View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

An Implementation of a Predictable Cache-coherent Multi-core System

by
Paulos Tegegn

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

© Paulos Tegegn 2019

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 2 and 3 has been incorporated within a paper that has been submitted for publication. The paper is co-authored by Kaushik, A. M., my supervisor Patel, H., and myself. In Chapters 6, Zhuanhao Wu aided in implementing the tethered system.

Abstract

Multi-core platforms have entered the realm of the embedded systems to meet the ever growing performance requirements of the real-time embedded applications. Real-time applications leverage the hardware parallelism from multi-cores while keeping the hardware cost minimum. However, when the real-time tasks are deployed on the multi-core platforms, they experience interference due to sharing of hardware resources such as shared bus, last level cache and main memory. As a result, it complicates computing the worst-case execution time of the real-time tasks. In this thesis, I present a hardware prototype that implements a predictable cache-coherent real-time multi-core system. The designed hardware follows the design guidelines outlined in the predictable cache coherence protocol. The hardware uses a latency insensitive interfaces to integrate the multi-core components such as the processor, cache controller and interconnecting bus. The prototyped multi-core hardware is synthesized and implemented in a low-cost and high-performing FPGA board. The hardware is validated and verified on a tethered system that enables the design to run multi-threaded pthread applications.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Professor Hiren Patel for his relentless patience, guidance, and training over the past two years. Your help has been invaluable. I thank my readers Professor Nachiket Kapre and Mahesh Tripunitara for reviewing this thesis.

I thank my colleagues and friends Anirudh M. Kaushik, Zhuanhao Wu and Nivedita Sritharan in the Computer Architecture and Embedded Systems Research group at University of Waterloo for the guidance, support and expert advice.

I thank my family for providing me with unfailing support and continuous encouragement throughout my years of study. Thank you.

Thanks to all my friends here at Waterloo.

Dedication

For my family

Table of Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Contributions 2

1.2 Overview 2

2 Related Work 4

2.1 Real-time systems 4

2.2 Interconnect Architecture 5

3 Background 6

3.1 Hardware Cache Coherence 6

3.1.1 Conventional Coherence Protocol 8

3.1.2 Predictable cache coherence for real-time systems: PMSI 10

4 System Overview 15

4.1 Baseline System Model 15

5 Implementation **17**
5.1 Hardware Design . . . 17
5.1.1 Processor . . . 20
5.1.2 Cache Controller . . . 21
5.1.3 Snooping Bus . . . 27
5.1.4 Memory Controller . . . 30
5.2 Access to DDR3 Memory . . . 33

6 Testing Hardware Environment **34**
6.1 Development FPGA Board . . . 34
6.2 Dual-Core Top Design . . . 35
6.3 Tethered System . . . 35
6.4 RISCV Core . . . 36
6.5 Handler . . . 37
6.6 Communication Channels . . . 37
6.6.1 GPIO . . . 38
6.6.2 Shared Memory on DDR3 DRAM . . . 38

7 Evaluation **40**
7.1 Test Applications . . . 42

8 Conclusions and Future Work **44**

References **45**

List of Figures

3.1 Snooping based cache coherence example. 8
3.2 MSI cache coherence protocol with stable states and respective transitions. 9
3.3 PMSI Architectureal support 11

4.1 Baseline Multi-core System Model 16

5.1 Valid and Ready handshake interface. 18
5.2 Input and output ports of the Data Array interface 19
5.3 Basic interface ports of the FIFO design 20
5.4 The processor interfaces and the message format of the cache-request and cache-response channel. 21
5.5 Cache controller interfaces and the message format of memory-request and memory-response 22
5.6 A sample write data path diagram of the direct-mapped cache. 24
5.7 Cache Controller three stage state machine 25
5.8 Snooping bus 28
5.9 Memory controller 30
5.10 Memory controller state machine 31
5.11 Content Addressable Memory 32
5.12 DDR3 interface diagram. 33

6.1 Tethered System 36
6.2 Address mapping of RISCV run time to handler virtual address 39

List of Tables

3.1 Cache coherence problem example when core c_1 and c_2 access data at address A. "A:1" indicates that address A has value of 1. And the column tells whether it is in the cache or main-memory. Note that if the entry is empty in the cache column, it means no data is cached. 7
3.2 Optimized MSI Snooping Protocol - Cache Controller [20] 12
3.3 Optimized MSI snooping protocol of the memory controller [20] 13
3.4 Additional states and transitions in PMSI [10] 13
3.5 PMSI coherence table [10] 14

5.1 Cache controller event encoding. Note that ID denotes the Core ID number. 23
5.2 Enumeration of all instructions that the cache controller can perform on the cache and FIFO modules. 25
5.3 Cache coherence table logic for cache hit event. U represent the pending store operation after refill of the data line. 26
5.4 Cache coherence table logic for cache miss event. oldAddr is the evacuated cache line address due to the new address request. 27

7.1 Total resource utilization 40
7.2 Resource utilization per components of the dual-core top module. 41
7.3 comparison our RISCV core to the sodor 42
7.4 Worst case latency of memory request of SPLASH2 benchmarks 43

Chapter 1

Introduction

Modern real-time systems comprises multiple tasks with different timing and performance demands. These tasks may share the common hardware resources that they are deployed on. Several certification standards require that high critical tasks never exceed their temporal guarantees. if these tasks exceed their temporal requirements, it will lead to catastrophic consequences. For instance, the Anti-lock Braking System (ABS) of the automotive must finish before the asserted deadline, otherwise it will lead to deadly consequences.

As the real-time task performance requirement increases, multi-core platforms have gained interest to meet the demand trend. leveraging the multi-cores provides hardware parallelism that real-time applications can exploit while reducing the hardware cost. However, when real-time applications are deployed on multi-core platforms, It raises several issues. One problem is it makes the process of analyzing the worst case execution time of the tasks difficult because the off-the-shelf multi-core platforms are curated for average case performance. And when these real-time tasks run on the multi-cores, they may experience varies interference due to sharing common hardware resources such as last-level caches, main memory. The other challenge is successfully leveraging the parallelism of multi-cores while guaranteeing predictability. For example, some platforms such as Kalray's multi-core platforms do not support hardware cache coherence. that means it does not allow exchanging data with in many-core processors that will allow to use the available processing power. This results prohibiting parallel execution of task.

There has been limited research on bounding the interference when different tasks exchange shared data. Particularly, predictably exchanging shared data between tasks while delivering high-performance. The reason is many works assume tasks do not share data [9, 12, 6]. However, for tasks that do share data, designers use software techniques such

as enforcing non-caching of shared data, data duplication in the main memory [11], selective partitioning of the data to ensure predictability of the tasks [5, 8, 11, 6]. This places a burden on the designer to invest efforts in the real-time application and programming model to achieve predictable data sharing. Furthermore, these approaches pay a significant price on performance, and strain memory requirements and cost of real-time application deployment.

PMSI [10] proposed a set of design guidelines to enable predictable data sharing via hardware cache coherence. This technique enabled multiple copies of shared data to simultaneously exit in the private caches of multi-cores while guaranteeing predictable data sharing. Furthermore, enabling predictable data sharing through cache coherence gave significant performance advantages (up to 4X and 71% on average) over prior real-time approaches without the need to make changes to the application to identify shared data.

1.1 Contributions

This thesis proposes a hardware implementation of a predictable cache coherence protocol for multi-core real-time systems. The contributions are as follows:

- A hardware prototype that implements the first predictable cache coherent real-time multi-core system.
- A latency insensitive interfaces that is used in the cache coherence policy for data exchanging between multi-core components.
- The RTL of the multi-core hardware implementation is synthesized and verified on small size FPGA board (Pynq).
- A tethered system where the multi-core hardware is enabled to run multi-threaded pthread applications.

1.2 Overview

The remainder of this thesis is organized into several sections as follows: Chapter 2 discusses the related work in real-time multi-core systems and interconnect protocols. chapter 3 provides a background on a conventional cache coherence protocol in a multi-core system

as well as a predictable cache coherence protocol. Chapter 4 gives the system model of the designed multi-core hardware and how it is mapped in the FPGA hardware. Chapter 5 details the cache coherent hardware design, its components and the latency insensitive interface used to integrate the components. Chapter 6 describes the tethered system created in the FPGA hardware, and the support for atomic instructions in the core and cache controller in order to run pthread applications. Chapter 7 shows the results of the designed resource utilization and the used test applications to validate the correctness. Finally, Chapter 7 provides a conclusion and description of future work to improve the system.

Chapter 2

Related Work

2.1 Real-time systems

Multi-core real-time systems is composed of components such as interconnects, main memory and last level caches that are shared among cores. When a core tries to make access to these shared components, it could experience interference from other cores. This will result in non-trivial worst case timing analysis of real-time tasks. Prior works followed several approaches to achieve predictable data sharing.

1. The first approach is by disabling caching of exchanged data in private caches [9, 12, 6]. This achieved bounded latency in accessing shared data at the cost of reduced average cache performance. The reason is it is not benefiting from low latency private cache hits.

2. The second way to attain predictability is by replicating the exchanged data [11]. This approach minimizes the opportunity to execute tasks in parallel.

3. The other approach is by scheduling tasks that share data on the same core by applies support on the OS layer [5, 8, 11, 6].

These approaches obtain predictable data sharing at the cost of reduced average-case performance. Similarly, Pyka et al. [16] modified the applications to use software locks when tasks are accessing the same data. However, only one core were able to have access to the communicated data at any time instance.

Hassan et al. [10] proposed PMSI, a predictable cache coherence protocol that was built on a set of general design guidelines to realize predictable data sharing through hardware cache coherence. PMSI significantly improved average-case performance of tasks compared to prior real-time approaches while maintaining predictability.

2.2 Interconnect Architecture

There has been different open source interconnect standards [22, 18] published to be used in architecting system-on-chip designs. AMBA ACE [22] is one standard that can be used in integrating components in SoC. TileLink [18] is the new open standard that came with the popularity of RISCV ISA. Tilelink is an interconnect protocol used in cache coherence data transactions. cache coherence transactions are the data transactions used in implementing a cache coherence policy. The purpose of TileLink is to isolate the design of on-chip interconnects and the implementation of cache coherence protocol in the cache controllers. Tilink is the closest interconnect standard that can be used in designing predictable multi-core system.

Chapter 3

Background

This chapter introduces the principle of cache coherence in a multi-core system. I will use a conventional cache-coherence protocol for illustration. This will provide the reader with the necessary background to discuss the PMSI cache coherence protocol [10], which this thesis implements in hardware.

3.1 Hardware Cache Coherence

All cores in modern multi-core platforms have access to data in the main memory, and employ a private cache. The reason for using a private cache is to utilize spatial and temporal locality to improve application performance. The task running on the core can access exclusive or shared data from main memory. A data access is called exclusive when only one core accesses the data, and shared access when more than one core access the data. If the task running on the core accesses the data exclusively, it sees the correct view of the memory and benefits from the private cache due to low access latency. However, if it accesses shared data, it could see different values of the same data than what the other cores see. Table 3.1 shows the problem. Suppose there are two cores c_1 and c_2 accessing the data at address A in main memory. First, c_1 reads the data at address A. Then, core c_2 loads the same data. At this point both cores have the same data for address A. Later, core c_2 modifies address A, which makes the cached data in core c_1 stale. When c_1 reads address A again, it will read the old value. This results in incorrect execution of the parallel program. Hence, all cores must have the same view of the shared data to guarantee correct program execution. Hardware cache coherence is a mechanism that guarantees coherent access to shared data.

Event	c_1 Cache	c_2 Cache	Main-memory
			A:0
c_1 Load:A	A:0		A:0
c_2 Load:A	A:0	A:0	A:0
c_2 Store:A=1	A:0	A:1	A:1
c_1 Load:A	A:0	A:1	A:1

Table 3.1: Cache coherence problem example when core c_1 and c_2 access data at address A. "A:1" indicates that address A has value of 1. And the column tells whether it is in the cache or main-memory. Note that if the entry is empty in the cache column, it means no data is cached.

Hardware cache coherence is a set of rules that data exchanging components follow such that the shared data is always coherent. Each cache and the memory employ a state machine controller called cache controller and memory controller respectively. These controllers implement the set of rules. These set of rules, also known as the cache coherence protocol, consists of the states that represents the read/write access permissions of the data and the transitions between states with its corresponding triggering events that occur due to core's memory activity. These coherence controllers exchange messages with each other through snooping or directory based interconnect to maintain coherent view of the main memory. This work chose snooping based interconnect because it provides low-latency coherence message transactions and simpler design for implementation in a small FPGA board. The granularity of data exchanged between cores and shared memory is in cache line size.

The following example demonstrates the basic principle of cache coherence by using Figure 3.1. This example uses core c_0 accessing a cache line Z. Suppose core c_0 issues a store request to cache line Z ①. Core c_0's cache controller checks if cache line Z exists in the private cache ②. If it was a cache hit, the controller would perform a write operation into the line and acknowledge the processor marking the request completed. If it is a cache miss ③, the controller issues a coherence message on the snooping bus; thereby broadcasting it to all cores and the shared memory asking for the cache line in modified state and waits for a response ④. A coherence message is said to be broadcasted on the bus when all cores and the shared memory observe the coherence message. The shared memory observes the memory request and responds with the data to the requesting core c_0 ⑤. When the cache controller receives the requested cache line from the bus, it fills the line in the private cache, performs the write operation ⑥, and acknowledges ⑦ the core marking the request completed.

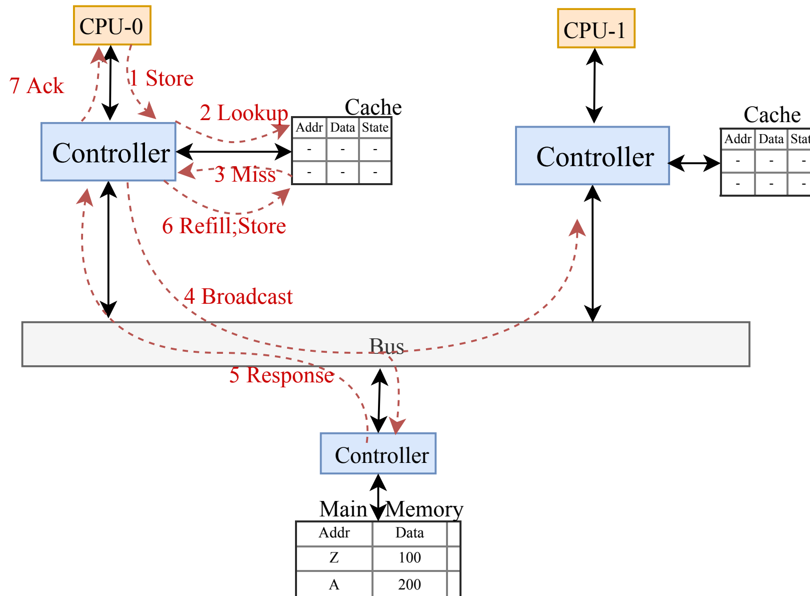


Figure 3.1: Snooping based cache coherence example.

3.1.1 Conventional Coherence Protocol

The Modified-Shared-Invalid (MSI) cache coherence protocol is a basic protocol that several modern cache coherence protocols, such as MESIF, and MOESI protocols, are built upon. As illustrated in Figure 3.2, MSI consists of three **stable states**: (1) *Invalid (I)* denotes the cache does not hold a valid cache line. (2) *Modified (M)* denotes the core has modified and owns the cache line. We refer to a core that owns cache lines as **owners** (3) *Shared (S)* denotes that a core has read-only access to the cache line. More than one core may have the cache line in S state. The objective of the cache coherence protocol is to enforce single-writer-multiple-reader (SWMR) invariant at all times [20].

The coherence messages are the triggering events that walk the cache line state through the coherence protocol. Owner or remote core's memory activities initiate these coherence messages. A cache controller generates a GetM(Z) coherence message when it observes a store memory request from its own processor. GetM(Z) entails the core's intent to keep the line in M once it receives the requested line from the bus. Similarly, GetS(Z) entails the core's intent to cache the line in S after it receives the line. When a cache controller wants to write back a line that exists in M, it generates PutM(Z). When a core issues a store request to a line that is present in the private cache as S, the cache controller generates an Upg(Z). Note that after the bus orders the coher-

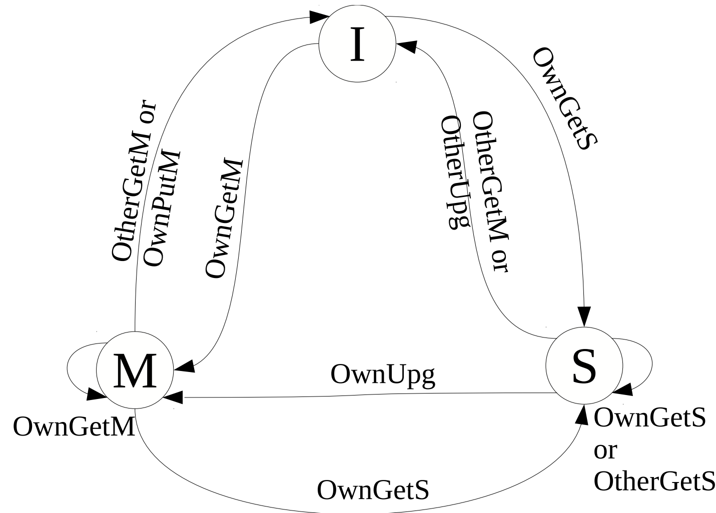


Figure 3.2: MSI cache coherence protocol with stable states and respective transitions.

ence message to all endpoints, the requesting cache controller sees its own requests as OwnGetM(Z)/OwnGetS(Z)/OwnPutM(Z)/OwnUpg(Z), while other cores see the messages as OtherGetM(Z)/OtherGetS(Z)/OtherPutM(Z)/OtherUpg(Z). Transitions between states occur based on the initial state, its own processor memory request, and the coherence message as shown in Figure 3.2. For instance, when a core sees OtherGetM(Z) to a line in S state, it moves to I.

The MSI coherence protocol discussed so far assumes the bus to be atomic. No other memory request is broadcasted on the bus while one core has already ordered a memory request and is waiting the memory response. This simplifies the protocol at the cost of performance. However, high performance coherence protocols use nonatomic transactions to improve bandwidth of the bus. This requires introducing additional states in the protocol.

The high performing MSI coherence protocol introduces **transient states**. These intermediate states allow cores to keep track of interleaving memory requests broadcasted on the bus. A memory request may experience other requests while waiting for permission to broadcast the coherence message on the bus or waiting for a response to already broadcasted coherence message. For example, the IM^{AD} transient state is an intermediate state between the invalid (I) state and the modified (M) state. When a core issues a store request to a cache line that is present in the private cache as (I) state, cache controller will identify it as a cache miss by marking the line as IM^{AD} state. Then, the controller generates a pending write request coherence message and waits for the bus permission to

broadcast it on the bus. Here A indicates the core is waiting to broadcast its coherence message on the bus and D indicates core is waiting for data response. When a core sees its own coherence message ordered on the bus, the controller marks the cache line to another transient state IM^D. Once the core receives the data, it performs the write operation, and updates the line as M state.

I present the full coherence specification of the state MSI protocol in Table 3.2 and Table 3.3. The additional transient states are due to non atomic transaction. As a result, it increased the overall complexity of the protocol.

3.1.2 Predictable cache coherence for real-time systems: PMSI

Although conventional coherence protocols provide cores access to shared data, it does not address the problem of providing bounded latency in accessing the shared data necessary for real-time multi-core systems. PMSI [10] work proposed a template to realize cache coherence protocols that enables sharing data predictably in multi-core systems. The template consists of design guidelines applied to the cache controllers, memory controllers, and the arbitration policy. Cache coherence protocols must adhere to these guidelines in order to guarantee bounded latency for shared data accesses. PMSI is a concrete realization of the proposed template that extends the MSI cache coherence protocol by equipping it with predictable data sharing.

PMSI is an extension of MSI with new transient and stable states, and state transitions. Table 3.4 summarizes the additional coherence states and the state transitions. As shown in Figure 3.3, PMSI adds the following four architectural components that coordinate with the coherence protocol to guarantee predictable data sharing:

1. The pending request lookup table (PR LUT) at the memory controller. The PR LUT records pending requests to cache lines. PR LUT records multiple pending requests to a cache line in the broadcast order. Similarly, memory controller responds to the requesting core in broadcast order.

2. Each cache controller contains two FIFO buffers: pending request (PR) and pending write-back (PWB) buffer. The PR buffer holds request coherence messages waiting for the cache controller to broadcast them on the bus, while PWB buffer holds the pending write-back response generated in the cache controller due to either remote core memory activity or cache eviction due to replacement.

3. Each cache controller services requests, responses, and write-backs in work conserving round-robin arbitration. When a core gets its turn to use the bus, either it services the coherence message in PR buffer or responses from PWB buffer. Furthermore, serving requests in the PR buffer consists either generating request coherence message to the bus or receiving a response from the memory controller to already broadcasted request coherence message.

4. PMSI uses a time-division multiplexing (TDM) bus arbitration policy to avoid starvation cores in accessing the shared resources. TDM arbitrates accesses to shared bus and memory in a predictable manner. The TDM arbitration policy allocates slots fixed time for each core to access the shared bus and the shared memory.

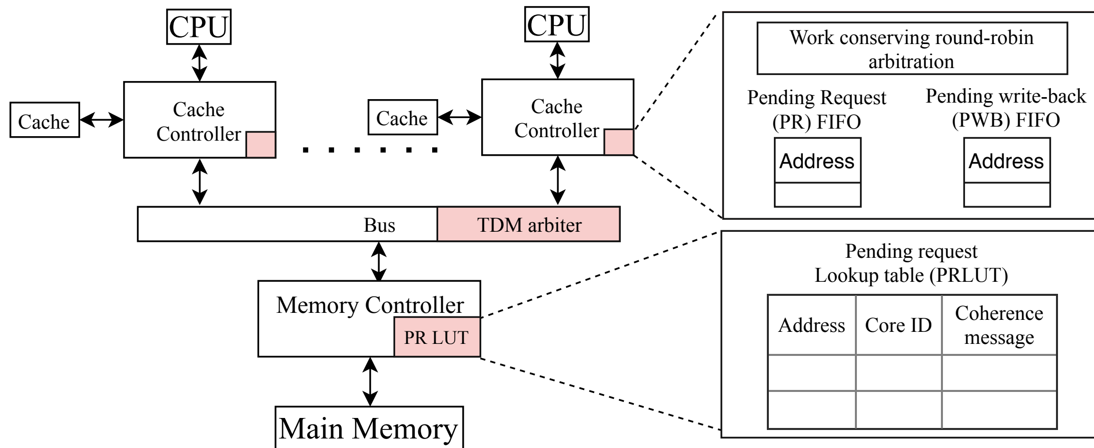


Figure 3.3: PMSI Architectureal support

The following example illustrate an example of unpredictable circumstance and how the PMSI extension avoids it. This sample of interference is between two actions from the same core. The first action is core having a pending request waiting to be issued on the underlying memory. The second is core's pending write-backs to lines cached in modified state due to remote core requests. If we look at conventional MSI protocol, pending request has lower priority than write-back response. As a result, the pending request can be indefinitely postponed because of back to back write-backs due to remote probing. PMSI solves the source of unpredictability by allocating a slot between pending request and pending write-back in round robin. Therefore, the latency of the remote core's memory request is bounded.

T	Load	Store	Replac-ement	Own GetS	Own GetM	Own PutM	Other GetS	Other GetM	Other PutM	Own Data
I	issue GetS $/IS^{AD}$	issue GetM $/IM^{AD}$					-	-	-	
IS^{AD}	stall	stall	stall	IS^D			-	-	-	$/IS^A$
IS^D	stall	stall	stall				-	IS^DI		load hit/S
IS^A	stall	stall	stall	load hit/S			-	-		
IS^DI	stall	stall	stall				-	-		load hit/S
IM^{AD}	stall	stall	stall		IM^D		-	-	-	IM^A
IM^D	stall	stall	stall				IM^DS	IM^DI		store hit/M
IM^A	stall	stall	stall		store hit/M		-	-	-	
IM^DI	stall	stall	stall				-	-		store hit, send data/I
IM^DS	stall	stall	stall				-	IM^DSI		store hit, send data/S
IM^DSI	stall	stall	stall					-		store hit, send data/I
S	hit	issue GetM $/SM^{AD}$	I				-	I		
SM^{AD}	hit	stall	stall		SM^D		-	IM^{AD}		SM^A
SM^D	hit	stall	stall				SM^DS	SM^DI		store hit/M
SM^A	hit	stall	stall		store hit/M		-	IM^A		
SM^DI	hit	stall	stall				-	-		store hit, send data/I
SM^DS	hit	stall	stall					SM^DSI		store hit, send data/S
SM^DSI	hit	stall	stall				-	-		store hit, send data/I
M	hit	stall	issue PutM/ MI^A				send data/S	send data/I		
MI^A	hit	stall	stall			send data/I	send data/II^A	send data/II^A		
II^A	hit	stall	stall			-/I	-	-	-	

Table 3.2: Optimized MSI Snooping Protocol - Cache Controller [20]

T	GetS	GetM	PutM from Owner	PutM from Non-Owner	Data
$IorS$	send data to requester	send data to requester, set owner to requester		-	
M	clear Owner/$IorS^D$	set owner to requester	clear Owner/$IorS^D$		write data to memory/$IorS^S$
$IorS^D$	stall	stall	stall		write data to memory/$IorS$
$IorS^A$	clear Owner/$IorS$		clear Owner/$IorS$		

Table 3.3: Optimized MSI snooping protocol of the memory controller [20]

Design guideline	Coherence protocol extensions
Cores write-back owned cache lines that are requested by other cores in their allocated slots	1) New transient states:MI^{wb} and MS^{wb} 2) New state transitions between M and MS^{wb}, M and MI^{wb} and MS^{wb} and MI^{wb} based on the requesting cores' memory activity
Cores upgrade cache lines in their allocated slots	1) New stable state SM^w that indicates a pending upgrade 2) New state transition between S and SM^w and SM^w and I based on the requesting cores' memory activity

Table 3.4: Additional states and transitions in PMSI [10]

T	Load	Store	Replacement	Own Data	Own Upg	Own PutM	Other GetS	Other GetM	Other Upg	Other PutM	Own GetS	Own GetM
I	issue GetS $/IS^{AD}$	issue GetM $/IM^{AD}$	X	X		X					X	X
S	hit	issue Upg $/SM^{W}$	I		X	X		I	I	X	X	X
M	hit	hit	issue PutM $/MI^{Wb}$	X	X	X	issue PUTM $/MS^{Wb}$	issue PUTM $/MI^{Wb}$	X	X	X	X
IS^{AD}	X	X	X		X	X					IS^{D}	X
IS^{D}	X	X	X	read/S	X	X		$IS^{D}I$	$IS^{D}I$		X	X
IM^{AD}	X	X	X		X	X					X	IM^{D}
IM^{D}	X	X	X	write/M	X	X	$IM^{D}S$	$IM^{D}I$	X		X	X
SM^{W}	X	X	X		write/M	X		I	I		X	X
MI^{Wb}	hit	hit		X	X	send data/I			X	X	X	X
MS^{Wb}	hit	hit	MI^{Wb}	X	X	send Data/S			X	X	X	X
$IM^{D}I$	X	X	X	write/M		X			X		X	X
$IS^{D}I$	X	X	X			X			X		X	X
$IM^{D}S$	X	X	X			X		$IM^{D}I$	X		X	X

Table 3.5: PMSI coherence table [10]

Chapter 4

System Overview

This chapter discusses the main components in the system model and its interfaces.

4.1 Baseline System Model

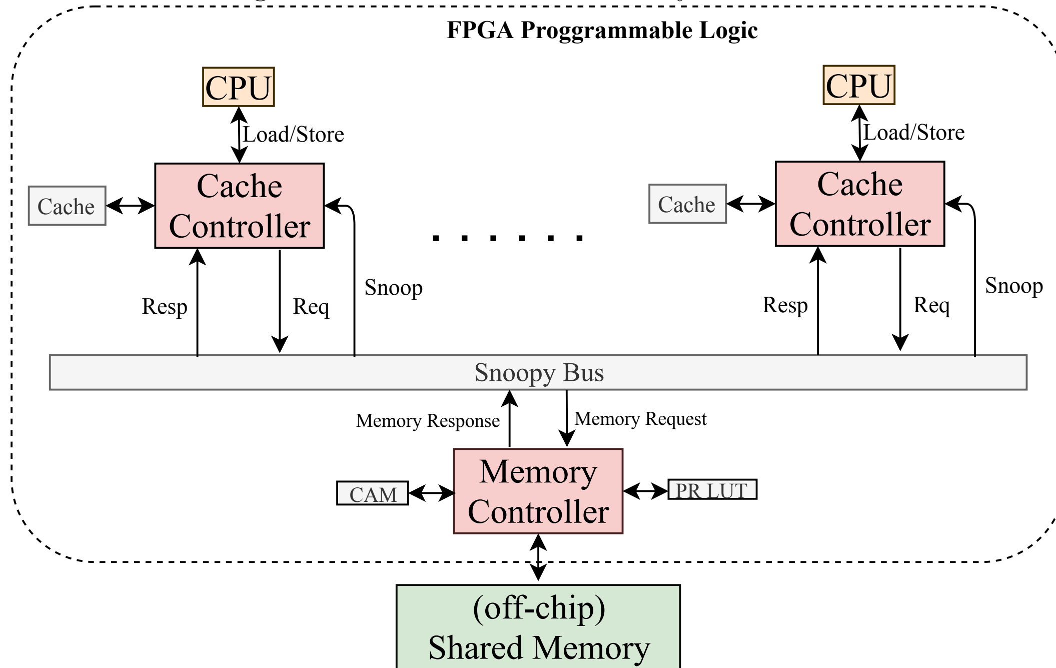
The hardware prototype implementation is based on the shared-memory system model with multi-core processors. The processor is a traditional in-order CPU and has 5 pipeline stages. It executes instructions and issues store and load memory requests to the underlying memory hierarchy. The core is comprised of instruction and data private caches. This private cache helps to lower the memory request latency and increase the memory bandwidth by utilizing the temporal and spatial locality. The private cache used here is chosen to allow implementation in small FPGAs. Hence, it is a write-back, direct-mapped cache that supports a simple replacement policy. However, it can be extended to have high performing caches with higher associativity as well. The cache controller is the unit that services requests coming from processor and coherence requests from the interconnect. Multiple copies of cores are connected by the snoopy bus to the shared memory.

The memory controller is similar to cache controller except it does not generate coherence messages into the snooping bus. The memory controller responds to coherence requests, and keeps track of the overlapping pending requests by using the hash table and pending lookup table modules.

The baseline system is based on the snooping protocol. The idea of snooping protocol is that all coherence controllers, cache controllers, and memory controllers, observe coherence messages in the same order of broadcast and collectively perform tasks in tandem to

maintain coherence [20]. The controller generates the coherence messages due to load and store activities of the processor. The bus acts as an ordering point to coherence messages by making sure all coherence controllers snoop the same series of coherence messages in the same order. The response messages from shared memory to the requesting cores use separate on-to-one interconnect. This is because these response messages do not affect the serialization of the coherence transactions.

Figure 4.1: Baseline Multi-core System Model



The interconnecting shared bus allows the cores in the multi-core system to access the shared memory. It broadcasts the core's load and store requests to all connected endpoints including the shared memory. The bus implements an arbitration scheme to equally share the shared bus among the cores. In order to avoid starvation of any core from accessing the bus, the arbitration scheme needs to be predictable. Time Division Multiplexing (TDM) is one predictable arbitration scheme that avoids unbound interference of remote cores to a given memory request. TDM allocates slots of fixed time duration to each core to access the shared bus. This work uses TDM arbitration scheme because it enables predictable access to the shared bus and it is easier to implement.

Chapter 5

Implementation

This chapter discusses the design of the core components of the multi-core system and the interfacing protocol. This includes the processor, cache controller, the memory controller and snooping bus. Furthermore, it discusses the private cache organization used in the core, how the coherence controllers implement the cache coherence protocol, and the additional modules used in the memory controller to track pending requests.

5.1 Hardware Design

The objective is to design a simple memory hierarchy system that can be synthesized in small-sized FPGA boards and can easily be modified for exploration of different cache coherence protocols. The prototyped hardware is verified on Xilinx Pynq FPGA. The cores and the memory hierarchies are implemented in the programmable logic. The off-chip SDRAM is used as main memory. Since the programmable size of Pynq is small, the core design size must be small enough to fit multiple copies of it in the fabric. Therefore, this work chose RISCV processor with RV32UI target. These processors are in-order and have 5 pipeline stages.

Interface Architecture

The implemented design extensively uses the latency insensitive **Valid Ready** handshake protocol to connect modules. The modules follow this handshake protocol to talk to one another whether to send and receive data. The diagram in Figure 5.1 illustrates the one

directional handshake specification. The **module A** and **module B** are connected to each other. **Module A** is the source module that outputs the data. **Module B** is the destination module that receives the data. The two modules are connected through three signals. **Data** is the wire that passes the actual data from the source to destination. I will use message or packet to denote the data transmitted. **Valid** and **Ready** are the handshaking wires, which allows the source and destination to communicate when it is time to pass the data through the **Data** wire. When **Valid** is high, it tells the source module has asserted a valid data on the **Data** wire. When **Ready** is high, it tells that the destination module is ready to accept data. if both **Valid** and **Ready** are asserted, data has been exchanged. The one way connection interconnect between source and destination module is called a channel. Two modules can have more than one channel. From this time on wards, I will ignore the **Valid** and **Ready** control wires, and only show the **Data** wires with the direction to denote the channel.

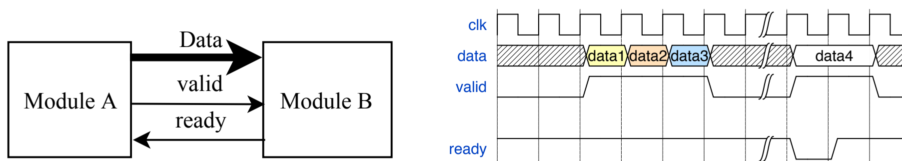


Figure 5.1: Valid and Ready handshake interface.

Cache

The private cache is the module where core stores the cache line data and its state. The controller performs read and write operations on the private cache. The cache is reconfigurable to allow variable cache line size and number of cache line entries. In this work, the private cache is set to **1k bytes direct mapped** data cache. Each cache line is **64 bytes**, with **16 lines** per cache. The private cache has **Data Array** and **Tag Array** sub units.

The **Data Array** is a dual port simple memory and synthesized as block RAM. The Data Array design interface is shown in Figure 5.2. The inputs and outputs of the design follow the timing standard of the BRAM interface. The read and write timing is shown on the right side of Figure 5.2. BRAM reading timing shows that the data is returned on the **read_data** wire after one cycle the address is asserted in the **addr** wire. Similarly, in the writing timing, when the data and address are asserted on the **write_data** and **write_addr** wire, the data is written on the address location one cycle later. This implies the Data Array can accept a new read or write request every cycle.

The Tag Array has similar top level design interface as the Data Array. However, because the Tag Array memory requirement is less than what a minimum block RAM offers, it is implemented as distributed RAM. So, the read timing is different. The read port timing has no latency limitation, while the write operation takes a cycle. Additionally, the Tag Array has another read port with different clock. This port is used for debugging purpose. The debug port allows to see the states of all cache lines.

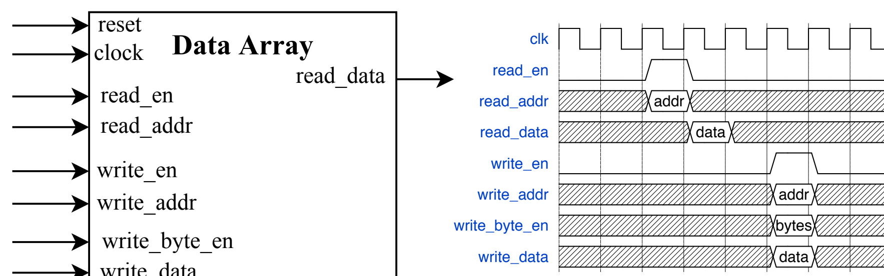


Figure 5.2: Input and output ports of the Data Array interface

The cache is designed to be extended to other types of cache by replacing it with higher associativity and larger sized caches with out affecting the cache controller functionality.

FIFOs

FIFOs are one of the building blocks used in the design. It allows different components to asynchronously operate independent of each other. It is a module where a data enters on the input port and leaves on the output port. The number of entries that the FIFOs can buffer before it gets full tells how deep the FIFO is. The FIFO width is the width of the data that enters and leaves the FIFO. The FIFO width and depth are parameterized. The FIFO design uses distributed RAM to store the data because most of the instantiated FIFOs require relatively less memory. The top level design interface is shown in Figure 5.3. Because the read output is configured to have no registers, the read timing is latency insensitive. It also has two more wires to give information whether the FIFO is full or empty. The surrounding logic should never read when the FIFO is empty and write when it is full.

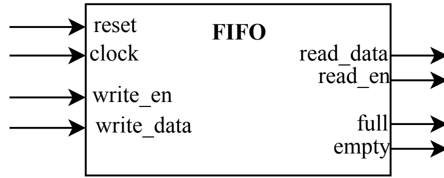


Figure 5.3: Basic interface ports of the FIFO design

5.1.1 Processor

The processor implements a 5 stage RISCV based instruction set architecture (ISA). RISCV is an open hardware ISA which is gaining a lot of attention because of many features: a fewer instruction set, hence the area and power usage will be less; and the RTL can be provided if the industry requires certifications. The design implements **RV32IA** RISCV target ISA [24, 25]. **RV32I** stands for the base instruction set with 32 bits wide instructions and registers, and **A** denotes the support for atomic instructions. The processor is in-order, that means it can only issue one memory request at a time.

The RISCV processor is the central processing unit that initiates data transactions. It issues requests to the cache controller to perform load, store and atomic instructions. Figure 5.4 shows the interfaces of the core module. The processor issues a memory request operation through **cache-request** channel, and receives the response on the **cache-response** channel. Once the processor issues a memory request, it will wait for the response to its request.

The list of memory operation that the processor orders on to the cache controller are as follows.

- **Load and Store**: read and write operations at the specified *address* in the main memory. It is encoded in **cache-request** *mem_type* bit. The packet is a load type when *mem_type=1* and store type when *mem_type=0*.
- **Load-Reserve (LR)**: Load a word from the specified *address* and reserve the *address* in registers.
- **Store-Conditional (SC)**: Store a word on the specified *address* if it matches with the reserved address.
- **AMO operations**: Atomically load a data from the specified *address*, modify the data based on the *amo_alu_op*, then store the result back. Swap, add, logical OR,

logical XOR, and signed and unsigned interger maximum and minimum are the supported AMO operations encoded in *amo_alu_op*.

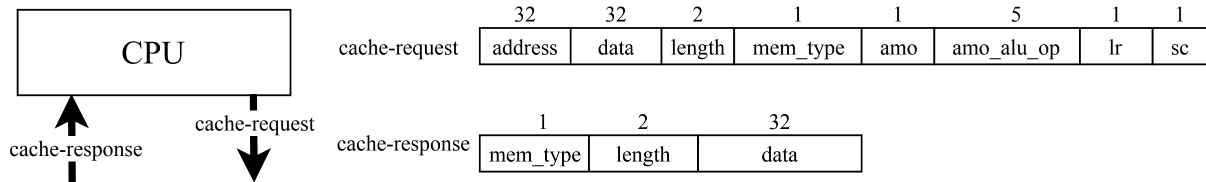


Figure 5.4: The processor interfaces and the message format of the cache-request and cache-response channel.

5.1.2 Cache Controller

The cache controller is a fundamental component of the memory hierarchy. It services memory requests coming from its own processor and coherence requests generated from the remote core. The controller is prototyped as a state machine that realizes the coherence protocol. As depicted in Figure 5.5, the controller module interfaces with the core, the cache, and the snooping bus. The controller inherently uses FIFO buffers to asynchronously operate on multiple requests.

The cache controller has three input channels:

1. **Cache-Request**: This is the channel where the cache controller receives memory requests from the processor. As described in Processor section, load, store, LR/SC and AMO operations are the type of memory request supported.
2. **Memory-Response**: This is the channel where the controller receives memory responses of the requested cache line from main memory through the bus.
3. **Snoop**: This is the channel through which the bus tells the processors what memory request message is broadcasted on the bus.

The controller also has two output channels:

1. **Cache-Response**: This is output channel through which the controller replies response messages to the requesting processor.

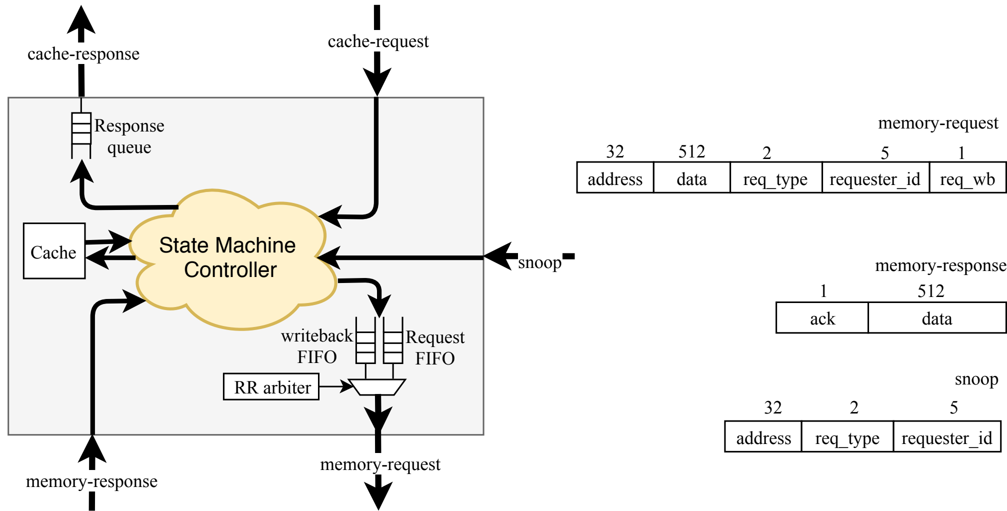


Figure 5.5: Cache controller interfaces and the message format of memory-request and memory-response

2. **Memory-Request**: The controller propagates processor memory requests to the underlying bus through memory-request channel.

Although there are three input channels, the cache controller can only process one packet at a time. The cache controller gives priority from high to low in the following order: **cache response**, **snoop**, and **cache request**. The cache controller must assert *valid* and drive the request data on the data wire when it wants to tell the bus its intent to send a memory request. This is because the bus uses the signal to perform work-conserving arbitration between accepting a request from the controller or responding to pending request by replying the data. The cache controller uses the *valid* signal from the three input channels and assign the *ready* signal to the highest priority channel. Note that a combinational loop in the *valid ready* hand shake signal can never happen because the bus pieces of logic that drive the *valid* signal on **snoop** and **memory-response** channels do not check the corresponding controller's *ready* signal. Also the processor designed in a way so that it does not use the **cache-request** *ready* signal to drive the *valid* signal.

The packets that the controller receives from the three input channels are encoded into cache events. Table 5.1 shows the encoding of packets coming from input channels into **cache events**. Every event has associated address. Events may also have additional arguments depending on the event type. For example, **store** event has the data *word*

argument.

Input channel	Condition	Event
cachere request	mem_type = 1	Load(Addr)
	mem_type = 0 or is_amo = 1	Store(Addr, word)
	is_amo = 1	Store(Addr, word)
snoop	req_type = GetS(Addr) & requester_id = ID	OwnGetS(Addr)
	req_type = GetM(Addr) & requester_id = ID	OwnGetM(Addr)
	req_type = Upg(Addr) & requester_id = ID	OwnUpg(Addr)
	req_type = PutM(Addr) & requester_id $\neq$ ID	OwnPutM(Addr)
	req_type = GetS(Addr) & requester_id $\neq$ ID	OtherGetS(Addr)
	req_type = GetM(Addr) & requester_id $\neq$ ID	OtherGetM(Addr)
	req_type = Upg(Addr) & requester_id $\neq$ ID	OtherUpg(Addr)
	req_type = PutM(Addr) & requester_id $\neq$ ID	OtherPutM(Addr)
memory response	data	MemData

Table 5.1: Cache controller event encoding. Note that ID denotes the Core ID number.

The **cache-response** and **memory-request** are the channels through which the controller responds to the processor and the remote core coherence requests respectively. Both channels uses FIFOs to enable the cache controller processes requests asynchronous from the core and the bus. A small piece of logic, which is independent from cache controller, drive the packets into to output channels.

Following the PMSI [10] template realization, two FIFOs and a multiplexer are used to allow arbitration between request and write-back on the **memory request** channel. As shown in Figure 5.5, the multiplexer arbitrates the **memory-request** channel to the the request and write-back FIFOs in work conserving round-robin fashion.

The private cache organization that is used in the design is direct-mapped cache. The advantages is that it is simple to implement and verify, and reading a cache line from the Data Array only takes a cycle. The drawback is that the cache will have high cache miss rate compared to higher associative caching schemes. The cache design can be extended to a other schemes such as associative caches.

The cache unit allows separate read and write operations on the Tag and Data Array. An example of the Data Array write operation on cache hit case is shown in Figure 5.6. During cache write, the address is split into three parts: tag, index, and offset. The index and offset goes to the Data Array while only the index goes to the Tag Array. The tag

read from the tag array is compared to the tag of the requested address. In another cycle, the write state machine controller writes the registered store data into the data array.

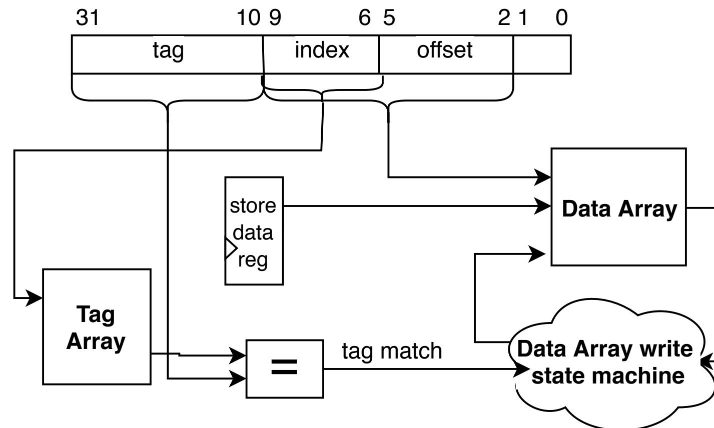


Figure 5.6: A sample write data path diagram of the direct-mapped cache.

Cache Coherence Protocol Implementation

The cache controller takes three steps to process the incoming packets from the input channels.

1. When the cache controller receives a packet from the input channels, it encodes the message into an event based on the Table 5.1. The controller also reads the data and Tag Array to determine the cache state and tag match status.

2. Using the event and the cache state, the controller looks up the **coherence-table** module to determine the list of operations. The **coherence-table** module is a combinational logic that determines the state that performs the list of instructions for a given event and cache state. The **coherence-table** returns the starting state in the state machine that executes the list of instructions.

3. The controller execute the list of operations by walking through the controller state machine.

The possible operations that the cache controller can take is divided into two:

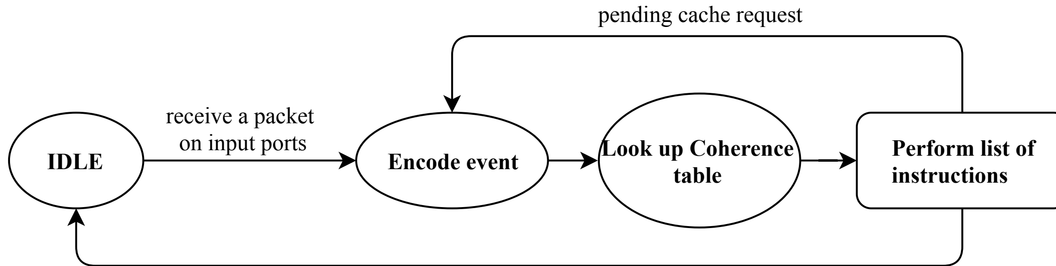


Figure 5.7: Cache Controller three stage state machine

1. Read and write operations on the cache unit.
2. Push packets into request, write-back and cache response FIFOs.

Table 5.2 enumerates all possible actions per module.

Module	Instruction	Notation
TAG Array	write to tag array of the cacheline at address *Addr*	Wr(Addr, state, dirty, valid)
	read the cacheline's Tag at address *Addr*	Rd(addr, state, dirty, valid)
	fill the data array line at address *Addr* with *data*	WrLine(addr, Data)
Data Array	read cache line data at address *addr*	RdLine(Addr, Data)
	write a *word* in to data array at address *addr*	WrWord(addr, word)
	read a *word* from data Array at address *addr*	RdWord(addr, word)
Cache Response FIFO	push a cache response packet	push(word)
Request FIFO	push a memory request packet (any request type)	push(addr, broadcast_type, core_id)
Writeback FIFO	push a memory request packet (PutM() only)	push(addr, putM, core_id)

Table 5.2: Enumeration of all instructions that the cache controller can perform on the cache and FIFO modules.

The **coherence-table** module is based on the PMSI [10] cache coherence specification, which is shown in Tables 5.3 and 5.4. Table 5.3 shows the coherence logic for cache hit event while Table 5.4 shows the coherence logic for cache miss event. The coherence logic for cache miss event is shown separately because replacements occur only in a few cases, which is during cache stable states. The tables show the list of operations for a given event and cache state. The controller performs the list of operations using states in the state machine. For example, when Load(Addr) event occur on cache state=I, dirty=0, valid=0 and core c_0, indexing **coherence-table** returns two operations: write(Addr, state=IS^{AD}, dirty=0, valid=1) on Tag Array and push(GetM(Addr), core_id=0) on request FIFO.

Event	cache state	TAG Array	Data Array	cache Resp FIFO	REQ FIFO	WB FIFO
Load	I	Wr(Addr, IS^{AD}, 0 , 1)			push(Addr, getM, ID)	
	S		RdWord(Addr)	push(word)		
	$IS^{U}I$	Wr(Addr, I, 0, 1)	RdWord(Addr)	push(word)		
	M			push(word)		
Store	M	Wr(Addr, M, 1, 1)	WrWord(Addr)	push(ack)		
	$IM^{U}I$	Wr(Addr, MI^{A}, 1, 1)	WrWord(Addr)	push(ack)		push(Addr, putM, ID)
	$IM^{U}S$	Wr(Addr, MS^{A}, 0, 1)	WrWord(Addr)	push(ack)		push(Addr, putM, ID)
	S	Wr(Addr, SM^{W}, 0, 1)			push(Addr, Upg, ID)	
	I	Wr(Addr, IM^{AD}, 0, 1)			push(Addr, getM, ID)	
OwnGetM	IM^{AD}	Wr(Addr, IM^{D}, 0, 1)				
OwnGetS	IS^{AD}	Wr(Addr, IS^{D}, 0, 1)				
OtherGetM	M	Wr(Addr, MI^{A}, 1 , 1)	RdLine(Addr)			push(Addr, putM, ID)
	S	Wr(Addr, I, 0 , 0)				
	IM^{D}	Wr(Addr, $IM^{DU}I$, 0, 1)				
	IS^{D}	Wr(Addr, $IS^{D}I$, 0, 1)				
	MS^{A}	Wr(Addr, MI^{A}, 1, 1)				
	SM^{W}	Wr(Addr, IM^{W}, 0, 1)				
OtherGetS	M	Wr(Addr, MS^{A}, 0, 1)	RdLine(Addr)			push(Addr, putM, ID)
	IM^{D}	Wr(Addr, $IM^{DU}S$, 0 ,1)				
OwnUpg	SM^{W}	Wr(Addr, M, 1, 1)	WrWord(Addr)	push(ack)		
	IM^{W}	Wr(Addr, IM^{AD}, 0 ,1)			push(Addr, getM, ID)	
OtherUpg	S	Wr(Addr, I, 0, 0)				
	SM^{W}	Wr(Addr, IM^{W}, 0, 1)				
OwnPutM	MI^{A}	Wr(Addr, I, 0 , 0)				
	MS^{A}	Wr(Addr, S, 0, 1)				
MemData	IM^{D}	Wr(Addr, M, 0, 1)	WrLine(Addr)			
	IS^{D}	Wr(Addr, S, 0, 1)	WrLine(Addr)			
	$IM^{DU}I$	Wr(Addr, $IM^{U}I$, 0, 1)	WrLine(Addr)			
	$IM^{DU}S$	Wr(Addr, $IM^{U}S$, 0, 1)	WrLine(Addr)			
	$IS^{D}I$	Wr(Addr, $IS^{U}I$, 0, 1)	WrLine(Addr)			

Table 5.3: Cache coherence table logic for cache hit event. U represent the pending store operation after refill of the data line.

The cache controller executes the list of operations by walking the states in the state machines. In the above Load(Addr) example, the controller performs the two operations using one state that writes the cache state=IS^{AD} on the Tag Array and pushs the memory request packet with GetM(Addr) and core_id=0 attributes on the request FIFO.

The coherence logic divides some of the transitions between intermediate states. This is because the operation done during the transition requires multiple cycles. As a result, the cache controller takes the three step process more than once for some of the events. For instance, when the controller receives data packet on **memory response** while the cache line state=IM^{D}, dirty=0, valid=0, based on the cache coherence, it needs to perform fill the cache line and write word on Data Array operations. The controller keeps pending request to perform the three step process twice. So, at the end of the step process, the controller checks for the pending request before going to IDLE.

Furthermore, cache fill and store operation during state transition from $IM^{D}I$ to I

Event	Cache State	TAG Array	Data Array	cache Resp FIFO	REQ FIFO	WB FIFO
Load	M	Wr(Addr, MI^A, 0, 1)	RdLine(oldAddr)			push(oldAddr, putM, ID)
	S	Wr(Addr, IM^{AD}, 0, 1)			push(addr, getS, ID)	
Store	M	Wr(Addr, MI^A, 0, 1)	RdLine(oldAddr)			push(oldAddr, putM, ID)
	S	Wr(Addr, IM^{AD}, 0, 1)			push(addr, getM, ID)	

Table 5.4: Cache coherence table logic for cache miss event. oldAddr is the evacuated cache line address due to the new address request.

is divided in to two. U is used to tell the pending store operation after filling the line. The reason to add this new state is the refill and store operation is done in Data Array takes two steps. Although the cache controller enforces performing the two tasks as atomic operation.

Cache Coherence Example in Action

I will use the following example to illustrate the process. Suppose the cache controller receives a packet with addr=A, mem_type=0, and is_amo=0 on cache request channel. (1) based on the encoding table, the controller encodes the packet into Store(A). The controller also reads the tag and Data Array and finds that it is in I state. (2) The controller uses the collected data in the first state, and looks up the **coherence-module**. The **coherence-module** returns the state that performs two tasks: write(state=IS^{AD}) on Tag Array and push(GetM(A), core_id=0) on the request FIFO. (3) The controller walks the state machine executing the two task. Once it completes performing the two tasks, it goes back to $IDLE$ state. When the control signal driver at the **memory-request** channel gets a *ready* signal, it asserts GetM(A) broadcast packet. After the coherence message is ordered on the bus, the controller sees its own broadcast event on the **snoop** channel. The controller encodes the snooped packet into OwnGetM(A) event and traverses through the three step process performing write(state=IS^D) on Tag Array before going to $IDLE$ state. Finally, the controller receives the data event at the **memory-response** channel. and sends an acknowledgment to the processor through **cache response** channel after it fills the cache line and perform the initial store operation in the Data Array.

5.1.3 Snooping Bus

The snooping bus connects all cores of the multi-core platform and the shared memory. All endpoints on the bus observe load and store requests and coherence messages in the same order they were broadcasted on the bus. In this work, the bus uses TDM arbitration

scheme, and each core will have a dedicated time slot where the bus allocates to cores to access the bus. The arbitration can be extended to support different arbitration schemes.

Coherence request messages are broadcasted on bus to allow all cache controller see the requested message, however, data responses are exchanged between cache controllers and the memory controller.

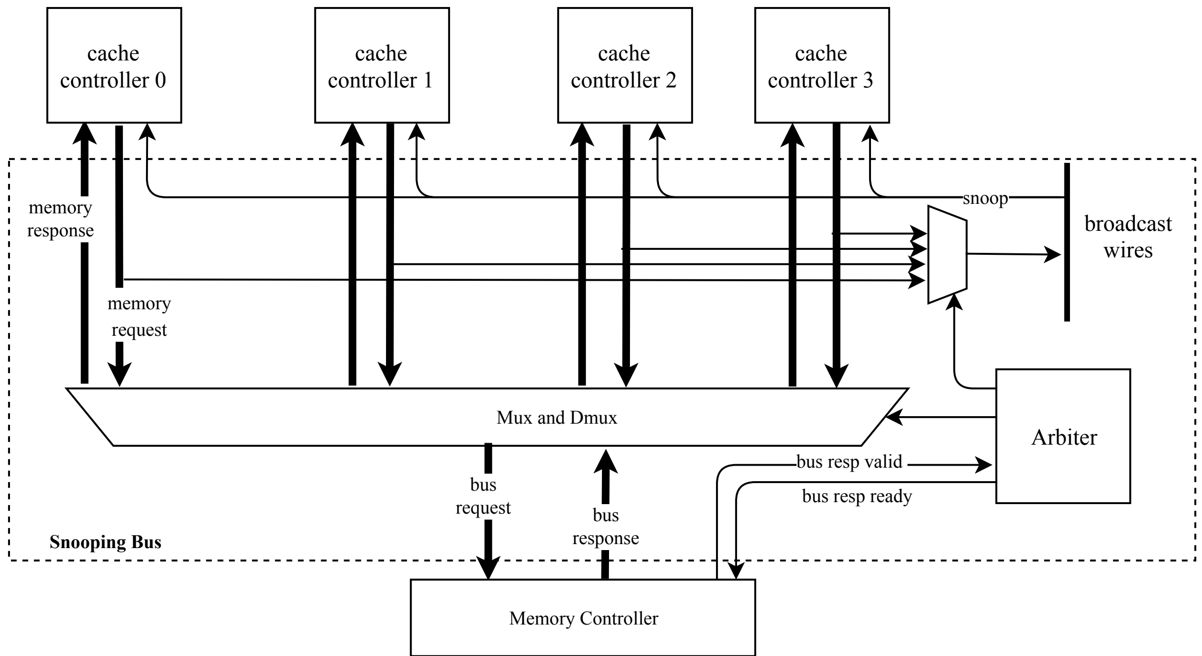


Figure 5.8: Snooping bus

Each core is connected to the bus through three different channels.

1. **Memory-request**:The cache controller issues memory requests GetM(Z)/GetS(Z)/PutM(Z)/Upg(Z) through this channel. Request coherence messages are only broadcasted at the beginning of owner's TDM slot. The bus orders these coherence messages to all end points connected to the bus. GetM(Z)/GetS(Z)/Upg(Z) coherence request messages are generated due to owner cores memory requests, while PutM(Z) is due to remote core memory activity.

2. **Memory-response**: The bus replies to the requesting core with requested data through the **memory-response** channel.

3. **Snoop**: The bus uses the **snoop** channel to broadcast the coherence message to all cores connected to the bus.

In addition, the bus interfaces to the memory controller use two channels.

1. **Bus-Request**: The bus sends memory requests to the memory controller, which is generated from cores. The bus arbitrates accessing **bus-request** channel by using TDM arbitration schemes.

2. **Bus-Response**: The bus receives data for GetM(Z)/GetS(Z) memory requests from the memory controller through **bus-response** channel.

The memory controller provides information about which core has pending responses by using **bus-response** *valid* signals. The **bus-response** *valid* and *ready* wire width is as many as the number of core. For example, in the Figure 5.8, there are 4 cores, therefore the **bus-response** has 4 *valid* and *ready* wires. The bus picks the memory response from the memory controller based on the arbitration scheme by asserting the right *ready* signal. Enabling the bus to see the current pending response in memory controller is important because only the arbiter decides what response to be replied to the requesting core based on the arbitration scheme, and the memory controller does not know anything about the arbitration.

The following example illustrates how the bus orders the coherence messages and reply the response data. Suppose the current cycle is the beginning of core c_1 TDM slot, and core c_1 has a request packet on the **memory-request** channel. The TDM arbiter asserts the ready control signal of **memory-request** channel. Then, it asserts the mux controller wire so that the memory-request drives the wires of bus-request. The state machine in TDM arbiter waits until all of cache controllers and the memory controller receives the requested packet on the snoop and bus-request channels respectively. Once the message is broadcasted, the arbiter waits for the response on the **bus-response** channel by checking the second *valid* wire for the remaining time of the slot. When the memory controller replies the data in the same slot, the arbiter asserts the demux controller so that the bus-response drives the **memory-response** channel of core c_1.

At the beginning of a slot, The TDM arbiter decides whether the slot is request or write-back slot by reading *req_wb* bit range in the **memory-request** packet. When the slot is for memory request, it arbitrate between memory request and data response in work conserving round-robin. Otherwise, it allocates the slot for write-back.

5.1.4 Memory Controller

Memory controller is the coherence controller at the shared memory side. It is similar to cache controller except it does not issue coherence message. The memory controller services GetM(Z)/GetS(Z)/PutM(Z) /Upg(Z) coherence requests from bus. Similar to cache controller, it is implemented as a state machine controller that realizes the memory coherence protocol. Figure 5.10 shows the state machine. Note that the memory controller we refer here is not the DRAM memory controller that performs the reads and writes from the DRAM memory. The memory controller interfaces with the bus through the **bus-request** and **bus-response** channel. It receives coherence requests from the bus through the **bus-request**, and replies the data to the bus through the **bus-response**.

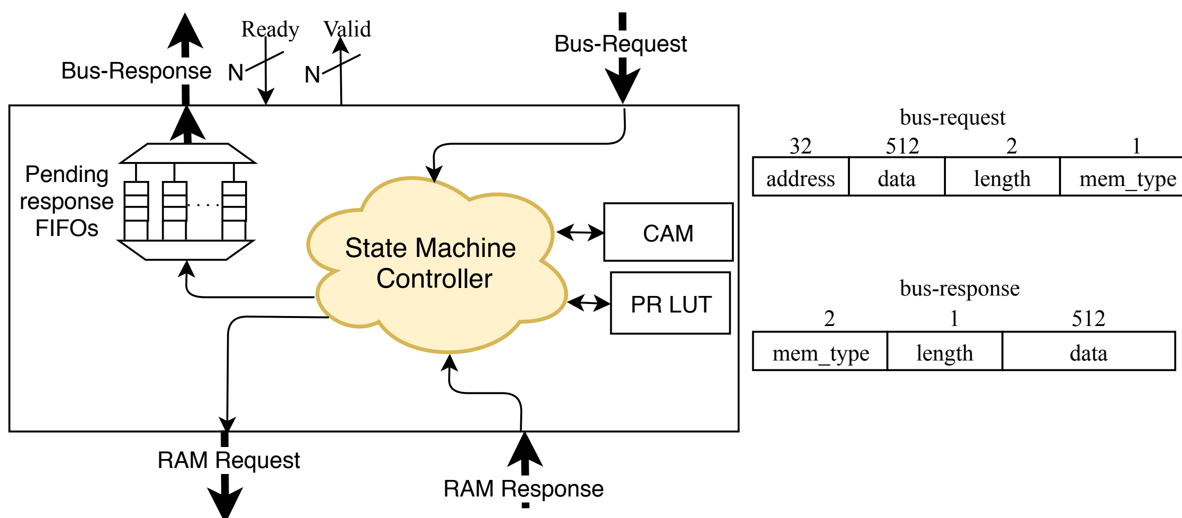


Figure 5.9: Memory controller

As shown in 5.9, the controller has as many *valid* signals as the number of cores on **bus response** channel. The controller uses this *valid* signal to inform the bus about the pending response to all cores. Since there is only one data wire on **bus-response** channel, the bus must tell the memory controller which core's data response to read by asserting the corresponding *ready* signal. And, the controller asserts the correct data response on the data wires based on the asserted *ready* signal. For instance, Suppose we have four cores c_0, c_1, c_2, c_3, and hence there will be four *valid* and *ready* signals on the bus response channel. And suppose there are pending responses to c_0 and c_1. The memory controller drives the *valid* signals corresponding to c_0, c_1. If the bus wants to read the response data

to c_0, It must assert the *ready* signal corresponding to c_0. Then the memory controller drives the response data to c_0 on the data wire of the channel. Note that here there is no combinational loop in the *valid ready* signal because the memory controller logic that derive **bus-response** *valid* signals does not depend on any of **bus response** *ready* signal.

The memory controller also has additional interface to the off-chip memory by using DRAM controller. The memory controller issues read and write operations to main memory through **RAM request** and **RAM response** channels.

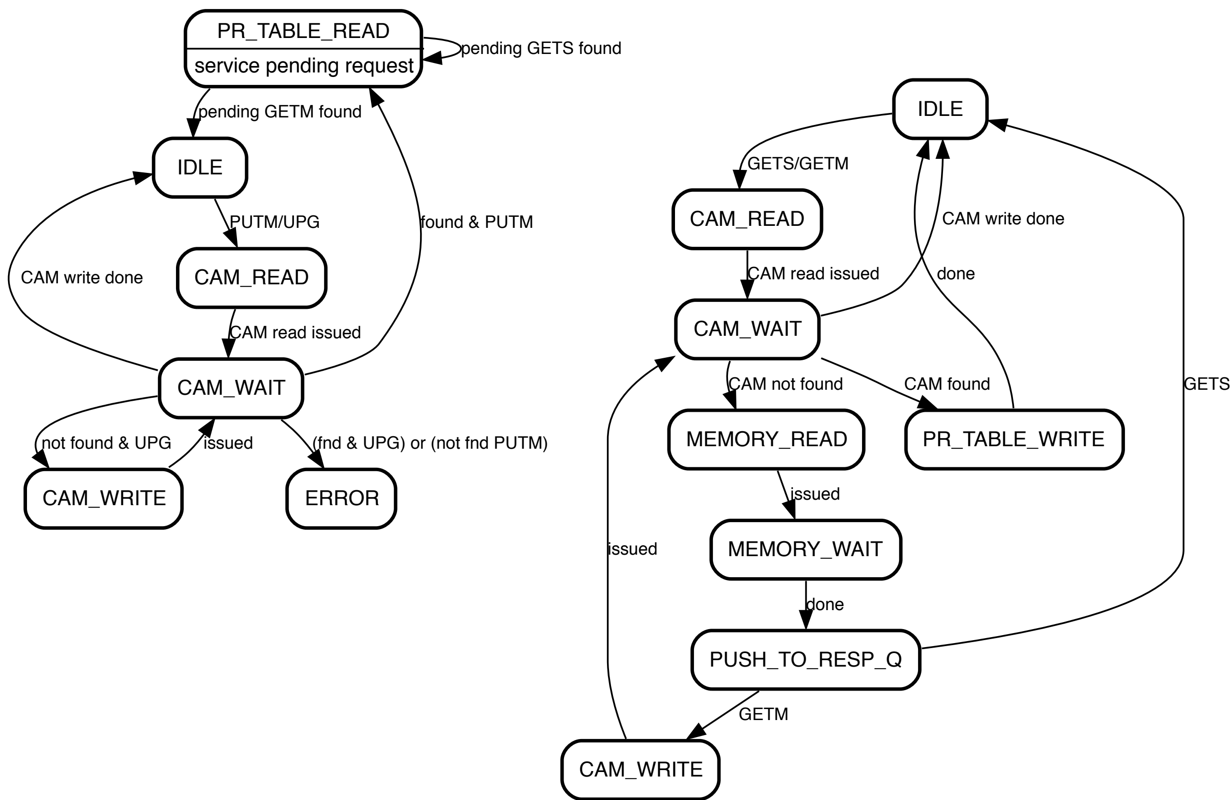


Figure 5.10: Memory controller state machine

Content Addressable Memory

The memory controller keeps information about all the cache lines stored in the cores private caches. The controller instantiates a **Content Addressable Memory** to track the cached addresses. **Content Addressable Memory** (CAM) is used to provide faster

lookup to check if a request address is cached or not. It is capable of performing parallel searching over all stored data in the CAM to find out whether the content exists. If it exists it return the address in the CAM where that content is stored. The content is the cache line addresses. The size of the CAM is large enough to hold all the cacheable memory address in all cores.

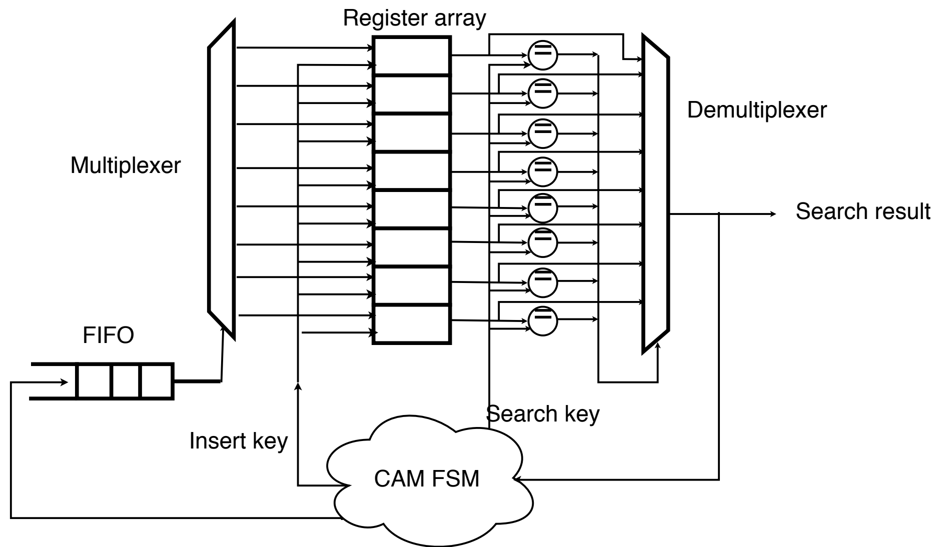


Figure 5.11: Content Addressable Memory

PR LUT

In addition, memory controller uses **PR LUT** module to track pending request per cache line. **PR LUT** is implemented as a full associative cache. The entry size of the full associative cache is same as the number of cores. This is because given that we have in-order processors, in the worst case, there will be one pending request per core where each requested addresses are different. Hence, one entry per core is needed in the associative cache. The other extreme case is when all core request on the same cache line. a FIFO per entry in the associative cache is used to track pending requests on the same address in the order of broadcast.

The memory controller uses **CAM** and **PR LUT** module to track which cache line has been cached in the private caches, and all the pending requests per cache line.

5.2 Access to DDR3 Memory

The multi-core design is built on programmable logic of the Pynq board [27] and uses the DDR3 memory on processing system as the main memory. The design interfaces with the High Performance (HP) AXI slave port, which provides connection from PL to PS to access the DDR3 memory. The HP AXI port has 64-bit data width. So, it requires a streaming interfacing module to convert the 512-bit data coming from the designed memory controller to 64-bit HP AXI port back and forth. The streaming interface is implemented using the HLS. Figure 5.12 shows how the design connects to the DDR3 controller through HP AXI port by using the intermediate module written in HLS.

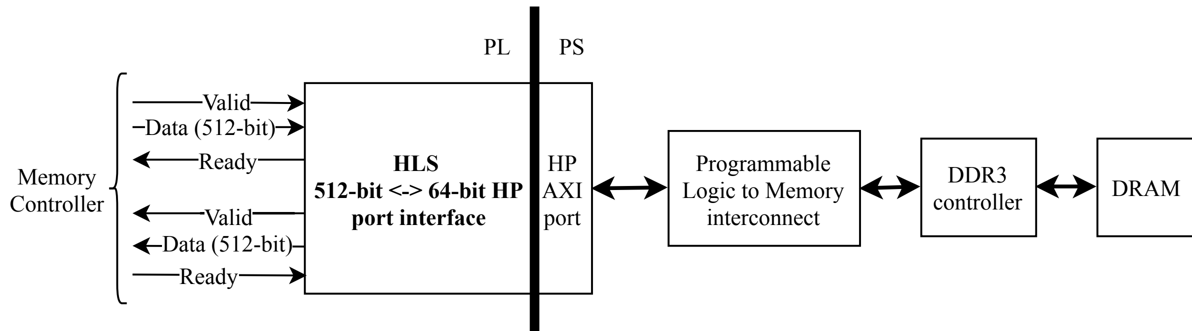


Figure 5.12: DDR3 interface diagram.

Chapter 6

Testing Hardware Environment

The baseline RISCV core is not a standalone processor, so it does not support running an operating system. It depends on another system (host system) to begin program execution, and handle syscalls such as file read and write operations. In order to enable the baseline RISCV core run pthread applications without supporting an OS, the top design must emulate three thread related syscalls: Clone, Futex, and Exit syscalls and atomic memory operations: Load Reserve and Store Conditional, which are used in the pthread application. The design should also handle the remaining general syscalls such as mmap, and brk for correct program execution. This chapter describes the tethered system setup and the extensions on the RISCV core to enable the top design run pthread applications.

6.1 Development FPGA Board

Pynq [27] is a development board based on Zynq-7020 System-on-Chip (SoC) manufactured by Xilinx. Zynq-7020 is composed of the processing systems (PS) and Programmable Logic (PL). The PS has permanently embedded hard components. These components are dual-core ARM Cortex-A9 [3, 2], 512MB DDR3 16-bit bus DRAM, and GPIO peripherals. The ARM core on the PS is a general purpose processor that runs Linux OS. The DDR3 controller is hardcore. So, PL can only access it through AXI slave ports on PS. On the other side, The PL is a Xilinx Artix-7 FPGA, which is used to build custom hardwares. The FPGA has 13,300 logic slices, each with four 6-input LUTs and 8 flipflops. It has 630 KB of block RAM.

The SoC embedded system is designed for software applications that run on the PS side and leverage the customized accelerator hardware on the PL. However, this work uses

a target design on the PL as the main processing unit and a host program running on PS as support system that provides facilities to the target design.

6.2 Dual-Core Top Design

The top design used as a testing platform is a dual-core system. The dual-core design is synthesized on the PL of the board. The top design integrates the two RISCV processors, L1 cache controllers, the snooping bus and the memory controller to build a dual-core system. Each core has 1K bytes data and instruction cache. The L1 caches use direct-mapped scheme with 16 entries of each having 64 bytes cacheline. The interconnecting bus uses TDM arbitration of 128 cycles slot width. The reason why we chose 128 is the latency of accessing the DDR RAM takes approximately 50 cycles, and it takes 10 cycles to propagate the request to DRAM and back to the core. So, the total latency to access uncached data from the DRAM takes 60 cycles. So, the 128 cycles slot width fulfills the PMSI guidelines. Although the designed memory controller does not use caches, it plays a part of the role in maintaining the coherence. The designed memory controller is parametrized to have as many number of total address entries as in the private caches.

6.3 Tethered System

In this work we use tethered top design system to run pthread applications. As shown in Figure 6.1, the tethered system has two components. (1) The dual-core (target) design on PL and (2) the handler program (host) running on the PS. The host helps the dual-core design in running the pthread applications. The baseline RISCV core can only run bare metal applications, and does not support running an OS. So, the riscv core need to emulate certain syscalls to run the pthread application.

The RISCV core traps all of the syscall and handles it either in trap subroutine or delegates it to the host program. The main syscalls involved in thread managements are **clone**, **futex** and **exit**. These are handled inside the RISCV trap routines. In addition, there are general syscalls used in the pthread applications, such as **read** and **mmap**, which are required for correct execution. The target design delegates the these syscalls to the host system. The handler program dispatches the syscalls on behalf of the target top design.

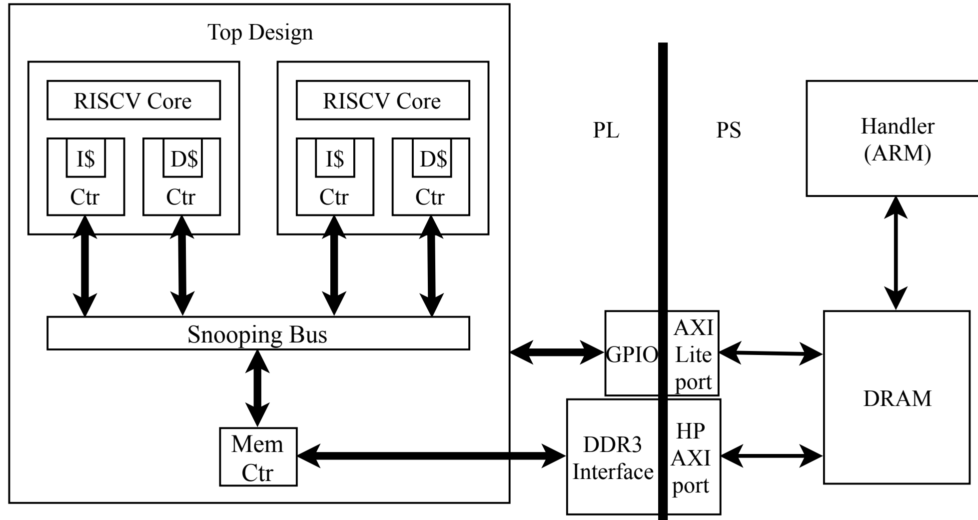


Figure 6.1: Tethered System

6.4 RISCV Core

This target system implements a simple thread scheduler subroutine that assigns threads to cores. Since the tethered system does not support an OS, there is no thread scheduler. So, the RISCV core uses a scheduler subroutine that statically assign threads to core one-to-ones. Therefore, a core can not run more than one thread simultaneously. The RISCV core supports thread management (thread creation, synchronization, and termination) by handling the **Clone**, **Futex**, and **Exit** syscalls in the trap subroutines. So, there are three trap subroutines for each of the syscalls. The core traps these syscalls and redirect them to the corresponding trap handler.

Clone is the syscall that the caller thread invokes to spawn another thread. clone subroutine checks the hardware thread context and picks the first idle core it finds. each core will use specific address values to store their the context to notify their running status. when the core is done executing the spawned thread (reaches exit) and it goes back to idle routine.

Futex is the thread synchronization syscall. The RISCV core supports only **FUTEX_WAIT** and **FUTEX_WAKE** operations. In both cases, futex simply checks that the passed argument is valid and then return. Such implementation is valid because futex is used in locks and futex caller should check the lock status after a **FUTEX_WAIT** operation is done. And since no thread is waiting, **FUTEX_WAKE** operation will always

succeed.

The core handles the three syscalls, and it delegates the rest to the handler program on the PS. When the core delegates the syscalls, it executes instructions that flushes the private cache lines, so that the handler can read the up-to-date data from shared memory. This is a way used to make the handler program coherent with the RISCV core.

The RISCV pthread programs are compiled with the clone, futex and exit subroutines. The trap vector redirects the execution to these subroutines when the traps occur. There are additional start and idle subroutines. All program will begin from the start subroutine. The start routine has two paths of execution. One that goes to the main of the program, and the other goes to idle subroutine. Idle subroutines are a simple while loop read operation to check their thread context if it is assigned a thread to run.

6.5 Handler

Handler is a program running on the PS that provides facilities to the target design. The RISCV core needs these facilities to run RISCV programs. The handler and the top design shares the DDR3 main memory. The handler serves as bootloader. It loads the necessary kernel environment on the RISCV stack address space. It also reads the RISCV ELF binary program and loads it on the RISCV program address space. When the handler finishes bootstrapping, it sets the starting address of the top design and deassert the reset to trigger the dual core top design start executing the loaded program. The handler waits for syscalls by polling from the target design for syscall events. By the time the handler sees a syscall events, it must have up-to-date view of the shared memory for correct emulation of the syscall. So, it polls the state of all lines in the caller private cache and wait until the core finish flushes the lines before dispatching the syscalls. The host program notifies the waiting RISCV core by writing at the global variable **fromhost** address on the shared memory when the handler finishes running the syscall.

6.6 Communication Channels

There are two ways where the handler program and the top design communicates.

6.6.1 GPIO

We instantiate GPIO IPs on the PL to enable the dual-core communicates with the host. These GPIOs modules are assigned some address spaces on the DDR3 RAM. The host program maps the physical address to its virtual address to have access to it. The target system uses these GPIOs to redirect the syscalls to the host, and to pass the syscall arguments. The handler uses it to reset the top design, assign start address of the RSICV cores program counter and check the status of each cache line.

6.6.2 Shared Memory on DDR3 DRAM

The handler allocates a contiguous array of memory on the DRAM to reserve the RISCV address space. It uses the **sds_alloc_non_cacheable**, **sds_mmap** and **sds_munmap** functions in **libsds** library that comes with the **xilinx-linux** on the PS. Usually the RISCV programs does not need all 512MB storage, and the **sds** library has limitation that at max it can only allocate 128MB contiguous array of memory [28]. So, the handler only allocates 128MB for RISCV address space. The handler access these address space by mmaping the physical address to its virtual address space. The dual-core design access the 132MB by using the HLS module explained in the section 5.2. The handler notifies the syscall caller by writing on the global variable **fromhost** address in shared memory. The host reads the global address parsing the RISV program ELF binary. The delegator core waits and reads the the fromhost global variable for completion of the host handling the syscall. The RISCV core avoids reading the stale value of **fromhost** by continuously evacuating the line in private cache and reading it again.

Figure 6.2 shows how RISCV program are mapped in RISCV address space, Handler virtual address space, and the physical memory. The syscall arguments passed to the host are in the RISCV address space. So the host program should be able to work on the same address space as the RISCV address space. Therefore, the RISCV program entry point is carefully selected so that it is available in the host virtual address space. And the designed DDR3 interfacing module uses a **dram_base** offset to convert the RISCV memory request address to physical address.

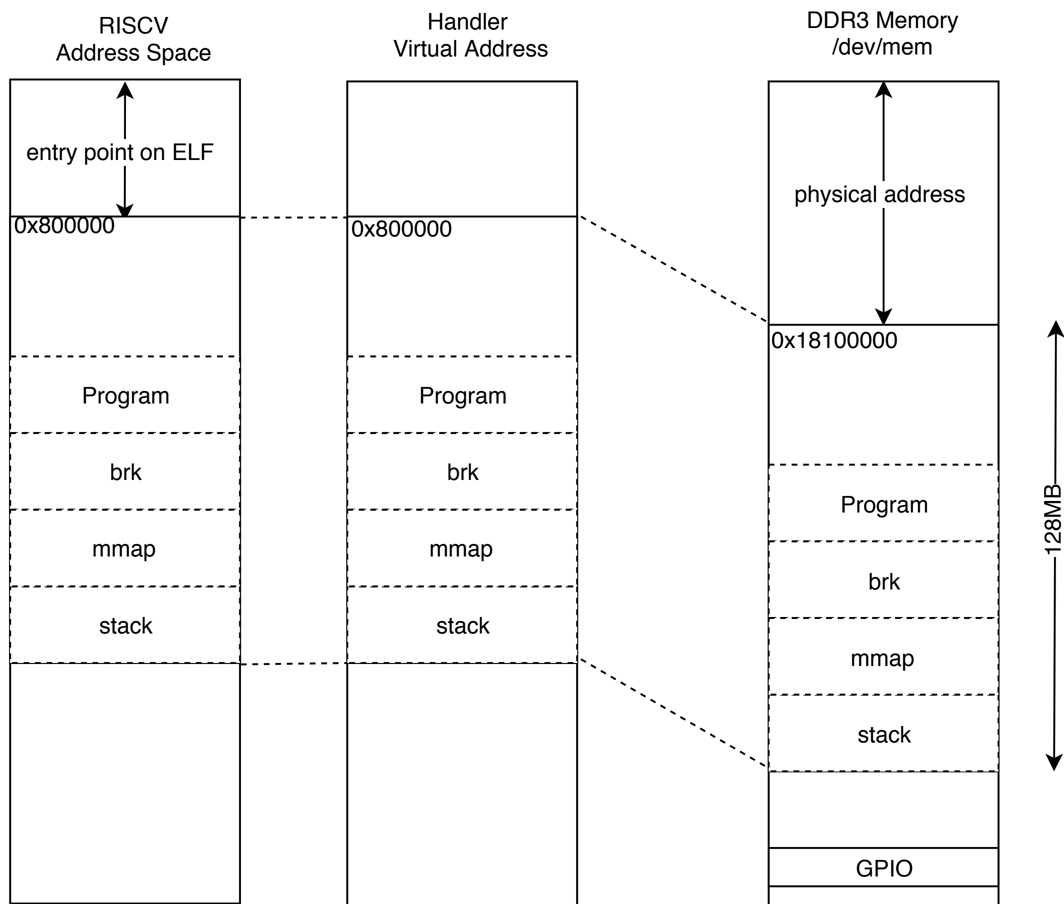


Figure 6.2: Address mapping of RISCV run time to handler virtual address

Chapter 7

Evaluation

The dual-core system is implemented in Verilog. It is simulated and tested on the open source RTL simulator Verilator[19]. The top design is sythesized with Xilinx Vivado 2018.2. SPLASH2 benckmark [26] programs were compiled and set up with gcc-8.2.0 and binutils 2.31.1 on linux machine. The dual-core top design is implemented on a Xilinx Zynq-7020 SoC board. It has 53,200 4-input LUTs, 106,400 flipflops, and 630KB of BRAM. The Vivado is setup to use the default strategy for synthesis and implementation.

Table 7.1 shows the total resource utilization of the design. The result includes the external IPs used to communicate with the Prosessing System unit of the board, generate clocks and debug the top design. These components are the clock wizard, GPIO, AXI interconnect, HP AXI IPs and the debugging IPs (Integrated Logic Analyzer). The dual-core top design only takes 58.84% LUTs, 54.37% LUTRAM and 20.84% FF of the utilized resources. However majority of the BRAM is used by the top design (87.67%). The external components takes the remaining percentage of resource utilization.

Resource	Utilization	Available	Utilization %
LUT	45943	53200	86.36
LUTRAM	8917	17400	51.25
FF	40207	106400	37.79
BRAM	73	140	52.14
IO	2	125	1.60
MMCM	1	4	25.00

Table 7.1: Total resource utilization

Name	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	BRAM
tb_dual_core	27033	4848	8378	64
core[0].dcache_ctr	3433	820	1154	16
core[0].dcache_ctr.tag_array_cache	70	40	13	0
core[0].dcache_ctr.data_array_cache	2351	0	0	16
core[0].dcache_ctr.coherence_table	88	0	0	0
core[0].dcache_ctr.request_fifo	12	372	7	0
core[0].dcache_ctr.cache_response_fifo	8	24	6	0
core[0].dcache_ctr.snoop_response_fifo	19	0	7	0
core[0].dcache_ctr.rr_arbiter	563	0	1	0
core[0].icache_ctr	3044	820	1135	16
core[0].processor	2349	0	254	0
core[1].dcache_ctr	3405	820	1161	16
core[1].icache_ctr	3112	820	1157	16
core[1].processor	2540	0	273	0
bus	2927	0	565	0
memory_ctr	4547	1568	462	0
memory_ctr.cam	1851	0	315	0
memory_ctr.pr_lookup_table	499	192	99	0
memory_ctr.resp_queues[0]	716	344	5	0
memory_ctr.resp_queues[1]	14	344	4	0
memory_ctr.resp_queues[2]	12	344	5	0
memory_ctr.resp_queues[3]	352	344	4	0

Table 7.2: Resource utilization per components of the dual-core top module.

The resource utilization of the cache controller, bus, memory controller of the dual-system is shown in Table 7.1. The shaded rows show the utilization of sub-modules inside the cache controller and memory controller. Initially, the Data Array was implemented using distributed Memory (LUTs as Memory), and the utilization was 3444 LUTs. We managed to free 1093 LUTs by mapping the module to use 16 BRAMs. Although the Data Array is using BRAMs, the LUTs utilization is still high. This is because the logic around the Data Array that generates the byte enable data and control wires requires higher number of LUTs. The request data can be any of the 4 byte coming from *data* bit range of the **cache-request** packet. Any of the 4 bytes need to be shifted to the right byte in the 64 bytes cache line based on the address. Therefore, the control logic that maps the request data byte to corresponding byte to the input wires of the BRAM use large LUT resource. One other way to decrease the LUTs usage is by disabling the byte writing and reading on the Data Array, and only supporting cache line size read and write. This comes with the cost of increased Data Array write and read latency. So, we chose the byte enable

design to keep the latency low.

Table 7.2 also shows there is high LUTs usage by the CAM unit compared to other memory controller sub-modules. This is because of the parallel comparison of over all stored data.

Name	LUT as Logic	LUT as Memory	LUT Flip Flop Pairs	BRAM
sodor_zynq [14]	4122	0	549	0
RocketChip(Tinycore) [4]	2388	48	1240	0
Our RISCV processor	2349	0	254	0

Table 7.3: comparison our RISCV core to the sodor

Rocket Chip [4] is an open-source SoC design generator that compiles synthesizable RTL of RISCV cores, caches and interconnects. we picked TinyCore configuration that has same RISCV ISA target (RV32IA) but uses the smallest resources. The result in Table 7.3 shows that our RISCV processor compares well with TinyCore RISCV core in terms of resource utilization. We also compared our processor to Sodor's [14] 5 stage piplined RISCV core implementation. Our processor used 50% less than the Sodor core.

7.1 Test Applications

The design is tested using three different RISCV compliance tests according to our RISCV target. We used RV32UI, RV32MI and RV32UA set of test programs in RISCV-test suits to cover machine-level, user-level, and atomic instructions respectively. We also created memory instruction generator that emits memory requests based on the EEMBC [15] program to verify the implemented coherence protocol in the coherence table. The memory request traces are chosen in such a way that covers all the state transitions in the coherence table. The coherence verification was done before integrating the memory hierarchy with the core.

Finally, we integrated the modules into a dual-core top design and verified the dual-core design by running the SPLASH2 benchmarks. All of the benchmark program we used is statically linked pthread programs. The programs are setup to use two threads. we were are able to run 7 of the benchmarks up until finish point. The worst case latency of a memory request is shown in the Table 7.4

SPLASH2 programs	LU	BARNS	FFT	FMM	RADIX	CHOLESKY
Worst Case Latency (cycles)	1593	1983	1509	1456	1102	1094

Table 7.4: Worst case latency of memory request of SPLASH2 benchmarks

Chapter 8

Conclusions and Future Work

This thesis describes a design and implementation of a predictable cache-coherent multi-core system. The coherence is based on the PMSI [10] guidelines. The core used in the design comprises 5 stage pipelined and fully bypassed RISCV processor. The processors implements the RISCV RV32IA target ISA. Each core has data and instruction caches. And each cache has 1Kbytes size with direct-mapped cache organization. The cache controller realizes the coherence protocol using state machines. Last-level memory controller is implemented as a directory on the memory side. The cores and the last-level memory controller is connected through snooping bus. The snooping bus uses TDM arbitration scheme.

A dual-core system was put together and simulated in the Verilator simulator. The design was validated by running RISCV regression test programs. We also validated the coherence manually by using memory instruction generator to cover all the state in the coherence table. Finally, We synthesized and verified the design on the Pynq board by running SPLASH2 benchmark programs.

After looking at the resource utilization, Data Array takes the major resource in the cache controller unit. As a future work, investigating a better design of Data array that will use less resource will help in fitting more core in the design. The other result we saw is the CAM unit does not scale well when the private cache line size is increased. Replacing the CAM unit with HASH table that maps to Block RAM will help in increasing the private cache sizes.

References

[1] Predictable cache coherence for multi-core real-time systems.

[2] ARM. Cortex-R5 and Cortex-R5F Technical Reference Manual. 2011.

[3] ARM. ARM Architecture Reference Manual ARMv8. 2013.

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[5] J. M. Calandrino and J. H. Anderson. On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, July 2009.

[6] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 57–68, Nov 2016.

[7] ARM Cortex. Cortex-A9 MPCore. *Technical Reference Manual*, 2009.

[8] Giovani Gracioli and Antônio Augusto Fröhlich. On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures. *SIGOPS Oper. Syst. Rev.*, 49(2):2–16, January 2016.

[9] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77, Dec 2009.

[10] M. Hassan, A. M. Kaushik, and H. Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, April 2017.

[11] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234, April 2017.

[12] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, Toulouse, France, November 2010.

[13] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Ninth European Dependable Computing Conference*, 2012.

[14] The Regents of the University of California. Riscv-sodor. `https://www.librecores.org/codelec/riscv-sodor`, April 2019.

[15] Jason Poovey et al. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

[16] A. Pyka, M. Rohde, and S. Uhrig. Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 107–114, July 2014.

[17] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[18] Henry Cook SiFive. Diplomatic design patterns : A tilelink case study. 2017.

[19] Wilson Snyder. Verilator. url=https://www.veripool.org/wiki/verilator, April 2019.

[20] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[21] JEDEC Standard. Ddr3 sdram standard. *JESD79-3, Jun*, 2007.

[22] Ashley Stevens. Introduction to amba® 4 ace and big. little processing technology. *ARM White Paper, CoreLink Intelligent System IP by ARM*, 2011.

[23] Dana Vantrease, Mikko H Lipasti, and Nathan Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011.

[24] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual volume ii: Privileged architecture version 1.9.1. Technical Report UCB/EECS-2016-161, EECS Department, University of California, Berkeley, Nov 2016.

[25] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: User-level isa, version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[26] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[27] Xilinx. Pynq. http://www.pynq.io/, April 2019.

[28] Xilinx. Pynq xlnk module. https://pynq.readthedocs.io/en/v2.4/pynq_package/pynq.xlnk.html, April 2019.