Simplifying System-on-Chip Design through Architecture and System CAD Tools

by

Lesley Shannon

A Thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy,
Department of Electrical and Computer Engineering,
University of Toronto

Simplifying System-on-Chip Design through Architecture and System CAD Tools

Lesley Shannon
Doctor of Philosophy,
2006
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Historically designers created computing systems by combining Integrated Circuits (ICs) on Printed Circuit Boards (PCBs), whereas now they are able to form complete Systems-on-Chip (SoCs). For the purpose of this study, SoCs are defined as a collection of functional units on one chip that interact to perform a desired operation. These modules are typically of a coarse granularity to promote reuse of previously designed Intellectual Property (IP). The decreasing size of process technologies enables designers to implement increasingly complex SoCs using both Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). The impact of increasing design complexity is increased design time and costs for electronics. Therefore, this research investigates methods to facilitate the design of SoCs through both architecture and CAD tools.

This thesis has two main contributions. The first is an architectural framework for SoCs, wherein they are modelled as Systems Integrating Modules with Predefined Physical Links (SIMPPL). The strength of the model is the Computing Element (CE) abstraction that separates the module's datapath from system-level control and communications to facilitate design reuse. Although SIMPPL can be used to build SoCs for ASICs or FPGAs, using an FPGA provides designers with a reprogrammable implementation platform. Thus, our second contribution is to develop a design infrastructure that leverages the advantages of reconfigurability.

Acknowledgements

First, I would like to my parents and my brother, Matthew: I finally did it. I know you are probably wondering what took me so long, but I finally finished. Thank you for believing that I *would* finish... eventually. Also, thank you to my extended family for your love and support, and for keeping me grounded. All work and no play makes Lesley a dull girl – and a little stir crazy.

To the many friends I have made throughout my graduate studies at the University of Toronto. SF2206, LP392 and EA306 will always hold a special place in my heart. Thank you for all the long talks, extra-curricular events and crazy antics that helped to calm the madness of a stressful environment. My door will always be open to each of you.

To my many friends outside of work. You have given me sound advice and made sure that I have remembered that the world is a beautiful place built on science, but filled with art and music. Thank you for nurturing the non-engineering parts of my soul.

Lastly, but certainly not least, I must thank my supervisor, Dr. Paul Chow. My Doctorate studies have been a fantastic experience and I know that I interminably indebted to you for this. Over the years I have seen how crucial the rapport between student and supervisor is to the graduate experience. All I can say is that if my students find me to be half as good a supervisor as you have been to me, I will consider myself a success.

I would also like to acknowledge the financial support, as well as the equipment and software donations, that I have received from the following organizations that have made this research possible: the Canadian Microelectronics Corporation, the Natural Sciences and Engineering Research Council, the O'Brien Foundation, the Ontario Government, the University of Toronto, the Walter C. Sumner Foundation, and Xilinx.

Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CAD	Computer Aided Design
CE	Computing Element
CLB	Configurable Logic Block
EX IR	Executing Instruction Register
FPGA	Field Programmable Gate Arrays
FSL	Fast Simplex Links
FSM	Finite State Machine
HDL	Hardware Description Language
IC	Integrated Circuit
IDCT CE	Inverse Discrete Cosine Transform CE
IP	Intellectual Property
IQ CE	Inverse Quantizer
ISS	Instruction Set Simulator
LSB	Least Significant Bits
LUTs	Look Up Tables
MC/PR CE	Motion Control/Picture Reconstruction CE
MDM	Microprocessor Debug Module
MHS	Microprocessor Hardware Specification
MMR CE	Missing Macroblock Replacer CE
MPD	Microprocessor Peripheral Definition
MSS	Microprocessor Software Specification
NoC	Network-on-Chip
OCP	Open Core Protocol
OPB	On-chip Peripheral Bus
PAO	Peripheral Analyze Order
PC	Program Counter
PCB	Printed Circuit Board
PE	Processing Element
SCS	SIMPPL Control Sequencer
SIMPPL	Systems Integrating Modules with Predefined Physical Links
SnoopP	Snooping Profiler
SoC	System on Chip
SUT	System Under Test
VLD/RLD CE	Variable Length Decoder/Run Level Decoder CE
VSIA	Virtual Socket Interface Alliance
WOoDSTOCK	Watching Over Data STreaming On Computing element linKs
XMP	Xilinx Microprocessor Project
XPS	Xilinx Platform Studio

Chapter 1

Introduction

Historically designers created computing systems by combining Integrated Circuits (ICs) on Printed Circuit Boards (PCBs). However, due to the decreasing size of process technologies, designers have been able to implement these same systems as Systems-on-Chip (SoCs) using Application Specific Integrated Circuits (ASICs) since the late 1990's. The term System-on-Chip (SoC) has been used with many different connotations in previous work. For this study, we define an SoC as a collection of functional units on one chip that interact to perform a desired operation. These modules are typically of a coarse granularity so that previously designed Intellectual Property (IP) modules can be reused to try and reduce the design time of more complex systems. Examples of IP modules range from data intensive processing cores, such as FIR filters and FFTs, to more control intensive cores, such as memory controllers and processors.

1.1 Motivation

IP reuse is more challenging in hardware designs than reusing software functions in new software applications. Software designers benefit from a fixed implementation platform with a highly abstracted programming interface, enabling them to focus on adapting the functionality to the new application. Unfortunately, to reuse hardware IP [1, 2], designers need to consider changes to the module's:

- functionality,
- physical interface, and
- communication protocols.

Depending on the amount of time required to adapt IP to a new application, there may be little benefit in reusing the IP to create new SoCs. However, if we create a framework for describing SoCs to simplify the integration of IP modules, it would allow hardware designers to focus on adapting IP functionality similar to software designers updating software functions.

Another challenge for SoC designers is verifying design functionality and performance in a timely fashion. Designers have traditionally relied on simulation and estimation to evaluate their systems. Given the potential size and complexity of SoCs, simulation can be a very time consuming process that takes orders of magnitude longer than on-chip execution. However, if an SoC is implemented on an ASIC, it has a restrictive design environment that is not easily altered post fabrication. Therefore, the importance of correctly implementing a design on an ASIC the first time necessitates lengthy simulation times to prevent a costly redesign.

Now that commercial Field Programmable Gate Arrays (FPGAs) are also large enough to implement entire systems on one chip [3, 4], as opposed to just the glue logic, they offer a unique opportunity for SoC designers. When using a reconfigurable implementation platform, there is no cost to reprogramming the hardware, therefore, we can develop a new design infrastructure where system evaluation is performed on-chip. It would provide greater flexibility to the designer and allow a new approach to the design process. For example, Hemmert et al. [5] introduced a debugger for hardware designs capable of running on an FPGA for the benefit of accelerated speed of execution during the debugging process. Recent work allows designers to incorporate a *Statistics Module* into a soft processor to obtain a variety of run-time statistics that can be dynamically reconfigured [6]. Furthermore, designing for a reconfigurable implementation platform enables designers to easily respecify the system's architecture if the on-chip evaluation determines that the current architecture fails to meet design specifications.

1.2 Objective

The objective of this research is twofold:

- To study how creating a framework for SoC architectures can facilitate both IP reuse and system design.

- To develop a set of on-chip CAD tools that can exploit FPGAs to reduce time spent evaluating the functionality and performance of SoCs.

Defining a system framework requires that the system-level communication structure used to integrate the IP modules be characterized. It also necessitates the specification of the physical interface as well as the communication protocols for IP modules, to facilitate their reuse in different applications. Finally, a formalized method for adapting how an IP module is used by different systems without necessitating significant redesign is desired to facilitate design reuse.

Having created a system framework, it is possible to develop on-chip CAD tools that can be tailored to different SoC architectures. Given the unrestricted ability to reprogram an FPGA, on-chip CAD tools can be used during the design process to evaluate functionality and performance. By performing these operations on the runtime platform, designers can reduce simulation time and overall design time.

1.3 Contributions

This thesis can be divided into two significant contributions:

- an architectural framework for SoCs, and
- a design infrastructure developed to leverage the advantages of reconfigurability and a defined SoC model.

The proposed framework models SoCs as Systems Integrating Modules with Predefined Physical Links (SIMPPL [7]) to expedite system integration. Within this framework, IP modules are abstracted as Computing Elements (CEs) to reduce the complexities of adapting IP to new applications. A lightweight controller has been created to provide a fixed system-level interface for the IP module with standardized communication protocols. It also executes a program that dictates how the IP is used in the system, thus localizing the system-level control to simplify any necessary functional redesign of the IP for other applications.

Designers implementing SoCs on FPGAs can leverage configurability by moving the evaluation of the system on-chip. This can reduce system design time by decreasing the amount of time spent simulating the system's runtime behaviour, while still providing accurate information. To this end, two on-chip profiling tools, SnoopP [8] and WOoDSTOCK [7] have been designed. Furthermore, fixing the SoC architectural framework allows us to create a system specification tool [7] that can facilitate the redesign of the system-level architecture and an on-chip verification environment [9] for SoCs implemented on FPGAs using the SIMPPL model.

1.4 Thesis Organization

This thesis is divided into eight chapters. Chapter 2 summarizes the previous work done on IP reuse and on-chip communication structures and presents the SIMPPL framework for SoC design. Chapter 3 describes the Computing Element (CE) abstraction that is central to this model. To demonstrate how the SIMPPL framework and CE abstraction can facilitate design, three applications are implemented as SoCs as described in Chapter 4. The remainder of the thesis document discusses designing SoCs on FPGAs. Chapter 5 provides an overview of current research and describes the system-level design tools created for generating and verifying SoCs within the SIMPPL framework on FPGAs. Along with these tools, the design infrastructure for SoCs on FPGAs also includes two profiling tools for evaluating system performance. The first is SnoopP, a *Snoop*ing *P*rofiler for measuring the performance of applications on processors at runtime, which is discussed in Chapter 6, and WOoDSTOCK is the other. Chapter 7 describes how Watching Over Data STreaming On Computing element linKs (WOoDSTOCK) can be used to detect processing load imbalances in systems modelled using SIMPPL. Finally, the conclusions and potential future work for this thesis are summarized in Chapter 8.

Chapter 2

Modelling SoCs: SIMPPL

This chapter begins by summarizing popular methods of simplifying IP reuse in Section 2.1, followed by a discussion of some of the previous work investigating on-chip interconnect structures in Section 2.2. It concludes with a presentation of the SIMPPL system framework for SoC design in Section 2.3.

2.1 IP Reuse

Multiple books exist discussing the complexities involved in reusing legacy IP in new designs [1, 2]. Although IP reuse can reduce design time, problems that arise when incorporating previously designed modules into new designs are of significant concern. This has led to the development of well-defined IP design methodologies [10, 11] to ensure reusability of cores with fixed interfaces and functionality. It does not, however, address the common situation where a module has defined functionality but requires the ability to interface with different communication structures.

The Spirit Consortium [12] has created two specifications for facilitating IP reuse. The first is the IP meta-data description, which provides a generic method for describing IP modules. The consortium has also created an IP tool integration API that allows designers to integrate tools into an IP framework for SoC design.

The VSI Alliance has proposed the Open Core Protocol (OCP) [13] to enable the separation of external core communications from the IP core's functionality, similar to the SIMPPL model. Both communication models are illustrated in Figure 2.1. The SIMPPL model targets the direct communication model using a defined, point-to-point interconnect structure for all on-chip communications. In contrast, OCP is used to provide a well-

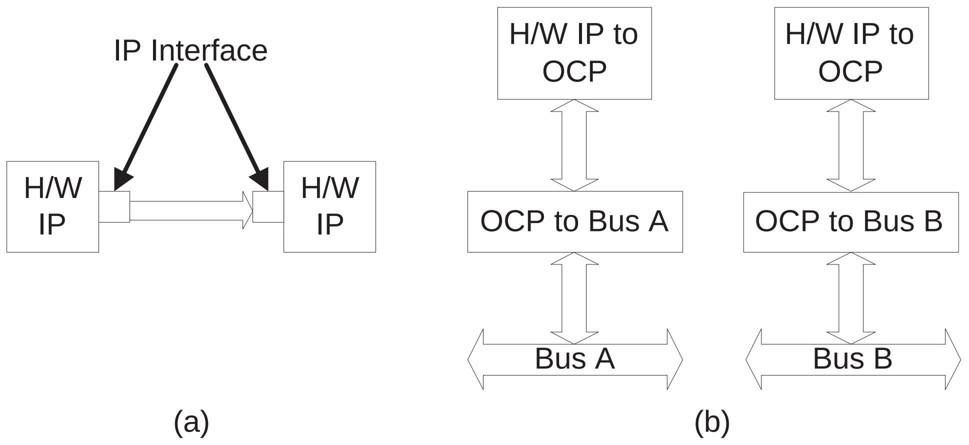

Figure 2.1: Standardizing the IP interface using (a) SIMPPL for point-to-point communications and (b) OCP for different bus standards.

defined socket interface for IP that allows a designer to attach interface modules that act as adaptors to different bus standards that include point-to-point interconnect structures as shown in Figure 2.1(a). This allows a designer to easily connect a core to all bus types supported by the standard.

More recently, an Interface Adaptor Logic (IAL) layer has been proposed [14] that uses a socket interface for IP modules, similar to the OCP. However, unlike OCP, it is specifically aimed at IP reuse in reconfigurable SoCs. FPGA companies also recognize the importance of simplifying the inclusion of previously designed IP into newer system designs. Xilinx provides its own bus-interface module for interconnecting IP with a defined socket interface [15].

All the protocols presented in this section make it easier to port IP among different bus standards. For example, the OCP and the IAL layer provide standardized adapters that allow cores of fixed functionality to connect to a variety of bus standards. The SIMPPL model, however, has a fixed interface, supporting only point-to-point connections with the objective of allowing is to enable designers to treat IP modules as programmable coarse-grained functional units. Designers can then reprogram the IP module's usage in the system to adapt to the requirements of new applications.

2.2 On-Chip Communication Structures

Many different on-chip interconnect strategies have been proposed for SoC design, including hierarchical buses that use bridges to connect to each other [16, 17, 18], but the maximum bandwidth for each bus is limited by the number of modules connected to it. The WISHBONE [19] SoC interconnect architecture provides multiple different interconnect structures, allowing the designer to select the bus architecture for a particular system. Since all the Wishbone interconnects are designed as single-level buses, the standard provides the user with a simpler design approach, unless components running at different clock rates must share the same bus.

Berkeley's SCORE [20] architecture divides system computations into fixed-size pages and uses the data abstraction of streams to pass data between pages. Streams provide a high-level description of point-to-point communication, comparable to the SIMPPL internal communication link, but without defining a physical connection. Adaptive System-on-chip (aSOC) [21] uses a physical implementation of a point-to-point communication architecture for heterogeneous systems, where unlike the SIMPPL model, the communication interface for each module is tailored in hardware to optimize the module's performance.

Networks provide another form of scalable on-chip communication. Multiple Network-on-Chip (NoC) topologies have been studied for ASIC designs [22, 23]. One popular NoC topology is the mesh [24, 25], which has also been investigated on an FPGA platform [26]. The SIMPPL model, however, can be used to implement any fixed point-to-point network topology, allowing the designer to choose the appropriate topology for each application.

2.3 SIMPPL Model

The proposed SIMPPL model represents SoCs as Systems Interfacing Modules with Predefined Physical Links (SIMPPL) [7], implementing an SoC as a combination of different Computing Elements (CEs) that are connected via communication links. Figure 2.2 illustrates a possible SoC architecture described using the SIMPPL model, where the solid lines indicate *internal links* and the dotted lines indicate *I/O communication links*. I/O communication links may require different protocols to interface with off-chip hardware peripherals, but the internal links are standardized physical links with defined communication protocols to make the actual interconnection of CEs a trivial problem and to create a framework for systems design. The current work using the SIMPPL model assumes that the internal links are n-bit wide Asynchronous FIFOs with a user defined depth. Using asynchronous FIFOs simplifies multi-clock domain systems, allowing designers to isolate different clock domains in different CEs and buffer the data transfers between CEs. Point-to-point links not only offer higher bandwidth than shared buses, but recent work has demonstrated that commercial FPGA routing fabrics can implement network topologies where CEs have a high

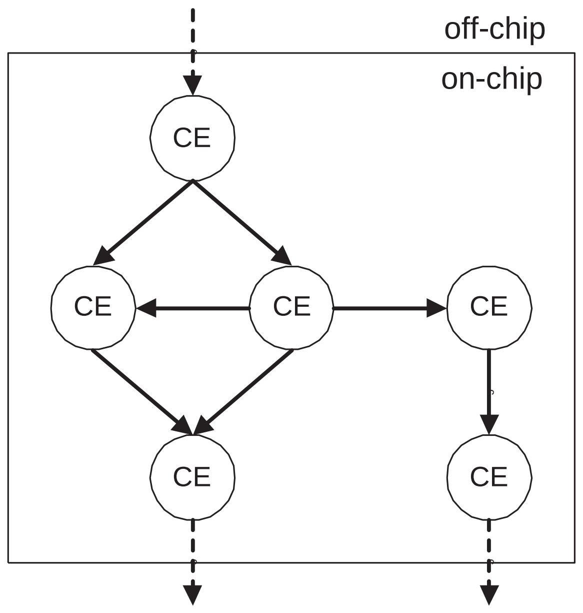

Figure 2.2: A generic SoC described using the SIMPPL model.

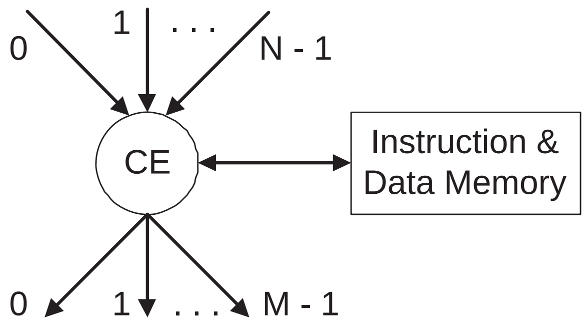

Figure 2.3: The system generator's generic computing element.

degree of connectivity without performance degradation due to routing congestion [27].

Each CE has the generic structure shown in Figure 2.3, where each CE has *N* input links and *M* output links. Internal links connect a CE to other CEs, where input links connect to *parent* CEs and output links connect to *child* CEs. The information passed between CEs is abstracted from the links themselves and instead, the data transfers are adapted to the specific requirements of each CE. This format of communicating data between modules is akin to software design, where the stack provides the physical interface between software functions, similar to the proposed internal links. However, the information passed on the stack, such as the number of parameters, is determined by the individual function calls. In the SIMPPL model, the size and nature of the data in the packet communicated between the IP modules performs this task. Each module has internal protocols capable of properly creating and interpreting the information in a packet.

A proposed model for the future of SoC design using many interacting heterogeneous processors [28] can also have the same structure as a SIMPPL SoC, however the SIMPPL model is more general, allowing CEs to depict either processors (software CEs) or dedicated logic modules (hardware CEs). The SIMPPL model representation of SoCs is more reminiscent of Kahn process networks [29], particularly Data process networks [30], in that it is a collection of CEs interconnected via unidirectional links and well suited to data intensive applications. However, unlike these models that assume the internal links have unbounded capacity, the SIMPPL model uses real FIFOs that have limited capacity. Work at Philips Research produced YAPI [31], an application model based on Kahn process networks that has been extended to support non-deterministic events and decouple the data types used for communications and computation.

Although the SIMPPL model allows non-deterministic events, they are supported by the CE abstraction. The abstraction allows inter-CE synchronization to be programmed to meet the specific requirements of each application. The SIMPPL model only provides a physical structure for the system and is oblivious to the meaning of the data flowing between CEs, deferring the interpretation of the data to the CE abstraction discussed in the following chapter.

Chapter 3

The Computing Element Abstraction

The Computing Element (CE) is an abstraction of software or hardware IP that facilitates design reuse by separating the datapath (computation), the inter-CE communication, and the control. Researchers have demonstrated some of the advantages of isolating independent control units for a shared datapath to support sequential procedural units in hardware [32]. This is similar to when a CE is implemented as software on a processor (software CE), the software is designed with the communication protocols, the control sequence, and the computation as independent functions. Should a software CE need to be reused and updated for a new application, the software changes should be localized to only the control sequence functions.

Typically, complex control is easier to implement in software than in hardware. Figure 3.1 illustrates the desired functionality of a hardware CE. Using a microcontroller as the CE communication interface isolates the Processing Element's (PE's) functionality from the rest of the system. The PE now operates as a coarse grain functional unit that is only accessible via the microcontroller. The PE's local control is encapsulated in the CE's local program. Its instructions, along with data requests from adjoining CEs, are interpreted by

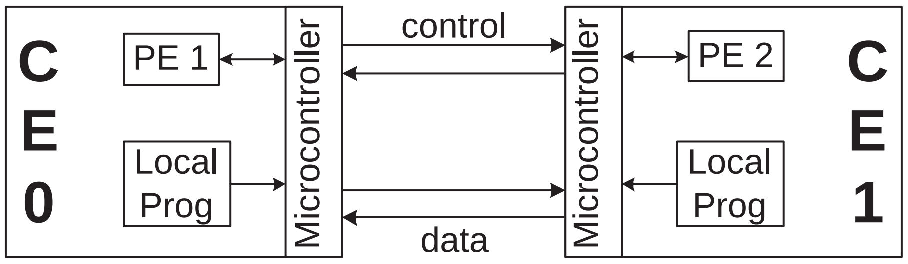

Figure 3.1: The concept for the hardware CE.

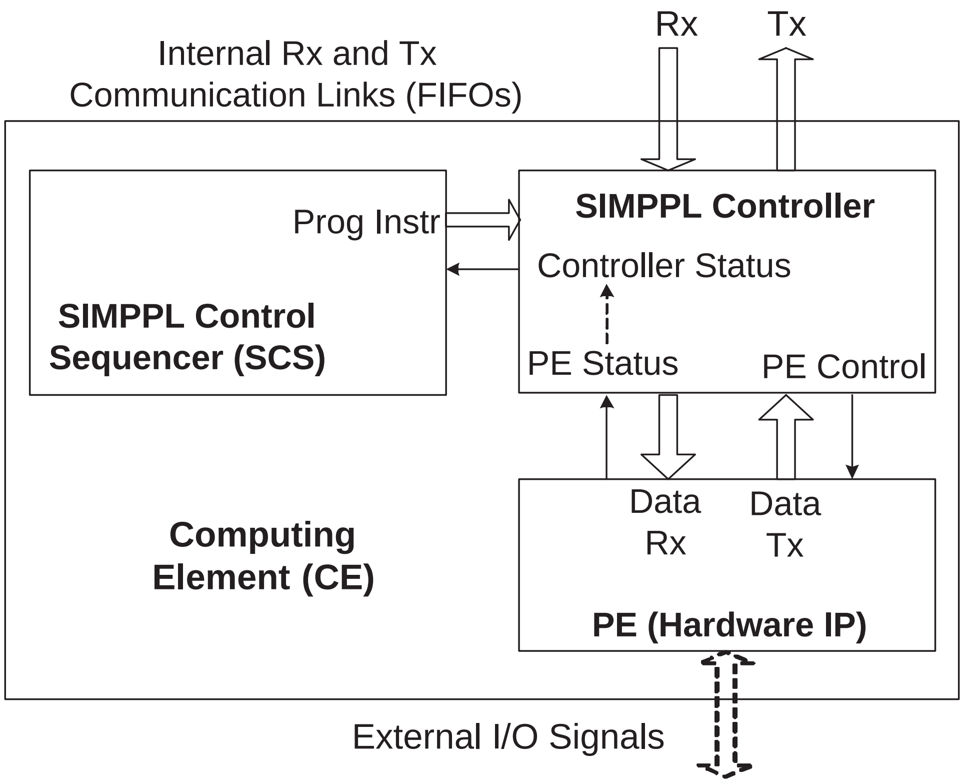

Figure 3.2: The hardware CE abstraction.

the microcontroller and executed by the PE.

However, general purpose microcontrollers are too big and too slow for the hardware-to-hardware interactions of dedicated logic modules in hardware CEs. Ideally, a controller customized to each CE's datapath could be used as a generic system interface, optimized for that specific CE's datapath. To this end, we've created two versions of a fast, programmable, lightweight controller – an execution-only (*execute*) version and a run-time debugging (*debug*) version – that are both adaptable to different types of computations suitable to SoC designs on both ASICs and FPGAs.

Figure 3.2 illustrates how the control, communications and the datapath are decoupled in hardware CEs. The Processing Element (PE) represents the datapath of the CE or the IP module, where an IP module implements a functional block having data ports and control and status signals. It performs a specific function, be it a computation or communication with an off-chip peripheral, and interacts with the rest of the system via the SIMPPL controller, which interfaces with the internal communication links. The SIMPPL Control Sequencer (SCS) module allows the designer to specify, or "program", how the PE is used in the SoC. It contains the sequence of instructions that are executed by the controller for a given application. The controller then manipulates the control bits of the PE based on the current instruction being executed by the controller and the status bits provided by the PE. Section 3.3.2 illustrates a programming example for the SCS.

The remainder of this chapter is divided into the following sections. Section 3.1 provides details on the underlying SIMPPL controller architecture and Section 3.2 outlines the additional functionality and hardware of the "debug" version of the controllers. Finally, the SIMPPL Controller Sequencer's interface and programming model are discussed in Section 3.3.

3.1 SIMPPL Controller

The SIMPPL controller acts as the physical interface of the IP core to the rest of the system. Its instruction set is designed to facilitate controlling the core's operations and reprogramming the core's use for different applications. Details on the controller's architecture and the instructions it supports are given below.

3.1.1 Controller Architecture

Figure 3.3 illustrates the SIMPPL controller's datapath architecture. The controller executes instructions received via both the internal receive (Rx) link and the SCS. Instructions from the Rx Link are sent by other CEs as a way to communicate control or status information from one CE to another CE, whereas instructions from the SCS implement local control. Instruction execution priority is determined by the value of the *Cont Prog* bit so that designers can vary priority of program instructions depending on how a CE is used in an application. If this status bit is high, then the "program" (SCS) instructions have the highest priority, otherwise the Rx link instructions have the highest priority. Since the user must be able to properly order the arrival of instructions to the controller from two sources, allowing multiple instructions in the execution pipeline greatly complicates the synchronization required to ensure that the correct execution order is achieved. Therefore, the SIMPPL controller is designed as a single-issue architecture, where only one instruction is in flight at a time, to reduce design complexity and to simplify program writing for the user. The SIMPPL controller also monitors the PE-specific status bits that are used to generate status bits for the SCS, which are used to determine the control flow of a program as discussed in Section 3.3.1.

The format of an output data packet sent via the internal transmit (Tx) link is dictated

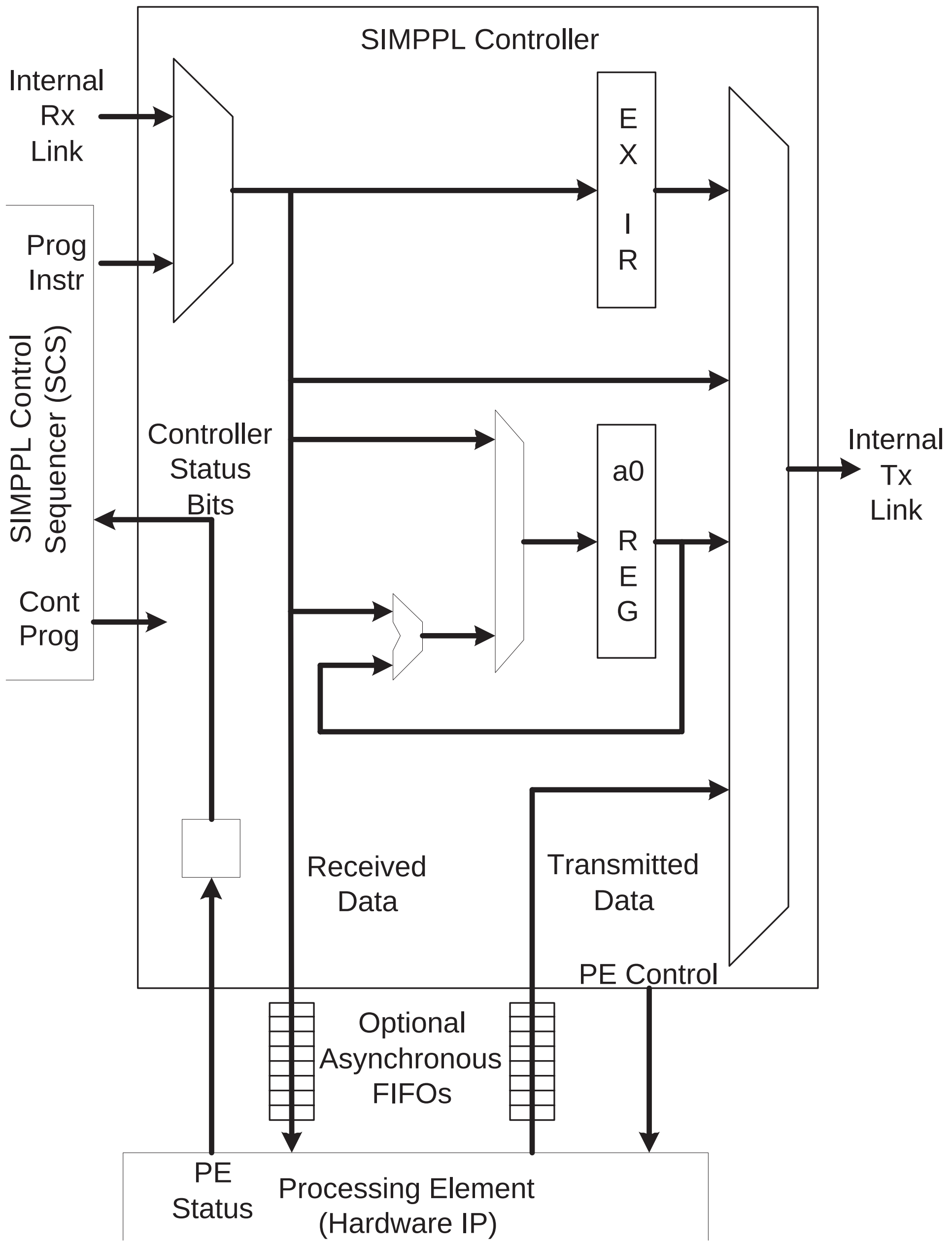

Figure 3.3: An overview of the SIMPPL controller datapath architecture.

by the instruction currently being executed. The inputs multiplexed to the Tx link are the Executing Instruction Register (EX IR), an immediate address that is required in some instructions, the address stored in the address register *a0* and any data that the hardware IP transmits. Data can only be received and transmitted via the internal links and cannot originate from the SCS. Furthermore, the controller can only send and receive discrete packets of data, which may not be sufficient for certain types of PEs requiring continuous data streaming. To solve this problem, the controller supports the use of optional asynchronous FIFOs to buffer the data transmissions between the controller and the PE. The designer can then clock the controller at a faster rate than the PE to guarantee that it accurately receives/transmits at the necessary data rate.

3.1.2 Controller Instruction Set

Although the current SIMPPL controller uses a 33-bit wide FIFO, the data word is only 32-bits. The remaining bit is used to indicate whether the transmitted word is an instruction or data. Figure 3.4 provides a description of the generic data packet structure transmitted over an internal link. The instruction word is divided into the least significant byte, which is designated for the opcode, and the upper 3 bytes, which represents the Number of Data Words (NDW) sent or received in a data transmission instruction. The current instruction set uses only the five Least Significant Bits (LSBs) of the opcode byte to represent the instruction. The remaining bits are reserved for future extensions of the controller instruction set.

Designers can choose to reduce the resource usage of SoCs using the SIMPPL model that do not require a 32-bit data word length or address space. If the width of the data word transmitted/received by a CE is less than 32-bits and the maximum number of data words, the NDW value, is less than 2^{23}, then the designer may choose to reduce the width of the FIFOs used as internal Rx and Tx links for that CE. For example, if the width of the data words being processed by a CE is 24-bits, the internal links can be 25-bits wide, where 24-bits are used for the data word and one bit is used as the control bit. The opcode of the instruction word would still be the eight LSBs, however, there would only be two bytes to represent the NDW value for the instruction, decreasing the packet size that could be received or transmitted by the CE.

All SIMPPL controller instruction packets have three components: (1) the instruction word; (2) the address or state word (optional); and (3) the data words (optional). The instruction set is divided into two groups; instructions that perform a control operation, and

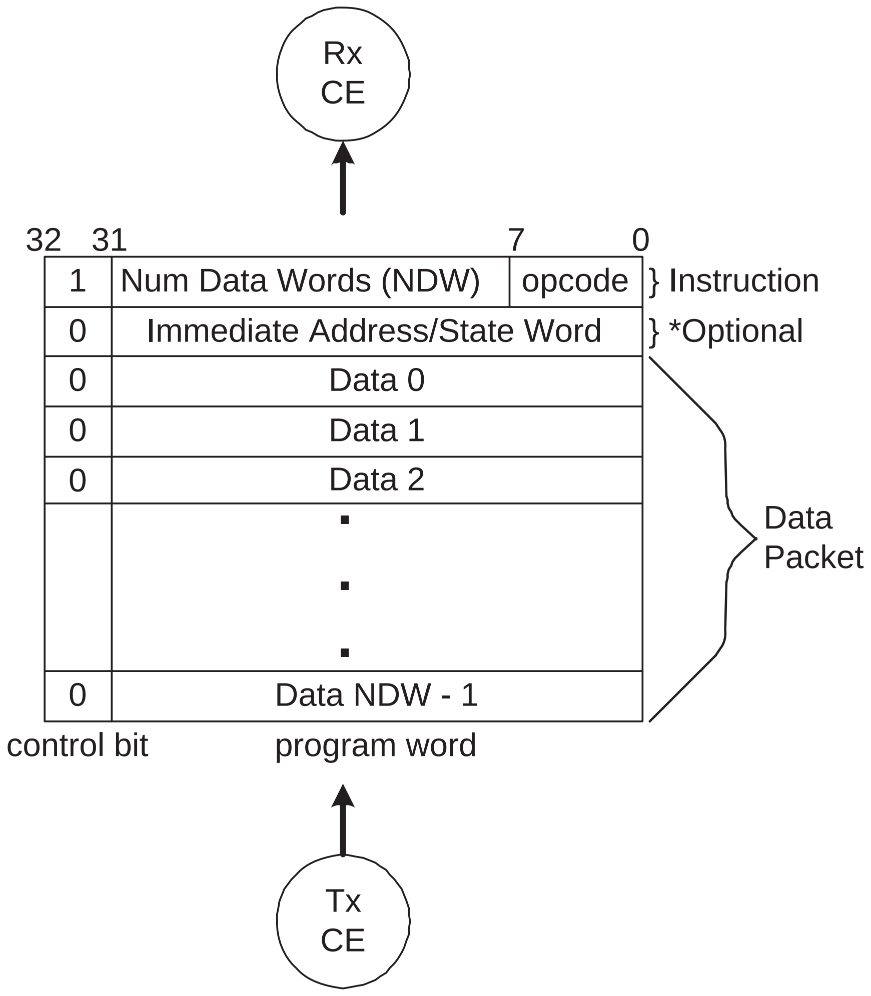

Figure 3.4: An internal link's data packet format.

those that transfer data. Instructions resulting in data transfers are further subdivided into three different categories: (1) read requests, (2) receives, and (3) writes. A read request is issued by the program of one CE and sent to another CE requesting that data be transmitted back to the original CE. A receive instruction must then be generated as the first transmitted word to accompany the data sent back to the initiating CE, since all transfers via internal links start with an instruction. Finally, the program can also use a write instruction to accompany data words transmitted to another CE.

Table 3.1 contains all the instructions currently supported by the SIMPPL controller and Appendix A.1 lists the instructions and their corresponding opcodes. The objective is to provide a minimal instruction set to reduce the size of the controller, while still providing sufficient programmability such that the cores can be easily reconfigured for any potential application. Although some instructions required to fully support the reconfigurability of

Table 3.1: The current instruction set supported by the SIMPPL controller.

Instruction Type	**Rd Req**	**Rx**	**Wr**	**Issue Instr**	**Exec. Instr**	**Addr Field**	**Data Field**
Imm. Data Transfer	X	X	X	S/R	S/R		X
Imm. Data + Imm. Addr.	X	X	X	S/R	S/R	X	X
Addr. Reg. Initialization			X	S	S	X	
Addr. Reg. Arithmetic			X	S	S		
Imm. Data + Indir. Addr.	X	X	X	S	S	X	X
Imm. Data + Autoinc.	X	X	X	S	S	X	X
Bypass				S/R	S/R		X
No-op				S	R		
Reset				S	R		

some types of hardware PEs may be missing, the instructions in Table 3.1 support the hardware CEs that have been built to date. Furthermore, the controller supports the expansion of the instruction set to meet future requirements.

The first column in Table 3.1 describes the operation being performed by the instruction. Columns 2 through 4 are used to indicate whether the different instruction types can be used to *request data* (Rd Req), *receive data* (Rx), or *write data* (Wr). The next two columns are used to denote whether each instruction may be issued from or executed from the *SCS* (S) or *internal Receive Communication Link* (R). Finally, the last two columns are used to denote whether the instruction requires an *address field* (Addr Field) or a *data field* (Data Field) in the packet transmission.

The first instruction type described in Table 3.1 is the immediate data transfer instruction. It consists of one instruction word of the format shown in Figure 3.4, excluding the address field, where the two LSBs of the opcode indicates whether the data transfer is a read request, a write, or a receive. The immediate data plus immediate address instruction is similar to the immediate data transfer instruction except that an address field is required as part of the instruction packet.

Instructions that use the *a0* register have a one or two-word format, but are not transmitted as they only make sense in the context of the local controller. The initialization of the local address register with an immediate value is a two word instruction, where the first contains the opcode and the second is the new address. The address register arithmetic

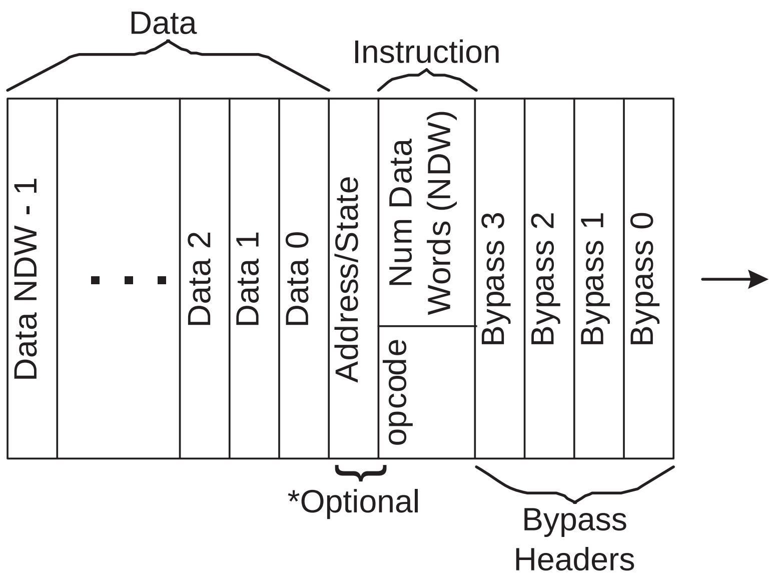

Figure 3.5: A Data packet with four bypass instructions.

instructions are single word instructions used to add or subtract an offset to the current local address register value. The value in the address register can provide the immediate address for any data transfer instructions sent to other CEs, using indirect addressing with an optional post-increment.

The remaining instructions provide control functionality for the controller. The *bypass* instruction allows a packet of data received from one CE to bypass the current CE, such that the bypass instruction header is removed and the enclosed instruction is forwarded without execution. Figure 3.5 illustrates a data packet that is encompassed within four bypass instructions. By prepending *N* bypass instructions to a data packet, the packet will bypass *N* controllers before the $N{+}1^{th}$ controller processes the actual data packet. The *no-op* instruction can be used in combination with SCS status bits to provide handshaking controls between CEs. This will be further discussed in Section 3.3.1. Finally, the *reset* instruction can be transmitted from the CE to reset the controller and PE of the receiving CE.

Designers can reduce the size of the controller by tailoring the instruction set to the PE. Although some CE's may receive and transmit data, thus requiring the full instruction set, others may only produce data or consume data. The Producer controller (Producer) is designed for CE's that only generate data. It does not support any instructions that may read data from a CE. The Consumer controller (Consumer) is designed for CEs that receive input data without generating output data. It does not support any instructions that try to

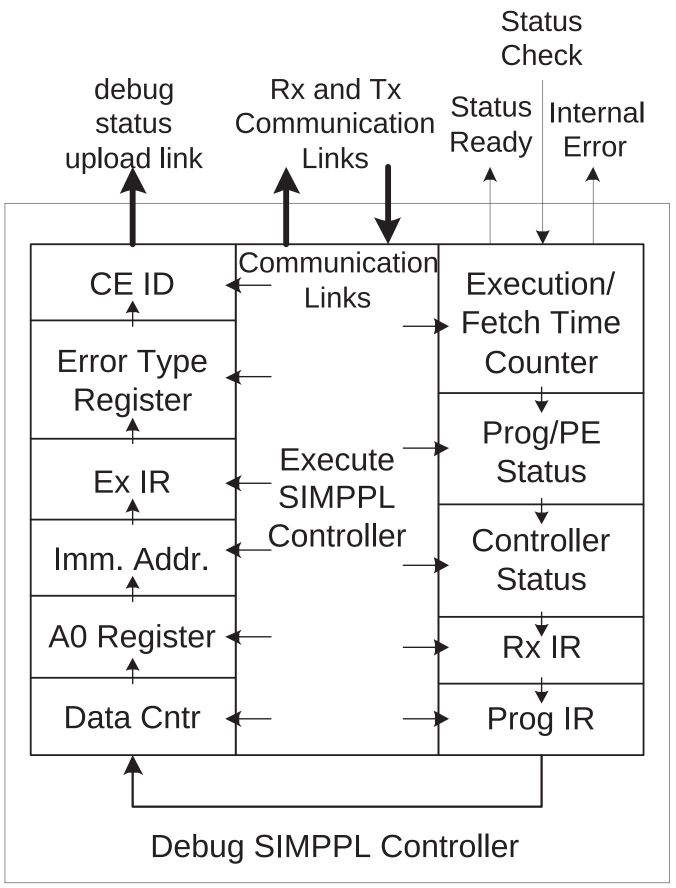

Figure 3.6: The SIMPPL debug controller architecture.

write PE data to a Tx link.

3.2 Debug Controllers

Here we introduce a debug SIMPPL controller (debug controller), based on the execute SIMPPL controller (execute controller) described in Section 3.1. This extension of the original architecture allows designers to detect low-level programming and integration errors for individual CEs.

3.2.1 Debug-Controller Architecture and Interface

Figure 3.6 shows the architecture of a debug controller, with the execute controller described in Section 3.1 forming the central component. While the execute controller has three states in the instruction execution state machine: fetch, decode, and execute, the de-

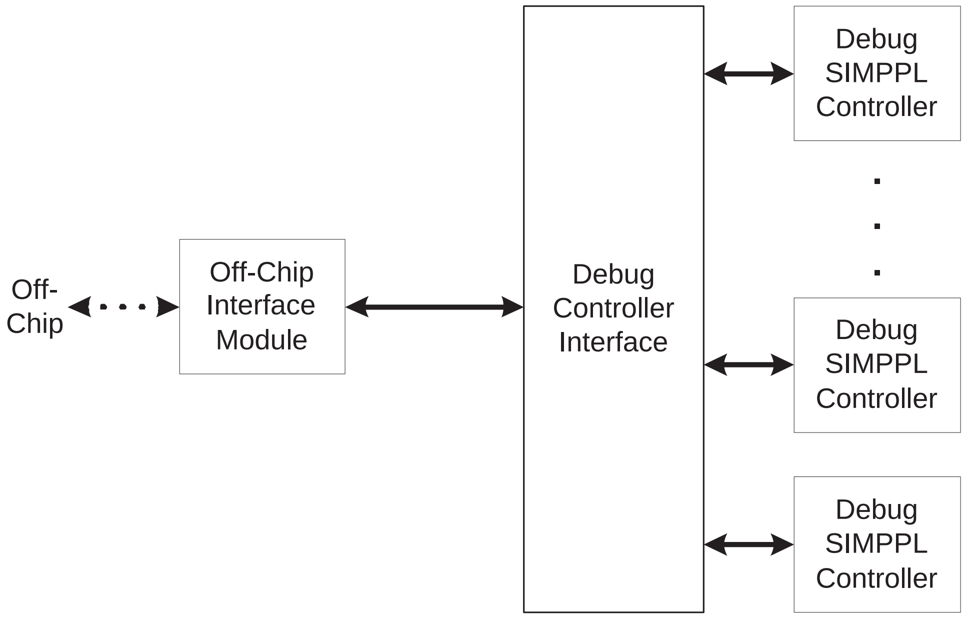

Figure 3.7: The SIMPPL debug controller interface.

bug controller has a fourth state – the *stall* state. An input signal (Status Check) has been added to the debug controller to allow designers to request a status check of the CE while the system is running. Additional output signals are used to indicate if a run-time error has occurred in the CE(*int_error*) and when the CE's status information is ready to be accessed (*status_ready*). The controller enters the stall state if an error occurs during the execution of an instruction or if a status check has been requested (*status_check*). The stall state allows the controller to upload all of the status information about the current executing instruction to the debug status upload link before executing the next instruction.

Eleven status registers have been added to the debug controller architecture, as shown in Figure 3.6, to store run-time status information about the CE. These include the CE's ID register, registers that store information about the instruction currently executing (Ex IR, Imm Addr, A0 register, Data Cntr, Execution/Fetch Time Counter), the current state of the CE (Error Type Register, Prog/PE Status, Controller Status), and the "next" instructions available from the program and from the receive link (Rx IR and Prog IR). The status registers are connected to form a large shift register to upload the values from the CE to the debug status upload link. The debug controller requires twelve cycles, or one cycle plus the number of status registers, in the stall state to upload all of the status information from the CE to the link, assuming the upload link is not full. Otherwise, the controller will remain in the stall state until all the status register values have been uploaded.

The debug status upload link is implemented as an additional Asynchronous FIFO link

Table 3.2: The current error cases detectable using the debug controller.

Error Case	Error Code	Error Type
Instruction word not in Fetch Cycle	8000 0001	Programming
Data word in Fetch Cycle	4000 0001	Programming
Execution Time Overflow	2000 0001	Programming
Fetch Time Overflow	1000 0001	Programming
Writing to a Full Tx Link	0800 0001	Integration
Reading from an Empty Rx Link	0400 0001	Integration
Writing data to the PE when it is not ready	0200 0001	Integration
Writing an address to the PE when it is not ready	0100 0001	Integration
Reading data from the PE when it is not ready	0080 0001	Integration
Executing an invalid instruction	0040 0001	Programming

that is used to upload the debugging information to the debug interface shown in Figure 3.7. The debug controller interface connects via a bus to an off-chip peripheral interface module that allows users to read the available status information off-chip from the controllers. The interface also contains a status register that indicates which CEs have status information available and what, if any, CEs have encountered run-time errors. Alternatively, if a debug controller is implemented in ASIC technology, the status information can be downloaded off-chip by implementing the registers using scannable flipflops.

3.2.2 Debugger Options and Detectable Errors

The debug controller supports two different run-time operations: error detection and status checks. When the *Status check* signal is set high for a clock cycle, it triggers the CE to upload status information after the execution of the current instruction completes. This allows the designer to check what instruction is being executed by a CE at random points of operation of the application. The *Status Check* can also be tied high for the duration of the profile period to obtain a continuously running profile of the CE, however, the CE will stall if the upload link becomes full.

Column 1 of Table 3.2 lists the error cases that the debug controller is currently able to detect, but the number of detectable error cases may be extended if a future need is determined. The second column in the table indicates the error code that is uploaded from the debug controller when an error occurs. The final column indicates whether an error case is the result of a programming error or a CE/system integration error.

3.3 SIMPPL Control Sequencer

The SIMPPL Control Sequencer provides the local program that specifies how the PE is to be used by the system. For example, a CE that has an audio sampling PE can be reprogrammed to generate packets of different formats depending on the requirements of the application. In this section, we discuss the SCS's architecture for both ASIC and FPGA platforms and provide a programming example. We then conclude with a discussion of how the CE abstraction allows a designer to dynamically generate program instructions, which we refer to as dynamic programming.

3.3.1 SCS Interface

The operation of a SIMPPL controller is analogous to a generic processor, where the controller's instruction set is akin to assembly language. For a processor, programs consist of a series of instructions used to perform the designed operations. Execution order is dictated by the processor's Program Counter (PC), which specifies the address of the next instruction of the program to be fetched from memory. While a SIMPPL controller and program perform the equivalent operations to a program running on a generic processor, the controller uses a remote PC in the SCS to select the next instruction to be fetched.

Figure 3.8 illustrates the SCS structure and its interface with the SIMPPL controller via six standardized signals. The 32-bit *program word* and the *program control bit*, which indicates if the program word is an instruction or address, are only valid when the *valid instruction* bit is high. The *valid instruction* signal is used by the SIMPPL controller in combination with the *program instruction read* to fetch an instruction from the Store Unit and update the PC. The *continue program* bit indicates whether the current program instruction has higher priority than the instructions received on the CE Rx link. It can be used in combination with PE-specific and controller *status bits* to help ensure the correct execution order of instructions.

For example, if the SCS has a status bit that indicates when the controller is executing an instruction from a Rx Link (*exec_rx_instr*), it can be used to stall the CE until it has received a packet from an adjacent CE. To perform this handshaking, the SCS program initially stalls the controller by setting the *valid instruction* bit low. When the controller

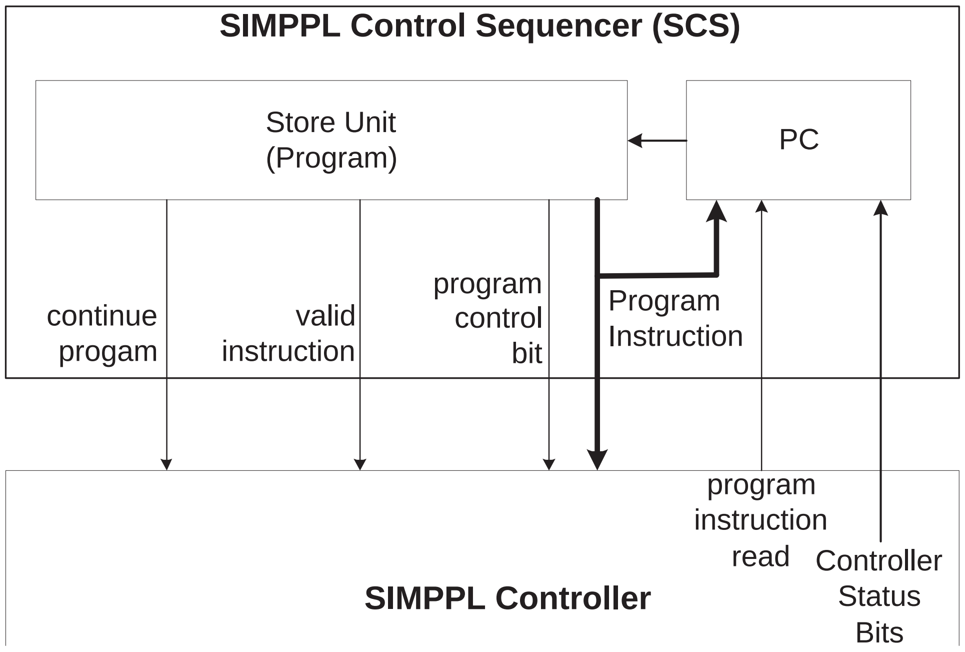

Figure 3.8: The standard SIMPPL control sequencer structure and interface to the SIMPPL controller.

receives an instruction on the Rx Link, it acts as a request signal and the *exec_rx_instr* will go high. In response to this request, the SCS's *valid instruction* signal then goes high along with the *continue program* so that the next instruction executed by the controller is an SCS instruction to acknowledge the received request.

Although a PC is traditionally implemented as a counter, the SCS's remote PC can also be constructed as a Finite State Machine (FSM). This allows branches to be executed implicitly as transitions in the PC's FSM depending on the control and status signal values. The PC FSM is application-specific and uses the current PC and status bit values to generate the correct index to the store unit to select the correct instruction to be fetched and sent to the controller. This reduces the size of both the SIMPPL controller and the program located in the store unit by eliminating the need for branch instructions in the instruction set. Furthermore, it reduces the performance overhead of using the SIMPPL controller as an interface since it does not have to execute conditional or explicit branch instructions.

If an SoC is implemented on an FPGA, the designer can choose to implement the program's store unit in an on-chip memory. Yet many CEs only require small SCSs for an application, thus the instructions can be stored as a separate FSM. When an SoC is implemented as an ASIC, the designer could choose to design each SCS for its specific application by instantiating a small memory for the Store Unit and then implementing

```
write start addr to a0;
for (i=0; i< 1024; i++)
{
   while (!valid_sensor_data);
   write 8 data words starting at addr (a0);
   a0 = a0 + 8;
}
```

Figure 3.9: Pseudocode for the sensor unit's SCS program.

the PC as application-specific dedicated logic. However, one of the benefits of the CE abstraction is that it decouples the control from the datapath to support programmability. Hardwiring the PC means that the designer cannot alter the CE's program post-fabrication. To allow post-fabrication programmability, ASIC designers can implement a small memory for the instruction words and a small region of programmable fabric that enables designers to change the PC to support a variety of SCS programs for the CE. The following example demonstrates how to write a program and use the SIMPPL controller interface.

3.3.2 Static Programming Example

Assume a hardware system that consists of two PEs: 1) a memory, and 2) a sensor unit used to measure multiple environmental quantities at set time intervals. The total storage requirements for each set of measurements is 32 bytes (eight data words) and the memory is large enough to store 1024 samples. The user wants to store the first 1024 samples to experimentally measure when the environmental system reaches steady state before deciding how often to record samples and upload the results to a host PC. The sensor unit has a status bit, *valid_sensor_data*, that indicates when a set of measurements is available for reading. The sensor unit's SIMPPL controller passes the status information to its SCS to indicate that data is available for transmission to the memory unit. The pseudocode for the sensor unit's SCS program is given in Figure 3.9. At present, we do not have compiler support for the SIMPPL controller and all programs (SCSs) are hand generated. Figure 3.10 illustrates pseudo-HDL implementations of the sensor CE's Program Counter FSM and the *valid_instruction* signal that dictate the program instruction and if it is available to be fetched by the SIMPPL controller using the *prog_instr_read* signal.

The PC requires four states to implement the pseudocode in Figure 3.9 and the PC state only changes after an instruction has been read or all 1024 samples have been written to memory. The first two states, *Write a0* state and *Write address* state, write the starting

```
if (rst=1) {
  PCstate <= Write a0 state;
else
  PCstate <= nextPC;
}

//Next-state state machine for the PC:
case (PCstate) {
    Write a0 state:  //Instruction to initialize a0
        if ((prog_instr_read) && (rst=0))
          nextPC = Write address state;
        else
          nextPC = Write a0 state;
    Write address state: //New address for a0
        if (prog_instr_read)
          nextPC = Write autoinc state;
        else
          nextPC = Write address state;
    Write autoinc state: //Write data to (a0)+
        if ((prog_instr_read) && (SampleCntr=1024))
          nextPC = Done state;
        else
          nextPC = Write autoinc state;
    Done state:
      nextPC = Done state;
  }


/*Used to indicate when the instruction is valid.
 *Stalls the processor when there is no valid
 *instruction. */
case (PCstate) {
  Write a0 state:
    valid_instruction = 1;
  Write address state:
    valid_instruction = 1;
  Write autoinc state:
    valid_instruction = valid_sensor_data;
  Done state:
    valid_instruction = 0;
}
```

Figure 3.10: Pseudo-HDL code to implement the state machine for the sensor unit's program counter and the valid instruction signal.

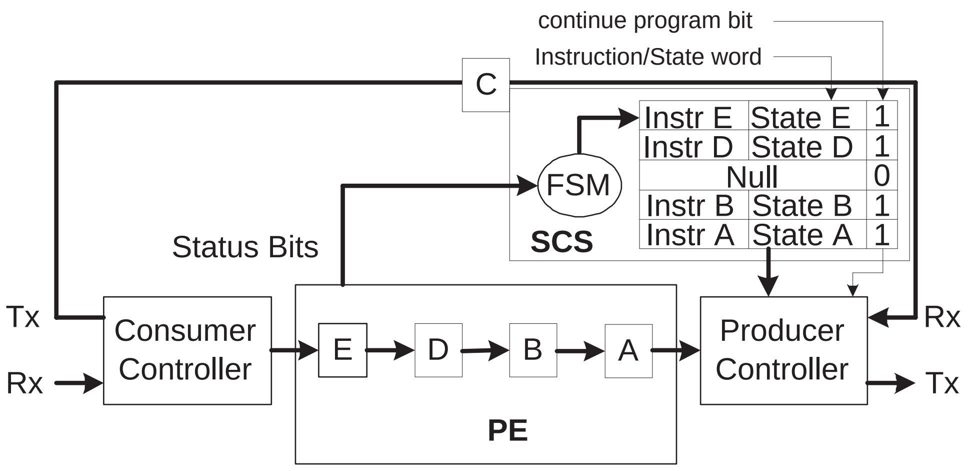

Figure 3.11: A CE with multiple packets of data in flight.

address of the memory unit to the *a0* register. The third state (*Write autoinc* state) writes eight data words to the memory unit starting at address *(a0)* and then post-increments *a0* by eight. While the *valid instruction* signal is high during the first two states to initialize the address register, it is assigned the value of the *valid_sensor_data* status bit in the *Write autoinc* state because the data write instruction should only occur when the sensor has new data to transmit to the memory. A separate counter state machine(*SampleCntr*), not shown in Figure 3.10, is used to count the number of times the sensor unit measurements are sent to the memory unit. When the *SampleCntr* equals 1024, the program has completed so the PC goes to the *Done* state, where no further instructions are executed, and the *valid_instruction* signal goes low permanently.

3.3.3 Dynamic Programming Architecture

For some applications, a designer may wish to have a CE support multiple processing operations that are data packet dependent. If the CE is pipelined with independent Producer and Consumer controllers for the PE, then the Consumer may receive a variety of instruction packets that should result in the Producer generating different instruction packets depending on the received data. The following example demonstrates how the Consumer and Producer controllers can work together to correctly process the received instruction packets and generate the appropriate output instruction packets, even in the presence of bypass instructions.

Figure 3.11 illustrates a CE that receives packets *A* through *E* in order, where packet *C* is to bypass the PE entirely, and generates the appropriate program instructions for the

Producer's SCS. For the purpose of this example, the Consumer does not have an SCS and the order of packets received by a CE must be maintained when they are transmitted to the subsequent CE. Therefore, it is imperative that data packets *A* and *B*, which were inflight when packet *C* arrived, are transmitted first. To enable this functionality, the instructions from the Producer's Rx Communication Link and those created in the Producer's SCS have variable processing priority determined by the value of the *continue program* status bit. When the *continue program* status bit is set, the controller continues to fetch available instructions from the SCS, even if there are data packets to be processed on the receive link. Therefore, each Producer's SCS uses a 35-bit wide FIFO to store the instruction word, the control bit, the valid instruction bit and the *continue program* bit as well. The FIFO acts as the Store Unit where the maximum depth is equal to the maximum number of data packets that can be processed concurrently. The PE enqueues valid instructions into the FIFO for every data packet in flight, setting the *continue program* bit for each instruction, as indicated in Figure 3.11.

To ensure that bypassed packets are transmitted in the proper order, the PE must detect if the Consumer receives a *bypass* instruction. In this situation, the PE will queue a null instruction into the FIFO with the *continue program* and *valid instruction* bits set low, as shown in Figure 3.11. To guarantee that instructions are enqueued in the Producer's FIFO in the correct order, the SCS state machine must push the correct instruction onto the FIFO before the Consumer controller finishes reading the current data packet. The Producer will then dequeue the instructions and transmit the data packets in order. When the "null" instruction is detected with the *continue program* and valid_instruction bits set low, the Rx Communication Link will be given priority. The bypassed packet will then be retransmitted by the Producer to the subsequent CE and the "null" instruction will be dequeued from the FIFO.

Thus, for the example shown in Figure 3.11, the Producer will transmit packets *A* and *B* from the PE. It will then detect a "null" instruction, with the *continue program* bit set low, and process packet *C* from the bypass link, while simultaneously dequeuing the "null" instruction. This will be followed by packets *D* and *E* being sent to the next CE.

3.4 Summary

The Computing Element abstraction simplifies the reuse of software and hardware IP by isolating the functionality of the IP, i.e. the Processing Element (PE), from the system-level communication and control. This is particularly important for hardware reuse, where redesign can be extremely costly. The SIMPPL Controller has been created to facilitate the reuse of hardware PEs. It provides a fixed physical interface to the PE as well as a fixed set of communication protocols for transmitting data among the CEs. Although the controller's underlying architecture is optimized to act as a PE's system interface, it can also be extended to provide runtime debugging capabilities for verification of the PE and its integration with the controller. CE's use a SIMPPL Control Sequencer (SCS) to store the local program that dictates how the PE will be used for the given application. The SCS allows both static and dynamic programming models to increase the flexibility of the CE abstraction and facilitate the reuse of CEs over a variety of applications.

Chapter 4

Implementing SIMPPL SoCs

To investigate the usage of a programmable controller interface for IP modules, three SoCs are created using the SIMPPL framework. All three of the SoCs are implemented on a Xilinx Multimedia board. The board's resources include a Virtex II 2000, five ZBT memory banks, a YCrCb video decoder that runs at 27MHz, and an RGB video DAC operating at 25 MHz. Section 4.1 of this chapter describes the nature of the three applications and the effects of using SIMPPL on system design time. Next, an overview of the CEs used to implement each application is given in Section 4.2. The chapter then concludes with a detailed examination of the effects of using the SIMPPL framework on a non-trivial SoC design in Section 4.3.

4.1 SIMPPL SoC Applications

Figure 4.1 illustrates the system level connections for two video-based systems. The first is a video streaming system, which does not include the Switch CE. Instead, it uses one of the two memory banks to buffer the video feed from the video camera while the other bank is displayed using the video DAC driving an SVGA monitor. The second system is a video snapshot system, which includes the Switch CE and only allows the user to update the SVGA display with a new image when the switch is toggled. The Vid_In CE interfaces with a video decoder to read in data in YCrCb format and then convert it to RGB format. The Vid_Out CE receives data in RGB format and transmits it to a video DAC used to drive an SVGA monitor. These CEs, in combination with two external memory banks controlled by the Mem CE, are used to implement a video streaming and a video snapshot application.

The video recorder and video display need to be synchronized because the system may

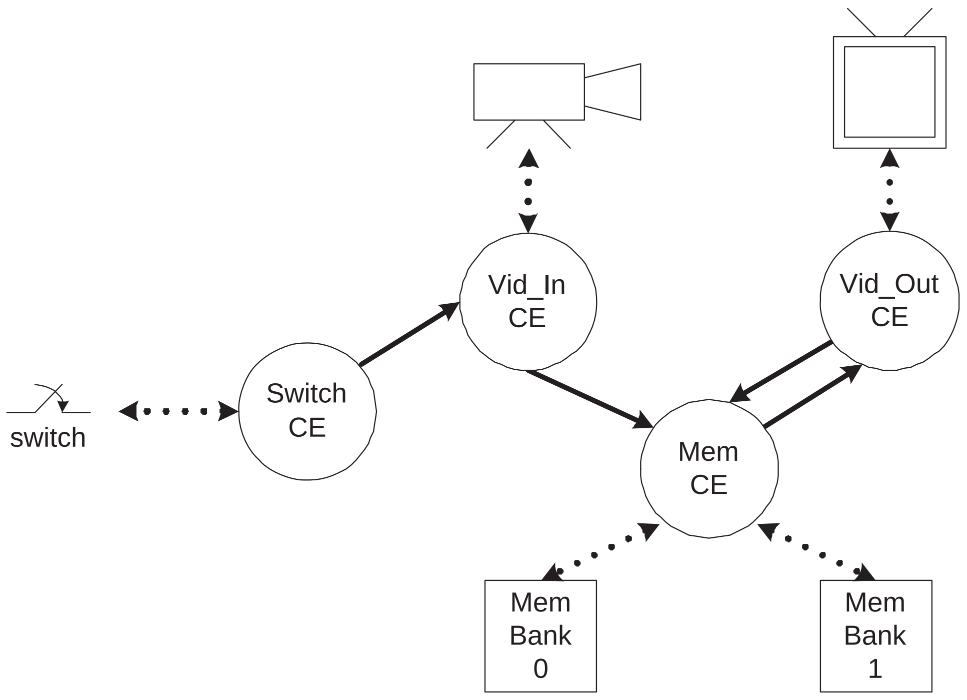

Figure 4.1: The SIMPPL model for the video streaming and snapshot applications.

come out of reset when the video recorder is mid-frame. Although the video applications require synchronization between the Vid_In CE and Vid_Out CE to properly display the video camera images, they do not communicate directly. Since the user is able to write individual programs to control the operation of the Vid_In, Vid_Out, and Mem CEs, there are multiple ways to implement this system. The straightforward approach is to have the Vid_In and Vid_Out CEs become active as soon as the system comes out of reset, and have the Mem CE only execute the memory reads and writes requested via the internal links from the Video CEs. However, this would not guarantee synchronization between the video data being received and the video data written to the SVGA. Therefore, to achieve synchronization between the two Video CEs, the Vid_In CE starts running as soon as the system comes out of reset and the Vid_Out CE stalls, waiting for an indication that the Vid_In CE has started writing a new frame to the Mem CE.

Another significant design challenge for the video systems is the different operating frequencies of the CEs. Fortunately, the CE abstraction and asynchronous FIFO communication links effectively isolate the different clock domains to simplify their integration and synchronization. For instance, the Vid_In and Input Switch CEs operate at 27 MHz, however, the Mem CE operates at 54 MHz. Furthermore, the Vid_Out CE uses an asynchronous FIFO interface between its PE and controller so that the controller can run at 50

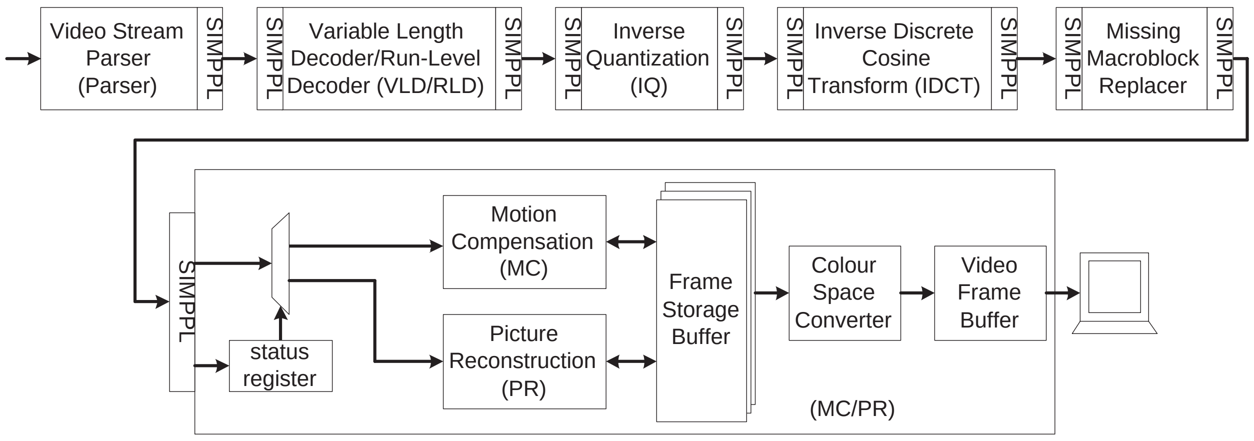
Video Stream Parser (Parser)
SIMPPL
Variable Length Decoder/Run-Level Decoder (VLD/RLD)
Inverse Quantization (IQ)
Inverse Discrete Cosine Transform (IDCT)
Missing Macroblock Replacer
Motion Compensation (MC)
Picture Reconstruction (PR)
status register
Frame Storage Buffer
Colour Space Converter
Video Frame Buffer
(MC/PR)

Figure 4.2: The SIMPPL model for an MPEG-1 video decoder.

Table 4.1: Table of the System Integration times for SoCs.

SoC Design	**System Integration Time**
Custom Streaming Video System	140 hours
SIMPPL Streaming Video System	4.5 hours
SIMPPL Snapshot Video System	1.5 hours
SIMPPL MPEG-1 Video Decoder System	18 hours

MHz to guarantee that valid data will be available for the PE, which runs at 25 MHz to match the video DAC's operating frequency.

Figure 4.2 illustrates the third system designed using the SIMPPL model. It is an MPEG-1 video decoder that runs at 30 frames per second, generating 320 by 240 pixel images on an SVGA monitor. The decoder was designed and implemented by four graduate students as a course project [9]. The synchronization challenge for this system is to maintain the order of packets processed in the system while ensuring that certain instruction packets are only processed by selected CEs. Recalling the discussion in Section 3.3.3 and the CE architecture in Figure 3.11, the bypass instruction allows such packets to bypass processing by a CE, but the *continue program* status bit can be used to ensure that the bypassed packet maintains its position in the data stream.

4.1.1 SIMPPL SoC Implementation Statistics

Table 4.1 summarizes the time required to integrate the CEs and create the SCSs for the systems shown in Figures 4.1 and 4.2. Before the SIMPPL model was defined, a novice designer created a custom version of the video streaming application. The student found it difficult to create the proper system-level control due to the multiple clock domains and synchronization requirements. After some redesign, the modules were reused and integrated with SIMPPL controllers to create the Vid_In, Vid_Out, and Mem CEs (Figure 4.1), which required approximately 40 hours. However, the integration of the CEs and the design of their respective SCSs took only 4.5 hours for the SIMPPL Streaming Video SoC, which is less than 3.5% of the time required to implement the system-level integration for the custom design. The CE abstraction simplified the system-level integration by isolating the different clock domains, which greatly reduced integration time. The SIMPPL Snapshot Video system only required the addition of the Input Switch CE and minor adjustments to the SCSs previously used in the streaming video system, reducing the system integration time to 1.5 hours. Thus, not only does the SIMPPL framework reduce system integration time, but it also facilitates the reuse of CEs for new applications.

Recalling that the SIMPPL MPEG-1 Video Decoder System was built by a four person design team, it took 18 person-hours to properly connect all the CEs and to generate the appropriate SCSs. Integrating all the MPEG-1 hardware PEs with Producer and Consumer controllers required an additional 39 person-hours, or 2.4%, of the total system design time of 1607 person-hours. For complex designs, system integration can be a significant portion of the total design time, however, the SIMPPL framework limits the system integration for the MPEG-1 Video Decoder to 1.1% of the total design time. Furthermore, the CE abstraction hides the implementation details of the CE from the rest of the system so that changes to the PE do not necessitate redesign at the system-level. For example, the Video Stream Parser CE is currently implemented as a software CE on a MicroBlaze, due to design time constraints. However, the fixed communication links allow it to be swapped out in favour of a hardware CE implementation in the future without any changes to the rest of the system.

4.2 CE Implementations

This section describes the implementation statistics for the controllers implemented on FPGAs and ASICs, possible different CE architectures, and the different CEs that have been created and tested to date on an FPGA for the SoCs described in Section 4.1.

Table 4.2: SIMPPL Controller implementation statistics.

Controller Type	FPGA platform			ASIC platform–Area		ASIC platform–Speed	
	Area		Max. Freq. (MHz)	Area ($10^3 um^2$)	Max. Freq. (MHz)	Area (10^3 um^2)	Max. Freq. (GHz)
	LUTs	Flipflops					
Consumer Execute	277	117	287	5.25	183	12.16	1.59
Producer Execute	355	125	285	5.42	184	13.16	1.56
Full Execute	346	115	283	5.49	183	13.71	1.59
Consumer Debug	1002	477	180	19.17	165	29.62	1.24
Producer Debug	955	478	199	19.24	164	28.01	1.09
Full Debug	946	478	185	19.48	166	29.59	1.09

4.2.1 Controller Implementation Statistics

Table 4.2 summarizes the area and operating frequency measurements obtained for the different types of SIMPPL controllers implemented on both FPGA and ASIC platforms. The ASIC measurements are obtained using Synopsys synthesis tools for a 90 nm standard cell process. The *ASIC platform–Area* values are minimized for area and the operating frequency is left unconstrained, whereas the *ASIC platform–Speed* values minimize the operating frequency and leave the area unconstrained. To obtain comparable operating frequency measurements on an FPGA, the Virtex4 LX 40 -12 is used since it is the highest speed grade device fabricated in a 90 nm technology available from Xilinx. The FPGA measurements are generated using Xilinx's Place and Route tool from the ISE tool suite version 7.1.4.

Column 1 lists all of the different types of debug and execute SIMPPL controllers. Although the regularity of the controller's architecture can allow them to be autogenerated, the Consumer and Producer controllers are currently hand tailored from the Full controller. Columns 2 through 4 report the resource usage and maximum operating frequency for the controllers on the FPGA platform. All of the execute controllers achieved at least a 280 MHz operating frequency, and the Producer controller requires the most logic resources: 355 LUTs and 125 flipflops. Although it may seen peculiar that the Producer would require more resources than the Full controlle given its reduced instruction set, these controllers have been optimized for speed and not area. Therefore, the tools replicate logic to reduce the delay along the critical path, while using additional resources.

The debug controllers have a reduced operating frequency of 180 to 199 MHz while utilizing 945 to 1002 LUTs and 478 flipflops. The additional flipflops used in the debug controllers are attributed mainly to the eleven 32-bit debug status registers used to upload run-time information from the CE. The extra LUTs are required for implementing the multiplexing and shift logic for the status registers along with the error detection and uploading functionality. However, when designing on an FPGA, designers may choose to instantiate CEs with debug controllers to verify functionality on-chip and then use execute controllers for the final implementation to free up resources or increase the operating frequency if necessary.

The fifth and sixth columns in Table 4.2 report ASIC synthesis results for the controllers when they are optimized for area. While these implementations of the controller occupy a minimal area, the maximum operating frequency of 183 MHz is comparable to the worst operating frequency on the FPGA of 180 MHz. However, the final column demonstrates that the debug controllers can achieve a minimum operating frequency of 1 GHz when optimized for speed. This requires an approximate increase in area of 50% from the debug controllers optimized for area. The size increase for the execute controllers optimized for speed is approximately 2.5 times that of the area of the execute controllers optimized for area.

The results shown in Table 4.2 demonstrate that the critical path delay for the FPGA implementations is approximately 5.5 times greater than the speed-optimized ASIC implementations with the exception of the Consumer Debug controller that has an increased delay of 6.9 times that of the speed optimized ASIC implementation. This concurs with recent research that found the critical path delay on FPGA implementations to be three to four times that on ASICs [33]. The operating frequency of the speed-optimized SIMPPL controllers is likely fast enough for most applications, however, there is also the consideration that the area overhead of using these controllers is not significant.

The MIPS core is on the order of 10mm^2 in 90nm technology according to industry sources. The maximum number of Consumer debug controllers, the largest of the SIMPPL controllers, that can be packed into 10mm^2 is 337. While most current SoC designs would probably have significantly fewer CEs, the area of the controllers could be reduced and the operating frequency increased if the designer used a fully custom version of the controller. Previous work indicates that the maximum operating frequency should increase by a factor of 6 to 8 times [34] that of the standard cell implementation and the full custom layout area

Table 4.3: Execution overhead clock cycles for the Consumer, Producer, and Full SIMPPL Controllers.

Measured Quantity	**Producer Controller**	**Consumer Controller**	**Full Controller**
Instr. Fetch Overhead	1 cycle	1 cycle	1 cycle
Instr. Decode Overhead	1 cycle	1 cycle	1 cycle
Instr. Execute Overhead	2 cycles	4 cycles	4 cycle
Total Overhead	4 cycle	6 cycles	6 cycles

could be reduced to 6.9% that of a standard cell version [35].

Table 4.3 summarizes the execution overhead for the different versions of the controllers. One clock cycle is required to fetch the instruction from either the SCS or a Rx Link and another clock cycle is required to decode the instruction. The Execution-stage clock-cycle overhead is dependent on the current instruction being executed, thus the maximum execution-stage clock-cycle overhead for each controller is dependent on the instruction set supported by that controller. The Producer's maximum overhead of two clock-cycles occurs when a Write Immediate address plus Autoincrement instruction is issued. Four clock-cycles is the maximum execution overhead incurred by both the Full and Consumer Controllers when a Read Request plus Autoincrement instruction is issued.

The total instruction execution overhead of the SIMPPL controller ranges from a maximum of four clock cycles for the Producer Controller and six clock cycles for the Full and Consumer Controllers. Additional clock cycles of overhead may be incurred depending on the nature of the PE, ranging from one clock-cycle of overhead for buffering data transferred between the controller and the PE to multiple cycles for resource arbitration. However, depending on the nature of the PE, status bits may be used to provide early warning of the availability or need for data, allowing the designer to hide some of the overhead incurred by the controller during the PE processing to decrease the effective latency attributed to passing data packets between CEs. For example, an early warning signal is used by the Vid_Out CE to request data to ensure it is available to write to the display. It masks the total controller latency overhead of six cycles with no impact on the functionality or performance of the PE.

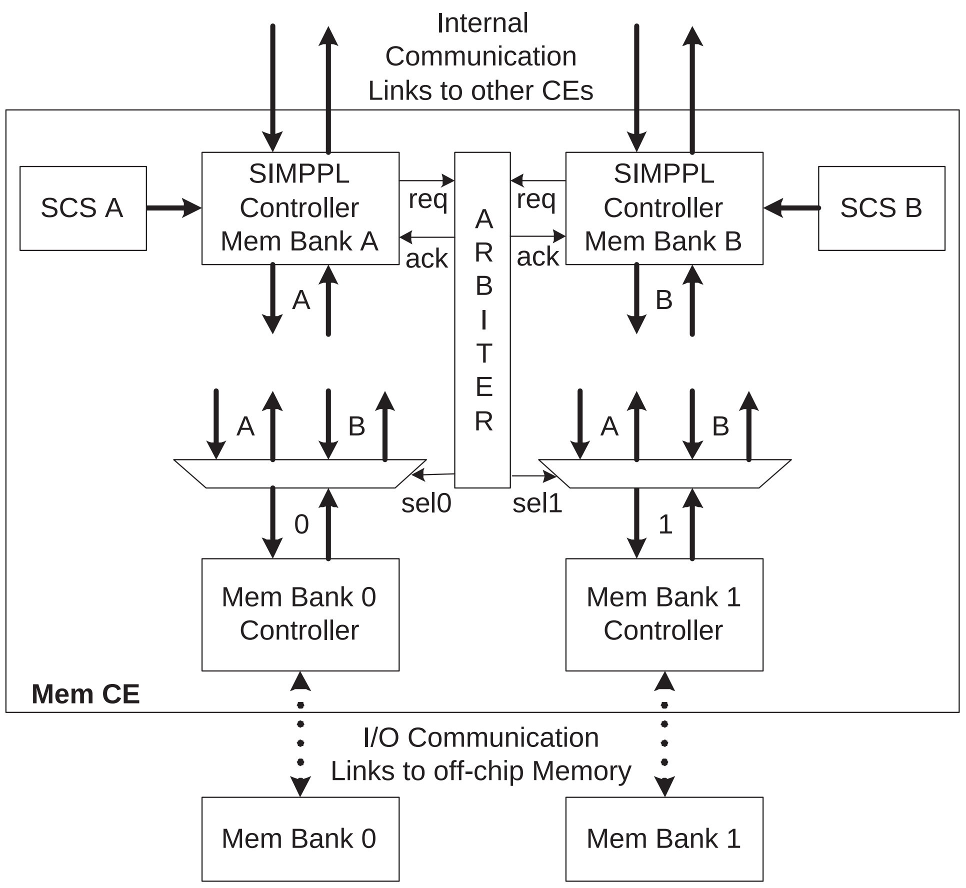
Internal Communication Links to other CEs
SCS A
SIMPPL Controller Mem Bank A
req
ack
A R B I T E R
req
ack
SIMPPL Controller Mem Bank B
SCS B
A
B
A
B
A
B
sel0
sel1
0
1
Mem Bank 0 Controller
Mem Bank 1 Controller
Mem CE
I/O Communication Links to off-chip Memory
Mem Bank 0
Mem Bank 1

Figure 4.3: The shared Computing Element architecture for a shared memory CE.

4.2.2 CE Architectures

To date, we have investigated three different hardware CE architectures. The first architecture, called the *Basic* architecture, is a direct implementation of the hardware CE abstraction shown in Figure 3.2. It is used for PEs that do not support the independent data transfers and have only one Rx and one Tx link. Examples of CEs with the Basic architecture are the Vid_In, Vid_Out, and Switch CEs. Figure 4.3 illustrates a second hardware CE architecture, the *Shared* architecture, which is designed to support parallel access to shared PE resources. For example, the shared architecture is useful for implementing a shared memory CE with two memory banks. It uses two controllers and an arbiter to determine which controller has access to which PE, in this case the Memory Bank Controllers.

The arbiter module interfaces with both SIMPPL controllers to service requests for memory bank accesses and to acknowledge that control of a memory bank has been granted. It generates the select signals used to multiplex the I/O signals from the two SIMPPL controllers to each of the two memory bank controllers. The arbiter is designed

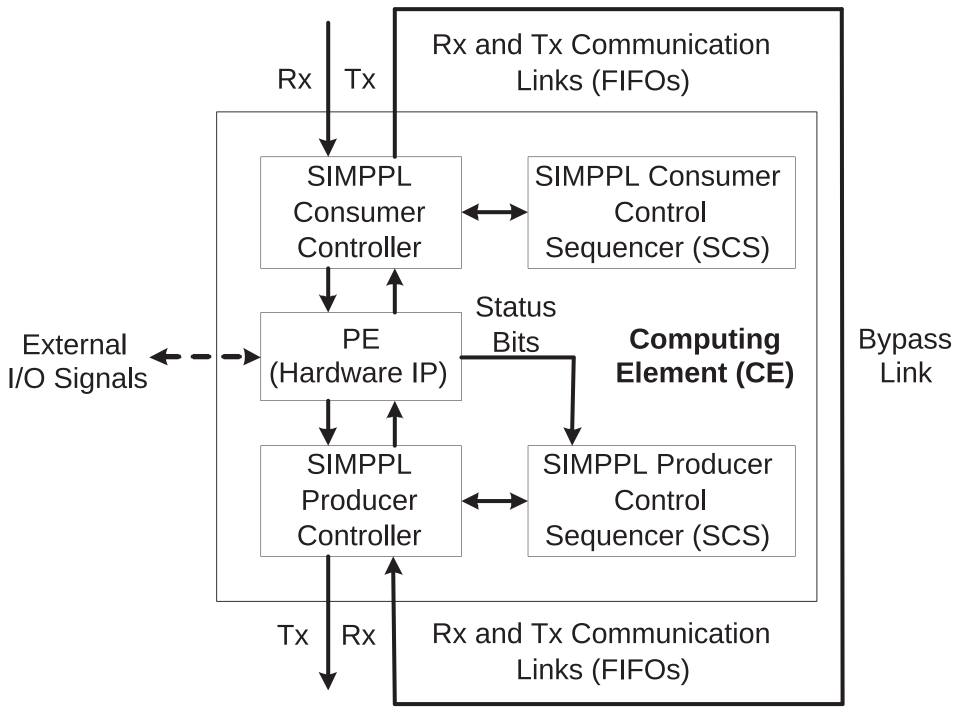
Rx
Tx
Rx and Tx Communication Links (FIFOs)
SIMPPL Consumer Controller
SIMPPL Consumer Control Sequencer (SCS)
Status Bits
External I/O Signals
PE (Hardware IP)
Computing Element (CE)
Bypass Link
SIMPPL Producer Controller
SIMPPL Producer Control Sequencer (SCS)
Tx
Rx
Rx and Tx Communication Links (FIFOs)

Figure 4.4: The pipelined Computing Element architecture.

as a separate module so that the user can adapt the arbiter to suit different applications.

Finally, Figure 4.4 shows a block diagram of the *Pipelined* architecture utilized for the CEs in the MPEG-1 video decoder application. All of the PEs in an MPEG-1 application are implemented in a pipelined format to allow multiple data packets to be processed concurrently Therefore, each PE has independent input and output Consumer and Producer SIMPPL controllers respectively, where each controller has its own SIMPPL Control Sequencer (SCS). Using independent controllers for receiving and transmitting data allows the Consumer to receive a new data packet for processing while the Producer transmits a packet to the adjacent CE.

4.2.3 CE Implementation Statistics

To investigate the CE architectures we described in the previous section and to demonstrate the benefits of the SIMPPL model, we implemented the three different SoCs described in Section 4.1 on an FPGA using the nine hardware CEs described in Table 4.4. All the results reported in the table were obtained using version 7.1.4 of the Xilinx ISE tools.[1] The first

[1]The LUTs and Flipflops resource usage of the Mem CE were reported using version 6.2.2 of the ISE tools as a bug in the current version of the tools causes an error during synthesis.

Table 4.4: Implemented CEs.

CE Name	**Architecture Type**	**Controllers**	**PE Resources**	
			LUTs	**Flipflops**
Input Switch CE	Basic	Producer	0	0
Vid_In CE	Basic	Producer	128	211
Vid_Out CE	Basic	Consumer	96	52
Mem CE	Shared	Full Full	187	148
VLD/RLD CE	Pipelined	Consumer Producer	606	699
IQ CE	Pipelined	Consumer Producer	429	201
IDCT CE	Pipelined	Consumer Producer	1091	1217
MMR CE	Pipelined	Consumer Producer	141	152
MC/PR CE	Basic	Consumer	1705	742

column provides the names of the CEs that have been designed. Column 2 describes which of the three architectures described in the previous section has been used to implement the CE. The third column lists which controller type(s) are used in the CE and the final two columns give the number of LUTs and Flipflops used to implement the PEs.

The Input Switch CE has no PE resources because the logic value on the switch is provided directly to the SCS as a status bit. However, the Vid_In and Vid_Out PEs read and write data to off-chip peripherals. The Vid_In PE also performs a 4-stage pipelined conversion of the YCrCb input to RGB format that makes it larger than the Vid_Out PE. The Mem PE comprises the two Memory Bank Controllers and the arbiter.

The remaining hardware CEs are: a Variable Length Decoder/Run-Level Decoder (VLD/RLD CE), an Inverse Quantizer (IQ CE), an Inverse Discrete Cosine Transform (IDCT CE), a Missing Macroblock Replacer (MMR CE), and a Motion Control/Picture Reconstruction (MC/PR CE). These hardware CEs are used to implement an MPEG-1 video decoder. The range in PE resource usage is due to the varied complexity of the PEs

Table 4.5: Table of the resource usage of the individual modules and total system.

Module Name	**Number of LUTs (PE/SCS)**	**Number of Flipflops (PE/SCS)**	**Number of PE BRAMs/ Multipliers**	**Consumer (LUTs/ Flipflops)**	**Producer (LUTs/ Flipflops)**	**% Overhead (LUTs/ Flipflops)**
VLD/RLD CE	606/92	699/98	9/0	119/70	217/42	71/30
IQ CE	429/86	201/13	2/2	126/70	302/106	120/94
IDCT CE	1091/67	1217/24	3/16	126/70	302/106	45/16
MMR CE	141/47	152/32	0/0	115/69	217/42	269/94
MC/PR CE	1705/0	742/0	2/5	115/69	0/0	7/9
Total System	7248/292	4118/167	16/23	601/348	1038/296	27/20

being implemented in the decoder.

4.3 Detailed Analysis of an SoC Implementation

This section provides the implementation statistics for the resource usage and the design time of the MPEG-1 video decoder running at 30 frames per second to generate 320 by 200 pixel images on a monitor using the Xilinx Multimedia Board's Virtex 2V2000. A four-person design team, initially unfamiliar with the model, implemented the MPEG-1 video decoder as shown in Figure 4.2 using SIMPPL to provide an independent evaluation of the proposed SIMPPL framework [9]. The design team partitioned the video decoder into PEs before selecting this system-level architecture. While the PE partitions are not made to ensure reusability, they are still readily adaptable to CEs. All the video decoder CEs are implemented in hardware except for the Parser. Due to design time limits, the team used software running on a processor, a software CE, to read the MPEG encoded data from external memory to generate the data-packets for the VLD/RLD CE.

4.3.1 Resource Usage

Table 4.5 summarizes the resource usage for the hardware CEs in the MPEG-1 video decoder system. The first column gives the name of the hardware CE for which the resource usage measurements will be reported in the remaining columns. This excludes the Parser

CE as it is a software CE implemented on a MicroBlaze and the focus here is on the overhead for hardware PEs. Columns 2 and 3 report the number of LUTs and flipflops used respectively by the PE and SCS. The Virtex 2V2000 provides 56 dedicated Block RAMs (BRAMs) and 56 hard multipliers as extra design resources, along with the homogeneous array of LUTs and flipflops in their Combinational Logic Blocks (CLBs). Column 4 reports the number of BRAMs and multipliers used in the PEs as none are required for the SCSs. Since neither the Producer controller nor the Consumer controller use BRAMs or multipliers, the total logic resource usage for the controllers is reported in terms of LUTs and flipflops in Columns 5 and 6 respectively. Finally, Column 7 reports the percentage of extra LUTs and flipflops required in addition to the PE to create each CE. This is calculated by totaling the number of the LUTs/flipflops used by the two controllers and the SCS and then dividing it by the LUTs/flipflops used by the PEs.

The Consumer and Producer controllers have relatively consistent resource usage among the CEs as they all support the same instruction sets. The IQ and IDCT CE have slightly larger controllers because they execute bypass instructions, and therefore have to support variable instruction priority. SCS resource usage for all five CEs is minimal, the maximum being 92 LUTs and 98 flipflops by the VLD/RLD CE's SCS. This is due to the fact that its Producer has to generate both packets for the adjacent IQ CE and bypass packets. The great variance in the CE sizes arises from the different PEs they include, how complex they are algorithmically and how much of their design can be moved into BRAMs and Multipliers.

The MMR PE is the smallest because it is a patch to fix an error. Ideally, its functionality should have been encompassed in the MC/PR CE, but a redesign of the PE required more testing and risked more errors than adding a separate module to implement the extra functionality. Although the percentage overhead of adapting the MMR PE into a CE was 269% in terms of LUTs and 94% in terms of flipflops, the percentage overhead of the LUTs for the MMR CE is greater than 100% because the PE uses less LUTs than the combination of the producer and consumer controllers, however, it minimized the design time required to fix the error as discussed in Section 4.3.2.

Row 7 summarizes the complete hardware MPEG-1 video decoder system resource usage in terms of LUTs, flipflops, BRAMs and multipliers. Columns 2 and 3 of Row 7 report the total number of LUTs and flipflops in the complete hardware system, which comprises the resources required for the FIFOs used as Communication Links between CEs, the system reset manager, and the CEs themselves, and the totals for the SCSs. The fifth and sixth columns in the final row report the total overhead of the Consumer and Producer controllers

Table 4.6: Table of the CE design and integration times required for the system given in person-hours.

Measured Design Time	**VLD/RLD CE**	**IQ CE**	**IDCT CE**	**MMR CE**	**MC/PR CE**	**Percentage of Total Time**
PE design and initial debugging	480	80	96	19	700	85.6%
PE-Consumer Integration	3.5	4	1	0.25	15	1.5%
PE-Producer Integration	10	4	1	0.25	0	0.9%
Producer's SCS Design	1.5	3	1	0.5	0	0.4%
CE Testing	2	5	3	0.5	5	1.0%
Second Phase PE Verification	9	8	16	1.5	125	9.9%

Total System Integration Design Time	12
Total System Design Time	1607

used in the system and the final column is the percent overheard of converting all the PEs to CEs with respect to the total system resource usage. Based on this calculation, the total system overhead of the SIMPPL model for the MPEG-1 design is 27% of the LUTs and 20% of the flipflops used by the system. However, these numbers do not reflect the necessity of system integration and control logic if the SIMPPL framework is not used. Thus, even if the SIMPPL model with its increased flexibility and simplified integration is not employed, a portion of this extra logic is still required to implement dedicated protocols for passing data correctly between the different PEs shown in the original MPEG-1 block diagram in Figure 4.2. A reasonable first order approximation of the necessary dedicated system control logic is the sum of all the SCSs used in the system as they implement the PE control state machines. Therefore, an approximation of the overhead of the SIMPPL framework is calculated as the resources used by all the controllers divided by the total system resources. The approximated actual overhead of the SIMPPL framework is reduced to 23% of the LUTs and 16% of the flipflops in this MPEG-1 system.

4.3.2 Design Time Statistics

Table 4.6 provides a synopsis of the design times required for the different phases of the MPEG-1 video decoder system's hardware design in terms of hours. The first column reports the portion of the design time being measured and Columns 2 through 6 list the different hardware CEs for which these numbers are reported. Column 7 reports the percentage of the total design time that the sum of the hardware CE design times attributes to each phase of the design.

PE design and initial debugging are reported in the second row. This is the design time required to create the initial PE design, debug it in simulation and, where possible, perform some initial on-chip debugging. These values vary greatly depending on the complexity of the PE and how much independent algorithmic development was required. As previously mentioned, the MMR PE is relatively simple and both the IQ PE and IDCT PE are well-defined with readily available example implementations. Both the VLD/RLD PE and the MC/PR PE required a significant portion of time to develop and debug their algorithms.

Before trying to convert their PEs into CEs, the design team members each required about ten hours to learn about the details of the SIMPPL model. This includes understanding the operation of the Consumer and Producer controllers and how the SCS directs the local operations of the PE. These hours are not included in Table 4.6 as they are a one-time, non-recurring cost for using the SIMPPL framework as opposed to being specifically attributable to the MPEG-1 video decoder design. The PE-Consumer and PE-Producer integration times measure the time required to adapt the PE interface to the controller's requirements. This was most costly in the cases of the MC/PR PE-Consumer integration and the VLD/RLD PE-Producer integration, which required that their PE interfaces be adapted to the controller requirements. The Producer's SCS design time was dependent on the complexity of the dynamic program that could be run by the Producer. Once both the Consumer and Producer had been integrated along with SCS, CE testing was performed to verify that packets were being properly received and retransmitted by the CE.

At this point, the design team was able to perform a second phase of PE verification using the on-chip testbed to thoroughly test the operation of their CE. Once the CEs had been verified, the correct system-level connections and constraint specification files were generated in 12 hours. This is less than 1% of the reported total design time, 1607 hours. 1376 hours, or 85.6%, of the time was required for the PE design and initial debugging phase and another 9.9% for the on-chip phase of PE verification. This means that only 4.5% of the time was actually used to generate the CEs from their PEs and then integrate them into the system. For complex designs, system integration can traditionally account for as much as 30% of design time [36], more than six times the system integration time for the MPEG-1 video decoder using the SIMPPL model. This allowed the designers to focus the majority of their efforts on creating properly functioning PEs, as opposed to system-level control and communication protocols.

An interesting point of comparison is to look at the design times required for transforming the IQ PE into the IQ CE versus the IDCT PE into the IDCT CE. Both CEs were created by the same individual, where the IQ CE was developed first. As can be seen from Columns 3 and 4 in Rows 3 through 6 of Table 4.6, adapting the IQ PE into a CE required significantly longer than for the IDCT CE. This was due to the designer's learning curve for understanding the SIMPPL CE model. Once the IQ CE was completed, the designer was sufficiently comfortable with the model to successfully implement a comparable interface in six hours as opposed to the 16 hours initially required.

Similarly, by the time the need for the MMR CE was determined, the design team had almost completed the system. The large resource usage overhead of using SIMPPL controllers to adapt the MMR PE into a CE required only 1.5 hours and the increase to the system integration time was nominal due to the designer's familiarity with the model. If an application-specific system model had been used for the video decoder, the time to fix the MC/PR module and reintegrate it into the system would have been significant and the entire verification phase would have to have been repeated. Instead, the complete design of the MMR CE, from the initial PE design to the final on-chip verification required only 22 hours.

4.4 Summary

Three different SoCs have been implemented using the SIMPPL framework: a streaming video system, a snapshot video system, and an MPEG-1 video decoder. A detailed discussion of the design time required to implement and integrate the hardware CEs for the MPEG-1 video decoder illustrated that the designers spent most of the their time creating and verifying the PE's fuctionality (85.6%). In contrast, the SIMPPL framework allowed the designers to perform the system integration in 12 person-hours, less than 1% of the total system design time of 1607 person-hours.

The CE abstraction also allows designers to select an architecture for each CE that enables designers to capitalize on the functionality of their respective PEs. The SIMPPL Controller's architecture and instruction set can be tailored to the functionality of the PE. The performance, area and latency characteristics of these different variations of the controller for both FPGA and ASIC platforms are described to indicate the overhead of using the controller as a PE's system-interface.

Chapter 5

Designing SoCs on FPGAs

Although SIMPPL SoCs may be designed for ASICs or FPGAs, all of these designs have been implemented on FPGAs to exploit the shorter design cycle of a pre-fabricated platform. This is not the only advantage of using FPGAs in SoC design. They also enable designers to use on-chip design tools for profiling performance and verifying functionality, instead of simulation and estimation, since the design platform is reprogrammable.

This chapter opens with a summary of previous work and available design tools for SoCs on FPGAs in Section 5.1. Section 5.2 begins with a discussion of how profiling and simulation can be used to assess system performance and functionality. This is followed by a presentation of the proposed design methodology, highlighting the CAD tools created for SoCs on FPGAs. Finally, Section 5.3 outlines the experimental platform for all of the tools and systems created during this research. It concludes with a detailed description of two components of the FPGA design environment for the SIMPPL framework – specifically, the System Generator and the on-chip testbed. The remaining on-chip profiling tools, SnoopP and WOoDSTOCK, are discussed in Chapters 6 and 7, respectively.

5.1 Current Status of SoC Design for FPGAs

This section presents the current status of research being done using FPGAs to design SoCs and the design tools that are commercially available from FPGA vendors.

5.1.1 Systems Research using FPGAs

Initially researchers used FPGAs to implement purely dedicated blocks of hardware. However, in recent years the sizes of FPGAs have been large enough to include processors. Thus researchers have implemented Multi-Processor SoCs [37, 38, 39] and embedded computing systems [40] on FPGAs. Embedded computing systems are stand-alone computing systems that typically comprise one or more processors plus dedicated logic modules to meet design specifications, such as cost, area, and performance. There has been considerable research to investigate the numerous issues that arise from designing a system's hardware and software concurrently, called Hardware/Software Codesign.

A basic Hardware/Software Codesign flow starts with a description of the application that is *partitioned* into hardware and software components. The processes running on each component are *scheduled* to provide the necessary communication between modules. The behaviour of the two environments and their interface can be approximated using *cosimulation* techniques. If this model of the system does not meet the necessary specifications, the designer may need to return to the first phase of the process and re-partition the design. However, if the design constraints appear to be met, the design can be *cosynthesized* to the target platform and then *coverified* to ensure the required functionality. Hardware/software codesign research aspires to address the challenges resulting from each of these complex problems.

Most of the previous work on hardware/software codesign using FPGAs, however, uses the FPGAs to speed up the portions of an application that fail to meet the required specifications. Specifically, FPGAs have been used as part of the implementation platform whereas we are proposing that they can also be used during the system design process. Previous systems have used one or more FPGAs that are configured once per application [41], or dynamically reconfigured on Dynamically Programmable Gate Arrays (DPGAs) at runtime [42], to implement different functions. These systems benefit from the lower redesign costs of reconfigurable technology, but they do not utilize the technology to obtain feedback as to the actual system performance during the design process.

The recent advent of soft processor cores for FPGAs adds greater flexibility to SoC designs. Soft processors are provided as synthesizable HDL with no fixed on-chip placement, which enables designers to customize the processor core as an application-specific embedded processor. Hebert et al. [43] use soft-cores to simplify the creation of application-tailored processors by reducing the design time and removing the need for processor specific emulators. In this case, the designers use reconfigurable technology for prototyping. Some of the more recent hardware/software codesign research uses reconfigurable processors as the platform for implementation [44, 45]. These architectures combine a Reconfigurable Functional Unit (RFU) with the microprocessor, but do not use the reconfigurability to provide runtime performance information to benefit the partitioning process. Although the precise details of how applications are profiled for all of these projects are not provided, they simulate system performance to obtain this data. Preliminary work on a simulator of a reconfigurable architecture that does use runtime profiling information to guide partitioning was presented at DAC 2003 [46]. Based on the profiling data it dynamically remaps one of a restricted category of inner loops to a reconfigurable fabric using on-chip place and route tools.

Tools developed for embedded systems with reconfigurable hardware include a partitioner by Rakhmatov et al. [47] for dynamically reconfigurable systems that minimizes the energy-delay cost due to computation and configuration. Noguera and Badia [48] also present a dynamic scheduling methodology for runtime reconfigurable architectures in hardware/software codesign. To obtain a schedule that minimizes runtime reconfiguration overhead, the scheduler relies on a partitioner to create a good mapping of the algorithm to hardware and software. The partitioner's choices are based on delay and area estimates and data from software profiling tools.

5.1.2 Commercial System Design Tools for FPGAs

Xilinx and Altera both support the design of systems combining a processor with dedicated hardware. Altera provides designers with the System On a Programmable Chip (SOPC) Builder [49], which hooks into the Quartus II tools [50]. The user specifies a complete system from IP and user designed components and then the SOPC Builder generates the system.

Having created a design, the user can both debug and simulate its performance. ModelSim [51] can simulate a Nios system design, including the peripherals. This is done by simulating the entire system, including the processor, at the RTL or gate-level. While it provides cycle-accurate information, it is extremely slow making the simulation of larger applications prohibitive. To facilitate on-board debugging of the software, Altera provides multiple options. A simple monitor program called GERMS allows basic debugging operations, and for more complex options, there is GNU's *gdb*, but it can only run on a processor instantiated on an FPGA. Finally, Altera has partnered with First Silicon Solutions [52] to provide a core that connects to the Nios processor and acts as a system analyzer.

Xilinx provides users with a similar tool set, the Embedded Development Kit

(EDK) [53]. It is available as a separate environment for designing embedded systems on FPGAs. Similar to the Altera SOPC Builder, it generates the necessary hardware and software interfaces to facilitate the design of an embedded system. As with Nios, the complete design, including the processor, can be simulated at the HDL/gate level to obtain a complete simulation.

To simplify the debugging of designs run on a MicroBlaze processor, Xilinx provides an Instruction Set Simulator (ISS) that may be run in a cycle-accurate mode on a host computer. Currently, this cycle-accurate simulator supports a limited selection of peripherals, but allows for faster simulation of the processor than is possible with gate level simulation.

The users can also insert a Xilinx command stub (*xmdstub*) into their design, which attaches a monitor program to the design so that the user is able to debug the executable on the board. They access their executable via the XMD command window or the *gdb* interface on the host. As the XMD window is a TCL shell, users can add their own commands to interface with a design implemented on an FPGA. Finally, Xilinx supports an IP core, the Microprocessor Debug Module (MDM) that enables the user to perform JTAG-based debugging on a configurable number of MicroBlaze processors.

Both companies provide numerous tools for debugging application software as well as some profiling tools, such as gprof, that are able to run locally on their soft processor. However, neither supplies tools capable of providing cycle-accurate performance information for an application running in real time on a soft processor core instantiated on an FPGA without requiring instrumentation of the source code. The importance of obtaining precise performance measurements for quality design implementation on FPGAs necessitates the usage of runtime monitoring tools for designs too large for proper simulation. All of the existing commercial FPGA tools of which the authors are aware are intrusive, which is a factor in system design. Section 5.2.1.1 describes gprof's functionality, explaining the source of inaccuracies in its profiling results.

5.2 Moving from Off-Chip Estimation to On-Chip Evaluation

A common problem in system design is that the quality of the implementation is dependent on the designer's decisions as to whether functionality is provided through software or hardware. However, these choices are often based on estimates or models. Since dividing the design into hardware and software modules, or partitioning, often occurs at the very beginning of the design process, there is no precise feedback available to the designer. The next section contains a brief discussion of some profiling and codesign simulation tools and how they help select partitions for a design.

5.2.1 Simulating versus Profiling Hardware/Software Codesigns

Conventional cosimulation environments emulate systems that combine a microcontroller with dedicated hardware to implement an embedded system [54, 55]. However, their simulation techniques result in only an approximation of the actual system performance. Mentor Graphics offers Seamless [56], a hardware/software co-verification simulation tool that enables a designer to interface an ISS with memory and dedicated logic to detect scheduling problems. However, the cost of simulating both hardware and software causes simulations to run at 1000 to 5000 instructions/second.

Simulation and profiling are integral methods for acquiring performance information about an application, varying in both speed and accuracy. The most accurate and detailed information is obtained by simulating a system's performance on a cycle-accurate simulator. However, this accuracy incurs significant overhead, and consequently, may be too slow for large systems. Instruction-level simulators provide a coarser level of granularity of simulation, sacrificing accuracy for faster execution.

Most modern microprocessor's include a limited number of hardware performance counters that can be used to profile the runtime execution to count "Events" that measure different aspects of performance. A Performance Application Programming Interface (PAPI) [57] provides users with a high-level interface for the usage of these counters. By annotating the application with calls to PAPI functions, the user can count numerous different kinds of Events [58]. The accuracy of PAPI's results is dependent on a large enough code space such that the overhead of the PAPI sampling code does not dominate the counter values [59]. Intel provides a commercial performance analyzer, similar to PAPI, called *VTune*. It provides a graphical interface that allows users to instrument their software post-compilation to utilize the hardware counters on their processors to profile performance [60]. Another popular profiling tool for measuring performance from GNU [61] is described in detail in Section 5.2.1.1. Unlike PAPI, it is a statistical profiler that does not use the hardware counters. However, similar to PAPI, the results are of limited precision.

5.2.1.1 GNU's gprof

When creating an SoC, one approach is to begin by implementing the entire design in software and then profiling it. Using this information, a designer moves components to hardware to meet the necessary performance constraints. Tools such as *gprof* can measure the performance of a software implementation as well as determine the characteristics of an application's execution [62] to help guide a designer's partitioning choices. This profiling may be done on a different system from the actual target system, which leads to inaccuracies in the results. Changing the platform affects the Instruction Set Architecture (ISA), the microarchitecture, the compiler and potential optimizations, resulting in variances in the executable that is profiled.

To use *gprof*, the designer must compile and link the application with the profiling options enabled. Unlike PAPI, where the user manually inserts the profiling routines into the application, the compiler automatically generates the extra code necessary for generating the profile information used by *gprof*. It inserts this code into the application to count the function calls and to generate an interrupt that samples the PC. While this method allows a precise tabulation of the number of times each function is called, the timing information it obtains from the execution is not as accurate.

At specific intervals, normally every 10ms (100Hz), *gprof* samples the PC [62]. Depending on the value of the PC, it increments the execution time of the appropriate function by the sample time. This means that unless the total runtime of the application is significantly larger than the sampling period, the measured execution time for each function may not reflect the actual execution time. Therefore, for smaller executables, applications are run numerous times so that the profiling information accumulates for a substantial runtime.

Obviously, there is a trade off between using a statistical runtime profiler and simulation to profile software execution on a processor. Statistical profilers obtain values that are imprecise and there is overhead to running the profiling software. However, the runtime profiling overhead is negligible compared with the time required to provide cycle accurate information by simulation. In other words, while *gprof* may add additional seconds or minutes to a software application's execution, cycle-accurate simulation requires seconds to minutes to simulate each cycle of a hardware system, depending on its complexity.

5.2.2 Proposed Design Methodology

Figure 5.3 illustrates the proposed SoC design methodology using the SIMPPL SoC framework for FPGAs. The portions of the design process for which design infrastructure has

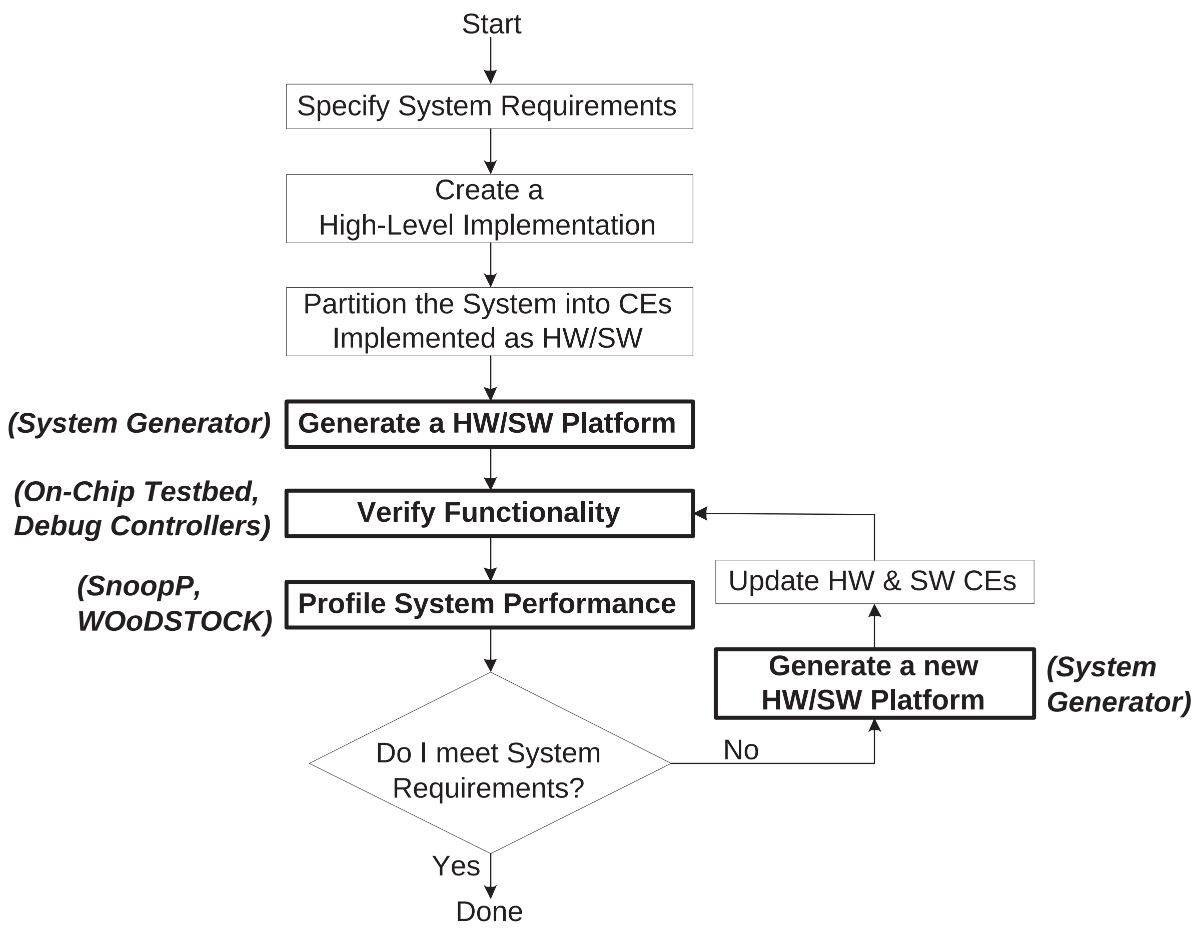
Start
Specify System Requirements
Create a High-Level Implementation
Partition the System into CEs Implemented as HW/SW
(System Generator)
Generate a HW/SW Platform
(On-Chip Testbed, Debug Controllers)
Verify Functionality
(SnoopP, WOoDSTOCK)
Profile System Performance
Update HW & SW CEs
Generate a new HW/SW Platform
(System Generator)
Do I meet System Requirements?
No
Yes
Done

Figure 5.1: An SoC Design Methodology for Reconfigurable Platforms.

been developed are highlighted and the names of the corresponding tools given. The process begins by having designers specify the requirements for the system and then creating a high-level implementation of the system that can be used to verify the final version. Next, the system is partitioned into modules that are realized as independent CEs in hardware or software. The system-level interconnections between CEs, along with templates for both hardware and software CEs, are generated using the System Generator as described in Section 5.3.2. This allows designers to focus on the implementation of the individual CEs, especially their PEs.

Having created the system, designers can perform preliminary functional testing in simulation. However, on-chip testing can be used to provide more thorough testing of the SoC. The debug controllers, previously mentioned in Section 3.2.1, allow for fine-grained testing of the integration of the PE, controller, and SCS. An on-chip testbed, discussed in Section 5.3.3, has also been created that enables designers to use a larger number of test vectors to verify PE functionality than is practical in simulation. After verifying functionality, designers may need to profile the system to ensure that it meets the original performance specifications. Chapter 6 presents SnoopP, an on-chip software profiler, and Chapter 7 describes WOoDSTOCK, an on-chip profiler for monitoring system communication to determine load balancing.

Using the profiling information obtained on-chip, designers can assess if the SoC design meets the original specifications. If not, they can choose to repartition the system into different CEs or implement some of the software CEs as hardware CEs to improve performance. Then a new system architecture can be generated using the System Generator if necessary, while the designers update the necessary CEs. The designer iterates between evaluating functionality and performance and redesigning the SoC until the implementation meets system requirements, signifying the completion of the process. The benefits of using the SIMPPL framework and on-chip design tools as proposed in this design methodology are summarized in the following section.

5.2.2.1 Benefits of Designing SIMPPL SoCs using FPGAs

The premise of using on-chip design tools is that simulating the cycle-accurate performance of a reconfigurable circuit is extremely computationally intensive and should only be used to determine preliminary functionality and not performance. Since an FPGA design platform is reconfigurable, it is possible to include on-chip profiling and verification tools during the design process that allow designers to obtain accurate information quickly. Furthermore, the SIMPPL system framework facilitates the creation of generic tools that can be tailored to system-specific requirements. For example, the CE abstraction facilitates verification of the PE's functionality. A CE can be instantiated on an FPGA platform and test vectors supplied over the internal Rx Link. This method can be used to quickly integrate the CE with a testbed that generates numerous test vectors because the Rx and Tx Links and communication protocols are fixed. Designers can then obtain direct feedback on the system's actual behaviour by using the reconfigurable environment to evaluate the system orders of magnitude faster than simulation, which can reduce overall design time.

However, profiling on-chip means that results are not only obtained quickly, but designers can also profile their system using runtime data to detect data-dependent behaviour. The accuracy of the profiler is determined by the operating frequency of the profiler relative to that of the rest of the system. If the operating frequency is the same as that of the system, the profiler will provide clock-cycle accurate results. The designer can also run the profiler at a slower operating frequency and obtain a statistical profile, similar to *gprof*, if that is sufficient for the application.

All of the on-chip tools are independent hardware modules that are scalable and adaptable to the system-specific architectures of different applications. Both of the profiling tools use snooping to detect the events they are measuring. They monitor signals inherent to the system so that the system's operation is unaffected by the profiling. Neither SnoopP nor WOoDSTOCK, discussed in the following chapters, insert extra software code into the application, which ensures that software performance is unchanged by the addition of a profiler. Furthermore, the profiling tools do not add extra hardware circuitry into the application's processing path, ensuring that the functional operation of the system is not altered by the profiler. While the profiling tools are non-intrusive to the system's processing, the additional circuitry may have side effects on the system's performance by reducing the maximum clock frequency of the design depending on the percentage of the chip resources required to implement the system. In situations where the application uses the majority of the chip resources, a larger chip from the same family can be used during the design process to reduce the effects of the on-chip design tools on the maximum clock frequency.

The most important benefits to designing SoCs on an FPGA are that there is no need to finalize the partitioning of the design at the beginning of the design process or to create a complex cosimulation environment to model communication between hardware and software. The system can be run on the reconfigurable fabric where the precise interaction between hardware and software can be tested. It is also easy to iterate between partitioning and profiling the design and the accuracy of on-chip profiling information provides better feedback for the partitioning process.

5.3 Designing SIMPPL SoCs on FPGAs

This section describes the experimental platform used to create and test all of the system design tools for FPGAs described during the remainder of this thesis. The first tool described herein is the System Generator, which allows designers to create all the system-level files for a SIMPPL SoC implemented as any combination of software and hardware CEs. Finally, the on-chip testbed created for verifying CEs using the defined structure of the SIMPPL network is discussed.

5.3.1 Experimental Platform

The benefits described above are applicable to designing on both Altera and Xilinx FPGAs with either NIOS or MicroBlaze soft processors. To test these tools and demonstrate how they could be used as part of an on-chip design methodology, a Xilinx Multimedia Board with a Virtex II 2000 is used. All the systems and on-chip software are synthesized and compiled with Xilinx's EDK tools. Xilinx's MicroBlaze soft processor can be configured for application-specific requirements to improve the performance of the system. However, since the objective of this work is to study the tools for a methodology, the default parameters for the MicroBlaze core are adequate. These include a software implementation of the multiply/divide instructions and no data or instruction caches.

The MicroBlaze also includes eight master and eight slave ports for asynchronous FIFOs. Xilinx has included read and write access instructions to these ports, which they call Fast Simplex Links (FSLs) [63], so that data transfers can be implemented directly in software. To access FIFO ports directly from a NIOS processor, designers can create application-specific instructions that will also allow them to access the FIFOs from software. All of the on-chip tools currently use the MDM to upload information from the chip to the host computer, where the *xmd* monitor program provides the user interface. To run these tools on Altera chips requires the present I/O interface to be adapted to Avalon bus protocols.

5.3.2 System Generator

The System Generator builds the top-level connectivity for all the components to create a user-specified application-specific architecture. It also generates template CEs for both the hardware and software CEs to act as place holders for the actual CEs used in the final design. To exercise all the communication links connected to each CE, the template CEs: detect available data on input links; read in data from input links; detect if output links are full; and write output data to output links. This functionality is implemented using C-source code for software CEs and HDL source code for hardware CEs. It can be used to model the sequential consumption and generation of data by all the system CEs, which can be used by designers to verify the system-level connectivity. The template source code files for both hardware and software CEs are easily replaced with the application-specific CE designs. The template CEs also facilitate generating example systems for testing these tools; although these CE templates may not exactly model a particular internal functionality, the system-level communication provides suitable benchmarks for WOoDSTOCK.

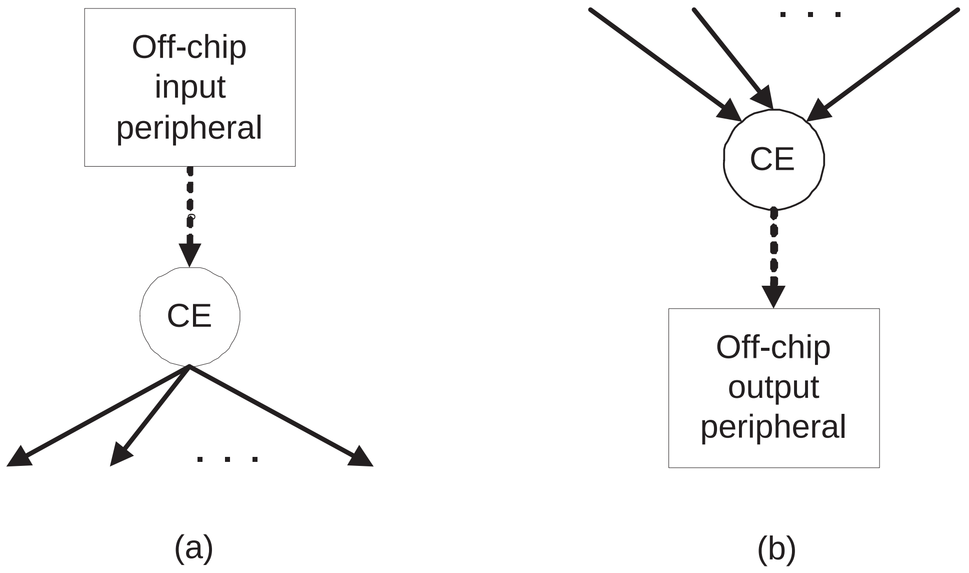

Figure 5.2: (a) A CE connected to an off-chip input peripheral. (b) A CE connected to an off-chip output peripheral.

A limitation of the System Generator is that the I/O communication links cannot be automatically generated as part of the system infrastructure. This is because the physical interface and logic communication protocols for I/O communication links are dependent on the nature of the interface for each off-chip device. Thus, for the purpose of the hardware and software CE templates, off-chip peripherals that produce/consume system data are modelled as part of the CE to which they are connected. If there are no internal input links to a CE, as shown in Figure 5.2(a), then it generates internally the output data packets to be sent to its children CEs. These packets model input data received from an off-chip hardware peripheral and processed to generate the CE's output data packets. Similarly, if a CE has no output links, as seen in Figure 5.2(b), all data packets are consumed to model output data generated for an off-chip hardware peripheral.

The System Generator currently creates all the necessary source files to describe a unique project for the Xilinx Platform Studio (XPS) software, however, it could easily be adapted to generate the appropriate system files for Altera's SOPC Builder. These files are generated based on an input description file of the system provided by the user to the System Generator, as shown in Figure 5.3. The format of the input file is shown in Appendix B. It describes the number of CEs, internal links, clock domains, and external memory banks in the system. For each CE, the user details its clock domain and how it generates outputs as a function of its inputs. For software CEs, there is a final option of

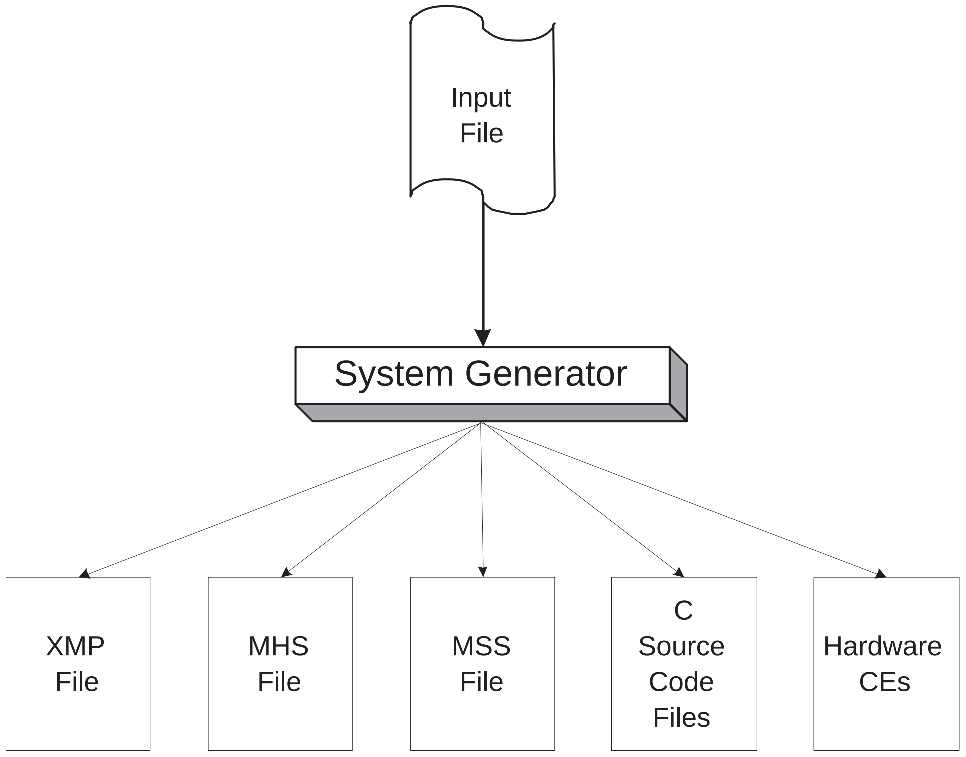

Figure 5.3: System Generator Design Flow.

selecting if an external memory is included.

Based on this information, the System Generator creates the necessary files to describe and build the bitstream for an SoC using XPS. The Xilinx Microprocessor Project (XMP) file contains the project settings used by XPS to build the system. The hardware system is described in the Microprocessor Hardware Specification (MHS) file in terms of the IP cores, such as MicroBlazes and FSLs, that comprise the system. The corresponding software drivers for these cores are given in the Microprocessor Software Specification (MSS) file.

The software CE template consists of a soft processor with its own local instruction and data memory (recall Figure 2.3) and the C source code file that includes read and write functions for the input and output links and a sample main program. The main reason for using local memory is that sharing memory creates possible data hazards. Even if two processors share a block of memory but have two distinct address spaces, there will be bus contentions causing interference in the execution results. Here, it is assumed that each of these modules should have the same performance independent of the number of other CEs in the system and that there is no need to share data between two CEs unless it is sent via a link. Each CE's template source program file is stored on its local memory and

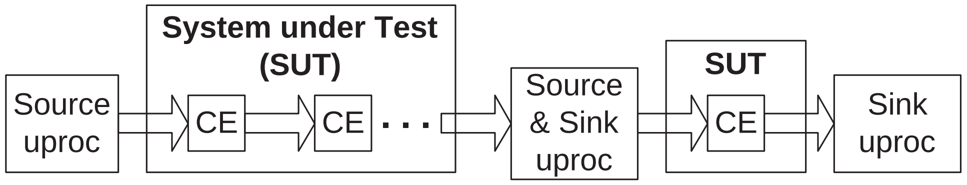

Figure 5.4: The on-chip testbed for debugging CEs.

provides functions for receiving and transmitting over the links and constants representing the processing time required to generate output data for the CE. The project file is designed to include all of the executable source files for software CEs in the bitstream so that all the processors begin running their program immediately after download to the FPGA.

The hardware CE template consists of files used to describe IP cores to XPS. A Hardware Description Language (HDL) file is generated containing the state machines for accessing the input and output links as well as a demonstration of how output generation can be synchronized to the availability of inputs. XPS requires that a Microprocessor Peripheral Definition (MPD) file be included with the core to define the ports and parameters of the interface to the rest of the system. Finally, a Peripheral Analyze Order (PAO) file is also necessary to indicate the hierarchy of HDL files to the synthesis tools.

5.3.3 On-Chip Testbed

One contribution to the on-chip design infrastructure is an on-chip testbed for systems designed using the SIMPPL model. The standardized physical interface and communication protocols of a CE allow the designer to use a flexible testbed architecture as shown in Figure 5.4. CEs can be verified individually, as independent processing stages, or in combination with adjacent CEs. Furthermore, since the design is implemented on an FPGA, it is possible to run the testbed on-chip to verify the behaviour of CEs with a large number of data packets to obtain quick and accurate results. Previous work demonstrated that debugging [64, 5] and profiling [7] designs using on-chip resources results in a significant reduction of the time required to obtain information for the designer. Since design verification commonly requires greater than 50% of the overall design time, sometimes as much as 70% [36], it may be possible to reduce the percentage of time spent verifying the design, and thus reduce the overall design time.

The testbed comprises the processors and the software required to generate (Source) and interpret (Sink) data packet streams for the CEs. The MPEG-1 video decoder described in Section 4.1 is designed for a Xilinx Virtex2V2000, so the MicroBlaze soft processor is used in this testbed. High-level functions are built to generate each data packet from the instruction and data pointer specified by the user. The user can then quickly alter the number and types of data packets sent by the Source to the System Under Test (SUT) by changing the instructions in the source code and then compiling and downloading the processor executable to the Source Processor. Creating the data stream using software allows a significantly quicker turnaround time for testing the SUT with different data packet streams than is possible with the source data stream coded as a separate hardware module. The Sink Processor runs a program that detects and interprets packets received from the SUT and then allows the user to log them. The Sink processor program can also be combined with the Source processor program to allow designers to log the intermediate state of the design as shown in Figure 5.4.

The on-chip testbed facilitated the detection of a significant PE error that required the redesign of the MPEG-1 video decoder pipeline. Using the MPEG-1 pipeline from the VLD/RLD CE to the MC/PR CE as the SUT, the design team found that a portion of the design specification for the MC/PR PE had not been implemented. The team created the new CE called the Missing Macroblock Replacer (MMR) CE and inserted it into the decoder pipeline just before the MC/PR CE to correct the error as shown in Figure 4.2. The modularity and structure of SIMPPL made this change to the pipeline very easy.

Although the on-chip testbed runs orders of magnitude faster than in simulation, it does not likely exhibit the exact runtime behaviour of the final system. A runtime data stream could be irregular with data words sometimes arriving every clock cycle and sometimes delayed for numerous clock cycles, thus the Source and Sink may process data slower or faster than the system at runtime. However, the Consumer and Producer controllers, which interface the CE with its preceding and subsequent CEs, are able to abstract runtime data behaviour from the PE as they separate the communication protocols from the actual data processing. Both are able to properly stall the PE if there are no source data packets in the Rx Communication Link or no space in the Tx Communication Link so that the PE exhibits correct runtime behaviour independent of the data rate.

5.4 Summary

This chapter introduces the second main contribution of the thesis – work to develop a design methodology and infrastructure tailored to designing SoCs on an FPGA platform using the SIMPPL framework. A review of the current research and the available design tools for SoCS implemented on FPGAs is provided. On overview of our proposed new design methodology, highlighting the use of on-chip design tools that can provide accurate information in less time than simulation, is given. This includes two of the tools created for this infrastructure, the System Generator and an On-chip Testbed.

Chapter 6

SnoopP

This chapter describes the architecture (Section 6.1) and experimental evaluation (Section 6.2) of our on-chip software profiler called SnoopP. To demonstrate the accuracy of profiling an application with SnoopP, the results are compared with those obtained using gprof.

Concurrent work has also been done at the University of Ljubljana to develop a profiler for soft processors that is similar to SnoopP, called COMET [65]. COMET is demonstrated using the NIOS processor and the main goal is to use COMET as part of a hardware/software design flow to help estimate performance and guide hardware/software partitioning.

6.1 General Architecture

SnoopP is designed as an independent hardware module that the user includes in their system design. The internal structure, shown in Figure 6.1, subdivides into two components – the clock-cycle counters and the system bus interface. The former profiles the system with the user-specified number of counters while the latter provides off-chip access to their values. To profile source code execution, SnoopP connects to a bus that displays the executing Program Counter (*PC_EX*) and a *valid_instr* signal that is high when the value on the *PC_EX* bus is valid. Each code segment counter increments every time the value of the *PC_EX* bus is both valid and in range.

When designers clock SnoopP using the system-level clock, as shown in Figure 6.1, this results in an accurate clock-cycle count of the time spent in a code segment. To obtain a cycle-accurate profile of data region accesses, instead of source code execution, SnoopP's

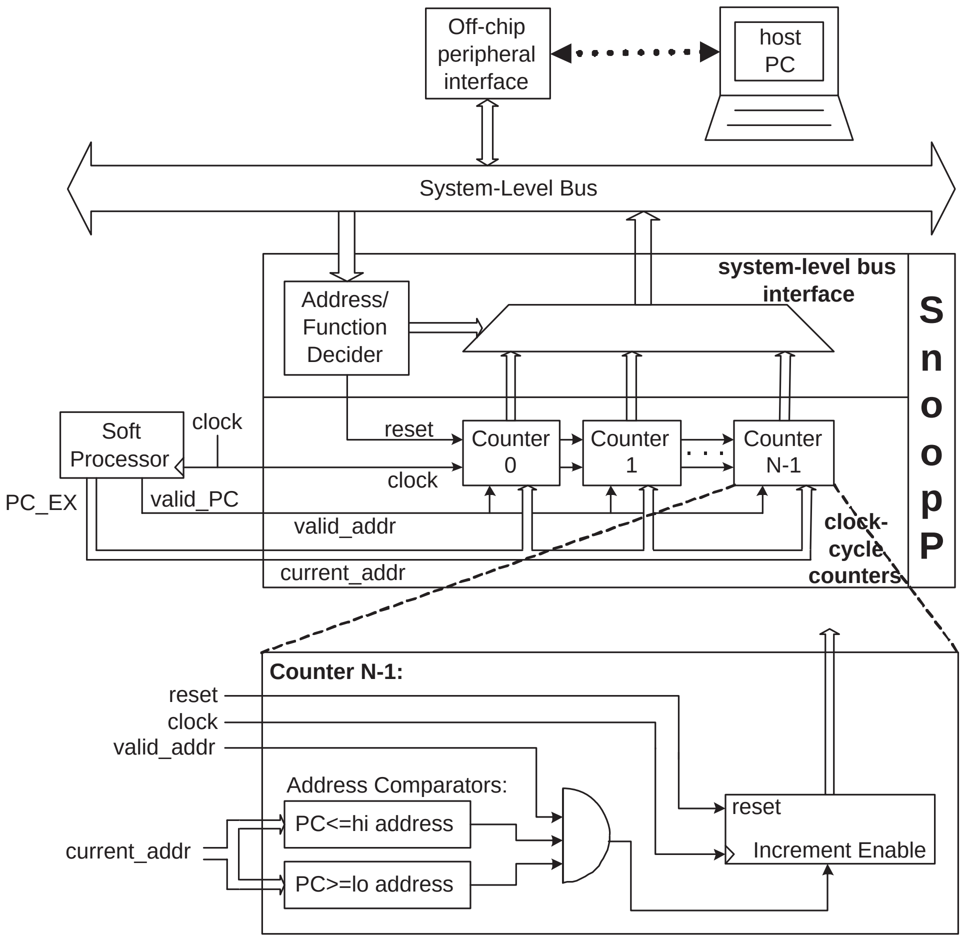

Figure 6.1: The Generic SnoopP Architecture.

counter_address bus needs to be connected to the system address bus and a "valid data address" signal. It should also be noted that SnoopP can be used to obtain a statistical profile of data accesses or program execution, similar to gprof. Using an independent clock to drive the module, instead of the system-level clock, allows the user to choose an appropriate clock frequency that provides them with an adequate granularity for their profiling data.

Counter N-1 is magnified to illustrate the internal workings of a clock-cycle counter. To determine if the address, for example the PC_EX, is in range, comparators check to see if the present PC_EX value is between the specified low and high addresses. If the PC is valid and is presently accessing an address within these bounds, then the counter value is incremented. The counters are memory mapped to the system-level bus, which enables a user to read and reset the counters from a host computer via the off-chip interface module connected to the system-level bus. Thus, SnoopP allows the designer to measure the exact number of clock cycles the program spends executing specified code segments or accessing data memory at runtime.

When SnoopP is used on a Xilinx FPGA, it interfaces with the On-chip Peripheral Bus (OPB) standard to communicate information off-chip via the MDM described previously. Xilinx's xmd terminal runs on the host computer, providing a user interface to the on-chip system. It supports TCL scripts that can be used to read and reset the counters in SnoopP as well as to analyze the measured results.

6.1.1 Design Decisions

The objective is to make the SnoopP circuit as small and as fast as possible so that it does not impact the system design process. However, to be a useful software profiler, it must allow the user flexibility in specifying address ranges and the number of counters for the system. The decisions outlined below are an attempt to balance these issues. SnoopP allows the user to choose up to a maximum of 16 profiling counters to limit the circuit size. Each counter requires two comparators to determine if the 32-bit address is in a valid range. The user can obtain the addresses for the upper and lower bound parameters of the address ranges by assembling the code or reading the symbol table. To provide complete flexibility in specifying the address ranges, SnoopP allows designers to select address ranges as small as a specific address to an entire 32-bit address space. This means that when SnoopP profiles source code, a code segment could be anywhere from a single instruction to an entire program.

However, this flexibility has potential performance costs as the comparators must be large enough to differentiate between individual addresses. Furthermore, the user needs to be able to memory map SnoopP to any address space that is available in their design. Currently, designers must select the number of counters, their individual address ranges, and their location in the memory map pre-synthesis to limit the effects of parameterization on SnoopP's critical path delay. Since it is desirable to be able to reload the address boundaries between application runs, the designer can easily change SnoopP to support runtime programmable address ranges if clock speed is not a concern. However, a better option would be to enable the designer to update the bitstream to change the hardwired address ranges without re-synthesizing the design. Currently, there are some tools that could be used by designers to do this, but there is no clear, user-friendly tool flow that provides easy access for making these changes. However, such a tool is feasible to implement for future work.

When using SnoopP, it is important to remember that only address accesses in contiguous regions of memory are counted. For example, to accurately profile how long a function *A* with subfunction calls *X*, *Y*, and *Z* takes to execute, the user must assign a counter to the function as well as to each of the subfunctions called during its execution (i.e. *A*, *X*, *Y*, and *Z*). Furthermore, if another function *B* calls any of these subfunctions, for instance *Y*, it may not be possible to distinguish which portion of the subfunction *Y*'s execution time is due to function *A* versus function *B*.

Since most software programs require many cycles for completion, 46-bit counters are used to store the clock cycle counts, which is equivalent to letting the profiler run at 100 MHz for eight days. When profiling source code, the decision to count clock cycles, as opposed to the number of executed instructions, is based on the desire to be precise as to the actual time spent executing each code segment. Given that most code segments will include a branch and/or a memory fetch, there will likely be pipeline stalls that could significantly increase the time spent executing a segment. This stall time is not accounted for if only the number of executed instructions are counted.

While this architecture provides the user with significant flexibility for profiling software, the hardware required to implement SnoopP with the maximum 16 counters translates into a maximum circuit size that utilizes 849 flipflops and 1349 LUTs for logic. The 16 46-bit counters require 736 flipflops, accounting for 87% of the flipflops utilized by SnoopP. Alternatively, SnoopP's counters could also be implemented using an on-chip BRAM, however, this would reduce the flexibility of the code segment definitions. The user would have to constrain their definition such that no more than two code segments overlap for a given address. This limitation arises because the BRAMs allow up to two concurrent memory accesses. The remaining 13% of the flipflops in SnoopP latch internal control signals to prevent the system's critical path from being in SnoopP when the system is synthesized.

For simple soft processor systems, SnoopP does not limit the maximum clock speed and, ideally, the profiling circuit will never be on the system's critical path as its maximum operating frequency is 127MHz. However, if the design is approaching the capacity of the FPGA, it may be unavoidable. If necessary, SnoopP can be pipelined to reduce the delay path in faster systems. This includes latching the *current_address* and *system_ABus* buses and the *valid_addr* and *system_ABus_select* signals. These additions have not been incorporated into the present version of SnoopP as they are unnecessary and increase the size of the circuit.

To implement the 32 32-bit comparators used to determine if an address is within each counter’s address range requires 1024 LUTs. This encompasses 76% of the LUTs employed in the SnoopP, and does not include the logic required to interface SnoopP to the OPB. The OPB interface must use two comparators to resolve that the user has accessed the SnoopP memory space. More logic is required to select the counter operation and to implement a 16-to-1 multiplexer that drives the appropriate value onto the OPB. Thus, the resources necessary to implement SnoopP using 16 counters is actually larger than what is required to implement a MicroBlaze processor, and an area for future study is possible methods of minimizing the size of SnoopP, such as reducing the resolution of the comparators for the address ranges.

6.2 Experimental Evaluation

This section illustrates how SnoopP is used to profile the source code of two different benchmarks. It details the methodology used and the issues encountered when profiling each application.

6.2.1 Methodology

The first benchmark, Dhrystone [66], is a relatively small application whereas the second, a cipher block chaining implementation of the Rijndael algorithm (AES) [67], is significantly larger. There are two possible methods of using SnoopP to profile software performance. The first is to use *gprof* to obtain an initial profile of executable performance. This information can be used to try and assign the counters to what *gprof* determines are the important regions of the executable.

The other method is to perform all the profiling using SnoopP. To do this, the user divides the software executable into groups of functions forming continuous address blocks and obtains an initial profile. The regions that require the largest percentage of execution time can be subdivided further to determine which specific functions take the most execution time. Depending on the size of these regions, the number of functions and the division of execution time, the user may have to iterate through this process until a suitable performance profile has been obtained.

For the purpose of this study, *gprof* is used to provide a baseline comparison of the varied accuracy between statistical and clock cycle accurate profilers. The application is initially profiled with *gprof* on a Sun Ultra 80 Model 4450 running version 5.8 of the Solaris OS. The design is then run on a MicroBlaze processor instantiated on the FPGA and

Table 6.1: gprof Statistics on Functions comprising the Dhrystone Benchmark after One Hundred and One Million Passes.

Function Name	**100 Passes**	**One Million Passes**	
	Total Calls	**Total Calls**	**Percent Time**
internal_mcount	—	—	31.5
main	1	1	11.2
Proc_8	100	1000000	10.4
Func_1	300	3000000	9.6
Proc_7	300	3000000	6.1
Func_2	100	1000000	6.1
Proc_1	100	1000000	5.9
Proc_6	100	1000000	4.5
Proc_2	100	1000000	3.7
Func_3	100	1000000	3.5
_mcount	—	—	3.2
Proc_3	100	1000000	1.9
Proc_4	100	1000000	1.6
Proc_5	100	1000000	0.8

profiled with SnoopP, for more precise performance information, and Xilinx's version of *gprof* tailored to run on the MicroBlaze, *mb-gprof*. These results are compared to determine if using *mb-gprof* on the MicroBlaze obtains profiling data that correlates better to the SnoopP profile than using *gprof* on the Sun station. Both the Dhrystone and AES applications are compiled using *gcc -O2* for both the Sun and MicroBlaze platforms, which optimizes the application's source code without inlining functions. As the authors are unfamiliar with the executional behaviour of both benchmarks, the *gprof* profiling results are used to guide the assignment of counters to different code segments. It prevents the intentional assignment of the counters to known problem areas and ensures that the method is totally dependent on profiling information.

6.2.2 Dhrystone

Dhrystone is a synthetic benchmark for testing a system's integer performance that Xilinx uses to measure MicroBlaze processor performance. Table 6.1 summarizes the results obtained using *gprof* on the Sun workstation. Column 1 contains the function names for which *gprof* returns results. Columns 2 and 3 list the number of times each function is called when Dhrystone makes one hundred passes and one million passes through the main loop respectively. Finally, Column 4 reports the percentage of execution time that *gprof* attributes to each function when the main loop makes a million passes. *gprof* is unable to obtain the same type of statistical timing information when Dhrystone makes only one hundred passes of the main loop as it completes execution in less than 10ms on the workstation.

As can be seen from Columns 2 and 3, *Func_1* and *Proc_7* are called three times more than the other procedures in the benchmark. However, this does not provide any indication as to which functions are most costly to implement in software. The functions *internal_mcount* and *_mcount* are part of the profiler and count the number of times a function is called during execution. While *gprof* does not report the number of times these functions are called, their combined overhead accounts for 34.7% of the execution time calculated by *gprof*. Although this data dominates the results, the increased execution time provides a clearer picture of where most of the execution time is probably spent. It is also interesting to note that while *Func_1* and *Proc_7* have three times the number of function calls, the executable appears to spend most of its execution time in *main* and *Proc_8*.

Since this application only has a few functions, it is possible to assign the counters in SnoopP to almost every function, profiling 91.5% of the static code size. Table 6.2 outlines how the application is partitioned into profiling segments. It includes the number of instructions per code segment and the percentage of the static code size utilized by each function to give better context to the profiling results. The table illustrates that *main* accounts for 43.5% of the static code size whereas *Proc_8* is only 6.9%. When selecting which regions should be profiled, all the initialization and clean up portions of the executable were ignored as they add little overhead and cannot be moved to hardware.

SnoopP profiled the Dhrystone application for both 100 and one million passes on a MicroBlaze processor that initially implemented integer multiplies and divides using software functions. This application was also profiled using *mb-gprof* on the same FPGA using the same configuration of the MicroBlaze. We set the configuration parameters for *mb-gprof* to mimic those used in *gprof* on the Sun station. However, the clock frequency of the Sun station's processor is 450MHz, whereas, the MicroBlaze processor is running at 27 MHz.

Table 6.2: Dhrystone SnoopP Counter Assignments.

Counter Number	**Function Name**	**Number of Instructions**	**Percentage of Static Code Size**
0	main	376	43.5%
1	Proc_1	70	8.1%
2	Proc_2	16	1.9%
3	Proc_3	18	2.1%
4	Proc_4	24	2.8%
5	Proc_5	7	0.8%
6	Proc_6	42	4.9%
7	Proc_7	5	0.6%
8	Proc_8	60	6.9%
9	Func_1	11	1.3%
A	Func_2	49	5.7%
B	Func_3	9	1.0%
C	__divsi3	38	4.4%
D	malloc	11	1.3%
E	__mulsi3	22	2.5%
F	strcmp	32	3.7%

Therefore, to sample the PC after the same number of clock cycles on both platforms, the PC on the MicroBlaze should be sampled at 6Hz since the Sun station processor is sampled at 100Hz. To determine if varying the sampling rate of *mb-gprof* significantly affected the results, we profiled these systems using both 6Hz and 100Hz sampling frequencies.

Figure 6.2 graphs the profiling results for the Dhrystone application executing 100 and one million passes using all three profilers – *gprof*, *mb-gprof*, and SnoopP. The x-axis lists the functions that have been profiled and the y-axis describes the percentage of the application's overall execution time attributed to each function. The absence of a data point for any of the six profiles plotted here indicates that the profiler did not return any profiling data for the specific function. For example, the *_mcount* and *internal_mcount* functions are part of *gprof* and *mb-gprof*, not Dhrystone. Thus, SnoopP obtains no profiling data for these functions.

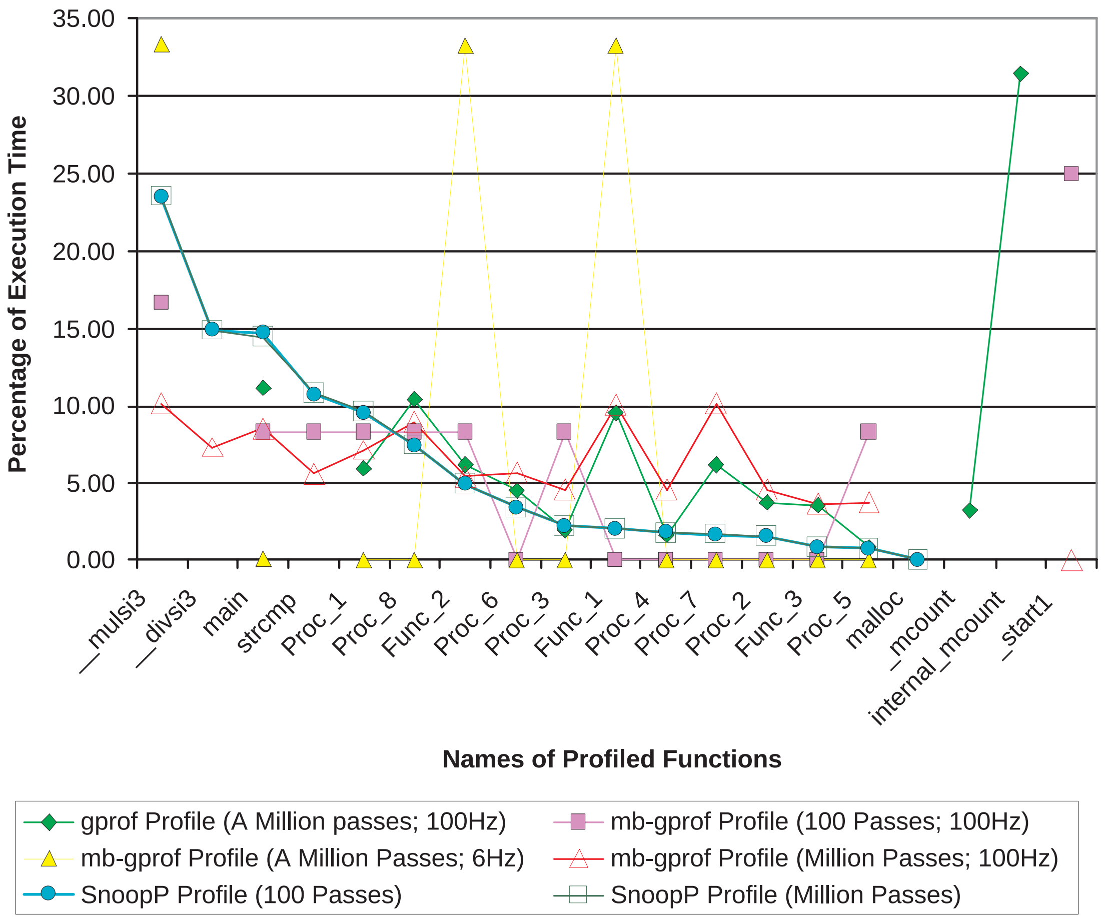

Figure 6.2: Profiling Results for Dhrystone using gprof on a Sun station and mb-gprof and SnoopP on a MicroBlaze using software implementations of the multiply and divide functions.

Figure 6.2 also does not contain a profile reporting the percentage of execution time when Dhrystone ran for 100 passes and was sampled at 6Hz using *gprof*. This is because, similar to when Dhrystone was profiled on the Sun station, the execution time of the application is less than the sampling frequency. Therefore, the figure only plots the *mb-gprof* profiling results when sampling at 100Hz for Dhrystone running 100 passes and at both 6Hz and 100Hz when Dhrystone executes one million passes.

Comparing the plots of the SnoopP profiles for 100 passes and one million passes illustrates the consistency of profiling information obtained using SnoopP. While *gprof* is only able to obtain a timing profile by executing the Dhrystone main loop a million times versus one hundred times, SnoopP obtained results that vary by no more than 0.24% for both cases (please see Appendix D for the detailed results). The resulting variance is easily explained by the diminishing significance of initialization code within *main* with respect to the longer execution time of the main processing loop. Therefore, SnoopP is able to obtain more accurate and consistent results than *gprof* in only 0.01% of the execution time.

By comparing the plots of the profiles obtained by SnoopP versus *gprof*, we see that there is a significant difference between results. Not only are the execution time percentages different, but *gprof* ranks *Proc_8*, *Func_1*, and *Proc_7* as the top three of the application's functions consuming processing time. In contrast, SnoopP shows that the application functions *Proc_1*, *Proc_8*, and *Func_2* actually consume the most processing time on a MicroBlaze. Furthermore, the software implementations of the integer multiply and integer divide functions along with *main* require just over 53% of the processing time. Therefore, if the partitioning choices are based on the profiling results obtained from *gprof*, the designer would not select the appropriate functions to implement in hardware.

Similar to *gprof*, the plots of the profiles obtained using *mb-gprof* are also quite erratic relative to the highly consistent SnoopP profiles. In all three cases, the software multiply function *__mulsi3* is reported as one of the top two functions consuming the most processing time. This coincides with the data reported using SnoopP, however, the data reported for the remaining functions in the *mb-gprof* profiles do not demonstrate any strong correlation.

Based on both SnoopP profiles, the two functions requiring the largest percentage of the execution time are *__mulsi3* and *__divsi3*. These implement software versions of the integer multiply and divide functions respectively. The divide function is only called once per loop in *main* whereas the multiply function is called multiple times per loop in both *main* and *Proc_8*. Figure 6.3 illustrates how the inclusion of the hardware multiplier while still implementing the divide function in software can affect the performance profile. The Dhrystone application is reprofiled on this new MicroBlaze platform, using the same configurations of SnoopP and *mb-gprof*. Since SnoopP's profile is so consistent, only the data for Dhrystone executing 100 passes is included here.

As shown in Figure 6.3, the *gprof* and *mb-gprof* profiles are still highly erratic, and not particularly consistent with each other or with the similar profiles obtained on the previous platform using the software multiply function. For example, *Proc_7* shows the strongest correlation for the three different *mb-gprof* profiles as it is listed as one of the top three consumers of execution time in each case. However, the results obtained from SnoopP demonstrate that *Proc_7* actually requires less than 3% of the overall execution time, as shown in Table D.2, and would thus be a bad choice for implementation as a hardware function. The *mb-gprof* profiling results also suggest that the *__divsi3* function is a costly

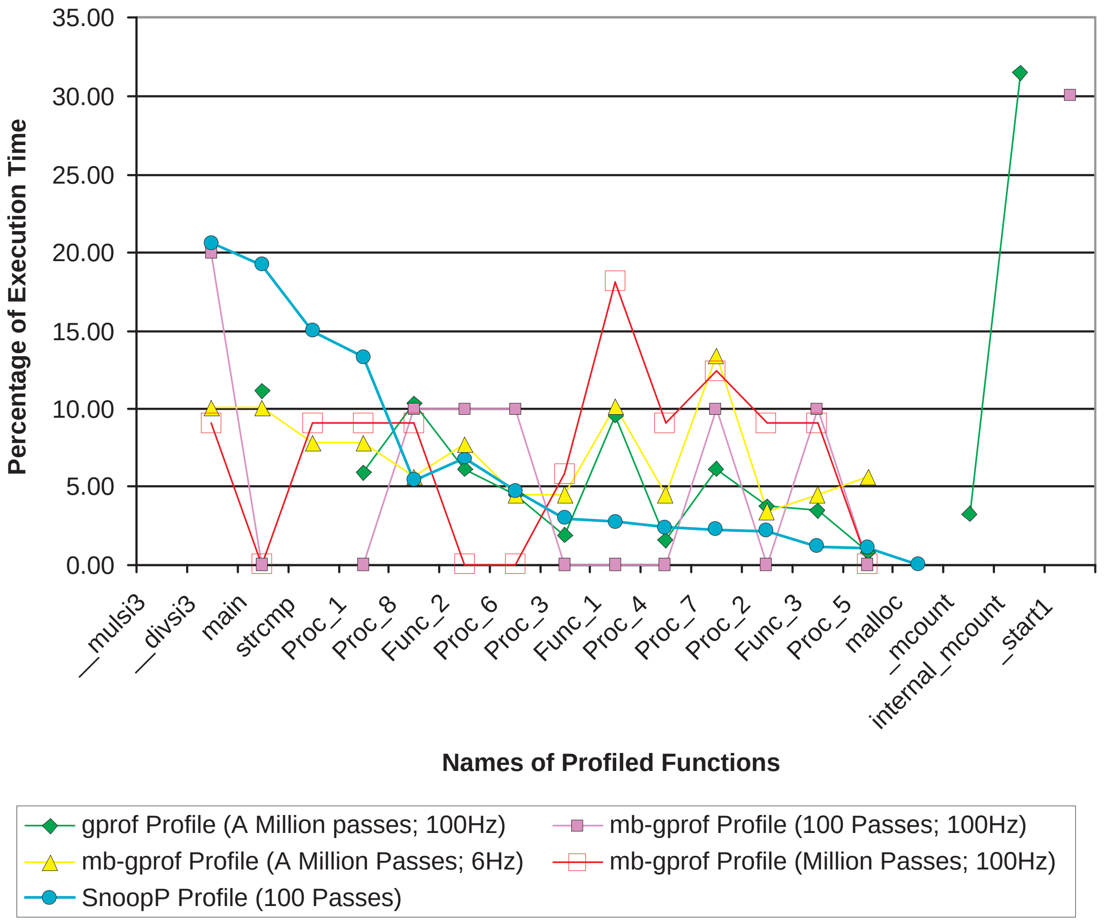

Figure 6.3: Profiling Results for Dhrystone using gprof on a Sun station and mb-gprof and SnoopP on a MicroBlaze that includes a hardware multiplier and a software implementation of the divide function.

software function as it is listed as one of the top four consumers of processing time in all three columns, whereas the remaining functions demonstrate no consistent correlation in profiled execution time.

The removal of the software multiply instruction reduces the overall instruction count by 7%, which generally increases the percentage execution time of all of the functions except for *Proc_8*. Since *Proc_8* calls *__mulsi3*, these results are not intuitive. However, the reason for the decrease in execution time is mainly due to the fact that the number of instructions in the function dropped from 60 to 33 due to optimizations that were possible with the removal of the software multiply. With respect to the tradeoff between performance improvement versus extra resources, using a hardware multiplier reduced the exe-

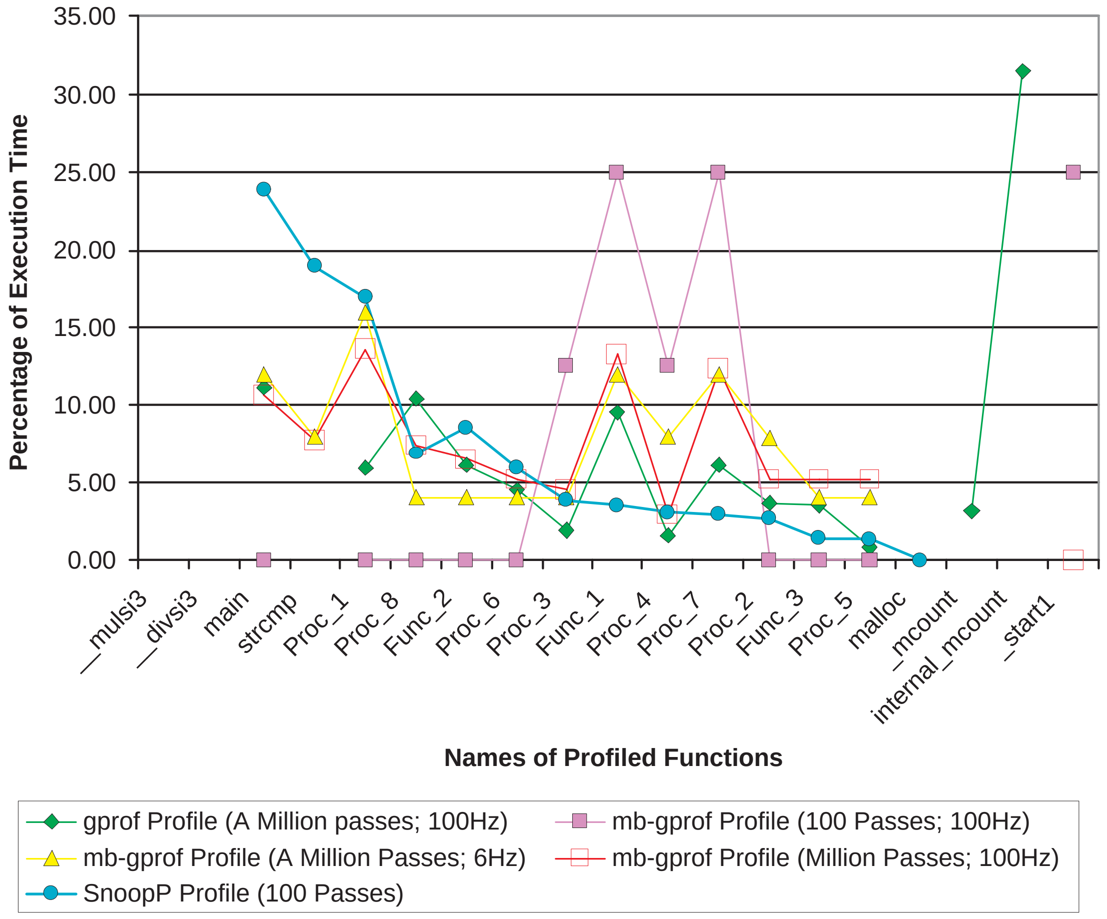

Figure 6.4: Profiling Results for Dhrystone using gprof on a Sun station and mb-gprof and SnoopP on a MicroBlaze that includes a hardware multiplier and a hardware divider.

cution time of the application from 1.33 million clock cycles to 960 thousand clock cycles, by approximately 28%. The extra resources required to implement the multiplier were 39 LUTs, 50 FlipFlops, and three 18x18 dedicated multipliers.

Given the significant improvement in performance obtained using minimal hardware resources to implement the hardware multiplier, we decided to investigate the benefits of including the hardware divider in combination with the multiplier. Figure 6.4 contains all the plots for the profiles of Dhrystone on this new MicroBlaze configuration. As before, the *gprof* and *mb-grof* plots all appear erratic with respect to the SnoopP profile. Only two functions demonstrate any correlation in terms of consistently consuming significant percentages of execution time, *Proc_7* and *Func_1*. However, as previously mentioned, *Proc_7* actually requires very little processing time. Similarly, SnoopP illustrates that *Func_1* actually uses less than 4% of the overall execution time as shown in Table D.2.

Including a hardware divider further reduced the code size of the application by another 4.5%, which increased the percentage of execution time of all the applications functions. The additional resources required to implement the hardware divider are 117 LUTs and 109 FlipFlops, but it further reduces the execution time by another 15%. Depending on the requirements of the application, the designer may feel that this is an acceptable tradeoff.

In summary, the results obtained on the FPGA with SnoopP vary from those obtained using *gprof* on the Sun station because: 1) the Sun's processor has a distinctive instruction set architecture from the MicroBlaze, and 2) statistically sampling the program to determine which functions require the longest execution time introduces inaccuracies in the *gprof* results. Consequently, SnoopP only required one hundred passes of the main loop in Dhrystone whereas a million passes were needed to obtain statistically meaningful timing information using *gprof*.

mb-gprof may be able to provide slightly better profiling data than running *gprof* on the Sun station, but this is only because it is able to profile software versions of the multiply and divide functions that are implemented in hardware on the Sun's processor. However, the overall accuracy of the profiles obtained using *mb-gprof* are still sufficiently inaccurate that using SnoopP to profile the system would provide better data to guide the designer's choice as to which functions should be implemented in hardware. Furthermore, since the time required to profile an application using SnoopP is the same as the time required to profile the application using *mb-gprof*, there is greater value in using SnoopP to obtain clock-cycle accurate results than the statistical results acquired using *mb-gprof*. Although the results from *gprof* are also statistical, the Sun station runs more than 16 times faster than the MicroBlaze. Thus, the time required to obtain these initial results to guide the assignment of the SnoopP counters is reduced to hours instead of days, which allows designers to reduce the time spent profiling the application.

6.2.3 AES

AES is a more realistic benchmark for SnoopP to profile [67]. Like Dhrystone, it also uses only integer mathematical operations. However, it is a popular design for hardware, which can greatly increase the throughput rate of the encryption.

Originally, *gprof* profiled the application with two different keys encrypting ten thousand blocks each. However, since the *internal_mcount* function tied for third in terms of processing time, the number of keys used to encrypt the ten thousand blocks was increased

Table 6.3: gprof Statistics on Functions Comprising the AES Benchmark for 2 and 400 Different Keys with 10 Thousand Blocks Each.

Function Name	**2 Keys**		**400 Keys**	
	Total Calls	**Percent Time**	**Total Calls**	**Percent Time**
MixColumns	180000	50.6	36000000	49.5
rijndaelEncrypt	20000	36.4	4000000	38.0
cipherInit	20000	5.2	4000000	2.2
internal_mcount	—	5.2	—	3.5
rijndaelCBC_MCT	1	1.3	1	1.8
_mcount	—	1.3	—	0.2
blockEncrypt	20000	0.0	4000000	4.8
HexToBin	3	0.0	—	—
makeKey	2	0.0	400	0.0
rijndaelKeySched	2	0.0	400	0.0
main	1	0.0	1	0.0
makeMCTs	1	0.0	—	—

to four hundred. The results from these different runs are found in Table 6.3. As can be seen from the table, the percentage of execution time for each function changes as the execution time is increased. Furthermore, the longer run of the executable also altered the ranking of functions incurring the longest execution time, where *blockEncrypt* now accrues a measurable percentage of the execution time while *cipherInit*'s relative execution time has decreased.

The AES executable is significantly larger than that of Dhrystone, hence it is impossible to assign individual counters to each of the functions. Instead, one counter is assigned to count all the clock cycles used in the executable and the rest are assigned to functions deemed important by the profiling results from *gprof*. Remembering that the counters will only increment when the program is inside their respective code segments, counters are also assigned to the functions called by these main functions. Table 6.4 summarizes what functions are chosen for profiling and the number of static instructions comprising each.

Figure 6.5 plots the profiles obtained using *gprof*, *mb-gprof*, and SnoopP using both two

Table 6.4: AES SnoopP Counter Assignments.

Counter Number	**Function Name**	**Number of Instructions**	**Percentage of Static Code Size**
0	Entire Program	14948	100.0%
1	MixColumns	124	0.8%
2	rijndaelEncrypt	103	0.7%
3	blockEncrypt	342	2.3%
4	cipherInit	81	0.5%
5	AddRoundKey	25	0.2%
6	Substitution	23	0.2%
7	ShiftRows	73	4.9%
8	main	144	1.0%
9	__modsi3	38	0.3%
A	__mulsi3	22	0.1%
B	__divsi3	38	0.3%
C	mul	44	0.3%
D	memcpy	11	0.1%
E	sprintf	26	0.2%
F	_vfprintf_r	2082	13.9%

and four hundred keys. There is only one plot for the SnoopP profile using both two and four hundred keys as there is no change in any of the values when the number of keys is increased (please see Appendix D for details). This reinforces the fact that the on-chip profiling with SnoopP provides valuable information with significantly less loop iterations. The majority of the execution time is spent in *__modsi3*. This function is called in *ShiftRows*, *rijndaelKeySched*, *_vfprintf_r*, and *mul*, which explains its dominance. *__modsi3* is an internal function used to implement a software version of the modulus function, similar to *__mulsi3*, and *__divsi3*.

The percentage of the total execution time measured by the SnoopP counters equalled 95.41%. Since one counter is used to measure the application's total execution time and the remaining fifteen are assigned to look at specific functions, this is reasonably good coverage. It translates into monitoring only 3176 instructions, which is 21% of the total

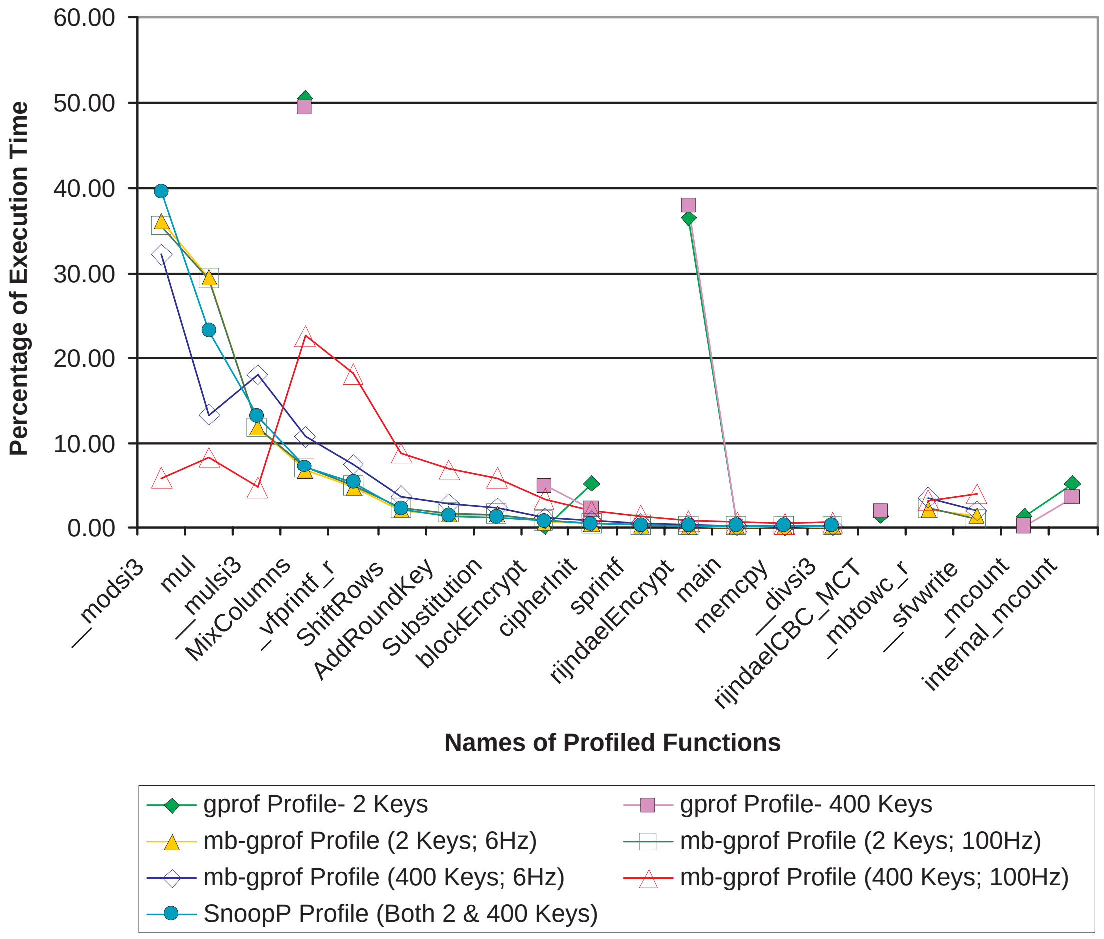

Figure 6.5: Profiling Results for AES using gprof on a Sun station and mb-gprof and SnoopP on a MicroBlaze using software implementations of the multiply and divide functions.

application to achieve over 95% of the executional time coverage.

MixColumns and *rijndaelEncrypt*, ranked first and second by *gprof* in terms of sampled execution time, are expected to require significant execution time. Although their execution does not obviously dominate the results obtained using SnoopP, the reason is that the time spent in function calls is not counted in these percentages. For example, *MixColumns* calls *__mulsi3* seven times and *mul* twice, which calls *__modsi3* once. Similarly, *rijndaelEncrypt* calls *AddRoundKey* thrice, *Substitution* twice, *ShiftRows* twice, *MixColumns* once, and *__mulsi3* once.

In this case, profiling AES demonstrates that the initial *gprof* profile can provide valuable information on potential performance hot spots at a coarse granularity, but that it may not precisely indicate the low-level cause of the increased execution time. The on-chip profiling results suggest that the first function to implement as hardware would be the modulus function. It has the largest execution time and is called from multiple functions. The other most obvious function to implement in hardware is *mul*, which requires 75% more execution time than the next most time consuming function, *__mulsi3*, the software multiply function. SnoopP's results also indicated that there will likely be little benefit to implementing the *MixColumns* or *rijndael_Encrypt* functions in hardware, contrary to the profile given by gprof.

The correlation between the profiling data for AES with two keys obtained using *mb-gprof* sampling at 6Hz and 100Hz is extremely high. Furthermore, the correlation of these results with the data obtained by SnoopP is also very high, particularly for the top five consumers of processing time. The results from profiling AES with 400 keys using *mb-gprof*, for both sampling frequencies of 6Hz and 100Hz, are also shown in Figure 6.5 These results required a little over seven days to obtain and are thus not particularly practical measurements for a system design process. However, it is rather suprising to note from the plots that these results correlate quite poorly with both the SnoopP results and each other. This suggests that while the profiling results for two keys are highly accurate, it is an unpredictable event arising from chance and circumstance.

6.3 Summary

Using SnoopP to profile a system produces consistent, fast, clock cycle accurate profiles of execution performance as demonstrated by the results of profiling Dhrystone and AES. While, *gprof* is able to obtain a basic overview of software performance, it needs numerous more loops of the main algorithm to obtain its percentage of execution time per function. Moreover, the profile is statistical and does not match the exact results measured by SnoopP. However, the initial profile from *gprof* is very useful in determining which code segments likely require the most execution time. It greatly facilitates the assignment of the SnoopP counters to the appropriate code segments.

Using *mb-gprof* on a MicroBlaze provides significantly more variable results than using *gprof*. Sometimes it is able to obtain highly accurate profiling information for an application, as shown with AES having two keys, however, sometimes this data is unreliable, as illustrated with Dhrystone. Therefore, using SnoopP instead of *mb-gprof* to obtain runtime profiling data on-chip allows users to: 1) reduce profiling time as the profiling functions added to application for *mb-gprof* increase execution time, and 2) obtain more reliable results than using *mb-gprof* as the accuracy of the profile is unpredictable.

Chapter 7

WOoDSTOCK

When creating systems, designers need not only consider the independent performance of each module in the design, but they must also ensure that there is load balancing among the CEs. If not, system stalls may be the reason a design fails to meet performance requirements. This chapter describes how Watching Over Data STreaming On Computing element linKs (WOoDSTOCK) can be used to monitor system behaviour, where Section 7.1 describes the architecture and Section 7.2 provides two case studies demonstrating how WOoDSTOCK detects potential bottlenecks.

7.1 Multi-CE Profiling Architecture

The SIMPPL model described in Section 2.3 resembles a multiprocessor system, where software designers are able to obtain some run-time statistics about an application's behaviour on their system. Of particular interest is the ability to determine the stall time of individual processors in the system. Typically, a scheduler monitors when a processor is waiting for another processing task, but as the scheduler is unaware of the nature of the actual tasks, it only provides system-level information.

WOoDSTOCK is able to provide analogous information to designers due to the standardized interface used in the SIMPPL model. WOoDSTOCK highlights problems arising from inter-CE communication and indicates to the user when a particular CE creates a system bottleneck. Like the scheduler, WOoDSTOCK is similarly unaware of the actual computation performed on a CE. Therefore, the precise cause of a system bottleneck is determined using a combination of the system performance results along with user's knowledge of the design and independent CE profiling. In the case where a software CE is causing the bottleneck, SnoopP may be used to detect the source of the bottleneck.

Figure 7.1 illustrates the connections between WOoDSTOCK and a multi-CE system. Each diamond represents a monitor that is associated with a specific CE. A monitor is a piece of hardware that records the behaviour of the traffic on all the internal input and output links connected to its CE through internal counters. These counters are used to measure the total possible stalling/starving time for a CE during the profiling period, which is set based on the program execution of a specially selected *base* processor, labeled as *CE0* in Figure 7.1, or an independent execution time counter. If the user sets the active monitoring period of the system based on the region of a *base* processor's source code, the addresses of the instructions bounding the code region are provided to WOoDSTOCK as start and stop points. If an independent execution time counter is used to select the monitoring period, then the minimum and maximum counter values act as the start and stop points. The *running* signal, shown in Figure 7.1, is enabled and disabled when the start and stop values, respectively, are seen as valid values on the counter bus. This signal is used to enable or disable the system's monitors. Each of the CEs and their monitors in Figure 7.1 are labelled for the purpose of differentiating the base processor (CE0) from the remaining CEs (1,2,3).

7.1.1 Bottleneck Detection

WOoDSTOCK assumes that the only signals a monitor can connect to are the full and empty status signals of the asynchronous FIFOs implementing the internal input and output links of its respective CE. These signals are used to generate enable signals for the counters used to profile the system. The counters are used to measure the number of clock cycles where a CE is potentially starving or stalling the system. A more naive approach would be to assign individual counters to the full and empty signals of each link in the system. However, this provides less useful information to the designer as the relationship between these status signals is required to determine if a CE is a system bottleneck as shown in the following paragraph.

Figure 7.2 illustrates examples of the three types of system bottlenecks that WOoDSTOCK can be used to detect. Figure 7.2(a) shows an *interior bottleneck,* where CE 1 has both internal input and output links and is stalling the system. To understand how WOoDSTOCK determines there is a bottleneck, consider when FIFO_1 becomes full. CE 1 may not be consuming the data produced by CE 2 fast enough. However, CE 1 may also be stalled because it cannot write to FIFO_0 if it too is full, in which case a child CE is the

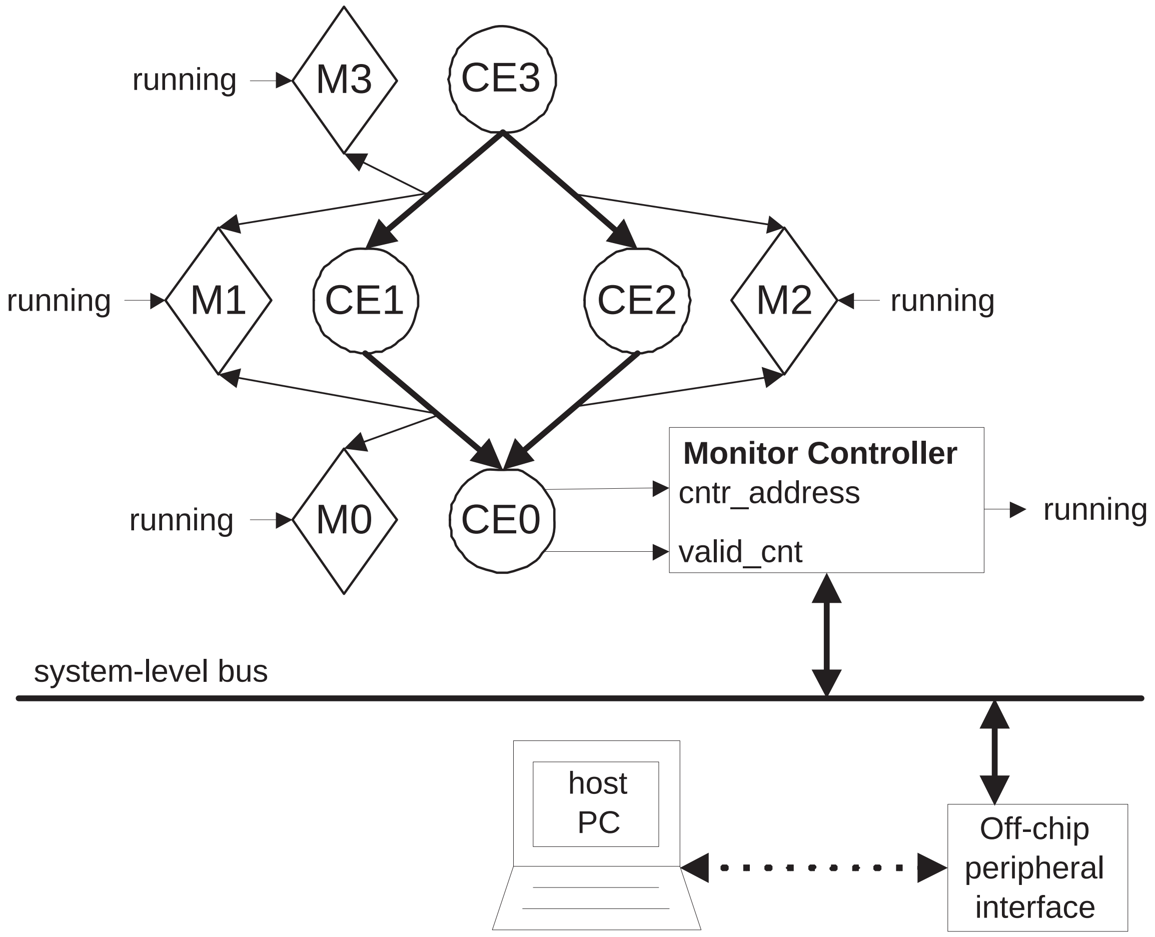

Figure 7.1: The WOoDSTOCK architecture.

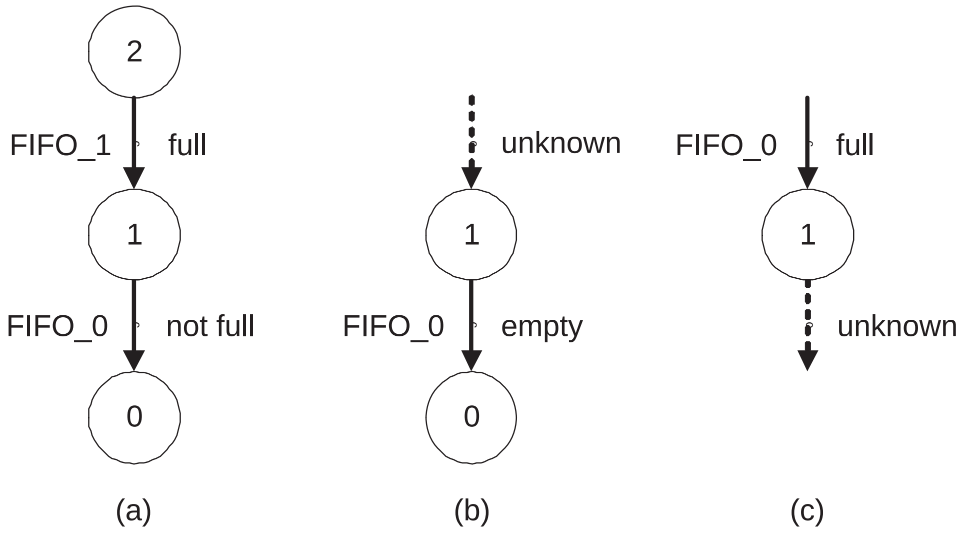

Figure 7.2: Examples of the different types of bottlenecks detectable by WOoDSTOCK: (a) interior bottleneck, (b) input bottleneck, and (c) output bottleneck.

Table 7.1: Example output equations for the systems in Figure 7.2.

Bottleneck Example	**Output Equation**
Interior Bottleneck	FIFO_1_full and (not FIFO_0_full)
Input Bottleneck	FIFO_0_empty
Output Bottleneck	FIFO_0_full

bottleneck and not CE 1. To differentiate between these situations, a CE is defined to be an interior bottleneck when all the input links that provide data to generate a specific output are full and the link at the output is not full as depicted in Figure 7.2(a). The specification of the output link as "not full", as opposed to empty, delineates an important aspect of the system monitoring tool. WOoDSTOCK is unaware of the nature of the data being transferred between CEs, so if CE 1 produces a data packet that CE 0 requires in its entirety to continue processing, then the link should normally be empty when the system is balanced. However, if CE 1 produces a data packet that is consumed as multiple individual data packets by CE 0, then there will normally be data in this FIFO even when the system is balanced. Therefore, the output link must be only "not full" instead of "empty" to produce a bottleneck.

A CE that has internal output links and no internal input links may cause an *input bottleneck*. This occurs when either the off-chip hardware peripheral supplying input to the CE is too slow or the processing time of the CE is too slow. In either case, the system is starved for data. To detect this situation, WOoDSTOCK monitors the empty status signal of the output link. Figure 7.2(b) shows CE 1 as the potential cause of an input bottleneck. The status of the I/O communication link is unknown and FIFO_0 is empty. However, CE 1 may not be a bottleneck if CE 0 consumes data at the same rate as CE 1 produces it. This situation would also cause FIFO_0 to be empty for the majority of the system's run-time. Since the results from these measurements are not conclusive on their own, the designer needs to see how this information fits in with the results obtained from monitoring the rest of the system.

Output bottlenecks arise in CEs that have internal input links and no internal output links. They occur due to the slow processing rate of either the CE or an off-chip peripheral. Both cases result in the input links to the CE becoming full as illustrated in Figure 7.2(c). While the state of the I/O communication links is unknown, FIFO_0 becomes full stalling the system. In situations where a CE stalls or starves because of an off-chip peripheral's slow data rate, this is still measured as being caused by the CE implementation. Therefore, the user must be sufficiently familiar with the CE's processing to determine the precise cause of the bottleneck.

To generate a system-specific monitoring system, the user writes a description of the system that states the required combination of data on internal input links used to produce an output for a given output link. WOoDSTOCK uses this information to create an *output equation* for each CE output described in terms of link empty and full status signals. Table 7.1 shows the appropriate output equations for CE 1 in each of the systems in Figure 7.2. These equations generate counter specific enable signals that are combined with the *running* signal to enable all the appropriate counters during each sampling clock cycle.

The frequency of WOoDSTOCK's sampling clock can be set to any rate, depending on the desired profiling accuracy. If sampling is done using the fastest system clock, then the measured results are precise. However, a slower clock may be used to do the sampling and obtain a statistical measurement of system performance. This information can still help detect system bottlenecks, but the system may need to be profiled for longer run-times to observe the problem.

7.1.2 Implementation and Design Decisions

WOoDSTOCK generates the necessary system dependent VHDL files to implement the monitoring system, along with the files required by XPS to interface WOoDSTOCK into a MicroBlaze system as shown in Figure 7.3. The internal structure of WOoDSTOCK is subdivided into two components — the system monitors and the OPB interface. The former profiles the system links based on the user-provided system profile (recall Figure 7.1) while the latter provides off-chip access to their counter values. Similar to SnoopP, WOoDSTOCK is memory mapped to the OPB as a slave device. It uses the MDM module as an off-chip interface to the xmd control window running on a host computer allowing users to remotely read and reset the counters.

While WOoDSTOCK does not differentiate between monitoring hardware and software CEs, the MDM module requires that there be at least one MicroBlaze processor in the system. However, systems are often comprised of a combination of hardware and software, which means there should be at least one processor in the system to fulfill the MDM's requirements. WOoDSTOCK also connects to the FIFO status signals that indicate when the FSL is empty and when the FSL is full. These signals will be referred to as *fsl_empty* and *fsl_full* respectively. By monitoring their runtime values, WOoDSTOCK enables the appropriate counters based on the user-defined output equations.

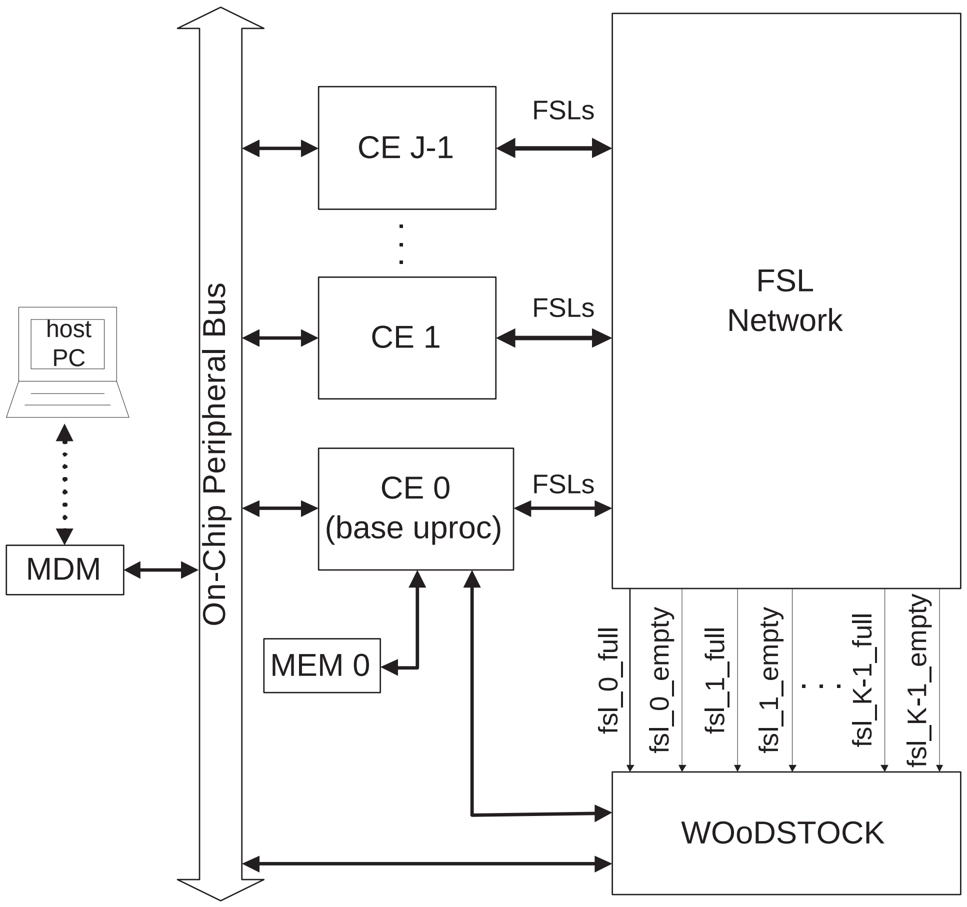

Figure 7.3: The interface of WOoDSTOCK with the Fast Simplex Link (FSL) network of a multi-CE system and a host PC via the Microprocessor Debug Module.

The objective is to make the WOoDSTOCK circuit as small and as fast as possible so that it does not impact the embedded system design. However, to be a useful system profiler, it must allow the user flexibility to assign the appropriate number of counters for the system. The decisions outlined below are an attempt to balance these considerations.

The size of the overall circuit depends mostly on the number of counters required to store the system profiling data. The counter size is again set to 46-bits, however, the maximum clock speed for the monitoring system is dependent on the complexity of the system being monitored. Additional logic resources are used to generate the running signal for the counters from two 32-bit comparators for the start and stop counter values, which are hardwired for the system to reduce the required logic.

Similarly, the OPB interface needs two comparators to determine if the user has accessed WOoDSTOCK's address space plus logic to multiplex the selected counter onto the OPB. Finally, the logic required to generate the counter specific enable signals depends on

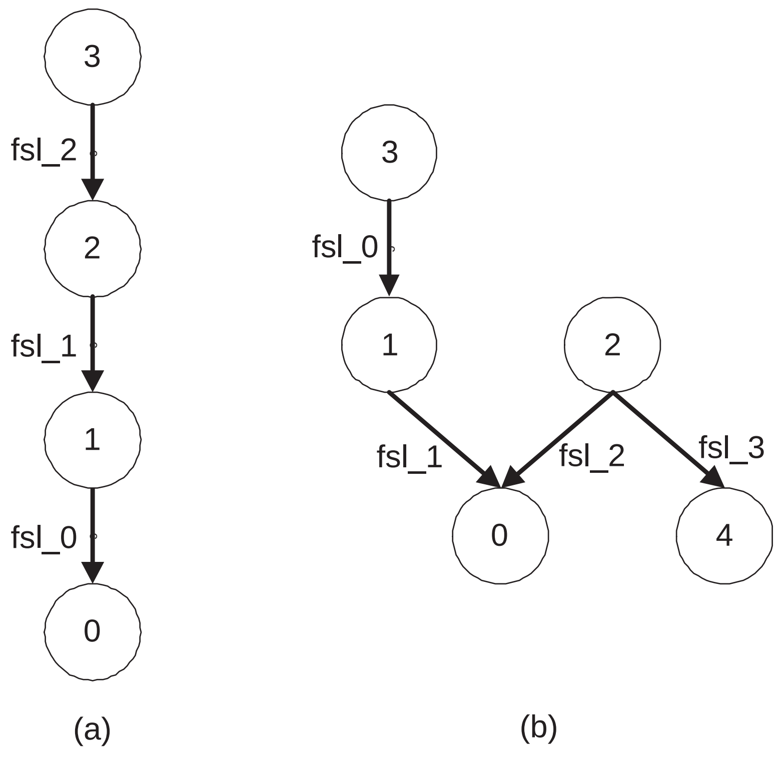

Figure 7.4: Two application architectures described with the SIMPPL model: (a) a pipelined system (b) a system with branching.

the complexity of each output equation. However, the logic resources used are negligible relative to the size of a counter.

7.2 Case Studies

This section uses WOoDSTOCK in two different case studies to show that the measurement approach works and to demonstrate how the information it provides can help to refine a design. It details the issues encountered while profiling each system and concludes with a discussion of the advantages of on-chip system profiling.

7.2.1 Methodology

The two benchmarks illustrated in Figure 7.4 are the case studies used to demonstrate the functionality of WOoDSTOCK. We use the System Generator described in Section 5.3.2 to generate benchmarks. The CEs in these benchmarks are implemented using the default version of the MicroBlaze soft processor. Each MicroBlaze has eight built-in FSL (FIFO)

Table 7.2: Table for pipelined system counter results describing the counter enables, what the counters represent, and reporting the measured results as percentages of the total monitor run time given in Counter 4 to the nearest million clock cycles.

Cntr	Enable Condition	Possible Meaning	20 Data Packets			100 Data Packets			200 Data Packets		
			Con A	Con B	Con C	Con A	Con B	Con C	Con A	Con B	Con C
0	fsl_0_full	CE 0 slow	0	0	0	0	0	0	0	0	0
1	fsl_1_full and (not fsl_0_full)	CE 1 slow	0	0	0	0	0	0	0	0	0
2	fsl_2_full and (not fsl_1_full)	CE 2 slow	0	0	0	0	0	0	0	0	30.8
3	fsl_2_empty	CE 3 slow?	100.0	100.0	23.6	100.0	100.0	5.2	100.0	100.0	2.7
4	running	monitors on	688	368	366	3248	1648	1646	6448	3248	3246

receive and transmit ports and the send and receive functions are generated based on macros provided by Xilinx to read and write from these ports. The time required for a CE to process input data to produce output data is modelled using the *delay* parameter in the main loop of the generated source code template. The source code for each MicroBlaze is then compiled and stored in its local on-chip memories and accessed via Local Memory Buses. Each system configuration is profiled for varying lengths of time to determine the initial effects of system start up on the results. The main processing loop of the base processor uses a *for* loop to set the number of data packets it consumes. Therefore, we can vary the profiling period is by changing the upperbound of the *for* loop.

The first benchmark is a simple pipelined design that is quite common to hardware design. The second is an imaginary system used to highlight the increasing difficulties of analyzing a design that is less intuitive. For both benchmarks, WOoDSTOCK uses the global system clock as its sampling clock. Different configurations of each system are created by varying the delays used to model CE processing times. These processing delays are used to create system imbalances that WOoDSTOCK should report as well as balanced systems to determine how this affects the results obtained by WOoDSTOCK.

7.2.2 Pipelined System Example

The pipelined system in Figure 7.4(a) requires 5 counters to monitor the system. The first three columns of Table 7.2 list the counters, the output equations used to generate their respective enables, and the possible meaning of these conditions. A question mark in Column 3 indicates that this CE may actually not be the source of any system performance problems as it only represents a possible input bottleneck. Counters 0, 1, and 2 monitor FSL full signals to determine if CEs 0, 1 , or 2, respectively, are stalling the system. Counter 3 counts the number of clock cycles for which fsl_2 is empty and Counter 4 stores the total run-time of the monitoring system. Configuration *A* has equal processing delays for all the CEs except for CE 3. Its delay is twice as long as the rest to model a situation where CE 3 requires twice the processing time of the rest of the CEs so CE2 should be starved for data. The *for* loop of CE 0, the base processor, is then set to consume 20, 100, and 200 data packets, respectively to create profiles of varying length. While WOoDSTOCK is able to obtain more accurate information about system performance as the profiling time is increased, the consumption of 200 data packets is sufficient to demonstrate WOoDSTOCK's functionality for the different configurations of the pipelined and branching systems.

The results for Configuration A are found in Table 7.2 in the subcolumns labelled *Con A*. All values in the configuration columns are reported as the percentage of the monitor run-time, which is given to the nearest million clock cycles in the final row of the table (Counter 4). Anytime a counter's value was actually zero, the percentage is reported as *0*, whereas if the value is simply negligible relative to the profiling period, the percentage is reported as *0.0*. This same method is applied to counters that run for almost the entire profiling period, *100.0* versus *100*. As illustrated by the table, the only counter to be incremented monitors when there is no data in fsl_2. Without any knowledge of individual CE behaviour, it is impossible to determine if the system is balanced or if CE 3 is too slow and starving the system. In either case, the processing time of CE 3 needs to be reduced if the designer wishes to try and improve system performance.

The second system configuration reduces CE 3's processing delay by 50% so that all the CEs have equal delays for processing time. This should balance the pipelined system, yet, the results in Table 7.2, in the subcolumns labelled *Con B*, show that the third counter is still enabled for almost 100% of the monitor run-time even though the system should now be balanced. This is because CE 2 consumes data at the same rate as CE 3 produces, thus fsl_2 remains empty most of the time. Instead, the decreased total run-time for the system in Configuration B proves that CE 3 is an input bottleneck in Configuration A. The overall run-time is reduced by approximately 50%, mirroring the decrease in CE 3's processing

Table 7.3: Table for branching system counter results describing the counter enables, what the counters represent, and reporting the measured results as percentages of the total monitor run time given in Counter 7 to the nearest million clock cycles.

Cntr	Enable Condition	Possible Meaning	20 Data Packets			100 Data Packets			200 Data Packets		
			Con A	Con B	Con C	Con A	Con B	Con C	Con A	Con B	Con C
0	fsl_1_full	CE 0 slow	21.4	10.1	0	83.7	82.2	0	91.8	91.1	0
1	fsl_2_full	CE 0 slow	0	0	0	0	0	0	0	0	0
2	fsl_0_full and (not fsl_1_full)	CE 1 slow	0	0	0	0.0	0.0	0	0.0	0.0	0
3	fsl_2_empty	CE 2 slow?	2.4	94.9	2.3	0.50	99.0	0.5	0.2	100.0	0.2
4	fsl_3_empty	CE 2 slow?	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
5	fsl_0_empty	CE 3 slow?	83.3	94.9	100.0	17.3	18.8	100.0	8.7	9.4	100.0
6	fsl_3_full	CE 4 slow	0	0	0	0	0	0	0	0	0
7	running	monitors on	672	632	352	3232	3192	1632	6432	6392	3232

delay. Since no other bottlenecks have appeared in the system, the designer may not realize that the system's performance has been maximized if they are insufficiently aware of the individual processing requirements of each CE.

If the user assumes that CE 3 is still an input bottleneck to the system, they may choose to further reduce its processing delay. Configuration C allows CE 3 to run faster by decreasing the processing delay to 90% of the processing delay used by the rest of the system. As can be seen from the data in the subcolumns labelled *Con C*, for smaller run-times, no bottlenecks are introduced into system communications. In fact, the percentage of time for which there is no data in fsl_2 decreases to 23.6% when the base processor consumes only 20 data packets, compared to the other two configurations where there is no data in fsl_2 for almost 100% of the run-time. However, as the system continues to run, fsl_2 becomes full as CE 2 acts as an interior bottleneck because it cannot consume data as quickly as CE 3 produces it. This is reflected in the value of Counter 2 when the base processor consumes 200 data packets and highlights the importance of running systems for long periods of time to achieve a more steady-state view of the system. Furthermore, if CE 3 had still been an input bottleneck to the system, the decrease in the processing delay of CE 3 should have been mirrored in the total run-time of the system, which remained almost unchanged from Configuration B to Configuration C.

7.2.3 Branching System Example

Figure 7.4(b)'s branching system requires eight counters that are enabled based on the functions described in Column 2 of Table 7.3 when the monitors are running. The results in the table are presented following the same format as Table 7.2. Counters 0 and 1 monitor fsl_1 and fsl_2 to determine if CE 0 is stalling the system. Similarly, counters 2 and 6 measure when CE 1 and CE 4, respectively, stall the system. Counters 3 and 4 count the number of clock cycles for which fsl_2 and fsl_3 are empty as does counter 5 for fsl_0. This information can help to determine if either CE 2 or CE 3 are producing output data too slowly, and thus starving their respective children CEs. The possible interpretations for the counter values are summarized in Column 3.

In this system, each data packet to and from each link is processed independently. For example, in CE 2 an output is generated for fsl_2 after a processing delay and an output is generated for fsl_3 after a separate processing delay. Therefore, for CE 2, the time between generating outputs for fsl_2 is the sum of these two delays. Similarly, in CE 0, data words are read from fsl_1 followed by a processing delay before data words are read from fsl_2 followed by an independent processing delay. In this case, for CE 0, the time between reading inputs from fsl_1 is the sum of these two delays. The first configuration of this system has all of the processing delays for each link set to the same value. This creates an imbalanced system as CE 0 and CE 2 have an effective per link processing delay that is twice that of the other CEs. Again, the base processor's *for* loop is set to consume 20, 100, and 200 data packets, which is sufficient to demonstrate the system imbalances for the following configurations.

Table 7.3 summarizes the results for Configuration A in the subcolumns labelled *Con A* where all values are in terms of the percentage of the monitor's run-time for which the counter was enabled. The total profiling period is reported to the nearest million clock cycles in Counter 7's row. The importance of running the system for a significant period of time is highlighted by the results for counter 0, which vary from 21.4% to 91.8%. The larger value from the long run-time clearly indicates that CE 0 is stalling the system by not consuming data quickly enough.

To try and remove this bottleneck, CE 0's processing delays for input data read from fsl_1 and fsl_2 are reduced to 50% of the delays for the rest of the system. This means that the combined effective per link processing delays for fsl_1 and fsl_2 are now the same as the rest of the system, with the exception of CE 2's processing delays, which are left unchanged. The results for this configuration are found in Table 7.3 in the subcolumns labelled *Con B*. From these results, it appears that CE 0 is still stalling the system, however closer inspection disproves this theory. While fsl_1 is still becoming blocked as the run-time increases, the period for which fsl_2 is empty has increased dramatically (see Counter 3). This may indicate that CE 2 cannot keep up with its child nodes. If this is the case, CE 0 is now starved for data on fsl_2 and still not able to keep up with its parent node CE 1. This also is reflected in the overall run-time that remains basically unchanged between Configuration A and Configuration B as the profiling period increases. If CE 0 were the only bottleneck in the system, the system's performance should have increased noticeably. Therefore, CE 2 must also be a system bottleneck, failing to provide data at the necessary production rate.

By reducing CE 2's processing delay for generating outputs for fsl_2 and fsl_3 to 50% of the original processing delay, the system should be balanced. This is designated as Configuration C and the results are found in the subcolumns labelled *Con C* in Table 7.3. In this case, none of the links become full so the system never stalls. This produces the expected increase in the overall system performance by decreasing the overall run-time by approximately 50% from the Configuration *A*.

7.3 Summary

WOoDSTOCK is able to detect bottlenecks in system performance and the removal of these bottlenecks dramatically improves the overall performance as demonstrated in the above examples. WOoDSTOCK required 579 LUTs and 331 flipflops to monitor the pipelined example and 928 LUTs and 478 flipflops to monitor the branching example. If these results are normalized in terms of the number of counters in each system, the pipelined example uses 115.8 LUTs and 66.2 flipflops per counter and the branching example uses 116 LUTs and 59.8 flipflops per counter. These results highlight that the increased size of WOoDSTOCK is mainly due to the extra counters and that overhead logic needed to provide a user interface can be considered minimal.

The system must be run for a significant period of time to obtain accurate results using WOoDSTOCK. This may be on the order of minutes to hours depending on system complexity, and is necessary to account for the initial effects of starting up the system. If these results are to be found via simulation, the required time could be excessive. Although

WOoDSTOCK obtains only a macroscopic view of system performance, combined with an understanding of the individual CEs, it provides greater insight into system behaviour that can guide the redesign of a system. Finally, while a designer should be sure that there are no CEs stalling the system, interpreting the meaning of the measured results for more complex systems requires that the Counter values not be viewed in isolation as demonstrated in the branching example.

Chapter 8

Conclusions and Future Work

Technological advancements enable designers to create increasingly complex SoCs on ASICs, increasing both the design time and cost for developing electronics. IP reuse is considered a possible solution for reducing both design time and complexity, but the lack of a universally accepted standard limits the possible benefits. Thus, different groups are attempting to develop a standard, however, there are many barriers to creating a unifying solution. Now that FPGAs are also large enough to implement these complex SoC designs, FPGA companies are providing both IP and system design tools to try and facilitate the design process. However, SoC design design methodologies do not exploit the benefits of a reconfigurable implementation platform.

8.1 Conclusions

The contributions of this thesis are divided into two areas. The first is an architectural framework to facilitate design reuse and system integration for SoCs implemented on ASICs or FPGAs. The second is the development of a set of design tools for a design infrastructure to leverage the benefits of designing a reconfigurable platform.

8.1.1 SoC Architecture

This thesis introduced the SIMPPL model for SoC designs implemented on both ASIC and FPGA platforms. Systems are modelled as a network of Computing Element(s) connected via asynchronous FIFOs. The CE abstraction decouples the system-level control from the Processing Element and provides a fixed communication interface and protocols to the rest of the system. The SIMPPL controller acts as the system interface and processes the instructions that allow the designer to program the use of a PE within the system. The current instruction set is limited to minimize the size of the controller by supporting instructions that are required to transfer data between CEs, however, it is extensible to support the needs of future applications. The execute controllers run at approximately 280 MHz on an FPGA and 1.56 GHz on an ASIC in 90 nm technology, whereas the debug controllers run at 160 MHz and 1.09 GHz, respectively. The standard cell implementation for the controllers ranges from 5 251 um^2 to 29 615 um^2 depending on which type and version of the controller is used and whether it is optimized for speed or area. The sizes of the controllers can be further reduced if they are implemented as custom cells.

The usage of SIMPPL controllers as the physical and communication protocol interface between CEs incurred latency and area overhead for the designs. However, they greatly facilitated system-level design by reducing complexity and simplifying the reprogramming of CEs for different applications. For example, the system integration time for each of the SIMPPL modelled systems was less than 20 hours compared to the 140 hours required for the custom designed video streaming system.

Besides reducing system integration time, the SIMPPL model facilitates debugging at both coarse and fine-grain levels. The fixed internal communication links simplify the design of on-chip testbeds that allow CEs to be tested with a large number of vectors in real time to verify the PE's functionality. To detect low-level programming errors, we have created a debug version of each of the three types of controllers that provides access to the run-time status of the controller when an error occurs.

8.1.2 SoC Design Tools

This research also resulted in the development of an SoC design methodology tailored to FPGAs. Since the FPGA fabric is easily reprogrammed, it allows a design methodology that incorporates on-chip design tools. The benefits accrued from such an approach are similar to those experienced by software designers who typically design on a processor-based platform as opposed to a processor simulator. Designers can obtain accurate results quickly using tools that are tailored to their specific design. Moreover, running a design on an FPGA is orders of magnitude faster than simulating it, allowing a larger number of test vectors to be used to verify functionality.

Assuming designers create their SoC architecture using the SIMPPL framework, it is possible to develop an on-chip design infrastructure to support this design methodology.

To date, the tools created to leverage the architectural format and the FPGA's reprogrammable fabric include an on-chip testbed, two on-chip profiling tools, and a system-level specification tool to facilitate system-level integration.

8.2 Future Work

Currently, the SCSs are handwritten for each application. The three types of debug and execute controllers have also been custom designed. Future work will investigate the development of a controller specification platform and a high-level programming environment.

The on-chip testbed currently uses soft processors to provide the test vectors and store the resulting outputs. While this allows designers to update the testbed without resynthesizing the design, soft processors include functionality unnecessary for the testbed that require extra resources. Therefore, a possible improvement for the on-chip testbed is to develop a debugging module having a source and sink unit that requires less logic resources than a MicroBlaze. It should also be modular and scalable to system-level requirements and still allow users to download different test vectors without necessitating resynthesis of the bitstream. Another possible extension for the testbed work is to develop a post-silicon debug infrastructure for SIMPPL SoCs implemented as ASICs

Finally, this research also demonstrated that on-chip profiling tools quickly obtain accurate results that can be used by the designer to make better design decisions to reduce design time. Future work should investigate possible design implementation improvements for these tools, such as pipelining SnoopP to increase the operating frequency. New on-chip tools that monitor other runtime characteristics, such as power consumption, should also be investigated. Another possibility is to consider the incorporation of the current SoC design tools into an embedded SoCs design methodology. It should be possible to automate more of the process by creating new tools that utilize the on-chip profiling results to generate a new partitioning, mapping, and scheduling for an application on a newly generated system architecture.

Bibliography

[1] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, Norwell, Massachusetts, 1998.

[2] H. Chang, L. Cooke, M. Hung, G. Martin, A. J. McNelly, and L. Todd. *Surviving the SOC revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, Norwell, Massachusetts, 1999.

[3] Altera's Home Page. Online: http://www.altera.com.

[4] Xilinx's Home Page. Online: http://www.xilinx.com.

[5] K. S. Hemmert, J. L. Tripp, B. Hutchings, and P. A. Jackson. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 228–237, April 2003.

[6] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood, and R. Cytron. Cycle-Accurate Microarchitecture Performance Evaluation. In *Proceedings of the Workshop on Introspective Architecture*, February 2006.

[7] L. Shannon and P. Chow. Maximizing System Performance: Using Reconfigurability to Monitor System Communications. In *IEEE Int. Conference on Field-Programmable Technology*, pages 231–238, December 2004.

[8] L. Shannon and P. Chow. Using Reconfigurability to Achieve Real-Time Profiling for Hardware/Software Codesign. In *ACM Int. Symposium on Field-Programmable Gate Arrays*, pages 190–199, February 2004.

[9] L. Shannon, B. Fort, S. Parikh, A. Patel, M. Saldana, and P. Chow. A System Design Methodology for Reducing System Integration Time and Facilitating Modular Design Verification. In *IEEE International Conference on Field-Programmable Logic and Applications*, pages 289–294, August 2006.

[10] W. Savage, J. Chilton, and R. Camposano. IP Reuse in the System on a Chip Era. In *Proc. from the 13th Int. Symposium on System Synthesis*, pages 2–7, September 2000.

[11] G. Martin. Design methodologies for system level IP. In *Proc. of IEEE Conference on Design Automation and Test in Europe*, pages 286–302, February 1998.

[12] The Spirit Consortium Home Page. http://www.spiritconsortium.org.

[13] VSIA Home Page. Online: http://www.vsia.org.

[14] T. Lee and N. W. Bergmann. An Interface Methodology for Retargettable FPGA Perihperals. In *Int. Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 1–7, July 2003.

[15] Xilinx. OPB IPIF Architecture, 2003. Online: http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_ipif.pdf.

[16] P. J. Aldworth. System-on-a-Chip Bus Architecture for Embedded Applications. In *Proc. IEEE Int. Conf. on Computer Design*, pages 297–298, October 1999.

[17] D. Flynn. AMBA: Enabling reusable on-chip design. *IEEE Micro*, 17(1):20–27, July 1997.

[18] IBM Corporation. The CoreConnect bus architecture. Online: www.ibm.com/chips/products/coreconnect.

[19] OpenCores. Specification for the WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP cores, Revision B.3, September 2002. Online: http://www.opencores.org/projects.cgi /web/wishbone/wbspec_b3.pdf.

[20] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract. In *Int. Conference on Field Programmable Logic and Applications*, pages 605–614, August 2000. (For more information on SCORE, goto http://brass.cs.berkeley.edu/documents/ score_tutorial.pdf).

[21] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier. An Architecture and Compiler for Scalable On-Chip Communication. *IEEE Transactions on VLSI Systems*, 12(7):711–726, July 2004.

[22] A. Adrianhantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino. Spin: A Scalable, Packet Switched, on-Chip Micro-Network. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 70–73, March 2003.

[23] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance Evaluation and Design Trade-offs for Network-on-Chip. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005.

[24] S. Kumar, A. Jantsch, J. Soininen, and M. Forsell. A Network on Chip Architecture and Design Methodology. In *Proceedings of the IEEE Annual Symposium on VLSI*, pages 105–112, 2002.

[25] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnect networks. In *Proc. for the ACM/IEEE Design Automation Conference*, pages 684–689, June 2001.

[26] G. Brebner and D. Levi. Networking on Chip with Platform FPGAs. In *IEEE Int. Conference on Field-Programmable Technology*, pages 13–20, December 2003.

[27] M. Saldana, L. Shannon, and P. Chow. The Routability of Multiprocessor Network Topologies in FPGAs. In *IEEE/ACM Workshop on System Level Interconnect Predictions*, March 2006.

[28] P. Magarshack and P. G. Paulin. System-on-chip Beyond the Nanometer Wall. In *Proceedings of 40th Design Automation Conference*, pages 419–424, June 2003.

[29] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IPIF Congress 74*, 1974.

[30] E. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[31] E. Kock, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, K. Vissers, and G. Essink. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of the 37th Design Automation Conference*, pages 402–405, June 2000.

[32] K. Jasrotia and J. Zhu. Stacked FSMD: A Power Efficient Micro-Architecture for High Level Synthesis. In *International Symposium on Quality Electronic Design*, pages 425–430, March 2004.

[33] I. Kuon and J. Rose. Measuring the Gap between FPGAs and ASICs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 21–30, February 2006.

[34] D. Chinnery and K. Keutzer. Closing the Gap Between ASIC and Custom: An ASIC Perspective. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[35] W. Dally and A. Chang. The Role of Custom Design in ASIC Chips. In *Proceedings of the 37th Design Automation Conference*, pages 643–647, June 2000.

[36] MEDEA+ EDA Roadmap 2003: Executive Summary Europe. Online: http://www.medea.org/webpublic/ publications/publ_relation_eda.htm.

[37] E. Salminen, A. Kulmala, and T.D. Hamalainen. HIBI-based multiprocessor SoC on FPGA. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2005.

[38] P. Huerta, J. Castillo, J.I. Martnez, and V. Lopez. A MicroBlaze based Multiprocessor SoC. *WSEAS Transactions on Circuits and Systems*, 4(5):423–430, May 2005.

[39] K. Ravindran, N. Satish, Y. Jina, and K. Keutzer. An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 487–492, August 2005.

[40] M. Horauer, F. Rothensteiner, M. Zauner, E. Armengaud, A. Steininger, H. Friedl, and R. Pallierer. An FPGA based SoC Design for Testing Embedded Automotive Communication Systems employing the FlexRay Protocol. In *Proceedings of the Austrochip Conference*, October 2004.

[41] S. Kimura, Y. Itou, and M. Hirao. A Hardware/Software Codesign Method for a General Purpose Reconfigurable Co-Processor. In *5th International Workshop on Hardware/Software Co-Design (Codes/CASHE '97)*, March 1997.

[42] J. Fleischmann, K. Buchenrieder, and R. Kress. Codesign of Embedded Systems Based on Java and Reconfigurable Hardware Components. In *Design Automation and Test in Europe*, March 1999.

[43] O. Hebert, I. C. K., and Y. Savaria. A Method to Derive Application-Specific Embedded Processing Cores. In *Proceedings of the Eighth International Symposium on Hardware/Software Codesign*, May 2000.

[44] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Design Automation Conference*, June 2000.

[45] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[46] G. Stitt, R. Lysecky, and F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. In *Proceedings of the 40th Design Automation Conference*, pages 250–255, June 2003.

[47] D. N. Rakhmatov and S. B. K. Vrudhula. Hardware/Software Bipartitioning for Dynamically Reconfigurable Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[48] J. Noguera and R. M. Badia. Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[49] Altera's SOPC Builder. Online: http://www.altera.com/products/software/system/products/sopc/sop-index.html.

[50] Altera's Quartus II. Online: http://www.altera.com/products/software/pld/products/q2/qts-index.html.

[51] Model Technology Home Page. Online: http://www.model.com.

[52] First Silicon Solutions Home Page. Online: http://www.fs2.com.

[53] Xilinx's EDK Design Tools. Online: http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[54] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, Dordrecht, The Netherlands, 1997.

[55] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, September 1993.

[56] Mentor Graphic's Seamless Co-verification Simulator. Online: http://www.mentor.com/seamless.

[57] PAPI's Home Page. Online: http://icl.cs.utk.edu/projects/papi/.

[58] S. Browne, J.Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3), 2000.

[59] W. Korn, P.J. Teller, and G. Castillo. Just how accurate are performance counters? In *20th IEEE International Performance, Computing, and Communications Conference*, April 2001.

[60] B. Sprunt. Pentium 4 Performance-Monitoring Features. *IEEE Micro*, 22(4), 2002.

[61] GNU's Not Unix! The GNU Project and Free Software Foundation (FSF). Online: http://www.gnu.org.

[62] GNU gprof Manual. Online: http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[63] MicroBlaze Processor Reference Guide. Online: http://www.xilinx.com/bvdocs/ipcenter/data_sheet/FSL_V20.pdf.

[64] T. Rissa, W. Luk, and P. Cheung. Automated Combination of Simulation and Hardware Prototyping. In *Int. Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004.

[65] M. Finc and A. Zemva. Profiling soft-core processor applications for hardware/software partitioning. *Journal of Systems Architecture*, 51(5):315–329, May 2005.

[66] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10), 1984.

[67] J. Daemen and V. Rijmen. Rijndael, the Advanced Encryption Standard. *Dr. Dobb's Journal*, 26(3), 2001.

Appendix A

SIMPPL Controllers HDL Source Code

A.1 Full Instruction Set

```
//***********************************************************************
//    File: instr_defines.v
//    Used by: All SIMPPL Controller modules and their Control Sequencers
//    Description: Defines the SIMPPL Controller Opcodes.
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//***********************************************************************


//Create constants for all the instruction opcode:
‘define reset 8’h00
‘define noop 8’h04

‘define readA_Imm 8’h01
‘define readR_Imm 8’h02
‘define write_Imm 8’h03

‘define readA_Abs_addr 8’h05
‘define readR_Abs_addr 8’h06  // These first 8 instructions will be
‘define write_Abs_addr 8’h07  // read and transmitted.

‘define read_Areg 8’h0A //NOT IMPLEMENTED YET!!!
‘define write_Areg 8’h0B

‘define add_imm_Areg 8’h18
‘define sub_imm_Areg 8’h08
‘define bypass 8’h0C
//‘define wait_rx 8’h0C //deprecated and removed

‘define readA_Areg_indirect 8’h0D
‘define readR_Areg_indirect 8’h0E
‘define write_Areg_indirect 8’h0F

‘define readA_Areg_autoinc 8’h1D
‘define readR_Areg_autoinc 8’h1E
‘define write_Areg_autoinc 8’h1F
```

A.2 Consumer Execute Controller

```
//**********************************************************************
//    File: fsl_ce_for_fifo_plus.v
//    Uses: fifo_plus.v and a SIMPPL Control Sequencer
//    Descriptions: Provides a SIMPPL interface to a 16 deep 32-bit word
//     synchronous FIFO (SRL)
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//**********************************************************************

'include "instr_defines.v"

module consumer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  // Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  // Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface
  pe_rst,
  pe_data_in,
  pe_write_data,
  pe_write_addr,
  pe_done_packet,
  pe_can_write_data,
  pe_can_write_addr,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  exec_bypass_instr,
  exec_rx_instr,

  // outputs
  led0, // for debug
  led1 // for debug
);

  input clk;
  input rst;

  output led0, led1;

// Slave FSL Signals
  output FSL_S_Clk;
  output FSL_S_Read;
  input  [0:31] FSL_S_Data;
  input  FSL_S_Control;
  input  FSL_S_Exists;
  // Master FSL Signals
 output FSL_M_Clk;
 output FSL_M_Write;
 output [0:31] FSL_M_Data;
 output FSL_M_Control;
 input FSL_M_Full;

 //PE Interface
 output pe_rst;
 output [31:0] pe_data_in;
 output pe_write_data;
 output pe_write_addr;
 output pe_done_packet;
 input pe_can_write_data;
 input pe_can_write_addr;

 //Program Interface
 input [0:31] prog_instr;
 input program_cbit, valid_instr;
 output prog_instr_read;
 input cont_prog;
 output exec_rx_instr, exec_bypass_instr;

//Connections for inputs and outputs:
  wire clk;
  wire rst;

  wire FSL_S_Clk;
  wire FSL_S_Read;
  wire [0:31] FSL_S_Data;
  wire FSL_S_Control;
  wire FSL_S_Exists;

  wire FSL_M_Clk;
  wire FSL_M_Write;
  wire [0:31] FSL_M_Data;
  wire FSL_M_Control;
  wire FSL_M_Full;

  //PE Interface
  wire pe_rst;
  wire [31:0] pe_data_in;
  wire pe_write_data;
  wire pe_write_addr;
  wire pe_done_packet;
  wire pe_can_write_data;
  wire pe_can_write_addr;

  //Program interface and Instruction Register Read signals:
  wire [0:31] prog_instr;  //connects to the program instrucion input port
  wire program_cbit; //connects to the program’s control bit
  wire cont_prog; //Used to override the receives higher priority
  wire prog_instr_read; //reading from the port into the program IR

  //Status bits:
  wire exec_bypass_instr; //Status bit allowing writes to the fifo
  wire exec_rx_instr; //NEW TO REPLACE WAIT RECEIVE INSTRUCTION

  wire  led0, led1; //for debug

 //IR signals:
  wire  rx_IR_full;
  wire [0:31] rx_IR;
  wire  rx_cbit;  //receive control bit
  wire prog_IR_full;
  wire [0:31] prog_IR;
  wire prog_cbit;  //program control bit
  reg exec_prog; //am I executing the program or an rx_IR**only valid when
```

```
   //ex_IR_full is high
  //Address registers:
  reg [0:31] a0_reg; //Address register 0
  reg [0:23] a0_operand; //latches the data portion of the instruction
  reg write_a0;

  //Mux control signals:
  wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
  reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

  //Used to generate Computing element resets:
  wire control_reset;  //combines system reset and instruction reset
  reg instr_reset; //generated via the reset instruction- clock period long

  //Used to interface the ex_IR, A0 and data with the fifo:
  reg [1:0] mux_select_fifo_in; //used to select
  wire [0:7] ex_IR_fifo_in;
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  // Indicates when the address register needs to be written to
   //the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg rx_data_req; // Used to indicate when data should be read in as part of
   // the received instruction.
  reg rx_addr_req; // Used to indicate when an address is read in as part of
   // the received instruction.
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // Used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

//state and delay signals:
  reg [1:0] pres_state;
  reg [1:0] next_state;

  wire fifo_full;
  reg fifo_write; //Used to write fifo to the FSL Master side

  wire write_data_to_pe; //Used to read data from the FSL Slave side to the PE

//States for overall execution path state machine:
‘define get_next_instr 0
‘define decode_instr 1
‘define execute_instr 2

//for debug
assign led0 = pe_done_packet ? 1’b0 : 1’b1;
assign led1 = pe_can_write_data ? 1’b0 : 1’b1;


//Slave signals
assign FSL_S_Clk = clk;
assign FSL_S_Read = ((reading_rx_IR == 1’b1) || (write_data_to_pe == 1’b1));
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1'b1) ? ~ex_cbit_input : ex_cbit_input;
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4'b0000;
assign ex_IR_fifo_in[4] = (opcode == `bypass) ? opcode[3] : 1'b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1'b1) ? opcode[1:0] : ~opcode[1:0];

//External PE interface
assign pe_rst = control_reset;
assign pe_data_in = FSL_S_Data;
assign pe_write_data = write_data_to_pe;
assign pe_write_addr = ((reading_rx_IR == 1'b1) && (rx_addr_req == 1'b1));

//Status bits passed to the program from the PE via the controller
assign pe_done_packet = (pres_state == `get_next_instr);
assign exec_bypass_instr = tx_bypass;

//New status bit that can be used to determine if the program should
//get out of a "Wait Receive" stall state
//Status bit 2:
assign exec_rx_instr = ((pres_state != `get_next_instr) && (exec_prog == 1'b0));

assign ex_instr_read = ((pres_state == `get_next_instr)
 && (control_reset == 1'b0)
 && (((rx_IR_full && ~cont_prog) || prog_IR_full)));

//Used to mux the outputs from the two different IRs:
assign sel_ex_IR_input = (pres_state == `get_next_instr) ?
((rx_IR_full==1'b1) && (cont_prog==1'b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_cbit_input = (mux_select_fifo_in == 2'b00) ? IR_req_W0 : 1'b0;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b0) && (prog_IR_full == 1'b1));

////Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)) ||
((rx_addr_req == 1'b1) && (pe_can_write_addr == 1'b1)))
&& (sel_ex_IR_input == 1'b1) && (rx_IR_full == 1'b1));

assign a0_mux_output = ((opcode[0] == 1'b1) && (opcode[2] == 1'b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal and the video reset signal:
assign control_reset = rst || instr_reset;

always @ (posedge clk) begin
  if (rst == 1'b1)
instr_reset <= 1'b0;
  else if ((pres_state == `execute_instr) && (opcode == 8'h00) &&
  (exec_prog == 1'b0))
```

```
instr_reset <= 1'b1;
  else
instr_reset <= 1'b0;
end

always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1'b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end
//The Address register
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_reg <= 32'h00000000;
  else if (write_a0 == 1'b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end

//Used to store the initial Num Data Words (NDW) value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end

//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ( (change_a0 == 1'b1) &&
             (Addr_req_W == 1'b0) )
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == `decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == `execute_instr))) && (opcode == `bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == ‘execute_instr) && (opcode != ‘bypass)))
   && (fifo_write == 1’b1))
    tx_bypass <= 1’b0;
  else
    tx_bypass <= tx_bypass;
end

//CHANGE FOR BYPASS
//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    IR_req_W0 <= 1’b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == ‘decode_instr) && (((exec_prog == 1’b1) &&
  (opcode[3:0] != 4’b1000) && (opcode[3:0] != 4’b1011)) ||
  (opcode[1:0] == 2’b10)))
    IR_req_W0 <= 1’b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b00))
    IR_req_W0 <= 1’b0;
  else
    IR_req_W0 <= IR_req_W0;
end

//CHANGE FOR BYPASS
//Used to indicate when to write immediate values from the receive
//link or the program
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    IR_req_W1 <= 1’b0;
  else if ((pres_state == ‘decode_instr) &&
  //program sending a read request plus abs address
  (((exec_prog == 1’b1) && (opcode[3:0] == 4’h6))
  //controller received a bypass instruction
  || ((exec_prog == 1’b0) && (opcode == ‘bypass))))
    IR_req_W1 <= 1’b1;
  else if (((opcode[3:0] == 4’h6) ||
  ((opcode == ‘bypass) && (data_cntr == 24’h000001))) &&
  (fifo_write == 1’b1) &&  (mux_select_fifo_in == 2’b01))
    IR_req_W1 <= 1’b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//The received instruction includes an address
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    rx_addr_req <= 1’b0;
  else if ((pres_state == ‘decode_instr) && (opcode[2] == 2’b1) &&
   ((opcode[1] == 1’b1) || (opcode[0] == 1’b1)) &&
   (exec_prog == 1’b0))
    rx_addr_req <= 1’b1;
  else if ((rx_addr_req == 1’b1) && (reading_rx_IR == 1’b1))
    rx_addr_req <= 1’b0;
  else
    rx_addr_req <= rx_addr_req;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    Addr_req_W <= 1’b0;
  else if ((pres_state == ‘decode_instr) && (opcode[3:2] == 2’b11) &&
   (opcode[1:0] != 2’b00))
    Addr_req_W <= 1’b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b10))
```

```
    Addr_req_W <= 1'b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Should ensure that data is properly read from the fsl to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     rx_data_req <= 1'b0;
  else if ((pres_state == `decode_instr) &&
((exec_prog == 1'b0) && (opcode[0] == 1'b1)))
     rx_data_req <= 1'b1;
  else if ((data_cntr == 24'h000001) && (write_data_to_pe == 1'b1))
     rx_data_req <= 1'b0;
  else
     rx_data_req <= rx_data_req;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_cntr <= 24'h000000;
  else if ((pres_state == `decode_instr) &&
((exec_prog == 1'b0) && ((opcode[0] == 1'b1) ||
          (opcode == `bypass))))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2'b11) && (write_data_to_pe == 1'b1)) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)
  && (fifo_write == 1'b1)))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end


// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W ) begin
if ((pres_state == `execute_instr) && (IR_req_W0 == 1'b1))
      mux_select_fifo_in = 2'b00;
else if ((pres_state == `execute_instr) && (IR_req_W1 == 1'b1))
      mux_select_fifo_in = 2'b01;
else if ((pres_state == `execute_instr) && (Addr_req_W == 1'b1))
      mux_select_fifo_in = 2'b10;
else
      mux_select_fifo_in = 2'b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1'b1)
    opcode <= 8'b0;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end

// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1'b1)
    exec_prog <= 1'b1;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b1))
    exec_prog <= 1'b0;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b0))
    exec_prog <= 1'b1;
  else
    exec_prog <= exec_prog;
end
```

```
// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand) begin
  case (mux_select_fifo_in)
    2'b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2'b01:
      fifo_mux_out = ex_IR;
    2'b10:
      fifo_mux_out = a0_reg;
    2'b11:
      fifo_mux_out = 32'h00000000;
  endcase
end

// FSM register
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    pres_state <= `get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or control_reset or exec_prog or opcode or ex_instr_read
    or rx_data_req or IR_req_W0 or IR_req_W1 or Addr_req_W or change_a0 or
    rx_addr_req) begin
  case (pres_state)
    `get_next_instr:
      begin
if ((ex_instr_read == 1'b1))
         next_state = `decode_instr;
        else
         next_state = `get_next_instr;
      end
    `decode_instr:
  next_state = `execute_instr;
    `execute_instr:
      begin
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
if ((IR_req_W0 == 1'b1) || (IR_req_W1 == 1'b1) ||
   (Addr_req_W == 1'b1) || (change_a0 == 1'b1) ||
    (rx_data_req == 1'b1)  || (rx_addr_req == 1'b1))
  next_state = `execute_instr;
else
  next_state = `get_next_instr;
        end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or fifo_full) begin
  case(mux_select_fifo_in)
     2'b00:
       begin
         if ((IR_req_W0 == 1'b1) && (fifo_full == 1'b0))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b01:
       begin
         if ((IR_req_W1 == 1'b1) && (fifo_full == 1'b0))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b10:
       begin
         if ((Addr_req_W == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b11:
            fifo_write = 1’b0;
  endcase
end

assign write_data_to_pe = ((pres_state == ‘execute_instr)
&& (rx_addr_req == 1’b0) && (rx_data_req == 1’b1)
&& (pe_can_write_data == 1’b1)
&& (FSL_S_Exists == 1’b1));

endmodule
```

A.3 Consumer Debug Controller

```
//***********************************************************************
//    File: fsl_ce_for_fifo_plus.v
//    Uses: fifo_plus.v and a SIMPPL Control Sequencer
//    Descriptions: Provides a SIMPPL interface to a 16 deep 32-bit word
//     synchronous FIFO (SRL)
//    Uses the SIMPPL Communication Protocol
//
//    LLS Octoober 2005
//***********************************************************************

`include "instr_defines.v"

module consumer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  // Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  // Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface
  pe_rst,
  pe_data_in,
  pe_write_data,
  pe_write_addr,
  pe_done_packet,
  pe_can_write_data,
  pe_can_write_addr,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  exec_bypass_instr,
  exec_rx_instr,

  //Debugging ports:
  //Debugging download link
  debug_link_clk,
  debug_link_write,
  debug_link_data,
  debug_link_control,
  debug_link_full,

  //Debug Control Signals
  int_error,
  chk_status,
  status_rdy,
  cont_execution,

  // outputs
  led0, // for debug
  led1 // for debug
);

  input clk;
  input rst;

  output led0, led1;

// Slave FSL Signals
  output FSL_S_Clk;
  output FSL_S_Read;
  input  [0:31] FSL_S_Data;
  input  FSL_S_Control;
  input  FSL_S_Exists;

  // Master FSL Signals
 output FSL_M_Clk;
 output FSL_M_Write;
 output [0:31] FSL_M_Data;
 output FSL_M_Control;
 input FSL_M_Full;

  //Debugging ports:
  //Debugging download link
  output debug_link_clk;
  output debug_link_write;
  output [0:31] debug_link_data;
  output debug_link_control;
  input debug_link_full;
  //Debugging handshake signals
  output int_error;
  input chk_status;
  output status_rdy;
  input cont_execution;


 //PE Interface
 output pe_rst;
 output [31:0] pe_data_in;
 output pe_write_data;
 output pe_write_addr;
 output pe_done_packet;
 input pe_can_write_data;
 input pe_can_write_addr;

 //Program Interface
 input [0:31] prog_instr;
 input program_cbit, valid_instr;
 output prog_instr_read;
 input cont_prog;
 output exec_rx_instr, exec_bypass_instr;


  //Program interface and Instruction Register Read signals:
  wire [0:31] prog_instr;  //connects to the program instrucion input port
  wire program_cbit; //connects to the program’s control bit
  wire cont_prog; //Used to override the receives higher priority
  wire prog_instr_read; //reading from the port into the program IR

  //Status bits:
  wire exec_bypass_instr; //Status bit allowing writes to the fifo
  wire exec_rx_instr; //NEW TO REPLACE WAIT RECEIVE INSTRUCTION

  wire  led0, led1; //for debug

  //For Debugging  ****
  reg [3:0] status_cntr;
  reg int_debug_link_write;
  reg prev_status_rdy;
  reg int_chk_status;
  reg int_int_error;
  wire CE_error;
  wire error0, error1, error2, error3, error4;
  wire error5, error6, error7, error8, error9;
  reg err_cbit;
  reg ex_cbit;

  //Debug ****
  reg [0:31] a0_reg_bak, imm_addr, controller_status_reg, prog_PE_status_reg;
  reg [0:31] ex_IR_bak, data_cntr_bak, prog_IR_bak, rx_IR_bak;
  reg[0:31] err_type_reg, err_word_reg, exec_time_cntr, ce_id_reg;


 //IR signals:
  wire  rx_IR_full;
  wire [0:31] rx_IR;
  wire  rx_cbit;  //receive control bit
  wire prog_IR_full;
  wire [0:31] prog_IR;
  wire prog_cbit;  //program control bit
  reg exec_prog; //am I executing the program or an rx_IR**only valid when
   //ex_IR_full is high
  //Address registers:
  reg [0:31] a0_reg; //Address register 0
  reg [0:23] a0_operand; //latches the data portion of the instruction
  reg write_a0;

  //Mux control signals:
  wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
  reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

  //Used to generate Computing element resets:
  wire control_reset;  //combines system reset and instruction reset
  reg instr_reset; //generated via the reset instruction- clock period long

  //Used to interface the ex_IR, A0 and data with the fifo:
  reg [1:0] mux_select_fifo_in; //used to select
  wire [0:7] ex_IR_fifo_in;
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  // Indicates when the address register needs to be written to
   //the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg rx_data_req; // Used to indicate when data should be read in as part of
   // the received instruction.
  reg rx_addr_req; // Used to indicate when an address is read in as part of
   // the received instruction.
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // Used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire ex_IR_cbit;  //wire used as mux output from
      //prog and rx IR's cbits
  wire ex_IR_full; //used to mux prog an rx full output signals
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

//state and delay signals:
  reg [1:0] prev_state; //Debug
  reg [1:0] pres_state;
  reg [1:0] next_state;

  wire fifo_full;
  reg fifo_write; //Used to write fifo to the FSL Master side

  wire write_data_to_pe; //Used to read data from the FSL Slave side to the PE

//States for overall execution path state machine:
`define get_next_instr 0
`define decode_instr 1
`define execute_instr 2
`define status_chk 3 //Debug

//Debug: ****
`define num_regs 11 //Debug
`define ce_id 32'h0000abbc //Debug: CE ID
`define bit 31-14 //The subtracted number represents the power of two bit


//for debug
assign led0 = int_int_error ? 1'b0 : 1'b1;
assign led1 = (status_cntr == `num_regs) ? 1'b0 : 1'b1;


//Slave signals
assign FSL_S_Clk = clk;
assign FSL_S_Read = ((reading_rx_IR == 1'b1) || (write_data_to_pe == 1'b1));
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1'b1) ? ~ex_cbit_input : ex_cbit_input;
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4'b0000;
assign ex_IR_fifo_in[4] = (opcode == `bypass) ? opcode[3] : 1'b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1'b1) ? opcode[1:0] : ~opcode[1:0];

//External PE interface
assign pe_rst = control_reset;
assign pe_data_in = FSL_S_Data;
assign pe_write_data = write_data_to_pe;
assign pe_write_addr = ((reading_rx_IR == 1'b1) && (rx_addr_req == 1'b1));

//Status bits passed to the program from the PE via the controller
assign pe_done_packet = (pres_state == `get_next_instr);
assign exec_bypass_instr = tx_bypass;

//New status bit that can be used to determine if the program should
//get out of a "Wait Receive" stall state
//Status bit 2:
assign exec_rx_instr = ((pres_state != `get_next_instr) && (exec_prog == 1'b0));

assign ex_instr_read = ((pres_state == `get_next_instr)
 && (control_reset == 1'b0)
 && (((rx_IR_full && ~cont_prog) || prog_IR_full)));

//Used to mux the outputs from the two different IRs:
assign sel_ex_IR_input = (pres_state == `get_next_instr) ?
((rx_IR_full==1'b1) && (cont_prog==1'b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_IR_cbit = sel_ex_IR_input ? rx_cbit : prog_cbit ; //Debug
assign ex_cbit_input = (mux_select_fifo_in == 2'b00) ? IR_req_W0 : 1'b0;
assign ex_IR_full = sel_ex_IR_input ? rx_IR_full : prog_IR_full ;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b0) && (prog_IR_full == 1'b1));

////Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)) ||
((rx_addr_req == 1'b1) && (pe_can_write_addr == 1'b1)))
&& (sel_ex_IR_input == 1'b1) && (rx_IR_full == 1'b1));

assign a0_mux_output = ((opcode[0] == 1'b1) && (opcode[2] == 1'b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal and the video reset signal:
assign control_reset = rst || instr_reset;

always @ (posedge clk) begin
  if (rst == 1'b1)
instr_reset <= 1'b0;
  else if ((pres_state == `execute_instr) && (opcode == 8'h00) &&
  (exec_prog == 1'b0))
instr_reset <= 1'b1;
  else
instr_reset <= 1'b0;
end

//Used to add/subtract the a0 operand to the address register
always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1'b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end

//The Address register
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_reg <= 32'h00000000;
  else if (write_a0 == 1'b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end

//Used to store the initial Num Data Words (NDW) value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end
```

```
//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W or ex_IR_full or opcode) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ((change_a0 == 1'b1) && (Addr_req_W == 1'b0) &&
  (((ex_IR_full == 1'b1) && (opcode == 'write_Areg)) ||
   (opcode != 'write_Areg)))
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == 'decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == 'decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == 'execute_instr))) && (opcode == 'bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == 'execute_instr) && (opcode != 'bypass)))
   && (fifo_write == 1'b1))
    tx_bypass <= 1'b0;
  else
    tx_bypass <= tx_bypass;
end

//CHANGE FOR BYPASS
//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W0 <= 1'b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == 'decode_instr) && (((exec_prog == 1'b1) &&
  (opcode[3:0] != 4'b1000) && (opcode[3:0] != 4'b1011)) ||
  (opcode[1:0] == 2'b10)))
    IR_req_W0 <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b00))
    IR_req_W0 <= 1'b0;
  else
    IR_req_W0 <= IR_req_W0;
end

//CHANGE FOR BYPASS
//Used to indicate when to write immediate values from the receive
//link or the program
always @ (posedge clk) begin
```

```
  if (control_reset == 1'b1)
    IR_req_W1 <= 1'b0;
  else if ((pres_state == `decode_instr) &&
  //program sending a read request plus abs address
  (((exec_prog == 1'b1) && (opcode[3:0] == 4'h6))
  //controller received a bypass instruction
  || ((exec_prog == 1'b0) && (opcode == `bypass))))
    IR_req_W1 <= 1'b1;
  else if (((opcode[3:0] == 4'h6) ||
  ((opcode == `bypass) && (data_cntr == 24'h000001))) &&
  (fifo_write == 1'b1) &&  (mux_select_fifo_in == 2'b01))
    IR_req_W1 <= 1'b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//The received instruction includes an address
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    rx_addr_req <= 1'b0;
  else if ((pres_state == `decode_instr) && (opcode[2] == 2'b1) &&
   ((opcode[1] == 1'b1) || (opcode[0] == 1'b1)) &&
   (exec_prog == 1'b0))
    rx_addr_req <= 1'b1;
  else if ((rx_addr_req == 1'b1) && (reading_rx_IR == 1'b1))
    rx_addr_req <= 1'b0;
  else
    rx_addr_req <= rx_addr_req;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    Addr_req_W <= 1'b0;
  else if ((pres_state == `decode_instr) && (opcode[3:2] == 2'b11) &&
   (opcode[1:0] != 2'b00))
    Addr_req_W <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b10))
    Addr_req_W <= 1'b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Should ensure that data is properly read from the fsl to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     rx_data_req <= 1'b0;
  else if ((pres_state == `decode_instr) &&
((exec_prog == 1'b0) && (opcode[0] == 1'b1)))
     rx_data_req <= 1'b1;
  else if ((data_cntr == 24'h000001) && (write_data_to_pe == 1'b1))
     rx_data_req <= 1'b0;
  else
     rx_data_req <= rx_data_req;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_cntr <= 24'h000000;
  else if ((pres_state == `decode_instr) &&
((exec_prog == 1'b0) && ((opcode[0] == 1'b1) ||
          (opcode == `bypass))))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2'b11) && (write_data_to_pe == 1'b1)) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)
  && (fifo_write == 1'b1)))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end


// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W ) begin
if ((pres_state == ‘execute_instr) && (IR_req_W0 == 1’b1))
      mux_select_fifo_in = 2’b00;
else if ((pres_state == ‘execute_instr) && (IR_req_W1 == 1’b1))
      mux_select_fifo_in = 2’b01;
else if ((pres_state == ‘execute_instr) && (Addr_req_W == 1’b1))
      mux_select_fifo_in = 2’b10;
else
      mux_select_fifo_in = 2’b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    opcode <= 8’b0;
  else if ((ex_instr_read == 1’b1) && (pres_state == ‘get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end

// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    exec_prog <= 1’b0;
  else if ((ex_instr_read == 1’b1) && (sel_ex_IR_input == 1’b1))
    exec_prog <= 1’b0;
  else if ((ex_instr_read == 1’b1) && (sel_ex_IR_input == 1’b0))
    exec_prog <= 1’b1;
  else
    exec_prog <= exec_prog;
end

// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand) begin
  case (mux_select_fifo_in)
    2’b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2’b01:
      fifo_mux_out = ex_IR;
    2’b10:
      fifo_mux_out = a0_reg;
    2’b11:
      fifo_mux_out = 32’h00000000;
  endcase
end

// FSM register
always @ (posedge clk) begin
  if (rst == 1’b1)
    pres_state <= ‘get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or control_reset or exec_prog or opcode or ex_instr_read
    or rx_data_req or IR_req_W0 or IR_req_W1 or Addr_req_W or change_a0 or
    rx_addr_req or error0 or error1 or error2 or error3 or error4 or error5 or
    error6 or error7 or error8 or error9 or int_int_error or int_chk_status or
    status_cntr or cont_execution) begin
  case (pres_state)
```

```
    `get_next_instr:
      begin
        if ((error1 == 1'b1) || (error3 == 1'b1) || (error5 == 1'b1)
|| (error9 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else if ((ex_instr_read == 1'b1))
         next_state = `decode_instr;
        else
         next_state = `get_next_instr;
      end
    `decode_instr:
      begin
        if ((error0 == 1'b1) || (error5 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else
        next_state = `execute_instr;
      end
    `execute_instr:
      begin
        if ((error0 == 1'b1) || (error2 == 1'b1) || (error4 == 1'b1) || //Debug
            (error5 == 1'b1) || (error6 == 1'b1) || (error7 == 1'b1) || //Debug
            (error8 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
        else if ((int_int_error == 1'b1) || ((int_chk_status == 1'b1) && //Debug
((IR_req_W0 == 1'b0) && (IR_req_W1 == 1'b0) &&  //Debug
   (Addr_req_W == 1'b0) && (change_a0 == 1'b0) &&  //Debug
   (rx_data_req == 1'b0) && (rx_addr_req == 1'b0)))) //Debug
  next_state = `status_chk; //Debug
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
else if ((IR_req_W0 == 1'b1) || (IR_req_W1 == 1'b1) ||
   (Addr_req_W == 1'b1) || (change_a0 == 1'b1) ||
    (rx_data_req == 1'b1)  || (rx_addr_req == 1'b1))
  next_state = `execute_instr;
else
  next_state = `get_next_instr;
      end
    `status_chk: //Debug ****
      begin
if ((status_cntr == `num_regs-1) &&
(cont_execution == 1'b1) && (int_int_error == 1'b0)) //Debug
next_state = `get_next_instr;
else
next_state = `status_chk;
      end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or
    fifo_full or ex_IR_full) begin
  case(mux_select_fifo_in)
     2'b00:
       begin
         if ((IR_req_W0 == 1'b1) && (fifo_full == 1'b0) &&
 (int_int_error == 1'b0)) //Debug
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b01:
       begin
         if ((IR_req_W1 == 1'b1) && (fifo_full == 1'b0) &&
 (int_int_error == 1'b0) && (ex_IR_full == 1'b1)) //Debug
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b10:
       begin
         if ((Addr_req_W == 1'b1) && (fifo_full == 1'b0) &&
 (int_int_error == 1'b0))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b11:
            fifo_write = 1'b0;
  endcase
end

assign write_data_to_pe = ((pres_state == `execute_instr)
&& (rx_addr_req == 1'b0) && (rx_data_req == 1'b1)
&& (pe_can_write_data == 1'b1)
&& (FSL_S_Exists == 1'b1));

//Debug register download order:
//0)int_debug_link_data <= err_type_reg;
//1)int_debug_link_data <= ex_IR_bak;
//2)int_debug_link_data <= imm_addr;
//3)int_debug_link_data <= a0_reg_bak;
//4)int_debug_link_data <= data_cntr_bak;
//5)int_debug_link_data <= prog_IR_bak;
//6)int_debug_link_data <= rx_IR_bak;
//7)int_debug_link_data <= controller_status_reg;
//8)int_debug_link_data <= prog_PE_status_reg;
//9)int_debug_link_data <= exec_time_cntr;

assign debug_link_clk = clk; //Debug
assign debug_link_data = ce_id_reg; //Debug
assign debug_link_control = 1'b0; //Debug
assign int_error = int_int_error; //Debug
assign CE_error = (error0 || error1 || error2 || error3 || error4 || error5 ||
error6 || error7 || error8 || error9);

//Debug Errors:
//Error0: reading an instruction outside the "fetch" stage
assign error0 = ((((prog_cbit == 1'b1) && (reading_prog_IR == 1'b1)) ||
 ((rx_cbit == 1'b1) && (reading_rx_IR == 1'b1))) &&
 (pres_state != `get_next_instr));
//Error1: in the "fetch" stage but the IR has a data word
assign error1 = ((pres_state == `get_next_instr) &&
(((reading_rx_IR == 1'b1) && (rx_cbit == 1'b0)) ||
((reading_prog_IR == 1'b1) && (prog_cbit == 1'b0))));
//Error2: in the "execute" stage, but the execution time cntr has overflowed
assign error2 = ((pres_state == `execute_instr) &&
(exec_time_cntr[`bit] == 1'b1));
//Error3: in the "fetch" stage, but the next instruction time cntr overflows
assign error3 = ((pres_state == `get_next_instr) &&
(exec_time_cntr[`bit] == 1'b1));
//Error4: trying to write to a full FSL
assign error4 = ((FSL_M_Write == 1'b1) && (FSL_M_Full == 1'b1));
//Error5: trying to read from an empty FSL
assign error5 = ((FSL_S_Read == 1'b1) && (FSL_S_Exists == 1'b0));
//Error6: trying to write data to the PE when the PE isn't ready
assign error6 = ((pe_write_data == 1'b1) && (pe_can_write_data == 1'b0));
//Error7: trying to write an address to the PE when the PE isn't ready
assign error7 = ((pe_write_addr == 1'b1) && (pe_can_write_addr == 1'b0));
//Error8: trying to read data from the PE when there is none ready
//assign error8 = ((pe_read_data == 1'b1) && (pe_can_read == 1'b0)); //producer
assign error8 = 1'b0;
//Error9: trying to execute an invalid instruction
assign error9 = ((pres_state == `get_next_instr) && (((reading_rx_IR == 1'b1) &&
((rx_IR[24:27] != 4'h0) || (rx_IR[30:31] == 2'b10) ||
(rx_IR[28:29] == 2'b10) || ((rx_IR[28] == 1'b1) &&
(rx_IR[31] == 1'b1)))) || ((reading_prog_IR == 1'b1) &&
((prog_IR[24:26] != 3'b000) || (prog_IR[27:28] == 2'b10) ||
(prog_IR[28:31] == 4'b1010) || (prog_IR[30:31] == 2'b01) ||
((prog_IR[31] == 1'b1) && ((prog_IR[29] == 1'b1) ||
(prog_IR[28] == 1'b0) || (prog_IR[27] == 1'b1))) ||
((prog_IR[27] == 1'b1) && (prog_IR[29:30] == 2'b10))))));

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_int_error <= 1'b0;
  else if (CE_error == 1'b1)
    int_int_error <= 1'b1;
  else
    int_int_error <= int_int_error;
end

//Debug
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_state <= `get_next_instr;
  else
    prev_state <= pres_state;
end

//Debug:
//Status info is ready when all the registers have been downloaded to the
//debug link fifo
assign status_rdy = (status_cntr == `num_regs-1) ? 1'b1 : 1'b0; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_status_rdy <= 1'b0;
  else
    prev_status_rdy <= status_rdy;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_chk_status <= 1'b0;
  else if (chk_status == 1'b1)
    int_chk_status <= 1'b1;
  else if (status_rdy == 1'b1)
    int_chk_status <= 1'b0;
  else
    int_chk_status <= int_chk_status;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_debug_link_write <= 1'b0;
  else if (status_cntr == `num_regs)
    int_debug_link_write <= 1'b0;
  else if ((pres_state == `status_chk) && (debug_link_full == 1'b0) &&
   (status_rdy == 1'b0) && (status_cntr != `num_regs))
    int_debug_link_write <= 1'b1;
  else
    int_debug_link_write <= 1'b0;
end

assign debug_link_write = int_debug_link_write; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    status_cntr <= 4'h0;
  else if ((prev_state == `status_chk) && (pres_state == `get_next_instr))
```

```
    status_cntr <= 4'h0;
  else if ((pres_state == `status_chk) && (int_debug_link_write == 1'b1))
    status_cntr <= status_cntr + 1;
  else
    status_cntr <= status_cntr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    ce_id_reg <= `ce_id;
  else if (int_debug_link_write == 1'b1)
    ce_id_reg <= err_type_reg;
  else
    ce_id_reg <= `ce_id;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    err_type_reg <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    err_type_reg <= err_word_reg;
  else if (int_int_error == 1'b0)
    begin
      err_type_reg[0] <= error0;
      err_type_reg[1] <= error1;
      err_type_reg[2] <= error2;
      err_type_reg[3] <= error3;
      err_type_reg[4] <= error4;
      err_type_reg[5] <= error5;
      err_type_reg[6] <= error6;
      err_type_reg[7] <= error7;
      err_type_reg[8] <= error8;
      err_type_reg[9] <= error9;
      err_type_reg[10:30] <= 22'h000000;
      err_type_reg[31] <= CE_error;
    end
  else if (pres_state == `get_next_instr)
      err_type_reg <= 32'h00000000;
  else
    err_type_reg <= err_type_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    err_word_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    err_word_reg <= ex_IR_bak;
  else if ((reading_prog_IR == 1'b1) && (CE_error == 1'b1))
    err_word_reg <= prog_IR;
  else if ((reading_rx_IR == 1'b1) && (CE_error == 1'b1))
    err_word_reg <= rx_IR;
  else if ((fifo_write == 1'b1) && (CE_error == 1'b1))
    err_word_reg <= fifo_mux_out;
  else if (pres_state == `get_next_instr)
      err_word_reg <= 32'h00000000;
  else
    err_word_reg <= err_word_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    err_cbit <= 1'h0;
  else if ((reading_prog_IR == 1'b1) && (CE_error == 1'b1))
    err_cbit <= prog_cbit;
  else if ((reading_rx_IR == 1'b1) && (CE_error == 1'b1))
    err_cbit <= rx_cbit;
  else if ((fifo_write == 1'b1) && (CE_error == 1'b1) && (tx_bypass == 1'b1))
    err_cbit <= ~ex_cbit_input;
  else if ((fifo_write == 1'b1) && (CE_error == 1'b1) && (tx_bypass == 1'b0))
    err_cbit <= ex_cbit_input;
  else
    err_cbit <= err_cbit;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    ex_IR_bak <= 32'h00000000;
  else if (((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
  || ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0)))
    ex_IR_bak <= ex_IR;
  else if (debug_link_write == 1'b1)
    ex_IR_bak <= imm_addr;
  else
    ex_IR_bak <= ex_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    ex_cbit <= 1'b0;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    ex_cbit <= ex_IR_cbit;
  else
    ex_cbit <= ex_cbit;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    imm_addr <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    imm_addr <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    imm_addr <= a0_reg_bak;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01) &&
   (opcode != `bypass))
    imm_addr <= ex_IR;
  else
    imm_addr <= imm_addr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    a0_reg_bak <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    a0_reg_bak <= a0_reg;
  else if (int_debug_link_write == 1'b1)
    a0_reg_bak <= data_cntr_bak;
  else if (write_a0 == 1'b1)
    a0_reg_bak <= a0_mux_output;
  else
    a0_reg_bak <= a0_reg_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    data_cntr_bak <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    data_cntr_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    data_cntr_bak <= prog_IR_bak;
  else if (pres_state == `decode_instr)
    data_cntr_bak <= {8'h00,a0_operand};
  else if (((mux_select_fifo_in == 2'b11) && (write_data_to_pe == 1'b1)) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)
  && (fifo_write == 1'b1)))
    data_cntr_bak <= data_cntr_bak - 1;
  else
    data_cntr_bak <= data_cntr_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prog_IR_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    prog_IR_bak <= rx_IR_bak;
  else if ((pres_state == `status_chk) && (prev_state == `status_chk))
    prog_IR_bak <= prog_IR_bak;
  else
    prog_IR_bak <= prog_IR;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    rx_IR_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    rx_IR_bak <= controller_status_reg;
  else if ((pres_state == `status_chk) && (prev_state == `status_chk))
    rx_IR_bak <= rx_IR_bak;
  else
    rx_IR_bak <= rx_IR;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    controller_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    controller_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    controller_status_reg <= prog_PE_status_reg;
  else if (((pres_state == `get_next_instr) || (pres_state == `decode_instr))
   || ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      controller_status_reg[0] <= ex_cbit;
      controller_status_reg[1] <= IR_req_W0;
      controller_status_reg[2] <= IR_req_W1;
      controller_status_reg[3] <= Addr_req_W;
      //controller_status_reg[4] <= data_req_W;
      controller_status_reg[4] <= 1'b0;
      controller_status_reg[5] <= tx_bypass;
      controller_status_reg[6] <= exec_prog;
      controller_status_reg[7] <= rx_addr_req;
      controller_status_reg[8] <= rx_data_req;
      controller_status_reg[9] <= change_a0;
      controller_status_reg[10:15] <= 6'h00;
      controller_status_reg[16] <= ex_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      //controller_status_reg[20] <= data_req_W;
      controller_status_reg[20] <= 1'b0;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      controller_status_reg[23] <= rx_addr_req;
      controller_status_reg[24] <= rx_data_req;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else if (pres_state == `execute_instr)
    begin
      controller_status_reg[0:15] <= controller_status_reg[0:15];
      controller_status_reg[16] <= ex_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      //controller_status_reg[20] <= data_req_W;
      controller_status_reg[20] <= 1'b0;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      controller_status_reg[23] <= rx_addr_req;
      controller_status_reg[24] <= rx_data_req;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else if ((pres_state == `status_chk) && (prev_state != `status_chk))
    begin
      controller_status_reg[0:15] <= controller_status_reg[0:15];
      controller_status_reg[16] <= ex_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      //controller_status_reg[20] <= data_req_W;
      controller_status_reg[20] <= 1'b0;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      controller_status_reg[23] <= rx_addr_req;
      controller_status_reg[24] <= rx_data_req;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else
    controller_status_reg <= controller_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prog_PE_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    prog_PE_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    prog_PE_status_reg <= exec_time_cntr;
  else if ((pres_state == `get_next_instr) || (pres_state == `decode_instr) ||
  ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      prog_PE_status_reg[0] <= rx_IR_full;
      prog_PE_status_reg[1] <= prog_IR_full;
      prog_PE_status_reg[2] <= rx_cbit;
      prog_PE_status_reg[3] <= prog_cbit;
      prog_PE_status_reg[4] <= valid_instr;
      prog_PE_status_reg[5] <= cont_prog;
      //prog_PE_status_reg[6] <= pe_can_read;
      prog_PE_status_reg[6] <= 1'b0;
      prog_PE_status_reg[7] <= pe_can_write_data;
      prog_PE_status_reg[8] <= pe_can_write_addr;
      prog_PE_status_reg[9] <= err_cbit;
      prog_PE_status_reg[10:11] <= pres_state;
      prog_PE_status_reg[12:13] <= mux_select_fifo_in;
      prog_PE_status_reg[14:15] <= 2'h0;
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      //prog_PE_status_reg[22] <= pe_can_read;
      prog_PE_status_reg[22] <= 1'b0;
      prog_PE_status_reg[23] <= pe_can_write_data;
      prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[25] <= err_cbit;
      prog_PE_status_reg[26:27] <= pres_state;
      prog_PE_status_reg[28:29] <= mux_select_fifo_in;
      prog_PE_status_reg[30:31] <= 2'h0;
    end
  else if (pres_state == `execute_instr)
    begin
      prog_PE_status_reg[0:15] <= prog_PE_status_reg[0:15];
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      //prog_PE_status_reg[22] <= pe_can_read;
      prog_PE_status_reg[22] <= 1'b0;
      prog_PE_status_reg[23] <= pe_can_write_data;
      prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[25] <= err_cbit;
      prog_PE_status_reg[26:27] <= pres_state;
      prog_PE_status_reg[28:29] <= mux_select_fifo_in;
      prog_PE_status_reg[30:31] <= 2'h0;
    end
  else if ((pres_state == `status_chk) && (prev_state != `status_chk))
    begin
      prog_PE_status_reg[0:15] <= prog_PE_status_reg[0:15];
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      //prog_PE_status_reg[22] <= pe_can_read;
      prog_PE_status_reg[22] <= 1'b0;
      prog_PE_status_reg[23] <= pe_can_write_data;
      prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[25] <= err_cbit;
      prog_PE_status_reg[26:27] <= prog_PE_status_reg[26:27];
      prog_PE_status_reg[28:29] <= prog_PE_status_reg[28:29];
      prog_PE_status_reg[30:31] <= 2'h0;
    end
  else
    prog_PE_status_reg <= prog_PE_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    exec_time_cntr <= 32'h00000000;
  else if (((status_rdy == 1'b1) && (prev_status_rdy == 1'b0)) ||
  ((pres_state == `get_next_instr) && (prev_state == `execute_instr))
  || (pres_state == `decode_instr))
    exec_time_cntr <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    exec_time_cntr <= `ce_id; //END OF PIPELINED DEBUG DATA
  else if((pres_state == `execute_instr) || (pres_state == `get_next_instr))
    exec_time_cntr <= exec_time_cntr + 1;
  else
    exec_time_cntr <= exec_time_cntr;
end
endmodule
```

A.4 Producer Execute Controller

```
//************************************************************************
//    File: fsl_ce_for_cntr_pe.v
//    Uses: cntr_pe.v and a SIMPPL Control Sequencer
//    Description: Provides a SIMPPL interface to the four bit counter.
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//************************************************************************
//

`include "instr_defines.v"

module producer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  //Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  //Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface:
  pe_rst,
  pe_data_out,
  pe_read_data,
  pe_can_read,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  valid_data,
  exec_rx_instr,

  // outputs
  led0, // for debug
  led1 // for debug
  );

  input clk, rst;
  output FSL_M_Clk, FSL_M_Write;
  output [0:31] FSL_M_Data;
  output FSL_M_Control;
  input FSL_M_Full;
  output FSL_S_Clk, FSL_S_Read;
  input [0:31] FSL_S_Data;
  input FSL_S_Control, FSL_S_Exists;
  output led0, led1; // for debug

  //PE Interface:
  output pe_rst;
  input [31:0] pe_data_out;
  output pe_read_data;
  input pe_can_read;
```

```
//Program Interface:
input [0:31] prog_instr;
input program_cbit, valid_instr;
output prog_instr_read;
input cont_prog;
output valid_data, exec_rx_instr;

wire clk, rst;
wire FSL_M_Clk, FSL_M_Full;
wire FSL_M_Control;
wire FSL_M_Write;
wire [0:31] FSL_M_Data;

//Slave FSL wires:
wire FSL_S_Clk;
wire FSL_S_Read;
wire [0:31] FSL_S_Data;
wire FSL_S_Control;
wire FSL_S_Exists;

//PE Interface:
wire pe_rst;
wire [31:0] pe_data_out;
wire pe_read_data;
wire pe_can_read;

//Program interface and Instruction Register Read signals:
wire [0:31] prog_instr;  //connects to the program instrucion input port
wire program_cbit; //connects to the program's control bit
wire valid_instr; //Used to indicate that the instruction from the program is
 //valid
wire cont_prog; //Used to override the receives higher priority
wire prog_instr_read; //reading from the port into the program IR

//Status bits:
//wire new_transaction; //status bit provided to the program for branching
wire valid_data; //status bit provided to the program for branching
wire exec_rx_instr; //status bit provided to be used in place of the
 //"wait rx" instr

wire led0, led1; // for debug

//IR signals:
wire  rx_IR_full;
wire [0:31] rx_IR;
wire  rx_cbit;  //receive control bit
wire prog_IR_full;
wire [0:31] prog_IR;
wire prog_cbit;  //program control bit
reg exec_prog; //am I executing the program or an rx_IR**only valid when
 //ex_IR_full is high

//Address registers:
reg [0:31] a0_reg; //Address register 0
reg [0:23] a0_operand; //latches the data portion of the instruction
reg write_a0;

//Mux control signals:
wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

wire control_reset;  //combines system reset and instruction reset
reg instr_reset; //generated via the reset instruction- clock period long

//Used to interface the ex_IR, A0 and data with the fifo:
reg [1:0] mux_select_fifo_in; //used to select
wire [0:7] ex_IR_fifo_in;
```

```
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  //Indicates when the address register needs to be written to
   //the fifo
  reg data_req_W;  // Indicates that there is data to be written to the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

  reg [1:0] pres_state;
  reg [1:0] next_state;

  reg fifo_write; //write data to the Master FSL
  wire fifo_full;

  wire read_data; //read data from the pe to the Master FSL

//States for overall execution path state machine:
‘define get_next_instr 0
‘define decode_instr 1
‘define execute_instr 2

// for debug
assign led0 = exec_rx_instr ? 1’b0 : 1’b1;
assign led1 = valid_data ? 1’b0 : 1’b1;

//Slave signals:
assign FSL_S_Clk = clk;
//Used to Read the Slave fsl data:
assign FSL_S_Read = reading_rx_IR;
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

//Master signals:
assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1’b1) ? ~ex_cbit_input : ex_cbit_input;
//Used to Write the data from the counter to the FSL:
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4’b0000;
assign ex_IR_fifo_in[4] = (opcode == ‘bypass) ? opcode[3] : 1’b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1’b1) ? opcode[1:0] : ~opcode[1:0];

assign pe_rst = control_reset;
assign pe_read_data = read_data;

//Status bits received from the PE and sent to the program via the controller:
assign valid_data = pe_can_read;

//Status bit 2:
assign exec_rx_instr = ((pres_state != ‘get_next_instr) && (exec_prog == 1’b0));
```

```
//Used to Read an instruction into the Executing IR:
assign ex_instr_read = ((pres_state == ‘get_next_instr)
&& (control_reset == 1’b0)
  && (((rx_IR_full && ~cont_prog )|| prog_IR_full)));

//Used to mux the outputs from the two different IRs such that the rx_IR
//takes precedence when the cont_prog flag is low:
assign sel_ex_IR_input = (pres_state == ‘get_next_instr) ?
((rx_IR_full==1’b1) && (cont_prog==1’b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_cbit_input = (mux_select_fifo_in == 2’b00) ? IR_req_W0 : 1’b0;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1’b1) ||
((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
|| ((write_a0 == 1’b1) && (opcode[0] == 1’b1) &&
(opcode[2] == 1’b0)))
&& (sel_ex_IR_input == 1’b0) && (prog_IR_full == 1’b1));

//Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1’b1) ||
((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
|| ((write_a0 == 1’b1) && (opcode[0] == 1’b1) &&
(opcode[2] == 1’b0)))
&& (sel_ex_IR_input == 1’b1) && (rx_IR_full == 1’b1));

//Used to multiplex an immediate or incremented/decremented value to a0
assign a0_mux_output = ((opcode[0] == 1’b1) && (opcode[2] == 1’b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal:
assign control_reset = rst || instr_reset;

//Used to generate the controller’s reset signal from the reset instruction:
always @ (posedge clk) begin // or posedge rst) begin
  if (rst == 1’b1)
instr_reset <= 1’b0;
  else if ((pres_state == ‘execute_instr) && (opcode == 8’h00) &&
  (exec_prog == 1’b0))
instr_reset <= 1’b1;
  else
instr_reset <= 1’b0;
end

//Used to add/subtract the a0 operand to the address register
always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1’b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end

//The address register
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    a0_reg <= 32’h00000000;
  else if (write_a0 == 1’b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end
```

```
//Used to store the initial NDW value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end

//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ( (change_a0 == 1'b1) &&
            (Addr_req_W == 1'b0) )
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

//Used to indicate that the value of the address register will change
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == `decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == `execute_instr))) && (opcode == `bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == `execute_instr) && (opcode != `bypass)))
   && (fifo_write == 1'b1))
    tx_bypass <= 1'b0;
  else
    tx_bypass <= tx_bypass;
end

//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W0 <= 1'b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == `decode_instr) && (((exec_prog == 1'b1) &&
  ((opcode[3:0] != 4'b1000) && (opcode[3:0] != 4'b1011)))
  || (opcode[1:0] == 2'b10)))
    IR_req_W0 <= 1'b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b00))
    IR_req_W0 <= 1’b0;
  else
    IR_req_W0 <= IR_req_W0;
end

// Used to indicate when to write immediate values from the receive
// link or program
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    IR_req_W1 <= 1’b0;
  else if ((pres_state == ‘decode_instr) && ((opcode[2] == 1’b1) &&
  (((exec_prog == 1’b1) && ((opcode[0] == 1’b1) || (opcode[1]==1’b1))
  && (opcode[3] == 1’b0)) //Not an Areg operation
  || ((exec_prog == 1’b0) && ((opcode == ‘bypass) ||
  (opcode[1:0] == 2’b10))))))
    IR_req_W1 <= 1’b1;
  else if (((opcode != ‘bypass) ||
  ((opcode == ‘bypass) && (data_cntr == 24’h000001))) &&
  (fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
    IR_req_W1 <= 1’b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    Addr_req_W <= 1’b0;
  else if ((pres_state == ‘decode_instr) && (opcode[3:2] == 2’b11) &&
   (opcode[1:0] != 2’b00))
    Addr_req_W <= 1’b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b10))
    Addr_req_W <= 1’b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Controls the request to write data to the fifo
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    data_req_W <= 1’b0;
  else if ((pres_state == ‘decode_instr) &&
(((exec_prog == 1’b1) && (opcode[0] == 1’b1)) ||
       ((exec_prog == 1’b0) && (opcode[1:0] == 2’b10))) &&
       (opcode[3:2] != 2’b10))
    data_req_W <= 1’b1;
  else if ((data_cntr == 24’h000001) && (fifo_write == 1’b1)
   //needed for when only 1 data word is sent
  && (mux_select_fifo_in == 2’b11))
    data_req_W <= 1’b0;
  else
    data_req_W <= data_req_W;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    data_cntr <= 24’h000000;
  else if ((pres_state == ‘decode_instr) &&
(((exec_prog == 1’b1) && (opcode[0] == 1’b1)) ||
       ((exec_prog == 1’b0) && ((opcode[1:0] == 2’b10) ||
        (opcode == ‘bypass)))) && (opcode[3:2] != 2’b10))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2’b11) ||
  ((opcode == ‘bypass) && (mux_select_fifo_in == 2’b01)))
  && (fifo_write == 1’b1))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end

// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W) begin
if ((pres_state == ‘execute_instr) && (IR_req_W0 == 1’b1))
      mux_select_fifo_in = 2’b00;
else if ((pres_state == ‘execute_instr) && (IR_req_W1 == 1’b1))
      mux_select_fifo_in = 2’b01;
else if ((pres_state == ‘execute_instr) && (Addr_req_W == 1’b1))
      mux_select_fifo_in = 2’b10;
else
      mux_select_fifo_in = 2’b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    opcode <= 8’b0;
  else if ((ex_instr_read == 1’b1) && (pres_state == ‘get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end
// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    exec_prog <= 1’b1;
  else if ((ex_instr_read == 1’b1) && (sel_ex_IR_input == 1’b1))
    exec_prog <= 1’b0;
  else if ((ex_instr_read == 1’b1) && (sel_ex_IR_input == 1’b0))
    exec_prog <= 1’b1;
  else
    exec_prog <= exec_prog;
end

// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand
    or pe_data_out) begin
  case (mux_select_fifo_in)
    2’b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2’b01:
      fifo_mux_out = ex_IR;
    2’b10:
      fifo_mux_out = a0_reg;
    2’b11:
      //fifo_mux_out = cntr_pe_data;
      fifo_mux_out = pe_data_out;
  endcase
end

always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1’b1)
    pres_state <= ‘get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or ex_instr_read or IR_req_W0 or IR_req_W1 or Addr_req_W
    or change_a0 or data_req_W) begin
  case (pres_state)
    ‘get_next_instr:
      begin
if ((ex_instr_read == 1’b1))
         next_state = ‘decode_instr;
```

```
        else
         next_state = `get_next_instr;
      end
    `decode_instr:
      next_state = `execute_instr;
    `execute_instr:
      begin
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
if ((IR_req_W0 == 1'b1) || (IR_req_W1 == 1'b1) ||
   (Addr_req_W == 1'b1) || (change_a0 == 1'b1) ||
   (data_req_W == 1'b1)) //||
  next_state = `execute_instr;
else
  next_state = `get_next_instr;
        end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or
    data_req_W or fifo_full or valid_data )begin
  case(mux_select_fifo_in)
     2'b00:
       begin
         if ((IR_req_W0 == 1'b1) && (fifo_full == 1'b0) &&
 //Instr does not tx data
 ((data_req_W == 1'b0) ||
//Instr does tx data- wait for data to be available before
//sending instruction
((data_req_W == 1'b1) && (valid_data == 1'b1))))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b01:
       begin
         if ((IR_req_W1 == 1'b1) && (fifo_full == 1'b0))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b10:
       begin
         if ((Addr_req_W == 1'b1) && (fifo_full == 1'b0))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
     2'b11:
       begin
         if ((data_req_W == 1'b1) && (fifo_full == 1'b0) &&
 (valid_data == 1'b1))
            fifo_write = 1'b1;
 else
            fifo_write = 1'b0;
       end
  endcase
end

assign read_data = ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b11));

endmodule
```

A.5 Producer Debug Controller

```
//***********************************************************************
//    File: fsl_ce_for_cntr_pe.v
//    Uses: cntr_pe.v and a SIMPPL Control Sequencer
//    Description: Provides a SIMPPL interface to the four bit counter.
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//***********************************************************************
//

`include "instr_defines.v"

module producer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  //Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  //Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface:
  pe_rst,
  pe_data_out,
  pe_read_data,
  pe_can_read,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  valid_data,
  exec_rx_instr,

  //Debugging ports:
  //Debugging download link
  debug_link_clk,
  debug_link_write,
  debug_link_data,
  debug_link_control,
  debug_link_full,

  //Debug Control Signals
  int_error,
  chk_status,
  status_rdy,
  cont_execution,

  // outputs
  led0, // for debug
  led1 // for debug
  );

  input clk, rst;
```

```
output FSL_M_Clk, FSL_M_Write;
output [0:31] FSL_M_Data;
output FSL_M_Control;
input FSL_M_Full;
output FSL_S_Clk, FSL_S_Read;
input [0:31] FSL_S_Data;
input FSL_S_Control, FSL_S_Exists;
output led0, led1; // for debug

//Debugging ports:
//Debugging download link
output debug_link_clk;
output debug_link_write;
output [0:31] debug_link_data;
output debug_link_control;
input debug_link_full;
//Debugging handshake signals
output int_error;
input chk_status;
output status_rdy;
input cont_execution;

//PE Interface:
output pe_rst;
input [31:0] pe_data_out;
output pe_read_data;
input pe_can_read;

//Program Interface:
input [0:31] prog_instr;
input program_cbit, valid_instr;
output prog_instr_read;
input cont_prog;
output valid_data, exec_rx_instr;


//Program interface and Instruction Register Read signals:
wire [0:31] prog_instr;  //connects to the program instrucion input port
wire program_cbit; //connects to the program's control bit
wire valid_instr; //Used to indicate that the instruction from the program is
 //valid
wire cont_prog; //Used to override the receives higher priority
wire prog_instr_read; //reading from the port into the program IR

//Status bits:
wire valid_data; //status bit provided to the program for branching
wire exec_rx_instr; //status bit provided to be used in place of the
 //"wait rx" instr

wire led0, led1; // for debug

//For Debugging  ****
reg [3:0] status_cntr;
reg int_debug_link_write;
reg prev_status_rdy;
reg int_chk_status;
reg int_int_error;
wire CE_error;
wire error0, error1, error2, error3, error4, error5, error6, error7, error8;

//Debug ****
reg [0:31] a0_reg_bak, imm_addr, controller_status_reg, prog_PE_status_reg;
reg [0:31] ex_IR_bak, data_cntr_bak, prog_IR_bak, rx_IR_bak;
reg[0:31] err_type_reg, exec_time_cntr;


//IR signals:
wire  rx_IR_full;
```

```
  wire [0:31] rx_IR;
  wire  rx_cbit;  //receive control bit
  wire prog_IR_full;
  wire [0:31] prog_IR;
  wire prog_cbit;  //program control bit
  reg exec_prog; //am I executing the program or an rx_IR**only valid when
   //ex_IR_full is high

  //Address registers:
  reg [0:31] a0_reg; //Address register 0
  reg [0:23] a0_operand; //latches the data portion of the instruction
  reg write_a0;

  //Mux control signals:
  wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
  reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

  wire control_reset;  //combines system reset and instruction reset
  reg instr_reset; //generated via the reset instruction- clock period long

  //Used to interface the ex_IR, A0 and data with the fifo:
  reg [1:0] mux_select_fifo_in; //used to select
  wire [0:7] ex_IR_fifo_in;
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  //Indicates when the address register needs to be written to
   //the fifo
  reg data_req_W;  // Indicates that there is data to be written to the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire [0:31] ex_IR_cbit;  //wire used as mux output from
      //prog and rx IR’s cbits
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

  reg [1:0] prev_state; //Debug
  reg [1:0] pres_state;
  reg [1:0] next_state;

  reg fifo_write; //write data to the Master FSL
  wire fifo_full;

  wire read_data; //read data from the pe to the Master FSL


//States for overall execution path state machine:
‘define get_next_instr 0
‘define decode_instr 1
‘define execute_instr 2
‘define status_chk 3 //Debug

//Debug: ****
‘define num_regs 9 //Debug
‘define bit 31-27 //The subtracted number represents the power of two bit


// for debug
```

```
assign led0 = exec_rx_instr ? 1'b0 : 1'b1;
assign led1 = valid_data ? 1'b0 : 1'b1;

//Slave signals:
assign FSL_S_Clk = clk;
//Used to Read the Slave fsl data:
assign FSL_S_Read = reading_rx_IR;
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

//Master signals:
assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1'b1) ? ~ex_cbit_input : ex_cbit_input;
//Used to Write the data from the counter to the FSL:
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4'b0000;
assign ex_IR_fifo_in[4] = (opcode == `bypass) ? opcode[3] : 1'b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1'b1) ? opcode[1:0] : ~opcode[1:0];

assign pe_rst = control_reset;
assign pe_read_data = read_data;

//Status bits received from the PE and sent to the program via the controller:
assign valid_data = pe_can_read;

//Status bit 2:
assign exec_rx_instr = ((pres_state != `get_next_instr) && (exec_prog == 1'b0));

//Used to Read an instruction into the Executing IR:
assign ex_instr_read = ((pres_state == `get_next_instr)
&& (control_reset == 1'b0)
  && (((rx_IR_full && ~cont_prog )|| prog_IR_full)));

//Used to mux the outputs from the two different IRs such that the rx_IR
//takes precedence when the cont_prog flag is low:
assign sel_ex_IR_input = (pres_state == `get_next_instr) ?
((rx_IR_full==1'b1) && (cont_prog==1'b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_IR_cbit = sel_ex_IR_input ? rx_cbit : prog_cbit ; //Debug
assign ex_cbit_input = (mux_select_fifo_in == 2'b00) ? IR_req_W0 : 1'b0;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b0) && (prog_IR_full == 1'b1));

//Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b1) && (rx_IR_full == 1'b1));

//Used to multiplex an immediate or incremented/decremented value to a0
```

```
assign a0_mux_output = ((opcode[0] == 1'b1) && (opcode[2] == 1'b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal:
assign control_reset = rst || instr_reset;

//Used to generate the controller's reset signal from the reset instruction:
always @ (posedge clk) begin
  if (rst == 1'b1)
instr_reset <= 1'b0;
  else if ((pres_state == `execute_instr) && (opcode == 8'h00) &&
  (exec_prog == 1'b0))
instr_reset <= 1'b1;
  else
instr_reset <= 1'b0;
end

//Used to add/subtract the a0 operand to the address register
always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1'b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end

//The address register
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_reg <= 32'h00000000;
  else if (write_a0 == 1'b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end

//Used to store the initial NDW value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end

//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ( (change_a0 == 1'b1) &&
            (Addr_req_W == 1'b0) )
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

//Used to indicate that the value of the address register will change
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == `decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == `execute_instr))) && (opcode == `bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == `execute_instr) && (opcode != `bypass)))
   && (fifo_write == 1'b1))
    tx_bypass <= 1'b0;
  else
    tx_bypass <= tx_bypass;
end

//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W0 <= 1'b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == `decode_instr) && (((exec_prog == 1'b1) &&
  ((opcode[3:0] != 4'b1000) && (opcode[3:0] != 4'b1011)))
  || (opcode[1:0] == 2'b10)))
    IR_req_W0 <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b00))
    IR_req_W0 <= 1'b0;
  else
    IR_req_W0 <= IR_req_W0;
end

// Used to indicate when to write immediate values from the receive
// link or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W1 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[2] == 1'b1) &&
  (((exec_prog == 1'b1) && ((opcode[0] == 1'b1) || (opcode[1]==1'b1))
  && (opcode[3] == 1'b0)) //Not an Areg operation
  || ((exec_prog == 1'b0) && ((opcode == `bypass) ||
  (opcode[1:0] == 2'b10))))))
    IR_req_W1 <= 1'b1;
  else if (((opcode != `bypass) ||
  ((opcode == `bypass) && (data_cntr == 24'h000001))) &&
  (fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
    IR_req_W1 <= 1'b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    Addr_req_W <= 1'b0;
  else if ((pres_state == `decode_instr) && (opcode[3:2] == 2'b11) &&
   (opcode[1:0] != 2'b00))
    Addr_req_W <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b10))
    Addr_req_W <= 1'b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Controls the request to write data to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_req_W <= 1'b0;
  else if ((pres_state == `decode_instr) &&
(((exec_prog == 1'b1) && (opcode[0] == 1'b1)) ||
       ((exec_prog == 1'b0) && (opcode[1:0] == 2'b10))) &&
       (opcode[3:2] != 2'b10))
    data_req_W <= 1'b1;
  else if ((data_cntr == 24'h000001) && (fifo_write == 1'b1)
   //needed for when only 1 data word is sent
  && (mux_select_fifo_in == 2'b11))
    data_req_W <= 1'b0;
  else
    data_req_W <= data_req_W;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_cntr <= 24'h000000;
  else if ((pres_state == `decode_instr) &&
(((exec_prog == 1'b1) && (opcode[0] == 1'b1)) ||
       ((exec_prog == 1'b0) && ((opcode[1:0] == 2'b10) ||
        (opcode == `bypass)))) && (opcode[3:2] != 2'b10))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2'b11) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)))
  && (fifo_write == 1'b1))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end

// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W) begin
if ((pres_state == `execute_instr) && (IR_req_W0 == 1'b1))
      mux_select_fifo_in = 2'b00;
else if ((pres_state == `execute_instr) && (IR_req_W1 == 1'b1))
      mux_select_fifo_in = 2'b01;
else if ((pres_state == `execute_instr) && (Addr_req_W == 1'b1))
      mux_select_fifo_in = 2'b10;
else
      mux_select_fifo_in = 2'b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    opcode <= 8'b0;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end

// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1'b1)
    exec_prog <= 1'b1;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b1))
    exec_prog <= 1'b0;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b0))
    exec_prog <= 1'b1;
  else
    exec_prog <= exec_prog;
end

// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand
    or pe_data_out) begin
  case (mux_select_fifo_in)
    2'b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2'b01:
      fifo_mux_out = ex_IR;
    2'b10:
      fifo_mux_out = a0_reg;
    2'b11:
      fifo_mux_out = pe_data_out;
  endcase
end

always @ (posedge clk) begin
  if (control_reset == 1'b1)
    pres_state <= `get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or ex_instr_read or IR_req_W0 or IR_req_W1 or Addr_req_W
    or change_a0 or data_req_W or error0 or error1 or error2 or error3 or
    error4 or error5 or error6 or error7 or error8 or int_int_error or
    int_chk_status or status_cntr or cont_execution) begin
  case (pres_state)
    `get_next_instr:
      begin
        if ((error1 == 1'b1) || (error3 == 1'b1) || (error5 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else if ((ex_instr_read == 1'b1))
         next_state = `decode_instr;
        else
         next_state = `get_next_instr;
      end
    `decode_instr:
        if ((error0 == 1'b1) || (error5 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else
        next_state = `execute_instr;
    `execute_instr:
      begin
        if ((error0 == 1'b1) || (error2 == 1'b1) || (error4 == 1'b1) || //Debug
            (error5 == 1'b1) || (error6 == 1'b1) || (error7 == 1'b1) || //Debug
            (error8 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
        else if ((int_int_error == 1'b1) || ((int_chk_status == 1'b1) && //Debug
((IR_req_W0 == 1'b0) && (IR_req_W1 == 1'b0) &&  //Debug
   (Addr_req_W == 1'b0) && (change_a0 == 1'b0) &&  //Debug
   (data_req_W == 1'b0)))) //Debug
  next_state = `status_chk; //Debug
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
else if ((IR_req_W0 == 1'b1) || (IR_req_W1 == 1'b1) ||
   (Addr_req_W == 1'b1) || (change_a0 == 1'b1) ||
   (data_req_W == 1'b1))
  next_state = `execute_instr;
else
  next_state = `get_next_instr;
      end
    `status_chk: //Debug ****
      begin
```

```
//if ((((status_cntr == ‘num_regs-2) && (int_debug_link_write == 1’b1))
// || (status_cntr == ‘num_regs-1))
if ((status_cntr == ‘num_regs-1) &&
(cont_execution == 1’b1) && (int_int_error == 1’b0)) //Debug
next_state = ‘get_next_instr;
else
next_state = ‘status_chk;
      end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or
    data_req_W or fifo_full or valid_data )begin
  case(mux_select_fifo_in)
     2’b00:
       begin
         if ((IR_req_W0 == 1’b1) && (fifo_full == 1’b0) &&
 //Instr does not tx data
 ((data_req_W == 1’b0) ||
//Instr does tx data- wait for data to be available before
//sending instruction
((data_req_W == 1’b1) && (valid_data == 1’b1))))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b01:
       begin
         if ((IR_req_W1 == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b10:
       begin
         if ((Addr_req_W == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b11:
       begin
         if ((data_req_W == 1’b1) && (fifo_full == 1’b0) &&
 (valid_data == 1’b1))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
  endcase
end

assign read_data = ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b11));

//Debug register download order:
//0)int_debug_link_data <= err_type_reg;
//1)int_debug_link_data <= ex_IR_bak;
//2)int_debug_link_data <= imm_addr;
//3)int_debug_link_data <= a0_reg_bak;
//4)int_debug_link_data <= data_cntr_bak;
//5)int_debug_link_data <= prog_IR_bak;
//6)int_debug_link_data <= rx_IR_bak;
//7)int_debug_link_data <= controller_status_reg;
//8)int_debug_link_data <= prog_PE_status_reg;
//8)int_debug_link_data <= exec_time_cntr;

assign debug_link_clk = clk; //Debug
assign debug_link_data = err_type_reg; //Debug
assign debug_link_control = 1’b0; //Debug
assign int_error = int_int_error; //Debug
assign CE_error = (error0 || error1 || error2 || error3 || error4 || error5 ||
error6 || error7 || error8);

//Debug Errors:
assign error0 = ((((prog_cbit == 1'b1) && (reading_prog_IR == 1'b1)) ||
 ((rx_cbit == 1'b1) && (reading_rx_IR == 1'b1))) &&
 (pres_state != `get_next_instr));
assign error1 = ((pres_state == `get_next_instr) &&
(((rx_IR_full == 1'b1) && (rx_cbit == 1'b0)) ||
((prog_IR_full == 1'b1) && (prog_cbit == 1'b0))));
assign error2 = ((pres_state == `execute_instr) &&
(exec_time_cntr[`bit] == 1'b1));
assign error3 = ((pres_state == `get_next_instr) &&
(exec_time_cntr[`bit] == 1'b1));
assign error4 = ((FSL_M_Write == 1'b1) && (FSL_M_Full == 1'b1));
assign error5 = ((FSL_S_Read == 1'b1) && (FSL_S_Exists == 1'b0));
//assign error6 = ((pe_write_data == 1'b1) && (pe_can_write_data == 1'b0));
assign error6 = 1'b0;
//assign error7 = ((pe_write_addr == 1'b1) && (pe_can_write_addr == 1'b0));
assign error7 = 1'b0;
assign error8 = ((pe_read_data == 1'b1) && (pe_can_read == 1'b0)); //producer

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_int_error <= 1'b0;
  else if (CE_error == 1'b1)
    int_int_error <= 1'b1;
  else
    int_int_error <= int_int_error;
end

//Debug
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_state <= `get_next_instr;
  else
    prev_state <= pres_state;
end

//Debug:
//Status info is ready when all the registers have been downloaded to the
//debug link fifo
assign status_rdy = (status_cntr == `num_regs-1) ? 1'b1 : 1'b0; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_status_rdy <= 1'b0;
  else
    prev_status_rdy <= status_rdy;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_chk_status <= 1'b0;
  else if (chk_status == 1'b1)
    int_chk_status <= 1'b1;
  else if (status_rdy == 1'b1)
    int_chk_status <= 1'b0;
  else
    int_chk_status <= int_chk_status;
end

//Debug:
always @ (posedge clk) begin
```

```
  if (rst == 1'b1)
    int_debug_link_write <= 1'b0;
  else if ((pres_state == `status_chk) && (debug_link_full == 1'b0) &&
   (status_rdy == 1'b0))
    int_debug_link_write <= 1'b1;
  else
    int_debug_link_write <= 1'b0;
end

assign debug_link_write = int_debug_link_write; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    status_cntr <= 4'h0;
  else if ((prev_state == `status_chk) && (pres_state == `get_next_instr))
    status_cntr <= 4'h0;
  else if ((pres_state == `status_chk) && (int_debug_link_write == 1'b1))
    status_cntr <= status_cntr + 1;
  else
    status_cntr <= status_cntr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    err_type_reg <= 32'h00000000;
  else if (int_int_error == 1'b0)
    begin
      err_type_reg[0] <= error0;
      err_type_reg[1] <= error1;
      err_type_reg[2] <= error2;
      err_type_reg[3] <= error3;
      err_type_reg[4] <= error4;
      err_type_reg[5] <= error5;
      err_type_reg[6] <= error6;
      err_type_reg[7] <= error7;
      err_type_reg[8] <= error8;
      err_type_reg[9:30] <= 22'h000000;
      err_type_reg[31] <= CE_error;
    end
  else if (int_debug_link_write == 1'b1)
    err_type_reg <= ex_IR_bak;
  else
    err_type_reg <= err_type_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    ex_IR_bak <= 32'h00000000;
  else if (((pres_state == `status_chk) || (pres_state == `execute_instr)) &&
  (next_state == `get_next_instr))
    ex_IR_bak <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    ex_IR_bak <= imm_addr;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    ex_IR_bak <= ex_IR;
  else
    ex_IR_bak <= ex_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    imm_addr <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    imm_addr <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    imm_addr <= a0_reg_bak;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01) &&
   (opcode != `bypass))
    imm_addr <= ex_IR;
  else
    imm_addr <= imm_addr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    a0_reg_bak <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    a0_reg_bak <= a0_reg;
  else if (int_debug_link_write == 1'b1)
    a0_reg_bak <= data_cntr_bak;
  else if (write_a0 == 1'b1)
    a0_reg_bak <= a0_mux_output;
  else
    a0_reg_bak <= a0_reg_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    data_cntr_bak <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    data_cntr_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    data_cntr_bak <= prog_IR_bak;
  else if (pres_state == `decode_instr)
    data_cntr_bak <= {8'h00,a0_operand};
  else if (((mux_select_fifo_in == 2'b11) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)))
  && (fifo_write == 1'b1))
    data_cntr_bak <= data_cntr_bak - 1;
  else
    data_cntr_bak <= data_cntr_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prog_IR_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    prog_IR_bak <= rx_IR_bak;
  else if (pres_state != `status_chk)
    prog_IR_bak <= prog_IR;
  else
    prog_IR_bak <= prog_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    rx_IR_bak <= 32'h00000000;
  else if (int_debug_link_write == 1'b1)
    rx_IR_bak <= controller_status_reg;
  else if (pres_state != `status_chk)
    rx_IR_bak <= rx_IR;
  else
    rx_IR_bak <= rx_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
```

```
    controller_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    controller_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    controller_status_reg <= prog_PE_status_reg;
  else if (((pres_state == `get_next_instr) || (pres_state == `decode_instr))
   || ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      controller_status_reg[0] <= ex_IR_cbit;
      controller_status_reg[1] <= IR_req_W0;
      controller_status_reg[2] <= IR_req_W1;
      controller_status_reg[3] <= Addr_req_W;
      controller_status_reg[4] <= data_req_W;
      controller_status_reg[5] <= tx_bypass;
      controller_status_reg[6] <= exec_prog;
      //controller_status_reg[7] <= rx_addr_req;
      //controller_status_reg[8] <= rx_data_req;
      controller_status_reg[7] <= 1'b0;
      controller_status_reg[8] <= 1'b0;
      controller_status_reg[9] <= change_a0;
      controller_status_reg[10:15] <= 6'h00;
      controller_status_reg[16] <= ex_IR_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      controller_status_reg[20] <= data_req_W;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      //controller_status_reg[23] <= rx_addr_req;
      //controller_status_reg[24] <= rx_data_req;
      controller_status_reg[23] <= 1'b0;
      controller_status_reg[24] <= 1'b0;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else if (pres_state == `execute_instr)
    begin
      controller_status_reg[0:15] <= controller_status_reg[0:15];
      controller_status_reg[16] <= ex_IR_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      controller_status_reg[20] <= data_req_W;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      //controller_status_reg[23] <= rx_addr_req;
      //controller_status_reg[24] <= rx_data_req;
      controller_status_reg[23] <= 1'b0;
      controller_status_reg[24] <= 1'b0;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else
    controller_status_reg <= controller_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prog_PE_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    prog_PE_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    prog_PE_status_reg <= exec_time_cntr;
  else if ((pres_state == `get_next_instr) || (pres_state == `decode_instr) ||
  ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      prog_PE_status_reg[0] <= rx_IR_full;
      prog_PE_status_reg[1] <= prog_IR_full;
      prog_PE_status_reg[2] <= rx_cbit;
      prog_PE_status_reg[3] <= prog_cbit;
      prog_PE_status_reg[4] <= valid_instr;
      prog_PE_status_reg[5] <= cont_prog;
      prog_PE_status_reg[6] <= pe_can_read;
      //prog_PE_status_reg[7] <= pe_can_write_data;
      //prog_PE_status_reg[8] <= pe_can_write_addr;
      prog_PE_status_reg[7] <= 1'b0;
      prog_PE_status_reg[8] <= 1'b0;
      prog_PE_status_reg[9:10] <= pres_state;
      prog_PE_status_reg[11:12] <= mux_select_fifo_in;
      prog_PE_status_reg[13:15] <= 3'h0;
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      prog_PE_status_reg[22] <= pe_can_read;
      //prog_PE_status_reg[23] <= pe_can_write_data;
      //prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[23] <= 1'b0;
      prog_PE_status_reg[24] <= 1'b0;
      prog_PE_status_reg[25:26] <= pres_state;
      prog_PE_status_reg[27:28] <= mux_select_fifo_in;
      prog_PE_status_reg[29:31] <= 3'h0;
    end
  else if (pres_state == `execute_instr)
    begin
      prog_PE_status_reg[0:15] <= prog_PE_status_reg[0:15];
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      prog_PE_status_reg[22] <= pe_can_read;
      //prog_PE_status_reg[23] <= pe_can_write_data;
      //prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[23] <= 1'b0;
      prog_PE_status_reg[24] <= 1'b0;
      prog_PE_status_reg[25:26] <= pres_state;
      prog_PE_status_reg[27:28] <= mux_select_fifo_in;
      prog_PE_status_reg[29:31] <= 3'h0;
    end
  else
    prog_PE_status_reg <= prog_PE_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    exec_time_cntr <= 32'h00000000;
  else if (((status_rdy == 1'b1) && (prev_status_rdy == 1'b0)) ||
  ((pres_state == `get_next_instr) && (prev_state == `execute_instr))
  || (pres_state == `decode_instr))
    exec_time_cntr <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    exec_time_cntr <= 32'hdeaddead; //END OF PIPELINED DEBUG DATA
  else if((pres_state == `execute_instr) && (pres_state == `get_next_instr))
    exec_time_cntr <= exec_time_cntr + 1;
  else
    exec_time_cntr <= exec_time_cntr;
end

endmodule
```

A.6 Full Execute Controller

```
//**********************************************************************
//    File: fsl_ce_for_cntr_pe.v
//    Uses: cntr_pe.v and a SIMPPL Control Sequencer
//    Description: Provides a SIMPPL interface to the four bit counter.
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//**********************************************************************
//

`include "instr_defines.v"

module producer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  //Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  //Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface:
  pe_rst,
  pe_data_out,
  pe_read_data,
  pe_can_read,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  valid_data,
  exec_rx_instr,

  // outputs
  led0, // for debug
  led1 // for debug
  );

  input clk, rst;
  output FSL_M_Clk, FSL_M_Write;
  output [0:31] FSL_M_Data;
  output FSL_M_Control;
  input FSL_M_Full;
  output FSL_S_Clk, FSL_S_Read;
  input [0:31] FSL_S_Data;
  input FSL_S_Control, FSL_S_Exists;
  output led0, led1; // for debug

  //PE Interface:
  output pe_rst;
  input [31:0] pe_data_out;
  output pe_read_data;
  input pe_can_read;
```

```
//Program Interface:
input [0:31] prog_instr;
input program_cbit, valid_instr;
output prog_instr_read;
input cont_prog;
output valid_data, exec_rx_instr;

wire clk, rst;
wire FSL_M_Clk, FSL_M_Full;
wire FSL_M_Control;
wire FSL_M_Write;
wire [0:31] FSL_M_Data;

//Slave FSL wires:
wire FSL_S_Clk;
wire FSL_S_Read;
wire [0:31] FSL_S_Data;
wire FSL_S_Control;
wire FSL_S_Exists;

//PE Interface:
wire pe_rst;
wire [31:0] pe_data_out;
wire pe_read_data;
wire pe_can_read;

//Program interface and Instruction Register Read signals:
wire [0:31] prog_instr;  //connects to the program instrucion input port
wire program_cbit; //connects to the program’s control bit
wire valid_instr; //Used to indicate that the instruction from the program is
 //valid
wire cont_prog; //Used to override the receives higher priority
wire prog_instr_read; //reading from the port into the program IR

//Status bits:
//wire new_transaction; //status bit provided to the program for branching
wire valid_data; //status bit provided to the program for branching
wire exec_rx_instr; //status bit provided to be used in place of the
 //"wait rx" instr

wire led0, led1; // for debug

//IR signals:
wire  rx_IR_full;
wire [0:31] rx_IR;
wire  rx_cbit;  //receive control bit
wire prog_IR_full;
wire [0:31] prog_IR;
wire prog_cbit;  //program control bit
reg exec_prog; //am I executing the program or an rx_IR**only valid when
 //ex_IR_full is high

//Address registers:
reg [0:31] a0_reg; //Address register 0
reg [0:23] a0_operand; //latches the data portion of the instruction
reg write_a0;

//Mux control signals:
wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

wire control_reset;  //combines system reset and instruction reset
reg instr_reset; //generated via the reset instruction- clock period long

//Used to interface the ex_IR, A0 and data with the fifo:
reg [1:0] mux_select_fifo_in; //used to select
wire [0:7] ex_IR_fifo_in;
```

```
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  //Indicates when the address register needs to be written to
   //the fifo
  reg data_req_W;  // Indicates that there is data to be written to the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

  reg [1:0] pres_state;
  reg [1:0] next_state;

  reg fifo_write; //write data to the Master FSL
  wire fifo_full;

  wire read_data; //read data from the pe to the Master FSL

//States for overall execution path state machine:
‘define get_next_instr 0
‘define decode_instr 1
‘define execute_instr 2

// for debug
assign led0 = exec_rx_instr ? 1’b0 : 1’b1;
assign led1 = valid_data ? 1’b0 : 1’b1;

//Slave signals:
assign FSL_S_Clk = clk;
//Used to Read the Slave fsl data:
assign FSL_S_Read = reading_rx_IR;
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

//Master signals:
assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1’b1) ? ~ex_cbit_input : ex_cbit_input;
//Used to Write the data from the counter to the FSL:
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4’b0000;
assign ex_IR_fifo_in[4] = (opcode == ‘bypass) ? opcode[3] : 1’b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1’b1) ? opcode[1:0] : ~opcode[1:0];

assign pe_rst = control_reset;
assign pe_read_data = read_data;

//Status bits received from the PE and sent to the program via the controller:
assign valid_data = pe_can_read;

//Status bit 2:
assign exec_rx_instr = ((pres_state != ‘get_next_instr) && (exec_prog == 1’b0));
```

```
//Used to Read an instruction into the Executing IR:
assign ex_instr_read = ((pres_state == ‘get_next_instr)
&& (control_reset == 1’b0)
  && (((rx_IR_full && ~cont_prog )|| prog_IR_full)));

//Used to mux the outputs from the two different IRs such that the rx_IR
//takes precedence when the cont_prog flag is low:
assign sel_ex_IR_input = (pres_state == ‘get_next_instr) ?
((rx_IR_full==1’b1) && (cont_prog==1’b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_cbit_input = (mux_select_fifo_in == 2’b00) ? IR_req_W0 : 1’b0;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1’b1) ||
((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
|| ((write_a0 == 1’b1) && (opcode[0] == 1’b1) &&
(opcode[2] == 1’b0)))
&& (sel_ex_IR_input == 1’b0) && (prog_IR_full == 1’b1));

//Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1’b1) ||
((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
|| ((write_a0 == 1’b1) && (opcode[0] == 1’b1) &&
(opcode[2] == 1’b0)))
&& (sel_ex_IR_input == 1’b1) && (rx_IR_full == 1’b1));

//Used to multiplex an immediate or incremented/decremented value to a0
assign a0_mux_output = ((opcode[0] == 1’b1) && (opcode[2] == 1’b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal:
assign control_reset = rst || instr_reset;

//Used to generate the controller’s reset signal from the reset instruction:
always @ (posedge clk) begin // or posedge rst) begin
  if (rst == 1’b1)
instr_reset <= 1’b0;
  else if ((pres_state == ‘execute_instr) && (opcode == 8’h00) &&
  (exec_prog == 1’b0))
instr_reset <= 1’b1;
  else
instr_reset <= 1’b0;
end

//Used to add/subtract the a0 operand to the address register
always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1’b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end

//The address register
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    a0_reg <= 32’h00000000;
  else if (write_a0 == 1’b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end
```

```
//Used to store the initial NDW value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end

//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ( (change_a0 == 1'b1) &&
             (Addr_req_W == 1'b0) )
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

//Used to indicate that the value of the address register will change
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == `decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == `execute_instr))) && (opcode == `bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == `execute_instr) && (opcode != `bypass)))
   && (fifo_write == 1'b1))
    tx_bypass <= 1'b0;
  else
    tx_bypass <= tx_bypass;
end

//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W0 <= 1'b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == `decode_instr) && (((exec_prog == 1'b1) &&
  ((opcode[3:0] != 4'b1000) && (opcode[3:0] != 4'b1011)))
  || (opcode[1:0] == 2'b10)))
    IR_req_W0 <= 1'b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b00))
    IR_req_W0 <= 1’b0;
  else
    IR_req_W0 <= IR_req_W0;
end

// Used to indicate when to write immediate values from the receive
// link or program
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    IR_req_W1 <= 1’b0;
  else if ((pres_state == ‘decode_instr) && ((opcode[2] == 1’b1) &&
  (((exec_prog == 1’b1) && ((opcode[0] == 1’b1) || (opcode[1]==1’b1))
  && (opcode[3] == 1’b0)) //Not an Areg operation
  || ((exec_prog == 1’b0) && ((opcode == ‘bypass) ||
  (opcode[1:0] == 2’b10))))))
    IR_req_W1 <= 1’b1;
  else if (((opcode != ‘bypass) ||
  ((opcode == ‘bypass) && (data_cntr == 24’h000001))) &&
  (fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01))
    IR_req_W1 <= 1’b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    Addr_req_W <= 1’b0;
  else if ((pres_state == ‘decode_instr) && (opcode[3:2] == 2’b11) &&
   (opcode[1:0] != 2’b00))
    Addr_req_W <= 1’b1;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b10))
    Addr_req_W <= 1’b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Controls the request to write data to the fifo
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    data_req_W <= 1’b0;
  else if ((pres_state == ‘decode_instr) &&
(((exec_prog == 1’b1) && (opcode[0] == 1’b1)) ||
       ((exec_prog == 1’b0) && (opcode[1:0] == 2’b10))) &&
       (opcode[3:2] != 2’b10))
    data_req_W <= 1’b1;
  else if ((data_cntr == 24’h000001) && (fifo_write == 1’b1)
   //needed for when only 1 data word is sent
  && (mux_select_fifo_in == 2’b11))
    data_req_W <= 1’b0;
  else
    data_req_W <= data_req_W;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1’b1)
    data_cntr <= 24’h000000;
  else if ((pres_state == ‘decode_instr) &&
(((exec_prog == 1’b1) && (opcode[0] == 1’b1)) ||
       ((exec_prog == 1’b0) && ((opcode[1:0] == 2’b10) ||
        (opcode == ‘bypass)))) && (opcode[3:2] != 2’b10))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2’b11) ||
  ((opcode == ‘bypass) && (mux_select_fifo_in == 2’b01)))
  && (fifo_write == 1’b1))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end

// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W) begin
if ((pres_state == `execute_instr) && (IR_req_W0 == 1'b1))
      mux_select_fifo_in = 2'b00;
else if ((pres_state == `execute_instr) && (IR_req_W1 == 1'b1))
      mux_select_fifo_in = 2'b01;
else if ((pres_state == `execute_instr) && (Addr_req_W == 1'b1))
      mux_select_fifo_in = 2'b10;
else
      mux_select_fifo_in = 2'b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    opcode <= 8'b0;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end
// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    exec_prog <= 1'b1;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b1))
    exec_prog <= 1'b0;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b0))
    exec_prog <= 1'b1;
  else
    exec_prog <= exec_prog;
end

// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand
    or pe_data_out) begin
  case (mux_select_fifo_in)
    2'b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2'b01:
      fifo_mux_out = ex_IR;
    2'b10:
      fifo_mux_out = a0_reg;
    2'b11:
      //fifo_mux_out = cntr_pe_data;
      fifo_mux_out = pe_data_out;
  endcase
end

always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1'b1)
    pres_state <= `get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or ex_instr_read or IR_req_W0 or IR_req_W1 or Addr_req_W
    or change_a0 or data_req_W) begin
  case (pres_state)
    `get_next_instr:
      begin
if ((ex_instr_read == 1'b1))
         next_state = `decode_instr;
```

```
        else
         next_state = ‘get_next_instr;
      end
    ‘decode_instr:
      next_state = ‘execute_instr;
    ‘execute_instr:
      begin
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
if ((IR_req_W0 == 1’b1) || (IR_req_W1 == 1’b1) ||
   (Addr_req_W == 1’b1) || (change_a0 == 1’b1) ||
   (data_req_W == 1’b1)) //||
  next_state = ‘execute_instr;
else
  next_state = ‘get_next_instr;
        end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or
    data_req_W or fifo_full or valid_data )begin
  case(mux_select_fifo_in)
     2’b00:
       begin
         if ((IR_req_W0 == 1’b1) && (fifo_full == 1’b0) &&
 //Instr does not tx data
 ((data_req_W == 1’b0) ||
//Instr does tx data- wait for data to be available before
//sending instruction
((data_req_W == 1’b1) && (valid_data == 1’b1))))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b01:
       begin
         if ((IR_req_W1 == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b10:
       begin
         if ((Addr_req_W == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b11:
       begin
         if ((data_req_W == 1’b1) && (fifo_full == 1’b0) &&
 (valid_data == 1’b1))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
  endcase
end

assign read_data = ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b11));

endmodule
```

A.7 Full Debug Controller

```
//*************************************************************************
//    File: fsl_ce_for_cntr_pe.v
//    Uses: cntr_pe.v and a SIMPPL Control Sequencer
//    Description: Provides a SIMPPL interface to the four bit counter.
//    Uses the SIMPPL Communication Protocol
//
//    LLS April 2005
//*************************************************************************
//

‘include "instr_defines.v"

module producer_controller (
  // inputs
  clk,
  rst, // rst should be driven by FSL_Rst

  //Master FSL Signals
  FSL_M_Clk,
  FSL_M_Write,
  FSL_M_Data,
  FSL_M_Control,
  FSL_M_Full,

  //Slave FSL Signals
  FSL_S_Clk,
  FSL_S_Read,
  FSL_S_Data,
  FSL_S_Control,
  FSL_S_Exists,

  //PE Interface:
  pe_rst,
  pe_data_out,
  pe_read_data,
  pe_can_read,

  //Program Interface
  prog_instr,
  program_cbit,
  prog_instr_read,
  valid_instr,
  cont_prog,
  valid_data,
  exec_rx_instr,

  //Debugging ports:
  //Debugging download link
  debug_link_clk,
  debug_link_write,
  debug_link_data,
  debug_link_control,
  debug_link_full,

  //Debug Control Signals
  int_error,
  chk_status,
  status_rdy,
  cont_execution,

  // outputs
  led0, // for debug
  led1 // for debug
  );

  input clk, rst;
```

```
output FSL_M_Clk, FSL_M_Write;
output [0:31] FSL_M_Data;
output FSL_M_Control;
input FSL_M_Full;
output FSL_S_Clk, FSL_S_Read;
input [0:31] FSL_S_Data;
input FSL_S_Control, FSL_S_Exists;
output led0, led1; // for debug

//Debugging ports:
//Debugging download link
output debug_link_clk;
output debug_link_write;
output [0:31] debug_link_data;
output debug_link_control;
input debug_link_full;
//Debugging handshake signals
output int_error;
input chk_status;
output status_rdy;
input cont_execution;

//PE Interface:
output pe_rst;
input [31:0] pe_data_out;
output pe_read_data;
input pe_can_read;

//Program Interface:
input [0:31] prog_instr;
input program_cbit, valid_instr;
output prog_instr_read;
input cont_prog;
output valid_data, exec_rx_instr;


//Program interface and Instruction Register Read signals:
wire [0:31] prog_instr;  //connects to the program instrucion input port
wire program_cbit; //connects to the program’s control bit
wire valid_instr; //Used to indicate that the instruction from the program is
 //valid
wire cont_prog; //Used to override the receives higher priority
wire prog_instr_read; //reading from the port into the program IR

//Status bits:
wire valid_data; //status bit provided to the program for branching
wire exec_rx_instr; //status bit provided to be used in place of the
 //"wait rx" instr

wire led0, led1; // for debug

//For Debugging  ****
reg [3:0] status_cntr;
reg int_debug_link_write;
reg prev_status_rdy;
reg int_chk_status;
reg int_int_error;
wire CE_error;
wire error0, error1, error2, error3, error4, error5, error6, error7, error8;

//Debug ****
reg [0:31] a0_reg_bak, imm_addr, controller_status_reg, prog_PE_status_reg;
reg [0:31] ex_IR_bak, data_cntr_bak, prog_IR_bak, rx_IR_bak;
reg[0:31] err_type_reg, exec_time_cntr;


//IR signals:
wire  rx_IR_full;
```

```
  wire [0:31] rx_IR;
  wire  rx_cbit;  //receive control bit
  wire prog_IR_full;
  wire [0:31] prog_IR;
  wire prog_cbit;  //program control bit
  reg exec_prog; //am I executing the program or an rx_IR**only valid when
   //ex_IR_full is high

  //Address registers:
  reg [0:31] a0_reg; //Address register 0
  reg [0:23] a0_operand; //latches the data portion of the instruction
  reg write_a0;

  //Mux control signals:
  wire [0:31] a0_mux_output; //multiplexer output to the a0 reg
  reg [0:31] fifo_mux_out; //output from multiplexer into buffering fifo

  wire control_reset;  //combines system reset and instruction reset
  reg instr_reset; //generated via the reset instruction- clock period long

  //Used to interface the ex_IR, A0 and data with the fifo:
  reg [1:0] mux_select_fifo_in; //used to select
  wire [0:7] ex_IR_fifo_in;
  reg [7:0] opcode;  //Used to store the opcode for 2-part instructions

  //Control registers:
  reg change_a0;  //Flag to indicate when instruction will alter the value of a0
  reg IR_req_W0;  // Indicates when the instruction writes to the fifo
  reg IR_req_W1;  // Indicates when the instruction writes an immed to the fifo
  reg Addr_req_W;  //Indicates when the address register needs to be written to
   //the fifo
  reg data_req_W;  // Indicates that there is data to be written to the fifo
  reg tx_bypass; // Used to alter the control bit transferred as part of a
   // bypass instruction (from program or rx link).
  reg [0:31] new_a0_total;  //Output from Accumulator;
  reg [23:0] data_cntr; // used to store the number of data words left to write

  wire [0:31] ex_IR;  //wire used as mux output from prog and rx IRs
  wire [0:31] ex_IR_cbit;  //wire used as mux output from
      //prog and rx IR's cbits
  wire ex_cbit_input; //wire used as mux output from prog and rx cbits
  wire ex_instr_read;  //reads an instruction into the ex_IR
  wire reading_prog_IR;  //reading the prog_IR clears it
  wire reading_rx_IR;  //reading the rx_IR clears it
  wire sel_ex_IR_input; // selects the input to the ex_IR

  reg [1:0] prev_state; //Debug
  reg [1:0] pres_state;
  reg [1:0] next_state;

  reg fifo_write; //write data to the Master FSL
  wire fifo_full;

  wire read_data; //read data from the pe to the Master FSL


//States for overall execution path state machine:
`define get_next_instr 0
`define decode_instr 1
`define execute_instr 2
`define status_chk 3 //Debug

//Debug: ****
`define num_regs 9 //Debug
`define bit 31-27 //The subtracted number represents the power of two bit


// for debug
```

```
assign led0 = exec_rx_instr ? 1'b0 : 1'b1;
assign led1 = valid_data ? 1'b0 : 1'b1;

//Slave signals:
assign FSL_S_Clk = clk;
//Used to Read the Slave fsl data:
assign FSL_S_Read = reading_rx_IR;
assign rx_IR = FSL_S_Data;
assign rx_cbit = FSL_S_Control;
assign rx_IR_full = FSL_S_Exists;

//Master signals:
assign FSL_M_Clk = clk;
assign FSL_M_Data = fifo_mux_out;
assign FSL_M_Control = (tx_bypass == 1'b1) ? ~ex_cbit_input : ex_cbit_input;
//Used to Write the data from the counter to the FSL:
assign FSL_M_Write = fifo_write;
assign fifo_full = FSL_M_Full;

assign ex_IR_fifo_in[0:3] = 4'b0000;
assign ex_IR_fifo_in[4] = (opcode == `bypass) ? opcode[3] : 1'b0;
assign ex_IR_fifo_in[5] = opcode[2];
assign ex_IR_fifo_in[6:7] = (exec_prog == 1'b1) ? opcode[1:0] : ~opcode[1:0];

assign pe_rst = control_reset;
assign pe_read_data = read_data;

//Status bits received from the PE and sent to the program via the controller:
assign valid_data = pe_can_read;

//Status bit 2:
assign exec_rx_instr = ((pres_state != `get_next_instr) && (exec_prog == 1'b0));

//Used to Read an instruction into the Executing IR:
assign ex_instr_read = ((pres_state == `get_next_instr)
&& (control_reset == 1'b0)
  && (((rx_IR_full && ~cont_prog )|| prog_IR_full)));

//Used to mux the outputs from the two different IRs such that the rx_IR
//takes precedence when the cont_prog flag is low:
assign sel_ex_IR_input = (pres_state == `get_next_instr) ?
((rx_IR_full==1'b1) && (cont_prog==1'b0)) : ~exec_prog;
assign ex_IR = sel_ex_IR_input ? rx_IR : prog_IR ;
assign ex_IR_cbit = sel_ex_IR_input ? rx_cbit : prog_cbit ; //Debug
assign ex_cbit_input = (mux_select_fifo_in == 2'b00) ? IR_req_W0 : 1'b0;

assign prog_IR = prog_instr;
assign prog_cbit = program_cbit;
assign prog_IR_full = valid_instr;

// Reads the prog_IR into the ex_IR depending on the mux select:
assign reading_prog_IR =  (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b0) && (prog_IR_full == 1'b1));

//Used to Read the Program Instructions:
assign prog_instr_read = reading_prog_IR;

// Reads the rx_IR into the ex_IR depending on the mux select:
assign reading_rx_IR = (((ex_instr_read == 1'b1) ||
((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
|| ((write_a0 == 1'b1) && (opcode[0] == 1'b1) &&
(opcode[2] == 1'b0)))
&& (sel_ex_IR_input == 1'b1) && (rx_IR_full == 1'b1));

//Used to multiplex an immediate or incremented/decremented value to a0
```

```
assign a0_mux_output = ((opcode[0] == 1'b1) && (opcode[2] == 1'b0)) ?
ex_IR : new_a0_total;

//Generates the overall controller reset signal:
assign control_reset = rst || instr_reset;

//Used to generate the controller's reset signal from the reset instruction:
always @ (posedge clk) begin
  if (rst == 1'b1)
instr_reset <= 1'b0;
  else if ((pres_state == `execute_instr) && (opcode == 8'h00) &&
  (exec_prog == 1'b0))
instr_reset <= 1'b1;
  else
instr_reset <= 1'b0;
end

//Used to add/subtract the a0 operand to the address register
always @ (opcode or a0_reg or a0_operand) begin
  if (opcode[4] == 1'b1)
    new_a0_total = a0_reg + a0_operand;
  else
    new_a0_total = a0_reg - a0_operand;
end

//The address register
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_reg <= 32'h00000000;
  else if (write_a0 == 1'b1)
    a0_reg <= a0_mux_output;
  else
    a0_reg <= a0_reg;
end

//Used to store the initial NDW value
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    a0_operand <= 32'h00000000;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    a0_operand <= ex_IR[0:23];
  else
    a0_operand <= a0_operand;
end

//Used to write the new value to the address register
always @ (control_reset or change_a0 or Addr_req_W) begin
  if (control_reset == 1'b1)
     write_a0 = 1'b0;
  else if ( (change_a0 == 1'b1) &&
            (Addr_req_W == 1'b0) )
     write_a0 = 1'b1;
  else
     write_a0 = 1'b0;
end

//Used to indicate that the value of the address register will change
always @ (posedge clk) begin
  if (control_reset == 1'b1)
     change_a0 <= 1'b0;
  else if (write_a0 == 1'b1)
     change_a0 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[3:0] == 4'b1000) ||
  //autoincrement instructions
  ((opcode[4:2] == 3'b111) && ((opcode[1]==1'b1) || (opcode[0]==1'b1)))
  //write to the Adress register
  || (opcode[4:0] == 5'b01011)))
     change_a0 <= 1'b1;
  else
     change_a0 <= change_a0;
end

//Used to indicate when to negate the control bit when executing a
//bypass instruction
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    tx_bypass <= 1'b0;
  else if((((exec_prog == 1'b0)  && (pres_state == `decode_instr)) ||
  ((exec_prog == 1'b1) && (IR_req_W0 == 1'b0) &&
  (pres_state == `execute_instr))) && (opcode == `bypass))
    tx_bypass <= 1'b1;
  else if (((exec_prog == 1'b0)  ||
   ((exec_prog == 1'b1) && (IR_req_W0 == 1'b1)
   && (pres_state == `execute_instr) && (opcode != `bypass)))
   && (fifo_write == 1'b1))
    tx_bypass <= 1'b0;
  else
    tx_bypass <= tx_bypass;
end

//Used to indicate when to write an instruction from the receive link
//or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W0 <= 1'b0;
//Set IR_req_W0 when you are in the decode stage and the instruction should
//be sent (examples of NOT sent instructions include: write_a0,
//add_a0, sub_a0
  else if ((pres_state == `decode_instr) && (((exec_prog == 1'b1) &&
  ((opcode[3:0] != 4'b1000) && (opcode[3:0] != 4'b1011)))
  || (opcode[1:0] == 2'b10)))
    IR_req_W0 <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b00))
    IR_req_W0 <= 1'b0;
  else
    IR_req_W0 <= IR_req_W0;
end

// Used to indicate when to write immediate values from the receive
// link or program
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    IR_req_W1 <= 1'b0;
  else if ((pres_state == `decode_instr) && ((opcode[2] == 1'b1) &&
  (((exec_prog == 1'b1) && ((opcode[0] == 1'b1) || (opcode[1]==1'b1))
  && (opcode[3] == 1'b0)) //Not an Areg operation
  || ((exec_prog == 1'b0) && ((opcode == `bypass) ||
  (opcode[1:0] == 2'b10))))))
    IR_req_W1 <= 1'b1;
  else if (((opcode != `bypass) ||
  ((opcode == `bypass) && (data_cntr == 24'h000001))) &&
  (fifo_write == 1'b1) && (mux_select_fifo_in == 2'b01))
    IR_req_W1 <= 1'b0;
  else
    IR_req_W1 <= IR_req_W1;
end

//Controls the request to write an address register to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    Addr_req_W <= 1'b0;
  else if ((pres_state == `decode_instr) && (opcode[3:2] == 2'b11) &&
   (opcode[1:0] != 2'b00))
    Addr_req_W <= 1'b1;
  else if ((fifo_write == 1'b1) && (mux_select_fifo_in == 2'b10))
    Addr_req_W <= 1'b0;
  else
    Addr_req_W <= Addr_req_W;
end

//Controls the request to write data to the fifo
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_req_W <= 1'b0;
  else if ((pres_state == `decode_instr) &&
(((exec_prog == 1'b1) && (opcode[0] == 1'b1)) ||
       ((exec_prog == 1'b0) && (opcode[1:0] == 2'b10))) &&
       (opcode[3:2] != 2'b10))
    data_req_W <= 1'b1;
  else if ((data_cntr == 24'h000001) && (fifo_write == 1'b1)
   //needed for when only 1 data word is sent
  && (mux_select_fifo_in == 2'b11))
    data_req_W <= 1'b0;
  else
    data_req_W <= data_req_W;
end

//Loads the number of data values into the counter
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    data_cntr <= 24'h000000;
  else if ((pres_state == `decode_instr) &&
(((exec_prog == 1'b1) && (opcode[0] == 1'b1)) ||
       ((exec_prog == 1'b0) && ((opcode[1:0] == 2'b10) ||
        (opcode == `bypass)))) && (opcode[3:2] != 2'b10))
    data_cntr <= a0_operand;
  else if (((mux_select_fifo_in == 2'b11) ||
  ((opcode == `bypass) && (mux_select_fifo_in == 2'b01)))
  && (fifo_write == 1'b1))
    data_cntr <= data_cntr - 1;
  else
    data_cntr <= data_cntr;
end

// Generate the select signal to the mux feeding the internal buffer:
always @ (pres_state or IR_req_W0 or IR_req_W1 or Addr_req_W) begin
if ((pres_state == `execute_instr) && (IR_req_W0 == 1'b1))
      mux_select_fifo_in = 2'b00;
else if ((pres_state == `execute_instr) && (IR_req_W1 == 1'b1))
      mux_select_fifo_in = 2'b01;
else if ((pres_state == `execute_instr) && (Addr_req_W == 1'b1))
      mux_select_fifo_in = 2'b10;
else
      mux_select_fifo_in = 2'b11;
end

// Used to update opcode with the next instruction to be executed:
always @ (posedge clk) begin
  if (control_reset == 1'b1)
    opcode <= 8'b0;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    opcode <= ex_IR[24:31];
  else
    opcode <= opcode;
end

// Used to update ex_IR with the next instruction to be executed:
always @ (posedge clk) begin // or posedge control_reset) begin
  if (control_reset == 1'b1)
    exec_prog <= 1'b1;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b1))
    exec_prog <= 1'b0;
  else if ((ex_instr_read == 1'b1) && (sel_ex_IR_input == 1'b0))
    exec_prog <= 1'b1;
  else
    exec_prog <= exec_prog;
end

// Mux the input into the internal buffer:
always @ (mux_select_fifo_in or ex_IR or ex_IR_fifo_in or a0_reg or a0_operand
    or pe_data_out) begin
  case (mux_select_fifo_in)
    2'b00:
      fifo_mux_out = {a0_operand,ex_IR_fifo_in};
    2'b01:
      fifo_mux_out = ex_IR;
    2'b10:
      fifo_mux_out = a0_reg;
    2'b11:
      fifo_mux_out = pe_data_out;
  endcase
end

always @ (posedge clk) begin
  if (control_reset == 1'b1)
    pres_state <= `get_next_instr;
  else
    pres_state <= next_state;
end

// Determine next state
always @ (pres_state or ex_instr_read or IR_req_W0 or IR_req_W1 or Addr_req_W
    or change_a0 or data_req_W or error0 or error1 or error2 or error3 or
    error4 or error5 or error6 or error7 or error8 or int_int_error or
    int_chk_status or status_cntr or cont_execution) begin
  case (pres_state)
    `get_next_instr:
      begin
        if ((error1 == 1'b1) || (error3 == 1'b1) || (error5 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else if ((ex_instr_read == 1'b1))
         next_state = `decode_instr;
        else
         next_state = `get_next_instr;
      end
    `decode_instr:
        if ((error0 == 1'b1) || (error5 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
else
        next_state = `execute_instr;
    `execute_instr:
      begin
        if ((error0 == 1'b1) || (error2 == 1'b1) || (error4 == 1'b1) || //Debug
            (error5 == 1'b1) || (error6 == 1'b1) || (error7 == 1'b1) || //Debug
            (error8 == 1'b1)) //Debug
         next_state = `status_chk; //Debug
        else if ((int_int_error == 1'b1) || ((int_chk_status == 1'b1) && //Debug
((IR_req_W0 == 1'b0) && (IR_req_W1 == 1'b0) &&  //Debug
   (Addr_req_W == 1'b0) && (change_a0 == 1'b0) &&  //Debug
   (data_req_W == 1'b0)))) //Debug
  next_state = `status_chk; //Debug
  //Continue executing until the instruction/address has been
  //completed- that data can be written on its own.
else if ((IR_req_W0 == 1'b1) || (IR_req_W1 == 1'b1) ||
   (Addr_req_W == 1'b1) || (change_a0 == 1'b1) ||
   (data_req_W == 1'b1))
  next_state = `execute_instr;
else
  next_state = `get_next_instr;
      end
    `status_chk: //Debug ****
      begin
```

```
//if ((((status_cntr == ‘num_regs-2) && (int_debug_link_write == 1’b1))
// || (status_cntr == ‘num_regs-1))
if ((status_cntr == ‘num_regs-1) &&
(cont_execution == 1’b1) && (int_int_error == 1’b0)) //Debug
next_state = ‘get_next_instr;
else
next_state = ‘status_chk;
      end
  endcase
end

always @ (mux_select_fifo_in or IR_req_W0 or IR_req_W1 or Addr_req_W or
    data_req_W or fifo_full or valid_data )begin
  case(mux_select_fifo_in)
     2’b00:
       begin
         if ((IR_req_W0 == 1’b1) && (fifo_full == 1’b0) &&
 //Instr does not tx data
 ((data_req_W == 1’b0) ||
//Instr does tx data- wait for data to be available before
//sending instruction
((data_req_W == 1’b1) && (valid_data == 1’b1))))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b01:
       begin
         if ((IR_req_W1 == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b10:
       begin
         if ((Addr_req_W == 1’b1) && (fifo_full == 1’b0))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
     2’b11:
       begin
         if ((data_req_W == 1’b1) && (fifo_full == 1’b0) &&
 (valid_data == 1’b1))
            fifo_write = 1’b1;
 else
            fifo_write = 1’b0;
       end
  endcase
end

assign read_data = ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b11));

//Debug register download order:
//0)int_debug_link_data <= err_type_reg;
//1)int_debug_link_data <= ex_IR_bak;
//2)int_debug_link_data <= imm_addr;
//3)int_debug_link_data <= a0_reg_bak;
//4)int_debug_link_data <= data_cntr_bak;
//5)int_debug_link_data <= prog_IR_bak;
//6)int_debug_link_data <= rx_IR_bak;
//7)int_debug_link_data <= controller_status_reg;
//8)int_debug_link_data <= prog_PE_status_reg;
//8)int_debug_link_data <= exec_time_cntr;

assign debug_link_clk = clk; //Debug
assign debug_link_data = err_type_reg; //Debug
assign debug_link_control = 1’b0; //Debug
assign int_error = int_int_error; //Debug
assign CE_error = (error0 || error1 || error2 || error3 || error4 || error5 ||
error6 || error7 || error8);

//Debug Errors:
assign error0 = ((((prog_cbit == 1'b1) && (reading_prog_IR == 1'b1)) ||
 ((rx_cbit == 1'b1) && (reading_rx_IR == 1'b1))) &&
 (pres_state != `get_next_instr));
assign error1 = ((pres_state == `get_next_instr) &&
(((rx_IR_full == 1'b1) && (rx_cbit == 1'b0)) ||
((prog_IR_full == 1'b1) && (prog_cbit == 1'b0))));
assign error2 = ((pres_state == `execute_instr) &&
(exec_time_cntr[`bit] == 1'b1));
assign error3 = ((pres_state == `get_next_instr) &&
(exec_time_cntr[`bit] == 1'b1));
assign error4 = ((FSL_M_Write == 1'b1) && (FSL_M_Full == 1'b1));
assign error5 = ((FSL_S_Read == 1'b1) && (FSL_S_Exists == 1'b0));
//assign error6 = ((pe_write_data == 1'b1) && (pe_can_write_data == 1'b0));
assign error6 = 1'b0;
//assign error7 = ((pe_write_addr == 1'b1) && (pe_can_write_addr == 1'b0));
assign error7 = 1'b0;
assign error8 = ((pe_read_data == 1'b1) && (pe_can_read == 1'b0)); //producer

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_int_error <= 1'b0;
  else if (CE_error == 1'b1)
    int_int_error <= 1'b1;
  else
    int_int_error <= int_int_error;
end

//Debug
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_state <= `get_next_instr;
  else
    prev_state <= pres_state;
end

//Debug:
//Status info is ready when all the registers have been downloaded to the
//debug link fifo
assign status_rdy = (status_cntr == `num_regs-1) ? 1'b1 : 1'b0; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prev_status_rdy <= 1'b0;
  else
    prev_status_rdy <= status_rdy;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    int_chk_status <= 1'b0;
  else if (chk_status == 1'b1)
    int_chk_status <= 1'b1;
  else if (status_rdy == 1'b1)
    int_chk_status <= 1'b0;
  else
    int_chk_status <= int_chk_status;
end

//Debug:
always @ (posedge clk) begin
```

```
  if (rst == 1'b1)
    int_debug_link_write <= 1'b0;
  else if ((pres_state == `status_chk) && (debug_link_full == 1'b0) &&
   (status_rdy == 1'b0))
    int_debug_link_write <= 1'b1;
  else
    int_debug_link_write <= 1'b0;
end

assign debug_link_write = int_debug_link_write; //Debug

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    status_cntr <= 4'h0;
  else if ((prev_state == `status_chk) && (pres_state == `get_next_instr))
    status_cntr <= 4'h0;
  else if ((pres_state == `status_chk) && (int_debug_link_write == 1'b1))
    status_cntr <= status_cntr + 1;
  else
    status_cntr <= status_cntr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    err_type_reg <= 32'h00000000;
  else if (int_int_error == 1'b0)
    begin
      err_type_reg[0] <= error0;
      err_type_reg[1] <= error1;
      err_type_reg[2] <= error2;
      err_type_reg[3] <= error3;
      err_type_reg[4] <= error4;
      err_type_reg[5] <= error5;
      err_type_reg[6] <= error6;
      err_type_reg[7] <= error7;
      err_type_reg[8] <= error8;
      err_type_reg[9:30] <= 22'h000000;
      err_type_reg[31] <= CE_error;
    end
  else if (int_debug_link_write == 1'b1)
    err_type_reg <= ex_IR_bak;
  else
    err_type_reg <= err_type_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    ex_IR_bak <= 32'h00000000;
  else if (((pres_state == `status_chk) || (pres_state == `execute_instr)) &&
  (next_state == `get_next_instr))
    ex_IR_bak <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    ex_IR_bak <= imm_addr;
  else if ((ex_instr_read == 1'b1) && (pres_state == `get_next_instr))
    ex_IR_bak <= ex_IR;
  else
    ex_IR_bak <= ex_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    imm_addr <= 32'h00000000;
  else if (pres_state == `get_next_instr)
    imm_addr <= 32'h00000000;
  else if (debug_link_write == 1’b1)
    imm_addr <= a0_reg_bak;
  else if ((fifo_write == 1’b1) && (mux_select_fifo_in == 2’b01) &&
   (opcode != ‘bypass))
    imm_addr <= ex_IR;
  else
    imm_addr <= imm_addr;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1’b1)
    a0_reg_bak <= 32’h00000000;
  else if (pres_state == ‘get_next_instr)
    a0_reg_bak <= a0_reg;
  else if (int_debug_link_write == 1’b1)
    a0_reg_bak <= data_cntr_bak;
  else if (write_a0 == 1’b1)
    a0_reg_bak <= a0_mux_output;
  else
    a0_reg_bak <= a0_reg_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1’b1)
    data_cntr_bak <= 32’h00000000;
  else if (pres_state == ‘get_next_instr)
    data_cntr_bak <= 32’h00000000;
  else if (int_debug_link_write == 1’b1)
    data_cntr_bak <= prog_IR_bak;
  else if (pres_state == ‘decode_instr)
    data_cntr_bak <= {8’h00,a0_operand};
  else if (((mux_select_fifo_in == 2’b11) ||
  ((opcode == ‘bypass) && (mux_select_fifo_in == 2’b01)))
  && (fifo_write == 1’b1))
    data_cntr_bak <= data_cntr_bak - 1;
  else
    data_cntr_bak <= data_cntr_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1’b1)
    prog_IR_bak <= 32’h00000000;
  else if (int_debug_link_write == 1’b1)
    prog_IR_bak <= rx_IR_bak;
  else if (pres_state != ‘status_chk)
    prog_IR_bak <= prog_IR;
  else
    prog_IR_bak <= prog_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1’b1)
    rx_IR_bak <= 32’h00000000;
  else if (int_debug_link_write == 1’b1)
    rx_IR_bak <= controller_status_reg;
  else if (pres_state != ‘status_chk)
    rx_IR_bak <= rx_IR;
  else
    rx_IR_bak <= rx_IR_bak;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1’b1)
```

```
    controller_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    controller_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    controller_status_reg <= prog_PE_status_reg;
  else if (((pres_state == `get_next_instr) || (pres_state == `decode_instr))
   || ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      controller_status_reg[0] <= ex_IR_cbit;
      controller_status_reg[1] <= IR_req_W0;
      controller_status_reg[2] <= IR_req_W1;
      controller_status_reg[3] <= Addr_req_W;
      controller_status_reg[4] <= data_req_W;
      controller_status_reg[5] <= tx_bypass;
      controller_status_reg[6] <= exec_prog;
      //controller_status_reg[7] <= rx_addr_req;
      //controller_status_reg[8] <= rx_data_req;
      controller_status_reg[7] <= 1'b0;
      controller_status_reg[8] <= 1'b0;
      controller_status_reg[9] <= change_a0;
      controller_status_reg[10:15] <= 6'h00;
      controller_status_reg[16] <= ex_IR_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      controller_status_reg[20] <= data_req_W;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      //controller_status_reg[23] <= rx_addr_req;
      //controller_status_reg[24] <= rx_data_req;
      controller_status_reg[23] <= 1'b0;
      controller_status_reg[24] <= 1'b0;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else if (pres_state == `execute_instr)
    begin
      controller_status_reg[0:15] <= controller_status_reg[0:15];
      controller_status_reg[16] <= ex_IR_cbit;
      controller_status_reg[17] <= IR_req_W0;
      controller_status_reg[18] <= IR_req_W1;
      controller_status_reg[19] <= Addr_req_W;
      controller_status_reg[20] <= data_req_W;
      controller_status_reg[21] <= tx_bypass;
      controller_status_reg[22] <= exec_prog;
      //controller_status_reg[23] <= rx_addr_req;
      //controller_status_reg[24] <= rx_data_req;
      controller_status_reg[23] <= 1'b0;
      controller_status_reg[24] <= 1'b0;
      controller_status_reg[25] <= change_a0;
      controller_status_reg[26:31] <= 6'h00;
    end
  else
    controller_status_reg <= controller_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    prog_PE_status_reg <= 32'h00000000;
  else if ((status_rdy == 1'b1) && (prev_status_rdy == 1'b0))
    prog_PE_status_reg <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    prog_PE_status_reg <= exec_time_cntr;
  else if ((pres_state == `get_next_instr) || (pres_state == `decode_instr) ||
  ((pres_state == `execute_instr) && (prev_state == `decode_instr)))
    begin
      prog_PE_status_reg[0] <= rx_IR_full;
      prog_PE_status_reg[1] <= prog_IR_full;
      prog_PE_status_reg[2] <= rx_cbit;
      prog_PE_status_reg[3] <= prog_cbit;
      prog_PE_status_reg[4] <= valid_instr;
      prog_PE_status_reg[5] <= cont_prog;
      prog_PE_status_reg[6] <= pe_can_read;
      //prog_PE_status_reg[7] <= pe_can_write_data;
      //prog_PE_status_reg[8] <= pe_can_write_addr;
      prog_PE_status_reg[7] <= 1'b0;
      prog_PE_status_reg[8] <= 1'b0;
      prog_PE_status_reg[9:10] <= pres_state;
      prog_PE_status_reg[11:12] <= mux_select_fifo_in;
      prog_PE_status_reg[13:15] <= 3'h0;
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      prog_PE_status_reg[22] <= pe_can_read;
      //prog_PE_status_reg[23] <= pe_can_write_data;
      //prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[23] <= 1'b0;
      prog_PE_status_reg[24] <= 1'b0;
      prog_PE_status_reg[25:26] <= pres_state;
      prog_PE_status_reg[27:28] <= mux_select_fifo_in;
      prog_PE_status_reg[29:31] <= 3'h0;
    end
  else if (pres_state == `execute_instr)
    begin
      prog_PE_status_reg[0:15] <= prog_PE_status_reg[0:15];
      prog_PE_status_reg[16] <= rx_IR_full;
      prog_PE_status_reg[17] <= prog_IR_full;
      prog_PE_status_reg[18] <= rx_cbit;
      prog_PE_status_reg[19] <= prog_cbit;
      prog_PE_status_reg[20] <= valid_instr;
      prog_PE_status_reg[21] <= cont_prog;
      prog_PE_status_reg[22] <= pe_can_read;
      //prog_PE_status_reg[23] <= pe_can_write_data;
      //prog_PE_status_reg[24] <= pe_can_write_addr;
      prog_PE_status_reg[23] <= 1'b0;
      prog_PE_status_reg[24] <= 1'b0;
      prog_PE_status_reg[25:26] <= pres_state;
      prog_PE_status_reg[27:28] <= mux_select_fifo_in;
      prog_PE_status_reg[29:31] <= 3'h0;
    end
  else
    prog_PE_status_reg <= prog_PE_status_reg;
end

//Debug:
always @ (posedge clk) begin
  if (rst == 1'b1)
    exec_time_cntr <= 32'h00000000;
  else if (((status_rdy == 1'b1) && (prev_status_rdy == 1'b0)) ||
  ((pres_state == `get_next_instr) && (prev_state == `execute_instr))
  || (pres_state == `decode_instr))
    exec_time_cntr <= 32'h00000000;
  else if (debug_link_write == 1'b1)
    exec_time_cntr <= 32'hdeaddead; //END OF PIPELINED DEBUG DATA
  else if((pres_state == `execute_instr) && (pres_state == `get_next_instr))
    exec_time_cntr <= exec_time_cntr + 1;
  else
    exec_time_cntr <= exec_time_cntr;
end

endmodule
```

Appendix B

Input File Format for the System Generator

```
/* Filename: gen_ublaze_system_input_format.c
 * Author:   Lesley Shannon
 * Date:     July 28, 2005
 * Description:
 *   This file describe the format for the input file used
 *   to describe the system to be generated by the system
 *   generator.  To provide the information for the input
 *   file, draw a DAG and label the nodes (CEs) from 0 to
 *   NumNodes-1 and label the edges (Links) from 0 to
 *   NumLinks-1.
 *********************************************************/

NumNodes
NumLinks
NumClocks
UseMDM?
ExtResetHigh?

Node0 Software? ClockID NumInputs NumOutputs [OPB Peripheral?: if SW]/
  [FSL Switch?: if HW] [ExtMem?: if SW]/[Associated Proc: if Switch]
*Input(s)
*Output(s)

Node1 Software? ClockID NumInputs NumOutputs [OPB Peripheral?: if SW]/
  [FSL Switch?: if HW] [ExtMem?: if SW]/[Associated Proc: if Switch]
*Input(s)
*Output(s)

...

Node(NumNodes-1) Software? ClockID? NumInputs NumOutputs
  [OPB Peripheral?: if SW]/[Switch?: if HW] [ExtMem?: if SW]/
  [Associated Proc: if Switch]
*Input(s)
*Output(s)


Link0 FSL? [default parameters?: if FSL]/[Master?: if wire]
SourceType SourceID SinkType SinkID
[Clk: if FSL using default parameters, a synchronous clock]
[Asynch? use_control?
MClk SClk
width depth: if an FSL not using default parameters]

Link1 FSL? [default parameters?: if FSL]/[Master?: if wire]
SourceType SourceID SinkType SinkID
```

```
[Clk: if FSL using default parameters, a synchronous clock]
[Asynch? use_control?
MClk SClk
width depth: if an FSL not using default parameters]

...

Link(NumLinks-1) FSL? [default parameters?: if FSL]/
  [Master?: if wire]
SourceType SourceID SinkType SinkID
[Clk: if FSL using default parameters, a synchronous clock]
[Asynch? use_control?
MClk SClk
width depth: if an FSL not using default parameters]

************************************************
Notes:
*Nodes may be either Processors (software) or dedicated
hardware modules (hardware) or switches (hardware,
but designed to act as an interface between fsls
from a processor and the wires in a network)
*Links may be either FSLs or wires connecting to FSLs (the
user must specify whether they connect to the Master
or Slave side of an FSL) or FSLs via switches
*Wires connect to either nodes or FSLs, but cannot connect
together two FSLs;
*FSLS connect to either wires or nodes;
*SourceType/SinkType: 0 = CE; 1 = FSL; 2 = wire;
```

Appendix C

On-chip Testbed Source and Sink Packet Interpreters

C.1 Transmitter Testbed

```
/****************************************************************************/
/* simppl_test.c                                                            */
/* =============                                                            */
/*                                                                          */
/* 05/18/2005                                                               */
/* apatel                                                                   */
/*                                                                          */
/* Test application skeleton for validating SIMPPL Systems.                 */
/*                                                                          */
/****************************************************************************/

#include "xparameters.h"
#include "xutil.h"
#include "mb_interface.h"

typedef unsigned char u8;
typedef unsigned int u32;

#define printf xil_printf

#define GET_NDW(x)  (((x) >> 8) & 0xFFFFFF)
#define GET_OP(x)   ((x) & 0xFF)

#define DEBUG 1
// Opcode Definitions - Taken from instr_defines.v

#define OP_RESET                0x00
#define OP_NOOP                 0x04
#define OP_BYPASS               0x0C

#define OP_READA_IMM            0x01
#define OP_READR_IMM            0x02
#define OP_WRITE_IMM            0x03

#define OP_READA_ABS_ADDR       0x05
#define OP_READR_ABS_ADDR       0x06  // These first 9 instructions will be
#define OP_WRITE_ABS_ADDR       0x07  // read and transmitted.

//***These instructions are NOT transmitted***//
#define OP_READ_AREG            0x0A //NOT IMPLEMENTED YET!!!
#define OP_WRITE_AREG           0x0B

#define OP_ADD_IMM_AREG         0x18
```

```
#define OP_SUB_IMM_AREG         0x08

#define OP_READA_AREG_INDIRECT  0x0D
#define OP_READR_AREG_INDIRECT  0x0E
#define OP_WRITE_AREG_INDIRECT  0x0F

#define OP_READA_AREG_AUTOINC   0x1D
#define OP_READR_AREG_AUTOINC   0x1E
#define OP_WRITE_AREG_AUTOINC   0x1F

//#define DEBUG

u32 test_case_1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
u32 address_1 = 0x00002200;

u32 test_case_2[6] = {0, 1 , 2, 3, 4, 5 };
u32 address_2 = 0x00001100;

u32 test_case_3[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
u32 address_3 = 0x00005500;

u32 test_case_4[2] = {0, 1};
u32 address_4 = 0x0000ee00;

u32 test_case_5[35] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
 25, 26, 27, 28, 29, 30, 31, 32, 33, 34};
u32 address_5 = 0x0000ab00;

u32 test_case_typ[64]={ 75,20,9,0,0,0,0,0,
                        21,12,0,0,0,0,0,0,
                        8,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0
                     };
u32 macrotype;
u32 scalecode;
u32 address_typ;

void reset();
void noop();
void bypass(u32, u32, u32, u32, u32, u32*);
//0: ndw; 1: # of bypasses; 2: opcode of bypassed instruction (sub_opcode);
//3: sub_ndw; 4: Sub-Address field (if necessary), otherwise -1
//5: Pointer to Payload

void reada_imm(u32, u32*); //NDW; Pointer to Payload
void readr_imm(u32);       //NDW
void write_imm(u32, u32*); //NDW; Pointer to Payload

void reada_abs_addr(u32, u32, u32*);  //NDW; Address; Pointer to Payload
void readr_abs_addr(u32, u32);  //NDW; Address; Pointer to Payload
void write_abs_addr(u32, u32, u32*);  //NDW; Address; Pointer to Payload

//***THESE INSTRUCTIONS ARE NEVER TRANSMITTED TO A CONTROLLER***

//void read_areg();
//void write_areg();
//
//void add_imm_areg(u32);
//void sub_imm_areg(u32);
//
//void reada_areg_indirect();
//void readr_areg_indirect();
//void write_areg_indirect();
```

```
//
//void reada_areg_autoinc();
//void readr_areg_autoinc();
//void write_areg_autoinc();

void simppl_packet_error();
void simppl_rx_packet(u32);

u32 g_addr;
u32* fsl_data;

int main (void)
{

    u32 i, rx_data;

#ifdef DEBUG
    printf("Testing TX 123...\r\n");
#endif

    //A sample transmitted program:
      reset();
      noop();
      bypass(2, 2, OP_NOOP, 0, -1, 0x0);

      reada_imm(10, test_case_1);
      readr_imm(64);
      write_imm(6, test_case_2);

      reada_abs_addr(9, address_3, test_case_3);
      readr_abs_addr(96, 0xDEADBEEF);
      write_abs_addr(2, address_4, test_case_4);

      bypass(1, 1, OP_READR_IMM, 16, -1, 0x0);
      bypass(2, 1, OP_READR_ABS_ADDR, 48, 0x00001122, 0x0);
      bypass(35, 2, OP_WRITE_ABS_ADDR, 32, 0x0000aabb, test_case_5);
      bypass(5, 4, OP_WRITE_ABS_ADDR, 0, 0xdeadbeef, 0x0);

#ifdef DEBUG
      printf("\r\nDone!\r\n");
      for (;;)
      {
      }
#endif
   return 0;
}

void reset()
{
    u32 ndw, opcode, control_word;

    opcode = OP_RESET;
    ndw = 0;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Reset Instruction Transmitted **\r\n");
#endif
    //g_addr = 0;
}

void noop()
{
    u32 ndw, opcode, control_word;

    opcode = OP_NOOP;
    ndw = 0;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** No-Op Instruction Transmitted **\r\n");
#endif
}

//0: NDW; 1: Pointer to Payload
void reada_imm(u32 ndw, u32 *fsl_data)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_READA_IMM;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Read ACK of Imm Data Transmitted **\r\n");
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif

    for (i = 0; i < ndw; i++)
    {
        microblaze_bwrite_datafsl(fsl_data[i], 0);
#ifdef DEBUG
        printf(" 0x%08x [%d]\r\n", fsl_data[i], i);
#endif
    }
}

//0: NDW
void readr_imm(u32 ndw)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_READR_IMM;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Read Request for Imm Data Transmitted **\r\n");
#endif
}

//0: NDW; 1: Pointer to Payload
void write_imm(u32 ndw, u32 *fsl_data)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_WRITE_IMM;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Write Imm Data Transmitted **\r\n");
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif

    for (i = 0; i < ndw; i++)
    {
        microblaze_bwrite_datafsl(fsl_data[i], 0);
#ifdef DEBUG
        printf(" 0x%08x [%d]\r\n", fsl_data[i], i);
#endif
```

```
    }
}

//0: NDW; 1: Address; 2: Pointer to Payload
void reada_abs_addr(u32 ndw, u32 address, u32 *fsl_data)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_READA_ABS_ADDR;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Read ACK of Data from Abs Addr Transmitted **\r\n");
#endif

    microblaze_bwrite_datafsl(address, 0);
#ifdef DEBUG
    printf(" Address = 0x%08x\r\n", address);
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif

    for (i = 0; i < ndw; i++)
    {
        microblaze_bwrite_datafsl(fsl_data[i], 0);
#ifdef DEBUG
        printf(" 0x%08x [%d]\r\n", fsl_data[i], i);
#endif
    }
}

//0: NDW; 1: Address
void readr_abs_addr(u32 ndw, u32 address)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_READR_ABS_ADDR;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Read Request for data from Abs Addr Transmitted **\r\n");
#endif

    microblaze_bwrite_datafsl(address, 0);
#ifdef DEBUG
    printf(" Address = 0x%08x\r\n", address);
#endif DEBUG

}

//0: NDW; 1: Address; 2: Pointer to Payload
void write_abs_addr(u32 ndw, u32 address, u32 *fsl_data)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_WRITE_ABS_ADDR;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Write Data to Abs Addr Instruction Transmitted **\r\n");
#endif

    microblaze_bwrite_datafsl(address, 0);
```

```
#ifdef DEBUG
    printf(" Address = 0x%08x\r\n", address);
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif

    for (i = 0; i < ndw; i++)
    {
        microblaze_bwrite_datafsl(fsl_data[i], 0);
#ifdef DEBUG
        printf(" 0x%08x [%d]\r\n", fsl_data[i], i);
#endif
    }
}

//0: ndw; 1: # of bypasses; 2: opcode of bypassed instruction (sub_opcode);
//3: sub_ndw; 4: Sub-Address field (if necessary), otherwise -1
//5: Pointer to Payload
void bypass(u32 ndw, u32 num_bypasses, u32 sub_opcode, u32 sub_ndw,
        u32 sub_address, u32 *fsl_data)
{
    u32 opcode, control_word;
    int i;

    opcode = OP_BYPASS;
    control_word = (ndw<<8) + opcode;

    microblaze_bwrite_cntlfsl(control_word, 0);
#ifdef DEBUG
    printf("\r\n** Bypass Instruction Transmitted **\r\n");
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif
    ndw--;

    for (i=1; i<num_bypasses; i++, ndw--)
    {
    control_word = (ndw<<8) + opcode;
    microblaze_bwrite_datafsl(control_word, 0);
#ifdef DEBUG
    printf(" Bypass Recursion Count = %d\r\n", i);
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif
    }

    control_word = (sub_ndw<<8) + sub_opcode;
    microblaze_bwrite_datafsl(control_word, 0);
#ifdef DEBUG
    printf(" Bypassed Instruction = 0x%08x\r\n", control_word);
#endif

    if ((sub_opcode == OP_READR_ABS_ADDR) || (sub_opcode == OP_WRITE_ABS_ADDR)
    || (sub_opcode == OP_READA_ABS_ADDR))
    {
    microblaze_bwrite_datafsl(sub_address, 0);

         if (ndw != 0)
    ndw--;
#ifdef DEBUG
    printf(" Bypassed Instruction Address = 0x%08x\r\n", sub_address);
#endif
    }
#ifdef DEBUG
    printf("** Bypassed Data **\r\n");
#endif

    for (i = 0; i < ndw; i++)
    {
        microblaze_bwrite_datafsl(fsl_data[i], 0);
#ifdef DEBUG
```

```
        printf(" 0x%08x [%d]\r\n", fsl_data[i], i);
#endif
    }
#ifdef DEBUG
    printf("** End Bypass Instruction **\r\n");
#endif
}

/*
void read_areg()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void write_areg()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void add_imm_areg(u32 ndw)
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void sub_imm_areg(u32 ndw)
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void reada_areg_indirect()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void readr_areg_indirect()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void write_areg_indirect()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void reada_areg_autoinc()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void readr_areg_autoinc()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}

void write_areg_autoinc()
{
    printf("\r\n** Not a Transmittable instruction **\r\n");
}
*/
```

C.2 Receiver Testbed

```
/***************************************************************************/
/* simppl_test.c                                                           */
/* =============                                                           */
/*                                                                         */
/* 05/18/2005                                                              */
/* apatel                                                                  */
/*                                                                         */
/* Test application skeleton for validating SIMPPL Systems.                */
/*                                                                         */
/***************************************************************************/

#include "xparameters.h"
#include "xutil.h"
#include "mb_interface.h"

typedef unsigned char u8;
typedef unsigned int u32;

#define printf xil_printf

#define GET_NDW(x)  (((x) >> 8) & 0xFFFFFF)
#define GET_OP(x)   ((x) & 0xFF)

// Opcode Definitions - Taken from instr_defines.v

#define OP_RESET                0x00
#define OP_NOOP                 0x04

#define OP_READA_IMM            0x01
#define OP_READR_IMM            0x02
#define OP_WRITE_IMM            0x03

#define OP_READA_ABS_ADDR       0x05
#define OP_READR_ABS_ADDR       0x06  // These first 8 instructions will be
#define OP_WRITE_ABS_ADDR       0x07  // read and transmitted.

#define OP_READ_AREG            0x0A //NOT IMPLEMENTED YET!!!
#define OP_WRITE_AREG           0x0B

#define OP_ADD_IMM_AREG         0x18
#define OP_SUB_IMM_AREG         0x08
#define OP_BYPASS               0x0C

#define OP_READA_AREG_INDIRECT  0x0D
#define OP_READR_AREG_INDIRECT  0x0E
#define OP_WRITE_AREG_INDIRECT  0x0F

#define OP_READA_AREG_AUTOINC   0x1D
#define OP_READR_AREG_AUTOINC   0x1E
#define OP_WRITE_AREG_AUTOINC   0x1F

//#define DEBUG

//Test example:
//u32 test_case_1[16] = {};
//tx_write_abs_addr(address, test_case_1, 16);


void reset();
void noop();
void reada_imm(u32);
void readr_imm();
void write_imm(u32);

void reada_abs_addr(u32);
void readr_abs_addr();
void write_abs_addr(u32);

void read_areg();
void write_areg();

void add_imm_areg(u32);
void sub_imm_areg(u32);
void bypass(u32, u32);

void reada_areg_indirect();
void readr_areg_indirect();
void write_areg_indirect();

void reada_areg_autoinc();
void readr_areg_autoinc();
void write_areg_autoinc();

void simppl_packet_error();
void simppl_rx_packet(u32);

u32 g_addr;

int main (void)
{

    u32 i, j;

#ifdef DEBUG
    printf("\r\nTesting RX 123...\r\n");
#endif

#if 0
    for (i = 0; i < 1024; i++)
    {
        j = 0;
        microblaze_bread_datafsl(j, 0);
        printf("Read back fsl[%d] = %08x\r\n", i, j);
    }
#endif

    for (j = 0; j < 200; j ++)
    {
        simppl_rx_packet(0);
    }

    printf("Done!\r\n");
    for (;;)
    {
    }
}

void simppl_rx_packet(u32 recursion_count)
{
    u32 fsl_data;
    u32 ndw;
    u8  opcode;

    // Step 1 - Fetch the instruction word and decode it
    microblaze_bread_datafsl(fsl_data, 0);

    opcode = GET_OP(fsl_data);
    ndw = GET_NDW(fsl_data);

    switch (opcode)
    {
        case OP_RESET:
            reset();
        break;
```

```
case OP_NOOP:
    noop();
break;

case OP_READA_IMM:
    reada_imm(ndw);
break;

case OP_READR_IMM:
    readr_imm();
break;

case OP_WRITE_IMM:
    write_imm(ndw);
break;

case OP_READA_ABS_ADDR:
    reada_abs_addr(ndw);
break;

case OP_READR_ABS_ADDR:
    readr_abs_addr();
break;

case OP_WRITE_ABS_ADDR:
    write_abs_addr(ndw);
break;

case OP_READ_AREG:
    read_areg();
break;

case OP_WRITE_AREG:
    write_areg();
break;

case OP_ADD_IMM_AREG:
    add_imm_areg(ndw);
break;

case OP_SUB_IMM_AREG:
    sub_imm_areg(ndw);
break;

case OP_BYPASS:
    bypass(ndw, recursion_count);
break;

case OP_READA_AREG_INDIRECT:
    reada_areg_indirect();
break;

case OP_READR_AREG_INDIRECT:
    readr_areg_indirect();
break;

case OP_WRITE_AREG_INDIRECT:
    write_areg_indirect();
break;

case OP_READA_AREG_AUTOINC:
    reada_areg_autoinc();
break;

case OP_READR_AREG_AUTOINC:
    readr_areg_autoinc();
break;
```

```
        case OP_WRITE_AREG_AUTOINC:
            write_areg_autoinc();
        break;

        default:
            simppl_packet_error(fsl_data);
    }


}

void reset()
{
    printf("\r\n** Reset Instruction Received **\r\n");
    g_addr = 0;
}

void noop()
{
    printf("\r\n** No-Op Instruction Received **\r\n");
}

void reada_imm(u32 ndw)
{
    u32 fsl_data;
    int i;

    printf("\r\n** Read ACK + Imm Data **\r\n");
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);

    for (i = 0; i < ndw; i++)
    {
        microblaze_bread_datafsl(fsl_data, 0);
        printf(" 0x%08x [%d]\r\n", fsl_data, i);
    }
}

void readr_imm()
{
    printf("\r\n** Read Request Imm Data **\r\n");
}

void write_imm(u32 ndw)
{
    u32 fsl_data;
    int i;

    printf("\r\n** Write Imm Data **\r\n");
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);

    for (i = 0; i < ndw; i++)
    {
        microblaze_bread_datafsl(fsl_data, 0);
        printf(" 0x%08x [%d]\r\n", fsl_data, i);
    }
}

void reada_abs_addr(u32 ndw)
{
    u32 fsl_data;
    int i;

    printf("\r\n** Read ACK Data from Abs Addr **\r\n");

    microblaze_bread_datafsl(fsl_data, 0);
    printf(" Address = 0x%08x\r\n", fsl_data);
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);

    for (i = 0; i < ndw; i++)
    {
        microblaze_bread_datafsl(fsl_data, 0);
        printf(" 0x%08x [%d]\r\n", fsl_data, i);
    }
}

void readr_abs_addr()
{
    u32 fsl_data;
    int i;

    printf("\r\n** Read Request from Abs Addr **\r\n");

    microblaze_bread_datafsl(fsl_data, 0);
    printf(" Address = 0x%08x\r\n", fsl_data);
}


void write_abs_addr(u32 ndw)
{
    u32 fsl_data;
    int i;

#ifdef DEBUG
    printf("\r\n** Write Data to Abs Addr **\r\n");
#endif

    microblaze_bread_datafsl(fsl_data, 0);
#ifdef DEBUG
    printf(" Address = 0x%08x\r\n", fsl_data, i);

    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);
#endif
    for (i = 0; i < ndw; i++)
    {
        microblaze_bread_datafsl(fsl_data, 0);
#ifdef DEBUG
        printf(" %d [%d]\r\n", fsl_data, i);
#endif
    }
}

void read_areg()
{
    printf("\r\n** Read Addr Reg **\r\n");
    printf(" NOT IMPLEMENTED YET\r\n");
}

void write_areg()
{
    u32 fsl_data;

    printf("\r\n** Write Data to Addr Reg **\r\n");
    microblaze_bread_datafsl(fsl_data, 0);
    printf(" Address = 0x%08x\r\n", fsl_data);
    g_addr = fsl_data;
}


void add_imm_areg(u32 ndw)
{
    printf("\r\n** Add Imm to Addr Reg **\r\n");
    printf(" Immediate   = 0x%08x\n", ndw);
    g_addr = (g_addr + ndw) & 0x00ffffff;
    printf(" New Address = 0x%08x\n", g_addr);
```

```
}

void sub_imm_areg(u32 ndw)
{
    printf("\r\n** Subtract Imm from Addr Reg **\r\n");
    printf(" Immediate   = 0x%08x\n", ndw);
    g_addr = (g_addr - ndw) & 0x00ffffff;
    printf(" New Address = 0x%08x\n", g_addr);
}
void bypass(u32 ndw, u32 recursion_count)
{
    int i;
    u32 sub_opcode;

    printf("\r\n** Bypass Data **\r\n");
    printf(" Recursion Count = %d\r\n", recursion_count);
    printf(" Payload size = %d (0x%08x)\r\n", ndw, ndw);

    simppl_rx_packet(recursion_count+1);

    if (recursion_count == 0)
    {
        printf("** End Bypass Instruction **\r\n");
        return;
    }

}


void reada_areg_indirect()
{ printf("** reada_areg_indirect not implemented! **\r\n"); }
void readr_areg_indirect()
{ printf("** readr_areg_indirect not implemented! **\r\n"); }
void write_areg_indirect()
{ printf("** write_areg_indirect not implemented! **\r\n"); }

void reada_areg_autoinc()
{ printf("** reada_areg_autoinc not implemented! **\r\n"); }
void readr_areg_autoinc()
{ printf("** readr_areg_autoinc not implemented! **\r\n"); }
void write_areg_autoinc()
{ printf("** write_areg_autoinc not implemented! **\r\n"); }

void simppl_packet_error(u32 fsl_data)
{
    printf("Unrecognized Opcode: 0x%08x\n", fsl_data);
}
```

Appendix D

Execution Profile Data from SnoopP Experiments

Table D.1: mb-gprof Statistics on Functions comprising the Dhrystone Benchmark after One Hundred and One Million Passes.

Function Name	**SW Multiply & Divide**			**HW Multiply & SW Divide**			**HW Multiply & Divide**		
	100 Passes	**One Million Passes**		**100 Passes**	**One Million Passes**		**100 Passes**	**One Million Passes**	
	Percent Time (100Hz)	**Percent Time (6Hz)**	**Percent Time (100Hz)**	**Percent Time (100Hz)**	**Percent Time (6Hz)**	**Percent Time (100Hz)**	**Percent Time (100Hz)**	**Percent Time (6Hz)**	**Percent Time (100Hz)**
_start1	25.00	—	0.00	30.00	—	—	25.00	—	0.00
__mulsi3	16.67	33.33	10.20	—	—	—	—	—	—
Func_2	8.33	33.20	5.38	10.00	7.82	0.00	0.00	4.00	6.50
Proc_1	8.33	0.00	7.17	0.00	7.91	9.09	0.00	15.97	13.63
Proc_3	8.33	0.00	4.55	0.00	4.50	5.79	12.50	4.00	4.55
Proc_5	8.33	0.00	3.73	0.00	5.62	0.00	0.00	4.00	5.19
Proc_8	8.33	0.02	8.97	10.00	5.60	9.09	0.00	4.00	7.47
main	8.33	0.11	8.55	0.00	10.14	0.00	0.00	12.03	10.72
strcmp	8.33	—	5.52	—	7.87	9.09	—	8.00	7.79
Func_1	0.00	33.27	10.07	0.00	10.16	18.18	25.00	12.00	13.31
Proc_7	0.00	0.04	10.21	10.00	13.46	12.40	25.00	12.00	12.34
Func_3	0.00	0.00	3.59	10.00	4.50	9.09	0.00	4.00	5.19
Proc_2	0.00	0.00	4.55	0.00	3.39	9.09	0.00	7.97	5.19
Proc_4	0.00	0.04	4.55	0.00	4.45	9.09	12.50	8.00	2.92
Proc_6	0.00	0.00	5.66	10.00	4.47	0.01	0.00	4.03	5.20
__divsi3	—	—	7.31	20.00	10.11	9.09	—	—	—

Table D.2: Cycle-Accurate Results using SnoopP to Profile Dhrystone on MicroBlaze systems that include and exclude the Hardware Multiplier and Divider.

Function Name	**Percentage of Execution Time (100 Passes)**	**Percentage of Execution Time (A Million Passes)**	**Percentage of Execution Time (100 Passes) HW multiply**	**Percentage of Execution Time (100 Passes) HW multiply & divide**
__mulsi3	23.49	23.56	—	—
__divsi3	14.88	14.93	20.68	—
main	14.75	14.51	19.22	23.90
strcmp	10.80	10.83	15.00	18.99
Proc_1	9.62	9.65	13.37	16.92
Proc_8	7.49	7.52	5.39	6.83
Func_2	4.88	4.89	6.78	8.58
Proc_6	3.39	3.40	4.72	5.97
Proc_3	2.17	2.17	3.01	3.81
Func_1	1.98	1.98	2.75	3.48
Proc_4	1.75	1.76	2.44	3.08
Proc_7	1.64	1.65	2.28	2.89
Proc_2	1.54	1.54	2.13	2.70
Func_3	0.82	0.83	1.15	1.45
Proc_5	0.76	0.77	1.06	1.34
malloc	0.02	0.00	0.03	0.04

Table D.3: The Results from Profiling AES on-chip with SnoopP for Both 2 and 400 Keys.

Function Name	**Percentage of Execution Time**
__modsi3	39.59
mul	23.30
__mulsi3	13.17
MixColumns	7.10
_vfprintf_r	5.37
ShiftRows	2.18
AddRoundKey	1.44
Substitution	1.23
blockEncrypt	0.70
cipherInit	0.46
sprintf	0.32
rijndaelEncrypt	0.17
main	0.14
memcpy	0.13
__divsi3	0.11

Table D.4: mb-gprof Statistics on Functions Comprising the AES Benchmark for 2 and 400 Different Keys with 10 Thousand Blocks Each.

Function Name	**2 Keys**		**400 Keys**	
	Percent Time (6Hz)	**Percent Time (100Hz)**	**Percent Time (6Hz)**	**Percent Time (100Hz)**
__modsi3	36.20	35.57	32.21	5.80
mul	29.50	29.37	13.41	8.42
__mulsi3	11.85	11.88	18.04	4.71
MixColumns	6.63	7.01	10.84	22.57
_vfprintf_r	4.71	4.80	7.49	18.19
_mbtowc_r	2.20	2.35	3.45	3.16
ShiftRows	2.05	2.34	3.58	8.86
AddRoundKey	1.50	1.69	2.66	6.77
Substitution	1.49	1.52	2.27	5.79
__sfvwrite	1.31	0.88	2.02	3.99
blockEncrypt	0.66	0.74	1.05	3.35
cipherInit	0.46	0.42	0.71	1.90
vfprintf	0.37	0.31	0.44	1.39
sprintf	0.27	0.31	0.46	1.43
memmove	0.26	0.20	0.47	0.93
rijndaelEncrypt	0.20	0.18	0.29	0.82
memcpy	0.14	0.11	0.19	0.51
__divsi3	0.07	0.14	0.16	0.63
localeconv	0.07	0.04	0.06	0.20
main	0.06	0.13	0.19	0.59
HexToBin	0.00	0.00	0.00	0.00
makeKey	0.00	0.00	0.00	0.00
rijndaelKeySched	0.00	0.0	0.00	0.0
_start1	—	0.00	—	0.00
strncpy	—	—	—	0.00

Appendix E

SnoopP HDL Source Code

```
--snoopy_types.vhd file
----------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


package snoopy_types is
  constant cntrsize : integer := 32;
  type cntrArray is array (natural range <>) of std_logic_vector(0 to cntrsize-1);

end package snoopy_types;
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- snoopy_wrapper.vhd file
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_Logic_arith.all;
use IEEE.STD_Logic_unsigned.all;

use Work.snoopy_types.all;

entity snoopy is
  generic(C_OPB_AWIDTH: INTEGER := 32;
  C_OPB_DWIDTH: INTEGER := 32;
  NUM_COUNTERS: INTEGER := 2; --Max of 10, Min of 1
  C_BASEADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FFE0";
  C_HIGHADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FFFF";
  RESET_ADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FFE4";
  INSTR_LOWERBND0: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND0: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND1: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND1: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND2: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND2: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND3: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND3: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND4: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND4: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND5: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND5: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND6: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND6: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND7: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND7: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND8: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND8: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND9: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND9: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND10: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND10: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND11: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND11: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND12: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND12: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND13: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND13: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND14: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND14: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154";
  INSTR_LOWERBND15: STD_LOGIC_VECTOR(0 to 31) := X"0080_0140";
  INSTR_UPPERBND15: STD_LOGIC_VECTOR(0 to 31) := X"0080_0154"
  );
  port( OPB_Clk: in STD_LOGIC;
OPB_Rst: in STD_LOGIC;

OPB_ABus: in STD_LOGIC_VECTOR(0 to C_OPB_AWIDTH-1);
OPB_BE: in STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH/8 -1);
OPB_DBus: in STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
OPB_RNW: in STD_LOGIC;
OPB_select: in STD_LOGIC;
OPB_seqAddr: in STD_LOGIC;

snoopy_DBus: out STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
snoopy_errAck: out STD_LOGIC;
snoopy_retry: out STD_LOGIC;
snoopy_toutSup: out STD_LOGIC;
snoopy_xferAck: out STD_LOGIC;

PC_EX: in STD_LOGIC_VECTOR(0 to 31);
valid_instr: in STD_LOGIC);

end entity snoopy;

architecture behaviour of snoopy is
  signal int_data_val: cntrArray(0 to NUM_COUNTERS-1);
  signal int_snoopy_DBus: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
  signal instr_lowerbnds: cntrArray(0 to 15);
  signal instr_upperbnds: cntrArray(0 to 15);
  signal rd_xferAck: STD_LOGIC;
  signal wr_xferAck: STD_LOGIC;
  signal int_snoopy_xferAck: STD_LOGIC;
  signal reset: STD_LOGIC;
  signal cntr_fx_sel: STD_LOGIC_VECTOR(0 to 1);

COMPONENT var_instr_cntrs IS
   GENERIC(C_OPB_AWIDTH: INTEGER := 32;
  C_OPB_DWIDTH: INTEGER := 32;
  NUM_COUNTERS: INTEGER := NUM_COUNTERS --Max of 10, Min of 1
  );
   PORT(clk: in STD_LOGIC;
reset: in STD_LOGIC;
instr_lowerbnds: in cntrArray(0 to NUM_COUNTERS-1);
instr_upperbnds: in cntrArray(0 to NUM_COUNTERS-1);
cntrs: out cntrArray(0 to NUM_COUNTERS-1);
cntr_fx_sel: in STD_LOGIC_VECTOR(0 to 1);
PC_EX: in STD_LOGIC_VECTOR(0 to 31);
valid_instr: in STD_LOGIC);
END COMPONENT var_instr_cntrs;

COMPONENT opb_output IS
   GENERIC(C_OPB_AWIDTH: INTEGER := C_OPB_AWIDTH;
   C_OPB_DWIDTH: INTEGER := C_OPB_DWIDTH;
   NUM_COUNTERS: INTEGER := NUM_COUNTERS; --Max of 10, Min of 1
RESET_ADDR: STD_LOGIC_VECTOR(0 to 31) := RESET_ADDR;
```

```
   C_BASEADDR: STD_LOGIC_VECTOR(0 to 31) := C_BASEADDR;
   C_HIGHADDR: STD_LOGIC_VECTOR(0 to 31) := C_HIGHADDR
);
   PORT( OPB_ABus: in STD_LOGIC_VECTOR(0 to C_OPB_AWIDTH-1);
OPB_Clk: in  STD_LOGIC;
OPB_RNW: in STD_LOGIC;
OPB_select: in STD_LOGIC;
cntrs: in cntrArray(0 to NUM_COUNTERS-1);
snoopy_DBus: out STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
reset: out STD_LOGIC;
snoopy_xferAck: out STD_LOGIC);
END COMPONENT opb_output;

begin

   instr_lowerbnds(0) <= INSTR_LOWERBND0;
   instr_upperbnds(0) <= INSTR_UPPERBND0;
   instr_lowerbnds(1) <= INSTR_LOWERBND1;
   instr_upperbnds(1) <= INSTR_UPPERBND1;
   instr_lowerbnds(2) <= INSTR_LOWERBND2;
   instr_upperbnds(2) <= INSTR_UPPERBND2;
   instr_lowerbnds(3) <= INSTR_LOWERBND3;
   instr_upperbnds(3) <= INSTR_UPPERBND3;
   instr_lowerbnds(4) <= INSTR_LOWERBND4;
   instr_upperbnds(4) <= INSTR_UPPERBND4;
   instr_lowerbnds(5) <= INSTR_LOWERBND5;
   instr_upperbnds(5) <= INSTR_UPPERBND5;
   instr_lowerbnds(6) <= INSTR_LOWERBND6;
   instr_upperbnds(6) <= INSTR_UPPERBND6;
   instr_lowerbnds(7) <= INSTR_LOWERBND7;
   instr_upperbnds(7) <= INSTR_UPPERBND7;
   instr_lowerbnds(8) <= INSTR_LOWERBND8;
   instr_upperbnds(8) <= INSTR_UPPERBND8;
   instr_lowerbnds(9) <= INSTR_LOWERBND9;
   instr_upperbnds(9) <= INSTR_UPPERBND9;
   instr_lowerbnds(10) <= INSTR_LOWERBND10;
   instr_upperbnds(10) <= INSTR_UPPERBND10;
   instr_lowerbnds(11) <= INSTR_LOWERBND11;
   instr_upperbnds(11) <= INSTR_UPPERBND11;
   instr_lowerbnds(12) <= INSTR_LOWERBND12;
   instr_upperbnds(12) <= INSTR_UPPERBND12;
   instr_lowerbnds(13) <= INSTR_LOWERBND13;
   instr_upperbnds(13) <= INSTR_UPPERBND13;
   instr_lowerbnds(14) <= INSTR_LOWERBND14;
   instr_upperbnds(14) <= INSTR_UPPERBND14;
   instr_lowerbnds(15) <= INSTR_LOWERBND15;
   instr_upperbnds(15) <= INSTR_UPPERBND15;

   cntr_fx_sel <= OPB_ABus(C_OPB_AWIDTH-4 to C_OPB_AWIDTH-3);

--Instantiates the proper number of counters and attaches them to the PC_EX bus
--looking for valid instructions that are in the specified range for each counter
V0: var_instr_cntrs port map ( OPB_Clk, reset, instr_lowerbnds(0 to NUM_COUNTERS-1),
instr_upperbnds(0 to NUM_COUNTERS-1),
int_data_val, cntr_fx_sel, PC_EX, valid_instr);

--Attaches the counters via a mux to the OPB bus.  Outputs the proper counter values
--when a valid address is read by the Master and Resets the counters when the Master
--writes to the reset address
O0: opb_output port map (OPB_ABus, OPB_Clk, OPB_RNW, OPB_select, int_data_val,
int_snoopy_DBus, reset, int_snoopy_xferAck);


   snoopy_errAck <= '0';
   snoopy_retry <= '0';
   snoopy_toutSup <= '0';
   snoopy_DBus <= int_snoopy_DBus;
   snoopy_xferAck <= int_snoopy_xferAck;

end architecture behaviour;
----------------------------------------------------------------------------



----------------------------------------------------------------------------
-- var_instr_cntrs.vhd file
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_Logic_arith.all;
use IEEE.STD_Logic_unsigned.all;

use Work.snoopy_types.all;

entity var_instr_cntrs is

  generic(C_OPB_AWIDTH: INTEGER := 32;
  C_OPB_DWIDTH: INTEGER := 32;
  NUM_COUNTERS: INTEGER := 2 --Max of 10, Min of 1
  );
  port( clk: in STD_LOGIC;
reset: in STD_LOGIC;
        instr_lowerbnds: in cntrArray(0 to NUM_COUNTERS-1);
        instr_upperbnds: in cntrArray(0 to NUM_COUNTERS-1);
cntrs: out cntrArray(0 to NUM_COUNTERS-1);
cntr_fx_sel: in STD_LOGIC_VECTOR(0 to 1);
PC_EX: in STD_LOGIC_VECTOR(0 to 31);
valid_instr: in STD_LOGIC);

end entity var_instr_cntrs;

architecture behaviour of var_instr_cntrs is
  signal int_data_val: cntrArray(0 to NUM_COUNTERS-1);
  signal int_snoopy_DBus: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
  signal enable: STD_LOGIC_VECTOR(0 to NUM_COUNTERS-1);
  signal int_pc_ex: STD_LOGIC_VECTOR(0 to 31);

COMPONENT clk_cntr IS
PORT(
clk : in STD_LOGIC;
reset : in STD_LOGIC;
enable : in STD_LOGIC;
sel_bits: in STD_LOGIC_VECTOR(0 to 1); --enable different
cnt : out STD_LOGIC_VECTOR(0 TO 31));
END COMPONENT clk_cntr;

COMPONENT valid_pc_addr IS
GENERIC( NUM_MSBS : INTEGER := 30);
PORT( pc_ex : in STD_LOGIC_VECTOR(0 TO 31);
lowerbnd : in STD_LOGIC_VECTOR(0 TO 31);
upperbnd : in STD_LOGIC_VECTOR(0 TO 31);
clk : in STD_LOGIC;
reset : in STD_LOGIC;
enable : out STD_LOGIC);
END COMPONENT valid_pc_addr;

COMPONENT reg IS
PORT(  clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
enable : IN STD_LOGIC;
d    : IN STD_LOGIC;
q     : OUT STD_LOGIC);
END COMPONENT reg;

begin
```

```
--latch each 'valid' PC_EX:
gen0: FOR i IN 0 TO 31 GENERATE
  regs: reg port map (clk, reset, valid_instr, pc_ex(i), int_pc_ex(i));
end generate;


GEN1: FOR i in 0 to (NUM_COUNTERS-1) GENERATE
enc: valid_pc_addr port map (int_pc_ex, instr_lowerbnds(i), instr_upperbnds(i),
clk, reset, enable(i));
cnt: clk_cntr port map (clk, reset, enable(i), cntr_fx_sel,
int_data_val(i));
     END GENERATE;

   cntrs <= int_data_val;

end architecture behaviour;
----------------------------------------------------------------------------



----------------------------------------------------------------------------
--clk_cntr.vhd
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_Logic_arith.all;
use IEEE.STD_Logic_unsigned.all;

entity clk_cntr is
   port(clk: in STD_LOGIC;
reset: in STD_LOGIC;
enable: in STD_LOGIC;
sel_bits: in STD_LOGIC_VECTOR(0 to 1); --enable different
cnt: out STD_LOGIC_VECTOR(0 TO 31));
end entity clk_cntr;

architecture behaviour of clk_cntr is
   constant cntrsize : INTEGER := 46; --46 bits
   signal int_cnt : STD_LOGIC_VECTOR(0 TO cntrsize-1);
begin

   process(clk, reset)
   begin
  if (reset = '1') then
 int_cnt <= (OTHERS => '0');
  elsif ( (rising_edge(clk)) and (enable = '1') ) then
 int_cnt <= int_cnt + '1';
  else
 int_cnt <= int_cnt;
  end if;
   end process;

--***REMEMBER THE "000...000" TERM SIZE IS DETERMINED BY THE CNTRSIZE***
cnt <=  (("000000000000000000") & int_cnt(0 to cntrsize-33))
WHEN sel_bits = "00" ELSE
int_cnt(cntrsize-32 to cntrsize-1) WHEN sel_bits = "01"
ELSE (OTHERS=> '0');

end architecture behaviour;
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--valid_pc_addr.vhd
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_Logic_arith.all;
use IEEE.STD_Logic_signed.all;
```

```
entity valid_pc_addr is
  generic(NUM_MSBS :  INTEGER := 24);
  port( pc_ex : in STD_LOGIC_VECTOR(0 TO 31);
lowerbnd : in STD_LOGIC_VECTOR(0 to 31);
upperbnd : in STD_LOGIC_VECTOR(0 to 31);
clk : in STD_LOGIC;
reset : in STD_LOGIC;
enable : out STD_LOGIC);

end valid_pc_addr;

architecture behaviour of valid_pc_addr is
  signal met_lowerbnd: STD_LOGIC;
  signal met_upperbnd: STD_LOGIC;
  signal base_sub: STD_LOGIC_VECTOR(0 to NUM_MSBS-1);
  signal high_sub: STD_LOGIC_VECTOR(0 to NUM_MSBS-1);

COMPONENT flipflop IS
PORT( clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
reset_val: IN STD_LOGIC;
d    : IN STD_LOGIC;
q     : OUT STD_LOGIC);
END COMPONENT flipflop;

begin

--in range = equal or greater than
base_sub <= pc_ex(0 TO NUM_MSBS-1) - lowerbnd(0 TO NUM_MSBS-1);
high_sub <= upperbnd(0 TO NUM_MSBS-1) - pc_ex(0 TO NUM_MSBS-1);

ff0: flipflop port map (clk, reset, '1', base_sub(0), met_lowerbnd);
ff1: flipflop port map (clk, reset, '1', high_sub(0), met_upperbnd);

        --the address is guaranteed to be valid due to the latching of the pc_ex,
        --this determines if it is in range for a counter (stalls will also be
        --caught as the latched value only changes when there is a valid address
        --in the pc_ex
enable <= met_lowerbnd nor met_upperbnd; --if either is 1, disable

end behaviour;
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
--flipflop.vhd
-----------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

--Entity Declaration:
entity flipflop is
port( clk : in STD_LOGIC;
reset : in STD_LOGIC;
reset_val : in STD_LOGIC;
d : in STD_LOGIC;
q     : out   STD_LOGIC);
end flipflop;

--Architecture Description:
architecture Behaviour OF flipflop IS
  signal int_q : STD_LOGIC;

begin
       process(clk, reset)
       begin
                if (reset = '1') then
                        int_q <= reset_val;
                elsif (rising_edge(clk))  then
                        int_q <= d;
                else
                        int_q <= int_q;
                end if;
        END PROCESS;
q <= int_q;
END Behaviour;
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--reg.vhd
----------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

--Entity Declaration:
entity reg is
port(  clk : in STD_LOGIC;
reset : in STD_LOGIC;
enable : in STD_LOGIC;
d : in STD_LOGIC;
q     : out   STD_LOGIC);
end reg;

--Architecture Description:
architecture Behaviour OF reg IS
  signal int_q : STD_LOGIC;

begin
       process(clk, reset)
       begin
                if (reset = '1') then
                        int_q <= '0';
                elsif ((rising_edge(clk)) and (enable = '1'))  then
                        int_q <= d;
                else
                        int_q <= int_q;
                end if;
        END PROCESS;
q <= int_q;
END Behaviour;
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--opb_output.vhd
----------------------------------------------------------------------------

library IEEE;
use IEEE.STD_Logic_1164.all;
use Work.snoopy_types.all;

entity opb_output is
  generic(C_OPB_AWIDTH: INTEGER := 32;
  C_OPB_DWIDTH: INTEGER := 32;
  NUM_COUNTERS: INTEGER := 2; --Max of 10, Min of 1
  RESET_ADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FFE4";
  C_BASEADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FF00";
  C_HIGHADDR: STD_LOGIC_VECTOR(0 to 31) := X"FFFF_FFFF"
);
  port( OPB_ABus: in STD_LOGIC_VECTOR(0 to C_OPB_AWIDTH-1);
OPB_Clk: in STD_LOGIC;
OPB_RNW: in STD_LOGIC;
OPB_select: in STD_LOGIC;
cntrs: in cntrArray(0 to NUM_COUNTERS-1);
snoopy_DBus: out STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
reset: out STD_LOGIC;
snoopy_xferAck: out STD_LOGIC);
```

```
end entity opb_output;

architecture behaviour of opb_output is
  signal valid_hi_reset_bits: STD_LOGIC;
  signal valid_reset_control: STD_LOGIC;
  signal valid_read_control: STD_LOGIC;
  signal valid_lo_addr_bits: STD_LOGIC;
  signal valid_hi_addr_bits: STD_LOGIC;
  signal sel_bits: STD_LOGIC_VECTOR(0 to 3);
  signal mux3_out: cntrArray(0 to 7);
  signal mux2_out: cntrArray(0 to 3);
  signal mux1_out: cntrArray(0 to 1);
  signal mux0_out: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
  signal mux_out: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
  signal int_snoopy_xferAck: STD_LOGIC;
  signal int_reset: STD_LOGIC;
  signal int_snoopy_DBus: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1);
  signal lowerbnd: STD_LOGIC_VECTOR(0 to C_OPB_AWIDTH-1):= C_BASEADDR;
  signal upperbnd: STD_LOGIC_VECTOR(0 to C_OPB_AWIDTH-1):= C_HIGHADDR;
  signal gnd_bus: STD_LOGIC_VECTOR(0 to C_OPB_DWIDTH-1):= X"0000_0000";

COMPONENT mux IS
PORT( in_0 : in STD_LOGIC_VECTOR(0 to 31);
in_1 : in STD_LOGIC_VECTOR(0 to 31);
sel_bit : in STD_LOGIC;
out_val : out STD_LOGIC_VECTOR(0 to 31));

END COMPONENT mux;

COMPONENT valid_opb_addr IS
   generic( NUM_MSBS :  INTEGER := 24);
   port( addr : in STD_LOGIC_VECTOR(0 TO 31);
lowerbnd : in STD_LOGIC_VECTOR(0 to 31);
upperbnd : in STD_LOGIC_VECTOR(0 to 31);
valid_addr : in STD_LOGIC;
enable : out STD_LOGIC);
END COMPONENT valid_opb_addr;

begin


--Assumptions (for now):
--valid addresses occur every 8 bytes to allow for 64 bit counters
--I've hard-wired a 16 to 1 mux architecture (not the most efficient)
--xferAck isn't guaranteed to be high for only one clk cycle

OE0:  valid_opb_addr port map (OPB_ABus, lowerbnd, upperbnd, OPB_select,
valid_hi_addr_bits);

  valid_lo_addr_bits <= '1' WHEN OPB_ABus(C_OPB_AWIDTH-2 to C_OPB_AWIDTH-1) = "00"
ELSE '0';
  valid_hi_reset_bits <= '1' WHEN  ((valid_hi_addr_bits = '1') and
(OPB_ABus(C_OPB_AWIDTH-8 to C_OPB_AWIDTH-3) = RESET_ADDR(24 to 29)) )
  ELSE '0';

--OPB_select is included in the determination of valid_hi_addr_bits
  valid_reset_control <= valid_lo_addr_bits and valid_hi_reset_bits and (not(OPB_RNW));
  valid_read_control <= valid_lo_addr_bits and valid_hi_addr_bits and OPB_RNW;

  sel_bits <= OPB_ABus(C_OPB_AWIDTH-8 to C_OPB_AWIDTH-5);

--HOW DO I GENERATE XFERACK AND VALID HI BITS FOR READ?

G3: IF (NUM_COUNTERS>1) GENERATE
G3A:  FOR i in 0 to ((NUM_COUNTERS/2)-1) GENERATE
M3: mux port map (cntrs(2*i), cntrs((2*i)+1), sel_bits(3), mux3_out(i));
END GENERATE;
      END GENERATE;
```

```
G3_0: IF (NUM_COUNTERS mod 2=0) GENERATE -- even
G3_0A: FOR i in (NUM_COUNTERS/2) to 7 GENERATE
M3_0: mux port map(gnd_bus, gnd_bus, sel_bits(3), mux3_out(i));
END GENERATE;
      END GENERATE;

G3_1: IF (NUM_COUNTERS mod 2=1) GENERATE -- odd
M3_1:        mux port map(cntrs(NUM_COUNTERS-1), gnd_bus, sel_bits(3),
mux3_out(NUM_COUNTERS/2));
G3_1A: FOR i in (NUM_COUNTERS/2)+1 to 7 GENERATE
M3_1A: mux port map(gnd_bus, gnd_bus, sel_bits(3), mux3_out(i));
END GENERATE;
      END GENERATE;


G2: FOR i in 0 to 3 GENERATE
M2: mux port map(mux3_out(2*i), mux3_out((2*i)+1), sel_bits(2), mux2_out(i));
END GENERATE;


G1: FOR i in 0 to 1 GENERATE
M1: mux port map(mux2_out(2*i), mux2_out((2*i)+1), sel_bits(1), mux1_out(i));
END GENERATE;


G0:     mux port map(mux1_out(0), mux1_out(1), sel_bits(0), mux0_out);
MOUT: mux port map(gnd_bus, mux0_out, valid_read_control, mux_out);

----Process for xmd output:
   process(OPB_Clk)
   begin
if( rising_edge(OPB_Clk) ) then
if ( valid_read_control = '1' ) then
int_reset <= '0';
int_snoopy_DBus <= mux_out;
int_snoopy_xferAck <= '1';
elsif ( valid_reset_control = '1' ) then
int_reset <= '1';
int_snoopy_DBus <= gnd_bus;
int_snoopy_xferAck <= '1';
else
int_reset <= '0';
int_snoopy_DBus <= gnd_bus;
int_snoopy_xferAck <= '0';
         end if;
    else
int_reset <= int_reset;
int_snoopy_DBus <= int_snoopy_DBus;
int_snoopy_xferAck <= int_snoopy_xferAck;
end if;
   end process;

  reset <= int_reset;
  snoopy_Dbus <= int_snoopy_DBus;
  snoopy_xferAck <= int_snoopy_xferAck;

end architecture behaviour;
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--mux.vhd
----------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

--Entity Declaration:
entity mux is
```

```
--GENERIC(bus_width: INTEGER := 32);
PORT( in_0: in STD_LOGIC_VECTOR(0 to 31);
in_1: in STD_LOGIC_VECTOR(0 to 31);
sel_bit:  in STD_LOGIC;
out_val: out STD_LOGIC_VECTOR(0 to 31));
end entity mux;

--Architecture Description:
architecture behaviour of mux is
begin
out_val <= in_0 WHEN sel_bit = '0' ELSE in_1;
end behaviour;
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--valid_opb_addr.vhd
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_Logic_arith.all;
use IEEE.STD_Logic_signed.all;

entity valid_opb_addr is
  generic(NUM_MSBS :  INTEGER := 24);
  port( addr : in STD_LOGIC_VECTOR(0 TO 31);
lowerbnd : in STD_LOGIC_VECTOR(0 to 31);
upperbnd : in STD_LOGIC_VECTOR(0 to 31);
valid_addr : in STD_LOGIC;
enable : out STD_LOGIC);

end valid_opb_addr;

architecture behaviour of valid_opb_addr is
  signal met_lowerbnd: STD_LOGIC;
  signal met_upperbnd: STD_LOGIC;
  signal base_sub: STD_LOGIC_VECTOR(0 to NUM_MSBS-1);
  signal high_sub: STD_LOGIC_VECTOR(0 to NUM_MSBS-1);
  signal in_range: STD_LOGIC;

begin

--in range = equal or greater than
base_sub <= addr(0 TO NUM_MSBS-1) - lowerbnd(0 TO NUM_MSBS-1);
high_sub <= upperbnd(0 TO NUM_MSBS-1) - addr(0 TO NUM_MSBS-1);

met_lowerbnd <= base_sub(0); --+ve means MSB is 0
met_upperbnd <= high_sub(0); --+ve means MSB is 0

in_range <= met_lowerbnd nor met_upperbnd; --if either is 1, disable
enable <= in_range and valid_addr;

end behaviour;
----------------------------------------------------------------------------
```

Appendix F

Input File Format for WOoDSTOCK

```
/* Filename: gen_monitor_input_format.c
 * Author:   Lesley Shannon
 * Date:     July 28, 2005
 * Description:
 *   This file describe the format for the input file used
 *   to describe the system to be monitored by WOoDSTOCK.
 *   To provide the information for the input file,
 *   draw a DAG and label the nodes (CEs) from 0 to
 *   NumNodes-1 and label the edges (Links) from 0 to
 *   NumLinks-1.
 *********************************************************/

NumNodes
NumLinks

Node0 NumInputs NumOutputs NumEquations
*Input(s)
*Output(s)
Equation0
*Equation1...

Node1 NumInputs NumOutputs NumEquations
*Input(s)
*Output(s)
Equation0
*Equation1...

****************************************************
Notes:

*The equations are used to indicate the relationship
between input packets received by the node to
output packets generated by the node.  Equations
use boolean equations where '*' indicates 'and'
and '+' indicates 'or'.  An equation is terminated
using a semicolon.
```