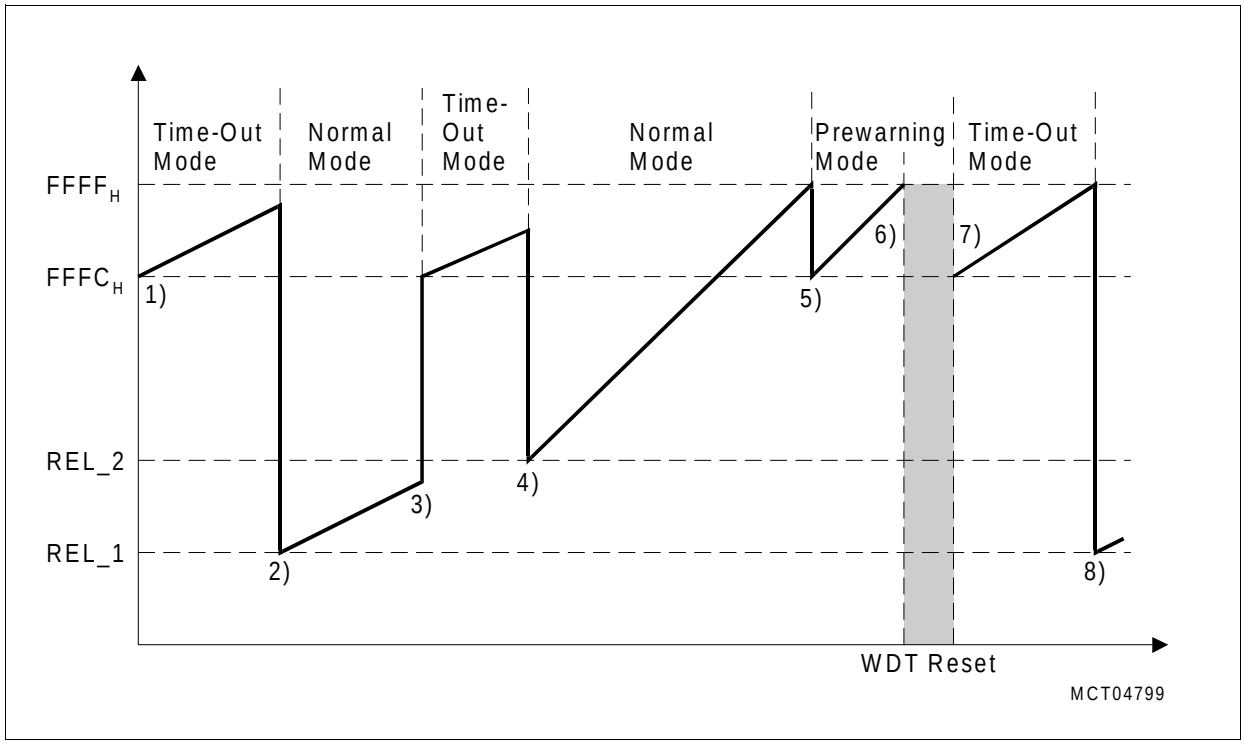


Figure 20-2 Example for an Operation Sequence of the Watchdog Timer

20.4.1 WDT Register Overview

Two control registers, WDT_CON0 and WDT_CON1, and one status register, WDT_SR, serve for communication of the software with the WDT. This section provides a short overview and describes the access mechanisms of the WDT registers. Detailed layout and bit descriptions of the registers are given in **Section 20.6**.

Register WDT_CON0 holds the ENDINIT bit, a register lock status bit (WDTLCK), an 8-bit user-definable password field (WDTPW), and the user-definable reload (start) value (WDTREL) for the Watchdog Timer in Normal Mode.

Register WDT_CON1 contains two bits. Bit WDTIR is a request bit for the Watchdog Timer input frequency selection, while bit WDTDR is a request bit for the Disable Mode of the WDT. These two bits are only request bits in that they do not actually control the input frequency and disabling of the WDT. They can be modified only when the ENDINIT bit is 0, but they will have an effect only when ENDINIT is properly set to 1 again.

The status register WDT_SR holds information about the current conditions of the WDT. It contains the current timer count value (WDTTIM), three bits indicating the mode of operation (WDTTO for Time-Out Mode, WDTPR for Prewarning Mode, and WDTDS for Disable Mode), and the error indication bits for timer overflow (WDTOE) and access error (WDTAE).

While WDT_SR is a read-only register, the control registers can be read and written. Reading these registers is always possible; a write access, however, must follow certain

protocols. Register WDT_CON1 is Supervisor Mode and EndInit-protected, thus, Supervisor Mode must be active and bit ENDINIT must be 0 for a successful write to this register. If one or both conditions are not met, a bus error will be generated, and the bits in WDT_CON1 will be not modified.

Register WDT_CON0 requires a much more complex write procedure as it has a special write protection mechanism. Proper access to WDT_CON0 always requires two write accesses in order to modify its contents. The first write access requires a password to be written to the register to unlock it. This access is called *Password Access*. Then, the second access can modify the register's contents. It is called *Modify Access*. When the modify access completes, WDT_CON0 is locked again automatically. (Even if no parameters are changed in the second write access, it is still called a *modify access*.) If the *Modify Access* sets WDT_CON0.ENDINIT = 0, then other protected system registers, such as WDT_CON1, are unlocked and can be modified.

Note: WDT_CON0 is automatically re-locked after a modify access, so a new password access must be performed to modify it again. Note further that the WDT switches to Time-Out Mode as a side-effect of a successful password access, so that protected registers can remain unlocked at most for the duration of one Time-out Period. Otherwise, the system will be forced to reset.

20.4.2 Modes of the Watchdog Timer

The Watchdog Timer can operate in one of four different modes:

- Time-Out Mode
- Normal Mode
- Disable Mode
- Prewarning Mode

The following description provides a short overview of these modes and how the WDT changes from one mode to the other. As well as these major operating modes, the WDT has special behavior during power-saving and OCDS suspend modes. Detailed discussions of each of the modes can be found in **Section 20.4.6**.

Figure 20-3 provides a state diagram of the different modes of the WDT and the transition possibilities. Please refer to the description for the conditions for changing from one state to the other.

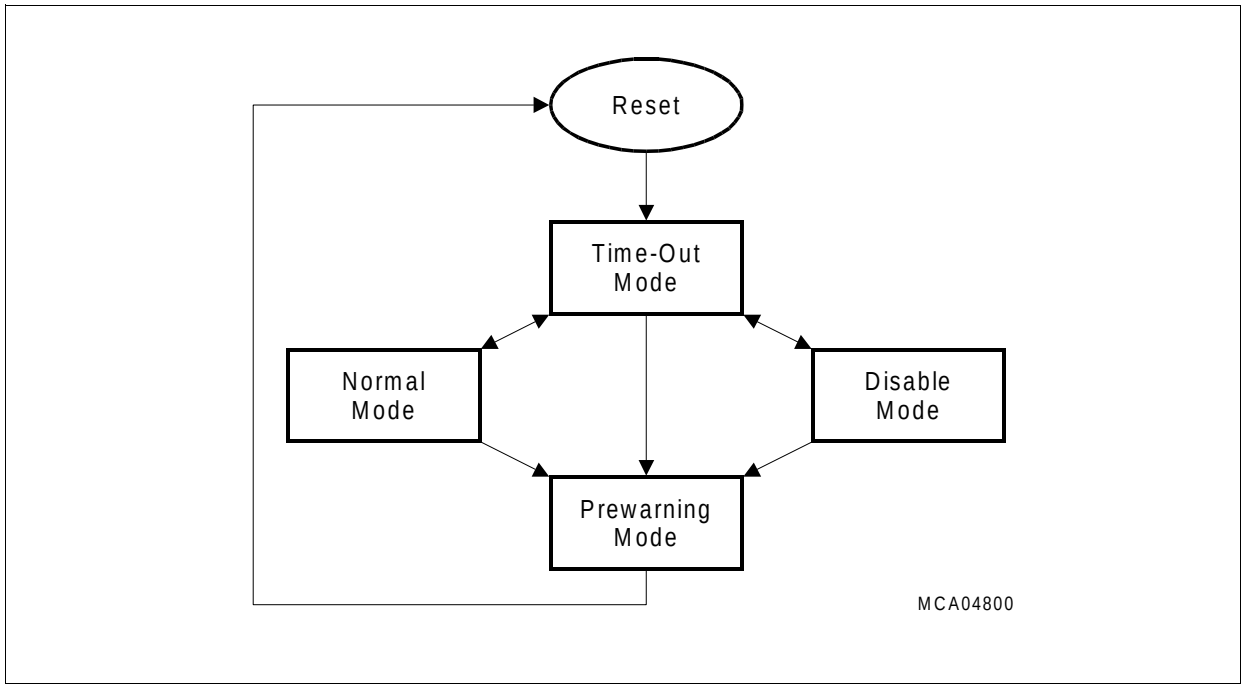


Figure 20-3 State Diagram of the Modes of the WDT

20.4.2.1 Time-Out Mode

The Time-Out Mode is the default mode after a reset. It is also always entered when a valid password access to register WDT_CON0 is performed (see **Section 20.4.3**). The timer is set to a predefined value and starts counting upwards. Time-Out Mode can only be exited properly by setting ENDINIT to one with a correct access sequence. If an improper access to the WDT is performed, or if the timer overflows before ENDINIT is set to 1, a Watchdog Timer NMI request (WDT_NMI) is requested, and Prewarning Mode is entered. A reset of the TC11IB is imminent and can no longer be stopped.

A proper exit from Time-Out Mode can either be to the Normal or the Disable Mode, depending on the state of the disable request bit, WDTDR, in register WDT_CON1.

20.4.2.2 Normal Mode

In Normal Mode (WDTDR = 0), the WDT operates in a standard Watchdog fashion. The timer is set to a user-defined start value, and begins counting up. It has to be serviced before the counter overflows. Servicing is performed through a proper access sequence to the WDT control register WDT_CON0. This reloads the timer with the start value, and normal operation continues.

If the WDT is not serviced before the timer overflows, or if an invalid access to the WDT is performed, a system malfunction is assumed. Normal Mode is terminated, a Watchdog Timer NMI request (WDT_NMI) is requested, and Prewarning Mode is entered. A reset of the TC11IB is imminent and can no longer be stopped.

Because servicing the WDT is an access sequence, first requiring a valid password access to register WDT_CON0, the WDT will enter Time-Out Mode until the second proper access is performed.

20.4.2.3 Disable Mode

Disable Mode is provided for applications which truly do not require the Watchdog Timer function. It can be entered from Time-Out Mode when the disable request bit WDTDR is set to 1. The timer is stopped in this mode. However, disabling the WDT does only stop it from performing the standard Watchdog function (Normal Mode), eliminating the need for timely service of the WDT. It does not disable Time-Out and Prewarning Mode. If an access to register WDT_CON0 is performed in Disable Mode, Time-Out Mode is entered if the access was valid, and Prewarning Mode is entered if the access was invalid. Thus, the ENDINIT monitor function as well as (a part of) the system malfunction detection will still be active.

20.4.2.4 Prewarning Mode

Prewarning Mode is entered always when a Watchdog error is detected. This can be an overflow of the timer in Normal or Time-Out Mode, or an invalid access to register WDT_CON0. Instead of immediately generating a reset of the device, as known from other Watchdog timers, the TC11IB Watchdog Timer provides the system with a chance to save important state information before the reset occurs. This is done through first activating an NMI trap request to the CPU, warning it about the coming reset (reset prewarning). If the CPU is still able to do so (depending on the type and severity of the detected malfunction), it can react on the Watchdog NMI request and can save important system state to memory. This saved system state can then be examined during debugging to determine the cause of the malfunction. If the part would be immediately reset on the detection of a Watchdog error, this debugging information would never be available, and investigating the cause of the malfunction would be a very difficult task.

In Prewarning mode, after having generated the NMI request, the WDT counts for a specified period of time, and then generates a Watchdog reset for the device. This reset generation cannot be avoided in this mode; the WDT does not react anymore to accesses to its registers, nor will it change its state. This is to prevent a malfunction from falsely terminating this mode, disabling the reset, and letting the device to continue to function improperly.

Note: In Prewarning Mode, it is not required for the part waits for the end of this mode and the reset. After having saved required state in the NMI routine, software can execute a soft reset to shorten the time. However, the state of the Watchdog Status Register should also be saved in this case, because the error flags contained in it will be cleared due to the soft reset (this is not the case if the Watchdog reset is awaited).

20.4.3 Password Access to WDT_CON0

A correct password must be written to register WDT_CON0 in order to unlock it for modifications. Software must either know the correct password in advance or it can compute it at runtime. The password required to unlock the register is formed by a combination of bits in registers WDT_CON0 and WDT_CON1, plus a number of guard bits. **Table 20-2** summarizes the requirements for the password.

Table 20-2 Password Access Bit Pattern Requirements

Bit Position	Required Value
0	Current state of the ENDINIT bit, WDT_CON0.ENDINIT
1	Fixed; must be written with 0.
2	Current state of the input frequency request bit, WDT_CON1.WDTIR
3	Current state of the input frequency request bit, WDT_CON1.WDTDR
[7:4]	Fixed; must be written to 1111_B
[15:8]	Current value of user-definable password field, WDT_CON0.WDTPW
[31:16]	Current value of user-definable reload value, WDT_CON0.WDTREL

When reading register WDT_CON0, bit positions [7:4] always return 0s. As can be seen from **Table 20-2**, the password is designed such that it is not possible to just read the contents of a register and use this as the password. The password is never identical to the contents of WDT_CON0 or WDT_CON1, it is always required to modify the read value (at least bits 1 and [7:4]) to get the correct password. This prevents a malfunction from accidentally reading a WDT register's contents and writing it to WDT_CON0 as an unlocking password.

If the password matches the requirements, WDT_CON0 will be unlocked as soon as the password access has finished. The unlocked condition will be indicated by WDT_CON0.WDTLCK = 0.

If WDT_CON0 is successfully unlocked, a subsequent write access can modify it, as described in **Section 20.4.4**.

If an improper password value is written to WDT_CON0 during the password access, a Watchdog Access Error condition exists. Bit WDTAE is set and the Prewarning Mode is entered.

The user-definable password, WDTPW, provides additional options for adjusting the password requirements to the application's needs. It can be used, for instance, to detect unexpected software loops or to monitor the execution sequence of routines. See **Section 20.5.4**.

20.4.4 Modify Access to WDT_CON0

If WDT_CON0 is successfully unlocked as described in **Section 20.4.3**, the following write access to WDT_CON0 can modify it. However, also this access must follow certain requirements in order to be accepted and regarded as valid. **Table 20-3** lists the required bit patterns. If the access does not follow these rules, a Watchdog Access Error condition is detected, bit WDTAE is set and the Prewarning Mode is entered.

Table 20-3 Modify Access Bit Pattern Requirements

Bit Position	Value
0	User definable; desired value for the ENDINIT bit, WDT_CON0.ENDINIT.
1	Fixed; must be written with 1.
2	Fixed; must be written with 0.
3	Fixed; must be written with 0.
[7:4]	Fixed; must be written with $1111_{B.}$
[15:8]	User-definable; desired value of user-definable password field, WDT_CON0.WDTPW.
[31:16]	User-definable; desired value of user-definable reload value, WDT_CON0.WDTREL.

After the modify access has completed, WDT_CON0.WDTLCK is set to 1 again by hardware, automatically re-locking WDT_CON0. Before the register can be modified again, a valid password access must be executed again.

20.4.5 Term Definitions for WDT_CON0 Accesses

To simplify the descriptions in the following sections, a number of terms are defined to indicate the type of access to register WDT_CON0:

Watchdog Access Sequence: Two accesses to register WDT_CON0 consisting of first a *Password Access* followed by a *Modify Access*. The two accesses do not have to be adjacent accesses, any number of accesses to other addresses can be between these accesses unless the *Time-out Period* is not exceeded.

Password Access: The first access of a *Watchdog Access Sequence* to register WDT_CON0 intended to open WDT_CON0 for modifications. This access needs to write a defined password value to WDT_CON0 in order to successfully open WDT_CON0.

Valid Password Access: A *Password Access* with the correct password value. A Valid Password Access opens register WDT_CON0 for one, and only one, *Modify Access*. Bit WDTLCK is set to 0 after this access. The Watchdog Timer is placed into the Time-Out Mode after a Valid Password Access in Normal Mode or Disabled Mode.

Modify Access: The second access of an *Watchdog Access Sequence* to register WDT_CON0 intended to modify parameters in WDT_CON0. The parameters that can be modified are WDTREL, WDTPW and ENDINIT. Special guard bits in WDT_CON0 must be written with predefined values in order for this access to be accepted.

Valid Modify Access: A *Modify Access* with the correct guard bit values. The values written to WDTREL, WDTPW, and ENDINIT are in effect after completion of this access. Bit WDTLCK is automatically set to 1 after this access. Register WDT_CON0 is locked until it is re-opened with a *Valid Password Access* again.

20.4.6 Detailed Descriptions of the WDT Modes

The following subsections provide detailed descriptions of each of the modes of the WDT. The entry conditions and actions, operation in this mode, as well as exit conditions and the succeeding mode are listed for each mode.

20.4.6.1 Time-Out Mode Details

Time-Out Mode is the default after reset, and is entered each time a Valid Password Access to register WDT_CON0 is performed.

Table 20-4 WDT Time-Out Mode

State / Action	Description
Entry	– Automatically after any reset. – If a valid password was written to WDT_CON0 in Normal or Disable Mode
Actions on Entry	– WDTTIM is set to $FFFC_H$; WDTTO is set to 1; WDTDS is set to 0. – ENDINIT = 0 if mode entered through reset; otherwise, it retains its previous value. – Bits WDTAE and WDTOE depend on their state before the reset if the reset was caused by the Watchdog. For any other reset (POR, HRST, SRST, PWDRST), they are 0. – WDTIS retains its previous value. – After reset, EndInit is 0. Thus, access to EndInit-protected registers is enabled. If Time-Out Mode was entered through other reasons, ENDINIT might or might not be 0.

Table 20-4 WDT Time-Out Mode

State / Action	Description
Operation	– Timer starts counting up from $FFFC_H$; increments with clock rate determined through WDTIS (0 after reset, slowest clock). – Access to registers WDT_CON0 is possible. Access to register WDT_CON1 is possible if ENDINIT = 0. – Restarting Time-Out Mode is not possible: A valid password access in this mode does not invoke another Time-out sequence (it does not reload the timer, etc.). A modify access to WDT_CON0 writing a 0 to ENDINIT does not terminate Time-Out Mode. – It is not possible to change the reload value or frequency in Time-Out Mode, as this would require setting EndInit to 1, which terminates Time-Out Mode. Reload value is not used until Normal mode is entered.
Exit	a) Writing ENDINIT to 1 with a valid Modify Access (a Valid Password Access must have been executed first). b) Timer WDTTIM overflows from $FFFF_H$ to 0000_H. c) An invalid access to WDT_CON0 (either during the password or the modify access)
Next Mode	Depending on the Exit condition: a1) If WDTDR = 0 (no disable request), the WDT enters the Normal Mode. a2) If WDTDR = 1 (disable request), the WDT enters the Disable Mode. b) Bit WDTOE is set to 1, and the WDT enters the Prewarning Mode. c) Bit WDTAE is set to 1, and the WDT enters the Prewarning Mode

20.4.6.2 Normal Mode Details

Normal Mode can be entered from Time-Out Mode only if bit WDT_CON1.WDTDR is set to 0 before proper termination of Time-Out Mode. The WDT operates as a standard Watchdog in this mode, requiring timely service to prevent a timer overflow.

Table 20-5 WDT Normal Mode

State / Action	Description
Entry	– Only from Time-Out Mode by writing ENDINIT to 1 with a Valid Modify Access (a Valid Password Access must have been executed first), while bit WDTDR = 0.
Actions on Entry	– WDTTIM is loaded with the value of WDTREL. – Bits WDTAE, WDTOE, WDTPR, WDTTO, and WDTDS are cleared to 0.

Table 20-5 WDT Normal Mode

State / Action	Description
Operation	– WDTTIM starts counting up from reload value with frequency selected through WDTIS.
Exit	a) A valid password access to register WDTCON. b) Timer WDTTIM overflows from $FFFF_H$ to 0000_H. c) An invalid access to WDT_CON0 (either during the password or the modify access)
Next Mode	Depending on Exit condition: a) Time-Out Mode. b) Prewarning Mode, bit WDTOE is set to 1 (overflow error). c) Prewarning Mode, bit WDTAE is set to 1 (access error).

20.4.6.3 Disable Mode Details

Disable Mode is provided for applications which truly do not require the Watchdog Timer function. It can only be entered from Time-Out Mode if bit WDT_CON1.WDTDR is set to 1 before proper termination of Time-Out Mode. The counter stops in this mode, eliminating the need for a WDT service. However, if an access to register WDT_CON0 is performed, the WDT will leave Disable Mode. Disable Mode does not stop the detection of access errors and the entry of Prewarning Mode nor the entry of Time-Out Mode on a Valid Password Access.

Table 20-6 WDT Disable Mode

State / Action	Description
Entry	– Only from Time-Out Mode by writing ENDINIT to 1 with a Valid Modify Access (a Valid Password Access must have been executed first), while bit WDTDR = 1.
Actions on Entry	– Bits WDTAE, WDTOE, WDTPR, and WDTTO are cleared. Bit WDTDS is set to 1. – Timer WDTTIM is stopped (it retains its current value).
Operation	–
Exit	a) Valid password access to register WDTCON. b) Invalid access to WDT_CON0 (either during the password or the modify access)
Next Mode	Depending on Exit condition: a) Time-Out Mode. b) Prewarning Mode, bit WDTAE is set to 1 (access error).

20.4.6.4 Prewarning Mode Details

Prewarning Mode is always entered immediately after a Watchdog error condition was detected. This can be either an access error to register WDT_CON0 or an overflow of the counter in Normal or Time-Out Mode. This mode indicates that a reset of the device is imminent. Operation of the WDT in this mode can not be altered or stopped, except through a reset.

Table 20-7 WDT Prewarning Mode

State / Action	Description
Entry	Detection of a Watchdog error: – Overflow of timer WDTTIM. – Access error to register WDT_CON0 (either on a password or modify access) in Time-Out, Normal, or Disable modes.
Actions on Entry	– NMIWDT in register NMISR is set (this triggers an NMI request to the CPU). – WDTTIM is set to $FFFC_H$. – WDTPR is set to 1; WDTDS is set to 0; WDTIS retains its value. – WDTTO retains its previous value: if entry into Prewarning Mode was from Time-Out Mode, WDTTO is 1. In all other cases, WDTTO is 0. – Bits WDTAE and WDTOE indicate whether Prewarning Mode was entered due to an access or an overflow error. They have been set accordingly on exit of the previous mode.
Operation	– Timer WDT_TIM starts counting up from $FFFC_H$ with frequency selected through WDTIS. – Register WDT_CON0 can be accessed in this mode as usual. However, the WDT will not change its mode anymore, regardless whether valid or invalid accesses are made to WDT_CON0. For invalid accesses to WDT_CON0 (password or modify access), however, bit WDTAE in WDT_SR will be set. – Register WDT_CON1 can not be written to in Prewarning Mode, even if bit ENDINIT = 0. Write access to WDT_CON1 is totally prohibited. A write attempt will generate a bus error in this mode.
Exit	– Prewarning Mode can not be disabled, prolonged, or terminated (except through a reset). The timer will increment until it overflows from $FFFF_H$ to 0000_H, which then causes a system reset. Bit WDTRST in register RSTSR is set in this case.
Next Mode	Reset

Note: In Prewarning Mode, it is not required that the part waits for the end of the Time-out Period and the reset. After having saved required state in the NMI routine, software can execute a soft reset to shorten the time. However, the state of the Watchdog Status Register should also be saved in this case, since the error flags contained in it will be cleared due to the soft reset (this is not the case if the Watchdog reset is awaited).

20.4.6.5 WDT Operation During Power-Saving Modes

If the CPU is in Idle Mode, Sleep Mode, or Deep Sleep Mode, it cannot service the Watchdog Timer because no software is running. When in Deep Sleep, only an external event can awaken the system. Excluding this case, and the case where the system is running normally, a strategy for managing the WDT is needed while the CPU is in Idle Mode or Sleep Mode. There are two ways to manage the WDT in these cases. First, the Watchdog can be disabled before idling the CPU. This has the disadvantage that the system will no longer be monitored during the idle period.

A better approach to this problem relies upon a wake-up features of the WDT. Whenever the CPU is put in Idle or Sleep Mode and the WDT is not disabled, it causes the CPU to be awakened at regular intervals. The Watchdog Timer triggers an NMI trap request when its count value (WDT_SR.WDTTIM) transitions from $7FFF_H$ to 8000_H, that is, when the most significant bit of the WDT counter changes its state from 0 to 1. The WDT also sets the NMISR.NMIWDT bit at this time to indicate to the CPU that the WDT caused the NMI. The CPU is awakened by the NMI trap, and can then service the Watchdog Timer in the usual manner, reset NMISR.NMIWDT, and then return to its former power-management mode.

This operation does not cause a WDT error condition. The WDT continues to operate in Normal Mode after generating this wake-up NMI. However, if the CPU does not service the WDT in the NMI trap routine, it will continue to run, eventually causing an overflow, which will cause the WDT to enter Prewarning Mode.

Note: Before switching into a non-running power-management mode, software should perform a Watchdog service sequence. With the Modify Access, the Watchdog reload value, WDT_CON0.WDTREL, should be programmed such that the wake-up occurs after a period which best meets application requirements. The maximum period between NMI requests is one-half of the maximum Watchdog Timer period.

20.4.6.6 WDT Operation in OCDS Suspend Mode

When the On-Chip Debugging System (OCDS) is enabled after reset (through the $\overline{\text{OCDSE}}$ pin), the WDT will automatically stop when OCDS Suspend Mode is activated. It will resume operation after the Suspend Mode is deactivated.

20.4.6.7 Double Watchdog Error.

It is possible that severe system malfunctions may not be corrected even by a system reset. If application code cannot be executed properly because of a system fault, then the WDT initialization code itself might not be able to execute to service the WDT, with the result that two WDT-initiated resets might occur back-to-back. A feature of the WDT detects such Double Watchdog Errors and suspends all system operations after the second reset occurs. This feature prevents the TC11IB from executing random wrong code for longer than the Time-out Period, and prevents the TC11IB from being repeatedly reset by the Watchdog Timer.

Double Watchdog Errors are detected with the aid of the error-indication flags WDT_SR.WDTOE and WDT_SR.WDTAE. Ordinarily, software clears these bits to 0 during normal WDT service. But, these bits are not cleared when a reset is caused by the WDT. Because the error bits are preserved across resets, the WDT can examine them if it times out again. If either error bit is still set when a new Watchdog Timer Error occurs, then there must have been a preceding WDT-initiated reset without intervening software service of the WDT. Hence, this is a Double Watchdog Error condition. In this case the WDT will generate another reset after the termination of the Prewarning Mode, but this time the TC11IB will be held in the reset state until a power-up reset is generated by external hardware.

20.4.7 Determining WDT Periods

The WDT uses the same clock, f_{SYSTEM}, as the System Control Unit (SCU) in which it is integrated. In the TC11IB, this clock is equal to the system clock, f_{SYSCLK} = 48 MHz. A clock divider in front of the Watchdog Timer provides two output frequencies, f_{SYSCLK}/256 and f_{SYSCLK}/16384. Bit WDTIS selects between these options.

When the WDT is in Normal Mode, the duration of a WDT cycle is defined as a Normal Period, as described in **Section 20.4.7.2**.

When the WDT is in Time-Out Mode or Prewarning Mode, the duration of a WDT cycle is defined as a Time-out Period, as described in **Section 20.4.7.1**.

The general form to calculate a Watchdog period is:

$$period = \frac{(2^{16} - startvalue) \times 256 \times 2^{(1 - WDTIS) \times 6}}{f_{SYSCLK}} \qquad [20\text{-}1]$$

The parameter *startvalue* represents the fixed value $FFFC_H$ for the calculation of the Time-out Period, and the user-programmable reload value WDTREL for the calculation of the Normal Period. Note that the exponent (1 - WDTIS) $\times$ 6 results to 0 if WDTIS is 1, and to 6 if WDTIS is 0. This results in the value 256 being multiplied by either 1 (2^0 = 1) or by 64 (2^6), giving the two divider factors 256 and 16384.

Note: Because there is no synchronization of the clock divider to the mode transitions of the Watchdog, the next clock pulse, incrementing the counter, may come after one clock divider period, or immediately after the counter was reloaded. Thus, it is recommended that the reload value is programmed to a value which results in one clock pulse more than the required period.

20.4.7.1 Time-out Period

The duration of Time-Out Mode and Prewarning Mode is determined by the Time-out Period described here. The Time-out Period that occurs immediately after reset is governed entirely by system defaults, as no software is been able to run at this point; it is described separately below.

Time-out Period After Reset

After reset, the initial count value for the timer is fixed at $FFFC_H$ when the WDT clock starts running. The WDT counts up at a rate determined by WDT_SR.WDTIS, which is 0 after any reset ($f_{SYSCLK}/16384$). Counting up from $FFFC_H$, it takes four clocks for the counter to overflow, so the Time-out Period defaults to a period of $4 \times 16384/f_{SYSCLK} = 65536/f_{SYSCLK}$. This establishes the real-time deadline for software to initialize the Watchdog and critical system registers, and to then set ENDINIT. For example, the Time-out Period after reset would correspond to 1.36 ms @ 48 MHz system frequency.

Changing the input frequency selection via WDT_CON1.WDTIR during this initial Time-out Period has no immediate effect, because frequency selection is actually determined by WDT_SR.WDTIS, but WDT_CON1.WDTIR is only copied into WDT_SR.WDTIS after WDT_CON0.ENDINIT has been set to 1, that is, after Time-Out Mode has been properly exited. Hence, the new input frequency will become effective only in a subsequent Time-out Period.

Time-out Period During Normal Operation

As after reset, the WDT counter is initially set to $FFFC_H$ when Time-Out Mode is entered, and Time-Out Mode expires when the counter overflows. However, there are two differences to the Time-out Period after reset. First, the input frequency can be either $f_{SYSCLK}/256$ or $f_{SYSCLK}/16384$, depending on the programmed state of bit WDT_SR.WDTIS before the Time-out Period was entered. Second, because there is no synchronization of the clock divider to the mode transitions of the Watchdog, the next clock pulse, incrementing the counter to $FFFD_H$, may come after one clock divider period, or immediately after the counter was initially set to $FFFC_H$. Thus, the minimum duration of the Time-out Period in the latter case will only be three counter clocks. The possible minimum and maximum periods are given in **Table 20-8**.

Table 20-8 Time-out Period During Normal Operation

WDTIS	Min/ Max	Period	Example @ f_{SYSCLK} = 48 MHz
0	min.	$3 \times 16384/f_{SYSCLK} = 49152/f_{SYSCLK}$	1.02 ms
	max.	$4 \times 16384/f_{SYSCLK} = 65536/f_{SYSCLK}$	1.36 ms
1	min.	$3 \times 256/f_{SYSCLK} = 768/f_{SYSCLK}$	16 µs
	max.	$4 \times 256/f_{SYSCLK} = 1024/f_{SYSCLK}$	21 µs

The WDT input clock rate can not be changed during the Time-out Period. The control bit for the input clock rate, WDT_SR.WDTIS, is loaded from WDT_CON1.WDTIR when WDT_CON0.ENDINIT is set to 1, that is, after Time-Out Mode has been properly exited. Hence, the new input frequency will become effective only in the subsequent Time-out Period.

Note: In Prewarning Mode, it is not required that the part waits for the end of the Time-out Period and the reset. After having saved required state in the NMI routine, software can execute a soft reset to shorten the time. However, the state of the Watchdog Status Register should also be saved in this case, since the error flags contained in it will be cleared due to the soft reset (this is not the case if the Watchdog reset is awaited).

20.4.7.2 Normal Period

The duration of Normal Mode can be varied by two parameters: the input clock and the reload value.

The system clock, f_{SYSCLK}, can be divided by either 256 or 16384. WDT_SR.WDTIS selects the input clock divider. The default value of WDTIS after reset is 0, corresponding to a frequency of $f_{SYSCLK}/16384$.

When the Watchdog Timer is serviced in Normal Mode, it is reloaded with the 16-bit reload value, WDT_CON0.WDTREL.

The Watchdog Timer Period can be varied over a wide range with these two parameters. Again, since there is no synchronization of the clock divider to the mode transitions of the Watchdog, the next clock pulse, incrementing the counter, may come after one clock divider period, or immediately after the counter was reloaded. Thus, it is recommended that the reload value is programmed to a value which results to one clock pulse more than the required period. Using a reload value of $FFFF_H$ could therefore lead to an immediate overflow of the timer. Thus, the examples given in **Table 20-9** are only shown with a maximum reload value of $FFFE_H$.

Table 20-9 Timer Periods in Normal Mode

WDTIS	Reload Value	Min/ Max	Period	Example @ f_{SYSCLK} = 48 MHz
0	0000_H	min.	$65535 \times 16384/f_{SYSCLK} = 1073725440/f_{SYSCLK}$	22.4 s
		max.	$65536 \times 16384/f_{SYSCLK} = 1073741824/f_{SYSCLK}$	22.4 s
	$FFFE_H$	min.	$1 \times 16384/f_{SYSCLK} = 16384/f_{SYSCLK}$	341.3 µs
		max.	$2 \times 16384/f_{SYSCLK} = 32768/f_{SYSCLK}$	682.7 µs
1	0000_H	min.	$65535 \times 256/f_{SYSCLK} = 16776960/f_{SYSCLK}$	349.5 ms
		max.	$65536 \times 256/f_{SYSCLK} = 16777216/f_{SYSCLK}$	349.5 ms
	$FFFE_H$	min.	$1 \times 256/f_{SYSCLK} = 256/f_{SYSCLK}$	5.3 µs
		max.	$2 \times 256/f_{SYSCLK} = 512/f_{SYSCLK}$	10.7 µs

20.4.7.3 WDT Period During Power-Saving Modes

Care needs to be taken when programming the WDT reload value before going to Idle or Sleep Mode. As described in **Section 20.4.6.5**, the state of bit 15 of the Watchdog counter is used to wake up from these modes through a Watchdog NMI request. Thus, the reload value should be chosen such that it is less then $7FFE_H$ (bit 15 = 0), otherwise an immediate wake-up could occur. Only half of the maximum periods shown in **Table 20-9** can be used for the wake-up period.

20.5 Handling the Watchdog Timer

This section describes methods of handling the Watchdog Timer function.

20.5.1 System Initialization

After any reset, the Watchdog Timer is put in Time-Out Mode, and WDT_CON0.ENDINIT is 0, providing access to sensitive system registers. Changes to the operation of the Watchdog Timer controlled by WDT_CON1 become effective only after WDT_CON0.ENDINIT has been set to 1 again. Thus, changes to the WDT mode bits in WDT_CON1 do not interfere with the Time-out operation of the Watchdog Timer after reset. **Table 20-10** shows the default contents of the Watchdog Timer registers.

Table 20-10 Watchdog Timer Default Values After Reset

Register	Default Contents	Description
WDT_CON0	FFFC 0002_H	Reload value is $FFFC_H$, WDTPW is 0; WDT_CON0 is locked (WDTLCK = 1); ENDINIT is 0.
WDT_CON1	$0000\ 0000_H$	Watchdog Timer disable request is 0; input clock request set to $f_{SYSCLK}/16384$.
WDT_SR	FFFC $001U_H$	The Watchdog counter contains $FFFC_H$ (the initial Time-out value); WDT is operating in Time-Out Mode (WDTTO = 1); WDT is enabled (WDTDS = 0); input clock is $f_{SYSCLK}/16384$. Bits WDTOE and WDTAE are set to 0 after a power-on, a hard or a soft reset. In case of a reset caused by the WDT, these two bits are set depending on the error condition that caused the Watchdog reset.

Because the Watchdog Timer is in Time-Out Mode after reset, WDT_CON0.ENDINIT must be set to 1 before the Time-out Period expires. This means that initialization of ENDINIT-protected system registers must be complete before the expiration of the Time-out Period, defined in **Section 20.4.7.1**. To set WDT_CON0.ENDINIT to 1, a Valid Password Access to WDT_CON0 must be performed first. During the subsequent Valid Modify Access, WDT_CON0.ENDINIT must be set to 1, which will exit Time-Out Mode. The Watchdog Timer is switched to the operation determined by the new values of WDTIS and WDTDS.

Note: The action described above must absolutely be performed during initialization of the device to properly terminate this mode. Even if the Watchdog function will not be used in an application and the WDT will be disabled, a valid access sequence to the WDT is mandatory. Otherwise, the Watchdog counter will overflow, Prewarning Mode will be entered, and a Watchdog reset will occur at the end of the Time-out Period.

Bit fields WDT_CON0.WDTREL and WDT_CON0.WDTPW can optionally be changed during the Valid Modify Access, but it is not required. WDT_CON0.ENDINIT can be set to 1 or 0, however, setting ENDINIT to 0 does not stop Time-Out Mode. Any values written to WDTREL, WDTPW, and ENDINIT are stored in WDT_CON0, and WDT_CON0 is automatically locked (WDTLCK = 1) after the modify access is finished.

20.5.2 Re-opening Access to Critical System Registers

If some or all of the system's ENDINIT-protected registers must be changed during run time of an application, access can be re-opened. To do this, WDT_CON0 must first be unlocked with a Valid Password Access. In the following Valid Modify Access, ENDINIT

can be set to 0. Access to ENDINIT-protected registers is now open again. However, when WDT_CON0 is unlocked, the WDT is automatically switched to Time-Out Mode. Thus, the access window is time-limited. Time-Out Mode is only terminated after ENDINIT has been set to 1 again, requiring another Valid Password and Valid Modify Access to WDT_CON0.

If the WDT is not used in an application and is therefore disabled (WDT_SR.WDTDS = 1), the above described case is the only occasion when WDT_CON0 must be accessed again after the system is initialized. If there are no further changes to critical system registers needed, no further accesses to WDT_CON0, WDT_CON1, or WDT_SR are necessary. However, it is always recommended that the Watchdog Timer be used in an application for safety reasons.

20.5.3 Servicing the Watchdog Timer

If the Watchdog Timer is used in an application and is enabled (WDT_SR.WDTDS = 0), it must be regularly serviced to prevent it from overflowing.

Service is performed in two steps. a Valid Password Access followed by a Valid Modify Access. The Valid Password Access to WDT_CON0 automatically switches the WDT to Time-Out Mode. Thus, the modify access must be performed before the Time-out expires or a system reset will result.

During the following modify access, the strict requirement is that WDT_CON0.ENDINIT as well as bit 1 and bits [7:4] are written with 1’s, while bits [3:2] are written with 0’s.

Note: ENDINIT must be written with 1 even if it is already set to 1 to perform a proper service.

Changes to the reload value WDTREL, or the user-definable password WDTPW, are not required. However, changing WDTPW is recommended so that software can monitor Watchdog Timer service operations throughout the duration of an application program (see **Section 20.5.4**).

If WDT service is properly executed, Time-Out Mode is terminated, and the Watchdog Timer switches back to its former mode of operation, and Watchdog Timer service is complete.

20.5.4 Handling the User-Definable Password Field

WDT_CON0.WDTPW is an 8-bit field that can be set by software to any arbitrary value during a Modify Access. Settings of this field have no effect on the operation of the WDT, other than the role it plays in forming the password bit pattern, as discussed in **Section 20.4.3**.

The purpose of this field is to support further enhancements to the password protection scheme. For the following description, it is assumed that software does at least not fully compute the value for the Password Access from the contents of registers WDT_CON0

and WDT_CON1, but uses a predefined constant, embedded in the instruction stream, for the password (this is at least necessary for the user-definable password field WDTPW). For example, software can modify this field each time it executes a Watchdog service sequence. The next service sequence needs to take this new value into account for its Password Access. And it again changes the value during its Modify Access. Up to 256 different password values can be used. In this way, each service sequence is unique. If a malfunction occurs that, for instance, would result in the omission of one or more of these service sequences, the next service sequence would most probably not write the correct password. This service sequence would rely on the password value programmed during the normally preceding service sequence. However, if this one was skipped, the password value required by the contents of the Watchdog registers is the one programmed at the last service sequence executed before the malfunction had occurred. A Watchdog error condition would be detected in this case.

In the same manner, the Watchdog would detect the malfunction if a service sequence would be executed twice due to a falsely performed jump. **Figure 20-4** illustrates these examples.

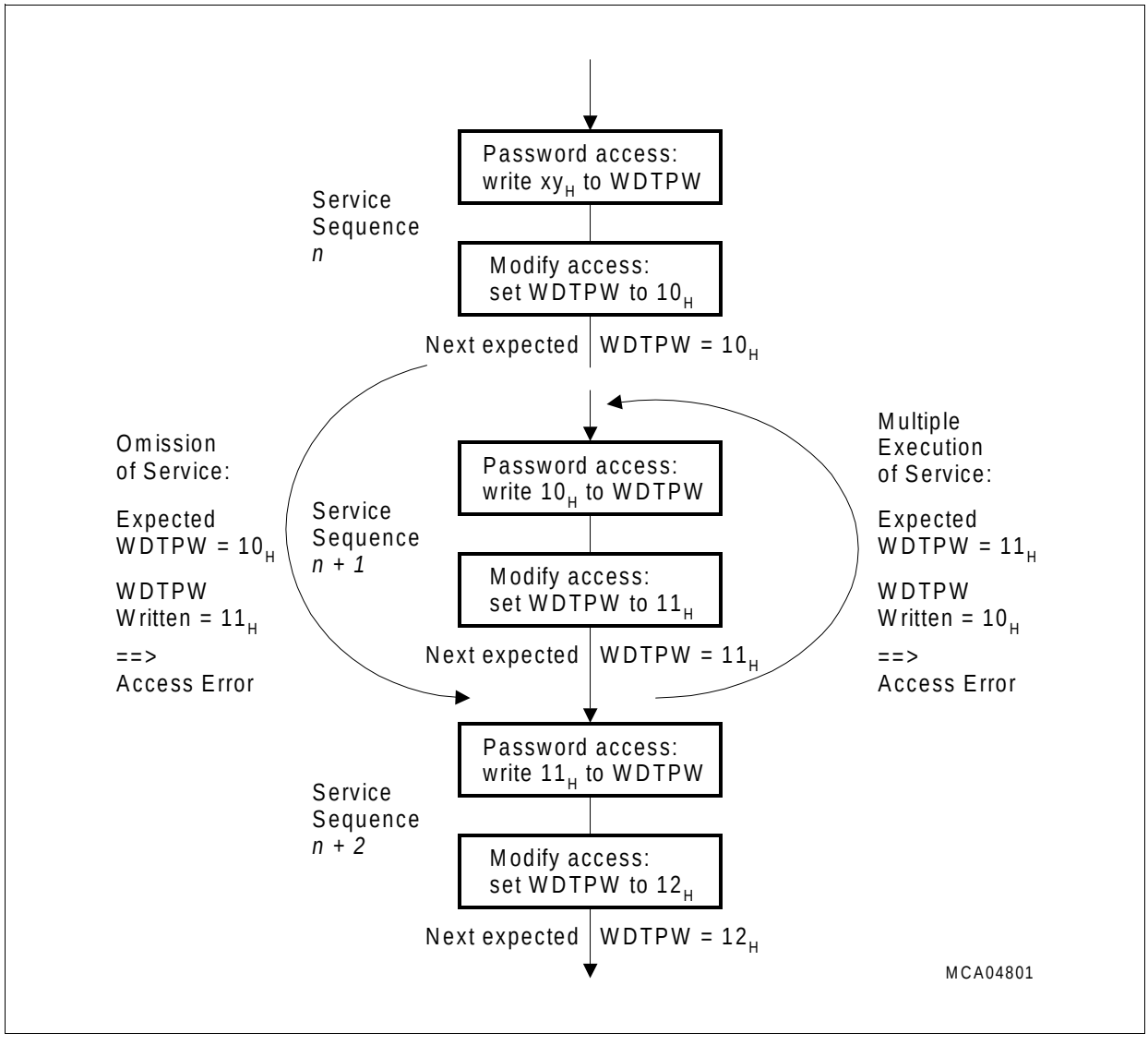


Figure 20-4 Detection of False Jumps and Loops

Other schemes are possible. Consider the case in which a routine determines some conditions that alter the program flow. One of two or more different paths will be executed next depending on these conditions. Before branching to the appropriate routine(s), software performs a Watchdog service and sets the new password value for WDTPW such that it depends on these conditions, that is, some or all of these condition codes can be incorporated into WDTPW. The next service sequence is performed at the point where the different paths come together again. To determine the correct password, software uses a value returned from the path which was executed. This value must match the value in WDTPW, otherwise the wrong path was executed. **Figure 20-5** shows an example for this.

It is also possible to have the different paths of a program compute the full or partial password to unlock register WDT_CON0. The password will only match at the next service sequence if all the expected paths and calculation routines have been executed properly. If one or more steps would have been omitted or a wrong path was executed due to a malfunction, the Watchdog failure mechanism will detect this and issue a reset of the device (after the prewarning phase).

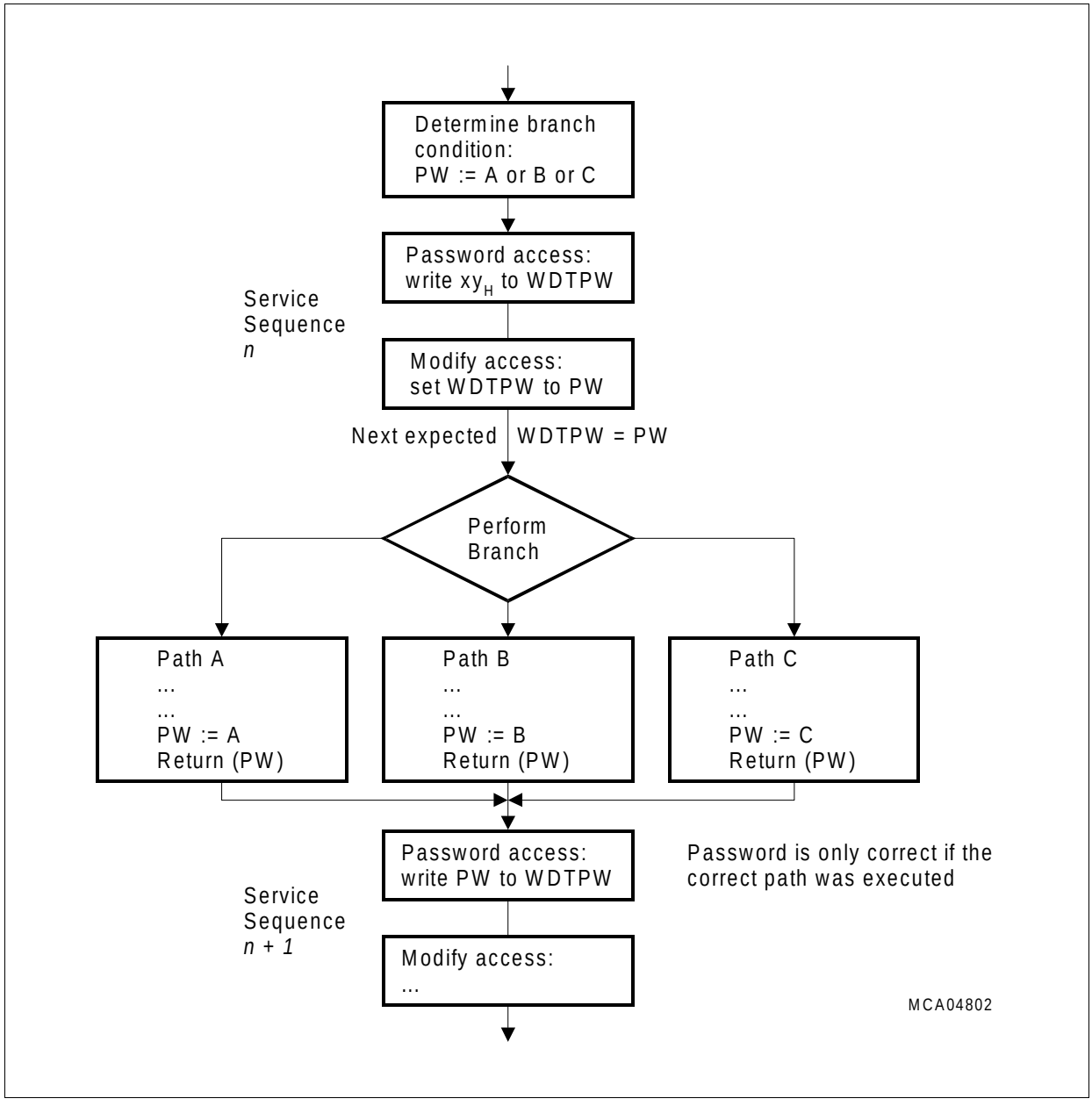


Figure 20-5 Monitoring Program Sequences

20.5.5 Determining the Required Values for a WDT Access

As described in **Section 20.4.3** and **Section 20.4.4**, the values required for the password and modify accesses to register WDT_CON0 are designed such that they can be derived from the values read from registers WDT_CON0 and WDT_CON1. However, at least some bits have to be modified in order to get the correct write value. This makes it very unlikely that a false operation derives values from reading these registers which inadvertently affect the WDT operation when written back to WDT_CON0. Even if a false write operation would have written the correct password to WDT_CON0, one further, different correct value needs to be written to this register in order to have an effect. In addition, the WDT switches to Time-Out Mode after the Valid Password Access, providing only a time-limited window for the second access.

While computing the required values from the current contents of the Watchdog registers is one option, the method of using predetermined values, set at compile-time of the program, may be the better approach in many cases. Usually, handling the Watchdog Timer is performed by one and only one task. Thus, the problem will not occur that another task might have changed some of the parameters which must not be modified (which would require reading the contents, modifying the value appropriately, and then writing it back). The one task handling the Watchdog Timer function would always “know” how it has programmed the WDT last time, and would therefore also “know” the next password value for opening WDT_CON0. In fact, this method would actually detect the case if another task had illegally modified the Watchdog registers, since the predetermined password might not work anymore, and a Watchdog error condition is generated.

In addition, accessing the WDT with predetermined values has the obvious benefit of shorter code, as no computing steps need to be performed.

20.6 Watchdog Timer Registers

Three registers are provided with the Watchdog Timer: WDT_CON0, WDT_CON1, and WDT_SR, as shown in **Figure 20-6**. They are located in the System Control Unit (SCU) Module.

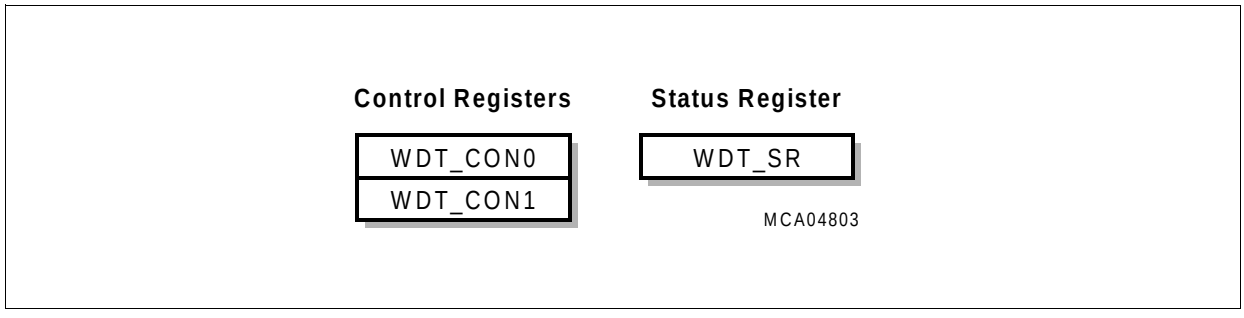


Figure 20-6 Watchdog Registers

Table 20-11 WDT Kernel Registers

Register Short Name	Register Long Name	Offset Address	Description see
WDT_CON0	Watchdog Timer Control Register 0	0020_H	**Page 20-27**
WDT_CON1	Watchdog Timer Control Register 1	0024_H	**Page 20-29**
WDT_SR	Watchdog Timer Status Register	0028_H	**Page 20-30**

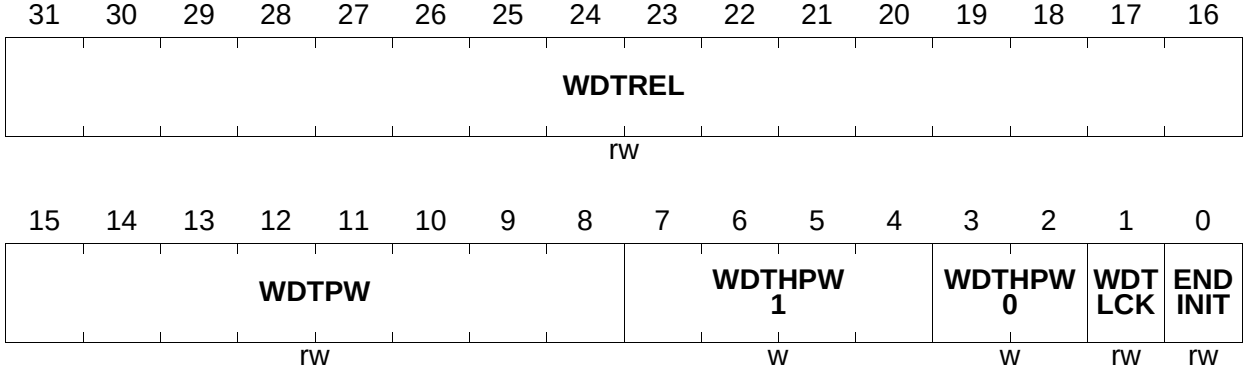
In the TC11IB, the registers of the Watchdog Timer are located in the address range of the SCU:

– Module Base Address: F000 0000_H
  Module End Address; F000 $00FF_H$
– Absolute Register Address = Module Base Address + Offset Address

20.6.1 Watchdog Timer Control Register 0

WDT_CON0 manages password access to the Watchdog Timer. It also stores the timer reload value, a user-definable password field, a lock bit, and the end-of-initialization (ENDINIT) control bit.

WDT_CON0
Watchdog Timer Control Register 0 **Reset Value: FFFC 0002$_H$**



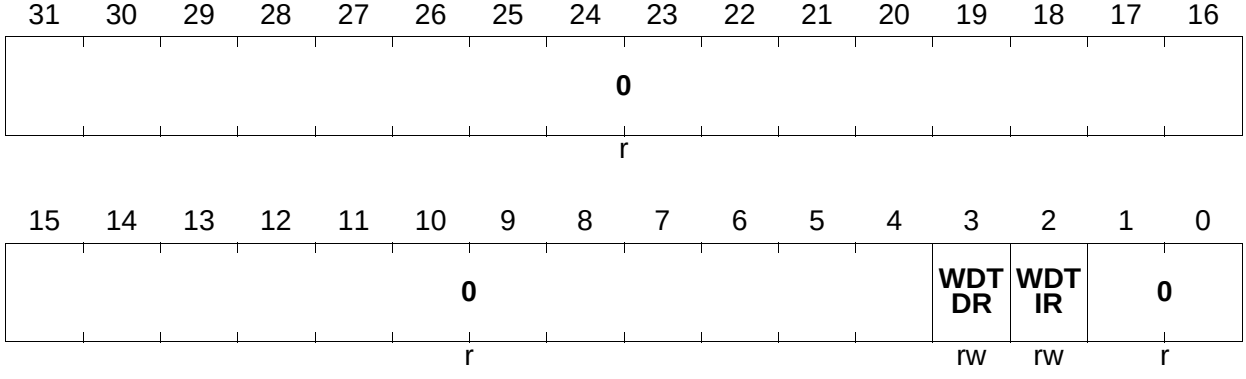
Field	Bits	Type	Description
ENDINT	0	rw	**End-of-Initialization Control Bit** 0 Access to Endinit-protected registers is permitted (default after reset). 1 Access to Endinit-protected registers is not permitted. ENDINIT controls the access to critical system registers. During a password access it must be written with its current value. It can be changed during a modify access to WDT_CON0.

Field	Bits	Type	Description
WDTLCK	1	rw	**Lock Bit to Control Access to WDT_CON0** 0 Register WDT_CON0 is unlocked. 1 Register WDT_CON0 is locked (default after reset). The actual value of WDTLCK is controlled by hardware. It is set to 0 after a successful password access to WDT_CON0 and automatically set to 1 again after a successful modify access to WDT_CON0. During a write to WDT_CON0 the value written to this bit is only used for the password-protection mechanism and is not stored. This bit must be set to 0 during a password access to WDT_CON0 and set to 1 during a modify access to WDT_CON0. That is, the inverted value read from WDTLCK always must be written to itself.
WDTHPW0	[3:2]	w	**Hardware Password 0** This field must be written with the value of the bits WDT_CON1.WDTDR and WDT_CON1.WDTIR during a password access. This field must be written with 0's during a modify access to WDT_CON0. When read, these bits always return 0.
WDTHPW1	[7:4]	w	**Hardware Password 1** This field must be written to 1111_B during both, a password access and a modify access to WDT_CON0. When read, these bits always return 0.
WDTPW	[15:8]	rw	**User-Definable Password Field for Access to WDT_CON0** This bit field must be written with its current contents during a password access. It can be changed during a modify access to WDT_CON0.
WDTREL	[31:16]	rw	**Reload Value for the Watchdog Timer** If the Watchdog Timer is enabled and in Normal Timer Mode, it will start counting from this value after a correct Watchdog service. This field must be written with its current contents during a password access. It can be changed during a modify access to WDT_CON0 ($FFFC_H$ = default after reset).

20.6.2 Watchdog Timer Control Register 1

WDT_CON1 manages operation of the WDT. It includes the disable request and frequency selection bits. It is ENDINIT-protected.

WDT_CON1
Watchdog Timer Control Register 1 **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
WDTIR	2	rw	**Watchdog Timer Input Frequency Req. Control Bit** 0 Request to set input frequency to f_{SYSCLK}/16384 1 Request to set input frequency to f_{SYSCLK}/256 This bit can only be modified if WDT_CON0.ENDINIT is set to 0. WDT_SR.WDTIS is updated by this bit only when ENDINIT is set to 1 again. As long as ENDINIT is left at 0, WDT_SR.WDTIS controls the current input frequency of the Watchdog Timer. When ENDINIT is set to 1 again, WDT_SR.WDTIS is updated with the state of WDTIR.
WDTDR	3	rw	**Watchdog Timer Disable Request Control Bit** 0 Request to enable the Watchdog Timer. 1 Request to disable the Watchdog Timer. This bit can only be modified if WDT_CON0.ENDINIT is set to 0. WDT_SR.WDTDS is set to this bit's value when ENDINIT is set to 1 again. As long as ENDINIT is left at 0, bit WDT_SR.WDTDS controls the current enable/disable status of the Watchdog Timer. When ENDINIT is set to 1 again with a valid modify access, WDT_SR.WDTDS is updated with the state of WDTDR.

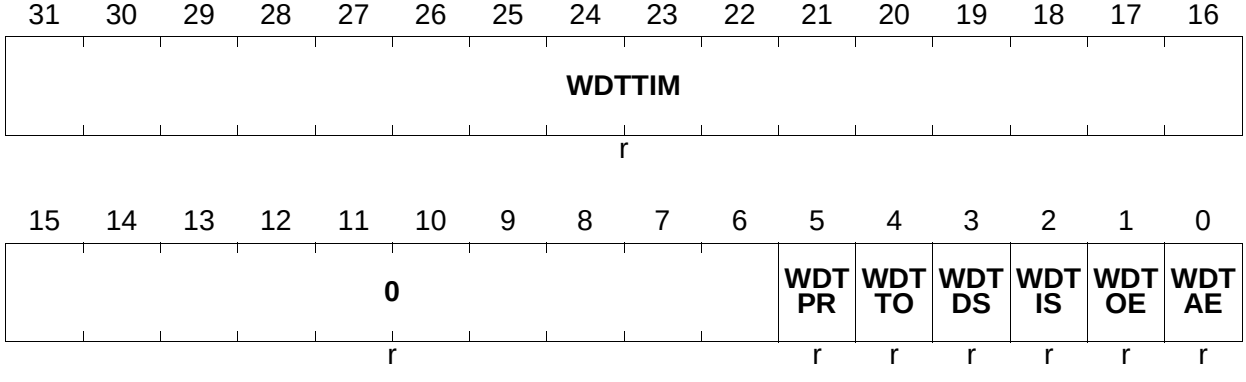
Field	Bits	Type	Description
0	[1:0], [31:4]	r	**Reserved**; read as 0; should be written with 0;

20.6.3 Watchdog Timer Status Register

WDT_SR shows the current state of the WDT. Status include bits indicating reset prewarning, Time-out, enable/disable status, input clock status, and access error status.

The reset value for this register is depending on the cause of the reset. For any reset other than a Watchdog reset, the reset value is FFFC 001U_H. After a Watchdog reset, bits WDTAE and WDTOE indicate the type of Watchdog error which occurred before the Watchdog reset. Either one or both bits can be set. These bits are not reset on a Watchdog reset. Bits WDTDS and WDTIS are always 0 after any reset.

WDT_SR
Watchdog Timer Status Register **Reset Value: FFFC 0010$_H$**



Field	Bits	Type	Description
WDTAE	0	r	**Watchdog Access Error Status Flag** 0 No Watchdog access error. 1 An Watchdog access error has occurred. This bit is set by hardware when an illegal password access or modify access to register WDT_CON0 was attempted. This bit is only reset through: – a power-on, hardware, or software reset occurs – WDT_CON0.ENDINIT is set to 1 during a valid modify access. However it is not possible to reset this bit if the WDT is in Prewarning Mode, indicated by WDT_SR.WDTPR = 1.

Field	Bits	Type	Description
WDTOE	1	r	**Watchdog Overflow Error Status Flag** 0 No Watchdog overflow error 1 A Watchdog overflow error has occurred. This bit is set by hardware when the Watchdog Timer overflows from $FFFF_H$ to 0000_H. This bit is only reset when: – a power-on, hardware, or software reset occurs; – WDT_CON0.ENDINIT is set to 1 during a valid modify access. However it is not possible to reset this bit if the Watchdog Timer is in Prewarning Mode, indicated by WDT_SR.WDTPR = 1.
WDTIS	2	r	**Watchdog Input Clock Status Flag** 0 Watchdog Timer input clock is f_{SYSCLK}/16384 (default after reset). 1 Watchdog Timer input clock is f_{SYSCLK}/256. This bit is updated with the state of bit WDT_CON1.WDTIR after WDT_CON0.ENDINIT is written with 1 during a valid modify access to register WDT_CON0.
WDTDS	3	r	**Watchdog Enable/Disable Status Flag** 0 Watchdog Timer is enabled (default after reset). 1 Watchdog Timer is disabled. This bit is updated with the state of bit WDT_CON1.WDTDR after WDT_CON0.ENDINIT is written with 1 during a valid modify access to register WDT_CON0.
WDTTO	4	r	**Watchdog Time-Out Mode Flag** 0 Normal mode 1 The Watchdog is operating in Time-Out Mode (default after reset) This bit is set to 1 when Time-Out Mode is entered, automatically after a reset and after every password access to register WDT_CON0. It is automatically reset by hardware when Time-Out Mode is properly terminated through a valid modify access to WDT_CON0. It is left set when a Watchdog error occurs during Time-Out Mode, and Prewarning Mode is entered.

Field	Bits	Type	Description
WDTPR	5	r	**Watchdog Prewarning Mode Flag** 0 Normal mode (default after reset) 1 The Watchdog is operating in Prewarning Mode This bit is set to 1 when a Watchdog error is detected. The Watchdog Timer has issued an NMI trap and is in Prewarning Mode. A reset of the chip occurs after the prewarning period has expired.
WDTTIM	[31:16]	r	**Watchdog Timer Value** Reflects the current content of the Watchdog Timer.
0	[15:6]	r	**Reserved**; read as 0;

21 On-Chip Debug Support

The On-Chip Debug Support (OCDS) of the TC11IB consists of four building blocks:

- OCDS module in the TriCore CPU
- OCDS module in the PCP
- Trace module of the TriCore
- Debugger Interface (Cerberus)

Figure 21-1 shows a basic block diagram of the building blocks.

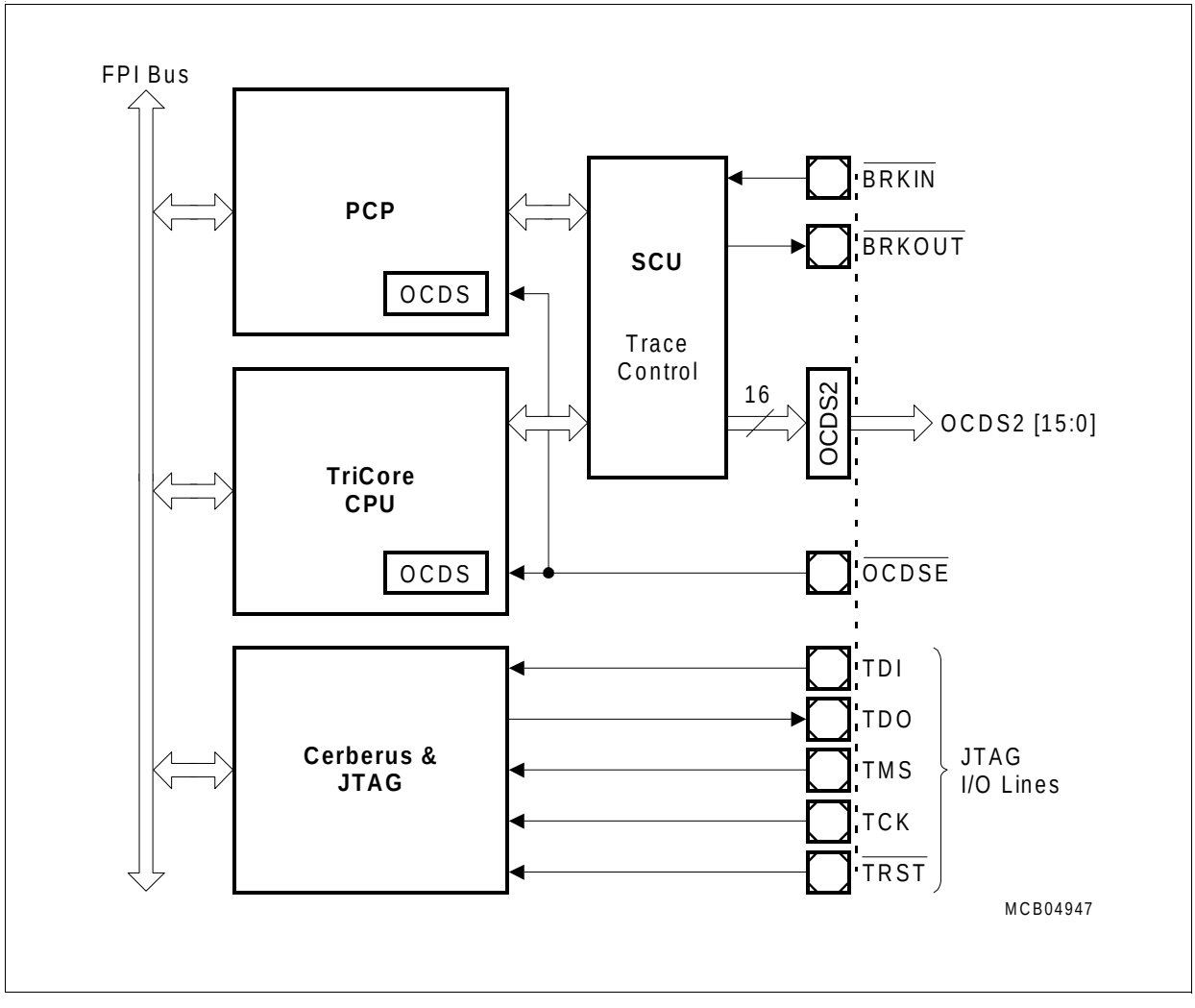


Figure 21-1 OCDS Basic Block Diagram

21.1 TriCore CPU Debug Support

The TriCore CPU in the TC11IB provides On-Chip Debug Support (OCDS) with the following features:

- On-chip breakpoint hardware
- Support of an external break signal

21.1.1 Basic Concepts

The TriCore breakpoint concept has two parts. The first part defines the generation of debug events and the second part defines what actions are taken when a debug event is generated.

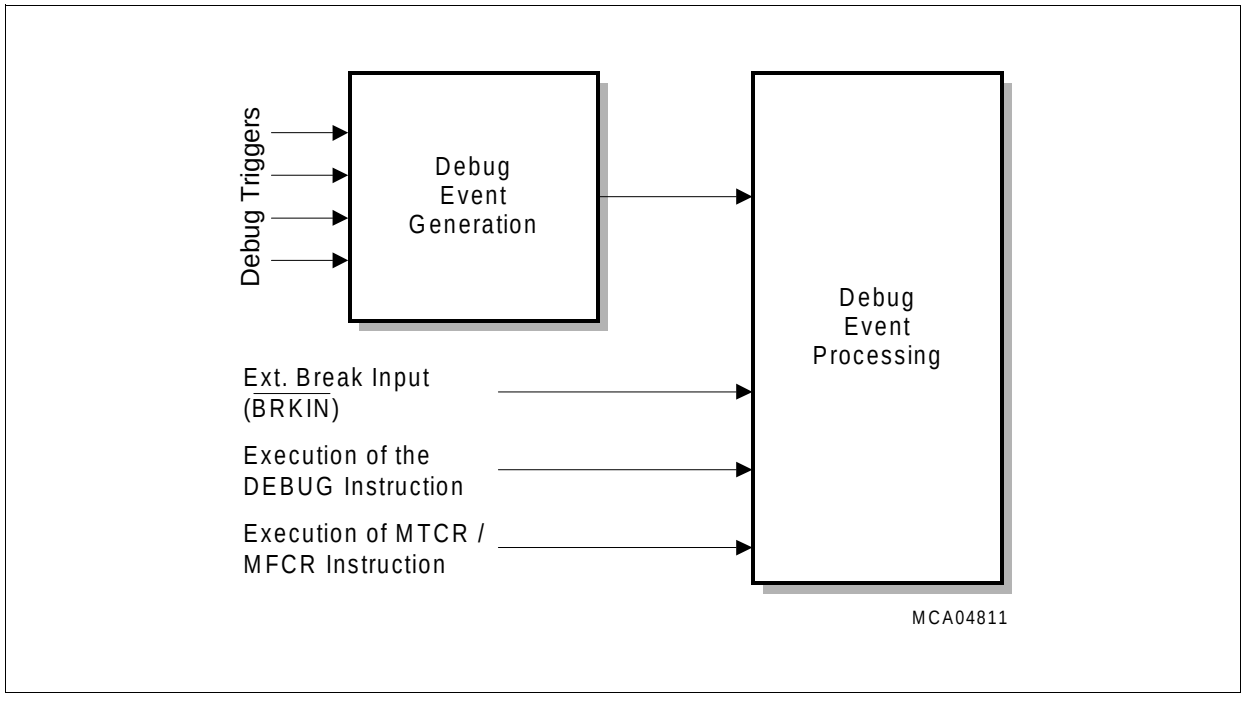


Figure 21-2 Basic TriCore Debug Concept

21.1.2 Debug Event Generation

In order for any debug event to be generated, the debug enable bit DBGSR.DE in the Debug Status Register must be set. If this bit is set, debug events can be generated by the:

– An active (low) signal at the OCDS Break Input pin $\overline{\text{BRKIN}}$
– Execution of a debug instruction
– Execution of a MTCR/MFCR instruction
– Debug event generation unit

21.1.2.1 External Debug Break Input

An external debug break pin is provided to allow the emulator to interrupt the processor asynchronously. The action that is performed when the external debug break input is activated is defined by the contents of the External Break Input Event Specifier Register EXEVT.

Note: The CPU core detects the active edge of $\overline{BRKIN}$ and performs the action specified in EXEVT at the first available opportunity.

21.1.2.2 Software Debug Event Generation

The TriCore architecture also supports a mechanism through which software can explicitly generate a debug event. This can be used, for instance, by a debugger to patch code held in RAM in order to implement breakpoints. A special DEBUG instruction is defined which is a user mode instruction, and its operation depends on whether the debug mode is enabled.

If debug mode is enabled (DBGSR.DE = 1), the DEBUG instruction causes a debug event to be raised and the action defined in the Software Break Event Specifier Register SWEVT is taken. If the debug mode is not enabled, then the DEBUG instruction is treated as a NOP instruction.

Both 16-bit and 32-bit forms of the DEBUG instruction are provided.

21.1.2.3 Execution of a MTCR or MFCR Instruction

In order to protect the emulator resource, a debug event is raised whenever a MTCR or MFCR instruction is used to read or modify an user core SFR. That means that an event is not raised when the user reads or modifies one of the dedicated debug core SFRs:

- DBGSR or
- CREVT or
- SWEVT or
- EXEVT or
- TR0EVT or
- TR1EVT

The action that is performed when a MTCR or MFCR instruction is executed on user core SFRs defined by the content of the Emulator Resource Protection Event Specifier Register CREVT.

21.1.2.4 Debug Event Generation from Debug Triggers

The debug event generation unit is responsible for generating debug events when a programmable set of debug triggers are active. Debug triggers may come from the following sources:

- Code protection logic
- Data protection logic

These debug triggers provide the inputs to a programmable block of combinational logic that outputs debug events.

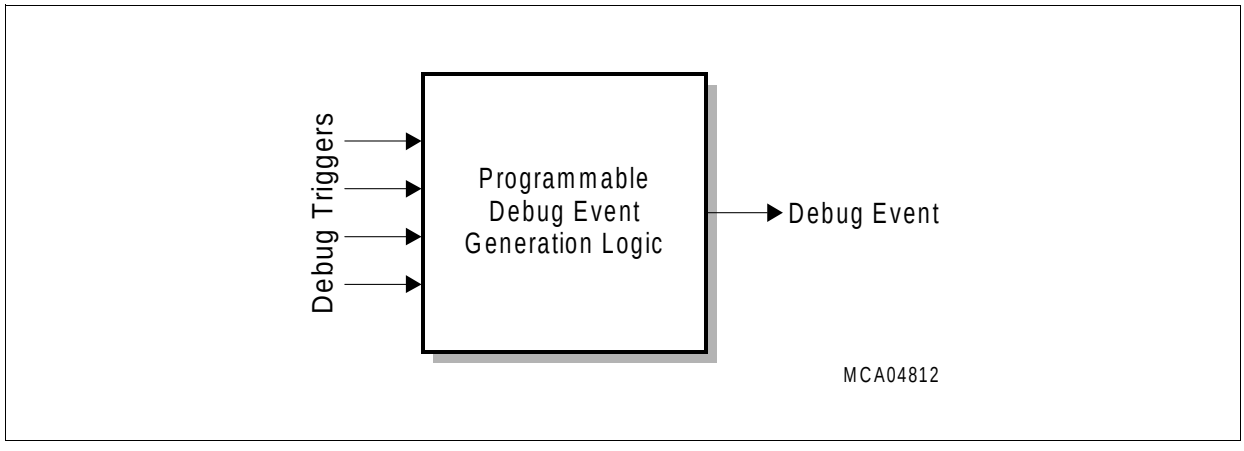


Figure 21-3 Debug Event Generation Logic

The aim is to be able to specify the breakpoints which use fairly simple criteria purely in the on-chip debug event generation unit, and to rely on help from the external debug system or debug monitor to implement more complex breakpoints.

21.1.3 Debug Triggers

21.1.3.1 Protection Mechanism

The TriCore debug system is also integrated into the protection mechanism which can generate the following types of debug triggers:

- Trigger on execution of an instruction at a specific address
- Trigger on execution of an instruction within a range of addresses
- Trigger on the loading of a value from a specific address
- Trigger on the loading of a value from anywhere in a range of addresses
- Trigger on the storing of a value to a specific address
- Trigger on the storing of a value to anywhere in a range of addresses

Informations on the generation of debug triggers by the protection mechanism are given in the TriCore Architecture Manual.

21.1.3.2 Combination of Triggers

In the TC11IB the first two code and data ranges can be used to generate one debug event. The TriCore CPU in the TC11IB allows one code range and one data range to be combined for a debug event generation. The combination is controlled by the Trigger Event n Specifier Registers TRnEVT (n = 0, 1).

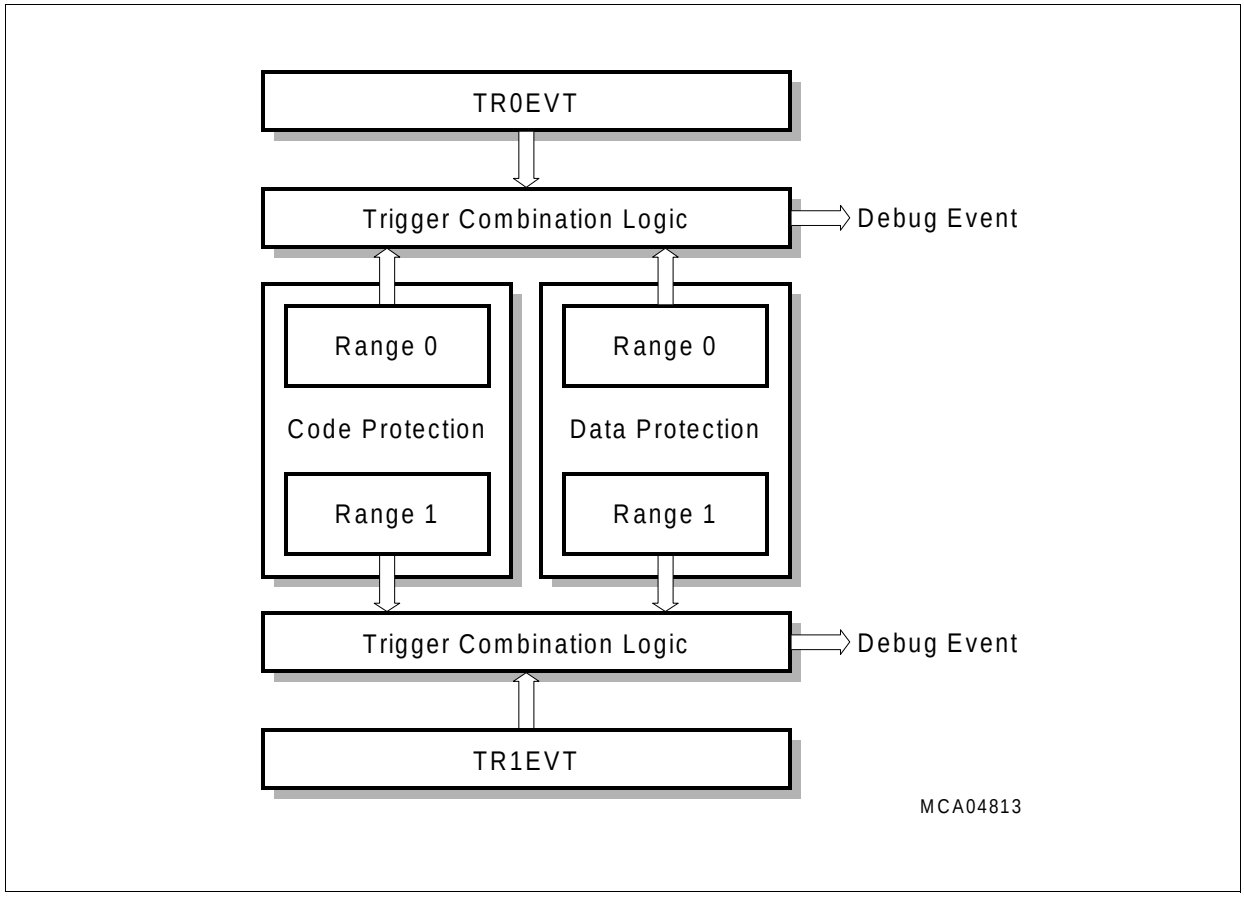


Figure 21-4 Combination of Data and Code Triggers

For example, the inputs from range 0 of the code protection logic can be combined with the inputs from range 0 of the data protection logic. This combination and the action taken if a debug event is generated are controlled by the TR0EVT register.

The debug event generation logic places certain restrictions on which debug triggers can be combined in order to produce a debug event whose action is marked as “break before make”.

All debug events that are produced from a combination of triggers which include inputs from the data protection logic are treated as “break after make”, irrespective of the event specifier.

21.1.4 Actions Taken on a Debug Event

When a debug event is generated, one of the following actions is taken.

21.1.4.1 Assert an External Pin $\overline{\text{BRKOUT}}$

A signal can be asserted on the external pin $\overline{\text{BRKOUT}}$. This could be used in critical routines where the system cannot be interrupted to signal to the external world that a particular event has happened. This feature could also be useful to synchronize the internal and external debug hardware.

For example, when the CPU writes to an off-chip location through the external bus interface, this could be detected and the external pin asserted. This could then be used as the input trigger to an analyzer to capture the bus cycles on the external interface pins.

21.1.4.2 Halt

The halt mode performs a selective cancellation of:

- All instruction after and including the instruction that caused the breakpoint if EXEVT.BBM = 1.
- All instructions after the instruction that caused the breakpoint if EXEVT.BBM = 0.

Once the pipeline has been cancelled, it enters a halt mode where no more instructions are executed. It then relies on the external debug system to interrogate the target purely through the mapping of the architectural state into the FPI address space without any help from the core.

While halted, the core will not respond to any interrupts, and will only resume execution once the external debug hardware clears the halt bit by writing 10_B to the DBGSR.HALT bit field.

When the halt mode is entered, the following actions are also performed:

- The DBGSR.EVTSRC bit field is updated.
- The breakout pin $\overline{\text{BRKOUT}}$ is asserted for one cycle.

21.1.4.3 Breakpoint Trap

The breakpoint trap is designed to be used to enter a debug monitor without using any user resource. It relies upon the following emulator resources:

- The debug monitor is held in the emulator region at address DE00 0000_H.
- There is a 4-word area of RAM available at address DE80 0000_H which can be used to store critical state during the debug monitor entry sequence.

When a breakpoint trap is taken, the following actions are performed:

- Write PSW to DE80 0000_H
- Write PCXI to DE80 0004_H
- Write A10 to DE80 0008_H

– Write A11 to DE80 000C$_H$
– A11 = breakpoint PC
– PC = DE00 0000$_H$
– PSW.PRS = 0
– PSW.IO = 2
– PSW.GW = 0
– PSW.IS = 1
– ICR.IE = 0

The corresponding return sequence is provided through the RFM (return from monitor) instruction. This effectively perform the reverse of the above:

– Branch to A11
– Restore PSW from DE80 0000$_H$
– Restore PCXI from DE80 0004$_H$
– Restore A10 from DE80 0008$_H$
– Restore A11 from DE80 000C$_H$

This provides an automated route into the debug monitor which does not use any user resource. The RFM instruction is then used to return control the original task.

When the debug trap is taken, the following actions are also performed:

– The EVTSRC bit field in DBGSR is updated.
– The $\overline{\text{BRKOUT}}$ pin is asserted for one cycle.

21.1.4.4 Software Breakpoint

When a debug event is raised, the system can enter the software debug mode. The software debug mode is basically an interrupt. The software breakpoint interrupt is controlled in the Software Breakpoint Service Request Control Register SBSRC.

21.1.5 OCDS Registers

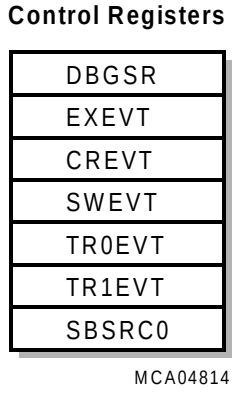


Figure 21-5 OCDS Registers

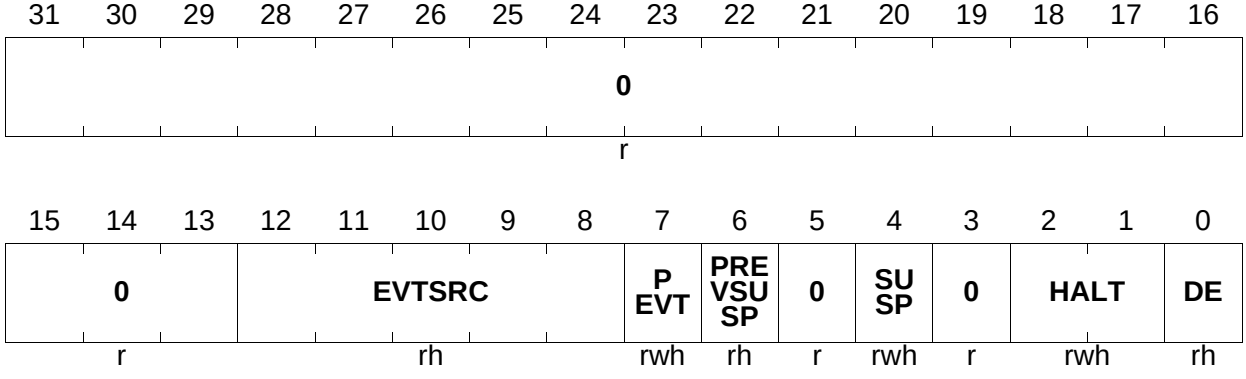
Table 21-1 OCDS Registers

Register Short Name	Register Long Name	Offset Address	Description see
DBGSR	Debug Status Register	0000_H	Page 21-9
EXEVT	External Break Input Event Specifier Register	0008_H	Page 21-11
CREVT	Emulator Resource Protection Event Specifier Register	$000C_H$	Page 21-12
SWEVT	Software Break Event Specifier Register	0010_H	Page 21-13
TR0EVT	Trigger Event 0 Specifier Register	0020_H	Page 21-14
TR1EVT	Trigger Event 1 Specifier Register	0024_H	Page 21-14
SBSRC0	Software Breakpoint Service Request Control Register 0	$00BC_H$ [1)]	Page 21-15

1) The SBSRC register is located in the address range of the CPU slave interface CPS (see Section 21.6).

DBGSR
Debug Status Register

Reset Value: 0000 0000$_H$

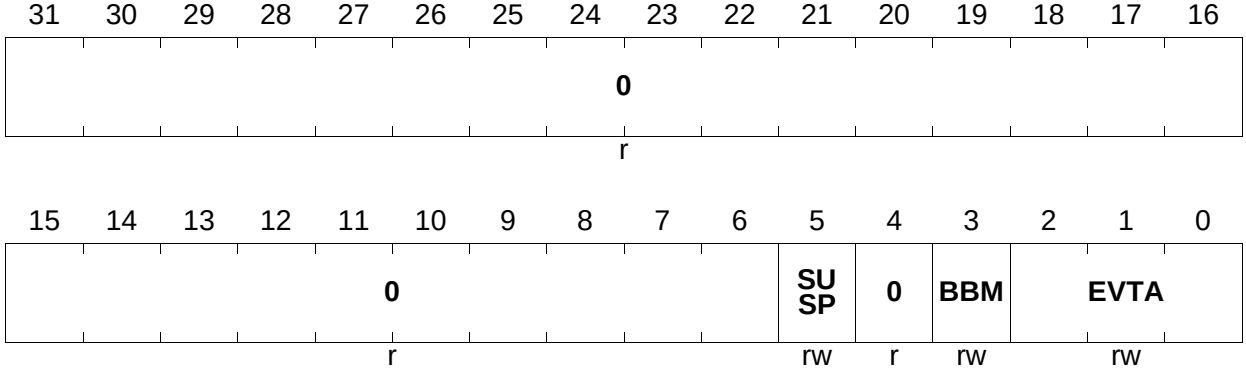


Field	Bits	Type	Description
DE	0	rh	**Debug Enable** Indicates whether debug support was enabled at reset 0 Debug disabled 1 Debug enabled
HALT	[2:1]	rwh	**CPU Halt Request / Status Field** HALT can be set or cleared by software. HALT[0] is the actual halt bit. HALT[1] is a mask bit to specify whether HALT[0] is to be updated on a software write or not. HALT[1] is always read as 0. HALT[1] must be set to one in order to update HALT[0] by software (R: read; W: write). 00 R: CPU running / W: HALT[0] unchanged 01 R: CPU halted / W: HALT[0] unchanged 10 R: n.a. / W: reset HALT[0] 11 R: n.a. / W: if debug support is enabled (DE = 1), set HALT[0]; if debug support is not enabled (DE = 0), HALT[0] is left unchanged
SUSP	4	rwh	**Current State of the Suspend Signal** 0 Suspend inactive 1 Suspend active
PREVSUSP	6	rh	**Previous State of the Suspend Signal** 0 Previous suspend inactive 1 Previous suspend active
PEVT	7	rwh	**Posted Event** 0 No posted event 1 Posted event

Field	Bits	Type	Description
EVTSRC	[12:8]	rh	**Event Source** 0 EXTEVT 1 CREVT 2 SWEVT 16 + n TRnEVT (n = 0, 1) other Reserved
0	3, 5, [31:13]	r	**Reserved**; read as 0; should be written with 0.

EXEVT
External Break Input Event Specifier Register

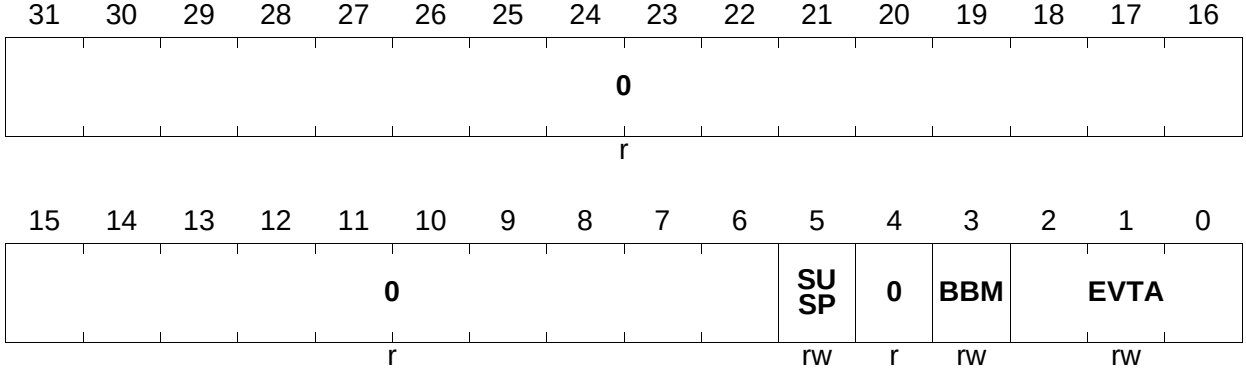
Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Specifies the action associated with the event: 000 None; disabled 001 Assert external pin $\overline{\text{BRKOUT}}$ 010 Halt 011 Breakpoint trap 100 Software breakpoint 0 101 Reserved, same behavior as 000 110 Reserved, same behavior as 000 111 Reserved, same behavior as 000
BBM	3	rw	**Break Before Make or Break After Make Selection** 0 Break after make 1 Break before make
SUSP	5	rw	**OCDS Suspend Signal State** Value to be assigned to the OCDS suspend signal when the event is raised.
0	4, [31:6]	r	**Reserved**; read as 0; should be written with 0.

CREVT

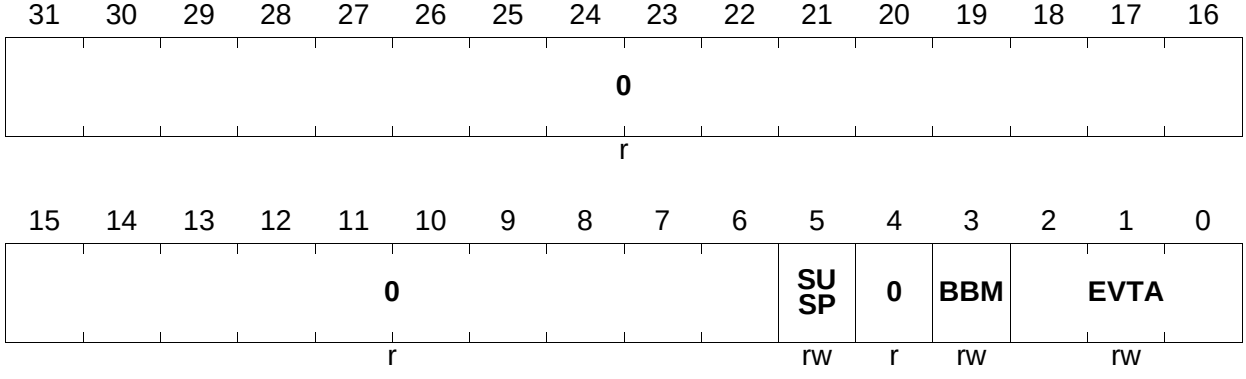
Emulator Resource Protection Event Specifier Register Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Specifies the action associated with the event: 000 None; disabled 001 Assert external pin $\overline{\text{BRKOUT}}$ 010 Halt 011 Breakpoint trap 100 Software breakpoint 0 101 Reserved, same behavior as 000 110 Reserved, same behavior as 000 111 Reserved, same behavior as 000
BBM	3	rw	**Break Before Make or Break After Make Selection** 0 Break after make 1 Break before make
SUSP	5	rw	**OCDS Suspend Signal State** Value to be assigned to the OCDS suspend signal when the event is raised.
0	4, [31:6]	r	**Reserved**; read as 0; should be written with 0.

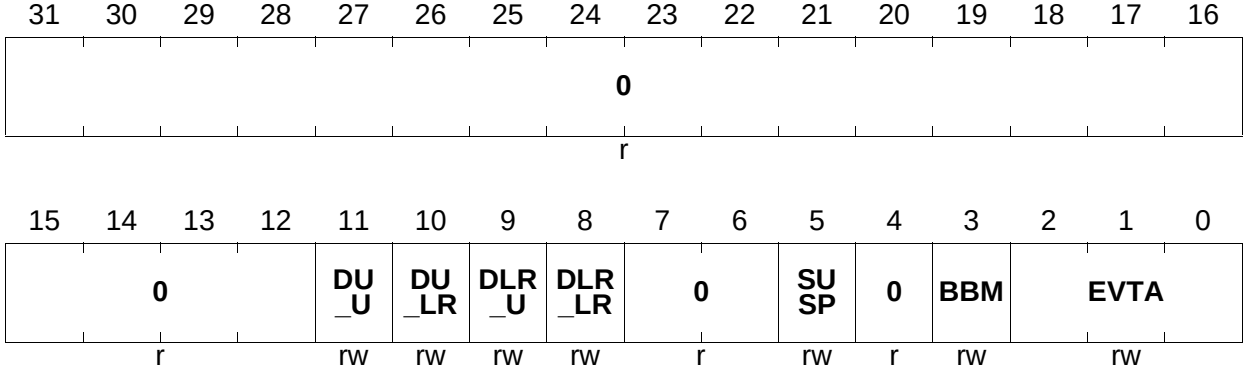
SWEVT
Software Break Event Specifier Register

Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Specifies the action associated with the event: 000 None; disabled 001 Assert external pin $\overline{\text{BRKOUT}}$ 010 Halt 011 Breakpoint trap 100 Software breakpoint 0 101 Reserved, same behavior as 000 110 Reserved, same behavior as 000 111 Reserved, same behavior as 000
BBM	3	rw	**Break Before Make or Break After Make Selection** 0 Break after make 1 Break before make
SUSP	5	rw	**OCDS Suspend Signal State** Value to be assigned to the OCDS suspend signal when the event is raised.
0	4, [31:6]	r	**Reserved**; read as 0; should be written with 0.

TR0EVT
Trigger Event 0 Specifier Register **Reset Value: 0000 0000$_H$**
TR1EVT
Trigger Event 1 Specifier Register **Reset Value: 0000 0000$_H$**



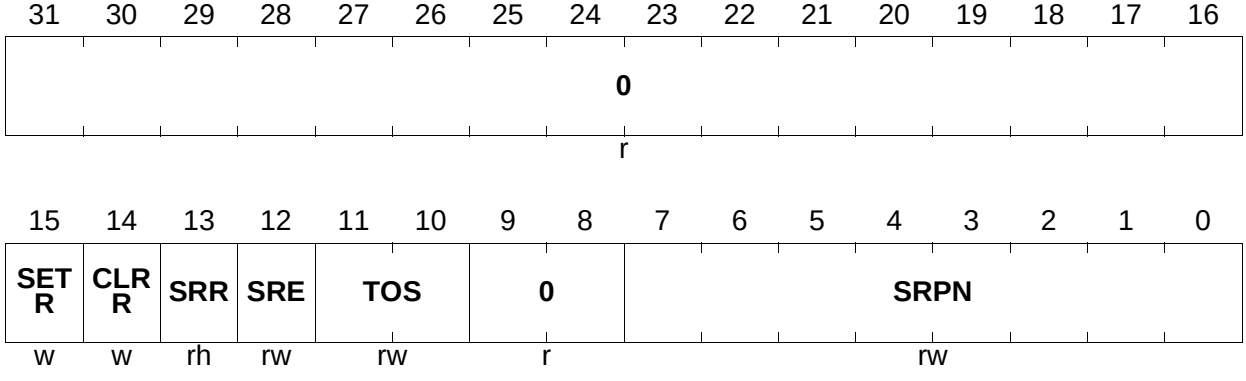
Field	Bits	Type	Description
EVTA	[2:1]	rw	**Event Associated** Specifies the action associated with the event: 000 None; disabled 001 Assert external pin $\overline{\text{BRKOUT}}$ 010 Halt 011 Breakpoint trap 100 Software breakpoint 0 101 Reserved, same behavior as 000 110 Reserved, same behavior as 000 111 Reserved, same behavior as 000
BBM	3	rw	**Break Before Make or Break After Make Selection** 0 Break after make 1 Break before make *Note: This bit will be ignored for events which include the data protection inputs because these events only can be break after make.*
SUSP	5	rw	**OCDS Suspend Signal State** Value to be assigned to the OCDS suspend signal when the event is raised.
DLR_LR	8	rw	Controls combination of D_{LR} and C_{LR}
DLR_U	9	rw	Controls combination of D_{LR} and C_U
DU_LR	10	rw	Controls combination of D_U and C_{LR}
DU_U	11	rw	Controls combination of D_U and C_U

Field	Bits	Type	Description
0	4,[7:6], [31:12]	r	**Reserved**; read as 0; should be written with 0.

The software breakpoint is controlled by the Software Breakpoint Service Request Control Register.

SBSRC0
Software Breakpoint Service Request Control Register 0

Reset Value: 0000 0000$_H$



Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number** 00_H Software breakpoint service request is never serviced 01_H- Software breakpoint service request is on lowest priority FF_H Software breakpoint service request is on highest priority
TOS	[11:10]	rw	**Type of Service Control** 00 CPU service is initiated 01 PCP request is initiated 1X Reserved
SRE	12	rw	**Service Request Enable** 0 Software breakpoint service request is disabled 1 Software breakpoint service request is enabled

Field	Bits	Type	Description
SRR	13	rh	**Service Request Flag** 0 No software breakpoint service request is pending 1 A software breakpoint service request is pending
CLRR	14	w	**Request Clear Bit** CLRR is required to reset SRR. 0 No action 1 Clear SRR; bit value is not stored; read always returns 0; no action if SETR is set too
SETR	15	w	**Request Set Bit** SETR is required to set SRR. 0 No action 1 Set SRR; bit value is not stored; read always returns 0; no action if SETR is set too
0	[9:8], [31:16]	r	**Reserved**; returns 0 if read; should be written with 0.

*Note: Further details on interrupt handling and processing are described in **Chapter 15** of this User's Manual.*

21.2 PCP Debug Support

A special PCP instruction, DEBUG, is provided for debugging the PCP. It can be placed at important locations inside the code to track and trace program execution. The execution of the instruction depends on a condition code specified with the instruction. The actions programmed for this instruction will only take place if the specified condition is true.

Further details on the PCP debugging features are described in **Chapter 17** of this User's Manual.

21.3 Multi-Core Debug Support

The TC11IB supports multi-core debug. It touches the following points:

- Break-input distribution to the various cores and break-output generation
- Suspend signal generation for the peripherals
- Instruction trace for the cores.

21.3.1 Break and Suspend Control

In a System On a Chip (SOC), there can be several processor cores (MPUs, MCUs, DSPs, etc.) for which software is developed **Figure 21-6**. For debugging this software, it is required, that the run control (OCDS) of one processor can break also the other processor cores. This is configured in a central break and suspend signal switch, which is located in Cerberus. But the processor cores must have break in and out signals, which are low active. The suspend signal requests are high active. The structure includes two independent break buses, break in/out sources and suspend generation. These signals are controlled by Special Function Registers. The external debugger can write these registers directly with Cerberus or indirectly with a monitor which communicates across for instance a serial interface.

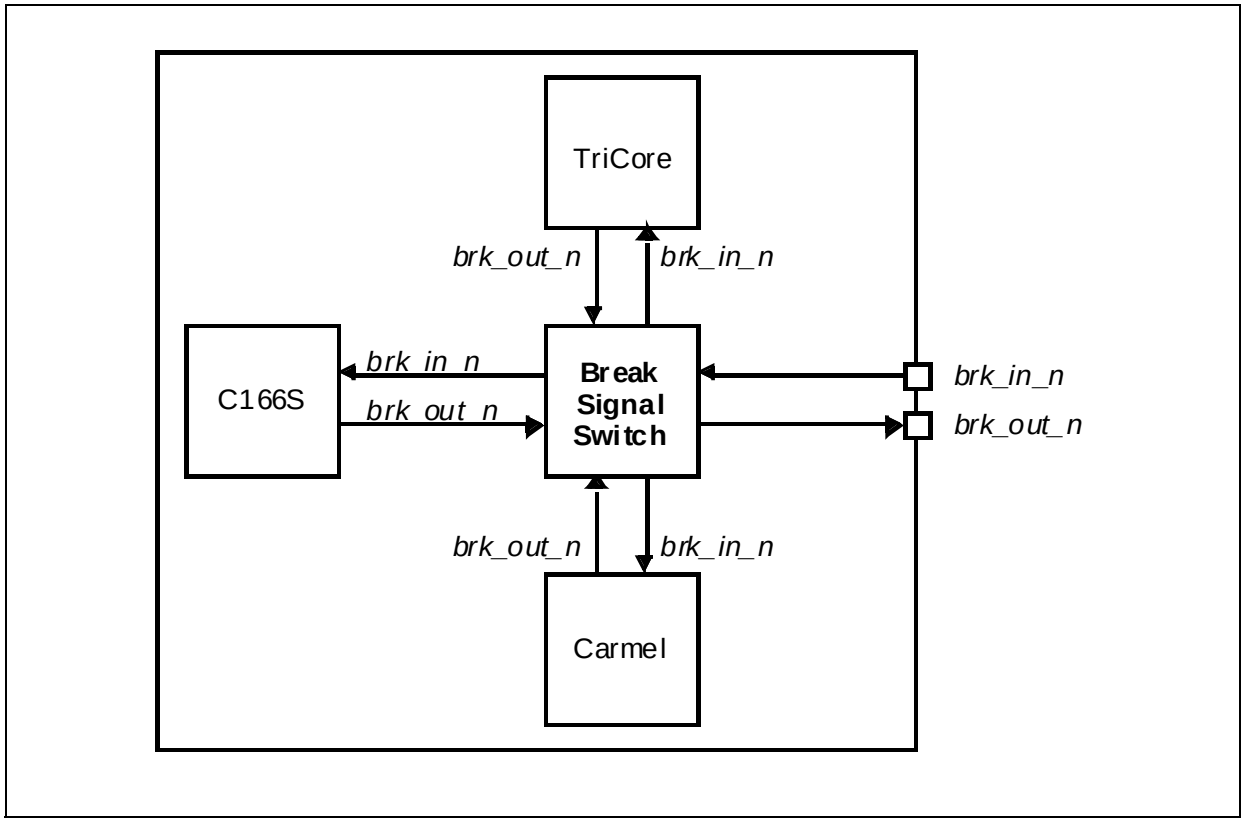


Figure 21-6 Example for a Multi-Core System On a Chip

Differences between Break and Suspend Signals

There are clear differences between break and suspend signals (**Table 21-2**). However if a processor core's OCDS does not provide both types, the MultiCore Debug Switch allows with some restrictions also to use the one type for the other purpose.

Table 21-2 Differences between Break and Suspend Signals

	Break Signals	Suspend Signals
Time to be active	A break output is active as long as the break condition is true. For instance as long as the program is in a predefined address range.	A suspend output is activated on a breakpoint hit or through writing the suspend bit in the control register of the OCDS. It is deactivated "manually" by the external debugger or by the monitor.
Function	An active break output signals to an external debugger, that the break condition is true. This is without any impact on the system behavior. The break output can be used to break other cores.	Suspend signals request sensitive parts of the system (e.g. peripherals) to suspend. When suspend is deactivated, these parts resume.
	Break inputs are provided to asynchronously break a core. This is done either by the external debugger or by another core via the break switch.	

21.3.1.1 Break Bus Switch

The Break Bus Switch is shown in **Figure 21-7**.

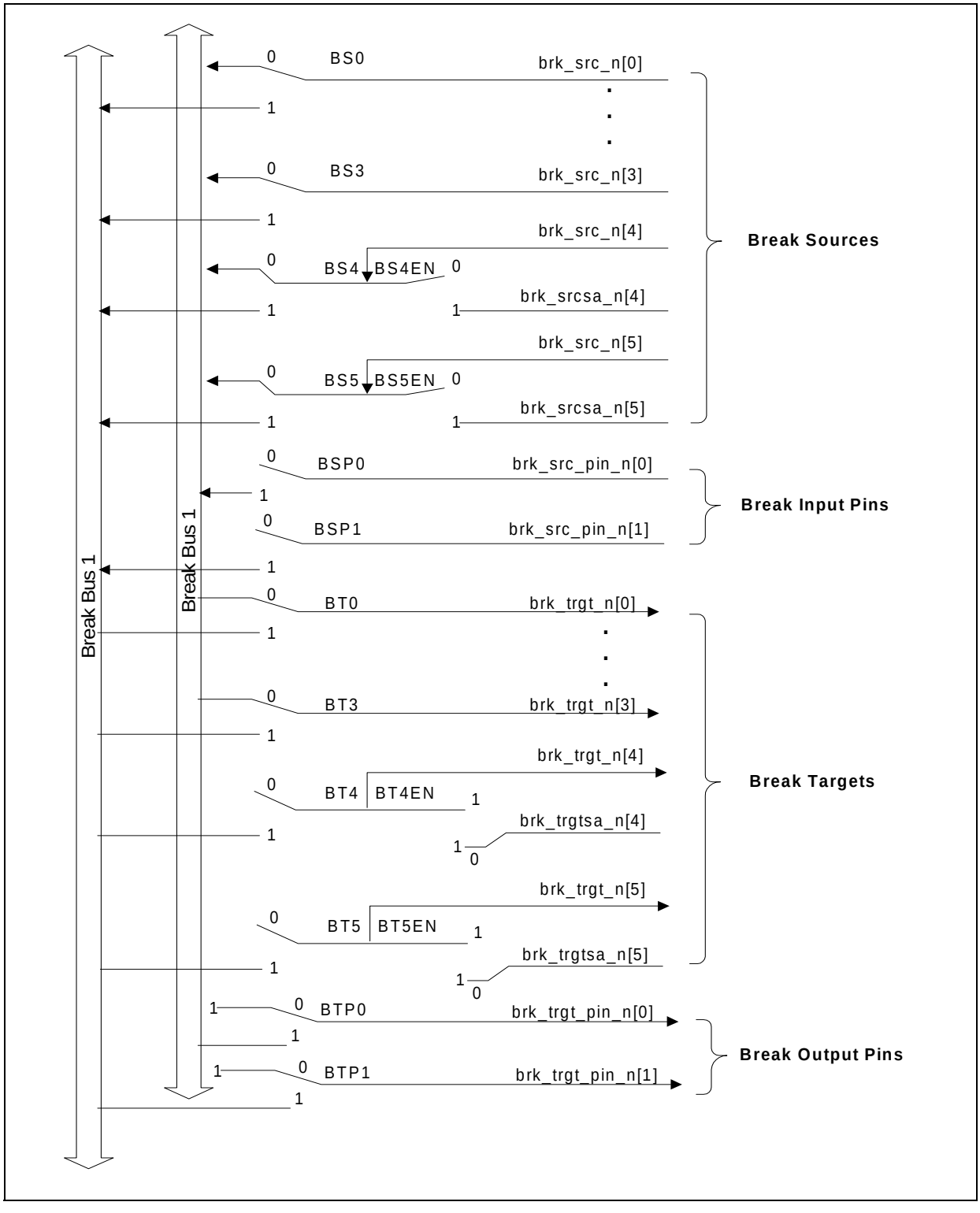


Figure 21-7 Break Bus Switch

21.3.1.2 Suspend Signal Generation

The Suspend Signal Generation is shown in **Figure 21-8**.

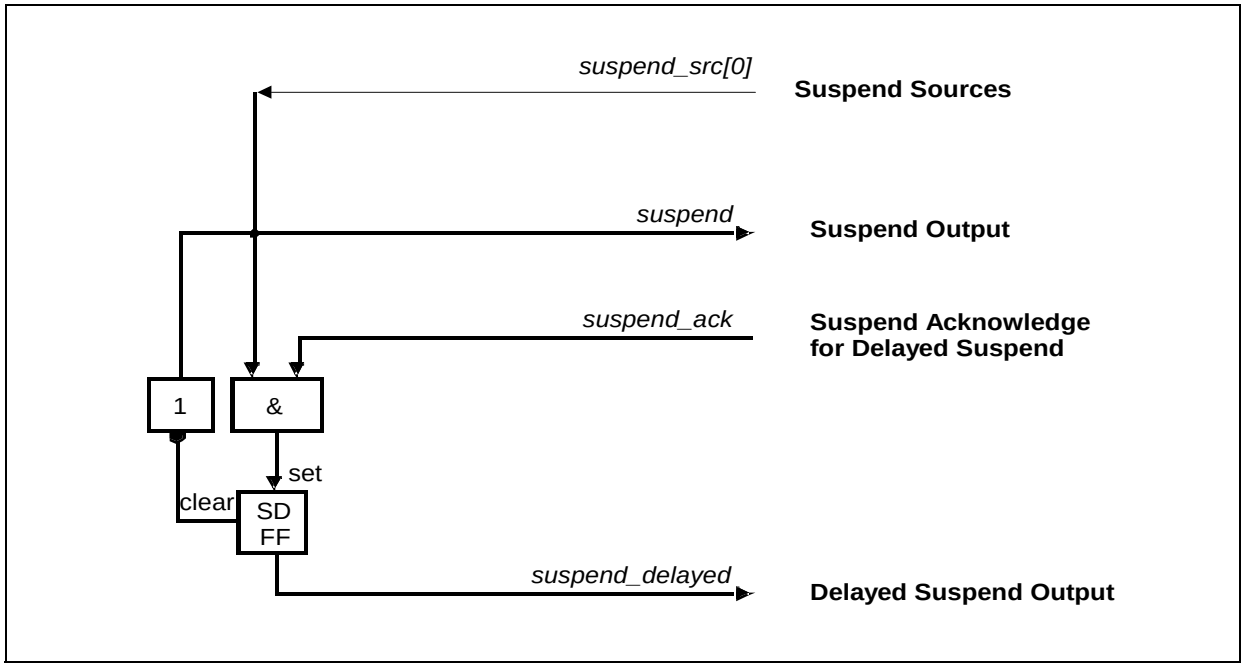


Figure 21-8 Suspend Signal Generation

Suspending Rules

If several bus transaction sources (processor cores) use the same target (peripheral), the sources need to be suspended (broke) prior to or simultaneously with the target. Otherwise transactions may be lost and a restart is not possible anymore. This can be accomplished simply, when the last broken processor core (transaction source) generates the suspend request.

Restart Rules

On a restart, the transaction targets need to restarted prior to or simultaneously with the associated transaction sources. This requirement can be fulfilled only, when there is only one single active suspend request source. In this case this transaction source is started first and with it simultaneously the suspended targets and then the other transaction sources.

The simultaneous restart of source and target can be done through writing with one access the suspend and the restart bit in the OCDS control register.

Delayed Suspend

This feature provides a second delayed conditional suspend signal. It is used in cases, where the rest of the system can be suspended only after certain parts have reached a

certain state after the first level suspend request. A system part with such a requirement can be for instance a bus bridge holding transactions in its FIFO. The transactions need to be finished before the slaves at the lower part of the bus can be suspended.

In general the first level suspend signal *suspend_o* will stop all primary masters from issuing transaction requests. The condition for the second level suspend request is, that all bus bridges have finished their transactions. In this case the system asserts *suspend_ack_i* and the second level suspend signal *suspend_delayed_o* will request all remaining slaves to suspend.

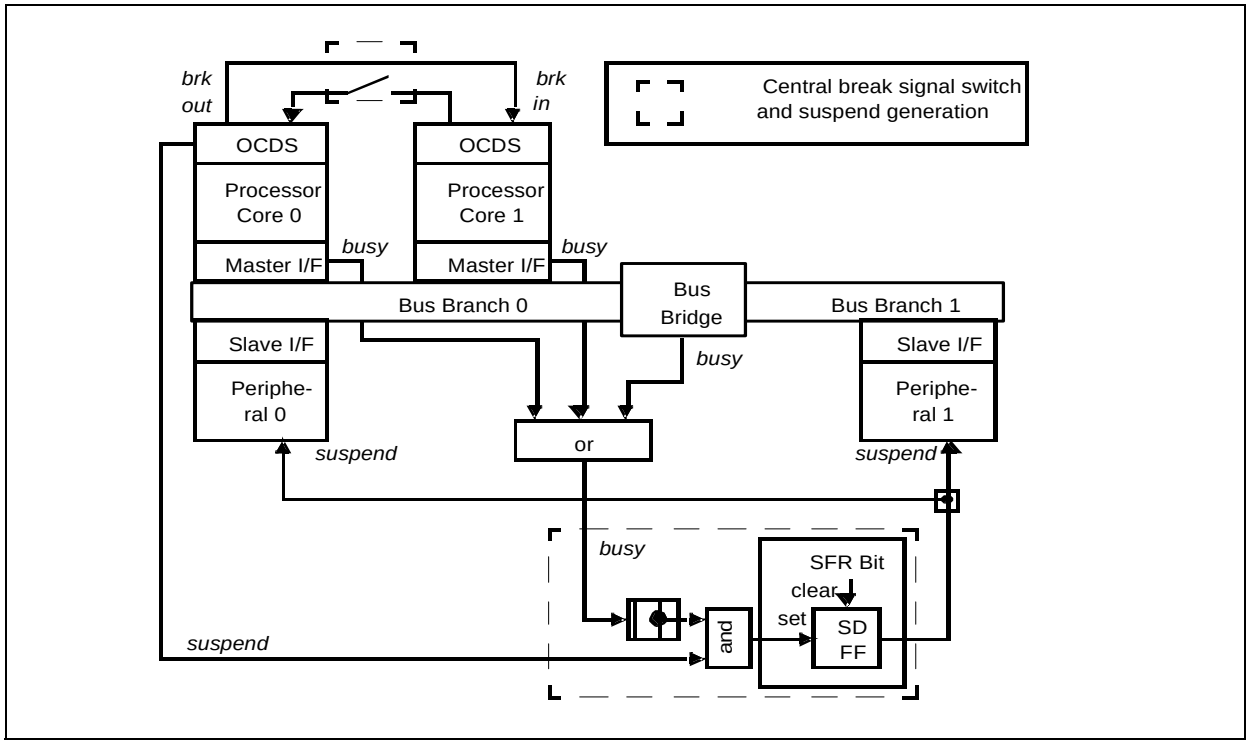


Figure 21-9 Delayed Suspend Request Example

Figure 21-9 shows an example for the delayed suspend request. The goal to halt the system (including peripherals) in a restart capable state, when Processor Core 0 hits a breakpoint. The central break signal switch is programmed that the *brk_out* of Processor Core 0 breaks Processor Core 1. This ensures that both primary masters stop to start bus transactions, thus their *busy* signal will get inactive after their write back buffer is emptied. In the second phase, the Bus Bridge will serve all pending transactions and inactivate its *busy* signal. When all busy signals are inactive the delayed suspend flip-flop SDFF will be set and the suspend request signal for the peripherals gets active.

To restart the system, Processor Core 0 releases the suspend request, then SDFF is cleared and then the Processor Cores are restarted in user mode.

In TC11IB, only Tricore be the primary suspend source, while PCP should do indirectly through *break_out* signals. The target for the first level suspend signal is the FPI masters

on the Fast FPI bus (PCI, Ethernet, XMU-EBU) preventing any new transactions to take place. FFI (FPI bridge) has to complete all transactions from Fast to Slow FPI and issue *suspend_ack* signal. PCI bridge will deassert *pcim_fpi_busy* to indicate the master data path is suspended, and similarly with the slave data path through *pcis_fpi_busy* signal. These three signals form the suspend_ack_i to Cerberus. Second level suspend signal is issued to the rest of the system, but pure-passive slave modules that totally depends on the bus for activities may not need this signal (e.g. LMU). The two masters on the slow FPI bus are indirectly prevented from initiating FPI transactions. PCP has to get the break signal when Tricore breaks (software must ensure), while the other master is Cerberus. During suspend mode any reads to BPI registers are possible, while write accesses may be acknowledged with a bus error.

21.3.2 Registers

Table 21-3 Multi-Core Debug Support Registers

Register Short Name	Register Long Name	Offset Address	Description see
MCDBBS	Multi-Core Debug Break Bus Switch Status and Control Register	0070_H	Page 21-23
MCDSSG	multi-Core Debug Suspend Signal Generation Register	0074_H	Page 21-24

MCDBBS
Break Bus Switch Status and Control **(Reset value: FF00'0000$_H$)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BBS1	BBS0	1	1	1	1	BSS PCP	BSS CPU	0					BTP	0	BSP
rh	rh	rh	rh	rh	rh	rh	rh	r					rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0						BT PCP	BT CPU	0						BS PCP	BS CPU
r						rw	rw	r						rw	rw

Field	Bits	Typ	Description
BSi	[1:0]	rw	**Break source switch** 0 Break source i connected to break bus 0. 1 Connected to break bus 1.
BTi	[9:8]	rw	**Break target switch :** 0 Break target i connected to break bus 0. 1 Connected to break bus 1.
BSP	[16]	rw	**Break source input pins enable:** 0 Break source pin i disabled. 1 Enabled.
BTP	[18]	rw	**Break target output pins enable:** 0 Break target pin i disabled. 1 Enabled.
BSSi	[25:24]	rh	**Status of break source i (*brk_src_n_i[i]*).**
BBSi (i = 0,1)	[31:30]	rh	**Status of break bus i.**
1	[29:26]	r	**Reserved.**

Field	Bits	Typ	Description
0	[23:19] 17 [15:10 [7:2]	r	**Reserved.**

.

MCDSSG
Suspend Signal Generation Status and Control **(Reset value: 0000'0000$_H$)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0						**SDS**	**SOS**	**0**						**SSS 1**	**SSS 0**
r						rh	rh	r						rh	rh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CON 3	**CON 2**	**CON 1**	**CON 0**	**0**											
rw	rw	rw	rw	r											

Field	Bits	Typ	Description
CONi (i = 0-3)	[15:12]	rw	**Controls the associated *mcdssg_con_o[i]* signal.**
SSSi (i = 0-1)	[21:16]	rh	**Status of suspend source i** (*suspend_src_i[i]*).
SOS	24	rh	**Status of the suspend output.**
SDS	25	rh	**Status of the delayed suspend output.**
0	[11:0] [23:18] [31:26]	r	**Reserved.**

*Note: The **CONi (i = 0-3)** bits control the associated mcdssg_con_o[i] signals. They are provided for systems which require for instance additional control of the suspend source. One example is a processor core which does not have a dedicated suspend request output. In this case the break out signal could be used and it is enabled as a suspend source with one of the **CONi (i = 0-3)** bits.*

21.4 Trace Module

This chapter describes the PC trace support implemented in the TC11IB.

21.4.1 Overview

Every cycle, 16 bits of information are sent out about the current state of the CPU core. These bits include the following 3 groups:

- 5 bits of pipeline status information
- An 8-bit indirect PC bus
- 3 bits of breakpoint qualification information

From this information, an emulator can reconstruct a cycle-by-cycle break down of the execution of the CPU. It should be possible to follow in real-time the current PC facilitating advanced tools such as profilers, coverage analysis tools etc. The information may also be captured and used to reconstruct, off-line, a cycle-accurate disassembly of the code being executed within the CPU.

The following sections describe the 3 groups of signals listed above and how they may be used to reconstruct the real time trace.

21.4.2 Pipeline Status Signals

Each cycle, a 5-bit code is sent out over the status signals. The meaning of this code is shown in **Table 21-4** for every cycle except for the first cycle after an indirect branch, when an indirect address sync code is sent (see **Section 21.4.3.1**).

Table 21-4 Pipeline Status Codes

Status	PC increment	Jump	Indirect	Description	Unique[1)]
00000_B	0	no	no	nop	yes
00001_B	2	no	no	–	yes
00010_B	2	yes	no	–	yes
00011_B	2	yes	yes	–	yes
00100_B	0	yes	yes	trap	yes
00101_B	4	no	no	–	yes
00110_B	4	yes	no	–	yes
00111_B	4	yes	yes	–	yes
01000_B	0	yes	yes	interrupt	yes
01001_B	6	no	no	–	yes
01010_B	6	yes	no	–	yes
01011_B	6	yes	yes	–	yes

Table 21-4 Pipeline Status Codes (cont'd)

Status	PC increment	Jump	Indirect	Description	Unique[1)]
01100_B	–	–	–	Reserved: overrun sync pattern	no
01101_B	8	no	no	–	yes
01110_B	8	yes	no	–	yes
01111_B	8	yes	yes	–	yes
10000_B	–	–	–	Reserved	no
10001_B	10	no	no	–	no
10010_B	10	yes	no	–	no
10011_B	–	–	–	Reserved: invalid for core 1	no
10100_B	–	–	–	Reserved	no
10101_B	12	no	no	–	no
10110_B	12	yes	no	–	no
10111_B	–	–	–	Reserved: invalid for core 1	no
11000_B	–	–	–	Reserved	no
11001_B	–	–	–	Reserved	no
11010_B	–	–	–	Reserved	no
11011_B	–	–	–	Reserved	no
11100_B	–	–	–	Reserved	no
11101_B	–	–	–	Reserved	no
11110_B	–	–	–	Reserved	no
11111_B	–	–	–	Reserved	no

1) See **Section 21.4.2.1**.

Quick Decoding of the Pipeline Status Codes

The pipeline status code is split up into two fields:

- Bits [4:2] indicate PC increment
- Bits [1:0] indicate code type

The code type field can have the values shown in **Table 21-5**.

Table 21-5 Code Type

Code Type	Increment PC	Jump	Indirect	Description
00	no	no	–	Special code, trap, interrupt etc.
01	yes	no	–	Sequential code
10	yes	yes	no	Relative branch
11	yes	yes	yes	Indirect branch

For sequential code, the new value of the PC determined from:

```
new_PC = PC + ((PC increment + 1) * 2)
```

21.4.2.1 Synchronizing with the Status and Indirect Streams

Unless the emulator follows the execution of the core from reset, there needs to be a way for the emulator to synchronize with the information coming out of the chip. This process can be performed in two stages.

1. The emulator would synchronize with the pipeline status stream.
2. The emulator would synchronize with the indirect PC stream so that the first PC could be obtained at the next indirect branch.

Pipeline Status Stream

Many of the most common pipeline status codes are unique and no equivalent indirect sync code exists. The unique codes are identified in the last column of **Table 21-4**. By waiting until it sees one of these unique codes, the emulator can synchronize with the pipeline status stream.

Indirect PC Stream

Once the emulator has synchronized with pipeline status stream, it can wait for the first indirect branch. Provided there has not been an overrun, the emulator will then be able to determine the PC, and using that as a starting point it will be able to reconstruct the trace.

21.4.3 Indirect Addresses

The target address of an indirect branch, interrupt, or trap entry are sent out one byte at a time over a dedicated 8-bit bus.

A FIFO is implemented to de-couple the generation of the indirect addresses by the core from the trickling of the addresses out of the chip. The pipeline status and indirect sync encoding has been designed to support a FIFO of up to 4 entries. However, the implementation may have fewer entries. If the FIFO fills up, an indirect address overrun is signalled through a special status code (01100_B).

21.4.3.1 Indirect Sync

The indirect sync is a 5-bit code sent over the status bus after every indirect branch. This code is used to synchronize the status stream to the indirect addresses being sent over the indirect PC bus.

The sync code is interpreted in the following manner:

Table 21-6 Indirect Sync Format

Bits	Name	Description
4	Overrun	Used to determine whether an overrun occurred. 0 Overrun 1 No overrun
[3:0]	Offset	1100_B means overrun occurred Other combinations: number of cycle before first byte of indirect target address will be seen on indirect PC bus.

Overrun

The overrun case occurs when the FIFO fills up that decouples the generation of indirect addresses within the CPU from the ability to transmit those addresses over the 8 bit indirect PC bus. When this scenario arises, the PC of the indirect jump which causes the overrun is lost. This is communicated to the emulator through the indirect PC overrun code 01100_B.

The emulator will then not be able to reconstruct the trace between the time of the indirect jump which caused the overrun, and the next indirect jump that does not also encounter an overrun condition.

The overrun condition should only occur very rarely in normal code. The most common source of indirect branches is when the jump is associated with a return from a function or trap/interrupt handler (RET or RFE). The context model in the first implementation restrict the execution of back-to-back context restores, such that the worst case would be 3 context restores separated by 3 cycles followed by a number of context restores separated by a minimum of 5 cycles. Each context restore generates an indirect PC.

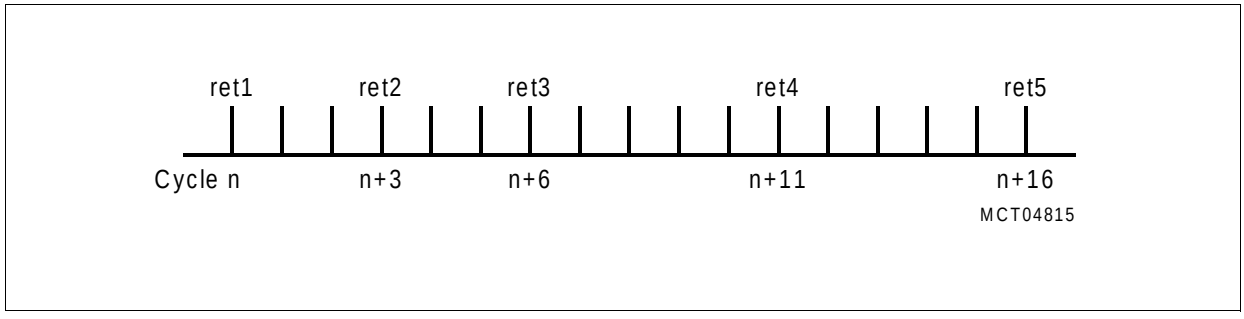


Figure 21-10 Worst Case Back to Back Returns

The FIFO, which is designed to decouple the generation of addresses by the CPU core and the sending out of the indirect PC's over four cycles, can easily handle this scenario. Hence even if the core performs a large number of back to back returns, an overrun would never be generated.

The only scenario that can result in an overrun is several back-to-back jump indirect instructions. This scenario should very seldom be encountered in normal code.

21.4.3.2 Example

```
;a2 contains the address of dest1
;a3 contains the address of dest2
        ji    a2
dest1:  ji    a3

dest2:  add   d9, d8, d7
        ld.w  d5, [a0]24
        ld.w  d6, [a0]28
```

Table 21-7 Trace Example

Cycle	Status					Indirect_pc
	Code	Sync	PC increment	Jump taken	Indirect	
0	–	no	4	yes	yes	X
1	00000_B	yes	–	–	–	t1[7:0]
2	–	no	4	yes	yes	t1[15:8]
3	00010_B	yes	–	–	–	t1[23:16]
4	–	no	8	no	no	t1[31:24]
5	–	no	4	no	no	t2[7:0]

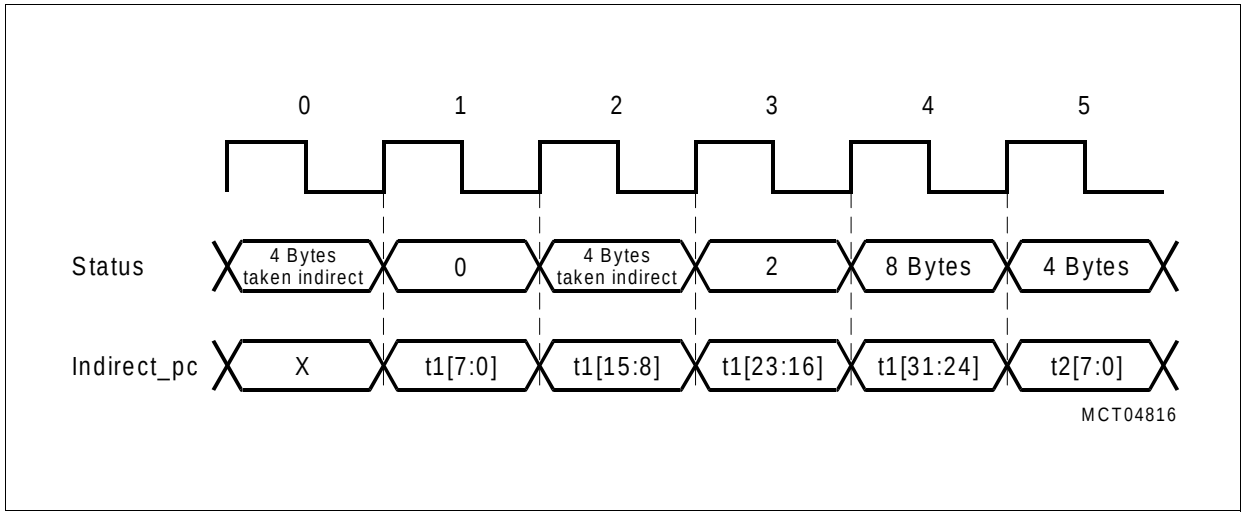


Figure 21-11 Example Output

21.4.3.3 Breakpoint Qualification

The breakpoint qualification lines provide the emulator with a qualification as to which debug events have been raised. The output codes are listed below:

Table 21-8 Breakpoint Qualification Codes

Code	Description
000	reserved
001	EXEVT
010	CREVT
011	SWEVT
100	TR0EVT
101	TR1EVT
110	reserved
111	reserved

The breakpoint qualification information is kept in synchronous with the pipeline status information so that a debug event generated on cycle n will be visible at the external pins at the same time as the pipeline status information associated with cycle n.

21.4.4 Trace Output Control

This part of the SCU controls the interconnections of OCDS2 Port to the trace interfaces of the CPU and PCP (see also **Chapter 4** in this User's Manual).

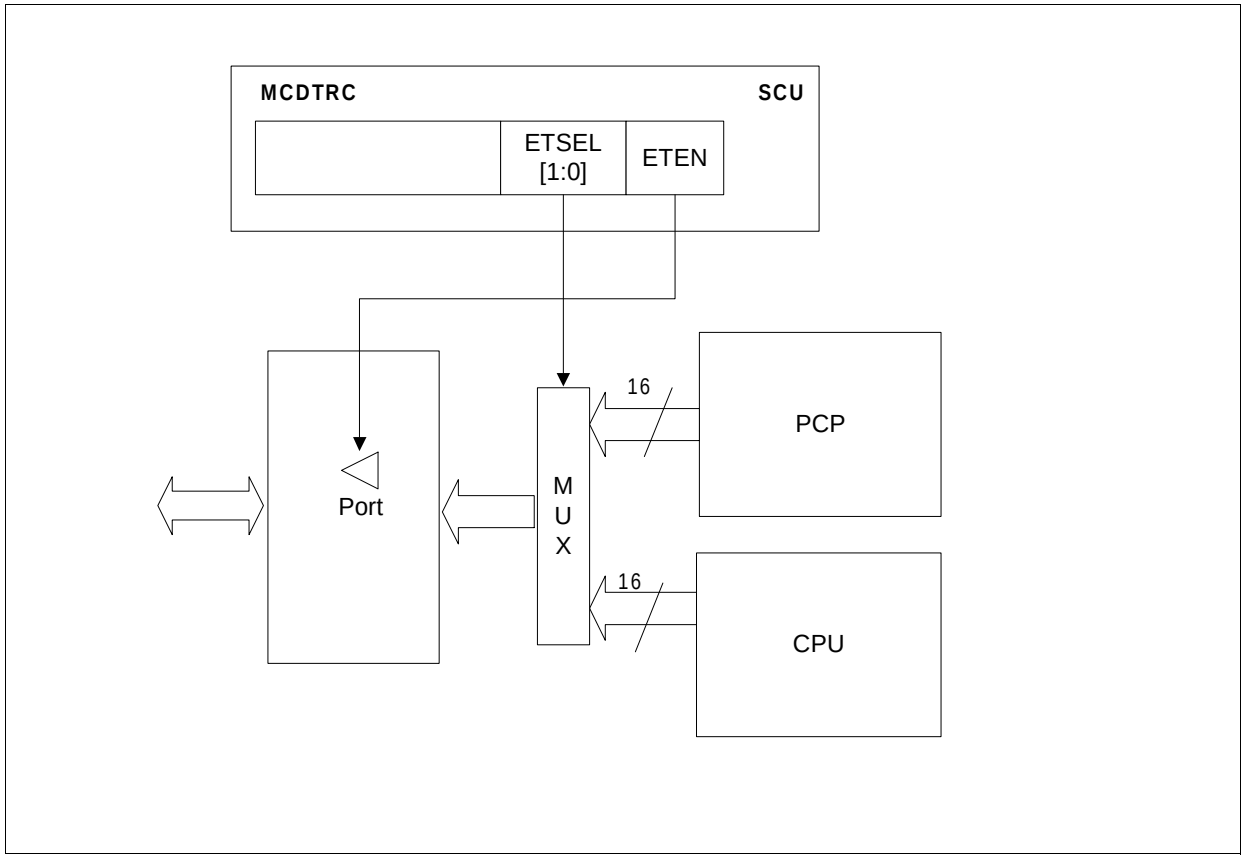


Figure 21-12 Port OCDS2 Trace Control within the SCU

21.5 Debugger Interface (Cerberus)

The Cerberus debug port is provided to debug and emulation tool vendors. The external debug hardware can access the OCDS registers and arbitrary memory locations across the FPI Bus (**Figure 21-1**). The interface is based on the JTAG standard, and uses only the dedicated JTAG port pins.

The description in this section gives a rough overview for the system programmer of the TC11IB on the operations the debugger interface can perform in the system (and thus affect system behavior). It also describes how the Cerberus is disabled to ensure security in the final product. The Cerberus should not be used by an application software since this will disturb the tool behavior. The information of this section is not sufficient to design tools for the TC11IB. For tool developers detailed specifications of Cerberus and the JTAG IO client are necessary.

Features

- External debugger uses the JTAG pins only
- Allows to address the complete FPI Bus address space
- Performance optimized (protocol)
- Does not use any user resources
- Minimum run time impact
- Generic memory read/write functionality
- Writes words, half words and bytes
- Block read and write support
- Full support for communication between monitor and external debugger
- Supports OCDS for several CPUs on the same FPI Bus
- Multi-Core debug switch supports several CPUs with OCDS
- Minimized run tine impact through optional lowest FPI master priority in RW mode
- Full control through optional highest FPI master priority in RW mode
- Pending reads (writes) can be optional triggered from the OCDS module (tracing)

Applications

- Download of programs and data
- Control of the OCDS blocks
- Data acquisition

Performance

The maximum JTAG port clock frequency is 20 MHz. The following performance figures can be achieved:

Table 21-9 Cerberus Performance (Net Data Rates)

Operation	JTAG clock 200 kHz	JTAG clock 10 MHz	JTAG clock 20 MHz
Random read	48 kbps	2.4 Mbps	4.6 Mbps

Table 21-9 Cerberus Performance (Net Data Rates)

Operation	JTAG clock 200 kHz	JTAG clock 10 MHz	JTAG clock 20 MHz
Random write	50 kbps	2.5 Mbps	4.9 Mbps
Block read	104 kbps	5.2 Mbps	10.0 Mbps
Block write	114 kbps	5.7 Mbps	11.2 Mbps

21.5.1 RW Mode

The RW mode is used to read or write memory locations by a JTAG host via the JTAG interface. The RW mode needs the FPI Bus master interface of the Cerberus to actively request data reads or writes.

21.5.1.1 Entering RW Mode

RW mode is entered when the RWDIS bit in the IOSR register is 0 and the JTAG host writes a 1 to the MODE bit of the IOCONF register.

21.5.1.2 Data Type Support

The default data type is a 32-bit word. It is used for single word transfers and block transfers. For reading 16-bit registers without getting an FPI Bus error, the IO_READ_HWORD JTAG instruction is provided. If the JTAG host wants to read a byte, it has to read the associated word or half-word. In all cases, the read value is 32-bit value and the JTAG host has to extract the needed part by itself.

Writes to bytes or half-words are supported with the IO_WRITE_BYTE and IO_WRITE_HWORD JTAG instructions. With this instructions the JTAG host must also shift in the full 32-bit word, but only the selected byte or half-word is actually written. Its position is defined by the lowest 2 (bytes) or the second (half-word) address bit in IOADDR.

21.5.1.3 FPI Bus Master Interface

The FPI Bus master interface executes the actual read or write of memory locations. It is configured by the IOCONF register and the transactions are requested by the JTAG shift core.

FPI Bus Master Priority Control

There are two different requirements for the RW mode access priority:

– The Cerberus is used to configure the OCDS registers in a CPU. Under this conditions, the Cerberus must be able to set these registers immediately.
– The RW mode is used to read registers while a user program is running.

Under these conditions it is important to influence the real time behavior as little as possible. To allow both options, the FPI Bus master priority can be configured with the FPIPRIO bit in the IOCONF register.

FPI Bus Supervisor Mode

For full debugging support, the external debugger needs the option to access memory locations which are only accessible in supervisor mode. This can be configured with the SVMODE bit in the IOCONF register.

21.5.2 Communication Mode

In the communication mode the Cerberus has no access to the FPI Bus, but a communication between an JTAG host and a software monitor, which is embedded in the application program, can be established via the Cerberus registers.

The communication mode is the default mode after reset. If the Cerberus is in RW mode, the communication mode is entered when the JTAG host writes a 0 into the *mode* bit of the IOCONF register.

21.5.3 System Security

After power-on reset, the Cerberus is in communication mode and needs at least 126 t_{CK} clock cycles to bring the system to its control. If the user program running on the CPU sets bit RWDIS immediately after reset, there is no way anymore from outside to set the Cerberus into RW mode via the JTAG interface.

21.5.4 Triggered Transfers

Triggered transfers are an OCDS specific feature of the Cerberus. They can be used to read from or write to a certain memory location, when an OCDS trigger becomes active. Triggered transfers are executed when:

- the Cerberus is in RW mode
- the TRGEN bit in register IOCONF is 1
- the JTAG shift core has requested a transaction
- and a positive edge occurs on the *transfer_trigger* signal (BRKOUT becomes active)

Triggered transfers behave like normal transfers, except that there must be additionally a positive edge on the *transfer_trigger* signal after the JTAG shift core requests the transfer. In the TC11IB, a *trigger_transfer* signal can be generated by the CPU or by the PCP. Another exception is that in case of IO_READ_WORD, IO_READ_HWORD, and IO_READ_BLOCK JTAG instructions the read data is followed by a dirty bit.

21.5.4.1 Tracing of Memory Locations

The main application for triggered transfers is to trace a certain memory location. If a certain memory location is written by a user program, the OCDS module activates a trigger signal. Which trigger signal is selected is defined by the content of the *channel* bit field in register IOCONF. The Cerberus is configured to read the memory location on the occurrence of this trigger signal. The maximum transfer rate that can be reached is defined by:

$$N_{INSTR} = \frac{46}{t_{INSTR} \times f_{JTAG}}$$

N_{INSTR} is the number of instruction cycles that need to be between two CPU accesses to the memory location. t_{INSTR} is the instruction cycle time of the CPU and f_{JTAG} is the clock rate of the JTAG interface (t_{CK}). For example, if t_{INSTR} = 25 ns and f_{JTAG} = 10 MHz, accesses in every 184th instruction cycle can be fully traced. In many cases this will be sufficient to trace for instance the task ID register. The factor 46 is the sum of 32 for the data, 10 for the JTAG state machine, I/O instruction and start bit and 4 for the synchronization between *transfer_trigger* events and the shift out.

It is recommended that triggered transfers are done with the highest FPI Bus master priority and SVMODE = 1, because otherwise another higher priority master could change the desired data value before it is actually read.

21.5.5 Trace with External Bus Address

This is a special operating mode of the master interface for faster tracing. In this mode the data is not shifted out via the JTAG port, but immediately forwarded to an external bus address. The data is then captured from the external bus by the debugger (“trace box”). This kind of tracing can be enabled in communication mode only and can be used in parallel to it.

The condition for transfers is, that MODE = 0, TRGEN = 1, EXBUSTRA = 1 (all are bits in IOCONF) and a positive edge on the *transfer_trigger* signal. With EXBUSHW the FPI Bus access for the source read can be switched between word and half word.

The external bus address is defined by:

31			0
TRADDR	10_H	$F0_H$	68_H

The TRADDR register sets the most significant bits, the rest is hardwired to $10F068_H$. It is recommended that also this kind of triggered transfers are done with the highest FPI Bus master priority and SVMODE = 1.

21.5.6 Reset Behavior

Reset from the JTAG side

The behavior of the registers is specified in **Table 21-10**.

Reset from the bus/CPU side

In this case all instructions except IO_CONFIG are responded with an indefinite number of busy bits (Error state). The external host has to acknowledge this state with the IO_SUPERVISOR instruction. The reason is to notify the external host, that something possibly unexpected has happened and that it has to check for instance the communication channel to the monitor.

Table 21-10 Register reset behavior

Register	JTAG Reset	Bus/CPU Reset	Description
IOADDR	undefined	undefined	Address for next RW mode accesses
IODATA	undefined	undefined	Data register for RW mode
COMDATA	unchanged	0	Data register for communication mode
IOCONF	client specific	client specific	IO client configuration register
IOSR	client specific	client specific	IO client status register

21.5.7 Power Saving

The Cerberus is in power saving mode when it is not selected from the JTAG side. The only register that is always accessible and working is IOSR.

21.5.8 Registers

Table 21-11 Register Summary

Register	Size	Address	Description
IOCONF	12 Bits	Accessible via JTAG only	Configuration register
IOADDR	32 Bits	Accessible via JTAG only	Address for next RW mode accesses
IODATA	32 Bits	Accessible via JTAG only	RW mode data register
COMDATA	32 Bits	F000 0468_H	Communication mode data register

Table 21-11 Register Summary

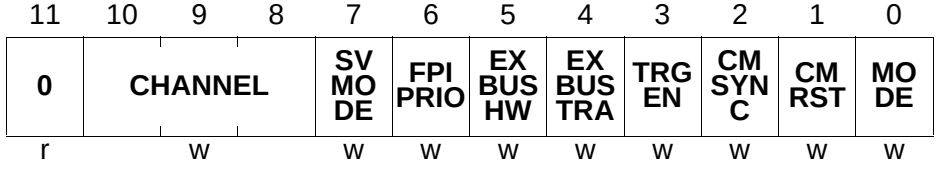
Register	Size	Address	Description
IOSR	32 Bits	F000 046C$_H$	Status register
TRADDR	8 Bits	Accessible via JTAG only	External bus trace mode address

Note: The Cerberus has fixed absolute register addresses. This makes a debug monitor independent of a TriCore product.

21.5.8.1 IOCONF Register

The IOCONF register is used to configure the Cerberus.The IOCONF register is a write-only register for the JTAG host and not accessible from the FPI Bus side.

IOCONF
Cerberus I/O Configuration Register **Reset Value: 0000_H**



Field	Bits	Type	Description
MODE	0	w	**Mode Selection** This bit defines whether the Cerberus is in RW mode or in communication mode. 0 Communication mode selected 1 RW mode selected
CMRST	1	w	**Communication Mode Bit Reset** This bit is provided to reset the CWSYNC and HWSYNC bits in register IOSR to abort requests in communication mode. This reset is not static, it is only done once when the IOCONF register is updated. 0 Bits CWSYNC and HWSYNC are not affected 1 Bits CWSYNC and HWSYNC in register IOSR are reset
CMSYNC	2	w	**Communication Mode Bit Set** This bit sets the bit COMSYNC in register IOSR. 0 Bit COMSYNC in register IOSR is not affected 1 Bit COMSYNC in register IOSR is set
TRGEN	3	w	**Triggered Transfer Enable** This bit enables triggered transfers in RW mode. 0 Triggered transfers in RW mode are disabled 1 The next RW mode transfers must be triggered by a *transfer_trigger* signal
EXBUSTRA	4	w	**Enable Triggered Transfers to External Bus Address** This bit enables triggered transfers to an external bus address. 0 Trace with external bus address disabled 1 Trace with external bus address enabled

Field	Bits	Type	Description
EXBUSHW	5	w	**FPI Bus Read Size Selection** This bit distinguishes between FPI Bus word and half word reads of the trace source for the external bus trace. 0 The trace source is read with an FPI Bus word access 1 The trace source is read with an FPI Bus half word access
FPIPRIO	6	w	**FPI Bus Master Priority of the Cerberus** This bit sets the priority of the Cerberus FPI Bus master interface in RW mode. 0 Next FPI Bus master request is done with lowest priority 1 Next FPI Bus master request is done with highest priority
SVMODE	7	w	**Supervisor Mode Selection** This bit sets the supervisor mode for the FPI Bus master interface in RW mode. 0 The next RW transfers are not done in supervisor mode 1 The next RW transfers are done in FPI Bus supervisor mode
CHANNEL	[10:8]	w	**Transfer Trigger Selection** This bit field sets the associated bit field in register IOSR and selects whether CPU or PCP can activate the *transfer_trigger* signal. 001_B CPU is selected to activate the *transfer_trigger* signal 010_B PCP is selected to activate the *transfer_trigger* signal All other combinations are reserved and must not be used.
0	11	r	**Reserved**; should be written with 0.

21.5.8.2 IOSR Register

The IOSR register is used in communication mode. It allows to disable the Cerberus from the CPU side. The IOSR register is only accessible from the FPI Bus.

IOSR
Status and Control Register **(Reset value: 0000'0000$_H$)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	CHNL			0		CLT ON	DBG ON	COM SY NC	CW ACK	CW SY NC	CR SY NC	RW EN P	RW EN	RW DIS P	RW DIS
r	rh			r		rh	rh	rh	w	rh	rh	w	rw(h)	w	r(w)

Field	Bits	Typ	Description
RWDIS	0	rw 1)2)	**RW mode protection:** 0 RW mode is enabled. 1 RW mode is disabled.
RWDISP	1	w	0 Bit protection: RWDIS unchanged. 1 RWDIS will be changed.
RWEN	2	rw(h) 3)	Used by user program for security. Reset by a JTAG reset (h) or a power on reset (optional) only and not by a CPU reset.
RWENP	3	w	0 Bit protection: RWEN unchanged. 1 RWEN will be changed.
CRSYNC	4	rh	Read sync bit for Communication Mode. 0 No receive request pending. 1 External debugger requests value (COMDATA).
CWSYNC	5	rh	**Write sync bit for Communication Mode.** 0 No send request pending. 1 External debugger offers value (COMDATA).
CWACK	6	w	**Write request acknowledge in Communication Mode.** 0 No action. 1 Acknowledge that send value was read from COMDATA by the monitor.

Field	Bits	Typ	Description
COMSYNC	7	rh	**High level sync bit for Communication Mode.**
DBGON	8	rh	0 No external debugger present. 1 External debugger present.
CLNTON	9	rh	0 Client not selected. 1 Client selected.
CHNL	[14:12]	rh	Selects the addressed CPU (monitor) in Communication Mode.
0	[11:10] [31:15	r	**Reserved.**

1) Write only in communication mode.

2) Write only in FPI supervisor mode.

3) Reset by a JTAG reset only and not by a FPI Bus reset.

21.5.8.3 TRADDR Register

The TRADDR register is used for tracing with external bus address. It defines the uppermost 8 bits of the external bus address. It is set with the IO_SET_TRADDR instruction by the external JTAG host.

21.5.8.4 IOADDR, COMDATA and RWDATA Registers

These registers are used as address, data, and control registers in communication and RW mode.

21.6 OCDS Register Address Ranges

In the TC11IB, the registers for on-chip OCDS control are located at the following address ranges:

- Core Debug Registers (except SBSRC0)
  Module Base Address: F7E1 FD00$_H$
  Module End Address: F7E1 FDFF$_H$
- CPU Slave Registers (SBSRC only)
  Module Base Address: F7E0 FF00$_H$
  Module End Address: F7E0 FFFF$_H$
- On-Chip Debug Support (Cerberus) Registers
  Module Base Address: F000 0400$_H$
  Module End Address: F000 04FF$_H$
- Absolute Register Address = Module Base Address + Offset Address
  (offset addresses see **Table 21-1**)

22 Register Overview

This chapter defines all registers of the TC11IB and provides the complete address range as well. It also defines the read/write access rights of the specific address ranges/ registers.

Throughout the tables in this chapter, the “Access Mode” “Read” and “Write”, and “Reset Values” columns indicate access rights and values using symbols listed in **Section 22-1**.

Table 22-1 Address Map Symbols

Symbol	Description
U	Access permitted in User Mode 0 or 1.
SV	Access permitted in Supervisor Mode.
R	Read-only register.
32	Only 32-bit word accesses are permitted to that register/address range.
E	Endinit protected register/address.
PW	Password protected register/address.
NC	No change, indicated register is not changed.
BE	Indicates that an access to this address range generates a Bus Error.
nBE	Indicates that no Bus Error is generated when accessing this address range, even though it is either an access to an undefined address or the access does not follow the given rules.
nE	Indicates that no Error is generated when accessing this address or address range, even though the access is to an undefined address or address range. True for CPU accesses (MTCR/MFCR) to undefined addresses in the CSFR range.
X	Undefined value or bit.

22.1 Segments 0 - 14

22.1.1 Address Map

Table 22-2 shows the block address map of Segment 0 to 14.

Note: Bold entries in column "Access Mode" are links to register definitions of the corresponding functional unit.

Table 22-2 Block Address Map of Segments 0 to 14

Segment	Description	Address Range	Access Mode		Size
			Read	Write	
0-7	MMU(Virtual Address)/ PCI (Physical Address) Space	$0000\ 0000_H$ - $7FFF\ FFFF_H$	nE (via F_FPI)	nE (via F_FPI)	2 GBytes
8	EBU Space (external), mapped from segment 10	$8000\ 0000_H$ - $8FFF\ FFFF_H$	nE (via LMB)	nE (via LMB)	256 MBytes
9	PCI Space, mapped from Segment 11	$9000\ 0000_H$ - $9FDF\ FFFF_H$	nE (via F_FPI)	nE (via F_FPI)	254 MBytes
	Com-DRAM, mapped from Segment 11	$9FE0\ 0000_H$ - $9FEF\ FFFF_H$	nE (via F_FPI)	nE (via F_FPI)	1 MBytes
	Reserved	$9FF0\ 0000_H$ - $9FFF\ FFFF_H$	BE	BE	1 MBytes
10	EBU Space	$A000\ 0000_H$ - $AFBF\ FFFF_H$	nE (via LMB)	nE (via LMB)	252 MBytes
	LMU non-cached executable space	$AFC0\ 0000_H$ - $AFC7\ FFFF_H$	nE (via LMB)	nE (via LMB)	512 KBytes
	Reserved	$AFC8\ 0000_H$ - $AFFF\ FFFF_H$	nE (via LMB)	nE (via LMB)	3.5 MBytes

Table 22-2 Block Address Map of Segments 0 to 14 (cont'd)

Segment	Description	Address Range	Access Mode		Size
			Read	Write	
11	PCI Space	B000 0000$_H$ - BFDF FFFF$_H$	nE (via F_FPI)	nE (via F_FPI)	254 MBytes
	Com-DRAM Space	BFE0 0000$_H$ - BFEF FFFF$_H$	nE (via F_FPI)	nE (via F_FPI)	1 MBytes
	Reserved	BF00 0000$_H$ - BFFF EFFF$_H$	BE	BE	1 MBytes
12	Local Memory, eDRAM	C000 0000$_H$ - C007 FFFF$_H$	nE (via LMB)	nE (via LMB)	512 KByte
	Reserved	C008 0000$_H$ - CFFF FFFF$_H$	BE	BE	255.5 MBytes
13	Data Scratchpad Memory (SRAM)	D000 0000$_H$ - D000 5FFF$_H$	nE (via LMB)	nE (via LMB)	24 KBytes
	Reserved	D000 6000$_H$ - D3FF FFFF$_H$	BE	BE	~64 MBytes
	Code Scratchpad Memory (SRAM)	D400 0000$_H$ - D400 5FFF$_H$	nE (via LMB)	nE (via LMB)	24 KBytes
	Reserved	D400 6000$_H$ - D7FF FFFF$_H$	BE	BE	~64 MBytes
	External Peripheral Space	D800 0000$_H$ - DDFF FFFF$_H$	nE (via LMB)	nE (via LMB)	96 MBytes
	External Emulator Space	DE00 0000$_H$ - DEFF FFFF$_H$	nE (via LMB)	nE (via LMB)	16 MBytes
	Reserved	DF00 0000$_H$ - DFFF BFFF$_H$	BE	BE	~16 MBytes
	Boot ROM Space	DFFF C000$_H$ - DFFF FFFF$_H$	nE (via S_FPI)	nE (via S_FPI)	16 KBytes

Table 22-2 Block Address Map of Segments 0 to 14 (cont'd)

Segment	Description	Address Range	Access Mode		Size
			Read	Write	
14	External Peripheral Space	E000 0000$_H$ - E7FF FFFF$_H$	nE (via LMB)	nE (via LMB)	128 MBytes
	Local Memory	E800 0000$_H$ - E807 FFFF$_H$	nE (via F_FPI)	nE (via F_FPI)	512 KBytes
	Reserved	E808 0000$_H$ - E83F FFFF$_H$	nBE (via F_FPI)	nBE (via F_FPI)	3.5 MBytes
	Data Scratchpad SRAM	E840 0000$_H$ - E840 5FFF$_H$	nE (via F_FPI)	nE (via F_FPI)	24 KBytes
	Reserved	E840 6000$_H$ - E84F FFFF$_H$	nBE (via F_FPI)	nBE (via F_FPI)	~1 MBytes
	Code Scratchpad SRAM	E850 0000$_H$ - E850 5FFF$_H$	nE (via F_FPI)	nE (via F_FPI)	24 KBytes
	Reserved	E850 6000$_H$ - E85F FFFF$_H$	nBE (via F_FPI)	nBE (via F_FPI)	~1 MBytes
	Reserved	E860 0000$_H$ - EFFF FFFF$_H$	BE	BE	122 MBytes

22.2 Segment 15 (Peripheral Units)

22.2.1 Address Map

Table 22-3 and **Table 22-4** show the memory map and registers of Segment 15.

Note: Bold entries in column "Access Mode" are links to the register definitions of the corresponding functional unit.

Table 22-3 Block Address Map of Segment 15

Unit	Address Range	Access Mode		Size
		Read	Write	
System Control Unit (SCU) and Watchdog Timer (WDT)	$F000\ 0000_H$ - $F000\ 00FF_H$	see **Page 22-9**		256 Bytes
PCI Software Interrupt Request Register Space	$F000\ 0100_H$ - $F000\ 01FF_H$	see **Page 22-10**		256 Bytes
Bus Control Unit 1 (BCU1)	$F000\ 0200_H$ - $F000\ 02FF_H$	see **Page 22-13**		256 Bytes
System Timer (STM)	$F000\ 0300_H$ - $F000\ 03FF_H$	see **Page 22-13**		256 Bytes
On-Chip Debug Support (Cerberus)	$F000\ 0400_H$ - $F000\ 04FF_H$	see **Page 22-14**		256 Bytes
Reserved	$F000\ 0500_H$ - $F000\ 05FF_H$	BE	BE	–
General Purpose Timer Unit 0 (GPTU0)	$F000\ 0600_H$ - $F000\ 06FF_H$	see **Page 22-14**		256 Bytes
General Purpose Timer Unit 1(GPTU1)	$F000\ 0700_H$ - $F000\ 07FF_H$	see **Page 22-17**		256 Bytes
Asynchronous/Synchronous Serial Interface (ASC)	$F000\ 0800_H$ - $F000\ 08FF_H$	see **Page 22-19**		256 Bytes
Asynchronous Serial Interface (16X50)	$F000\ 0900_H$ - $F000\text{-}09FF_H$	see **Page 22-20**		256 Bytes
High-Speed Synchronous Serial Interface (SSC)	$F000\ 0A00_H$ - $F000\text{-}0AFF_H$	see **Page 22-22**		256 Bytes
MultiMediaCard Interface (MMCI)	$F000\ 0B00_H$ - $F000\text{-}0BFF_H$	see **Page 22-22**		256 Bytes
Service Request Control (SRC)	$F000\ 0C00_H$ - $F000\text{-}0DFF_H$	see **Page 22-22**		2×256 Bytes

Table 22-3 Block Address Map of Segment 15 (cont'd)

Unit		Address Range	Access Mode		Size
			Read	Write	
Reserved		$F000\ 0E00_H$ - $F000\ 27FF_H$	BE	BE	–
Port 0		$F000\ 2800_H$ - $F000\ 28FF_H$	see **Page 22-28**		256 Bytes
Port 1		$F000\ 2900_H$ - $F000\ 29FF_H$	see **Page 22-29**		256 Bytes
Port 2		$F000\ 2A00_H$ - $F000\ 2AFF_H$	see **Page 22-29**		256 Bytes
Port 3		$F000\ 2B00_H$ - $F000\ 2BFF_H$	see **Page 22-30**		256 Bytes
Port 4		$F000\ 2C00_H$ - $F000\ 2CFF_H$	see **Page 22-30**		256 Bytes
Port 5		$F000\ 2D00_H$ - $F000\ 2DFF_H$	see **Page 22-31**		256 Bytes
Port 6 & 7 (no registers available)		$F000\ 2E00_H$ - $F000\ 3EFF_H$	BE	BE	2×256 Byte
PCP	Peripheral Control Processor (PCP)	$F000\ 3F00_H$ - $F000\ 3FFF_H$	see **Page 22-31**		256 Bytes
	Reserved	$F000\ 4000_H$ - $F000\ FFFF_H$	BE	BE	–
	PCP Data Memory (PRAM)	$F001\ 0000_H$ - $F001\ 0FFF_H$	nE	nE	4 KBytes
	Reserved	$F001\ 1000_H$ - $F001\ FFFF_H$	BE	BE	–
	PCP Code Memory	$F002\ 0000_H$ $F002\ 3FFF_H$	nE	nE	16 KBytes
Reserved		$F002\ 4000_H$ - $F017\ FFFF_H$	BE	BE	–
ComDRAM		$F018\ 0000_H$ - $F018\ FFFF_H$	see **Page 22-33**		64 KBytes
Reserved		$F019\ 0000_H$ - $F03F\ FFFF_H$	BE	BE	–

Table 22-3 Block Address Map of Segment 15 (cont'd)

Unit	Address Range	Access Mode		Size
		Read	Write	
PCI Interface (PCI)	$F040\ 0000_H$ - $F040\ 00FF_H$	see **Page 22-33**		256 Bytes
Reserved	$F040\ 0100_H$ - $F0FF\ FFFF_H$	BE	BE	–
PCI Configuration Space 1(PCI_CS1)	$F100\ 0000_H$ - $F100\ 00FF_H$	see **Page 22-37**		256 Bytes
PCI Configuration Space 2(PCI_CS2)	$F100\ 0100_H$ - $F100\ 01FF_H$	see **Page 22-38**		256 Bytes
Reserved	$F100\ 0200_H$ - $F1FF\ FFFF_H$	BE	BE	–
Bus Control Unit 0 (BCU0)	$F200\ 0000_H$ - $F200\ 00FF_H$	see **Page 22-39**		256 Bytes
Ethernet Controller Module (Ethernet)	$F200\ 0100_H$ - $F200\ 05FF_H$	see **Page 22-40**		5×256 Bytes
Reserved	$F200\ 0600_H$ - $F7E0\ FEFF_H$	BE	BE	–

Table 22-3 Block Address Map of Segment 15 (cont'd)

Unit		Address Range	Access Mode		Size
			Read	Write	
CPU	CPU Slave Interface Registers (CPS)	$F7E0\ FF00_H$ - $F7E0\ FFFF_H$	see Page 22-48		256 Bytes
	Reserved	$F7E1\ 0000_H$ - $F7E1\ 7FFF_H$	nE	nE	–
	Memory Management Unit (MMU)	$F7E1\ 8000_H$ - $F7E1\ 80FF_H$	see Page 22-49		48 × 256 Bytes
	Reserved	$F7E1\ 8100_H$ - $F7E1\ BFFF_H$	nE	nE	–
	Memory Protection Register	$F7E1\ C000_H$ - $F7E1\ EFFF_H$	see Page 22-49		48 × 256 Bytes
	Reserved	$F7E1\ F000_H$ - $F7E1\ FCFF_H$	nE	nE	–
	Core Debug Register (OCDS)	$F7E1\ FD00_H$ - $F7E1\ FDFF_H$	see Page 22-52		256 Bytes
	Core Special Function Registers (CSFR)	$F7E1\ FE00_H$ - $F7E1\ FEFF_H$	see Page 22-52		256 Bytes
	General Purpose Register (GPR)	$F7E1\ FF00_H$ - $F7E1\ FFFF_H$	see Page 22-53		256 Bytes
Reserved		$F7E2\ 0000_H$ - $F7FF\ FFFF_H$	BE	BE	–
External Bus Unit (EBU)		$F800\ 0000_H$ - $F800\ 02FF_H$	see Page 22-44		3 × 256 Bytes
Reserved		$F800\ 0300_H$ - $F800\ 03FF_H$	BE	BE	–
Local Memory Unit (LMU)		$F800\ 0400_H$ - $F800\ 04FF_H$	see Page 22-48		256 Bytes
Reserved		$F800\ 0500_H$ - $F87F\ FBFF_H$	BE	BE	–
Data Memory Unit (DMU)		$F87F\ FC00_H$ - $F87F\ FCFF_H$	see Page 22-55		256 Bytes
Program Memory Unit (LMU)		$F87F\ FD00_H$ - $F87F\ FDFF_H$	see Page 22-55		256 Bytes

Table 22-3 Block Address Map of Segment 15 (cont'd)

Unit	Address Range	Access Mode		Size
		Read	Write	
LMB Bus Control Unit (LCU)	$F87F\ FE00_H$ - $F87F\ FEFF_H$	see Page 22-56		256 Bytes
LMB to FPI Bus Bridge (LFI)	$F87F\ FF00_H$ - $F87F\ FFFF_H$	see Page 22-56		256 Bytes
Reserved	$F880\ 0000_H$ - $FFFF\ FFFF_H$	BE	BE	–

22.2.2 Registers

Table 22-4 shows the address map with all register of Segment 15.

Note: Addresses listed in column "Address" of Table 22-4 are word (32-bit) addresses.

Table 22-4 Detailed Address Map of Segment 15

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
System Control Unit (SCU) with Watchdog Timer (WDT)					
–	Reserved	$F000\ 0000_H$-$F000\ 0004_H$	BE	BE	–
SCU_ID	SCU Module Identification Register	$F000\ 0008_H$	U, SV	BE	$XXXXXXXX_H$
–	Reserved	$F000\ 000C_H$	BE	BE	–
RST_REQ	Reset Request Register	$F000\ 0010_H$	U, SV	U, SV, E	$0000\ 0000_H$
RST_SR	Reset Status Register	$F000\ 0014_H$	U, SV	–	according boot cfg.
–	Reserved	$F000\ 0018_H$-$F000\ 001C_H$	BE	BE	–
WDT_CON0	Watchdog Timer Control Register 0	$F000\ 0020_H$	U, SV	U, SV, PW	$FFFC\ 0002_H$
WDT_CON1	Watchdog Timer Control Register 1	$F000\ 0024_H$	U, SV	U, SV, E	$0000\ 0000_H$
WDT_SR	Watchdog Timer Status Register	$F000\ 0028_H$	U, SV	U, SV, NC	$FFFC\ XX0X_H$
NMISR	NMI Status Register	$F000\ 002C_H$	U, SV	U, SV	$0000\ 0000_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PMG_CON	Power Management Control Register	F000 0030$_H$	U, SV	U, SV, E	0000 0001$_H$
PMG_CSR	Power Management Control and Status Reg.	F000 0034$_H$	U, SV	U, SV	0000 0100$_H$
–	Reserved	F000 0038$_H$-F000 003C$_H$	BE	BE	–
PLL_CLC	PLL Clock Control Reg.	F000 0040$_H$	U, SV	U, SV, E	000F 01XX$_H$
–	Reserved	F000 0044$_H$	nBE	nBE	–
GS_CON	General System Control Register	F000 0048$_H$	U, SV	SV, E	0000 0000$_H$
–	Reserved	F000 004C$_H$	BE	BE	–
MCDTRC	Trace Control Register	F000 0050$_H$	U, SV	SV	0000 0000$_H$
–	Reserved	F000 0054$_H$	BE	BE	–
FFI_CON	FFI Bridge Control Reg.	F000 0058$_H$	U, SV	SV, E	0000 0404$_H$
–	Reserved	F000 005C$_H$-F000 006C$_H$	BE	BE	–
MANID	Manufacturer Identification Register	F000 0070$_H$	U, SV	BE	XXXX XXXX$_H$
CHIPID	Chip Identification Reg.	F000 0074$_H$	U, SV	BE	XXXX XXXX$_H$
RTID	Redesign Tracing Identification Register	F000 0078$_H$	U, SV	BE	XXXX XXXX$_H$
–	Reserved	F000 007C$_H$-F000 00FC$_H$	BE	BE	–
PCI Software Interrupt Request Register					
–	Reserved	F000 0100$_H$-F000 017C$_H$	BE	BE	–
PCI_SW_IRQ0	PCI Software Interrupt Request 0 Register	F000 0180$_H$	–	SV	–
PCI_SW_IRQ1	PCI Software Interrupt Request 1Register	F000 0184$_H$	–	SV	–
PCI_SW_IRQ2	PCI Software Interrupt Request 2 Register	F000 0188$_H$	–	SV	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_SW_IRQ3	PCI Software Interrupt Request 3 Register	F000 018C$_H$	–	SV	–
PCI_SW_IRQ4	PCI Software Interrupt Request 4 Register	F000 0190$_H$	–	SV	–
PCI_SW_IRQ5	PCI Software Interrupt Request 5 Register	F000 0194$_H$	–	SV	–
PCI_SW_IRQ6	PCI Software Interrupt Request 6 Register	F000 0198$_H$	–	SV	–
PCI_SW_IRQ7	PCI Software Interrupt Request 7 Register	F000 019C$_H$	–	SV	–
PCI_SW_IRQ8	PCI Software Interrupt Request 8 Register	F000 01A0$_H$	–	SV	–
PCI_SW_IRQ9	PCI Software Interrupt Request 9 Register	F000 01A4$_H$	–	SV	–
PCI_SW_IRQ10	PCI Software Interrupt Request 10 Register	F000 01A8$_H$	–	SV	–
PCI_SW_IRQ11	PCI Software Interrupt Request 11 Register	F000 01AC$_H$	–	SV	–
PCI_SW_IRQ12	PCI Software Interrupt Request 12 Register	F000 01B0$_H$	–	SV	–
PCI_SW_IRQ13	PCI Software Interrupt Request 13 Register	F000 01B4$_H$	–	SV	–
PCI_SW_IRQ14	PCI Software Interrupt Request 14 Register	F000 01B8$_H$	–	SV	–
PCI_SW_IRQ15	PCI Software Interrupt Request 15 Register	F000 01BC$_H$	–	SV	–
PCI_SW_IRQ16	PCI Software Interrupt Request 16 Register	F000 01C0$_H$	–	SV	–
PCI_SW_IRQ17	PCI Software Interrupt Request 17 Register	F000 01C4$_H$	–	SV	–
PCI_SW_IRQ18	PCI Software Interrupt Request 18 Register	F000 01C8$_H$	–	SV	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_SW_IRQ19	PCI Software Interrupt Request 19 Register	F000 01CC$_H$	–	SV	–
PCI_SW_IRQ20	PCI Software Interrupt Request 20 Register	F000 01D0$_H$	–	SV	–
PCI_SW_IRQ21	PCI Software Interrupt Request 21 Register	F000 01D4$_H$	–	SV	–
PCI_SW_IRQ22	PCI Software Interrupt Request 22 Register	F000 01D8$_H$	–	SV	–
PCI_SW_IRQ23	PCI Software Interrupt Request 23 Register	F000 01DC$_H$	–	SV	–
PCI_SW_IRQ24	PCI Software Interrupt Request 24 Register	F000 01E0$_H$	–	SV	–
PCI_SW_IRQ25	PCI Software Interrupt Request 25 Register	F000 01E4$_H$	–	SV	–
PCI_SW_IRQ26	PCI Software Interrupt Request 26 Register	F000 01E8$_H$	–	SV	–
PCI_SW_IRQ27	PCI Software Interrupt Request 27 Register	F000 01EC$_H$	–	SV	–
PCI_SW_IRQ28	PCI Software Interrupt Request 28 Register	F000 01F0$_H$	–	SV	–
PCI_SW_IRQ29	PCI Software Interrupt Request 29 Register	F000 01F4$_H$	–	SV	–
PCI_SW_IRQ30	PCI Software Interrupt Request 30 Register	F000 01F8$_H$	–	SV	–
PCI_SW_IRQ31	PCI Software Interrupt Request 31 Register	F000 01FC$_H$	–	SV	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
Bus Control Unit 1(BCU1)					
–	Reserved	F000 0200$_H$ - F000 0204$_H$	BE	BE	–
BCU1_ID	BCU1 Module Identification Register	F000 0208$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 020C$_H$	BE	BE	–
BCU1_CON	BCU1 Control Register	F000 0210$_H$	U, SV, 32	U, SV, 32	4009 FFFF$_H$
–	Reserved	F000 0214$_H$- F000 021C$_H$	BE	BE	–
BCU1_ ECON	BCU1 Error Control Capture Register	F000 0220$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
BCU1_ EADD	BCU1 Error Address Capture Register	F000 0224$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
BCU1_ EDAT	BCU1 Error Data Capture Register	F000 0228$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
–	Reserved	F000 022C$_H$- F000 02F8$_H$	BE	BE	–
BCU1_SRC	BCU1 Service Request Control Register	F000 02FC$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
System Timer (STM)					
STM_CLC	STM Clock Control Reg.	F000 0300$_H$	U, SV	U, SV, E	0000 0000$_H$
–	Reserved	F000 0304$_H$	BE	BE	–
STM_ID	STM Module Identification Register	F000 0308$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 030C$_H$	BE	BE	–
STM_TIM0	STM Timer Register 0	F000 0310$_H$	U, SV	U, SV	0000 0000$_H$
STM_TIM1	STM Timer Register 1	F000 0314$_H$	U, SV	U, SV	0000 0000$_H$
STM_TIM2	STM Timer Register 2	F000 0318$_H$	U, SV	U, SV	0000 0000$_H$
STM_TIM3	STM Timer Register 3	F000 031C$_H$	U, SV	U, SV	0000 0000$_H$
STM_TIM4	STM Timer Register 4	F000 0320$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
STM_TIM5	STM Timer Register 5	F000 0324$_H$	U, SV	U, SV	0000 0000$_H$
STM_TIM6	STM Timer Register 6	F000 0328$_H$	U, SV	U, SV	0000 0000$_H$
STM_CAP	STM Timer Capture Reg.	F000 032C$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0330$_H$-F000 03FC$_H$	BE	BE	–
On-Chip Debug Support (Cerberus)					
–	Reserved	F000 0400$_H$-F000 0404$_H$	BE	BE	–
JPD_ID	JTAG/OCDS Module Identification Register	F000 0408$_H$	U, SV	BE	XXXX XXXX$_H$
–	Reserved	F000 040C$_H$-F000 0464$_H$	BE	BE	–
COMDATA	Cerberus Communication Mode Data Register	F000 0468$_H$	SV	SV	0000 0000$_H$
IOSR	Cerberus Status Register	F000 046C$_H$	SV	SV	0000 0000$_H$
MCDBBS	Multi-Core Debug Break Bus Switch Status and Control Register	F000 0470$_H$	U,SV	U,SV	FF00 0000$_H$
MCDSSG	Multi-Core Debug Suspend Signal Generation Register	F000 0474$_H$	U,SV	U,SV	0000 0000$_H$
–	Reserved	F000 0478$_H$-F000 04FC$_H$	BE	BE	–
General Purpose Timer Unit 0 (GPTU0)					
GPTU0_CLC	GPTU0 Clock Control Reg.	F000 0600$_H$	U, SV	U, SV, E	0000 0002$_H$
–	Reserved	F000 0604$_H$	nBE	nBE	–
GPTU0_ID	GPTU0 Module Identification Register	F000 0608$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 060C$_H$	nBE	nBE	–
GPTU0_T01IRS	GPTU0 Timers T0 and T1 Input and Reload Source Selection Register	F000 0610$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
GPTU0_ T01OTS	GPTU0 Timers T0 and T1 Output, Trigger and Service Req. Register	F000 0614$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2CON	GPTU0 Timer T2 Control Register	F000 0618$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2RCCON	GPTU0 Timer T2 Reload/ Capture Control Register	F000 061C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2AIS	GPTU0 Timer T2/T2A Ext. Input Selection Register	F000 0620$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2BIS	GPTU0 Timer T2B External Input Selection Register	F000 0624$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2ES	GPTU0 Timer T2 External Input Edge Selection Reg.	F000 0628$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ OSEL	GPTU0 Output Source Selection Register	F000 062C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0 _OUT	GPTU0 Output Register	F000 0630$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T0DCBA	GPTU0 Timer T0 Count Register (T0D, T0C, T0B, T0A)	F000 0634$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T0CBA	GPTU0 Timer T0 Count Register (T0C, T0B, T0A)	F000 0638$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T0RDCBA	GPTU0 Timer T0 Reload Register (T0RD, T0RC, T0RB, T0RA)	F000 063C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T0RCBA	GPTU0 Timer T0 Reload Register (T0RC, T0RB, T0RA)	F000 0640$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T1DCBA	GPTU0 Timer T1 Count Register (T1D, T1C, T1B, T1A)	F000 0644$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T1CBA	GPTU0 Timer T1 Count Register (T1C, T1B, T1A)	F000 0648$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
GPTU0_ T1RDCBA	GPTU0 Timer T1 Reload Register (T1RD, T1RC, T1RB, T1RA)	F000 064C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T1RCBA	GPTU0 Timer T1 Reload Register (T1RC, T1RB, T1RA)	F000 0650$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_T2	GPTU0 Timer T2 Count Register	F000 0654$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2RC0	GPTU0 Timer T2 Reload/ Capture Register 0	F000 0658$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T2RC1	GPTU0 Timer T2 Reload/ Capture Register 1	F000 065C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ T012RUN	GPTU0 Timers T0, T1, T2 Run Control Register	F000 0660$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0664$_H$- F000 06D8$_H$	BE	BE	–
GPTU0_ SRSEL	GPTU0 Service Request Source Select Register	F000 06DC$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC7	GPTU0 Service Request Control Register 7	F000 06E0$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC6	GPTU0 Service Request Control Register 6	F000 06E4$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC5	GPTU0 Service Request Control Register 5	F000 06E8$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC4	GPTU0 Service Request Control Register 4	F000 06EC$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC3	GPTU0 Service Request Control Register 3	F000 06F0$_H$	U, SV	U, SV	0000 0000$_H$
GPTU0_ SRC2	GPTU0 Service Request Control Register 2	F000 06F4$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
GPTU0_ SRC1	GPTU0 Service Request Control Register 1	F000 06F8$_H$	U, SV	U, SV	0000 0000$_H$
GPTU_ SRC0	GPTU0 Service Request Control Register 0	F000 06FC$_H$	U, SV	U, SV	0000 0000$_H$
General Purpose Timer Unit 1 (GPTU1)					
GPTU1_ CLC	GPTU1 Clock Control Reg.	F000 0700$_H$	U, SV	U, SV, E	0000 0002$_H$
–	Reserved	F000 0704$_H$	nBE	nBE	–
GPTU1_ID	GPTU1 Module Identification Register	F000 0708$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 070C$_H$	nBE	nBE	–
GPTU1_ T01IRS	GPTU1 Timers T0 and T1 Input and Reload Source Selection Register	F000 0710$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T01OTS	GPTU1 Timers T0 and T1 Output, Trigger and Service Req. Register	F000 0714$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2CON	GPTU1 Timer T2 Control Register	F000 0718$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2RCCON	GPTU1 Timer T2 Reload/ Capture Control Register	F000 071C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2AIS	GPTU1 Timer T2/T2A Ext. Input Selection Register	F000 0720$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2BIS	GPTU1 Timer T2B External Input Selection Register	F000 0724$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2ES	GPTU1 Timer T2 External Input Edge Selection Reg.	F000 0728$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ OSEL	GPTU1 Output Source Selection Register	F000 072C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ OUT	GPTU1 Output Register	F000 0730$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
GPTU1_ T0DCBA	GPTU1 Timer T0 Count Register (T0D, T0C, T0B, T0A)	F000 0734$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T0CBA	GPTU1 Timer T0 Count Register (T0C, T0B, T0A)	F000 0738$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T0RDCBA	GPTU1 Timer T0 Reload Register (T0RD, T0RC, T0RB, T0RA)	F000 073C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T0RCBA	GPTU1 Timer T0 Reload Register (T0RC, T0RB, T0RA)	F000 0740$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T1DCBA	GPTU1 Timer T1 Count Register (T1D, T1C, T1B, T1A)	F000 0744$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T1CBA	GPTU1 Timer T1 Count Register (T1C, T1B, T1A)	F000 0748$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T1RDCBA	GPTU1 Timer T1 Reload Register (T1RD, T1RC, T1RB, T1RA)	F000 074C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T1RCBA	GPTU1 Timer T1 Reload Register (T1RC, T1RB, T1RA)	F000 0750$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_T2	GPTU1 Timer T2 Count Register	F000 0754$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2RC0	GPTU1 Timer T2 Reload/ Capture Register 0	F000 0758$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T2RC1	GPTU1 Timer T2 Reload/ Capture Register 1	F000 075C$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ T012RUN	GPTU1 Timers T0, T1, T2 Run Control Register	F000 0760$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0764$_H$- F000 07D8$_H$	BE	BE	–
GPTU1_ SRSEL	GPTU1 Service Request Source Select Register	F000 07DC$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
GPTU1_ SRC7	GPTU1 Service Request Control Register 7	F000 07E0$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC6	GPTU1 Service Request Control Register 6	F000 07E4$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC5	GPTU1 Service Request Control Register 5	F000 07E8$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC4	GPTU1 Service Request Control Register 4	F000 07EC$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC3	GPTU1 Service Request Control Register 3	F000 07F0$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC2	GPTU1 Service Request Control Register 2	F000 07F4$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC1	GPTU1 Service Request Control Register 1	F000 07F8$_H$	U, SV	U, SV	0000 0000$_H$
GPTU1_ SRC0	GPTU1 Service Request Control Register 0	F000 07FC$_H$	U, SV	U, SV	0000 0000$_H$
Asynchronous/Synchronous Serial Interface (ASC)					
ASC_ CLC	ASC Clock Control Reg.	F000 0800$_H$	U, SV	U, SV, E	0000 0002$_H$
–	Reserved	F000 0804$_H$	nBE	nBE	–
ASC_ID	ASC Module Identification Register	F000 0808$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 080C$_H$	BE	BE	–
ASC_CON	ASC Control Register	F000 0810$_H$	U, SV	U, SV	0000 0000$_H$
ASC_BG	ASC Baud Rate/Timer Reload Register	F000 0814$_H$	U, SV	U, SV	0000 0000$_H$
ASC_FDV	ASC Fractional Divider Register	F000 0818$_H$	U, SV	U, SV	0000 0000$_H$
ASC_PMW	ASC IrDA Pulse Mode and Width Register	F000 081C$_H$	U, SV	U, SV	0000 0000$_H$
ASC_ TBUF	ASC Transmit Buffer Register	F000 0820$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
ASC_ RBUF	ASC Receive Buffer Register	F000 0824_H	U, SV	U, SV	$0000\ 0000_H$
–	Reserved	F000 0828_H- F000 $083C_H$	BE	BE	–
ASC_RXFC ON	ASC Receive FIFO Control Register	F000 0840_H	U, SV	U, SV	$0000\ 0100_H$
ASC_TXFC ON	ASC Transmit FIFO Control Register	F000 0844_H	U, SV	U, SV	$0000\ 0100_H$
ASC_FTAT	ASC FIFO Status Register	F000 0848_H	U, SV	U, SV	$0000\ 0000_H$
–	Reserved	F000 $084C_H$- F000 $08D4_H$	BE	BE	–
–	Reserved	F000 $08D8_H$- F000 $08DC_H$	nBE	nBE	–
–	Reserved	F000 $08E0_H$- F000 $08EC_H$	BE	BE	–
ASC_ TSRC	ASC Transmit Interrupt Service Req. Control Reg.	F000 $08F0_H$	U, SV	U, SV	$0000\ 0000_H$
ASC_ RSRC	ASC Receive Interrupt Service Req. Control Reg.	F000 $08F4_H$	U, SV	U, SV	$0000\ 0000_H$
ASC_ ESRC	ASC Error Interrupt Service Req. Control Reg.	F000 $08F8_H$	U, SV	U, SV	$0000\ 0000_H$
ASC_ TBSRC	ASC Transmit Buffer Interrupt Service Req. Control Reg.	F000 $08FC_H$	U, SV	U, SV	$0000\ 0000_H$
Asynchronous Serial Interface (16X50)					
16X50_CLC	16X50 Clock Control Reg.	F000 0900_H	U, SV	U, SV, E	$0000\ 0002_H$
–	Reserved	F000 0904_H	nBE	nBE	–
16X50_ID	16X50 Module Identification Register	F000 0908_H	U, SV	BE	$XXXXXXXX_H$
–	Reserved	F000 $090C_H$- F000 $091C_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
16X50_THR	16X50 Transmit Holding Register	F000 0920$_H$	U, SV	U, SV	0000 00XX$_H$
16X50_RHR	16X50 Receive Holding Register				0000 00XX$_H$
16X50_DLL	16X50 Divisor Latch LSB Register				0000 0000$_H$
16X50_IER	16X50 Interrupt Enable Register	F000 0924$_H$	U, SV	U, SV	0000 0000$_H$
16X50_DLM	16X50 Divisor Latch MSB Register				0000 0000$_H$
16X50_ISR	16X50 Interrupt Status Register	F000 0928$_H$	U, SV	U, SV	0000 0001$_H$
16X50_FCR	16X50 FIFO Control Reg.				0000 0000$_H$
16X50_EFR	16X50 Enhanced Feature Register				0000 0000$_H$
16X50_LCR	16X50 Line Control Reg.	F000 092C$_H$	U,SV	U,SV	0000 0000$_H$
16X50_MCR	16X50 Modem Control Register	F000 0930$_H$	U, SV	U, SV	0000 0000$_H$
16X50_ XON1	16X50 XON Character 1 Register				0000 0011$_H$
16X50_LSR	16X50 Line Status Reg.	F000 0934$_H$	U, SV	U, SV	0000 0060$_H$
16X50_ XON2	16X50 XON Character 2 Register				0000 0000$_H$
16X50_MSR	16X50 Modem Status Register	F000 0938$_H$	U, SV	U, SV	0000 00X0$_H$
16X50_ XOFF1	16X50 XOFF Character 1 Register				0000 0013$_H$
16X50_SR	16X50 Scratchpad Reg.	F000 093C$_H$	U, SV	U, SV	0000 00XX$_H$
16X50_ XOFF2	16X50 XON Character 2 Register				0000 0000$_H$
–	Reserved	F000 0940$_H$- F000 09D4$_H$	BE	BE	–
–	Reserved	F000 09D8$_H$- F000 09DC$_H$	nBE	nBE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F000 09E0$_H$-F000 09F8$_H$	BE	BE	–
16X50_SRC	16X50 Interrupt Service Req. Control Reg.	F000 09FC$_H$	U, SV	U, SV	0000 0000$_H$
High-Speed Synchronous Serial Interface (SSC)					
SSC_CLC	SSC Clock Control Reg.	F000 0A00$_H$	U, SV	U, SV, E	0000 0002$_H$
–	Reserved	F000 0A04$_H$	nBE	nBE	–
SSC_ID	SSC Module Identification Register	F000 0A08$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 0A0C$_H$	BE	BE	–
SSC_CON	SSC Control Register	F000 0A10$_H$	U, SV	U, SV	0000 0000$_H$
SSC_BR	SSC Baud Rate Timer Reload Register	F000 0A14$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0A18$_H$-F000 0A1C$_H$	BE	BE	–
SSC_TB	SSC Transmit Buffer Register	F000 0A20$_H$	U, SV	U, SV	0000 0000$_H$
SSC_RB	SSC Receive Buffer Register	F000 0A24$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0A28$_H$-F000 0AF0$_H$	BE	BE	–
SSC_ TSRC	SSC Transmit Interrupt Service Req. Control Reg.	F000 0AF4$_H$	U, SV	U, SV	0000 0000$_H$
SSC_ RSRC	SSC Receive Interrupt Service Req. Control Reg.	F000 0AF8$_H$	U, SV	U, SV	0000 0000$_H$
SSC_ ESRC	SSC Error Interrupt Service Req. Control Reg.	F000 0AFC$_H$	U, SV	U, SV	0000 0000$_H$
MultiMediaCard Interface (MMCI)					
MMCI_CLC	MMCI Clock Control Reg.	F000 0B00$_H$	U, SV	U, SV, E	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F000 0B04$_H$	nBE	nBE	–
MMCI_ID	MMCI Module Identification Register	F000 0B08$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 0B0C$_H$	BE	BE	–
MMCI_DAT	MMCI Data Register	F000 0B10$_H$	U, SV	U, SV	0000 0000$_H$
MMCI_CMD	MMCI Command Register	F000 0B14$_H$	U, SV	U, SV	0000 0008$_H$
–	Reserved	F000 0B18$_H$-F000 0BF8$_H$	BE	BE	–
MMCI_SRC	MMCI Interrupt Service Request Control Register	F000 0BFC$_H$	U, SV	U, SV	0000 0000$_H$
Service Request Unit (SRC)					
–	Reserved	F000 0C00$_H$	nBE	nBE	–
BCU0_SRC	Service Request Control Reg. for BCU0 Interrupt	F000 0C04$_H$	U. SV	U,SV	0000 0000$_H$
–	Reserved	F000 0C08$_H$	nBE	nBE	–
PCI_SRC	Service Request Control Register for PCI Interrupt	F000 0C0C$_H$	U, SV	U,SV	0000 0000$_H$
TES0	Trigger Edge Select Register 0	F000 0C10$_H$	U, SV	U, SV	0000 0000$_H$
TES1	Trigger Edge Select Register 1	F000 0C14$_H$	U, SV	U, SV	0000 0000$_H$
FEN0	Filter Enable Register 0	F000 0C18$_H$	U, SV	U, SV	0000 0000$_H$
FEN1	Filter Enable Register 1	F000 0C1C$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC0	Service Request Control Reg. for Ext. Interrupt 0	F000 0C20$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC1	Service Request Control Reg. for Ext. Interrupt 1	F000 0C24$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC2	Service Request Control Reg. for Ext. Interrupt 2	F000 0C28$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC3	Service Request Control Reg. for Ext. Interrupt 3	F000 0C2C$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EINT_SRC4	Service Request Control Reg. for Ext. Interrupt 4	F000 0C30$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC5	Service Request Control Reg. for Ext. Interrupt 5	F000 0C34$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC6	Service Request Control Reg. for Ext. Interrupt 6	F000 0C38$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC7	Service Request Control Reg. for Ext. Interrupt 7	F000 0C3C$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC8	Service Request Control Reg. for Ext. Interrupt 8	F000 0C40$_H$	U, SV	U, SV	0000 0000$_H$
EINT_SRC9	Service Request Control Reg. for Ext. Interrupt 9	F000 0C44$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC10	Service Request Control Reg. for Ext. Interrupt 10	F000 0C48$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC11	Service Request Control Reg. for Ext. Interrupt 11	F000 0C4C$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC12	Service Request Control Reg. for Ext. Interrupt 12	F000 0C50$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC13	Service Request Control Reg. for Ext. Interrupt 13	F000 0C54$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC14	Service Request Control Reg. for Ext. Interrupt 14	F000 0C58$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC15	Service Request Control Reg. for Ext. Interrupt 15	F000 0C5C$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC16	Service Request Control Reg. for Ext. Interrupt 16	F000 0C60$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC17	Service Request Control Reg. for Ext. Interrupt 17	F000 0C64$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC18	Service Request Control Reg. for Ext. Interrupt 18	F000 0C68$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC19	Service Request Control Reg. for Ext. Interrupt 19	F000 0C6C$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EINT_ SRC20	Service Request Control Reg. for Ext. Interrupt 20	F000 0C70$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC21	Service Request Control Reg. for Ext. Interrupt 21	F000 0C74$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC22	Service Request Control Reg. for Ext. Interrupt 22	F000 0C78$_H$	U, SV	U, SV	0000 0000$_H$
EINT_ SRC23	Service Request Control Reg. for Ext. Interrupt 23	F000 0C7C$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC0	Service Request Control Register for PCI Software Interrupt 0	F000 0C80$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC1	Service Request Control Register for PCI Software Interrupt 1	F000 0C84$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC2	Service Request Control Register for Software Interrupt 2	F000 0C88$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC3	Service Request Control Register for Software Interrupt 3	F000 0C8C$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC4	Service Request Control Register for Software Interrupt 4	F000 0C90$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC5	Service Request Control Register for Software Interrupt 5	F000 0C94$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC6	Service Request Control Register for Software Interrupt 6	F000 0C98$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC7	Service Request Control Register for Software Interrupt 7	F000 0C9C$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_ SRC8	Service Request Control Register for Software Interrupt 8	F000 0CA0$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_SW_SRC9	Service Request Control Register for Software Interrupt 9	F000 0CA4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC10	Service Request Control Register for Software Interrupt 10	F000 0CA8$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC11	Service Request Control Register for Software Interrupt 11	F000 0CAC$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC12	Service Request Control Register for Software Interrupt 12	F000 0CB0$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC13	Service Request Control Register for Software Interrupt 13	F000 0CB4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC14	Service Request Control Register for Software Interrupt 14	F000 0CB8$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC15	Service Request Control Register for Software Interrupt 15	F000 0CBC$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC16	Service Request Control Register for Software Interrupt 16	F000 0CC0$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC17	Service Request Control Register for Software Interrupt 17	F000 0CC4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC18	Service Request Control Register for Software Interrupt 18	F000 0CC8$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC19	Service Request Control Register for Software Interrupt 19	F000 0CCC$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_SW_SRC20	Service Request Control Register for Software Interrupt 20	F000 0CD0$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC21	Service Request Control Register for Software Interrupt 21	F000 0CD4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC22	Service Request Control Register for Software Interrupt 22	F000 0CD8$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC23	Service Request Control Register for Software Interrupt 23	F000 0CDC$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC24	Service Request Control Register for Software Interrupt 24	F000 0CE0$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC25	Service Request Control Register for Software Interrupt 25	F000 0CE4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC26	Service Request Control Register for Software Interrupt 26	F000 0CE8$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC27	Service Request Control Register for Software Interrupt 27	F000 0CEC$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC28	Service Request Control Register for Software Interrupt 28	F000 0CF0$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC29	Service Request Control Register for Software Interrupt 29	F000 0CF4$_H$	U, SV	U, SV	0000 0000$_H$
PCI_SW_SRC30	Service Request Control Register for Software Interrupt 30	F000 0CF8$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_SW_SRC31	Service Request Control Register for Software Interrupt 31	F000 0CFC$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0D00$_H$-F000 0D0C$_H$	BE	BE	–
Ethernet_MACTX0SRC	Service Request Control Register for Ethernet MAC Tx 0 Interrupt	F000 0D10$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_MACRX0SRC	Service Request Control Register for Ethernet MAC Rx 0 Interrupt	F000 0D14$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_MACTX1SRC	Service Request Control Register for Ethernet MAC Tx 1 Interrupt	F000 0D18$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_MACRX1SRC	Service Request Control Register for Ethernet MAC Rx 1 Interrupt	F000 0D1C$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_RBSRC0	Service Request Control Register for Ethernet RB Interrupt 0	F000 0D20$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_RBSRC1	Service Request Control Register for Ethernet RB Interrupt 1	F000 0D24$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_TBSRC	Service Request Control Register for Ethernet TB Interrupt	F000 0D28$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_DRSRC	Service Request Control Register for Ethernet DR Interrupt	F000 0D2C$_H$	U, SV	U, SV	0000 0000$_H$
Ethernet_DTSRC	Service Request Control Register for Ethernet DT Interrupt	F000 0D30$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 0D34$_H$-F000 0DFC$_H$	BE	BE	–
Port 0					

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F000 2800$_H$-F000 280C$_H$	BE	BE	–
P0	Port 0 Data Output Reg.	F000 2810$_H$	U, SV	U, SV	XXXX 0000$_H$
P0_IN	Port 0 Data Input Register	F000 2814$_H$	U, SV	U, SV	XXXX XXXX$_H$
P0_DIR	Port 0 Direction Register	F000 2818$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 281C$_H$-F000 2840$_H$	BE	BE	–
P0_ ALTSEL0	Port 0 Alternate Function Select Register 0	F000 2844$_H$	U, SV	U, SV	0000 0000$_H$
P0_ ALTSEL1	Port 0 Alternate Function Register 1	F000 2848$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 284C$_H$-F000 28FC$_H$	BE	BE	–
Port 1					
–	Reserved	F000 2900$_H$-F000 290C$_H$	BE	BE	–
P1	Port 1 Data Output Reg.	F000 2910$_H$	U, SV	U, SV	XXXX 0000$_H$
P1_IN	Port 1 Data Input Register	F000 2914$_H$	U, SV	U, SV	XXXX XXXX$_H$
P1_DIR	Port 1 Direction Register	F000 2918$_H$	U, SV	U, SV	XXXX 0000$_H$
P1_OD	Port 1 Open Drain Control Register	F000 291C$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2920$_H$-F000 2940$_H$	BE	BE	–
P1_ ALTSEL0	Port 1 Alternate Function Select Register 0	F000 2944$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 2948$_H$-F000 29FC$_H$	BE	BE	–
Port 2					
–	Reserved	F000 2A00$_H$-F000 2A0C$_H$	BE	BE	–
P2	Port 2 Data Output Reg.	F000 2A10$_H$	U, SV	U, SV	XXXX 0000$_H$
P2_IN	Port 2 Data Input Register	F000 2A14$_H$	U, SV	U, SV	XXXX XXXX$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
P2_DIR	Port 2 Direction Register	F000 2A18$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2A1C$_H$-F000 2A40$_H$	BE	BE	–
P2_ ALTSEL0	Port 2 Alternate Function Select Register 0	F000 2A44$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2A48$_H$-F000 2AFC$_H$	BE	BE	–
Port 3					
–	Reserved	F000 2B00$_H$-F000 2B0C$_H$	BE	BE	–
P3	Port 3 Data Output Reg.	F000 2B10$_H$	U, SV	U, SV	XXXX 0000$_H$
P3_IN	Port 3 Data Input Register	F000 2B14$_H$	U, SV	U, SV	XXXX XXXX$_H$
P3_DIR	Port 3 Direction Register	F000 2B18$_H$	U, SV	U, SV	XXXX 0000$_H$
P3_OD	Port 3 Open Drain Control Register	F000 2B1C$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2B20$_H$-F000 2B40$_H$	BE	BE	–
P3_ ALTSEL0	Port 3 Alternate Function Select Register 0	F000 2B44$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2B48$_H$-F000 2BFC$_H$	BE	BE	–
Port 4					
–	Reserved	F000 2C00$_H$-F000 2C0C$_H$	BE	BE	–
P4	Port 4 Data Output Reg.	F000 2C10$_H$	U, SV	U, SV	XXXX 0000$_H$
P4_IN	Port 4 Data Input Register	F000 2C14$_H$	U, SV	U, SV	XXXX XXXX$_H$
P4_DIR	Port 4 Direction Register	F000 2C18$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2C1C$_H$-F000 2C40$_H$	BE	BE	–
P4_ ALTSEL0	Port 4 Alternate Function Select Register	F000 2C44$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F000 2C48$_H$-F000 2CFC$_H$	BE	BE	–
Port 5					
–	Reserved	F000 2D00$_H$-F000 2D0C$_H$	BE	BE	–
P5	Port 5 Data Output Reg.	F000 2D10$_H$	U, SV	U, SV	XXXX 0000$_H$
P5_IN	Port 5 Data Input Register	F000 2D14$_H$	U, SV	U, SV	XXXX XXXX$_H$
P5_DIR	Port 5 Direction Register	F000 2D18$_H$	U, SV	U, SV	XXXX 0000$_H$
–	Reserved	F000 2D1C$_H$-F000 2D40$_H$	BE	BE	–
P5_ ALTSEL0	Port 5 Alternate Function Select Register 0	F000 2D44$_H$	U, SV	U, SV	0000 0000$_H$
P5_ ALTSEL1	Port 5 Alternate Function Select Register 1	F000 2D48$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F000 2D4C$_H$ - F000 2DFC$_H$	BE	BE	–
Peripheral Control Processor (PCP)					
PCP_CLC	PCP Clock Control Reg.	F000 3F00$_H$	U,SV,	U,SV, E	0000 0000$_H$
–	Reserved	F000 3F04$_H$	BE	BE	–
PCP_ID	PCP Module Identification Register	F000 3F08$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F000 3F0C$_H$	BE	BE	–
PCP_CS	PCP Control/Status Register	F000 3F10$_H$	U, SV, 32	SV, E, 32	0000 0000$_H$
PCP_ES	PCP Error/Debug Status Register	F000 3F14$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
–	Reserved	F000 3F18$_H$-F000 3F1C$_H$	BE	BE	–
PCP_ICR	PCP Interrupt Control Register	F000 3F20$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCP_ITR	PCP Interrupt Threshold Control Register	F000 3F24$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
PCP_ICON	PCP Interrupt Configuration Register	F000 3F28$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
PCP_SSR	PCP Stall Status Register	F000 3F2C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
PCP_FTD	PCP Feature Disable/Test Register	F000 3F30$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
–	Reserved	F000 3F34$_H$- F000 3FCC$_H$	BE	BE	–
PCP_ SRC11	PCP Service Request Control Register 11	F000 3FD0$_H$	U, SV, 32	U,SV, 32	0000 1400$_H$
PCP_ SRC10	PCP Service Request Control Register 10	F000 3FD4$_H$	U, SV, 32	U,SV, 32	0000 1400$_H$
PCP_SRC9	PCP Service Request Control Register 9	F000 3FD8$_H$	U, SV, 32	U,SV, 32	0000 1400$_H$
PCP_SRC8	PCP Service Request Control Register 8	F000 3FDC$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
PCP_SRC7	PCP Service Request Control Register 7	F000 3FE0$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
PCP_SRC6	PCP Service Request Control Register 6	F000 3FE4$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
PCP_SRC5	PCP Service Request Control Register 5	F000 3FE8$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
PCP_SRC4	PCP Service Request Control Register 4	F000 3FEC$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
PCP_SRC3	PCP Service Request Control Register 3	F000 3FF0$_H$	U, SV, 32	U,SV, 32	0000 1400$_H$
PCP_SRC2	PCP Service Request Control Register 2	F000 3FF4$_H$	U, SV, 32	U,SV, 32	0000 1400$_H$
PCP_SRC1	PCP Service Request Control Register 1	F000 3FF8$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCP_SRC0	PCP Service Request Control Register 0	F000 3FFC$_H$	U, SV, 32	U,SV, 32	0000 1000$_H$
ComDRAM					
ComDRAM_CLC	ComDRAM Clock Register	F018 0000$_H$	U,SV	SV,E	0000 0000$_H$
ComDRAM_OCDSS	ComDRAM OCDS Suspend Register	F018 0004$_H$	U,SV	SV	0000 0000$_H$
–	Reserved	F018 0008$_H$ - F018 003C$_H$	BE	BE	–
ComDRAM_RST	ComDRAM Reset Register	F018 0040$_H$	U,SV	SV	0000 0000$_H$
–	Reserved	F018 0044$_H$ - F018 019C$_H$	BE	BE	–
ComDRAM_REFCON	ComDRAM Refresh Control Register	F018 01A0$_H$	U,SV	SV	0000 0025$_H$
ComDRAM_MODE	ComDRAM Mode Register	F018 01A4$_H$	U,SV	SV	0000 0000$_H$
–	Reserved	F018 01A8$_H$- F018 FFFF$_H$	BE	BE	–
PCI Interface (PCI)					
PCI_CLC	PCI Clock COntrol Reg.	F040 0000$_H$	U, SV	SV,E	0000 0001$_H$
PCI_FPIID	PCI FPI Identification Reg.	F040 0004$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F040 0008$_H$	BE	BE	–
PCI_SRST	PCI Soft Reset Register	F040 0010$_H$	U, SV	SV	0000 0000$_H$
PCI_PTFERRADDR	PCI PCI_FPI Access Error Address Register	F040 0014$_H$	U, SV	SV	FFFF FFFF$_H$
PCI_FTPERRADDR	PCI FPI_PCI Access Error Address Register	F040 0018$_H$	U, SV	SV	FFFF FFFF$_H$
PCI_FTPERRTAG	PCI FPI_FCI Access Error Tag Register	F040 001C$_H$	U, SV	SV	0000 000F$_H$
PCI_IRR	PCI Interrupt Request Reg	F040 0020$_H$	U, SV	U,SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_IRA/ PCI_IR	PCI Interrupt Request Acknowledge Register/ PCI Interrupt Register	F040 0024$_H$	U, SV	U,SV	0000 0000$_H$
PCI_IRM	PCI Interrupt Mask Reg.	F040 0028$_H$	U, SV	U,SV	FFFF FFFF$_H$
PCI_EOI	PCI End-of-Interrupt Reg	F040 002C$_H$	U, SV	U,SV	0000 0000$_H$
PCI_MODE	PCI Mode Register	F040 0030$_H$	U, SV	SV	0000 010B$_H$
PCI_ID	PCI Module Identification Register	F040 0034$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_SUBID	PCI Subsystem Vendor Identification Reg.	F040 0038$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_PM	PCI Power Management Register	F040 003C$_H$	U, SV	SV	0001 1011$_H$
PCI_CC1	PCI Class Code 1 Register	F040 0040$_H$	U, SV	SV	0002 8000$_H$
PCI_ BAR11M	PCI BAR 11 Mask Register	F040 0044$_H$	U, SV	SV	0000 0008$_H$
PCI_ BAR12M	PCI BAR 12 Mask Register	F040 0048$_H$	U, SV	SV	0000 0008$_H$
PCI_ BAR13M	PCI BAR 13 Mask Register	F040 004C$_H$	U, SV	SV	8000 0008$_H$
PCI_ BAR14M	PCI BAR 14 Mask Register	F040 0050$_H$	U, SV	SV	0FC0 0000$_H$
PCI_ BAR15M	PCI BAR 15 Mask Register	F040 0054$_H$	U, SV	SV	8000 0000$_H$
PCI_ BAR16M	PCI BAR 16 Mask Register	F040 0058$_H$	U, SV	SV	8000 0001$_H$
PCI_ CBCP1	PCI CardBus CIS Pointer 1 Register	F040 005C$_H$	U, SV	SV	0000 0000$_H$
PCI_ SUBID1	PCI Subsystem Identification Register of Function 1	F040 0060$_H$	U,SV	BE	XXXXXXXX$_H$
PCI_PTFAD DRM11	PCI Address Map 11 Register	F040 0064$_H$	U, SV	SV	BFE0 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_PTFADDRM12	PCI Address Map 12 Register	F040 0068$_H$	U, SV	SV	E800 0000$_H$
PCI_PTFADDRM13	PCI Address Map 13 Register	F040 006C$_H$	U, SV	SV	A000 0000$_H$
PCI_PTFADDRM14	PCI Address Map 14 Register	F040 0070$_H$	U, SV	SV	F000 0001$_H$
PCI_PTFADDRM15	PCI Address Map 15 Register	F040 0074$_H$	U, SV	SV	F000 0001$_H$
PCI_PTFADDRM16	PCI Address Map 16 Register	F040 0078$_H$	U, SV	SV	E000 0000$_H$
PCI_SEGEN	FPI Segment Enable Reg.	F040 007C$_H$	U, SV	SV	0000 003F$_H$
PCI_CC2	PCI Class Code 2 Register	F040 0080$_H$	U, SV	SV	FF00 0000$_H$
PCI_BAR21M	PCI BAR 21 Mask Register	F040 0084$_H$	U, SV	SV	8000 0008$_H$
PCI_BAR22M	PCI BAR 22 Mask Register	F040 0088$_H$	U, SV	SV	0078 0008$_H$
PCI_BAR23M	PCI BAR 23 Mask Register	F040 008C$_H$	U, SV	SV	8000 0008$_H$
PCI_BAR24M	PCI BAR 24 Mask Register	F040 0090$_H$	U, SV	SV	0FE0 0000$_H$
PCI_BAR25M	PCI BAR 25 Mask Register	F040 0094$_H$	U, SV	SV	0FFF F000$_H$
PCI_BAR26M	PCI BAR 26 Mask Register	F040 0098$_H$	U, SV	SV	0FFF FFE1$_H$
PCI_CBCP2	PCI CardBus CIS Pointer 2 Register	F040 009C$_H$	U, SV	SV	0000 0000$_H$
PCI_SUBID2	PCI Subsystem Identification Register of Function 2	F040 00A0$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_PTFADDRM21	PCI Address Map 21 Register	F040 00A4$_H$	U, SV	SV	BFE0 0000$_H$
PCI_PTFADDRM22	PCI Address Map 22 Register	F040 00A8$_H$	U, SV	SV	E800 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_PTFADDRM23	PCI Address Map 23 Register	F040 00AC$_H$	U, SV	SV	A000 0000$_H$
PCI_PTFADDRM24	PCI Address Map 24 Register	F040 00B0$_H$	U, SV	SV	F000 0001$_H$
PCI_PTFADDRM25	PCI Address Map 25 Register	F040 00B4$_H$	U, SV	SV	F800 0001$_H$
PCI_PTFADDRM26	PCI Address Map 26 Register	F040 00B8$_H$	U, SV	SV	A000 0000$_H$
PCI_ADMSK11L	FPI Address Mask 11 Low Register	F040 00BC$_H$	U, SV	SV	0000 0000$_H$
PCI_FTPADDRM0	FPI Address Map 0 Register	F040 00C0$_H$	U, SV	SV	0000 0000$_H$
PCI_FTPADDRM1	FPI Address Map 1 Register	F040 00C4$_H$	U, SV	SV	1000 0000$_H$
PCI_FTPADDRM2	FPI Address Map 2 Register	F040 00C8$_H$	U, SV	SV	2000 0000$_H$
PCI_FTPADDRM3	FPI Address Map 3 Register	F040 00CC$_H$	U, SV	SV	3000 0000$_H$
PCI_FTPADDRM4	FPI Address Map 4 Register	F040 00D0$_H$	U, SV	SV	4000 0000$_H$
PCI_FTPADDRM5	FPI Address Map 5 Register	F040 00D4$_H$	U, SV	SV	5000 0000$_H$
PCI_FTPADDRM6	FPI Address Map 6 Register	F040 00D8$_H$	U, SV	SV	6000 0000$_H$
PCI_FTPADDRM7	FPI Address Map 7 Register	F040 00DC$_H$	U, SV	SV	7000 0000$_H$
PCI_FTPADDRM11L	FPI Address Map 11 Low Register	F040 00E0$_H$	U, SV	SV	B000 0000$_H$
PCI_FTPADDRM11H	FPI Address Map 11 High Register	F040 00E4$_H$	U, SV	SV	B800 0000$_H$
PCI_BURSTLEN	FPI Burst Length Register	F040 00E8$_H$	U, SV	SV	0000 0203$_H$
PCI_SSERR	PCI Set PCI SERR Reg.	F040 00EC$_H$	U, SV	SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_ DFSTADDR	DMA FPI Start Address Register	F040 00F0$_H$	U, SV	SV	8000 0000$_H$
PCI_ DPSTADDR	DMA PCI Start Address Register	F040 00F4$_H$	U, SV	SV	0000 0000$_H$
PCI_ TRANLEN	DMA Transfer Length Register	F040 00F8$_H$	U, SV	SV	0000 0000$_H$
PCI_ DMACON	DMA Control/Status Register	F040 00FC$_H$	U, SV	SV	0000 0000$_H$
PCI Configuration Space 1 (PCI_CS1)					
PCI_CS1_ID	PCI_CS1 Module Identification Register	F100 0000$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_CS1_ COMMAND	PCI_CS1 Command/ Status Register	F100 0004$_H$	U, SV	U, SV	0290 0000$_H$
PCI_CS1_ REVID	PCI_CS1 Module Revision Identification Register	F100 0008$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_CS1_ CONTROL	PCI_CS1 Control Register	F100 000C$_H$	U, SV	U, SV	2000 0000$_H$
PCI_CS1_ BAR1	PCI_CS1 Base Address Register 1	F100 0010$_H$	U, SV	U, SV	0000 0008$_H$
PCI_CS1_ BAR2	PCI_CS1 Base Address Register 2	F100 0014$_H$	U, SV	U, SV	0000 0008$_H$
PCI_CS1_ BAR3	PCI_CS1 Base Address Register 3	F100 0018$_H$	U, SV	U, SV	0000 0000$_H$
PCI_CS1_ BAR4	PCI_CS1 Base Address Register 4	F100 001C$_H$	U, SV	U, SV	0000 0000$_H$
PCI_CS1_ BAR5	PCI_CS1 Base Address Register 5	F100 0020$_H$	U, SV	U, SV	0000 0000$_H$
PCI_CS1_ BAR6	PCI_CS1 Base Address Register 6	F100 0024$_H$	U, SV	U, SV	0000 0000$_H$
PCI_CS1_ CCP	PCI_CS1 CardBus CIS Pointer Register	F100 0028$_H$	U, SV	U, SV	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_CS1_ SUBID	PCI_CS1 Module Subsystem Identification Register	F100 002C$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F100 0030$_H$	BE	BE	–
PCI_CS1_ POINTER	PCI_CS1 Capabilities Pointer Register	F100 0034$_H$	U, SV	BE	0000 0040$_H$
–	Reserved	F100 0038$_H$	BE	BE	–
PCI_CS1_ INTCTRL	PCI_CS1 Interrupt Control Register	F100 003C$_H$	U, SV	U, SV	6608 0100$_H$
PCI_CS1_ PMC	PCI_CS1 Power Management Capabilities Register	F100 0040$_H$	U, SV	BE	406B 0001$_H$
PCI_CS1_ PMCSTAT	PCI_CS1 Power Management Control Status Register	F100 0044$_H$	U, SV	U, SV	0000 0000$_H$
–	Reserved	F100 0048$_H$- F100 004C$_H$	BE	BE	–
PCI_CS1_P TFERRADD	PCI_CS1 Error Address PCI_PCI Access Register	F100 0050$_H$	U, SV	U, SV	FFFF FFFF$_H$
PCI_CS1_F TPERRADD	PCI_CS1 Error Address FPI_PCI Access Register	F100 0054$_H$	U, SV	U, SV	FFFF FFFF$_H$
PCI_CS1_F TPERRTAG	PCI_CS1 Error Tag FPI_PCI Access Register	F100 0058$_H$	U, SV	U, SV	0000 000F$_H$
–	Reserved	F100 005C$_H$- F100 00FC$_H$	BE	BE	–
PCI Configuration Space 2(PCI_CS2)					
PCI_CS2_ID	PCI_CS2 Module Identification Register	F100 0100$_H$	U, SV	BE	XXXXXXXX$_H$
PCI_CS2_ COMMAND	PCI_CS2 Command/ Status Register	F100 0104$_H$	U, SV	U, SV	0290 0000$_H$
PCI_CS2_ REVID	PCI_CS2 Module Revision Identification Register	F100 0108$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F100 010C$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PCI_CS2_BAR1	PCI_CS2 Base Address Register 1	$F100\ 0110_H$	U, SV	U, SV	$0000\ 0000_H$
PCI_CS2_BAR2	PCI_CS2 Base Address Register 2	$F100\ 0114_H$	U, SV	U, SV	$0000\ 0008_H$
PCI_CS2_BAR3	PCI_CS2 Base Address Register 3	$F100\ 0118_H$	U, SV	U, SV	$0000\ 0000_H$
PCI_CS2_BAR4	PCI_CS2 Base Address Register 4	$F100\ 011C_H$	U, SV	U, SV	$0000\ 0000_H$
PCI_CS2_BAR5	PCI_CS2 Base Address Register 5	$F100\ 0120_H$	U, SV	U, SV	$0000\ 0000_H$
PCI_CS2_BAR6	PCI_CS2 Base Address Register 6	$F100\ 0124_H$	U, SV	U, SV	$0000\ 0001_H$
PCI_CS2_CCP	PCI_CS2 CardBus CIS Pointer Register	$F100\ 0128_H$	U, SV	U, SV	$0000\ 0000_H$
PCI_CS2_SUBID	PCI_CS2 Module Subsystem Identification Register	$F100\ 012C_H$	U, SV	BE	$XXXXXXXX_H$
–	Reserved	$F100\ 0130_H$-$F100\ 0138_H$	BE	BE	–
PCI_CS2_INTCTRL	PCI_CS2 Interrupt Control Register	$F100\ 013C_H$	U, SV	U, SV	$0000\ 0200_H$
–	Reserved	$F100\ 0140_H$	BE	BE	–
PCI_CS2_PMCSTAT	PCI_CS2 Power Management Control Status Register	$F100\ 0144_H$	U, SV	U, SV	$0000\ 0000_H$
–	Reserved	$F100\ 0148_H$-$F100\ 01FC_H$	BE	BE	–
Bus Control Unit 0 (BCU0)					
–	Reserved	$F200\ 0000_H$ - $F200\ 0004_H$	BE	BE	–
BCU0_ID	BCU0 Module Identification Register	$F200\ 0008_H$	U, SV, 32	BE	$XXXXXXXX_H$
–	Reserved	$F200\ 000C_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
BCU0_CON	BCU0 Control Register	F200 0010$_H$	U, SV, 32	U, SV, 32	4009 FFFF$_H$
–	Reserved	F200 0014$_H$-F200 001C$_H$	BE	BE	–
BCU0_ ECON	BCU0 Error Control Capture Register	F200 0020$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
BCU0_ EADD	BCU0 Error Address Capture Register	F200 0024$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
BCU0_ EDAT	BCU0 Error Data Capture Register	F200 0028$_H$	U, SV, 32	U, SV, 32	0000 0000$_H$
–	Reserved	F200 002C$_H$-F200 00FC$_H$	BE	BE	–
Ethernet Controller Module (Ethernet)					
–	Reserved	F200 0100$_H$-F200 0114$_H$	BE	BE	–
Ethernet_ DRISFIFO	DR Interrupt Status FIFO Register	F200 0118$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_ DRFFCR	DR FIFO Full Counter Register	F200 011C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_ DRCMD	DR Command Register	F200 0120$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ DRMOD	DR Mode Register	F200 0124$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
–	Reserved	F200 0128$_H$	BE	BE	–
Ethernet_ DRFRDA	DR First Rx Descriptor Address Register	F200 012C$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ DRIMR	DR Interrupt Mask Register	F200 0130$_H$	U, SV, 32	U,SV, 32	0000 7F05$_H$
Ethernet_ DRCONF	DR Configuration Register	F200 0134$_H$	U, SV, 32	U,SV, 32	8300 0060$_H$
–	Reserved	F200 0138$_H$-F200 0214$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
Ethernet_ DTCMD	DT Command Register	F200 0218$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
–	Reserved	F200 021C$_H$	BE	BE	–
Ethernet_ DTFTDA	DT First Tx Descriptor Address Register	F200 0220$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ DTIMR	DT Interrupt Mask Register	F200 0224$_H$	U, SV, 32	U,SV, 32	0000 0007$_H$
Ethernet_ DTCONF	DT Configuration Register	F200 0228$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ DTCONF3	DT Configuration 3 Reg.	F200 022C$_H$	U, SV, 32	U,SV, 32	0009 0000$_H$
Ethernet_ DTISFIFO	DT Interrupt Status FIFO Register	F200 0230$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_ DTFFCR	DT FIFO Full Counter Register	F200 023C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
–	Reserved	F200 0240$_H$- F200 030C$_H$	BE	BE	–
Ethernet_ MACCTRL	MAC Control Register	F200 0310$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACCAMCTRL0	MAC CAM Control Register 0	F200 0314$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACTXCTRL	MAC Transmit Control Register	F200 0318$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACTXSTAT	MAC Transmit Status Register	F200 031C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_ MACRXCTRL	MAC Receive Control Register	F200 0320$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACRXSTAT	MAC Receive Status Register	F200 0324$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
Ethernet_ MACSMDATA	MAC Station Management Data Register	F200 0328$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACSMCTRL	MAC Station management Control Register	F200 032C$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACCAMADDR	MAC CAM Address Register	F200 0330$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACCAMDATA	MAC CAM Data Register	F200 0334$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACCAMCTRL1	MAC CAM Control Register 1	F200 0338$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACMERRCNT	MAC Missed Error Count Register	F200 033C$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACPSECNT	MAC Pause Count Register	F200 0340$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACRPSECNT	MAC Remote Pause Count Register	F200 0344$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACTX0IMR	MAC Transmit 0 Interrupt Mask Register	F200 0348$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACTX0ISR	MAC Transmit 0 Interrupt Status Register	F200 034C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_ MACTX1IMR	MAC Transmit 1 Interrupt Mask Register	F200 0350$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_ MACTX1ISR	MAC Transmit 1 Interrupt Status Register	F200 0354$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F200 0358$_H$-F200 0364$_H$	BE	BE	–
Ethernet_MACRX0IMR	MAC Receive 0 Interrupt Mask Register	F200 0368$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_MACRX0ISR	MAC Receive 0 Interrupt Status Register	F200 036C$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_MACRX1IMR	MAC Receive 1 Interrupt Mask Register	F200 0370$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_MACRX1ISR	MAC Receive 1 Interrupt Status Register	F200 0374$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
–	Reserved	F200 0378$_H$-F200 0414$_H$	BE	BE	–
Ethernet_RBCC	RB Channel Command Register	F200 0418$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_RBCBL	RB Channel Burst Length Register	F200 041C$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$
Ethernet_RBFPM	RB Free Pool Monitor Reg.	F200 0420$_H$	U, SV, 32	U,SV, 32	0000 0020$_H$
Ethernet_RBFPTH	RB Free Pool Threshold Register	F200 0424$_H$	U, SV, 32	U,SV, 32	2000 0001$_H$
–	Reserved	F200 0428$_H$	BE	BE	–
Ethernet_RBFPCNT	RB Free Pool Count Register	F200 0430$_H$	U, SV, 32	SV,32 NC	0020 00FF$_H$
–	Reserved	F200 0434$_H$-F200 0510$_H$	BE	BE	–
Ethernet_TBISR	TB Interrupt Status Register	F200 0514$_H$	U, SV, 32	SV,32 NC	0000 0000$_H$
Ethernet_TBCC	TB Channel Command Register	F010 0354$_H$	U, SV, 32	U,SV, 32	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
Ethernet_ TBCPR	TB Channel Parameter Register	F010 0358$_H$	U, SV, 32	U,SV, 32	0020 0000$_H$
–	Reserved	F200 035C$_H$- F200 05FC$_H$	BE	BE	–
External Bus Unit (EBU)					
EBU_CLC	EBU Clock Control Reg.	F800 0000$_H$	U, SV	U, SV, E	0000 0000$_H$
–	Reserved	F800 0004$_H$	BE	BE	–
EBU_ID	EBU Module Identification Register	F800 0008$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F800 000C$_H$	BE	BE	–
EBU_CON	EBU External Bus Configuration Register	F800 0010$_H$	U, SV	SV	0000 0028$_H$ 0001 0068$_H$
–	Reserved; this location must not be written	F800 0014$_H$- F800 001C$_H$	nBE	nBE	–
EBU_ BFCON	EBU Burst Flash Access Control Register	F800 0020$_H$	U, SV	SV	0100 01D0$_H$
EBU_ SDRMREF0	EBU SDRAM Type 0 Refresh Control Register	F800 0040$_H$	U, SV	SV	0000 0000$_H$
–	Reserved	F800 0044$_H$	BE	BE	–
EBU_ SDRMREF1	EBU SDRAM Type 1 Refresh Control Register	F800 0048$_H$	U, SV	SV	0000 00D0$_H$
–	Reserved	F800 004C$_H$	BE	BE	–
EBU_ SDRMCON0	EBU SDRAM Type 0 Configuration Register	F800 0050$_H$	U, SV	SV	0000 0000$_H$
–	Reserved	F800 0054$_H$	BE	BE	–
EBU_ SDRMCON1	EBU SDRAM Type 1 Configuration Register	F800 0058$_H$	U, SV	SV	0000 00D0$_H$
–	Reserved	F800 005C$_H$	BE	BE	–
EBU_ SDRMOD0	EBU SDRAM Type 0 Mode Register	F800 0060$_H$	U, SV	SV	0000 0000$_H$
–	Reserved	F800 0064$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EBU_ SDRMOD1	EBU SDRAM Type 1 Mode Register	F800 0068$_H$	U, SV	SV	0000 00D0$_H$
–	Reserved	F800 006C$_H$	BE	BE	–
EBU_ SDRSTAT0	EBU SDRAM Type 0 Status Register	F800 0070$_H$	U, SV	SV, NC	0000 0000$_H$
–	Reserved	F800 0074$_H$	BE	BE	–
EBU_ SDRSTAT1	EBU SDRAM Type 1 Status Register	F800 0078$_H$	U, SV	SV, NC	0000 00D0$_H$
–	Reserved	F800 007C$_H$	BE	BE	–
EBU_ ADDSEL0	EBU Address Select Register 0	F800 0080$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 0084$_H$	BE	BE	–
EBU_ ADDSEL1	EBU Address Select Register 1	F800 0088$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 008C$_H$	BE	BE	–
EBU_ ADDSEL2	EBU Address Select Register 2	F800 0090$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 0094$_H$	BE	BE	–
EBU_ ADDSEL3	EBU Address Select Register 3	F800 0098$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 009C$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EBU_ ADDSEL4	EBU Address Select Register 4	F800 00A0$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 00A4$_H$	BE	BE	–
EBU_ ADDSEL5	EBU Address Select Register 5	F800 00A8$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 00AC$_H$	BE	BE	–
EBU_ ADDSEL6	EBU Address Select Register 6	F800 00B0$_H$	U, SV	U, SV	0000 0000$_H$ A000 0001$_H$
–	Reserved	F800 00B4$_H$- F800 00BC$_H$	BE	BE	–
EBU_ BUSCON0	EBU Bus Configuration Register 0	F800 00C0$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00C4$_H$	BE	BE	–
EBU_ BUSCON1	EBU Bus Configuration Register 1	F800 00C8$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00CC$_H$	BE	BE	–
EBU_ BUSCON2	EBU Bus Configuration Register 2	F800 00D0$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00D4$_H$	BE	BE	–
EBU_ BUSCON3	EBU Bus Configuration Register 3	F800 00D8$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00DC$_H$	BE	BE	–
EBU_ BUSCON4	EBU Bus Configuration Register 4	F800 00E0$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00E4$_H$	BE	BE	–
EBU_ BUSCON5	EBU Bus Configuration Register 5	F800 00E8$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00EC$_H$	BE	BE	–
EBU_ BUSCON6	EBU Bus Configuration Register 6	F800 00F0$_H$	U, SV	U, SV	8092 8000$_H$ 8092 807F$_H$
–	Reserved	F800 00F4$_H$- F800 00FC$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EBU_ BUSAP0	EBU Bus Access Parameter Register 0	F800 0100$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 0104$_H$	BE	BE	–
EBU_ BUSAP1	EBU Bus Access Parameter Register 1	F800 0108$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 010C$_H$	BE	BE	–
EBU_ BUSAP2	EBU Bus Access Parameter Register 2	F800 0110$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 0114$_H$	BE	BE	–
EBU_ BUSAP3	EBU Bus Access Parameter Register 3	F800 0118$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 011C$_H$	BE	BE	–
EBU_ BUSAP4	EBU Bus Access Parameter Register 4	F800 0120$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 0124$_H$	BE	BE	–
EBU_ BUSAP5	EBU Bus Access Parameter Register 5	F800 0128$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 012C$_H$	BE	BE	–
EBU_ BUSAP6	EBU Bus Access Parameter Register 6	F800 0130$_H$	U, SV	U, SV	FFFF FFFF$_H$
–	Reserved	F800 0134$_H$- F800 015C$_H$	BE	BE	–
EBU_ EMUAS	EBU Emulator Address Select Register	F800 0160$_H$	U, SV	U, SV	DE00 0031$_H$
–	Reserved	F800 0164$_H$	BE	BE	–
EBU_ EMUBC	EBU Emulator Bus Configuration Register	F800 0168$_H$	U, SV	U, SV	0190 2077$_H$
–	Reserved	F800 016C$_H$	BE	BE	–
EBU_ EMUBAP	EBU Bus Emulator Region Access Parameter Register	F800 0170$_H$	U, SV	U, SV	5248 4911$_H$
–	Reserved	F800 0174$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
EBU_ EMUOVL	EBU Emulator Overlay Memory Chip Select Generation Register	F800 0178$_H$	U, SV	U, SV	0000 0000$_H$
EBU_ EXTCON	EBU External Access Configuration Register 8	Only from external (400000$_H$)	–	–	3C0B FFA0$_H$
–	Reserved	F800 017C$_H$- F800 02FC$_H$	BE	BE	–
Local Memory Unit (LMU)					
LMU_MODE	LMU Mode Register	F800 0400$_H$	U, SV, 32	SV,32	0000 0000$_H$
–	Reserved	F800 0404$_H$	BE	BE	–
LMU_ REFRATE	LMU Refresh Rate Register	F800 0408$_H$	U, SV, 32	SV,32	0140 5014$_H$
–	Reserved	F800 040C$_H$	BE	BE	–
LMU_ID	LMU Module Identification Register	F800 0410$_H$	U, SV, 32	BE	XXXXXXXX$_H$
–	Reserved	F800 0414$_H$- F800 04FC$_H$	BE	BE	–
CPU Slave Interface Registers (CPS)					
–	Reserved	F7E0 FF00$_H$- F7E0 FF14$_H$	BE	BE	–
CPU_ID	CPU Module Identification Register	F7E0 FF18$_H$	U, SV	BE	XXXXXXXX$_H$
–	Reserved	F7E0 FF1C$_H$ - F7E0 FFB8$_H$	BE	BE	–
CPU_ SBSRC0	Software Break Service Request Control Reg. 0	F7E0 FFBC$_H$	U, SV	SV	0000 0000$_H$
–	Reserved	F7E0 FFC0$_H$ - F7E0 FFEC$_H$	BE	BE	–

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
CPU_SRC3	CPU Service Request Control Register 3	F7E0 FFF0$_H$	U, SV	SV	0000 0000$_H$
CPU_SRC2	CPU Service Request Control Register 2	F7E0 FFF4$_H$	U, SV	SV	0000 0000$_H$
CPU_SRC1	CPU Service Request Control Register 1	F7E0 FFF8$_H$	U, SV	SV	0000 0000$_H$
CPU_SRC0	CPU Service Request Control Register 0	F7E0 FFFC$_H$	U, SV	SV	0000 0000$_H$
Memory Management Unit (MMU)					
MMU_CON	MMU Configuration Reg.	F7E1 8000$_H$	U, SV, 32	SV,32	0000 07E0$_H$
MMU_ASI	MMU Address Space Identifier Register	F7E1 8004$_H$	U, SV, 32	SV,32	0000 001F$_H$
MMU_ID	MMU Module Identification Register	F7E1 8008$_H$	U, SV, 32	BE	XXXXXXXX$_H$
MMU_TVA	MMU Translation Virtual Address Register	F7E1 800C$_H$	U, SV, 32	SV,32	0000 0000$_H$
MMU_TPA	MMU Translation Physical Address Register	F7E1 8010$_H$	U, SV, 32	SV,32	0000 0000$_H$
MMU_TPX	MMU Translation Page Index Register	F7E1 8014$_H$	U, SV, 32	SV,32	0000 0000$_H$
MMU_TFA	MMU Translation Fault Address Register	F7E1 8018$_H$	U, SV, 32	SV,32	0000 0000$_H$
–	Reserved	F7E1 801C$_H$-F7E1 80FC$_H$	BE	BE	–
Memory Protection Registers					
DPR0_0L	Data Seg. Protect. Reg. Set 0, Range 0, Lower	F7E1 C000$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_0U	Data Seg. Protect. Reg. Set 0, Range 0, Upper	F7E1 C004$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_1L	Data Seg. Protect. Reg. Set 0, Range 1, Lower	F7E1 C008$_H$	U, SV, 32	SV, 32	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
DPR0_1U	Data Seg. Protect. Reg. Set 0, Range 1, Upper	F7E1 C00C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_2L	Data Seg. Protect. Reg. Set 0, Range 2, Lower	F7E1 C010$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_2U	Data Seg. Protect. Reg. Set 0, Range 2, Upper	F7E1 C014$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_3L	Data Seg. Protect. Reg. Set 0, Range 3, Lower	F7E1 C018$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR0_3U	Data Seg. Protect. Reg. Set 0, Range 3, Upper	F7E1 C01C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 C020$_H$-F7E1 C3FC$_H$	nE	nE	–
DPR1_0L	Data Seg. Protect. Reg. Set 1, Range 0, Lower	FFFF C400$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_0U	Data Seg. Protect. Reg. Set 1, Range 0, Upper	F7E1 C404$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_1L	Data Seg. Protect. Reg. Set 1, Range 1, Lower	F7E1 C408$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_1U	Data Seg. Protect. Reg. Set 1, Range 1, Upper	F7E1 C40C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_2L	Data Seg. Protect. Reg. Set 1, Range 2, Lower	F7E1 C410$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_2U	Data Seg. Protect. Reg. Set 1, Range 2, Upper	F7E1 C414$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_3L	Data Seg. Protect. Reg. Set 1, Range 3, Lower	F7E1 C418$_H$	U, SV, 32	SV, 32	0000 0000$_H$
DPR1_3U	Data Seg. Protect. Reg. Set 1, Range 3, Upper	F7E1 C41C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 C420$_H$-F7E1 CFFC$_H$	nE	nE	–
CPR0_0L	Code Seg. Prot. Register Set 0, Range 0, Lower	F7E1 D000$_H$	U, SV, 32	SV, 32	0000 0000$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
CPR0_0U	Code Seg. Prot. Register Set 0, Range 0, Upper	F7E1 $D004_H$	U, SV, 32	SV, 32	0000 0000_H
CPR0_1L	Code Seg. Prot. Register Set 0, Range 1, Lower	F7E1 $D008_H$	U, SV, 32	SV, 32	0000 0000_H
CPR0_1U	Code Seg. Prot. Register Set 0, Range 1, Upper	F7E1 $D00C_H$	U, SV, 32	SV, 32	0000 0000_H
–	Reserved	F7E1 $D010_H$- F7E1 $D3FC_H$	nE	nE	–
CPR1_0L	Code Seg. Prot. Register Set 1, Range 0, Lower	F7E1 $D400_H$	U, SV, 32	SV, 32	0000 0000_H
CPR1_0U	Code Seg. Prot. Register Set 1, Range 0, Upper	F7E1 $D404_H$	U, SV, 32	SV, 32	0000 0000_H
CPR1_1L	Code Seg. Prot. Register Set 1, Range 1, Lower	F7E1 $D408_H$	U, SV, 32	SV, 32	0000 0000_H
CPR1_1U	Code Seg. Prot. Register Set 1, Range 1, Upper	F7E1 $D40C_H$	U, SV, 32	SV, 32	0000 0000_H
–	Reserved	F7E1 $D410_H$- F7E1 $DFFC_H$	nE	nE	–
DPM0	Data Memory Protection Mode Register 0	F7E1 $E000_H$	U, SV, 32	SV, 32	0000 0000_H
–	Reserved	F7E1 $E004_H$- F7E1 $E07C_H$	nE	nE	–
DPM1	Data Memory Protection Mode Register 1	F7E1 $E080_H$	U, SV, 32	SV, 32	0000 0000_H
–	Reserved	F7E1 $E084_H$- F7E1 $E1FC_H$	nE	nE	–
CPM0	Code Memory Protection Mode Register 0	F7E1 $E200_H$	U, SV, 32	SV, 32	0000 0000_H
–	Reserved	F7E1 $E204_H$- F7E1 $E27C_H$	nE	nE	–
CPM1	Code Memory Protection Mode Register 1	F7E1 $E280_H$	U, SV, 32	SV, 32	0000 0000_H

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
–	Reserved	F7E1 E284$_H$- F7E1 FCFC$_H$	nE	nE	–
Core Debug Register (OCDS)					
DBGSR	Debug Status Register	F7E1 FD00$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 FD04$_H$	nE	nE	–
EXEVT	External Break Input Event Specifier Register	F7E1 FD08$_H$	U, SV, 32	SV, 32	0000 0000$_H$
CREVT	Emulator Resource Protection Event Specifier Register	F7E1 FD0C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
SWEVT	Software Break Event Specifier Register	F7E1 FD10$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 FD14$_H$ - F7E1 FD1C$_H$	nBE	nBE	–
TR0EVT	Trigger Event 0 Specifier Register	F7E1 FD20$_H$	U, SV, 32	SV, 32	0000 0000$_H$
TR1EVT	Trigger Event 1 Specifier Register	F7E1 FD24$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1FD28$_H$- F7E1FD3C$_H$	nBE	nBE	–
DMS	Debug Monitor Start Address Register	F7E1 FD40$_H$	U, SV, 32	BE	DE00 0000$_H$
DCX	Debug Context Save Area Pointer Register	F7E1 FD44$_H$	U, SV, 32	SV, 32	DE80 0000$_H$
–	Reserved	F7E1FD48$_H$- F7E1FDFC$_H$	nBE	nBE	–
Core Special Function Registers (CSFR)					
PCXI	Previous Context Information Register	F7E1 FE00$_H$	U, SV, 32	SV, 32	0000 0000$_H$
PSW	Program Status Word	F7E1 FE04$_H$	U, SV, 32	SV, 32	0000 0B80$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PC	Program Counter	F7E1 FE08$_H$	U, SV, 32	SV, 32	acc. boot cfg.
–	Reserved	F7E1FE0C$_H$-F7E1FE10$_H$	nBE	nBE	–
SYSCON	System Configuration Register	F7E1 FE14$_H$	U, SV, 32	SV,32	0000 0000$_H$
–	Reserved	F7E1 FE18$_H$-FFFF FE1C$_H$	nBE	nBE	–
BIV	Interrupt Vector Table Pointer	F7E1 FE20$_H$	U, SV, 32	SV, E; 32	0000 0000$_H$
BTV	Trap Vector Table Pointer	F7E1 FE24$_H$	U, SV, 32	SV, E; 32	A000 0100$_H$
ISP	Interrupt Stack Pointer	F7E1 FE28$_H$	U, SV, 32	SV, E; 32	0000 0100$_H$
ICR	ICU Interrupt Control Register	F7E1 FE2C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 FE30$_H$-F7E1 FE34$_H$	nBE	nBE	–
FCX	Free Context List Head Pointer	F7E1 FE38$_H$	U, SV, 32	SV, 32	0000 0000$_H$
LCX	Free Context List Limit Pointer	F7E1 FE3C$_H$	U, SV, 32	SV, 32	0000 0000$_H$
–	Reserved	F7E1 FE40$_H$-F7E1 FEFC$_H$	nBE	nBE	–
General Purpose Register (GPR)					
D0	Data Register D0 (DGPR)	F7E1 FF00$_H$	–	–	XXXX XXXX$_H$
D1	Data Register D1 (DGPR)	F7E1 FF04$_H$	–	–	XXXX XXXX$_H$
D2	Data Register D2 (DGPR)	F7E1 FF08$_H$	–	–	XXXX XXXX$_H$
D3	Data Register D3 (DGPR)	F7E1 FF0C$_H$	–	–	XXXX XXXX$_H$
D4	Data Register D4 (DGPR)	F7E1 FF10$_H$	–	–	XXXX XXXX$_H$
D5	Data Register D5 (DGPR)	F7E1 FF14$_H$	–	–	XXXX XXXX$_H$
D6	Data Register D6 (DGPR)	F7E1 FF18$_H$	–	–	XXXX XXXX$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
D7	Data Register D7 (DGPR)	F7E1 FF1C$_H$	–	–	XXXX XXXX$_H$
D8	Data Register D8 (DGPR)	F7E1 FF20$_H$	–	–	XXXX XXXX$_H$
D9	Data Register D9 (DGPR)	F7E1 FF24$_H$	–	–	XXXX XXXX$_H$
D10	Data Register 10 (DGPR)	F7E1 FF28$_H$	–	–	XXXX XXXX$_H$
D11	Data Register 11 (DGPR)	F7E1 FF2C$_H$	–	–	XXXX XXXX$_H$
D12	Data Register 12 (DGPR)	F7E1 FF30$_H$	–	–	XXXX XXXX$_H$
D13	Data Register 13 (DGPR)	F7E1 FF34$_H$	–	–	XXXX XXXX$_H$
D14	Data Register 14 (DGPR)	F7E1 FF38$_H$	–	–	XXXX XXXX$_H$
D15	Data Register 15 (DGPR)	F7E1 FF3C$_H$	–	–	XXXX XXXX$_H$
–	Reserved	F7E1 FF40$_H$- F7E1 FF7C$_H$	nE	nE	–
A0	Address Reg. 0 (AGPR) Global Address Register	F7E1 FF80$_H$	–	–	XXXX XXXX$_H$
A1	Address Reg. 1 (AGPR) Global Address Register	F7E1 FF84$_H$	–	–	XXXX XXXX$_H$
A2	Address Register 2 (AGPR)	F7E1 FF88$_H$	–	–	XXXX XXXX$_H$
A3	Address Register 3 (AGPR)	F7E1 FF8C$_H$	–	–	XXXX XXXX$_H$
A4	Address Register 4 (AGPR)	F7E1 FF90$_H$	–	–	XXXX XXXX$_H$
A5	Address Register 5 (AGPR)	F7E1 FF94$_H$	–	–	XXXX XXXX$_H$
A6	Address Register 6 (AGPR)	F7E1 FF98$_H$	–	–	XXXX XXXX$_H$
A7	Address Register 7 (AGPR)	F7E1 FF9C$_H$	–	–	XXXX XXXX$_H$
A8	Address Reg. 8 (AGPR) Global Address Register	F7E1 FFA0$_H$	–	–	XXXX XXXX$_H$
A9	Address Reg. 9 (AGPR) Global Address Register	F7E1 FFA4$_H$	–	–	XXXX XXXX$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
A10 (SP)	Address Reg. 10 (AGPR) Stack Pointer	F7E1 FFA8$_H$	–	–	XXXX XXXX$_H$
A11 (RA)	Address Reg. 11 (AGPR) Return Address	F7E1 FFAC$_H$	–	–	XXXX XXXX$_H$
A12	Address Reg. 12 (AGPR)	F7E1 FFB0$_H$	–	–	XXXX XXXX$_H$
A13	Address Reg. 13 (AGPR)	F7E1 FFB4$_H$	–	–	XXXX XXXX$_H$
A14	Address Reg. 14 (AGPR)	F7E1 FFB8$_H$	–	–	XXXX XXXX$_H$
A15	Address Reg. 15 (AGPR)	F7E1 FFBC$_H$	–	–	XXXX XXXX$_H$
–	Reserved	F7E1FFC0$_H$-F7E1FFFC$_H$	nE	nE	–
Data Memory Unit (DMU)					
–	Reserved	F87F FC00$_H$	nBE	nBE	–
–	Reserved	F87F FC04$_H$	BE	BE	3)
DMU_ID	DMU Module Identification Register	F87F FC08$_H$	U, SV	BE	XXXX XXXX$_H$
–	Reserved	F87F FC0C$_H$	BE	BE	3)
DMU_CON	DMU Control Register	F87F FC10$_H$	U, SV	SV	0000 0032$_H$
–	Reserved	F87F FC14$_H$	BE	BE	3)
DMU_STR	DMU Synchronous Trap Flag Register	F87F FC18$_H$	U, SV 2)	1)	0000 0000$_H$
–	Reserved	F87F FC1C$_H$	BE	BE	3)
DMU_ATR	DMU Asynchronous Trap Flag Register	F87F FC20$_H$	U, SV 2)	1)	0000 0000$_H$
–	Reserved	F87F FC24$_H$-F87F FCFC$_H$	BE	BE	3)
Program Memory Unit (PMU)					
–	Reserved	F87F FD00$_H$-F87F FD04$_H$	BE	BE	–
PMU_ID	PMU Module Identification Register	F87F FD08$_H$	U, SV	BE	XXXX XXXX$_H$
–	Reserved	F87F FD0C$_H$	BE	BE	–
PMU_CON0	PMU Control Register 0	F87F FD10$_H$	U, SV	SV,32	0000 0002$_H$

Table 22-4 Detailed Address Map of Segment 15 (cont'd)

Short Name	Description	Address	Access Mode		Reset Value
			Read	Write	
PMU_CON1	PMU Control Register 1	F87F FD14$_H$	U,SV	SV,32	0000 0000$_H$
PMU_CON2	PMU Control Register 2	F87F FD18$_H$	U, SV	SV,32	0000 0032$_H$
–	Reserved	F87FFD1C$_H$-F87F FDFC$_H$	BE	BE	–
LMB Bus Control Unit (LCU)					
–	Reserved	F87F FE00$_H$	nBE	nBE	–
–	Reserved	F87F FE04$_H$	BE	BE	
LCU_ID	LCU Module Identification Register	F87F FE08$_H$	U, SV, 32	BE	XXXX XXXX$_H$
–	Reserved	F87FFE0C$_H$-F87FFE1C$_H$	BE	BE	
LCU_EATT	LCU Error Attribute Capture Register	F87F FE20$_H$	U, SV, 32	SV,32	0000 0000$_H$
LCU_EADD	LCU Error Address Capture Register	F87F FE24$_H$	U, SV, 32	SV,32	0000 0000$_H$
LCU_EDAT	LCU Error Data Capture Register	F87F FE28$_H$	U, SV, 2*32,	SV, 2*32	0000 0000$_H$
–	Reserved	F87FFE30$_H$-F87FFEF8$_H$	BE	BE	
LCU_SRC	LCU Service Request Control Register	F87F FEFC$_H$	U, SV	U,SV	0000 0000$_H$
LMB to FPI Bus Bridge(LFI)					
–	Reserved	F87F FF00$_H$	nBE	nBE	–
–	Reserved	F87F FF04$_H$	BE	BE	
LFI_ID	LCU Module Identification Register	F87F FF08$_H$	U, SV	BE	XXXX XXXX$_H$
–	Reserved	F87F FF0C$_H$	BE	BE	
LFI_CON	LFI Configuration Register	F87F FF10$_H$	U, SV	U,SV	0000 0306$_H$
–	Reserved	F87F FF14$_H$-F87F FFFC$_H$	BE	BE	

1) Access to the DMU registers must only be made with double-word-aligned word accesses. An access not conforming to this rule, or an access which does not follow the specified privilege mode (supervisor mode), or

a write access to a read-only register, will cause a bus error if the access was from the FPI Bus, or to a trap, flagged with a DMU Control Register Error Flag (see DMUSTR/DMUATR registers) in case of a CPU load/ store access.

2) Reading this register in supervisor mode returns the contents and then clears the register. Reading it in user mode only returns the contents of the register and does not clear its bits. No error will be reported in this case.

3) A read access to this range will lead to a bus error on data load operation trap. A write access to this range will lead to a bus error on data store operation trap.

23 Index

23.1 Keyword Index

This section lists a number of keywords which refer to specific details of the TC11IB in terms of its architecture, its functional units. or functions. Bold page number entries identify the main definition material for a topic.

D

E

F

I

L

M

N

O

P

R

S

T

W

23.2 Register Index

This section lists the references to the Special Function Registers of the TC11IB.

D

E

F

G

R

S

T

W

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG