

# **Cache Coherence Implementation on Ring Bus**

*A Project Report*

*submitted by*

**KARILLI SATISH KUMAR**

*in partial fulfilment of the requirements  
for the award of the degree of*

**MASTER OF TECHNOLOGY**

*Under the guidance of*

**Prof V. Kamakoti**



**DEPARTMENT OF ELECTRICAL ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY MADRAS**

**JUNE 2016**

# THESIS CERTIFICATE

This is to certify that the thesis titled **Cache Coherence Implementation on Ring Bus**, submitted by **Karilli Satish Kumar**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master Of Technology**, is a bona fide record of the research 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**

Dr V. Kamakoti

Professor

Dept. of Computer Science

IIT-Madras, 600 036

Place: Chennai

Date: 18th June 2016

## **ACKNOWLEDGEMENTS**

I would like to express my sincere gratitude to my guide, Dr. V.Kamakoti for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to Mr. G.S. Madhusudan for his encouragement and motivation through out the project. His valuable suggestions and constructive feedback were very helpful in moving ahead with my project work.

I would like to thank my co-guide Dr.Nitin Chandrachoodan and faculty advisor Dr.Shreepad Karmalkar. who have patiently listened, evaluated, and guided us through out the program.

My special thanks to my fellow labmates Rahul,Neel Gala,sirisha,mohmodh, Arjun,abinay, venkata krishna,phani,Arnab for their help and support.

# ABSTRACT

**KEYWORDS:** Cache coherence, Ring interconnect, MOESI protocol

It is desirable to pack in as many cores per chip as possible resulting in increased performance per unit area. An interconnect is required for inter-core and off-chip transport of user packets and also the coherency data. The conventional bus interconnect although ordered and excellent for cache coherence traffic using snoopy protocol it doesn't support higher bandwidths as core count increases. Thus here we are implementing a ring interconnect with TLM interfaces by parameterise the no of cores in the ring interconnect.

We have used L1 cache as a private cache to all the cores in a ring interconnect to reduce the average memory latency and memory traffic but private caches lead to the possibility of cache incoherence problem. Over the years distinct kind of protocols like MSI, MESI, MOESI etc. were proposed to solve this problem. Here we have implemented the MOESI cache coherence protocol in all private L1 caches in a ring interconnect. Thus we maintained the cache coherence among all the cores in a ring interconnect. The code for the entire project is written in a HDL namely Bluespec System Verilog (BSV).

# Contents

<b>ACKNOWLEDGEMENTS</b>	<b>i</b>
<b>ABSTRACT</b>	<b>ii</b>
<b>LIST OF TABLES</b>	<b>vi</b>
<b>LIST OF FIGURES</b>	<b>viii</b>
<b>ABBREVIATIONS</b>	<b>ix</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Overall Microcontroller Architecture . . . . .	1
<b>2 INTERCONNECT TOPOLOGIES</b>	<b>2</b>
2.1 Ring Interconnect . . . . .	2
2.2 Mesh Interconnect . . . . .	3
2.3 Ring-Mesh Hybrid Interconnect . . . . .	5
2.4 Requirements to be met for Cache Coherence . . . . .	6
<b>3 CACHE COHERENCE</b>	<b>8</b>
3.1 Modified: . . . . .	8
3.2 Owned: . . . . .	8
3.3 Exclusive: . . . . .	8
3.4 Shared: . . . . .	9
3.5 Invalid: . . . . .	10
3.6 Cache controller . . . . .	10
<b>4 BLUESPEC SYSTEM VERILOG</b>	<b>11</b>
4.1 Key Features of BSV . . . . .	11
4.2 Study of the Bluespec System Verilog build process . . . . .	11
4.3 Bluespec SystemVerilog Constructs . . . . .	12

4.3.1	Rules . . . . .	12
4.3.2	Modules . . . . .	12
4.3.3	Interfaces . . . . .	13
4.3.4	Methods . . . . .	13
4.3.5	Functions . . . . .	13
4.4	Application Areas of Bluespec System Verilog . . . . .	13
4.5	Building a design in Bluespec System Verilog . . . . .	13
<b>5</b>	<b>TLM INTERFACE</b>	<b>14</b>
5.1	BSV TLM interfaces . . . . .	14
5.1.1	Data structures . . . . .	14
5.1.2	TLM Request . . . . .	15
5.1.3	TLM Response . . . . .	15
5.1.4	Interfaces . . . . .	16
5.1.5	TLM advantages . . . . .	16
<b>6</b>	<b>DESIGN AND IMPLEMENTATION</b>	<b>17</b>
6.1	Interfaces . . . . .	19
6.1.1	TLMRecvIFC . . . . .	20
6.1.2	TLMSendIFC . . . . .	21
6.2	Fields in TLM packet . . . . .	21
6.2.1	Address . . . . .	22
6.2.2	Data . . . . .	22
6.2.3	Command . . . . .	22
6.2.4	Custom . . . . .	22
6.2.5	Lock . . . . .	22
6.3	Working of Fifos . . . . .	23
6.3.1	NODE_IN_REQ FIFO . . . . .	23
6.3.2	NODE_OUT_REQ FIFO . . . . .	24
6.3.3	CAHE_REQ FIFO . . . . .	24
6.3.4	REQ2_CAHCE FIFO . . . . .	24
6.4	Operation Of Coherent Ring . . . . .	24
6.4.1	Rule internal . . . . .	25

6.4.2	Rule internal_from_cache . . . . .	26
6.4.3	Rule external . . . . .	26
6.5	Priority Logic . . . . .	26
6.5.1	Priority and Routing . . . . .	28
<b>7</b>	<b>SIMULATIONS RESULTS AND SYNTHESIS REPORT</b>	<b>30</b>
7.1	Hardware Design Flow . . . . .	30
7.2	Simulation . . . . .	31
7.2.1	Test Case:1 . . . . .	32
7.2.2	Test Case:2 . . . . .	35
7.3	Synthesis . . . . .	41
7.3.1	Device utilization summary . . . . .	42
7.3.2	Timing Report: . . . . .	43
<b>8</b>	<b>CONCLUSION AND FUTURE WORK</b>	<b>44</b>
8.1	Conclusion . . . . .	44
8.2	Future work . . . . .	44

## List of Tables

3.1	MOESI Table . . . . .	10
-----	-----------------------	----



## List of Figures

2.1	ring interconnect . . . . .	2
2.2	ring interconnect routing . . . . .	3
2.3	mesh interconnect . . . . .	4
2.4	mesh structure . . . . .	4
2.5	mesh routing . . . . .	5
2.6	Ring-Mesh Hybrid Interconnect . . . . .	5
2.7	graphical visualization of Ring_Mesh_hybrid Interconnect . . . . .	6
3.1	Modify,Owned and Exclusive . . . . .	9
3.2	Shared And Invalid . . . . .	9
4.1	BSV Flow . . . . .	12
5.1	TLM Interfaces . . . . .	14
6.1	ring interconnect daiagram . . . . .	17
6.2	ring with 8 nodes . . . . .	18
6.3	bottom up approach . . . . .	19
6.4	top down approach . . . . .	19
6.5	code snippet of interfaces . . . . .	20
6.6	single node with interfaces and fifos . . . . .	20
6.7	FIFO VIEW OF SINGLE NODE . . . . .	23
6.8	WORKING OF RULES . . . . .	25
6.9	priority ordering logic table . . . . .	27
6.10	priority of nodes . . . . .	27
6.11	ROUTING OF PACKETS WITH PRIORITY ORDER . . . . .	28
6.12	rule for deq . . . . .	29
7.1	Flow of synthesis . . . . .	31
7.2	BSV SIMULATION RESULTS of 1ST and 2ND CLOCK CYCLES	33
7.3	BSV SIMULATION RESULTS of 3RD CLOCK CYCLE . . . . .	34

7.4	BSV SIMULATION RESULTS OF 1ST AND 2ND CLOCK CYCLES	36
7.5	BSV SIMULATION RESULTS OF 3RD AND 4TH CLOCK CYCLES	37
7.6	BSV SIMULATION RESULTS OF 5TH AND 6TH CLOCK CYCLES	38
7.7	BSV SIMULATION RESULTS OF 7TH AND 8TH CLOCK CYCLES	39
7.8	BSV SIMULATION RESULTS OF 9TH AND 10TH CLOCK CY- CLES . . . . .	40
7.9	Device utilization summary . . . . .	42
7.10	timing report . . . . .	43

## **ABBREVIATIONS**

<b>IITM</b>	Indian Institute of Technology, Madras
<b>RISE</b>	Reconfigurable and Intelligent Systems Engineering
<b>BSV</b>	Bluespec System Verilog
<b>HDL</b>	Hardware Description Language
<b>FIFO</b>	First In First Out
<b>RISC</b>	Reduced Instruction Set Computer
<b>RTL</b>	Register Transfer Language
<b>CMP</b>	Chip-Multiprocessors
<b>TLM</b>	Transcaction level modeling

# Chapter 1

## INTRODUCTION

### 1.1 Overall Microcontroller Architecture

The processor design team of Reconfigurable and Intelligent Systems Engineering[RISE] lab in the computer science department of IIT-Madras has been actively involved in building few processors for academic purposes and other applications. The processor strictly follows the RISC-V instruction set architecture[ISA]. Entire design of the processor is done using a Hardware Description Language[HDL] named Bluespec System Verilog[BSV]. The I-Class processor is a 32-bit in-order variant aimed at 50-250MHz microcontroller variants which have an optional memory protection and the design consumes very low power. The integration forms the basis for synchronizing the core to different peripherals in the microcontroller, which have various operating frequencies. My project work involves maintainance of cache coherence among all private L1 caches in a ring interconnect network. Here we have implemented a ring interconnect with TLM interfaces by parameterising the no of cores in a ring interconnect. And implemented MOESI protocol in all private L1 caches in a ring interconnect for cache coherence. Thus we have maintained cache coherence among all the cores in a ring interconnect.

## Chapter 2

### INTERCONNECT TOPOLOGIES

The components of an interconnect have been briefly discussed here. Topology is probably a design choice that has deep impact on the interconnect performance. A topology primarily decides the minimum number of hops that a packet makes from source to destination. Also since the number of hops require storing and forwarding of packets, the power consumption depends directly on the number of hops. A metric for determining the relative merit of the topologies is firstly the number of physical links between the two nodes and secondly the complexity to physically route the wires of the interconnect.

Three basic interconnect topologies are

- Ring Interconnect
- Mesh Interconnect
- Ring-Mesh hybrid interconnect or Torus

#### 2.1 Ring Interconnect

In Ring interconnect all the cores or nodes are connected in a ring fashion, where packets can move in two directions to reach destination node, we have chosen the ring interconnect.

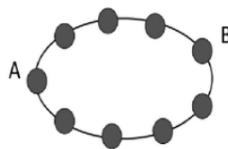


Figure 2.1: ring interconnect

The rings main function is to facilitate the transfer of packets as the case may be between the nodes or between caches and nodes. The top level of the ring is presented in figure 2.1. Note that by convention we use East and West to indicate the directions.

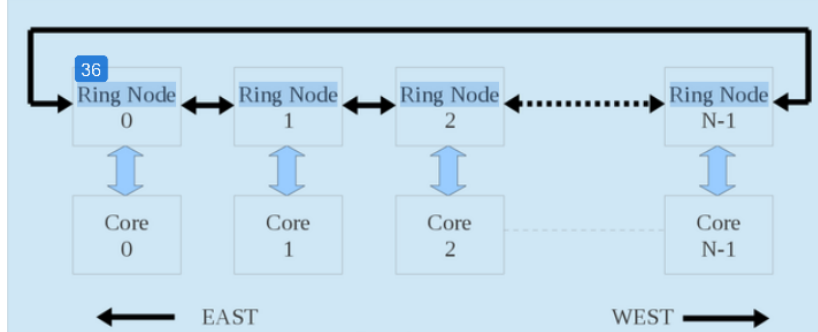


Figure 2.2: ring interconnect routing

To implement the above two functionalities discussed, multiple cores as they are referred to in the code are instantiated. Rules are written to fetch packets from east facing ports of the ring and push them into west facing ports of ring as shown in the Figure 2.2.

In the ring module the packets which are coming from the east or from the cache block are first buffered at that node and in the next clock cycle depending on the logic of ring module one of the packet either from cache or node will traverse through the ring bus. That wraps the discussion on the modules of the ring interconnect. Next we see the mesh interconnect.

## 2.2 Mesh Interconnect

In Mesh interconnect all the cores or nodes are connected in a mesh fashion, where packets can move in all four directions to reach its destination node. The nodes which are at the corners can move only two directions and the nodes which are at the edges of mesh can move only in three directions. The Figure 2.3 shows the mesh network.

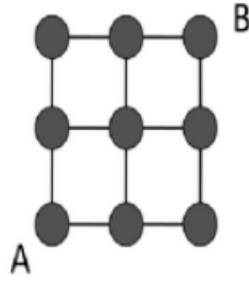


Figure 2.3: mesh interconnect

The mesh is observed to be made with the nodes which contain the sub-modules as in the ring interconnect. However, the difference here is the routing logic since the organisation of the nodes is different than in the ring. A major difference here is the prominence of the standard routing algorithm i.e. Dimension Order Routing using the XY routing algorithm. Each node in the interconnect is assigned two co-ordinate values as is required for implementation of a 2D mesh. Figure 2.4 shows the numbering of nodes in the mesh interconnect.

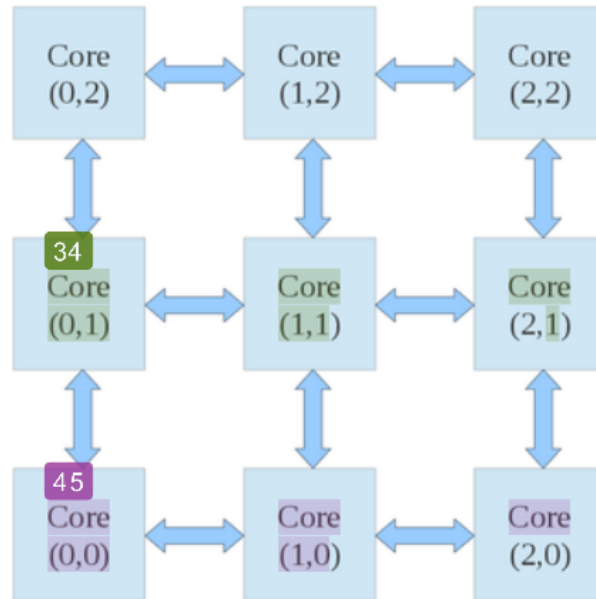


Figure 2.4: mesh structure

This mesh interconnect must facilitate transfer of packets between north and south directions also in addition to the east and west ports. The nodes at periphery of the interconnect should be treated as special cases .since they don't have in-out ports in all four directions. Figure 2.5 shows different paths of routing packets in mesh interconnect.

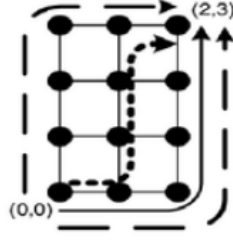


Figure 2.5: mesh routing

As in the case of the ring the parameters number of nodes and the link width are parameterized and therefore adjustable. The number of nodes are specified indirectly by giving the number of nodes at one side of the mesh and the square of this number gives the actual number of nodes in the interconnect.

## 2.3 Ring-Mesh Hybrid Interconnect

The interconnects seen above were the two most basic and widely used interconnects in the commercial implementations. However, the ring interconnect latency grows linearly with the number of tiles and the number of wires becomes extremely large in case of mesh with large number of nodes due to multiple physical links per node. Hence, we look for a different topology that combines the benefits of both the designs. This is a new topology that is a combination of the two. A ring-mesh hybrid connects multiple nodes in a ring and multiple such rings are instantiated. Each of the rings is further connected in a mesh superstructure. Figure 2.6 shows the Ring-Mesh Hybrid Interconnect. The

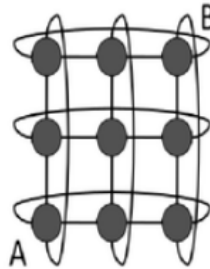


Figure 2.6: Ring-Mesh Hybrid Interconnect

intuition behind such a design is that applications can be assigned multiple cores of a ring for computations. Since the number of tiles on a ring are limited and performance of a ring is proved to be better than other topologies for limited number of cores, the applications' per packet latency is reduced.



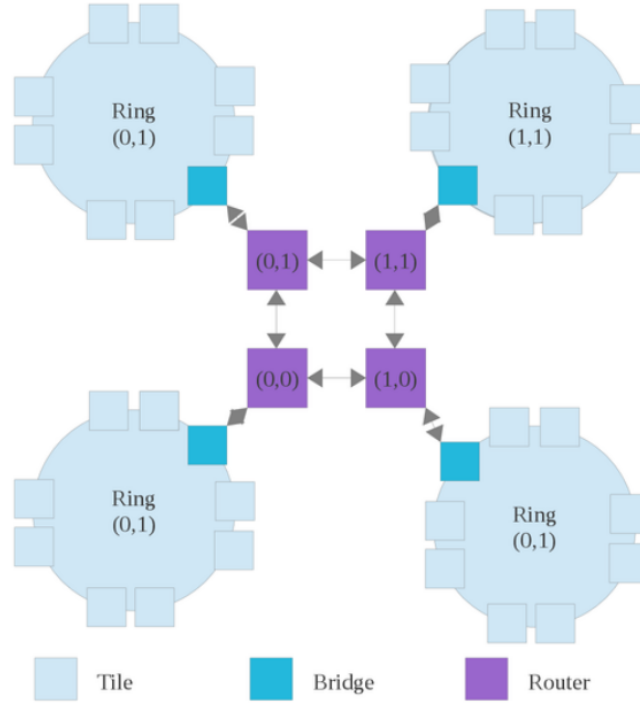


Figure 2.7: graphical visualization of Ring\_Mesh\_hybrid Interconnect

A graphical visualization of such a topology is presented at figure 2.7. The figure shows a network of 32 tiles wherein there are 8 tiles per ring and the collection of 4 such rings are connected in a mesh. Addressing of the tiles is hierarchical. Each 5 ring in the network gets associated with a two coordinate address and within the ring each tile has a unique tile ID which in conjunction with the ring address is used by the routing logic to route the packets. The router is identified with the same two coordinate address as that of the ring.

## 2.4 Requirements to be met for Cache Coherence

The two most widely used coherence protocols are the Snoopy and Directory based protocols. The traffic in both cases may be classified as point-to-point, point-to- multi point or broadcast traffic. In case of a Directory based implementation, the majority of the traffic is only point-to-point. Hence, the traffic on the interconnect is not much. However, there may be a requirement to send out point-to-multi point messages like in case of invalidation of a cache line shared by many.

Snoopy protocol on the other hand relies heavily on broadcast traffic. The request

messages are all of broadcast type however, the response messages are unicast. So broadly the messages in the cache coherent systems may be classified as requests and responses. It is also interesting to note that the requests tend to be short and responses containing the cache lines is generally longer. Thus it makes sense to send them over two physically different links . An important take away from current discussion is that the interconnect should be able to provide support for the coherence protocol selected and further should be designed keeping in view the nature of traffic that is intended to be sent over it and thus arrive at an acceptable level of trade-off between performance and area.

## Chapter 3

### CACHE COHERENCE

In a shared system, caches maintains same data and number of processors perform different kind of operations on cache. So the problem of data inconsistency can happen. Inconsistency can be avoided by cache coherence protocol.

We have a number of cache coherence protocols, namely MSI, MESI, MOESI, MESIF etc. In this work we are implementing MOESI protocol.

#### **3.1 Modified:**

If a cache Block is in Modified state, then concerned processor can change data of cache block and can read data of cache block. Figure 3.1 shows what are other cache block states when cache block is in M,O,E states.

#### **3.2 Owned:**

As an extension of MESI protocol, We restrict write-back of data from cache to main memory. New state “O” is invented to avoid unnecessary write-back of data to main memory during transition from “M” to “S”. Cache block in owned state is not consistent with main memory.

#### **3.3 Exclusive:**

Exclusive state is added to MSI protocol for solving the issue of unnecessary broadcast of invalidation message.

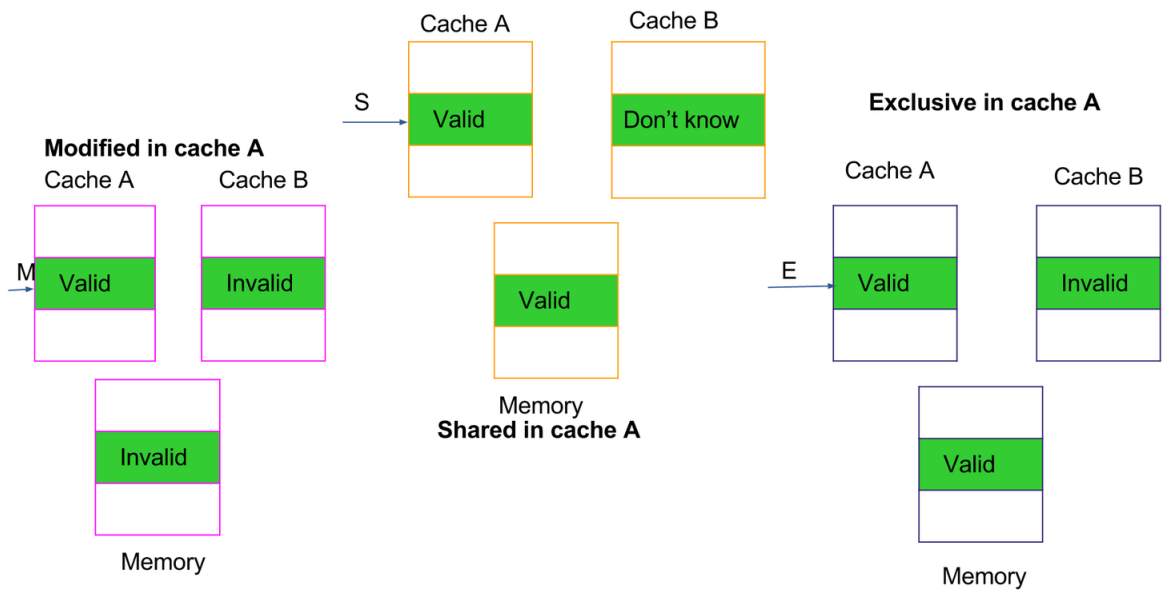


Figure 3.1: Modify, Owned and Exclusive

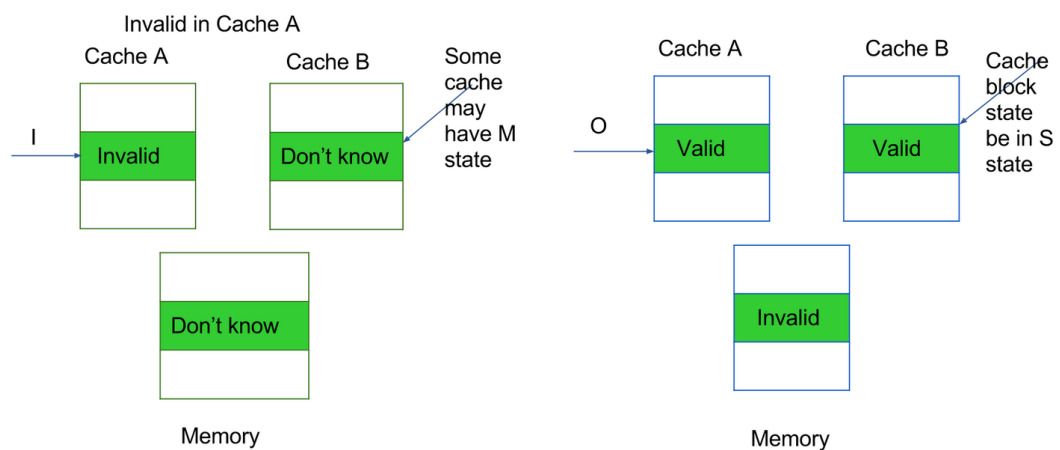


Figure 3.2: Shared And Invalid

### 3.4 Shared:

Cache block is one of several copies in the system. This cache block does not have right to modify the copy. Other remaining processors in the system may have copies of the cache block in the Shared state. In this state processor can read data of cache block it can not perform write operation. If any of other processor is not performing write operation, then block is in shared state. Figure 3.2 shows what are other cache block states when cache block is in S,I states.

	Modified	Own	Exclusive	Shared	Invalid
Modified	False	False	False	False	True
Own	False	False	False	True	True
Exclusive	False	False	False	False	True
Shared	False	True	False	True	True
Invalid	True	True	True	True	True

Table 3.1: MOESI Table

### 3.5 Invalid:

Invalid means either the block is absent or read/write operation is restricted. If a cache block is in invalid state, then processor can not do either read or write operation. Table 3.1 gives details of states of blocks validity.

### 3.6 Cache controller

It performs two more extra operations

- Cache controller has extra three bits along with Tag, data, V/I bits. Three bits represents state of the block (any of M,O,E,S,I). It updates block status for every transaction and if it needs to change of block status of cache forwards intention status of block address and status to snooping mechanism.
- It reads data from other cache requests and verifies whether incoming address is valid or not; it presents it updates block status otherwise leaves it. Figure 3.3 shows the entire structure of communication between processor and cache through TileLink bus.

## Chapter 4

### BLUESPEC SYSTEM VERILOG

BSV is a HDL used in design of electronic systems such as FPGA, ASIC etc. It is a very high level language and results in synthesizable hardware which can run on FPGA emulation platforms. BSV substantially extends the design subset of System Verilog and also increases the programmer's coding efficiency. It has more polymorphism than System Verilog.

#### 4.1 Key Features of BSV

- High level atomic rules in place of Verilog's always block.
- High level interfaces instead of Verilog's port list.
- Powerful Parametrization and Polymorphism.
- Powerful static checking.
- Fully synthesizable at all levels of abstraction.

#### 4.2 Study of the Bluespec System Verilog build process

The following are the steps involved in building a BSV design:

- A developer writes a Bluespec System Verilog program. It may be optionally have Verilog, System Verilog, VHDL and C components.
- The Bluespec System Verilog program is compiled in to Verilog or Bluesim. Then it has two different stages:
  1. pre elaboration - It do parsing and also do type checking.
  2. post elaboration -It does code generation.
- The compilation output is either linked into a simulation environment or processed by a synthesis tool. Once the Verilog or Bluesim implementation is generated, the workstation provides the following tools to help analyse your design:

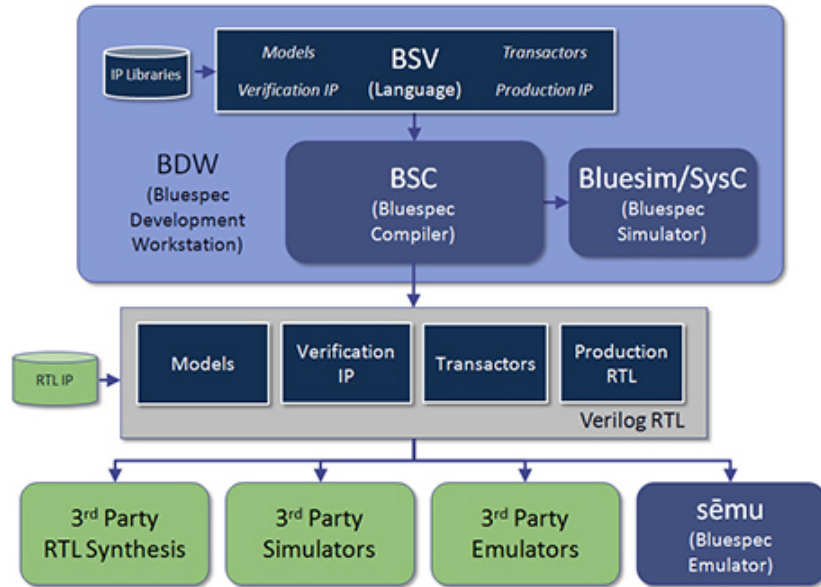


Figure 4.1: BSV Flow

1. Interface with an external waveform viewer with additional Bluespec provided annotations, including structure and type definitions. Figure 4.1 shows Structure of BSV Compiler and BSV synthesis.
2. Schedule Analysis viewer provides multiple perspectives of a modules schedule.
3. Scheduling graphs displaying schedules, conflicts, and dependencies among rules and methods.

## 4.3 Bluespec SystemVerilog Constructs

### 4.3.1 Rules

Rules are used to explain how the data shifts from one state to another state, instead of the Verilog methods of uses always blocks. Every Rule has two components:

- Rule conditions : In rule condition we declare condition like in while in c. if condition satisfied then goes to rule body.
- Rule body : It is a set of actions these explains state transitions.

### 4.3.2 Modules

A module has of three kind of things: state, rules that operate on that state, and an interface that has inputs and outputs of module. A module definition specifies a scheme that can be instantiated multiple times.

### **4.3.3 Interfaces**

Interfaces give a means to group of wires into bundles with mentioned uses, explained by methods. An interface is a tend to remind one of something of a struct, where each member is a method. Interfaces may have other interfaces also.

### **4.3.4 Methods**

Signals and buses are driven in and out of modules using methods. These methods are grouped together into interfaces. There are three kinds of methods:

- Value Methods: It takes zero or more parameters and returns a value.
- Action Methods: It takes zero or more parameters and It performs an action inside of module.
- Action Value Methods: It takes Zero or more parameters, and performs an action, and returns the result.

### **4.3.5 Functions**

Functions are simply parametrized combinational circuits. Function application simply connects a parametrized combinational circuit to actual inputs.

## **4.4 Application Areas of Bluespec System Verilog**

- Modeling for Software development
- Modeling for Architecture Exploration
- Verification
- IP creation

## **4.5 Building a design in Bluespec System Verilog**

- The designer writes the BSV code and it may contain Verilog, Verilog Hardware Description Language and C components.
- The Bluespec System Verilog code is compiled into either Verilog or a Bluesim. This step has 2 stages:
  1. Pre elaboration does parsing and it also does type checking.
  2. Post elaboration does code generation.

The compiled output is either linked to a simulation environment or processed by synthesis tool.



## Chapter 5

### TLM INTERFACE

Transaction Level modeling (TLM) is used to implement a digital systems where communications happens between one to another module. Communication of modules have FIFOs are as channels or buses.

- Every TLM has transaction requests and responses to communicate with other modules.
- TLM interface has
  1. SEND INTERFACE
  2. RECEIVE INTERFACE
- Every send and receive interfaces have request and responses.

## 5.1 BSV TLM interfaces

### 5.1.1 Data structures

IN BSV TLM package has two data structures.

- TLMRequest
- TLMResponses

Figure 5.1 shows Interfaces and data structures of TLM.

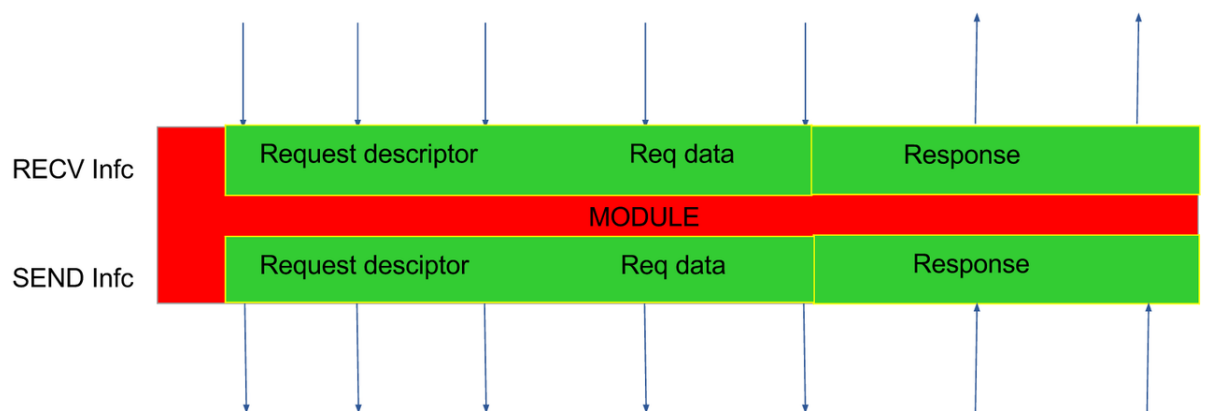


Figure 5.1: TLM Interfaces

### 5.1.2 TLM Request

Each TLMRequest has two control signals and data. TLMRequest has tagged with Request descriptor or Request data.

- Request descriptor has control signal information, these control signals are declared in TLM packages by default and those are
  1. command
  2. mode
  3. addr
  4. data
  5. burst length
  6. byte enable
  7. burst mode
  8. burst size
  9. prty
  10. lock
  11. thread id
  12. transaction id
  13. export id
  14. custom
- Request data has data signals these are
  1. erase
  2. data
  3. transaction-id
  4. custom

### 5.1.3 TLM Response

TLMResponse has valid values of members those are

1. command
2. data
3. status
4. prty
5. thread id
6. transaction id
7. export id
8. custom

### 5.1.4 Interfaces

Interfaces of TLM define how TLM blocks are interconnected and how they communicated. The TLM package includes two basic interfaces:

- TLMSendIFC interface
- TLMRecvIFC interface

These two interfaces use Get and Put sub interfaces as requests and responses.

- TLMRecvIFC interface receives (Put) requests and generates (Get) responses.
- TLMSendIFC interface generates (Get) requests and receives (Put) responses.

### 5.1.5 TLM advantages

TLM interfaces has following advantages:

1. TLM model is accuracy.
2. By using TLM interfaces we can connect easily to other modules.
3. Most of the design errors can be detected during the TLM verification phase.
4. TLM code is more compact and readable than its RTL (VHDL or Verilog or Bluespec) equivalent.

## Chapter 6

### DESIGN AND IMPLEMENTATION

Here we will discuss how we have implemented the coherent ring interconnect to achieve cache coherence. In this coherent Ring interconnect design all the cores or nodes are connected in a ring fashion, where packets can move in a ring in clock wise direction to reach any other node in a ring interconnect.

The routing of Basic ring interconnect will be shown in the Figure 6.1

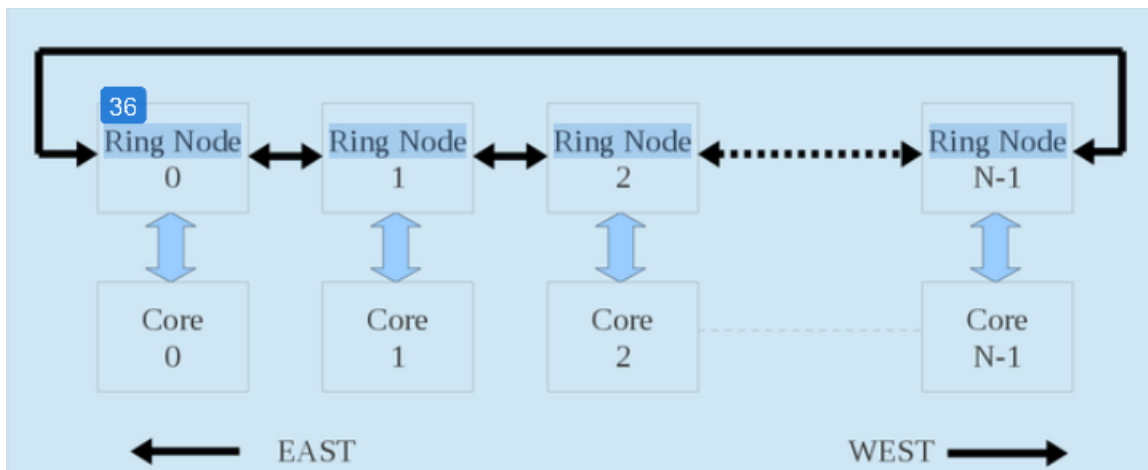


Figure 6.1: ring interconnect daiagram

We identify that the interconnect itself is the topmost module of our design. The main components of the ring module are the nodes and the caches that connect them. We observe that the rings main function is to facilitate the transfer of packets . It may be between the nodes or between the nodes and caches. The top level of the ring is presented shown in the Figure 6.2. Note that by convention we use East and West to indicate the directions.

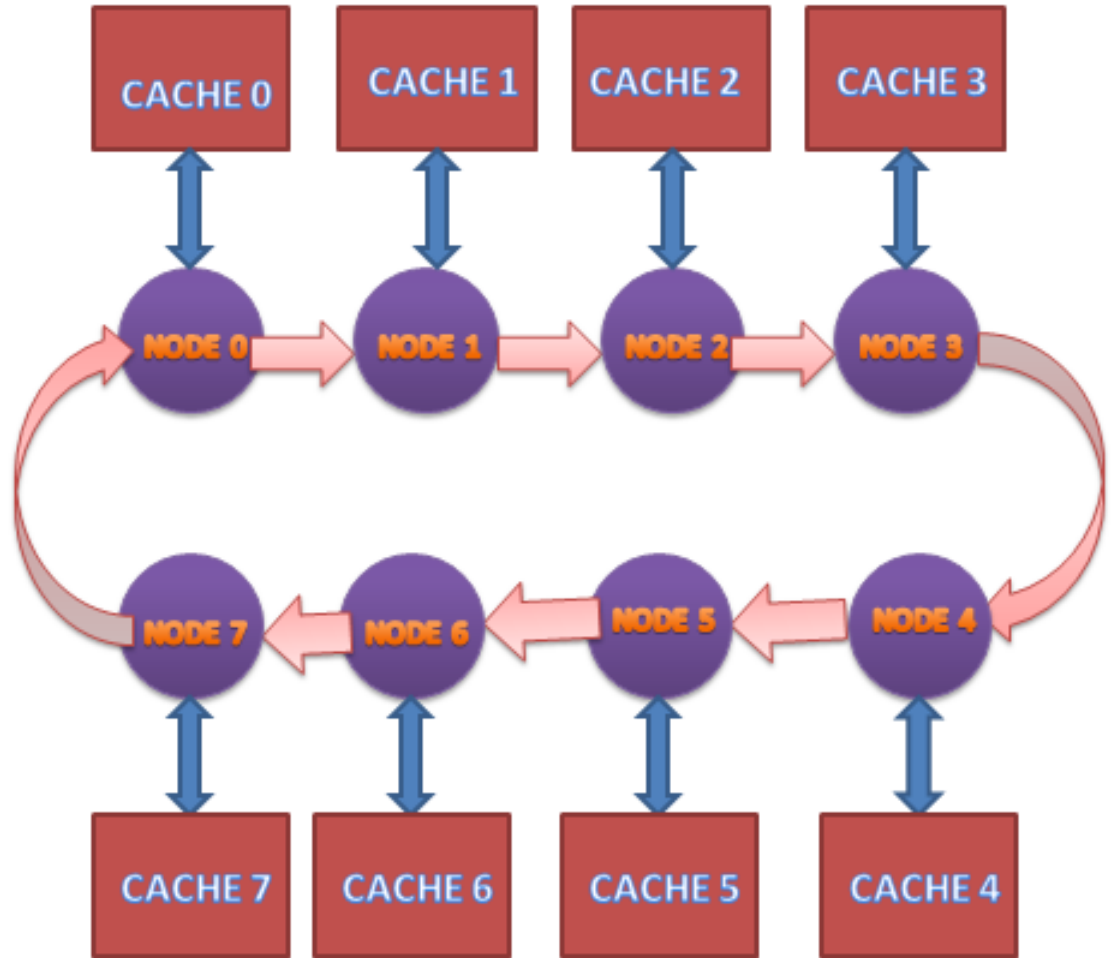


Figure 6.2: ring with 8 nodes

If we go through this coherent ring interconnect design, any Implementation of the design is carried out in either top down approach or a bottom up approach. In top down approach, the highest module in the hierarchy is implemented first and functionality of the lower level modules is assumed whereas in the bottom up approach, the lower level modules are first created and the internal working of each lower level is well known prior to dealing with the higher level modules. Figures 6.3 and Figure 6.4 gives an idea of these two approaches. The selection of an approach is left on the coder and is made as per the level of comfort one has with the concept. We have implemented the design in a top down approach.

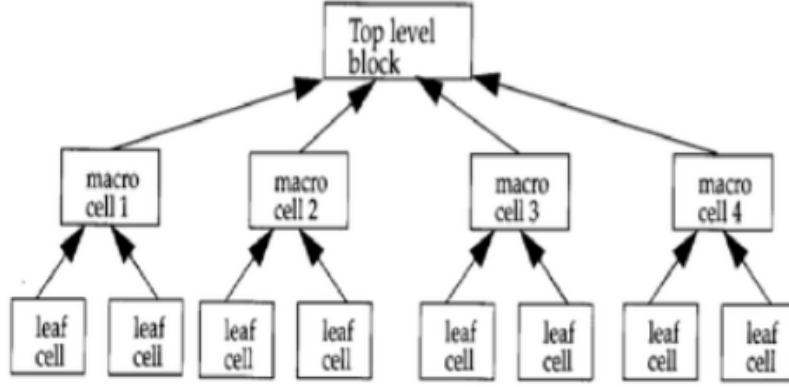


Figure 6.3: bottom up approach

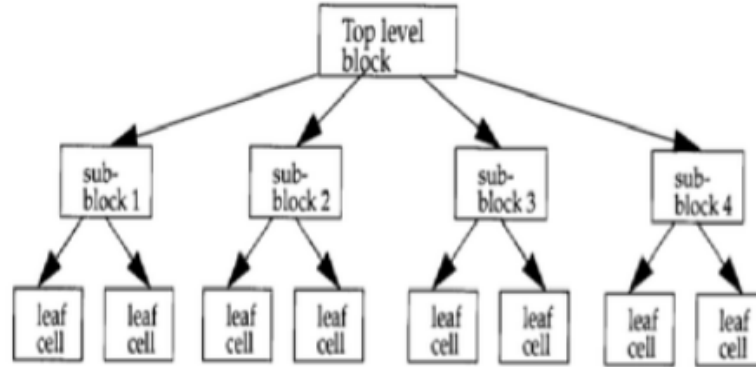


Figure 6.4: top down approach

We are following top down approach in our project. Since Bluespec System Verilog allows behavioural modeling of the design, we describe the working of the ring using rules and interfaces. Put simply, the rules are the part of the code that are fired or executed at every clock cycle in parallel to other rules if all their implicit and explicit conditions are met. The interfaces define the input and output signals of the module.

## 6.1 Interfaces

Here we have used two basic types of TLM interfaces .

- TLMRecvIFC interface
- TLMSendIFC interface

The TLM interfaces will act as input and output ports for communication with other modules. These interfaces use basic Get and Put sub interfaces as the requests and

responses, The TLMSendIFC interface generates (Get) requests and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests and generates (Get) responses. Interfaces used in our code are shown in the Figure 6.5

```
interface Ring_Ifc;

    interface Vector #(No_of_nodes, TLMSendIFC #(Req_from_node,Rsp_to_node)) node_out_ifc;
    interface Vector #(My_No_of_nodes, TLMRecvIFC #(Req_from_node,Rsp_to_node)) node_in_ifc;
    interface Vector #(1, TLMRecvIFC #(Req_from_node,Rsp_to_node)) node_in_ifc3;
    interface Vector #(No_of_nodes, TLMRecvIFC #(Req_from_node,Rsp_to_node)) cache_ifc;
    interface Vector #(No_of_nodes, TLMSendIFC #(Req_from_node,Rsp_to_node)) req2_cache_ifc;

endinterface
```

Figure 6.5: code snippet of interfaces

we can see how these interfaces connected to the node in the Figure 6.6. It is single node which shows all the interfaces that are connected to one particular node in a ring interconnect. We just replicated this node to design the complete ring interconnect.

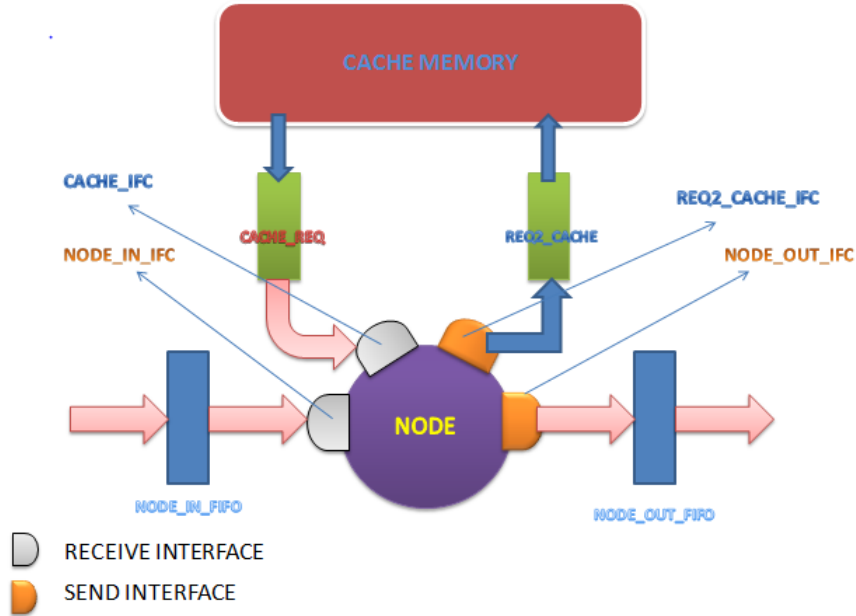


Figure 6.6: single node with interfaces and fifos

### 6.1.1 TLMRecvIFC

The TLMRecvIFC interface receives the requests and transmits the responses.

```

interface Vector #(My_No_of_nodes, TLMRecvIFC #(Req_from_node,Rsp_to_node))
node_in_ifc;

interface Vector (1, TLMRecvIFC (Req_from_node,Rsp_to_node)) node_in_ifc3;

interface Vector #(No_of_nodes, TLMRecvIFC#(Req_from_node,Rsp_to_node))
cache_ifc;

```

node\_in\_ifc is the vector of receive interfaces which receive the requests from the previous nodes connected through the ring. cache\_ifc is the receive interface which receive the requests from the caches connected to the nodes in the ring interconnect.

### 6.1.2 TLMSendIFC

The TLMSendIFC interface transmits the requests and receives the responses.

```

interface Vector #(No_of_nodes, TLMSendIFC #(Req_from_node,Rsp_to_node))
node_out_ifc;

interface Vector #(No_of_nodes, TLMSendIFC #(Req_from_node,Rsp_to_node))
req2_cache_ifc;

```

node\_out\_ifc is the vector of send interfaces which forwards the request from one node to other node in a ring interconnect . req2\_cache\_ifc is the send interface which sends the requests from nodes to caches.

The packets that route through the ring interconnect are TLM packets which comes from cache module. we will send that packet into our ring interconnect.It will route through all the nodes and update all the other caches block status thus coherence maintained.

## 6.2 Fields in TLM packet

- address
- data
- command
- custom
- lock



### **6.2.1 Address**

This field has 32 bit address. It gives the address of the shared memory block requested by the cache .It helps us to identify the block in any cache and update its status.

### **6.2.2 Data**

This field contains 32 bit data. It gives the data of the cache request .It helps us to get the data of that block in any cache

### **6.2.3 Command**

This field tell about type of request the core was received either READ or WRITE.

### **6.2.4 Custom**

We have used this field to send/receive cache block status .i.e we are using MOESI protocol in our project.The no of states required to implement MOESI protocol are 5 .For that we need 3bits to represent the cache block status.So this custom field will be Bit(3) type which will inform about the status of the block.

### **6.2.5 Lock**

This field have Bool as its data type .It is used to know weather the cache has send the request or not.

We used these feilds like this

```
req_from_node.addr = 32'h00000001;
```

```
req_from_node.data = 0;
```

```
req_from_node.command = READ;
```

```
req_from_node.custom = 001;
```

```
req_from_node.lock = True;
```

## 6.3 Working of Fifos

The memory elements used in our design are FIFOs. we have used some special FIFOs like BYPASSFIFOs to resolve the timing issues occurred when feedback is connected. Which can enq and deq at same time (i.e in the same clock cycle). where as in ordinary FIFO we can only enq or deq one at a time. Figure 6.7 will show how the packets route from one fifo to another fifo in our design.

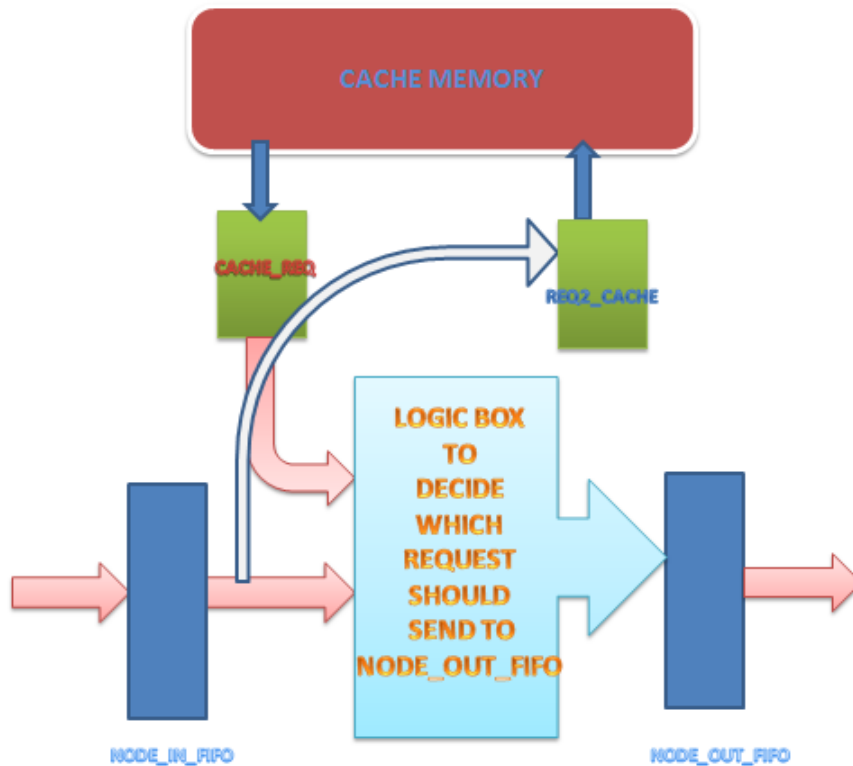


Figure 6.7: FIFO VIEW OF SINGLE NODE

Let us look into the purpose of each FIFO.

- **NODE\_IN\_REQ FIFO**
- **NODE\_OUT\_REQ FIFO**
- **CACHE\_REQ FIFO**
- **REQ2\_CAHE FIFO**

### 6.3.1 NODE\_IN\_REQ FIFO

node\_in\_fifo is vector of fifos. We have used all these fifos as bypassfifos except the initial fifo which is ordinary fifo. The reason for using ordinary fifo is to connect the

feedback from NODE7 to NODE0. Here the node\_in\_fifos vector is used to store the packets coming into the node. These packets will be sent to req2\_cache fifo which will send the updates to cache. These packets will also be sent to bypassfifos depends on the cache requests at that time.

### **6.3.2 NODE\_OUT\_REQ FIFO**

node\_out\_fifo is vector of bypassfifos. We have used all these fifos as bypassfifos. So we can do enq and deq at same time (i.e in the same clock cycle). Here the node\_out\_req fifos vector is used to store the packets that is received from the cache\_req fifo or node\_in\_req fifo depends on which rule fired first. These packets will be sent to node\_in\_fifos of other node. This fifos will facilitate the routing of the packets inside the ring interconnect.

### **6.3.3 CACHE\_REQ FIFO**

cache\_req\_fifo is vector of bypassfifos. We have used all these fifos as bypassfifos. In this fifos we can do enq and deq at the same time (i.e in the same clock cycle). This vector fifos are used to store the packets that is received from the cache. These packets will be sent to node\_out\_req fifos depends on the cache request lock status.

### **6.3.4 REQ2\_CACHE FIFO**

req2\_cache\_fifo is vector of bypassfifos. We have used all these fifos as bypassfifos. In this fifos we can do enq and deq at same time (i.e in the same clock cycle). Here the req2\_cache fifos vector is used to store the packets that is received from the node. These packets will be sent to cache to update the cache memory status.

## **6.4 Operation Of Coherent Ring**

As a part of coherent ring design we have used rules as a part of our code. which will trigger at every clock pulse. we have designed each node operation in with help

of 3rules.which will fire one after the other because of using bypass fifos .Which will allow enq first and then deq.Hence rules will be fired in a particular order.Three rules were used to rout the packets in ring interconnect

- Rule internal:
- rule internal\_from\_cache:
- rule external:

Figure 6.8 explains the working of these three rules

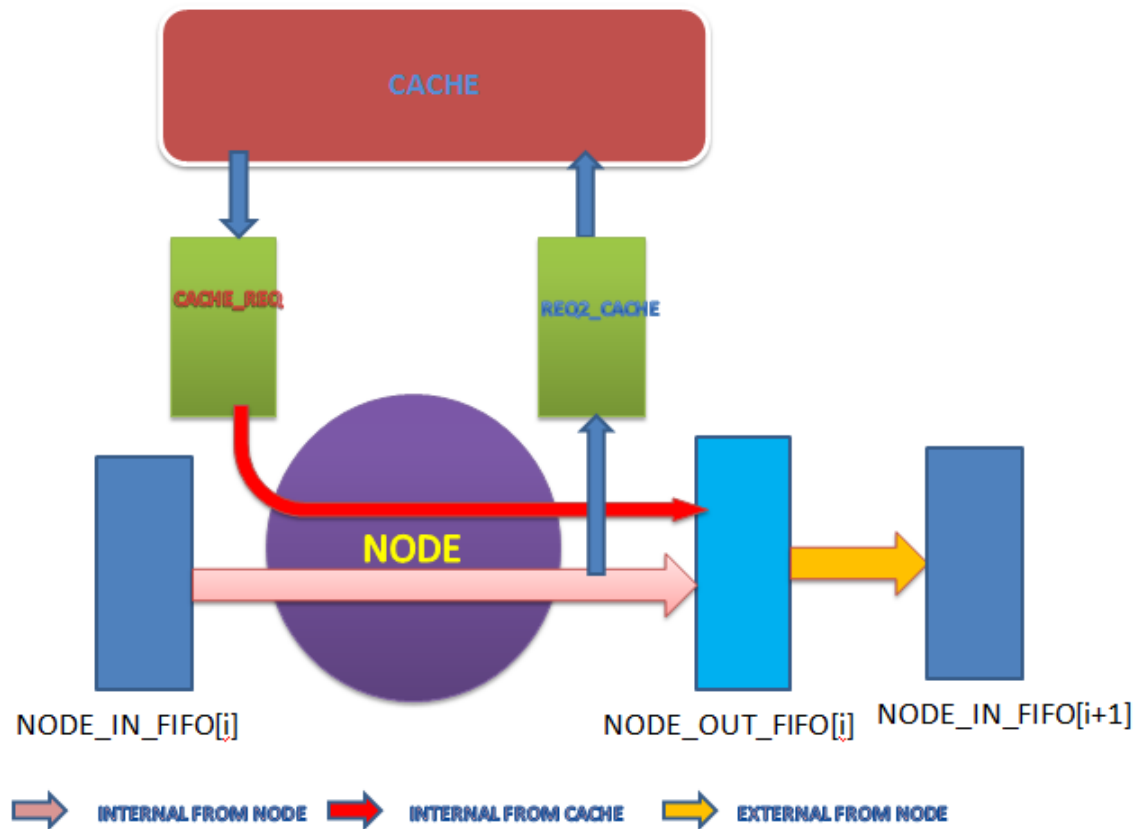


Figure 6.8: WORKING OF RULES

### 6.4.1 Rule internal

The pink arrow shows the transfer of packet from node\_in\_req fifo to node\_out\_req fifo. The request is transferred from node in fifo to node out fifo with in the same node. This rule will fire when there is no cache request .When ever this rule fires a wire named **wr\_lock** will become False. If all the nodes **wr\_locks** becomes False then it indicates that all the requests are serviced. Than cache fifos will enqueue new set of cache requests.

### 6.4.2 Rule internal\_from\_cache

The red arrow shows the transfer of packet from cache\_req fifo to node\_out\_req fifo. This rule will fire when there is a cache request. When ever this rule fires the request packet from cache is transferred from cache fifo to node out fifo of the same node and wr\_lock will become True. This wire wr\_lock is used in priority ordering of ring interconnect.

### 6.4.3 Rule external

The yellow arrow shows the transfer of packet from node\_out\_req fifo to node\_in\_req fifo of the next node. This rule will always fire what ever may be the cache request. This rule helps in forwarding the packet inside the ring. The packet which reached the node\_in\_req fifo of next node will forward through the rule internal at next node and so on.

## 6.5 Priority Logic

Some times we may face situation like more than one core put requests at the same instance of time. Then we need to give priority to one among them. After the completion of one request we will service the request of other core. In our coherent ring we have given a priority ordering in the clock wise order. According to the given priority ordering node0 has given 1st priority and node7 has given the last priority. We have implemented the priority ordering with the help of the following logic which is included in the Figure 6.9.

Let all the cores sends its cache requests at same time then cache0 request will be serviced first and cache7 request will be serviced at the end. Figure 6.9 will explain the cache requests and the operation of our coherent ring. Here True means cache sent the request. False means cache has not sent request. Dont care means what ever may be the cache request either its True or False.

CACHE0	CACHE1	CACHE2	CACHE3	CACHE4	CACHE5	CACHE6	CACHE7	OPERATION
TRUE	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	Request from cache0 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	TRUE	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care	Request from cache1 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	TRUE	Don't care	Don't care	Don't care	Don't care	Don't care	Request from cache2 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	TRUE	Don't care	Don't care	Don't care	Don't care	Request from cache3 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	FALSE	TRUE	Don't care	Don't care	Don't care	Request from cache4 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	Don't care	Don't care	Request from cache5 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	Don't care	Request from cache6 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	Request from cache7 is serviced and new cache requests are not enqueued into cache_req FIFO
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	NO cache Requests are serviced and New cache requests are enqueued into cache_req FIFO

Figure 6.9: priority ordering logic table

So if we receive a request from cache0 independent of the other caches requests always cache0 request will be served with highest priority .Similarly cache1 has more priority independent of other caches requests when cache0 has not sent request and so on.

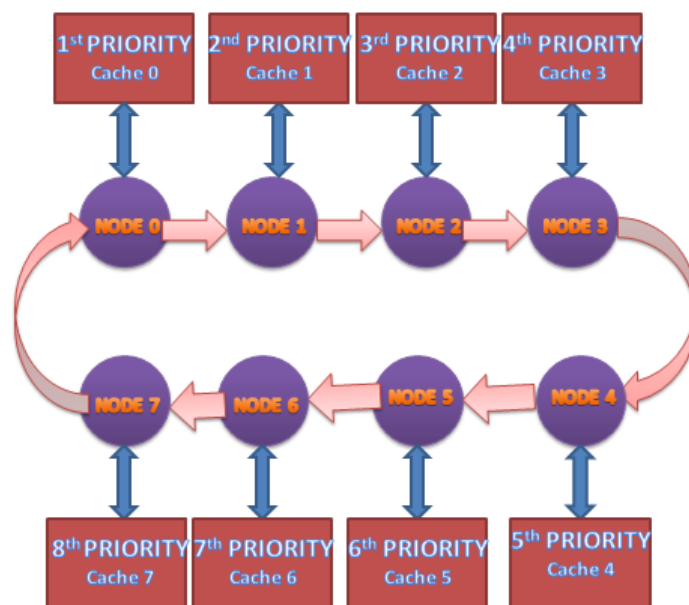


Figure 6.10: priority of nodes

Figure 6.10 shows the priority ordering of different cores in a ring interconnect.

### 6.5.1 Priority and Routing

Now we look into the routing of the packets through the coherent network when more than one core send requests at same time. We already know from the priority ordering cache0 request will be served first if it sends a request. all the other nodes are activated their "rule internal" and deactivated the rule "internal\_from\_cache". so the request packet received by the cache0 will be routed successfully through the coherent ring network and updates all the caches connect to the ring. We can see the routing of packets in the Figure 6.11

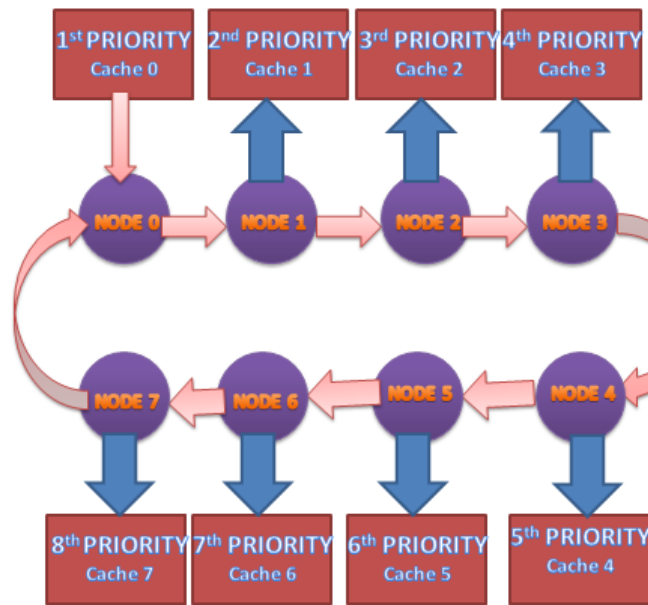


Figure 6.11: ROUTING OF PACKETS WITH PRIORITY ORDER

Figure 6.11 shows that when it services one request all the other nodes are disconnected from their respective caches through receive interfaces. But always connected through send interfaces. Through these send interfaces block status of caches will update.

If all the cache requests are serviced. `wr_lock` of all the nodes will become False then rule "rule\_for\_deq" will be fired. All cache\_req fifos will be dequeued. And new cache requests will enqueue into the cache fifos. Figure 6.12 shows the code snippet for this rule.

```

rule rule_for_deq|( wr_lock0 == False  && wr_lock1 == False  && wr_lock2
False&& wr_lock6 == False&& wr_lock7 == False);
    cache_req[0].deq;
    cache_req[1].deq;
    cache_req[2].deq;
    cache_req[3].deq;
    cache_req[4].deq;
    cache_req[5].deq;
    cache_req[6].deq;
    cache_req[7].deq;
    token0 <=True;
    token1 <=True;
    token2 <= True;
    token3 <=True;
    token4 <=True;
    token5 <=True;
    token6 <= True;
    token7 <=True;
endrule

```

Figure 6.12: rule for deq

Because of the rule shown in the Figure 6.12 ,we can dynamically enqueue the new cache requests into cache fifos when ever the ring becomes idle .That wraps the discussion on the Design and Implementation .



## **Chapter 7**

# **SIMULATIONS RESULTS AND SYNTHESIS REPORT**

### **7.1 Hardware Design Flow**

Coding the design in a high level language is job only half complete. The final realization of the hardware is the ultimate goal of any project. The hardware or the VLSI design flow as depicted in Figure 7.1 gives the major steps taking the design towards physical realization. A short detour explaining this flow is in order at this stage.

The design of any product starts with an idea. The idea is born out of a client requirement. This idea is put down as a higher level behavioural model of the final product using the high level languages like BSV. The behavioural model is then compiled into a RTL using a suitable compiler. RTL are generally the description of the circuit at the module level where input output interfaces, clock and other signals are visible. Any design can be described in RTL using the Huffman's model.

Once the RTL is arrived at, the next step in the design flow is the logic synthesis. Using commercial EDA tools, the designer converts the RTL into a netlist which is nothing but a list of gates and wires whose input output are specified. The EDA tools gives a lot of options like types of gates to be used, constraints for the design with respect to the power, area and timing, thus a highly optimized netlist is achieved after logic synthesis.

On getting the netlist, more EDA tools are used to do place and route of gates and wires or floor planning as it is popularly called. The result of place and route is the mask that could be handed over to the foundry for carrying out the fabrication of the chip. Two most important part of the design flow are the testing and verification. Testing is done to ensure final chip does not suffer from manufacturing defects and verification is done at each stage of the flow to ensure the design meets the requirements as were

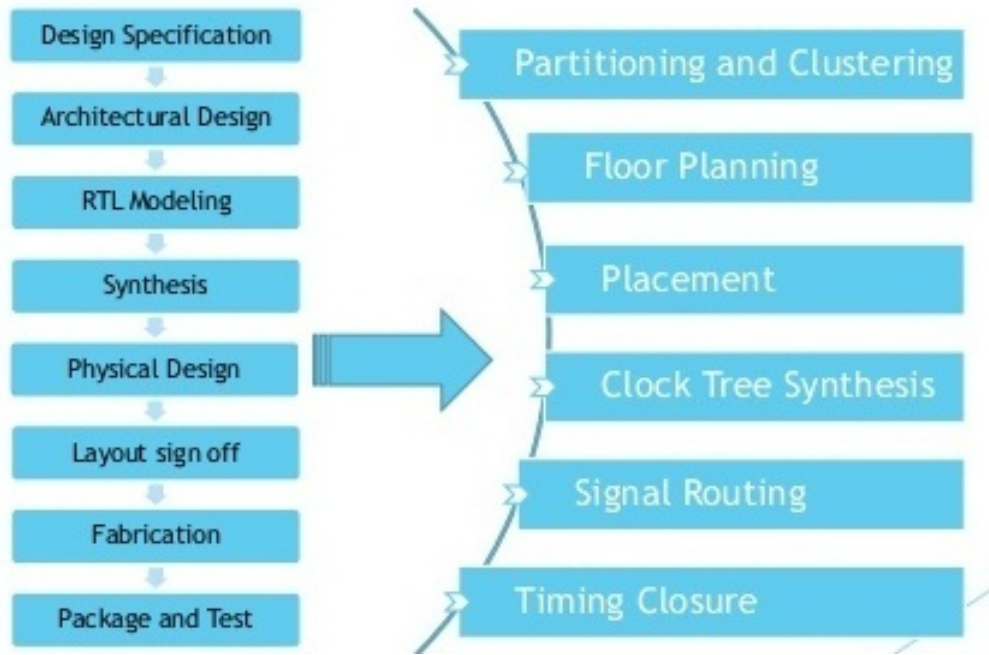


Figure 7.1: Flow of synthesis

originally projected. In our case however, we limit the scope to design, implementation of the design in BSV, simulations and logic synthesis is done to verify performance.

Since the design process has been dealt with in earlier chapters adequately. We already looked at implementation in the BSV. As previously mentioned we have adopted the top down approach for coding the design. Hence, higher level module of ring was first implemented using behavioural modeling and then the behaviour is realized using the lower modules that implements the functionality and are instantiated in the higher modules.

## 7.2 Simulation

On completion of the BSV coding, the project is compiled with BSV compiler which gives options to compile for BSV simulator or to generate verilog files for further processing. In our case we need both. We simulate the design using the Bluesim simulator and observe the number of clock cycles which was shown to be a good indicator of the latency in the network. The results of the simulation are observed on the BSV GUI and recorded for analysis. The Simulation Results for some test cases are shown below

In our project we have represented MOESI protocol states as 3 bit binary numbers.

- MODIFIED :001
- OWNED :010
- EXCLUSIVE:011
- SHARED :100
- INVALID :101

### **7.2.1 Test Case:1**

Let core0 and core1 have a particular shared memory block in both of their private L1 caches. Let Core0 modified the data in that memory block. At the same instance of time core1 want to read the data from that memory block. According to cache coherence core1 should receive the latest updated value of that particular memory block. For this to be happen first the block status of cache0 should changed to “MODIFY” and propagate the block status “INVALID” to all the other caches connected in the coherent ring interconnect. When the packets were routing in the ring interconnect core1 will update its private cache block status to “INVALID”. So that the read request received by cache1 will not service the wrong data(old data) to core1. In the next clock cycle core1 READ request will be serviced with updated data by making its cache block status “SHARED” and sending the block status update “OWNED” through the ring bus. So the cache0 will update its block status to “OWNED” and cache1 which is in "SHARED" mode can get the updated data. Hence data consistency is maintained. We can see the Blue spec simulation results for this test case in the figures 7.2 and 7.3.

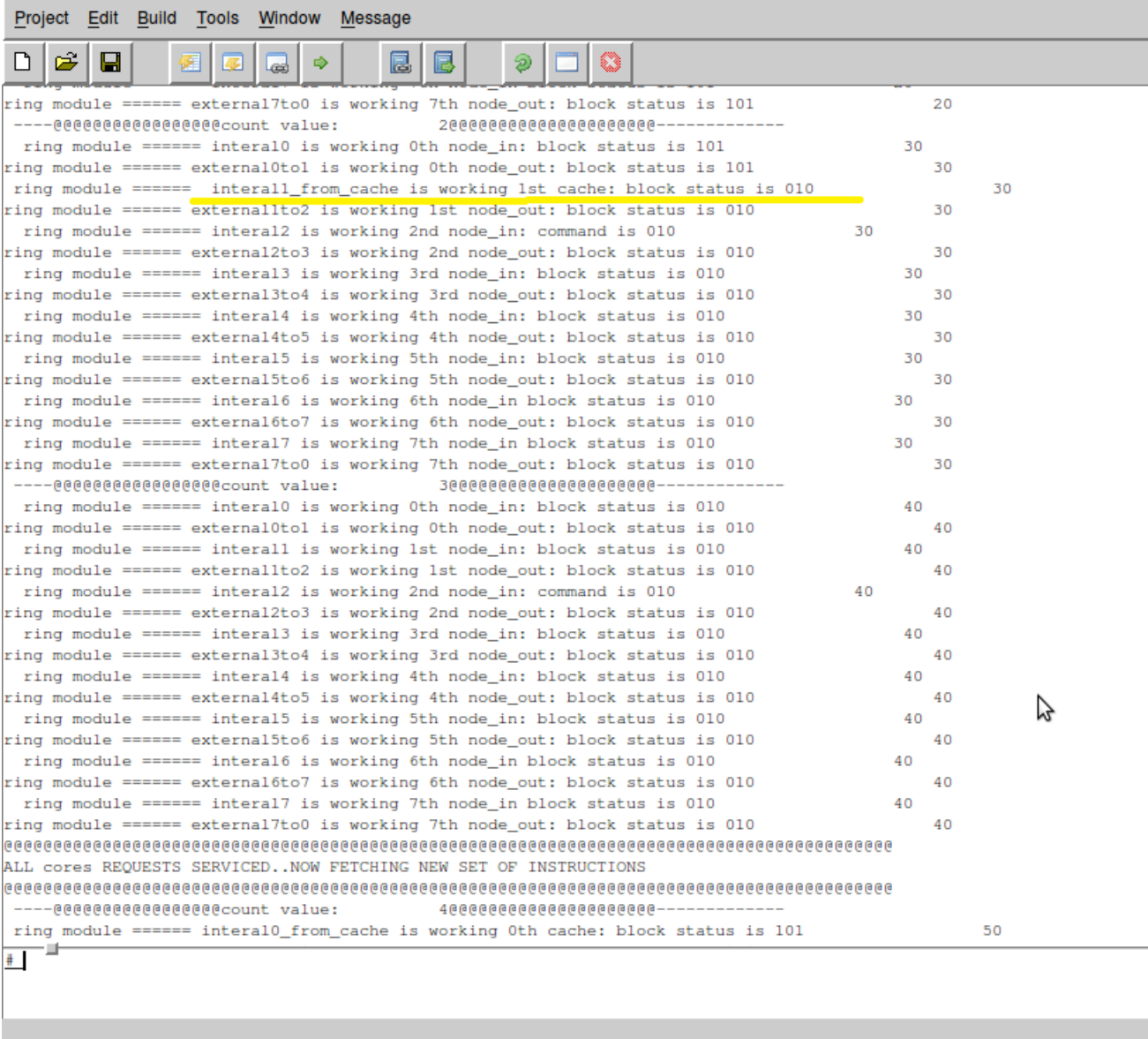
```

Project Edit Build Tools Window Message
Simulation shared library created: out.so
Simulation executable created: ./out
+ ./out
----@@@@@@@@@@@@@@@@@count value:          1@@@@@@@@@@@@@@@@@-----
ring module ===== internal0_from_cache is working 0th cache: block status is 101
ring module ===== external0to1 is working 0th node_out: block status is 101
ring module ===== internal1 is working 1st node_in: block status is 101
ring module ===== external1to2 is working 1st node_out: block status is 101
ring module ===== internal2 is working 2nd node_in: command is 101
ring module ===== external2to3 is working 2nd node_out: block status is 101
ring module ===== internal3 is working 3rd node_in: block status is 101
ring module ===== external3to4 is working 3rd node_out: block status is 101
ring module ===== internal4 is working 4th node_in: block status is 101
ring module ===== external4to5 is working 4th node_out: block status is 101
ring module ===== internal5 is working 5th node_in: block status is 101
ring module ===== external5to6 is working 5th node_out: block status is 101
ring module ===== internal6 is working 6th node_in block status is 101
ring module ===== external6to7 is working 6th node_out: block status is 101
ring module ===== internal7 is working 7th node_in block status is 101
ring module ===== external7to0 is working 7th node_out: block status is 101
----@@@@@@@@@@@@@@@@@count value:          2@@@@@@@@@@@@@@@@@-----
ring module ===== internal0 is working 0th node_in: block status is 101
ring module ===== external0to1 is working 0th node_out: block status is 101
ring module ===== internal1_from_cache is working 1st cache: block status is 010
ring module ===== external1to2 is working 1st node_out: block status is 010
ring module ===== internal2 is working 2nd node_in: command is 010
ring module ===== external2to3 is working 2nd node_out: block status is 010
ring module ===== internal3 is working 3rd node_in: block status is 010
ring module ===== external3to4 is working 3rd node_out: block status is 010
ring module ===== internal4 is working 4th node_in: block status is 010
ring module ===== external4to5 is working 4th node_out: block status is 010
ring module ===== internal5 is working 5th node_in: block status is 010
ring module ===== external5to6 is working 5th node_out: block status is 010
ring module ===== internal6 is working 6th node_in block status is 010
ring module ===== external6to7 is working 6th node_out: block status is 010
ring module ===== internal7 is working 7th node_in block status is 010
ring module ===== external7to0 is working 7th node_out: block status is 010
----@@@@@@@@@@@@@@@@@count value:          3@@@@@@@@@@@@@@@@@-----
#

```

Figure 7.2: BSV SIMULATION RESULTS of 1ST and 2ND CLOCK CYCLES

In the Figure 7.2 in first clock cycle core0 want to write the data into cache0 .So cache0 will change its state to “MODIFIED ” and cache0 will propagate block status “INVALID”(101) to all the other caches. In the Figure 7.2 we can see the propagation of INVALID(101) signal through the ring interconnect. So cache1 which have the same memory block will update its status to "INVALID".



```

Project Edit Build Tools Window Message
ring module ===== external7to0 is working 7th node_out: block status is 101      20
----@count value:      2@count value:-----
ring module ===== internal0 is working 0th node_in: block status is 101      30
ring module ===== external0to1 is working 0th node_out: block status is 101      30
ring module ===== internal1_from_cache is working 1st cache: block status is 010      30
ring module ===== external1to2 is working 1st node_out: block status is 010      30
ring module ===== internal2 is working 2nd node_in: command is 010      30
ring module ===== external2to3 is working 2nd node_out: block status is 010      30
ring module ===== internal3 is working 3rd node_in: block status is 010      30
ring module ===== external3to4 is working 3rd node_out: block status is 010      30
ring module ===== internal4 is working 4th node_in: block status is 010      30
ring module ===== external4to5 is working 4th node_out: block status is 010      30
ring module ===== internal5 is working 5th node_in: block status is 010      30
ring module ===== external5to6 is working 5th node_out: block status is 010      30
ring module ===== internal6 is working 6th node_in: block status is 010      30
ring module ===== external6to7 is working 6th node_out: block status is 010      30
ring module ===== internal7 is working 7th node_in: block status is 010      30
ring module ===== external7to0 is working 7th node_out: block status is 010      30
----@count value:      3@count value:-----
ring module ===== internal0 is working 0th node_in: block status is 010      40
ring module ===== external0to1 is working 0th node_out: block status is 010      40
ring module ===== internal1 is working 1st node_in: block status is 010      40
ring module ===== external1to2 is working 1st node_out: block status is 010      40
ring module ===== internal2 is working 2nd node_in: command is 010      40
ring module ===== external2to3 is working 2nd node_out: block status is 010      40
ring module ===== internal3 is working 3rd node_in: block status is 010      40
ring module ===== external3to4 is working 3rd node_out: block status is 010      40
ring module ===== internal4 is working 4th node_in: block status is 010      40
ring module ===== external4to5 is working 4th node_out: block status is 010      40
ring module ===== internal5 is working 5th node_in: block status is 010      40
ring module ===== external5to6 is working 5th node_out: block status is 010      40
ring module ===== internal6 is working 6th node_in: block status is 010      40
ring module ===== external6to7 is working 6th node_out: block status is 010      40
ring module ===== internal7 is working 7th node_in: block status is 010      40
ring module ===== external7to0 is working 7th node_out: block status is 010      40
#####
ALL cores REQUESTS SERVICED..NOW FETCHING NEW SET OF INSTRUCTIONS
#####
----@count value:      4@count value:-----
ring module ===== internal0_from_cache is working 0th cache: block status is 101      50
#

```

Figure 7.3: BSV SIMULATION RESULTS of 3RD CLOCK CYCLE

Now in the 3rd clock cycle when core1 READ request is serviced it will change its cache block status to “SHARED” mode and propagates “OWNED”(010) block status to all the other caches. In the figure 7.3 we can see the propagation of OWNED(010) signal through the Ring interconnect. Finally the memory block in cache0 will be in “OWNED” state and cache1 will be in “SHARED” state. Both the memory blocks have same updated data which was updated by WRITE request of core0. Once all the requests are serviced caches will enqueue new requests. Hence cache coherency is maintained.

### 7.2.2 Test Case:2

Let us consider the worst case in which all the cores wanted to WRITE into the same memory block at the same instance. All the cache blocks get the write requests from their respective cores. As per our design we have given more priority to core0 and then core1, core2 and so on. So first core0 request will be serviced, i.e memory block in cache0 will be changed to “MODIFIED” state and update its value through WRITE operation. Propagate the block status “INVALID” to all the other caches. In the 2nd clock cycle core1 request will be serviced this time core1 will update the memory block. Finally in the 8th clock cycle expect cache7 all the caches blocks status becomes INVALID. cache7 only has the latest updated value with block status “MODIFIED”. So cache coherency is maintained even in worst case situation. We can see the Blue Spec simulation results of Test Case:2 in the figures 7.4 to 7.8.





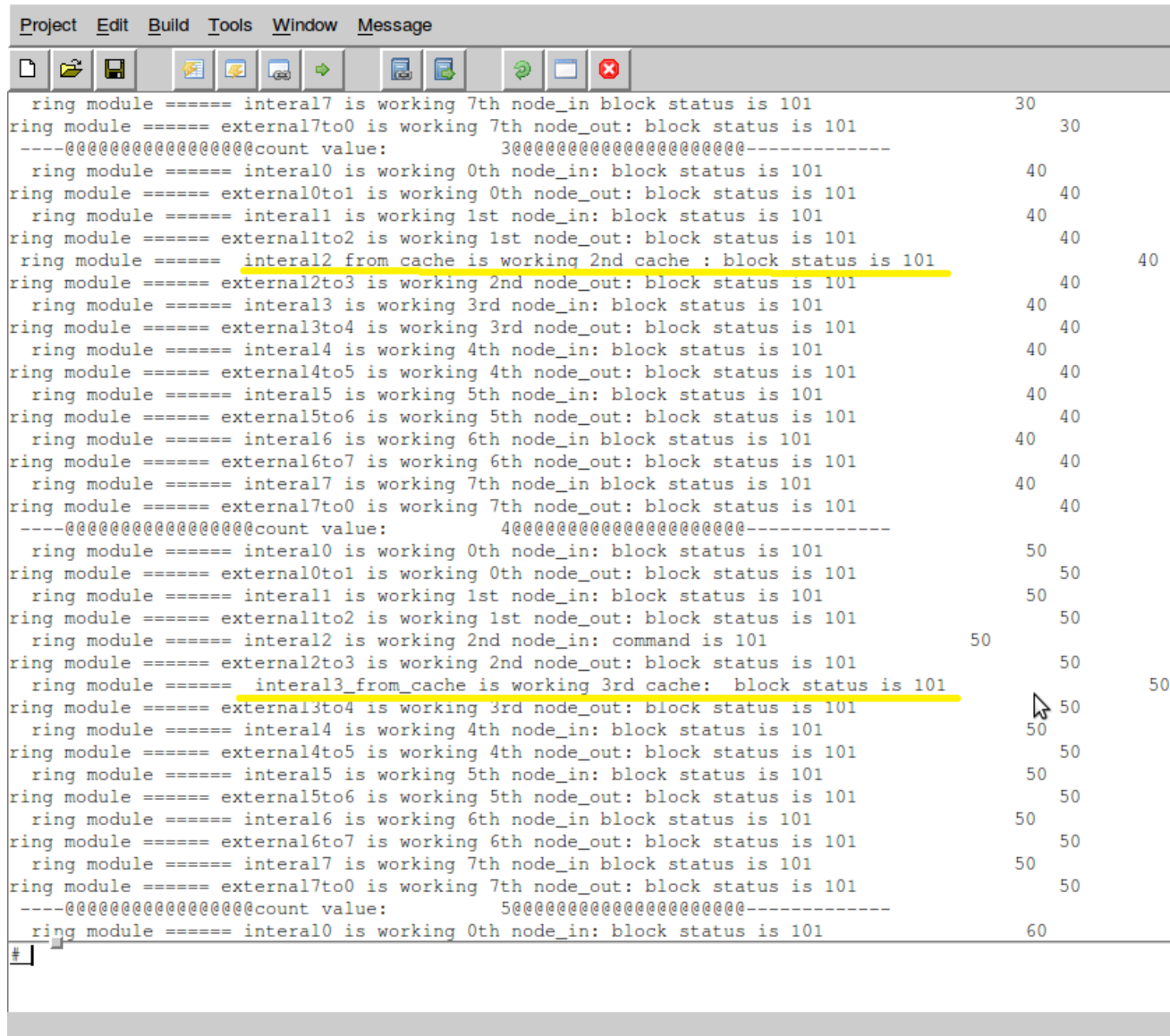


Figure 7.5: BSV SIMULATION RESULTS OF 3RD AND 4TH CLOCK CYCLES

In the figure 7.5 according to priority logic in third clock cycle core2 writes the data into cache2 .So cache0 changes its block status to “MODIFIED ” state and it propagates block status “INVALID”(101) to all other caches. In the figure 7.5 we can see the propagation of INVALID(101) signal through the Ring interconnect. So memory blocks in all the other caches will be in “INVALID” state.

In fourth clock cycle core3 writes the data into cache3 .So cache3 changes its block status to “MODIFIED ” state and it propagates the block status “INVALID”(101) to all other caches. In the figure 7.5 we can see the propagation of INVALIDATE(101) signal through the Ring . So memory blocks in all the other caches will be in “INVALID” state.



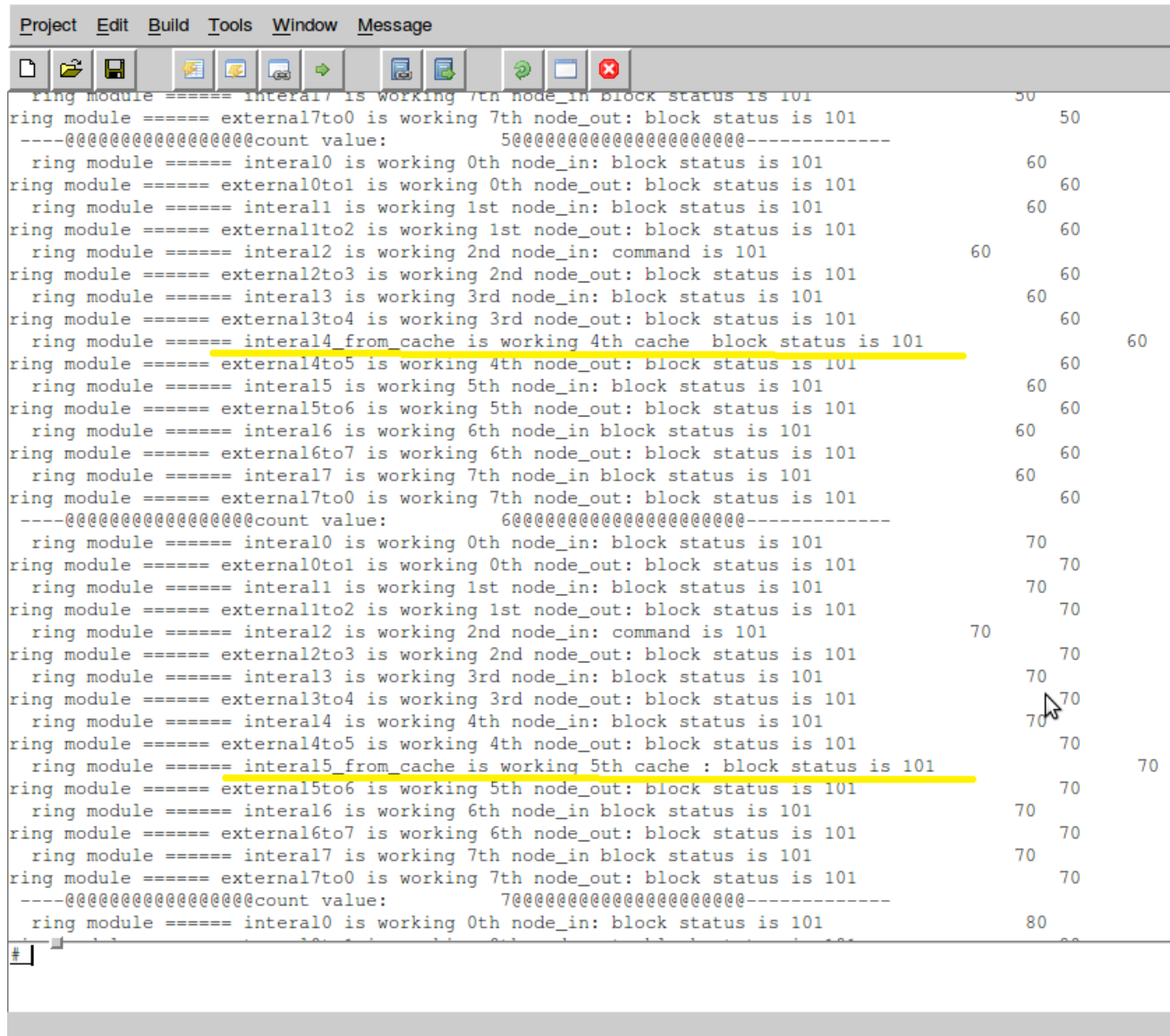


Figure 7.6: BSV SIMULATION RESULTS OF 5TH AND 6TH CLOCK CYCLES

In the figure 7.6 according to priority logic in fifth clock cycle core4 writes the data into cache4 .So cache4 changes its block status to “MODIFIED ” state and it propagates block status “INVALID”(101) to all other caches. In the figure 7.6 we can see the propagation of INVALID(101) signal through the Ring interconnect.So memory blocks in all the other caches will be in “INVALID” state.

In sixth clock cycle core5 writes the data into cache5 .So cache5 changes its block status to “MODIFIED ” state and it propagates the block status “INVALID”(101) to all other caches. In the figure 7.6 we can see the propagation of INVALIDATE(101) signal through the Ring . So memory blocks in all the other caches will be in “INVALID” state.

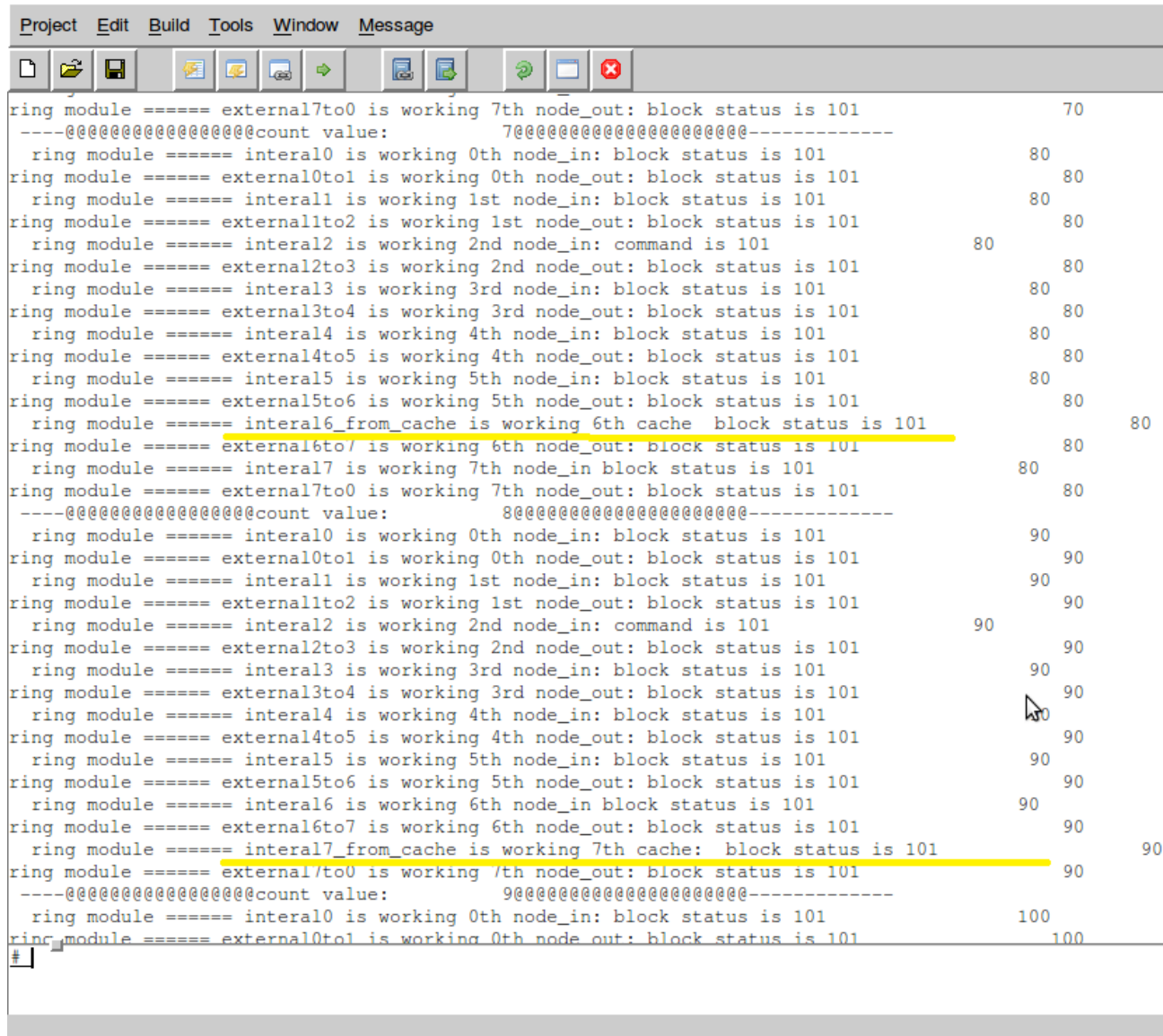


Figure 7.7: BSV SIMULATION RESULTS OF 7TH AND 8TH CLOCK CYCLES

In the figure 7.7 according to priority logic in seventh clock cycle core6 writes the data into cache6 .So cache6 changes its block status to “MODIFIED ” state and it propagates block status “INVALID”(101) to all other caches. In the figure 7.7 we can see the propagation of INVALID(101) signal through the Ring interconnect. So memory blocks in all the other caches will be in “INVALID” state.

In 8 th clock cycle core7 writes the data into cache7 .So cache7 changes its block status to “MODIFIED ” state and it propagates the block status “INVALID”(101) to all other caches. In the figure 7.7 we can see the propagation of INVALIDATE(101) signal through the Ring . So memory blocks in all the other caches will be in “INVALID” state.

```

Project Edit Build Tools Window Message
ring module ===== external7to0 is working 7th node_out: block status is 101 90
----@count value: 9-----
ring module ===== internal0 is working 0th node_in: block status is 101 100
ring module ===== external0to1 is working 0th node_out: block status is 101 100
ring module ===== internal1 is working 1st node_in: block status is 101 100
ring module ===== external1to2 is working 1st node_out: block status is 101 100
ring module ===== internal2 is working 2nd node_in: command is 101 100
ring module ===== external2to3 is working 2nd node_out: block status is 101 100
ring module ===== internal3 is working 3rd node_in: block status is 101 100
ring module ===== external3to4 is working 3rd node_out: block status is 101 100
ring module ===== internal4 is working 4th node_in: block status is 101 100
ring module ===== external4to5 is working 4th node_out: block status is 101 100
ring module ===== internal5 is working 5th node_in: block status is 101 100
ring module ===== external5to6 is working 5th node_out: block status is 101 100
ring module ===== internal6 is working 6th node_in: block status is 101 100
ring module ===== external6to7 is working 6th node_out: block status is 101 100
ring module ===== internal7 is working 7th node_in: block status is 101 100
ring module ===== external7to0 is working 7th node_out: block status is 101 100
=====
ALL cores REQUESTS SERVICED..NOW FETCHING NEW SET OF INSTRUCTIONS
=====
----@count value: 10-----
ring module ===== internal0_from_cache is working 0th cache: block status is 101 110
ring module ===== external0to1 is working 0th node_out: block status is 101 110
ring module ===== internal1 is working 1st node_in: block status is 101 110
ring module ===== external1to2 is working 1st node_out: block status is 101 110
ring module ===== internal2 is working 2nd node_in: command is 101 110
ring module ===== external2to3 is working 2nd node_out: block status is 101 110
ring module ===== internal3 is working 3rd node_in: block status is 101 110
ring module ===== external3to4 is working 3rd node_out: block status is 101 110
ring module ===== internal4 is working 4th node_in: block status is 101 110
ring module ===== external4to5 is working 4th node_out: block status is 101 110
ring module ===== internal5 is working 5th node_in: block status is 101 110
ring module ===== external5to6 is working 5th node_out: block status is 101 110
ring module ===== internal6 is working 6th node_in: block status is 101 110
ring module ===== external6to7 is working 6th node_out: block status is 101 110
ring module ===== internal7 is working 7th node_in: block status is 101 110
ring module ===== external7to0 is working 7th node_out: block status is 101 110
#

```

Figure 7.8: BSV SIMULATION RESULTS OF 9TH AND 10TH CLOCK CYCLES

In the ninth clock cycle the INVALID signal sent by cache7 routing in the ring. All cache requests were serviced by this time. Finally one cache i.e cache7 will have the most updated data. Hence data consistency is maintained. Once all the cache requests were serviced caches will now enqueue new requests from 10th clock cycle onwards.

## 7.3 Synthesis

Further the design is compiled to generate the verilog files which are required for the EDA tool to complete the logic synthesis as discussed in the design flow diagram. We not only receive an optimized netlist after logic synthesis but also reports for area and timing which are required for analysis.

The synthesis tool accepts the verilog files of the design and runs the synthesis algorithm for logic minimization. The synthesis culminates with generation of synthesized design schematic and detailed synthesis report with hardware units used in the final design. It is possible to selectively visualize the flow of the signals and the modules of interest making it convenient for the designer to verify the correctness of the design.

### 7.3.1 Device utilization summary

```
=====
Final Register Report

Macro Statistics
\# Registers          : 9017
Flip-Flops           : 9017

=====
\subsection{Device utilization summary:}
Device utilization summary:

Selected Device : 7a100tfgg484-3

Slice Logic Utilization:

    Number of Slice Registers:      9017 out of 126800   7\%
    Number of Slice LUTs:          19033 out of 63400   30\%
    Number used as Logic:           19033 out of 63400   30\%

Slice Logic Distribution:

    Number of LUT Flip Flop pairs used: 19223
    Number with an unused Flip Flop: 10206 out of 19223   53\%
    Number with an unused LUT:       190 out of 19223    0\%
    Number of fully used LUT-FF pairs: 8827 out of 19223   45\%
    Number of unique control sets:    25

IO Utilization:

    Number of IOs:                  20450
    Number of bonded IOBs:          16010 out of 285 5617\%(*)

Specific Feature Utilization:

    Number of BUFG/BUFGCTRLs:        2 out of 32   6\%

=====
```

Figure 7.9: Device utilization summary

## 7.3.2 Timing Report:

Timing Summary:

Speed Grade: -3

Minimum period: 8.858ns (Maximum Frequency: 112.893MHz)

Minimum input arrival time before clock: 8.895ns

Maximum output required time after clock: 8.739ns

Maximum combinational path delay: 8.776ns

Timing Details:

All values displayed in nanoseconds (ns)

Timing constraint: Default period analysis for Clock 'CLK'

Clock period: 8.858ns (frequency: 112.893MHz)

Total number of paths / destination ports: 1015241542 / 11730

Delay: 8.858ns (Levels of Logic = 17)

Source: cache\_req\_0\_rv\_358 (FF)

Destination: node\_in\_req3\_0/data0\_reg\_356 (FF)

Source Clock: CLK rising

Destination Clock: CLK rising

Data Path: cache\_req\_0\_rv\_358 to node\_in\_req3\_0/data0\_reg\_356

Cell:in->out	fanout	Delay	Delay	Logical Name (Net Name)
FD:C->Q	2	0.361	0.299	cache_req_0_rv_358 (cache_req_0_rv_358)
LUT3:I2->O	4	0.097	0.525	cache_req_0_rv\$port1__read<358>1 (cache_req_0_rv\$port1__read<358>)
LUT6:I3->O	2	0.097	0.384	CAN_FIRE_RL_interal0_from_cache<359>1_1 (CAN_FIRE_RL_interal0_from_cache<359>1)
LUT6:I4->O	3	0.097	0.389	node_in_req_0_rv\$port1__read<359>1 (node_in_req_0_rv\$port1__read<359>)
LUT6:I4->O	2	0.097	0.299	WILL_FIRE_RL_interal11_1 (WILL_FIRE_RL_interal11)
LUT6:I5->O	3	0.097	0.389	node_in_req_1_rv\$port1__read<359>1 (node_in_req_1_rv\$port1__read<359>)
LUT6:I4->O	2	0.097	0.299	WILL_FIRE_RL_interal21_1 (WILL_FIRE_RL_interal21)
LUT6:I5->O	3	0.097	0.389	node_in_req_2_rv\$port1__read<359>1 (node_in_req_2_rv\$port1__read<359>)
LUT6:I4->O	2	0.097	0.299	WILL_FIRE_RL_interal31_1 (WILL_FIRE_RL_interal31)
LUT6:I5->O	2	0.097	0.383	node_in_req_3_rv\$port1__read<359>1 (node_in_req_3_rv\$port1__read<359>)
LUT6:I4->O	553	0.097	0.475	WILL_FIRE_RL_interal41 (WILL_FIRE_RL_interal4)
LUT6:I5->O	8	0.097	0.543	node_in_req_4_rv\$port1__read<358>1 (node_in_req_4_rv\$port1__read<358>)
LUT4:I1->O	6	0.097	0.402	node_out_req_5_rv\$port1__read<359>1 (RDY_node_out_ifc_5_tx_get_OBUF)
LUT6:I4->O	1460	0.097	0.503	WILL_FIRE_RL_interal61 (WILL_FIRE_RL_interal6)
LUT6:I5->O	190	0.097	0.426	node_in_req_6_rv\$port1__read<359>1 (node_in_req_6_rv\$port1__read<359>)
LUT6:I5->O	359	0.097	0.452	node_in_req3_0/d0h1 (node_in_req3_0/d0h)
LUT6:I5->O	1	0.097	0.379	node_in_req3_0\$D_IN<178>1_SW0 (N1122)
LUT5:I3->O	1	0.097	0.000	node_in_req3_0/d0di_d0h_or_8_OUT<178>1 (node_in_req3_0/d0di_d0h_or_8_OUT<178>)
FD:D		0.008		node_in_req3_0/data0_reg_178
Total		8.858ns		(2.018ns logic, 6.840ns route) (22.8% logic, 77.2% route)

Figure 7.10: timing report

## **Chapter 8**

### **CONCLUSION AND FUTURE WORK**

#### **8.1 Conclusion**

The effort of the work presented here is to solve the coherence problem in multi-core system in a ring interconnect. The Cache module with (MOESI) protocol Using TLM (Transaction level modeling) have been successfully implemented in Ring interconnect with Parameterised no of nodes in BSV. All the nodes in a ring interconnect are communicated through TLM send and receive interfaces. Communication between cache and node is done with TLM send and receive interfaces and also implemented the priority ordering in the interconnect. This routing between all the nodes will be done with latency of ONE clock cycle.

#### **8.2 Future work**

The research on coherence on interconnect design for multi-core networks is relatively new field. With acceptance of the fact that multi-cores are the only viable option for performance scaling for next few decades to come.

Shrinking the transistor sizes do come with possibility of manufacturing defects and process variations. So although as of now ring interconnect is assumed to be fault free, reliable solution for packet exchange, the future interconnects will have to care for such challenges by exploring the possibility of using encoding schemes and other such protocols so that the interconnect can continue to provide reliable end to end solutions.

Here we implemented MOESI Protocol in L1 cache and maintained a coherence in ring interconnect network. To further this implementation we can implement All protocols in one module by giving some input bit it will select which protocol to use (like 1-MSI, 2-MESI, 3-MOESI, 4-MESIF etc) to achieve more efficient in timing point of view.

Not only ring bus topology, We can also further implement cache coherence to Ring,Mesh and torus. As the no of cores in processor increases its better to go for mesh ,torus so that latency will be reduced. We can also implement all topologies in a single system and can access any one by giving request (Like 1-Ring, 2-Mesh ,3-Star etc) depends on no of cores in the processor.



## Bibliography

- [1] Bluespec Inc. Bluespec System Verilog Reference Guide, Revision 30 July 2014.
- [2] Coherence Ordering for Ring-based Chip Multiprocessors, 39th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-39), 2006
- [3] RING-DATA ORDER: A new cache coherence protocol for ring-based multi-cores, High Performance Computing Simulation, 2009. HPCS '09. International Conference on, June 2009
- [4] The Performance of Cache-Coherent Ring-based Multiprocessors, Luiz An Barroso and Michel Dubois
- [5] RISE Lab, Shakti Series Processors.ppt
- [6] A Cache Coherence Protocol for the Bidirectional Ring Based Multiprocessor, Hitoshi Oi and N. Ranganathan