Abstract

DWIEL, BRANDON H. FPGA Modeling of Diverse Superscalar Processors. (Under the direction of Dr. Eric Rotenberg.)

There is increasing interest in using Field Programmable Gate Arrays (FPGAs) as platforms for computer architecture simulation. A wide range of FPGA-based modeling approaches have proliferated in recent years, as recently documented in the FPGA Architecture Model Execution (FAME) taxonomy. As multi-core and many-core have been the key drivers of FAME technologies, the emphasis of model development has been in the "uncore" (caches, cache coherence, interconnection networks, etc.). Hence, there is a lack of FPGA models of complex superscalar cores.

We frame the problem as follows. The ideal superscalar processor model should combine (1) the configurability of a software model (*i.e.*, ability to vary superscalar parameters such as fetch, issue, and retire widths, depths of pipeline stages, queue sizes, *etc.*), (2) the cycle-accuracy of a register-transfer-level (RTL) model, and (3) the simulation speed of FAME. Researchers have demonstrated mapping RTL models of fixed commercial designs to FPGAs. The FPGA models are cycle-accurate and fast, but key superscalar dimensions (*e.g.*, pipeline stage widths) are not configurable like software models. Approaches that synthesize software simulators to FPGAs provide a path toward fast simulation of arbitrary superscalar configurations but the cycle-accuracy is limited by the software simulator.

This thesis describes FPGA-Sim, a configurable and FPGA-synthesizable simulator that models the RTL designs of diverse out-of-order superscalar processors. FPGA-Sim is (1) configurable like a software model in major superscalar dimensions, (2) cycle-accurate, and (3) orders-of-magnitude faster than a C++ simulator.

FPGA Modeling of Diverse Superscalar Processors

by
Brandon H. Dwiel

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Rhett W. Davis	Dr. Greg Byrd

Dr. Eric Rotenberg
Chair of Advisory Committee

Biography

The author earned a bachelor of science in computer engineering from Southern Illinois University, Carbondale, in 2009. He joined North Carolina State University in 2010 to pursue a master of science degree in computer engineering.

Acknowledgements

This research was supported by NSF grants No. CCF-0811707 and CCF-1018517, Intel, and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Tables

List of Figures

CHAPTER

1

Introduction

There is increasing interest in using Field Programmable Gate Arrays (FPGAs) as platforms for computer architecture simulation. Interest is driven by the need to simulate multi-core architectures, the need for longer running benchmarks to evaluate these architectures at scale, and new capability to map large architectures to single-FPGA systems due to greater FPGA capacity and the perfecting of design multiplexing techniques. A wide range of FPGA-based modeling approaches have proliferated in recent years, as recently documented in the thorough FPGA Architecture Model Execution (FAME) taxonomy [1] . As multi-core and many-core have been the key drivers of FAME technologies, the emphasis of model development has been in the "uncore" (caches, cache coherence, interconnection networks, etc.). Hence, there is a lack of FPGA models of complex superscalar cores.

In this thesis, we frame the following challenge (and opportunity): The ideal superscalar processor model would combine (1) the configurability of a software model (i.e., ability to vary superscalar parameters such as fetch, issue, and retire widths, depths of pipeline stages, queue sizes, etc.), (2) the cycle-accuracy of a register-transfer-level (RTL) model, and (3) the simulation speed of FAME. Researchers have demonstrated mapping RTL models of fixed commercial designs to FPGAs [2, 3, 4] . The FPGA models are cycle-accurate and fast, but key superscalar dimensions (e.g., pipeline stage widths) are not configurable like software models. Other approaches, that synthesize software cycle-level simulators to FPGAs, suggest a path toward fast simulation of arbitrary superscalar configurations. Cycle-accuracy is limited by the software simulator, however.

This thesis describes FPGA-Sim, a configurable and FPGA-synthesizable simulator that models the RTL designs of diverse out-of-order superscalar processors. FPGA-Sim is (1) configurable like a software model in major superscalar dimensions, (2) cycle-accurate, and (3) orders-of-magnitude faster than a C++ simulator. There are two major aspects to this modeling effort. The first is creating the configurable RTL model of the superscalar processor. The second is automatically mapping any one of its configurations to an FPGA.

1.1 Configurable RTL Model

We leveraged the FabScalar tool-set [5] to develop the configurable RTL model. Our approach was to use FabScalar to generate the widest and deepest superscalar processor in its repertoire – a superset core. We then inserted Verilog preprocessor macros within each canonical pipeline stage to be able to narrow it to a desired width and collapse it to the desired degree of sub-pipelining. So the RTL model is statically configurable: the footprint of the core is that of the chosen configuration, not the superset core, and there is no extra synthesized logic or configuration bits for this configurability.

1.2 Automatic and Efficient FPGA Mapping

We faced many of the same challenges presented in past work, most notably, mapping multi-ported RAMs to the FPGA's dual-ported RAMs [6, 7, 8]. A RAM with R read ports and 1 write port can be implemented with R dual-ported RAMs (replication). Furthermore, W write ports can be implemented by further replicating each of these read-replicas: each read-replica now consists of W dual-ported RAMs instead of just 1 RAM, and control bits associated with each row keep track of which of the W RAMs has the latest-written value for that row. In our context, the complexity of mapping memories to RAMs is increased by three factors:

- Wide superscalar processors require not just multi-ported RAMs, but very highly ported ones (e.g., 12-ported register files, rename map tables, etc.) and very many of them throughout the pipeline. The resource requirement skyrockets as a result, and for wide configurations there are not enough.
- Superscalar processors have several highly ported CAMs as well (e.g., issue queue wakeup CAM). A typical FPGA does not have native CAMs.

- Our FPGA model must support arbitrary superscalar widths, hence, an arbitrary number of ports for each modeled RAM or CAM structure.

The first issue is addressed by using fewer RAMs than required and taking multiple FPGA cycles to complete all the reads and writes that would occur within a single processor cycle [6,7,8]. For the second issue, a CAM can be modeled with a RAM indexed by the value being searched for [9]. An entry in the RAM contains a bit vector indicating which entries in the modeled CAM match the value (*i.e.*, the bit vector is the match vector produced by the modeled CAM). Writing to the CAM involves setting a bit in one bit vector (the bit vector associated with the new value) and clearing a bit in another bit vector (the bit vector associated with the replaced value). The latter requires a second RAM that mirrors the contents of the modeled CAM, to know which value is being replaced. Finally, to support arbitrary superscalar widths, and at the same time provide flexibility in trading off space (resource utilization) and time (extra FPGA cycles), we developed an extensive library of RAM and CAM Verilog modules. All modules provide an abstraction of `Y` rows, `X` bits per row, `R` read ports, and `W` write ports. The library provides multiple variants for each `{R,W}` combination (`Y` and `X` are parameterized). Each variant makes a specific trade-off between resource utilization and FPGA cycles. One variant may use the most resources and fewest FPGA cycles, another variant may use the fewest resources and most FPGA cycles, and there are variants in between these two extremes.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 discusses the relevant related work to this thesis. Chapter 3 elaborates on the first aspect of this modeling effort: creating the configurable RTL model. Chapter 4 elaborates on the second aspect: automatically and efficiently mapping arbitrary superscalar configurations to a single FPGA. Chapter 5 provides an overview of the FPGA-Sim framework including the modules necessary for hardware simulation. Chapter 6 presents our methodology for demonstrating this work. Chapter 7 presents the experiments performed on five processor configurations. The experiments show that FPGA-Sim is orders-of-magnitude faster than C++ and RTL simulations and cycle-accurate with respect to the RTL modeled. Chapter 8 summarizes this thesis and gives areas of future work.

CHAPTER

2

Related Work

FPGA modeling of computer architectures has emerged as a promising alternative to software simulators for accelerating performance modeling. Several FPGA-accelerated simulators have been developed that exploit the inherent parallelism of multi-core systems to explore the performance of these systems quickly. The RAMP Gold project [1, 8] uses FPGAs for full-system simulation of large multiprocessor and many-core systems. The focus of the project is primarily modeling caches, cache coherence, and the interconnection network; therefore, simple in-order cores are used. ProtoFlex [10] applies FPGAs to model the functional execution of multiprocessor systems with up to 1000 nodes. The emphasis of our work is accelerating the simulation of complex superscalar processors and modeling the precise timings of instruction interactions. Because our work focuses primarily on complex processors and RAMP Gold and ProtoFlex focus on the complex interactions between processors, these works complement each other.

FAST [6, 11] proposes a general strategy for accelerating software simulators. The software simulator is divided into functional-only and timing-only models. The former runs on the CPU and the latter is synthesized to an FPGA. This distribution of effort is powerful because it matches the complexity of each model with the strengths of each platform (*i.e.*, full-system simulation of complex instruction sets is better suited for the software model and timing model simulation benefits greatly from FPGA-accelerated systems). New complexity, modeling limitations (accuracy), and performance limitations are introduced due to coordination between the two models. Nevertheless, FAST offers a rich set of modeling tradeoffs. Our approach is in a different regime of FPGA modeling, that relies

on the availability of RTL for diverse cores in a canonical superscalar template–a hardware-only and hardware-literal approach. Running untethered to a software model avoids the complexity of coordinating between different systems but requires more functionality to be implemented in hardware.

A-ports [7] discusses an approach to efficiently synthesize multi-ported memory structures on FPGAs by taking multiple FPGA cycles before advancing the modeled cycle. We leveraged this concept but, to the best of our knowledge, implemented it in a novel way that allows the ratio of FPGA-to-model cycles to be both static and dynamic.

A second approach to synthesize multi-ported memory structures was discussed in [2, 3, 4], that replicates dual-ported FPGA memories to mimic multi-ported memories. By using FPGA primative memories, highly-ported memories consume far fewer resources than using flip-flops. We combined this and the previous approach to construct a library of memories with different numbers of ports that are configurable in their depth and width and resources (more FPGA cycles results in fewer resources).

Other efforts to model performance using FPGAs include Pellauer et al.'s [12] HAsim simulator that divides the simulator into separate functional and timing partitions that interact to form a complete simulator. This approach is similar to the FAST methodology except that both the functional and timing models are implemented on an FPGA. They do this closely coupling the functional and timing models. The functional model provides the data necessary for the timing model to execute each instruction by running a few instructions ahead. HAsim still relies on a software model for rare events such as system calls which is different than our work. Additionally, FPGA-Sim uses only a timing model that performs the tasks of both the HAsim functional and timing models.

Tan et al. propose FAME [1], a comprehensive taxonomy of FPGA modeling techniques along a number of key dimensions. FAME provides a useful framework for discussing the expanding body of work in this area. FAME categorizes FPGA-accelerated simulators into four levels based on three defining implementation characteristics. The first characteristic is whether the model is directly mapped to an FPGA or has decoupled FPGA and model clocks. Decoupling the clocks allows for a more efficient mapping to an FPGA at the cost of additional logic for managing timing information. The second characteristic classifies simulators as being either full RTL or abstract models. Full RTL models are beneficial if the RTL is available and can be mapped to an FPGA and abstract simulators allow the model to be based off of a high-level description of the design. The third characteristic

separates simulators based on if the simulator is single-threaded or multi-threaded. This is different than the model being single- or multi-threaded in that threads refer to model instances and multi-threaded simulators map multiple instances to a single physical FPGA model.

The four levels of FAME are *Direct*, *Decoupled*, *Abstract* and *Multi-threaded*. Direct FAME simulators are directly mapped, full RTL, single-threaded simulators. Decoupled FAME characterizes simulators that are decoupled, full RTL and single-threaded. Abstract FAME simulators include decoupled, abstract, single-threaded models and Multi-threaded FAME includes decoupled, abstract, multi-threaded simulators. FPGA-Sim falls into the Decoupled FAME level because it uses a decoupled clock, uses a full RTL model and maps a single model instance to a physical FPGA model.

There is significant precedent for mapping commercial superscalar processors to FPGAs, achieving fast and cycle-accurate simulation [2, 3, 4]. Each model is for a fixed microarchitecture. While varying some parameters may be straightforward (*e.g.*, structure sizes), others may require significant manual effort (*e.g.*, pipeline stage widths). The goal of our work is to automate FPGA modeling of diverse superscalar cores, including along fundamental dimensions such as width and depth.

Veloce [13] is a product offered by the company Mentor Graphics that provides a debugging and emulation platform. While these products aim at faster simulation for design and verification, they are only the platform to run a design on. FPGA-Sim provides multiple processor configurations that can be run on many platforms, including Veloce.

CHAPTER

3

Configurable RTL Model

Like software simulators, FPGA-Sim must model superscalar processors of arbitrary configuration to provide flexibility to the user. Our approach was to use FabScalar [5] to generate the widest and deepest superscalar processor in its repertoire–a superset core. We then inserted Verilog preprocessor macros within each canonical pipeline stage to be able to narrow it to a desired width and collapse it to the desired degree of pipelining. Because the Verilog preprocessor is used, the RTL model is statically configurable: the footprint of the core is that of the chosen configuration, not the superset core, and there is no extra synthesized logic or configuration bits for this configurability. The preprocessor strips away excess superscalar ways in the front-end pipeline stages, execution lanes in the back-end, etc.

This chapter is organized as follows. Section 3.1 gives a brief background on FabScalar. Section 3.2 describes implementation techniques to transform the FabScalar-generated processors into a statically configurable processor.

3.1 FabScalar Background

We used FabScalar [5] to develop the configurable RTL model of the superscalar processor. The FabScalar tool generates synthesizable RTL of arbitrary cores within a canonical superscalar template. The template is shown in Figure 3.1. To generate a core of a desired configuration, the core generator selects a design for each canonical pipeline stage from a Canonical Pipeline Stage Library (CPSL). The CPSL contains multiple designs for

each canonical pipeline stage that differ in their complexity and degree of sub-pipelining. The complexity of a canonical stage depends on its superscalar width and the sizes of stage-specific structures (RAMs and CAMs). A canonical stage is nominally unpipelined, but it may be pipelined deeper to achieve a higher clock frequency or to accommodate increased complexity in the stage for the same frequency.

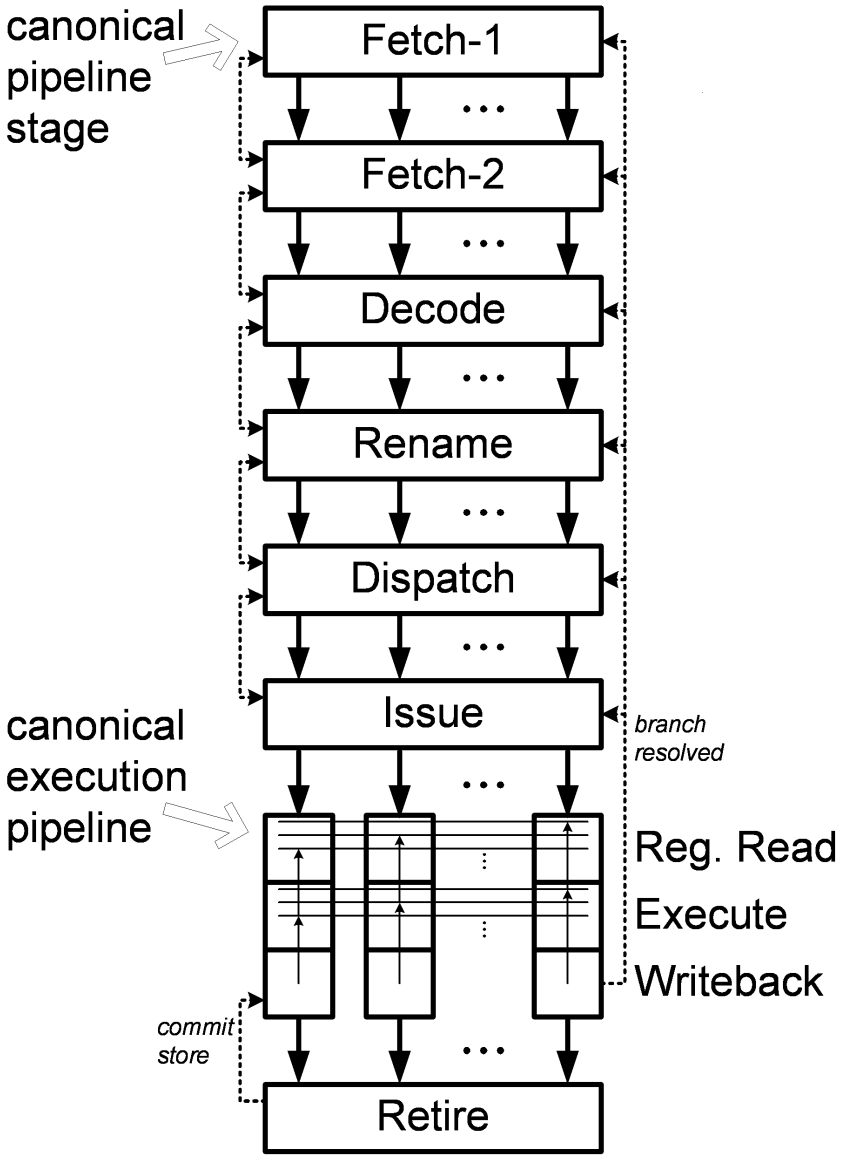


Figure 3.1: Canonical superscalar template

3.2 Statically Configurable Processor

Each of the core configurations that FPGA-Sim models is reproducible without the FPGA-Sim modifications by using the core generator distributed with FabScalar. However, unlike FabScalar, FPGA-Sim relies on a single universal core, which is a superset of all FabScalar cores, as the core generator. The user specifies the parameters of the desired core in a global parameter file and the synthesis tool strips away the unneeded logic. This was achieved by generating the deepest and widest core using FabScalar and then inserting Verilog 2001 preprocessor macros around code to be removed and rewriting simple logic blocks using for loops. An example of how the preprocessor macros (`` `ifdef `` and `` `endif ``) are used is shown in Figure 3.2. Note that only a portion of the `IssueQueue` port declaration is shown for space reasons. The preprocessor macros are used in the port declarations to remove the extra inputs and outputs. To specify the dispatch width of the processor, the user must set `DISPATCH_WIDTH` to the desired width and define the macros `DISPATCH_TWO_WIDE` up to and including the desired width. More information about the global parameters file can be found in Appendix A.

The other standard Verilog construct used for making the core configurable is the for-loop. For-loops used naively in Verilog have the possibility of generating errors or inefficient logic from synthesis. It is safe to use a for-loop to perform the same operation over each element in an array (Figure 3.3). In this example, for-loops are used to concatenate multiple signals into enough signals for the RSR and payload. The user defines `ISSUE_WIDTH` in the global parameters file and the correct number of RSR tags and payload packets are created by synthesis. The alternative is to completely unroll the loops and use preprocessor macros similar to the previous example. However, this approach adds many lines of code, making the file hard to read and prone to errors caused by misspellings.

The benefit of these two approaches is that they both are part of Verilog 2001 and are widely supported by EDA tools. One limitation of Verilog 2001 is that the preprocessor does not support arguments. This feature was introduced in SystemVerilog (Verilog 2005) but is not supported in the EDA tools used in this thesis (Xilinx ISE 10.1). To provide the same functionality, FPGA-Sim includes a Perl script for generating RAM module names that are dependent on user-specified variables. The RAM modules use parameters to control the width and depth but the number of ports cannot be parameterized using Verilog 2001. To distinguish the RAM modules, they follow the naming convention of

```
module IssueQueue(
  input                                    clk,
  input                                    reset,

  input                                    vstall_i,

  /* The number of dispatch Packets and AL IDs is dependent
   * on DISPATCH_WIDTH
   */
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket0_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst0ALid_i,

`ifdef DISPATCH_TWO_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket1_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst1ALid_i,
`endif

`ifdef DISPATCH_THREE_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket2_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst2ALid_i,
`endif

`ifdef DISPATCH_FOUR_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket3_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst3ALid_i,
`endif

`ifdef DISPATCH_FIVE_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket4_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst4ALid_i,
`endif

`ifdef DISPATCH_SIX_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket5_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst5ALid_i,
`endif

`ifdef DISPATCH_SEVEN_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket6_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst6ALid_i,
`endif

`ifdef DISPATCH_EIGHT_WIDE
  input  [`DISPATCH_IQ_PACKET_SIZE-1:0]    dispatchPacket7_i,
  input  [`SIZE_ACTIVELIST_LOG-1:0]        inst7ALid_i,
`endif
```

Figure 3.2: An example of adding configurability with preprocessor macros

```
always @ (*)
begin
  rsrTag_t[0] = rsr0Tag_i;

  /* Create one RSR tag for each issue lane */
  for (i = 1; i < `ISSUE_WIDTH; i = i + 1)
  begin
    rsrTag_t[i] = {rsrTag[i], rsrTagValid[i]};
  end

  /* Create one payload packet and valid signal for each issue lane */
  for (i = 0; i < `ISSUE_WIDTH; i = i + 1)
  begin
    payloadPacket[i] = {payloadGrantedEntry_t[i], payload_t[i]};
    payloadValid[i]  = selectValid[i];
  end
end
```

Figure 3.3: An example of adding configurability with for loops

SRAM_*M*R*N*W where *M* and *N* are the number of read and write ports, respectively. The number of ports is different for each RAM, as discussed previously, and is either dependent on the width of a pipeline stage or is constant. Before synthesis begins, the user executes the Perl script and specifies the fetch width, dispatch width and issue width as arguments. The script replaces the module names with the name appropriate for the configuration. The script is described in more detail in Appendix A.

CHAPTER

4

Efficient FPGA Mapping

The biggest challenge with synthesizing an out-of-order superscalar processor to an FPGA is efficiently mapping its wide data paths and many multi-ported RAM and CAM structures to FPGA resources. An FPGA is comprised of large dual-ported SRAMs called Block RAMs, special function units such as for digital signal processing, and configurable logic blocks (CLB) which are further broken into slices. Figure 4.1 shows a CLB with two slices, each having an input and output for communicating to slices within neighboring CLBs, and access to a switch matrix for global communication. CLBs populate the majority of the FPGA and are arranged in a grid, as shown in Figure 4.2. Slices can communicate directly with the slice above and below through a 1-bit uni-directional path. All other communication must pass through one or more switch matrices. Communication within a slice is fastest but slows down as more hops are needed to reach the slice within another CLB. Figure 4.3 shows the number of hops that it takes to communicate between CLBs for a Virtex-4 and Virtex-5 FPGA. Virtex-5 provides a significant improvement to the number of CLBs within two and three hops but logic should be contained within a slice or one hop away for improved performance.

Slices contain mostly look-up tables (LUTs) and storage elements, the two most abundant FPGA resources. A Xilinx Virtex-5 slice contains four 64x2-bit LUTs and four single-bit registers or latches. In addition, some slices contain resources to support forming multiple LUTs into RAMs, called distributed RAMs, and creating a 32-bit shift register from one LUT. These slices are called SLICEM (Figure 4.4) and the others are called SLICEL (Figure 4.5). These two figures show the LUTs on the left and the flip-flops

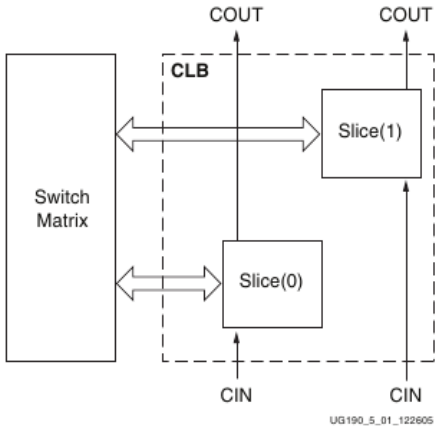


Figure 4.1: CLB Diagram with two slices connected to a switch matrix and connections for adjacent CLBs [14]

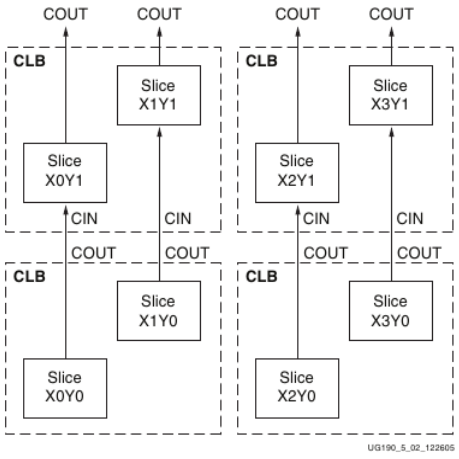


Figure 4.2: Vertical communication between CLBs [14]

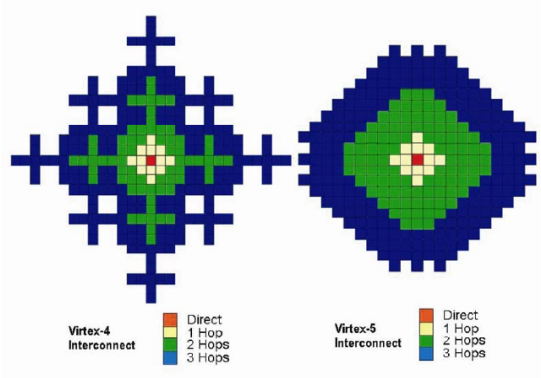


Figure 4.3: FPGA interconnect and the number of hops for Virtex-4 and Virtex-5 [15]

on the right. The LUTs in a SLICEM have the option to be configured as a `DPRAM64/32`, `SPRAM64/32`, `SRL32`, `SRL16`, `LUT`, `RAM` or `ROM` and have extra inputs for writing compared to the LUTs in a SLICEL that can only be configured as a `LUT` or `ROM`. A LUT can be configured as any six-input two-output logic function or two five-input one-output logic functions if the two functions share common inputs. As many as four LUTs can be combined within a slice to generate logic functions with up to eight inputs. If more than eight inputs are required, LUTs from multiple CLBs are used by communicating through the switch matrix. Therefore, logic functions requiring more than eight inputs must span multiple CLBs and suffer with lower performance. With a 32-bit data path, arithmetic functions alone have 64 inputs.

Memories on a Virtex-5 FPGA are limited to single-, dual-, or quad-ported distributed RAMs, or large (18 Kb or 36 Kb) dual-ported SRAM structures in specific locations, called Block RAMs. Table 4.1 shows the combinations of distributed RAMs that can be formed by using one SLICEM and the number of LUTs required by each.Each port can be used for either reading (R), writing (W) or reading and writing (R/W). The port configuration can be either single-ported (S) with one R/W port, dual-ported (D) with two R/W ports, quad-ported (Q) with three R ports and one W port, or simple dual-ported (SDP) with one R port and one W port. The width and depth of the RAM can be increased by using the resources of multiple SLICEM slices. Synthesis tools will map any memory with a combination of ports not listed to flip-flops instead of the more efficient distributed or block RAMs. Table 4.2 shows the resource utilization of a 4R2W (4 read ports, 2 write ports) 32x32-bit register file for a dual-issue superscalar processor,

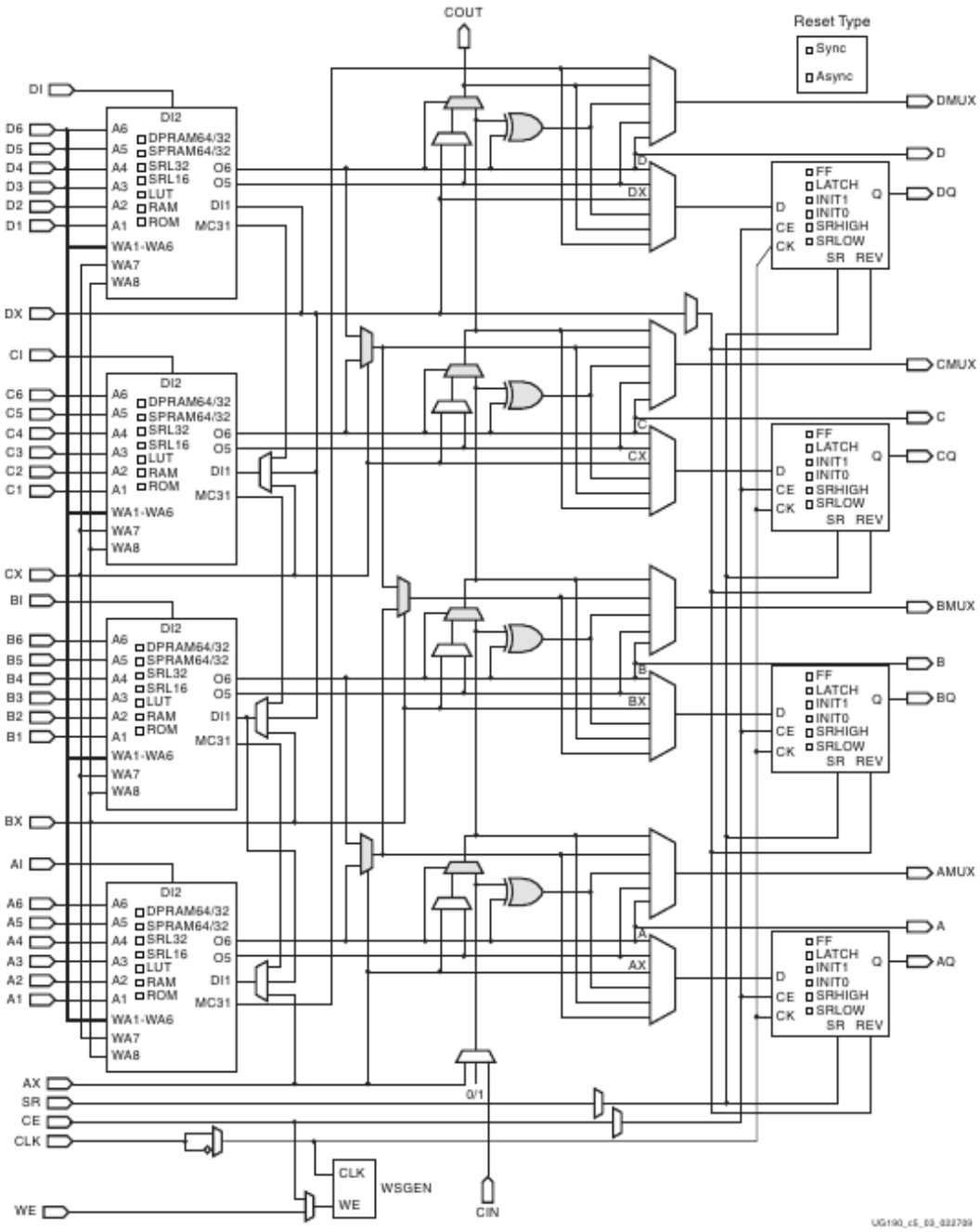


Figure 4.4: SLICEM Diagram [14]

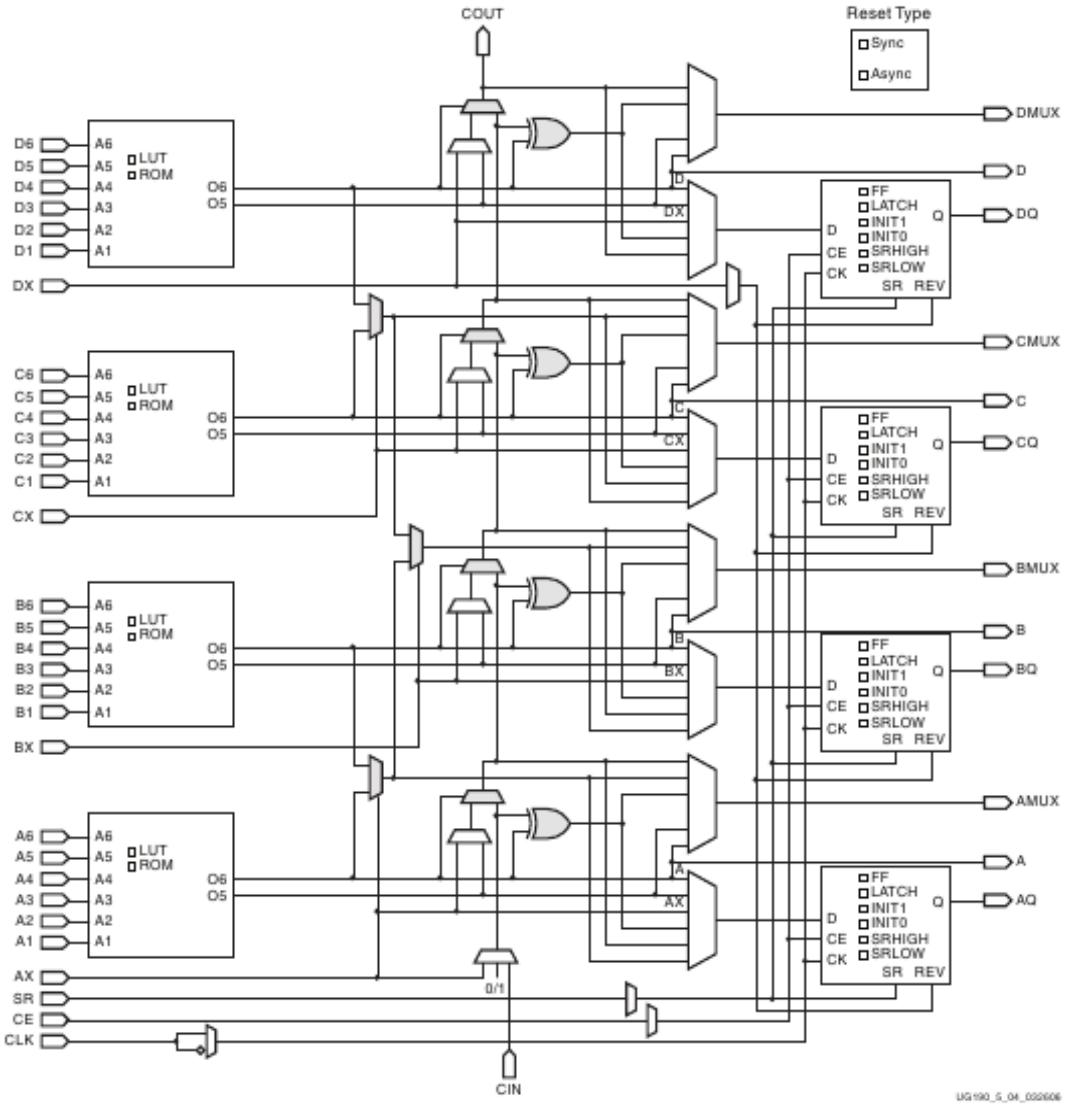


Figure 4.5: SLICEL Diagram [14]

Table 4.1: Distributed RAM configurations

Size	Ports	LUTs
32 x 1	S	1
32 x 1	D	2
32 x 2	Q	4
32 x 6	SDP	4
64 x 1	S	1
64 x 1	D	2
64 x 1	Q	4
64 x 3	SDP	4
128 x 1	S	2
128 x 1	D	4
246 x 1	S	4

Table 4.2: Resource utilization of a 32x32-bit register file with four read and two write ports

Coding Style	LUTs	Flip-Flops	Occupied Slices
Dist. RAM	373	32	94
Flip-Flops	1,882	1,024	472
Difference	5x	32x	5x

when we code the register file for distributed RAMs or use naive coding and leave it up to synthesis. The distributed RAM implementation uses eight 32x32-bit dual-ported RAMs to provide the necessary four read ports and two write ports. (As explained earlier: 4R1W requires four read-replicas, and making these 2W requires two replicas for each read-replica, yielding 4x2 = 8 replicas total.) A total of 180 LUTs are used with each dual-ported RAM consuming roughly 23 LUTs. The naïve implementation lets the synthesis tool decide how to implement the same register file. The entire register file is placed into 1,024 flip-flops and uses multiplexers for both reading from and writing to rows. Because a slice contains only four flip-flops, the register file occupies more than five times the number of slices.

Table 4.3: Memory structures in the superscalar processor and whether explicitly coded for Distributed RAM (d RAM), Block RAM (b RAM), or left up to synthesis

Structure	Read Ports	Write Ports	d / b RAM
L1 Instruction Cache (*2 banks*)	1/bank	1/bank	b RAM
Branch Target Buffer (*F banks*)	1/bank	1/bank	b RAM
Branch Predictor (*2 banks*)	2/bank	1/bank	b RAM
Fetch Queue	D	F	d RAM
Rename Map Table	2D	D	d RAM
Free List	D	R	d RAM
Architectural Map Table	R	R	d RAM
Active List (ROB)	R	D	d RAM
Memory Dependence Predictor	D	1	b RAM
Issue Queue Wakeup CAM	X	D	d RAM
Issue Queue Payload RAM	X	D	d RAM
Physical Register File	2X	X	d RAM
Load Queue CAM	M	M	Left up to synth
Store Queue CAM	M	M	Left up to synth
L1 Data Cache	M	1	b RAM

Out-of-order superscalar processors use a large variety of multi-ported RAMs. Table 4.3 shows the major RAM and CAM structures and the number of ports as a function of fetch width (F), dispatch width (D), issue width (X, the number of execution lanes), the number of load/store execution lanes (M), and retire width (R).

Managing the resources used by each superscalar structure in an efficient manner is a key necessity if larger and wider processors are to fit on a single FPGA. As a stark example, we found that naïvely synthesizing a 2-way superscalar processor with 64 physical registers and a 16-entry issue queue runs out of resources on the Virtex-5 FPGA used in this work. A potential work-around is to partition the design and synthesize it across multiple FPGAs. This approach incurs the design effort of RTL partitioning and extends simulation time (either by decreasing the FPGA frequency or adding FPGA stall cycles) due to frequent off-chip communication among partitioned modules. Moreover, it leads to inefficient utilization of individual FPGAs.

4.1 Clock Decoupling

Decoupling the modeled processor clock from the FPGA clock is vital to both fitting large processors on an FPGA and for mapping complex operations to the general-purpose structures of FPGAs. During each model cycle, multiple FPGA cycles may elapse. The model cycle advances only after all operations for the model cycle have completed, allowing complex operations to be performed efficiently over multiple FPGA cycles but completed entirely within one model cycle. The number of FPGA cycles needed to complete one model cycle is defined as the FPGA-to-model cycle ratio (FMR). Two prior implementations are 1) defining a static FMR for all simulated cycles which minimizes cycle time at the cost of an increased cycle count and 2) dynamically determining the FMR for each simulated cycle which minimizes the cycle count but hurts cycle time. FPGA-Sim introduces a new approach by defining a static FMR for events that occur in every model cycle and allowing infrequent events to delay the model clock even further when necessary.

Explicitly defining a static FMR [16] requires no global management to advance the model cycle because each stage knows exactly when the next model cycle begins by using a counter. The logic is placed locally within a module, eliminating the extra communication between modules that hurts cycle time and the time to perform place-and-route. Large designs benefit from this. However, because every operation must complete within the specified number of FPGA cycles, the FMR must be set to handle the worst case. As the difference between the FMR and the average number of cycles actually needed for the model cycle to complete increases, the performance of the simulator degrades due to more useless FPGA cycles.

Dynamically adjusting the FMR based on the events of a given model cycle minimizes the number of useless FPGA cycles and decreases the total number of FPGA cycles needed to complete the simulation. A centralized controller advances the model cycle once all stages have reported completion of the current cycle. Advancement to the next model cycle happens after the minimum number of FPGA cycles needed to complete the model cycle. Long latency events affect only the model cycles in which they occur instead of delaying every model cycle. This is ideal for an off-chip access to DRAM or a 32-cycle integer division operation which are infrequent. The problem becomes the routing between all of the stages and the centralized controller, increasing cycle time by up to 39% and place-and-route time 20-fold [17]. Complex out-of-order superscalar

processors exacerbate the problem because deep pipelines increase the number of registers to stall, each of the numerous memories needs routing to the controller, and high resource utilization complicates the routing.

FPGA-Sim combines both methods together: a minimum FMR (MFMR) is specified and only the long-latency modules are routed to the controller. Since the memories are accessed every cycle and the number of cycles they require is low, they are designed to complete within the MFMR and do not signal the controller to stall. Each memory minimizes the resources used (to as low as one RAM) for the given MFMR by using time multiplexing and RAM replication when there are insufficient cycles. Since there are many memories across all stages, the total amount of resources used by the design can be substantially lowered by raising the MFMR, allowing large processors to fit on a single FPGA.

To allow long latency events to still complete within one model cycle, a centralized controller stalls the model clock when one of these events has not completed within the MFMR cycles. Only the modules which can have a long latency event have the routing to request the additional time from the controller. In the current implementation, there are three such modules: instruction cache, data cache and the complex instruction arithmetic logic unit (ALU).

4.2 Modeling a Multi-Ported RAM

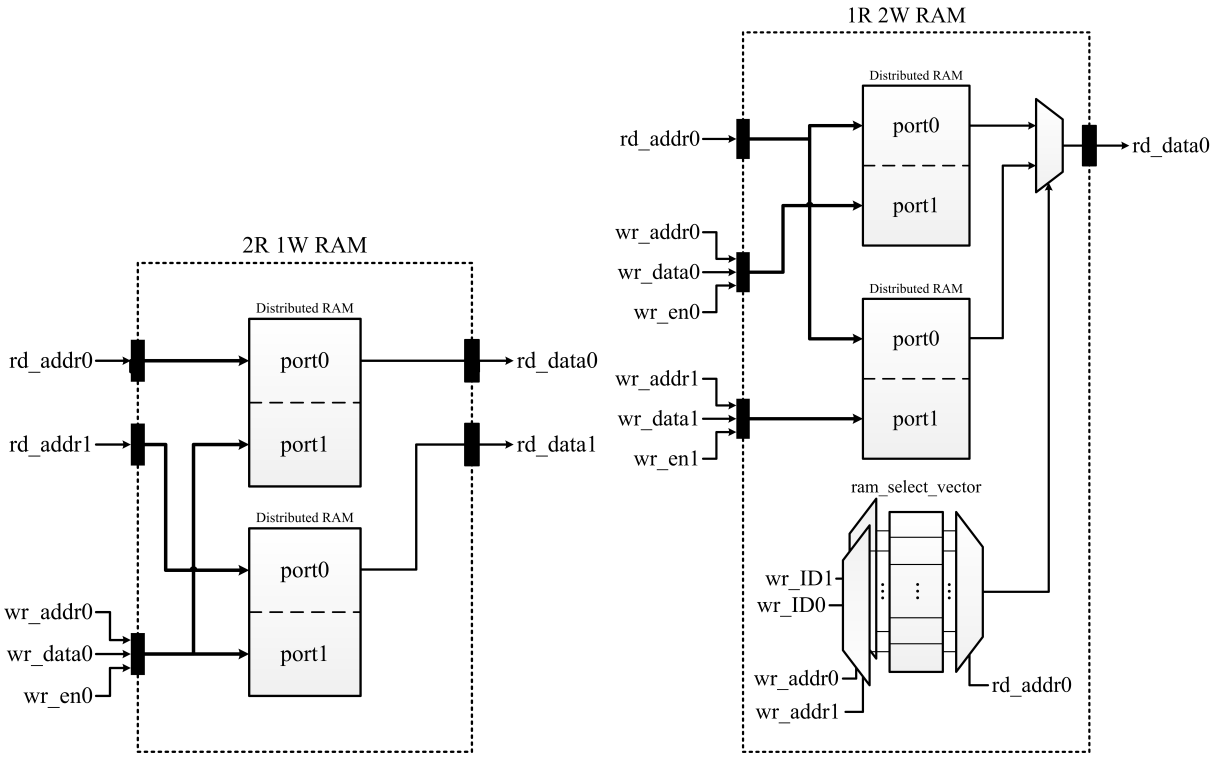
In general, FabScalar RAMs do not synthesize efficiently to the FPGA. They are synthesized to flip-flops, due to having many ports. Therefore, a library of FPGA-aware RAMs was designed to replace FabScalar RAMs. This is important for maintaining the close relationship between FPGA-Sim and FabScalar RTL. The synthesis tool substitutes the FabScalar RAMs with the equivalent FPGA-Sim RAMs without any logic changes made to the pipeline. Each FPGA-Sim RAM, whether using distributed or Block RAMs, provides the correct number of logical ports through RAM replication and time multiplexing. Thus, the number of RAMs used is dependent on three variables: the number of logical read ports, the number of logical write ports and the MFMR.

The first technique to add ports is to use multiple dual-ported RAMs to emulate a multi-ported RAM. Figure 4.6a shows a 2 read (R) and 1 write (W) RAM using two dual-ported distributed RAMs. The two logical read ports map to separate distributed RAMs and the logical write port goes to both so that they receive the same data. Each

read port has a dedicated distributed RAM to access that copy of the data. The cost of each read port is one additional distributed RAM, thus the cost of an M read, 1 write RAM is M distributed RAMs.

Write ports use a similar technique except that the distributed RAMs contain different data and the RAM with the correct value must be maintained. Figure 4.6b shows a 1 read, 2 write RAM including the additional logic required. The wiring to the distributed RAM ports follows the same concept as before, the two logical write ports map to separate RAMs and the logical read port maps to both. When performing a read, two different values are read, the valid value being from the logical write port that last wrote to the address. To manage which value is the most recent for each entry, a vector called `ram_select_vector` stores an ID of the last logical write port that updated each entry. The ID, `wr_ID`, is written to the entry in the `ram_select_vector` that has the same address as the entry in the distributed RAM being written. A read operation uses the ID to determine which value to get from the multiplexer. The `ram_select_vector` introduces complexity: the vector itself is a 1 read, 2 write RAM. However, the width is reduced to $\log_2(\#\ \textit{of write ports})$, reducing the number of flip-flops and multiplexers when the RAM width is large. A 1 read, N write RAM uses N dual-ported RAMs and the additional logic for the `ram_select_vector`, which is based on N and the depth of the RAM.

These two techniques work orthogonal to each other and are used together to increase both the number of read and write ports. As a general rule, MxN dual-ported RAMs are needed for an M read, W write RAM (Figure 4.7). This can be reduced by using time multiplexing which maps multiple logical ports to fewer physical ports over multiple cycles. Figure 4.8a contains a diagram of a 2R 2W RAM implemented with one dual-ported RAM using two cycles, *i.e.,* MFMR is two. The timing diagram is shown in Figure 4.8b. The signal `vstall` is the active low clock enable used to stall the pipeline, *i.e.*, the model clock advances to the next cycle at the rising edge of `clock` if `vstall` is low. Each physical port supports both a read and write operation. During the first cycle, the state machine routes `rd_addr0` to `port0_addr` and `rd_addr1` to `port1_addr`. At the rising edge of `clock`, the data read from each port is latched into a dedicated register and is available to the model. The state machine changes the select signals of the multiplexers in cycle two to allow the writes to happen. When the MFMR is large enough that the memory is implemented with one dual-ported RAM, `ram_select_vector` is removed to reduce resources further. However, as the MFMR is increased and more logical ports are mapped to a physical port,



(a): Adding read ports

(b): Adding write ports

Figure 4.6: Using multiple dual-ported RAMs to implement: (a) two read ports and (b) two write ports

the number of inputs to the multiplexer increases, potentially degrading performance more than an additional RAM would.

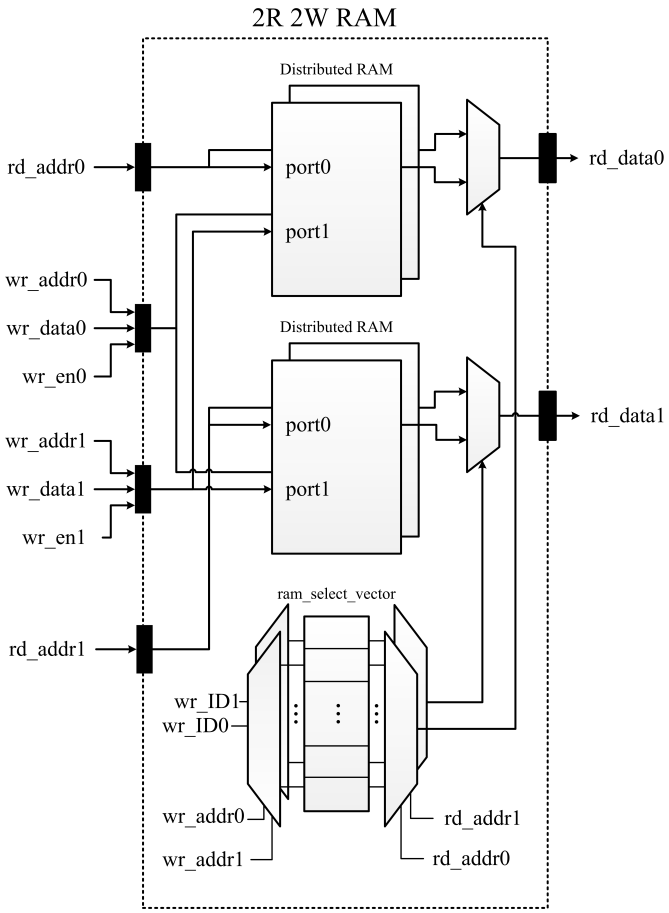
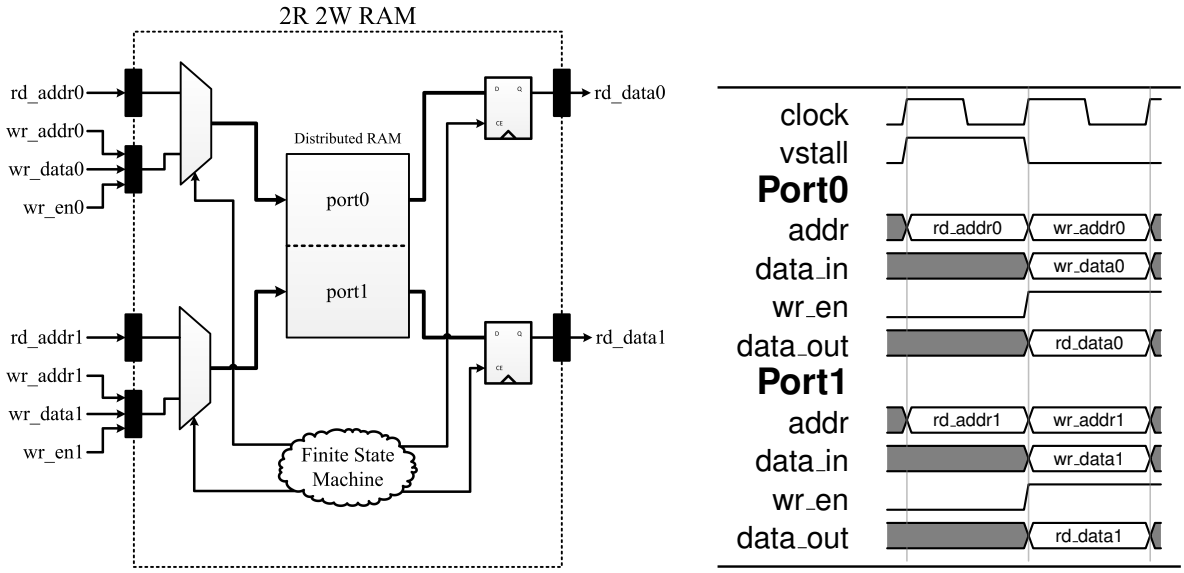


Figure 4.7: Implementing a 2R 2W RAM with RAM replication

Multiplexing the ports reduces the resources used but also improves the FPGA cycle time by placing registers immediately after the read ports of memory structures. FabScalar is designed to use memories with asynchronous read ports, *i.e.,* the data is available in the same cycle that the address is provided. However, time multiplexing requires the data to be latched into registers in order to preserve the data for the entire model cycle. Without the registers, the physical ports could not be shared with multiple logical ports without changing the data read in each FPGA cycle. The added benefit is that timing



(a): Block diagram (b): Timing diagram

Figure 4.8: Implementing a 2R 2W RAM with time multiplexing

paths through memories end at the read port, shortening the logic path.

Since time multiplexing pipelines the reads and writes over multiple FPGA cycles, the situation where the data read from one memory structure is used for calculating an input of another memory structure must be handled correctly. Figure 4.9 shows an example in the commit stage where three RAMs are sequentially accessed in one cycle. When an instruction is committed, `AL_head_ptr0` is used to read `log_dest0` (blue) from the head of the Active List (AL) and `log_dest0` becomes the address into the Architectural Map Table (AMT). In the same cycle, `phys_dest0` (red) is read from the AMT and, at the end of the cycle, is written into the Free List (FL). The timing diagram is shown in Figure 4.10a. The problem that time multiplexing introduces is that `log_dest0` is no longer available in cycle 1 and reaches the AMT in cycle 2 at the earliest. The AMT needs to be aware of this and wait until cycle 2 to begin reading. The timing diagram for the FPGA with an MFMR of four is shown in Figure 4.10b. In cycle 1, `AL_head_ptr0` is used to read `log_dest0` from the AL. Normally, the AMT would read `phys_dest0` in the same cycle but since `log_dest0` is a stale value, the reading is delayed a cycle. At the beginning of cycle 2 `log_dest0` is valid and `phys_dest0` is read from the AMT. In cycle 3, `phys_dest0` becomes available and is written to the FL at the next rising edge

of `clock`. The FL can use cycles 1 and 2 for reading since the read addresses are not dependent on a RAM. A minimum of three cycles is needed to complete this example, thus the MFMR is limited to three or greater.

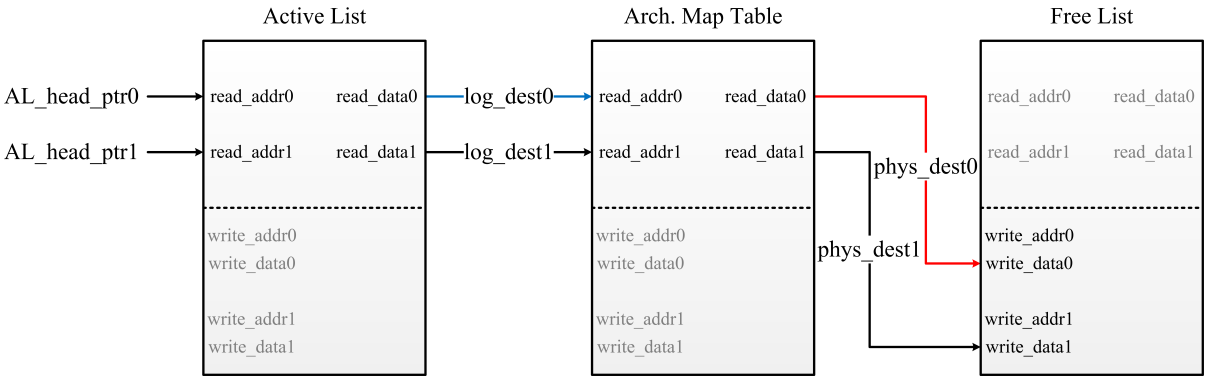
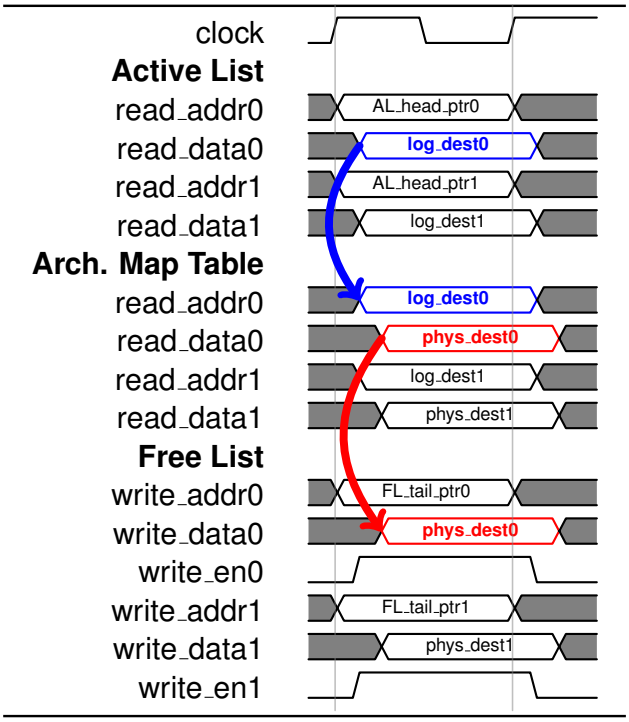
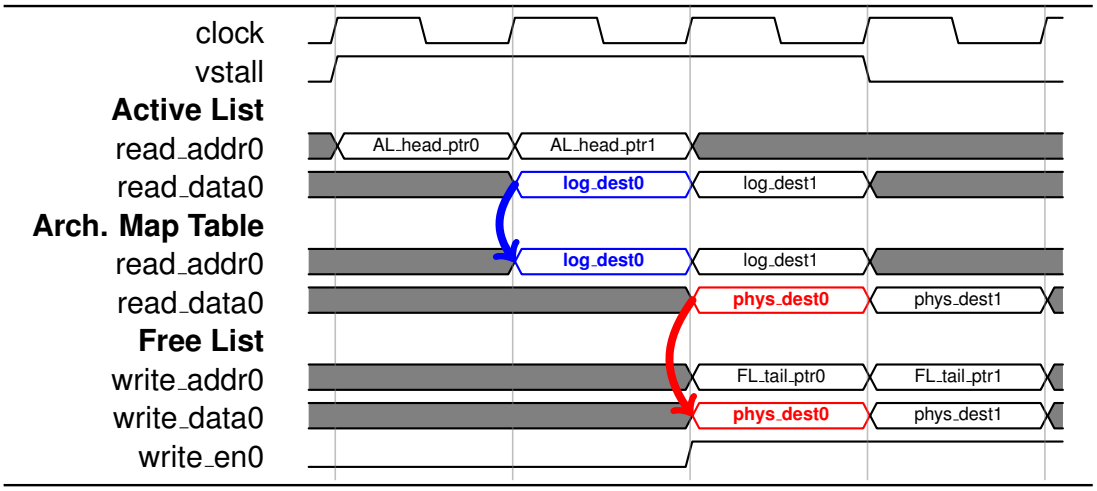


Figure 4.9: Logic path through multiple RAMs

The pervasiveness of multi-ported memories in a superscalar processor means that memories consume a large percentage of the total design. This percentage is reduced by increasing the MFMR because each memory module is designed to use the minimum resources for the possible MFMR values. The MFMR is used as a parameter to specify which design to synthesize. Setting the MFMR high enables large processors to fit on a single FPGA and setting it low increases the simulation speed when smaller processors are modeled.



(a): FabScalar



(b): FPGA-Sim

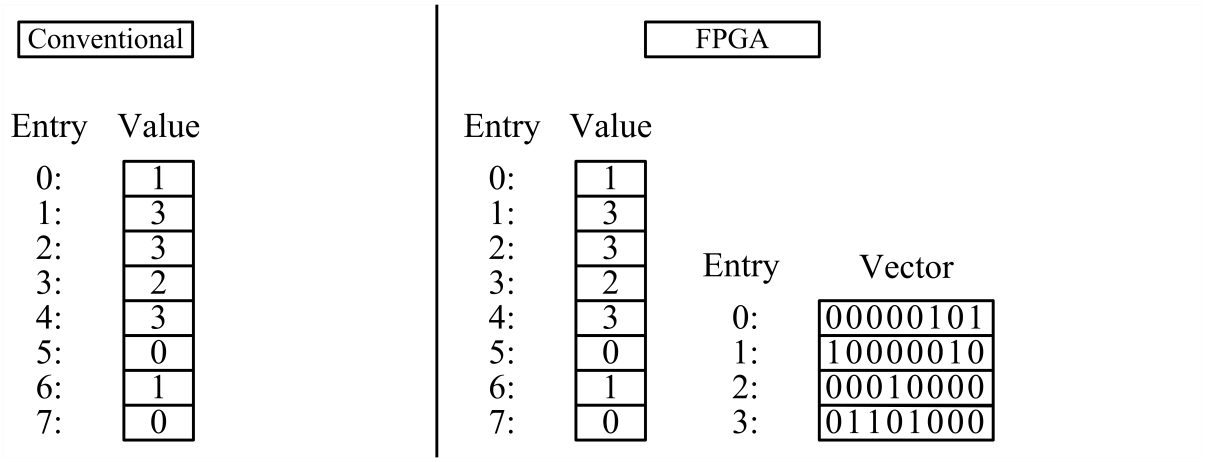
Figure 4.10: Timing diagrams of the AL, AMT and FL

4.3 Modeling a Multi-Ported CAM

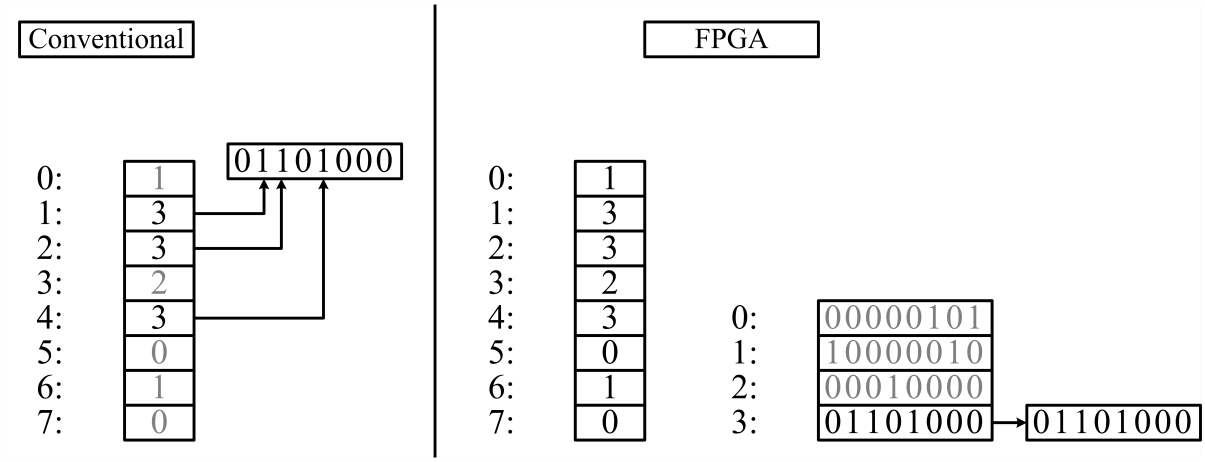
Modeling CAMs presents a new challenge beyond just adding more ports because there is no native CAM structure to replicate. Performing a serial search over multiple cycles is a simple approach, but negatively impacts simulation speed when the contents must be fully searched. The CAMs used in the issue stage search for all matching entries to wake up the consumer instructions. With an issue queue size of 32, each model cycle would need 32 FPGA cycles to perform a serial search. Instead, CAMs are modeled with RAMs that store the match-vectors for the data values [9].

The example in Figure 4.11 demonstrates the operations of a conventional CAM (left) and one modeled on an FPGA (right). The left side contains a conventional CAM with eight entries, each two bits wide. The FPGA implementation requires two structures, a RAM with the same dimensions and contents of the CAM (Value-RAM) and a second RAM with transposed dimensions of the CAM (Vector-RAM) (*i.e.*, the width is equal to the depth of the CAM and the depth is 2^{width}). The Vector-RAM is addressed with the match or write data and it contains bit-vectors of the matching entries for all data values. The initial state is shown in Figure 4.11a. A match operation for the data value 3 (Figure 4.11b) requires a comparison of each entry in the conventional CAM with the value 3 and the output bit of the entry (match line) is set to a zero if the two values are not equal or a one if they are equal. The FPGA implementation uses the value 3 to look up the match-vector in the vector-RAM. Both sides take one cycle to produce the match-vector. The bit cells of a conventional CAM are different than those a RAM, allowing a comparison with all entries in parallel. On the FPGA, the match-vector is updated by the write operation, shifting the complexity to the write operation to improve the time taken to match. The writing of the value 0 to entry 2 is shown for both designs in Figure 4.11c. The conventional CAM overwrites the previous value at entry 2 with 0 and future matches will compare with 0 at that entry instead of 3. To achieve the same effect on the FPGA, the match-vectors for both 3 and 0 are updated. The value-RAM provides the previous value (3) at entry 2 and is then updated with 0. Bit 2 of the match-vectors for the previous value (3) and new value (0) are set to 0 and 1, respectively. Updating a single bit of the match-vectors is simple because distributed RAMs are comprised of multiple 1- and 2-bit LUTs, each having a separate write enable signal.

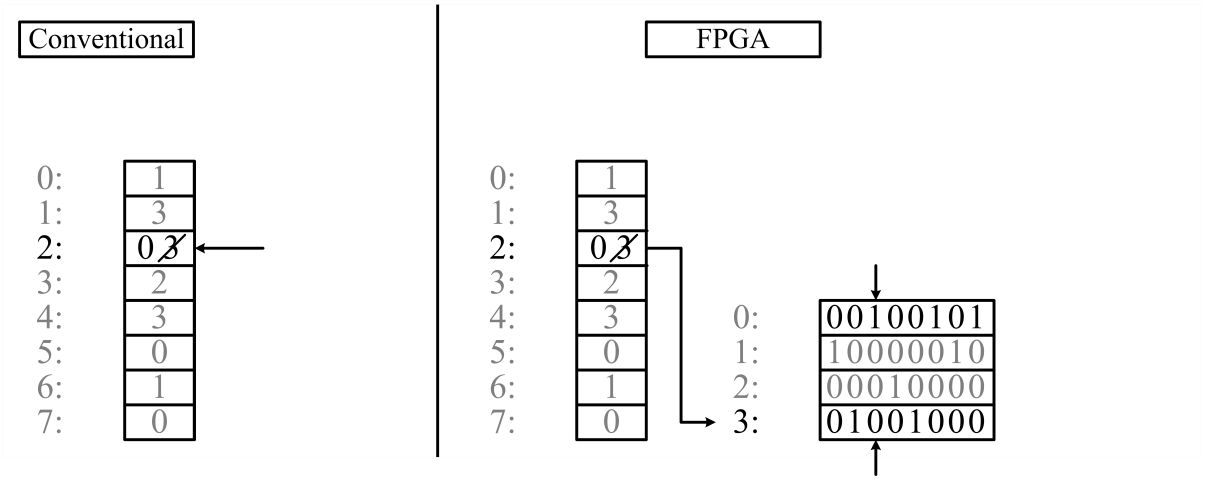
It was discussed previously that the number of match and write ports that the CAMs in the issue stage require is dependent on the issue width and dispatch width. Using



(a): Initial state



(b): Matching value '3'



(c): Writing value '0' at address '2'

Figure 4.11: CAM operations of a conventional CAM and one modeled on an FPGA

the modeled CAM described above, multiple ports are attained through replication and time multiplexing in a similar fashion as RAMs. The distinction is with the write ports since the match-vector contains one bit for each entry, and each entry can be updated by any logical write port. Determining which value is the most recent needs to be done for each bit instead of the whole value. The diagram of a 1 match (M) 2W CAM and the additional logic needed for adding write ports is shown in Figure 4.12. The logic for the second write port is not shown but it is identical to the logic for the first write port. A write requires one cycle for each of the two writes to `Vector_RAM_0`, one write to set a bit in the new value's match-vector and one to clear a bit in the old value's match-vector. The new value's match-vector is accessed during the *write* cycle when `write_data0` is selected as the address of `Vector_RAM_0`. This vector has the bit corresponding to `write_addr0` set to reflect that this bit will match when searching for `write_data0`. The previous value's match-vector is accessed in the *clear* cycle when `previous_data0` is selected. This vector is the match-vector for the data previously written to `write_addr0` and needs the bit corresponding to `write_addr0` cleared. In the first cycle (*write* cycle), the new value's match-vector is updated while `previous_data0` is read from `Value_RAM_0`. Before updating the previous value's match-vector in the second cycle (*clear* cycle), `previous_data0` is compared with `write_data0`. If there is a match then the bit is not cleared since the new match-vector is the same as the previous match-vector: clearing the bit would undo the setting of the bit. Still in cycle 2, `Value_RAM_0` is updated with `write_data0`. A vector called `mask_vector` records the last logical write port that updated each entry, similar to the `ram_select_vector`. When performing a match, the `mask_vector` provides a bit-mask for each physical write port (`mask_vector_0` for write port 0) that is bit-wise ANDed with the match-vector from `Vector_RAM_0`, clearing all bits except those last set by write port 0. The masked match-vectors from all Vector-RAMs are then bit-wise ORed to produce `match_vector0`.

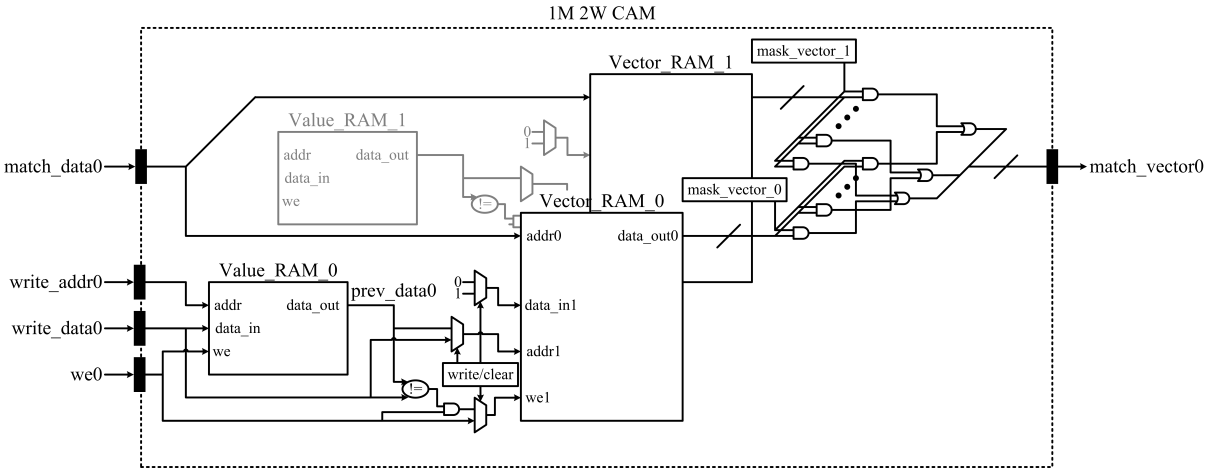


Figure 4.12: Multi-ported CAM modeled on an FPGA

CHAPTER

5

Simulating on an FPGA

This chapter describes the additional modules required to simulate the core on an FPGA. These modules are a substitute for the tasks that a software simulator relies on the operating system for (*e.g.,* communication, memory accesses). Figure 5.1 contains a diagram of the FPGA-Sim framework. The SD Controller (Section 5.1) loads the architectural register and memory state from an SD card at the beginning of the simulation. It interacts with the DRAM interface to load the memory state into DRAM and the pipeline to initialize the physical register file and PC. The DRAM Controller (Section 5.2) provides access to the system DRAM through the DRAM interface. The DRAM interface is the arbiter of requests sent to DRAM from the SD, instruction cache and data cache controllers. The instruction and data cache controllers (Section 5.3) manage on-chip caches composed of Block RAMs to reduce the number of requests sent to DRAM. Time multiplexing is handled by the Time Controller and it interacts with the two cache controllers and the pipeline. Performance counters are managed by the Stats module. Counters can be added to record simulator performance (*e.g.*, the number of elapsed FPGA cycles) or to record model performance (*e.g.*, the number of branch mispredictions). The statistics are sent to the host through a UART interface. The Trace Generator creates traces of committed instructions and sends them to either the host or the Checker Core. The traces are compared by the Checker Core (Section 5.4) which is a simple in-order core that runs the same workload as the model. Modules specific to FPGA-Sim were developed by the author but modules that work with an industry standard (*e.g.*, UART Controller) were obtained from a third party. Table 5.1 lists the modules and where they

were obtained from.

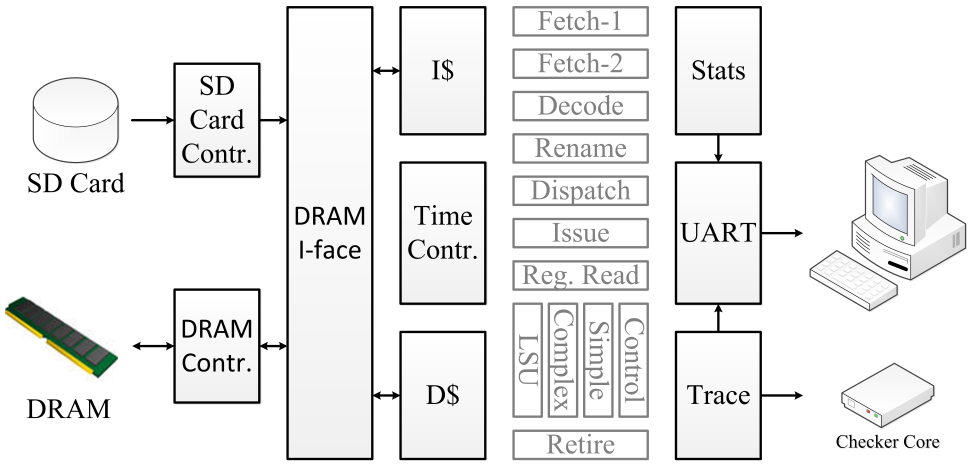


Figure 5.1: FPGA-Sim Framework

Table 5.1: FPGA-Sim Framework Sources

Module	Source
Instruction Cache Controller	Author
Data Cache Controller	Author
Time Controller	Author
DRAM Interface	Author
SD Controller	OpenCores [18]
DRAM Controller	Xilinx MIG [19]
Statistics Controller	Author
Trace Generator	Author
UART Controller	OpenCores [20]
Checker Core	Author

5.1 SD Card Controller

A software simulator initializes the architectural register and memory state of the simulated benchmark to kick-off execution, either from the first instruction of the benchmark or from some checkpoint at mid-execution. FPGA-Sim must similarly kick-off simulation on the FPGA. Programming the FPGA with a bitfile containing the initial architectural register and memory state would require more storage on the FPGA than is currently available and the bitfile would be tied to a single benchmark. Instead, the SD Controller reads the necessary data to initialize the architectural register and memory state from an SD Card prior to starting the simulation. This enables execution to start not only at the beginning of the benchmark, but also from some arbitrary point: any arbitrary region of the benchmark can be simulated by initializing the architectural state with a checkpoint from a previous execution. Controlling the start and stop points of execution allows the fast simulation of characteristic workloads by only executing the regions that represent the characteristic.

The checkpoint containing the program counter (PC), architectural register file contents and memory state is loaded using a state machine after the memory controller has initialized. The SD Card controller reads 512-byte blocks of data at a time, starting with the PC and register file contents. The controller has special access to the PC in the fetch stage and the physical register file in the register read stage for initializing the data. Together, the PC and register file are only 140 bytes (4 and 136, respectively), 372 bytes

are padded at the end to create a 512-byte block.

5.2 DRAM

Running large workloads on the FPGA requires a memory system larger than the storage an FPGA can provide. The system used in this work provides two channels of DDR2 SDRAM and the capacity of each channel is 2 GB. To avoid virtual-to-physical memory address translation, the virtual addresses are used as the physical addresses. Address translation is not required for single-threaded single-programmed workloads and the PISA ISA specifies a 31-bit address space, or 2 GB, which fits within a single channel.

FPGA-Sim accesses the DRAM through a memory controller synthesized to the FPGA. The memory controller was generated using Xilinx Memory Interface Generator (MIG) v2.2 and requires the interface to operate at 133 MHz or higher. To run FPGA-Sim with a slower clock, the memory controller was modified to use FIFOs that were already present to synchronize between FPGA-Sim's clock and a 133 MHz clock. The user accesses the memory by issuing a 3-bit command, 31-bit address and, optionally, the 256-bits of data for a write. For read and write operations, the data is divided into two 128-bit values. Three FIFOs store the address/command, write data and read data to support multiple outstanding requests. Block diagrams of the read and write interface are shown in Figure 5.2 and Figure 5.3, respectively. For both the read data FIFO and write data FIFO, the controller instantiates two FIFO36_72s to achieve the 128-bit data width required. The additional signals indicating the FIFOs are almost full (afull) and `app_wdf_mask_data` are not used by FPGA-Sim because the request FIFOs are large enough to buffer the maximum number of requests generated in a model cycle. The model cycle does not advance until the requests have completed.

The DRAM width is 64-bits and is accessed by the controller sequentially with a burst length (BL) of four. *I.e.,* four 64-bit words starting with a four-word-aligned address is accessed whenever there is a command issued from the FIFOs. Thus, the minimum amount of data accessed is 256 bits. The timing diagrams for the read and write operations are shown in Figure 5.4 and Figure 5.5, respectively. In each figure, two operations are requested with only one address for each operation, the controller generates the additional three to complete the burst. The 3-bit command for reading is `001` and `000` for writing. The latency between issuing a read and when the data is returned is shown and the latencies for multiple requests can overlap.

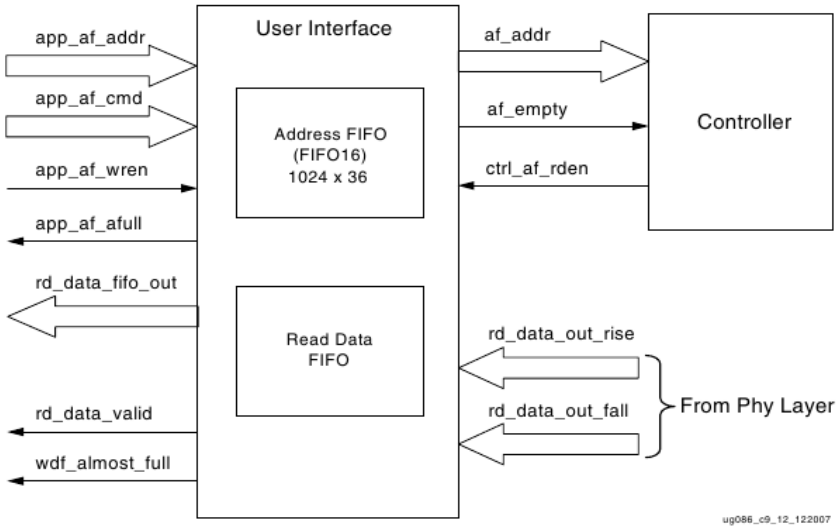


Figure 5.2: User interface for reading from DRAM [19]

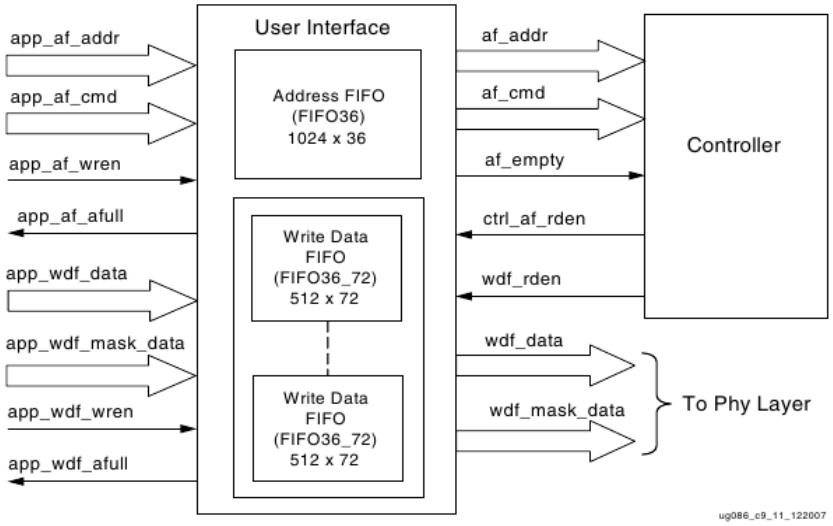


Figure 5.3: User interface for writing to DRAM [19]

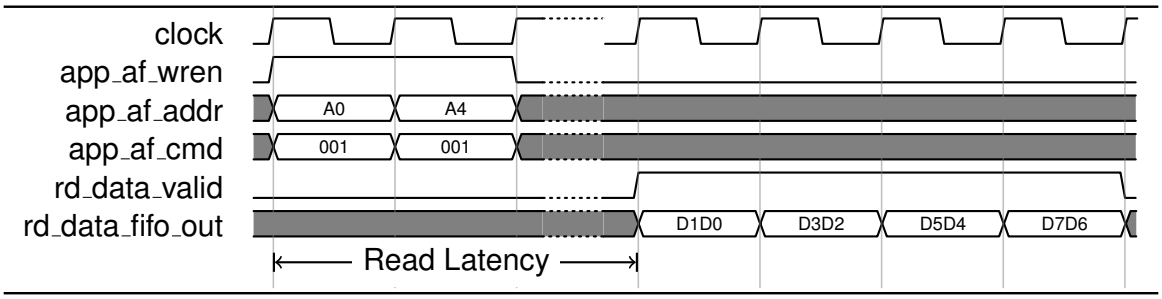


Figure 5.4: Timing diagram of two reads to DRAM (BL = 4) [19]

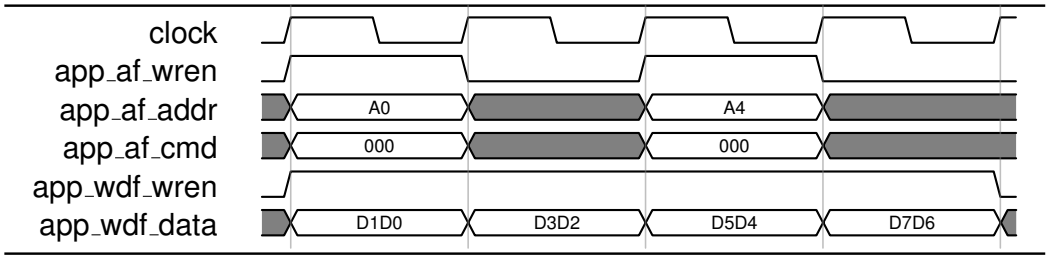


Figure 5.5: Timing diagram of two writes to DRAM (BL = 4) [19]

The `app_af_addr` is for the addresses and is used for both read and write operations. Thus, only one request may be issued to the memory controller per cycle. The DRAM Interface serializes the requests from the SD, instruction cache and data cache controllers.

5.3 Caches

The model cores expect all memory accesses to complete in the same cycle that they are issued, *i.e.*, perfect caches are modeled. FPGA-Sim provides this by stalling the model clock until the requests have completed. The performance of the simulator (not model) is improved with a 32 KB instruction cache and 32 KB data cache so that not every request is sent to the memory controller and DRAM.

Both caches are direct mapped. They are implemented using multiple 36 Kb dual-ported Block RAMs accessed in parallel. The cache block size for both caches is 32 bytes, the size of one access to the memory controller. Cache hits have a two FPGA-cycle latency and the latency of misses depends on the time to access the DRAM.

5.3.1 Instruction Cache

The Block RAMs in the instruction cache are configured in simple dual port (SDP) mode which limits the ports to one read and one write but provides an increased width of 64 bits from 32 bits. Four such Block RAMs are aggregated to construct one instruction cache bank. Four are used to get a width of one cache block (4x64 bits = 256 bits = 32 bytes). The instruction cache is comprised of two such banks, interleaved: the first bank contains blocks with even-addresses and the second bank contains blocks with odd-addresses. Each bank has 512 cache blocks. The 2-way interleaved instruction cache enables fetching up to 8 instructions at a time.

5.3.2 Data Cache

The data cache uses regular dual-ported Block RAMs with two read/write ports and a width of 32 bits. Accessing one cache block requires an array of eight 36 Kb Block RAMs. The data cache has two read ports and two write ports to support a processor load (R), processor store (W), writeback (R) and linefill (W), concurrently. The cache array is replicated to meet the higher port demand of the data cache.

5.4 Checker Core

Implementing new micro-architectural ideas introduces the chance of adding bugs to the design. To speed up the verification and debugging process, an optional checker core runs in parallel with the model core on a separate FPGA and detects bugs in the output of the model core. The checker core is a 5-stage in-order core and performs the same function as the functional simulator when simulating in software. The PCs and results of committing instructions are pushed into a FIFO in program order. The model core sends this trace to the checker core over a bus that connects the two FPGAs. The checker core provides the "golden reference" based on the instructions it commits and compares the instructions from both cores. The checker core signals to the host computer the instruction count, actual values and expected values of the offending instruction when a mismatch is detected. The checker core has been verified by comparing its committed instruction trace with that from the FabScalar functional simulator.

Typically, more information than what is provided by the checker core is needed to

determine the cause and location of a bug. Replacing debugging tools is not the goal of the checker core. Rather, the goal is to verify that the model core is working correctly in the common case and to provide a starting point for debugging. For workloads that run for millions of instructions and take hours to simulate in software, locating the first incorrect committed value is a significant advantage.

CHAPTER

6

Methodology

6.1 Core Configurations

Five different processor configurations that test each of the three major superscalar dimensions were synthesized to a single FPGA. The configurations, their MFMR values and their cycle times are shown in Table 6.1. Core-1 was selected as an "average" core that a user may decide is a good starting point for exploring conceived implementations. Core-2 aims to stress the back-end and resource utilization by issuing up to six instructions per cycle. We test Core-3 as a shallow, narrow core with smaller structure sizes. Core-4 tests the superscalar depth with a three-deep issue stage and four-deep register read stage. Finally, Core-5 stresses the front-end width along with the back-end width. The MFMR range begins with the lowest value that allowed the design to both fit on a single FPGA and finish synthesis in a reasonable amount of time. We stopped increasing the MFMR once all memories were using the minimum resources. A highly-detailed C++ simulator was compared against FPGA-Sim. The C++ simulator provides many of the same configurable parameters and the closest cores were used for the comparisons. The only significant difference was that the C++ simulator did not have a memory dependence predictor so all loads were issued speculatively.

The branch predictor is bi-modal with 64 K entries and the branch target buffer (BTB) contains 4 K entries. The back-end contains four types of functional units: simple, complex, control and memory. For issue widths greater than four, additional simple units are added to fill the width. In this case, cascaded select logic is used to issue multiple

Table 6.1: Core configurations

Parameter	Core-1	Core-2	Core-3	Core-4	Core-5
Fetch Width	4	4	2	4	5
Dispatch Width	4	4	2	4	5
Issue Width	4	6	4	4	5
Active List	128	128	64	128	128
Physical Register File	96	128	64	96	128
Issue Queue	32	32	16	32	32
Load/Store Queue	32	32	16	32	32
Fetch-to-Execute Depth	9	9	9	13	9
Branch Predictor	128 Kb	128 Kb	128 Kb	128 Kb	128 Kb
Branch Target Buffer	120 Kb	120 Kb	120 Kb	120 Kb	120 Kb
Mem. Dependence Pred.	20 Kb	0	20 Kb	0	0
Instruction Cache	perfect	perfect	perfect	perfect	perfect
Data Cache	perfect	perfect	perfect	perfect	perfect
MFMR	3-9	3-11	3-9	3-9	5-11
Cycle Time	20 ns	20 ns	20 ns	20 ns	20 ns

instructions of the same type. Load instructions execute speculatively with respect to prior unknown store addresses. If a store detects a prematurely executed load, the offending load initiates recovery when it reaches the head of the active list.

6.2 Experimental Setup

We performed all of the experiments on a BEE3 FPGA system with four Xilinx Virtex-5 LX155T FPGAs. Each FPGA has a single channel of 2 GB DDR2 DRAM, an SD Card reader, a UART port for communicating with the host and a 72-bit ring bus for communicating between the target core and checker core. The ring bus runs at 200 MHz, the DRAM runs at 133 MHz and the system clock runs at 50 MHz. FPGA-Sim can run on any FPGA system that has similar peripherals; the only platform-specific module is the memory controller. The five cores were synthesized for each MFMR value using Xilinx ISE 10.1. The workload consists of six 100 million instruction SimPoints [21] from the SPEC CPU2000INT benchmark suite. The SimPoints provide regions that are free of floating-point instructions and system calls; currently FPGA-Sim does not support these types of instructions.

CHAPTER

7

Results

This chapter presents the experimental results gathered for FPGA-Sim and the two software simulators. The results show that FPGA-Sim is, in fact, faster than and more accurate than the C++ simulator. Additionally, FPGA-Sim is many orders of magnitude faster than RTL simulations and just as accurate.

7.1 Resource Utilization

One challenge that has held back the simulation of out-of-order superscalar processors on FPGAs is the mapping to a single FPGA. With memories being so pervasive, large processors that would normally require multiple FPGAs can fit on a single FPGA by increasing the MFMR. Figure 7.1 and Figure 7.2 show the LUTs and flip-flops used for each of the five cores. Utilization is shown on the left vertical axis and absolute numbers of LUTs or flip-flops on the right. The maximum capacity on the FPGA of both LUTs and flip-flops is 97,280. The horizontal axis indicates the MFMR values used. Higher MFMR values reduce the resources used by the memories. These numbers represent the resources used by all modules required for running FPGA-Sim, excluding the checker core. Core-4 shows the largest improvement, with 17,511 (39%) fewer LUTs when switching from an MFMR of 3 to 9. Flip-flops also show a noticeable improvement with a maximum reduction of 6,089 (26%), again in Core-4. Flip-flops are used by the memories for storing the data output and also as the `ram_select_vector`. When the MFMR is increased so that all write ports of a memory are multiplexed to the same

RAM, the `ram_select_vector` becomes unnecessary and causes a decrease in the number of flip-flops. Both Core-1 and Core-4 have four-wide fetch, dispatch, issue and commit stages. The difference is that Core-1 has a load violation predictor and Core-4 has a three-deep issue stage compared to two-deep and a four-deep register read stage compared to one-deep. These differences cause Core-4 to use 5,501 (8%) more LUTs and 2,581 (10%) more flip-flops when MFMR is 3. The resource utilization of Core-5 shows that the front-end width has a greater impact than the issue width. Core-5 uses more resources than Core-2 for all values of MFMR except 11, despite having a narrower issue width. Core-3 uses the least resources by a large margin with nearly 30,000 (46%) fewer LUTs and 6,251 (23%) fewer flip-flops than the next smallest core, Core-1. Core-3 benefits from not only having a front-end half as wide as Core-1 but nearly every memory is half of the size as well. The small memories already use few resources and increasing the MFMR provides minimal benefit.

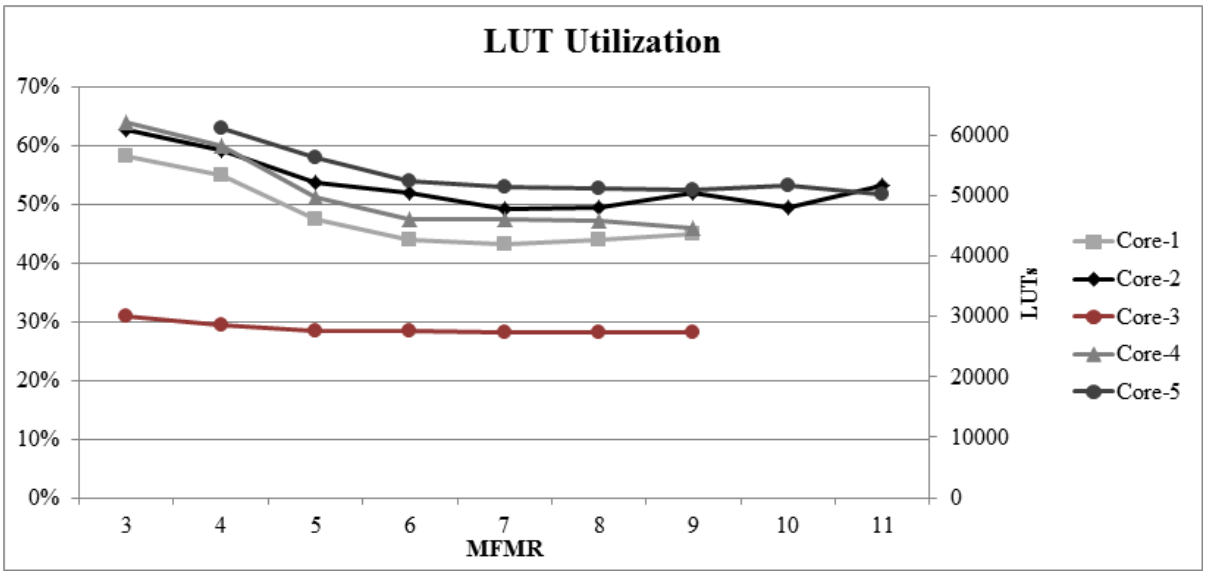


Figure 7.1: LUT utilization

The wider cores have a knee in the curve of LUTs at MFMR 7. As highly-ported memories perform more time multiplexing, the number of logical ports that must be multiplexed to a single physical port increases, requiring more inputs to the multiplexors

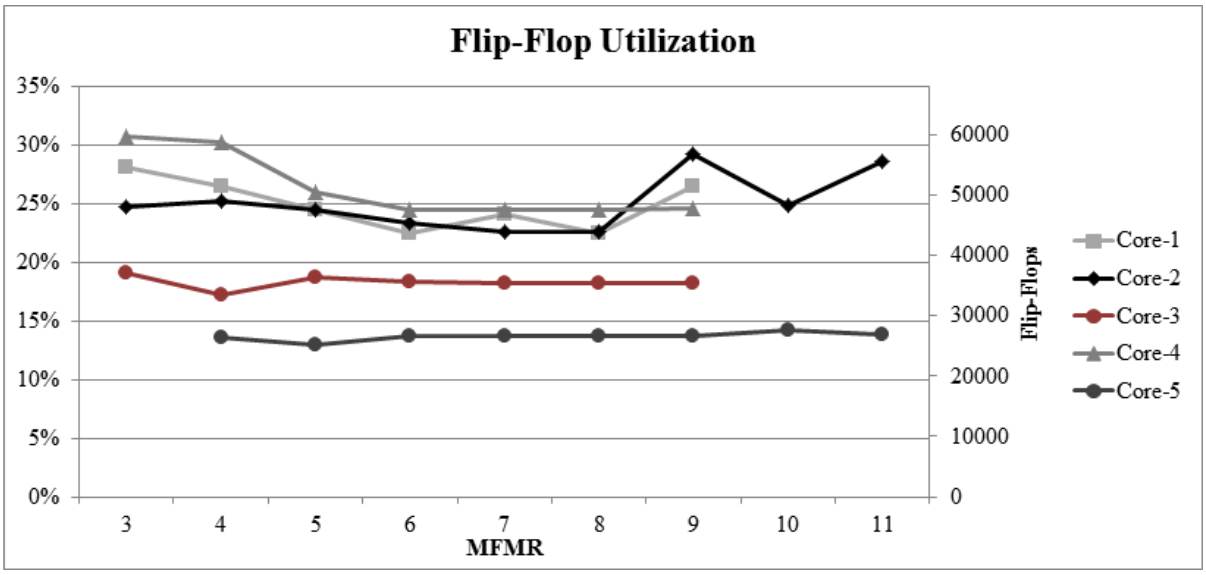


Figure 7.2: Flip-Flop utilization

that act against the benefits of using fewer structures. Because of this effect, setting the MFMR to an arbitrarily large value will not always provide the smallest design. For the designs studied here, increasing the MFMR beyond seven shows little improvement.

Increasing the MFMR is a powerful optimization that can reduce designs enough to fit onto an FPGA or to downgrade to a smaller FPGA, obviating the need to put effort into partitioning the design across multiple FPGAs and avoiding the associated communication. Figure 7.3 and Figure 7.4 show the same data, but also show the LUT and flip-flop capacities of various Xilinx FPGAs, respectively. The FPGAs listed are part of Xilinx's Virtex-5, Virtex-6 and Virtex-7 family of FPGAs and their per unit prices[1] at the time of this writing are shown with the model number, when available. The LUT and flip-flop capacities were calculated by multiplying the number of slices by the number of LUT and flip-flops per slice, respectively. The figures show how FPGAs have followed Moore's Law to make simulations of complex out-of-order superscalar processors on a single FPGA a practical endeavor. The maximum number of LUTs and flip-flops was used by Core-4 with an MFMR of 3 (62,070 LUTs and 29,904 flip-flops). Table 7.1 shows the percentages that this core would require if using a different FPGA. On the largest

[1]The prices were taken from www.avnet.com on October 26, 2011. For each model, the lowest priced unit was used.

FPGA available (Virtex-7 2000T), Core-4 would use up just over 5% of the LUTs and just over 1% of the flip-flops. With the remarkable capacities of FPGAs available today, larger and more complex cores can be simulated by using minimal effort to manage the resource usages.

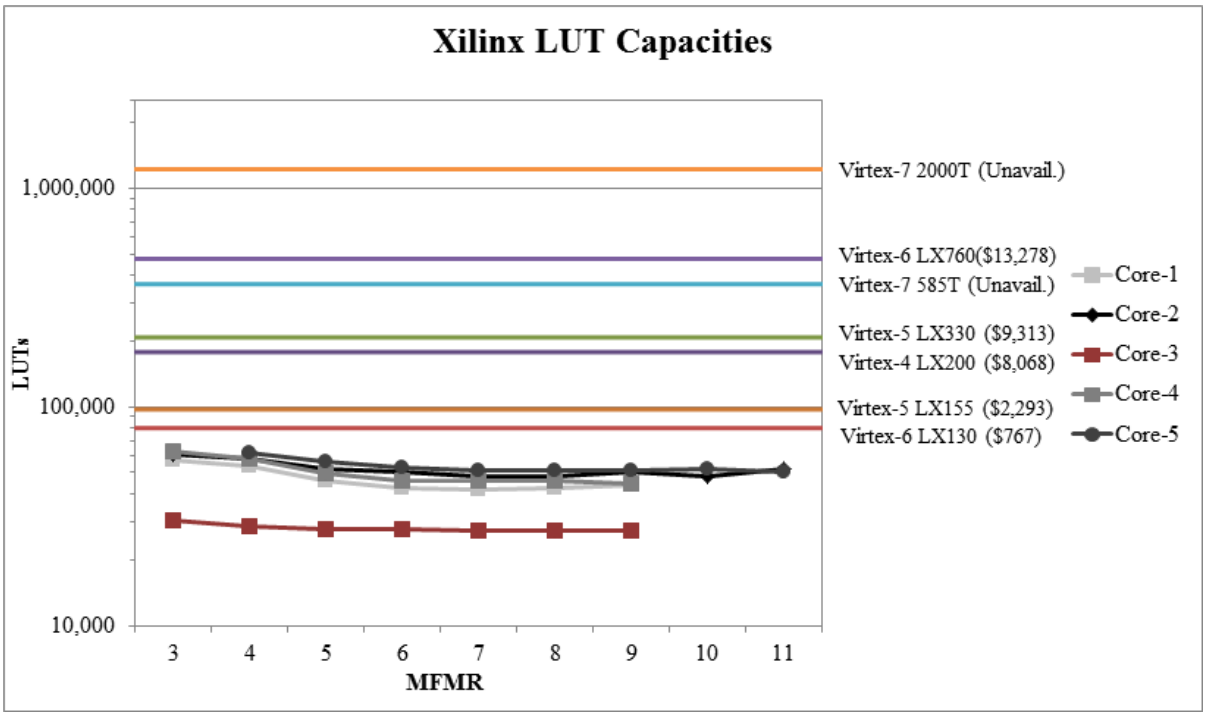


Figure 7.3: Estimated LUT utilization on various FPGAs

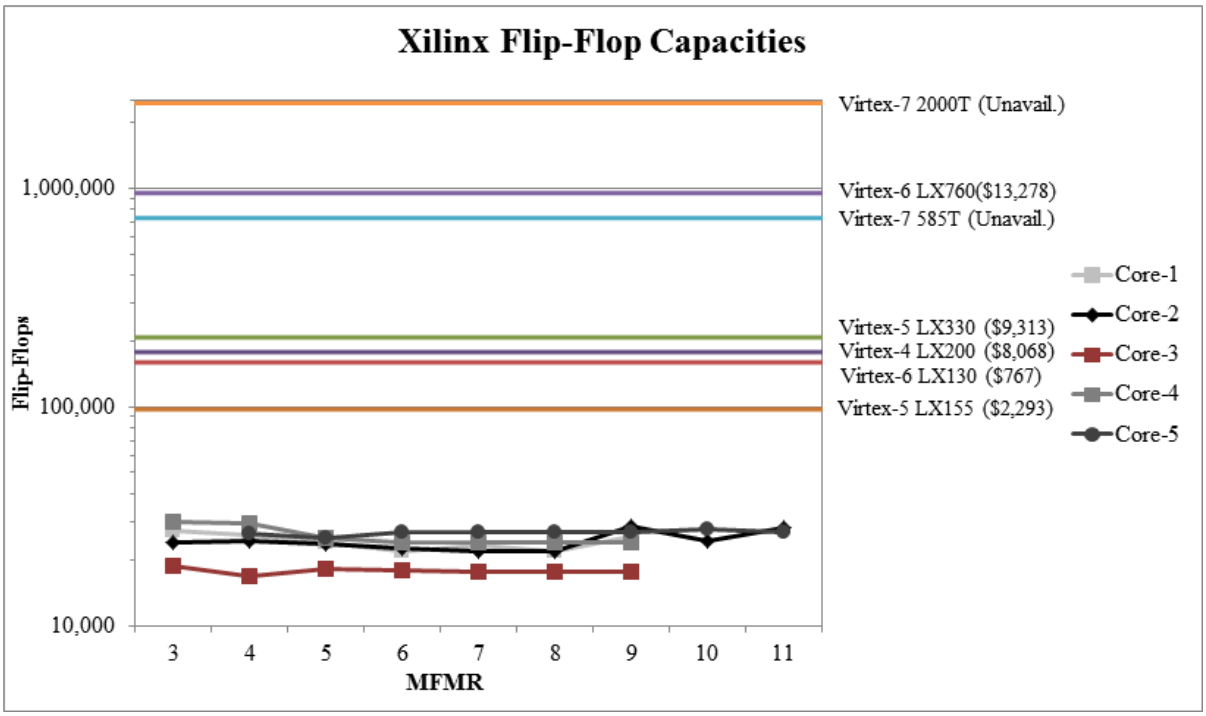


Figure 7.4: Estimated flip-flop utilization on various FPGAs

Table 7.1: Capacities of various FPGAs and the estimated utilization of Core-4

	LUTs		Flip-Flops	
FPGA	Capacity	Utilization	Capacity	Utilization
Virtex-5 XC5VLX155	97,280	63.8%	97,280	30.7%
Virtex-5 XC5VLX330	207,360	29.9%	207,360	14.4%
Virtex-6 XC6VLX130	80,000	77.6%	160,000	18.7%
Virtex-6 XC6VLX760	474,240	13.1%	948,480	3.2%
Virtex-7 XC7V585T	364,200	17%	728,400	4.1%
Virtex-7 XC7V2000T	1,221,600	5.1%	2,443,200	1.2%

Table 7.2: Block RAM usage and Synthesis Times

	Block RAMs		Synthesis Time		
Core	Total	Utilization	Min.	Max.	Avg.
Core-1	44	21%	2h 29m	5h 28m	4h 12m
Core-2	43	20%	4h 08m	7h 32m	5h 53m
Core-3	44	21%	1h 10m	1h 32m	1h 18m
Core-4	43	20%	3h 37m	6h 58m	4h 47m
Core-5	45	21%	4h 10m	5h 13m	4h 40m

7.2 Speed against C++, RTL

Fast simulations are important for being able to make informed design decisions based on many iterations. In fact, one of the main goals of FPGA-Sim is to provide fast simulations when even a C++ simulator is not fast enough. We measured the speed of FPGA-Sim as the number of model cycles simulated per second, and compared against a C++ timing simulator that models the same FabScalar-produced processors and RTL simulations of the same FabScalar core. The MFMR is five for all FPGA-Sim cores. Although cores one through four successfully run when MFMR is as low as three, Core-5 fails to meet the same timing constraint when MFMR is less than five. So, these results are pessimistic for cores one through four. Figure 7.5 shows the model cycles per second using a logarithmic scale on the vertical axis and the workloads simulated on the horizontal axis with the harmonic mean for all workloads on the far right. RTL simulations perform the worst because of the great detail being simulated. Moreover, the parallelism of event-based simulations is too fine-grained to exploit effectively. C++ achieves almost two orders of magnitude more model cycles per second than RTL. Both are software simulators but the advantage that C++ has is the abstraction used when modeling the core. The improvement gained by going from the RTL simulator to FPGA-Sim is between three and four orders of magnitude.

An interesting note is that all cores simulate at nearly the same rate with FPGA-Sim for a given workload. There is variability only when looking across the workloads because the FMR is dependent on the MFMR and long latency events (complex arithmetic, cache misses). The MFMR is constant across the cores and long latency events are primarily dependent on the workload. On the other hand, both of the software simulators perform

better when simulating Core-3, the smallest core.

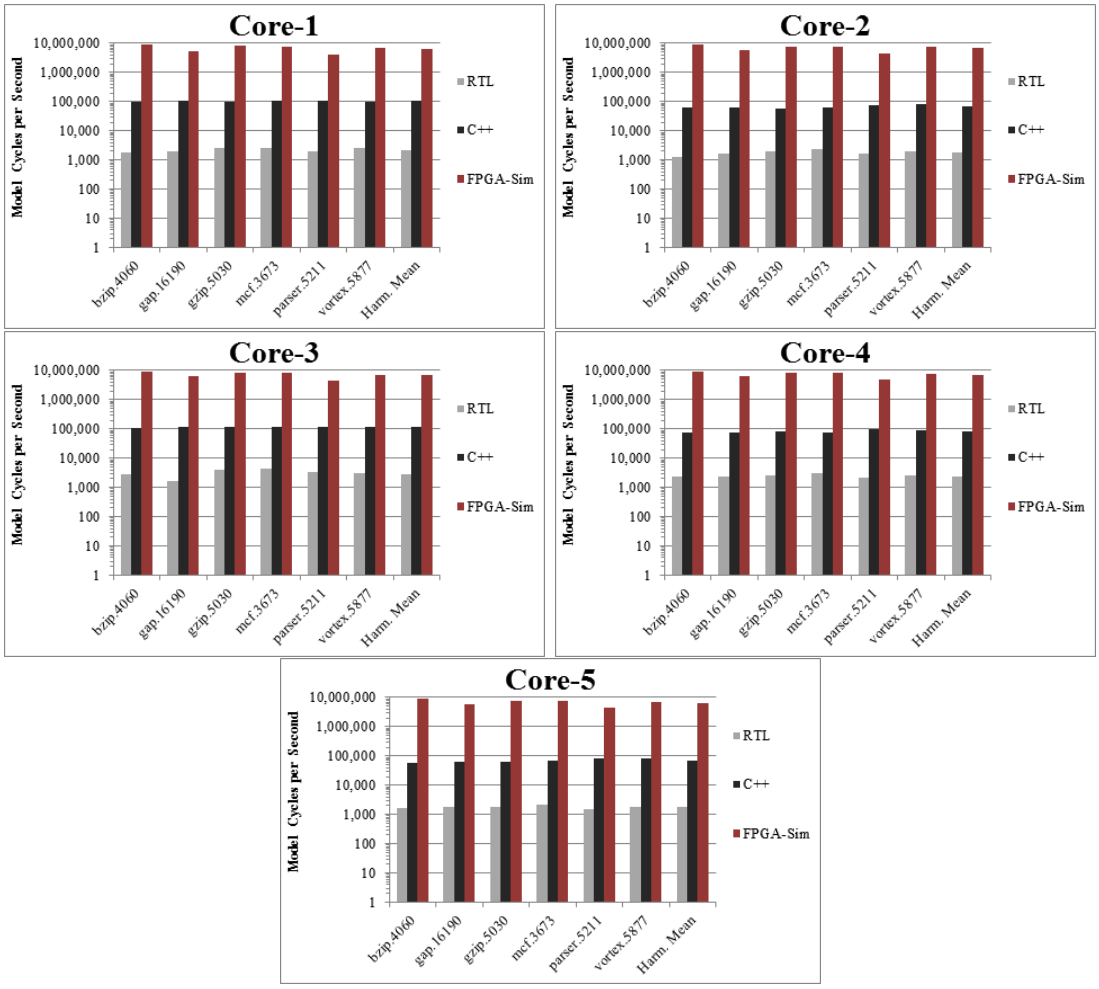


Figure 7.5: Simulated model cycles per second

The model cycles per second of FPGA-Sim was compared against C++ to show that there is a significant performance increase when using an FPGA as a simulation platform, even when compared to a lower-fidelity representation of the core. The speedup of FPGA-Sim over the C++ simulator (Figure 7.6) ranges between 37x for Core-3 running *parser.5211* and 154x for Core-5 running *bzip.4060*. As before, the cycles per second were measured when executing with MFMR equal to five. Core-2 consistently has the highest speedup over all of the workloads except *bzip.4060* because the C++ (and RTL)

simulator must do more work per cycle with this core and for large cores in general. For the same reason, the speedup of cores one and three are the lowest because they are the smallest cores. The average speedups for each of the workloads are shown in the figure and listed in Table 7.3 for clarity. The workload with the highest average speeup for all of the cores is *bzip.4060* with an average of 121x while the lowest is *parser.5211* with an average of 47x. Overall, FPGA-Sim performs 87x better, on average, than the C++ simulator for all of the cores and workloads.

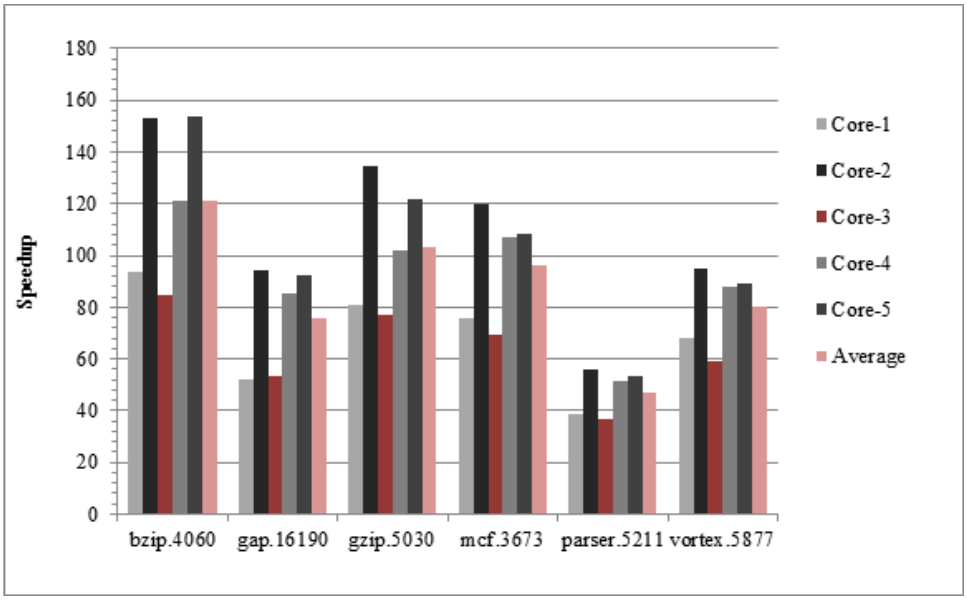


Figure 7.6: Speedup of FPGA-Sim over C++

Figure 7.7 shows the effective clock frequency as the MFMR is increased. The effective clock frequency is calculated by dividing the FPGA clock frequency by the FMR of the simulation. The plots are arranged by workload to highlight the point that the performance is largely dependent on the workload, not the core being simulated. *Bzip.4060* performs the best, reaching a peak effective frequency of 15.1 MHz with cores three and four. *Parser.5211* performs the worst, only getting as high as 5.8 Mhz with Core-4. Workloads that perform well with low MFMRs have the largest performance degradation as the MFMR is increased. This is because they are not frequently stalling for long-latency events. For example, the effective frequency of *bzip.4060* running on Core-2 is reduced by

Table 7.3: Average speedup over C++

Workload	Average Speedup
bzip.4060	121x
gap.16190	76x
gzip.5030	103x
mcf.3673	96x
parser.5211	47x
vortex.5877	80x
All	**87x**

45% when the MFMR is doubled from 3 to 6, 47% between 4 and 8, and 48% between 5 and 10. The frequency is reduced by nearly half each time the number of FPGA cycles afforded to memories is doubled. Workloads with many stalls for long-latency events perform worse than other workloads when the MFMR is low, but are not affected to the same extent by raising the MFMR. The effective frequency for Core-4 running *parser.5211* is reduced only by 18% when the MFMR is doubled from 3 to 6, 22% between 4 and 8, and 25% between 5 and 10. Increasing the MFMR from 3 to 11 lowers the frequency by 69% and 37% for bzip.4060 and *parser.5211*, respectively, on Core-4. An interesting exception is the relatively wide range of frequencies for *gzip.5030* when the MFMR is 3. Core-3 runs at 13 MHz and Core-2 runs at 10.8 Mhz. The difference is because of mispredicted branches causing pollution in the instruction cache. Core-3 resolves branches quicker because it has a smaller instruction window so the number of incorrect instructions fetched is fewer. Although the cause is micro-architectural differences, the caches are part of the simulator.

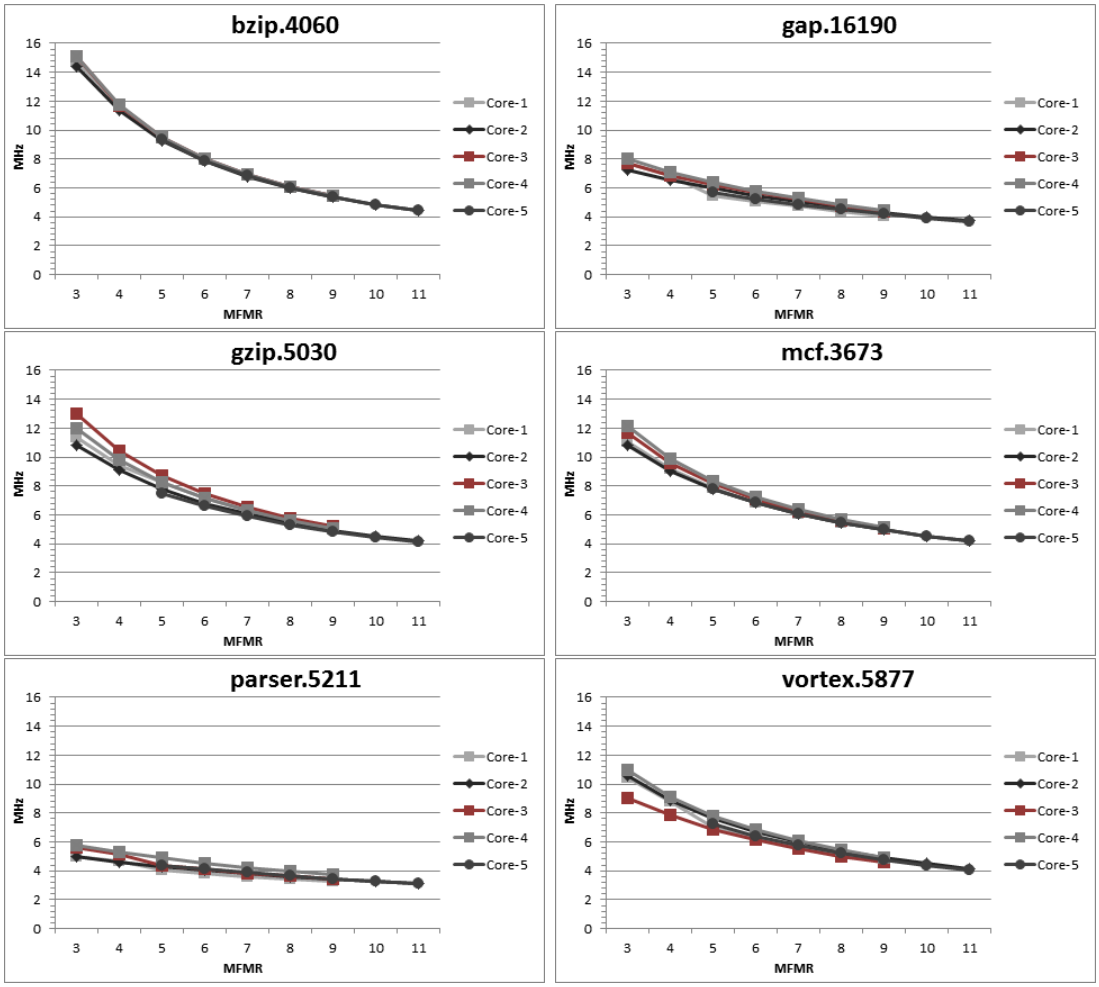


Figure 7.7: The effective frequencies

7.3 Accuracy

Three separate avenues are taken to verify the correctness of FPGA-Sim: matching the instructions per cycle (IPC) with the RTL simulation, matching on the checker core, and comparing the instruction trace with a software functional simulator. Producing the same IPC as the RTL simulation of the model core helps to verify the timing of the modeled core since a bug in the design will likely increase of decrease the cycle count. Figure 7.8 shows the IPCs for the different cores and workloads. The C++ simulator was configured as close to the RTL cores as the simulator allowed; however, there were still some large differences. FPGA-Sim matches the IPCs of the RTL cores exactly for 29 out of the 30 core+workload combinations and within 1% for the last combination. FPGA-Sim can be reliably used for any performance or sensitivity studies. The one workload that does not match (Core-1, *vortex.5877*) is currently being resolved. The C++ simulator performs well for some cases but is overall unreliable. Its IPCs match the RTL within 5% for 9 combinations. Its IPCs are off by more than 10% for 15 combinations, however.

Simulating with the checker core running in parallel provides additional assurance that each committed instruction is correct. This method is the fastest of the three since the work is done entirely on FPGAs and the only communication with the host is if and when there is an error. The third and most definitive step to verifying that the simulation is correct is to capture the instruction trace of the model core and compare it with the trace generated from the functional simulator used by FabScalar to ensure correctness. The PC and result of each committed instruction is sent to the host through a UART connection. The speed of host communication severely delays the simulation, making this method unreasonable for typical use. Table 7.4 lists the cores verified and the number in parentheses next to the core name is the MFMR value that was used. FPGA-Sim traces match exactly with the FabScalar functional simulator. Simulation was terminated arbitrarily at the instruction counts shown, due to time constraints; again, host communication is unreasonably slow.

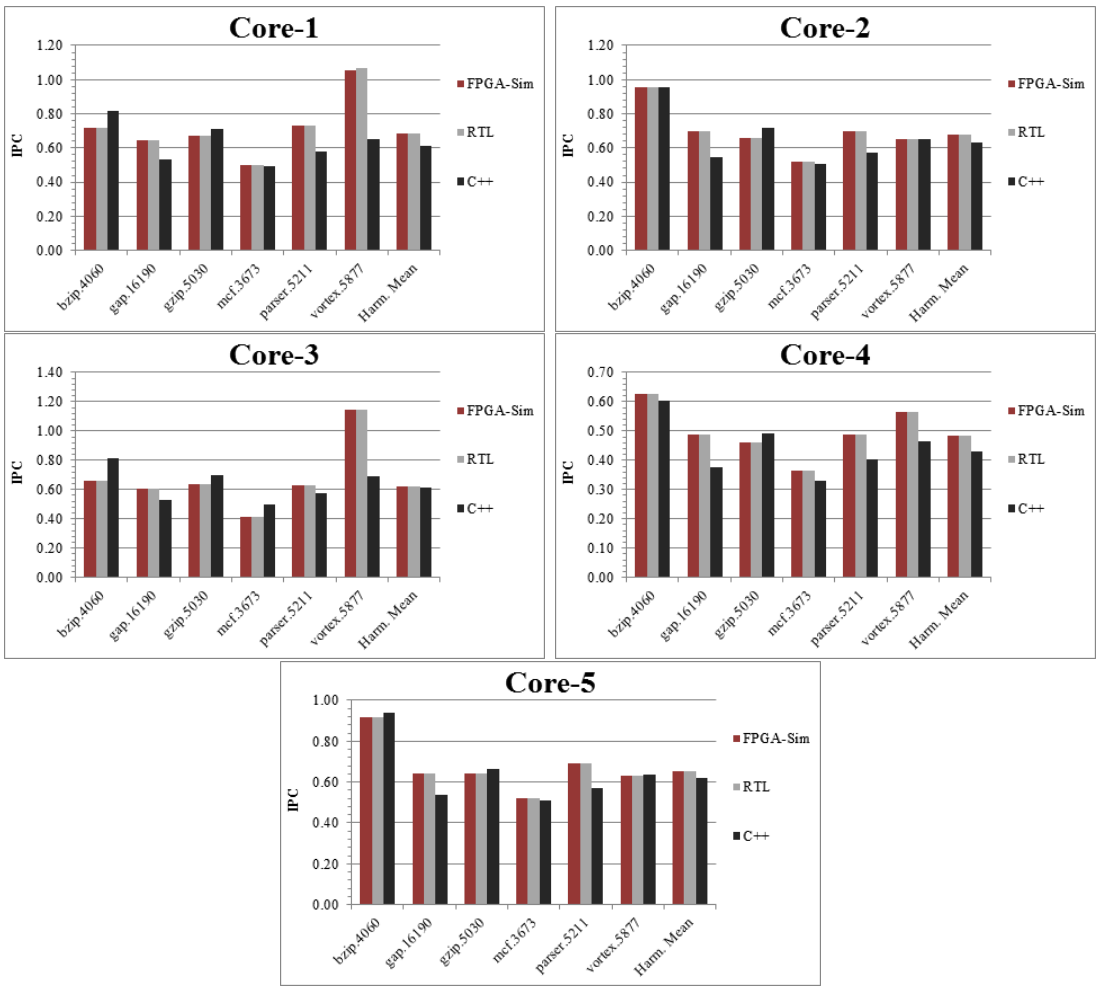


Figure 7.8: IPCs of FPGA-Sim, RTL and C++

Table 7.4: Number of instructions verified

Workload	Core-1 (6)	Core-2 (9)	Core-3 (9)	Core-4 (9)	Core-5 (9)
bzip.4060	19,017,398	14,174,876	10,616,918	17,365,606	13,413,091
gap.16190	10,626,418	28,392,187	13,136,862	10,711,432	24,679,765
gzip.5030	18,810,483	15,485,820	13,826,438	19,967,558	13,508,432
mcf.3673	11,700,649	34,271,783	35,679,441	35,661,629	10,367,709
parser.5211	25,000,549	23,702,156	11,396,448	22,145,545	19,883,183
vortex.5877	22,241,639	19,066,336	10,111,174	19,240,150	10,049,283

7.4 Analysis of FPGA cycles

Two factors determine the number of model cycles that FPGA-Sim simulates per second: the MFMR and the additional number of stall cycles caused by long-latency events. The maximum performance is limited by the chosen MFMR since each model cycle must use at least that many FPGA cycles. The long-latency events reduce the performance further depending on how many cycles they took to complete. A higher FPGA cycle count can be offset by reducing the cycle time; a possibility in the cases where increasing the MFMR reduces resource utilization. However, determining the minimum cycle time for each design would require multiple synthesis attempts and would be a hurdle to our goal of quickly simulating diverse cores. Figure 7.9, Figure 7.10, Figure 7.11, Figure 7.12, Figure 7.13 and Figure 7.14 show a breakdown of the contribution to FPGA cycles from the long-latency events for *bzip.4060*, *gap.16190*, *gzip.5030*, *mcf.3673*, *parser.5211* and *vortex.5877*, respectively. These are all cycles beyond the MFMR that delayed the simulation from advancing to the next model cycle. The events are: a multiplication instruction (6 cycles), a division/modulus instruction (32 cycles), a cache hit (3 cycles), a cache miss (22 cycles on average to access DRAM) or stalling for the checker core to catch up. When multiple events happen simultaneously, the single counter increased is, in descending priority: I-Cache, D-Cache, complex unit, checker core. I-Cache was given priority over D-Cache because requests from the I-Cache are serviced before the D-Cache requests. Only the breakdown for cores two and three are shown. The cycles attributed to the MFMR are not shown because they comprise the majority and always increase linearly as the MFMR is increased. Since these are omitted and only cycles that stall beyond the MFMR are counted, the number of cycles shown decreases as the MFMR increases even though the total number of cycles increases. Each workload exhibits a different mix of events. *Bzip.4060* is dominated by misses in the data cache. In fact, the cycles spent handling data cache misses is between 98.7% and 99.9% for both cores. Compared to Core-2, Core-3 has far fewer cycles caused by the instruction cache , hence far fewer total cycles, for *gzip.5030*. It was discussed previously that Core-3 had fewer instruction cache misses caused by pollution from mispredicted branches. The plots for *vortex.5877* show that Core-3 has 17% fewer cycles for all events and a larger percentage of complex cycles.

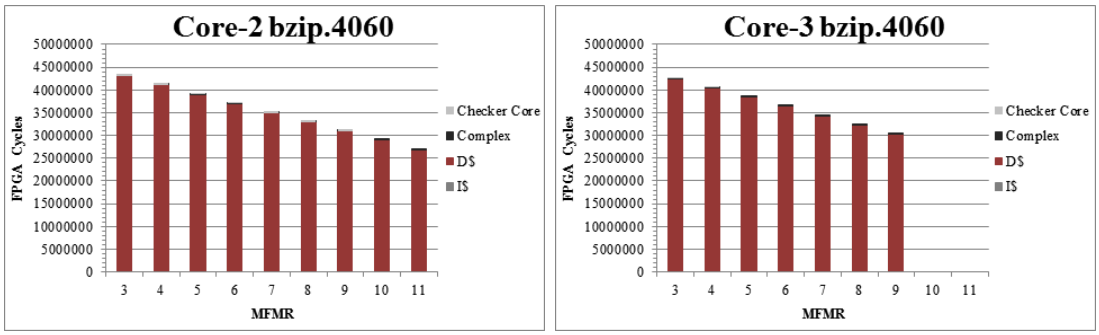


Figure 7.9: Breakdown of contributors to the FMR for bzip.4060

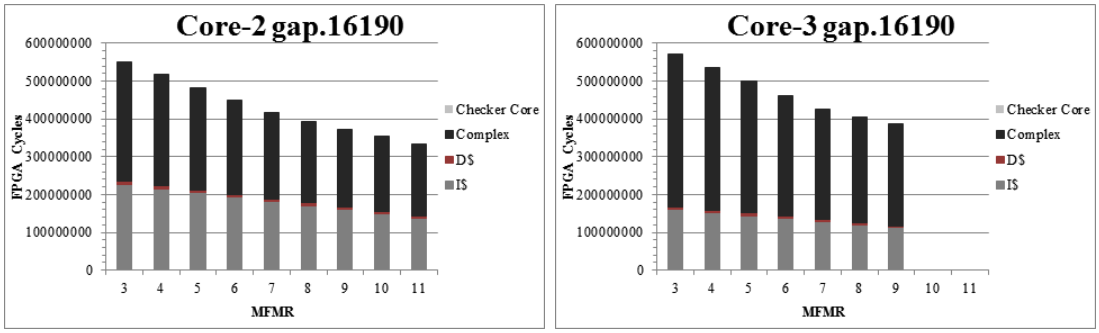


Figure 7.10: Breakdown of contributors to the FMR for gap.16190

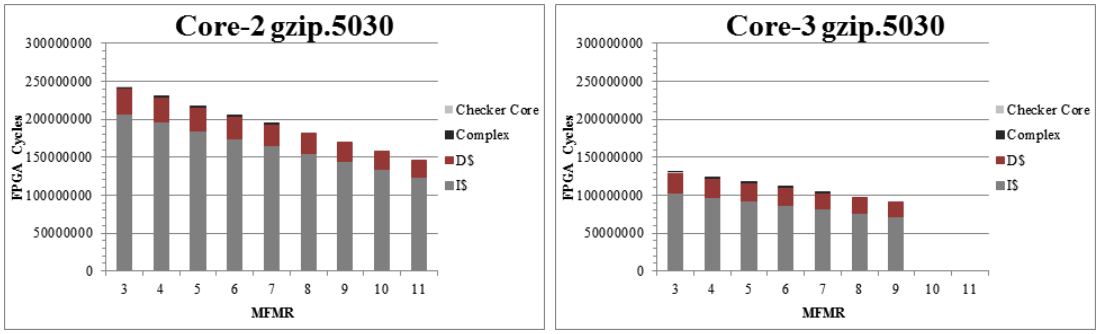


Figure 7.11: Breakdown of contributors to the FMR for gzip.5030

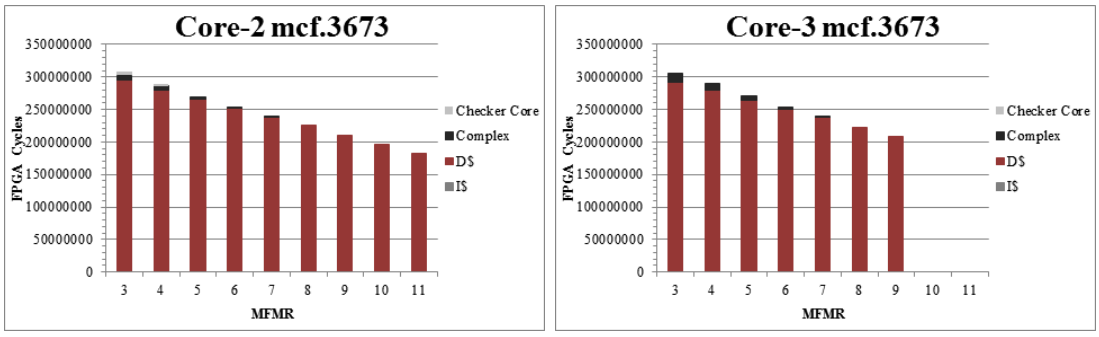


Figure 7.12: Breakdown of contributors to the FMR for mcf.3673

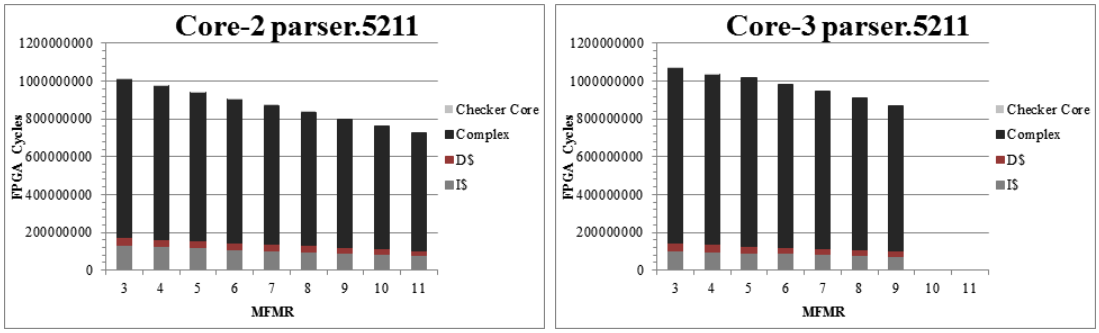


Figure 7.13: Breakdown of contributors to the FMR for parser.5211

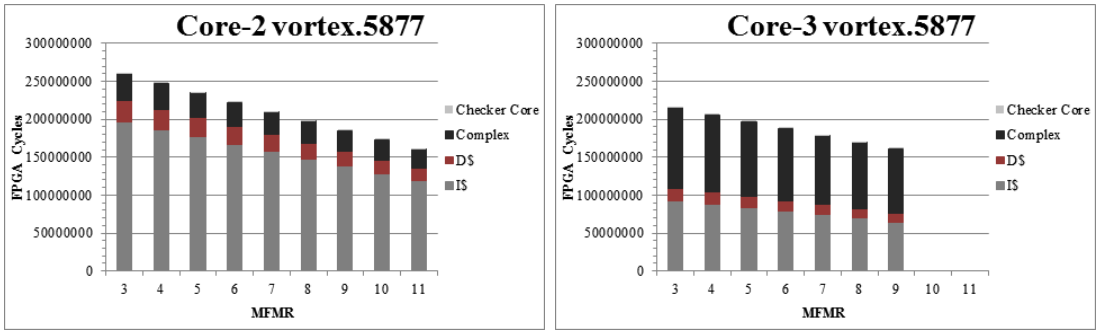


Figure 7.14: Breakdown of contributors to the FMR for vortex.5877

CHAPTER

8

Summary and Future Work

This work explored an exciting trend that is gaining traction in the computer architecture community, which is to use FPGAs as cost-effective hardware-accelerated simulators. The continued scaling of FPGA capacities has brought them into the spotlight as a worthy alternative to the flexible, well-understood, yet, somewhat inaccurate and slow method of using software simulators for architecture research. This work presented FPGA-Sim, a configurable and FPGA-synthesizable simulator that models the RTL designs of diverse out-of-order superscalar processors. This thesis demonstrates that, although following this trend requires more initial effort, it has been completed by one graduate student.

The challenges facing newcomers were addressed and techniques for overcoming them were presented. Among the challenges is mapping the highly-ported RAMs and CAMs found in out-of-order superscalar processors efficiently by using the generic look-up tables (LUTs) and other FPGA primitives. Implementations were presented and examined for using dual-ported RAMs to emulate a memory structure with far more ports. The methods were: RAM replication and time multiplexing. By adopting these implementations, future effort can be focused in other areas that advance FPGA-based simulation.

Hardware implementations were also presented for software-leveraged tasks. Decoupling the FPGA and model clocks provides the same necessary flexibility of software simulators and a new technique that allows both a static and dynamic number of FPGA cycles to complete one model cycle was presented. Operations typically inefficient or impossible for an FPGA are easily performed in a single cycle by virtualizing time for the modeled core.

The work was compared against the two common types of simulators to quantify the benefits of an FPGA-accelerated simulator. FPGA-Sim outperformed both the RTL simulator and a highly-detailed C++ simulator by up to 7,014x and 154x, respectively, when comparing the simulated model cycles per second. Furthermore, the IPCs reported by the C++ simulator differed by as much as 40% from the core being modeled whereas FPGA-Sim was 100% accurate for all but 1 simulation, and even that was 99% accurate. To demonstrate that the capacity of FPGAs is no longer an insurmountable hurdle for simulating complex designs, the resource utilization of all cores, both large and small, were compared with the capacities of FPGAs available today and in the near future. The largest core fell under 3% utilization of a future-announced FPGA.

FPGA-Sim is a ready-to-use simulator currently being used for fast RTL simulations. Extending the usefulness is one ongoing goal for this work and includes adding support for floating-point instructions and system calls. With these two features, simulating longer and more diverse workloads will benefit from the flexibility, speed and accuracy of FPGA-Sim.

References

[1] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic̀, and D. Patterson, "A Case for FAME: FPGA Architecture Model Execution," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ACM, 2010, pp. 290–301. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815999

[2] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh, "An FPGA-based Pentium® in a Complete Desktop System," in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. ACM, 2007, pp. 53–59. [Online]. Available: http://doi.acm.org/10.1145/1216919.1216927

[3] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang, "Intel® Atom™ Processor Core Made FPGA-Synthesizable," in *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2009, pp. 209–218. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508160

[4] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel® Nehalem Processor Core Made FPGA Synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2010, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723116

[5] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template," in *Proceeding of the 38th Annual International Symposium on Computer Architecture*. ACM, 2011, pp. 11–22. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000067

[6] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007, pp. 249–261.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Ports: an Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. ACM, 2008, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344685

[8] T. Zhangxi, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovič, "RAMP Gold: An FPGA-Based Architecture Simulator for Multiprocessors," in *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC)*, June 2010, pp. 463–468.

[9] J. L. Brelet, "Using Block RAM for High Performance Read/Write CAMs," Xilinx App. Note XAPP204, Tech. Rep., 2000. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp204.%pdf

[10] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, pp. 15:1–15:32, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534925

[11] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST Methodology for High-Speed SoC/Computer Simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2007, pp. 295–302.

[12] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2008, pp. 1–10.

[13] [Online]. Available: http://www.mentor.com/

[14] (2010) Virtex-5 FPGA User Guide. [Online]. Available: www.xilinx.com/support/documentation/user_guides/ug190.pdf

[15] (2006) Virtex-5 Platform FPGA Family Technical Backgrounder. [Online]. Available: www.xilinx.com/company/press/kits/v5/v5backgrounder.pdf

[16] M. M. Denneau, "The Yorktown Simulation Engine," in *19th Conference on Design Automation*, June 1982, pp. 55–59.

[17] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, pp. 16:1–16:26, September 2009. [Online]. Available: http://doi.acm.org/10.1145/1575774.1575775

[18] [Online]. Available: http://opencores.org/project,spimaster

[19] (2010) Memory Interface Solutions. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf%

[20] [Online]. Available: http://opencores.org/

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/605397.605403

APPENDIX

APPENDIX

A

Configuring a Core

Steps to configure the various pipeline widths, depths and structure sizes before synthesis or simulation.

1. In the top level source directory, edit FabScalarParam.v as follows:

   - Set `STALL_CYCLES` to the desired MFMR minus 1. *E.g.*, to have an MFMR of 5, set `STALL_CYCLES` to 4.
   - Set `FETCH_WIDTH` and `FETCH_WIDTH_LOG` appropriately.
   - Comment/uncomment `FETCH_TWO_WIDE`, `FETCH_THREE_WIDE`, etc., up to and including the value set for `FETCH_WIDTH`. *E.g.*, if `FETCH_WIDTH` was set to 4, `FETCH_TWO_WIDE`, `FETCH_THREE_WIDE` and `FETCH_FOUR_WIDE` should be uncommented and `FETCH_FIVE_WIDE` through `FETCH_EIGHT_WIDE` should be commented.
   - Repeat for the dispatch and issue widths (the commit width must remain at 4).
   - Uncomment `RR_TWO_DEEP` for a register read depth of 2, `RR_TWO_DEEP` and `RR_THREE_DEEP` for 3, etc.
   - Uncomment `ISSUE_THREE_DEEP` for an issue depth of 3. `ISSUE_TWO_DEEP` must remain uncommented.
   - Set the sizes of the structures in the "Structure Sizes" section.

- Comment/uncomment `ENABLE_LD_VIOLATION_PRED` to disable/enable the load violation predictor.

2. Run fpgaify.pl to change the RAM module names that depend on the width of a pipeline stage. The required arguments are:
   - -fw <fetch_width>
   - -dw <dispatch_width>
   - -iw <issue_width>