Simplifying the Integration of Processing Elements in Computing Systems using a Programmable Controller

Lesley Shannon and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
{lesley, pc}@eecg.toronto.edu

ABSTRACT

As technology sizes decrease and die area increases, designers are creating increasingly complex computing systems using FPGAs. To reduce design time for new products, the reuse of previously designed Intellectual Property (IP) cores is essential. However, since no universally accepted interface standards exist for IP cores, there is often a certain amount of redesign necessary before they are incorporated into the new system. Furthermore, the core’s functionality may need updating to support the requirements of the new application.

This paper demonstrates how the SIMPPL system model allows designers to rapidly implement on-chip systems comprising multiple Computing Elements (CEs). Furthermore, using a controller-based interface to manage inter-CE transfers enables users to easily adapt the control sequence of individual CEs to suit the needs of new applications without necessitating the redesign of other elements in the system. Two systems using three different hardware modules adapted to CEs are described to illustrate the power and simplicity of the SIMPPL model. It required a total of six hours to implement both designs on-chip once the individual CEs had been designed.

1. INTRODUCTION

Commercial Field Programmable Gate Arrays (FPGAs) are now large enough to support designs comprising multiple processors and dedicated hardware modules. To minimize the design time for such complex circuits, designers try to reuse previously designed modules, known as Intellectual Property (IP) cores. For example, an MPEG-4 design, as shown in Figure 1 [1] uses cores such as the DCT, IDCT, and Frame Store (memory) that are common to other multimedia applications. While this system may be viewed as dedicated hardware, it can also be thought of as a computing system that uses direct communication between different types of processing elements (PEs). The goal of achieving the rapid development of custom computing systems requires that low-level hardware design be minimized.

Integrating PEs at the physical level necessitates the development of a hardware interface that provides synchronization, control, and data transfer between cores. Synchronization supplies the handshaking needed to indicate when there is a valid data transfer and the control signals direct the operations to be performed and report the current status of the core. Finally, the data transferred between cores must be in the required format and sequence to be correctly interpreted by the receiving cores.

To simplify the physical level connections (physical interface) between PEs, a standardized hardware interface and communication structure is required. Currently, IP cores are connected together using direct communication, often implemented as random glue logic, or by adapting the cores to a common standard, such as a bus. However, designs that pass data from one module to the next, as in Figure 3, typically use a direct communication structure.

Previous work introduced the need for standardizing the physical interconnect between system modules so that the user can abstract the physical design information from the data transferred between modules [2]. Modelling designs as Systems Integrating Modules with Predefined Physical Links (SIMPPL) allows each module to represent a dedicated hardware or processor-based (software) Computing Element (CE) that connects to other CEs via fixed communication links. A CE combines a PE, an IP core that performs some given function, with the control sequence that dictates how the PE is used by the rest of the system.

Since the communication links are fixed in the SIMPPL model, the actual physical interfacing of CEs is a trivial problem. With a fixed physical interface, the mechanism for the physical transfer of data across a link is provided so that the designer can focus on the meaning of the data transfers, rather than how to connect the wires. This is similar to the software world where, when using procedures, a programmer is never concerned with the use of the stack or

Figure 1: Block diagram of the MPEG4 video coder.
2. BACKGROUND

The following is a discussion of some previous work investigating on-chip interconnect structures and methods of simplifying IP reuse. The section closes with a system-level description of the SIMPPL system architecture.

2.1 IP Reuse

Multiple books exist discussing the complexities involved in reusing legacy IP in new designs [3, 4]. The challenges of using IP to reduce design time due to 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 [5, 6] to ensure reusability of cores with fixed interfaces and fixed 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) [7] to enable the separation of external core communications from the IP core’s functionality, similar to the SIMMPL model. Both communication models are illustrated in Figure 2. OCP is used to provide a well-defined socket interface for IP that allows the designer to easily at-tach fixed interface modules that support different bus standards. This allows a designer to easily connect a core to all bus types supported by the standard. In contrast, the SIMPPL model targets the direct communication model using a fixed, point-to-point interconnect structure for all on-chip communications.

More recently an Interface Adaptor Logic layer has been proposed [8]. It is very similar to the OCP, using a fixed socket interface for IP modules, however unlike OCP, it is 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 even provides its own bus-interface module for IP with a defined socket interface [9]. These protocols make it easier to port IP among different bus standards, whereas SIMMPL addresses the problems of adapting an IP core’s functionality to the requirements of a new application.

2.2 On-Chip Communication Structures

Multiple different on-chip interconnect strategies have been proposed for SoC design, including hierarchical buses that use bridges to connect to each other [10, 11, 12]. These bus structures can all be mapped to an FPGA, but the maximum bandwidth for each bus is limited by the number of buses connected to it. The WISHBONE [13] SoC Interconnect architecture provides multiple different interconnect structures that can also be mapped to an FPGA, thereby allowing the designer to select the bus architecture for a particular system. Since all 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 [14] 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, without defining a physical connection. Adaptive System-on-chip (aSOC) [15] provides 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.

Another form of on-chip communication uses a network. MicroNetwork [16] has a pipelined data network to communicate between modules. It also includes the previously described OCP socket as an interface for the heterogeneous system modules and rotates the communications resources among the inter-module transfer requests. A Stanford project on scalable network fabrics [17] also uses a common network for passing data between modules where the system is partitioned into interconnected modules and the routing network is associated with the tiles such that packets are rerouted at each tile. This idea of using data packets is similar to the data-passing method used in the SIMPPL model, but the lack of flexibility in a predefined placement and structured network architecture is not as suitable to SoC designs on FPGAs as the point-to-point communications of the SIMPPL model.

2.3 SIMPPL Computing System Model

Figure 3 illustrates the previously proposed macro-level description of a system built using the SIMPPL architectural model [2]. The solid arrows indicate internal links and the dotted arrows indicate I/O communication links to ex-
Figure 3: A generic computing system described using the SIMPPL model.

External devices. The I/O communication links and protocols between a CE and an off-chip device are determined by the off-chip device, however, the internal communication links are fixed and the communication protocols between modules are abstracted from the physical links and may be adapted to the requirements of each CE.

An n-bit wide asynchronous FIFO is used as the standard internal link for this investigation of the SIMPPL model. Asynchronous FIFOs provide clocking flexibility to system designers as they allow CEs to send and transmit data at independent data rates. This decouples the CE’s inter-module communications from processing, thereby allowing independent clock domains for individual CEs without complicating the system level design. Since the number and type of data words transmitted or received by a CE is dependent on the nature of its computation, the width and depth of the FIFO can be altered to provide greater bandwidth and support data packets of varying lengths. To support the communication protocols described in the following section, the FIFO data-width is currently set to 33-bits but the depth is left variable.

3. SIMPPL CONTROLLER

The SIMPPL controller architecture provides the physical interface to the IP core and supports an instruction set designed to facilitate reprogramming the core’s operations for different applications. For example, a CE that has an audio sampling PE can be reprogrammed to sample the signal received from an external audio device at different rates depending on the requirements of the system. Details and an example of how the controller supports reprogramming are given below.

3.1 Controller Architecture

Figure 4 illustrates the main components of a hardware CE. The PE is used to perform a specific function, be it computational 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 represents the separation of the controller functionality from communications enabled by the fixed interface and communication protocols. The SCS stores the program and supplies the sequence of instructions to be executed by the controller for each application. The instruction set supported by the SIMPPL controller is described in Section 3.2 and provides the flexibility required to adapt the CE to reflect the requirements of each application. The protocol used to communicate over the internal links requires that all transmissions must initiate with an instruction to indicate to the receiving controller how to process the received information. This condition enables a receiving CE to correctly interpret the data packets sent from transmitting CEs and satisfies the need for handshaking in inter-CE communications.

Figure 5 illustrates the SIMPPL controller’s datapath architecture. Figure 4: The internal structure of a hardware CE.

Figure 5: An overview of the SIMPPL controller datapath architecture.
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 generates status bits that can be used by the SCS to determine if the program should branch. The status bits are specific to each PE so that they can better aid in the control of program execution order.

The format of an output 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, the immediate address that is part of 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 in a controller’s program. 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 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/produces at the necessary data rate.

3.2 Controller Instruction Set

While 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. This is shown in Figure 6, which 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) to represent the opcode. All SIMPPL controller instructions require at most two words – the instruction word and an optional immediate address data word.

The 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. The first is the read request. It is issued by the program of one CE and sent to another CE requesting that data be transmitted back to the original CE. Secondly, a receive instruction must be generated as the first transmitted word to accompany the data sent back to the first CE, since all transfers via internal links start with an instruction. Finally, the program can use a data write to accompany data words transmitted to another CE.

Table 1 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, that still provides sufficient programmability such that the cores are easily reconfigured for any potential application. While instructions that are needed to fully support the reconfigurability of some types of hardware may be missing, the instructions in Table 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 1 describes the operation being performed by the instruction. The following three columns are used to indicate whether the different instruction types can be used to request data (Rd), write data (Wr), or receive data (Rx). The final two columns are used to denote whether the instruction may be issued or executed from the Program memory (P) or internal receive communication link (R). Before describing these instructions in detail, it is important to realize that the instruction types are not fully mapped for issue and execution. As an example, it is only necessary to allow immediate data transfer instructions with and without an immediate address to be sent via the internal link as the receiving CE will be unaware of the value stored in the a0 register of the transmitting CE. The usage of a0 in program instructions helps to provide a level of indirection that reduces program complexity but is irrelevant to inter-CE communication.

The first instruction type described in Table 1 is the immediate data transfer instruction. It consists of one instruction word of the format shown in Figure 6, where the three most significant bytes give the number of data words being transferred and the least significant byte indicates the opcode. The two LSBs of the opcode indicate whether the data transfer is a read request, a write, or a receive. Similar
The address register arithmetic instructions are single word instructions used to add or subtract an offset to the current address. For the controller, the "Wait Receive" combined with the "Noop" instructions can be used to provide handshaking between CEs. For example, one controller can execute the wait instruction, which stalls the controller until an instruction is received via an internal link. When the instruction is received, the wait will be cleared and the received instruction will then execute on the controller. In the case where the user desires no operation to be performed, except the termination of the wait instruction, a noop should be used. Finally, the reset instruction can be used to clear all the status signals and registers for the controller.

3.3 SIMPPL Control Sequencer

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 the series of instructions used to perform the designed operations. Execution order is dictated by the processor’s Program Counter (PC), which commissions the next instruction 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 fetch instructions from the Store Unit, which acts like a memory.

Assume a hardware system that consists of a memory and a sensor unit used to measure multiple environmental quantities at a set time interval. The total storage requirements for each set of measurements is 32 bytes 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. The SIMPPL controller for the sensor unit has a status bit, valid_sensor_data indicating when a new set of measurements is available for storage in memory. The pseudocode for the sensor unit’s SIMPPL controller program is given in Figure 8. Figures 9 and 10 provide pseudo-HDL implementations of the state machines and output signals used in a SCS for the sensor unit.

The nextPC state only changes after an instruction has been read or all 1024 samples have been written to memory. Although the PC is commonly thought of as an address that selects the memory location of the next instruction to be executed, the store unit required for this example’s program is small enough that a state machine is sufficient. The valid_instruction signal is high during the initialization of the address register. However, since the data write instruction should only occur when the sensor has new data to transmit to the memory, it is assigned the value of the valid_sensor_data status bit in the Write autoinc state. When the program has completed, the PC goes to the Done state, where no further instructions are executed, and the valid_instruction signal goes low permanently.

4. SAMPLE SIMPPL MODULES

This section discusses the three hardware CEs designed to interface with three of the off-chip components on the Xilinx Multimedia board, the composite video input decoder, the video Digital-to-Analog Converters (DAC) for the VGA output, and two of the five 2MB ZBT memory chips. It begins with an overview of the shared memory CE architecture followed by a discussion of the implementation details.
if (rst==1) {
    PCstate <= Write a0 state;
} else {
    PCstate <= nextPC;
}

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

Figure 9: Pseudo-HDL code to implement the state machine for the sensor unit’s program counter.

4.1 Shared Memory CE Architecture

Figure 11 provides a high level view of a shared memory CE. For the two video applications described in Section 5, memory is required to store both the current image being written to the monitor and the next image being read in from the video recorder. These two frame buffers are stored on separate memory banks to allow parallel access to memory. Each memory bank has its own dedicated memory controller and since the video decoder interface core (VidIn CE) and SVGA output interface core (VidOut CE) need to make independent accesses to the memory banks, they also have independent SIMPPL controllers running individual programs.

An 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. The arbiter is currently designed to give higher priority to memory bank requests from the VidIn CE than from the VidOut CE. The VidIn CE is given whichever bank it requests as soon as it is available. However, the VidOut CE is given the requested bank only when it is not being used by the VidIn CE, otherwise, the VidOut CE is given access to the remaining unused bank. The arbiter 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 reuse the shared memory CE (Mem CE) and adapt the arbiter to suit different applications.

4.2 CE Descriptions

The VidIn CE interfaces with the video decoder and reads in the data in YCrCb format and then converts it

Figure 10: Pseudo-HDL code used to implement the output signals from the sensor unit’s SCS.

/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 11: The shared memory Computing Element.
The execution stage of the SIMPPL controller requires a different maximum number of clock cycles in overhead depending on the CE's functionality. The VidIn CE and the Mem CE require at most two extra cycles in the execute stage, which occurs when an instruction requires the controller to transmit a write instruction plus a destination address via internal link. In the case of the VidOut CE, a read request that uses indirect addressing and autoincrementing is the most costly instruction in the execute stage, requiring an extra four cycles. A final clock cycle of overhead is incurred by both Video CEs as they buffer the data sent/received by their PEs to guarantee proper functionality.

Table 2's penultimate row indicates the number of clock cycles of advanced warning that can be provided to the controller to indicate that a data transfer is imminent. In the case of the Mem CE, there can be no warning as memory accesses are non-deterministic from system to system. However, the Video CEs require data transfers to set intervals, so it is possible to use the control signals to indicate that the controller should be initiating a data transfer earlier to mask the overhead. In the case of the VidOut CE, there are only four clock cycles warning as that is the number of stages in the pipeline used to convert the data from YCrCb to RGB. The VidOut CE is able to provide a user defined amount of warning by setting the Hardware IP core's parameter NUM_CYCLES_PREFETCH. Currently, this parameter is set to 20 clock cycles, which is found to be more than sufficient for both of the sample systems.

Finally, the last row of the table indicates the total number of clock cycles of overhead incurred by using the SIMPPL controller interface for each of the CEs. The VidIn controller incurs only one clock cycle of overhead, which disappears if the buffering between the PE and the controller is eliminated. The VidOut CE is recorded as having negative overhead because it can mask the cost of the SIMPPL communication protocol. It avoids any effective overhead by having sufficient warning before any data transfers are required. Finally, the Mem SIMPPL controllers accrue a minimum of seven cycles of overhead, which can only be masked from the rest of the system by clocking the Memory CE faster than the rest of the system. In the two sample systems, the only possible data corruption arises from missing data transmitted by the VidIn CE to the Mem CE. However, with the buffering built into the system and the Video CEs having large down-times between data transfers, the Mem CE is able to ensure that all memory requests are properly serviced.

5. SAMPLE DESIGNS

This section describes the two sample systems implemented using the CEs described in Section 4. Figure 12 illustrates

\[\text{Table 2: Implementation Statistics for the Video In, Video Out, and Memory CEs.}\]

<table>
<thead>
<tr>
<th>Measured Quantity</th>
<th>VidIn CE</th>
<th>VidOut CE</th>
<th>Mem CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of LUTs</td>
<td>359</td>
<td>266</td>
<td>436</td>
</tr>
<tr>
<td>Number of flipflops</td>
<td>177</td>
<td>163</td>
<td>161</td>
</tr>
<tr>
<td>Instr. Fetch Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Instr. Decode Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Mem. Arb. Overhead</td>
<td>N/A</td>
<td>N/A</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Instr. Execute Overhead</td>
<td>2 cycles</td>
<td>4 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Buffering. Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>N/A</td>
</tr>
<tr>
<td>Early Indication Cycles</td>
<td>4 cycles</td>
<td>20 cycles</td>
<td>N/A</td>
</tr>
<tr>
<td>Total Overhead</td>
<td>1 cycle</td>
<td>-13 cycles</td>
<td>7 cycles</td>
</tr>
</tbody>
</table>

The first two are the fetch and decode stages, similar to the Video CEs, but the third is a memory arbitration request stage. For instructions that do not require access to memory, such as writes to the address register, the overhead is one cycle. However, if the instruction accesses memory, then the minimum overhead is three cycles, assuming no memory request conflicts.

To RGB format. The SIMPPL controller for the VidIn CE is clocked at 27MHz, the same frequency as the video input sampling rate from the video decoder. The VidOut CE communicates with the video DAC used to drive the SVGA monitor. It receives the data in RGB format via an internal link and writes the data to the video DAC driving the display. While the SVGA sampling rate is 25 MHz, the clock for the SIMPPL controller runs at 50 MHz to guarantee that there is always valid data to be written to the monitor. Finally, there is the Mem CE that is used to store video input data and to send video output data. Both external memory banks and SIMPPL controllers are clocked at 54MHz to ensure that the CE services memory requests faster than they are made by the VidIn CE and VidOut CEs.

4.3 CE Implementation Results

The remainder of Table 2 is concerned with the number of clock cycles of overhead incurred by adding an instruction packet header to each data transmission. Both the VidIn and VidOut CEs use three-stage state machines to control instruction execution, where one cycle of overhead is incurred during each of the fetch and decode stages, while the execute phase overhead varies. The Mem CE, however, uses a four-stage state machine to execute each instruction.

The simulation was used to study the properties of the SIMPPL controller and the number of latchs required. The number of latchs required are shown in Table 2. The number of latchs required is significantly lower for the VidIn CE and VidOut CE than the Mem CE. This is because the Video CEs are not required to support the full instruction set due to their PE's functionality. For example, the VidIn CE does not support any instructions that receive data transfers as there is no data consumed by its PE. Similarly, the VidOut CE does not produce data and, therefore, does not implement the data transmit instructions. The Memory banks, however, can both read and write data and so their controllers require the full instruction set, which almost doubles the number of supported instructions.

The execution stage of the SIMPPL controller requires a different maximum number of clock cycles in overhead depending on the CE's functionality. The VidIn CE and the Mem CE require at most two extra cycles in the execute stage, which occurs when an instruction requires the controller to transmit a write instruction plus a destination address via internal link. In the case of the VidOut CE, a read request that uses indirect addressing and autoincrementing is the most costly instruction in the execute stage, requiring an extra four cycles. A final clock cycle of overhead is incurred by both Video CEs as they buffer the data sent/received by their PEs to guarantee proper functionality.

Table 2's penultimate row indicates the number of clock cycles of advanced warning that can be provided to the controller to indicate that a data transfer is imminent. In the case of the Mem CE, there can be no warning as memory accesses are non-deterministic from system to system. However, the Video CEs require data transfers at set intervals, so it is possible to use the control signals to indicate that the controller should be initiating a data transfer earlier to mask the overhead. In the case of the VidOut CE, there are only four clock cycles warning as that is the number of stages in the pipeline used to convert the data from YCrCb to RGB. The VidOut CE is able to provide a user defined amount of warning by setting the Hardware IP core's parameter NUM_CYCLES_PREFETCH. Currently, this parameter is set to 20 clock cycles, which is found to be more than sufficient for both of the sample systems.

Finally, the last row of the table indicates the total number of clock cycles of overhead incurred by using the SIMPPL controller interface for each of the CEs. The VidIn controller incurs only one clock cycle of overhead, which disappears if the buffering between the PE and the controller is eliminated. The VidOut CE is recorded as having negative overhead because it can mask the cost of the SIMPPL communication protocol. It avoids any effective overhead by having sufficient warning before any data transfers are required. Finally, the Mem SIMPPL controllers accrue a minimum of seven cycles of overhead, which can only be masked from the rest of the system by clocking the Memory CE faster than the rest of the system. In the two sample systems, the only possible data corruption arises from missing data transmitted by the VidIn CE to the Mem CE. However, with the buffering built into the system and the Video CEs having large down-times between data transfers, the Mem CE is able to ensure that all memory requests are properly serviced.

5. SAMPLE DESIGNS

This section describes the two sample systems implemented using the CEs described in Section 4. Figure 12 illustrates

\[\text{Table 2: Implementation Statistics for the Video In, Video Out, and Memory CEs.}\]

<table>
<thead>
<tr>
<th>Measured Quantity</th>
<th>VidIn CE</th>
<th>VidOut CE</th>
<th>Mem CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of LUTs</td>
<td>359</td>
<td>266</td>
<td>436</td>
</tr>
<tr>
<td>Number of flipflops</td>
<td>177</td>
<td>163</td>
<td>161</td>
</tr>
<tr>
<td>Instr. Fetch Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Instr. Decode Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Mem. Arb. Overhead</td>
<td>N/A</td>
<td>N/A</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Instr. Execute Overhead</td>
<td>2 cycles</td>
<td>4 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Buffering. Overhead</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>N/A</td>
</tr>
<tr>
<td>Early Indication Cycles</td>
<td>4 cycles</td>
<td>20 cycles</td>
<td>N/A</td>
</tr>
<tr>
<td>Total Overhead</td>
<td>1 cycle</td>
<td>-13 cycles</td>
<td>7 cycles</td>
</tr>
</tbody>
</table>

The first two are the fetch and decode stages, similar to the Video CEs, but the third is a memory arbitration request stage. For instructions that do not require access to memory, such as writes to the address register, the overhead is one cycle. However, if the instruction accesses memory, then the minimum overhead is three cycles, assuming no memory request conflicts.

The execution stage of the SIMPPL controller requires a different maximum number of clock cycles in overhead depending on the CE's functionality. The VidIn CE and the Mem CE require at most two extra cycles in the execute stage, which occurs when an instruction requires the controller to transmit a write instruction plus a destination address via internal link. In the case of the VidOut CE, a read request that uses indirect addressing and autoincrementing is the most costly instruction in the execute stage, requiring an extra four cycles. A final clock cycle of overhead is incurred by both Video CEs as they buffer the data sent/received by their PEs to guarantee proper functionality.

Table 2's penultimate row indicates the number of clock cycles of advanced warning that can be provided to the controller to indicate that a data transfer is imminent. In the case of the Mem CE, there can be no warning as memory accesses are non-deterministic from system to system. However, the Video CEs require data transfers at set intervals, so it is possible to use the control signals to indicate that the controller should be initiating a data transfer earlier to mask the overhead. In the case of the VidOut CE, there are only four clock cycles warning as that is the number of stages in the pipeline used to convert the data from YCrCb to RGB. The VidOut CE is able to provide a user defined amount of warning by setting the Hardware IP core's parameter NUM_CYCLES_PREFETCH. Currently, this parameter is set to 20 clock cycles, which is found to be more than sufficient for both of the sample systems.

Finally, the last row of the table indicates the total number of clock cycles of overhead incurred by using the SIMPPL controller interface for each of the CEs. The VidIn controller incurs only one clock cycle of overhead, which disappears if the buffering between the PE and the controller is eliminated. The VidOut CE is recorded as having negative overhead because it can mask the cost of the SIMPPL communication protocol. It avoids any effective overhead by having sufficient warning before any data transfers are required. Finally, the Mem SIMPPL controllers accrue a minimum of seven cycles of overhead, which can only be masked from the rest of the system by clocking the Memory CE faster than the rest of the system. In the two sample systems, the only possible data corruption arises from missing data transmitted by the VidIn CE to the Mem CE. However, with the buffering built into the system and the Video CEs having large down-times between data transfers, the Mem CE is able to ensure that all memory requests are properly serviced.

5. SAMPLE DESIGNS

This section describes the two sample systems implemented using the CEs described in Section 4. Figure 12 illustrates
In CE to request MemB
In CE.
MemB to signify that
In MemA, it has
In CE to the memory and then in-
In CE
Out, and Mem CEs, there are multiple ways to

MemB controller is responsible for servic-
MemA. Thus, the
In CE data packet
Out CE. Since the Vid
Out CE stalls, wait-
In CE is using either Memory Bank 0 or
Out CE
MemB's SCS uses 3 LUTs and
In and Vid
In CE, which uses 29 LUTs and 20
Out CE once the first Vid
In CE has started writ-
Out CE should start writing data to the SVGA to
MemA
Out CE requires 32 LUTs and 23 

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 most obvious method would be 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 be-
ing received and the video data written to the SVGA. This
is because the video recorder may not be at the start of a
video frame when the system comes out of reset.

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, wait-
ing for an indication that the Vid_In CE has started writ-
ing a new frame to the Mem CE. Figure 13 illustrates the
pseudocode for the Vid_In CE, which uses 29 LUTs and 20
flipflops in its SCS. Note that the base address for Memory
Bank 0 is denoted as 0x80000000 and the base address for
Memory Bank 1 is 0x40000000. Because the video input
signal is interleaved, all the odd lines of the video frame are
transmitted first followed by the even lines. Since the mon-
tor used for these systems has a line length of 640 pixels,
only the first 640 pixels received from the video recorder
are stored to memory. However, this parameter in the SCS
program can be changed if a different monitor is used. The
program uses an indirect write plus autoincrement to Tx the
line of data from the Vid_In CE to the memory and then in-
crements a0 by 640 a second time so that the video data


5.1 Streaming Video

The first design implements a video streaming design, where each new video frame is stored in a memory bank.
The following new frame is stored in the other memory bank
so that the previously received frame can be concurrently
written to the SVGA monitor. 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 most obvious method would be 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 be-
ing received and the video data written to the SVGA. This
is because the video recorder may not be at the start of a
video frame when the system comes out of reset.

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, wait-
ing for an indication that the Vid_In CE has started writ-
ing a new frame to the Mem CE. Figure 13 illustrates the
pseudocode for the Vid_In CE, which uses 29 LUTs and 20
flipflops in its SCS. Note that the base address for Memory
Bank 0 is denoted as 0x80000000 and the base address for
Memory Bank 1 is 0x40000000. Because the video input
signal is interleaved, all the odd lines of the video frame are
transmitted first followed by the even lines. Since the mon-
tor used for these systems has a line length of 640 pixels,
only the first 640 pixels received from the video recorder
are stored to memory. However, this parameter in the SCS
program can be changed if a different monitor is used. The
program uses an indirect write plus autoincrement to Tx the
line of data from the Vid_In CE to the memory and then in-
crements a0 by 640 a second time so that the video data


frame = 0;
field = 0; // 0 = odd field; 1 = even field
while (1) {
    // ***Write-Frame-to-Memory Loop***
    while(!done_frame) {
        if ((field==0) && (frame==0))
            a0 = 0x800000000;
        else if ((field==1) && (frame==0))
            a0 = 0x800000280;
        else if ((field==0) && (frame==1))
            a0 = 0x400000000;
        else // ((field==1) && (frame==1))
            a0 = 0x400000280;
        while(!done_field) {
            while(wait_for_next_line);
            num_pixels = 640;
            while (((valid_video_data) &&(num_pixel > 0)) ) {
                write data word to (a0)+;
                num_pixels--;
            }
            a0 = a0 + 640; // skip next line
        }
        field = ~field;
    }
    frame = ~frame;
    // ***End of Write-Frame-to-Memory Loop***
}

Figure 13: Pseudocode for the Vid_In CE in the
streaming video design.
5.2 Video Camera Snap Shots

The objective for this system is to have the video camera behave as a still-shots camera. The user can indicate when a picture should be taken by toggling a switch on the board. Figure 14 illustrates the pseudocode for the VidIn CE, which uses 34 LUTs and 23 flipflops in its SCS. The only changes required to the control sequence are the addition of three lines to the main loop. Line A is used to wait for a snapshot request from the user and is implemented using a wait rx instruction. When the user toggles the switch, a noop is transmitted to the VidIn CE clearing the wait request (Line B) and then the controller waits for the start of the next valid frame to transmit to memory (Line C). The code used to transmit the frame to memory is the same as the “Write-Frame-to-Memory” loop used in the streaming video application shown in Figure 13.

As in the video streaming system, the SIMPPL MemA controller has a null program since all the necessary data transfers are initiated by the VidIn CE. However, the SIMPPL MemB requires 40 LUTs and 22 flipflops to implement the SCS, which runs a significantly more complex program (shown in Figure 15) than in the previous system. Since the SIMPPL MemB controller has status bits that can indicate when the VidIn core is accessing a memory bank, it is responsible for selecting and writing the current video frame to the VidOut CE line by line. In between frames, the program for SIMPPL MemB checks the new_frame variable to see if the VidIn has accessed a memory bank. This variable is updated in parallel with the program’s execution by using a separate state machine to check the controller’s status bits for memory accesses by the VidIn CE. If the new_frame variable is set, the memory writes the current frame to the VidOut CE again and then switches to writing the video frame stored in the other memory bank. The program delays switching the memory banks by a frame to ensure that the VideoIn controller has finished writing the new frame to the new bank. In this system, the VidOut CE only needs to indicate when a new line of video data should be written to the monitor, by clearing the wait_for_next_line request, and then wait for the data to arrive. This reduces the size of the VidOut’s SCS to 17 LUTs and 10 flipflops. Since the snapshot video system had virtually the same system level connections and by using the programs created for the streaming video system as a starting point, the new video system required approximately another 1.5 hours to get running on the board.

5.3 Summary

Although the two video systems are quite similar, the

```c
frame = 0;
field = 0; //odd field
while (1) {
    while(!take_picture_request); //A
    clear(take_picture_request); //B
    while(!start_of_new_frame); //C

    //***Write-Frame-to-Memory Loop
    //from Figure 13
}
```

Figure 14: Pseudocode for the VidIn CE in the snapshot video design.

```
while(!done_frame) {
    num_pixels = 640;
    while(wait_for_next_line);
    while ((valid_video_data)
        kK (num_pixel > 0)) {
        write data word from (a0)+;
        num_pixels--;
    }
}
```

Figure 15: Pseudocode for the SIMPPL MemB controller program in the snapshot video design.

```
//new_frame is set and cleared in a separate
//process.
clear_request(new_frame);
switch_frame = 0;
current_bank = 0;
while (1) {
    if (switch_frame==1)
        current_bank = "current_bank;

    //Set the address from which the next frame
    //is written to the SVGA
    if (current_bank==0)
        a0 = 0x800000000;
    else // (current_bank==1)
        a0 = 0x400000000;

    //Set the switch frame variable if a new
    //frame has started being written to memory
    switch_frame = new_frame;

    //Send a clear request as a new_frame
    //will be written after the current frame.
clear_request(new_frame);

    while(!done_frame) {
        num_pixels = 640;
        while(wait_for_next_line);
        while ((valid_video_data)
            kK (num_pixel > 0)) {
            write data word from (a0)+;
            num_pixels--;
        }
    }
```

```
frame = 0;
field = 0; //odd field
while (1) {
    while(!take_picture_request); //A
    clear(take_picture_request); //B
    while(!start_of_new_frame); //C

    //***Write-Frame-to-Memory Loop
    //from Figure 13
}
```

Although the two video systems are quite similar, the
Figure 16: The system connections for the two sample systems.

The system infrastructure and the controllers for each application. Although the streaming video system can be designed to limit the necessary redesign for the snapshot system if that is part of the initial specification, the system-level design requires significantly more work than using the SIMPPL model. The user must develop a hardware interface and communication protocols among the three cores before creating the control sequence for each system.

6. CONCLUSIONS AND FUTURE WORK

The SIMPPL controller for hardware CEs facilitates the reprogramming of CEs to suit different applications. Furthermore, the fixed internal communication links simplify the physical interfacing of each CE into the system. These two benefits greatly reduce the redesign time required to reuse the hardware CEs. Three different hardware CEs have been used to implement two systems with varied control sequences in a total of six hours on a Xilinx Virtex II FPGA.

By incorporating the SIMPPL controller into hardware CE designs, it is possible to create a library of cores that can be used to reduce the design time for future custom computing systems. Currently, an image processing group at the University of Toronto is investigating the idea of using the SIMPPL controller to help create a library of image processing IP cores. The next phase of the project is to study systems designed with the SIMPPL model using on-chip CAD tools to maximize the performance of such systems. Future work will focus on further reducing the design time of new systems. To this end, the development of tools that can be used to autogenerate application-specific CE control sequencers and even SIMPPL controllers with instruction sets customized to specific PEs will be investigated.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council and the Ontario Government. The authors would like to thank Nathalie Chan King Choy for her design contributions and Tomasz Czajkowski, Ian Kuon, and Peter Jamieson for their many helpful comments and suggestions. Finally, thanks to CMC/SOCRN for providing the prototyping system and Xilinx for providing the tools.

7. REFERENCES


