SIMPPL: An Adaptable SoC Framework Using a Programmable Controller IP Interface to Facilitate Design Reuse

Lesley Shannon, *Member, IEEE*, and Paul Chow, *Member, IEEE*

***Abstract*—As the complexity of designing system-on-chips increases, so does the need to abstract low-level design issues to improve designer productivity. The reuse of previously designed Intellectual Property (IP) modules is a common form of abstraction used to reduce design time. However, different applications typically use a variety of physical interfaces, communication protocols, and global system-level control for IP modules, which complicates design reuse. In this paper, we describe the SIMPPL system model and an abstraction for IP modules, called the computing element (CE), that facilitate the SoC design for both field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) platforms. The CE abstraction decouples the datapath and system-level communication from the application-specific control to promote design reuse by localizing control redesign of IP for new applications. The SIMPPL model facilitates multi-clock domain SoC designs and expedites system integration by defining the intermodule links and communication protocols.**

***Index Terms*—application-specific architectures, application-specific integrated circuits (ASICs), customizable controllers, design reuse, field-programmable gate arrays (FPGAs), Intellectual Property (IP) reuse, system integration, system-on-chip design.**

I. Introduction

THE term, system-on-chip (SoC), has been used with many different connotations in previous work. In this paper, we define an SoC as a collection of functional units 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 finite-impulse response (FIR) filters and fast Fourier transforms (FFTs) to more control intensive cores such as memory controllers and processors.

Unfortunately, reusing IP 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. Hardware designers not only need to consider changes to the module's functionality, but also to the physical interface and communication protocols [1], [2]. Depending on the amount of time required to adapt IP to a new application, there may be little benefit in reusing the IP.

In this paper, we describe how modeling SoCs as Systems Integrating Modules with Predefined Physical Links (SIMPPL [3]) expedites system integration. We also demonstrate how abstracting IP modules as computing elements (CEs) can reduce the complexities of adapting IP to new applications. The CE model separates the datapath of the IP from system-level control and communications. A lightweight controller provides the system-level interface for the IP module and executes a program that dictates how the IP is used in the system [4]. Localizing the control for the IP to this program simplifies any necessary redesign of the IP for other applications.

The remainder of this paper is structured as follows. Section II provides an overview of communication and IP core standards along with details of the SIMPPL model and the CE abstraction. The underlying SIMPPL controller architecture and instruction set are outlined in Section III and the SIMPPL controller sequencer's interface and programming model are discussed in Section IV. Section V details the additional functionality and hardware of the "debug" version of the controllers and Section VI outlines some SIMPPL SoC implementations. Finally, Section VII describes the implementation statistics for various controller and CE architectures and Section VIII concludes the paper along with providing suggestions for future work.

II. Background

This section begins with a description of the SIMPPL model and the CE abstraction. It is followed by a discussion of some previous work investigating on-chip interconnect structures and methods of simplifying IP reuse, demonstrating how SIMPPL fits into this work.

A. SIMPPL System Model

Fig. 1 illustrates the SIMPPL SoC architecture of a network of CEs comprising the hardware and software modules in the system. *I/O Communication Links* are represented as dotted arrows and are used to communicate with off-chip peripherals using the appropriate protocols. The solid arrows represent the *Internal Communication Links*. These are defined as point-to-

Manuscript received December 31, 2005; revised December 1, 2006.

L. Shannon was with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada. She is now with the School of Engineering Sciences, Simon Fraser University, Burnaby, BC V5A 1S6, Canada (e-mail: lshannon@ensc.sfu.ca).

P. Chow is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: pc@eecg.toronto.edu).

Digital Object Identifier 10.1109/TVLSI.2007.893645

1063-8210/$25.00 © 2007 IEEE

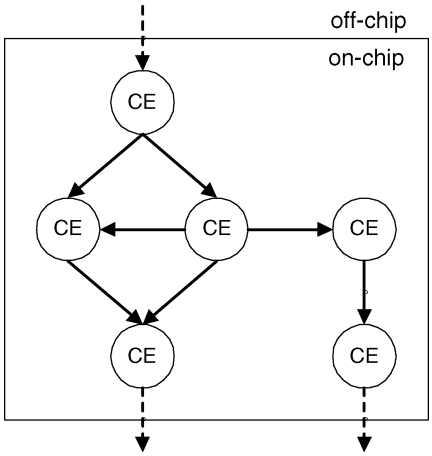

Fig. 1. Generic computing system described using the SIMPPL model.

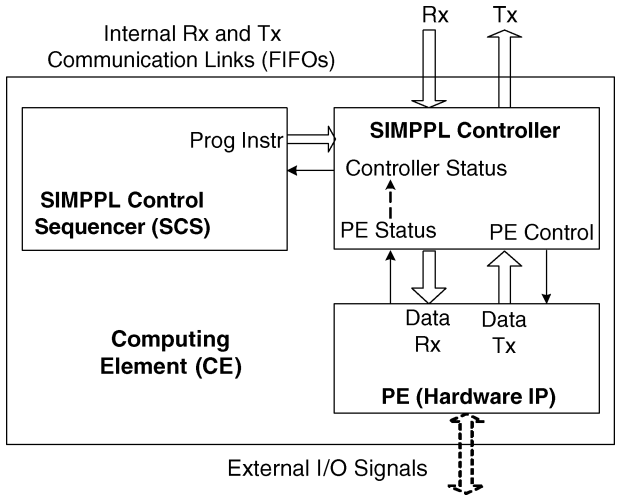

Fig. 2. Hardware CE abstraction.

point links to provide inter-CE communications where the communication protocols are abstracted from the physical links and implemented by the CEs.

For our current investigation, we are using n-bit wide asynchronous first-in–first-out (FIFOs) to implement the internal links in the SIMPPL model. Asynchronous FIFOs isolate clocking domains to individual CEs, allowing them to transmit and receive at data rates independent of the other CEs in the system. This simplifies system-level design by decoupling the processing rate of a CE from the inter-CE communication rate. Furthermore, the width and depth of the FIFO can be altered to provide greater bandwidth and to support data packets of varying lengths depending on the number and type of data words transmitted in a data packet. For the purposes of this discussion, we assume a FIFO width of 33 bits, but leave the depth variable.

The SIMPPL model representation of SoCs is reminiscent of Kahn process networks [5], particularly Data process networks [6], in that it is a collection of CEs interconnected via unidirectional links. However, unlike these models that assume the internal links have unbounded capacity, the SIMPPL model uses real FIFOs that have limited capacity. Recent work at Philips Research produced YAPI [7], an application model based on Kahn process networks that has been extended to support nondeterministic events and decouple the data types used for communications and computation. Although the SIMPPL model allows nondeterministic events, they are supported by the CE abstraction. 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 section.

B. SIMPPL CE Abstraction

The 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 [8]. 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, but general processors are too big and too slow for the hardware-to-hardware interactions of dedicated logic modules (hardware CEs). Ideally, a controller customized to the datapath of each CE could be used as a generic system interface, optimized for that specific CE's datapath. To this end, we have 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 application-specific integrated circuits (ASICs) and field-programmable gate array (FPGAs).

Fig. 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 to receive and transmit instruction packets. 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 IV-B illustrates a programming example for the SCS.

C. IP Reuse

Multiple books exist discussing the complexities involved in reusing legacy IP in new designs [1], [2]. The challenges of using IP to reduce design time due to problems that arise when

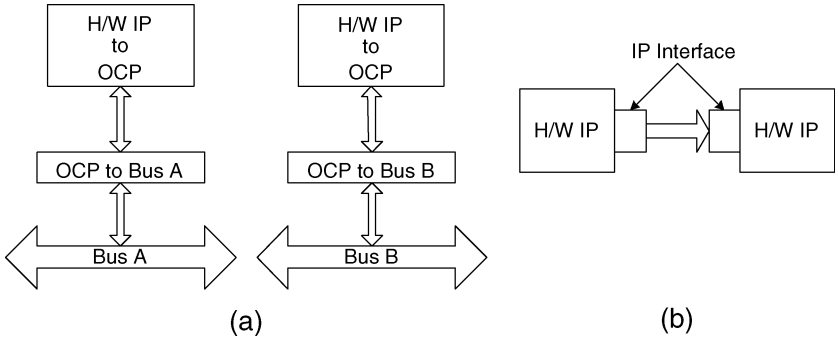

Fig. 3. Standardizing the IP interface using (a) OCP for different bus standards and (b) SIMPPL for point-to-point communications.

incorporating previously designed modules into new designs are of significant concern. This has led to the development of well-defined IP design methodologies [9], [10] 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 VSI Alliance has proposed the Open Core Protocol (OCP)[1] 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 Fig. 3. OCP is used to provide a well-defined socket interface for IP, which allows a designer to attach interface modules that act as adaptors to different shared bus and point-to-point communications standards. This allows a designer to easily connect a core to all bus types supported by the socket interface. In contrast, the SIMPPL model only targets the direct communication model and uses a defined, point-to-point interconnect structure for all on-chip communications, as shown in Fig. 3(b).

More recently, an interface adaptor logic layer has been proposed [11] 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 [12]. These protocols make it easier to port IP among different bus standards, whereas SIMPPL addresses the problems of adapting an IP core's functionality to the requirements of a new application.

D. 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 [13]–[15], but the maximum bandwidth for each bus is limited by the number of modules connected to it. The WISHBONE [16] 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.

[1]VSI Alliance home page. Available: http://www.vsia.org.

Berkeley's SCORE [17] 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. The adaptive SoC (aSOC) [18] 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 [19], [20]. One popular NoC topology is the mesh [21], [22], which has also been investigated on an FPGA platform [23]. 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.

III. SIMPPL Controller

The SIMPPL controller acts as the physical interface of the IP core to the rest of the system. It processes instruction packets received from other CEs and 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, the instructions it supports, and the format of its instruction packets are given below.

A. Instruction Packet Format

SIMPPL uses instruction packets to pass both control and data information over the internal communication links shown in Fig. 1. Fig. 4 provides a description of the generic instruction packet structure transmitted over an internal link. Although the current SIMPPL controller uses a 33-bit wide FIFO, the data word is only 32 bit. The remaining bit is used to indicate whether the transmitted word is an instruction or data. 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 (NDWs) sent or received in an instruction packet. 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.

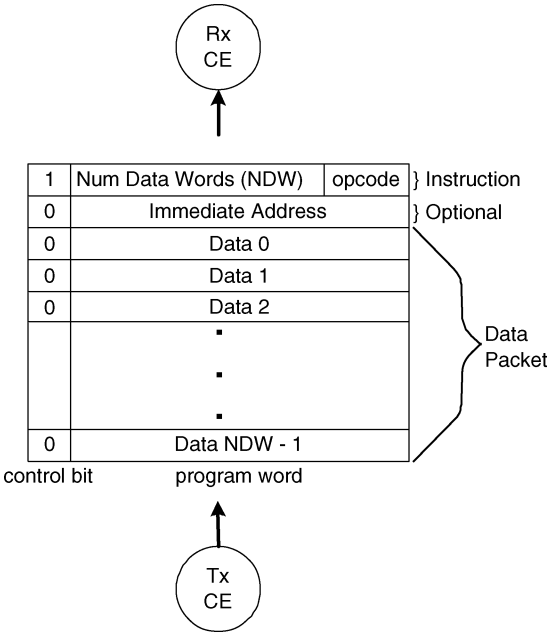

Fig. 4. An internal link's data packet format.

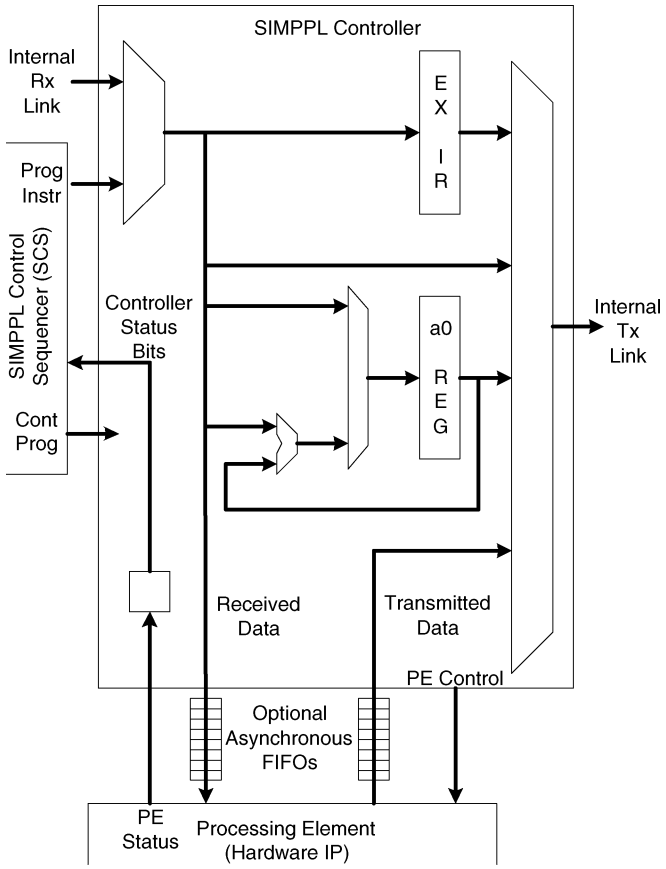

Fig. 5. An overview of the SIMPPL controller datapath architecture.

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 bit 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 bit, the internal links can be 25-bit wide, where 24 bit 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 (mandatory); 2) the address or status word (optional); and 3) the data words (optional). Each instruction packet begins with an instruction word that the controller interprets to determine how the packet is used by the CE. Since the SIMPPL model uses point-to-point communications, each CE can transfer/receive instruction packets directly to/from the necessary system CEs to perform the appropriate application-specific computations. Therefore, designers need not memory map the entire system, however, depending on the nature of the CE's operations, address or status information may be required. For example, CEs that provide access to memory require an address field so that designers can specify the data they wish to access. In contrast, functional units, such as a variable length decoder or run-level decoder, need not be memory mapped but may require status information about the system or data. Other types of functional units, such as an FIR or FFT, may require neither address nor status information. Hence this component of the address packet is optional. The remainder of the packet consists of NDW data words, as specified with the opcode.

B. Controller Architecture

Fig. 5 illustrates the SIMPPL controller's datapath architecture. The controller executes instructions received via the internal receive (Rx) link as well as those in the SCS. Instruction packets from the internal 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 internal 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 controller status bits for the SCS, which can then be used to determine the control flow of a program as will be discussed in Section IV-A. Finally, the SIMPPL controller's architecture includes the register $a0$. It is provided to allow designers to generate an address or to store local status information for the instruction packets transmitted

TABLE I
CURRENT INSTRUCTION SET SUPPORTED BY THE SIMPPL CONTROLLER

Instruction Type	Rd Req	Rx	Wr	Issue Instr	Exec. Instr	Addr Field	Data Field
Immediate Data Transfer	X	X	X	S/R	S/R		X
Immediate Data + Immediate Address	X	X	X	S/R	S/R	X	X
Register Initialization			X	S	S	X	
Register Arithmetic			X	S	S		
Immediate Data + Indirect Addressing	X	X	X	S	S	X	X
Immediate Data + Autoincrement	X	X	X	S	S	X	X
Bypass				S/R	S/R		X
No-op				S	R		
Reset				S	R		

to other CEs. The contents of $a0$ may represent the local address or status information of data from the transmitting CE or the address being accessed within the CE processing the instruction packet. The usage of the $a0$ register is left to the discretion of the designer, but is limited by the controllers current instruction set.

The format of an instruction data packet sent via the internal transmit (Tx) link is dictated by the instruction currently being executed. The inputs multiplexed to the Tx link are the instruction, an immediate address that is required in some instructions, the address or status stored in the 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.

C. Controller Instruction Set

The SIMPPL Controller's instruction set is divided into two groups; instructions that perform a control operation, and 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 I lists all the instructions currently supported by the SIMPPL controller. 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 some types of hardware may be missing, the instructions in Table I 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 I 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 I is the immediate data transfer instruction. It consists of one instruction word of the format shown in Fig. 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 $a0$ register with an immediate value is a two word instruction, where the first contains the opcode and the second is the new address. The register arithmetic instructions are single word instructions used to add or subtract an offset to the current value stored in the $a0$ register. A CE's $a0$ register can be used to store local status information, or to provide an immediate address for any data transfer instruction packets 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. Fig. 6 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 IV-A. Finally, the *reset* instruction can be transmitted from 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

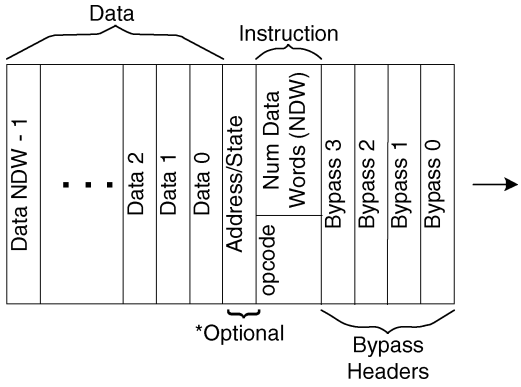

Fig. 6. A data packet with four bypass instructions.

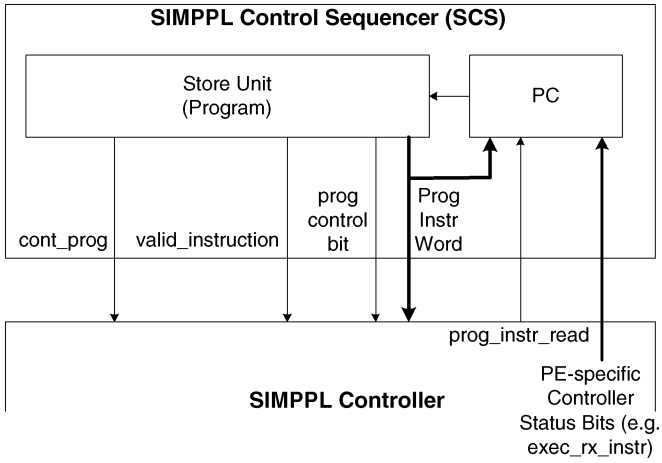

Fig. 7. Standard SIMPPL control sequencer structure and interface to the SIMPPL controller.

may only produce data or consume data. The Producer controller (Producer) is designed for CEs 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 write PE data to a Tx link.

IV. 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 architecture of the SCS for both ASIC and FPGA platforms, and to 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.

A. 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.

Fig. 7 illustrates the SCS structure and its interface with the SIMPPL controller via six standardized signal types. 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. Finally, the SCS has access to a set of PE-specific controller status bits that can be used to branch control within SCS program.

For example, if the SIMPPL Controller provides a status bit that indicates when the controller is executing an instruction from an internal Rx Link (e.g. *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 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' *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 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

```
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;
}
```

Fig. 8. Pseudocode for the sensor unit's SCS program.

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.

B. 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 Fig. 8. At present, we do not have compiler support for the SIMPPL controller and all programs (SCSs) are hand generated. Fig. 9 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 Fig. 8 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 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 $a0$ 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 Fig. 9, 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.

C. 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

```
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;
}
```

Fig. 9. Pseudo-HDL code to implement the state machine for the sensor unit's program counter and the valid instruction signal.

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.

Fig. 10 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

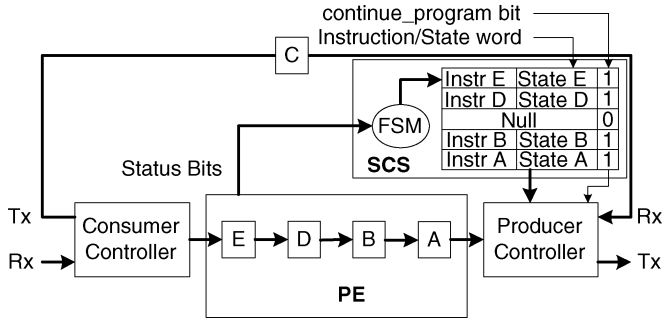

Fig. 10. A CE with multiple packets of data in flight.

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 Fig. 10.

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 Fig. 10. 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 Fig. 10, 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 dequeueing the "null" instruction. This will be followed by packets D and E being sent to the next CE.

V. Debug Controllers

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. Furthermore, since the CE is being tested on-chip, it is orders of magnitude faster than simulation. Here we introduce a debug SIMPPL controller (debug controller), based on the execute SIMPPL controller (execute controller) described in Section III, that allows the detection of low-level programming and integration errors.

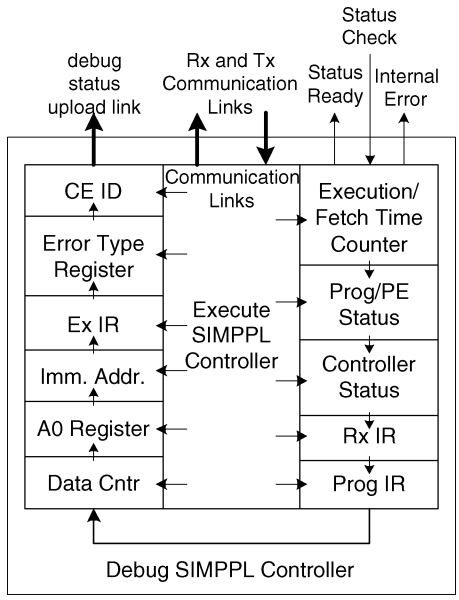

Fig. 11. The SIMPPL debug controller architecture.

A. Debug-Controller Architecture and Interface

Fig. 11 shows the architecture of a debug controller, with the execute controller described in Section III forming the central component. While the execute controller has three states in the instruction execution state machine: fetch, decode, and execute, the debug 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 Fig. 11, 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.

TABLE II
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

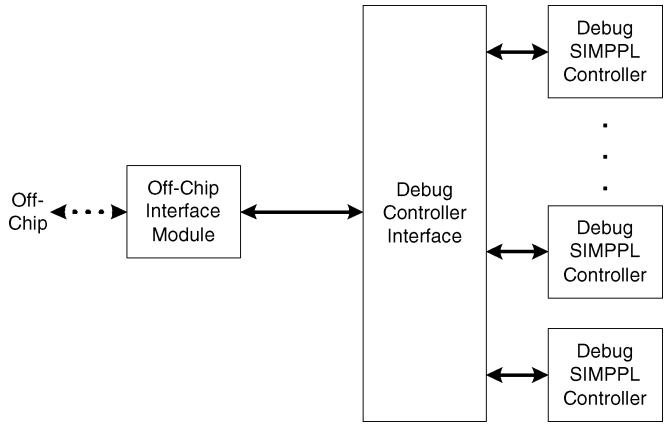

Fig. 12. SIMPPL debug controller interface.

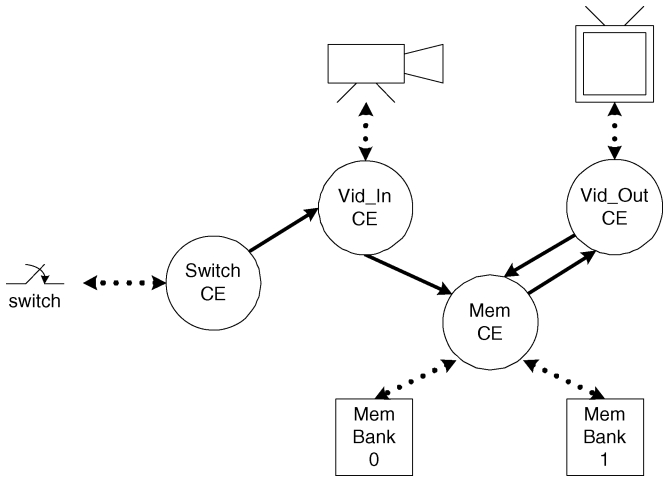

Fig. 13. SIMPPL model for the video streaming and snapshot applications.

The debug status upload link is implemented as an additional asynchronous FIFO link that is used to upload the debugging information to the debug interface shown in Fig. 12. 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 flip-flops.

B. 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 II 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.

VI. 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 27 MHz, and an RGB video DAC operating at 25 MHz. This section describes the nature of the three applications and discusses the effects on system design time.

A. SIMPPL SoC Applications

Fig. 13 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 the 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 to an SVGA monitor. By adding the Switch CE to the video system, the user can create a snapshot system, where the SVGA display is only updated 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,

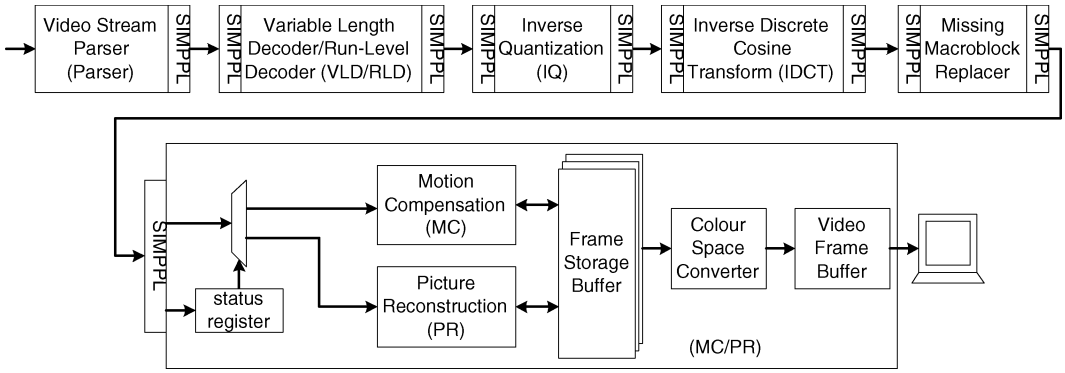

Fig. 14. The SIMPPL model for an MPEG-1 video decoder.

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 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 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.

Fig. 14 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 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 IV-C and the CE architecture in Fig. 10, 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.

TABLE III
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

B. SIMPPL SoC Implementation Statistics

Table III summarizes the time required to integrate the CEs and create the SCSs for the systems shown in Figs. 13 and 14. 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 PEs (Fig. 13), which required approximately 40 h. However, the integration of the CEs and the design of their respective SCSs took only 4.5 h 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 h. Thus, not only does the SIMPPL framework reduce system integration time, but it also facilitates the reuse of CEs for new applications.

It took 18 h to properly connect all the CEs and to generate the appropriate SCSs for the SIMPPL MPEG-1 Video Decoder System. Integrating all the MPEG-1 hardware PEs with Producer and Consumer controllers required an additional 39 h, or 2.4%, of the total system design time of 1607 h. 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

TABLE IV
SIMPPL CONTROLLER IMPLEMENTATION STATISTICS

Controller Type	FPGA platform			ASIC platform- Area		ASIC platform- Speed	
	Area		Max Frequency	Area	Max Frequency	Area	Max Frequency
	LUTs	Flipops	(MHz)	(10^3um^2)	(MHz)	(10^3 um^2)	(GHz)
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

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.

VII. 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 VI.

A. Controller Implementation Statistics

Table IV summarizes the area and operating frequency measurements obtained for the different types of SIMPPL controller implemented on both FPGA and ASIC platforms. The ASIC measurements are obtained using Synopsis 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' 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 2–4 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 required a maximum of 355 LUTs and 125 flip-flops. In contrast, the debug controllers can only operate at 180–199 MHz while utilizing 945–1002 LUTs and 478 flip-flops. The additional flip-flops used in the debug controllers are attributed mainly to the 11 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 IV 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 IV 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 [24]. 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 10 mm^2 in 90-nm technology according to industry sources. The maximum number of Consumer debug controllers, the largest of the SIMPPL controllers, that can be packed into 10 mm^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 [25] that of the standard cell implementation and the full custom layout area could be reduced to 6.9% that of a standard cell version [26].

Table V 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

TABLE V
EXECUTION OVERHEAD CLOCK CYCLES FOR THE CONSUMER, PRODUCER, AND FULL SIMPPL CONTROLLERS

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

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.

B. 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 Fig. 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 Vid_In, Vid_Out, Switch CEs. Fig. 15 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, labelled *SIMPPL Controller Mem Bank A* and *SIMPPL Controller Mem Bank B* in Fig. 15, that interface with an arbiter module to determine which controller has access to which PE, in this case the Memory Bank Controllers.

The arbiter module communicates with both SIMPPL controllers in the Shared architecture (Fig. 15) 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 as a separate module so that the user can adapt the arbiter to suit different applications.

Finally, Fig. 16 shows a block diagram of the *Pipelined* architecture utilized for the CEs in the MPEG-1 video decoder

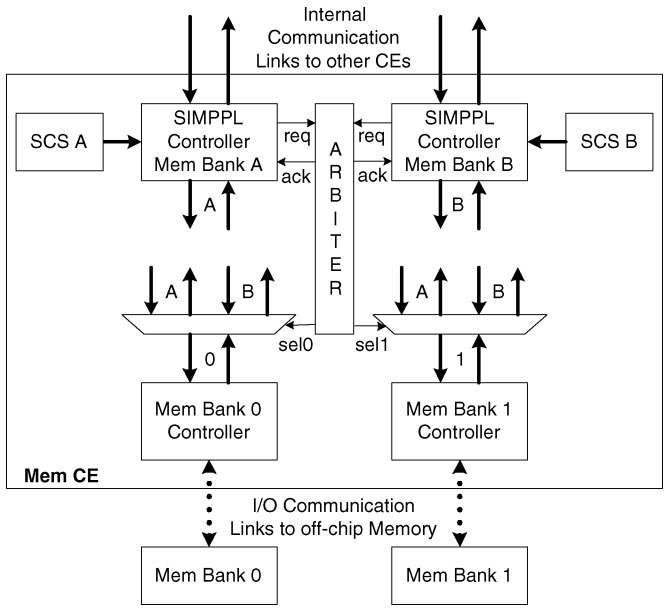

Fig. 15. Shared CE architecture for a shared memory CE.

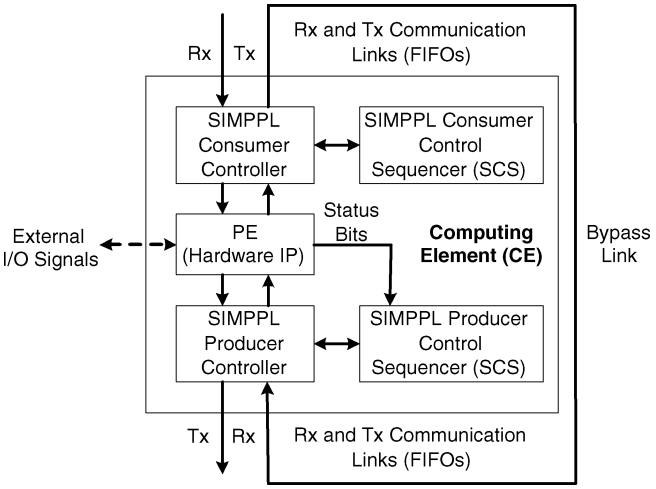

Fig. 16. The Pipelined CE architecture.

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.

C. 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 VI-A on an FPGA using the nine hardware CEs described in Table VI. All the results reported in the table were obtained using the version 7.1.4 of the Xilinx ISE tools.[2] The

[2]The LUTs and flip-flops 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 VI
IMPLEMENTED CEs

CE Name	Architecture Type	Controllers	PE Resources	
			LUTs	Flipops
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

first 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 flip-flops 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 being implemented in the decoder.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we discussed the SIMPPL model for SoC designs 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. We have created the SIMPPL controller to act as the system interface and to process 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 μm^2 to 29 615 μm^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 could be further reduced if they were 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 h compared to the 140 h 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. 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.

REFERENCES

[1] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*. Norwell, MA: Kluwer Academic, 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*. Norwell, MA: Kluwer Academic, 1999.
[3] L. Shannon and P. Chow, "Maximizing system performance: Using reconfigurability to monitor system communications," in *Proc. IEEE Int. Conf. on Field-Programm. Technol.*, Dec. 2004, pp. 231–238.
[4] ——, "Simplifying the integration of processing elements in computing systems using a programmable controller," in *proc. IEEE Symp. on Field-Programm. Custom Comput. Mach.*, Apr. 2005, pp. 63–72.
[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IPIF Congress 74*, 1974, pp. 471–475.
[6] E. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 471–475, May 1995.
[7] 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 *Proc. 37th Design Automat. Conf.*, Jun. 2000, pp. 402–405.
[8] K. Jasrotia and J. Zhu, "Stacked FSMD: A power efficient micro-architecture for high level synthesis," in *Proc. Int. Symp. on Quality Electronic Des.*, Mar. 2004, pp. 425–430.
[9] W. Savage, J. Chilton, and R. Camposano, "IP Reuse in the system on a chip era," in *Proc. 13th Int. Symp. on Syst. Synthesis*, Sep. 2000, pp. 2–7.
[10] G. Martin, "Design methodologies for system level IP," in *Proc. IEEE Conf. on Design Automat. and Test in Eur.*, Feb. 1998, pp. 286–302.
[11] T. Lee and N. W. Bergmann, "An interface methodology for retargettable FPGA perihperals," in *Proc. Int. Conf. on Eng. of Reconfigurable Systems and Algorithms*, Jul. 2003, pp. 1–7.
[12] Xilinx, OPB IPIF architecture. (2003) [Online]. Available: http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb ipif.pdf
[13] P. J. Aldworth, "System-on-a-chip bus architecture for embedded applications," in *Proc. IEEE Int. Conf. on Computer Design*, Oct. 1999, pp. 297–298.
[14] D. Flynn, "AMBA: Enabling reusable on-chip design," *IEEE Micro*, vol. 17, no. 1, pp. 20–27, Jul. 1997.
[15] IBM Corp., The CoreConnect Bus architecture. (1999) [Online]. Available: www.ibm.com/chips/products/coreconnect

[16] OpenCores, Specification for the WISHBONE system-on-chip (SoC) interconnect architecture for portable IP cores. (Sep. 2002) [Online]. Available: http://www.opencores.org/projects.cgi/web/wishbone/wbspec b3.pdf, Revision B.3
[17] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. De-Hon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Extended abstract," in *Int. Conf. on Field Programmable Logic and Appl.*, Aug. 2000, pp. 605–614 [Online]. Available: http://brass.cs.berkeley.edu/documents/score tutorial.pdf
[18] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Trans. VLSI Syst.*, vol. 12, no. 7, pp. 711–726, Jul. 2004.
[19] A. Adrianhantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino, "Spin: A scalable, packet switched, on-chip micro-network," in *Proc. Conf. on Design, Automat. and Test in Eur.*, Mar. 2003, pp. 70–73.
[20] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.
[21] S. Kumar, A. Jantsch, J. Soininen, and M. Forsell, "A network on chip architecture and design methodology," in *Proc. IEEE Annu. Symp. on VLSI*, 2002, pp. 105–112.
[22] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnect networks," in *Proc. ACM/IEEE Des. Automat. Conf.*, Jun. 2001, pp. 684–689.
[23] G. Brebner and D. Levi, "Networking on chip with platform FPGAs," in *IEEE Int. Conf. Field-Programm. Technol.*, Dec. 2003, pp. 13–20.
[24] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2006, pp. 21–30.
[25] D. Chinnery and K. Keutzer, "Closing the gap between ASIC and custom: An ASIC perspective," in *Proc. 37th Design Automat. Conf.*, Jun. 2000.
[26] W. Dally and A. Chang, "The role of custom design in ASIC chips," in *Proc. 37th Design Automat. Conf.*, Jun. 2000, pp. 643–647.

Lesley Shannon (S'03–M'06) received the B.Sc.Eng. degree in electrical engineering from the University of New Brunswick, Fredericton, NB, Canada, in 1999, and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, ON, Canada, in 2001 and 2006, respectively.

In 2006, she joined the School of Engineering Science, Simon Fraser University, Burnaby, BC, Canada, as an Assistant Professor. Her current research interests include computing system design, particularly, system-on-chip architectures, embedded computing systems, and reconfigurable computing, as well as computing system design methodologies and CAD tools.

Paul Chow (S'79–M'83) received the B.A.Sc. degree (with hons) in engineering science, and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 1977, 1979, and 1984, respectively.

In 1984 he joined the Computer Systems Laboratory at Stanford University, Stanford, CA, as a Research Associate, where he was a major contributor to an early RISC microprocessor design called MIPS-X, one of the first microprocessors with an on-chip instruction cache. Since January 1988, he has been with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, where he is now a Professor and holds the Dusan and Anne Miklas Chair in Engineering Design. His research interests include high performance computer architectures, architectures and compilers for embedded processors, VLSI systems design, and field-programmable gate array architectures, systems, and applications. From 1998 to 2001, he was the Chairman of the Technical Advisory Committee (TAC) for the Canadian Microelectronics Corporation (CMC). Since 2001, he has been a member of the CMC Board of Directors and still participates as a member of the TAC. In December 1999, he co-founded AcceLight Networks to build a high-capacity, carrier-grade, optical switching system. He was the Director of ASIC Technology from May 2000 to October 2002 and managed a group of close to 30 designers that developed over 40 large, high-performance field-programmable gate array designs.