TEST AND VERIFICATION PLAN

EVLA Correlator Chip

TVP Document: **A25082N0001**

Revision: DRAFT

Brent Carlson, August 26, 2004

National Research Council Canada
Herzberg Institute of Astrophysics
Dominion Radio Astrophysical Observatory

P.O. Box 248, 717 White Lake Rd
Penticton, B.C., Canada
V2A 6K3

Table of Contents

List of Abbreviations and Acronyms

ASIC – Application Specific Integrated Circuit. This is a chip built to perform a specific set of functions. This is also referred to as a "custom chip" or "custom ASIC".

EVLA – Expanded Very Large Array. This is the project that the EVLA correlator, including the EVLA correlator chip is being built for.

FPGA – Field Programmable Gate Array. A programmable chip that contains an array of logic functions surrounded by programmable interconnects.

FSM – Finite State Machine. Logic that implements some controller function that contains a finite number of states, responds to input stimuli, and produces output signals.

PAR – Refers to Place and Route of the correlator chip.

PPAR – Refers to Post Place and Route of the correlator chip.

RTL – Register Transfer Language. This is synthesizable Verilog (VHDL is another industry standard that is not used for the correlator chip) code used to build the chip's logic functions. RTL is a subset of the entire HDL (Hardware Description Language)

Structured ASIC – This is an ASIC that contains a sea of gates or cells and a fixed number of specialized functions such as PLLs and memory. The Strucutred ASIC is frozen to perform fixed functions once metal interconnect layers are added in the design and manufacturing process.

VLA – Very Large Array. Array of 27, 25 m antennas in New Mexico built in the 1970s and operating since ~1980.

VLSI – Very Large Scale Integration.

1 Revision History

Revision	Date	Changes/Notes	Author
I/C DRAFT	June 26, 2003	Incomplete DRAFT for discussion purposes	B. Carlson
DRAFT	August 26, 2004	DRAFT initial complete release	B. Carlson

2 Introduction

This document describes the test and verification plan for the EVLA Correlator Chip. Detailed requirements and functionality can be found in the EVLA Correlator Chip RFS document [1].

The Correlator Chip is the heart of the correlator and there are 64 of them on every Baseline Board, with a minimum of 160 of these boards in the system. Because of speed, logic density, cost, and quantity it is necessary to implement this chip in a full-custom VLSI ASIC (Application Specific Integrated Circuit) or in a "structured ASIC" device. (The correlator chip is about a 3.5 million gate design, and has about a 4 W power dissipation—implementation of this device in an FPGA is, at this point in time, technologically impossible and not feasible in terms of power and cost.) Fabrication of the chip is an expensive and lengthy process and necessarily requires careful testing at all stages of development to minimize the chance of failure and thus increased cost and development time. Depending on where a failure is found, it can cost from a few dollars to several million dollars to fix the problem. Clearly, testing and qualification of the correlator chip is crucial to the successful implementation of the correlator system.

It should be noted that the plan for testing the correlator chip outlined in this document has changed from that envisioned in the correlator Conceptual Design Review in November 2001. In that review, the plan was to implement a scaled-down version (i.e. fewer lags) of the chip in an FPGA, test the FPGA on the sky at the VLA site with EVLA antennas, and then proceed with implementation of the ASIC for production of the correlator. The plan outlined in this document has changed somewhat, eliminates the FPGA implementation and test, and proceeds directly to the ASIC implementation. The FPGA prototype testing was eliminated due to problems in getting even a scaled-down version of the FPGA to place and route at speed, power dissipation, packaging, ASIC conversion, and cost.

3 Overview

This section presents a brief overview of the plan for testing and verifying the functionality and performance of the correlator chip. Since the chip will be fabricated as a full custom or structure ASIC, testing and verifying the functionality and performance of the chip is important at all stages of development. For this reason, a rigorous procedure for testing the chip before moving to the next stage of implementation is contemplated.

The main phases/steps in testing and verifying the performance and functionality of the correlator chip are as follows:

1. Incremental RTL coding and testing of correlator chip modules.

2. Correlator chip RTL functional testing with test cases developed to exercise the chip in as many of its intended operating modes as possible. Comparison with behavioural simulation results.

3. Correlator chip RTL functional testing with random or psuedo-random inputs (random test bench).

4. Critical Design Review of the correlator chip design, RTL code, test bench, test cases, and test plan.

5. Correlator chip gate-level (post-place-and-route—PPAR) testing and verification with RTL functional test results.

6. Correlator chip PPAR testing and verification with the random test bench.

7. Physical prototype chip verification and testing.

8. Production chip verification and testing.

Each of these elements will be described in more detail in following sections.

Any vendor/manufacturing specific testing required in the chip implementation is beyond the scope of this test plan and is not included.

4 Pre-fabrication Test Plan

This section describes the test and verification plan for testing the chip before physical prototypes are fabricated. Rigour in this testing is absolutely essential to ensure that the prototypes function as intended.

4.1 Incremental RTL Coding and Testing

During the development of the RTL code for the chip, an incremental code and test strategy is used. This methodology is the first level of testing that, if performed rigorously, helps to ensure that further (functional) testing is only required to find obscure bugs that are the result of unanticipated interactions between functional blocks. There are no quantifiable data products from this testing, rather testing is done to the engineer's satisfaction for the particular function under test.

4.1.1 CMAM functionality qualification

An important module that dominates the silicon area and power dissipation of the chip, and is important for the fundamental operation of the correlator is the Complex Multiply-Accumulate Module or the "cmam". It is critically important that this module is producing correct results and thus extra effort must go into verifying that this is the case.

The synthesizable cmam module is compared within the simulator (Modelsim) with a non-synthesizable, but functionally equivalent module that exactly replicates its behaviour, but that is built using simple arithmetic equations. This behavioural model produces an output that must be exactly equivalent, at the bit level, with the synthesizable module that will eventually be implemented in silicon. This comparison is essential to ensure that there are no logic errors in the synthesizable module that could introduce systematic biases in the implemented correlator chip results. If iteration of the cmam is required during chip place-and-route (PAR) (e.g. to add or remove pipelining), then the behavioural model is updated accordingly and the test is re-run before the modified design is accepted.

4.1.2 Cross-clock domain signals

Any signals that cross from one clock domain to another (such as would be the case with microprocessor configuration and status), are clocked into the destination clock domain with 2 stages of flip-flops before entering the destination logic.

If more than one bit is clocked into another clock domain, and the destination domain contains a finite state machine (FSM), then logic is included to ensure that both bits change at the same time before entering the FSM.

4.2 Chip functional testing (test case generation and verification)

A top-level, sophisticated test bench allows configuration and testing of the chip in as many different modes as can be envisioned. The plan is to ensure that the test cases/configurations that are created will encompass any possible configurations created during the lifetime of the chip, and to ensure that as close as possible to 100% fault coverage (i.e. exercising 100% of the logic in the chip) is achieved.

A standard set of test vectors for X and Y input data streams are used to provide the chip with data for each test. Each set of test vectors has a recognizable result in the lag and frequency domains. A Windows-based GUI and a MatLab analysis program (developed by Aardvark Resources) facilitate verification that each test case is set up as intended, and to facilitate comparison of data produced by the correlator chip simulator with the behavioural simulator described in section 4.2.1 below.

The output files from this functional correlator chip RTL simulation will be frozen and saved as "golden files" for bit-by-bit comparison with the correlator chip PPAR simulations. This comparison is a crucial step that ensures that all functional and timing requirements have been met in the ASIC implementation of the design.

It is possible that the final RTL simulation and the resulting final golden files may not be completed until such time as the ASIC vendor indicates that there are no timing/power issues in PAR, and that the RTL design can be frozen.

In addition, the test bench is built so that any inputs that should not matter to the chip for the particular configuration and for any particular instants in time are set to "don't cares". This ensures that if signals entering the chip "leak" into unexpected places and cause errors, it will show up in the Modelsim wave window and the output files.

Refer to document A25082N0002 (TVP: EVLA Correlator Chip Functional Test Cases) by Aardvark Resources for a detailed description of all functional test cases.

Refer to the appendix of this document for further information on functional testing.

4.2.1 Behavioural simulation and comparison with RTL simulation

A behavioural simulator written in C[1], and derived from the simulation code used to study the behaviour of the critical signal processing elements of the correlator (NRC EVLA Memo# 001) is the "gold standard" against which correlator chip simulations are compared.

The fundamental code for the correlator is very simple and it is possible to verify the correctness of its output by inspection and with the knowledge of what the fundamental output of the correlator should be from first principles. (This code is derived from code

[1] Refer to Appendix V for a complete listing of this code.

originally developed for testing in the development and successful implementation of the space VLBI correlator [2].)

The test vectors that are fed into the behavioural simulator have two possible sources. The first source is from a C program that produces a file of test vectors. This same file is fed into the correlator chip test bench. The second source is from the correlator chip test bench: the *exact* set of test vectors that went into the correlator chip simulation are written to a file that can then go into the behavioural simulator. With proper structuring of the data valid bits to eliminate unimportant differences in cmam pipeline delays between the C code and the RTL code[2], this test allows an exact bit-by-bit comparison of the C behavioural simulation with the correlator chip simulation, verifying that they produce *exactly* the same result.

The comparison of the C behavioural simulation results with the correlator chip simulation forms a solid foundation to ensure that the correlator chip functions as it should.

4.3 Random/quasi-random input testing

A separate test bench generates a combination of random and quasi-random data on all of the inputs to the chip. This test bench can be set for purely random inputs or random inputs on data lines, and quasi-random inputs on control lines that allow the chip to produce some output data frames. This test is used to find bugs in FSMs that could cause the chip to hang up. This test produces no useful output that can be compared with anything. Determination of hung conditions and/or don't cares is made by observing the Modelsim wave window output.

4.4 Critical Design Review

A Critical Design Review of the correlator chip will be held after the correlator chip ASIC vendor is chosen but before final synthesis and PAR of the chip commences.

The review committee will consist of a chairman and internal and external reviewers and a final short report will be generated by the review committee before final sign-off of the chip for implementation by the ASIC vendor.

It should be noted that much work has already been done by more than one vendor in synthesis, PAR, and analysis of the design and so there should be no major surprises

[2] i.e. blank data valid a short period of time before and after a dump command (dump_sync) occurs so that differences in pipelining in the cmam between the RTL code and the C code are factored out.

coming out of this review. This review is used as a final critical look at design, testing, and power, speed, and packaging issues.

This review consists of three distinct parts and different sets of people will be required for each part.

4.4.1 Detailed design/code walk-through

This is a line-by-line walk-through of each RTL module with the intent of trying to find coding faults that could cause unusual or problematic behaviour in the actual device but that may not show up in any simulations. For example, a FSM with a missing or unaccounted for state could cause intermittent lock-up of the chip, but may not be found in any simulations.

This walk-through is expected to take 2 days and attendance by the following people is essential:

- Correlator chip engineer (B. Carlson)
- ASIC chip vendor engineer (TBD).
- External engineer from NRAO (TBD—optional)

4.4.2 Detailed test case review

This is a review of all of the functional test cases, the random test bench, and all test results including comparisons with the behavioural simulation. Reviewers should have a good understanding of the functionality of the correlator chip, and the functionality of the correlator system.

This part of the review is expected to take 1 or 2 days and attendance by the following people is essential:

- Correlator chip engineer (B. Carlson).
- ASIC chip vendor engineer (TBD).
- External engineer from NRAO (TBD).
- Test-case developer (R. Smegal).
- EVLA project scientist(s) (R. Perley, M. Rupen)

4.4.3 Review of vendor analysis, and final wrap-up

This is a review of the preliminary work done by the vendor in analysis of the correlator chip design for feasibility in terms of cost, power, speed, packaging, and reliability as well as a final wrap-up/summary of the first 2 parts of the review.

This part of the review is expected to take 1 day, and attendance by the following people is anticipated:

- All people in attendance thus far.
- EVLA Correlator project manager (P. Dewdney).
- EVLA project manager (P. Napier).

4.5 Post-place-and-route (PPAR) qualification and gate-level simulation

The ASIC vendor will perform PPAR static timing analysis to find any timing violations. The results of this analysis, as well as the characterization of some timing paths as false paths or multi-cycle paths must be carefully reviewed to ensure that no errors exist. This review will likely require a meeting between the ASIC vendor and the EVLA correlator chip design engineer.

Once PAR is complete, the ASIC vendor generates the gate-level netlist (.vo file) and the timing file (.sdf file) that are used for simulation. PPAR simulation is performed by a contractor (Aardvark Resources) or the EVLA correlator chip designer, and the outputs are compared with the RTL simulation golden files. Any differences in the output files indicate a timing or synthesis problem that must be fixed before final approval to build the chip can be given.

Due to potentially excessive simulation times, it may not be possible to perform a gate-level simulation on every single test case run for the RTL simulation, and thus one or maybe a few test cases that cover as much functionality of the chip should be run for comparison with the RTL golden files.

The random test bench must be run on the PPAR results to ensure that no lock-up or don't care conditions occur in the final design. Analysis is the same as the RTL simulation—observation of the Modelsim wave window results.

The IBIS or spice models for the I/O of the chip should be analyzed using Signal Vision or IS_analyzer to ensure that there are no board-level communication problems—although this will not likely be the case since everything is point-to-point LVTTL over very short distances and with one load.

4.5.1 Cross-clock domain signals

There is a potential problem in the PPAR (gate-level) simulation that will not show up in the actual chip or in the RTL simulation. This can occur when the two clock domains are not frequency synchronized and is due to the way Modelsim deals with internal setup and hold timing violations.

When an internal setup or hold violation occurs, Modelsim sets the output to a "don't care" state until the setup or hold violation is fixed on the next or subsequent clock cycle. If this "don't care" propagates into a FSM, then the FSM state is undefined and it will never recover without a reset.

This effect does not occur in the RTL simulation because there are no setup or hold requirements. This effect does not occur in the actual chip implementation if the design procedure outlined in section 4.1.2 is followed since the signal into the FSM will always be a 0 or a 1, and the FSM will operate properly on either case.

One way to work around this problem is to set the clocks in the test bench so that they are frequency synchronous and so that when a signal propagates into the other clock domain it most likely will not have a setup or hold-time violation even though there is no static timing analysis on that particular path. In the correlator chip, this means setting the MCB_CLK input to 32 MHz rather then 33 MHz. This workaround is no guarantee, though, and may require tweaking the 32 MHz clock phase in the test bench to avoid this condition.

5 Prototype Test Plan

A minimum of 200 correlator chip prototypes must be fabricated to meet the requirements for chip testing and prototype correlator testing. This is enough chips to populate 3 Baseline Boards. One Baseline Board will eventually be shipped to the VLA-site for testing on the sky where it will stay for further testing, one board will be shipped to the UK for e-MERLIN, and one will stay in Penticton for continued testing and software development.

It is anticipated that for cost and risk reasons, only one Baseline Board will be initially populated and tested. If testing is successful, the other two boards will be populated, otherwise a board re-spin will be done before continuing with the population and testing of the fixed board.

There are three components of testing the correlator chip prototypes described in the following sub-sections.

5.1 Test case verification

The goal of this testing is to ensure that the prototype chips are performing as expected, at speed, using the same set of test vectors as was used for the functional simulation and the PPAR simulation.

The baseline plan is that the prototype chip test board will be the prototype Baseline Board with vectors originating from the prototype Station Board. These Station Board vectors originate as vectors loaded into the Delay Module memory. The vectors loaded can be anything, and it is likely that they will be the same or similar vectors to what was used for the RTL and PPAR simulations.

Since all high-speed connections to each correlator chip are point-to-point (except for the low-speed MCB bus), it is likely that a separate test board for the correlator chip will not be necessary. However, it is *possible* to build a special test board with an FPGA to stimulate the correlator chip under test, but this incurs additional hardware and software engineering time that we can ill afford.

In principle it should be possible to obtain *exactly* the same results from this test as from the functional simulation, although in practice this may be difficult and time consuming to do because of the long times between time-tick markers in the real system versus what is used in the simulations. Every time there is a time-tick (SCHID_FRAME_), data gets replaced with ID codes and is invalid data for correlation. In the real system, this time tick is generated once every 10 milliseconds, whereas in the simulations it is typically generated every few tens or hundreds of microseconds. Thus, the accumulated data, and data valid counts will be slightly different in the two cases, precluding a direct bit-by-bit comparison. However, statistical comparison of floating point data products in both the lag and frequency domains should yield extremely close results and should be sufficient to qualify the prototype chips for this stage of testing.

5.2 "Sky" testing at the VLA-site

This will be the first time that data from the antennas will be processed in the correlator chip and successful completion and qualification of the chip at this stage is crucial for the entire project.

A number of test cases will be developed to ensure that the chip is functioning properly and that there are no functional or performance problems. Among them, deep integrations will be performed in the configurations most often required by the correlator during EVLA observations.

Advice from NRAO VLA science and operations people will be required to develop the test cases, and these test cases will be detailed in a separate correlator prototype test document.

The deep integration times and the configurations tested should be approximately at the 95^{th} percentile or better. For proper qualification, phase and amplitude closure tests and image processing of deep integrations will be performed to try to find systematic errors that may be present in the correlator chip due to incorrect functionality or performance problems. Although there is no model to compare the results with (except for perhaps the old VLA correlator) this is the final important qualifying test before the correlator chip is approved for production.

5.3 Environmental testing

This testing is used to ensure that the chip meets all of its environmental operating requirements. This includes timing into and out of the chip, operating temperature, power dissipation, clock speed etc. This test will involve measurements, burn-in, temperature cycling, vibration testing, and increasing the clock speed somewhat to determine the chip's performance robustness. The parameters for this testing are TBD and will involve obtaining the ASIC vendor's recommendations for this type of testing.

6 Production Test Plan

At least 12000 and perhaps as many as 17000 correlator chips will be produced for the full correlator build. These chips are tested by the ASIC manufacturer using "scan-insertion" testing techniques that test each die before it is packaged. We will thus take delivery of chips that are tested to ensure that all internal gates are switching.

There is no plan to test individual packaged chips before being soldered to destination Baseline Boards. Thus, any testing of production correlator chips falls under the scope of Baseline Board testing. Generally, this means JTAG and functional testing before the boards are accepted from the manufacturer, and functional and burn-in testing at our site before shipment to their final destination.

7 References

[1] Carlson, B. REQUIREMENTS AND FUNCTIONAL SPECIFICATION, EVLA Correlator Chip, A25082N0000, Revision 1.3, June 15, 200.

[2] Carlson, B.R., Dewdney, P.E., Burgess, T.A., Casorso, R.V., Petrachenko, W.T., Cannon, W.H., The S2 VLBI Correlator: A Correlator for Space VLBI and Geodetic Signal Processing, Publications of the Astronomical Society of the Pacific, 1999, 111, 1025-1047.

8 Appendix I – Functional test case example

This appendix includes an example of a functional test case used to verify that the correlator chip is functioning properly in one of its intended configurations. For more detailed information on the operation of the correlator chip, refer to [1].

```
// ********************************************************************************************************************
// File name : tc14_jul17-04_v2.v
// Project   : Correlator chip (25082) CORRCHIP_25082_TB_LIB
// Version   : V2
// Company   : National Research Council of Canada
//                  P.O. Box 248
//                  Penticton, BC, V2A 6K3
// Author    :  Brent Carlson
//                  phone: 250-490-4346
//                  email: Brent.Carlson@nrc.ca
//
// Contributors :  Rick Smegal
//
//
// Design description:
// This file contains a single testcase for the top-level correlator chip testbench.
// This test is based on the original work by
// Brent Carlson and Frances Lau.
//
//
// ********************************************************************************************************************
// ************************    REVISION HISTORY    ********************************************************************
// ********************************************************************************************************************
// FILE NAME              DATE            VERSION          COMMENTS
// tc14_mar13-04_v1.v     Mar 13, 2004    1                creation
// tc14_jul17-04_v2.v     Jul 17, 2004    2                new test bench, SE_CLK mod 2
//
//
//
// ********************************************************************************************************************
//
//
// ********************************************************************************************************************
// ********************************************************************************************************************
//                                  BEGINNING OF TEST CASE GENERATION/INITIALIZATION
// ********************************************************************************************************************
// ********************************************************************************************************************
//

// Notation for correlator cell concatenation : inpX(begin_cell:end_cell)inpY
// Cells not listed are disabled by an undefined setting of CCC data switches;
// a hyphen indicates inputs with 'no-data';
// inputs not shown have data valids reset (0) unless otherwise indicated.

tc = 14;
test_description[tc] =
"# ------------------------------------------------------------------------------
# TEST 14:
# ver: v2,04jul17
# Synopsis:  CONCAT AS QUADS, INP 0,1 ALL POLN PRODUCTS, MOD 2 CLK, 3 DUMPS ON X
# Description:
#        Purpose of Test:      all poln products with quad concatenation
#        Concatenation:        0(0:3)0, 0(4:7)1, 1(8:11)0, 1(12:15)1
#        Num of Dumps-input:   3-0, 3-1
#        Dump Control:         X
#        Num samples/dump-inp: 20000-0, 20000-1
#       Distinct samp/dump:   10000-0, 10000-1
#        Active SE clocks:     0, 1
#        Modulo se_clk gen:    2-0, 2-1
#        Compare results with: behavioural simulation test vectors for data set DS4
#        Additional details:
#
# ------------------------------------------------------------------------------";

//--------------------------------------------------------------------------------------------------------------------
```

```
//                    MCSR register:
//                    ------------------------------------------------------
//                    X_Y_DS | SyncER | TvER | AOV | OVR | PhEN | TVEN | CEN
//                    ------------------------------------------------------
//                      B7       B6      B5    B4    B3     B2     B1     B0
mcsr_mem[tc] = 8'b______1________0_______0_____0_____0______1______0______1;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//                      CCCSCR Input Selection:
//                      00 = adjacent CCC
//                      01 = primary CCC input
//                      10 = secondary CCC input
//                      11 = undefined: no lag frames produced
//
//                         CCC0,1 switch config register: (same for CCC2...CCC15)
//                        |--------- CCC1 ----------|---------- CCC0 -----------|
//                        ---------------------------------------------------------
//                        Y-S1 | Y-S0 | X-S1 | X-S0 | Y-S1 | Y-S0 | X-S1 | X-S0
//                        ---------------------------------------------------------
//                         B7     B6     B5     B4     B3     B2     B1     B0
cccscr_0_1_mem[tc] = 8'b___0______0______0______0______0______0______0______1;
cccscr_2_3_mem[tc] = 8'b___1______0______0______0______0______0______0______0;
cccscr_4_5_mem[tc] = 8'b___0______0______0______0______0______0______0______1;
cccscr_6_7_mem[tc] = 8'b___0______1______0______0______0______0______0______0;
cccscr_8_9_mem[tc] = 8'b___0______0______0______0______0______0______1______0;
cccscr_10_11_mem[tc] = 8'b_1______0______0______0______0______0______0______0;
cccscr_12_13_mem[tc] = 8'b_0______0______0______0______0______0______1______0;
cccscr_14_15_mem[tc] = 8'b_0______1______0______0______0______0______0______0;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//                       CCQR Input Selection:
//                            0 = primary
//                            1 = secondary
//
//                       CCQ0,1 switch config register
//                       ---------------------------------------------------------------------------------
//                       Q1-YSw1 | Q1-YSw0 | Q1-XSw1 | Q1-XSw0 | Q0-YSw1 | Q0-YSw0 | Q0-XSw1 | Q0-XSw0
//                       ---------------------------------------------------------------------------------
//                           B7       B6         B5       B4        B3        B2        B1        B0
ccqr_0_1_mem[tc] = 8'b_______1________1__________1________1_________0_________0_________0_________0;
ccqr_2_3_mem[tc] = 8'b_______1________1__________1________1_________1_________1_________1_________1;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//                     shift enable clock(X,Y-SE_CLK) generation for each test, for each input.
//                     On (1) or off (0).
//
//                     Input line number
//                     ------------------------------------------------------------
//                        7   |   6   |   5   |   4   |   3   |   2   |   1   |   0   |
//                     ------------------------------------------------------------
x_se_clk_mem[tc] = 8'b_____0_______0_______0_______0_______0_______0_______1_______1;
y_se_clk_mem[tc] = 8'b_____0_______0_______0_______0_______0_______0_______1_______1;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//                Modulo shift enable clock (X,Y-SE_CLK) generation for each test, for each input.
//                On (1) or off (0).
//                Is only looked at if the corresponding bit in x/y_se_clk_mem for the same test is set (1).
//                If modulo generation is on, then the se_clk is on when: !(samplenum % [input_number+2]).
//                However, this can be overridden by set_modulo_mode_mem below.
//                E.g. se_clk_0 would be on every other clock cycle, se_clk_1 would be on every third clock cycle etc.
//                This could result in the detection of input synchronization errors since a given se_clk may not
//                be 1 when schid_frame is asserted.
//
//            modulo X,Y-SE_CLK generation is on (1) or off (0).
//
//                     Input line number
//                     ------------------------------------------------------------
//                        7   |   6   |   5   |   4   |   3   |   2   |   1   |   0   |
//                     ------------------------------------------------------------
x_se_clkmod_mem[tc] = 8'b__0_______0_______0_______0_______0_______0_______1_______1;
y_se_clkmod_mem[tc] = 8'b__0_______0_______0_______0_______0_______0_______1_______1;
```

```
//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

// If modulo generation is on and this bit is reset (0), then modulo 2 clock
// generation is used no matter what the input number.  If this bit is set (1),
// then modulo (input_number+2) generation is used.
//
set_modulo_mode_mem[tc] = 0;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

// For each test, the contents of this memory is ANDed with the data valid lines
// for inputs X and Y.
//
//                      Input line number
//                      ---------------------------------------------------------------
//                         7   |   6   |   5   |   4   |   3   |   2   |   1   |   0   |
//                      ---------------------------------------------------------------
x_dvalid_mem[tc] = 8'b_____0_______0_______0_______0_______0_______0_______1_______1;
y_dvalid_mem[tc] = 8'b_____0_______0_______0_______0_______0_______0_______1_______1;

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

// This block defines temporary registers to hold data that
// is to be transmitted serially on one or more X,Y-DUMP_EN lines and assigns this data
// to a register corresponding to an active X,Y-DUMP_EN line.
// Different data may be assigned to each line, if required.
// The length of the serial sequence for dumpen and timestamp is defined by the
// parameter dts_len, and currently dts_len = 72.
// So, there are 72 bits total...here

fill_sync = 2'b01;                // 2 bits

// Dump control bits:
//       000 = First dump of data into LTA. Just save data in LTA bin.
//       001 = Add to existing LTA data and save in LTA.
//       010 = Last dump: add data to LTA, and flag data as ready for readout.
//       011 = Speed dump: bypass LTA directly to output. Data bias removed.
//       100 = Correlator chip dumps, discards data, clears registers.
//       1X1 = Reserved.
//       11X = Reserved.

dump_command = 3'b010;            // 3 bits: last dump

// Phase bin number that the data gets dumped/accumulated into.
//       Note: PB15 is the bank number (0 or 1)
//             With speed dumping, all 16 PB bits can be used for a total of 65,536 bins.

phase_bin = 55666;                // 16 bits
recirc_blk = 233;         // 8 bits
fill3 = 3'b111;            // 3 bits

// Holdoff held high to disable correlation until lag shift registers
// fill up with data.
holdoff = 40'b0000000000000000000000000000000000001111;     // 40 bits

// Either X or Y controls dumping, as defined by X/Y_DS (X/Y dump select) in the MCSR.
// Here, X controls dumping.
// dump_en is transmitted LSB first.
x_dumpen0_mem[tc] = {holdoff,fill3,recirc_blk,phase_bin,dump_command,fill_sync};
x_dumpen1_mem[tc] = {holdoff,fill3,recirc_blk,phase_bin,dump_command,fill_sync};

// these are dummies
x_dumpen2_mem[tc] = {70'bx,2'b0};
x_dumpen3_mem[tc] = {70'bx,2'b0};
x_dumpen4_mem[tc] = {70'bx,2'b0};
x_dumpen5_mem[tc] = {70'bx,2'b0};
x_dumpen6_mem[tc] = {70'bx,2'b0};
x_dumpen7_mem[tc] = {70'bx,2'b0};

//-------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

// This block defines temporary registers to hold the TIMESTAMP data and assigns
// this data to a register corresponding to the X,Y-TIMESTAMP data line.

// Timestamps
ts_fillsync = 2'b01;
ts_0 = 222333444;
```

```
ts_1 = 555666777;
ts_endfill = 0;
x_timestamp_mem[tc] = {ts_endfill,ts_1,ts_0,ts_fillsync};

// The Y timestamp is ignored because X controls dumping.
ts_fillsync = 2'b01;
ts_0 = 'hffffffff;
ts_1 = 'hffffffff;
ts_endfill = 0;
y_timestamp_mem[tc] = {ts_endfill,ts_1,ts_0,ts_fillsync};

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// These registers contains the dump period in 128 MHz clock cycles. All tests
// start with a dump and discard immediately after reset so that everything gets
// into a known state.
// The facility to set the dump period independently for each DUMP_EN line
// was added Feb. 5, 2004.
// NOTE: IT IS UP TO THE USER TO ENSURE THAT THERE
//    ARE NO "COLLISIONS"...THUS, ALL OF THE DUMP_PERIOD_MEM(0...7) MUST BE
//    HARMONICALLY RELATED.

// DEFAULT for tests in 128 MHz clock cycles is DUMP_PERIOD=5000

dump_period_mem0[tc] = 10000;
dump_period_mem1[tc] = 10000;
dump_period_mem2[tc] = 10000;
dump_period_mem3[tc] = 10000;
dump_period_mem4[tc] = 10000;
dump_period_mem5[tc] = 10000;
dump_period_mem6[tc] = 10000;
dump_period_mem7[tc] = 10000;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Set how often, in 256 MHz clock cycles, the X,Y-SCHID_FRAME pulses are generated.
// **MUST** BE 512 * 2^n
// DEFAULT for tests in 256 MHz clock cycles SCHID_PERIOD=512

schid_period_mem[tc] = SCHID_PERIOD;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Define memory that allows us to generate frame abort tests.  If the mem is
// set (1), then every other dump for each CCC is aborted partway through
// transmission.

frame_abort_mem[tc] = 0;  // set (1) to abort CCC first dump frames

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Generate input sync errs on the X and Y data at 256 MHz.  If bit in this memory is set (1) then
// error generation occurs.  If reset (0) then no error generation.

sync_err_mem[tc] = 0;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Set how often errors are generated for each test.
// If TVEN=1, then this is how often in samples.  If TVEN=0, then
// this is how often in terms of the input synchronization pattern (i.e. every
// sync_err_mod_mem sync patterns, the error is generated).

sync_err_mod_mem[tc] = 0;


//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Set how much the Y dump_sync and dump_en is later than the X dump_sync and dump_en
// in terms of 128 MHz clock cycles.

y_dump_offset_mem[tc] = 3;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------
```

```
// Set how much the Y schid_frame is offset from the X schid_frame
// in terms of 128 MHz clock cycles.

y_schid_offset_mem[tc] = 6;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Define X and Y station IDs for each test.
// define embedded identifiers...make them something easily recognized.

x_station_id_mem[tc] = 85;
y_station_id_mem[tc] = 170;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// Define X and Y sub-band IDs BASES.  The actual sub-band ID is this plus the
// input number (0...7)

x_sbid_mem[tc] = 0;
y_sbid_mem[tc] = 0;

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

// The number of 128 MHz clock cycles that the simulation runs for.  Once
// these number of clock cycles have elapsed, the LTA Controller emulator clears
// config_ready, waits for the remaining data frames to be read out and stored, and then
// tells the MCB interface emulator that it can RESET_pad the chip, load a new
// configuration, and then re-assert config_ready for the next test.
//
// NOTE: The first integration (dump interval) does not start until after the
// data has been synchronised and the first dump and discard occurs.
// This time is approximated in 128 MHz clock cycles by:
// (mcb_cen_set_wait*schid_period/512+128)*128/33 + first_dump_start + dts_len)
// This delay is approximately 2200 clock cycles for mcb_cen_set_wait=410,
// schid_period=512,first_dump_start=20, dts_len=72
// The time requred for configuration of the MCSR is not included in the above and
// additional time should allowed in setting the TEST_DURATION.

// DEFAULT test duration for tests in 128 MHz clock cycles is TEST_DURATION=10000

test_duration_mem[tc] = 35000; // dump period + additional 5000 cycles

//-----------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------

$display("---Test Case %d initialized",tc);

// ***********************************************************************************************************
```

9 Appendix II – Functional test results example

Examples of the "golden file" outputs produced by the functional test bench are presented below. The first example is the MCB configuration register file that is created near the end of test case execution when the configuration and status registers on the chip are read out and then saved in an ASCII format. The second example is lag frame data output acquired by the test bench from the correlator chip and written to an ASCII file.

MCB configuration/status register file example

```
# TEST 14:
# ver: v2,04jul17
# Synopsis:  CONCAT AS QUADS, INP 0,1 ALL POLN PRODUCTS, MOD 2 CLK, 3 DUMPS ON X
# Description:
#    Purpose of Test:      all poln products with quad concatenation
#    Concatenation:        0(0:3)0, 0(4:7)1, 1(8:11)0, 1(12:15)1
#    Num of Dumps-input:   3-0, 3-1
#    Dump Control:         X
#    Num samples/dump-inp: 20000-0, 20000-1
#    Distinct samp/dump:   10000-0, 10000-1
#    Active SE clocks:     0, 1
#    Modulo se_clk gen:    2-0, 2-1
#    Compare results with: behavioural simulation test vectors for data set DS4
#    Additional details:
#
# ------------------------------------------------------------------------------
10000101  MCSR: X/Y_DS=1 SyncER=0 TvER=0 AOV=0 OVR=0 PhEN=1 TVEN=0 CEN=1

00000001  CCCSCR_0_1:   CCC1: Y=adj X=adj   CCC0: Y=adj X=pri
10000000  CCCSCR_2_3:   CCC3: Y=sec X=adj   CCC2: Y=adj X=adj
00000001  CCCSCR_4_5:   CCC5: Y=adj X=adj   CCC4: Y=adj X=pri
01000000  CCCSCR_6_7:   CCC7: Y=pri X=adj   CCC6: Y=adj X=adj
00000010  CCCSCR_8_9:   CCC9: Y=adj X=adj   CCC8: Y=adj X=sec
10000000  CCCSCR_10_11: CCC11: Y=sec X=adj CCC10: Y=adj X=adj
00000010  CCCSCR_12_13: CCC13: Y=adj X=adj CCC12: Y=adj X=sec
01000000  CCCSCR_14_15: CCC15: Y=pri X=adj CCC14: Y=adj X=adj

11110000  CCQR_0_1: Q1: YSw1=sec YSw0=sec XSw1=sec XSw0=sec  Q0: YSw1=pri YSw0=pri XSw1=pri XSw0=pri
11111111  CCQR_2_3: Q3: YSw1=sec YSw0=sec XSw1=sec XSw0=sec  Q2: YSw1=sec YSw0=sec XSw1=sec XSw0=sec

00000000  XSTATUS: X-SDATA1: 0000, X-SDATA0: 0000
00000000  YSTATUS: Y-SDATA1: 0000, Y-SDATA0: 0000
00000000  XSTATUS: X-SDATA3: 0000, X-SDATA2: 0000
00000000  YSTATUS: Y-SDATA3: 0000, Y-SDATA2: 0000
00000000  XSTATUS: X-SDATA5: 0000, X-SDATA4: 0000
00000000  YSTATUS: Y-SDATA5: 0000, Y-SDATA4: 0000
00000000  XSTATUS: X-SDATA7: 0000, X-SDATA6: 0000
00000000  YSTATUS: Y-SDATA7: 0000, Y-SDATA6: 0000
00000000  XSTATUS: X-PHASE1: 0000, X-PHASE0: 0000
00000000  YSTATUS: Y-PHASE1: 0000, Y-PHASE0: 0000
00000000  XSTATUS: X-PHASE3: 0000, X-PHASE2: 0000
00000000  YSTATUS: Y-PHASE3: 0000, Y-PHASE2: 0000
00000000  XSTATUS: X-PHASE5: 0000, X-PHASE4: 0000
00000000  YSTATUS: Y-PHASE5: 0000, Y-PHASE4: 0000
00000000  XSTATUS: X-PHASE7: 0000, X-PHASE6: 0000
00000000  YSTATUS: Y-PHASE7: 0000, Y-PHASE6: 0000
00000000  XSTATUS: X-DVALID: 00000000
00000000  YSTATUS: Y-DVALID: 00000000
11111100  XSTATUS: X-SE_CLK: 11111100
11111100  YSTATUS: Y-SE_CLK: 11111100
00000000  XSTATUS: X-DUMP_EN: 00000000
00000000  YSTATUS: Y-DUMP_EN: 00000000
00000000  XSTATUS: X-TIMESTAMP=0 X-SCHID_SYNC=0 X-DUMP_SYNCs=0
00000000  YSTATUS: Y-TIMESTAMP=0 Y-SCHID_SYNC=0 Y-DUMP_SYNCs=0

00000000  DESSR_0_7: 00000000
00000000  DESSR_8_15: 00000000
```

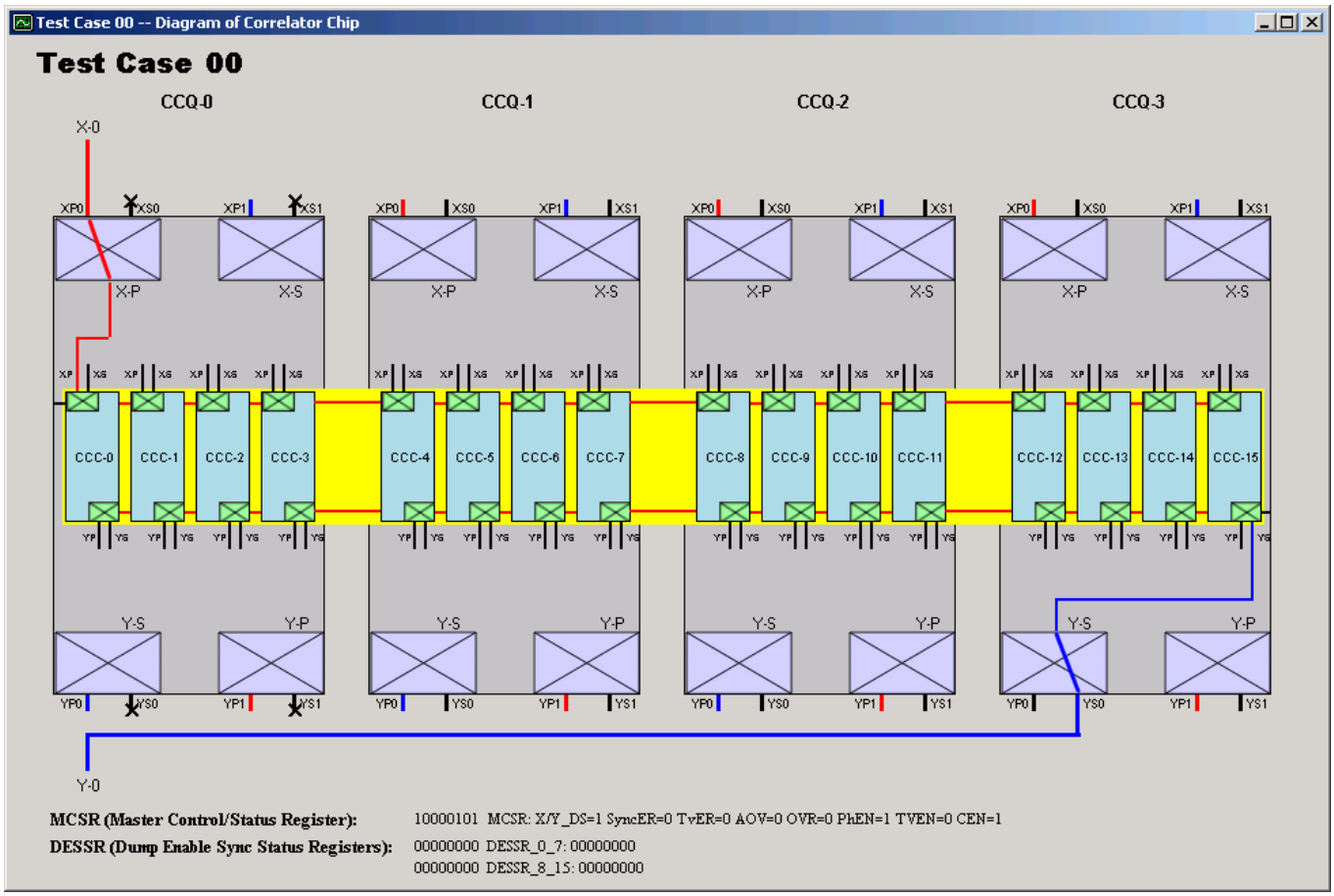
Lag frame data output file example

```
# TEST 14:
# ver: v2,04jul17
# Synopsis:  CONCAT AS QUADS, INP 0,1 ALL POLN PRODUCTS, MOD 2 CLK, 3 DUMPS ON X
# Description:
#       Purpose of Test:      all poln products with quad concatenation
#       Concatenation:        0(0:3)0, 0(4:7)1, 1(8:11)0, 1(12:15)1
#       Num of Dumps-input:   3-0, 3-1
#       Dump Control:         X
#       Num samples/dump-inp: 20000-0, 20000-1
#        Distinct samp/dump:   10000-0, 10000-1
#       Active SE clocks:     0, 1
#       Modulo se_clk gen:    2-0, 2-1
#       Compare results with: behavioural simulation test vectors for data set DS4
#       Additional details:
#
# ------------------------------------------------------------------------------
aaaaaaaa     W0: START SYNC WORD -- OK
e0010032     W1: B31=ASIC[1] Yin=1 Xin=1 YSyner=0 XSyner=0 ACC_OV=0 OVR=0 Rsrv=000000 NUM_CLAGS= 128 CCC= 6 Cmmd=010
c1aae055     W2: BBID-Y=6 SBID-Y= 1 SID-Y=170 BBID-X=7 SBID-X= 0 SID-X= 85
d97200e9     W3: LTA/Phase bin=55666 Recirc_blk-Y=  0 Recirc_blk-X=233
0d408a04     W4: TIMESTAMP-0= 222333444
211ecd59     W5: TIMESTAMP-1= 555666777
00004578     W6: DVCOUNT-Cntr=     17784
0000454f     W7: DVCOUNT-Edge=     17743
00134380     W8: DATA_BIAS   =   1262464
00135678     Lag    0 =    1267320
001345f6     Lag    0 =    1263094
00135e5e     Lag    1 =    1269342
001340b8     Lag    1 =    1261752
001359d4     Lag    2 =    1268180
0013376c     Lag    2 =    1259372
001357a8     Lag    3 =    1267624
0013333c     Lag    3 =    1258300
.
.
.
00135e36     Lag  127 =    1269302
00134936     Lag  127 =    1263926
1c71c71c     W265: END SYNC WORD -- OK
83f1011a     W266: Checksum calculated OK
```

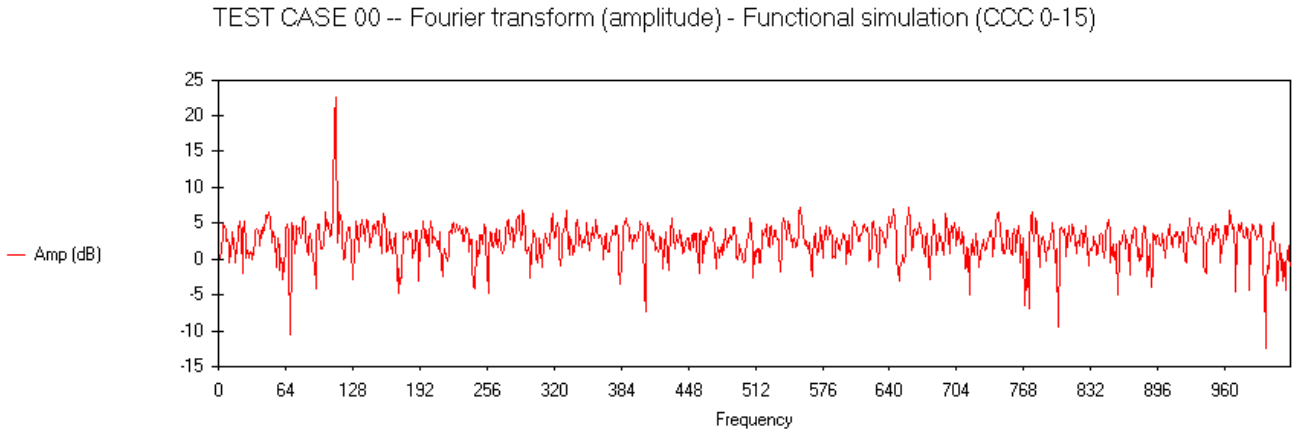
10 Appendix III – GUI analysis of functional test example

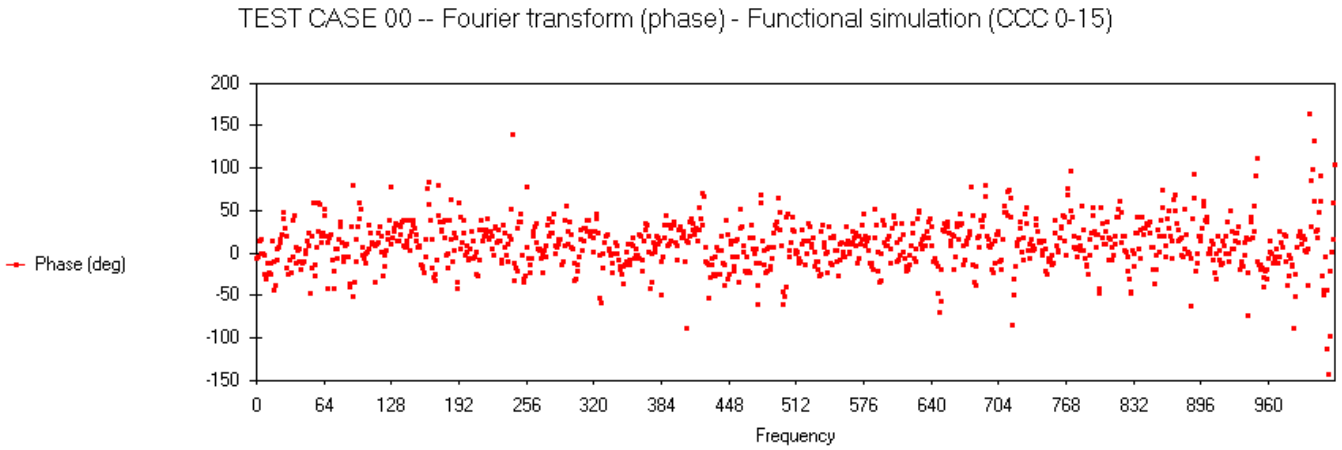
This appendix shows some example output from the Windows-based GUI that was built to test the correlator chip test bench output.

This window shows a diagram of the correlator chip indicating what inputs are active, how the input data is routed to the correlator chip cells (CCCs), and how the cells are concatenated (grouping shown in yellow boxes).



These two images show the correlator chip output in amplitude and phase versus frequency. In this case, there are 1024 spectral channels since all of the CCCs in the chip have been concatenated for one correlation. In the data, there is one spectral line and a continuum correlation.



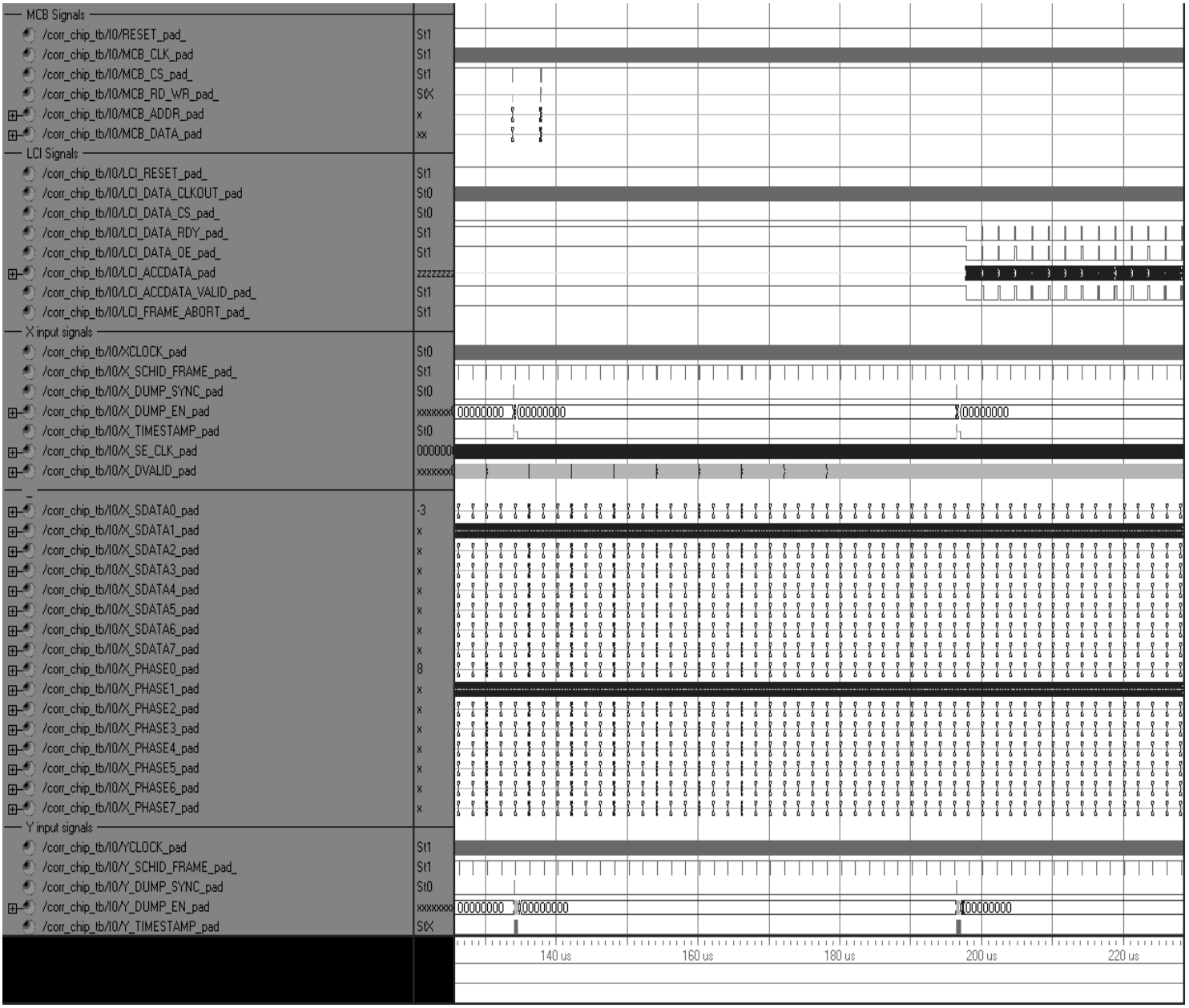


11 Appendix IV – Modelsim wave window output examples

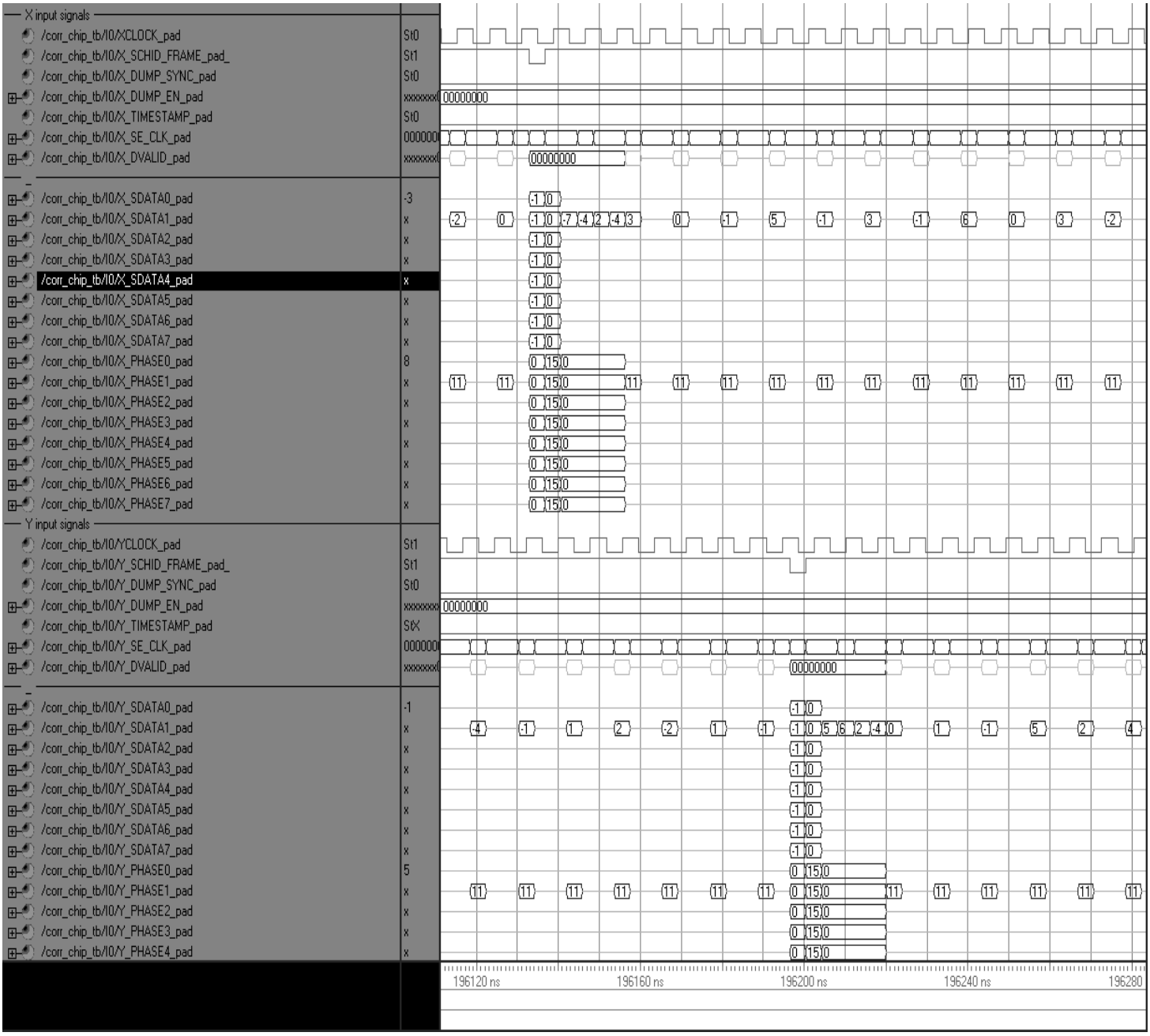
This appendix contains some examples of Modelsim wave window displays that show how the inputs are being stimulated in both the functional test case simulation and the random input simulation.

Functional test case simulation examples.

This is a "wide-angle" view of one functional simulation that shows the input stimuli and the resulting output data transfers on the LTA Controller Interface (LCI).

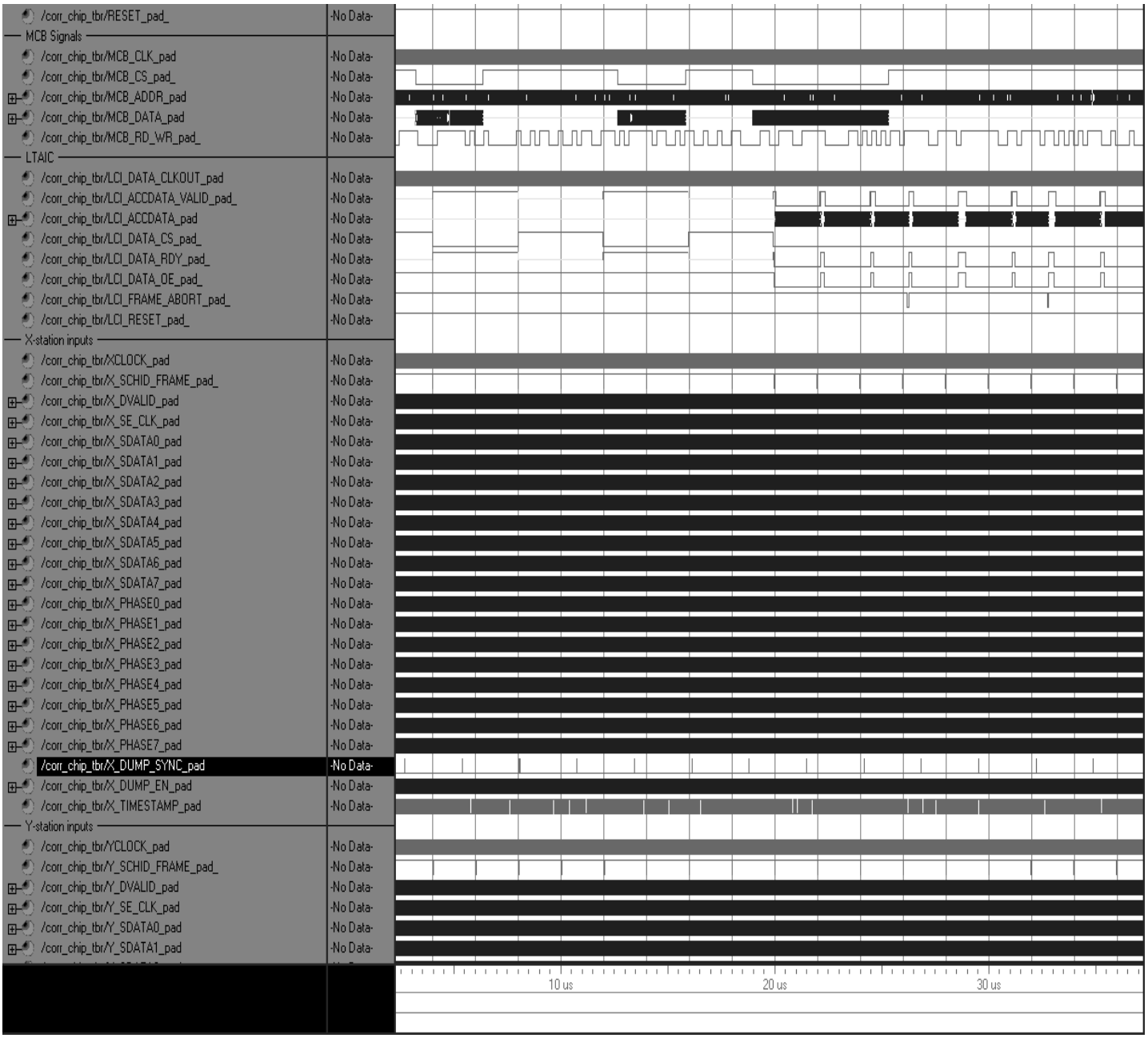


This is a zoomed-in picture showing the input stimuli, showing that when an input doesn't matter it is set to a don't care state.

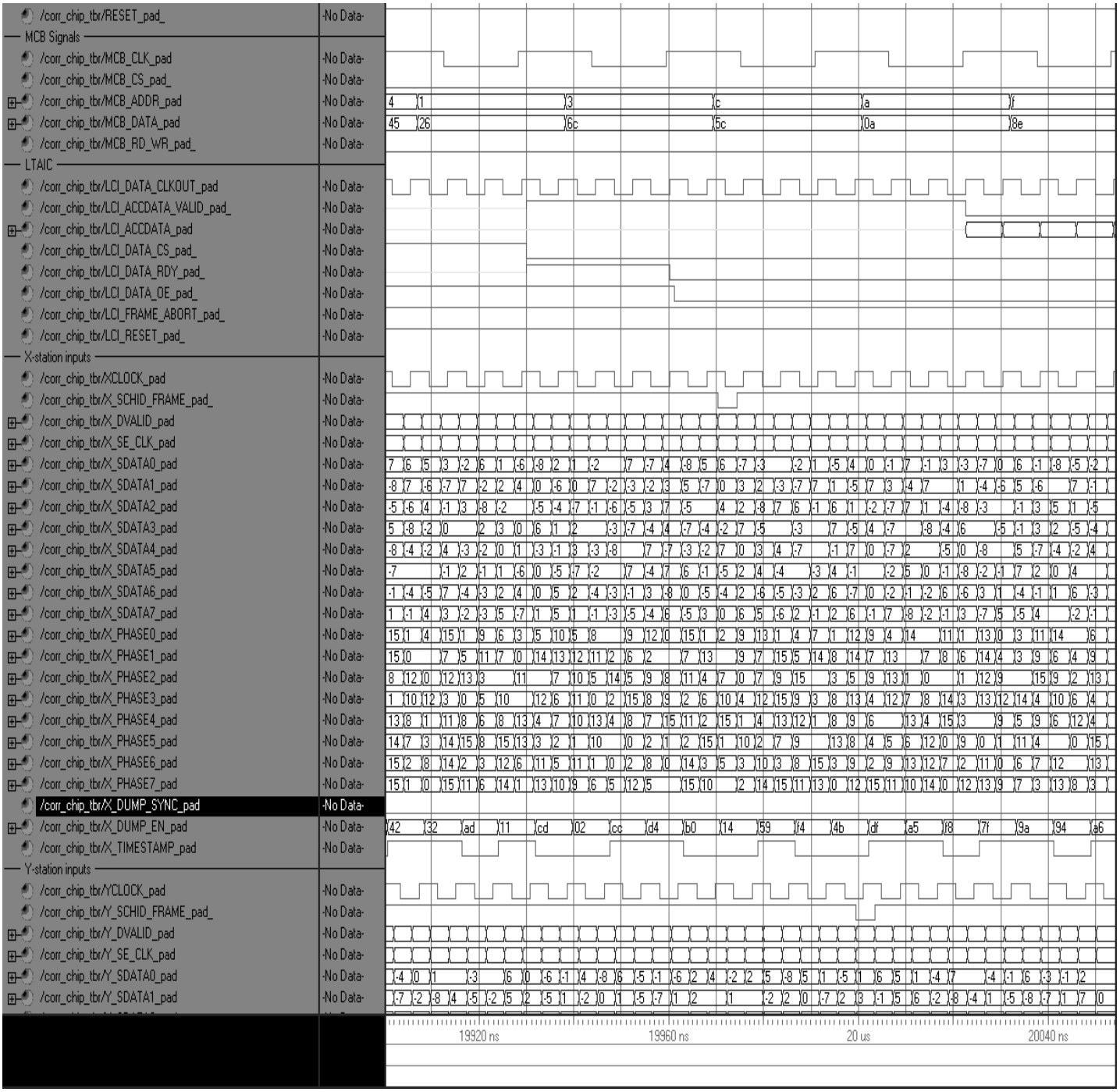


Random input test bench simulation

Wide-angle Modelsim wave file view showing random inputs on the MCB and data lines, and the generation of output frames with random frame abort requests.



This is a zoomed-in picture showing the random input stimuli on the data, phase, and control inputs.



12 Appendix V – C behavioural simulation code

This appendix contains the C program that forms the "gold standard" against which correlator chip RTL and eventually PPAR simulations are compared. This code is very simple and is derived from code used to study and develop an existing correlator [2]. Thus, there is great assurance that it is correct. Important code fragments and functions are highlighted in **bold**.

```
/* Correlator simulator program to provide a reference correlation for testing
 * testvectors generated by noisegenCCtest that go to into the correlator chip
 * testbench.
 */
/* $Log: $ */

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/stat.h>
#include <unistd.h>
#include <simparms_CCtest.h>


#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

#ifndef M_PI
#define M_PI 3.14159265358979323846264
#endif


/* X and Y cross-corr delay lines...only the LSN of each one contains actual
 * data read from the .vhex file. */
typedef struct
{
  int    data;
  int    phase;
  int    dvalid;
} CROSSLAG_T;

typedef struct
{
  double in_phase;
  double quad;
} COMPLEX_T;


static CROSSLAG_T* X_lag_line=NULL;
static CROSSLAG_T* Y_lag_line=NULL;


/* insertion index for the X and Y delay lines */
static int crosslag_insert_index = 0;

static COMPLEX_T* crossaccum=NULL;

static int crosssample_count = 0;
static int crossinsert_count=0;


/* 3-level fringe stopping functions */
static int sin16[] = { 0, 1, 1, 1, 1, 1, 1, 0, 0, -1, -1, -1, -1, -1, -1,0 };
static int cos16[] = { 1, 1, 1, 0, 0, -1, -1, -1, -1, -1, -1, 0, 0, 1, 1,1 };

/* 5-level fringe stopping functions. */
static int sin16_5[] ={ 0, 1, 1, 2, 2, 2, 1, 1, 0, -1, -1, -2, -2, -2, -1,-1 };
static int cos16_5[] ={ 2, 2, 1, 1, 0, -1, -1, -2, -2, -2, -1, -1, 0, 1, 1,2 };

static int num_lags;
static int crosscorr_dellen;

static COMPLEX_T* localaccum=NULL;
```

```
static int testnum;
static int numdumps;
static int sample7bit=0;

/* FUNCTION PROTOTYPES */

static
void
crosscorrelate
( int xdata,
  int xphase,
  int xdvalid,
  int ydata,
  int yphase,
  int ydvalid );

static
void
output_crosscorr_data
(int testnumX,
 int testnumY,
 int dumpnum);



void
main
( int argc,
  char* argv[] )
{
  long samplenum;
  int corrsamples; /* number of samples to correlate per dump */
  int dumpnum;
  int i;
  FILE* fpX;
  FILE* fpY;
  char fnameX[256];
  char fnameY[256];
  int testnumX;
  int testnumY;
  int itemp;
  int xdata, xphase, xdvalid;
  int ydata, yphase, ydvalid;
  int xdata_m;
  int ydata_m;

  if( argc != 7 && argc !=8 )
    {
      fprintf(stderr,
   "Use: corrsimCCtest <#lags> <X-in#> <Y-in#> <#samples;0=all> <test#> <#dumps;0=all> (-7[7-
bit])\n");
      exit(-1);
    }
  itemp = sscanf(argv[1],"%d",&num_lags);
  if( itemp != 1 )
    {
      fprintf(stderr,"***Error: could not determine #lags: '%s'\n",
              argv[1]);
      exit(-1);
    }
  if( num_lags % 2 )
    {
      fprintf(stderr,"***Error: #lags %d must be even!\n",num_lags);
      exit(-1);
    }
  crosscorr_dellen = ((num_lags/2)+1);

  /* dynamically allocate memory for lag correlator structures */
  X_lag_line = (CROSSLAG_T*)malloc(sizeof(CROSSLAG_T)*crosscorr_dellen);
  Y_lag_line = (CROSSLAG_T*)malloc(sizeof(CROSSLAG_T)*crosscorr_dellen);

  crossaccum = (COMPLEX_T*)malloc(sizeof(COMPLEX_T)*num_lags);
  localaccum = (COMPLEX_T*)malloc(sizeof(COMPLEX_T)*num_lags);

  if( X_lag_line==NULL || Y_lag_line==NULL || crossaccum==NULL || localaccum==NULL )
    {
      fprintf(stderr,"***Error: internal malloc() error!\n");
      exit(-1);
    }
```

```
itemp = sscanf(argv[2],"%d",&testnumX);
if( itemp != 1 )
  {
    fprintf(stderr,"***Error: could not determine X-input#: '%s'\n",
            argv[2]);
    exit(-1);
  }
itemp = sscanf(argv[3],"%d",&testnumY);
if( itemp != 1 )
  {
    fprintf(stderr,"***Error: could not determine Y-input#: '%s'\n",
            argv[3]);
    exit(-1);
  }
itemp = sscanf(argv[4],"%d",&corrsamples);
if( itemp != 1 )
  {
    fprintf(stderr,"***Error: could not determine #samples: '%s'\n",
            argv[4]);
    exit(-1);
  }
itemp = sscanf(argv[5],"%d",&testnum);
if( itemp != 1 )
  {
    fprintf(stderr,"***Error: could not determine test#: '%s'\n",
            argv[5]);
    exit(-1);
  }
itemp = sscanf(argv[6],"%d",&numdumps);
if( itemp != 1 )
  {
    fprintf(stderr,"***Error: could not determine #dumps: '%s'\n",
            argv[6]);
    exit(-1);
  }
if( argc==8 )
  {
    /* look for the -7 switch */
    if( !(strcmp(argv[7],"-7")) )
       {
         sample7bit=1;
         fprintf(stderr,"\nNote: 7-bit correlation\n");
       }
    else
       {
         fprintf(stderr,"***Error: unknown switch: '%s'\n",argv[7]);
         exit(-1);
       }
  }

/* open our input test files */
if( sample7bit )
  {
    sprintf(fnameX,"X_CCinput%d_7.vhex",testnumX);
    sprintf(fnameY,"Y_CCinput%d_7.vhex",testnumY);
  }
else
  {
    sprintf(fnameX,"X_CCinput%d.vhex",testnumX);
    sprintf(fnameY,"Y_CCinput%d.vhex",testnumY);
  }
fpX = fopen(fnameX,"r");
fpY = fopen(fnameY,"r");
if( fpX==NULL || fpY==NULL )
  {
    fprintf(stderr,
            "***Error: can't open '%s' or '%s' for reading\n",
            fnameX,fnameY);
    exit(-1);
  }

samplenum = 0;
dumpnum = 0;


/* ******************** Main sample loop *************************** */
while( fscanf(fpX,"%1x%1x%1x\n",&xdata,&xphase,&xdvalid) != EOF  &&
       fscanf(fpY,"%1x%1x%1x\n",&ydata,&yphase,&ydvalid) != EOF )
  {
    /* printf("samplenum: %d xdata: %d ydata: %d\n",samplenum,xdata,ydata); */
    if( sample7bit )
```

```
        {
          if( fscanf(fpX,"%1x%1x%1x\n",&xdata_m,&xphase,&xdvalid) != EOF &&
              fscanf(fpY,"%1x%1x%1x\n",&ydata_m,&yphase,&ydvalid) != EOF )
            {
              /* xdata contains the LSN, xdata_m contains the MSN...merge the two */
              xdata = (xdata&0x0f) | ((xdata_m&0x07)<<4);
              /* correct for sign extension loss */
              if( xdata > 63 )
                xdata = xdata-128;

              /* same for ydata */
              ydata = (ydata&0x0f) | ((ydata_m&0x07)<<4);
              /* correct for sign extension loss */
              if( ydata > 63 )
                ydata = ydata-128;

              /* don't worry about encoding data valid here for the actual test
               * with ModelSim...it will be done in the test bench */
            }
          else
            {
              break; /* all done */
            }
        }
      else
        {
          /* correct for sign-extension loss */
          if( xdata > 7 )
            xdata = xdata-16;
          if( ydata > 7 )
            ydata = ydata-16;
        }

      crosscorrelate(xdata,xphase,xdvalid,ydata,yphase,ydvalid);

      if( corrsamples > 0 )
        {
          /* performing multiple dumps... */
          if( !(samplenum % corrsamples) && samplenum > 10 )
            {
              /* gotta dump the data */
              output_crosscorr_data(testnumX,testnumY,dumpnum);
              crosssample_count=0;
              dumpnum++;
              if( (numdumps > 0) && (dumpnum >= numdumps) )
                exit(-1);
            }
        }
      samplenum++;
    } /* end while loop */
  output_crosscorr_data(testnumX,testnumY,dumpnum);
}


/* function to do a simple 'bi-directional' cross-correlation.  Note that the
 * center lag is at num_lags/2 */

static
void
crosscorrelate
( int xdata,
  int xphase,
  int xdvalid,
  int ydata,
  int yphase,
  int ydvalid )
{
  int lag;
  int i;
  int indexX;
  int indexY;
  double sin_part,cos_part;
  int phase;

  if( crossinsert_count == 0 )
    {
      for( i=0; i<crosscorr_dellen; i++ )
        {
          X_lag_line[i].data = 0;
          Y_lag_line[i].data = 0;
          X_lag_line[i].phase = 0;
```

```
          Y_lag_line[i].phase = 0;
           X_lag_line[i].dvalid = 0;
           Y_lag_line[i].dvalid = 0;
        }
      for( i=0; i<num_lags; i++ )
        {
          crossaccum[i].in_phase = crossaccum[i].quad = 0.0;
        }
    }

  /* insert data, phase, data valids into the delay line */
  X_lag_line[crosslag_insert_index].data = xdata;
  X_lag_line[crosslag_insert_index].phase = xphase;
  X_lag_line[crosslag_insert_index].dvalid = xdvalid;

  Y_lag_line[crosslag_insert_index].data = ydata;
  Y_lag_line[crosslag_insert_index].phase = yphase;
  Y_lag_line[crosslag_insert_index].dvalid = ydvalid;


  /* do not increment the insertion index yet! so that it points at the newest
   * sample. */

  crossinsert_count++;

  for( lag=0; lag<num_lags; lag++ )
    {

      if( !(lag % 2) )
        {
          /* even lags */
          indexX = (crosslag_insert_index + 1 + lag/2) % crosscorr_dellen;
          indexY = (crosslag_insert_index - lag/2);
          if( indexY < 0 )
            indexY += crosscorr_dellen;
        }
      else
        {
          /* odd lags */
          indexX = (crosslag_insert_index + 1 + (lag+1)/2) % crosscorr_dellen;
          indexY = (crosslag_insert_index - ( lag-1)/2);
          if( indexY < 0 )
            indexY += crosscorr_dellen;
        }

      phase = (X_lag_line[indexX].phase-Y_lag_line[indexY].phase);
      if( phase < 0 )
        phase += 16;
      else
        phase = phase % 16;

#if LEVEL5_FRINGE_STOPPING
      sin_part = (double)sin16_5[phase];
      cos_part = (double)cos16_5[phase];
#else
      sin_part = (double)sin16[phase];
      cos_part = (double)cos16[phase];
#endif


      crossaccum[lag].in_phase +=
        (double)(X_lag_line[indexX].data * X_lag_line[indexX].dvalid *
                 Y_lag_line[indexY].data * Y_lag_line[indexY].dvalid) * cos_part;

      crossaccum[lag].quad +=
        (double)(X_lag_line[indexX].data * X_lag_line[indexX].dvalid *
                 Y_lag_line[indexY].data * Y_lag_line[indexY].dvalid) * sin_part;


      /* maintain a data valid count at the center lag */
      if( lag == num_lags/2 )
         {
           if( X_lag_line[indexX].dvalid & Y_lag_line[indexY].dvalid )
             crosssample_count++;
         }
    }

  /* finally, increment the insertion index */
  crosslag_insert_index = (crosslag_insert_index+1) % crosscorr_dellen;
}
```

```
/* this function spits out cross-correlated data to a file */
static
void
output_crosscorr_data
(int testnumX,
 int testnumY,
 int dumpnum )
{
  FILE* fp;
  int lag;
  double max;
  char buf[256];
  char buf1[256];
  int i;

  if( crosssample_count <= 0 )
     return; /* nothing to do */

  /* normalize and copy data locally */
  for( lag=0; lag<num_lags; lag++ )
    {
      localaccum[lag].in_phase = crossaccum[lag].in_phase /
        (double)crosssample_count;
      localaccum[lag].quad = crossaccum[lag].quad /
        (double)crosssample_count;

      /* clear the active accumulators as well...since we could be doing
      * multiple dumps */
      crossaccum[lag].in_phase = 0.0;
      crossaccum[lag].quad = 0.0;
    }

  printf("Data valid count=%d\n",crosssample_count);

  /* The only normalization that is done is to divide by the data valid
   * count.  The Van Vleck normalization is not done, and is pointless
   * anyway for this test.
   */
  sprintf(buf,"CC%dx%d_test%02d_%dlagsD%d.dat",
          testnumX,testnumY,testnum,num_lags,dumpnum);

  fp = fopen(buf,"w");
  if( fp == NULL )
    {
      fprintf(stderr,"***Error: can't open file '%s'\n",
              buf);
      exit(-1);
    }

  for( lag=0; lag<num_lags; lag++ )
    {
      fprintf(fp,"%.15e %.15e\n",
              localaccum[lag].in_phase,
              localaccum[lag].quad);
    }
  fclose(fp);
}
```