Hybrid HW/SW CPU Simulation using Zync

Jingxi Xu

4th Year Project Report
Computer Science and Electronics
School of Informatics
University of Edinburgh

2017

Abstract

Software simulation of modern microprocessors can be implemented very efficiently using a functional-first approach and then a cycle-accurate model. However, the implementation of cache model can sometimes slow down the simulation process. The goal of this project is to implement the cache model in hardware rather than software, to evaluate if the speed of simulation can be improved and to what extent. I use the Zybo board, with a dual-core ARM Cortex A9 system and a FPGA fabric on it. The A9 core will run the software part of the ArcSim simulator while offloading its cache model to the FPGA fabric.

For the cache module designed in this project, it surprisingly turns out to be a slow-down compared to implementing the same logic in pure software. But this is because of the simplicity of the cache design. With the analysis on the AXI interface, I firmly believe that the more logic we implement in hardware, the more delay in AXI interface can be compensated and we can finally get a speed-up.

Since the cache module designed in this project does not achieve a speed-up on its own, there must be no improvement on speed if integrated into ArcSim. In this dissertation, however, I do a prediction on what is the maximum possible improvement of speed if we have enough logic on hardware and what is the relationship between the speed-up of a cache model and the speed-up of the whole simulator.

Acknowledgements

I would like to thank:

Prof. Nigel Topham for patiently supervising me through the year. He not only teaches me how to handle technical problems but also inspires me on how to be a good researcher and engineer.

Dr. Boris Grot for coordinating project group meetings and giving me valuable feedback on my project.

my parents for supporting me all the time.

Table of Contents

1 Introduction **7**

1.1 Overview 7

1.2 Motivation 8

1.3 Contributions 10

1.4 Outline 11

2 Evaluation of Previous Work **13**

2.1 FPGA-based Simulation 13

2.2 Hardware Accelerator Advantages and Limits 14

2.3 Other Related Literature Review 15

3 Background **17**

3.1 ArcSim 17

3.2 Cache Architecture 18

3.2.1 Cache Organization 18

3.2.2 Cache Addressing 18

3.2.3 Replacement Policy 19

3.2.4 Write Policy 19

3.3 Zynq-7000 All Programmable SoC 19

3.3.1 Zynq-7000 Family Architecture 20

3.3.2 PS-PL Interfaces 21

3.3.3 Another Overview of This Project 22

3.4 AMBA AXI Protocol 23

3.4.1 AXI Architecture 23

3.4.2 Two-way Handshake Mechanism 24

4 System Setup **27**

4.1 Boot Linux on ARM Cortex-A9 CPUs 27

4.2 SSH Across Different Networks 27

4.3 Mount Filesystem 28

5 Implementation **31**

5.1 Cache in Verilog 31

5.1.1 Cache Design 31

5.1.2 Cache Implementation 35

5.2 Create Custom AXI Slave Peripheral 37

5.3 Call Hardware Cache Model From Software Driver 37

6 Test and Evaluation 39
6.1 Time Types for Benchmarking Program 39
6.2 Evaluation of AXI Interface Time Cost 39
6.2.1 Method 40
6.2.2 Results 42
6.2.3 Discussion 43
6.3 Software Cache Simulation vs. Hybrid Cache Simulation 43
6.3.1 Validation 44
6.3.2 Results 44
6.3.3 Discussion 45
6.4 Prediction on Potential Speed-up for ArcSim 46
6.4.1 Implementation 46
6.4.2 Results 47
6.4.3 Discussion 47

7 Conclusion 51
7.1 Summary 51
7.2 Critical Analysis 51
7.2.1 Difficulties Handled 52
7.2.2 Possible Improvement 52

Appendix A Verilog Code for Cache Module 55

Appendix B Test Bench Code for Cache Module 61

Appendix C Software Cache vs. Hardware Cache 63

Bibliography 69

Chapter 1

Introduction

1.1 Overview

As said by [CSS+06], being able to accurately and quickly predict properties of computer systems is important for architects, designers, software developers and users of computers. Simulators can provide us with a window into inner workings of a computer and because they do not have the constraints as a real implementation, they can be made easier to probe, examine and modify.

Two most important properties of a system simulator are *fast* and *accurate to cycle level resolution*; however, it is hard to have both as the more detailed information we need, the more complex this simulator's architecture will be, so the speed is affected [CSS+06, KVBW+12]. Some conventional wisdom even says that no simulators can simultaneously have both two properties while also being complete, transparent, cheap and easy to create [CSS+06].

In this project, I am trying to figure out a hybrid hardware/software simulation way to help ArcSim simulator to achieve both fastness and cycle-accurateness, similar to the work done by [CSS+06].

ArcSim [Insa], written in C++, is a high-speed functional and cycle-accurate[1] instruction set simulator[2] of the *Encore* [Insb] processor.

[1] A cycle-accurate simulator is a computer program that simulates a microarchitecture on a cycle-by-cycle basis. It must ensure that all operations are executed in the proper virtual (or real if it is possible) time — branch prediction, cache misses, fetches, pipeline stalls, thread context switching, and many other subtle aspects of microprocessors [Wik17].

[2] An instruction set simulator (ISS) is a simulation model, usually coded in a high-level programming language, which mimics the behaviour of a mainframe or microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers [Wik15].

When simulating the target microarchitecture under cycle-accurate mode, the ArcSim first runs a functional simulation to interpret each instruction, and during this time, some timing information such as the number of cycles of memory access and the number of cycles of execution stage, will be recorded in the `inst` object corresponding to each instruction. Those information is then passed to the cycle model through the `inst` objects to provide detailed latency statistics. Inside the cycle model, as you can probably guess, there is a cache model which simulates the behaviour of cache, contributing to the cycle-accurate information we need, and that is what this project will mainly focus on.

Instead of running the whole simulator on a DICE machine, this project will run it on a Zybo board while at the same time, splitting it into two parts, implemented in hardware and software respectively. The cache model will be implemented in hardware (FPGA) and the other parts of this simulator will be run on the dual-core ARM Cortex A9 system (capable of running Linux) of the board. Figure 1.1 gives us a clear skeleton of this project.

For more details on ArcSim, see section 3.1.

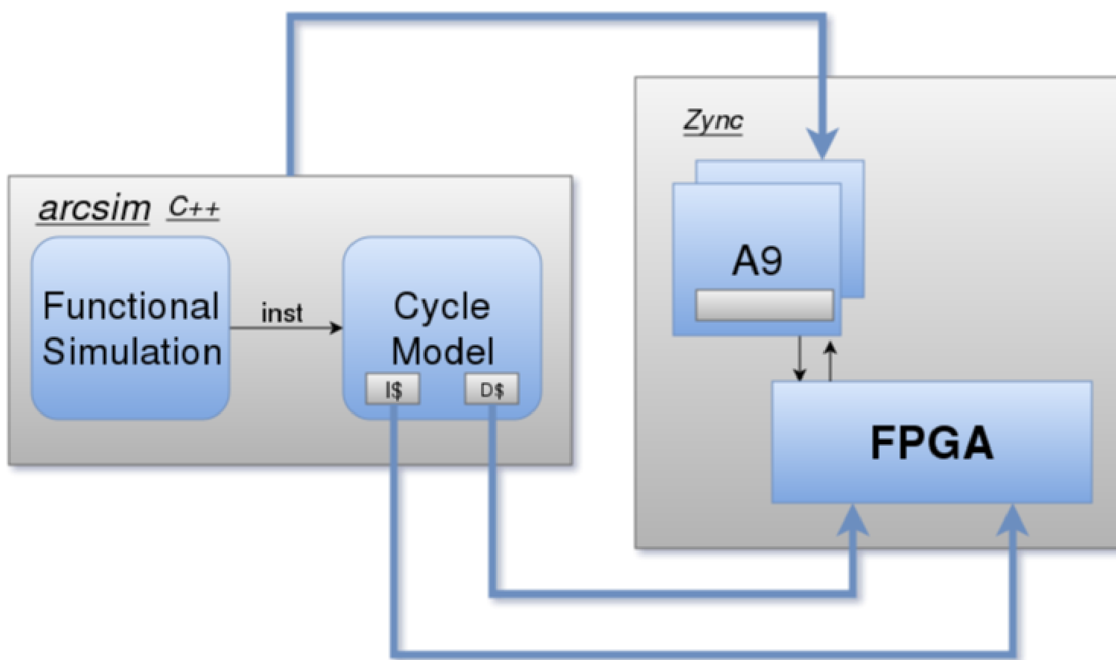

Figure 1.1: Overview of This Project

1.2 Motivation

When we run the functional simulation (see section 3.1 for details) of ArcSim using a little executable program called `speed`, we get the following execution profile shown in figure 1.2:

```
Instruction Execution Profile
-------------------------------------------------
                            Instructions   %Total
-------------------------------------------------
 Translated instructions:             0      0.00
 Interpreted instructions:   1000000054    100.00
 Total instructions:         1000000054    100.00
-------------------------------------------------

Simulation Time Statistics [Seconds]
-----------------------------------------------------
        Simulation   Tracing     Total
 Time:       23.84      0.00       23.84
-----------------------------------------------------

Interpreted instructions = 1000000054
Translated instructions  = 0
Total instructions       = 1000000054

Simulation time = 23.84 [Seconds]
Simulation rate = 41.95 [MIPS]
```

Figure 1.2: Functional Simulation

```
Instruction Execution Profile
-------------------------------------------------
                            Instructions   %Total
-------------------------------------------------
 Translated instructions:             0      0.00
 Interpreted instructions:   1000000054    100.00
 Total instructions:         1000000054    100.00
-------------------------------------------------

Simulation Time Statistics [Seconds]
-----------------------------------------------------
        Simulation   Tracing     Total
 Time:       47.96      0.00       47.96
-----------------------------------------------------

Interpreted instructions = 1000000054
Translated instructions  = 0
Total instructions       = 1000000054

Simulation time = 47.96 [Seconds]
Simulation rate = 20.85 [MIPS]
Cycle count     = 1000004710 [Cycles]
CPI             =  1.000
IPC             =  1.000
Effective clock = 20.9 [MHz]
```

Figure 1.3: Cycle-accurate Simulation

Then, if we enable the cycle-accurate mode (see section 3.1 for details), we get another profile shown in figure 1.3.

We can see that the functional simulation takes 23.84 seconds to finish while the cycle-accurate simulation takes 47.96 seconds, almost twice as much as it takes for a functional simulation. That is because after the cycle-accurate mode is enabled, the simulator has to do a large amount of extra calculation to get the delay information of this processor. It is probably intuitive to say that cache model contributes a lot to the slow-down of cycle-accurate simulation because for each instruction, we run instruction cache model to return the number of cycles it takes and similarly, for each memory access instruction, the data cache model is executed. It is also proven in section 6.4 that cache model contributes a lot to the slow-down of cycle-accurate simulation. Hence, if we want to improve the performance of cycle-accurate simulation of this simulator, it would be a good idea to first focus on improving the performance of cache model.

Since previous evaluation done by Rowson [Row94] has indicated that a hardware/-software co-simulation will have a better performance in terms of speed for a cycle-accurate model, it is worth trying to offload cache model in hardware, for instance, a FPGA fabric. Hence, instead of letting a CPU do the computational jobs cycle by cycle, which usually takes longer time and consumes higher amount of energy, we can directly implement those computational tasks on hardware.

In this project, I intend to develop a simple data cache model (always a good idea to start with something simple) in FPGA for this simulator and carry out evaluation on the speed of this hardware model compared to its software counterpart implementing the same logic. If the hardware model does show a speed-up, I will try to integrate it with ArcSim to see the overall performance. I also intend to evaluate the interface between software and hardware so that I can gain a better idea of whether the speed can be improved and to what extent. If the performance of cycle-accurate simulation is indeed improved or show a trend to be improved with just a little more future work, we may probably offload more elements of the simulator to hardware.

This project will give more motivation and numerical guide for future designers to work on this subject — hybrid hardware/software simulation.

1.3 Contributions

My contributions to this project mainly consists of the following points:

- Critically review previous work on this topic.
- Boot Linux on the dual-core ARM Cortex A9 system.
- Design and implement cache in hardware using Verilog.
- Design and implement the interface between the software simulation and the hardware simulation of cache.
- Evaluate the time cost of interface between hardware and software.

- Evaluate the performance of hardware cache model compared to its software counterpart.
- Evaluate the potential speed-up of the whole simulation system.
- Critically analyse my own work and propose appropriate future work for this subject.

1.4 Outline

The rest of the report is structured as follows:

Chapter 2 Evaluates some previous related research to gain a preliminary understanding of this topic and discusses what aspects from them can be applied to this project.

Chapter 3 Explains how I set up the system for this project and what difficulties I encounter during this process.

Chapter 4 Introduces some necessary fundamental terminologies, concepts and background knowledge for readers to better understand the general idea of this project.

Chapter 5 Describes in more details on how hardware cache module and interfaces are designed and implemented.

Chapter 6 Presents evaluation of the delay in hardware and software interface. Evaluates the performance of a simple hybrid simulation calling hardware cache module. Predicts potential performance improvement of the hybrid ArcSim simulator.

Chapter 7 Critically illustrates what has been done, what could have been done but is not because of time limitation, which part of this project should have been done in a better way and what aspect of this topic should be a major concern for future researchers.

Appendices and Bibliography

Chapter 2

Evaluation of Previous Work

2.1 FPGA-based Simulation

Khan et al. [KVBW+12] have well presented their work to implement a fast and cycle-accurate modelling of a multicore processor on FPGA called *Arete*. As with the increase of demand for design exploration and accurate performance estimates in modern simulators, more detailed models are required. This paper points out that even though FPGA-based simulators have drastically higher speed than software simulators, sacrificing fidelity is common.

As the aim of this project is to develop a hybrid hardware/software simulator, the main difficulties tackled by [KVBW+12] for FPGA-based simulation will also be the main focus of this project:

- *Programmability*: FPGAs are typically programmed in low-level RTL languages like Verilog, so the design is much less straightforward than software design using high-level programming languages and the architecture would be very inflexible.
- *Resource management*: Unlike software design, FPGA-based implementation has hard-resource constraints. Even though the logic of the design is correct but it does not fit into a particular FPGA if this logic overuses hardware resources such as LUTs, I/Os.
- *Interfacing with off-chip memory or host PC*: This problem would be of a bigger concern in this project than how it appears in [KVBW+12]. A key factor of this project is whether our speed-up in hardware can offset the time wasted in the interface. In addition, the problem of interfacing for this project is not with off-chip memory or host PC, but is between the simulator running on embedded Linux with hardware cache module on FPGA.

This paper also points out the key challenge for hardware-software co-design, which is to figure out the optimal hardware-software partitioning of algorithms for performance and power efficiency.

[CSS+06] does a good job in hardware-software co-design. It partitions a simulator into a software component and a hardware component implemented in FPGAs, and the resulting simulators are capable of 1M to 100M cycles per second and full cycle-accuracy.

2.2 Hardware Accelerator Advantages and Limits

Shand, Bertin and Vuillemin's paper [SBV91] provides some insights in the the use of harware accelerator and its limits. They present experiments in hardware/software design trade-off met in improving the speed of long integer multiplications by using PAM (*Programmable Active Memory*) [BRV89].

They state that in some intense computational task, the program contains a relatively simple inner loop which performs the bulk of this computation. Increasing the speed of this loop through hardware can lead to dramatic performance improvements. The paper mentions some application making use of such dedicated hardware such as *floating point coprocessors*, *vector coprocessors*, *graphic coprocessors*, etc. In the context of this project, cache model is the inner loop as it is frequently called by the program to achieve cycle-accurate simulation. We are trying to develop a hardware accelerator for implementing the cache model.

The authors also evaluate the limits of hardware accelerators. The main limit to performance achievable by hardware accelerators comes from the available communication bandwidth to the host and this limit is perfectly consistent with the third implementation difficulty presented by [KVBW+12]. In addition, [SBV91] makes another similar point that most of the work to develop a hardware accelerator goes into finding appropriate trade-offs between hardware and software processing in order to keep our application within the available bandwidth. It seems that deciding on how much computation to offload to hardware and evaluation on software and hardware interface time cost are quite necessary for this project.

Another limit demonstrated by [SBV91] is the economic consideration which requires using as little hardware resource as possible as a result of limitation on hardware resource. This limit is similar to the second difficulty mentioned in [KVBW+12]. To comply with this rule, we should not offload the part of simulator which is infrequently used but requires a lot of hardware to achieve useful speed-up. That being said, using programmable hardware is still more economically attractive than its alternative — using super-computers.

Moreover, [Row94] gives another drawback of the co-simulation system, and that is the difficulty of debugging the system with a hardware accelerator, especially when the hardware design is big and complex. In this project, every time we modify our hardware design, even just a little bit, we need to go through a lot of tedious steps (package IP, report IP status, generate output product, create HDL wrapper, etc.) to create the bitstream and then export it to FPGA. The synthesis process in Vivado, in particular, takes an extremely long time compared to compiling a software program. After this, we need to operate on the embedded Linux to see the results of the whole system. The

most efficient way to handle this problem is by modularizing our system into different parts and figure out a way to test each part to guarantee its correct functionality. In this way, the possibility of error occurring when testing the whole system is reduced to a large extent.

2.3 Other Related Literature Review

Paper [HHKC12, CCL05, AB86, SP05, PWKR02, VPNH10, DO04] also make use of hardware accelerators to improve the performance of speech recognition algorithms, real-time image feature extraction, artificial neural networks, etc. Most of them are based on FPGA and do get an impressive performance. Even though they are not explicitly explained as some other references in this dissertation, they contribute a lot to my understanding of this project and inspire my thoughts and ideas.

Chapter 3

Background

3.1 ArcSim

As said at the start of this dissertation, *ArcSim* [Insa] is a high speed functional and cycle-accurate instruction set simulator of the *Encore* [Insb] processor written in C++. It primarily has the following modes of simulation:

- *Functional simulation*: Most basic mode, just returns number of interpreted instructions, simulation time and rate of simulation. This is the default mode of simulation.
- *Cycle-accurate simulation*: This mode allows for cycle-accurate simulation of our microarchitecture, providing very detailed latency statistics for each instruction.
- *High-speed simulation*: This mode uses *Just-In-Time* (JIT) *Dynamic Binary Translation* (DBT) techniques to perform very high speed functional simulation at speeds approaching (or even exceeding) real time [TJ07]. Recently, this mode can be run together with cycle-accurate mode [BFT10].
- *Fast cycle-approximate simulation*: This mode enables fast prediction of cycle counts based on information gathered during fast functional simulation and prior training [Fra08].

To run the simulator, we will first need to compile our application using a suitable compiler to generate an ELF executable file. Giving compiler `arc-elf32-gcc` and the assembly file `speed.s` as an example:

```
arc-elf32-gcc -mA7 -o <output file> <input file>
```

Then, we can run ArcSim on the generated executable file by:

```
arcsim -v -c -e <ELF executable file>
```

Option [`-v`] tells the simulator to output performance statistics, option [`-c`] enables cycle-accurate mode and option [`-e`] indicates that the target file is of ELF executable type.

3.2 Cache Architecture

Cache plays a very important part in increasing the efficiency of memory access and data transfer. It makes use of the locality of reference [Den05] exhibited in computers to store frequently accessed data into an ultra-fast and compact memory component so that future requests can be served fast. Hennessy and Patterson's book [Pat11] helps me to review the knowledge of cache and then write this section.

3.2.1 Cache Organization

Cache block or *cache line* are the basic unit of a cache, usually consists of multiple bytes. Several blocks in a cache are grouped into a set.

Cache associativity means the number of blocks in a set, in other words, how many blocks a byte in memory can possibly go to in the cache. For instance, a *fully associative* cache basically means that a particular memory byte can go into every block in the cache; a *4-way set associative* cache means a particular memory byte can potentially go into a certain set of 4 blocks; and a *direct mapped* cache means this memory byte can only replace one particular block in cache.

Each block in cache will have a valid bit and a dirty bit attached to it. The *valid bit* of a cache block indicates whether or not this block has been loaded with valid data. In the initial state of a cache (no data has been written into the cache), all valid bits are 0. The *dirty bit* indicates whether this block has been modified since it was loaded from memory. The dirty bit is used when the cache is using a write-back policy.

3.2.2 Cache Addressing

Figure 3.1 shows us how a memory address is structured to be matched to a cache.

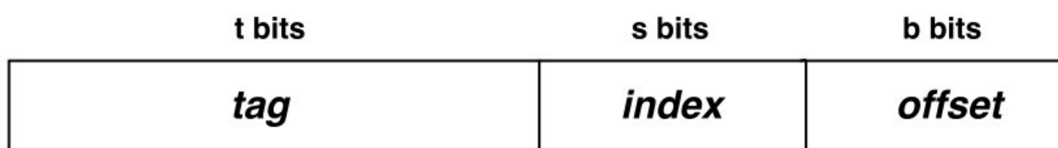

Figure 3.1: Cache Addressing

When we have a memory access to a particular address, the processing unit will first find which set this memory byte will possibly locate at based on the index part. Then it compares the tag of the address to each tag of the cache blocks in that set. If the tag of the address matches with one of the tags of cache blocks, that is a hit; otherwise, it is considered to be a miss. As a cache block usually contains a wide range of bytes, we need the offset part of the address to indicate where this particular memory byte lies in that cache line.

3.2.3 Replacement Policy

When there is a cache miss, and the blocks in the set where the data could go are all filled with valid content, then the cache has to use some replacement policy to find a victim block to be replaced by the new-coming data. There are lots of existed cache replacement policies and some of them are very complicated and tricky. Different replacement algorithms will generate different efficiencies and the policy of a cache is usually decided by specific requirements.

For the sake of this project, only the *first-in-first-out* policy is used. Its meaning is rather self-explanatory: the cache evict the first filled-in block regardless of how often or how recently it has been used.

3.2.4 Write Policy

When there is a *write-hit*, there are generally two ways to operate on cache:

Write-through Write is done both to the cache and to the memory

Write-back Write is done only to the cache. Write to the backing store is not implemented until the cache block containing the written data is about to be replaced by new content. In this case, we will need the dirty bit to tell the memory access instruction if this content in the block is different from that in the corresponding memory location.

When there is a *write-miss*, one of the following operations is carried out:

Write Allocate The data at the memory location is loaded into the corresponding cache block, followed by a *write-hit* operation.

No-write Allocate The data is written directly to the backing store.

Usually, a *write-through* cache will use *no-write allocate* policy and a *write-back* cache will use a *write allocate* policy.

3.3 Zynq-7000 All Programmable SoC

As said before, this project is based on the *Zybo Zynq-7000 ARM/FPGA SoC Trainer Board*. The embedded Linux, cache module, ArcSim simulator and the interconnects among them are all implemented on the *Zynq-7000 All Programmable SoC* architecture of the board. In this section, I will demonstrate the key features of the Zynq-7000 family which plays a vital role for this project.

3.3.1 Zynq-7000 Family Architecture

Figure 3.2 from [Sad14a] shows a brief overview of the Zynq-7000 family architecture.

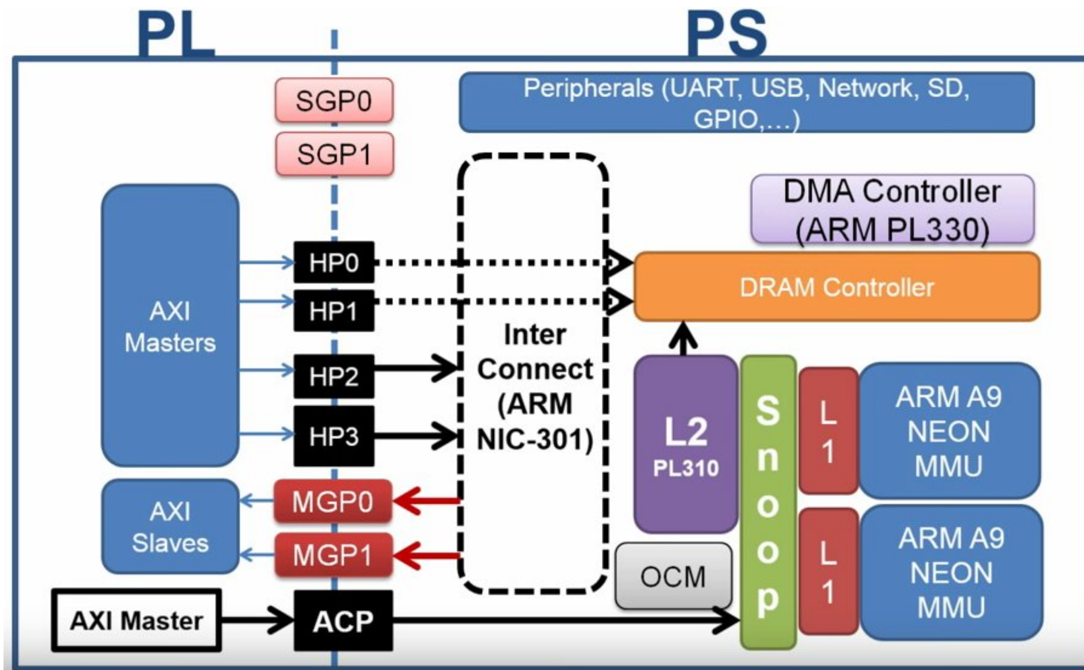

Figure 3.2: Brief Overview of Zynq System

The Zynq system consists mainly of two parts – a *processing system* (PS) and a *programmable logic* (PL). The *ARM Cortex-A9 CPUs* are the heart of PS and are where we will run the software side of ArcSim simulator to drive the cache module implemented in PL. Each CPU has a level-1 cache and the coherency is guaranteed by the Snoop control unit. The PS also has an on-chip memory (OCM), a level-2 cache, some external memory interfaces, and a rich set of peripheral interfaces. The interconnect device controls the communication among different units on the PS and also the communication with PL through PS-PL interfaces.

The PL is basically the FPGA fabric, which is where the hardware accelerator (cache model) is implemented on.

Each device in the Zynq system has its own address space and is not accessible by components other than themselves unless interfaces are provided, such as the AXI interfaces. Figure 3.3 from [Xil16b] illustrates the address space of different components in the Zynq system. The two columns circled by a red rectangular are the address spaces of the PL and are where the memory-mapped registers of the AXI slave will locate in.

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF [2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF [2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 3.3: System-Level Address Map

3.3.2 PS-PL Interfaces

Now, let us take a closer look specifically at the PS-PL AXI interfaces.

The four HP (*high performance*) ports (`HP0`, `HP1`, `HP2`, `HP3`) are the slave ports of the PS and we can design custom peripherals in PL with AXI master ports connected to these HP ports. It allows the AXI master peripherals to carry out read/write transactions to access DRAM or OCM of PS.

The two MGP (*master general purpose*) ports are the most important ports for this project and they are the only two master ports of PS. They provide the basis for the implementation of AXI protocol (explained in section 3.4) and allow the programs running on ARM Cortex-A9 cores to access the address space of PL through the AXI interface. They provide the way for PS to control and use the hardware. Furthermore, these two ports are the main means of the DMA controller on the PS to perform read and write to the logic on PL [Sad14b].

The ACP (*Accelerator Coherency Port*) is very similar to the HP ports and allows

the AXI master blocks on PL to initiate read/write transactions to the PS. The main difference between them is that the ACP allows the AXI master to first search caches of the CPUs because the ACP is connected directly to the Snoop control unit. If data exist in caches, then this transaction can be responded without looking into the DRAM memory of PS, leading to a faster and more energy-efficient data transfer.

Finally, the SGP (*slave general purpose*) ports are another two slave ports of the PS for master logic on PL to access different peripherals on PS.

3.3.3 Another Overview of This Project

After I learned about the Zynq system architecture, I can draw another skeleton of this project with a better view of how the cache module is integrated with the Zynq system and how AXI ports are connected. This is shown by figure 3.4.

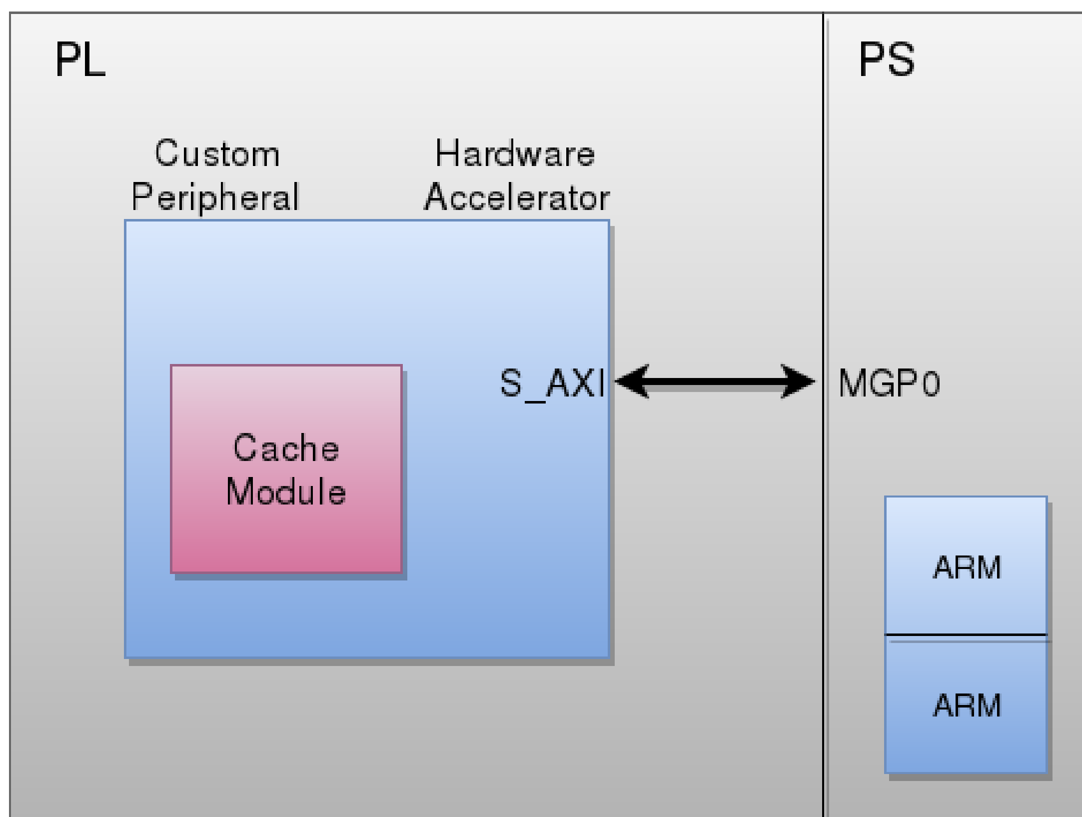

Figure 3.4: Another Overview of This Project

3.4 AMBA AXI Protocol

The AMBA (*Advanced Microcontroller Bus Architecture*) AXI (*Advanced eXtensible Interface*) protocol provides a way for the PS to communicate with PL. It defines channels and signals to guarantee the functionality of data transfer so that we can create custom IP peripheral in the PL and then add control and status monitoring capabilities by using memory-mapped registers which the processors (PS) can access via the AXI interconnect.

For a complete explanation and specification of the AXI protocol, please refer to [ARM04] and [Gri14]. In this section, I will selectively illustrate some main functionalities and mechanisms of the AXI protocol which are critical for understanding this project.

3.4.1 AXI Architecture

In this project, we are using an AXI4-Lite protocol which is a reduced form of the full AXI4 specifications and has single-beat transactions only (does not support for bursts or have transaction ID) [Sad15].

Five Channels

The functionality of AXI protocol is achieved mainly by five channels. Each of the five independent channels consists of a set of information signals and uses a two-way `VALID` and `READY` handshake mechanism, which is explained in section 3.4.2.

Figure 3.5 gives us a rough skeleton of the five channels with an arrowed line pointing from data provider (source) to data receiver (destination).

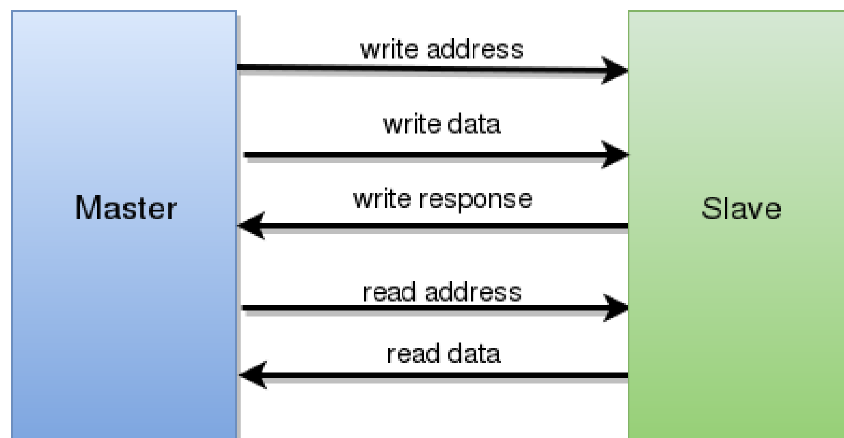

Figure 3.5: Skeleton of Five AXI Channels

Here are the definitions of the five channels:

- *Write Adrress Channel* carries all the required addresses and control information for a write transaction.
- *Write Data Channel* transfers the write data from master to slave.
- *Write Response Channel* provides a way for the slave to respond to write transactions. It carries information indicating the status of the write transaction including `OKAY`, `EXOKAY`, `SLVERR` and `DECERR`.
- *Read Address Channel* carries all the required addresses and control information for a read transaction.
- *Read Data Channel* transfers both the read data and any read response information from the slave back to the master.

Memory-mapped Registers

The AXI slave can have a user-defined number of memory-mapped registers which allow PS to write to the address space of PL and can be accessed and used by PL logic as control signals to achieve a particular function. They are the critical medium which contains data shared by PS and PL for communication and are the most important architecture in the project to guarantee the PS's control over PL.

3.4.2 Two-way Handshake Mechanism

As mentioned above, each channel of the AXI protocol uses a two-way `VALID`/`READY` handshake mechanism to transfer data.

To see the full set of defined signals of each channel for AXI4-Full protocol, please refer to [ARM04]. To see the full set of signals of each channel for AXI4-Lite protocol, please refer to [Gri14].

The source generates the `VALID` signal to indicate when the data or control information is available. The destination generates the `READY` signal to indicate when it is ready to accept the data or control information. The transfer can happen only when both the `VALID` and the `READY` signal are high.

Figure 3.6, 3.7 and 3.8 from [ARM04] show three scenarios when a handshake process happens with an arrow indicating the moment the transfer happens.

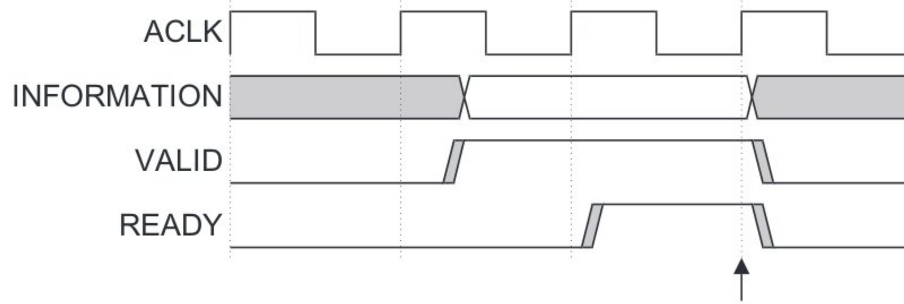

Figure 3.6: VALID Before READY Handshake

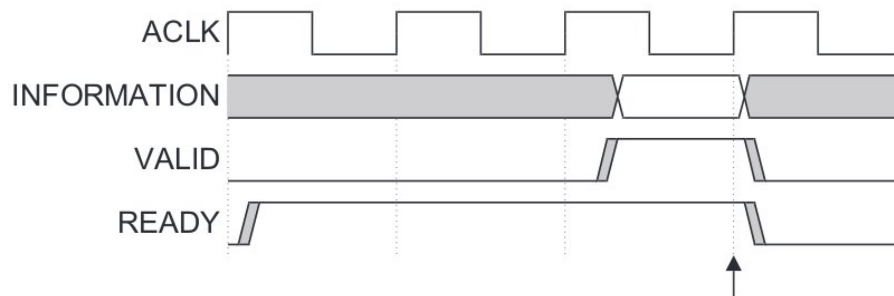

Figure 3.7: READY Before VALID Handshake

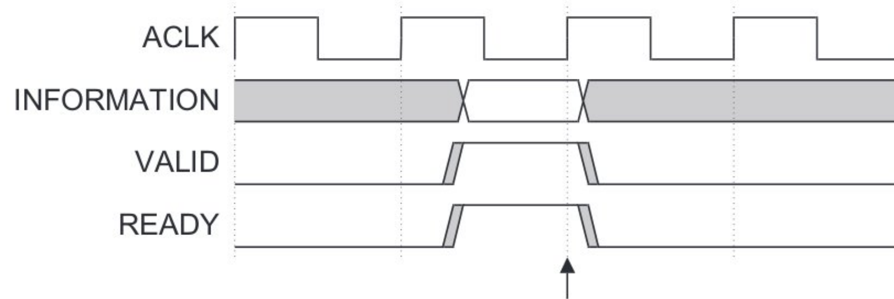

Figure 3.8: VALID With READY Handshake

It is worth noting that the source can generate the VALID signal even before the destination is ready to receive the data and vice versa. There is no specific requirement on

the order of VALID and READY assertions between source and destination.

It is a common misunderstanding on the handshake process that the sender must wait for the receiver to assert READY before it can assert VALID. This “wait for READY before asserting VALID” rule is ILLEGAL and will incur deadlock situation. READY can be asserted before VALID, but the sender should never wait for the assertion of READY as a premise to start the transaction [Gri14].

Chapter 4

System Setup

4.1 Boot Linux on ARM Cortex-A9 CPUs

This tutorial: `http://www.dbrss.org/zybo/tutorial4.html` was followed to boot Linux on the Zybo board.

The reason of running the software part of the simulator on a embedded Linux is that the bandwidth between PS and PL of Zynq system is much larger than that between external computers and FPGA on board. Thus, we save time on the communication between hardware and software.

4.2 SSH Across Different Networks

After I manage to run Linux on the Zybo board, to make it wirelessly accessible and able to update software and download some useful packages, I enable its Ethernet port for networking. As a result, I can also `ssh` (Secure Shell) into my user account on that embedded Linux system under the same network. But the problem is that when I want to use those Vivado or Xilinx SDK design tools on DICE machines, I cannot access my board connected to the router at home. That is because the network of a DICE machine is different from my home network. Thus, when I have a remote request sent to my home IP address from outside the network, the router simply does not know where to send my request [Tri16].

Figure 4.1 from [Tri16] illustrates this scenario. When we are using our laptop (225.213. 7.32) somewhere in the world and we want to access some files on our home laptop using `ssh`, we send this request to our public, or forward-facing IP address (127.34.73.214). Nevertheless, our router (192.128.1.1) cannot do anything because it does not know which device or port to forward this request to.

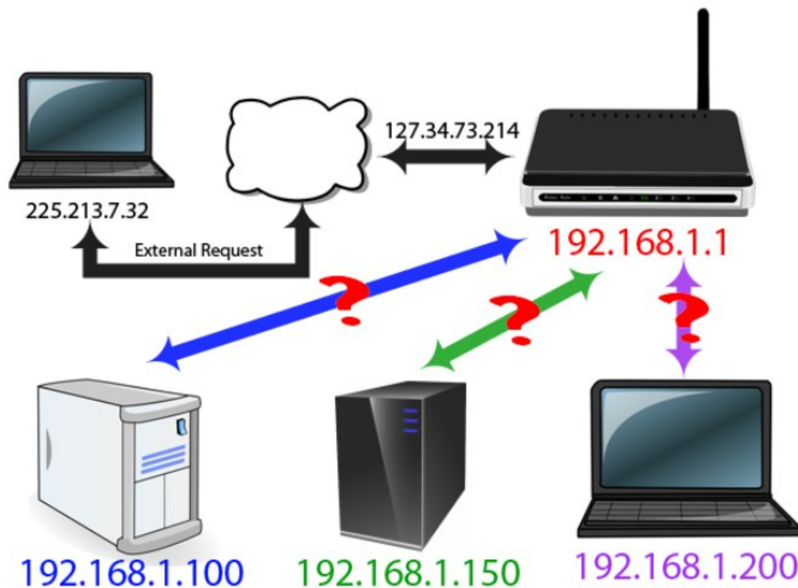

Figure 4.1: Remotely SSH to Another Network

As a result, we will need a port forwarding rule to tell our home network that when we access it using this program, it should send this request to this device at this port. Before doing so, we also need to assign a static IP address to our Zybo board because the address assigned to it might change when there is new device joining our home network [Tri11].

Then, we can configure our router by going to the router IP address in our web browser and add a port forwarding rule which specifies the IP address of our Zybo board and the port number 22 (the port number of `ssh`).

Finally, we can `ssh` to our board wherever we are by simply tapping the following command in terminal:

```
ssh -X -l user_name home_ip_address
```

It is worth noting that it would be better if we set a dynamic DNS service to link our home IP address to a memorable address name like `mysuperawesome.home.address` so that we do not need to check our numeric home network IP address every time before we `ssh` into our board because our home IP address can also change sometimes (although it might remain the same for several months or years) [Tri16].

4.3 Mount Filesystem

After enabling `ssh` to my home network from DICE machines, now I can edit files on my board using various tools on DICE by mounting the embedded Linux file system to the DICE machine, just by the following command:

```
sshfs <user name>@<home ip address>:/ <mount point path>
```

[/] means mounting from the root directory of the embedded Linux and `mount point path` is where on DICE I want to mount the directory.

After I finish editing files in my board or transferring files to my board, I can use the following command to unmount the file system from the DICE:

```
fusermount -uz <mount point path>
```

Command `fusermount` provides a secure method for non privileged users to mount and unmount their filesystems. Option [`-u`] represents `unmount` and option [`-z`] represents `lazy unmount`.

Chapter 5

Implementation

5.1 Cache in Verilog

To implement a cache in Verilog and then make it run on FPGA is not a complicated task requiring hundreds of lines of code, but it is tricky in the aspect that we need to convert our way of thinking in software to hardware. In hardware design, we do not have some handy data structures such as `class`, `struct`, `list` in software programming; neither can we write some tedious code in a concise form by using `for`, `while`, `if`, `else` wherever we want, but what we have is a series of `0`s and `1`s and what we need to do is connecting and modularizing them and using flip-flops when we need to store some values.

5.1.1 Cache Design

A relatively simple cache architecture is used to make everything run before modification. The following parameters are those of a typical and simple cache with an FIFO replacement policy which I simulate in hardware:

- Cache size: 32KB
- Cache line/block size: 64B
- Cache associativity: 4-way
- Cache replacement policy: First-In-First-Out
- Cache writing policy: Write-through and no-write allocate
- Number of blocks: 512
- Number of sets: 128

Also for the sake of simplicity, I use the most easy-to-implement write policy pair *write-through* and *no-write allocate*. I assume the following time costs for this model:

- Read hit: 1 cycle

- Read miss: 2 cycles
- Write hit: 3 cycles
- Write miss: 4 cycles

Because we are simulating Encore processor targeting an *ARCompact Instruction-set Architecture*, we have a 32-bit memory address, and then we can compute the sizes of different parts of this 32-bit address:

- Memory address size: 32 bits
- Tag size: 19 bits
- index size: 7 bits
- Offset size: 6 bits

Figure 5.1 shows us the overview of the cache module design in Verilog.

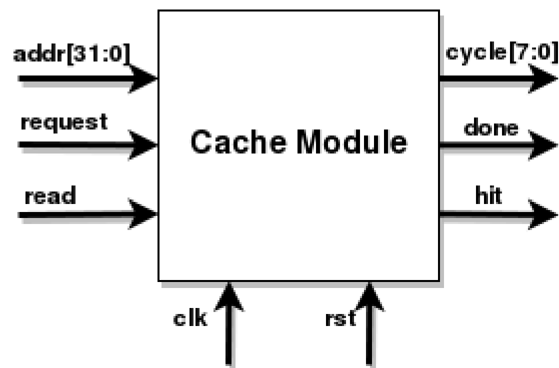

Figure 5.1: Cache Module Overview

Below is the definition of each input and output:

- `addr[31:0]`: The 32-bit memory address where the instruction want to access.
- `request`: Single-bit signal indicating that there is a valid memory access request and the cache module should start working to compute the latency.
- `read`: Single-bit signal indicating the type of this memory access instruction. 1 means a memory read instruction and 0 means a memory write instruction.
- `clk` and `rst`: Every operation of this module is synchronized with the input clock `clk`, including the synchronous reset `rst`.
- `cycle[7:0]`: This 7-bit output signal implies how many cycles it takes for this memory access instruction to finish. According to the previous time cost settings, the first three bits of this output are enough to give latency information.
- `done`: This single-bit signal is very important and is set to 1 only for 1 cycle after the cache module starts working. It indicates when the AXI slave should read the output data from cache module to corresponding memory-mapped registers.

- `hit`: Single-bit signal telling whether this memory access hits in cache or not.

Figures 5.2 to 5.5 show the waveforms of how this cache module works under read miss, read hit, write miss and write hit, for a random address `0x84fa7cc1`.

As the `request` signal is driven by the software running on the embedded Linux through the AXI interface, there should be two points that we should bear in mind when we are designing the hardware cache module (these rules hold true for other software-driven signals `addr` and `read` as well):

- Each value of `request` (either 1 or 0) will last for a very long period as software is much slower than hardware. Plus we know in section 6.2 that the write transaction from processing system to programmable logic takes over 150 cycles, so the value of `request` will remain the same for at least that amount of time.

- As the write to `request` is though the memory-mapped registers in AXI slave, the change of its value is synchronized with the clock. That is why in our waveforms, every modification on request takes place on the positive edge of `clk`.

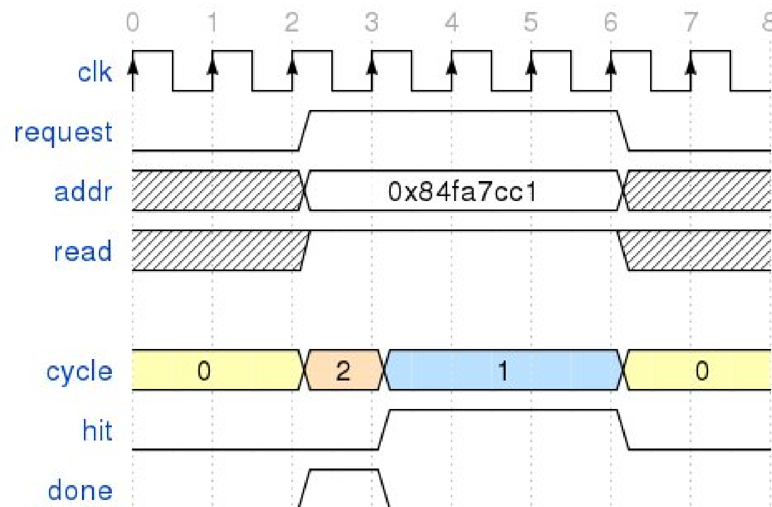

Figure 5.2: Waveform for Cache Read Miss

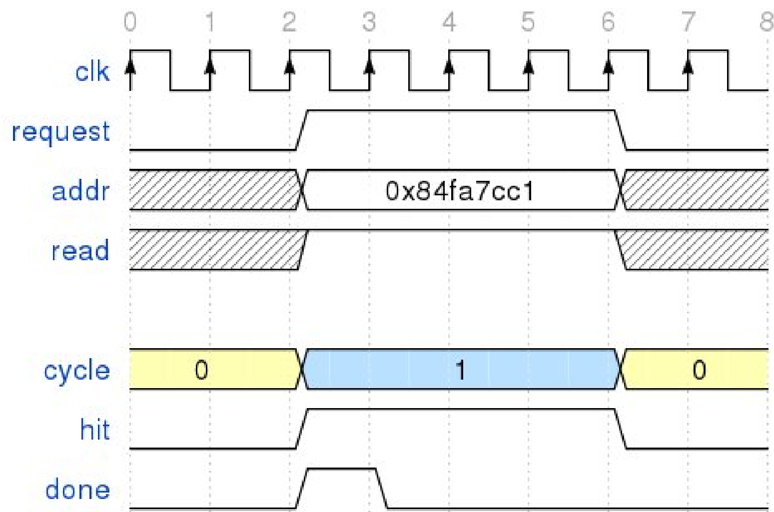

Figure 5.3: Waveform for Cache Read Hit

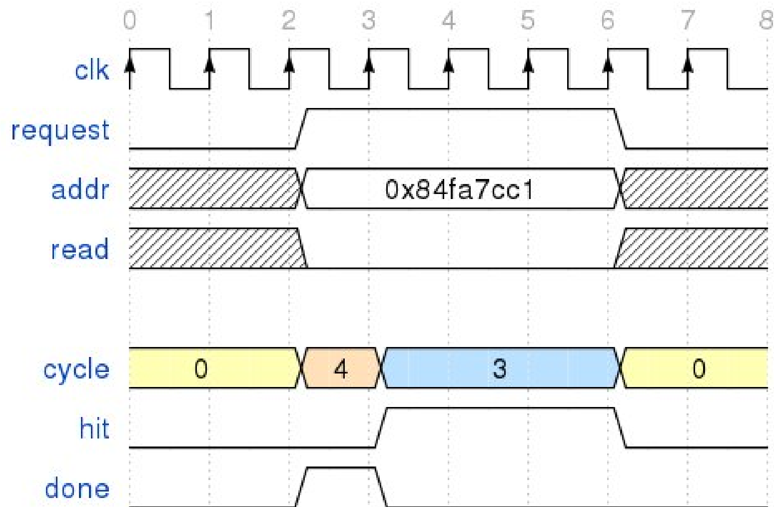

Figure 5.4: Waveform for Cache Write Miss

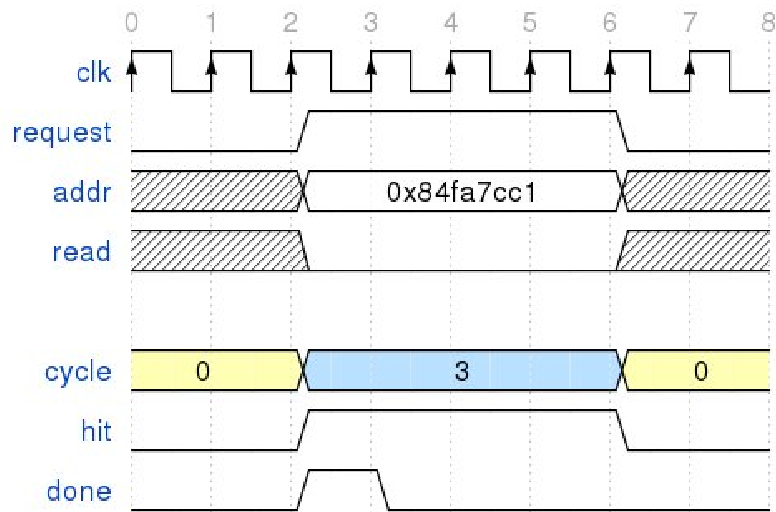

Figure 5.5: Waveform for Cache Write Hit

Only the mechanism for the cache read miss scenario is explained in this section as the other three are similar.

The software driver (simulator) running on Linux will set the values of `request`, `addr` and `read` at any time and the changes of them are synchronized with positive `clk` edge in the AXI slave. The cache module starts working and will output the results (`hit` and `cycle`) in the next cycle. The status of cache will be changed in the same cycle and in the following clock cycle, since the `request` signal is still high, the cache module will run again and because the missed block has already been available in cache at this moment, output `hit` will be 1 and `cycle` will be 1 (read hit). That is why we need the `done` signal to tell the AXI slave when to read the output data from cache module to corresponding registers. The data to read should be the results in the first cycle after `request` is asserted. Thus, the `done` signal is 1 only for the first cycle after `request` is asserted. In our AXI peripheral, the results are read only when `done` is high.

Then, the outputs will remain the same. The final stable result of a particular memory request will always be a read hit or write hit (`hit` = 1 and `cycle` = 1 or 3).

At the positive edge when `request` is set to 0 (4 cycles indicated in the waveforms are just a representation. The real time span during which `request` remains high will be much longer), the output will all be set to 0. As the `done` is also 0, these results will not be read to memory-mapped registers so the software driver will not be able to see these zeros.

5.1.2 Cache Implementation

To see the complete code for this cache module in verilog, please turn to appendix A.

`reg [18:0] tags [511:0]` is an array of tags of each block in the cache (there are 512 cache lines and each cache line has a 19-bit tag). `validBits[511:0]` is an array of the valid bits of those 512 blocks in the cache. Given a particular `index`, we can easily get the valid bits (`v0, v1, v2, v3`) and tags (`t0, t1, t2, t3`) of the four blocks in that set by:

```
v0 = validBits[index * 4 + 0]; t0 = tags[index * 4 + 0]
v1 = validBits[index * 4 + 1]; t1 = tags[index * 4 + 1]
v2 = validBits[index * 4 + 2]; t2 = tags[index * 4 + 2]
v3 = validBits[index * 4 + 3]; t3 = tags[index * 4 + 3]
```

It is worth noting that at first I did not use two-dimensional arrays in Verilog (for example, I defined `tags` simply as `reg [9727:0] tags`), and after one and half hours' synthesis, it turns out that even though this way of definition passes simulation, it overuses the LUTs on FPGA. Hence, it is in fact not feasible.

Then, we can use these valid bits and tags to judge whether this memory access is a hit or not and give the result in the output signal `hit`. Based on `read` and `hit`, we can compute the latency `cycle` of this memory access. We also need to update the cache status based on the replacement policy if there is a miss.

Since `request` is high for a long period, the cache operations are run at every clock positive edge. We need to use signal `done` to make sure that only results in the next cycle after assertion of `request` are read to memory-mapped registers by the AXI slave. To achieve this, we need another variable `integer loop`. `loop` is set to 1 in the next cycle after the assertion of `request` and is increased by one in each of the following cycles until `request` is set back to 0 when `loop` is also set to 0.

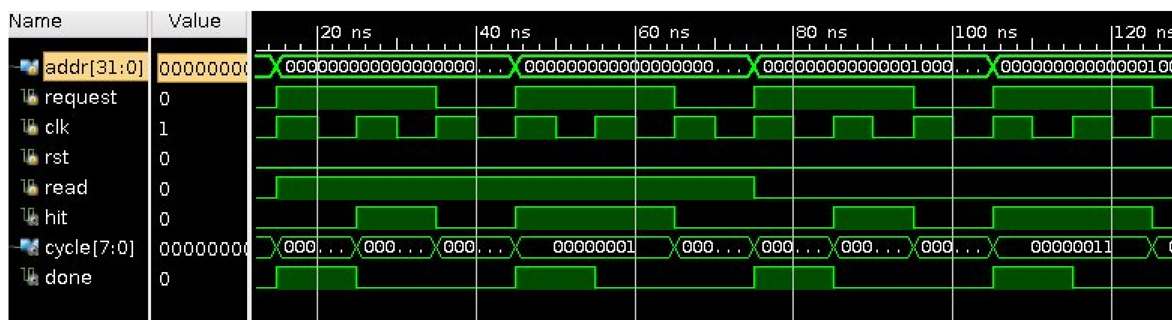

Figure 5.6: Wave Graph of Cache Design Simulation in Verilog

Figure 5.6 shows the wave graph of the cache module simulation. The four memory accesses are a read miss, a read hit, a write miss and a write hit. When `done` is 1, the corresponding values of `cycle` are 2, 1, 4 and 3 respectively. The complete test bench code to generate this simulation wave graph is in appendix B.

5.2 Create Custom AXI Slave Peripheral

I created an custom AXI IP block in Vivado with the following code of user logic. I commented out the default Verilog code for writing to `slv_reg1` so that this register can be assigned the result (`cycle`) from the cache module. The software driver can drive the hardware by writing to `slv_reg0` (request), `slv_reg2` (address) and `slv_reg3` (instruction type).

In the `always` block, `slv_reg1` is written based on `done` so that it gets the correct result.

```
wire hit;
wire [7:0] cycle;
wire done;
wire rst;

cache cache (
    .addr(slv_reg2),
    .read(slv_reg3),
    .request(slv_reg0),
    .hit(hit),
    .cycle(cycle),
    .done(done),
    .clk(S_AXI_ACLK),
    .rst(~S_AXI_ARESETN)
);

always @(posedge S_AXI_ACLK)
begin
    slv_reg1 <= slv_reg1;
    if (S_AXI_ARESETN == 1'b0) begin
        slv_reg1 <= 0;
    end
    else begin
        if (done == 1'b1)
            slv_reg1 <= cycle;
    end
end
```

AXI User Logic

5.3 Call Hardware Cache Model From Software Driver

After finishing the design of the whole hardware system (including our AXI slave peripheral and hardware cache module), we need to generate a bitstream file for this hardware design and them export it to FPGA from the embedded Linux. The following command can help us achieve that:

```
dd if=<bitstream file> of=/dev/xdevcfg
```

After this, the logic is written to FPGA on the board so that we can use the hardware in the software driver running on PS.

Remember that if we do not reset the board (normally by pushing the physical reset button or rebooting the Linux), the data written on the address space of PL is not automatically cleared after the completion of a program. The hardware cache status we get after simulating a set of memory accesses will remain unchanged for the next program. Thus, we might get two different results for the same set of memory accesses. To start a new test with an empty cache, we can run the above command again because it can reload the bitstram file and refresh the PL so all the old data in the cache is cleared.

To call the hardware cache module through AXI interface we will first need to get the base address of the memory-mapped registers from the Address Editor in Vivado. Then we need a function to get a virtual address for this physical base address so that we can write and read this address space by creating a pointer to it in our software driver. A function to achieve this following examples in [Fle14] and an example of calling hardware cache module in the software driver can be found in appendix C.

Chapter 6

Test and Evaluation

6.1 Time Types for Benchmarking Program

Before we start our tests on the time cost of difference implementations and architectures, we need to first clarify the following three time types [SGGS98] used for timing a program.

- *Real (wall) time* is the actual elapsed real-world time including time slices used by other processes and the time the process is blocked, such as waiting for I/O.
- *User CPU time* is the amount of CPU time spent in user mode (outside the kernel) to execute the process. It is the actual CPU executing time outside the kernel for this process only. The time spent by other processes or by the process being blocked is excluded.
- *System CPU time* is the amount of CPU time spent in kernel mode (inside the kernel) to execute the process. It is the actual CPU executing time inside the kernel for this process only. Similar as user CPU time, other processes and time the process spends blocked do not count towards this figure.

These three parameters also correspond to the three outputs (`real`, `user`, `sys`) of the terminal command `time` in Linux. It is quite obvious that we need to use the total CPU time spent by a process (system CPU time + user CPU time) for benchmarking different architectures and implementations in this project.

6.2 Evaluation of AXI Interface Time Cost

[CSS+06] has already stated the importance of taking care of the interface, "FAST performance hinges on the interface between the functional model and the timing model. Care must be taken to maximize the performance of the communication protocol, its implementation and the physical link."

It is obvious that the reason for offloading the cache simulation to FPGA is because the whole implementation (check cache hit or not, update cache, output cycle count) just takes several clock cycles; however, the primary drawback of this method is that we introduce new delay from transferring signals in AXI interfaces. It is quite possible that this newly-joined delay could compensate what we have achieved by taking advantage of the high speed of hardware.

The AXI protocol, as introduced in section 3.4, is not a simple protocol and has a bunch of signals to guarantee its functionality, so its influence on simulation speed could not be ignored. The real aim of this project is, to some extent, to figure out the **trade-off between how much functionality of the simulator should be offloaded to hardware and how much new delay would be introduced by the AXI interfaces for building this hybrid architecture**.

As a result, I find it quite important to spend some time on the test of just the AXI interface before the test of our new hybrid simulation system, so that we can gain a better understanding of the results from much more complicated situations.

6.2.1 Method

The analysis method would be similar for a single read transaction between PS and PL, so I will just explain the evaluation of time spent for a single write transaction.

From [Xil16a] we know that under the board type and configuration (illustrated in table 6.1, represented by `xc7z010clg400-3` in Vivado Design Suite) used for this project, the maximum frequency of the processing system is 866 MHz. So 1 clock cycle is $1/(866 \times 10^6) = 1.15 \approx 1ns$. I will just make a reasonable approximation in this dissertation that one clock cycle is $1ns$.

Option	Value
Family	Zynq-7000
Sub-family	Zynq-7000
Device Name	Z-7010
Part Number	XC7Z010
Package	clg400
Speed Grade	-3

Table 6.1: Board Type and Configuration for This Project

Now, I create an empty AXI-Lite slave peripheral which has no user logic. The only thing it does is enabling the PS (processing system) to write a value to or read a value from the PL (programmable logic) through the memory-mapped registers provided by the slave.

The following three pieces of code are executed on the embedded Linux to get the time cost for a single write (the code to time the execution of the loop is only presented in the first test, but is obviously needed in all tests):

```
clock_t begin = clock();
for(k = 0; k < 10; k++) {
  for (j = 0; j<1000; j++) {
    for (i = 0; i < 1000; i++) {
      // Do nothing
    }
  }
}
clock_t end = clock();
double time_spent = (double)(end - begin)/CLOCKS_PER_SEC;
```

Write Test 1

```
for(k = 0; k < 10; k++) {
  for (j = 0; j<1000; j++) {
    for (i = 0; i < 1000; i++) {
      // Write value i to a normal variable
      var = i;
    }
  }
}
```

Write Test 2

```
for(k = 0; k < 10; k++) {
  for (j = 0; j<1000; j++) {
    for (i = 0; i < 1000; i++) {
      // Write value i to slv_reg0 in PL
      *a = i;
    }
  }
}
```

Write Test 3

```
for(k = 0; k < 10; k++) {
  for (j = 0; j<1000; j++) {
    for (i = 0; i < 1000; i++) {
      // Write value i to a normal variable but through a pointer
      *ptr = i;
    }
  }
}
```

Write Test 4

Test 1 does not do anything in each iteration and is just for reference. `var` in test 2 is an integer previously defined. Variable `a` in test 1 is a pointer to where register `slv_reg0` in AXI slave locates at. Details on how we get a virtual address from the physical address automatically provided by the Vivado Address Editor can be found in section 5.3. `ptr` in test 4 is a pointer to a normal integer variable.

In fact, I disabled the PS's ability to write to `slv_reg1` and do `slv_reg1 <= slv_reg0` on each positive clock edge, so that I can print out the value of `slv_reg1` in my program to check that the number is written to `slv_reg0` correctly. This will not affect our test

results of the time cost for interfacing between PS and PL and is just for the purpose of validation.

6.2.2 Results

Table 6.2 shows the timing results for three tests (each result is the average value of running the same piece of code for 10 times).

Test	CPU Execution Time
1	0.124s
2	0.154s
3	2.003s
4	0.139s

Table 6.2: Timing Results for Write Tests

For each test except the first one, the write operation is implemented for 10^7 times.

From test 2 we know that a write to a variable in software takes:

$$(0.154 - 0.124) \div 10^7 = 3ns$$

From test 3 we know that a single write to the register in programmable logic has a time cost of:

$$(2.003 - 0.124) \div 10^7 = 188ns$$

From test 4, we know that a write to an integer variable in software through a pointer takes:

$$(0.139 - 0.124) \div 10^7 = 1.5ns$$

As mentioned before, the clock cycle for the Zynq system is $1ns$. Hence, we have:

Type of write	Number of clock cycles
Write in software	3
Write in software through pointer	1.5
Write to hardware through AXI	188

Table 6.3: Number of Cycles for Different Write Types

I understand that in each test, the operation also includes a read from integer variable `i` in software, but since this is made the same for all 3 tests (with an operation in each loop), the results shown in table 6.3 can still reflect the information we need.

6.2.3 Discussion

It is clear that because the system needs to go through the handshake protocol for each signal channel it uses if PS wants to communicate with PL, it takes much longer time to carry out a single write transaction between PS and PL. The signal transfers between PS and PL (`VALID`, `READY`, etc.) for guaranteeing the functionality of AXI protocol significantly slow down the program: more than 60 times slower than a single write in software and more than 120 times slower than a single write in software through pointer. The result that write through a pointer is significantly faster than write directly to the variable in software is an interesting phenomenon but is beyond the discussion of this project.

However, one key point here is that we does not have any user logic in our AXI peripheral for these test programs. If we offload more computation to hardware, the advantage of hardware will start showing up. Even though going though AXI interface will take much more time than a single write in software, we might just need to write to the hardware once at the start and leave all the calculation to be implemented on hardware, which will excitingly only need several extra clock cycles, and at the end of calculation, carry out another high-cost read to receive data from hardware. This is exactly the case of our cache simulation in hardware. Writing the address and instruction type information to hardware might take a long time, but all the remaining computation can be done in several cycles in hardware. What we need to do is simply read back the data through AXI interface after the computation finishes.

In comparison, as our program becomes more and more complicated, we will have more and more reads and writes in software, and unlike the case of hardware, the number of cycles for each operation will add up. 3 (or 1.5) might not be a large number on its own, but when added up in a rather big program, it will very likely surpass the time it takes to do several reads and writes through AXI interface. There must be a certain point where the total number of cycles for operations in software will compensate the number of cycles spent in the AXI interface plus the several extra cycles for computation in hardware. That will be the moment where hybrid simulation will have a higher speed as a whole.

6.3 Software Cache Simulation vs. Hybrid Cache Simulation

This section will focus on whether the number of cycles saved by hardware computation of the simple cache designed in 5.1 will compensate the number of cycles wasted in the AXI interface.

The code for this section is in Appendix C.

I write another cache model in C which runs purely in software on the A9 core and has the same architectures, parameters and specifications as the hardware cache module I implement in 5.1. I will compare the speeds of calculating the number of cycles of a

single memory access instruction by two cache models to see if using our hardware cache module through AXI interface will result in an improvement on speed.

6.3.1 Validation

To validate my software and hardware cache simulation, the program generates 1×10^6 (`#define NUM_REQUESTS 1000000`) memory access requests with random addresses and random instruction types (read or write). This set of memory access requests is run both on the software cache module and on the hardware cache module through the AXI interface.

Table 6.4 shows us the results when the set of random memory accesses are tested on both software cache model and hardware cache model.

No. Test	Read Hit	Read Miss	Write Hit	Write Miss
1	8	500618	4	499370
2	6	500031	6	499957
3	10	500100	9	499881
4	7	500106	10	499877
5	5	500051	5	499939
6	9	499988	15	499988

Table 6.4: Simulation Results for a Set of Random Memory Accesses

The software cache model and hardware cache model are getting exactly the same results for all the tests so we can reasonably say that both models are doing their job correctly. It is worth noting that the number of hit memory accesses is significantly smaller than missed instructions. This is reasonable as the requests we generated are completely random so there is no principle of locality applied on them [Den05]. Plus that the cache model are fairly simple with an FIFO replacement policy, the performance is expected to be extremely bad. The set of memory requests without any locality is not what happens in real world but it should not affect the comparison of speeds between two models.

6.3.2 Results

If we turn off validation (program does not print out statistics so no need to do extra work in the `for` loop). We can get the timing results for simulating the whole set of random memory accesses using different models. Each value is obtained as the average of 20 tests.

Model	CPU Execution Time
Software Cache Model	0.3451s
Hardware Cache Model	0.8274s

Table 6.5: Timing Results for Simulating the Whole set

Divide the values in table 6.5 by 10^6, we can get the timing results for simulating a single memory access using two models. The same assumption made in preivious section is still hold here that $1\,cycle \approx 1ns$.

Model	CPU Execution Time (in ns)	CPU Execution Time (in cycles)
Software Cache Model	345ns	345 cycles
Hardware Cache Model	827ns	827 cycles

Table 6.6: Timing Results for Simulating a Single Memory Access

The results are consistent with the results we get in 6.2. We can see from the code that it has 4 writes to the address space of PL in a single `for` loop for cache simulation in hardware. We know from 6.2 that a single write will take $188ns$ so one iteration will take at least $4 \times 188 = 752ns$. Plus that in one iteration, we also need to read from arrays and variables in software, the value $827ns$ indeed makes sense as the time cost for a single memory access simulation.

From the results we can see that calling the hardware cache module through AXI interface is slower than implementing the computation directly on software. **It suggests that under the specifications of our simple cache module in hardware, the delay on AXI interface is not compensated by the high-speed of computation in hardware.**

6.3.3 Discussion

This result might seem disappointed at first glance but it in fact indicates great possibilities if we compare it with previous results from experiments on AXI interface in 6.2.

Type of operation	Percent of slow-down
Write to a register in hardware	6167%
Cache simulation in hardware	140%

$P_{slowdown} = (t_{hardware} - t_{software}) \div t_{software} \times 100\%$

Table 6.7: Percent of Slow-down for Different Operations

From table 6.7 we can see that even though our hardware cache model still shows a slow-down compared to its counterpart in software, this slow-down is much more slight than that of a single write to a register in hardware. Actually, such a large amount of

relief on slow-down with such a small amount of computation offloaded to hardware is quite exciting.

As the cache module designed in this project is a fairly simple one and we get such a reduction on slow-down, we can reasonably say that there must be a point when time spent in AXI interface can be offset by the amount of time saved by hardware computation on FPGA and that point should be very close from what we have now. Off course, to get such a speed-up, we will need to keep the number of accesses to the address space of PL down but offload more computation to PL.

From this trend, it seems very likely to get a speed-up if our cache is designed to be a slightly more complicated one with for example, a write-back and write allocate policy and a *least recently used* replacement policy, because in that case we will have more hardware computation while communication between PS and PL will remain the same as the simple cache module in this project. The CPU execution time of the complicated hardware cache model should be the same as the simple one, but the execution time of the software cache model will increase much more significantly if we add more functionality to it.

6.4 Prediction on Potential Speed-up for ArcSim

As the cache module designed in this project does not achieve a speed-up on its own compared to software implementation executing the same logic, there is no point integrating this module with the ArcSim simulator to substitute corresponding cache simulation because we are bound to get a slow-down as well.

In this section, however, assume we have a hardware cache module with sufficient computation offloaded to hardware as depicted by previous section and we do get a speed-up of α on its own, we are trying to figure out to what extent the speed of the simulator as a whole can be improved.

6.4.1 Implementation

If we look into `MemoryModel.h` of the ArcSim simulator, we can find that the memory model of this simulator consists of two `CacheModels` (data and instructions), two `CCMModels` (data and instructions) and one `MainMemoryModel`.

For the sake of figuring out how much slow-down the modelling of cache contributes to the overall slow-down of cycle-accurate simulation, we need to comment out the operations for checking cache hit or miss and calculating latency and cycles. To do so, I comment out the operations in functions `is_dc_hit()` and `is_dirty_dc_hit()` and just make them return `false`. I also comment out operations in function `fetch()`, `read()` and `write()` so that they only return a `cycle` of value 0. All these functions I mentioned are in the `MemoryModel.h` file of the ArcSim simulator source code.

6.4.2 Results

Recompile this simulator on DICE machine and test on the same executable file called `speed` as we used in section 1.2. We can get the results shown in figure 6.1.

```
Instruction Execution Profile
-----------------------------------------------
                           Instructions   %Total
-----------------------------------------------
 Translated instructions:            0     0.00
 Interpreted instructions:  1000000054   100.00
 Total instructions:        1000000054   100.00
-----------------------------------------------

Simulation Time Statistics [Seconds]
---------------------------------------------------
        Simulation   Tracing    Total
 Time:       41.48      0.00      41.48
---------------------------------------------------

Interpreted instructions = 1000000054
Translated instructions  = 0
Total instructions       = 1000000054

Simulation time = 41.48 [Seconds]
Simulation rate = 24.11 [MIPS]
Cycle count     = 1000000371 [Cycles]
CPI             =  1.000
IPC             =  1.000
Effective clock = 24.1 [MHz]
```

Figure 6.1: Cycle-accurate Simulation Without Cache

If we compare the simulation time with that of figure 1.3, we can see there is a speed-up (there is an improvement on simulation rate as well) after we disable cache modelling. By running each simulator for ten times and computing the average value of simulation time, we get the results in table 6.8

Type of simulation	Simulation time
Cycle-accurate simulation with cache modelling	$48s$
Cycle-accurate simulation without cache modelling	$41s$

Table 6.8: Simulation Time for Different Simulation Types

6.4.3 Discussion

We can see from the table that $(48-41) \div 48 \times 100\% = 16\%$ of the cycle-accurate simulation time might be the subject of speed-up as the result of a hybrid HW/SW

simulation. If we have a speed-up of α for the cache model, according to Amdahl's Law [Amd67], the speed-up of the whole system would theoretically be:

$$S = \frac{1}{1 - 0.16 + \frac{0.16}{\alpha}} = \frac{\alpha}{0.84\alpha + 0.16}$$

When speed-up α goes to infinity, the maximum overall speed-up in theory would be:

$$S_{max} = \lim_{\alpha \to \infty} \frac{\alpha}{0.84\alpha + 0.16} = 1.2$$

Figure 6.2 plots the relationship between the speed-up of the simulator as a whole and the speed-up of the cache model.

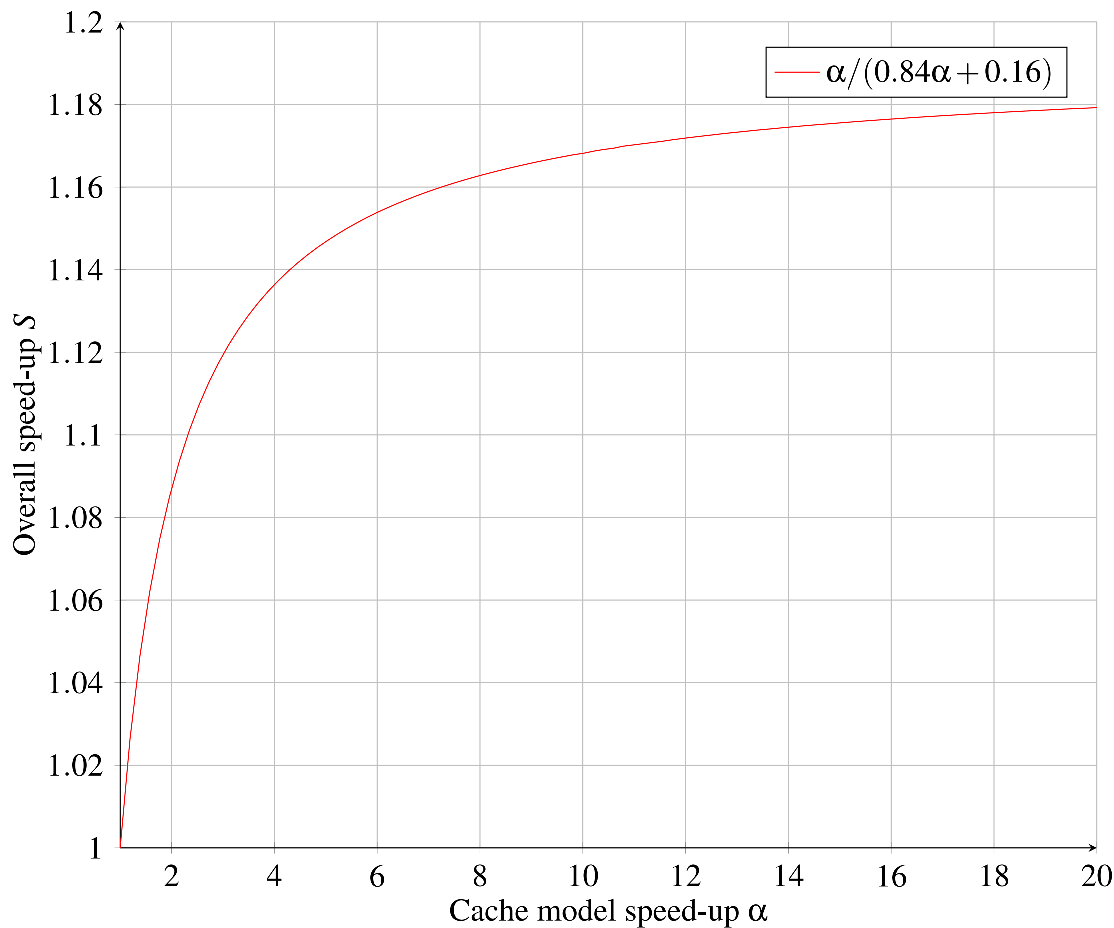

Figure 6.2: Plot of Overall Speed-up Along Cache Model Speed-up

The discussion in this section gives us a great guide for future designers in the design process. As stated by [Kri01], the fraction of task that does not use the improved feature limits the performance and speed-up. Designers should know the performance limits due to other slow parts of this program and try to improve the most frequently used components.

According to the evaluation in this section, I would say that choosing the cache model to be the first part of this simulator to be offloaded to hardware is a wise idea. If we recall the comparison between a functional simulation and cycle-accurate simulation in section 1.2, we see that the latter spends twice as much time as the former, and among those extra 23 seconds, cache makes up $7 \div 23 \times 100\% = 30\%$, almost $1/3$. Even though the unaffected part of this simulator is still the major part, if we are aiming specifically at relieving the slow-down of cycle-accurate model (the 23-second slow-down), offloading cache simulation to hardware would be undoubtedly a good idea.

Also, we can see from the structures of the ArcSim simulator, cache model is highly independent from the whole memory model and is easily structured and modularized, which is another advantage of implementing this part in hardware.

My suggestion for future designers would be to aim at a speed-up of cache model at value $4 \sim 8$. In that range, the speed-up of the whole simulator starts to increase slower and stabilize. It should be the most efficient range as it helps the whole simulator to achieve a high performance while consuming relatively small amount of hardware resource (less computation offloaded to hardware).

In comparison to previous work [SBV91, HHKC12, CCL05, AB86, SP05, PWKR02, VPNH10, DO04], I would not say the potential improvement of performance is impressive, but it is worth trying as a speed-up of $4 \sim 8$ on cache model should not be hard to realize according to our previous evaluation on the relief of slow-down with the amount of computation offloaded to hardware. In addition, it provides the basic structure for designers to offload more parts of ArcSim to FPGA.

Chapter 7

Conclusion

.

7.1 Summary

With the goal to develop a hybrid hardware/software simulator (cache model on hardware) to see to what extent the simulation can be speeded up, I start with the implementation of a simple cache model on FPGA. I compare the performance of a hybrid simulation calling this hardware cache model with a pure software simulation implementing the same cache architecture in software. I find that there is a slow-down for the hybrid simulation but by evaluating the AXI interface time cost and the performance of my two cache simulations, I figure out that this slow-down is as a result of the simplicity of the cache.

Finally, after analysing the cache model of ArcSim, I am confident to say that with this amount and complexity of computation, there must be a speed-up if we substitute this cache implementation with a hardware model. In addition, by using Amdahl's law, I figure out that the maximum speed-up of this simulator with a hardware cache model would be about 1.2 and the most efficient speed-up of the cache model which future designers should aim at is $4 \sim 8$.

7.2 Critical Analysis

In this section, I will analyse the difficulties handled in this project and which part of this project can be improved as future work.

7.2.1 Difficulties Handled

System Setup

This project requires a mixed use of computer and the Zybo board, so the system setup is a critical foundation for completing the following parts of this project. This particularly requires a convenient access to the board and an efficient file transfer between computer and the board. This difficulty is tackled by SSH across different networks and mounting remote file systems on a DICE machine, so it becomes much easier to make use of software tools on DICE machines to develop programs and systems on Zybo board.

Programming in Verilog

I was not confident in using low-level RTL languages like Verilog for FPGA programming before I start this project. I spend a long time in learning the syntax and grammar of hardware description language and converting my way of thinking from software to hardware. [Tal14] helps me significantly in learning Verilog.

Resource Management of FPGA

I use some very large arrays of registers in my cache design and it turns out to be the reason that this design takes one and half hour to synthesize, even though it gets the correct results for the test bench. Following the suggestion of my supervisor, I substitute those large size one-dimensional arrays with two-dimensional ones and then the usage of LUTs drops to within the normal range.

Debugging of a Hybrid System

Every time I intend to test my hybrid system I have to go through a bunch of steps to see the result. Among those steps are some fairly time-consuming ones such as synthesis. This issue is mitigated by dividing the whole system into smaller function units. Only when the previous units on which the following units are based are tested and guaranteed to work properly, can I continue to implement and test the following parts. If we write the code for the whole system and then test it all together, it is very likely to get errors and is difficult to locate those bugs.

7.2.2 Possible Improvement

Add More Features to Cache Design

I do not implement the cache model of ArcSim completely in hardware due to time constraints. Instead, I develop a simple cache model and do my evaluation based on

this model and the interface. Further work can be done in adding more features to the cache model according to that of ArcSim and then integrate it with the ArcSim to see the real speed-up.

Parametrize Cache Module

The cache module should be parametrized if time allows. In this way, we can not only carry out wider range of tests but also gain more speed-up as parametrizing in software will have higher time cost.

Generate Random Addresses

When I am generating random memory accesses in section 6.3 (code presented in appendix C), the random address generated by `rand()` has a range of $0 \sim 2147483647$ ($2^{31} - 1$). Thus, the most significant bit of the random address is always 0. Other measures should be taken if we want a set of random addresses covering every possibility of a 32-bit integer ($-2147483647 \sim 2147483647$), but this should have no influence on the speed comparison between two cache simulations.

Drive Request Signal in AXI Slave

Currently the `request` signal is driven by the software simulator. The simulator call cache module by writing 1 to the register where `request` points to. This works as we know software is much slower than hardware so the time between two statements in software is long enough for the hardware to finish its job. We do not even need a feedback from hardware to inform the simulator that the computation is done.

A drawback of this method is we need to initiate a write transaction from PS to PL specially for `request` signal. Another way to do this is to drive this signal in the AXI slave peripheral. Whenever the AXI slave sees a write to `address` or `read`, it asserts the request. After AXI sees the `done` signal from cache module, it can set `request` to 0. In this way, we reduce time spent in the AXI interface by omitting one write transaction but use more hardware resource.

Programming Language Inconsistency

I test the interface and compare the speeds of software simulation and hybrid simulation using C language; however, the ArcSim is written in C++. It is better to keep the programming language consistent to save troubles when we integrate the cache module with ArcSim. This is not a big deal because of the similarity between two languages.

Compile ArcSim on ARM

The ArcSim we evaluate is the one compiled on DICE. However, at the end, the software part of the simulator should run on ARM core. It is better to compile ArcSim on ARM and then carry out evaluation, even though the proportion of cache model usage should not be affected by where we run the simulator.

Appendix A

Verilog Code for Cache Module

```
`timescale 1ns / 1ps

//cache size: 32KB
//cache line size: 64B
//cache associativity: 4-way
//cache replacement policy: FIFO
//cache write policy: write-through and no-write allocate

//memory address size: 32 bits
//cache tag size: 19 bits: numBlocks: 512
//cache index size: 7 bits: numSets: 128
//cache offset size: 6 bits

//penalty
//read hit: 1 cycle
//read miss: 2 cycles
//write hit: 3 cycles
//write miss: 4 cycles

module cache (
    addr,
    read,
    request,

    hit,
    cycle,
    done,

    clk,
    rst
);

    input wire [31:0] addr;
    //if read equals 1, this is a load instruction;
    //otherwise, this is a store instruction
    input wire read;
    //indictes whether there is a valid memory access
    input wire request;

```

```
    //indicates whether this access hits or not
    output reg hit;
    //indicates how many cycles this memory access will take
    output reg [7:0] cycle;
    //indicates that it is the right time to read results
    output reg done;

    input clk;
    input rst;

    //array of the valid bit of each block
    reg [511:0] validBits;
    //records the first coming-in block in each set.
    //two bits are enough for each set
    reg [1:0] firstInBlock [127:0];
    //array of the tag of each block
    reg [18:0] tags [511:0];

    reg [511:0] next_validBits;
    reg [1:0] next_firstInBlock [127:0];
    reg [18:0] next_tags [511:0];

    reg v0, v1, v2, v3;
    reg [18:0] t0, t1, t2, t3;


    reg [7:0] index;
    reg [18:0] tag;

    reg hit0, hit1, hit2, hit3;

    reg [3:0] setValidBits;

    integer i;
    integer j;
    integer y;
    //to set "done" just for one clock cycle
    integer loop;

    // sequential logic, with synchronous reset
    always @(posedge clk) begin
        if (rst == 1'b1) begin
            hit <= 0;
            cycle <= 0;
            done <= 0;
            loop <= 0;

            validBits <= 0;
            for (i = 0; i < 512; i=i+1) begin
                tags[i] <= 0;
            end
            for (i = 0; i < 128; i=i+1) begin
                firstInBlock[i] <= 0;
            end
        end

```

```
96          else if (request == 1'b1) begin
97              loop = loop + 1;
98              if (loop == 1)
99                  done = 1;
100             else
101                 done = 0;
102             ////////////////////////////////////////////////
103             // compute hit and cycle
104             index = addr[12:6];
105             tag = addr[31:13];
106             // get the tags of a set, getTags
107             t0 = tags[index * 4 + 0];
108             t1 = tags[index * 4 + 1];
109             t2 = tags[index * 4 + 2];
110             t3 = tags[index * 4 + 3];
111 
112             // get the valid bits of a set, getValidBits
113             v0 = validBits[index * 4 + 0];
114             v1 = validBits[index * 4 + 1];
115             v2 = validBits[index * 4 + 2];
116             v3 = validBits[index * 4 + 3];
117 
118             //judge whether it is a cache hit
119             hit0 = (v0 == 1'b1 && tag == t0) ? 1'b1 : 1'b0;
120             hit1 = (v1 == 1'b1 && tag == t1) ? 1'b1 : 1'b0;
121             hit2 = (v2 == 1'b1 && tag == t2) ? 1'b1 : 1'b0;
122             hit3 = (v3 == 1'b1 && tag == t3) ? 1'b1 : 1'b0;
123 
124             hit = hit0 | hit1 | hit2 | hit3;
125 
126             case ({read, hit})
127                 2'b00:  //write miss
128                     cycle = 4;
129                 2'b01:  //write hit
130                     cycle = 3;
131                 2'b10:  //read miss
132                     cycle = 2;
133                 2'b11:  //read hit
134                     cycle = 1;
135                 default:
136                     cycle = 0;
137             endcase
138             ////////////////////////////////////////////////
139 
140             ////////////////////////////////////////////////
141             // decide the next status of the cache
142             next_validBits = validBits;
143             for (i = 0; i < 512; i=i+1) begin
144                 next_tags[i] = tags[i];
145             end
146             for (i = 0; i < 128; i=i+1) begin
147                 next_firstInBlock[i] = firstInBlock[i];
148             end
149 
150             if(hit == 0 && request == 1) begin
151                 setValidBits[0] = v0;
```

```
152                 setValidBits[1] = v1;
153                 setValidBits[2] = v2;
154                 setValidBits[3] = v3;
155 
156                 //if there is empty block in the cache set,
157                 //fill them first
158                 for (i = 0, j = 0; i != 4 && j != 1; i = i + 1)
    begin
159                     if(setValidBits[i] == 1'b0) begin
160                         next_validBits[addr[12:6] * 4 + i] = 1'b1;
161                         next_tags[index * 4 + i] = addr[31:13];
162                         j = 1;
163                     end
164                 end
165 
166                 //if all blocks are filled and no tag matches,
167                 //needs to find one to replace
168                 if (j == 0) begin
169                     //get the number of the first comming-in
170                     //block for set addr[12:6]
171                     y = firstInBlock[index];
172                     next_tags[index * 4 + y] = addr[31:13];
173 
174                     //update firstInBlock
175                     if (y == 3) begin
176                         next_firstInBlock[index] = 2'b00;
177                     end
178                     else begin
179                         next_firstInBlock[index] = y + 1;
180                     end
181                 end
182             end
183             ///////////////////////////////////////////////
184 
185             ///////////////////////////////////////////////
186             // update the status of cache
187             validBits = next_validBits;
188             for (i = 0; i < 512; i=i+1) begin
189                 tags[i] = next_tags[i];
190             end
191             for (i = 0; i < 128; i=i+1) begin
192                 firstInBlock[i] = next_firstInBlock[i];
193             end
194             ///////////////////////////////////////////////
195         end
196         else begin
197         // if request is 0, sets output to 0
198         // because done is 0 as well,
199         // hit(0) an cycle(0) will not be read from PS
200             hit <= 0;
201             cycle <= 0;
202             done <= 0;
203             loop <= 0;
204         end
205     end
```

```
206 endmodule
```

cache.v

Appendix B

Test Bench Code for Cache Module

```
`timescale 1ns / 1ps

module cache_tb();

    reg [31:0] addr;
    reg request;
    reg clk;
    reg rst;
    reg read;
    wire hit;
    wire [7:0] cycle;
    wire done;

    cache DUT(addr, read, request, hit, cycle, done, clk, rst);

    initial begin
        rst = 1;
        clk = 0;
        addr = 0;
        request = 0;
        read = 0;

        #6
        rst = 0;

  // Demo test set ****************************
        #9
        addr = 32'b0000000000000000000_0000100_000001;
        request = 1;
        read = 1;
        //1, read miss
        //cycle = 2

        #20
        request = 0;

        #10
        addr = 32'b0000000000000000000_0000100_000010;
        request = 1;
```

```
40          read = 1;
41          //2, read hit
42          //cycle = 1
43
44          #20
45          request = 0;
46
47          #10
48          addr = 32'b0000000000000010000_0000100_000001;
49          request = 1;
50          read = 0;
51          //3, write miss
52          //cycle = 4
53
54          #20
55          request = 0;
56
57          #10
58          addr = 32'b0000000000000010000_0000100_000000;
59          request = 1;
60          read = 0;
61          //4, write hit
62          //cycle = 3
63
64          #20
65          request = 0;
66
67          #15 $finish;
68    // Demo test set end ***********************
69      end
70
71      always
72          #5 clk = !clk;
73  endmodule
```

cache_tb.v

Appendix C

Software Cache vs. Hardware Cache

```
#include <stdint.h>
#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stddef.h>
#include <time.h>

//cache size: 32KB
//cache line size: 64B
//cache associativity: 4-way
//cache replacement policy: FIFO
//cache write policy: write-through and no-write allocate

//memory address size: 32 bits
//cache tag size: 19 bits: numBlocks: 512
//cache index size: 7 bits: numSets: 128
//cache offset size: 6 bits

//penalty
//read hit: 1 cycle
//read miss: 2 cycles
//write hit: 3 cycles
//write miss: 4 cycles

#define NUM_REQUESTS 1000000

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)
#define ADDER_BASE_ADDR 0x43c00000

void *getvaddr(int phys_addr)
{
    void *mapped_base;
    int memfd;
```

```
40
41      void *mapped_dev_base;
42      off_t dev_base = phys_addr;
43
44      // to open this, the program needs to run as root
45      memfd = open("/dev/mem", O_RDWR | O_SYNC);
46      if (memfd == -1) {
47        printf("cant open /dev/mem. \n");
48        exit(0);
49      }
50
51      // map one page of memory into user space such that the
52      // device is in that page, but it may not
53      // be at the start of the page
54
55      mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE,
        MAP_SHARED, memfd, dev_base & ~MAP_MASK);
56      if (mapped_base == (void*) - 1) {
57        printf("cant open the memory to user space. \n");
58        exit(0);
59      }
60
61      // get the address of the device in user space which
62      // will be an offset from the base that was mapped
63      // as memeory is mapped at the start of a page
64
65      mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
66      return mapped_dev_base;
67  }
68
69  // structure of cache line
70  typedef struct {
71    int validBit;
72    int tag;
73  } CacheLine;
74
75  // structure of cache
76  typedef struct {
77    CacheLine blocks[512];
78    int firstInBlock[128];
79  } Cache;
80
81  int memoryAccess_software (int addr, int read, Cache* cache) {
82    int index = addr >> 6 & 0x0000007f;
83    // logical left shit, bits vacated are filled with zeros
84    int tag = addr >> 13;
85    int hit = 0;
86    int cycle = 0;
87
88    int i;
89    // check if it hits or misses
90    for (i = 0; i < 4; i++) {
91      if (cache->blocks[index * 4 + i].tag == tag && cache->blocks[
        index * 4 + i].validBit == 1) {
92        hit = 1;
93        break;
```

```
94     }
95   }
96
97   // cache miss, update cache
98   if (hit == 0) {
99     int blockToFill = 0;
100    if (cache->firstInBlock[index] == 3)
101      blockToFill = 0;
102    else
103      blockToFill = cache->firstInBlock[index] + 1;
104    cache->blocks[index * 4 + blockToFill].tag = tag;
105    cache->blocks[index * 4 + blockToFill].validBit = 1;
106
107    cache->firstInBlock[index] = blockToFill;
108  }
109
110  // calculate cycle count
111  if (hit == 0 && read == 1)
112    cycle = 2;
113  else if (hit == 1 && read == 1)
114    cycle = 1;
115  else if (hit == 0 && read == 0)
116    cycle = 4;
117  else
118    cycle = 3;
119
120  return cycle;
121 }
122
123 int main()
124 {
125   //statistics
126   int numReadHit = 0;
127   int numReadMiss = 0;
128   int numWriteHit = 0;
129   int numWriteMiss = 0;
130
131   int numReadHit_harware = 0;
132   int numReadMiss_hardware = 0;
133   int numWriteHit_hardware = 0;
134   int numWriteMiss_hardware = 0;
135
136   // for random number generation
137   srand(time(NULL));
138
139   Cache cache;
140
141   // initialise software cache
142   int i;
143   for(i = 0; i < 512; i++) {
144     cache.blocks[i].validBit = 0;
145     cache.blocks[i].tag = 0;
146     if (i < 128)
147       cache.firstInBlock[i] = -1;
148     //printf("%d tag: %d, valid bit: %d\n", i, cache.blocks[i].tag,
        cache.blocks[i].validBit);
```

```
149   }
150
151   // initialise AXI slave
152   // slv_reg1 <= slv_reg0 + slv_reg2
153   int * dev_base_vaddr = (int*) getvaddr(ADDER_BASE_ADDR);
154   int * request  = dev_base_vaddr;  // slv_reg0
155   int * cycle = dev_base_vaddr + 1; //slv_reg1
156   int * addr = dev_base_vaddr + 2; //slv_reg2
157   int * read = dev_base_vaddr + 3; //slv_reg3
158   *addr = 0;
159   *read = 0;
160   *request = 0;
161
162   // generate requests
163   int addrs[NUM_REQUESTS];
164   int reads[NUM_REQUESTS];
165   for(i = 0; i < NUM_REQUESTS; i++) {
166     addrs[i] = rand();
167     reads[i] = rand()%2;
168   }
169
170   // time software memory access simulation
171   clock_t begin_soft = clock();
172   for(i = 0; i < NUM_REQUESTS; i++) {
173     int cycle_soft = memoryAccess_software(addrs[i], reads[i], &
      cache);
174     /****** For Validation ******/
175     // switch (cycle_soft) {
176     //   case 1:
177     //     numReadHit++;
178     //     break;
179     //   case 2:
180     //     numReadMiss++;
181     //     break;
182     //   case 3:
183     //     numWriteHit++;
184     //     break;
185     //   case 4:
186     //     numWriteMiss++;
187     //     break;
188     // }
189     /********** End **********/
190   }
191   clock_t end_soft = clock();
192   double time_spent_soft = (double)(end_soft - begin_soft)/
      CLOCKS_PER_SEC;
193   printf("%fs\n", time_spent_soft);
194
195   // time hardware memory access simulation
196   clock_t begin_hard = clock();
197   for(i = 0; i < NUM_REQUESTS; i++) {
198     *addr = addrs[i];
199     *read = reads[i];
200     *request = 1;
201     *request = 0;
202     /****** For Validation ******/
```

```
203     // switch (*cycle) {
204     //   case 1:
205     //     numReadHit_harware++;
206     //     break;
207     //   case 2:
208     //     numReadMiss_hardware++;
209     //     break;
210     //   case 3:
211     //     numWriteHit_hardware++;
212     //     break;
213     //   case 4:
214     //     numWriteMiss_hardware++;
215     //     break;
216     // }
217     /*********** End ***********/
218   }
219   clock_t end_hard = clock();
220   double time_spent_hard = (double)(end_hard - begin_hard)/
        CLOCKS_PER_SEC;
221   printf("%fs\n", time_spent_hard);
222
223   /****** For Validation ******/
224   // printf("software cache results:\n");
225   // printf("number of read hit: %d\n", numReadHit);
226   // printf("number of read miss: %d\n", numReadMiss);
227   // printf("number of write hit: %d\n", numWriteHit);
228   // printf("number of write miss: %d\n", numWriteMiss);
229   //
230   // printf("hardware cache results:\n");
231   // printf("number of read hit: %d\n", numReadHit_harware);
232   // printf("number of read miss: %d\n", numReadMiss_hardware);
233   // printf("number of write hit: %d\n", numWriteHit_hardware);
234   // printf("number of write miss: %d\n", numWriteMiss_hardware);
235   /*********** End ***********/
236
237   return 0;
238 }
```

SoftwareVsHardware.c

Bibliography

[AB86] Thomas S Anantharaman and Roberto Bisiani. A hardware accelerator for speech recognition algorithms. *ACM SIGARCH Computer Architecture News*, 14(2):216–223, 1986.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[ARM04] ARM. AMBA AXI Protocol. Specification v1.0, ARM, 2004.

[BFT10] Igor Böhm, Björn Franke, and Nigel Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. 1989.

[CCL05] Jian-Wen Chen, Cheng-Ru Chang, and Youn-Long Lin. A hardware accelerator for context-based adaptive binary arithmetic decoding in H. 264/AVC. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4525–4528. IEEE, 2005.

[CSS+06] Derek Chiou, Huzefa Sunjeliwala, Dam Sunwoo, John Xu, and Nikhil Patil. FPGA-based fast, cycle-accurate, full-system simulators. In *Proceedings of the second Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12, Austin, TX*, 2006.

[Den05] Peter J Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[DO04] James P Durbano and Fernando E Ortiz. FPGA-based acceleration of the 3D finite-difference time-domain method. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 156–163. IEEE, 2004.

[Fle14] Shane Fleming. Creating a simple AXI slave adder and interfacing with the Zynq. https://www.youtube.com/watch?v=XtvVfjIm9Xw, 2014. [Youtube Video; Accessed 1-April-2017].

[Fra08] Björn Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78. ACM, 2008.

[Gri14] Rich Griffin. Designing a Custom AXI-lite Slave Peripheral. Technical Report Version 1.0, SILICA, 2014.

[HHKC12] Feng-Cheng Huang, Shi-Yu Huang, Ji-Wei Ker, and Yung-Chang Chen. High-performance SIFT hardware accelerator for real-time image feature extraction. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(3):340–351, 2012.

[Insa] Institute for Computing Systems Architecture, The University of Edinburgh. ArcSim Instruction Set Simulator. http://groups.inf.ed.ac.uk/pasta/tools_arcsim.html. [Online; Accessed 3-April-2017].

[Insb] Institute for Computing Systems Architecture, The University of Edinburgh. Encore Embedded Processor. http://groups.inf.ed.ac.uk/pasta/hw_encore.html. [Online; Accessed 3-April-2017].

[Kri01] S Krishnaprasad. Uses and abuses of amdahl's law. *Journal of Computing Sciences in colleges*, 17(2):288–293, 2001.

[KVBW+12] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, et al. Fast and cycle-accurate modeling of a multicore processor. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 178–187. IEEE, 2012.

[Pat11] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[PWKR02] Mario Porrmann, Ulf Witkowski, Heiko Kalte, and Ulrich Ruckert. Implementation of artificial neural networks on a reconfigurable hardware accelerator. In *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, pages 243–250. IEEE, 2002.

[Row94] James A Rowson. Hardware/software co-simulation. In *Design Automation, 1994. 31st Conference on*, pages 439–440. IEEE, 1994.

[Sad14a] Mohammadsadegh Sadri. Brief overview of Zynq architecture. https://www.youtube.com/watch?v=Jpi4_Acyqnw&list=PLQGDgb5p8ClhvSztd-vHeBr4fo3EXfr85&index=13, 2014. [Youtube Video; Accessed 1-April-2017].

[Sad14b] Mohammadsadegh Sadri. Zynq AXI interfaces part 1 (lesson 3). https://www.youtube.com/watch?v=nAycgPUOiAI&t=1213s, 2014. [Youtube Video; Accessed 1-April-2017].

[Sad15] Mohammadsadegh Sadri. Creating custom AXI slave interfaces part 1 (lesson 6). https://www.youtube.com/watch?v=meQcwzC4Vtk&t=332s, 2015. [Youtube Video; Accessed 1-April-2017].

[SBV91] Mark Shand, Patrice Bertin, and Jean Vuillemin. Hardware speedups in long integer multiplication. *ACM SIGARCH Computer Architecture News*, 19(1):106–113, 1991.

[SGGS98] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-wesley Reading, 1998.

[SP05] K Sridharan and TK Priya. The design of a hardware accelerator for real-time complete visibility graph construction and efficient FPGA implementation. *IEEE Transactions on Industrial Electronics*, 52(4):1185–1187, 2005.

[Tal14] Deepak Kumar Tala. *Verilog Tutorial*. ASIC World, 2014.

[TJ07] Nigel Topham and Daniel Jones. High speed CPU simulation using JIT binary translation. In *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, 2007.

[Tri11] Yatri Trivedi. How to set up static DHCP so your computer's IP address doesn't change. http://www.howtogeek.com/69612/how-to-set-up-static-dhcp-on-your-dd-wrt-router/, 2011. [Online; Accessed 3-April-2017].

[Tri16] Yatri Trivedi. Forward ports on router. http://www.howtogeek.com/66214/how-to-forward-ports-on-your-router/, 2016. [Online; Accessed 3-April-2017].

[VPNH10] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134. IEEE, 2010.

[Wik15] Wikipedia. Instruction set simulator — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Instruction_set_simulator&oldid=656736749, 2015. [Online; Accessed 3-April-2017].

[Wik17] Wikipedia. Computer architecture simulator — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Computer_architecture_simulator&oldid=764006551, 2017. [Online; Accessed 3-April-2017].

[Xil16a] Xilinx. Zynq-7000 All Programmable SoC Overview. Product Specification DS190 (v1.10), Xilinx, 2016.

[Xil16b] Xilinx. Zynq-7000 All Programmable SoC Technical Reference Manual. Technical Reference Manual UG585 (v1.11), Xilinx, 2016.