

AFRL-IF-RS-TR-2002-276
Final Technical Report
October 2002



DYNAMIC RECONFIGURATION FOR ADAPTIVE COMPUTING SYSTEMS (DRACS)

BAE Systems - Information and Electronics Warfare Systems

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J471

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-276 has been reviewed and is approved for publication

APPROVED:

A handwritten signature in black ink that reads "Martin J. Walter". The signature is written in a cursive style with a long horizontal stroke extending to the right.

MARTIN WALTER
Project Engineer

FOR THE DIRECTOR:

A handwritten signature in black ink that reads "Michael L. Talbert". The signature is written in a cursive style with a large, sweeping initial "M".

MICHAEL L. TALBERT, Major, USAF
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank) 2. REPORT DATE
Aug 02 3. REPORT TYPE AND DATES COVERED
Final Jun 99 – Mar 02

4. TITLE AND SUBTITLE
DYNAMIC RECONFIGURATION FOR ADAPTIVE COMPUTING SYSTEMS (DRACS)

5. FUNDING NUMBERS
C - F30602-99-C-0164
PE - 62301E
PR - DRAC
TA - S0
WU - 01

6. AUTHOR(S)
John C. Zaino

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BAE Systems Information and Electronic Systems, Inc.
PO Box 868, MER 15-222
Nashua, NH 03061-0868

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFTC
3701 North Fairfax Drive 26 Electronic Pky
Arlington, VA 22203-1714 Rome, NY 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-276

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Martin Walter, IFTC, 315-330-4102, walterm@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 Words*)
The Dynamic Reconfiguration for Adaptive Computing Systems (DRACS) effort has exploited the emerging technology associated with run-time reconfigurable devices to develop system capabilities for run-time reconfiguration (RTR) of ACS hardware both in response to software control and in a data-driven manner. Using the DARPA-sponsored CSRC dynamically reconfigurable device, DRACS has demonstrated a host-driven design and reconfiguration as well as two data-driven designs, one Finite State Machine (FSM) driven and one additional host-driven demonstration. This report describes how these can be used to develop a "virtual co-processor" that supports multiple reconfigurable computing applications residing in a single piece of hardware. One demonstration focus selected involves a subset of a realistic system scenario for parameter measurement processing in electronic warfare that can be improved through the use of dynamic reconfiguration.

14. SUBJECT TERMS
Run-Time Reconfiguration, FPGA, Adaptive Computing

15. NUMBER OF PAGES
86

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

Table of Contents

1.0	Introduction.....	1
2.0	Supporting Technology Background.....	2
2.1	CSRC Architecture Description.....	3
2.1.1	Data Pipes.....	4
2.1.2	Context Switching Logic Array.....	5
2.1.3	Routing Modes of Operation.....	6
2.1.3.1	Bus Routing.....	6
2.1.3.2	Bitwise Routing.....	7
2.1.4	CSLC.....	8
2.1.5	CSIO.....	9
2.1.6	Data Sharing/Context Switching.....	10
2.1.7	Block RAM.....	11
2.1.8	High Speed Direct Connect Routing.....	12
2.1.9	Programming.....	12
2.1.10	CSRC Control Block.....	13
2.2	Reconfigurable Computing Module (RCM) Description.....	13
3.0	Technical Development.....	16
3.1	BAE SYSTEMS.....	16
3.1.1	Design Methodology.....	16
3.1.2	CSRC Tools	23
3.1.3	CSRC RCM Board Testing Environment.....	26
3.1.4	Host Driven Demonstration.....	28
3.1.5	Data Driven Demonstration.....	39

3.2 Virginia Tech.....	42
3.2.1 General.....	42
3.2.2 API.....	42
3.2.3 RCM OS.....	44
3.2.3.1 Cache Management.....	44
3.2.3.2 Data-Driven RTR.....	44
3.2.3.2.1 Implementing the Finite State Machine (FSM).....	46
3.2.4 Janus/JHDL Approach to CSRC Data/Control Driven RTR... 	49
3.2.5 Technical Discussion.....	52
3.3 Brigham Young University.....	59
3.3.1 General.....	59
3.3.2 JHDL Design Tool.....	60
3.3.3 RCM Board Model.....	66
3.3.4 JHDL (Software) Mode.....	67
3.3.5 Hardware Mode.....	68
3.3.6 Creating and Building a Design.....	69
3.3.7 Technical Discussion.....	71
4.0 Summary.....	72
Appendix A.....	73

List of Figures

2.1	Reconfigurable Benefits.....	3
2.1.1	16 Bit Data Pipe Comprised of CSLAs.....	4
2.1.2	Level 3 Routing Bridges Pipes.....	5
2.1.3	Prototype Context Switching Logic Array & Level 1 Routing.....	6
2.1.4	Prototype Context Switching Logic Array with Level 1 & Level 2 Routing.....	7
2.1.5	Context Switching Logic Cell Architecture.....	8
2.1.6	Context Switching Input/Output Cell Architecture.....	10
2.1.7	Private/Public Addressable Sharing Scheme.....	11
2.1.8	Direct/Shift Routing.....	12
2.2.1	RCM Block Diagram.....	14
2.2.2	RCM Circuit Card.....	16
3.1.1	Design Flow for the CSRC/RCM Board.....	17
3.1.2	Synplicity Window.....	19
3.1.3	Executing csrc.exe.....	21
3.1.4	CSRC Detailed Design View.....	22
3.1.5	CSRC Detailed Design View (Zoom).....	23
3.1.6	16-Bit Adder With Carry Chain.....	24
3.1.7	Overview of 16-Bit Adder With Carry Chain.....	25
3.1.8	16-Bit Adder Without Carry Chain.....	25
3.1.9	BAE SYSTEMS DRACS Test Bed.....	27
3.1.10	Typical EW Application of Signal Detection and Classification.....	28
3.1.11	Typical Parameter Measurements.....	29
3.1.12	Division of Channel into Sub-bands.....	30
3.1.13	CSRC Filters in Pulse Parameter Processing Path.....	31
3.1.14	Filters in CSRC Contexts with Shared All-pass Filter.....	31
3.1.15	Filters in CSRC Contexts with Two Filters per Context.....	32
3.1.16	Mixed Signal Test Input, Pulse-on-Pulse Condition.....	34
3.1.17	Mixed Test Signal Components, Signals 1 and 2.....	35
3.1.18	Design of Complex FIR Filters A&B.....	36
3.1.19	Unmodified FIR Filter.....	36
3.1.20	Modified FIR Filter.....	37
3.1.21	POP Signal Input Sequence.....	38
3.1.22	POP Signal Separation Results – Host Driven.....	39
3.1.23	POP Signal Input Sequence.....	40
3.1.24	POP Signal Separation Results – Data Driven.....	41

3.2.1	Mode 1 – Device Computes and Performs Context Switch.....	45
3.2.2	Mode 2 – State Analyzed and Processed by External FPGA.....	46
3.2.3	Mode 3 – State Analyzed and Processed by External CPU.....	46
3.2.4	State-Driven RTR.....	47
3.2.5	Traditional Approach to FSM Creation.....	48
3.2.6	Dynamic FSM Generation.....	48
3.2.7	Janus Environment Over JHDL.....	49
3.2.8	Janus Hardware Abstraction.....	50
3.2.9	Classic Janus Task Scheduling.....	51
3.2.10	Classic Janus Execution.....	51
3.2.11	Janus Task Scheduling.....	52
3.2.12	Enigma Processing.....	53
3.2.13	Enigma Implementation.....	54
3.2.14	Enigma GUI.....	54
3.2.15	Motion Detection Algorithm.....	57
3.2.16	Motion Detection Implementation.....	58
3.3.1	The Tool Path: JHDL Description to Circuit Simulation.....	62
3.3.2	JHDL Library of CSRC Primitives.....	63
3.3.3	The Tool Path: JHDL Board Level SW Simulation to VHDL Netlister...	64
3.3.4	The Tool Path: Multicontext Place and Route (CSRC Tools) to Bitstream File Generation.....	65
3.3.5	The Tool Path: JHDL Board Level Hardware Verification.....	66
3.3.6	Schematic Viewer – Shared Data Register in Design.....	67
3.3.7	Schematic Viewer – Non-Shared Data Registers in Design.....	68
3.3.8	Schematic Viewer – Incrementer Design.....	71

List of Tables

3.2.1	Area Study Results.....	59
-------	-------------------------	----

1.0 Introduction

The ability to rapidly reconfigure hardware is the essential quality that makes adaptive computing systems an attractive processing paradigm. Many of the most revolutionary performance results in the DARPA Adaptive Computing Systems (ACS) program have come from clever reconfiguration of the devices in problem-specific ways. While commercial FPGAs had never been constructed for real-time configuration, a new breed of run-time reconfigurable devices is becoming available from both DARPA ACS research and industrial efforts. The Dynamic Reconfiguration for Adaptive Computing Systems (DRACS) program has exploited this emerging technology to develop system capabilities for run-time reconfiguration (RTR) of ACS hardware both in response to software control and in a data-driven manner. DRACS provides an order of magnitude improvement in system capability by nearly instantaneously reconfiguring system functions to operational needs.

The DRACS program has made great strides towards filling the gap between the emerging RTR device technology and system insertion. We have created software for the development and management of run-time reconfiguration. We have leveraged emerging software technologies for providing high-level design entry and debugging. These technologies have been targeted to, but not fully limited to, the premier RTR device, the DARPA-sponsored Context Switching Reconfigurable Computing (CSRC) hardware. The RTR management software has been developed through extensions to the DARPA-sponsored System Level Architectures for Adaptive Computing (SLAAC) adaptive computing application programming interface as well as a custom API developed by Virginia Tech under subcontract to this program. Demonstration of the power and flexibility of the RTR system environment has been showcased through the development of a DoD electronic warfare RTR application and several other RTR applications of DoD (and commercial) interest.

The development approach for DRACS focused on the key technology of managing run-time reconfiguration in a systems context and on support of high level design and debug of run-time reconfiguration systems. The DRACS program has expanded on the existing SLAAC application programming interface to include run-time reconfiguration. These extensions include host-driven and data-driven reconfiguration. The host-driven reconfiguration extensions allow both user-level and system-level control of configurations, under software control. Current API extensions provide support for defining a virtual adaptive computing environment, consisting of more logical hardware configurations than physical hardware platforms, and for transparently managing these virtual configurations. Support is provided for caching of configurations and background loading of configurations into hardware. These extensions support development of a “virtual coprocessor,” e.g. one piece of hardware that will, during the course of system use, represent multiple physical designs, all transparently to the users of the system.

To support high level design and debug of run-time reconfiguration systems we provided coordination and guidance to Brigham Young University in its development of run-time reconfiguration extensions into its DARPA-sponsored JHDL design tool. These extensions allow modeling both of reconfigurable devices as well as a reconfigurable processing board containing multiple dynamically reconfigurable devices. The JHDL-based approach to

development of RTR systems provided capabilities toward enabling both rapid development and rapid debugging of dynamically reconfigurable systems.

The DRACS program has successfully demonstrated the power of dynamically reconfigurable hardware and the capabilities of the technologies being developed here in a sequence of demonstrations. Using the DARPA-sponsored CSRC dynamically reconfigurable device, DRACS has demonstrated a host-driven design and reconfiguration as well as two data-driven, one Finite State Machine (FSM) driven and one additional host-driven demonstration. DRACS has showed how these can be used to develop a “virtual co-processor” that supports multiple reconfigurable computing applications residing in a single piece of hardware. These demonstrations showcase both run-time reconfiguration and high level design tools on applications of interest to DoD. One demonstration focus selected involves a subset of a realistic system scenario for parameter measurement processing in electronic warfare that can be significantly improved through the use of dynamic reconfiguration.

The DRACS’ program development of a unified RTR environment and use of high level targeting tools, using DARPA’s CSRC device technology and the SLAAC and Virginia Tech APIs, have created a crucial path to the widespread use and acceptance of RTR technology.

2.0 Supporting Technology Background

The context switching reconfigurable computing (CSRC) technology that was previously developed by BAE SYSTEMS (then Sanders, A Lockheed Martin Co.) under the CSRC program extended commercially available field programmable gate array (FPGA) devices to include high speed changes between a number of programmed functions without the need for additional FPGAs. Each configuration, referred to as a *context*, in a CSRC FPGA has the functionality similar to that of many commercially available FPGAs. The context switching can occur at significantly higher speeds than the rate at which current FPGA technology can reconfigure. In addition, unlike commercial FPGAs, where reprogramming destroys any resident data, the CSRC FPGA affords the capability of data sharing between contexts.

The concept of virtual hardware is an obvious benefit of dynamic reconfiguration. If configurations can be swapped in and out of an FPGA upon demand at a real-time system rate, only the necessary hardware need be instantiated at any given time. In this manner, a virtually infinite algorithm cache or an infinite coprocessor can be conceived. In other words, a high level system scheduler can instantiate hardware as needed. In this manner, a reduction in size, weight, and power can be achieved. Additionally, given the CSRC FPGA, if the processing requirements specify a sequential application of algorithms, the context layers can be set up to share data such that the output of one algorithm is immediately available as the input to the next algorithm upon a context switch. This was not possible with contemporary FPGAs.

A natural extension of the algorithm cache mode of computation is the concept of mission phase reprogrammability. As seen in Figure 2.1, an entire mission can be mapped to a CSRC device. In this case, different contexts can house different algorithmic phases of a mission without requiring that an algorithm be confined to a single context, depicted as layers in Figure 2.1.

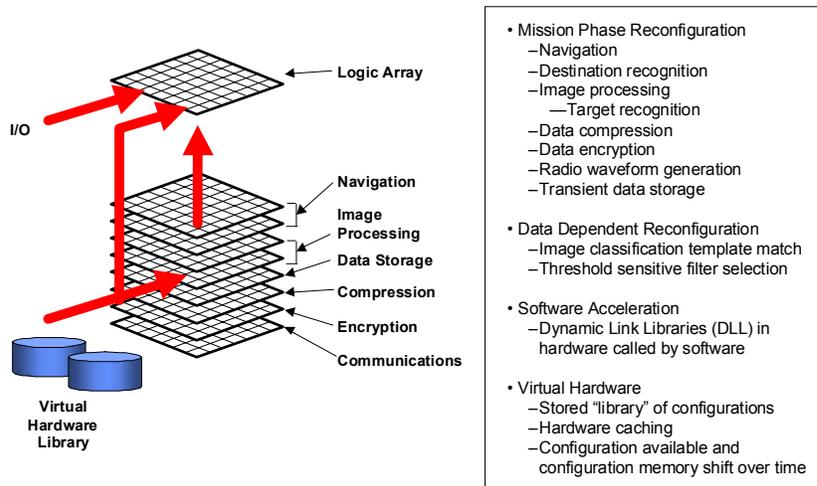


Figure 2.1: Reconfiguration Benefits

Although Figure 2.1 identifies the obvious modes of computation for gaining a performance enhancement, it is believed that the true potential of context switching requires a paradigm shift in algorithm implementation. The capabilities of the CSRC architecture, which extend dynamic reconfiguration to context switching, have the potential to provide improved implementations of signal processing algorithms over those currently available through commercial FPGAs. The inherent ability of CSRC to quickly perform different tasks and share results among different configurations allows one to approach algorithms from a different perspective, enabling mathematical implementations previously inconceivable without context switching.

A reconfigurable computing module (RCM) was designed, fabricated and tested successfully. The RCM fits into a standard computer PCI slot and contains two CSRC devices. The RCM has been integrated into a PC environment so that host programs can use the RCM to demonstrate CSRC technology. A sophisticated suite of development tools have been built so that designers may describe circuits and map them seamlessly onto the multi-layered contexts of the CSRC device. The DRACS project employs the CSRC device, the RCM, and the CSRC Toolkit to further enhance dynamic reconfigurable technology by affording the designer a design environment that facilitates the ease of developing runtime reconfigurable systems.

2.1 CSRC Architecture Description

Experience has shown that FPGAs afford the greatest performance benefit when they are used to implement algorithms with deep pipelines. However, pure dataflow algorithms are rare. In fact, generating pipeline control signals, implementing state machines, and interfacing with external RAM or other integrated circuits, are critical, although not typically areas of performance enhancement, to an FPGA's successful system integration. With this in mind, the CSRC device was designed to be a 4 bit DSP dataflow engine that is simultaneously capable of efficiently implementing glue logic. However, since FPGA performance enhancements are oftentimes achieved by implementing the minimum required bitwidth, the CSRC device was developed to allow users to implement scalable pipelines such that the wordwidth can be of any size.

2.1.1 Data Pipes

The CSRC device is arranged into 16-bit wide data pipes. Each pipe is formed by a plurality of context switching logic arrays (CSLAs) as seen in Figure 2.1.1. A single CSLA is capable of processing two 16-bit words and outputting a 16-bit result. The result of a CSLA is available as an input to the two adjacent CSLAs in the pipe. Hence, a pipe can naturally be used as a data path. Information can easily flow from one end of the pipe to the other. It is important to point out that in this device data can non-preferentially flow in both directions. This feature has great utility when sharing data among different contexts. For example, one context could process data from left to right, storing its' final result in the right-most set of registers. Note that is quite possible that the final result of a single context is actually an intermediate result of the entire algorithm. Given this situation, an incoming context can pick up where its predecessor context left off by acquiring the intermediate data deposited on the rightmost portion of the pipeline and processing it in a pipeline that flows from right to left. From this simple example, it can be seen that a data path that does not favor data flow in either direction is more efficient for context switching hardware because it alleviates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

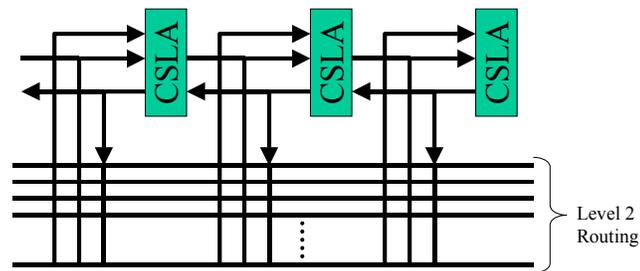


Figure 2.1.1: 16 Bit Data Pipe Comprised of CSLAs

Level 2 routing can be found alongside the pipe and consists of 16-bit buses. See Figure 2.1.1. These busses are not segmented and run the entire width of the CSRC device. This type of bussing scheme implies that a signal driven onto level 2 routing is available to any CSLA in the pipe. Additionally, this approach affords the possibility of faster and less complicated programming tools than segmented approaches because the timing is more deterministic. Each CSLA has two 16-bit inputs, each of which is capable of tapping into any of the Level 2 routing busses. Similarly, the CSLA's 16-bit output can drive any of the Level 2 routing busses. Note that Level 2 routing can be utilized as a bus architecture, can be broken down and utilized by individual bits, or can be employed as any combination of these.

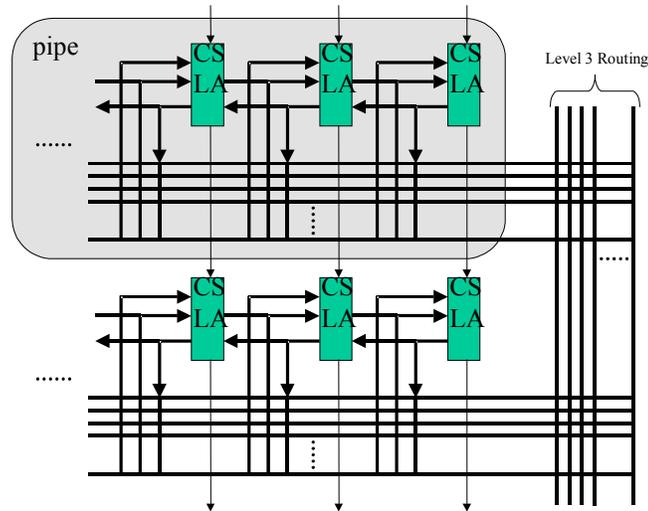


Figure 2.1.2: Level 3 Routing Bridges Pipes

The CSRC device is formed by stacking up pipes one on top of the other. Corresponding CSLAs on adjacent pipes have dedicated wiring that allows them to pass along their carry bit. This feature allows two adjacent pipes to be bundled together and be used as a single 32-bit wide data path. In actuality, physical 16-bit pipes can be broken down into smaller logical pipes. Although hardware is optimized to break pipes into nibbles, pipes can be n-bits wide.

As seen in Figure 2.1.2, information driven onto a given pipe’s Level 2 routing can be connected to Level 3 routing which in turn makes the data available to any Level 2 routing on the chip. Similar to the Level 2 routing structure, the Level 3 routing is not segmented and spans the device. Note that conceptually the Level 2 and Level 3 routing are perpendicular to each other.

I/O pins on the device are connected to Level 2 and Level 3 routing. All pins physically located on the top and bottom edges of the device connect to Level 3 routing. Pins on the left and right edges can connect to either Level 2 routing or directly into the dedicated routing that normally connects adjacent CSLAs.

2.1.2 Context Switching Logic Array

A single CSLA is primarily composed of 16 context switching logic cells (CSLCs) and Level 1 routing to interconnect them. Figures 2.1.3 and 2.1.4 depict a CSLA and the CSLA as it attaches to the Level 2 routing, respectively. Note that the routing structure depicted applies to the prototype IC. The final IC routing architecture is slightly more flexible but utilizes similar structure to that employed in the prototype IC. The CSLCs are numbered 0 through 15 and their carry-in and carry-out chains are hardwired appropriately so they can function as a single cohesive unit. Level 1 routing consists of three 16-bit busses. Two of these 16-bit busses are inputs from the Level 2 routing. The third 16-bit bus is hardwired to the outputs of the CSLCs. Level 1 routing was designed with two modes of operation in mind.

2.1.3 Routing Modes of Operation

As previously mentioned, it is believed that the most beneficial FPGA is capable of exploiting its inherent DSP strengths while simultaneously being capable of implementing the often required glue logic. Hence, the CSRC FPGA has been designed with two modes of operation in mind: (1) Deep pipeline mathematical operations that can be of arbitrary bitwidth & (2) Random logic implementations that encompass control, state machines, and interfacing with external RAM or other integrated circuits. As a direct result, the CSRC FPGA exhibits two types, or modes, of routing.

2.1.3.1 Bus Routing

The first operational mode of routing is bus routing. The design goal was to provide users with the ability to route entire 16-bit words in and out of CSLAs while maintaining bitwidth order (i.e. the most significant bit (MSB) in the MSB position and the least significant bit (LSB) in the LSB position.)

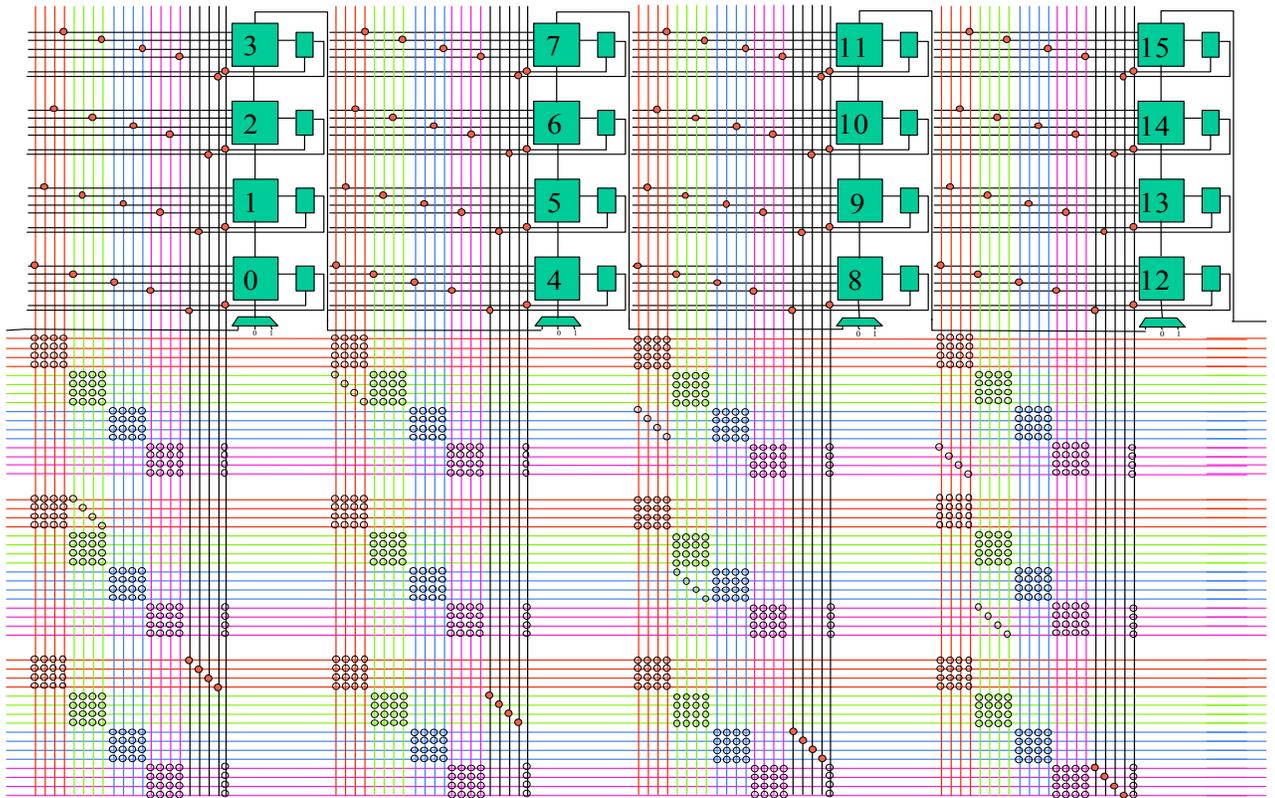


Figure 2.1.3: Prototype Context Switching Logic Array & Level 1 Routing

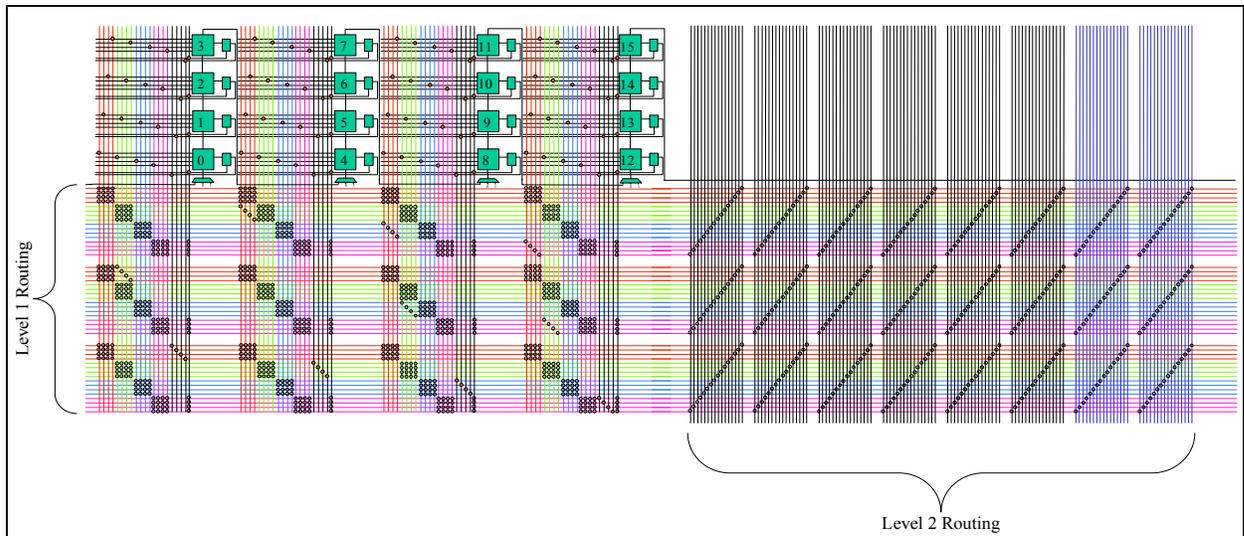


Figure 2.1.4: Prototype Context Switching Logic Array with Level 1 & Level 2 Routing

Given that the four data inputs to the CSLC are labeled as A, B, C, and D, enough programmable connections are contained in the Level 1 switching matrices to ensure that one of the input buses can be routed into the A inputs of all of the 16 CSLCs. The least significant bit of the bus feeds the A-input of the least significant CSLC and so on. In essence, this bus can be considered the A-input (16-bits wide) for the entire CSLA under this bus routing mode of operation. Note that the second 16-bit bus can be used to feed the B inputs of the CSLCs within a CSLA in a similar fashion. The final bus connection is hardwired to the 16-bit output of the CSLA and attaches to the Level 2 routing. Note that this output is also a direct connect between neighboring CSLAs. As previously described, this non-directional direct connect allows for fast routing between CSLAs within a pipe by alleviating the need for Level 2 routing if the output of a pipe stage is feeding a neighboring CSLA.

2.1.3.2 Bitwise Routing

The second mode of operation is bitwise routing. No matter how data processing intensive a design might be there is almost always a need for control logic whether it is simple glue logic or more complex state machines. For this reason the bitwise routing mode of operation is necessary. The basic premise is that the output of any given CSLC within a CSLA should have at least one possible path to connect to at least one input of all other CSLCs within the same CSLA. A simple pattern of programmable connections was developed to enable this feature. All the A-inputs of all the CSLCs in a CSLA can tap into the four least significant bits of all three Level 1 routing busses (this includes the output bus to provide a means of local feedback without having to waste Level 2 routing resources). Similarly the B-inputs and the C-inputs tap into the next 4 bit bundles within each level 1 routing bus, and finally the D-inputs tap into the four most significant bits on every bus. As a result, the four least significant CSLCs, which drive the corresponding four-least significant bits of the output bus, are capable of driving any A-input on any CSLC within the same CSLA. For this reason these four CSLCs are known as “A-drivers”

under the bitwise routing mode of operation. Similarly, B-drivers refers to CSLCs 4 through 7, C-drivers to CSLCs 8 through 11, and D-drivers to CSLCs 12 through 15. Furthermore, since connections between Level 1 and Level 2 and connections between Level 2 and Level 3 maintain proper bit order (LSBs to LSBs and MSBs to MSBs) any A-driver can drive the A-input of any CSLC anywhere in the chip. For these same reasons, the same functionality applies to the B, C, and D-drivers.

In addition to the four main inputs (A, B, C, & D), each CSLC has a clock enable / tri-state control line. Both of these control lines tap into the four most significant bits of the three Level 1 buses, hence, they are controlled by D-drivers. As seen in Figure 2.1.5, the clock enable / tri-state control line is a single control line to the CSLC. For this reason, the user can choose to use this control line to control either the clock enable or the tri-state buffer. Note that in the final CSRC IC, the tri-state functionality has been removed leaving only the enable control signal.

2.1.4 CSLC

The CSLC is the heart of computation for the CSRC device. As seen in Figure 2.1.5, the CSLC is composed of carry logic, a four input lookup table (CSLUT), a context switching flip-flop (CSFF) and a tri-state buffer. The carry logic unit is capable of generating carry bits for either additions or subtractions. The carry logic chain is connected by dedicated connections. The chain can be connected, disconnected, or fed a logic zero or logic one every four bits. In this manner, the bus routing mode can be utilized to generate a pipeline granularity of four bits. However, in reality, the buswidths can be of an arbitrary bitwidth, n . Note that bitwidths with a modulo $4 = m$, where m is greater than zero, will disallow m CSLCs from supporting a mathematical pipe that requires the starting of a carry chain.

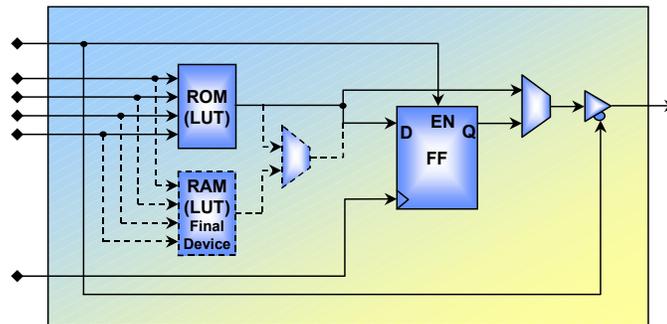


Figure 2.1.5: Context Switching Logic Cell Architecture

The outputs of the carry logic feed the CSLUT which consists of 16 context switching configuration bits (CSBits) that are multiplexed together. The 4 inputs serve as the select lines therefore implementing a programmable function. Note that the contents of each of the CSLUTs are unique in each context and specified in the configuration bitstream.

The CSBits implement context switching itself. Each CSBit holds a single programming bit for every context. However, only the active context's value drives whatever logic the CSBit is controlling.

Unlike some commercial FPGAs, the lookup table can not serve as a memory element because the CSLUT is composed of CSBits, not SRAM. Instead a separate context switching RAM (CSRam) provides memory storage facilities. The CSRam, which is only available in the final CSRC device, implements the *global sharing scheme*. This data sharing scheme is similar to traditional blackboard data sharing. Any data written to a CSRam memory is available to all the CSRam elements that are physically collocated among different contexts. Whatever data value is last written into the active CSRam before deactivation of the current context will be seen by all other collocated CSRams upon the activation of their respective contexts. In fact, one can envision writing to a CSRam in one context and having its contents be used as a LUT in another context. Additionally, it is the CSRam that will allow for large amounts of data passing between contexts to facilitate modes of computation such as moving the algorithm through the data. This mode of computation is advantageous due to the fact that the on/off chip accesses are minimized by loading the data on chip and keeping it on chip until the entire algorithm has been run on the data.

Both the CSRam and the CSLUT coexist in the final CSRC device and their outputs are multiplexed together. The select line of this MUX is yet another control line to the CSLC and it is connected to Level 1 routing in the same fashion as the clock enable / tri-state control (driven by D-drivers). The output of this MUX can then be registered or passed directly out of the CSLC as seen in Figure 2.1.5. Note that if the data is to be registered, it will be done in the context switching flip-flop (CSFF). During regular operation within a single context, the CSFF appears to the users as a normal D-flip-flop (DFF). The DFF connects to the global clock and it is controlled by the clock enable input to the CSLC.

2.1.5 CSIO

The context switching input/output cell is used to facilitate on/off chip data accesses. As can be seen in Figure 2.1.6, the CSIO cell is bi-directional, can provide latched or direct outputs, and has a programmable pull-up resistor on the output. In addition, the CSIO cell can tri-state its output. Since on/off chip access time is oftentimes a limiting factor of FPGAs, a programmable drive strength capability has been included to insure maximum performance. Finally, the flip-flop in the CSIO cell utilizes a different sharing scheme than the flip-flop in the CSLC. Note that since it is believed that sharing data between contexts within a CSIO cell is unlikely to be a key feature, the global sharing scheme is implemented for the CSIO cell DFFs rather than the more complex sharing scheme that is implemented in the CSLC's DFF.

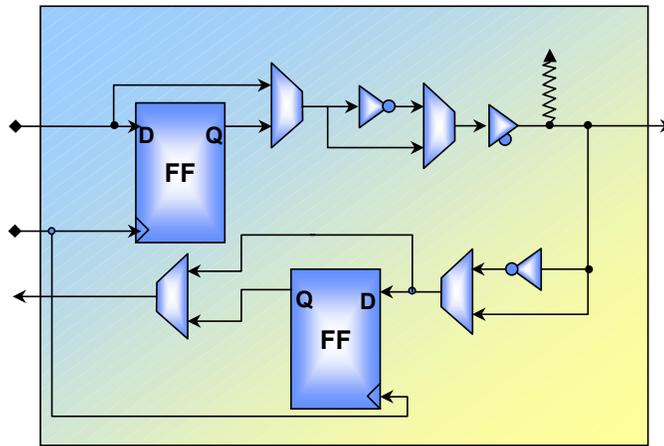


Figure 2.1.6: Context Switching Input/Output Cell Architecture

2.1.6 Data Sharing / Context Switching

Research indicates that the major benefits of context switching are afforded by sharing data between contexts and being able to switch between contexts very rapidly. For this reason, a great emphasis has been placed on the development of a device that meets both of these needs. The two sharing schemes that have been designed and implemented are Global Sharing and Private/Public Addressable Sharing (P/PASS). The global sharing scheme is used in the CSIO DFF and in the CSRam while P/PASS is used in the CSLCs by the CSFF. Global Sharing, as previously described, is simply a common memory element between all contexts. Hence, all contexts view these same memory elements and when any context writes to the memory element, the change is seen by all contexts upon their respective activation.

The data sharing scheme used by the CSFF, P/PASS, truly exposes the novelty of the CSFF and is depicted in Figure 2.1.7. With this type of sharing, each CSFF within each context supported in hardware has a corresponding register. These registers are known as *private* registers since they belong to a particular context and can only be accessed by a specific DFF within the context. Additionally, there is a single active register per CSFF. The active register is what the user actually utilizes during uninterrupted context execution. Upon switching contexts, the outgoing (active) context saves its intermediate values to its private registers. This feature enables many of the capabilities that the NSA would need to develop secure kernels by isolating intermediate data. Additionally, a context can *choose* to write its values to a *public* register (on a Logic Cell by Logic Cell basis) which can be addressed by any and all of the contexts. In this manner, the sharing of data between DFFs within contexts is enabled. The number of public registers available in a P/PASS implementation is independent of the number of contexts supported directly by hardware. Hence, public registers must be addressed when used. Note that the CSRC device affords two public registers. Upon activation, a context can choose to restore its previous state by reading from the private register or it can opt to load a state from either public register (on a Logic Cell by Logic Cell basis).

P/PASS provides a means to keep secure data isolated while at the same time allowing data to be shared (if so desired) using public registers. This architecture scales to implementations with more contexts than hardware supports, allows sharing data between contexts that do not necessarily follow one another in time, and provides a clean and solid foundation to add features such as interrupt handling and hardware recursion.

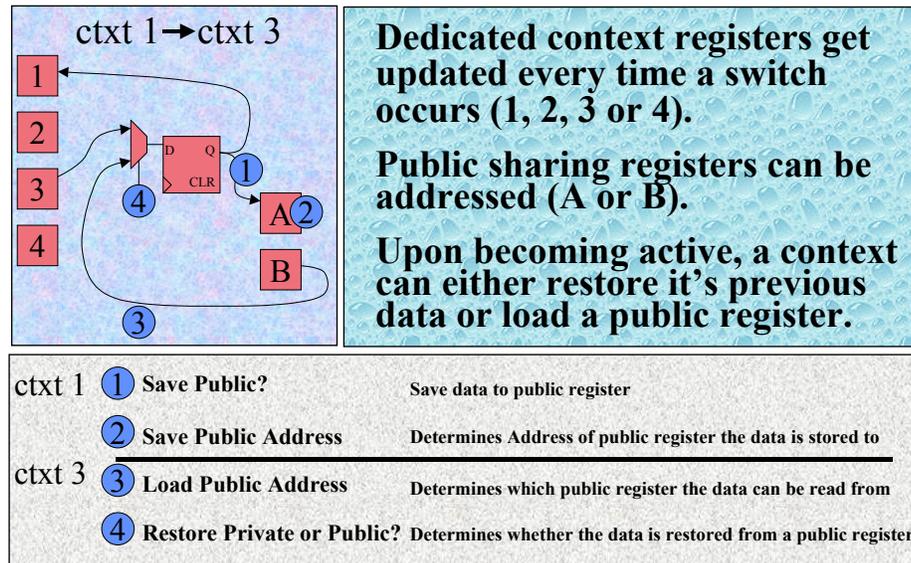


Figure 2.1.7: Private/Public Addressable Sharing Scheme

Since some modes of computation, such as the virtual coprocessor, require rapid reconfiguration, the CSRC device was designed to be capable of switching contexts on a single clock cycle. A key point to be made is that this single cycle context switching not only includes completely reconfiguring the CSRC device but completely executing all of the data sharing schemes. In fact, the active context can be swapped so rapidly that a context can be processing data on one clock edge, switch to a new configuration (including data sharing) and be processing data in the new configuration on the very next clock edge. SPICE simulations indicate that it is possible to switch contexts in fewer than 5 nanoseconds. A caveat to this rapid context switching is that time will be required to distribute the “switch to” lines throughout the chip. These lines indicate which context the device is supposed to switch to upon receiving the “switch” signal. However, given that the “switch to” lines are stable, the context switch can take place as described above. Note that this delay in switching is merely a latency and can therefore be factored into the logic that initiates a switch. Since the switch can be initiated by the active context or via external stimulus this latency is easily accounted for.

2.1.7 Block RAM

One of the new features added to the prototype CSRC device is the block RAM. The final CSRC device has two 256x8 dual port synchronous RAM blocks per pipe. Since there are eight pipes (each with 8 CSLAs in each pipe), there are 16 block RAMs in total. Traditionally,

commercial FPGA vendors tend to employ either block RAM (Altera) or distributed RAM (Xilinx). However, Sanders experience has shown that both types of RAM are valuable to computation. For this reason, the CSRC device employs both types of RAM. In fact, the CSRC device architecture emerged *prior* to commercial devices, such as the Xilinx Virtex family, that now utilize both distributed and block RAMs.

2.1.8 High Speed Direct Connect Routing

Subsequent to developing the prototype CSRC FPGA architecture, a means for implementing constant coefficient multiplies more efficiently was determined. Figure 2.1.8 depicts the additional routing developed for the final CSRC IC. As can be seen, the result of a logic cell can be forwarded to its counterparts on adjacent logic array or even fed back to itself. Optionally, the forwarded result of a logic array can be shifted down by 0-7 bits before forwarding. Since the relative routing delay is directly proportional to the level of routing (I, II, or III), it is advantageous to keep routes in lower level routes. Without this “fast routing”, communication between CSLAs requires the use of Level II routing. However, if data is routed between adjacent CSLAs, this direct routing can be used which alleviates the need to get onto Level II routing. Hence, routing delays go down and performance goes up.

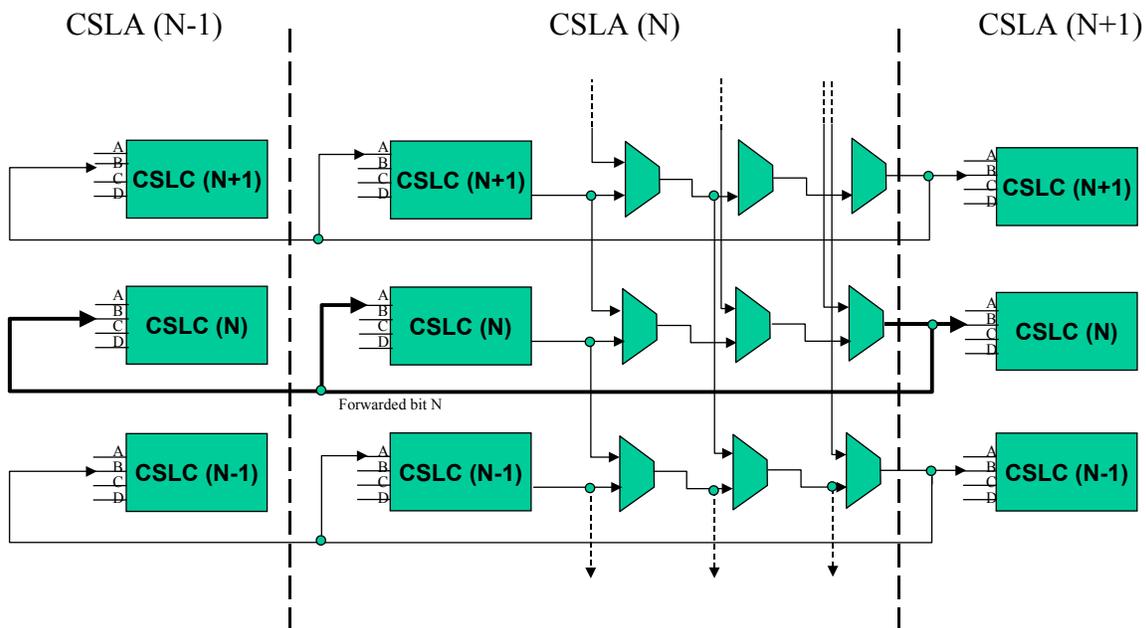


Figure 2.1.8: Direct / Shift Routing

2.1.9 Programming

The bitstreams for the CSRC FPGAs are downloaded serially (8 bit parallel for the final CSRC device). The user is required to specify which context is about to be downloaded and then supply a clock and data. By repeating this process four times, the user can configure all four on-chip configurations. Note that the configuration being downloaded can not be active during configuration download. However, inactive contexts can be downloaded while another context

is active and running. Additionally, a bitstream may be downloaded by the active context. In this manner, one can envision the possibility of passing compressed or encrypted bitstreams into the active context so that it may download an inactive context after uncompressing or decrypting the bitstream.

The device will power up, prior to downloading bitstream(s), in a known state possessed by all four configurations. This provides the user with the ability to determine if the device is operational prior to use. This “known state”, is both benign and affords built-in self-test (BIST). A random number generator passes data through *all* of the CSLCs and compares the results at the output, indicating pass or fail on an output pin. This pin can be monitored to verify that the device is functioning properly.

2.1.10 CSRC Control Block

The CSRC Control Block supports both external and internal programming as well as internal and external switching requests. Programming requests are denied if an attempt is made to program an active context because active contexts can not be programmed. A switching request is denied if an attempt is made to switch to a context that is either currently being programmed, not programmed, or is being requested for programming. An external switch will always take precedence over a simultaneous internal switch request to ensure that the user does not lose control of the device. Once the programming request of a context has been submitted and accepted, the control block will look for a preamble pattern in the bitstream to sync to. From that point on, the bitstream will be passed on the context being programmed. A counter will detect the end of the bitstream and complete the programming sequence.

2.2 Reconfigurable Computing Module (RCM) Description

The Reconfigurable Computing Module’s (RCM) primary objectives were to provide hardware to demonstrate the operation of the Context Switching Reconfigurable Computing (CSRC) device and to be a commercially viable processor with on board reconfigurable and context switching logic. The architecture of the RCM provides good general use and extensive flexibility in the configurations. See Figure 2.2.1.

The Reconfigurable Computing Module form factor was required to be on a commercial standard that would allow connection into a commercially available host computer system. The PCI long card form factor was selected because of its high performance, capabilities for full concurrency with processor/memories subsystems, ease of use and support of multiple families of processors as well as future generations of processors (by bridges or by direct integration).

Requirements on the RCM called for a node processor to be modern, main line with floating-point capability and extensive software development tool support. The processor selected was the MPC750 RISC Microprocessor. The MPC750 is targeted for low-cost and low-power systems and consists of a processor core and an internal L2 Tag combined with a dedicated L2 cache interface and a 60x bus. The PowerPC 750 microprocessors are super-scalar, capable of issuing three instructions per clock cycle into six independent execution units: two integer units,

floating point unit, branch processing unit, load/store unit and system register unit. The ability to execute multiple instructions in parallel, to pipeline instructions, and the use of simple instructions with rapid execution times yields maximum efficiency and throughput for PowerPC 750 systems.

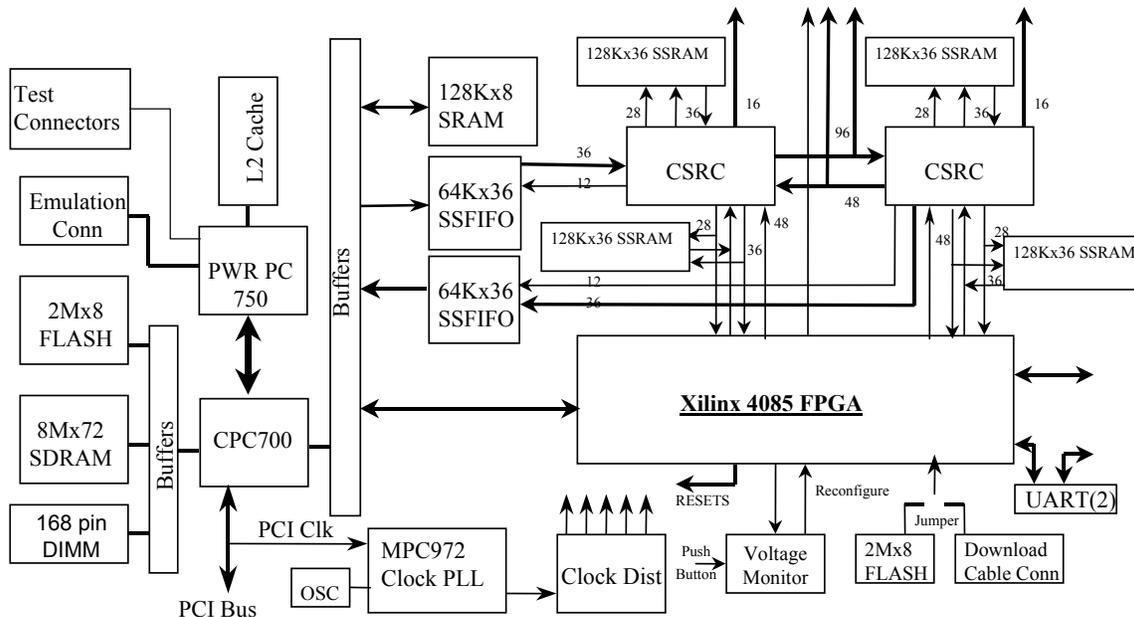


Figure 2.2.1: RCM Block Diagram

The processor is rated to operate at a core clock rate of 300 MHz and a bus clock ranging from 25 to 83.3 MHz. The MPC750 processor has complete support for a private L2 cache. A pair of 128K x 36 Synchronous SRAMS is connected to provide 1 Mbyte of cache with byte parity error detection capability. The clock speed of the L2 cache interface is controlled by the processor, it is synchronous with the CPU core and can be set to core L2 ratios of: 1, 1.5, 2, 2.5 or 3:1.

A CPC700 device provides the connection between the PCI bus and the processor. The CPC700 provides a PowerPC common hardware reference platform (CHRP) compliant bridge between the Power PC microprocessor and the PCI bus. The CPC700 integrates secondary cache control and a high performance memory controller. The CPC700 provides an integrated high-bandwidth, high-performance, TTL-compatible interface between a 60x processor, a secondary (L2) cache or additional 60x processors, the PCI bus and main memory. The initiator and target PCI interface is 32 bit wide and PCI 2.1 compliant.

The RCM's main memory is composed of SDRAM's arranged in up to 4 banks. Bank 0 consists of a 64Mbyte arranged as 8M x 64. There is one 168-pin DIMM connector to accept commercially available SDRAM modules. The main memory is controlled by the CPC700. The MPC106 provides byte parity. When less than the full 64 bit data bus is written with parity operating, the 106 registers the write data, reads the addressed 64 bit wide location, replaces the

read data with the appropriate write data and recalculates the error control bits on the full 64 bits before it writes to RAM.

The processor to CSRC communications is primarily through two sets of FIFOs that are 36 bits wide. The depth is dependent on the version of the RCM, the FIFO devices are pin compatible to support anywhere from 8K deep to 64K deep. The FIFOs have configurable almost full and almost empty flags. These flags are inputs to the support FPGA, which make the flag values available to the processor or CSRC devices. The FPGA could be configured to generate interrupts on flag conditions or just to provide flag status.

The RCM supports two CSRC devices in 560 BGA packages. Each device is connected to private SSRAM (128k x 36). There is another 128K x 8 SRAM that is shared with the support FPGA. The 64 signal lines connected to the SRAM and FPGA are general purpose and may be used for shared control of the SRAM or for communications between the CSRC and the FPGA, or a mix. The two CSRC devices are directly connected with 144 signal lines and there are 48 signal lines from each CSRC to the support FPGA. CSRC1 is connected to a 36 bit wide FIFO to the processor bus. The assumed data flow is thus from the processor through the CSRC1 to CSRC2 and back to the processor. If both devices need to send data to the node processor, the CSRC1 sends its data to CSRC2. CSRC1 is in control of this FIFO interface and when there is insufficient data, it will let the pipeline flush and suspend operations until new data is available. If there is a need for “direct access” to the main memory from the CSRCs, the more than one hundred connections between each CSRC and the FPGA can be used with an appropriate FPGA design to allow the CSRC to gain access to the processor bus. Either the FPGA provides clock domain translation or the CSRC operates synchronously with the processor section. The CSRC devices are configured via the support FPGA. The data may come from the host processor via the PCI bus, the node processor or from the configuration FLASH attached to the FPGA.

The Xilinx FPGA is intended to provide a variety of support functions. It is packaged in a 560 BGA. The FPGA contains as a minimum the ability to receive interrupt requests from the host processor, manage FIFO control flags, program the CSRC devices, and serve as a DMA controller to move data to and from the CSRC devices.

In Summary, the key identifiable features of the RCM board (Fig. 2.2.2) are as follows:

64 Mbytes SDRAM	32k bytes each (instruction and data) L1 cache
Up to 128Mbytes DIMM	1 Mbyte L2 cache
2 Independent memory banks per CSRC	2 CSRC ICs
Input & Output FIFOs (16k x 36)	Windows NT limitation - Only 128MB accessible from PC - Not true with Linux
300MHz PPC750	PPC has access to all memory space
Xilinx 4085	Xilinx, FIFOs, DIMM, SDRAM, SRAM memory mapped from PPC750
66MHz Local Bus	

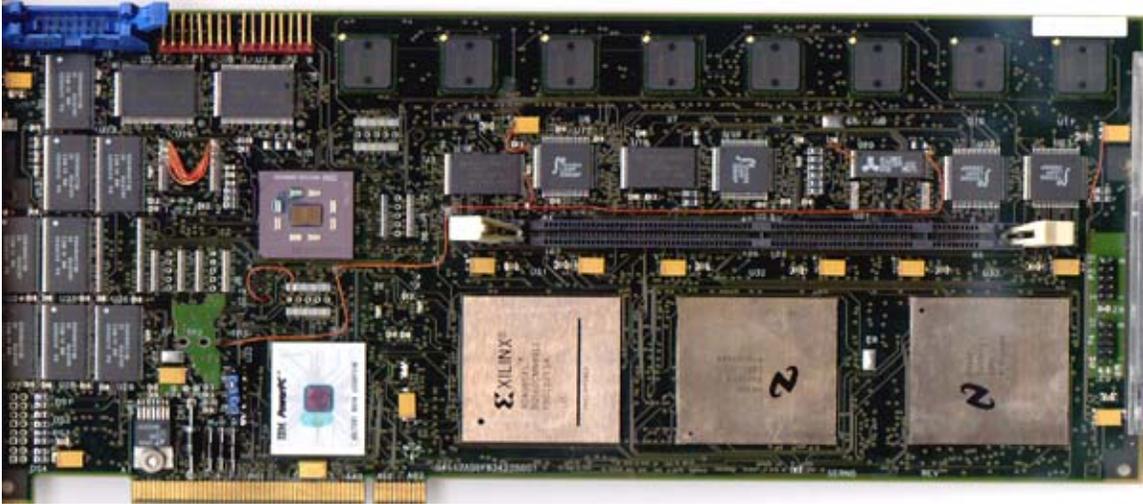


Figure 2.2.2: RCM Circuit Card

3.0 Technical Development

The DRACS team consists of BAE SYSTEMS, Virginia Tech and Brigham Young University (BYU). BYU was under a separate contract but performed development work in support of DRACS technology proliferation and insertion efforts. The following provides the final program status of the technical development efforts from each of these teams.

3.1 BAE SYSTEMS

3.1.1 Design Methodology

Fig. 3.1.1 represents the design flow methodology used for developing applications for the CSRC/RCM board.

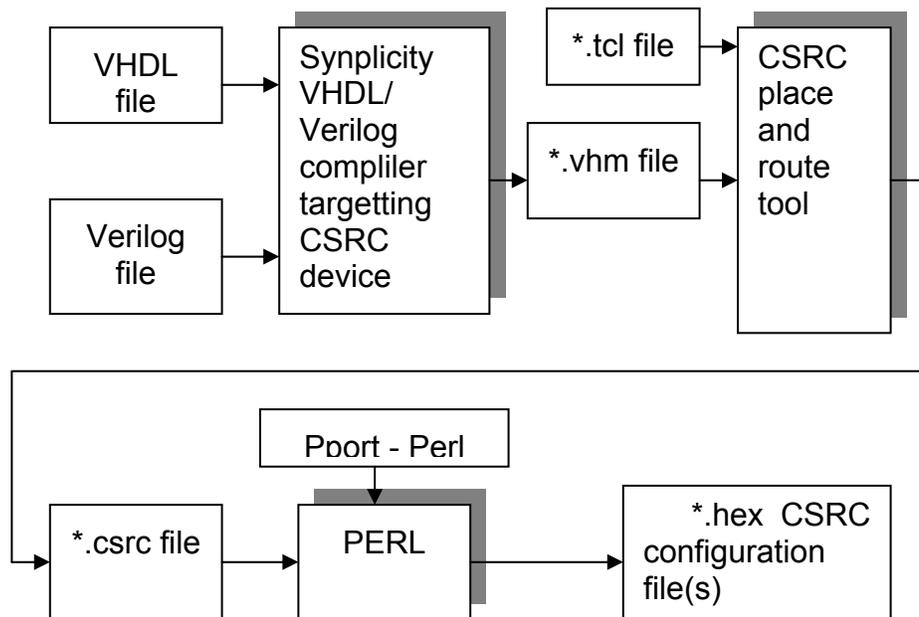


Figure 3.1.1: Design Flow for the CSRC/RCM Board

To better illustrate the design flow, all the files and relevant screen snapshots for a multiplier design are included below. The design process starts with a regular VHDL file such as the following for the multiplier, mul8.vhd:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mul8 is
    port(
        z : out unsigned(15 downto 0);
        a : in unsigned(7 downto 0);
        b : in unsigned(7 downto 0)
    );
end mul8;

architecture rtl of mul8 is

begin
    z <= a * b;
end rtl;
```

Synplify is then used to compile this file targeting the CSRC chip. In the Synplicity window (Fig. 3.1.2), the “Target” is Dyna Chip DY6000. This is the CSRC chip.

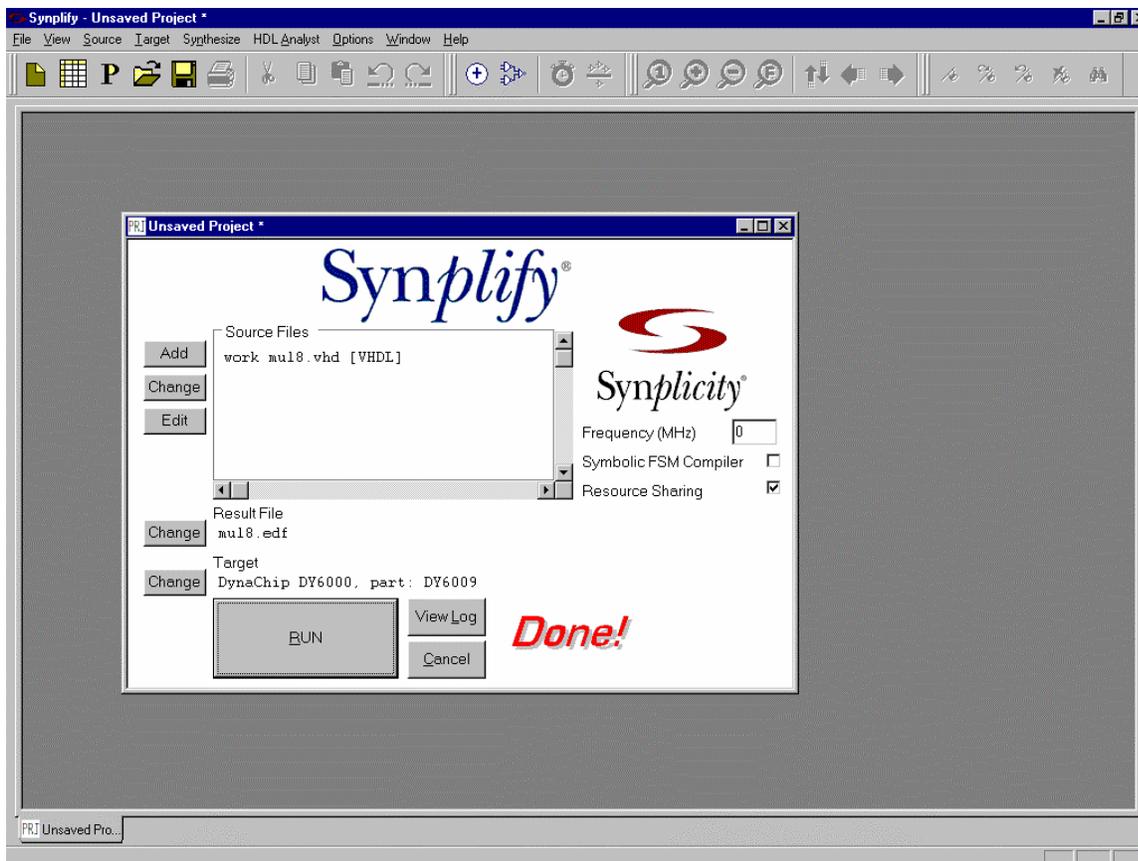


Figure 3.1.2: Synplicity Window

From this process, mul8.vhm file is produced. This is the HDL file which contains only primitives available in the CSRC device.

The next step is to run the CSRC place and route tool. At this stage one usually uses a constraint file to lock pins, specify locations of certain design primitives, etc. along with the *.vhm file. Here is an example of a constraint file which assigns I/O pins of the multiplier to specific locations. File mul8.tcl:

location "a_0" "F31"
location "a_1" "G30"
location "a_2" "G33"
location "a_3" "J30"
location "a_4" "F32"
location "a_5" "G31"
location "a_6" "H30"
location "a_7" "J31"

location "b_0" "G29"
location "b_1" "G32"
location "b_2" "H31"
location "b_3" "J32"
location "b_4" "F33"
location "b_5" "H29"
location "b_6" "H32"
location "b_7" "J33"

location "z_0" "N32"
location "z_1" "P31"
location "z_2" "R29"
location "z_3" "T30"
location "z_4" "N33"
location "z_5" "P32"
location "z_6" "R31"
location "z_7" "T29"
location "z_8" "P30"
location "z_9" "P33"
location "z_10" "R32"
location "z_11" "T31"
location "z_12" "P29"
location "z_13" "R30"
location "z_14" "R33"
location "z_15" "T32"

To run the place and route tools one has to execute the csrc.exe file as shown in Fig. 3.1.3:

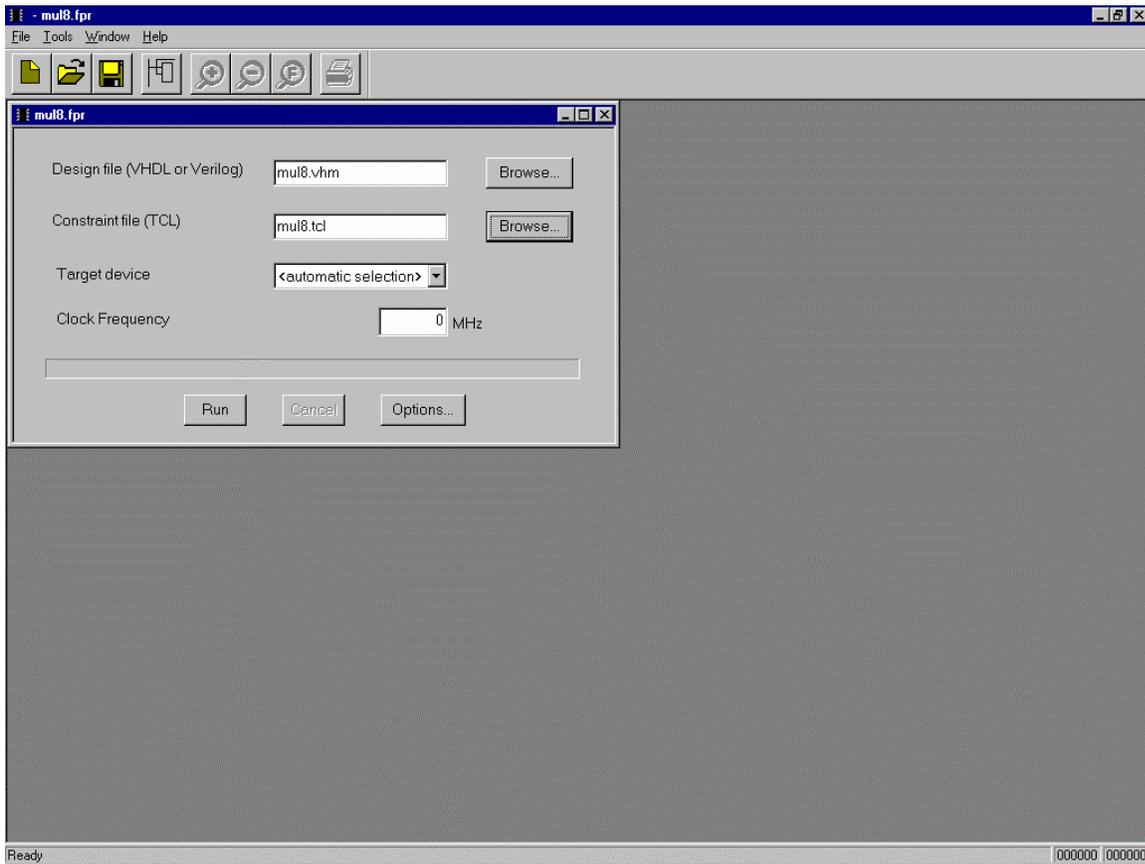


Figure 3.1.3: Executing csrc.exe

After running the CSRC tool, mul8.csrc file is generated as well as mul8.hex. Mul8.hex file is the configuration file ready to be downloaded to the CSRC device. Note that in the block diagram, Perl script processing follows place and route as a separate block. Perl script processing is in fact incorporated as part of the place and route tool processing. This provides more transparency and convenience to the user.

When the tool is finished, details of the design can be seen as shown in Fig. 3.1.4:

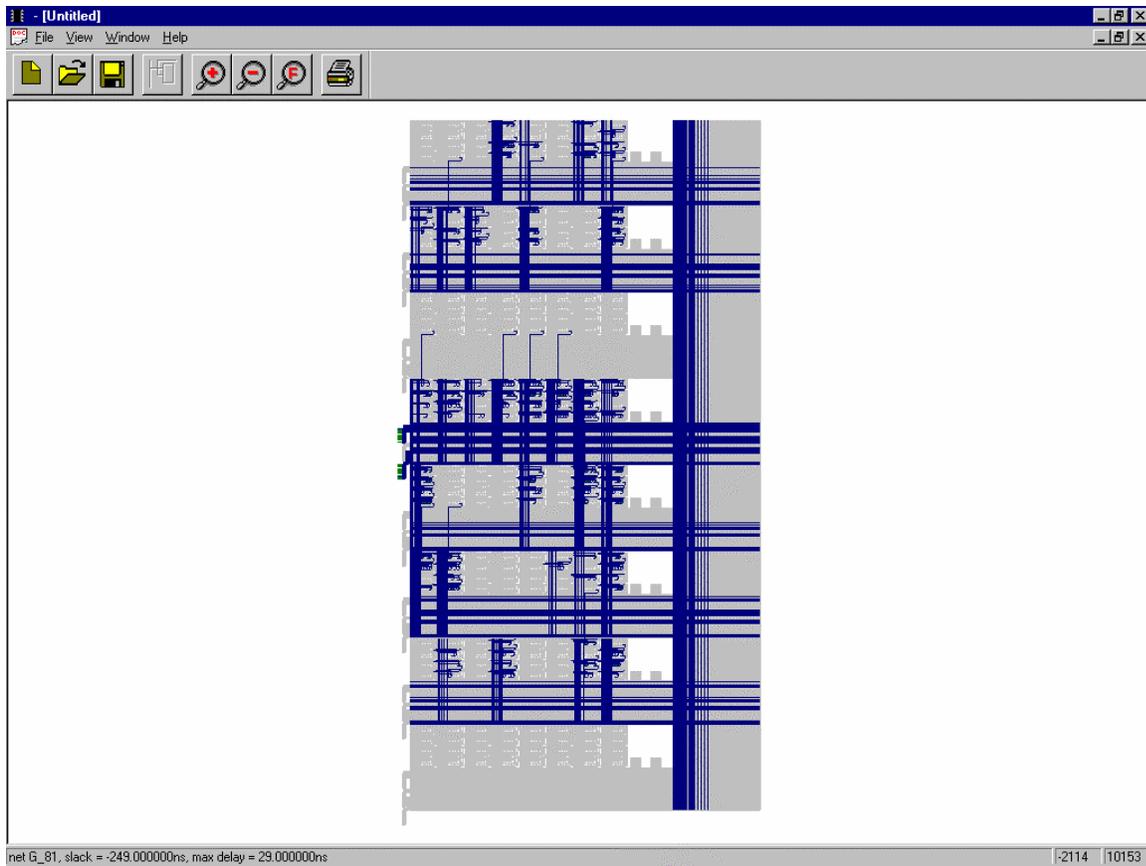


Figure 3.1.4: CSRC Detailed Design View

One can zoom in on a particular portion of the chip as shown in Fig. 3.1.5:

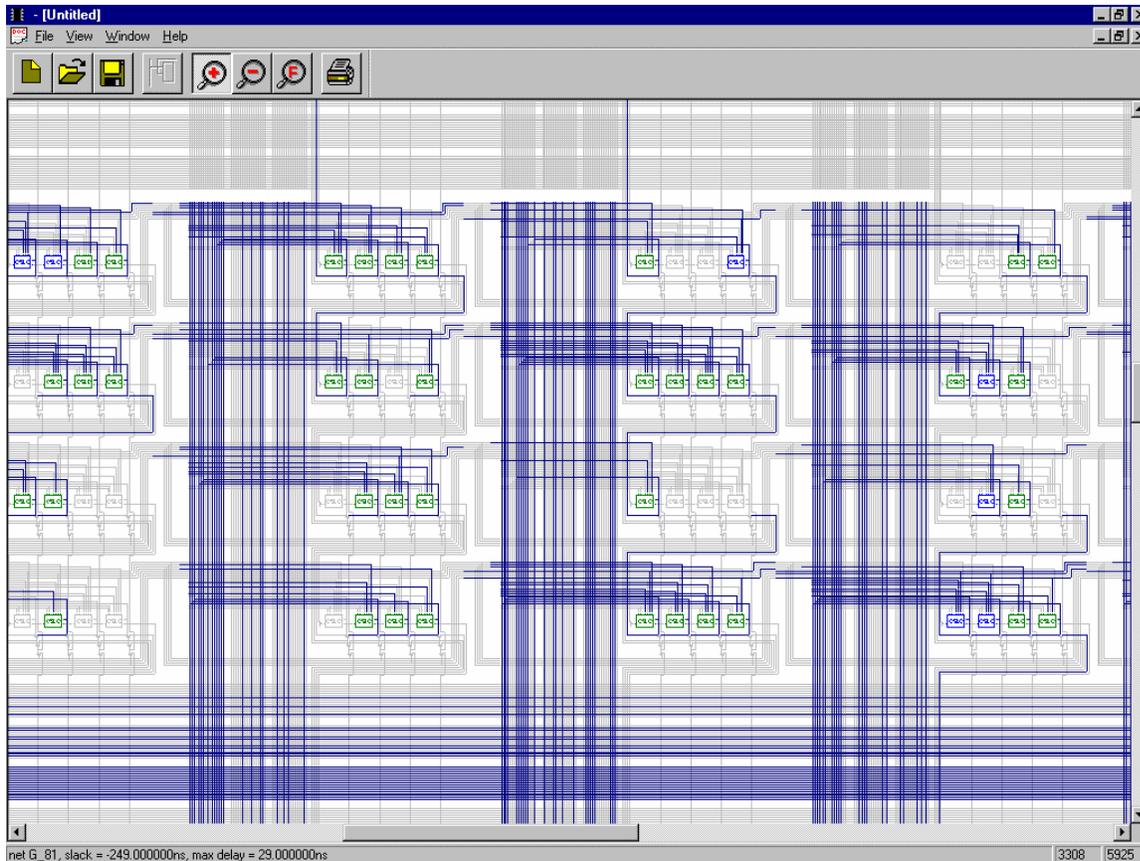


Figure 3.1.5: CSRC Detailed Design View (Zoom)

3.1.2 CSRC Tools

Although developed under the former CSRC program, the capabilities and features of the CSRC tools have, under the DRACS efforts, only then been fully implemented. As such, issues arose with the tools that required significant debugging efforts. The CSRC tools developer had been working closely with BAE SYSTEMS to provide debugging and required tools modification support. Significant progress was made in the debugging effort of the CSRC tools in this fashion, however these tools corrections were very slow to achieve as the tools developer's time became more and more limited and eventually a stopping point for continued forward progress on the program. The decision was made to bring the tools (source code and all) "in-house". This required an approximate 2 month delay in the program while bringing our in-house knowledge and capability up to speed on the immense tools code and functionality. Once this was achieved, significant, and much more efficient, debugging advances were achieved with the tools. In addition, now tools enhancements, such as carry chain, could be implemented.

Examples of the significant bugs that were resolved included fixing level 3) (L3) routing, development of fixes and the methodology for data/state sharing between contexts, resolution of problems in utilizing the chip's routing resources, and the resolution of logic synthesis errors, to name a few.

As a prime example demonstrating our in-house expertise acquired with the CSRC tools and demonstrating a major improvement to the tools, successful implementation and testing of a 16-bit adder using carry chain has been achieved. This accomplishment was achieved with new routines generated during the processes of mapping and binary stream generation. Results of this implementation can be found in the Figures below:

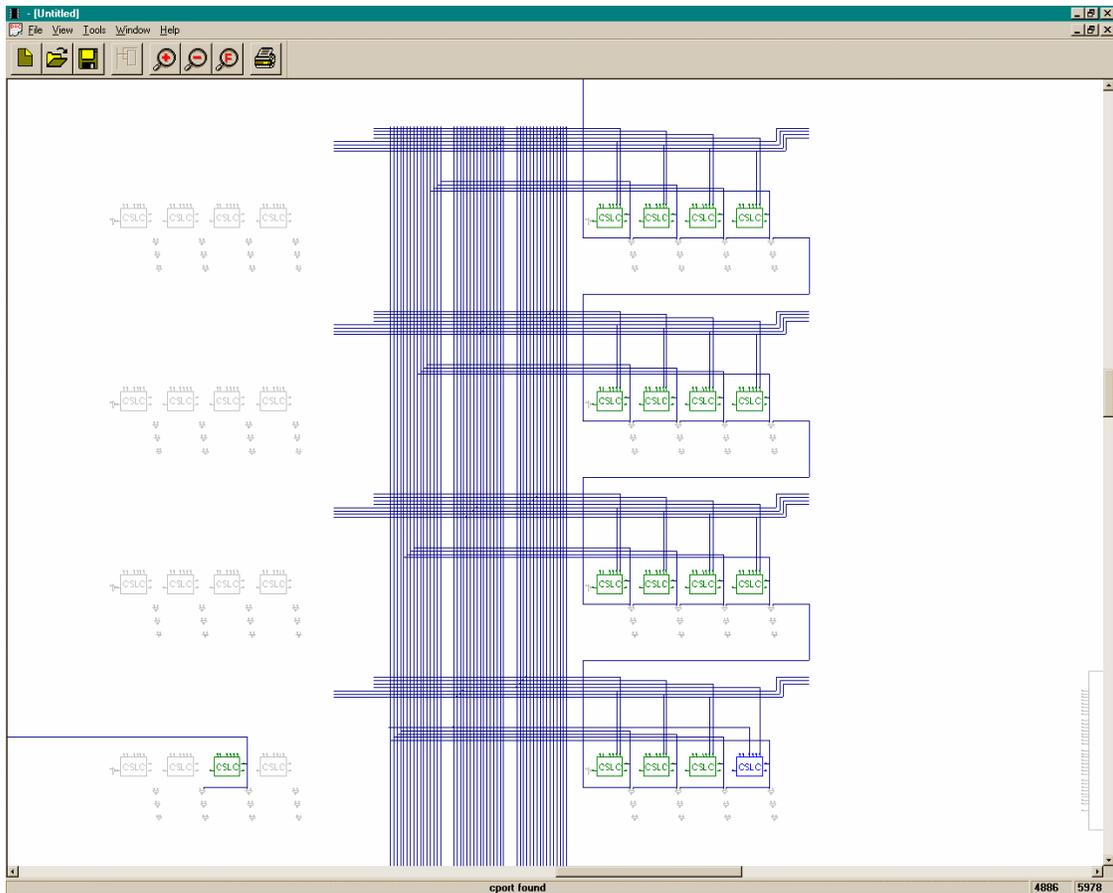


Figure 3.1.6: 16-Bit Adder With Carry Chain

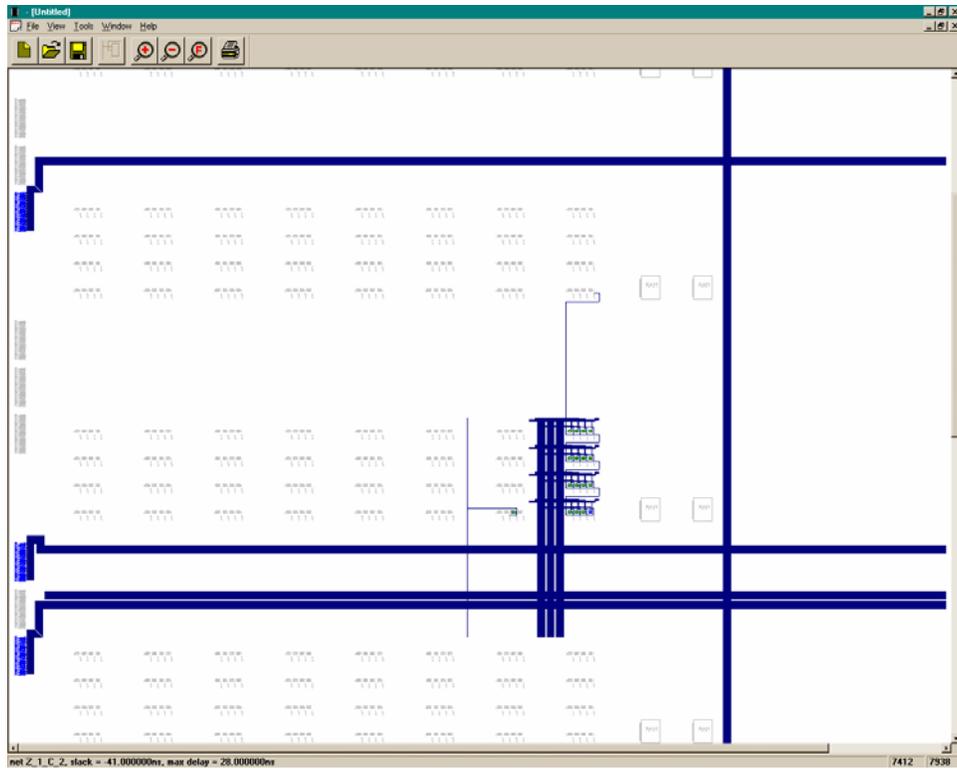


Figure 3.1.7: Overview of 16-Bit Adder With Carry Chain

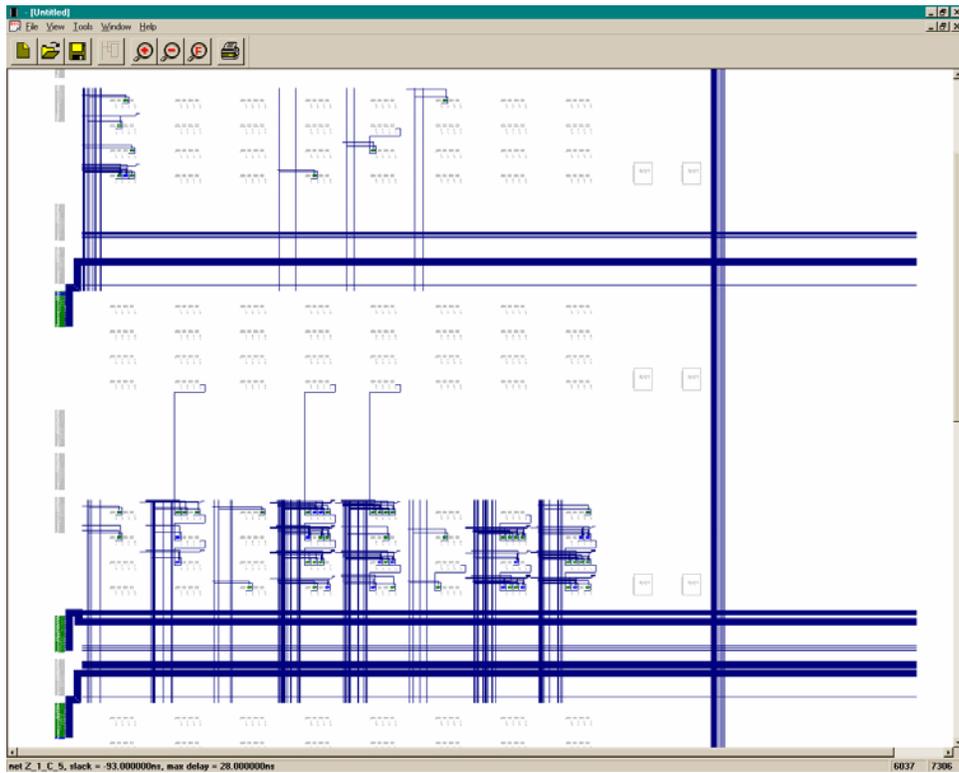


Figure 3.1.8: 16-Bit Adder Without Carry Chain

Once we had managed a preplacement of a carry chain, the mapping process needed to be revisited. The mapping process will randomly select logic elements (in a CSLC site) and move them within the CSRC to another CSLC if the cost of the move reduces the overall cost of the design. As it stood, this mapping process would either take an element of a carry chain, or it would supplant an element in the chain with a non-carry logic element. The mapper was revised so that a non-carry logic element could not be placed where a carry chain resided. Additionally, when a carry element was selected for movement, the entire carry chain would be moved. This is crucial, as the use of carry logic infers a strong constraint on placement of elements to neighboring CSLCs.

Lastly, the process of translating the design into a hardware readable format needed to be modified. The CSRC tool exports a text-based .csrc file that is processed by Perl script to create a binary representation that is readily acceptable by the CSRC device. The use of carry chain logic was already supported by the Perl script. All that was needed was to export the necessary commands to activate the carry chain in hardware.

The first issue was with routing, as the carry chain logic elements would utilize special Pips. The CSRC tool previously based the Pip selection on bit ordering of the logic. A trap was added for these special Pips.

The remaining issue was with activation of the carry chain in the device. The carry chain can span the entire CSRC device. Special codes are used to select the beginning and end of a particular carry segment within the overall device. Additionally, there are also codes that allow for carry to span CSLC nibbles as well as individual CSLCs. All of these cases have been trapped for and exported for accurate interpretation by the Perl script.

3.1.3 CSRC RCM Board Testing Environment

At BAE SYSTEMS we established a test bed for testing and demonstrating the RCM board/CSRC chip technology and devices and associated software/tools. A block diagram of our system is shown in Fig. 3.1.9.

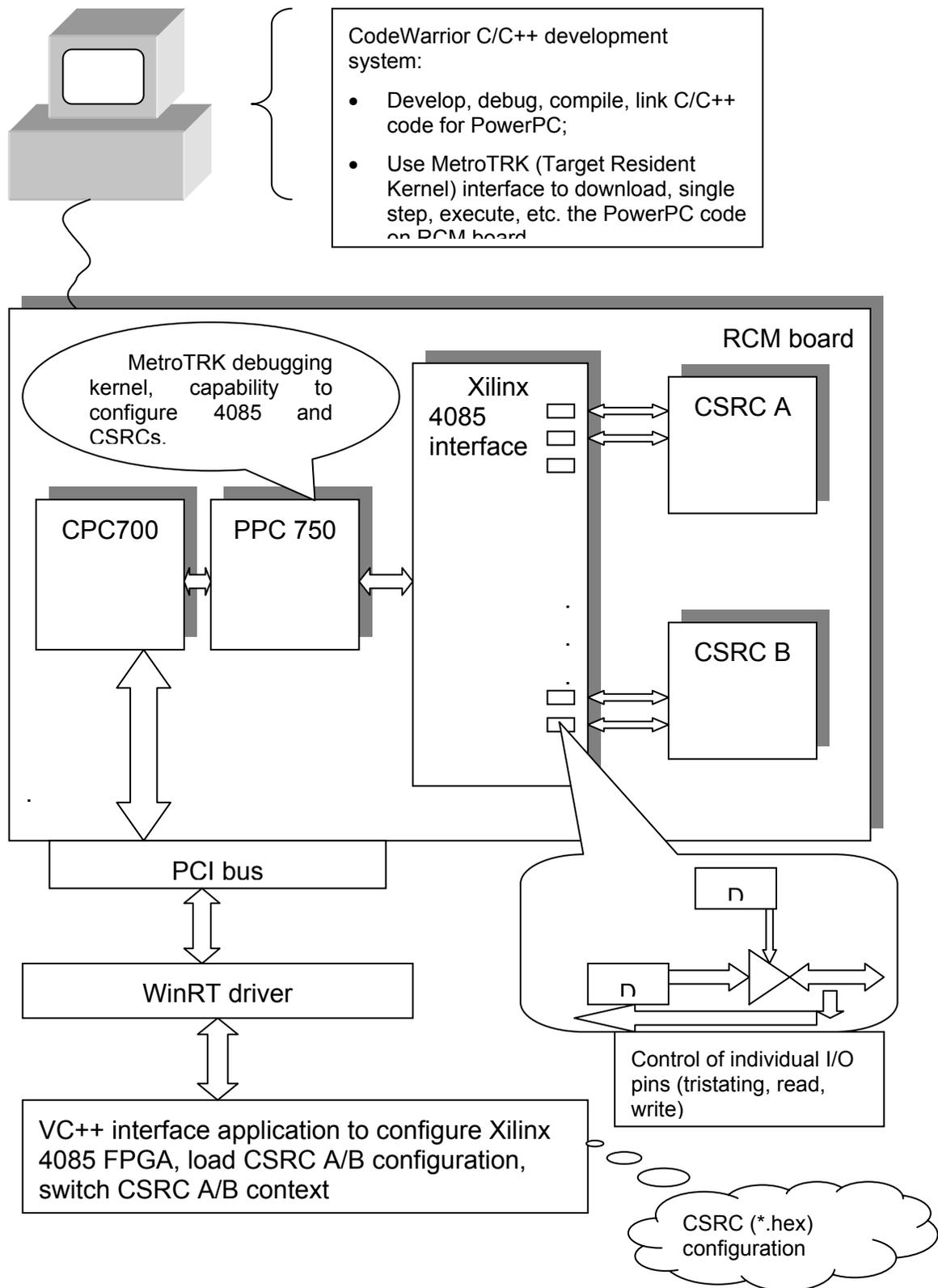


Figure 3.1.9: BAE SYSTEMS DRACS Test Bed

As part of our test bed, we developed a Xilinx 4085 interface. The current interface design supports both CSRC configuration and control of individual I/O pins for driving the data onto CSRC buses and reading the data from CSRC buses.

The testing/debugging process using our test bed involves the following:

- Power up the PC containing the CSRC RCM board
- Start the WinRT driver
- Execute the RCM/CSRC programming interface program (_RCM_board_configuration.exe)
- Load the 4085 configuration

At this point, the system is ready to load CSRC configuration(s), switch contexts, run and/or debug PowerPC code.

The testing of the CSRC design is done by writing some PowerPC application which drives a certain set of pins of either CSRC A or CSRC B and reads another set of CSRC A/B pins. Using the debugger, one can single step through the PPC code and examine variables containing the state of CSRC output pins.

The current 4085 design uses software control of the CSRC clocks. The BAE SYSTEMS test bed does not support FIFO. FIFO support is implemented via modifications to the application design code (VHDL) for implementing the Virginia Tech RTR environment FIFO support and running the application under the VT environment.

3.1.4 Host Driven Demonstration

The objective of the host driven demo was to demonstrate the context switching capability of the CSRC technology in a RTR environment applied to a DOD related application. To first lay the ground work or basis for the demonstration, consider Fig. 3.1.10 below. The figure represents a typical Electronic Warfare application of signal detection and classification.

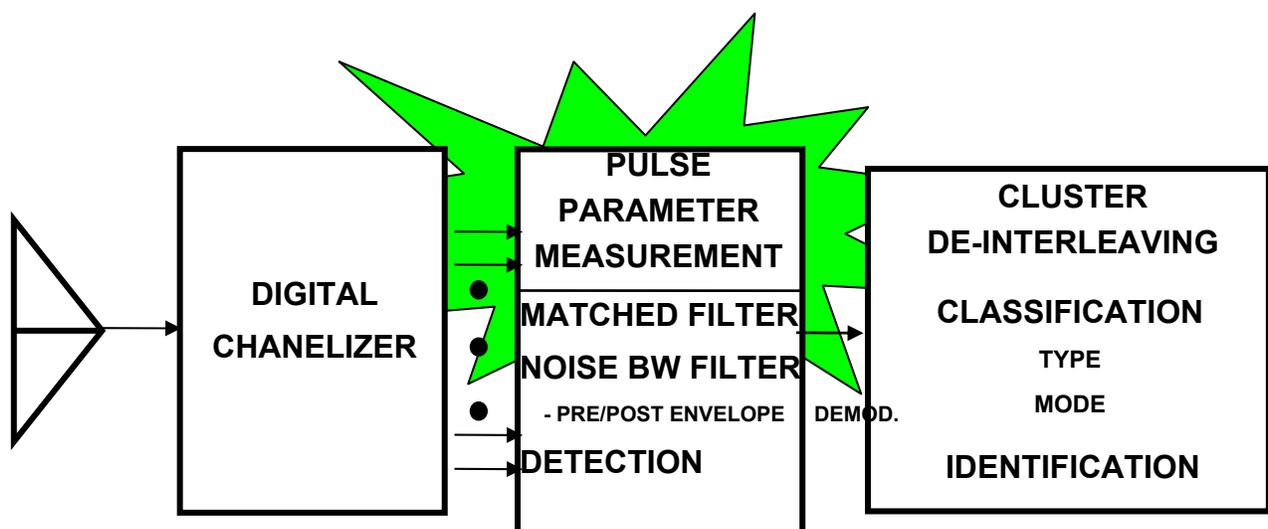


Figure 3.1.10: Typical EW Application of Signal Detection and Classification

As shown in the figure, parameter measurement must characterize and measure the critical parameters of each pulse of each signal in real-time as they arrive. Any number of signal parameters are used to separate and identify complex signals. Figure 3.1.11 shows examples of such measured parameters.

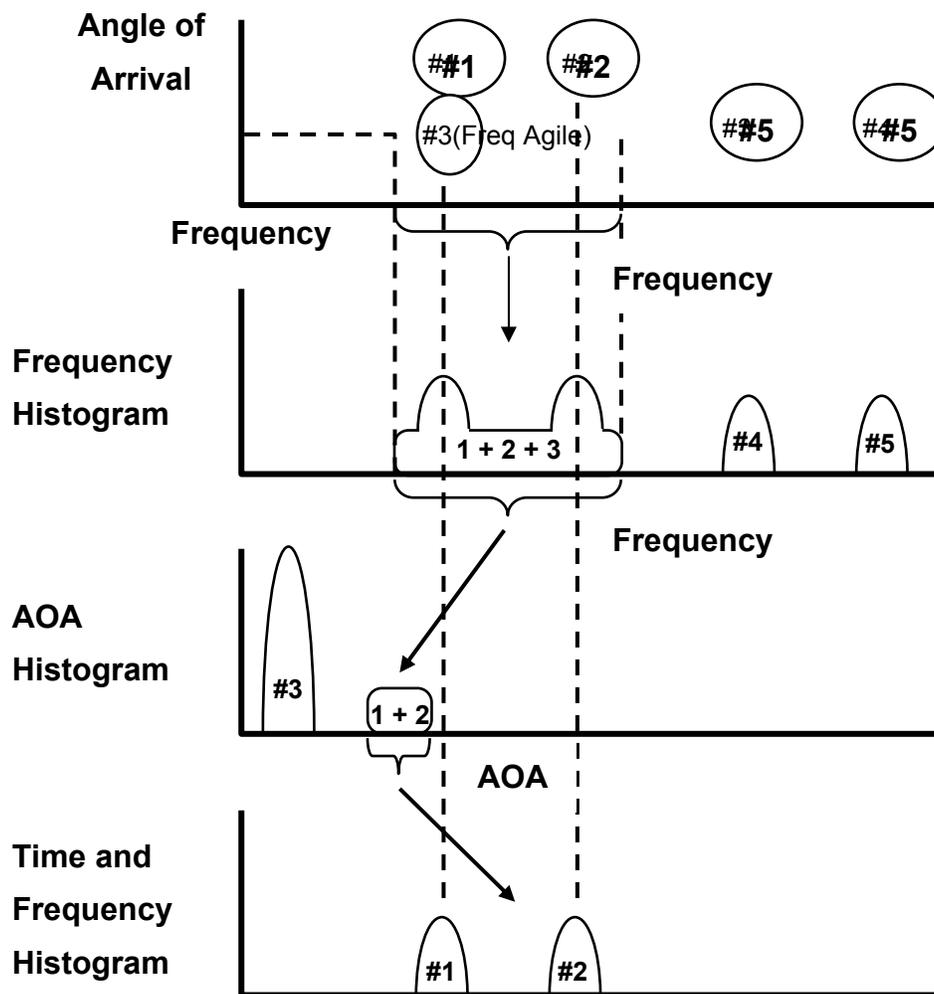


Figure 3.1.11: Typical Parameter Measurements

A typical benign environment for signals is a 40 MHz bandwidth, 0.5 to 10 us pulses, 500 us between pulses (pulse repetition interval or PRI) per signal, 1 to 5 signals, and a signal bandwidth that is 10% of the input bandwidth. Variance in the signal parameter measurement can make it difficult, or impossible to separate signals. In addition, more accurate measurement up front can significantly reduce down stream processing requirements. One method of improving measurement accuracy is by tailoring the parameter measurement bandwidth to the incoming signal. The presence of multiple in-band signals degrades the performance of the

pulse parameter measurement. Identifying and filtering interfering signals or signals that are not of interest dynamically and in real time will improve performance. Runtime reconfigurability, such as that provided with CSRC technology under DRACS, is an efficient mechanism to implement real time pulse by pulse measurement band tailoring. The rate of change in a real environment, the finite number of emitters and the periodic nature of the signals of interest all contribute to the temporal locality of this application. It is ideally suited to CSRC.

We focussed the demonstration towards supporting an F-22 channelizer application, through a CSRC application to enhance pulse parameter measurement capability. The objective is to provide the F-22 channelizer with additional capability to filter out unwanted frequency components in any of the channels by implementing programmable FIR filters in CSRC devices. Given a channelized receiver with N channels, the general concept is to subdivide each channel into M sub-channels and be able to apply band-pass filtering to an arbitrary number of these subchannels. For the demonstration environment, the following requirements are imposed:

- Let $M=4$, leading to 16 possible filters
- Continuous processing required
- Support 30-60 MHz data rates
- Dynamic filter switching required
- Maximize the amount of “good” data that reaches the Pulse Parameter Measurement Unit (PPMU)

Figures 3.1.12 shows a basic representation of the channel division and Figure 3.1.13 shows where CSRC filters would reside in a typical system.

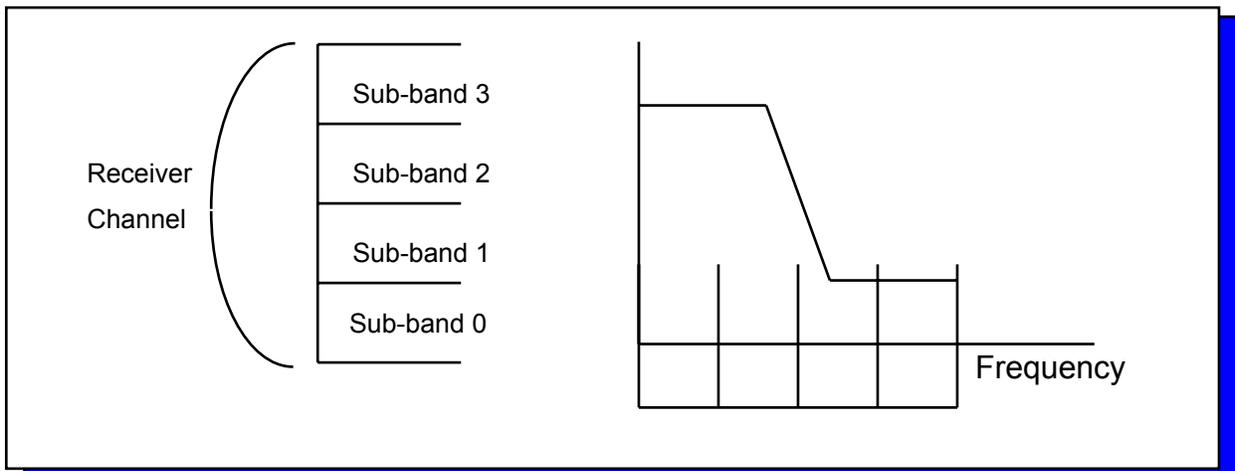


Figure 3.1.12: Division of Channel into Sub-bands

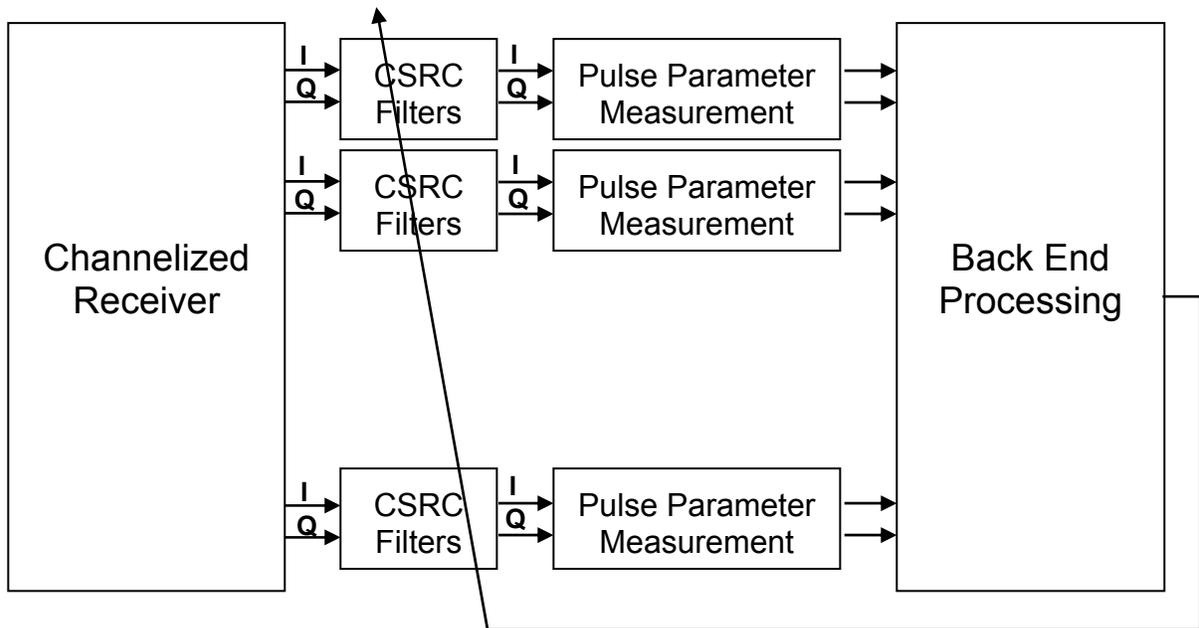


Figure 3.1.13: CSRC Filters in Pulse Parameter Processing Path

Three approaches to providing sub-band filtering were looked at. The first approach would require 16 contexts, each containing one of the desired filters (i.e. filters A through P). Switching to a new filter would be directed by the context manager. A caching strategy would need to be developed for the environment. A disadvantage with this method included a potential for data discontinuity of one filter length after the context switch. An advantage considered was that this method would allow for the largest filters by using the entire CSRC device.

The second approach looked at one filter (context), A, as an all-pass which required only one delay element. There would then be 15 contexts developed, each containing one of the desired filters, B through P, and the all-pass filter A. The context output would be user selectable as either filter A or the one of the other filters (see Figure 3.1.14).

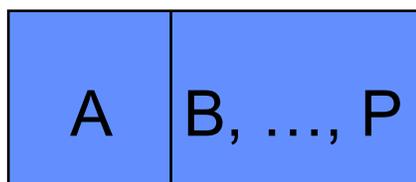


Figure 3.1.14: Filters in CSRC Contexts with Shared All-pass Filter

Since filter A is common to all contexts and located in the same place in each context, data sharing is perfect for this filter, i.e. no data discontinuities. After the context switch, filter A output would remain selected until the other filter pipeline fills. At this point, the context switch to the new filter would occur. Some related issues with this method include:

- There is a minimum number of contexts (M-1)
- The individual filters are smaller since they each must share the context with the all-pass filter A
- This method reintroduces a burst of all-pass data at every context switch.

The third method considered involved developing $M*(M-1)$ contexts, each containing two of the desired filters A through P (see Figure 3.1.15).

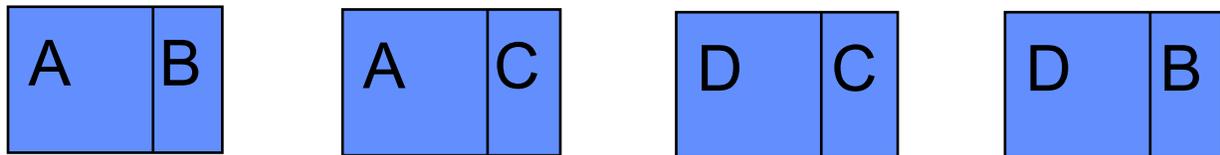


Figure 3.1.15: Filters in CSRC Contexts with Two Filters per Context

In this method, data sharing will always be perfect for one of the filters in a context, i.e. no data discontinuities. After the context switch, the formerly active filter output is kept selected until the other filter pipeline fills, at which point the output is switched to the new filter. Some disadvantages to this method include:

- Requires a large number of contexts, and as such places the greatest burden on a context caching algorithm
- The individual filters are half size since two filters occupy each context
- An all-pass filter must be explicitly switched to

On the other hand, this method makes it possible to track and filter chirp and other frequency agile signals. It also presents the smallest energy discontinuity to the PPMU.

Of the three options considered, option three represents the most flexible solution and was chosen for the demonstration.

With regards to the third method chosen, consider the two configurations below in a scenario of “smooth” switching and data sharing between filters without the introduction of any glitches at the output:

Configuration1: filter A and B (filter B is active in steady state)

Configuration2: filter B and C (filter C is active in steady state)

Assume that the currently active configuration is configuration 1. In this configuration, the data is coming out of filter B. When the switch is made to configuration 2, the data is shared between filter B in configuration 1 and filter B in configuration 2 (identical filters). The fact of switching is detected by circuitry in configuration 2 and the output data is still provided from the filter B until the data from filter C becomes available. Data from filter C will be available once the filter has charged up following being switched into the data stream. At that point, the output mux is switched and configuration 2 begins providing output data from filter C. Thus no switching artifacts are introduced.

The scenario for the host-driven demonstration, showing applicability towards F-22, involved simulated downstream identification by the PPMU of a pulse-on-pulse (POP) condition where two signals were mixed and inseparable within the 40 MHz bandwidth of a received channel. The following represents an example of a POP condition.

For illustration purposes, using a MATLAB-based signal generator completed under the DARPA sponsored Reconfigurable Algorithms for Adaptive Computing (RAAC) program, complex (I/Q) signals representing a pulse-on-pulse condition in a typical EW environment are as follows:

Signal 1:

- Pulse Width: 5000 ns
- Rise Time: 500 ns
- Frequency Offset: 14 MHz
- Signal-to-Noise Ratio 13

Signal 2:

- Pulse Width: 2000 ns
- Rise Time: 100 ns
- Frequency Offset: 25 MHz
- Signal-to-Noise Ratio 10

Common Parameters:

- Pulse Repetition (PRI) 500 us (kept common for demo purposes)
- Number of Pulses 10
- Samples Per Burst 1000
- Noise 2
- Number of Lead Samples 30

Represented below are the combined signal showing pulse-on-pulse (Figure 3.1.16) and individual Signals 1 and 2 (Figure 3.1.17).

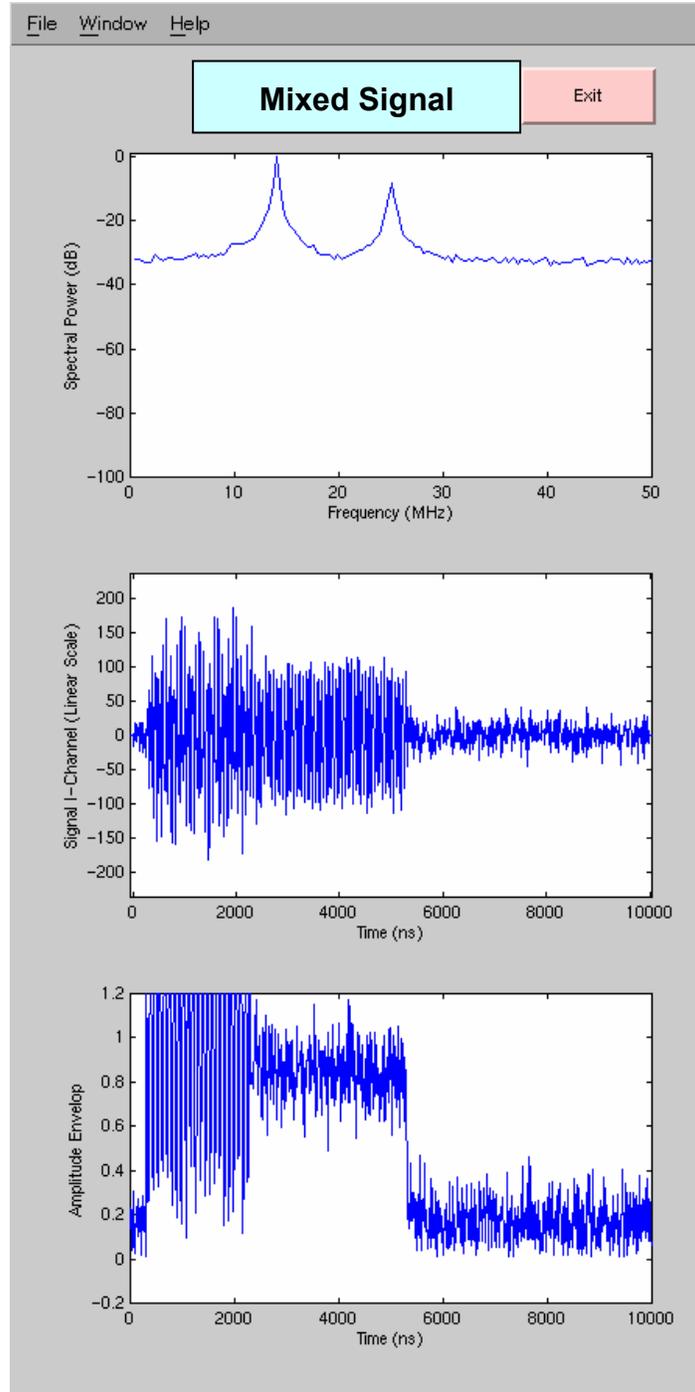


Figure 3.1.16: Mixed Signal Test Input, Pulse-on-Pulse Condition

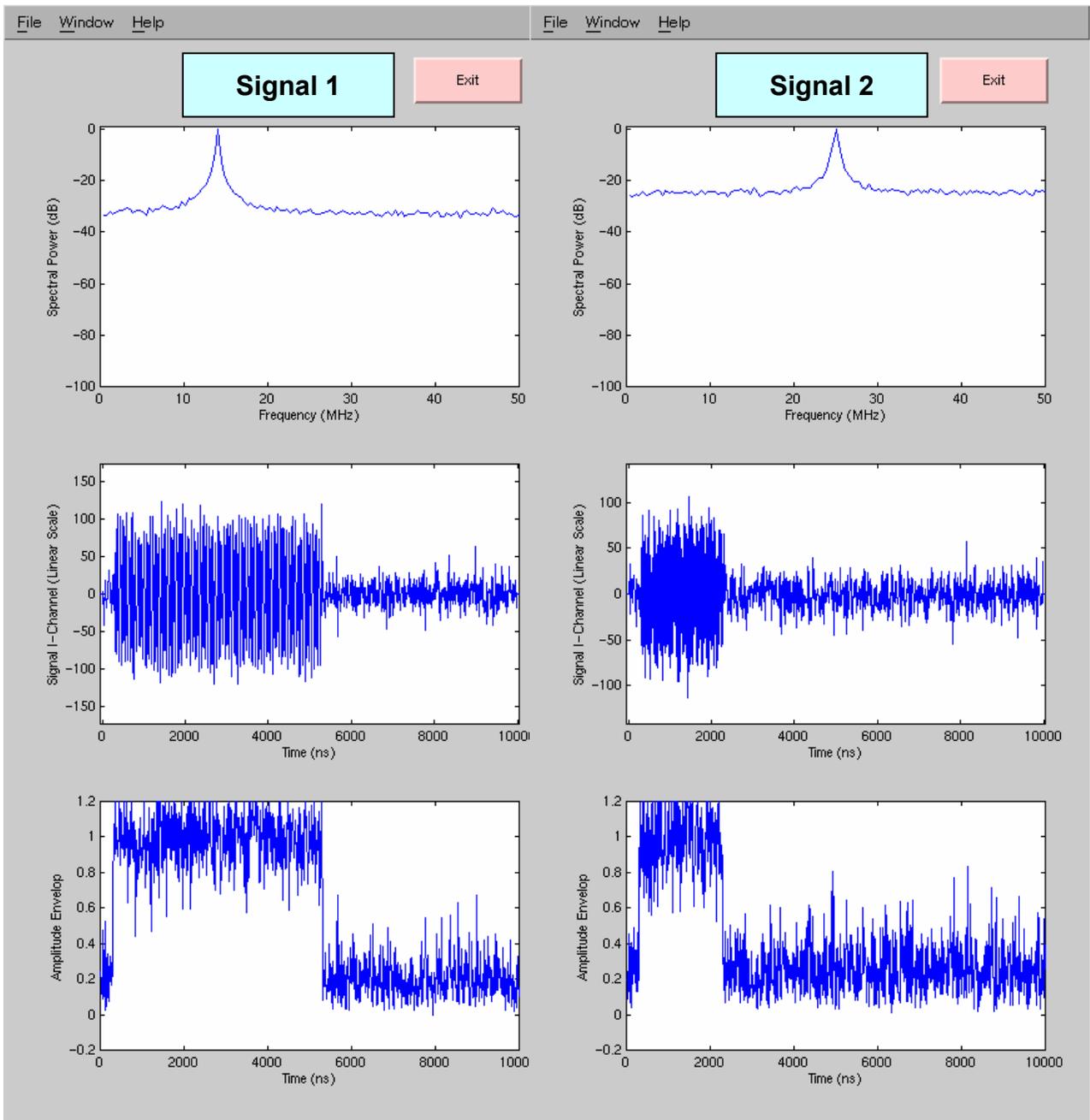


Figure 3.1.17: Mixed Test Signal Components, Signals 1 and 2

In preparation for the host-driven demo, we attempted to synthesize a circuit as shown in Fig. 3.1.18 containing two different 8-tap complex FIR filters (A and B). Input data was 8-bit wide and output data 16-bit wide.

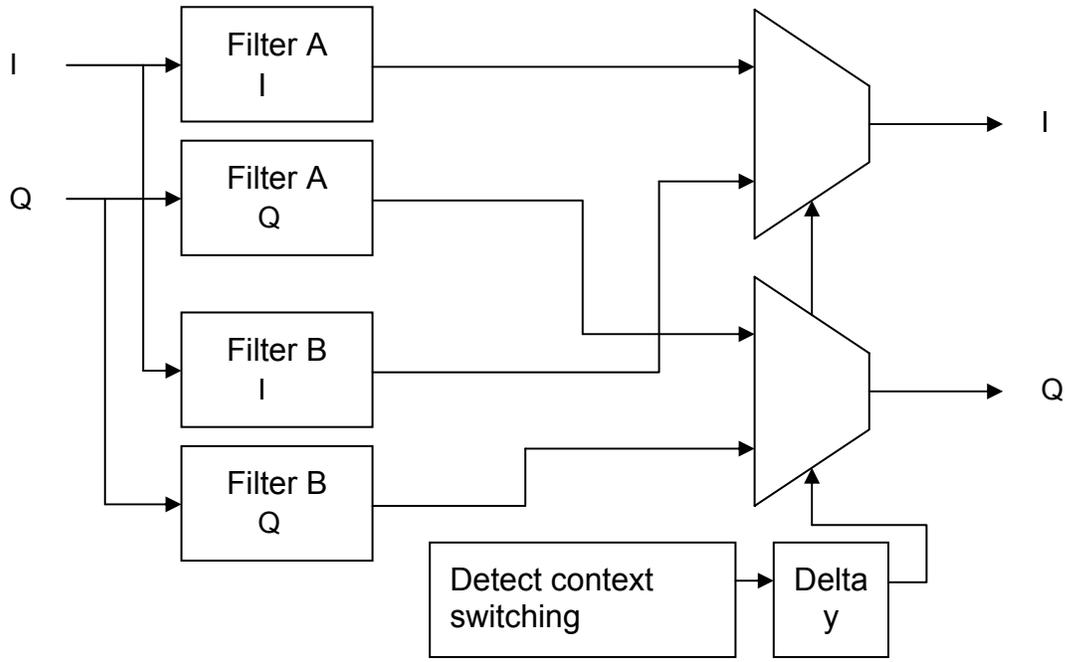


Figure 3.1.18: Design of Complex FIR Filters A&B

However, it turned out that the complete design would not fit into a CSRC context. We therefore decided to interleave I and Q data and to use modified FIR filters. This is a valid approach since the I and Q portions of the filter are identical (real coefficients).

Fig. 3.1.19 represents the original design filter:

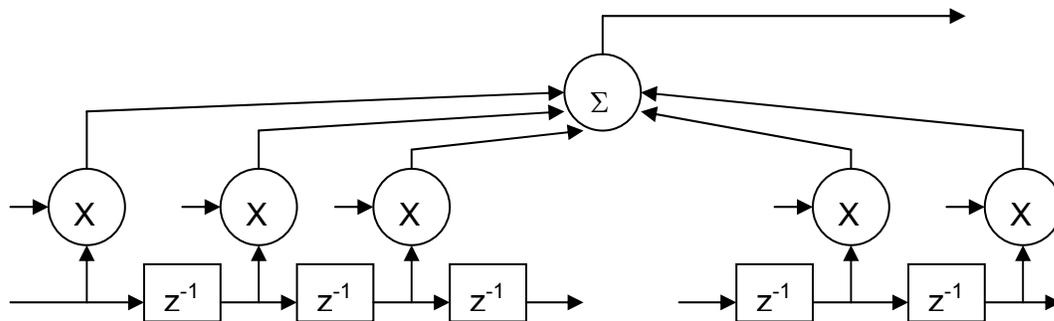


Figure 3.1.19: Unmodified FIR Filter

Fig. 3.1.20 represents the modified filter:

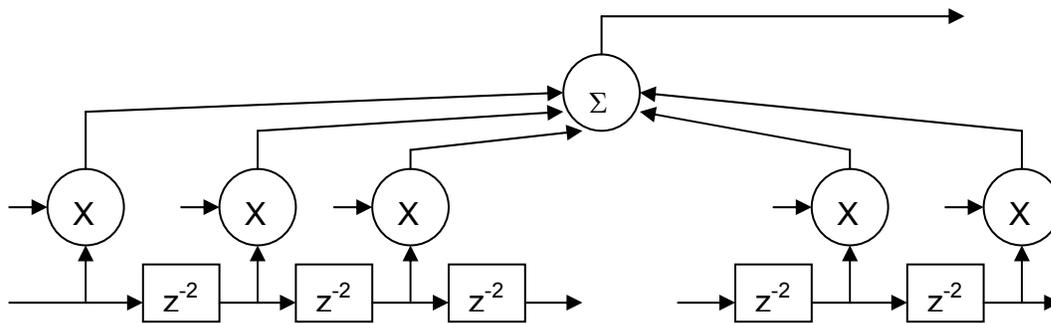


Figure 3.1.20: Modified FIR Filter

As shown above, the data presented to all the taps at any given moment comes from either I stream or Q stream. For the host-driven demo, three FIR filters were designed, filters A, B and C. Filter B was essentially an all-pass, wide band filter, and filters A and C were narrow-band filters each for a different upper or lower portion of the channel bandwidth. Due to routing issues with the CSRC tools, the final filter designs ended up as 5-tap 4-bit FIR filters. Of the 1024 available logic cells of a CSRC context, the filter designs required 352 logic cells for the context design with filters A and B, and 442 logic cells for the context design with filters B and C.

In preparation of the host environment for running the host-driven demonstration, BAE SYSTEMS developed a MATLAB host program that interfaces, through *.MEX files, to the C functions for interfacing with the test platform RCM module.

For the host-driven demo, the first active filter in CSRC at the start of the demonstration, filter B, had a “large” bandwidth sufficient to span the individual frequencies of the two mixed signals. In this situation, parameter measurements/pulse characteristics for each signal are corrupted by the presence of the other signal pulse occurring at the same time. Pulse-on-pulse conditions exist in actual systems/scenarios and contribute towards degradation of signal identification and classification in an EW system. Without an auto-detection routine for a pulse-on-pulse condition, the identification of a pulse-on-pulse condition was hard-coded within the host program to occur after a pre-determined number of input data samples. Identification of POP then initiated the context switch command (from host) to the RCM to switch in a band-pass filter with a narrow bandwidth. The filters residing on the four contexts of the CSRC (2 filters per context with one being the active filter) provided separation of the 40 MHz bandwidth into a wide band (filter B) and two narrow band regions (filters A and C). The specific sequence of filter changes and context switching was such that the signal of interest previously corrupted was isolated from the other signal. The figures below show the input data pulse sequence (POP existing) and the demonstration results of data output following filter and context switching to isolate the signal of interest.

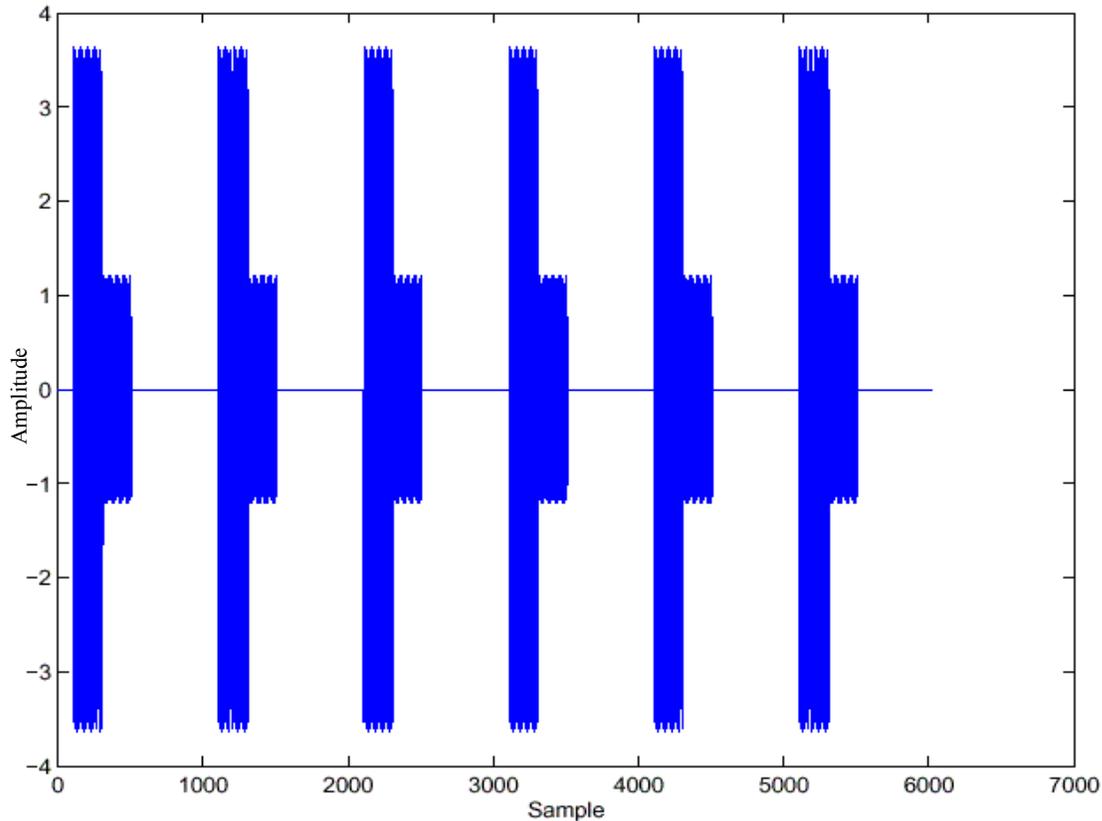


Figure 3.1.21: POP Signal Input Sequence

In the following figure, the sequence of filter changes and context switching used is revealed. With data passing through filter B, POP is detected (known by host in this case). In an attempt to isolate the signal of interest, the output of the active context containing filter B and C is switched to filter C output, thus providing separation of one of the signals through narrow band filter C. It is assumed for demo purposes that the isolated signal (as shown) is not the desired signal. In preparation for switching in sub-band filter A, the output of the active context is first switched back to filter B. The context switch command is initiated and the context switched to the second context, again looking at the output of filter B, but this time filter B of the second, now active context. This switching occurs in just 2 clock cycles (color change shows context switch) with no discontinuities in the data stream (since the filter B design in both contexts is the same and state information is immediately shared with the new filter B). After a set time (clock cycles) for charge up of filter A in the second context, the output is switched to narrow band filter A, which results in filter separation of the desired signal of interest. For display purposes of the output pulses to show proper operation of filter B in the second context, the switch to filter A was set to occur well after the charge up of filter A was completed.

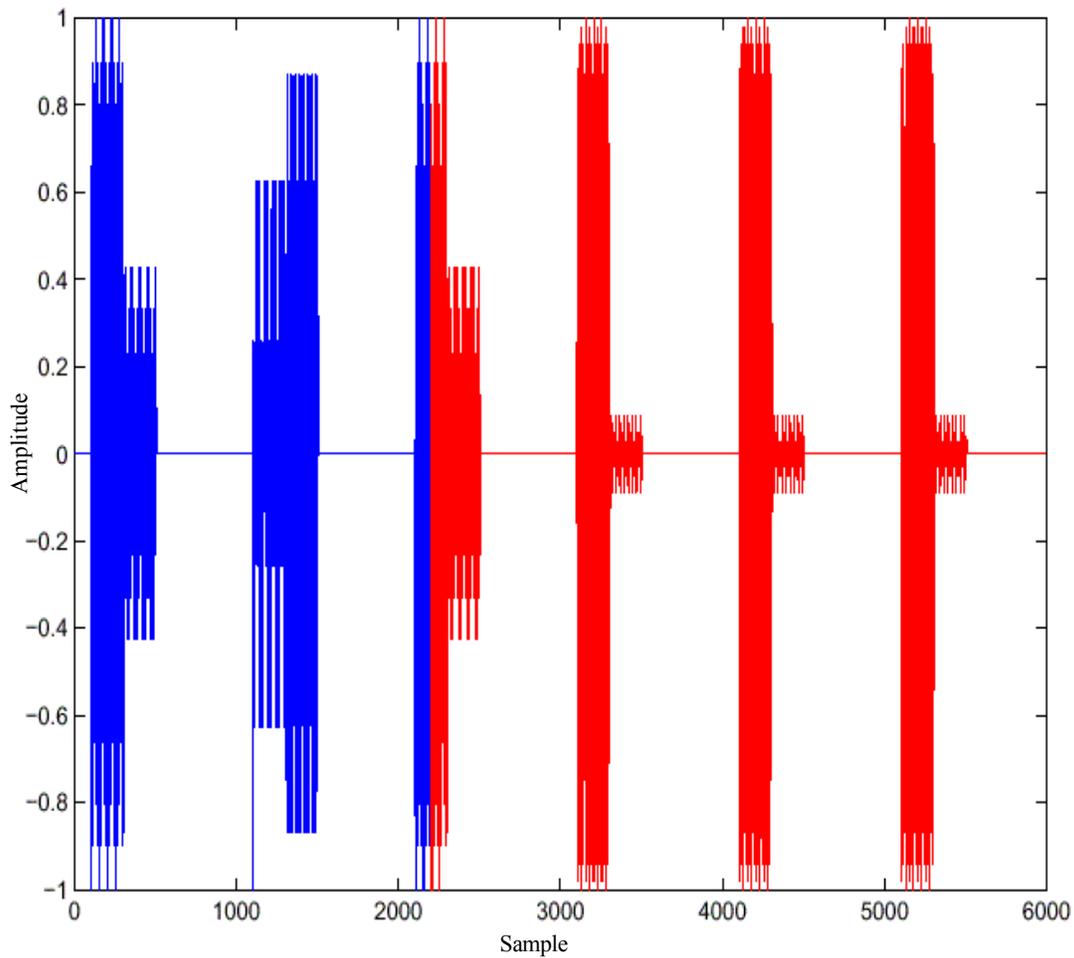


Figure 3.1.22: POP Signal Separation Results – Host Driven

With this signal separation, using the RCM/CSRC technology to “bring the hardware to the data”, downstream parameter measurements would not be corrupted thus allowing normal system performance for signal ID and classification.

3.1.5 Data Driven Demonstration

In a final data-driven system, the context switching control information for dynamic sub-band filtering would be driven from information derived from the pulse parameter measurement processing. For example, pulse identification and classification could directly determine the filtering strategy and hence the context switching strategy. In a data-driven scenario, a Finite State Machine could determine and control a context switch to a different appropriate sub-band filter/context, or similarly, from a software command sequence initiated from processing of the data stream. The use of a FSM for context switching is implemented as part of Virginia Tech’s two demonstrations (image processing/motion detection and encryption algorithm).

The data-driven demo of POP signal separation using sub-band filtering has been developed to use the VT RTR/ACS API (modified SLAAC API) environment and its support of FIFO data streaming. The context designs (VHDL) had to first be modified for proper design interface and use of VT RCMOS FIFO support and resynthesized through the CSRC tools. For this demo, the data driven initiation of the filter/context switching sequence is initiated via a POP detection algorithm processing the data stream as it passes (via FIFO) through the power PC on the RCM board. The power PC then initiates the filter and context switching sequence similar to that used for the host driven demo. Figure 3.1.23 below shows the input data stream of six pulses with POP (two intermixed signals). Similar to the results of the host-driven demonstration, Figure 3.1.24 captures the resultant output showing successful POP signal separation. Again, filter B is the wide band filter and filters A and C the narrow band filters:

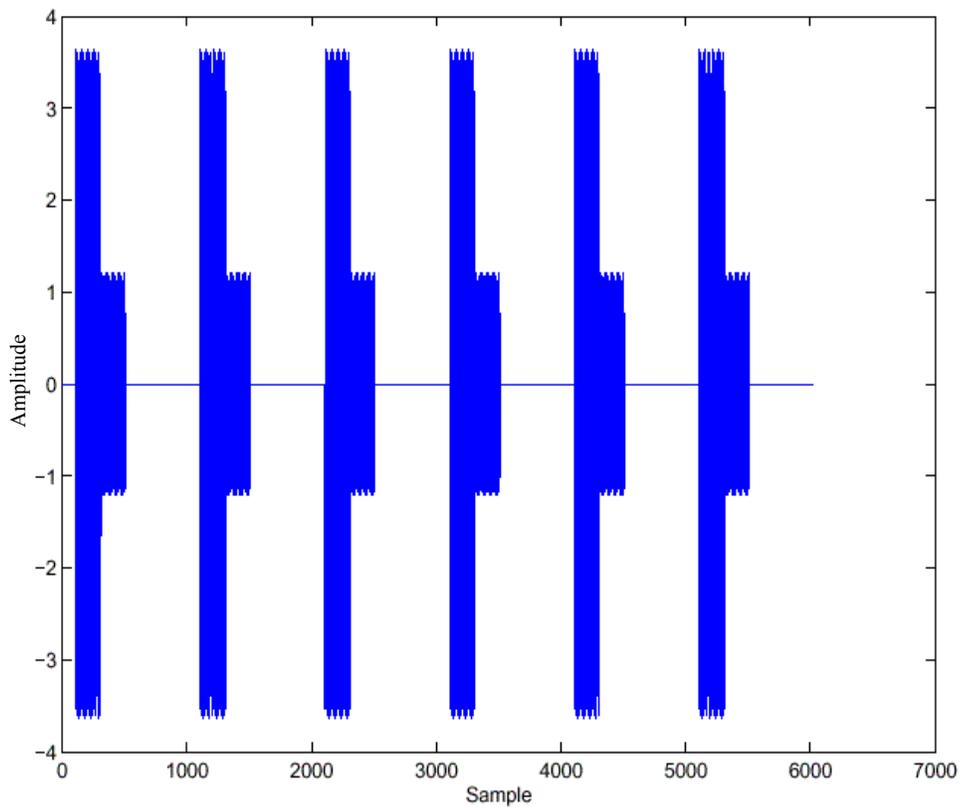


Figure 3.1.23: POP Signal Input Sequence

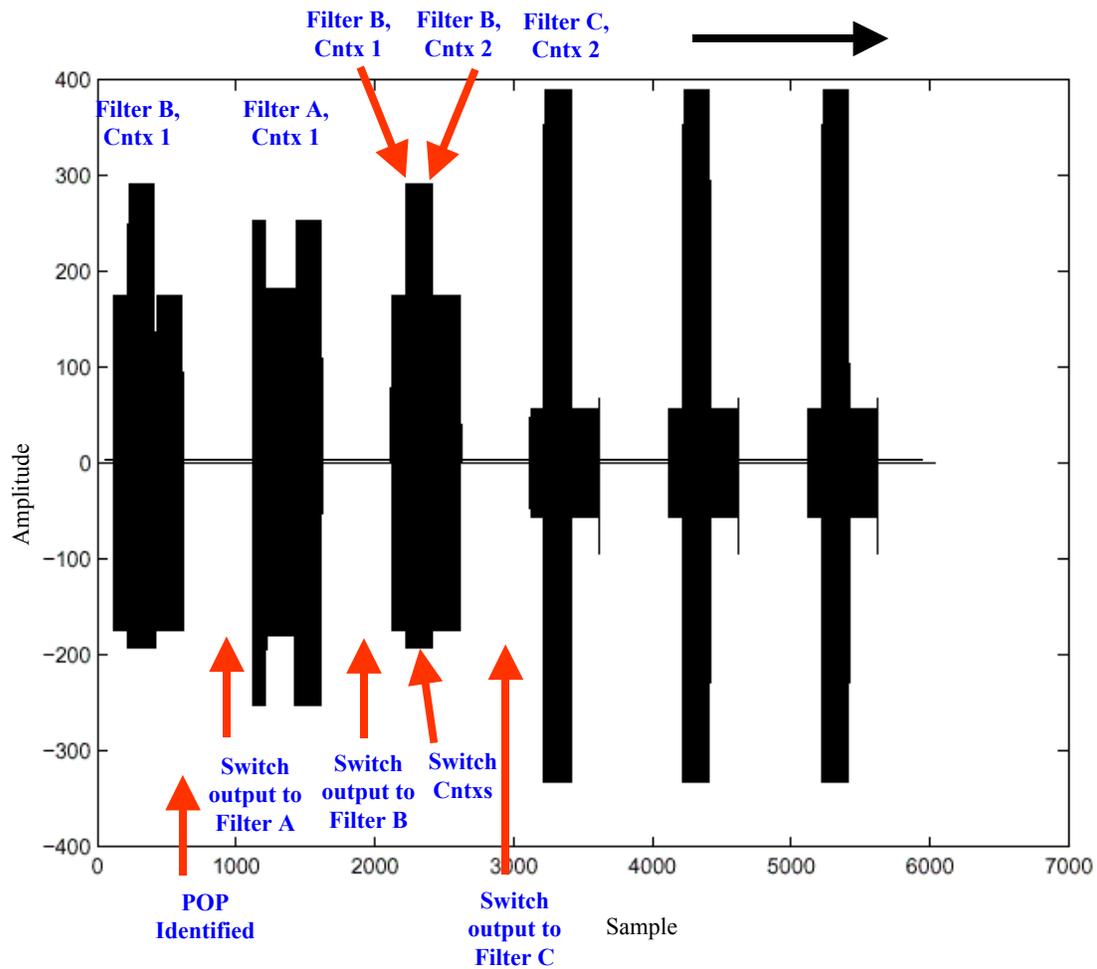


Figure 3.1.24: POP Signal Separation Results – Data Driven

Because of the processing for POP detection via C code on the power PC on the data stream, and without clock synchronization between the power PC and the FIFOs/CSRC boards, the clocking of the data through the system was essentially single stepped vs free running on the CSRC. This incurred a performance hit, but did not preclude the successful implementation of showing data-driven POP signal separation. In an improved or actual implementation, and if further time and resources allowed, this limitation could be resolved by implementing the POP detection algorithm in the CSRC context design. Context switching initiated from within the CSRC would then be used (requiring some additional modifications for control logic from within the CSRCs), at a speed determined by design limitations for using the CSRC clock.

The data-driven demonstration was capable of being run under either of two API's. The first was the Virginia Tech RTR/ACS API (modified SLAAC API). The second API was one developed by Virginia Tech as an alternative API to be used for RTR/context switching and our RCM boards. Using either API, the demonstration produced the same expected results (i.e. POP

signal separation). However, performance for speed was considerably different between the two API's. For the demonstration, the sample set used was 6,000 samples of 4-bit I and 4-bit Q data, for a total data size of 6 Kbytes. Actual on-board performance (i.e. the filtering process) was similar between the two API's as would be expected. The on-board processing of the full data set took 0.0403 seconds for the modified SLAAC API and 0.0387 seconds for the VT developed API. However, use of FIFO was relatively much slower for the SLAAC API. This was primarily associated with writing to (vs. reading from) the FIFO. Reading from FIFO using the modified SLAAC API required 0.0768 seconds while the VT developed API required 0.0681 seconds. However, writing the data set to FIFO required 5.5795 seconds for the modified SLAAC API and only 0.0604 seconds for the VT developed API. This is primarily the result of the SLAAC API not being developed for efficient use with the RCM boards. Modifying the SLAAC API for efficient operation with the RCM boards would improve the resultant performance numbers; however, this work was not performed as part of the DRACS program. Overall run time for the full data set was 9.1094 seconds using the modified SLAAC API and 2.2250 seconds using the VT developed API. These times take into account additional overhead associated with startup, shutdown and data display, each of which were similar in required processing time between the two API's.

3.2 Virginia Tech

3.2.1 General

The efforts of the Virginia Tech team under DRACS include the following:

- Design and implement the interface to allow the BAE SYSTEMS CSRC-based board (RCM) to be accessible through modifications to the SLAAC ACS API
- Extend the ACS API to support efficient host-driven and data-driven Run Time Reconfiguration (RTR)
- Extend the ACS API to provide debugging support for RTR
- Implement and evaluate applications that exploit or require RTR on the BAE SYSTEMS CSRC-based board

The two “styles” of runtime reconfiguration referred to under the DRACS efforts are Host-Driven RTR and Data-Driven RTR. Host-Driven RTR refers to when a device is reconfigured at the behest of a controlling process, e.g. a predetermined sequence of configurations or OS-driven time-sharing between jobs. Data-Driven RTR involves device reconfiguration according to the data being processed, e.g. filter selection depending on brightness.

3.2.2 API

Since the BAE SYSTEMS' CSRC provides for context switching on a clock cycle basis, orders of magnitudes faster than typical chips, effective use of the chip in an application environment requires support mechanisms for RTR that provide the user with usable abstractions and do not incur overhead expenses that overwhelm the advantages of the CSRC's fast context switching. To control the CSRC chip/RCM, the approach involves an API providing the required RTR

support. Rather than inventing another API, Virginia Tech used the ACS API from the SLAAC project. The ACS API provides for the control and operation of one or more adaptive computing boards. As an example, one of the main functions is *ACS_Configure()*, which sends a configuration to a particular board.

In order to implement the RCM API, associated node library components have been created. In addition, the API includes the necessary functions to manipulate memory, load and execute PowerPC tasks, and configure the Xilinx FPGA on the RCM. The API also provides the low-level operations necessary to interface with higher-level API's, and is sufficient for high-level development.

The ACS API extensions for Host-Driven and Data-Driven RTR are complete (including additional modifications for FIFO and debug support). Support is for a Linux based system.

3.2.3 RCM OS

The RCM Operating System developed at Virginia Tech is a shared tightly-coupled dual processor tasking environment. The RCM-OS uses a common interpretation of memory structures by the host and RCM. There is full access of RCM resources through light-weight library calls. Under the RCM-OS, the host functions include cache management, FPGA/CSRC configuration, implementation as the Host-Driven RTR mechanism for the ACS API, data streaming support, debug support, and node management.

3.2.3.1 Cache Management

In order to be efficient, a virtual hardware implementation requires fast switching between configurations. This is difficult, for example, on an Annapolis Microsystems WildForce board where the Xilinx 4K part takes 100's ms to configure and the configuration must be transferred over PCI. For efficient virtual hardware under DRACS, a cache hierarchy implementation combined with the CSRC chip is used. The intent of configuration caching is to have a common platform independent view presented to the programmer. The actual behavior is node dependent. For example, the BAE SYSTEMS' RCM board involves a mixture of on-chip context caching and dedicated HW/SW configuration caching engine on the ACS node. The WildForce board caching is managed by the host while the SLAAC-1V caching is managed by a special on-board configuration controller. Configuration caching avoids the expense of sending a configuration to the chip over PCI (and perhaps a network) every time the user wants to reconfigure. In the case of virtual hardware, we expect to find temporal locality in the configuration sequence that can be exploited in exactly the same fashion as in memory hierarchy. The Janus project at Virginia Tech found that most applications exhibit temporal locality.

Cache hierarchy can be categorized into four levels, with configuration data residing:

- Level 4: somewhere in the (embedded) network
- Level 3: in the main memory of the computer containing the board (saves cost of sending over the network)
- Level 2: in the PowerPC's memory on the RCM board (saves transfer cost on the PCI bus)
- Level 1: in the four contexts on each CSRC (saves the cost of downloading a configuration)

The goal is to keep the configurations most likely to be used next in the level one cache (CSRC contexts). Virginia Tech has implemented cache management using a latest used policy (note that cache prefetching is available for exploitation by higher level tools). The programmer can exercise the level of control desired over caching. It can be completely implicit, with the programmer just using *ACS_Configure()* as usual. The user can also exercise control by telling *ACS_Configure()* details such as which cache slot in which cache to configure.

3.2.3.2 Data-Driven RTR

Even more than virtual hardware, data-driven RTR can exploit the CSRC's fast context switching. Data-driven RTR provides for the hardware configuration to be selected according to the data encountered. Being able to quickly switch hardware configurations is required to meet real-time processing constraints in certain applications. Data-driven RTR can be described as one of three modes, depending on what is used to initiate the context switching. The first mode, Mode 1, is when the device computes and performs the context switching. This mode involves the lowest latency switching. This is represented by Figure 3.2.1 below.

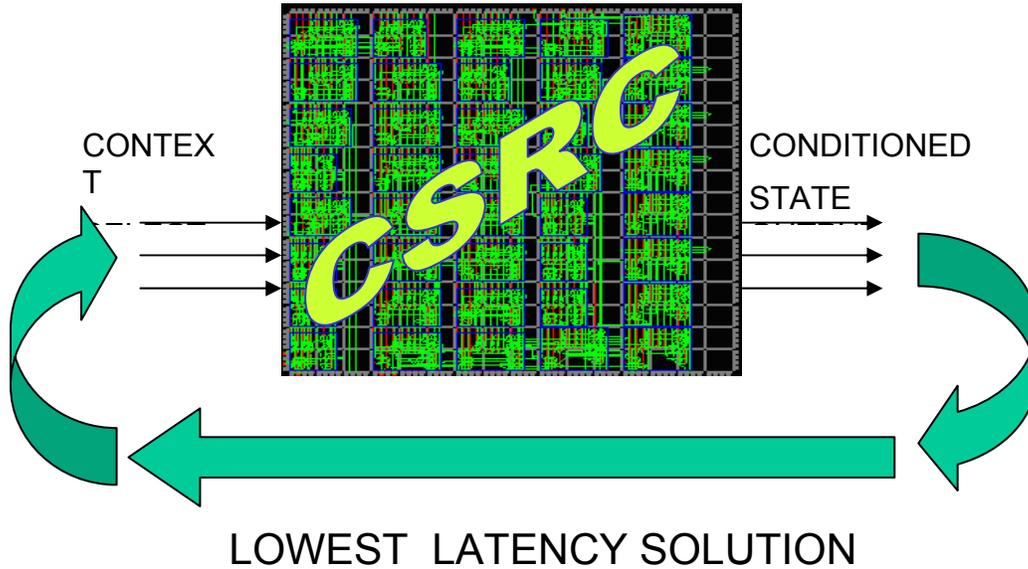


Figure 3.2.1: Mode 1 – Device Computes and Performs Context Switch

The second mode is when the state is analyzed and processed by the RCM board Xilinx FPGA external to the CSRC devices. This mode provides additional latency but also increased flexibility. This mode is represented below in Figure 3.2.2.

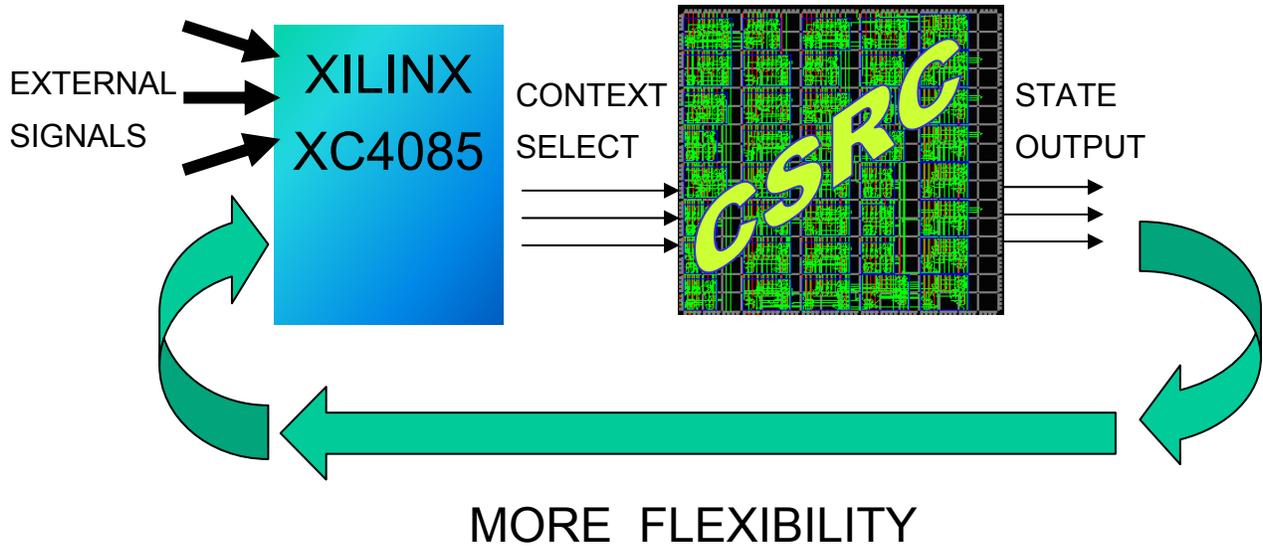


Figure 3.2.2: Mode 2 – State Analyzed and Processed by External FPGA

The third mode is when the state is analyzed and processed by an external CPU. This mode provides the highest latency but also the most flexibility. This mode is represented below in Figure 3.2.3.

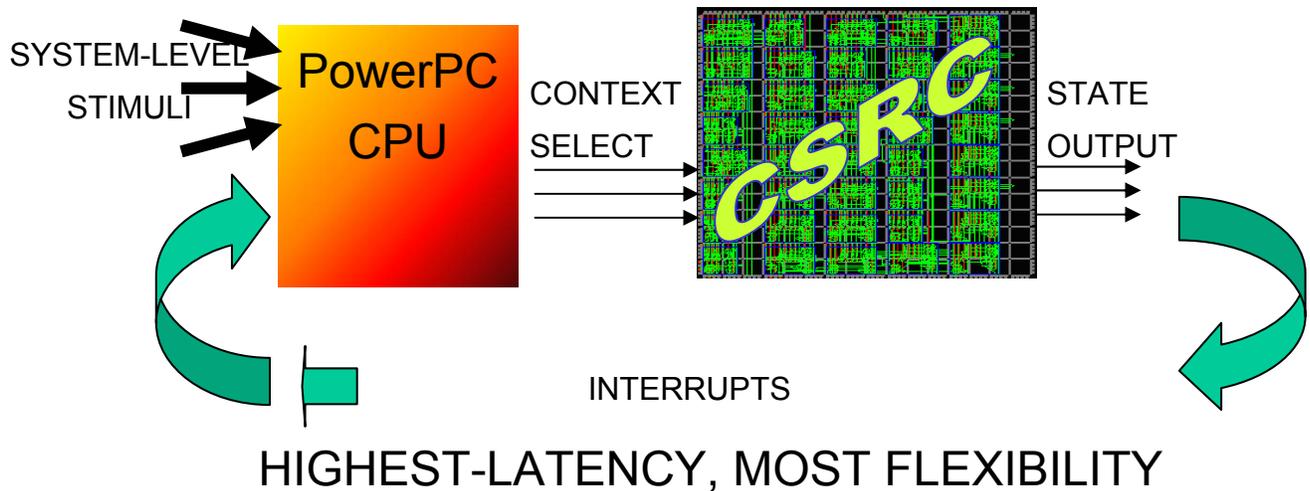


Figure 3.2.3: Mode 3 – State Analyzed and Processed by External CPU

3.2.3.2.1 Implementing the Finite State Machine (FSM)

Functionality has been added to the API to allow the user to define an FSM. On the RCM board, Virginia Tech has implemented a context manager FSM on the XC4085 Xilinx chip. The FSM takes inputs from the CSRC chip that are condition codes to indicate that data is encountered.

The FSM is then to drive context switching on the CSRC and interrupt the PPC to cause it to send a new configuration from its cache to the CSRC.

Figure 3.2.4 below represents RTR driven by states:

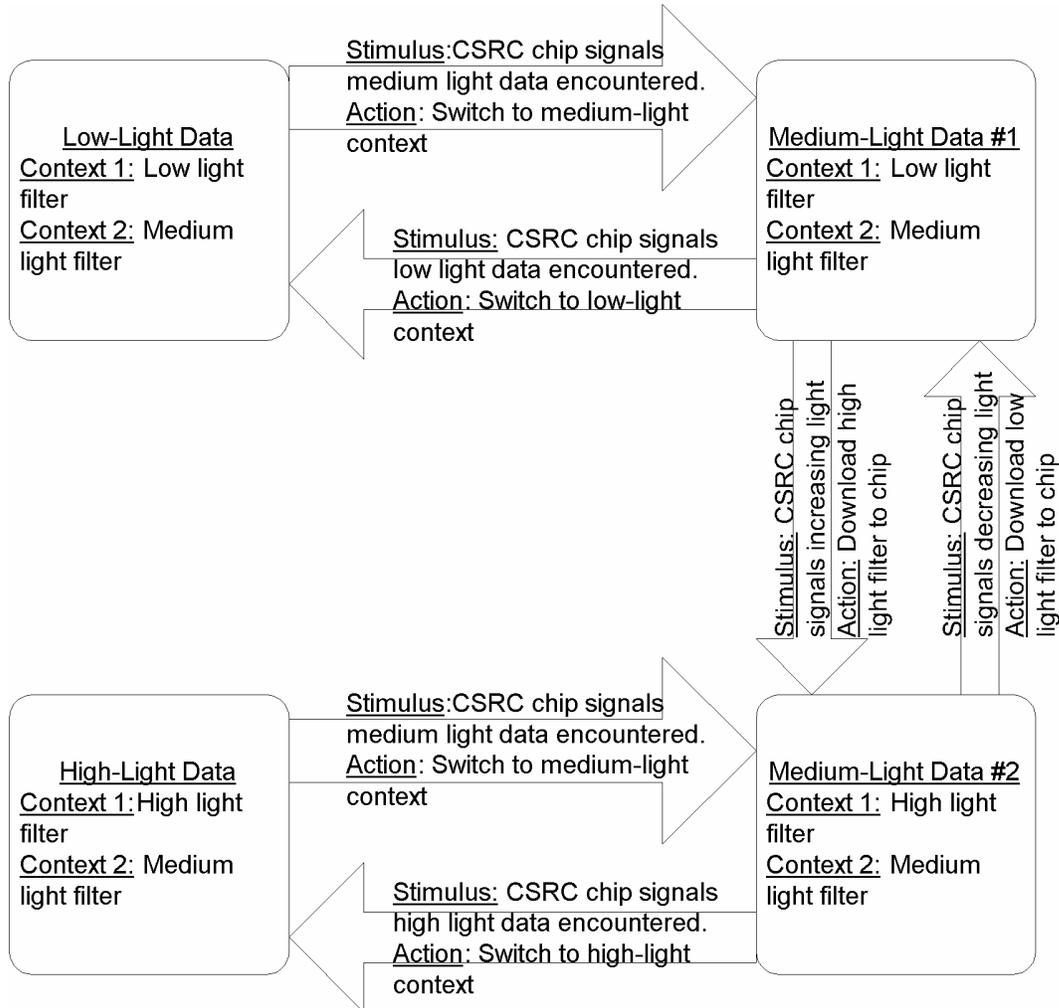


Figure 3.2.4: State-Driven RTR

Using Jbits technology developed at VT, the FSM can be reconfigured quickly. This allows the user to change a FSM without having to resynthesize the design. This is suitable for applications where a FSM controls reconfigurable hardware. The configuration tool is not difficult to use. As an example, a traditional run time FSM generation would involve a process similar to that shown in Figure 3.2.5.

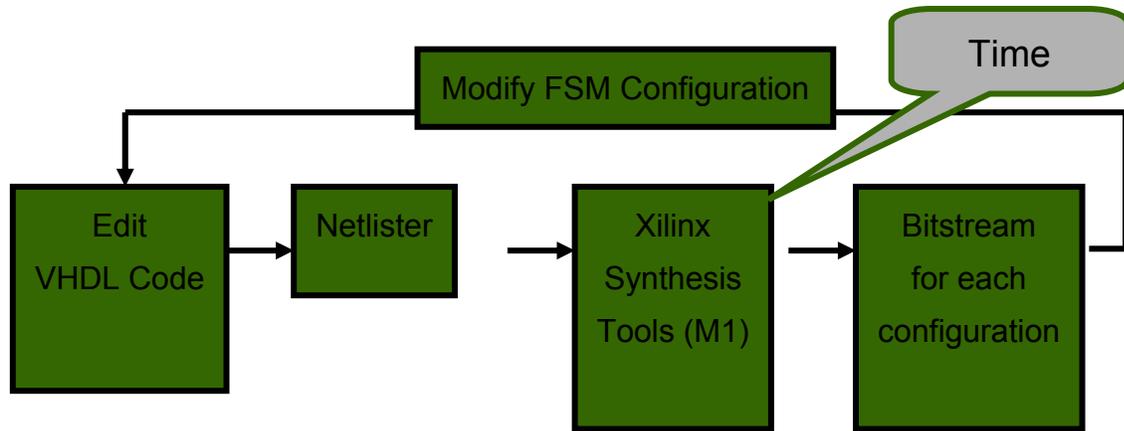


Figure 3.2.5: Traditional Approach to FSM Creation

However, through the use of Jbits technology, dynamic FSM generation can be realized as represented in Figure 3.2.6.

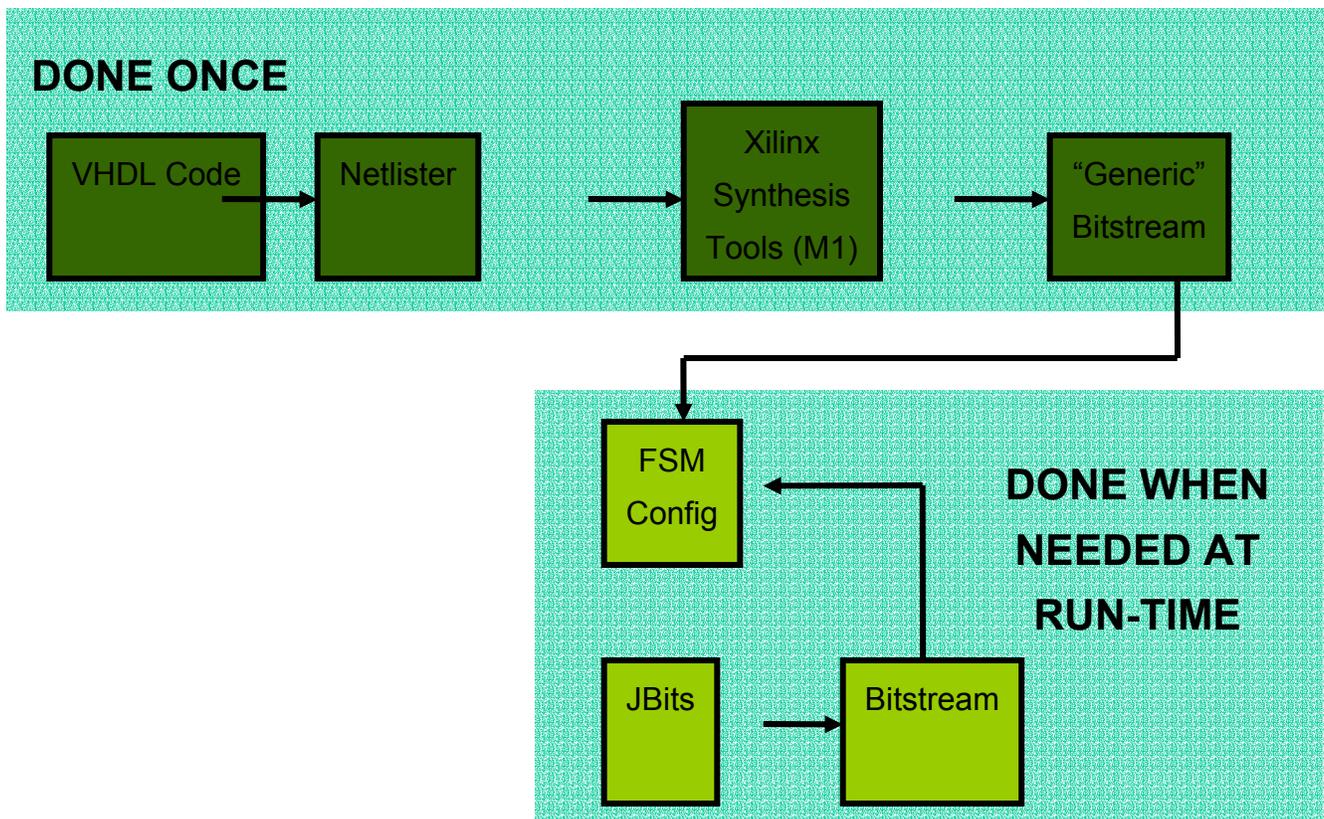


Figure 3.2.6: Dynamic FSM Generation

3.2.4 Janus/JHDL Approach to CSRC Data/Control Driven RTR

Virginia Tech has been developing an approach to CSRC data driven and control driven RTR that uses a layered software environment on top of JHDL, called Janus (Figure 3.2.7).

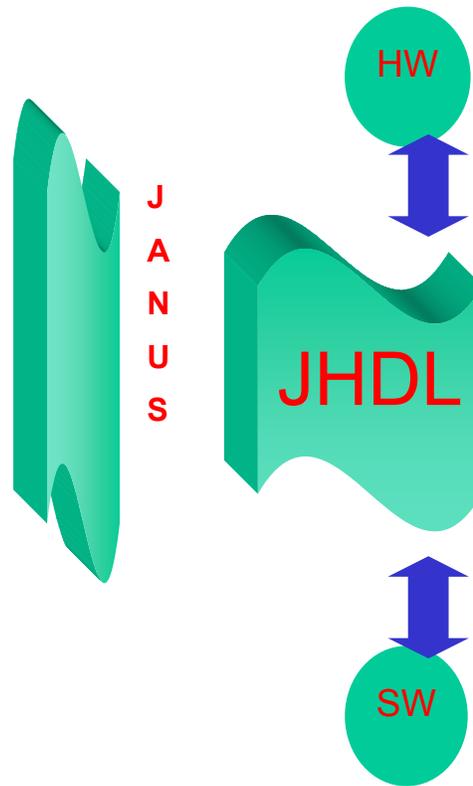


Figure 3.2.7: Janus Environment Over JHDL

Implicit with control-driven RTR, partitions are deduced at compile/run time and the scheduling is static (compile-time). Implicit with data-driven RTR, the scheduling is dynamic (run-time). The Janus environment retains all the advantages of JHDL, providing HW and SW with the same view.

The following Figure 3.2.8 provides a representation of the Janus hardware abstraction.

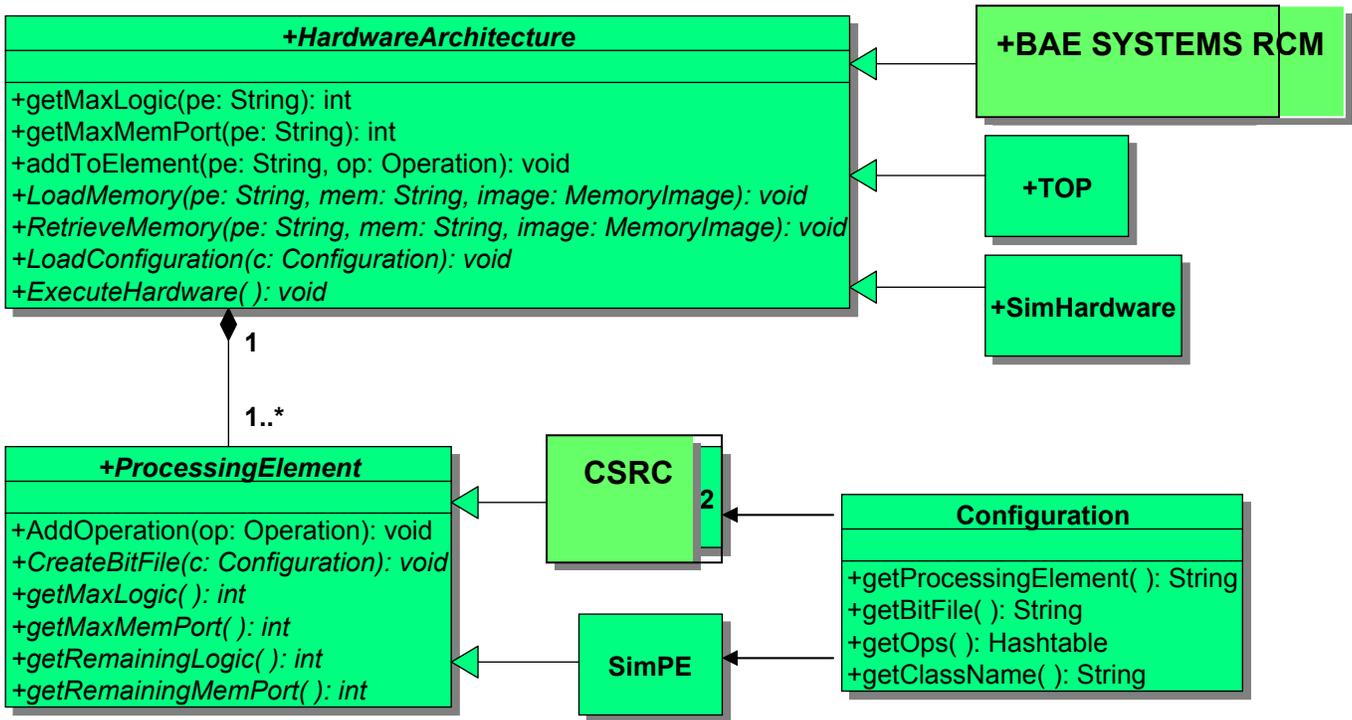


Figure 3.2.8: Janus Hardware Abstraction

The following two figures, Figure 3.2.9 and 3.2.10, provide a representation of a scheduling scheme for a classic Janus application and representation of a classic Janus execution respectively. As can be seen, the Janus scheduler properly schedules contexts for their order in hardware execution at run time.

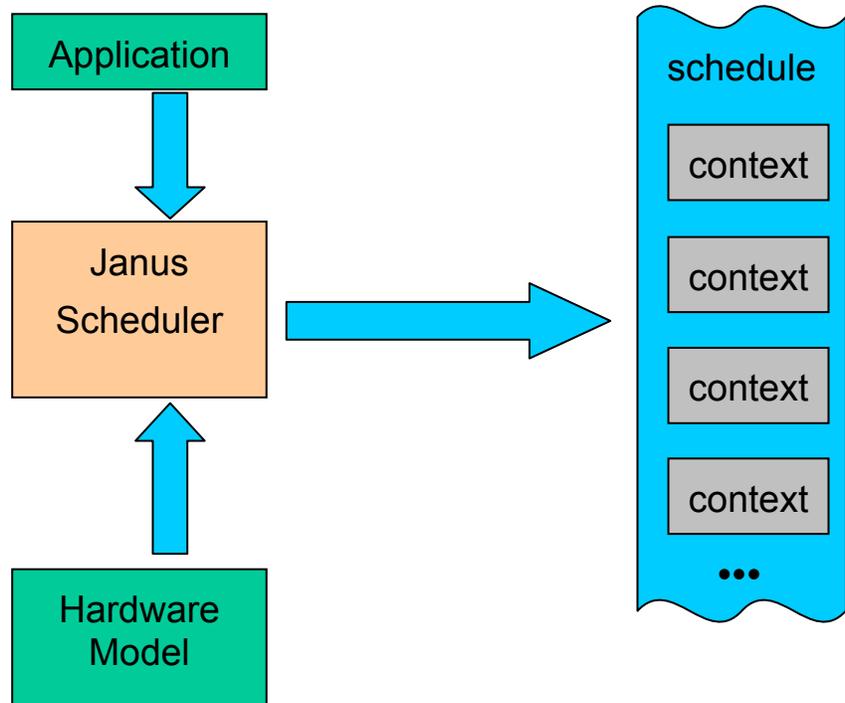


Figure 3.2.9: Classic Janus Task Scheduling

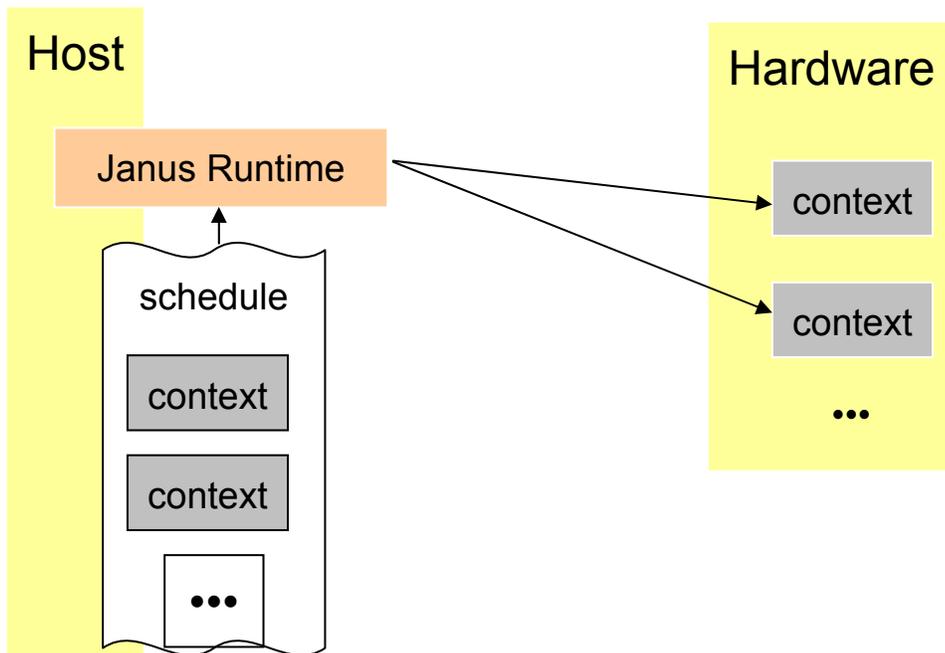


Figure 3.2.10: Classic Janus Execution

Incorporation of a state machine to the Janus environment can then be viewed as shown in Figure 3.2.11. In this case, the classic Janus execution scheme of using the schedule established at compile time to initiate the switching of contexts in the predetermined sequence at the

established points during execution from the host environment (Janus Runtime environment of Figure 3.2.10) is no longer used. Instead, the execution scheme of Figure 3.2.10 is replaced by Figure 3.2.11 which is simply the classic Janus task scheduling process of Figure 3.2.9 modified to incorporate a State Machine for determining when a context switch is to occur. The predetermined schedule with the switching sequence or order of contexts is still used; however, the conditions for when the switch to the next context will occur is now controlled in a data-driven fashion, each switch occurring once the appropriate state(s) or conditions exist for that next switch to occur.

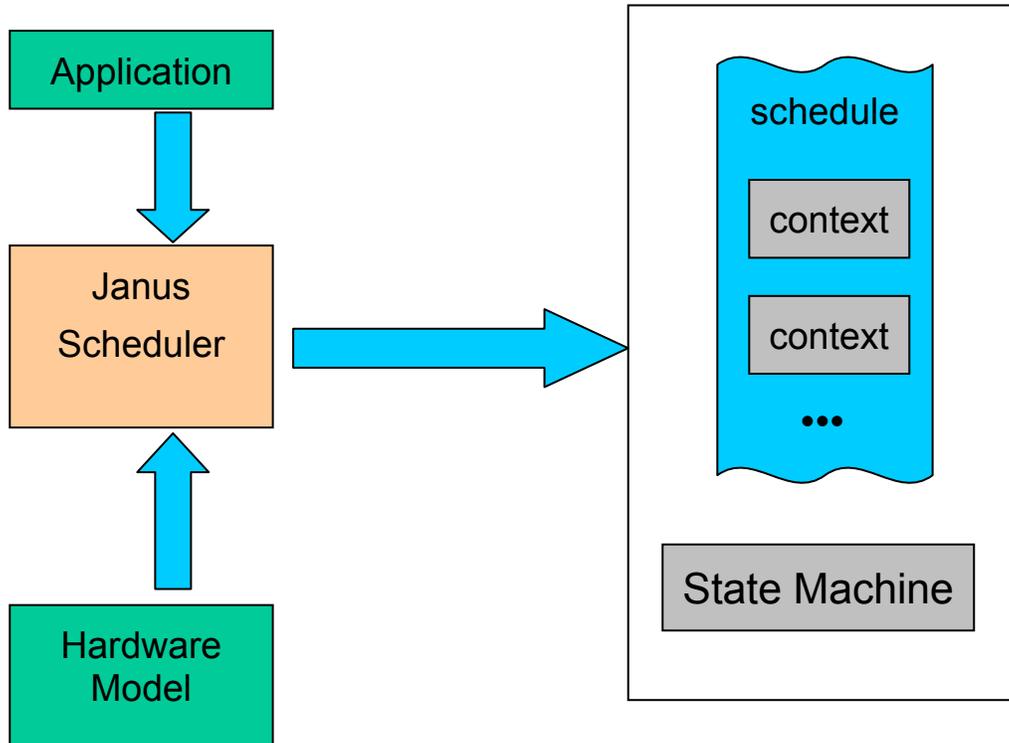


Figure 3.2.11: Janus Task Scheduling

3.2.5 Technical Discussion

The VT DRACS team has successfully completed their support of the project. The team has completed the implementation of FIFO and debug support, its data driven CSRC demonstration Enigma Encryption application, and its finite state machine (FSM) driven CSRC demo application, Motion Detection. The team also completed its efforts in implementation of the BAE SYSTEMS POP signal separation filter demo. Also, a paper publication effort has been completed.

As planned, the Enigma Encryption implementation has three encryption engines. The context in CSRC_A chip will receive a data stream that has a key (address) in the packet header, and will determine the right encryption engine, which is in one of the contexts of the CSRC_B chip. Then, the Xilinx chip generates a control signal to select the corresponding context in CSRC_B for encryption or decryption. Since the Enigma Encryption algorithm is symmetric the same implementation can be used for encryption operation as well as decryption operation. When a

normal data stream feeds data to Enigma, it will generate an encrypted data stream. When the encrypted data stream is fed in the same Enigma using the same key used during encryption, the stream will be decrypted into the original data stream. For the Enigma demo, Virginia Tech constructed a GUI to attach a key (address) in the header of a text stream and to display the encryption results or the decryption results. See Figs. 3.2.12 through 3.2.14 below for a graphical representation of the Enigma processing routine and the associated GUI .

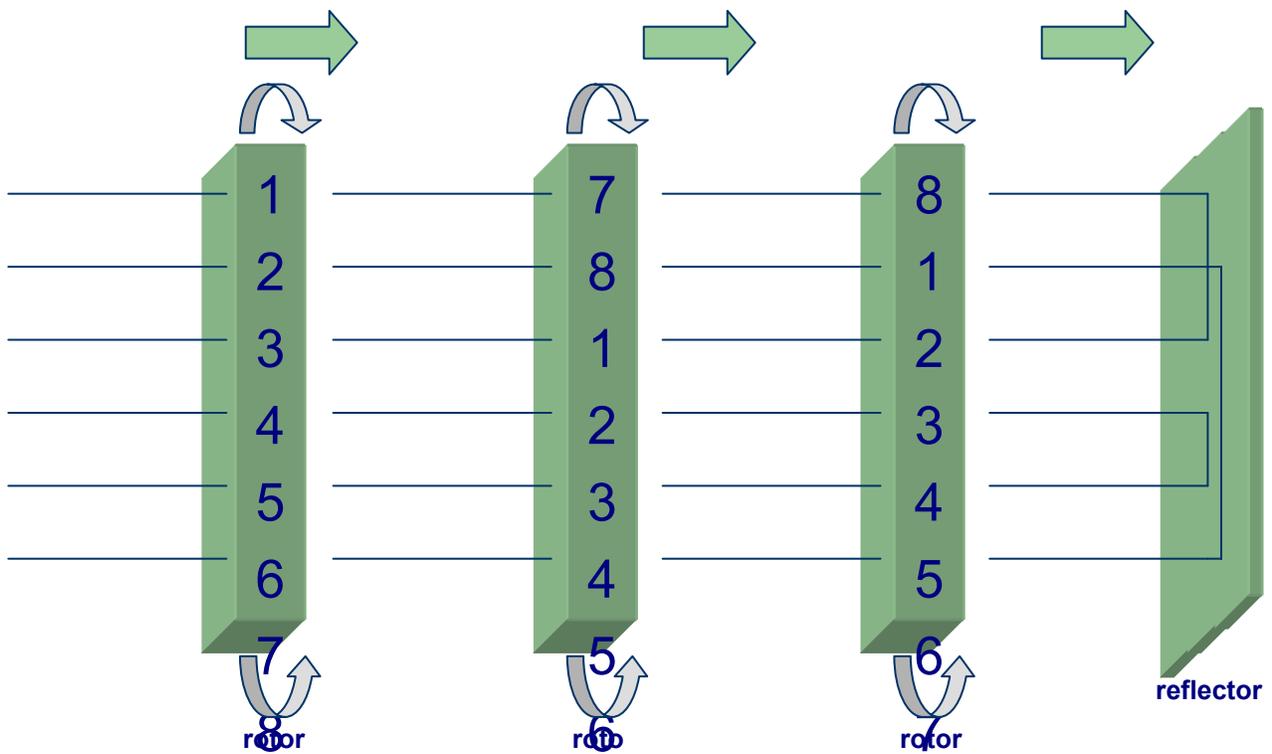


Figure 3.2.12: Enigma Processing

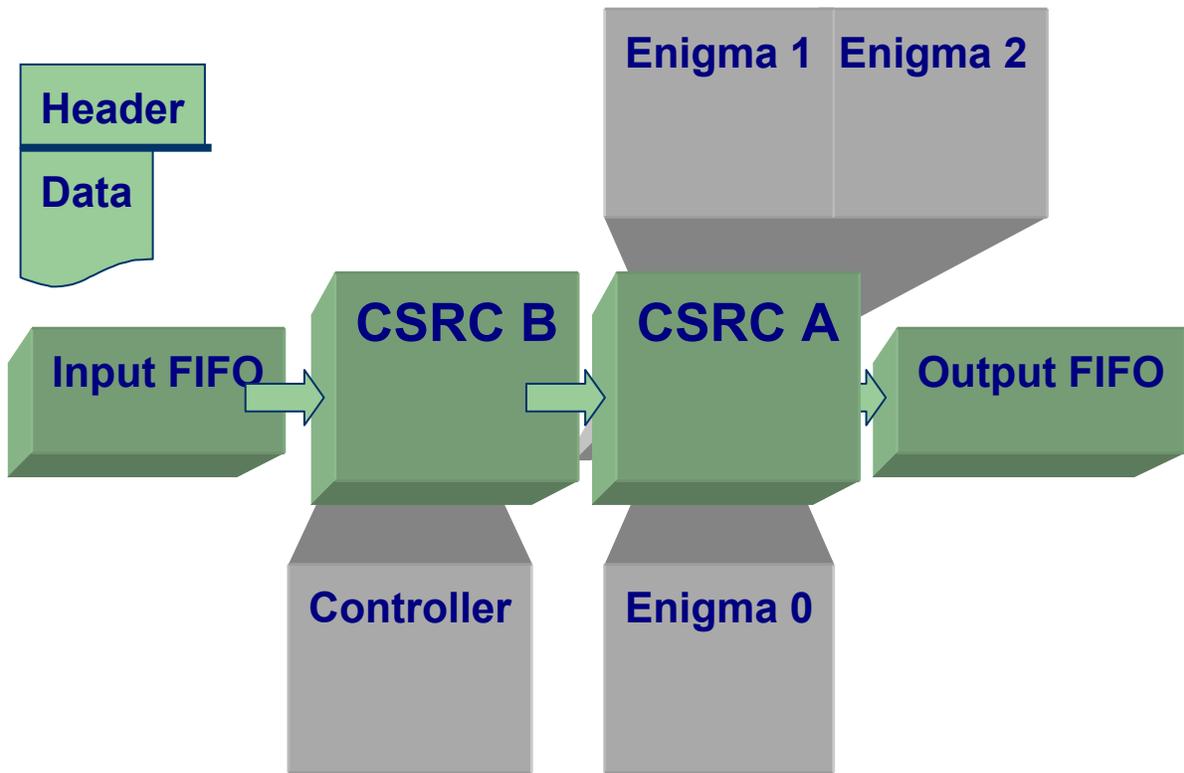


Figure 3.2.13: Enigma Implementation

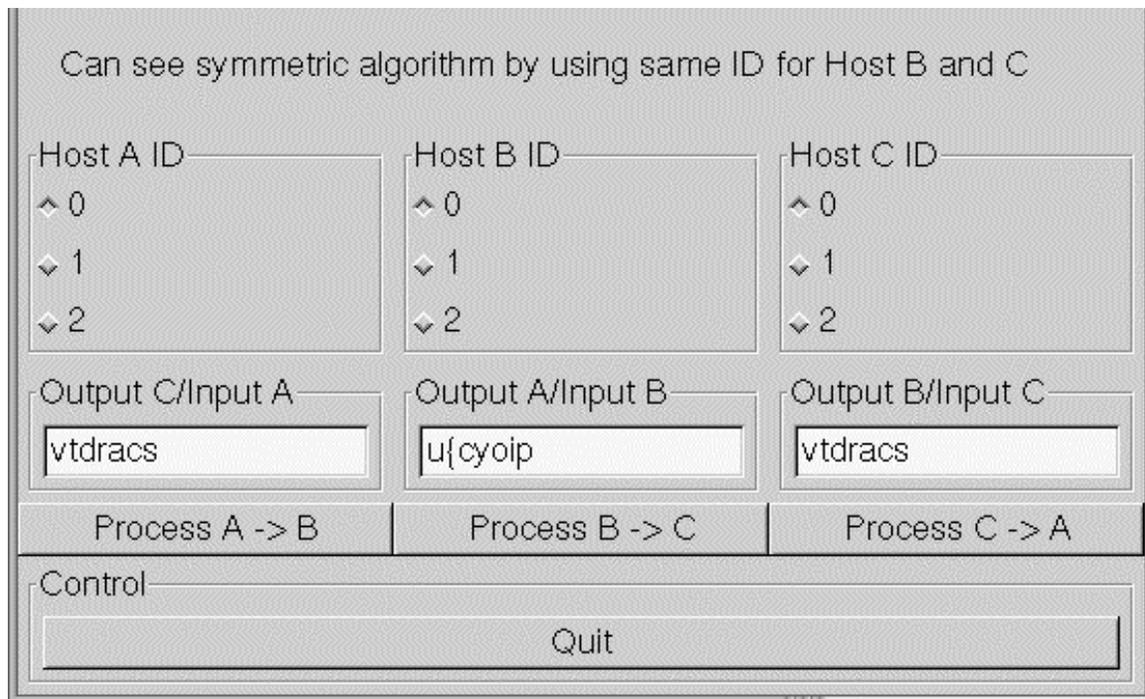


Figure 3.2.14: Enigma GUI

FIFO Operation

This section briefly describes the logic in controlling the FIFOs for streaming operations. On the RCM board, CSRCA has to be configured to drive the FIFO_RE_N signal for the input-FIFO, one clock edge before the data is required on the FIFO_2_CSRCA bus. CSRCA(0) line (F31 on CSRCA) is used as the FIFO_RE_N signal in the VT API. For the output-FIFO, the FIFO_WE_N has to be driven by the CSRCB on the line F31 of CSRCB. On the clock edge where the data has to be written on the output-FIFO, the FIFO_WE_N signal should be asserted. Apart from this, the FIFO_2_CSRCA lines and CSRCB_2_FIFO lines have to be mapped on to the respective pins on the CSRCs. Appendix A includes examples of sample code:

Finite State Machine

For data driven applications, a FSM is well suited to manage the transition from one context to another based upon events within the CSRCs. Due to the prototype nature of the CSRC devices, these chips have a limited amount of logic and routing resources. Complex control logic can potentially consume large amounts of valuable resources. The Xilinx XC4085 FPGA is perfectly positioned in the control pathways to address much of this control logic. The FPGA also has access to signals from both CSRCs as well as the both hardware FIFO units.

FSM Implementation

The context switching FSM is implemented in the Xilinx FPGA within a dual-ported RAM core. A behavioral VHDL representation of the RAM is used for modeling and synthesis. It is written so that the synthesis tools can instantiate the proper RAM core for the target hardware.

The FSM module is constructed to have has inputs for the current state, the current input, various programming signals, forced input signals, and control signals. It has two outputs: the current state and the corresponding state output variables. Internally, the current state and input are used to create an address into the RAM core. The value at that RAM location is both the next state and the output of the FSM for that next state. On each clock the FSM registers this memory value and effectively outputs the current state and output value associated with that state.

Programming the RAM is done with the Xilinx E clock emanating from the PowerPC, which is not controllable by software. The FSM itself runs off the CSRC clock. The current size of the hardware FSM is fixed at VHDL compile time. The various sizes can be found at runtime by reading the following values:

FSM_STATE_SIZE number of bits for state representation

FSM_INPUT_SIZE number of bits for input representation

FSM_OUTPUT_SIZE number of bits for state output representation

The number of address bits needed is:

$ADDRESS_BITS = FSM_STATE_SIZE + FSM_INPUT_SIZE$

The number of data bits needed is:

$$\text{DATA_BITS} = \text{FSM_STATE_SIZE} + \text{FSM_OUTPUT_SIZE}$$

The size of RAM used is:

$$\text{RAM_BITS} = 2^{\text{ADDRESS_BITS}} \times \text{DATA_BITS}$$

For each combination of state and input that can occur in the application the associated memory location must be set. The location can be found by concatenating state address and input bit vectors together. The value at that location is found by concatenating the next state address and output bit vectors together.

Note that it is not required to set all locations. If, for instance, only a few states and inputs are required only RAM locations related to them need to be set. (Please see VHDL source for exact implementation details.)

Programming

The first step in the creation of a context-switching FSM controller is creating the state transition table for it. Programming usually starts by disabling the active FSM through control registers in the Xilinx control FPGA. Next the memory is loaded. Then a pseudo next state and input are forced in order to force the initial state. Next a clock occurs to load the initial state and output. Finally all signals are re-enabled for normal operation.

It is wise to stop the FSM during programming to keep the system from changing into indeterminate states. This is not strictly required, however. Disabling is done by setting bit 1 of the FSM_CTRL register to a 1, and the FSM_STATE_IN to the desired “safe” state vector.

There are two modes available to accomplish FSM programming. The first is automatic memory writes that occur whenever the FSM_LD_DATA register is written to. The second is manual writes done by toggling FSM_LD_IN. The latter mode is chosen by setting FSM_CTRL(3) to 0 and 1 respectively.

The procedure is as follows:

1. Disable FSM: set FSM_CTRL(0) to 1
2. For each state and input combination used by the application program the FSM:
 - (a) set the memory location at FSM_LD_ADDR
 - (b) set the memory value at FSM_LD_DATA
 - (c) if FSM_CTRL(3) is 1 then toggle FSM_LD_IN
3. Initialize:
 - (a) use forced input: set FSM_CTRL(2) to 1 and FSM_INPUT_IN
 - (b) use forced state: set FSM_CTRL(1) to 1 and FSM_STATE_IN
 - (c) clock the FSM via the CSRC clock
4. Enable FSM: set FSM_CTRL(0) to 0

The FSM bits are available to various other configurable sections of the control logic. They can be routed to the CSRC inputs as well as to the input and output FIFO control signals. The FSM inputs can come from various locations. They are programmed in a similar address/data fashion as the FSM memory. For each bit of the input vector set FSM_INPUT_CTRL_ADDR to the bits address then set FSM_INPUT_CTRL_DATA to select that bits source. See the VHDL source for an illustrative example on how to create the mapping of the data field to the source locations.

FSM Demonstration

Virginia tech successfully demonstrated the use of the FSM for context switching control with an image processing demonstration as shown below.

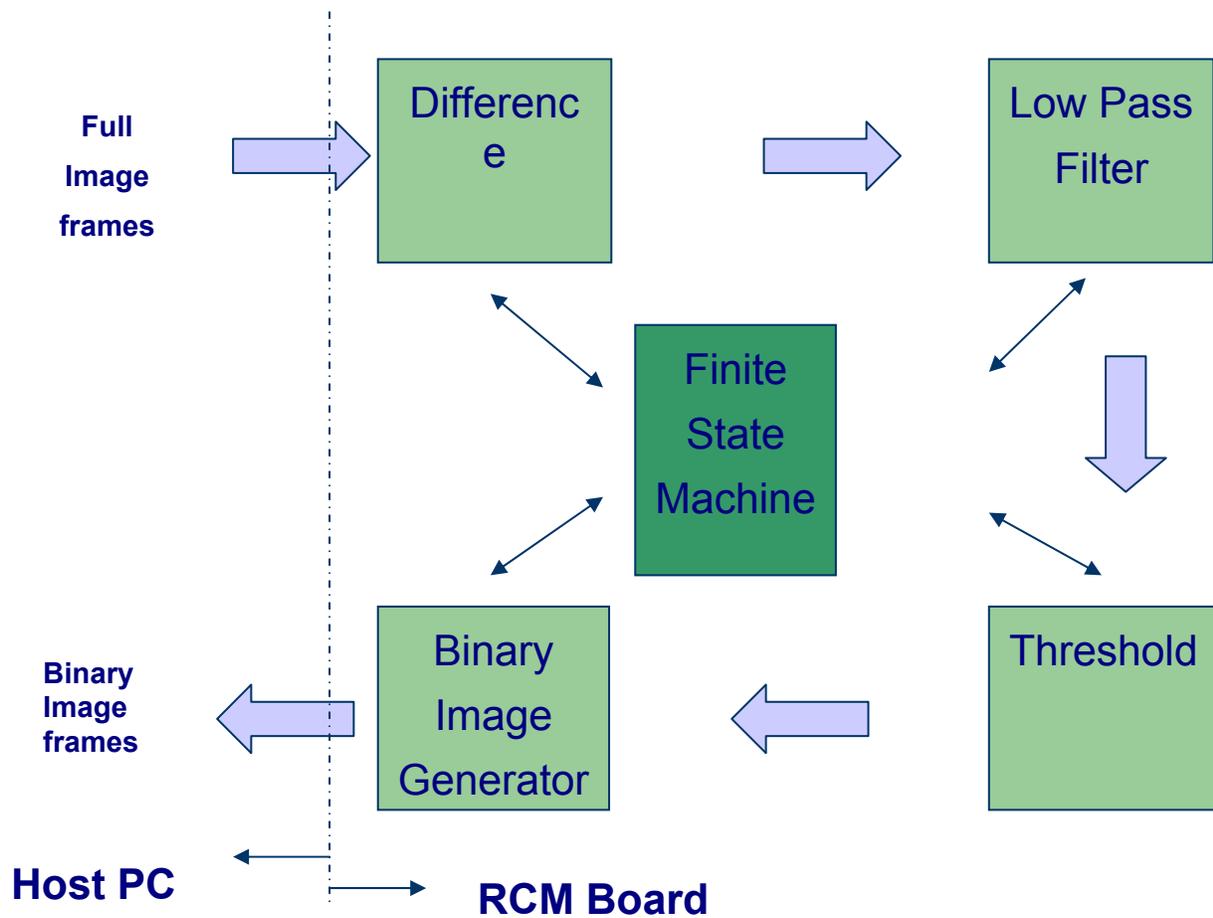


Figure 3.2.15: Motion Detection Algorithm

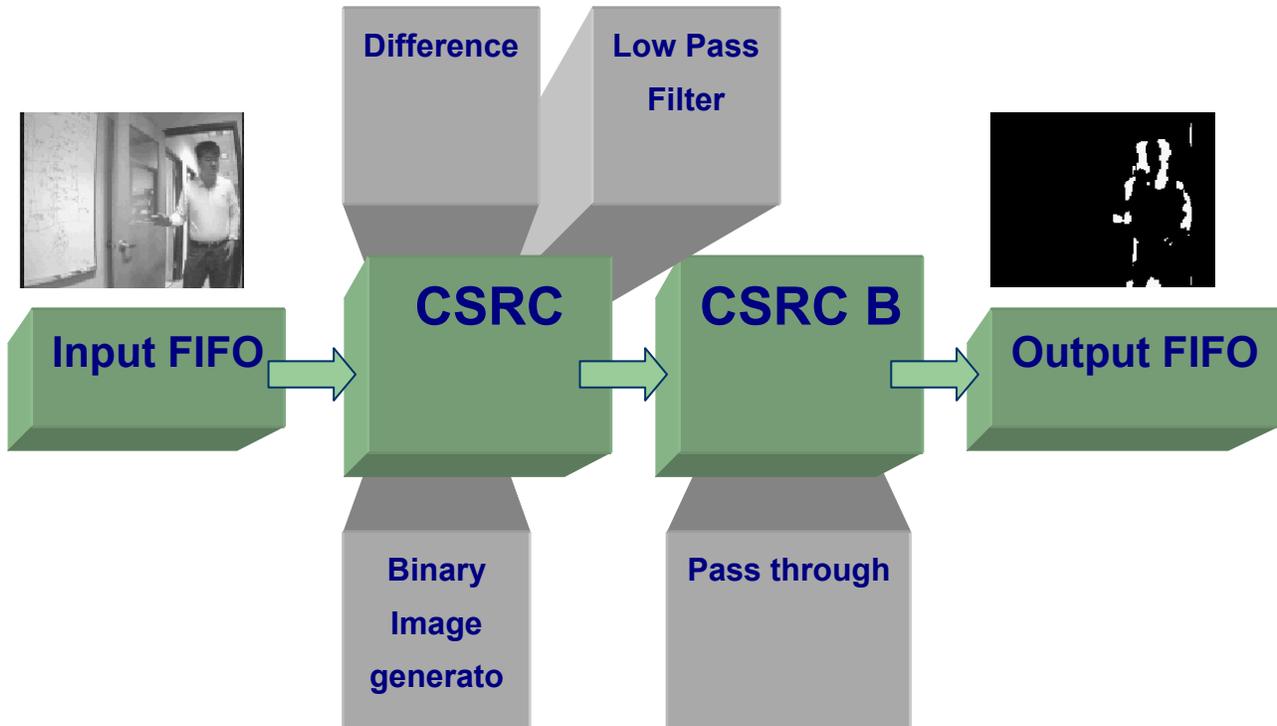


Figure 3.2.16: Motion Detection Implementation

The motion detection algorithm implementation demonstrates the control of CSRC contexts through the host programmable FSM. The motion detection algorithm basically consists of capturing an image (the demonstration used a live video feed), processing it and sending out a cropped image where there is motion. Such an application is useful for power critical remote sensing motion detection. The above figure shows the mapping of the algorithm to the CSRC contexts. The image is captured by the Host machine and passed to the RCM board through FIFOs. The four parts of the algorithm are implemented as four different contexts. The binary image generated by the CSRC is used to generate the cropped image by the host machine.

User Controlled Context Switching

A variation on the motion detection application was also developed as a host-driven application. A continuous video stream is processed by filters stored in contexts. A user request from the host causes the context to switch. This enables single-cycle algorithm changes. If more filters are needed than available contexts, then some cache manipulation may need to take place. The loading of new filters into contexts that are currently inactive takes place while another filter is running. This allows unlimited “virtual hardware” filters. Simple filters have been implemented on the CSRCs that include the basic passthrough, the motion detection difference filter and a decay filter.

Area Evaluation

A very strong argument in favor of multi-context devices is the reusability of area for emulating infinite hardware. To assess this claim, Virginia Tech conducted a study of the area requirement for the applications developed. See the below table:

<i>Application</i>	<i>4-context implementation</i>		<i>1-context implementation</i>
Motion	Difference	783.0	2055.0
Detection	Filter	1188.8	
Algorithm	Bin. image gen.	253.3	
Enigma Encryption	Enigma0	1331.8	5245.0
	Enigma1	1331.8	
	Enigma2	1331.8	
	Enigma3	1331.8	

Table 3.2.1 : Area Study Results

Table 3.2.1 shows the gate equivalent areas obtained for Synplify for the applications implemented in a single context and in the four-context device. The table also shows the area counts in each context for the application in the four-context implementation. A device with an equivalent gate count of the highest number will be required for that particular application. Motion detection application can be implemented in a three-context device with a minimum of 1188.8 equivalent gates without incurring any delay in reconfiguration. But with only a single context, the application requires a device with 2055.0 equivalent gates. For the Enigma encryption we see that in a four-context implementation we require at least 1331.8 equivalent gates for each of the enigma engines. In a single-context implementation of all four enigma engines bound together the application requires at least a 5245.0 equivalent gate device. It can be inferred that for applications which can be partitioned equally among all the available contexts, resource requirement will scale well with the number of contexts on the device.

3.3 Brigham Young University

3.3.1 General

As stated earlier, the team at BYU is not under contract with DRACS; however, they are in direct support of efforts associated with DRACS. BAE SYSTEMS has had the responsibility to collaborate and coordinate the integration of BYU's JHDL and BAE System's CSRC technology. The efforts of the BYU team associated with DRACS included the following:

- Develop JHDL-based run-time reconfiguration programming approaches
- Create a library of JHDL primitives for CSRC
- Develop a JHDL model of the CSRC board

- Benchmark the CSRC system and JHDL models with circuit examples

3.3.2 JHDL Design Tool

JHDL is a structural design tool, designed specifically to support RTR, that allows designers to express circuit organizations that dynamically change over time in a natural way, using only standard programming abstractions found in object-oriented languages. BYU extended the JHDL design paradigm from partially reconfigurable devices to context-switching devices, and has coordinated its design approach to be compatible with the DRACS extensions to the SLACC API, and has implemented support for both the CSRC device and the CSRC-based Reconfigurable Computing Module (RCM).

JHDL has advantages as a design tool. First of all, it meets the DRACS objectives of supporting high level system design of dynamically reconfigurable systems. JHDL manages FPGA resources in a manner that is similar to the way object-oriented languages manage memory: circuits are treated as distinct objects and a circuit is configured onto a configurable computing machine (CCM) by invoking its constructor, effectively “constructing” an instance of the circuit onto the reconfigurable platform just as object instances are allocated to memory with conventional object-oriented languages. This approach of using object constructors/destructors to control the circuit lifetime on a CCM is a powerful technique that naturally leads to a dual simulation/execution environment where a designer can seamlessly switch between either software simulation or hardware execution on a CCM within a single application description. Moreover, JHDL supports dual HW/SW execution: parts of the application described using JHDL circuit constructs can be executed on the CCM while the remainder of the application, the GUI for example, can run on the CCM host. JHDL is based on JAVA and therefore requires no language extensions and can be used with any standard Java 1.1 distribution. In addition, JHDL includes an integrated functional simulator with an integrated browser that is intuitively easy to use. Once the JHDL program is ready to be tested, it is loaded into the browser for simulation. The browser allows the designer to hierarchically navigate the circuit, examine signals, invoke methods, etc., during debugging.

JHDL uniquely attacks design-time and run-time issues related to DRACS efforts. For design-time issues, JHDL provides design and simulation support for context switching as well as module generation. Since contexts are an integral part of the design for an RCM/CSRC application, circuit elements can be grouped by context and simulation automatically switches contexts as a natural part of simulation. VHDL does not support this – all possible contexts would have to be elaborated and muxed together which is not practical, especially with dynamic reconfiguration. The capability to generate modules provides for efficient, high performance circuitry; faster place and route times; and a natural way to access unique CSRC capabilities (e.g. dedicated shift routing, data sharing, etc.) which would be difficult if not impossible to infer via VHDL synthesis.

Run-time issues addressed are board/device control and automatic context switching. For run-time issues (control), the JHDL environment fosters confidence that if simulation works, hardware will work as well. Features provided include:

- A context switch in hardware will be the same as shown in simulation
- Configurations are loaded directly to CSRC

- Automatic state extraction for debug (work in progress for providing automatic insertion of debug circuitry)
- Integration with the SLAAC runtime API, providing access to context caching algorithms, HW load/configuration, etc.

There are five major components to the JHDL environment at BYU related to the DRACS efforts. These include:

- JHDL library of CSRC primitives
- TechMapper to map to these primitives
- VHDL Netlister
- JHDL modifications for CSRC specific features
- RCM Board Model

The following series of figures show the tool path used for CSRC/RCM designs in the JHDL environment. Figure 3.3.1 describes the path from the JHDL description to the TechMapper, to the CSRC primitives and to the JHDL circuit level simulation:

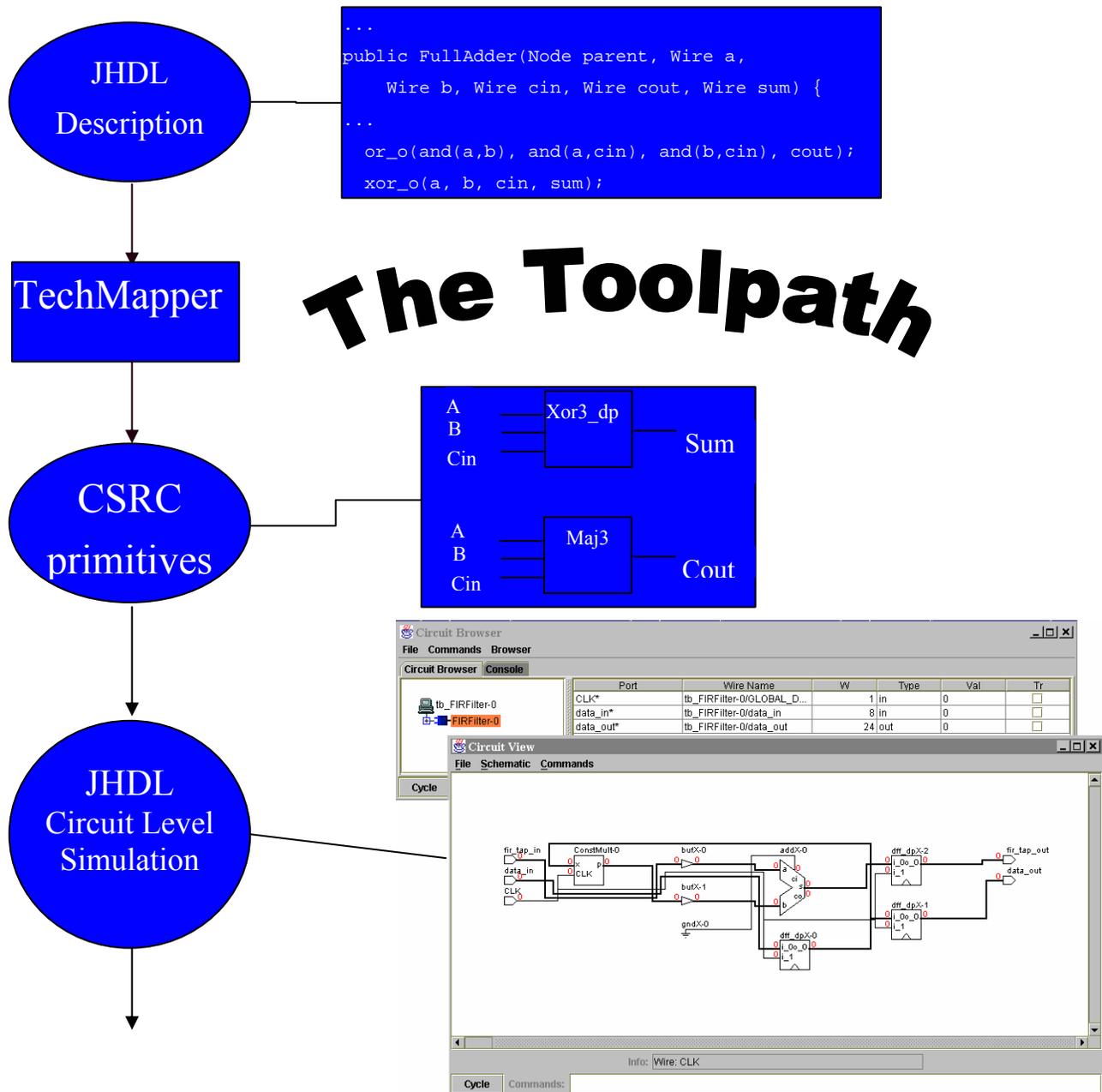


Figure 3.3.1: The Tool Path: JHDL Description to Circuit Simulation

The TechMapper provides for platform independence. It translates a Logic call like `regce(...)` into a platform dependent cell: `fdce(...)` in Xilinx and `dffe_dp(...)` in CSRC. It also provides capability for adding placement hints and data-sharing information and supplies the user access to netlisting for the appropriate platform. Implemented in the TechMapper are:

- Boolean gates
- Registers of all sorts (enable, preset, reset, etc...)
- Adders, subtractors, adder/subtractors
- Shifters, constants, muxes
- Buffers, I/O buffers and pads
- Block RAMs
- Partial placement support

Figure 3.3.2 shows examples of available primitives in the JHDL library of CSRC primitives.

IB	OB
and2_dp	and3_dp
nand2_dp	nand3_dp
or2_dp	or3_dp
nor2_dp	nor3_dp
xor2_dp	xor3_dp
xnor2_dp	xnor3_dp
not_dp	buf
mux3_dp	mux6_dp
maj3	dff_dp
dffr_dp	dffs_dp
dffe_dp	dffre_dp
dffse_dp	DL_ONE
DL_ZERO	blockram

Figure 3.3.2: JHDL Library of CSRC Primitives

Figure 3.3.3 shows the continuation of the Toolpath from Figure 3.3.1.

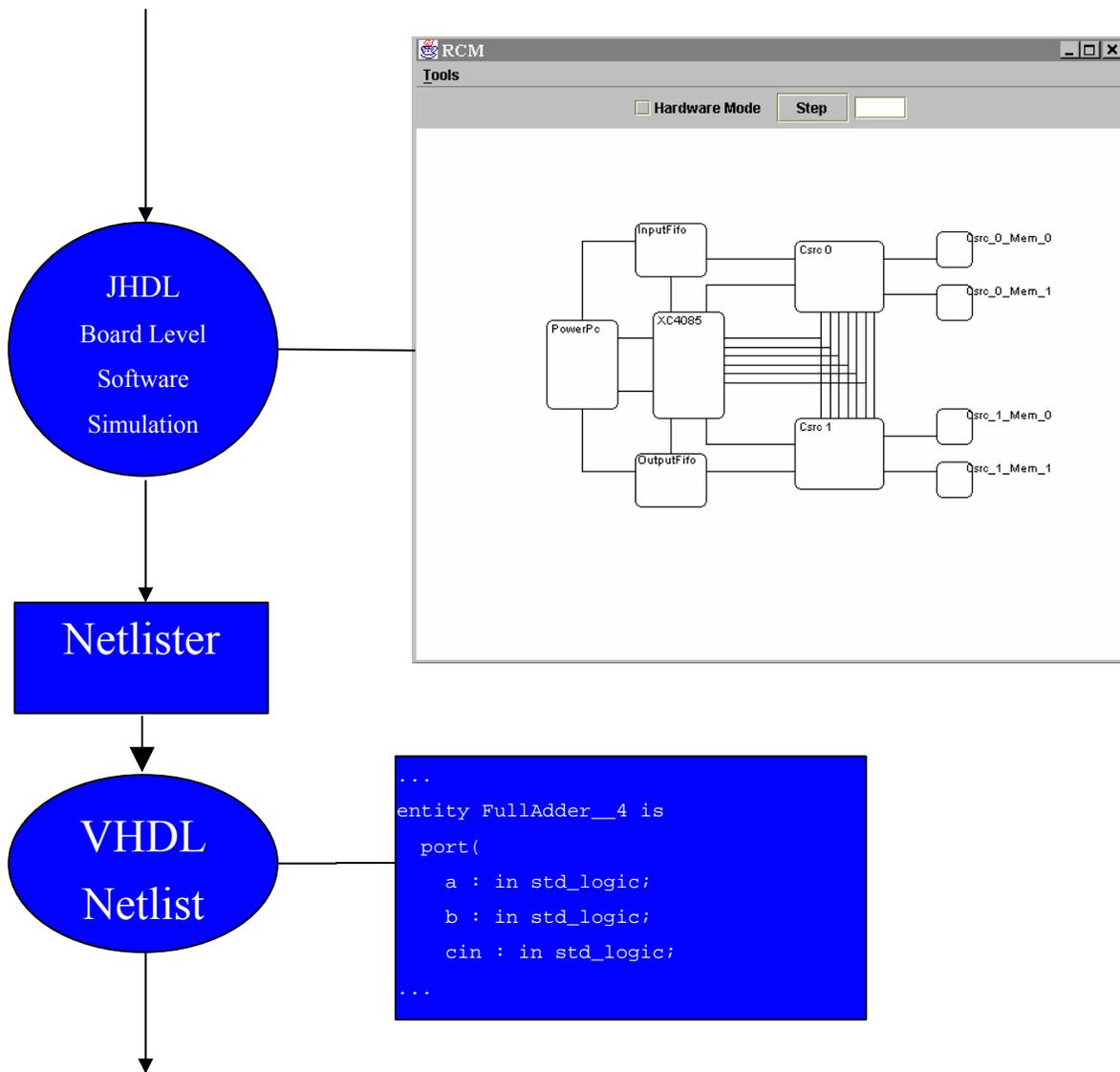


Figure 3.3.3: The Tool Path: JHDL Board Level SW Simulation to VHDL Netlister

With regards to the VHDL netlister, the CSRC place and route uses a mapped VHDL or Verilog netlist in the format Synplicity generates. The internal cell hierarchy of JHDL is translated into mapped VHDL format, consisting only of primitives known to the place and route tool. The netlister enables a second tool path for CSRC circuit designs, the two consisting of:

- VHDL → Synplicity → mapped VHDL/Verilog
- JHDL → mapped VHDL

The next figure for the Toolpath, Figure 3.3.4, shows the Bitstream file generation from the Multi-context Place and Route backend tools (CSRC tools).

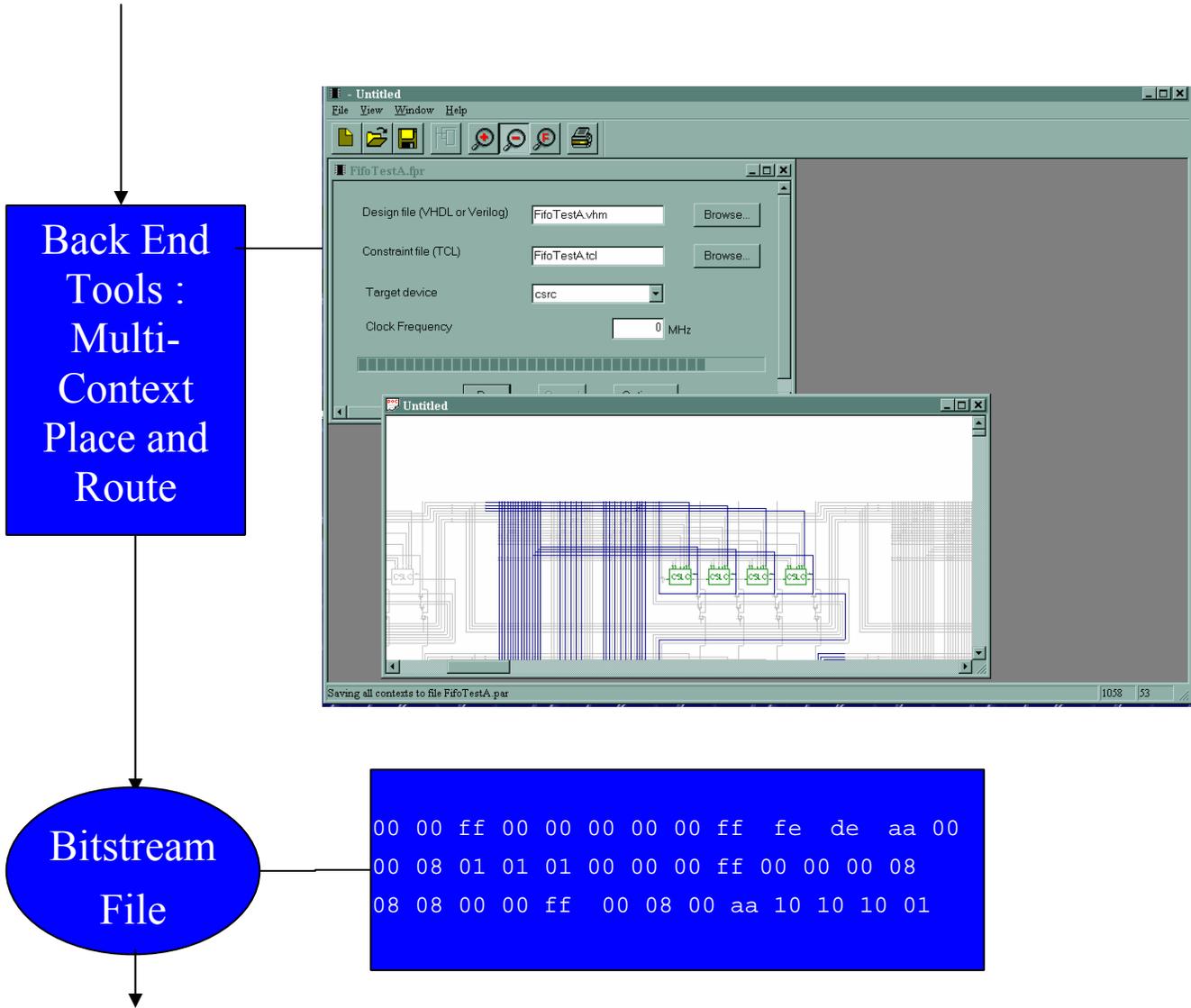


Figure 3.3.4: The Tool Path: Multicontext Place and Route (CSRC Tools) to Bitstream File Generation

The last of the Toolpath is shown in Figure 3.3.5, which provides an example of the JHDL board level hardware verification capabilities.

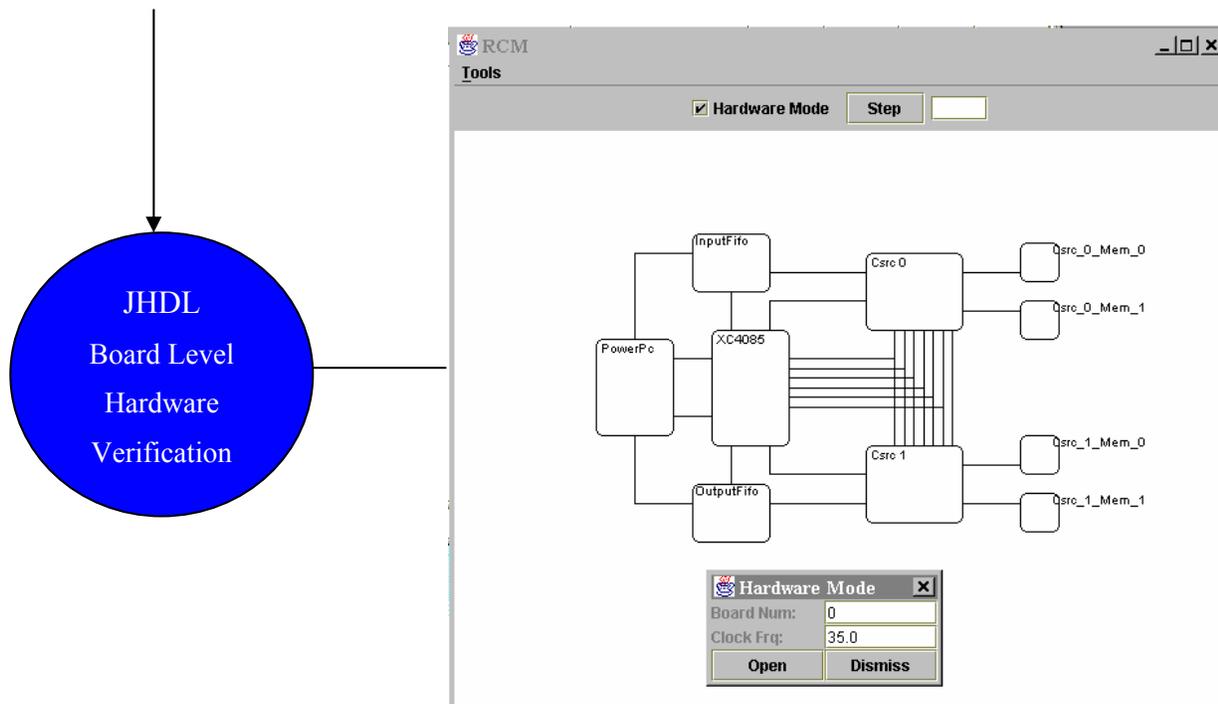


Figure 3.3.5: The Tool Path: JHDL Board Level Hardware Verification

To implement the JHDL environment for CSRC support, two specific JHDL feature modifications were completed. First, support was added for multi-context simulation. Second, a structure and modifications to the user interface were added that now allow for data sharing. Both of these features are discussed in greater detail in the following section dedicated to the RCM board model development and capabilities.

3.3.3 RCM Board Model

The RCM board model extends the JHDL simulator to provide a quick and robust simulation environment for board-level CSRC designs. This includes simulation of all board components, including FIFO's, memories, Xilinx component (XC4085) and CSRC's. JHDL provides an environment for rapid development, prototyping and debugging. The board model allows the user to load, simulate, and debug RCM board designs in software and hardware modes. It provides a user-friendly graphical interface for netlisting and host-driven context switching and has fully integrated the VT developed SLAAC API for controlling the board.

3.3.4 JHDL (Software) Mode

The JHDL mode is the software mode. It can be used without a board. In this mode, the user can simulate and debug his JHDL designs (written in Java) for the CSRC. This mode also provides VHDL netlisting support for user designs, and dynamically generates the TCL files needed for the backend CSRC tools. Other unique features provided for in the JHDL mode include simulation support for multiple contexts and support for data sharing between registers of different contexts. With regards to data sharing, individual designs may set registers as “shared”. When a context is activated, it’s shared registers are updated from the public registers. When the user switches to a new context, the values on the shared registers of the old context are saved to the public registers.

Data sharing is accomplished in the following manner. In the constructor, instance a register similar to:

```
reg_0(delay, Shared_0_Bus_Out);
```

The above establishes that the wire Shared_0_Bus_Out is the output wire. To set this register as shared, the register’s output wire is simply passed to a special function call:

```
SetAsShared(“incrVal”, Shared_0_Bus_Out);
```

The string “incrVal” simply denotes a shared ID. All registers that share with each other should have the same shared ID. This shared ID will appear after the register name in the schematic viewer. Figure 3.3.6 shows a typical example of the schematic viewer:

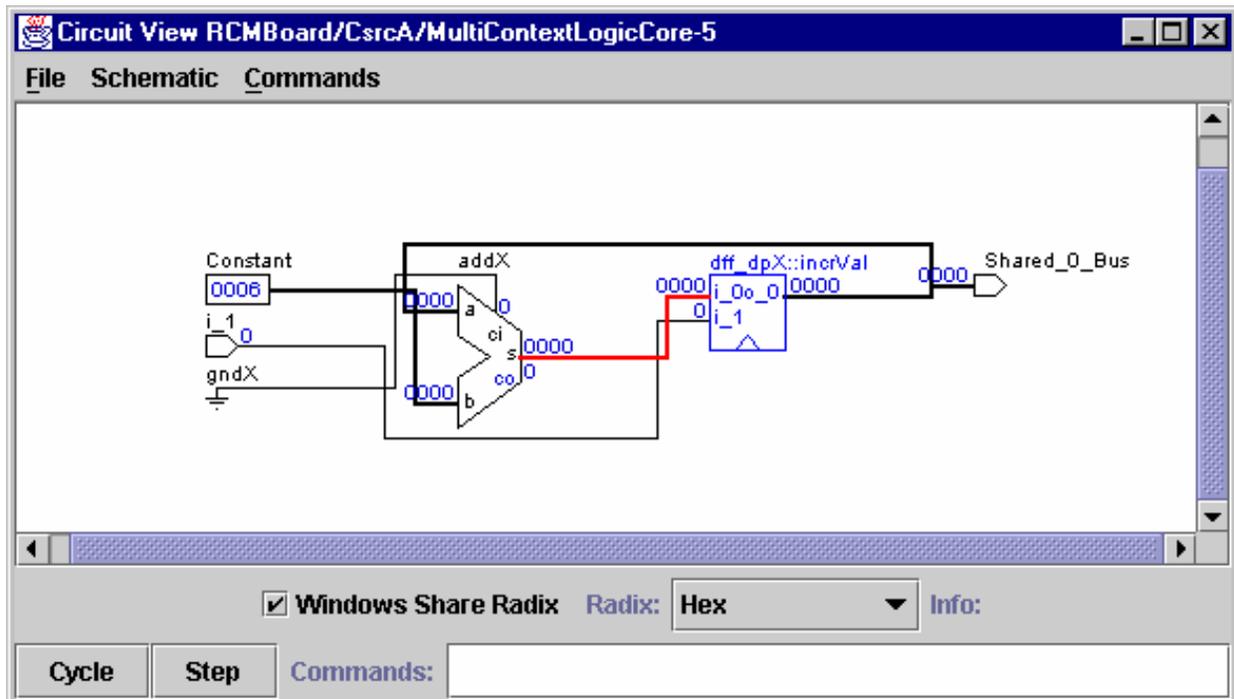


Figure 3.3.6: Schematic Viewer - Shared Data Register in Design

Figure 3.3.7 shows an example of registers that are not shared with each other (different names).

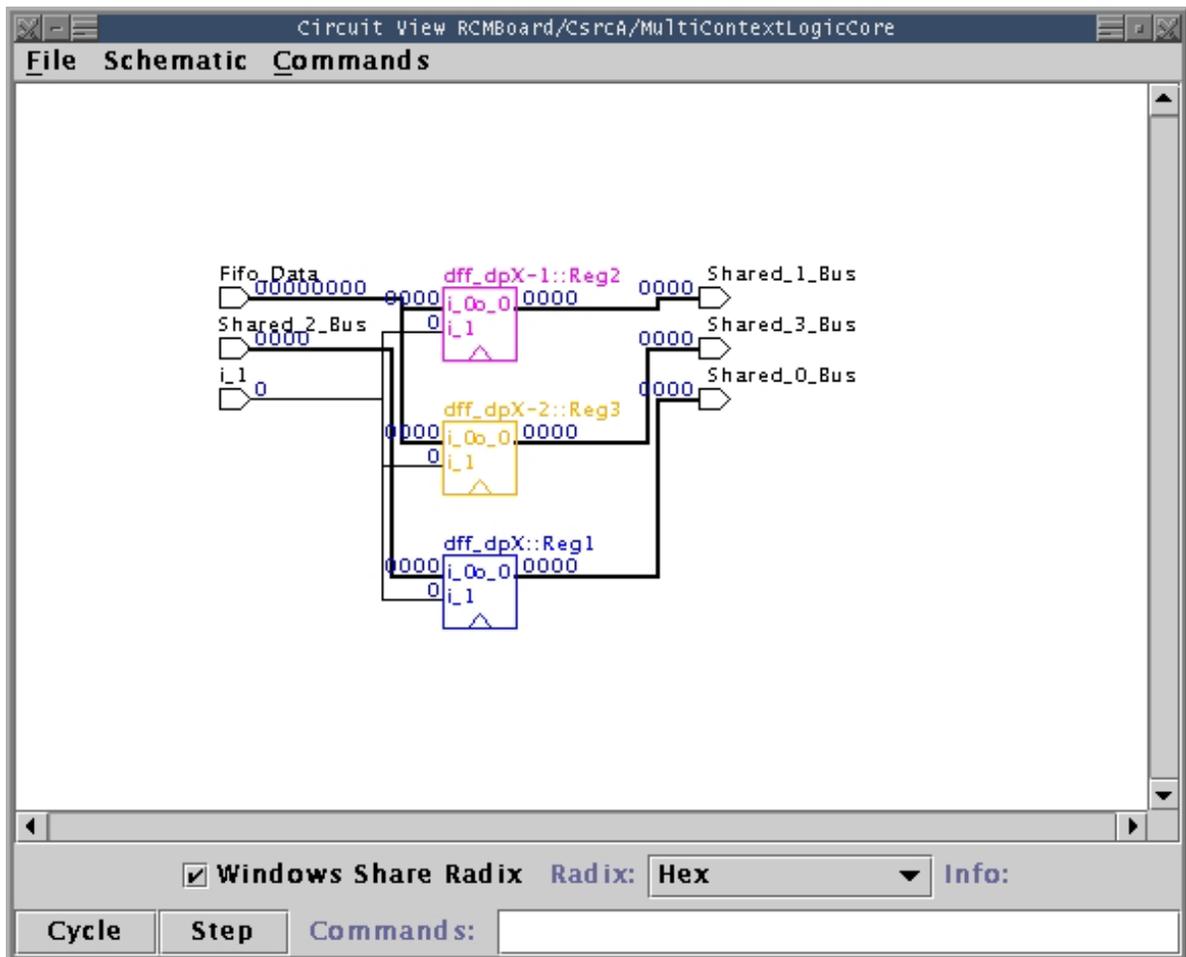


Figure 3.3.7: Schematic Viewer – Non-Shared Data Registers in Design

3.3.5 Hardware Mode

In hardware mode, the board model provides an interface to the RCM_API (extension of SLAAC API) developed by Virginia Tech. A Java Native Interface (JNI) is used to make calls to the c libraries. The hardware mode allows the user to download bitstreams onto the board and execute them without having to know the low-level implementation rules. The board model in hardware mode has the same basic look and feel as JHDL mode, although the lower level details are much different. The board model has recently been modified to fully support the caching scheme developed by Virginia Tech in the RCM API. With the nature in which the RCM API was integrated, it is easy to add support for additional API calls as they become available.

3.3.6. Creating and Building a Design

This section will describe how a basic design would be created and used in the JHDL environment with the RCM board model.

Environment Setup:

To begin, it should be verified that the latest “rcm” and JHDL jars are to be used. The JHDL is available for download from www.jhdl.org. Next, java needs to be set up, making sure to include the necessary jars in your Classpath. Next, if required or desired, the sample tutorial included in the documentation should be explored. The tutorial has been created to help users become familiar with the basics of board models. Documentation is included in the rcm jar.

Create a Basic Design:

Use a basic template such as:

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.CSRC.*;
import byucc.jhdl.platforms.rcm.*;

public class CSRCTemplate extends MultiContextLogicCore {

    public static CellInterface[] cell_interface = {
        //Put all your ports here
    };

    public CSRCTemplate(csrc parent) {
        super(parent);
        //Construct your circuit here
    }
}
```

Declare Ports:

Specify the ports to be used in the design, such as:

```
public static CellInterface[] cell_interface = {
```

```
    out("Fifo_Data",32),
    in("Shared_0_Bus",16),
    in("Shared_1_Bus",16),
};
```

Build the Circuit:

Put the detailed components of the circuit in the constructor method. First grab the wires that will be needed with wa() calls:

```
Wire Fifo_Data_Out = connect("Fifo_Data", wa("Fifo_Data"));
Wire Shared_0_Bus_In = connect("Shared_0_Bus", wa("Shared_0_Bus"));
Wire Shared_1_Bus_In = connect("Shared_1_Bus", wa("Shared_1_Bus"));
```

At this step, the wires are connected with calls to the JHDL Logic API:

```
buf_o(Shared_0_Bus_In,Fifo_Data_Out.range(15,0));
buf_o(Shared_1_Bus_In,Fifo_Data_Out.range(31,16));
```

The following is the source for a simple incrementer:

```
Wire constVal = constant(this,16,incrementValue);
Wire Shared_0_Bus_Out =connect("Shared_0_Bus",wa("Shared_0_Bus"));

Wire delay = add(Shared_0_Bus_Out,constVal);
reg_o(delay,Shared_0_Bus_Out);
```

Figure 3.3.8 shows what the incrementer looks like in the Schematic Viewer.

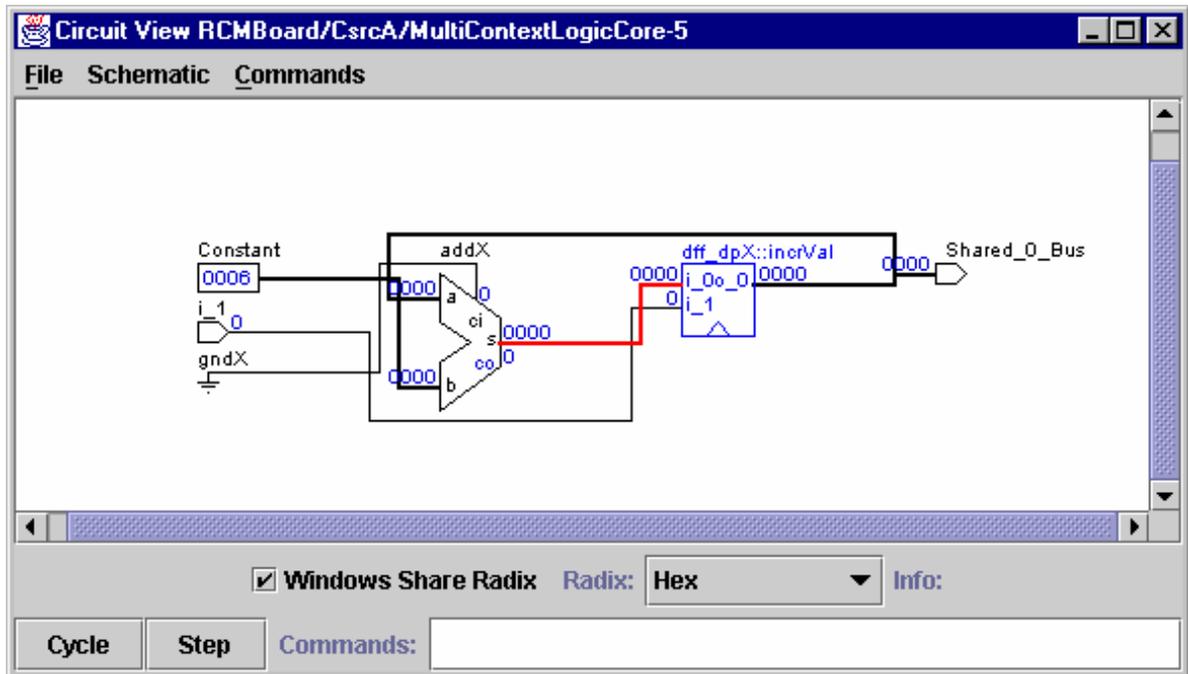


Figure 3.3.8: Schematic Viewer – Incrementer Design

3.3.7 Technical Discussion

The RCM board model developed at BYU reached a mature state. The RCM model has the ability to perform multi-context simulations, supports simulation of data sharing between contexts, has been updated for full integration of VT’s RTR SLAAC API/environment including VT’s caching scheme, has limited debug support, and is supported with comprehensive documentation.

With regards to developing a debugging environment, one of the initial debug approaches considered for the CSRC context-switched device was the placement of debug circuitry in a dedicated context. This approach was ruled out when it was determined that when the context switches are data driven, the context running on the device is not yet externally available. It would not be possible for the host computer to switch out of the current context into the debug context and back without knowing which context to return to. The debug scheme chosen could not require any context switches, thus defeating the primary intent of the project.

In place of the dedicated context debug scheme, design-level scan was chosen. Design-level scan consists of adding multiplexors and gates to the memory elements of a design--such as flip-flops and embedded RAMs--so that the state elements' values can be serially shifted out of the FPGA. The main downside is that this added user circuitry may impose a high overhead to

implement, even though it can be removed when debugging is complete. There are several benefits to using design-level scan. First, it can be added to any user design on the CSRC without additional hardware modifications. Secondly, the scan bitstream is generally small and easy to manipulate compared to a readback bitstream. Third, the circuit runs at full speed until the user is ready to take a snapshot of the circuit state, at which point the clock is still running in order to scan out the circuit state, although no useful work is being done by the circuit during those clock cycles. Fourth, scan allows the state of the circuit to be modified, providing the user with the ability to bring the circuit into a known state.¹ Most of the code to implement scan was already a part of JHDL and only slight modifications were needed to integrate it with the CSRC library primitives.

During development, BYU was unable to actually run any scan-instrumented designs in hardware. This was due to the fundamental need to produce bitstreams from JHDL designs. Early problems with the back-end place and route tools prevented BYU from generating test bitstreams through their toolpath.

Because of the time frame that was required to bring the CSRC tools in-house at BAE SYSTEMS and the time required to concentrate on resolving tools issues required to support the BAE SYSTEMS and VT demonstrations, support of CSRC tools issues particular to BYU's environment for implementing CSRC/RCM designs was necessarily limited. As a result, implementing designs in the BYU environment was limited. Full testing of circuit designs created from JHDL would have required additional CSRC tools fixes to allow for proper handling of generated netlists and successful place & route and bit stream generation for HW implementation.

4.0 Summary

DRACS technology using context switching in CSRCs remains a viable, exciting technology. The goals of this program have been achieved with much success, although room for further improvements certainly exist. Progress with the CSRC tools set reached a very efficient and workable stage following significant efforts to bring the tools (source code) and expertise in-house. This allowed resolution of many technical issues related to the CSRC tools and then provided for the capability to enhance the tools (e.g. basic carry chain support). Both Virginia Tech and BYU teams remained fully engaged and supportive in achieving their goals related to the DRACS program. Time and budget constraints (primarily while waiting for our required ramp up with bringing the CSRC tools in-house) unfortunately curtailed BYU's support for full implementation of JHDL for CSRC technology. However, great progress was achieved and fully documented for potential future follow-on efforts. Through the dedicated development efforts of the team members and the resultant program demonstrations, RTR using CSRC technology for bringing the "hardware to the data" in a viable, supported environment is now available to the ACS and other interested communities.

¹ Wheeler, T.B. (2001). Improving Design Observability and Controllability for Functional Verification of Fpga-Based Circuits Using Design Level Scan Techniques. Master's Thesis, Brigham Young University, Provo, Utah, Department of Electrical and Computer Engineering.

APPENDIX A

FIFO Example VHDL

```
-----  
-- FIFO STREAMING LOGIC EXAMPLE  
-- Created as a drop-in core for the CSRC device on the BAE Systems  
-- RCM board.  
--  
-- The bare-bone code below describes the use of the FIFO write  
-- enable signal.  
-- Created by K. Puttegowda on 1/25/02  
-----  
  
entity bin_image_gen is  
    generic (    NUMBER_OF_PIXEL_ADDRS    :    integer:= 9243 --10240 --20480  
                -- Number of address to hold pixels for image  
                );  
    port (  
  
        CLK                : in  std_logic;  -- clock  
  
        csrca2csrcb : out STD_LOGIC_VECTOR(15 downto 0);  
        FIFO_WE       : out std_logic; -- fifo read enable  
active low. Will be delayed by aclock in the xilinx  
        FIFO_RE       : out std_logic;  
        BIN_GEN       : in  std_logic;  
        DONE_BIN_GEN  : out std_logic  
    );  
end bin_image_gen;  
  
architecture rtl of bin_image_gen is  
  
    -- Internal signals  
  
    -- address count  
    signal address : std_logic_vector(13 downto 0);  
  
begin  
    FIFO_RE <= '1'; -- Read not enabled. Sorry should have been named RE_N  
    csrca2csrcb(15 downto 8) <= "11111111";
```

```

    csrca2csrcb(7 downto 0) <= "00000000"; -- data lines to FIFO. If the
design is on CSRCA these go through CSR CB. Must place a passthrough on
CSR CB
    seq:process (CLK)
    begin
        if clk'event and clk='1' then
            if BIN_GEN ='0' then
                address <= (others => '0');
            else
                if control = '1' then
                    address <= address + "000000000000001";-- counter
which counts the number of data
                else
                    address <= address;
                end if;
            end if;
        end if;
    end process;

    comb:process (address, BIN_GEN)
    begin
        if BIN_GEN='0' or conv_integer(address) = (NUMBER_OF_PIXEL_ADDRS +
1) then -- check start of processing end of count
            control <= '0'; -- control for enabling processing in CSRC
and FIFO control
        else
            control <= '1';
        end if;
    end process;

    FIFO_WE <= not control; -- FIFO write enable
    DONE_BIN_GEN <= not control ;

end rtl;

```

```

-----
--The bare-bone code below describes the use of fifo read enable signal
-----

```

entity difference is

```

generic (  NUMBER_OF_PIXEL_ADDRS   :   integer:= 9600 --10240 --20480
          -- Number of address to hold pixels for image
          );
port (
          CLK           : in  std_logic;  -- clock

          FIFO_2_CSRCA   : in  STD_LOGIC_VECTOR(15 downto 0);
          FIFO_RE        : out std_logic;  -- fifo read enable
active low. Will be delayed by aclock in the xilinx
          FIFO_WE        : out std_logic;  -- fifo read enable
active low. Will be delayed by aclock in the xilinx
          DIFF_CALC      : in  std_logic;
          DONE_DIFF      : out std_logic
        );
end difference;

architecture rtl of difference is

-- Internal signals

-- address count
signal address, n_address : std_logic_vector(13 downto 0);

begin
  FIFO_WE <= '1'; -- Write not enable.
  -----<= FIFO_2_CSRCA; -- Input FIFO data lines. delayed by a clock.
See spec sheets for timing diagrams
  seq:process (CLK)
  begin

    --if RESET='1' then

    if clk'event and clk='1' then
      if DIFF_CALC = '0' then
        address <= (others => '0');
      else
        address <= address + "000000000000001"; -- count for
number of bytes to be read from the FIFO
      end if;
    end if;
  end process;
end difference;

```

```

    comb:process (mode, DIFF_CALC, address, csrcmem_wc1, csrcmem_wc2,
csrcmem_wc3, csrcmem_wc4, control)
    begin

        if DIFF_CALC='1' and conv_integer(address) <
(NUMBER_OF_PIXEL_ADDRS) then -- check for end of count
            FIFO_RE <= '0';
        else
            FIFO_RE_d <= '1'; -- FIFO read controls
        end if;

    end process;

end rtl;

```

FIFO Example Constrain file(pipe_b_demo.tcl)

```

location "CSRCB_2_FIFO_15" "B7"
location "CSRCB_2_FIFO_14" "C8"
location "CSRCB_2_FIFO_13" "B9"
location "CSRCB_2_FIFO_12" "C10"
location "CSRCB_2_FIFO_11" "E8"
location "CSRCB_2_FIFO_10" "B8"
location "CSRCB_2_FIFO_9" "A9"
location "CSRCB_2_FIFO_8" "B10"
location "CSRCB_2_FIFO_7" "A7"
location "CSRCB_2_FIFO_6" "D9"
location "CSRCB_2_FIFO_5" "E10"
location "CSRCB_2_FIFO_4" "A10"
location "CSRCB_2_FIFO_3" "D8"
location "CSRCB_2_FIFO_2" "C9"
location "CSRCB_2_FIFO_1" "D10"
location "CSRCB_2_FIFO_0" "E11"

location "fifoout_we" "F31"

location "CSRCA_2_CSRCB_0" "B19"

```

```
location "CSRCA_2_CSRCB_1" "A20"  
location "CSRCA_2_CSRCB_2" "E20"  
location "CSRCA_2_CSRCB_3" "D21"  
location "CSRCA_2_CSRCB_4" "C19"  
location "CSRCA_2_CSRCB_5" "B20"  
location "CSRCA_2_CSRCB_6" "A21"  
location "CSRCA_2_CSRCB_7" "B22"  
location "CSRCA_2_CSRCB_8" "D19"  
location "CSRCA_2_CSRCB_9" "C20"  
location "CSRCA_2_CSRCB_10" "B21"  
location "CSRCA_2_CSRCB_11" "C22"  
location "CSRCA_2_CSRCB_12" "E19"  
location "CSRCA_2_CSRCB_13" "D20"  
location "CSRCA_2_CSRCB_14" "C21"  
location "CSRCA_2_CSRCB_15" "D22"  
  
location "csrca_fifoout_we" "B25"
```

FIFO Example Constrain file(bin_image_gen.tcl)

```
location "csrclmem_A_0" "AD29"  
location "csrclmem_A_1" "AF32"  
location "csrclmem_A_2" "AG32"  
location "csrclmem_A_3" "AH33"  
location "csrclmem_A_4" "AE32"  
location "csrclmem_A_5" "AF31"  
location "csrclmem_A_6" "AF29"  
location "csrclmem_A_7" "AH32"  
location "csrclmem_A_8" "AE31"  
location "csrclmem_A_9" "AF30"  
location "csrclmem_A_10" "AG31"  
location "csrclmem_A_11" "AG29"  
location "csrclmem_A_12" "AE30"  
location "csrclmem_A_13" "AG33"  
location "csrclmem_A_14" "AG30"  
location "csrclmem_A_15" "AH31"  
location "csrclmem_A_16" "AA32"  
location "csrclmem_A_17" "Y31"  
location "csrclmem_A_18" "W30"
```

location "csrcblmem_Q_0" "AA2"
location "csrcblmem_Q_1" "AB3"
location "csrcblmem_Q_2" "AJ2"
location "csrcblmem_Q_3" "AH3"
location "csrcblmem_Q_4" "AG4"
location "csrcblmem_Q_5" "AG1"
location "csrcblmem_Q_6" "AH5"
location "csrcblmem_Q_7" "AH2"
location "csrcblmem_Q_8" "AG3"
location "csrcblmem_Q_9" "AF4"
location "csrcblmem_Q_10" "AJ1"
location "csrcblmem_Q_11" "AG5"
location "csrcblmem_Q_12" "AG2"
location "csrcblmem_Q_13" "AF3"
location "csrcblmem_Q_14" "AH4"
location "csrcblmem_Q_15" "AH1"

location "csrcblmem_W_N" "Y32"
location "csrcblmem_E_N" "V29"
location "csrcblmem_G_N" "AA33"
location "csrcblmem_SE" "W31"
location "csrcblmem_SCK" "Y29"
location "csrcblmem_SD1" "Y33"

location "csrca2csrcb_0" "B19"
location "csrca2csrcb_1" "A20"
location "csrca2csrcb_2" "E20"
location "csrca2csrcb_3" "D21"
location "csrca2csrcb_4" "C19"
location "csrca2csrcb_5" "B20"
location "csrca2csrcb_6" "A21"
location "csrca2csrcb_7" "B22"
location "csrca2csrcb_8" "D19"
location "csrca2csrcb_9" "C20"
location "csrca2csrcb_10" "B21"
location "csrca2csrcb_11" "C22"
location "csrca2csrcb_12" "E19"
location "csrca2csrcb_13" "D20"

