Memory Subsytem Design for Multicore RISC-V processors

A Project Report

submitted by

S PAWAN KUMAR (CED16I043)

in partial fulfilment of requirements

for the award of the dual degree of

BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING KANCHEEPURAM

MAY 2021

DECLARATION OF ORIGINALITY

I, **S Pawan Kumar**, with Roll No: **CED16I043** hereby declare that the material presented in the Project Report titled **Memory Subsytem Design for Multicore RISC-V processors** represents original work carried out by me in the **Department of Computer Science and Engineering** at the Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram.

With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

S Pawan Kumar

Place: Chennai

Date: 04.05.2021

CERTIFICATE

This is to certify that the report titled **Memory Subsytem Design for Multicore RISC-V processors**, submitted by **S Pawan Kumar (CED16I043)**, to the Indian Institute of Information Technology, Design and Manufacturing Kancheepuram, for the award of the dual degree of **BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY** is a bona fide record of the work done by him/her under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Noor Mahammad SK
Project Guide
Assistant Professor
Department of CSE
IIITDM Kancheepuram, 600 127

Place: Chennai
Date: 04.05.2021

Dr. Neel Gala
External Guide
Chief Technological Officer
Incore Semiconductors Pvt. Ltd
Chennai,600 002

Place: Chennai
Date: 04.05.2021

ACKNOWLEDGEMENTS

The internship opportunity I had with InCore Semiconductors Pvt. Ltd. was a great chance for learning and professional development. Therefore, I consider myself a very lucky individual. First and foremost, I would like to thank Dr. Neel Gala for providing me with this amazing opportunity. I would also like to thank Dr. Neel Gala and Mr. Arjun Menon who took time out to guide me and participate in very lengthy discussions periodically. These discussions have contributed to key developments in the project. I would also like to thank my guide, Dr. Noor Mahammad for his support and guidance during such testing times.

I perceive this opportunity as a big milestone in my career development. I will strive to use the gained skills and knowledge in the best possible way, and I will continue to work on their improvement, in order to attain the desired career objectives.

ABSTRACT

All modern day processors use caching of memory as a means to provide higher performance and lower CPI. Such a processor may also have multiple cores/harts operating over a single shared address space. When multiple actors in a system cache the same memory and operate on it, it is highly likely that data is corrupted. In the absence of coherence, the outcomes of multiple programs running at the same time are non-deterministic and result in a harder programming model. Hence, such multicore systems keep the caches coherent with each other to provide guarantees on the outcomes based on the memory model. Design decisions in coherent cache subsystems span across multiple domains such as the coherence protocol, interconnect fabric and the physical implementation itself. This introduces multiple points for human error to creep in and cause problems. This report aims to provide a modular design for such a cache subsystem and alleviate human errors by automating protocol implementation in hardware. This helps reduce design-test-fix cycles and improve re-usability of the components. This report also identifies the RISCV specific features needed to be supported by the cache and provides implementation details regarding the same.

Towards आत्मनिर्भर् भारत् and Domain Specific Hardware Design...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ISA	Instruction Set Architecture
SSP	Stable State Protocol
TSO	Total Store Ordering
RVWMO	RISC-V Weak Memory Ordering
LR	Load Reserved
SC	Store Conditional
AMO	Atomic Memory Operation
LLC	Last Level Cache
SWMR	Single Writer Multiple Reader
FIFO	First In First Out
PLRU	Pseudo Least Recently Updated
MRU	Most Recently Updated
ALU	Arithmetic and Logic Updated

NOTATION

$xlen$	The bit width of the registers in the core.

CHAPTER 1

Introduction

1.1 Motivation

With the dramatical slowing down of Moore's law, the industries are relying on development of domain specific systems to achieve higher performance i.e. systems which cater to a specific purpose and offer high performance in all problems of that domain. Many such systems share memory among multiple actors in a system. In a shared memory system, each of the actors may perform memory operations(read/write) on a singular shared address space. These designs seek various optimisations and guarantees such as high performance, low power, and low cost. Of course, correctness is an inherent guarantee expected by such systems. Hence, a shared memory system needs to implement a coherency protocol which ensures memory consistency. However, such a system also needs a transport protocol and the physical transmission of the data i.e topology of the bus. Design decisions across all these layers impact the performance of the system. Along with these, there are a multitude of optimisations possible in the caches. Each design comes with its own trade-offs and offers different performance/area/power benefits.

A framework which automatically generates the memory subsystem based on the design choices would drastically reduce the development time. Having such an automated framework will also help reduce the turn-around time necessary to gain performance metrics and help implement speedy optimisations. Since the different components are reasonably independent, development/optimisations for each individual component can proceed concurrently if the interfaces are standardised. Such a framework also complements the primary offerings of InCore Semiconductors; a series of Processor(Chromite[1]) and SOC generators(Chromite-M[2]). This would enable stakeholders to generate custom hardware as per technological needs. The objective of this project is to follow a modular approach and automate the memory subsystem design for such multi-core systems.

CHAPTER 2

Memory Model

A memory consistency model, or simply, a memory model, is a specification of the allowed/legal outcomes of multithreaded programs which operate on a shared address space. For a multithreaded program executing with specific input data, it specifies what values dynamic loads(loads whose values change over time) may return. Unlike a single-threaded execution, multiple outcomes are usually legal. Memory models impact three different aspects of a design listed below.

- Performance of the hardware
- Complexity of the hardware implementation.
- Complexity of the programs which run correctly on the hardware.

The more constrained a model is, the lower the performance of the hardware conforming to it. The more relaxed the model, the complexity of the programs/hardware implementation increases. This chapter provides a brief overview of the different memory models and the memory model imposed by the RISC-V ISA.

2.1 Sequential Consistency

Sequential Consistency is the most intuitive model. It was first formalized by Lamport[3], who called a single processor (core) sequential if the result of an execution is the same as if the operations had been executed in the order specified by the program. He then called a multiprocessor sequentially consistent if the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program. The total order of operations is referred as memory order. In Sequential Consistency, the total/memory order always reflects and respects all the relevant program orders. The memory model requires that all cores preserve the program order of the local memory operations for all permutations of consecutive operations, namely:

- Load preceding Load
- Load preceding Store
- Store preceding Store
- Store preceding Load

This model is the easiest to implement, and writing programs which work according to this model is intuitive. However, due to the inherent ordering of the operations defined, cores cannot execute instructions speculatively/reorder the instructions freely. This translates to lower throughput of instructions and high CPI(Cycles per Instruction).

2.2 Total Store Order

Write Buffers are a very efficient performance optimisation in the modern day processors. A store is entered into the write buffer when the store instruction is committed in the instruction pipeline. The store buffer holds these stores within itself until the memory system is ready to commit these changes to memory. This typically occurs when the cacheline is in a Read-Write coherence state. Consequentially, the stores can be retired without waiting for coherence operations i.e the store enters the store buffer when the store instruction is committed but the actual change in the memory occurs when sufficient coherence permissions have been obtained. Stores are a pretty common operation in any program and being able to not stall the core due to delays in the memory offers huge performance benefits.

For a single-core processor, the effects of the write buffer can be made invisible to the program by ensuring that a load to address A returns the value of the most recent store to A from both the write buffer and the memory/cache. This is typically done by either by stalling a load of A if a store to A is in the write buffer or by passing the value of the most recent(as per program order) store to A to the load from A. When building a multicore processor, it seems natural to use multiple cores, each with its own bypassing write buffer, and assume that the write buffers continue to be invisible architecturally. This assumption is wrong, because the outcome of one of the resulting scenarios will violate SC. This resulted in development of a new memory model which requires that cores

preserve the program order of the local memory operations for the following combination of operations:

- Load preceding Load
- Load preceding Store
- Store preceding Store

It should be noted that the set of constraints are a subset of the set of constraints imposed by the sequential consistency model. Consequentially, the set of allowed outcomes under SC is a subset of the allowed outcomes under TSO.

2.3 RVWMO Memory Consistency Model

The standard memory model for RISC-V, called RVWMO (RISC-V Weak Memory Ordering)[4], is designed to provide flexibility for computer architects to build high-performance and scalable hardware while also supporting an intuitive and tractable programming model.

RVWMO is a variant of the release consistency[5] model. In release consistency, synchronization accesses (accesses which perform synchronization operations) are processor consistent with each other. All previous regular data accesses must be completed before a synchronization access can be performed and vice versa. Consequentially, the burden of synchronisation falls on the programmer. The memory will only be consistent immediately after a synchronization operation. It further differentiates between the entry operation(*acquire*) and exit operation(*release*) in synchronization.

Under RVWMO, code running on a single hart appears to adhere to program order as observed by the other memory operations in the same hart, but memory instructions from another hart may observe the same memory operations being executed in a different order. Therefore, code which share the same address space require explicit synchronization primitives to warrant ordering between memory instructions among different harts.

The RVWMO memory model enables developers, while simultaneously providing high performance programmes for language memory models, to construct basic implementations, aggregate implementations, implementations integrated into a much wider

system and subject to dynamic memory system interactions or other possibilities. The model imposes a set of 13 constraints which apply only to special scenarios in the instruction execution trace. Reorderings are allowed in all other scenarios.

A program/implementation is said to adhere to the RVWMO model if it follows the preserved program order[4] and also satisfies the following

- *load value axiom*
- *atomicity axiom*
- *progress axiom*

Additional information regarding the model and axioms can be found in the RISCV ISA Specification[4].

TSO and SC impose constraints on the ordering of instructions unconditionally, whereas RVWMO constraints are imposed conditionally. Hence, the set of allowed outcomes under SC/TSO is a strict subset of the set of allowed outcomes under RVWMO. Software written assuming RVWMO can be run on hardware which assumes SC/TSO but vice versa is not feasible.

CHAPTER 3

Memory Ordering Constraints

RISC-V hardware primarily follow a variant of release consistency(the *RCsc* variant)[6]. Hence, loads and stores can be reordered freely in RISC-V. However, some loads may be marked as *acquire* operations which must not be observed to succeed later memory operations and some stores may be marked as *release* operations which must not be observed to precede any earlier memory accesses[4]. Although the ISA does allow hardware to follow TSO2.2 only, the hardware must still support the instructions/constructs discussed in this chapter. The memory ordering constructs control the ordering of the memory instructions while execution and help provide guarantees to the programmer.

3.1 Fence Instruction

The base ISA for RISC-V[4] includes minimal support for memory ordering in the form of the *fence* instruction. The fence instruction is used to order device I/O and memory operations as observed by other actors in the system. The fence instruction orders operations in the predecessor set with respect to operations in the successor set. The sets can contain any combination of the following operations

- device input(I)
- device output(O)
- memory reads(R)
- memory writes(W)

The distinction between which addresses correspond to memory and which addresses correspond to devices is platform specific.

The *fence.i* instruction is supported by the ISA via an optional *Zifencei* extension. This instruction is used to synchronize the data and instruction streams. In a RISC-V system,

it is not mandatory for the instruction stream to be consistent with the data stream at all instances. Hence the need for an explicit instruction to synchronize them is needed. The *fence.i* instruction does not guarantee or enforce that other harts in a system will witness the changes by the hart executing the instruction. It only ensures that the local hart will observe all its own changes and other changes in the global memory order so far.

3.2 Atomic Memory operation Instructions

The instructions supported by the standard atomic-instruction extension(called *A*) allow atomic read-update-write operation on memory to support coordination between multiple actors access the same memory space[4]. There are two forms of instructions namely

- load-reserved/store-conditional
- atomic memory computations

Both these types of instructions provide multiple memory consistency orderings such as unordered, acquire, release and sequentially consistent. Each atomic instruction has two bits, *aq* and *rl* which are used to specify the aforementioned memory orderings. Table 3.1 contains the encodings and meanings for the different ordering constraints.

Ordering	Meaning	*aq*	*rl*
Unordered	No explicit ordering specified	0	0
Acquire	Ordered before successive memory operations	1	0
Release	Ordered after any preceding memory operations	0	1
Sequentially Consistent	No reordering allowed	1	1

Table 3.1: Memory ordering encoding and its meaning in Atomic Instruction

3.2.1 Load-Reserved/Store-Conditional

Complex memory operations on a single address location(word or double word) can be atomically performed by the use of LR(load reserved) and SC(Store Conditional) instructions. The LR instruction reads a word(or double word) from the target address, places it in the destination register(sign extending if necessary) and registers a *reservation set*. The size of the reservation set is platform dependent. Minimally, the reservation set should

contain all the bytes accessed by the instruction i.e 4 bytes in a RV32 system and 8 bytes in a RV64 system. The SC instruction conditionally writes a word(or double word) from the source register to a target address, the SC succeeds if and only if the reservation is still valid and it subsumes all the bytes being updated. If the SC fails, the instruction does not modify memory and it returns a non zero value(error code) in the destination register. Executing a SC instruction invalidates any reservation set held by the hart regardless of success or failure. Data modification to any byte(s) over which a reservation has been registered always invalidate the reservation set. If the data modification was by another actor(other hart or device) in the system, the modification succeeds and fails otherwise. The reservation set held by a hart is also invalidated if there is any SC executed by the hart. The behaviour of the SC while dealing with virtual addresses is platform dependent and is allowed to succeed or fail if the virtual addresses are different although the effective addresses are the same.

3.2.2 Atomic memory computation

These instructions perform a sequence of operations on a memory location atomically. The sequence is typically load, computation followed by a store to the same location. These instructions can either be emulated by using the LR/SC sequence in software in case the hardware does not support atomic operations. The atomic instructions always . All of these operations update the destination register with the value at the target address before modification. The list of operations supported are listed below:

- Swap: Update the target address with the value in the source register.
- Add: Add the value in source register to the value at target address and update the target address with the result.
- And: Update the target address with the bitwise and of the values at the target address and the source register.
- Or: Update the target address with the bitwise or of the values at the target address and the source register.
- Xor: Update the target address with the bitwise xor of the values at the target address and the source register.
- Max: Update the target address with the higher value among the values at the target address and the source address. Treat the values as signed numbers in two's complement representation.

- Min: Update the target address with the lower value among the values at the target address and the source address. Treat the values as signed numbers in two's complement representation.
- Max-U: Update the target address with the higher value among the values at the target address and the source address. Treat the values as unsigned numbers in binary representation.
- Min-U: Update the target address with the lower value among the values at the target address and the source address. Treat the values as unsigned numbers in binary representation.

These atomic operations form the basis for many of the synchronisation primitives used in modern programming languages such as mutex and semaphores. These can also be used to accelerate the memory primitives in modern day operating systems and programming languages.

CHAPTER 4

Cache Coherence Protocols

Coherence is the uniformity of the data in the memory as seen by all actors in a system at any given instant. The purpose of a coherence protocol is to ensure coherence by making sure the following invariants are preserved[7].

1. Single-Writer, Multiple-Read (SWMR) Invariant: At any given (logical) time instant, only one of the following conditions holds true for any memory location A:
    - A Single private cache has read-write permissions for A
    - Any number of private caches have read only permissions for A
2. Data-Value Invariant. No modifications to a memory location are observed between the start of an logical time slice and the end of the previous read-write logical time slice.

4.1 States

Four characteristics of a cache block are encoded in its state namely ownership, validity, exclusivity and dirtiness [8]. Exclusivity and ownership are relevant only in systems with multiple actors.

- Validity: The block has the recent value and read operations are allowed on the block. However, writes to the block are allowed only if it is also exclusive.
- Dirtiness: A cache block is termed dirty if its value has been updated after the most recent fetch and it does not resemble the value in LLC/memory. The cache controller is eventually updates the LLC/memory with this fresh value. This is the characteristic of a write-back cache.
- Exclusivity: A cache block is exclusive if only a single private cache contains the block. The LLC/memory might also hold the block if they are shared entities.
- Ownership: A controller is said to hold ownership of a block if it is responsible for sending responses to the coherence transactions for that block. Typical coherence protocols do not allow more than one owner for a block at any given instant. A block which is owned may not be replaced or flushed from a cache without transfer of ownership to another entity. Blocks which aren't owned may be flushed without any transactions(silently).

4.1.1 Stable States

Most coherence protocols use a subset of the classic five state MOESI model first introduced by Sweazey and Smith [8]. These states refer to the states of blocks in a cache. MSI are the three fundamental states. The O and E states may be used in advanced versions of the protocols. Each of these states has a different combination of the characteristics discussed above.

- M(odified): The block is valid, exclusive, owned, and potentially dirty. The cache has read-write permissions on the block. The cache is the owner of the block and hence the cache must respond to coherence requests. The value of the block at the LLC/memory could be stale.
- O(wned): The block is valid, owned, and potentially dirty, but not exclusive. The cache has read-only permissions on the block. Read-only copies of the block may exist in other private caches, but they are not owners. The value of the block at the LLC/memory could be stale.
- E(xclusive): The block is valid, exclusive, and clean. The cache has read-only permissions on the block. No other private caches have a valid copy of the block, and the copy of the block in the LLC/memory is recent. In some protocols the Exclusive state also implies that the private cache has ownership of the block.
- S(hared): The block is valid but not exclusive, not dirty, and not owned. The cache has a read-only copy of the block. Other private caches might contain valid, read-only copies of the block too.
- I(nvalid): The block is invalid. The cache either does not contain the block and hence has no read or write permissions on the block.

The most common transactions associated with these states are listed in Table 4.1.

4.1.2 Protocol Design Options

A coherence protocol can be designed in a variety of ways. There are many different protocols that are used for the same set of states and transactions. The protocol's design determines what events and transitions are possible at the coherence controllers. However, there is no way to present a list of possible events or transitions independent of the protocol itself unlike the states and transactions. Though there exists a vast design space, there are two main classes of coherence protocols namely snooping and directory.

Transaction	**Goal/Meaning**
GetShared (GetS)	Request block in Shared state
GetModified (GetM)	Request block in Modified state
Upgrade (Upg)	Upgrade block state from Shared or Owned to Modified; Upg (unlike GetM) does not require data to be sent to requestor
PutShared (PutS)	Evict block in Shared state
PutExclusive (PutE)	Evict block in Exclusive state
PutOwned (PutO)	Evict block in Owned state
PutModified (PutM)	Evict block in Modified state
Recall	Used by a higher level cache to initate eviction of a block in the lower level caches.

Table 4.1: Common Coherence Transactions

- Snooping protocol: The requests for a block by any cache controller is broadcasted to all the other controllers which are in the same coherence domain. The controllers change state based on the request and are trusted to perform the right operation. These protocols heavily rely on the interconnect to define the ordering of the messages and provide consistent deliveries to all actors.
- Directory protocol: A cache controller initiates a transaction for a block by unicasting it to the memory controller that is the block's house. The memory controller manages a directory that contains information about each block in the LLC/memory, such as the identifier of the current owner or the identifiers of current sharers. When a request arrives, the coherence controller looks up the directory state of the relevant block.

The decision between snooping and directory includes making different trade-offs in the domains of area, power and latency. Snooping protocols are technically basic, but they do not scale to several cores because of the increased bus traffic due to broadcasting. They also consume huge amounts of power on that scale. Since directory protocols are unicast, they are scalable, but many requests take longer because an additional message must be transmitted while the owner and the directory aren't the same entity. Furthermore, the protocol used has an impact on the interconnection network (for example, classical snooping protocols require a complete order for request messages).

Another factor which affects the protocols is the atomicity properties of the system. The relevant properties are as follows:

- Atomic Requests: This property states the cycle in which the coherence request is ordered is the same cycle that it is issued. This property eliminates the possibility of a request from other actors changing the state of a block between the cycles in which the request is ordered and issued.

- Atomic Transactions: The atomic transaction property states that subsequent request for the same block may not appear on the bus until after the first transaction completes (i.e., until after the response for the previous request has appeared on the bus). Allowing requests to different blocks does not impact the protocol because coherence transactions involving different blocks are independent of each other.

4.2 Stable State Snoopy MSI Protocol

This section describes the transitions between the stable states for the Snoopy protocol with the MSI stable states. This protocol assumes a write-back cache. A block is owned by the LLC/memory if its in I or S state. The block is owned by the private cache if its in the M state.

Three notational problems should be acknowledged. The arcs are labelled by coherence requests found on the bus in Figure 4.1. Other activities such as load, store and responses to coherence requests, are deliberately excluded. Secondly, the cache controller coherence events shall be marked as "own" to indicate whether or not the requesting cache control is the requestor. Third, in a notation that is cache centric, Figure 4.2 specifies the block's status in the LLC/memory.

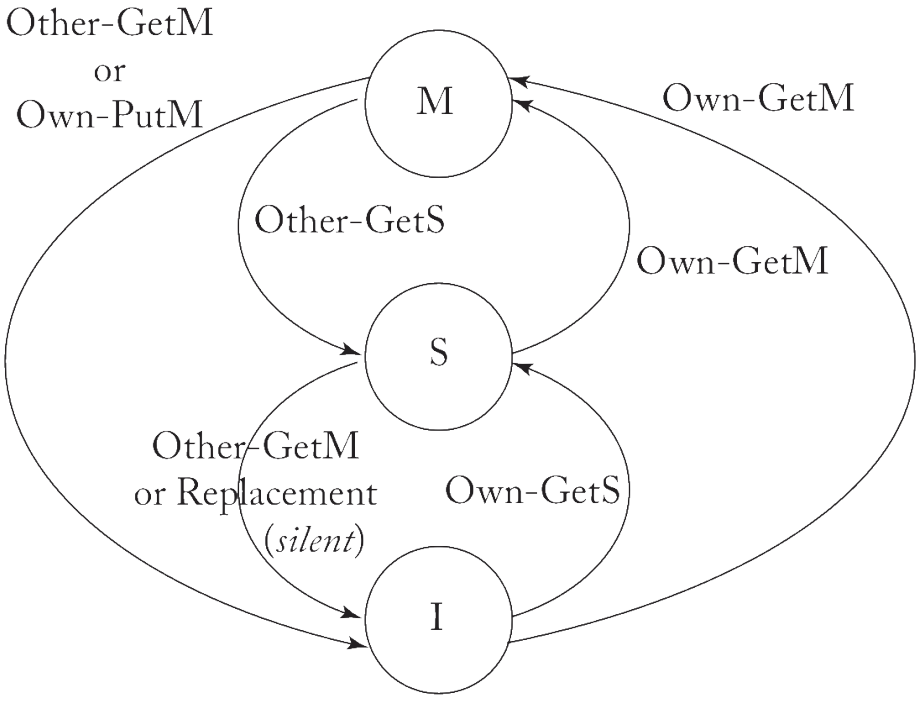

Figure 4.1: Transitions between stable states(MSI) at cache controller.

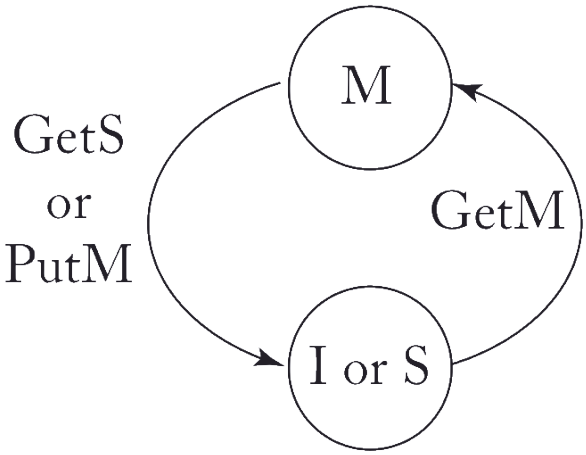

Figure 4.2: Transitions between stable states(MSI) at LLC/memory controller.

CHAPTER 5

Cache Coherence Controller

The coherence protocols discussed in Chapter 3 are overly simplistic and cannot be implemented as coherence controllers as is. Transitions between the stable states are not instant and neither are they direct. Often they involve exchange of multiple messages between different actors of the system. The status of sending/receiving these messages need to be tracked. This leads to multiple transient states being generated which indicate that the controller is waiting for an event from the bus in order to move into another state. The number of transient states depends on various design parameters such as bus architecture, nature of protocol, atomicity of transactions, atomicity of bus messages etc. In typical coherence protocols with dozens of states, it is easy for architects to make mistakes. An architect can forget about a possible coherence message that can arrive for a block in a certain state. An architect can introduce a bug in how an incoming message is handled for a block in a certain state. An architect can fail to think of a possible situation and end up with too few states. Hence the automated generation of controllers for a specified protocol is employed.

5.1 Protogen

Protogen[9] is a tool that automates the process of defining the coherence protocol for a system. It takes the description of a directory protocol(extensible to support snoopy) Using atomic transactions (i.e., no concurrency). Protogen then generates the corresponding protocol with non-atomic transactions for a multicore system. Starting with a Stable State Protocol, ProtoGen creates a protocol using transient states that is complete. This generates an auxiliary state as well as a permission for allowed operations. This is done without the need for an atomic system model which guarantees physically atomic transactions. The requirement is that there are correct and complete SSP descriptions for the cache and directory i.e the SSP must also satisfy SWMR. Single Writer Multiple Reader

(SWMR) and the data value invariant, the two invariants of coherency are satisfied by all protocols generated by protogen.

Protogen consumes the abstract specification of the SSP as input. To capture this, protogen defines a simple and powerful domain specific language. The specification describes the behaviour of the protocol for an individual cache block(as the behaviour for all the others is similar and independent). It specifies all of the following information:

- the subset of MOESIF which constitute the stable coherence states
- the stable auxiliary state at each cache block and each directory entry
- a list of operations on the cache block (loads, stores, eviction) and the transactions they initiate
- Tentative coherence messages (requests, forwarded requests, responses, and acknowledgments) that can be encountered for each stable state
- The target stable state(along with permissions and auxiliary states) to switch to on encountering a coherence message

A sample SSP for MI directory protocol is present in Appendix A.

ProtoGen produces finite state machines (FSMs) for the cache controllers including the transient states. These FSMs are expressed in a formal language and can be translated to any other format for specifying FSMs. Currently, protogen specifies the protocol in murφ(*murphi*)[10].

Protogen also allows different system constraints and other design choices to be specified. Due to this, a wide variety of protocols can be automatically generated for the same SSP. The supported constraints are:

- Atomic Transactions
- Atomic Requests

The supported design choices are:

- Blocking vs Non-blocking caches
- Access to the cache line in the transient states

Each configuration has a different impact on performance, area, power, latency and throughput. This enables us to generate coherence controllers which cater to a specific scenario with no additional cost.

5.2 Generating Cache Controllers in BSV

A tool, called *murphiToBSV* which translates murφ code(from Protogen) into Bluespec System Verilog(BSV)[11], a high level hardware definition language has been developed. This helps us leverage protogen to generate hardware coherence controllers. This tool exploits the similarities between the semantics of both the languages and the domain specific knowledge regarding cache protocols, nature of murφ code generated by Protogen and the cache design in bluespec to provide an efficient translation without the use of complex algorithms/parsers. The flow for generating the bluespec description is depicted in Figure5.1. The protocol is formally verified for safety, deadlock freedom and state

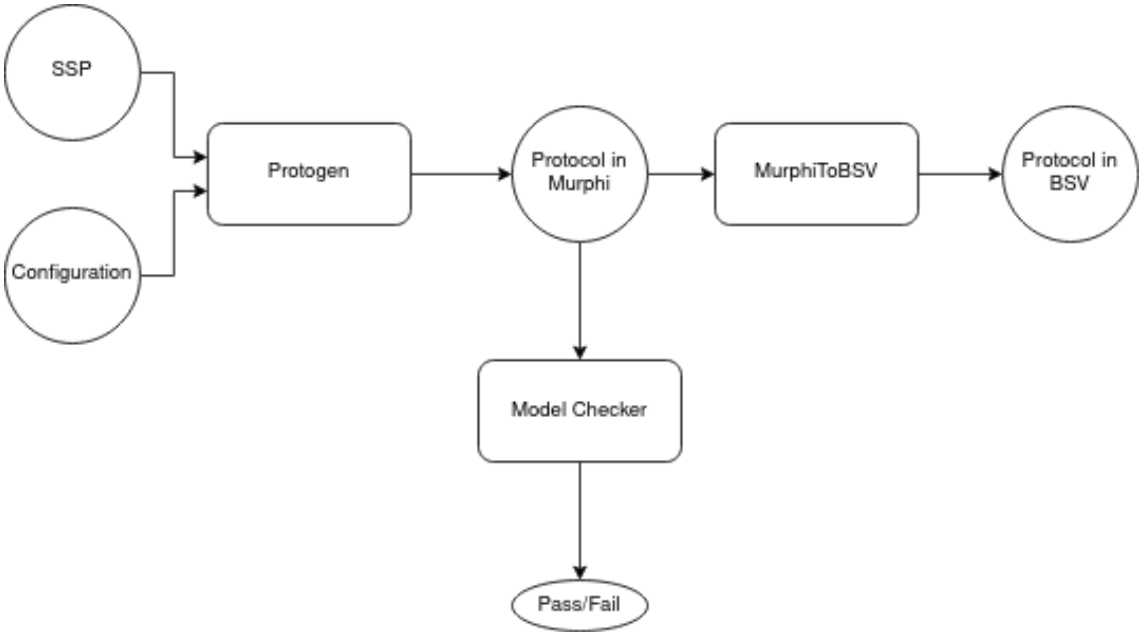

Figure 5.1: Cache Controller generation flow.

space validation using the murφ model checker[12]. The verification also ensures that the two fundamental invariants for coherence are never violated in the protocol. Using this as the source of truth ensures that no protocol implementation errors creep up into the hardware and drastically reduces resources necessary for verification.

The tool generates the necessary state transition functions for the caches. These are imported as a package in the cache and instantiated wherever necessary. This ensures that the code is modular and the protocol has minimal influence on the design of the cache. Thereby increasing code re-usability.

CHAPTER 6

Transport Bus

The transport bus consists of two individual but closely interconnected systems i.e the transport protocol and the physical transmission of the data itself. The physical transmission of the data is related to the underlying architecture and the network topology whereas the transport protocol is related to the meaning of the data being transmitted and all the necessary fields.

6.1 IncoreCLink

IncoreCLink is a custom transport protocol based on the Tilelink[13] specification. TileLink offers a highly concurrent and low overhead protocol. It is also deadlock free due to the static priority assigned to each of the channels. The protocol contains all the 5 unidirectional channels specified in the TL-C conformance level in [13]. The channel names, direction and class of messages are mentioned in Table 6.1. In case any agent has to resolve a conflict in processing packets from multiple channels, the priority order is as follows: A < B < C < D < E. All fields in the packets are exactly the same as specified in the specification[13]. Description of all the fields and their functions is out of the scope of this report. The relevant fields for coherence are *opcode* and *param*. The *opcode* field encodes the operation to be performed i.e Get Data, Acquire permissions etc. The *param* fields in the packets denote the requested/granted permissions for a block of data. The *param* field encodes the state transition. The protocol inherently supports coherency via an additional field in the message. The coherence states and the allowed operations on a block in that state are listed in Table 6.2. The *opcode* and *param* fields in the message together define a coherence operation. Table 6.3 represents the mapping of the coherence messages described in Table 4.1 to these fields in the message. These mappings are used to develop functions which translate coherence packets to IncoreCLink packets and vice versa.

Name	Direction	Class
Channel A	Master to Slave	Requests of operation on a specific address range, accessing or caching the data.
Channel B	Slave to Master	Requests for operation to be performed on the cached data by the master agent.
Channel C	Master to Slave	Data/Acknowledgement in response to a Channel B request.
Channel D	Slave to Master	Data/Acknowledgement in response to a Channel A request.
Channel E	Master to Slave	Final acknowledgement of a transfer, used for serialization.

Table 6.1: Channel description

State	Permissions
None	None
Branch	Read
Trunk	None
Tip (with Branches)	Read
Tip (without Branches)	Read & Write

Table 6.2: State to Permission Mapping

Coherence Message	Channel	Opcode	Param
GetS	A	AcquireBlock	None to Branch
GetM	A	AcquireBlock	None to Tip
PutM	C	ReleaseData	to Tip
Recall	B	ProbeBlock	to None

Table 6.3: Coherence Message Mapping

CHAPTER 7

Cache Architecture

7.1 L1 Data Cache

The L1 Data Cache services any requests for a memory operation from the core. The data stream for the core originates from this cache. The L1 data cache is a core local cache i.e private cache and is coherent with other L1 data caches in the system using a snoopy protocol. The design of the cache is blocking i.e on a miss the cache subsystem stalls until the miss can be serviced. The write policy followed in the design is Write-Back i.e the cache holds the updated value of the memory location until eviction and the next level cache(or memory) is only updated when the line is evicted from the cache. The cache consists of the following modules:

- Coherence Controller
- Cache Entries
- Fill Buffer
- Evict Buffer
- Reservation Handler
- Replacement Policy
- Core Request Handler

The coherence controller was generated as specified in Section 5.2. The interfaces used in the cache are generic such that any coherence protocol can be integrated with the cache. This enables reuse of the same cache design for multiple coherence protocols and lowers design overheads.

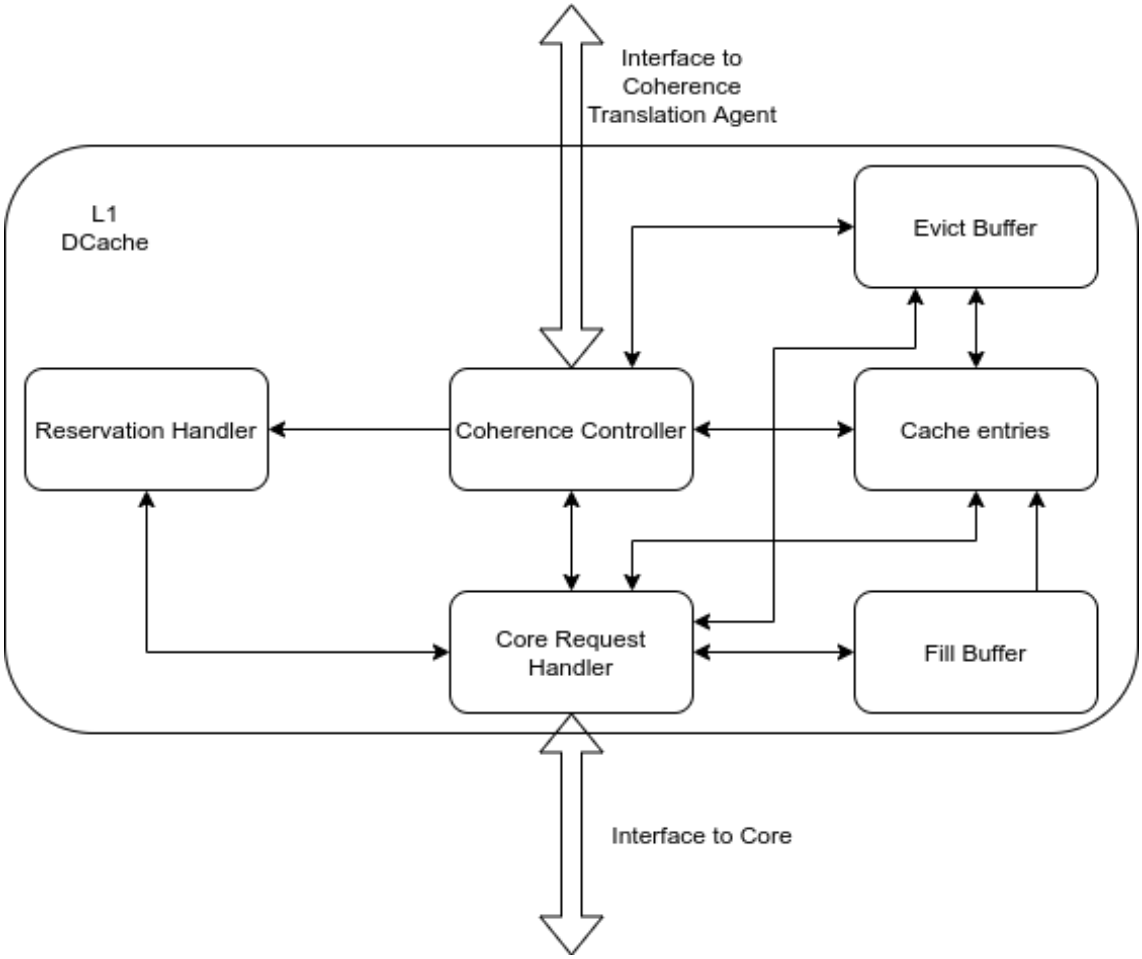

Figure 7.1: Block Diagram of L1 DCache.

7.1.1 Associativity

The cache is a n-way associative cache where n is a compile time parameter. The various variables and their meanings are listed below:

- $paddr$: The bit width of the address in the system.
- sz_line: Size of the line i.e each line holds sz_line bits.
- num_sets: The number of sets in the cache.
- n_l1: The number of ways in the cache. This variable inherently represents the associativity of the cache.

The bit width of the various fields in the system are calculated using the above variables. The bit width of the set index is $log_2(num\_sets)$. The width of the block index is $log_2(sz\_line)$. The physical address in the system is split into various fields as represented in Figure 7.2.

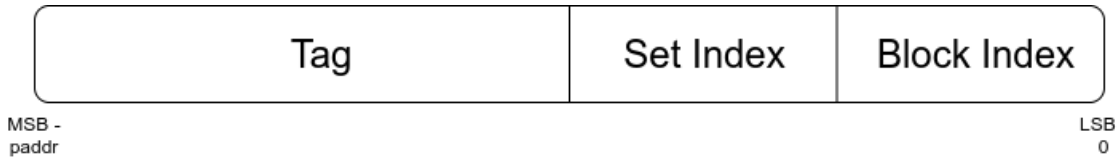

Figure 7.2: Split physical address to multiple fields.

7.1.2 Cache Entries

Each entry in the cache consists of three fields, namely:

- Tag
- Flags
- Cache line or Data

The *Cache Line* contains the actual data for the respective address(fetched from the next level cache or memory). The data may be fresh or stale depending on the operations performed so far. The size of the cache line is controlled via compile time macros. The *Tag* contains part of the actual address for the data. This is used to determine the cache hit and miss for a particular address. The size of the tag is $paddr - log_2(num\_sets) - log_2(sz\_line)$.

The *Flags* field constitutes the book keeping data structure for the cache which enables different modules of the cache to maintain memory consistency and control access. The various sub-fields are as follows:

- Valid: A single bit entry which indicates whether the cache entry is valid or not.
- Dirty: A single bit entry which indicates whether the cache entry has been modified and hence the value in memory has to be updated before eviction.
- State: A multi-bit entry which indicates the state of the cache entry based on the coherence protocol.
- Permissions: A 2 bit entry specifying the legal operations on the cache entry. There are three possible values for this field namely None, Read Only and Read-Write.

7.1.3 Fill Buffer

The fill buffer is a one entry buffer which temporarily holds the state of the cache entry until the line is acquired from the next module in the memory hierarchy on a miss. Once

the coherence transaction is complete i.e the line has been acquired, the core request is serviced and the line is written onto the cache storage. While writing the line back to the cache storage, an existing entry may be scheduled for eviction based on the replacement policy.

7.1.4 Replacement Policy

The replacement algorithm decides the way to replace in the event all the ways of a particular set contain valid entries. The design supports a parameterised interface to enable support for multiple replacement policies as listed below:

- Random: In this scheme, a random way in the set to be replaced.
- Round Robin(FIFO): In this scheme, the evictions are performed in the same order in which the ways were filled.
- Pseudo Least Recently Used(P-LRU): This scheme stores a single bit for each cache way. These bits are called MRU-bits. The MRU-bits for each way are set on access, indicating that the way was recently accessed. Whenever the last remaining in a set's MRU-bits is set, all other bits are reset. The leftmost way whose MRU-bit is 0 is replaced on a conflict.

All algorithms proactively retain dirty lines whenever possible to lower the traffic on the coherence bus. This results in a higher probability for a valid and non-dirty way to be evicted than warranted by the algorithm.

7.1.5 Evict Buffer

Whenever a cache entry is evicted, it is placed in the evict buffer. The evict buffer is a multi-entry buffer which temporarily holds the cache entry until the entry can be evicted and the coherence transaction for eviction is completed. This buffer also acts as a small victim cache in the event that there is a cache hit in the evict buffer i.e the coherence transaction for eviction has not begun. In such a scenario, the cache entry is swapped with one of the ways for the set based on the replacement policy.

7.1.6 Reservation Handler

This module maintains the reservation set held by the hart to support the LR(load reserved) and SC(store conditional) operations referred in 3.2.1. A single hart can have only one valid reservation registered as mandated by the ISA specification[4]. The size of the reservation set is equal to the size of the line(sz_line). The reservation is registered on a hit for a LR request. The reservation is invalidated whenever an SC request is encountered. A valid reservation is invalidated automatically when the corresponding cache entry is invalidated as a result of a coherence transaction. This ensures that the axioms for outcomes of LR-SC sequences are satisfied. Figure 7.3 shows the flowchart detailing the operation of this module in response to incoming requests.

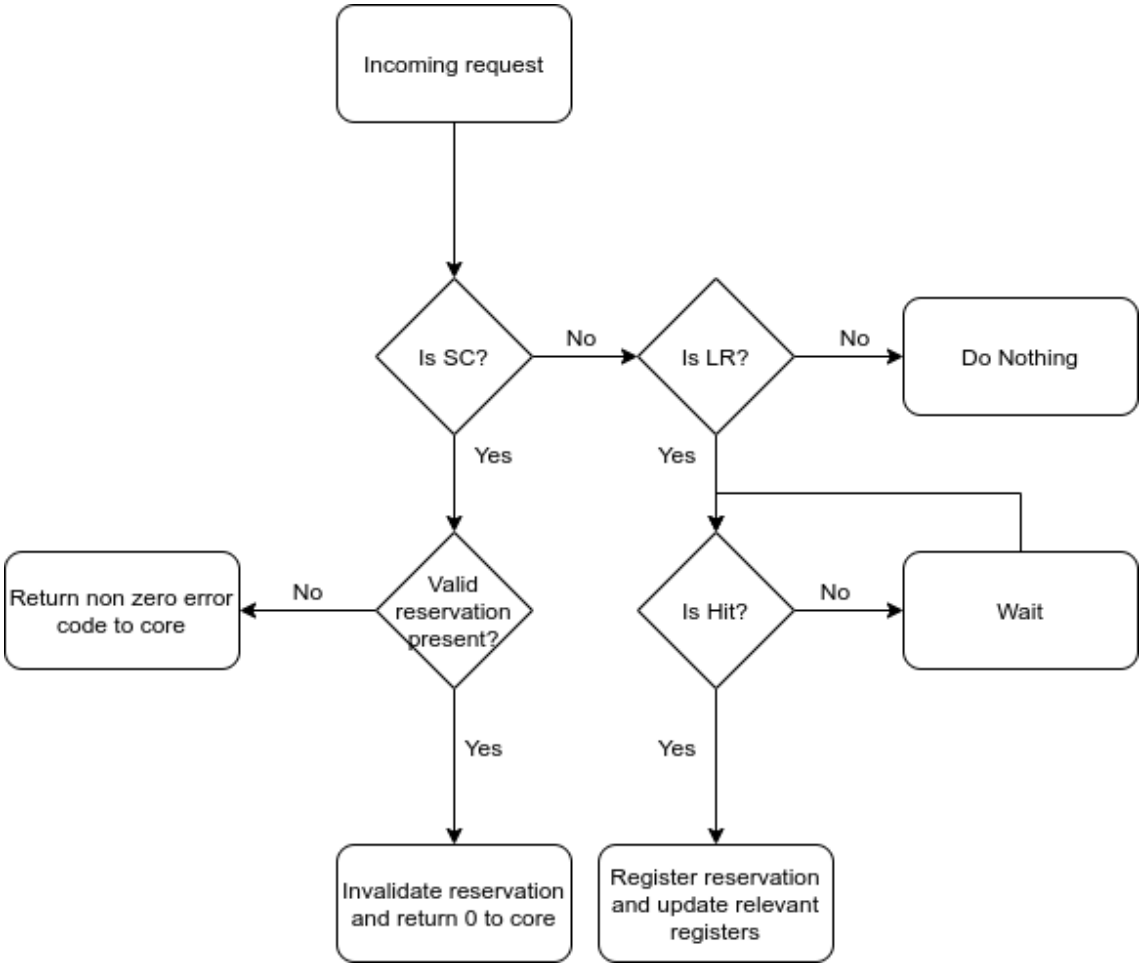

Figure 7.3: Flowchart detailing operation of the Reservation Handler.

7.1.7 Core Request Handler

This is the primary module which services core requests and initiates coherence transactions via the coherence controller. There are two kinds of requests:

- Fence request
- Memory request

Figure 7.4 shows the flowchart detailing the operation of this module in response to incoming requests.

7.1.7.1 Handling Fence requests

Typically, servicing a fence request is a multi-cycle operation. In a blocking cache, there is no reordering of memory operations. Hence to service a fence request, all the entries which are Valid and Dirty have to be written onto the next level cache or memory. This is performed using a combination of registers which keep track of the set number and way number. Each applicable entry is opportunistically placed in the evict buffer turn by turn until no such entries exist in the cache. Once the evict buffer has been emptied, the fence operation is marked as complete and a response is sent to the core indicating the same. The cache does not take in any new requests until the fence operation has completed.

7.1.7.2 Handing memory requests

A memory access request can be either a load, store or an atomic memory operation. For such requests, the address is treated as a hit, if one of the cache entries for the relevant set(indexed using the set index) satisfies the following conditions:

1. The tag in the tag array matches the tag derived from the address.
2. The valid bit is set.
3. The cache line has been obtained with the relevant permissions.

On a hit, the requested operation(load,store or atomic) is performed and the results are returned back to the core. The cache is also equipped with a miniature ALU which

handles the atomic memory operation requests. The ALU supports all the operations mentioned in 3.2.2.

In case condition 1 or 2 is not satisfied, it is treated as a miss and the entry has to be obtained from the next level in the memory hierarchy. A fill buffer entry is allocated and the coherence coherence transaction is initiated out based on the access permissions necessary. In case condition 3 is not satisfied and both conditions 1 and 2 are satisfied, it is treated as insufficient permissions. The coherence controller initiates an upgrade transaction to obtain the relevant permissions for the entry. The cache is stalled until the request can be serviced in both cases. All requests for an atomic operation are treated as a store access while initiating transactions and checking permissions.

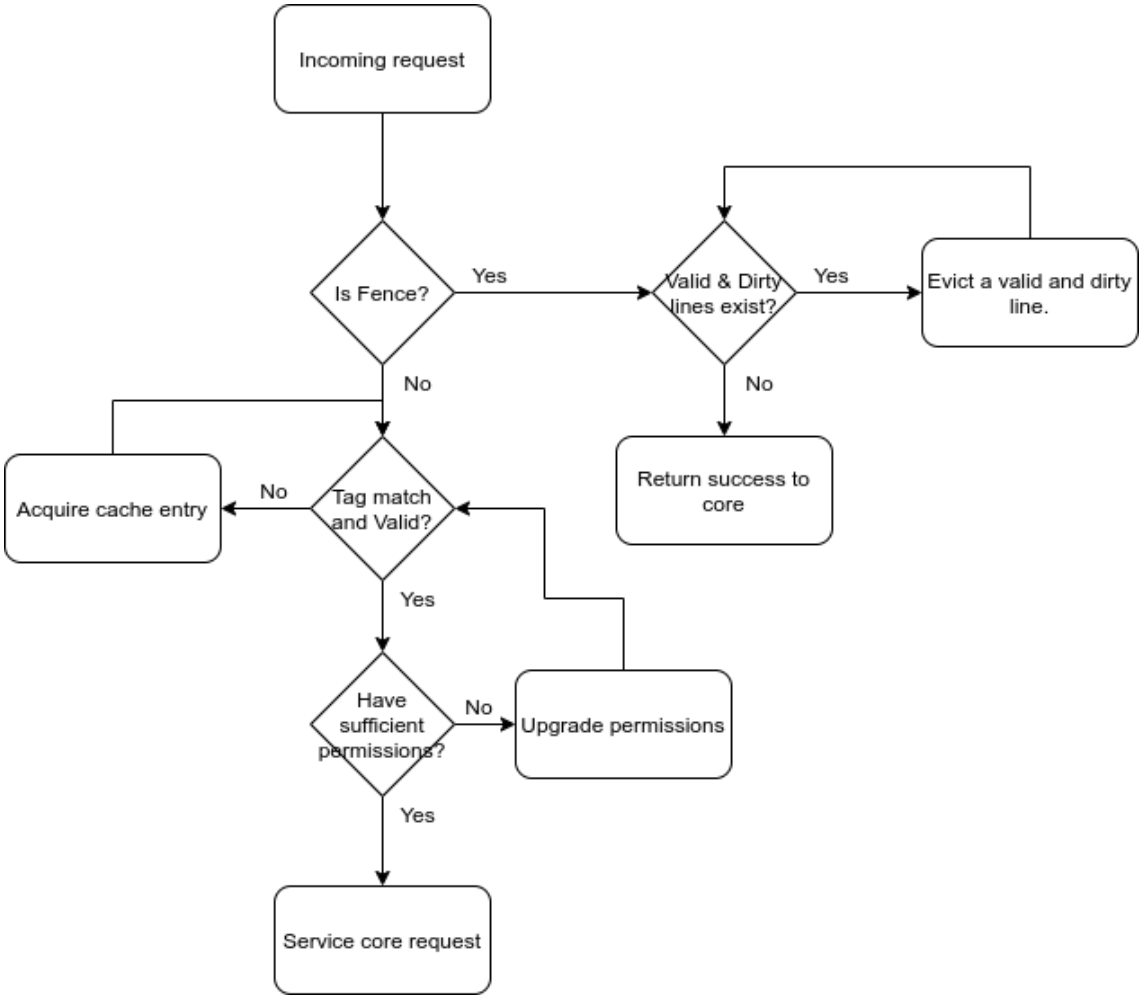

Figure 7.4: Flowchart detailing operation of the Core Request Handler.

7.2 L2 Cache

The L2 Cache lies between the L1 caches and the main memory in the cache hierarchy. The L2 cache is a shared cache and responds to requests from the L1 D and I caches. It also lies in the same coherence domain as the L1 DCaches. The functional modules in the L2 cache are as follows:

- Cache Entries
- Coherence Controller
- Memory request Handler
- CMO Handler

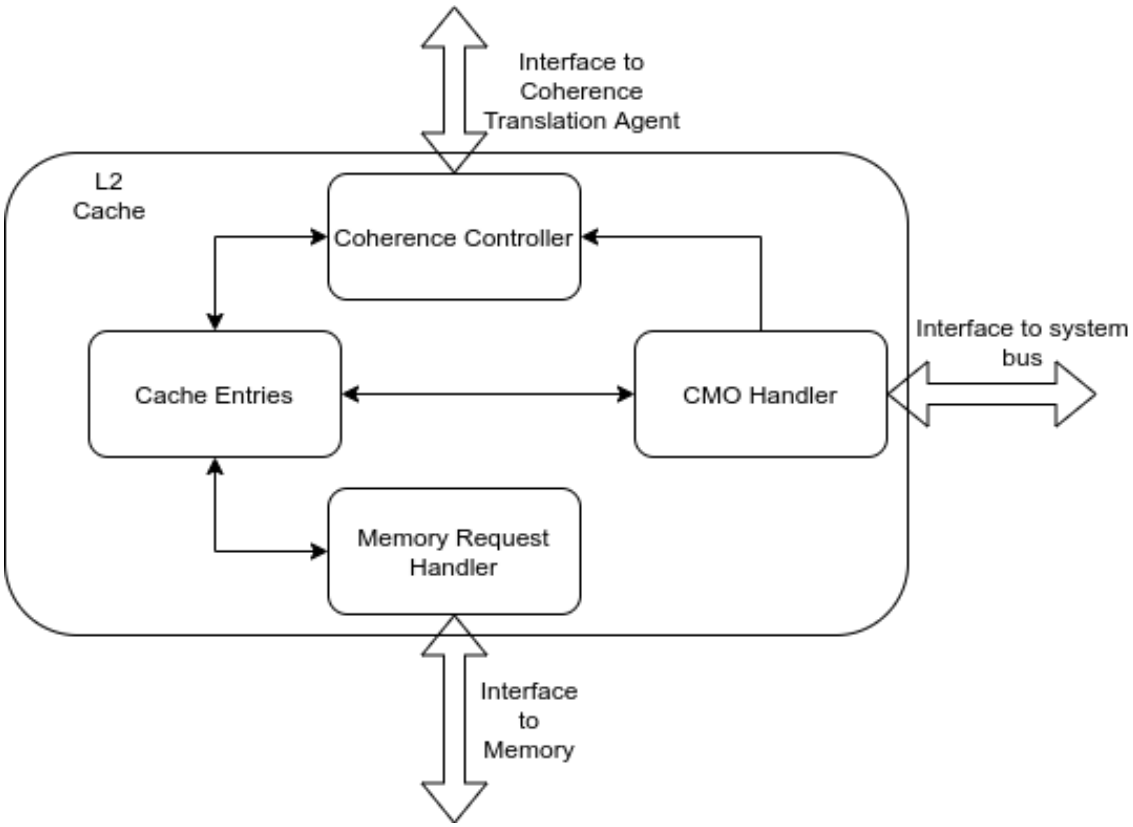

Figure 7.5: Block Diagram of L2 Cache.

The coherence controller was generated as specified in Section 5.2. The design for the cache entries[7.1.2] and associativity[7.1.1] are similar to L1 DCache. The size of the line(sz_line) and the physical address($paddr$) are equal to the respective variables in L1 DCache. The associated variables for the L2 Cache are:

- $l2\_num\_sets$: The number of sets in the cache.
- n_l2: The number of ways in the cache. This variable inherently represents the associativity of the cache.

The bit width of the various fields in the system are calculated using the above variables. The bit width of the set index is $log_2(l2\_num\_sets)$. The width of the block index is $log_2(sz\_line)$. The physical address in the system is split into various fields as represented in Figure 7.2. The size of the tag field in L2 cache is $paddr - log_2(l2\_num\_sets) - log_2(sz\_line)$

7.2.1 CMO Handler

Cache Management Operations(CMO) are operations which cause a change in the state of a cache entry without performing any memory accesses. These operations are usually used by the compiler to provide hints to the micro-architecture in order to improve performance of the software. Some of these operations may also be used to synchronise the value in memory and the cache subsystem when it goes stale. This scenario usually occurs when there are cacheable memory mapped devices. Other example use cases of these instructions include preventing timing side channel attacks, power management, persistence, debugging and software controlled consistency. This design focusses on providing support for such synchronisation mechanisms only. Future versions of the design may support advanced operations. These operations are supported via 2 memory mapped registers(*cmoaddr* and *cmostatus*) which are exposed to the core via the system bus fabric and are treated as device I/O. The functionalities of these registers are explained below.

7.2.1.1 *cmoaddr*

The *cmoaddr* register is a single register which specifies the address for the operation as well as the operation itself. The design assumes that each cache line will be 64 bits wide at the least. This allows the software to initiate a CMO by writing to a single memory mapped register and improve performance. Figure 7.6 represents the fields in this register. The operation to be performed is indicated by the three bit field called *op*. Currently the design supports 2 different operations namely:

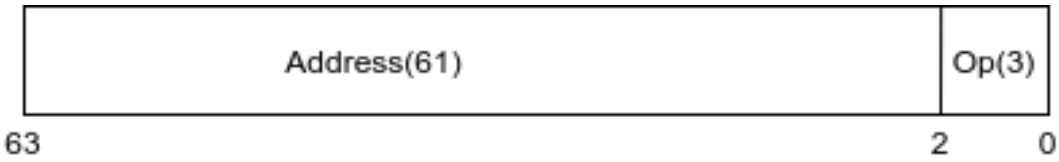

Figure 7.6: Fields in *cmoaddr* register.

- Flush($op == 0$): The line is recalled from the lower level caches and then evicted from the cache. If the line is dirty, it is written back to the memory to reflect the change.
- Invalidate($op == 1$): The line is recalled from the lower level caches and then evicted from the cache. However no update is made to the value in the memory i.e all the changes are discarded.

All other values of *op* are reserved for future use.

The address is a 61 bit field which is treated as a 64 bit value by assuming that the lower three bits are 0 always. The 64 bit value is then truncated to the required length($paddr$) and the operation is performed on the corresponding line. If the line doesn't exist in the cache, no operation is performed.

7.2.1.2 *cmostatus*

Figure 7.7: Fields in *cmostatus* register.

The *cmostatus* is a read-only *xlen* bit wide register. It houses 2 status bits which indicate the status of the CMO handler to the software. The rest of the bits are reserved and will always read as 0. The meaning of each individual bits are as follows:

- B: This bit indicates whether the CMO handler is busy(1) or free(0).
- S: This bit indicates whether the last operation was a success(0) or a failure(0). This bit is always cleared on a write to the *cmoaddr* register.

The *cmoaddr* register can only be written to if the B bit is 0, attempts to write the register result in bus errors otherwise. Table 7.1 shows the different encodings of the status bits and their inferences.

S	B	Meaning
0	0	The previous operation was a success and ready to receive next operation.
0	1	The handler is busy and cannot respond to any new operations.
1	0	The previous operation was a success and ready to receive next operation.
1	1	Never occurs.

Table 7.1: Status bit encoding for *cmostatus*.

CHAPTER 8

Implementation and Testing

8.1 Implementation Details

8.1.1 MurphiToBSV

The tool was implemented in python. The tool parses the files generated by protogen and generates the bluespec files for the coherence functions. As a byproduct, it also generates the translation functions based on the transport protocol. Currently the tool has support for generating translation functions for IncoreCLink and TileLink.

8.1.2 Coherence Controller

The following steps were involved in generation of the coherence controller description in BSV:

1. The Snoopy MSI with Recall protocol was chosen for the initial implementation. The SSP for the same was described using the DSL specified by protogen. Table 8.1 and Table 8.2 shows the relevant transitions in the SSP for the L1 and L2 controllers respectively.(All relevant book-keeping necessary for the cache block was specified in the SSP)
2. Protogen was used to generate the protocol implementation in murφ from the SSP. The protocol was generated for the following configuration:
   - Atomic Transaction
   - Atomic Requests
   - Blocking
   - Allow R/W accesses in transient states
3. The generated murφ code was verified for safety and deadlock using the murφ model checker[12].
4. On successful verification, MurphiToBSV tool was used to generate the cache controller and translation functions in BSV.

The generated code was tested for correctness using a nominal testbench with random test cases.

	Load	Store	Evict	Other GetM	Other GetS	Recall
M	-	Hit	PutM / I	PutM / I	PutM / S	PutM / I
S	-	GetM / M	- / I	- / I	-	- / I
I	GetS / S	GetM / M	-	-	-	-

Table 8.1: Snoopy MSI with recall: Stable state protocol specification for L1.

	Evict	GetM	GetS	PutM
M	Recall / I	Ack / M	Ack / S	- / I
S	Recall / I	Ack / M	Ack / S	-
I	-	Ack / M	Ack / S	-

Table 8.2: Snoopy MSI with recall: Stable state protocol specification for L2.

8.1.3 IncoreCLink

8.1.3.1 Development

The fabric was implemented according to the specifications and needs in BSV. Each channel in the specification is implemented as an individual FIFO structure in hardware. The topology of the implementation is catered towards a snoopy coherence protocol. Hence, it follows a mixed bus architecture. The channels A and B are implemented as a broadcast bus, as all the requests(GetS, GetM etc) are transmitted on these channels. This ensures that all the caches receive the coherence requests. The channels C,D and E are implemented as crossbar switches, this ensures maximum concurrency and ensures maximum throughput in the bus. The mixed bus architecture reduces power consumption while increasing the performance of the system.

The channels A and B are implemented as broadcast buses and channels C,D and E are implemented as crossbar switch fabrics. This results in the following behaviour:

- Any packet on channels A & B are broadcasted to every agent connected.
- All communication on channels C,D & E is point to point i.e the packet is routed only to the intended recipient.

8.1.3.2 Testing

Model agents which respond to the messages according to the specification were implemented in BSV. A Linear Feedback Shift Register(LFSR) was used to generate random traffic for the bus. The random traffic was injected on Channel A and all the traffic on the bus was monitored. The bus performance was verified to ensure that the performance was satisfactory. Using LFSR based testing strategies has a salient advantage. It enables one to generate synthesizable code which can then be run on a FPGA. This would prove useful for verification of the system during the final stages of development.

8.1.4 System Architecture

The memory subsystem architecture consists of the following components:

- L1 DCaches
- L2 Cache
- Coherence message translation agents for the L1 Caches
- Coherence message translation agent for the L2 Cache
- Coherence fabric

The architecture of the system is represented in Figure 8.1. The L1 DCaches and L2 Cache were implemented in BSV as per the design specification in . The values chosen for the implementation are listed in Table 8.3. The cache entries were implemented as 3

Variable	Value
$paddr$	32
$xlen$	32
sz_line	256
num_sets	64
n_l1	2
$l2\_num\_sets$	64
n_l2	4

Table 8.3: Values of different variable for caches.

individual entities. Each of the entities are implemented as an array of register files. The number of registers in the file is decided by the number of sets and the number of register

files in the array is decided by the number of ways in the cache. This ensures maximum concurrent access while satisfying parameters for timing, area and power. The latency of accesses for L1 DCaches are listed in Table 8.4 and for L2 Cache are listed in Table 8.5.

Scenario	Latency in Clock Cycles
Hit	1
Miss	5 to 6
Insufficient Permissions	5

Table 8.4: L1 DCache access latencies.

Scenario/Operation	Latency in Clock Cycles
Hit	1
Miss	2
Flush	4 to 8
Invalidate	4

Table 8.5: L2 Cache access latencies.

8.2 Synthesis Results

Hardware synthesis for each of the individual hardware modules were performed using Xilinx Vivado HLS Design Suite. Xilinx Artix-7 FPGA AC701 was used as the target board for the synthesis. The synthesis results are reported in Table 8.6. All the designs met the timing requirements of 15 ns.

Module	LUT	FF
L1 DCache	10330	2377
L2 Cache	6626	2679
L1 Coherence Agent	2183	3287
L2 Coherence Agent	1465	1564
Coherence Bus	9322	14684

Table 8.6: Hardware utilisation reports from synthesis.

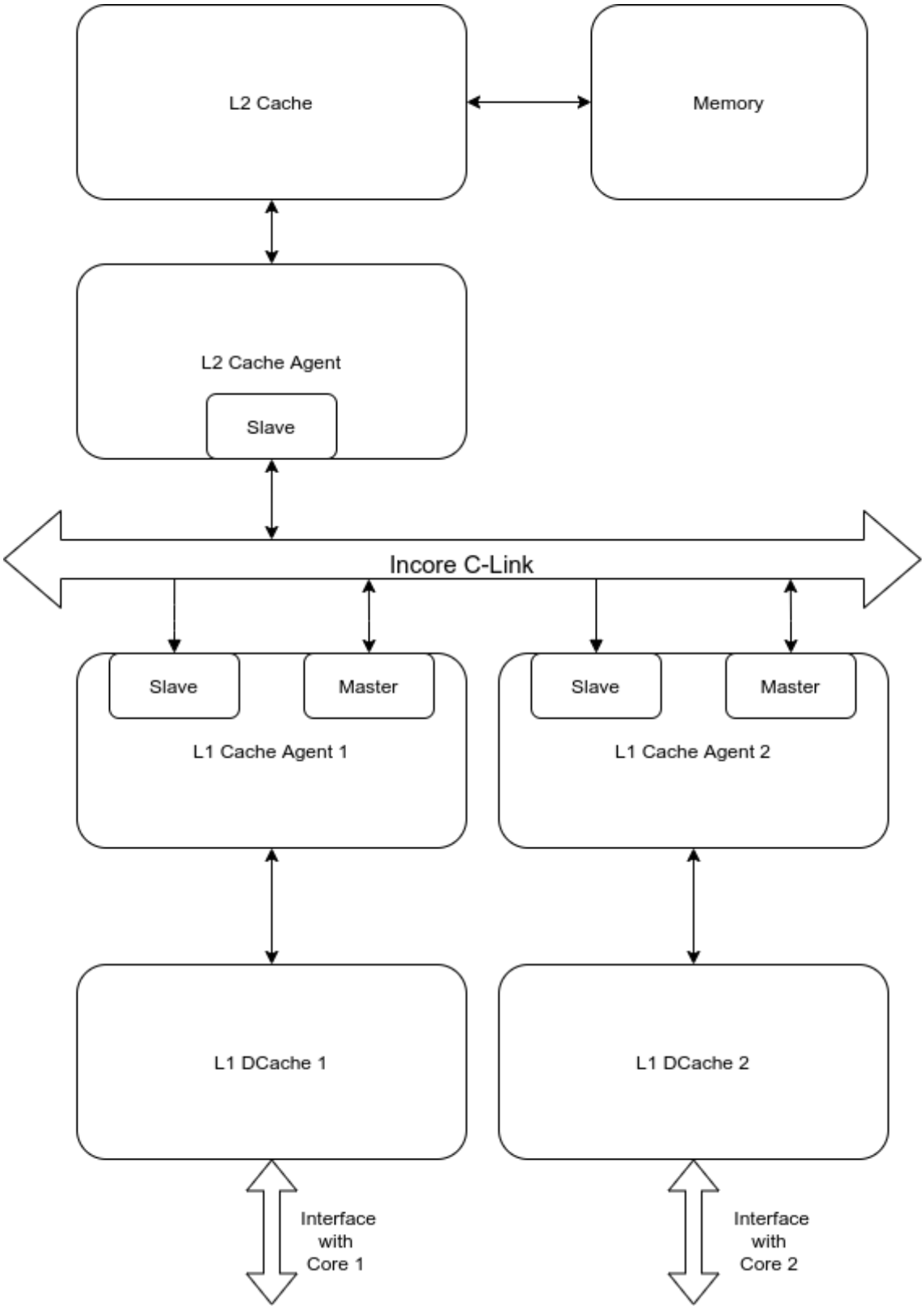

Figure 8.1: Architecture of the Memory Subsystem.

CHAPTER 9

Conclusion

Cache coherence has been a key design parameter since processors started encompassing multiple cores in order to exploit the task level parallelism in programs. There are numerous ways to implement coherency in a system and each method offers different trade-offs. As the industry shifts focus towards domain specific hardware, each method might find a particular use case which is optimal. This calls for a modular and scalable design with maximum re-usability across components. This report highlights the feasibility of such a design methodology and offers solutions to minimise human errors in design. The suggested method generates formally verifiable protocols, thereby increasing confidence in the design and decreasing the resources necessary for design verification in the later stages. The synthesis results are a testament to the feasibility of such a design and show promising improvements to the performance of the system.

9.1 Further Improvements

The cache entries can be implemented as BRAM arrays instead of register files. This would decrease the area and power consumed and improve performance on silicon tapeouts. The subsystem can be integrated with the cores and L1 ICaches and memory management units to instantiate a multicore system and run RISC-V programs on it. The standard memory model supported by RISCV is RVWMO, a weak memory model. Hence the coherence protocol can be enhanced to leverage the relaxed nature of the memory model to provide higher performance and greater concurrency. Further support for multiple bus protocols and multiple physical implementations of the same can be added to enhance the configurability of the design.

Another optimisation possible is to design the caches to be non-blocking i.e to handle hits with misses pending to be serviced. This would greatly enhance the throughput of the system and would help decrease the CPI of an out-of-order system. The system can also

be improved further by adding a victim cache in-between the L1 and the L2 caches. The CMO handler in the L2 cache can also be extended to support advanced operations like prefetch and zero allocate to be employed by just in time compilers and other software with self modifying code.

APPENDIX A

SSP of Directory MI

```
# NrCaches 3
Network { Ordered fwd;    //FwdGetS, FwdGetM, Inv, PutAck
          Ordered resp; // Data, InvAck
          Ordered req;   //GetS, GetM, PutM
          };
Cache {
    State I;
    Data cl;
} set[NrCaches] cache;
Directory {
    State I;
    Data cl;
    ID owner;
} directory;
Message Request{};
Message Ack{};
Message Resp{
    Data cl;
};
Message RespAck{
    Data cl;
};

Architecture cache {
    Stable{I, M}
    // I ////////////////////////////////////////
    Process(I, store, State){
```

```
28          msg = Request(GetM, ID, directory.ID);
29          req.send(msg);
30          await{
31              when GetM_Ack_D:
32                  cl=GetM_Ack_D.cl;
33                  State = M;
34                  break;
35          }
36      }
37      Process(I, load, State){
38          msg = Request(GetM, ID, directory.ID);
39          req.send(msg);
40          await{
41              when GetM_Ack_D:
42                  cl=GetM_Ack_D.cl;
43                  State = M;
44                  break;
45          }
46      }
47      // M ////////////////////////////////////////
48      Process(M, load){
49      }
50      Process(M, store, M){}
51      Process(M, Fwd_GetM, I){
52          msg = Resp(GetM_Ack_D, ID, Fwd_GetM.src, cl);
53          resp.send(msg);
54      }
55      Process(M, evict, State){
56          msg = Resp(PutM, ID, directory.ID, cl);
57          req.send(msg);
58          await{
59              when Put_Ack:
```

```
                State = I;
                break;
            }
        }
}

Architecture directory {
    Stable{I, M}
    // I ////////////////////////////////////////
    Process(I, GetM, M){
        msg = Resp(GetM_Ack_D, ID, GetM.src, cl);
        resp.send(msg);
        owner = GetM.src;
    }
    // M ////////////////////////////////////////
    Process(M, GetM){
       msg = Request(Fwd_GetM, GetM.src, owner);
       fwd.send(msg);
       owner = GetM.src;
    }
    Process(M, PutM){
       msg = Ack(Put_Ack, ID, PutM.src);
       fwd.send(msg);
       if owner == PutM.src{
            cl = PutM.cl;
            State=I;
       }
    }

}
```

REFERENCES

[1] "Chromite Core Documentation Chromite Core Generator 0.9.6 documentation." [Online]. Available: https://chromite.readthedocs.io/en/latest/index.html

[2] "Chromite M Soc Manual Chromite M SoC Manual 0.9.9 documentation." [Online]. Available: https://chromitem-soc.readthedocs.io/en/0.9.9/

[3] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[4] "Volume 1, Unprivileged Spec v.20191213." [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf

[5] S. Zhang, M. Vijayaraghavan, and Arvind, "Weak memory models: Balancing definitional simplicity and implementation flexibility," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 288–302.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 18, pp. 15–26, 10 1995.

[7] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A Primer on Memory Consistency and Cache Coherence, Second Edition," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294. [Online]. Available: https://www.morganclaypool.com/doi/10.2200/S00962ED2V01Y201910CAC049

[8] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Washington, DC, USA: IEEE Computer Society Press, 1986, p. 414423.

[9] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: Automatically generating directory cache coherence protocols from atomic specifications," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 247260. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00030

[10] D. L. Dill, "The murφ verification system," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 390–393.

[11] O. Arcas-Abella and N. Sonmez, *Bluespec SystemVerilog*. Cham: Springer International Publishing, 2016, pp. 165–172. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_9

[12] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli, "Exploiting transition locality in automatic verification of finite state concurrent systems," *Sttt*, vol. 6, no. 4, pp. 320–341, 2004.

[13] "Sifive tilelink specification v.1.8.1." [Online]. Available: https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf

[14] "Volume 2, Privileged Spec v.20190608." [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-privileged.pdf

[15] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," *SIGARCH Computer Architecture News*, vol. 14, no. 2, p. 414423, May 1986. [Online]. Available: https://doi.org/10.1145/17356.17404

[16] N. Oswald, V. Nagarajan, and D. J. Sorin, "Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 888899. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00077