CACHE MEMORY DESIGN AND ITS INTEGRATION WITH NOC FOR A MULTI-CORE RISCV PROCESSOR

A Project Report

submitted by

VEMPATI NOEL JONES

EE16B126

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY &

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **CACHE MEMORY DESIGN AND ITS INTEGRATION WITH NOC FOR A MULTI-CORE RISCV PROCESSOR**, submitted by **Vempati Noel Jones (EE16B126)**, to the Indian Institute of Technology, Madras, for the award of the degree of **the degree of Bachelors of Technology and Master of Technology**, is a bonafide record of the project work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V. Kamakoti
Project Guide
Professor
Department of Computer Science and Engineering
IIT-Madras, 600 036

Dr Janakiraman Viraraghavan
Project Co-Guide
Assistant Professor
Department of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 26^{th} June 2021

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my project advisor, Prof. V. Kamakoti sir, for giving me this opportunity to be part of this project. I would like to thank my mentor M.J. Shankar Raman sir for his immense guidance and support throughout the course of my project. I would like to express my gratitude to Dr. Janakiraman Viraraghavan sir for supporting me being my co-guide for the project.

Finally, I would like to thank my family and friends for their constant encouragement and support throughout my education and project.

ABSTRACT

KEYWORDS: SoC, Network on Chip, Cache Memory, LLC, Vivado Design Suite

SafeRV is a multi-core RISCV processor that uses a 2-level cache hierarchy and coherency is maintained using Directory-based MSI protocol. Each core has a private L1 cache that has separate Data and instruction caches. Whereas L2 cache is a memory side cache that acts as the LLC and is shared among all the cores These are integrated to SafeRV considering the Network on Chip and other compatibility requirements. Source Code is in Bluespec System Verilog. Compilation and verilog Generation were done using Bluespec Compiler and simulation in Vivado Design Suite.

Note: Due to Covid-19 norms and regulations, Hardware required for the project (i.e. FPGA board) present in the lab in campus was inaccessible. Hence entire work was based on simulations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

SoC	System on Chip
NoC	Network on Chip
LLC	Last Level Cache
MSI	Modified Shared Invalid
BSV	Bluespec System Verilog
BSC	BlueSpec Compiler
LFSR	Linear Feedback Shift Register
UART	Universal Asynchronous Receiver Transmitter
BRAM	Block Random Access Memory
CSR	Control Status Register
TLB	Translation Lookaside Buffer
VIPT	Virtually Indexed Physically Tagged
SRAM	Static Random Access Memory
PLRU	Pseudo Least Recently Used
PIPT	Physically Indexed and Physically Tagged

CHAPTER 1

SAFERV INTRODUCTION

SafeRV is a RISC-V based multi-core processor for safety critical applications. It has 4 Shakti C-Class cores (64 bit, RV64IMAFDC, 6 stage in-order pipeline) instantiated. Shakti C-Class core includes separate I-Cache and D-Cache and Performance Counters

Multi-core SoC employs OpenSMART NoC generator with criticality for the underlying fabric. NoC is generated using OpenSMART - all routers, network interfaces are generated. Just attach the required modules to them in a convenient layout.

Y/X	0	1	2	3
0	T	H	H	T
1	T	C	C	B
2	T	C	C	U
3	T	H	H	T

Table 1.1: OpenSMART layout for the SoC

H HART (Shakti C-Class Core)

C LLC bank

T Tie-off Node (Any packets received at this node will be dropped off safely

U UART

B BRAM

CSR-0x801 is used to drive the criticality of each packet generated by the cores.
A 256KB BRAM is used as a substitute for Main-Memory

It has banked distributed shared L2 cache. There are 4 such banks and are round robin based.

ProtoGen is used to generate Bluespec codes for cache controllers to implement directory-based MSI protocol.

ShaktiLink is used as the bus protocol which is a custom link designed so as to not make any changes to the cache controllers which were generated by ProtoGen.

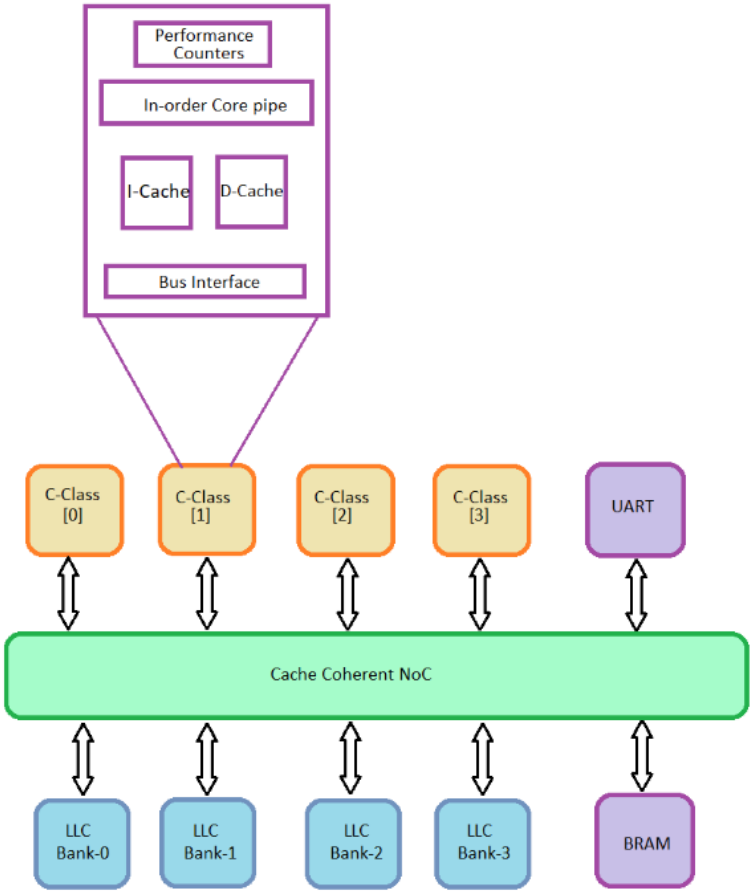

Figure 1.1: Block Diagram of SoC

CHAPTER 2

MAIN COMPONENTS OF SAFERV

2.1 Shakti C-Class Core

C-Class core is a member of the Shakti family of processors. It is an extremely configurable and commercial-grade 6-stage in-order core.

It supports the standard ISA=RV64IMAFDCSUN extensions based on RISCV-spec-2.2 and privilege-spec-1.10

It has parametrized blocking Instruction and Data caches.

It supports Single and Double Precision Floating point units.

It has early out multiplier and a restoring divider.

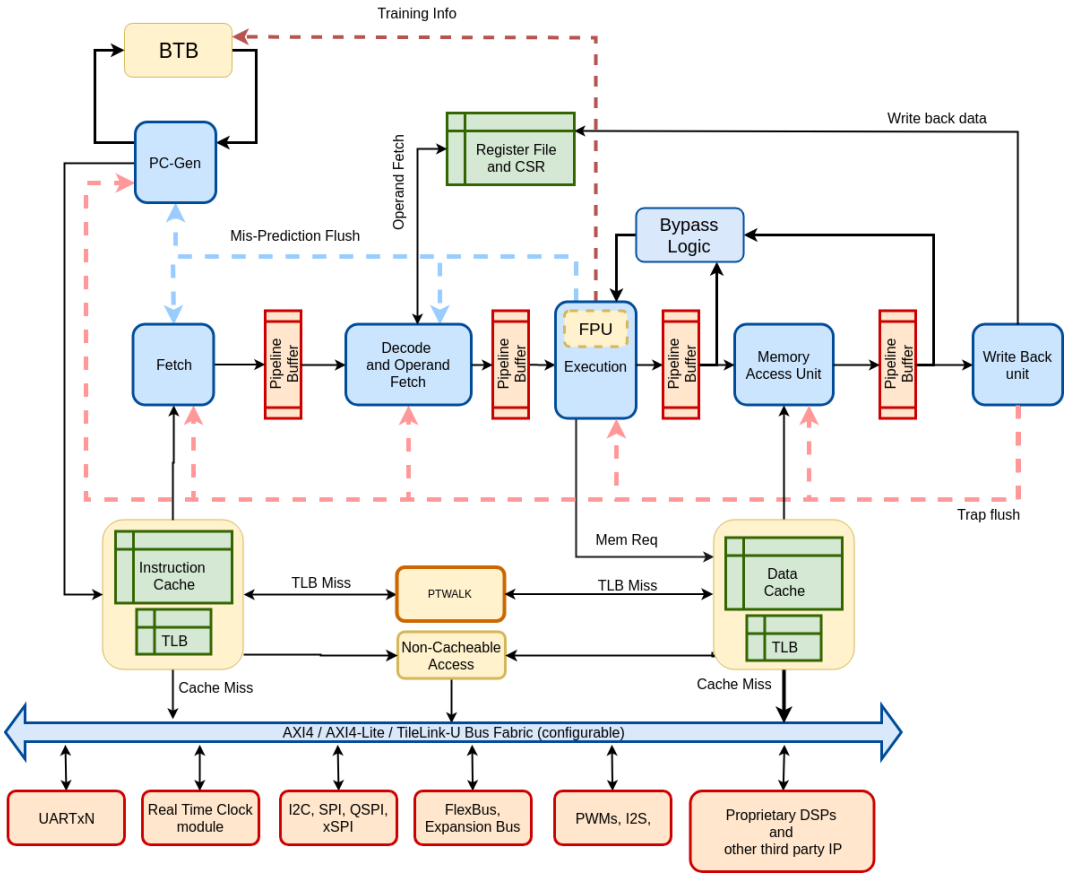

Figure 2.1: Block Diagram of Shakti C-Class core

2.2 Network on Chip

Network on Chip is a key component of almost all the chips today. It is the interconnect backbone connecting IPs communicating each other.
NoC differentiates between critical and non-critical traffic. It defines two criticality levels. HI critical traffic and LO critical traffic.
When in HI criticality mode priority is given to the HI critical VCs over LO critical ones (Round robin policy is used among HI critical VCs) If HI critical packets are blocked, then LO critical packets would make progress
When in LO criticality mode, a round robin policy is used among all VCs irrespective if crirticality.

The latency of traversal between two IPs is proportional to the number of hops between them. SMART NoC is a single-cycle multi-hop traversal network design that reduces the average flit hop counts in a mesh-based designs. OpenSMART is an automated tool for generating SMART NoC.

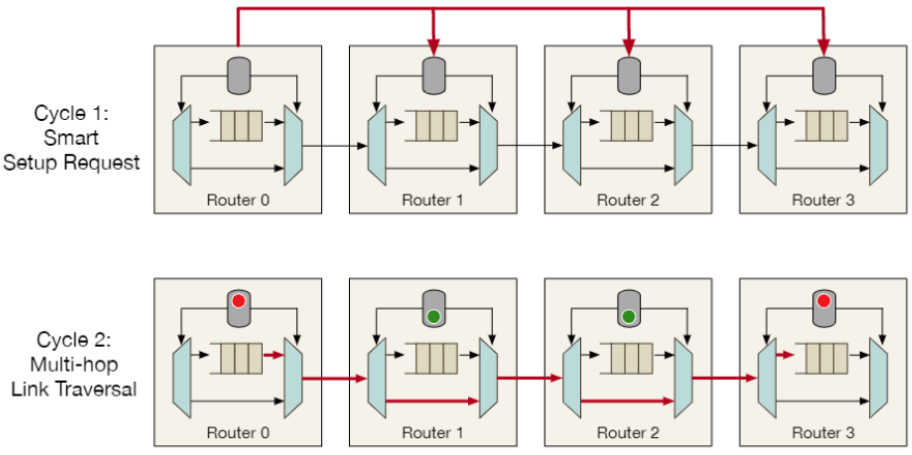

Figure 2.2: Example of single cycle multi hop traversal

SMART leverages wire delay of the underlying repeated wires and augments each router with the ability to request and setup bypass paths. OpenSMART takes SMART from a NoC optimization to a design methodology for SoC, enabling users to generate verified RTL for a class of user specified network configurations such as network size, topology, routing algorithm, number of VCs/buffer, router pipeline stages and so on.

Language	BSV and Chisel
Flow Control	VC, SMART
Topology	Mesh, Arbitrary topologies
Buffer Management	Credit
Router Micro architecture	1 cycle, 2 cycle, SMART
Packet size	1 flit
Traffic Generator support/Stress Test	Uniform-random, Bit-complement

Table 2.1: OpenSMART features

2.2.1 Additional Features added to OpenSMART

1. Support for various message classes
2. Multi-flit support
3. Support for criticality aware arbitration
4. LFSR based traffic generators which can be ported to FPGAs as well

2.2.2 Support for Performance counters at NoC level

1. Each router maintains a set of configurable registers which act as event filters
2. Filters include address range, inject/eject, asid, target etc
3. Each router sends out minimal number of toggling signals and thus provide run time monitoring without affecting the network traffic

2.3 Shakti Link

Shakti-Link is the custom interconnect standard used in the SoC. It is a highly modified version of TileLink which has 5 channels. Shakti Link is used due to overcome compatibility issues with cache controllers generated by ProtoGen

It is a master-slave protocol and each master will have a separate slave interface. This is just a interface that connects IP/Core and fabric.
node-slave of ShaktiLink is slave for a core(Master).
Similarly node-master is master for IP(Slave) like Directory Controller

ShaktiLink has three channels:

1. Request type channel (Channel A): From master to slave
2. Forward type channel (Channel B): From slave to master
3. Response type channel (Channel D): From slave to master

CHAPTER 3

CACHE MEMORY

SafeRV uses a 2 level cache hierarchy. Each core has a private L1 cache that has separate Data and Instruction caches in it. L2 is a memory side cache that acts as the LLC and is shared among all the cores

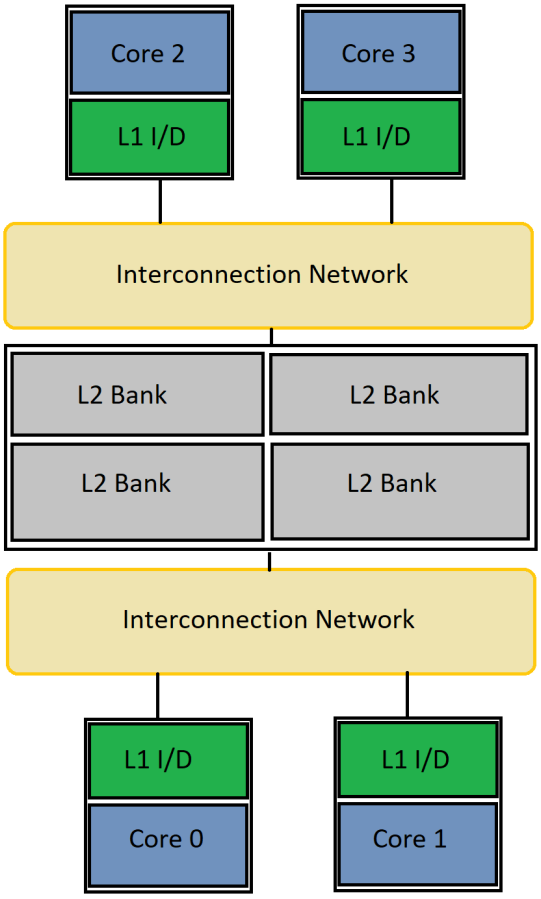

Figure 3.1: Cache Hierarchy

3.0.1 Cache size and other specifications

There is a 2-level cache hierarchy with coherence maintained using Directory-based MSI protocol. The cache controllers are generated using ProtoGen

ProtoGen generates Murphi code for a coherence protocol given a Stable state specification of a protocol. A bluespec back-end to generate BSV from the generated Murphi code was built in-house and added to ProtoGen.
Each core has a Data cache and Instruction cache which is local to the core. The shared cache, L2 is banked(4 such banks). L1 I-cache and D-cache are set associative, 128 sets. LLC(L2 cache) has a Round Robin Replacement policy.

3.1 Instruction-Cache

Features of Instruction Cache:

1. Direct Mapped
2. 128 Sets
3. VIPT cache
4. Blocking
5. Writes are not supported
6. All cache lines are flushed on a fence operation
7. Replacement policies: Random (4-bit LFSR based), Round Robin and Tree-PLRU (Max of 4 ways are supported)

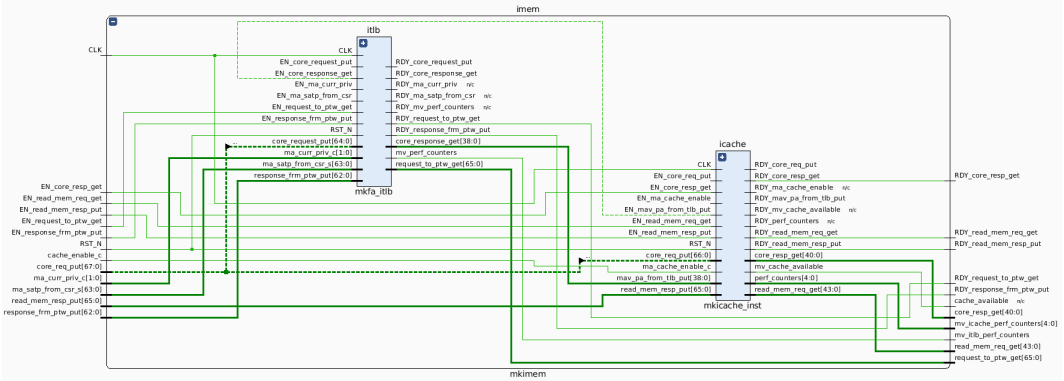

Figure 3.2: Imem internal modules and interfaces

I-Cache constitutes of the Cache module(all the above, fill buffer, SRAM and required logic) and the Cache replacement policy module. The I-Cache together with instruction TLB is referred to as instruction memory. This interfaces directly with the core on one side and with the fabric in the other side so as to communicate with next level cache/memory

Instruction SRAM: It stores the instructions -one per way. This RAM has s single port
Tag SRAM: It stores tag for each cache line. Design is similar to Instruction SRAM
Fill buffer stores cache lines for which requests are sent to he next level of memory hierarchy. It is implemented as a set of registers that are used in a circular fashion using head and tail pointers. Each fill buffer entry stores information on data in cache line, address of the cache line, valid bit and error bit.

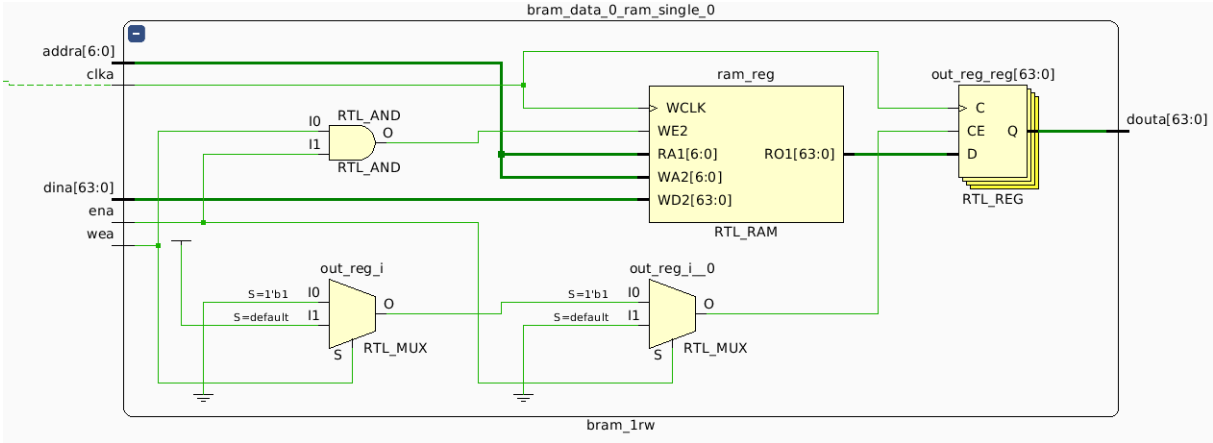

Figure 3.3: I-Cache Data RAM

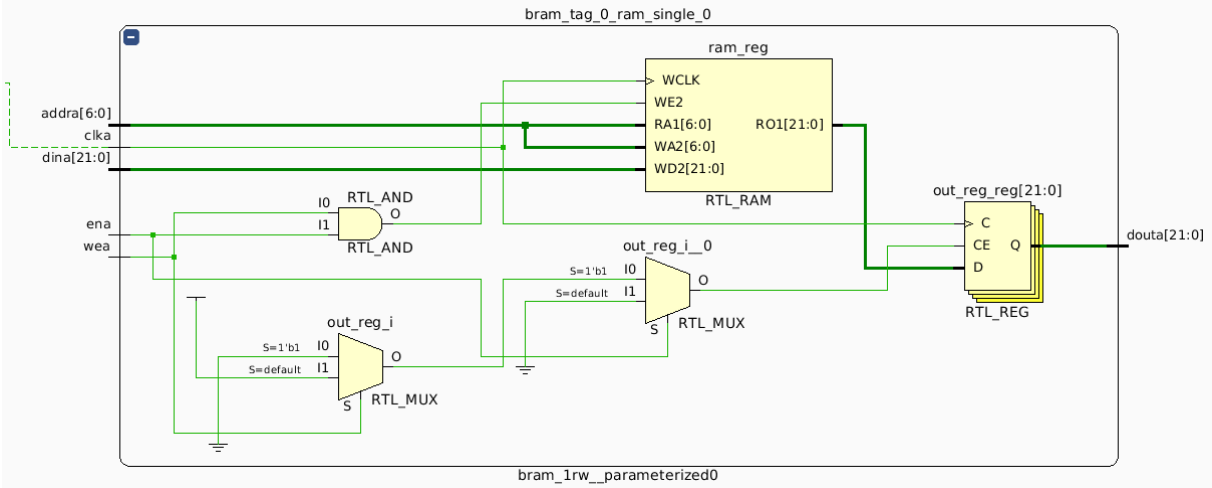

Figure 3.4: I-Cache Tag RAM

Both cacheable and non-cacheable reads and writes reach the icache. However all the non-cacheable reads are directly sent to the next level by bypassing both fill buffer and SRAMs.

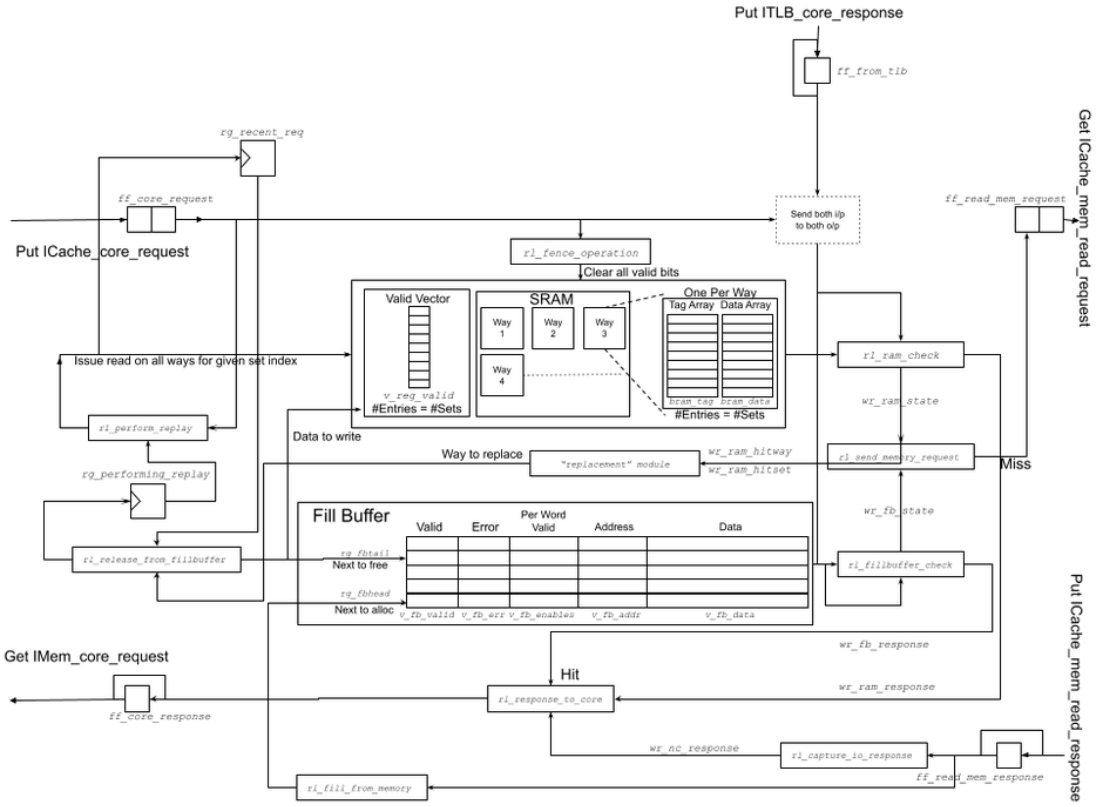

Figure 3.5: I-Cache components and interactions

3.2 Data Cache

Features of Data-cache:

1. Direct Mapped
2. 128 Sets
3. VIPT cache
4. Blocking(on a miss)
5. Write allocate(on write miss)
6. Write Back
7. All cache lines are flushed on a fence operation
8. Replacement policies: Random (4-bit LFSR based), Round Robin and Tree-PLRU (Max of 4 ways are supported)

D-cache constitutes of coherence logic, cache replacement policy module, store buffer module, and the cache(all the above,fill buffer,SRAM cells and required logic). D-cache together with data TLB is referred as data memory. This interfaces directly with the core on one side and with the fabric on the other side so as to communicate with the next level cache or memory.

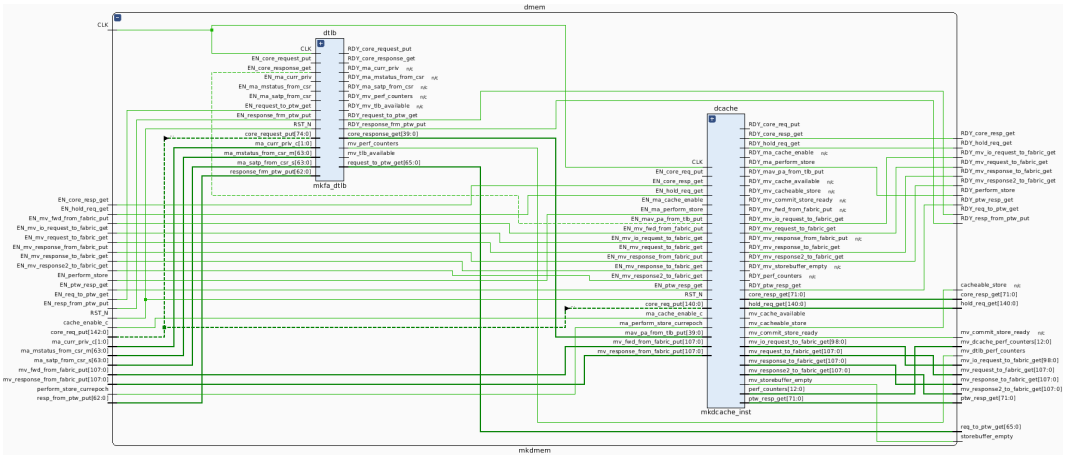

Figure 3.6: Dmem internal modules and interfaces

Data SRAM: Data is stored in multiple SRAMs- one per way. The ram has 2 ports- one port is used for all read/write operations and other port is used only when data needs to be accessed from SRAM due to an incoming coherence message.

Tag SRAM: Stores tag for each cache line. Design is similar to Data SRAM

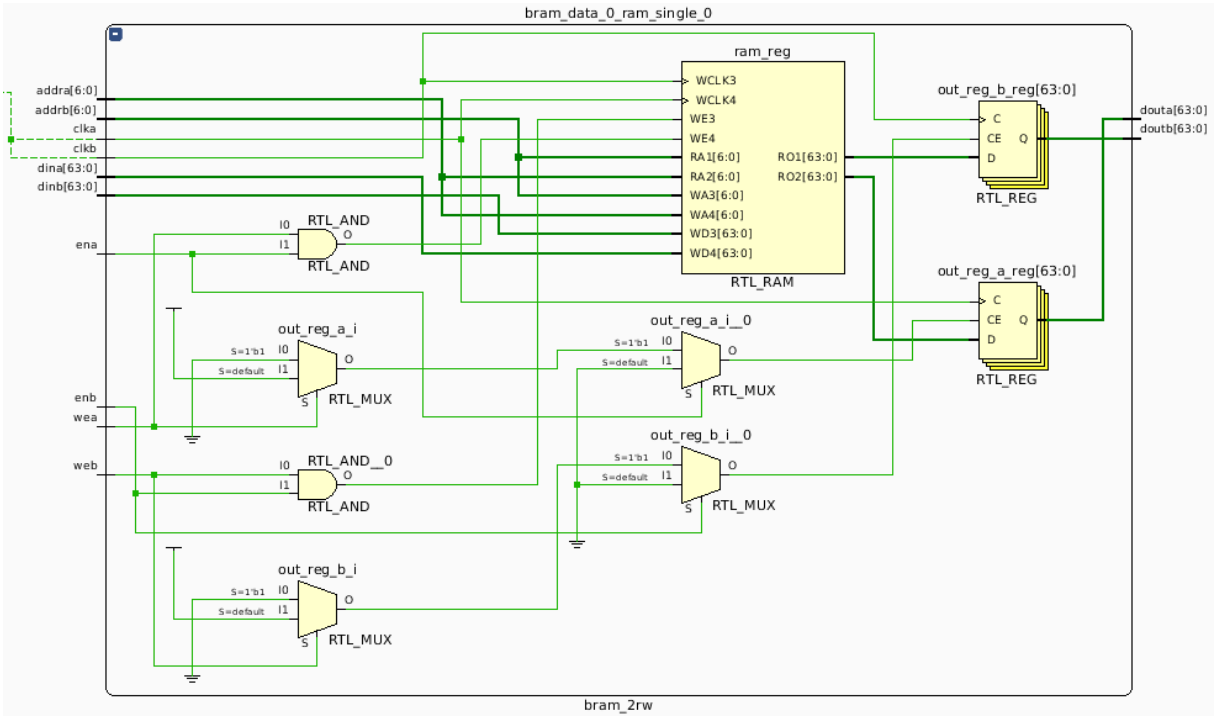

Figure 3.7: D-Cache Data RAM

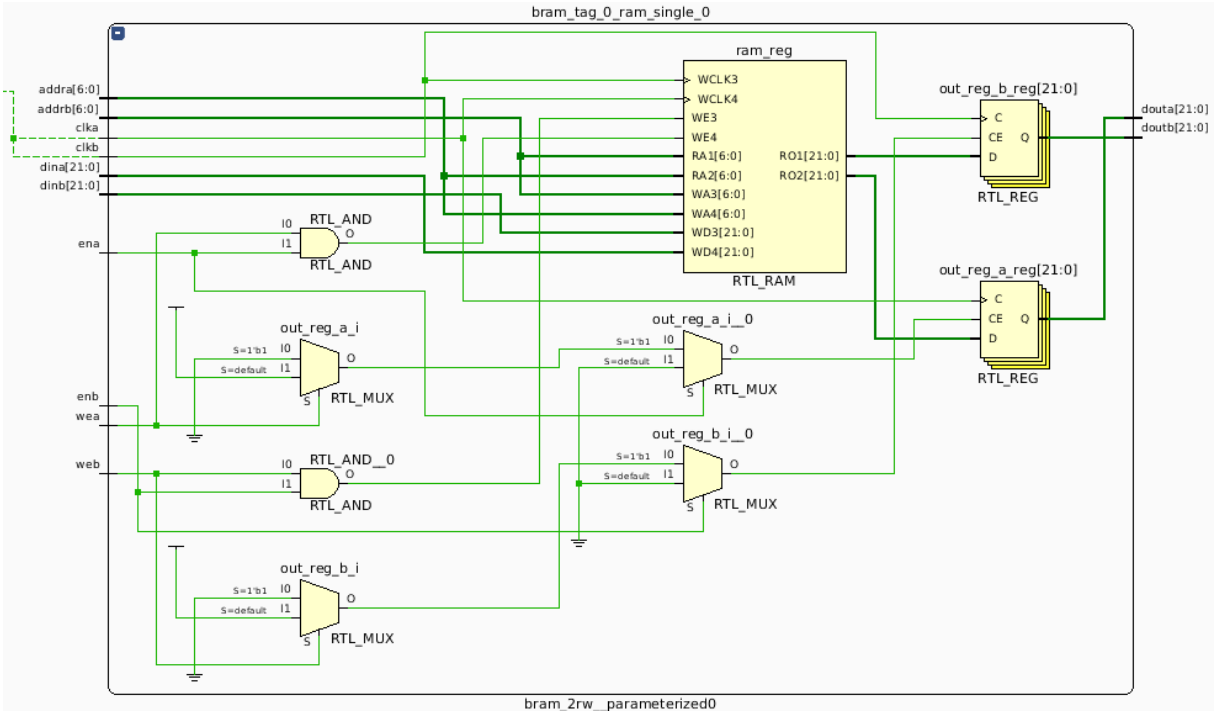

Figure 3.8: D-Cache Tag RAM

Coherence State: The coherence state is maintained in a vector of registers- one per cache line. It stores coherence states, acknowledgements expected and received.
Store Buffer: Stores uncommitted writes. Implemented as a circular queue using a set of registers.
Fill buffer: It stores cache lines for which requests are sent to the next level of memory hierarchy. It is implemented as a set of registers that are used in a circular fashion using head and tail pointers. Each fill buffer entry stores information on data in cache line. address of the data in the cache line, valid bit, error bit and coherence metadata.

Both cacheable and non-cacheable reads and writes reach the D-Cache. However, all non-cacheable reads and writes are sent directly to the next level by bypassing both fill buffer and SRAM cells. However all non-cacheable stores are first written to the store buffer until they are committed.

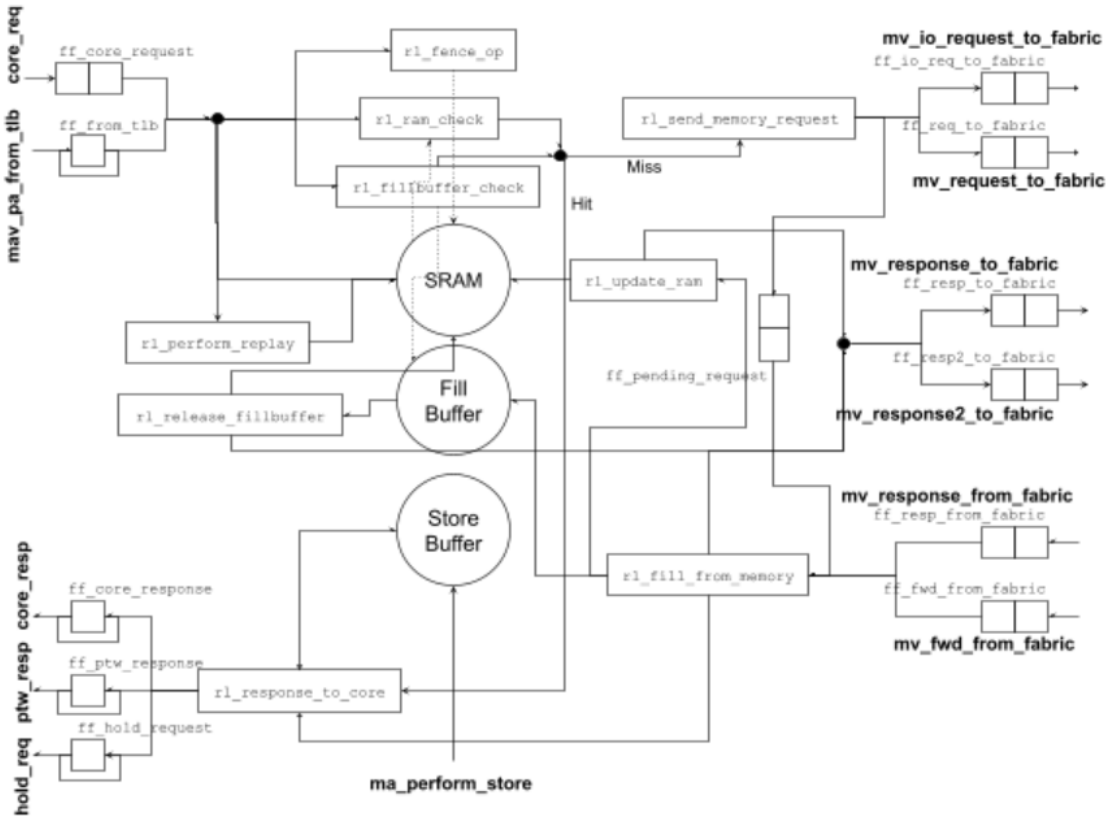

Figure 3.9: D-Cache components and interactions

3.2.1 Coherence

Coherence protocol is auto generated using the ProtoGen framework which automates the generation of a coherence protocol given the desired behavior. It takes as Stable state Specification of a Protocol(SSP) as an input and generates the logic needed for a coherence controller as a Murphi model. Generated output would include all possible transient states, transitions between them and actions to be taken on each incoming message. Using an in-house developed back-end, the Murphi model is converted to bluespec model which is used in the code.

An MSI protocol that is non-blocking is generated. Each cache line in the cache needs to maintain coherence state related and auxiliary information.
The transitions between states happen as per the following state diagram.

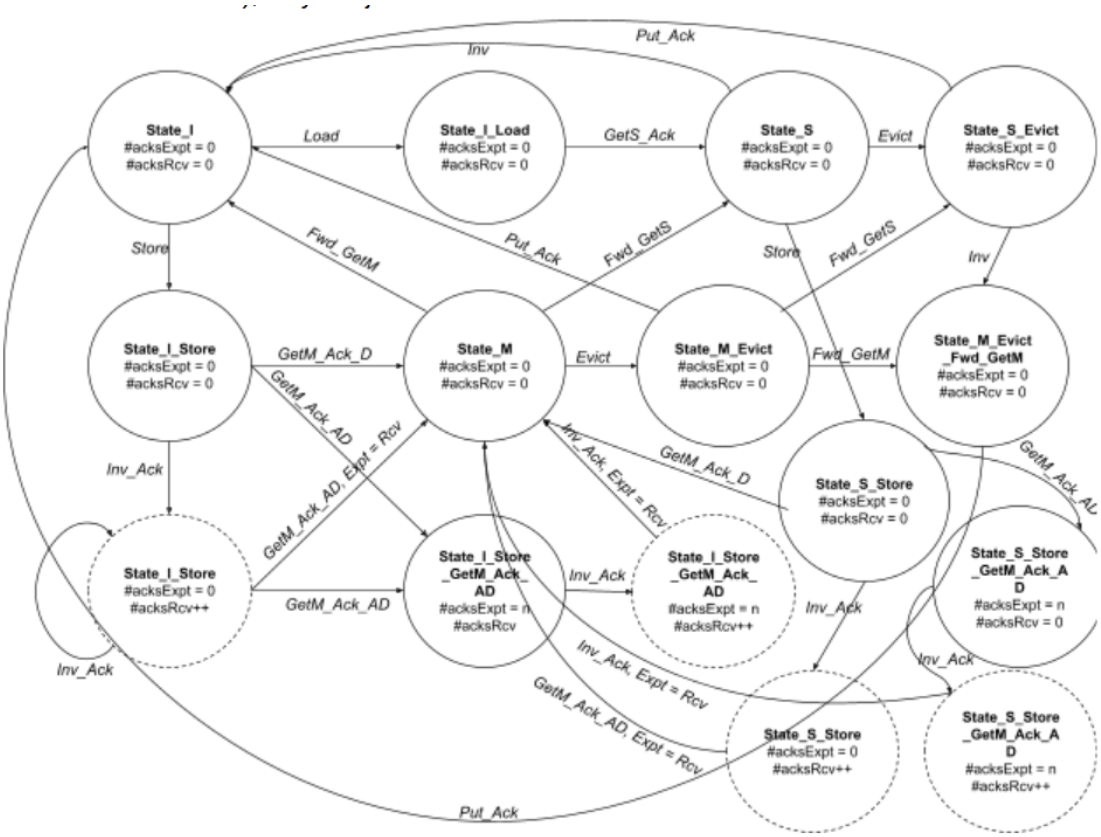

Figure 3.10: State diagram with possible transactions for a cache line

3.3 LLC

Features of LLC:

1. Number of ways is equal to number of cores i.e. 4
2. 128 Sets
3. PIPT cache
4. Blocking (on a miss)
5. Write allocate (on a write miss)
6. Write Back
7. Inclusion: Inclusive cache
8. Replacement policies: Round Robin

LLC is a memory side cache that is implemented as a distributed cache. Multiple instances of LLC cache banks are created in the SoC top level module and addresses are statistically partitioned between various LLC banks.
The design of LLC banks differ slightly from the design of L1 I-cache and D-cache. Due to the large number of lines in LLC, it would be significantly costlier to have the state of each line maintained in a register. Hence, unlike I-cache or D-cache, no valid bit registers or coherence metadata registers are used. Instead, the coherence metadata is directly stored in the SRAM as data of the cache lines. Also fill-buffers are not used for handling misses in the directory. The data is directly updated inline and any line to be evicted is stored in an independent register and then written to memory. The blocking nature of LLC makes sure that no requests/responses are missed due to this.

LLC uses dual ported SRAM to store data and tag(Port 1 for processing request from higher level caches and port 2 for processing requests from memory)

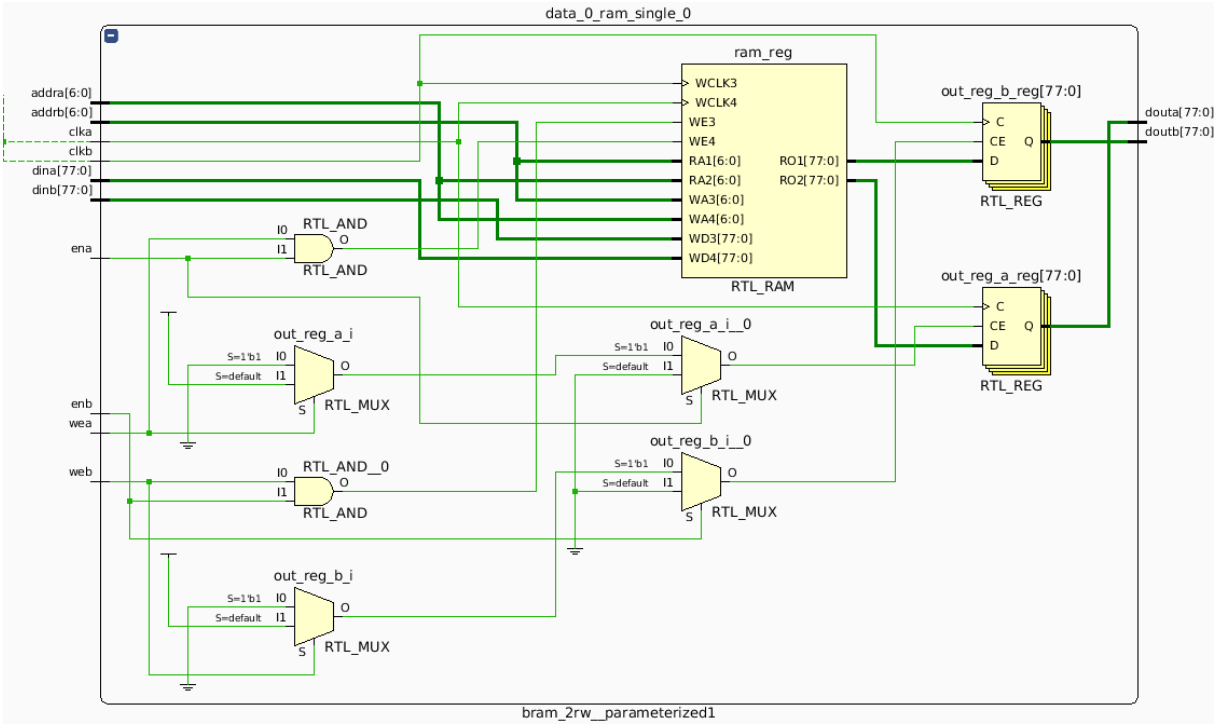

Figure 3.11: LLC Data RAM

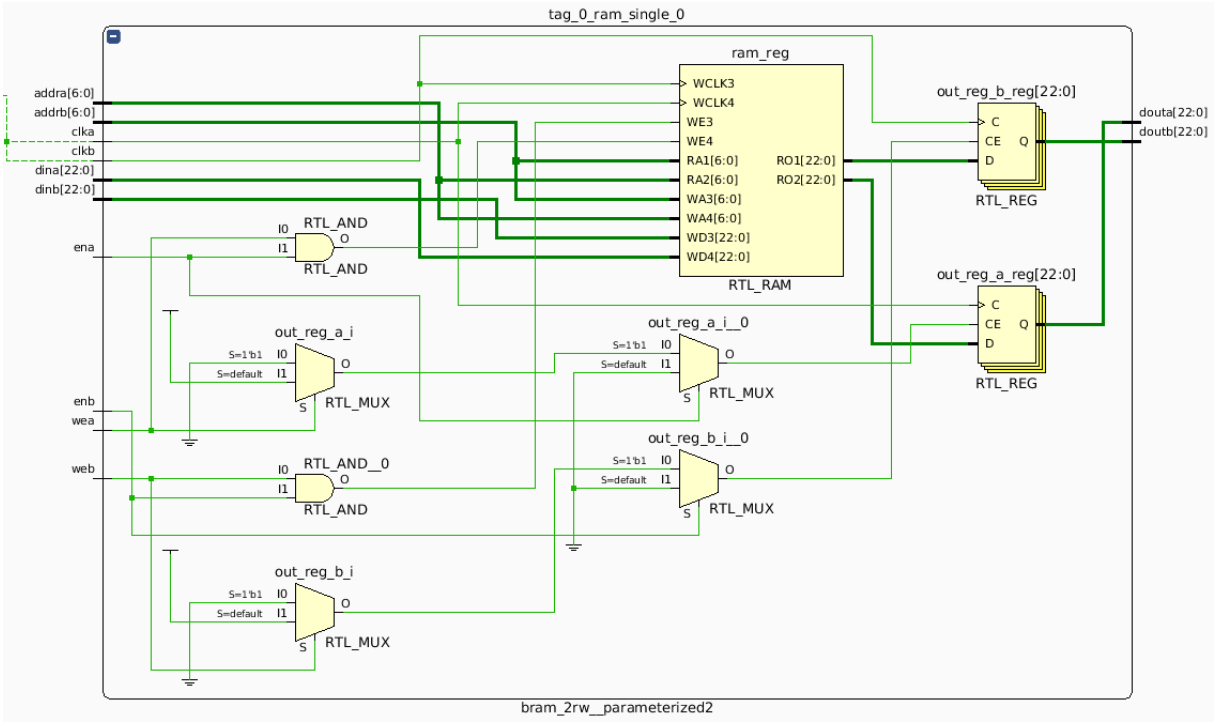

Figure 3.12: LLC Tag RAM

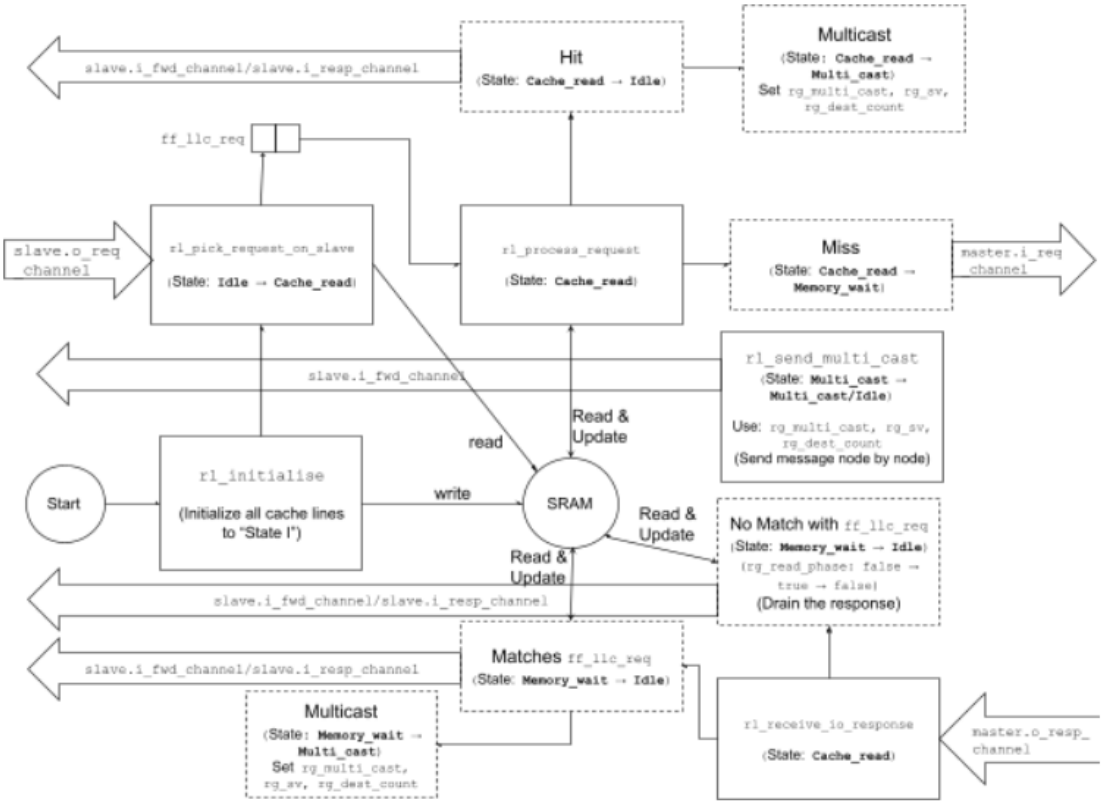

Figure 3.13: LLC components and interactions

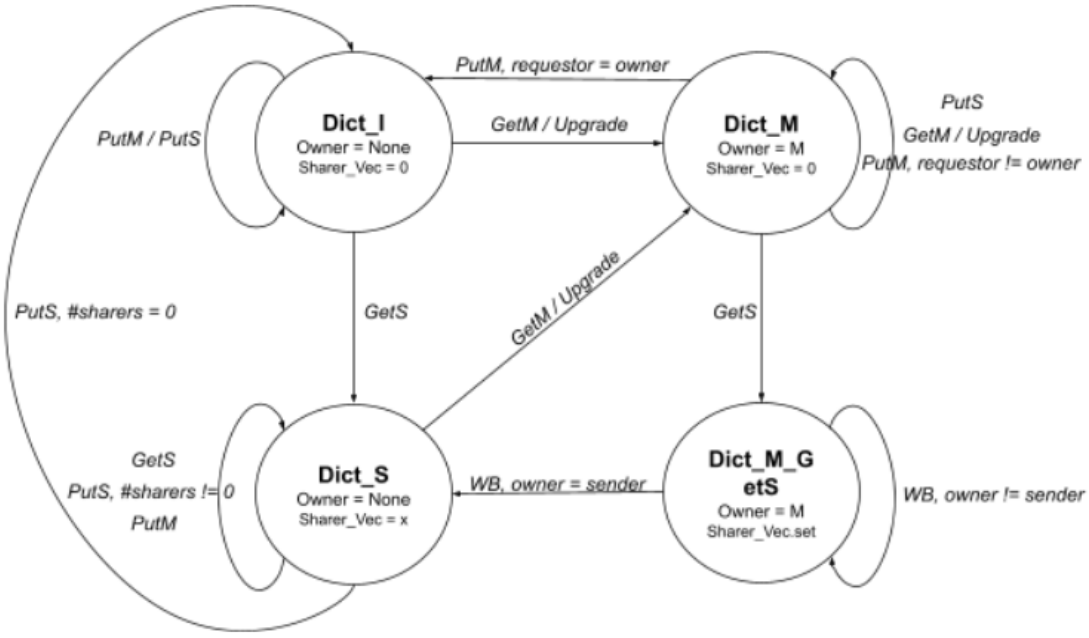

Figure 3.14: State diagram with possible transactions for a cache line in LLC

CHAPTER 4

TESTING AND INTEGRATION OF CACHES

4.1 I-Cache and D-Cache

Test benches were written in Bluespec System Verilog(BSV) for both Imem and Dmem as a whole which includes the following rules:

Common rules for imem and dmem;

1. Send core request for a particular address by enquiing the request in FIFO which is triggered by stimulus
2. Receive response from core and check if the output of cache is as expected or not.
3. Send memory read request
4. Send memory read response back
5. Checkout if the a test of simulation is passed
6. End the simulation and print the total test count of the simulation

Additional rules for dmem:

1. Perform store operation
2. Send memory write request
3. Send memory write response back

4.1.1 Simulation source

Python code was used as a simulation source which reads the parameters specified for the cache such as number of sets, ways, word size, blocksize, address width and replacement policy. It generates two files named "test.mem" and "data.mem". It fills random numbers in the data.mem file based on the max address and xlen.
The test format is as follows. "read/write: size: sign: delay/no-delay: fence/no-fence: Null: Address. These are written in "test.mem" file. Writing to test file algorithm is as follows:

```
if readwrite == 'read':
  rw=int(format(0,'#04b'),2)
elif readwrite == 'write':
  rw=int(format(1,'#04b'),2)
elif readwrite == 'atomic':
  rw=int(format(2,'#04b'),2)
else:
  rw=int(format(0,'#04b'),2)
if delaycycle == 'delay': d=0b1
else: d=0b0
if fencecycle == 'fence': f=0b1
else: f=0b0
if size== 'byte':
  s=int(format(0,'#04b'),2)
elif size == 'halfword':
  s=int(format(1,'#04b'),2)
elif size == 'word':
  s=int(format(2,'#04b'),2)
elif size == 'dword':
  s=int(format(3,'#04b'),2)
if sign == 'signed': sg=0b0
elif sign == 'unsigned': sg=0b1
# test format:
# read/write : size: sign: delay/nodelay :
                Fence/noFence : Null : Addr
upperbits=((rw<<6) | (sg<<5) | (s<<3) | (d<<2) | (f<<1) | f)
s = str(hex( (upperbits<<addr_width) | addr |
                    (data<<(addr_width+8))))
test_file.write(s[2:].zfill(nibbles)+'\n')
```

Then different test cases were defined in python to generate the test stimulus. Some of them are:

1. This test will generate consecutive requests on the same line. Each request will be a miss in the cache and probably a hit in the line buffer if present

```
write_to_file(0,read,word,unsigned,nodelay,fence)
for i in range(block_size):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+word_size
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

2. This test will generate consecutive requests to different sets of the cache. All should be a cold miss

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
for i in range(sets):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+(word_size*block_size)
for i in range(40):
    write_to_file(address,read,word,unsigned,delay,nofence)
address=4096
for i in range(sets):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+(word_size*block_size)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

3. This test will first fill a line and then generate a request on the same line after significant delay

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=address+4
write_to_file(address,read,word,unsigned,nodelay,nofence)
for i in range(20):
    write_to_file(address,read,word,unsigned,delay,nofence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

4. Filling a line, and after significant delay filling the next line and immediately generate request for the first line that was filled

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
for i in range(20):
```

```
    write_to_file(address,read,word,unsigned,delay,nofence)
address=address+(word_size*block_size)
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

5. This test will first fill a line. After significant delay it will then generate a request for all the words on the same line. They should all be hits in the cache

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
for i in range(20):
    write_to_file(address,read,word,unsigned,delay,nofence)
for i in range(block_size):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+word_size
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

6. Generating a cache request and then a IO request

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=32
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

7. This test will check for a possibility where a LB miss and cache miss will occur for different addresses (cache array output is registered). Request to a line which will be a cache miss, then request a word in the same line after a delay. Then request a line in different set would be miss

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
write_to_file(address,read,word,unsigned,nodelay,nofence)
for i in range(5):
    write_to_file(address,read,word,unsigned,delay,nofence)
address=4096+4
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=address+(word_size*block_size)
write_to_file(address,read,word,unsigned,nodelay,nofence)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

8. Creating thrashing scenario on the same set. Total requests is 2*ways

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
for i in range(ways+ways+ways+1):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+(word_size*block_size*sets)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

9. Fill a set completely. Then access the lines in that set from line 0 to 3. Then generate 2 misses to that set. This test is to assess how PLRU is affected by hits before generating those misses.

```
write_to_file(0,read,word,unsigned,nodelay,fence)
address=4096
for i in range(ways):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address+(word_size*block_size*sets)
address=4096+(word_size*block_size)
# a miss to fully fill the set
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096
# accessing lines from 0 to 3
address=address+(word_size*block_size*sets*(ways-1))
# this would set next_repl to be line 0
for i in range(ways):
    write_to_file(address,read,word,unsigned,nodelay,nofence)
    address=address-(word_size*block_size*sets)
address=4096
address=address+(word_size*block_size*sets*(ways))
# miss to the set, would replace line 0
#and set next_repl to line 2
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=address+(word_size*block_size*sets)
# another miss to the set, would replace line 2
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096+(word_size*block_size*2)
# miss to a different set to write back lb contents, line 0
# and 2 get replaced
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096+(word_size*block_size*sets)
# request to old line 2, should be a miss
write_to_file(address,read,word,unsigned,nodelay,nofence)
address=4096+(word_size*block_size*sets*(ways-1))
# request to old line 0, should be a miss
write_to_file(address,read,word,unsigned,nodelay,nofence)
write_to_file(maxaddr,atomic,dword,unsigned,delay,fence)
```

Generated files are used as stimulus for the testbench files for both Imem and Dmem and simulated.

4.1.2 Integration of I-Mem and D-Mem

Imem and Dmem interfaces with core on one side and the fabric on other side so as to communicate with next level cache or memory.

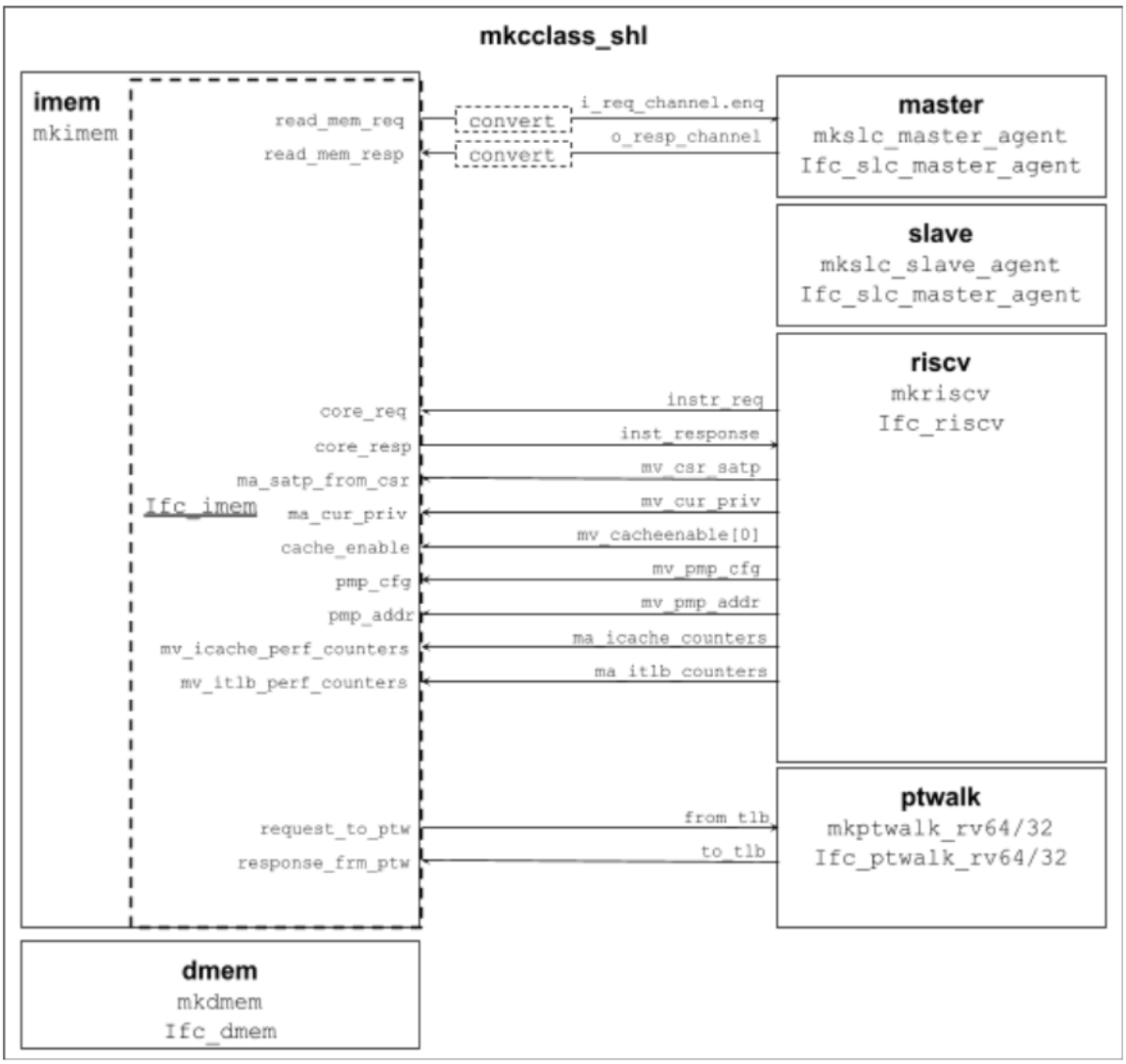

Figure 4.1: Imem integration

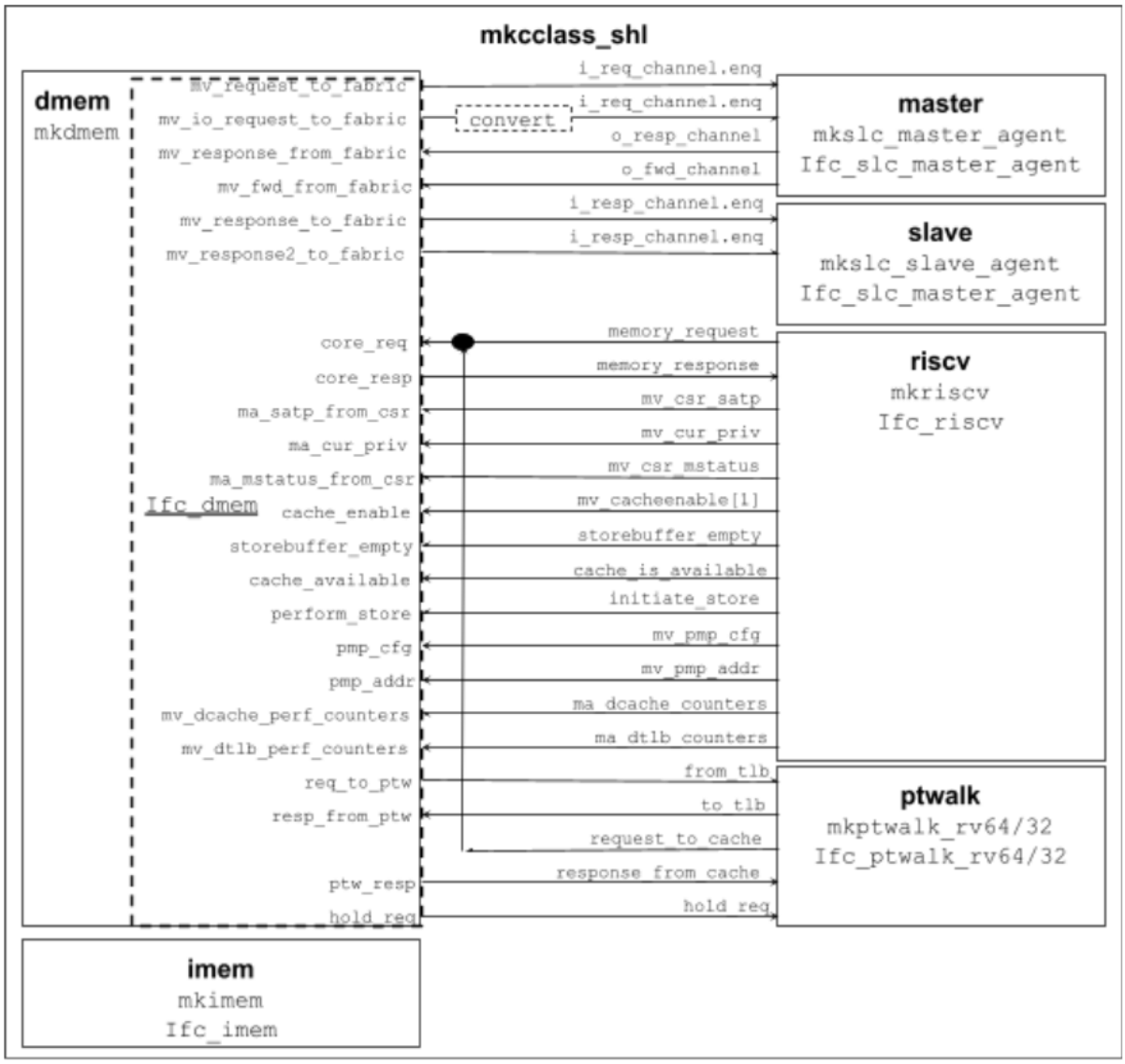

Figure 4.2: Dmem integration

4.2 Integration of LLC with NoC

Unlike I-Cache and D-Cache which are connected to RISCV core on one side and memory hierarchy on the other, LLC is uniform in the sense that it is connected to memory hierarchy on both sides. Hence it has a relatively uniform interface and connects to the NoC via ShaktiLink protocol.

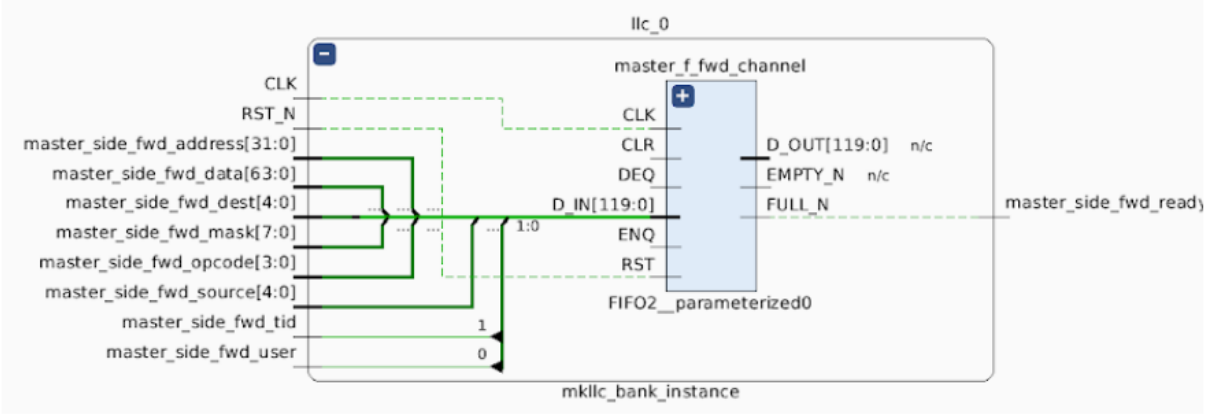

Figure 4.3: Master forward channel interface of LLC

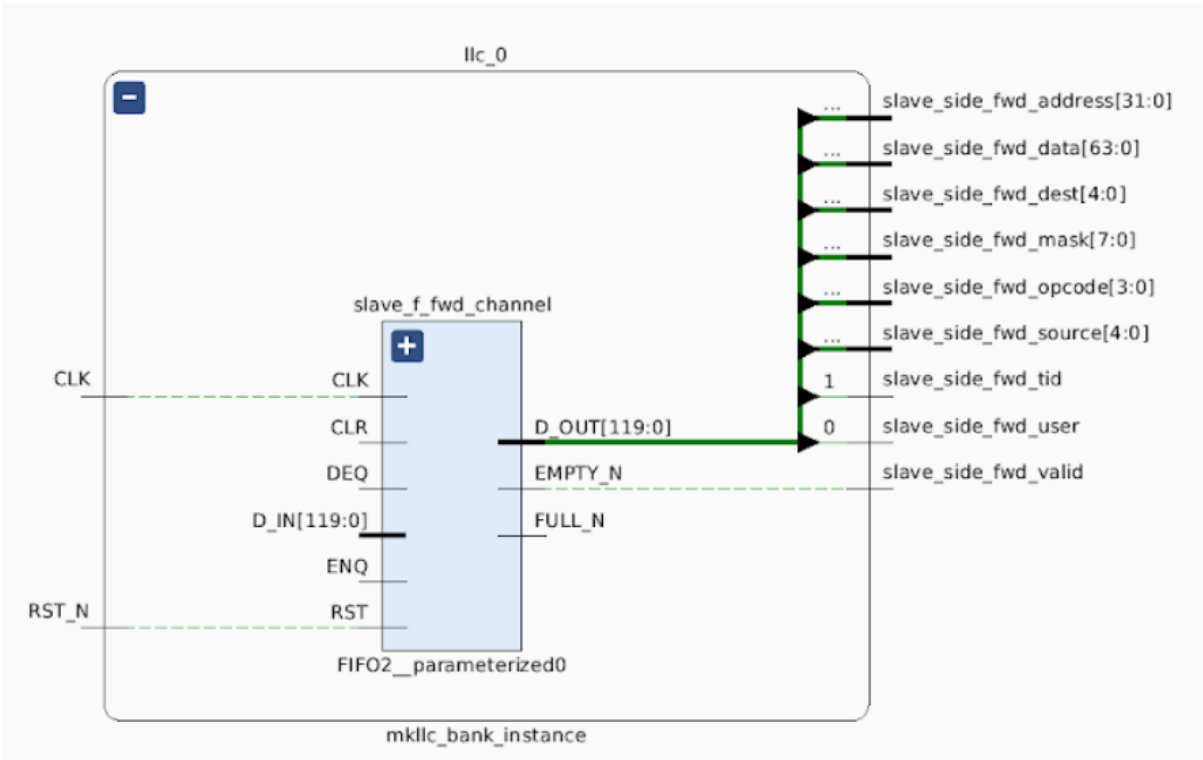

Figure 4.4: Slave forward channel interface of LLC

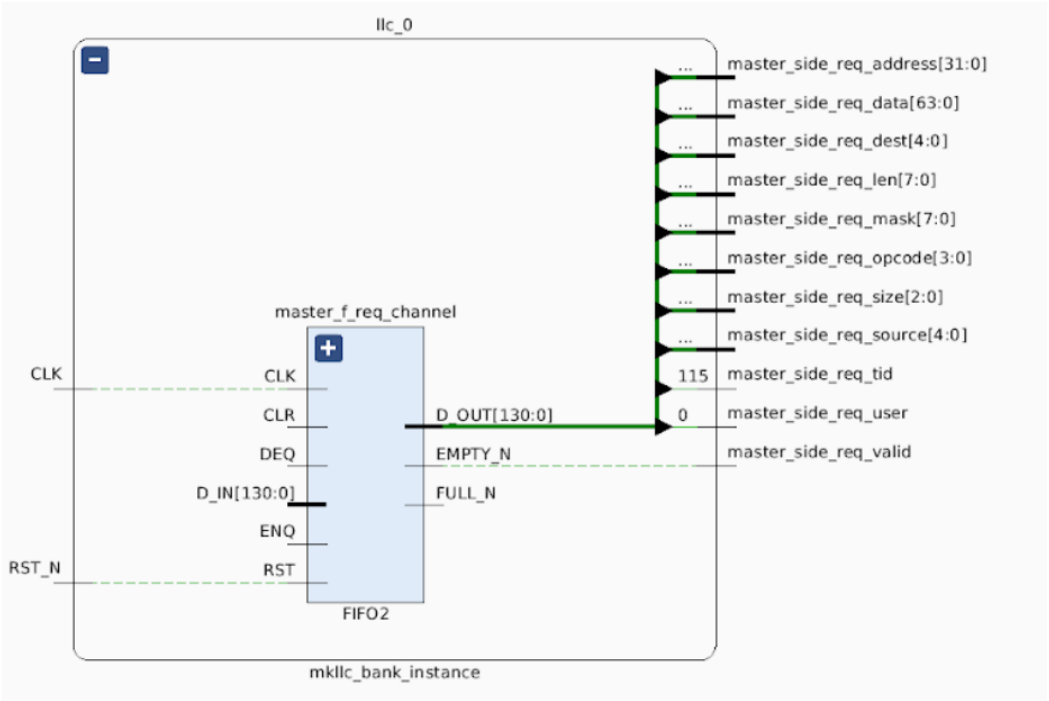

Figure 4.5: Master request channel interface of LLC

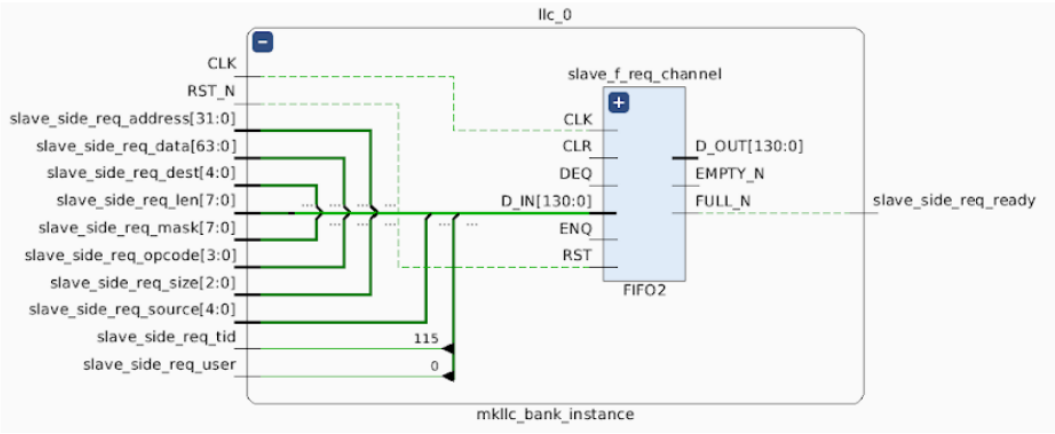

Figure 4.6: Slave request channel interface of LLC

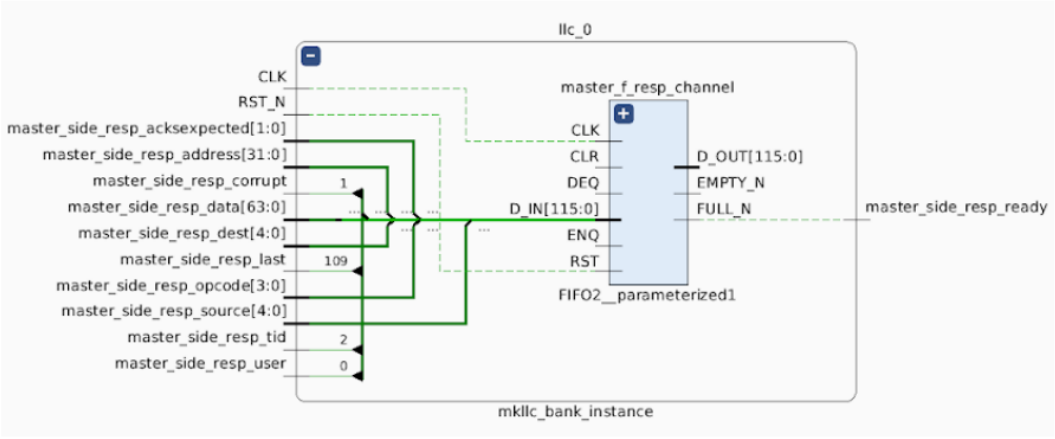

Figure 4.7: Master response channel interface of LLC

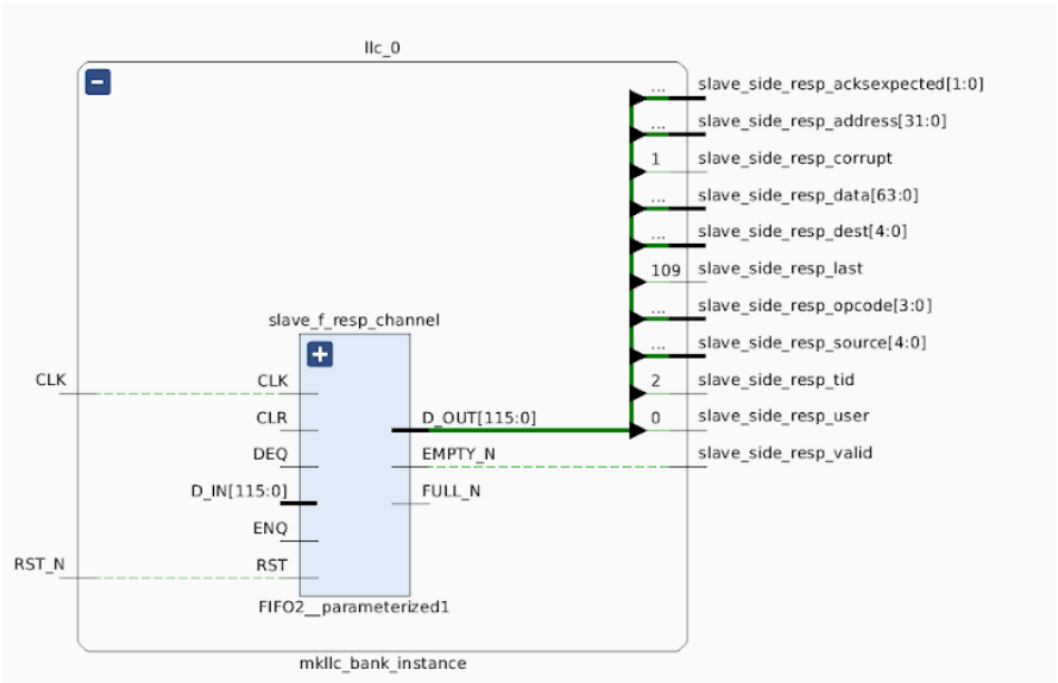

Figure 4.8: Slave response channel interface of LLC

CONCLUSION AND FUTURE WORK

Cache modules were studied, simulated using test bench files and other simulation sources, integrated within System on Chip considering the Network on Chip. This needs to be implemented and tested on Virtex Ultra scale VCU118 FPGA platform. Due to restrictions imposed by covid and inaccessibility of hardware, study was restricted to simulations. SoC level behavioral simulation and post synthesis simulations were done with defined test benches in BSV.

Future work may include setting up post implementation simulation and getting the SafeRV working on FPGA by generating bit stream file and programming it and doing further optimizations as required.

REFERENCES

1. Saferv repository: https://gitlab.com/shaktiproject/saferv/-/tree/master/multi-core
2. Shakti C-Class core: https://c-class.readthedocs.io/en/latest/index.html
3. Cache Design: https://chromite.readthedocs.io/en/latest/cache.html.
4. Hyoukjun Kwon, Tushar Krishna OpenSMART: Single-Cycle Multi-hop NoC generator in BSV and Chisel
5. ProtoGen: https://github.com/icsa-caps/ProtoGen