RING BASED ARCHITECTURE FOR DISTRIBUTED SHARED MEMORY CHIP MULTIPROCESSOR

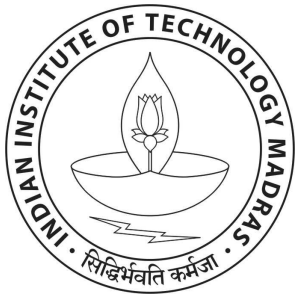
A Project Report

submitted by

GAURAV SEHGAL

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

MASTER OF TECHNOLOGY



DEPARTMENT OF
ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
May 2017

REPORT CERTIFICATE

This is to certify that the project report titled **RING BASED ARCHITECTURE FOR DISTRIBUTED SHARED MEMORY CHIP MULTI-PROCESSOR**, submitted by **GAURAV SEHGAL**, to the Indian Institute of Technology, Madras, as part of **Master of Technology**, is a bonafide record of the work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

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

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards several people who enabled me to reach this far with their timely guidance, support and motivation.

Firstly, I would like to thank my guide, Prof. V. Kamakoti and co-guide Prof. Nitin Chandrachoodan who despite his very busy schedule, always gave me a patient hearing and showed me the way through thick and thin on all matters concerned. His knowledge and an extraordinary ability to lighten the students with his positive approach towards things always infused me with great energy and positivity. The invaluable inputs and suggestions from him enabled me to achieve the desired goals during the project work.

I would like to extend a special thanks to my faculty advisor Prof. Shreepad Karmalkar and who continuously supported me throughout my course with his knowledge and advise. I would also like to thank Major Anirudh Agarwal, Mr. Rahul Mr.Abhinay whose persistent efforts and pursuit of practical knowledge has been motivational. The long discussions we had on deployment of latest systems in i-class processor. My special thanks and deepest gratitude to Dr.Neel Gala and Gopinathan who have been very supportive during the course of this project. They have enriched the project experience with their knowledge of the subject matter, active participation, deep understanding and invaluable suggestions. I thank to Major Ajay Fuloria, Major H S Mann, Arjun and all my RISE lab-mates whose acquaintance and support helped me in one way or other, throughout this learning process.

I would like to thank my parents for always standing by me. I also wish to express my gratitude towards Indian Navy for granting me this invaluable opportunity to pursue higher studies at IIT Madras.

Gaurav Sehgal

ABSTRACT

KEYWORDS: Cache Coherency, Snoop Based, Uni-directional, Bi-directional

The project involves design and implementation of "Ring Based Architecture for Distributed Shared Memory Multiprocessors". The work involves implementing snoop based ring architecture for distributed shared memory multiprocessors using flat MOESI and Tilelink protocol for scheduling the requests generated by processor cores during instruction execution.

An implementation of a uni-directional and bi-directional ring based architecture for distributed shared memory extended upto ring of rings, the messages will traverse all the way through the ring uni-directionally or bidirectionally respectively in Bluespec System Verilog which accepts memory access request from the processor and the processor searches its private cache and then schedules the request of the ring. The ring naturally orders requests sufficiently to enable directory-less coherence, but not in the total order that buses provide for snooping coherence. The process reduces the access time of fetching the data from the main memory in case of uni-directional as well as from coherent caches in case of bi-directional implementation as the same data might be available in one of the cores of the multi-core processor. The data from the responder reaches the initiator in a clockwise fashion in case of uni-directional implementation, whereas for bi-directional implementation the data can reach the initiator through dual path.

The complexity in the designing and verification of hierarchical coherence protocols has been reduced as each tier can be verified and evaluated in isolation. As a result, bug free design with protocol heterogeneity can be created without much difficulty. This would enable the architect to focus on performance enhancement rather than on debugging and verifying correctness of coherence implementation. The memory hierarchy being part of the flagship SHAKTI program of RISE Lab, successful implementation shall contributes for developing a indigenous Microprocessor.

This dissertation presents the implementation of a ring based topology that is simpler and more efficient than prior ring-based topologies. Our design uses simple ring networks that modifies the MCP based protocol for implementing snoopy based protocol by using clients of the MCP. The implementation provides a more scalable network architecture for implementing coherency protocols while retaining the key simplicities of ring network.

Contents

List of Tables

List of Figures

ABBREVIATIONS

BSV	Bluespec System Verilog
CMP	Chip Multi-Processor
DSM	Distributed Shared Memory
LLC	Last Level Cache
HDL	Hardware Description Language
PE	Processing Elements
SBP	Snoop Based Protocol
DBP	Directory Based Protocol
TCC	Theoretically Calculated Clock Cycle
BCC	Bi-directional Ring Clock Cycle
UCC	Uni-directional Ring Clock Cycle
UTL	Uni-directional Theoretical Latency
BTL	Bi-directional Theoretical Latency
UDL	Uni-directional Design Latency
BDL	Bi-directional Design Latency
AQ	Acquire
PF	Probe Forward
PR	Probe Request
SR	Search & Release
GF	Grant Forward
GR	Grant Request
VR	Voluntary Release
MI	Memory In
MO	Memory Out
LUT	Look Up Tables

Chapter 1

INTRODUCTION AND MOTIVATION

Multi-core processors dominate the microprocessor industry as the scaling of single core processor performance is rapidly reaching saturation. As the cores become more numerous and more diverse, interconnect scalability, performance and energy efficiency are the first-order concerns in the design of future CMP's. CMP's are built with greater number of cores, centralised interconnects are no longer scalable.

Cache hierarchy is often used with multi-core processor design in several applications for optimal performance. The use of shared memory presents numerous challenges with multiple caches sharing single memory, data traffic becomes huge and it may present a performance bottleneck.

The challenges in designing, testing and verifying advanced hierarchical cache coherence protocols can be overcome by the use of standardized coherence communication interface that provides protocol encapsulation. Among various interconnection networks that have been used for multiprocessor systems, the ring networks have the advantages of:

(a) Fixed node degree (modular expandability).

(b) Data path is very simple compared to a mesh router because it consist of several MUXes to allow data to enter and leave,one pipeline register.

(c) Simple network interface structure (fast operation speed).

(d) Low wiring complexity (fast transmission speed).

Because of these advantages several prototype and main stream commercial CMPs today most commonly use ring-based interconnects: the Intel Larrabee, IBM Cell, and more recently, the Intel Sandy Bridge.Onr16 Rings are a well-known network topology and the idea behind ring topology is very simple that all cores are connected in a loop that carries network traffic. At each core a new traffic can be injected in a ring and traffic in a ring can be removed when it reaches its destination.

Unfortunately, rings suffer from a fundamental scaling problem because of ring's bisection bandwidth does not scale with number of nodes in the network. Building more rings or a wider ring, serves as a stopgap measure but increases the cost. As commercially CMP's continue to increase core counts, a new network design will be needed that balances the simplicity and low overhead of rings with the scalability of more complex topologies. Hierarchical clustered cache design is one possible solution to this problem. Grouping cores and their caches in clusters reduces network congestion by localizing traffic among several hierarchy levels, potentially enabling much higher scalability.

The Ring Based Architecture for Distributed Shared Memory Multiprocessor form part of the SoC, hence its design is critical for both Area and Performance. The project being part of the flagship SHAKTI program of RISE Lab, successful implementation shall contributes for developing a indigenous Microprocessor.

In summary, major contribution:

- Inspired by Manager Client Pairing protocol.
- Modification to use Client module to maintain MOESI coherency for snoopy based protocol.
- Ring prioritizes request service to clients in clockwise direction and keep on snooping till the time request is not generated.
- In-ring buffering has been implemented and request are served in a round robin mechanism.

1.1 Problem Specification

To implement a Ring Based Distributed Shared Memory multiprocessors capable of interfacing with a MOESI protocol based private cache and last level cache of varying configurations in a unidirectional as well as bi-directional configuration for the purpose of execution of instructions at the processor core. The specifications are:

- Implementing MOESI based cache structure for ring architecture capable of being interface with the last level shared cache.
- The architecture design should be capable of unidirectional as well as bi-directional ring such that various operations never degrades the performance of ring.
- Designing of coherent cache structure in order to check the functionality and performance benchmarks respectively.
- Ring is designed in such a manner that multiple request from the respective cores could not be serviced by ring, rather request are serviced based on round robin mechanism i.e Client's are prioritise in the clockwise manner of their inter-connections with the ring, hence if request of any client has been serviced then the same client will get the service once rest of the clients request has been serviced by the ring in order to avoid data inconsistency.
- Bi-directional ring architecture, the request is to be forwarded in two direction (i.e clockwise and anti-clockwise)and processed simultaneously, data is serviced to client's in a bi-directional mechanism so there exist two paths for request/data to reach upto initiator.
- Ring should facilitate the testing of logic as well as cases through itself rather than the entire execution process to be followed.

- Ring should prioritise among the request itself i.e Read/Write request is prioritised over the Write-Back/Invalidate request. Write-Back/Invalidation are serviced using in-ring buffering.

1.2 Organization of the Report

The remaining part of the dissertation has been divided into following chapters for ease of understanding and coverage.

- Chapter II covers background to Ring architecture and functionalities which are focal to understanding the Ring Architecture module, cache coherency protocol specifically focused on MOESI based protocol and the various interfaces. It also highlights relevant details for implementing the design covered as part of literature survey done during Project phase I and thereafter.
- Chapter III includes the Bluespec System Verilog and related modules utilised for implementing the design.
- Chapter IV consists of functional description.
- Chapter V encompasses system architecture and implementation approach for attaining the objective.
- Chapter VI embraces the results and cognizance drawn based on application and limitations.
- Chapter VII epitomize the conclusion and future scope envisaged.

Chapter 2

BACKGROUND AND REVIEW OF RELATED WORK

2.1 Background

As we move towards multi-core processors, there is an increasing need for sophisticated coherence management. When cache hierarchies are used in the design, we also need to introduce coherence management hierarchies to achieve data consistency. The hierarchical coherence protocol based design increases the complexity in terms of the ease of designing, testing and verification. Also, this complexity increases exponentially with the addition of more coherence states in the protocol. This in turn, strains the interaction between the hierarchy tiers.

Ring based cache coherency protocol we discuss different classes of coherence protocols for a ring and concentrate on its design based on Shakti Programme. Design of a uni-directional as well as for bi-directional ring topology for cache coherency has been development and tested with an assumption of a ring comprising of quad-core processor with each core based on RISC-V having its own cache of the size of 32kB for data storage and a client for managing the MOESI coherency protocol for the respective caches. The ring has been designed in a manner that the it inherents the features of memory controller used for its request/data routing while implementing the methodology hence reducing an area for the separate memory controller.

2.2 Cache Coherence

In a shared memory environment, the private processor caches may contain copies of data which may be dirty with respect to main memory. Cache coherence handles the management and distribution of data in such cases. Without coherence, the consistency model of architecture could be violated. That is when a private processor cache commits a store which is not being observed in the local cache of other processors, it breaks the consistency among the copies of data stored in the private processor caches. This affects the fundamental way in which the processors communicate with each other in a shared memory environment. The loads and stores performed by every processor should be observed by every other processor for functional correctness and cache coherence ensures that this behaviour is maintained.

Cache coherence in hardware is accomplished through the addition of state bits to the data in cache, which indicates the coherence state of the data. The coherence state associated with the data depends on the coherence protocol used. The basic coherence states are invalid, shared and modified. Invalid state indicates whether the copy of data in cache is valid or not. Shared state of data implies that one or more of other caches also contain the copy of the same data. When a private processor cache commits a store, that copy of data is stored in modified state. These provide the basic mechanisms required to maintain coherency in the caches. A processor

can read from its local cache only if the data is valid. Also, when a local copy of data is written by the processor, all other shared copies are invalidated and the copy written is put in modified state. The two major classes are coherence protocols are broadcast based protocols and directory based protocols. In the directory based coherence management, the coherence state of all the data in the local caches is maintained in a directory. Whenever a processor has to perform read or write, the request is sent to the directory and the operation is performed if the required permissions are granted by the directory. In a broadcast based protocol, read or write miss is broadcast through a shared bus and every local cache is snooped to check if the requested data is present in them. Also, when a processor performs a store on a shared copy of data in its local cache, invalidation request is sent to other caches to invalidate the other copies of that data. The invalidation of the shared copies in the event of a write operation by the processor is done in invalidation based protocols. There is another variant of coherence protocol which handles write-hit in a different manner. Instead of invalidating the shared copies, the written value is broadcast to all the caches that share this data and they are updated with the new value. This protocol is referred to as the update-based protocol.

The choice of a coherence protocol primarily depends on the application. Different protocols can be best suited for different hardware implementations. In this dissertation, broadcast based MOESI coherence protocol is used.

2.3 MOESI Protocol

As already mentioned, we will be using broadcast based MOESI protocol, which will handle write-hit by invalidation. Initially, when the data is fetched from the memory, it is stored in Exclusive(E) state, as it will be the only copy of that data among the caches. When this data is read by some other cache, it is no longer an exclusive copy.So, the state is changed to Owner(O) and the cache which read this data will store it in Shared(S) state. When the data in Shared(S) state is read by other caches, its state does not change.

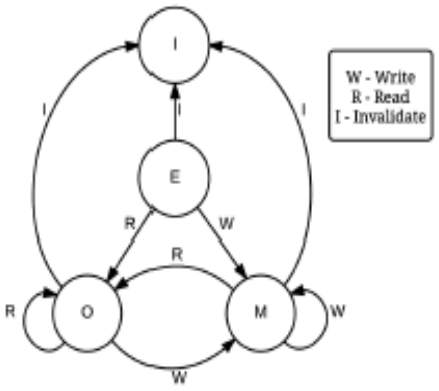


Figure 2.1: Block Diagram: State Transition beginning with data in E state

Reading of data in Owner(O) state, by other caches does not change the state of the local copy. However, writing of data in Owner(O) state or Shared(S) state, by the processor, changes its

state to Modified(M) and the invalidate request is sent to other shared copies. Writing of data in Modified(M) state, will cause it to remain in modified state. When data in Exclusive(E) state is written, its state is updated as Modified(M), but invalidate request is not sent out as there are no shared copies to invalidate. The data in any coherence state, when invalidated, is put in Invalid(I) state [1].

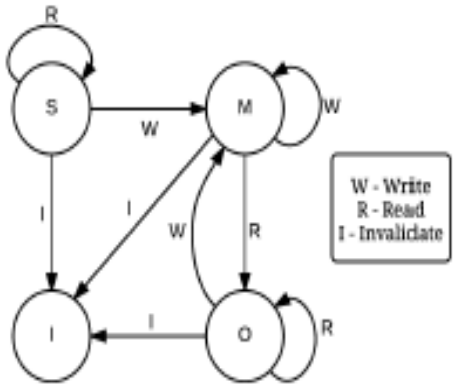


Figure 2.2: Block Diagram: State transition beginning with data in S state

2.4 Coherency Mechanism

The protocols are implemented in client interface connected to the ring as well as the core, the cache is connected to the processor. The requested generated by the processor is sent over ring via clients. For data messages, the interface examines each to determine if it should be pulled from the ring or forwarded to the next client. Common mechanisms of ensuring coherency have been listed below with each having its own benefits and drawbacks:

- **Snoop Based Protocol**: SBP is the most popular protocol because of its simplicity and influence in desktop and server solutions, as well as in recent SoCs. It relies on broadcasts, and snooping of those broadcasts, to ensure coherence.
- **Directory Based Protocol**: DBP employs a directory where information about each cache line is stored. This way the directory has full control over which core has loaded which cache line.
- **TokenB**: It is a protocol where each cache line has a number of tokens associated with it. The number of tokens has to be at least as high as the number of cores. Any core which possesses at least one token can read, any core that has all the tokens for the cache line has write privileges.
- **Snarfing**: Snarfing directly updates local cache data without going through a centralized memory.

2.5 Interconnect Topologies

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**: In Ring interconnect all the cores or nodes are connected in a ring fashion, request as well as data packets are send in uni-directionally (i.e either in clockwise or anticlickwise direction) or bi-directionally to reach destination client. The rings main function is to facilitate the transfer of packets, prioritising the service request among the clients and prioritising the request itself as the case may be or between caches and nodes.

- **Mesh Interconnect**: 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.

- **Ring-Mesh hybrid interconnect or Torus**: The ring and mesh interconnects 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 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.

The bi-directional ring interconnect for the cache coherency can be upgraded for Ring-Mesh hybrid based on the Manager Client Pairing [MCP] where clients have already been connected in a ring fashion and managers can be connected in the mesh form.

The Torus is used for multi-core processors as the resource requirement is more for the quad-core than the performance upgradation is relatively the same. Hence we have designed unidirectional ring interconnect as well as bi-directional ring interconnect as the resource area does not increases as much in relation to uni-direction ring but the performance is relatively improved on a much better scale. The MOESI protocol is implemented for the cache coherency.

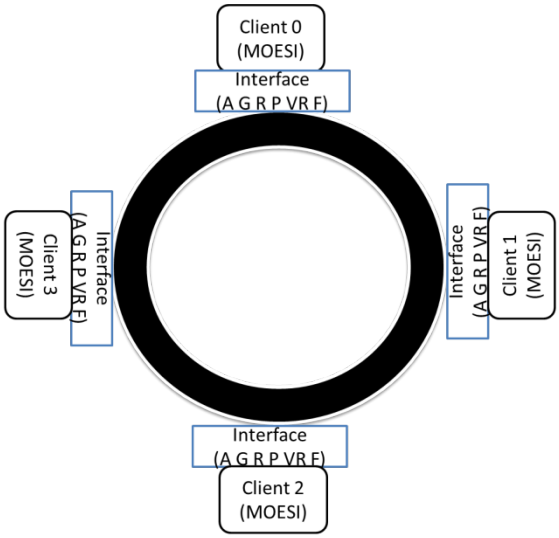


Figure 2.3: Block Diagram: Ring Interconnect

2.6 Related Work

To develop a better understanding for designing ring architecture I have gone through the research paper of Michael R. Marty and Mark D. Hill (University of Wisconsin Madison) on "Coherence Ordering for Ring-based Chip Multiprocessors" presented in proceedings of the 39th Annual IEEE/ACM Symposium on micro architecture[3], for extending the sphere of the design and for Jesse G.Beu, Michael C. Rosier, Thomas M. Conte research on "Manager-Client Pairing: A Framework for Implementing Coherence Hierarchie" presented in MICRO-44 Proceedings of the 44th Annual IEEE/ACM International Symposium on Micro architecture [2], for cache coherency interfacing and modification have studied from M.Tech Thesis on "Implementing Cache Coherence through Manager-Client Pairing" submitted in year 2016 [1]. In her work the author does a detailed analysis of the architecture of MOESI based cache and coherency implementation using a manager client pair. The work on extending the scope for implementation of the design to multiple hierarchical level i have studied research paper of Rachata Ausavarungnirum,Chris Fallin,Xiangyao Yu, Kevin Kai-Wei Chang,Greg Nazario,Reetuparna Das,Gabriel H.Loh and Onur Mutlu (Carnegie Mellon University, University of Mechigam, Massachusets Institute of Technology, Advanced Micro Devices) on "A case for hierarchical ring with deflection routing: An energy-efficient on-chip communication substrate" presented in ELSEVIER journal of Parallel Computing [4].

Chapter 3

BLUESPEC SYSTEM VERILOG

Bluespec System Verilog is a Hardware Description Language (HDL), which is used for specification, synthesis, modeling and verification of ASIC and FPGA design. With a radically different approach to highlevel synthesis, bluespec offers significantly higher productivity. It allows designers to express intended hardware through high-level constructs, where all behavior is described as a set of guarded atomic actions.

3.1 Limitation of Verilog

Verilog focusses more on simulation than logic synthesis. The source text of verilog often explicitly contains aspects of circuit that could be readily determined by the compiler, such as size of registers, width of busses etc. This makes the design less portable. Handling concurrency in hardware is relatively difficult in verilog as the designer should manage all the aspects of handshaking between combinational circuits. Shared use of register and other memory resources should also be elaborated. The behavioral specification of design in verilog often consumes multiple clock cycles. Attempts to resolve this problem results in a highly unreadable code with possible bugs. In practice,this problem is solved by separating the combinational and sequential parts of the circuit. Due to these shortcomings, the synthesis and verification of hardware in verilog is slowed down. This is a huge problem during the design of SOC.

3.2 Bluespec

Bluespec is based on atomic transactions, which increases the level of concurrency abstraction above SystemC and RTL without compromising the control over hardware design. It enables automatic synthesis of complex control logic, which is the source of many bugs. This results in highly adaptable, reusable and reconfigurable designs. Control adaptive parametrization in bluespec provides flexibility, where a significantly different micro-architecture can be generated by changing the parameters in the design with the associated control structures generated automatically. Bluespec allows user defined data types and static type checking. It provides several features of the modern high level languages and all of them can be synthesized.

In recent times, several attempts have been made to move the hardware design language towards a more software like specification of the circuit behaviour. Languages like C, C++ are used to express designs as sequential programs. However, the semantic gap between the software model and the hardware results in suboptimal designs with unpredictable speed and area. Bluespec System Verilog tackles this problem by building upon the traditional hardware semantics. It exploits advanced concepts from software only for static elaboration and static

verification. It uses the standard hardware structure model of verilog such as modules, module instances, hierarchy etc. For communication between modules it uses the System verilog model of interfaces and interface instances. These added with the advanced features of the high level languages, makes designing and verification in bluespec much faster.

3.3 Features of Bluespec

3.3.1 Modules and Interfaces

Module is the basic element of the hardware design hierarchy in bluespec. A module can be instantiated multiple times, and also different parameters can be passed during every instantiation. Unlike verilog, bluespec does not have input, output and in-out pins as interface to modules. Methods are used to drive signals and busses in and out of modules. These methods are grouped together into interfaces. Modules contain rules, which use methods in other modules.

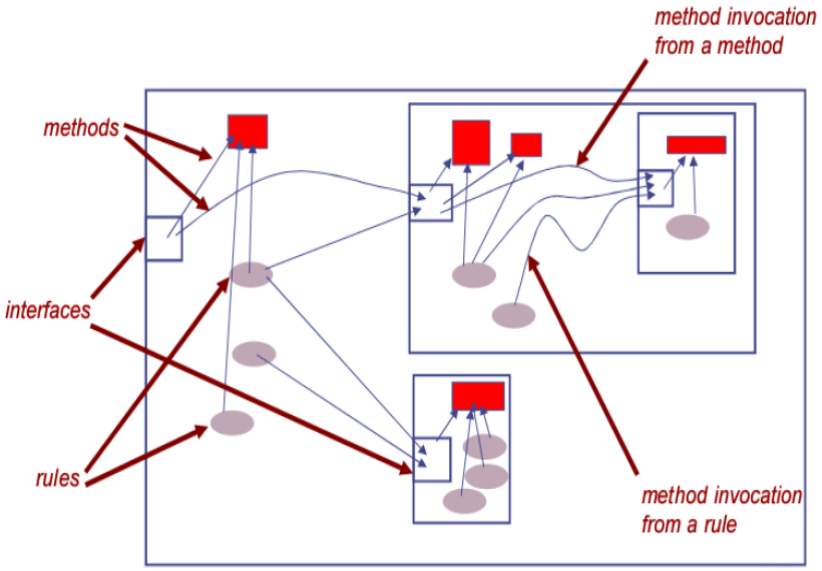


Figure 3.1: Block Diagram:Representation of Methods, Interfaces Rules in a module hierarchy

In BSV, the interface declaration is done separately, outside the module definition. This allows declaration of common interfaces which can be used in multiple modules,without having to declare them repeatedly. All the modules which share the same interfaces also share same methods and therefore share same number and type of inputs and outputs.

3.3.2 Data Types

In verilog, all the representation is done in bits. Also, ultimately in hardware all computation is done in bits. However, representation in terms of integers, floating point numbers, fixed point numbers etc, makes the process of coding much easier. Different representations may be more appropriate depending on the application environment. By separating out the type abstraction from its bit representation, we can easily change representations without modifying or breaking the rest of the program.

In BSV, every variable has a type and only the values of compatible types can be assigned to a variable. The BSV compiler provides a strong, static type-checking environment. Type checking is done before the program elaboration and it ensures that the object types are compatible and the conversion functions are valid for the context. Bluespec also allows the usage of user-defined types. BSV has a type class which can be considered as a set of types. It implements overloading across related data types. Overloading is the ability to use a common name for a collection of types, with the specific type for the variable being chosen by the compiler based on the types on which it is actually used. Functions and operators are shared by all the data types within a type class.

Some common scalar types used in Bits type class are Bit(n), Bool, UInt(n) and Int(n). The values stored in registers, FIFOs and other memory elements and also the values passed by wires, must be in the Bits type class. Other common data types include Integer, which belongs to the Arith type class and String, which belongs to the Literal type class etc.

3.3.3 Rules

Rules manage the movement of data from one state to another, within the module.It consists of two parts: rule conditions and rule body. Rule conditions are boolean expressions which decide whether the rule can be fired. Rule body is a set of actions for state transitions. Rules in BSV are atomic. The actions within the rule completely describes the state transition. The process of determining the functional correctness of a design is greatly simplified by one-rule-at-a-ime semantics. That is, because of the atomic property of rules, each rule can be looked at in isolation, without considering the actions of the other rules to determine functional correctness. Multiple rules can be executed concurrently in the hardware implementation.

The actions in a rule are executed simultaneously. This can be thought of as similar to the execution of non-blocking statements in always blocks of verilog. Also, as the rule has atomic property, the entire body of rule is executed and there is no partial exe- cution of a rule. When there are several rules within a module, the execution of rules is ordered by the compiler. No two rules can execute simultaneously. The ordering of the rules by the compiler is called scheduling.

3.3.4 Methods

method is a procedure which takes arguments and returns a value. It could also return a value without taking any arguments. It becomes a bundle of wires when translated into RTL. The method definition is written within the definition of the interface and it can be different in different modules sharing a common interface. A method also contains implicit conditions which are handshaking signals and logic automatically generated by the compiler. Methods are of three types: Value Methods, Action Methods and Action Value Methods. Value methods return a value. They do not alter any state within the module. Action methods cause actions to occur. They create state changes within the module. Action value methods are a combination of value methods and action methods. They cause state changes and also return values.

3.3.5 TLM Library

The TLM package includes definitions of interfaces, data structures, and module constructors which allow users to create and modify bus-based designs in a manner that is independent of any one specific bus protocol. Designs created using the TLM package are thus more portable as it allows the core design to be easily applied to multiple bus protocols.

The TLM interfaces define how TLM blocks interconnect and communicate. The TLM package includes two basic interfaces: The TLMSendIFC interface and the TLMRecvIFC interface. 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. Additional TLM interfaces are built up from these basic blocks. The TLMSendIFC interface transmits the requests and receives the responses. The TLMRecvIFC interface receives the requests and transmits the responses.

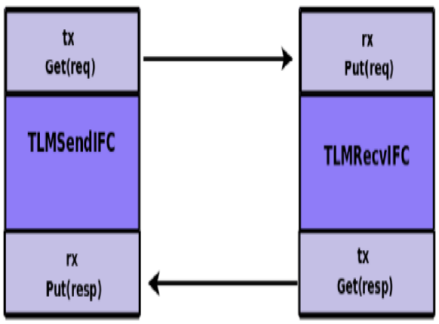


Figure 3.2: Block Diagram: Representation of TLMSendIFC and TLMRecvIFC

The two basic data structures defined in the TLM package are TLMRequest and TLMResponse. By using these types in a design, the underlying bus protocol can be changed without having to modify the interactions with the TLM objects. A TLM request contains either control information and data, or data alone. A TLMRequest is tagged as either a RequestDescriptor or RequestData. A RequestDescriptor contains control information and data while a RequestData

contains only data.

3.3.6 Tile Link

Tilelink is a protocol designed to be a substrate for cache coherence transactions implementing a particular cache coherence policy within an on-chip memory hierarchy. Its purpose is to orthogonalize the design of the on-chip network and the implementation of the cache controllers from the design of the coherence protocol itself. Any cache coherence protocol that conforms to TileLinkâs transaction structure can be used interchangeably with the physical networks and cache controllers we provide.

Tilelink is roughly analogous to the data link layer in the IP network protocol stack, but exposes some details of the physical link necessary for efficient controller implementation. It also codifies some transaction types that are common to all protocols, particularly the transactions servicing memory accesses made by agents.

Chapter 4

FUNCTIONAL DESCRIPTION

4.1 Core

Core has been designed on RISC-V architecture. Testing of ring architecture was initially done by designing a cache-mimic and memory-mimic instead of actual cache or main memory.

4.2 Cache

The cache implemented for the coherency is a distributed I-cache and D-cache of 32kB (non-blocking cache). It is a set associativity "4-ways". There are 512 in number cache lines. Cache has a block size of 16bytes.

4.3 MOESI based Client Module

The coherence protocol used here is MOESI protocol. It has five states: Modified(M), Owner(O), Exclusive(E), Shared(S) and Invalid(I). The client which has the data in the Exclusive state or the Modified state has the only copy of that data. Although the client with data in Owner state does not have a unique copy of the data, it is the only client among the sharers, which can respond to a request. Also, at any point of time, only one of these states(M,O,E) can be present among the clients of the same level. Therefore, data propagation can be easily handled by the manager as it can directly grant data from the only client in M/O/E state to the client which requested that data block.

Let us consider the handling of write operation in MOESI protocol. After write, the copy of the data block which is written, becomes dirty with respect to the sharers. In order to track this information, the Modified(M) state is used by the client. In the event of a write hit in the processor, the client changes the state of that data to Modified(M) and sends an invalidate request if the previous state of the data written was Owner(O) or Shared(S) state. During write-back, if the data block is in M/E state, it is directly written in the main memory. If in Owner(O) state, in addition to sending the data to main memory, all the other shared copies are invalidated. If the data is in Shared(S) or Invalid(I) state, the write-back operation is not performed. Table summarizes and enumerates a comprehensive list of the base functions required for communication between processors, clients, ring and memory in MOESI protocol. These will be used as an aid in the developing a generic protocol interface

The basic requirements that have to be satisfied by the agents involved in coherence management are as follows. Clients should be able to respond whether or not they have the requested

data block. If the cache associated with a client encounters a read or write miss, the client should be able to place a request to the ring. Ring should accept data requests from clients and provide data from the appropriate location. The client should also be able to send out invalidate request when necessary and the ring should forward it to all the other clients in the same level. In case of write back, the clients and ring should forward the data till the main memory.

As the management of coherency is completely handled by the client, the ring need not be aware of any internal change of coherence states. The ring only has to keep track of whether the data is obtained from which client and to be forwarded to the initiator. This information is used by the clients to appropriately update the coherence states.

Origin Agent	Action Type	Action	Description	Destination Agent	Response Action
Processor	Data Acquisition	GetData	R/W miss: Get data to complete CPU request	Client	GetData
	Invalidation	FwdInval	Write hit: Invalidate shared copies	Client	FwdInval
	Data Supply	DoWrite	Block replacement; writeback	Client	DoWrite
		GrantData	Supply Data	Client	GrantData
Client	Data Acquisition	GetData	Request from processor	Manager	GetData
		GetData	Request from manager; change state to O	Processor	GrantData
	Invalidation	DoInval	Invalidate copy in local cache if hit in O/S state	Processor	CompleteInvalidate
		FwdInval	Change state to M; forward request to manager if O/S state	Manager	DoInval
	Data Supply	DoWrite	Only when in M/O/E. Invalidate shared copies when O	Manager	DoWrite; DoInval
		GrantData	Got data from processor	Manager	GrantData
		GrantData	Got data from manager; State – E if from mem, S otherwise; send data to processor	Processor	Complete R/W; when write, send
Manager	Data Acquisition	GetData	Gets data from client if it is in M/O/E state	Client	GetData
		GetData	If no hit in any client	Memory	GrantData
	Invalidation	DoInval	Invalidate shared copies	Client	DoInval
	Data Supply	GrantData	Data supply from M/O/E clients or from main memory; indicate if data obtained from main memory	Client	GrantData
		DoWrite	Forward request to higher level	Memory	GrantWrite
Memory	Data Supply	GrantData	Supply Data	Manager	GrantData
		GrantWrite	Finish write	N/A	N/A

Figure 4.1: List of the base functions required for communication between processors, clients, ring and memory in MOESI protocol

The client has four interconnections Acquire(A), Probe(P), Grant(G), Release(R). The acquire signal is raised by the processor initiating the request to the ring. The probe is forwarded to the clients in a sequential fashion by the ring till the time the request is served. Corresponding to each probe client gives a response in form of release signal. Ring evaluates the release and checks for the data requested and received. Finally the response is routed towards the initiator on the grant signal by the ring.

4.4 Request Generated by Client

The clients are prioritised in the sequential clockwise manner and the clients are serviced by the ring on the priority token which defines the ring to which client it should provide service. If the clients are idle and does not have any request to be served by the ring the priority token is incremented automatically for the ring and hence it keeps on snooping for the request of the clients to be served. The request are of the following nature:

- **Read**: The request is generated by the client as a Read request in Regular mode. It generates an Acquire signal in one of the client which acts like an initiator and the based on the priority token the ring serves the client whosoever is prioritise for service. The signal is passed on to the successive client in the form of Probe. Based on which the client searches its respective cache and then responds back to ring in the form of Release signal. The ring introspects the response and find out weather it is a Hit/Miss and based on response takes necessary action of forwarding request in form of Probe signal or sends data in form of Grant signal. The Grant signal is forwarded till the time it reaches initiator.

- **Write**: The request is generated by the client as a Write request in Regular mode, served as the Write-back request by the ring. The Write request is received in the form of Release signal by the ring. The request if received in between of any service already initiated by ring to another client is firstly stored in the ring as Write-back request and served based on the priority token prioritising the client raising Write request. The request is converted into the Acquire signal, client who initiated is made as an initiator and then forwarded to successive clients as the Probe signal. In this case the request is completed once it reach the all the client holding the same set of data and finally the main memory. Release signal received post Probe returns the completion of Write request by the individual clients holding same set of data.

- **Invalidate**: The request is initiated by client as Write request in Controlled mode. The Invalidate request is received as a Release signal by the ring. The request if received in between of any service already initiated by ring to another client is firstly stored in the ring as Invalidate request and served based on the priority token prioritising the client raising Invalidate request. The request is converted into the Acquire signal, client who initiated is made as an initiator and then forwarded to successive clients as the Probe signal. Request is completely serviced once it reach the all the client holding same set of data including the main memory. Release signal received post Probe returns the completion of the Invalidation by the individual clients holding same set of data.

The ring has been designed in a manner that it will prioritise the Write-back over Invalidate request.

Chapter 5

SYSTEM ARCHITECTURE AND IMPLEMENTATION

5.1 Ring Architecture Module

A request in a ring protocol is active immediately, does not require retries to handle contention, and incurs minimal latency and bandwidth. The project tries to implement a class of protocols that achieves these goals by completing requests in unidirectional as well as bi-directional ring order.

5.1.1 Uni-directional Ring Architecture

This protocol is implemented by tokens, directly enforcing the coherence invariant by counting tokens. Forward progress exploits the ring order to guarantee that initial request always succeed. Token coherence associates a fixed number of tokens for each memory block in the system. To perform the Read Operation (RO) in any one of the memory the init token is enabled for the client initializing the RO request, directly forwarded to the next client in clockwise direction. The ring gets the response form the client and post evaluation of response performs the operation of checking whether the successive client need to ask for the RO request from the previous client or not. Once the data is located it is forwarded directly to the next client and then the ring performs the check weather this client is an initiator or not. If not then weather the successive client need to request for the data so that it can be served to the initiator or not. In this way, the coherence invariant is directly enforced by counting and exchanging tokens. The key insight is that token counting allows a requester to remove tokens off the ring to complete its request safely and potentially immediately.

To ensure starvation avoidance, requesting node must remove the incoming priority token from the ring and hold onto it until its request completes. Other non-priority tokens, in flight due to a write-back or exclusive request, must alesce with the priority token [3].

Tokens causes request message to move in a clockwise/anti-clockwise direction. Response message is not strictly sent to a particular requester and can instead be used by other requesters on the way. Response message includes a furthest destination field to indicate the furthest relative node on the ring that desires the tokens for a coherence request. Another key advantage is that RING-ORDER does not require any fixed synchrony in the ring or when snoop responses generate. One suspected negative aspect of this protocol is that if the data is found in the successive client then it has to travel along the entire ring to reach the initiator. The remedial for the same would be to design a bi-directional ring.

5.1.2 Bi-directional Ring Architecture

This protocol is implemented by adding four one bit flags along with the tokens of Acquire, Probe, Grant Release. The logical implementation has changed in a manner that the the initial request has been directed bi-directionally (i.e clockwise anticlockwise) from the initiator and the flag of prb is made high. When a response is received the rel flag is made high hence the data moves in the ring and directed based on the flags and token. The data once received in any of the cache is also sent in a bi-directional fashion, hence limitation of the methodology discussed for the uni-direction has been overcome. The data as well as request could reach upto the initiator or the requester respectively in two directions. The result for a quad-core processor might not produce very huge range of improvement relative to unidirectional implementation, improvement in throughput has been achieved and the results are more vibrant once the implementation crosses more than quad-core scope.

To further increase the throughput of multi-core processor the bi-directional can be enhanced to a hybrid network inclusive of bi-directional ring for clients and mesh for the manager which have already been implemented in MCP. The hardware will effectively increase but the trade off need to be made based on performance and work area and a mid way could be implemented. Hence for the quad-core we have implement bi-directional ring.

5.2 Initial Methodology for Implementation

There was an initial methodology of designing a ring module and instantiating the same with the clients but this leads to the more number of routing and requires an ordering point and a controller to keep maintain the data routing and to keep the track of the clients. The process complexity arises in designing the controller/ ordering point. In case of ordering point the data has to every-time pass thorough the node of the ordering point and then can be routed.

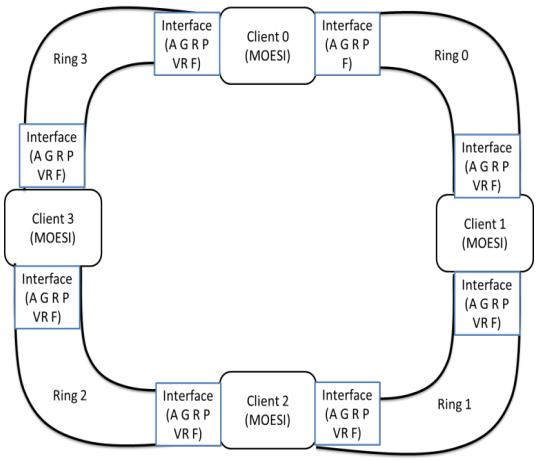


Figure 5.1: Block Diagram: Ring Architecture

5.3 Unidirectional Implementation

The ring structure will inter-connect the total number of clients instantiated in the ring fashion using get/put interfaces. The ring takes control of routing of data/request packet inside the ring, manages the request generation and completion based on token protocol. The request generation by the clients can be prioritized in a sequential manner the way they are connected with ring and the second request is served once the first is completed.

- **Case I**: The request is originated by "Client 0" i.e ***Acquire***, it means the data is not present in cache of "Client 0". The *Initiation Token* of "Client 0" is made true stating that it is the originator. The ***Acquire***is forwarded as a ***Probe*** to "Client 1" by the ring. "Client 1" takes ***Probe*** and searches its own cache returns ***Release*** to ring. Ring checks ***Release*** and find that data is not found in the cache of "Client 1" i.e **Miss**. The ring starts calculating that which client number has given the **Miss** and then makes *Forward Token* of successor i.e "Client 2" true and makes*Forward Token* of previous i.e "Client 1" false. Based on *Forward Token* true value "Client 2" ask for a "Probe" from "Client 1".Now "Client 2" on receiving ***Probe*** searches its own cache and returns ***Release***. Ring checks ***Release*** and find that data is found in the cache i.e **Hit**. The ring checks that which client number has given the "Hit" forwards ***Release*** as a "Grant" to "Client3"| and makes the *Data Token* of "Client 3" true. The ring checks weather "Client 3" is originator of the request or not. The ring then makes "Client 0" *Data Token* true and "Client 3" *Data Token* **False**. Based of *Data Token* true value "Client 0" ask for a ***Grant*** from "Client 3" Ring again checks weather "Client 0" is originator or not once reach originator it goes to ***Ideal*** state. Now "Client 1" can originate the request for the ring and hence, process continues.

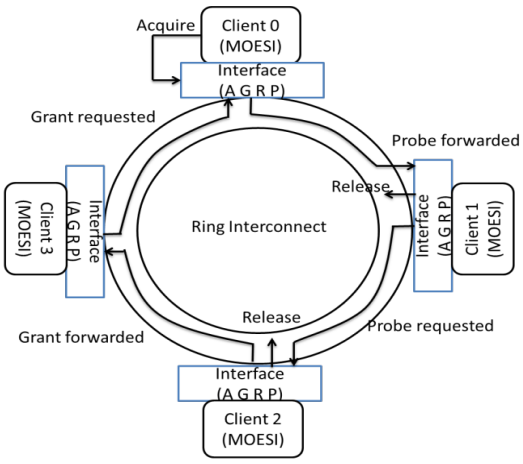


Figure 5.2: Implementation Diagram: Uni-Directional Ring Architecture

- **Case II**: If there is no request origination from "Client 0" it will give "Voluntary Release" and ring will check whether there is any request from the successive client in a sequence. The priority of the request if served based on the sequence of clients i.e in descending order " $Client0 \rightarrow Client1 \rightarrow Client2 \rightarrow Client3$" by using a priority token. Once a request is raised then till the time the request is not served none of the clients can raise another request.

5.4 Bi-directional Implementation

The ring structure inter-connects the total number of clients instantiated in the ring fashion using get/put interfaces. The ring takes control of routing of data/request packet inside the ring, manages the request generation and completion based on tokens and flag. The request generation by the clients can be prioritized in a sequential manner they are connected with ring and the second request is served once the first is completed.

- **Case I**: The request is originated by "Client 0" i.e ***Acquire***, it means the data is not present in cache of "Client 0". The *Initiation Token* and *Acquire Flag* of "Client 0" is made true stating that it is the originator. The ***Acquire***is forwarded as a ***Probe*** to "Client 1" "Client 3" simultaneously the ***Probe Flag*** is also made high by the ring. "Client 1" and "Client 3" takes ***Probe*** and searches its own cache returns ***Release*** to ring and ***Release Flag*** is made high. Ring checks ***Release*** and find that data is not found in the cache of "Client 1" i.e **Miss**. The ring starts calculating that which client number has given the **Miss** and then makes *Forward Token* of next client i.e "Client 2" true along with the ***Probe Flag*** and makes*Forward Token* of previous i.e "Client 1" false. Based on *Forward Token* being true and ***Probe Flag***being low value "Client 2" ask for a "Probe" from "Client 1".In the mean time the data might be found in "Client 3" i.e **Hit**.Now the data would be forwarded by "Client 3" in form of ***Grant*** to its adjuscent client i.e "Client 2" and "Client 0" makes the *Data Token* true along with ***Grant Flag*** is made high.The ring checks weather "Client 0" and "Client 2" is originator of the request or not.Now "Client 2" has received ***Probe*** as well as ***Grant*** so it first prioritise ***Probe*** over ***Grant*** and searches its own cache.The data has already reached its initiator "Client 0".Hence on receiving ***Release*** by ring from "Client 2"it is killed.This particular methodology of killing ***Release*** is applicable only for quad-core.

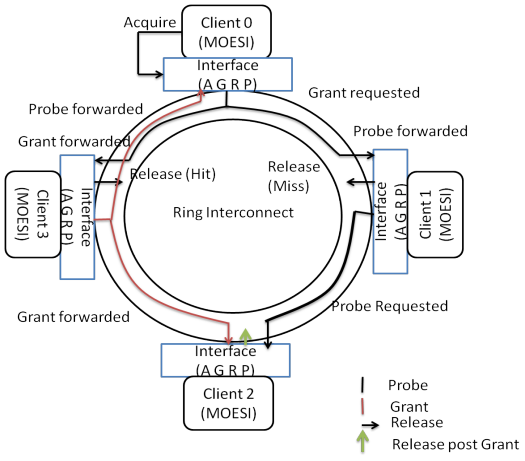


Figure 5.3: Implementation Diagram: Bi-directional Ring Architecture

- **Case II**: If there is no request origination from "Client 0" it will give "Voluntary Release" and ring will check whether there is any request from the successive client in a sequence. The priority of the request if served based on the sequence of clients i.e in descending order "$Client01 \rightarrow Client2 \rightarrow Client3$" by using a priority token. Once a request is

raised then till the time the request is not served none of the clients can raise another request.

5.5 Design Verification

The design of the ring interface with the request generated by the clients has been verified based on the layout described. The design testing was performed over the ring for the routing of packets in uni-direction as well as bi-directional architecture and running the test cases for the calculation of latency. To test the Hit/Miss for the data found in a cache, cache mimic was designed with a singular input of request and corresponding output of response. Similarly a dummy main memory was designed on the same line to test the design. The ring treats main memory in the form of an additional cache which was also designed as a dummy cache. The difference between the dummy memory and the cache mimic is that cache mimic is connected with the client and the processor whereas dummy memory is connected with the ring directly in a star formation i.e whenever request have been forwarded to all clients in the ring thereafter it would be forwarded to main memory.

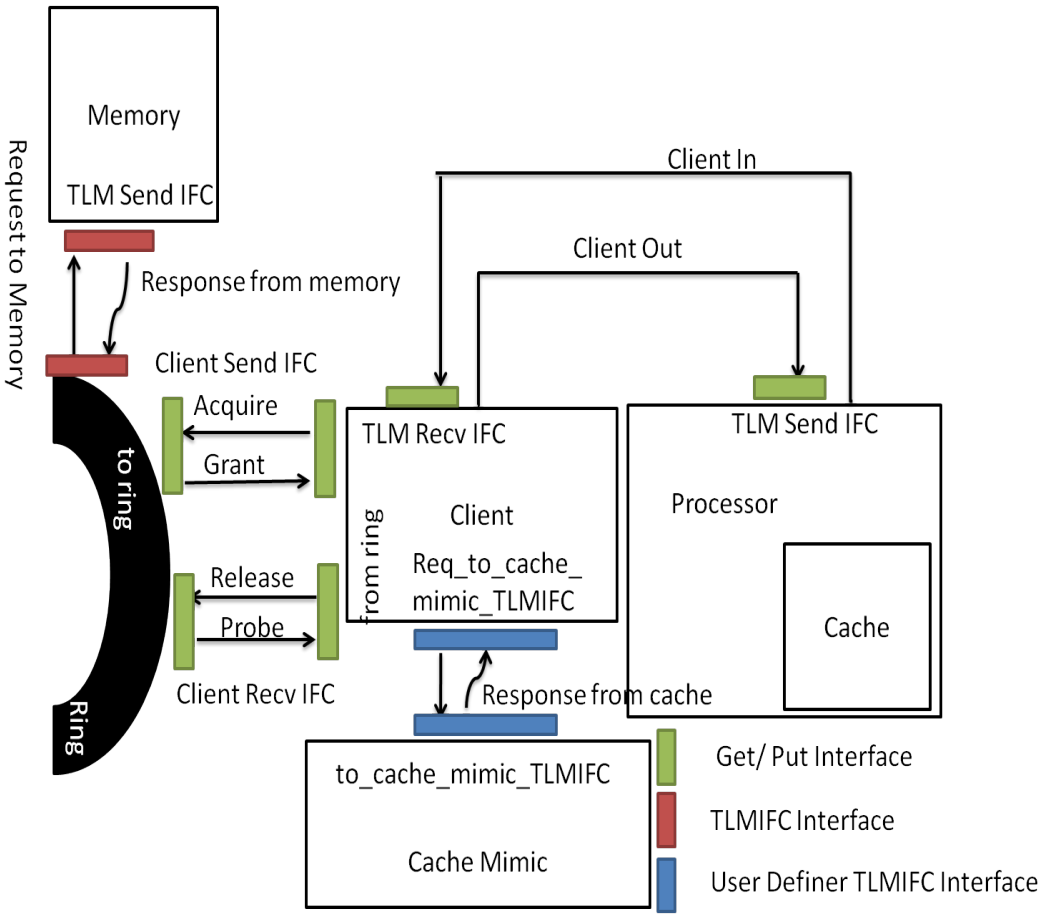


Figure 5.4: Block Diagram: Testing Layout with Interfaces

Chapter 6

RESULTS AND DESIGN CHALLENGES

6.1 Latency Calculation

The latency calculations have been done based on theoretical assumptions as well as designed based Bluespec compilation.Latency calculations varies based on design architecture (i.e uni-directional / bi-directional), calculated while compiling the code in Bluespec based on execution of rules and the timing predictions.

Priority Latency = (VR x 3) = (1 x 3)
= 3 clock cycle

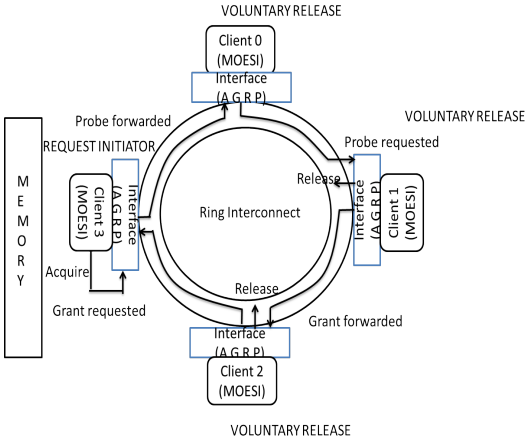


Figure 6.1: Ring Architecture Diagram: Voluntary Release

Table 6.1: Latency of Ring Architecture

Parameters	**Notations**	**TCC**	**BCC**	**UCC**
Acquire	AQ	2	1	1
Probe Forward	PF	2	2	2
Probe Request	PR	3	3	3
Search & Release	S& R	2	5	5
Grant Forward	GF	2	2	2
Grant Request	GR	3	2	2
Voluntary Release	VR	1	1	1
Memory In	MI	1	1	1
Memory Out	MO	1	2	2

TCC: Theoretically Calculated Clock Cycles
BCC: Bi-directional Ring Clock Cycles
UCC: Uni-directional Ring Clock Cycles

- **Case I. Data in Successive Client**: The latency would be minimum for uni-directional bi-directional ring when the hit is received in the successive client as it reduces the clock cycle of search. Whereas for bi-directional ring it comes out to be even one cycle less than that of uni-directional.

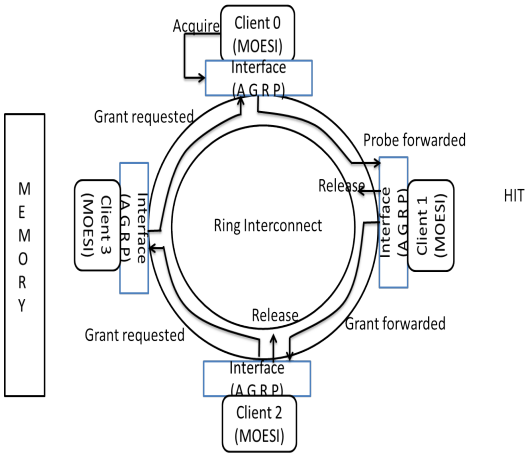


Figure 6.2: Uni-Directional Architecture Diagram: Minimum Latency

Uni-directional Theoretical Latency = AQ + PF + GF + (GRx 2) + SR
= 2 + 2 + 2 + (3 X 2) + 2
= 14 clock cycle / 17 clock cycle

Bi-directional Theoretical Latency = AQ + PF + PR + (SRx 2) + GF
= 2 + 2 + 3 + (2 X 2) + 2
= 13 clock cycle / 16 clock cycle

Bi-directional Design Latency = AQ + PF + PR + SR + GF
= 1 + 2(Min) + 3(Min) + 5 + 2
= 14 clock cycle

Uni-directional Design Latency = AQ + PF + GF + GR + SR
= 19 clock cycle

- **Case II. Data in Main Memory**:

  The latency would be maximum for the uni-directional ring when the hit is received in the main memory. It takes the clock cycle of search and release.Whereas latency would reduce drastically for bi-directional ring because it processed the request parallely to the clients. Hence takes less number of clock cycle for search and release.

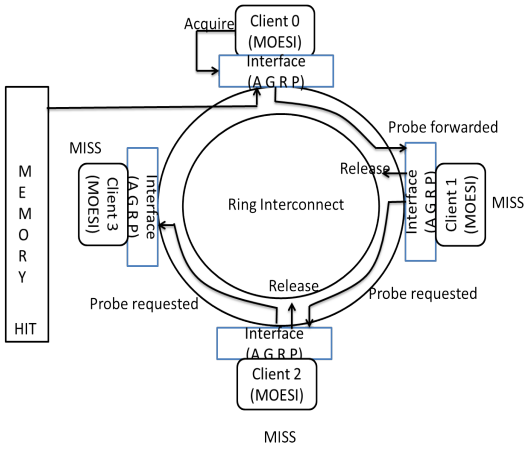


Figure 6.3: Uni-Directional Architecture Diagram: Maximum Latency

Uni-directional Theoretical Latency = AQ + PF + (PR X 2) + MO + MI + (SR X 3)
= 2 + 2 + (3 X 2) + 1 + 1 + (2 X 3)
= 18 clock cycle / 21 clock cycle

Bi-directional Theoretical Latency = AQ + PF + PR + MO + MI + (SR X 2)
= 2 + 2 + 3 + 1 + 1 + (2 X 2)
= 13 clock cycle / 16 clock cycle

Bi-directional Design Latency = AQ + PF + PR + MO + MI + SR + Memory to Client
= 1 + 3(Max) + 7(Max) + 2 + 1 + 5 + 3
= 22 clock cycle

Uni-directional Design Latency = AQ + PF + PR + MO + MI + SR
= 31 clock cycle

- **Case III. Data in Last Cache**: The latency comes out to be equivalent ot maximum latency case when data is found in last cache of the uni-directional ring architecture. Whereas its minimum for bi-directional ring architecture.

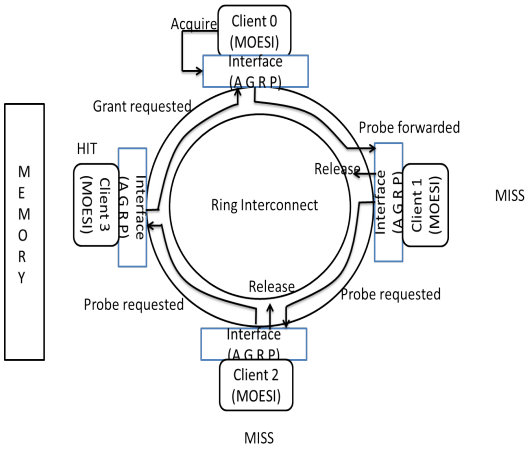


Figure 6.4: Uni-Directional Architecture Diagram: Data in Last Cache

Uni-directional Theoretical Latency = AQ + PF + (PR X 2) + GF + (SR X 3)
= 2 + 2 + (3 X 2) + 2 + (2 X 3)
= 18 clock cycle / 21 clock cycle

Bi-directional Theoretical Latency = AQ + PF + PR + GF + (SR X 2)
= 2 + 2 + 3 + 2 + (2 X 2)
= 13 clock cycle / 16 clock cycle

Bi-directional Design Latency = AQ + PF + PR + SR + GF
= 1 + 2(Min) + 3(Min) + 5 + 2
= 14 clock cycle

Uni-directional Design Latency = AQ + PF + PR + GF + SR
= 30 clock cycle

- **Case IV. Data in Second Cache**:

  The latency when data is found in second cache of the ring architecture. All cases arise once for the ring. Latency comes out to be maximum for bi-directional ring when data is found in second cache of the ring architecture as all cases have occured.

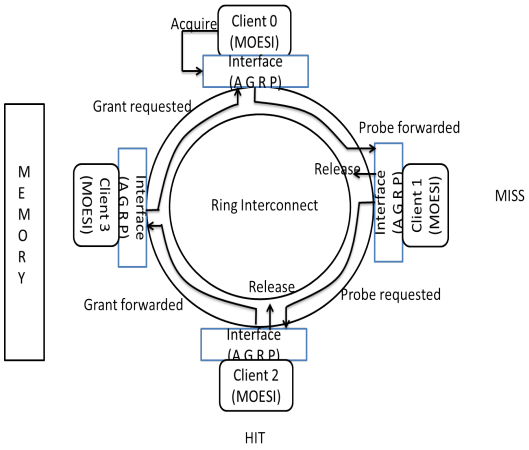


Figure 6.5: Uni-Directional Architecture Diagram: Data in Second Cache

Uni-directional Theoretical Latency = AQ + PF + PR + GF + GR+ (SR X 2)
= 2 + 2 + 3 + 2 + 3 + (2 X 2)
= 16 clock cycle / 19 clock cycle

Bi-directional Theoretical Latency = AQ + PF + PR + (SR X 2) + GF + GR
= 2 + 2 + 3 + (2 X 2) + 2 + 3
= 16 clock cycle / 19 clock cycle

Bi-directional Design Latency = AQ + PF + SR(Min) + PR + SR(Max) + GF + GR
= 2 + 2 + 5(Min) + 3 + 8(Max) + 2 + 3
= 25 clock cycle

Uni-directional Design Latency = AQ + PF + PR + GF + GR+ SR
= 23 clock cycle

To summarise the results for the afore mentioned cases it has been described in a tabular form. These results include the number of clock cycles consumed for Search and the cycles for which the ring waits for Release signal from the respective client for which Search signal has been sent.

Table 6.2: Latency Test Results in Clock Cycles

Scenario	UTL	BTL	UDL	BDL	Remarks
Case I	14 / 17	13 / 16	19	14	Best
Case II	18 / 21	13 / 16	31	22	Worst
Case III	18 / 21	13 / 16	30	14	Random
Case IV	16 / 19	16 / 19	23	25	All Signals

UTL: Uni-directional Theoretical Latency
BTL: Bi-directional Theoretical Latency
UDL: Uni-directional Design Latency
BDL: Bi-directional Design Latency

6.2 Average Network Latency

The average network latency calculation were carried out for uni-directional, bi-directional and ring of rings.

Unidirectional = (Best + Worst) / 2

Bidirectional = (B + SW) / 2

Ring of Rings = (2/16) x B + (1/16) x SW + (8/16) x HB + (4/16) x HW

Table 6.3: Ring Latency in Hops & Cycles

Design	Hops	Cycles	Remarks
Uni-directional (4-core)	13	5	Best
Uni-directional (4-core)	11	6	Worst
Uni-directional (4-core)	12	5.5	Average
Bi-directional (4-core)	6	4	B
Bi-directional (4-core)	8	6	SW
Bi-directional (4-core)	7	5	Average
Ring of Rings (16-core)	19	14	HB
Ring of Rings (16-core)	28	20	HW
Ring of Rings (16-core)	17.75	12.875	Average

6.3 Synthesis Results

The Vivado Synthesis results have shown the following results for the bi-directional as well uni-directional ring with four clients being initiated for an implementation of coherency protocol. The synthesis has been performed on Vertex Ultra Scale VCU 108 Evaluation Platform. Vivado synthesis shows that the clock frequency is same for bi-directional as well as uni-directional ring and number of LUT's are less for uni-directional ring rather than bi-directional ring but the relative reduction in clock cycles and hops with in the ring are more significant.

Table 6.4: Synthesis Results for Timing, Power and LUT

Parameters	Bi-directional Ring	Uni-directional Ring
CLK Frequency (MHz)	204.082	204.082
CLK Period (ns)	4.9	4.9
Total on Chip Power (Watt)	1.032	0.993
Dynamic Power (Watt)	0.123	0.085
Static Power (Watt)	0.908	0.908
LUT	14539	14465
LUTRAM	7936	7936
FIFO	5217	5204
BRAM	8	8
IO	73	73
BUFG	1	1

6.4 Design Challenges

The design is based on Client module of MCP. The client module deals with the ring through four signals in order to maintain cache coherency (A, G, P, R). Few typical challenges faced during implementation and optimisations done are as under:-

- **Handling Write-back/Invalidate Request**. The Write-back/Invalidate requests arrives at the clients via *Release* (R) signal hence it had a probability to arrive in between of an event or request service by the ring. However the request was handled by incorporating *in-ring buffering mechanism* with in ring architecture in order to resume the process.

- **Inevitable Timing Loss**.This is specific to bi-directional ring architecture with the quad-core implementation. When a request is generated and forwarded to adjacent clients. If one of the Client'x' has data and second Client'y' does not have a data. There architecture encounters an issue as '*Hit*' is always received post'*Miss*' signal has arrived as *Release*, hence the Client'y' forwards *Probe* signal to next Client'z'. Once Probe is forwarded ring reads data sent by Client'x' and forwards as a Grant to the initiator in a bi-directional format and the request has been service but the *Release* is pending from Client'z'. So to overcome a limitation a mechanism was designed that once the request is serviced the

ring would wait for all the responses pending based on request sent outward and would Kill the unwanted Release before allowing resources for next request. This lead to time penalty.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The underlying architecture of Ring Based Distributed Shared Memory multiprocessors is an effort towards generating the multifaceted ring architecture for of the flagship SHAKTI program of RISE Lab. Simulation of the output using test bench and verification by linking the compiled code with the simulation environment has been achieved. Apart from carrying out literature survey on ring structure system which included the unidirectional and bidirectional ring architecture, Distributed Shared Memory, theoretical latency, design latency and average network latency for the architecture has been undertaken. The comparative study for the utilization of uni-directional / bi-directional ring architecture is done in terms of increase in hardware and reduction in latency for optimum performance based on requirement. The scope of the design was extended to the implementation of **Ring of Ring** in a hierarchical manner for implementing 16-cores in a ring fashion of 4-core each ring.

7.1.1 Ring Square

Ring Square (Ring of Rings) has been implemented as a modification to MCP. The lower ring has four clients connected and a manager to send the request to the global ring connecting the lower rings. It has following salient features:-

- Exploits both the Manager as well as Client module of MCP.
- Modified to use only client modules for snoopy based protocol with lower ring.Utilises manager module for snoopy based protocol with global ring.
- In-ring buffering is implemented not only for Write-Back, Invalidation request but also for request arriving to global ring and are serviced on highest priority to avoid deadlock.

7.1.2 Bi-directional Ring

Bi-directional Ring implemented as a modification to MCP has following salient features:-

- Inspired by the Manager-Client Pairing protocol.
- Modified to use only client modules for snoopy based protocol.

- Client is used to maintain the MOESI coherency protocol.
- Ring controller mechanism is implicitly implemented using BSVâs atomic rule structures.
- Part of same SoC to maintain coherency for shared memory based Processors.
- Ring prioritizes request service to clients in clockwise direction and keep on snooping till the time request is not generated.
- The Read/Write is served on priority than Write-back and Invalidation.
- In-ring buffering requests are serviced in a round-robin mechanism.
- User Interface is through TLMs.
- VCU108 Vivado Synthesis:

  Frequency - 204MHz (4 Client)

  Area - < 15K LUTs

7.1.3 Uni-directional Ring

Uni-directional Ring implemented as modification to MCP has following salient features:-

- Inspired by the Manager-Client Pairing protocol.
- Modified to use only client modules for snoopy based protocol.
- Client is used to maintain the MOESI coherency protocol.
- Ring controller mechanism is implicitly implemented using BSVâs atomic rule structures.
- Part of same SoC to maintain coherency for shared memory based Processors.
- Ring prioritizes request service to clients in clockwise direction and keep on snooping till the time request is not generated.
- The Read/Write is served on priority than Write-back and Invalidation.
- In-ring buffering requests are serviced in a round-robin mechanism.
- User Interface is through TLMs.
- VCU108 Vivado Synthesis:

  Frequency - 204MHz (4 Client)

  Area - < 15K LUTs

7.2 Future Work

The project might be taken as an extension towards the implementation of same for the higher hierarchical structure keeping the same level of data encapsulation i.e Designing of MCP for Rings, Ring of Rings, Mesh of Rings. The design methodology could be chosen based on the application, area constrains in term of placement and routing, power consumption, throughput etc.

7.2.1 MCP of Rings

Ring is connected with clients and a bridge network could be formed among the rings using Manager of the MCP protocol. Manager forwards the request to other rings. The methodology helps in implementing sixteen in number clients with each ring having 8-cores attached to it. A comparative study for the *MCP of Ring* methodology and *Ring of Ring* methodology could be made so as to find analogy of which methodology would serve the purpose base on requirement specification.

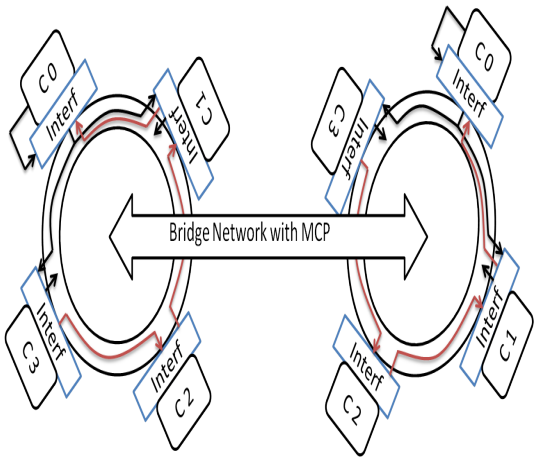


Figure 7.1: MCP of Rings : Interconnect of Rings with Manager Module

7.2.2 Ring Square

Ring square is also known as Ring of Ring's, Clients are connected via ring and ring's are connected via a higher level ring structure. The methodology could be beneficiary for designing processors for server based applications, with total number of cores equal to sixteen i.e four core on each ring. The higher level ring/global ring is connected to main memory and lower ring is connected in a star methodology with the higher/global ring. A continuously injection of the traffic on the higher ring is done via the lower rings Ring0, Ring1, Ring2, Ring3. The critical issue arises when lower Ring0 and Ring3 injects the traffic to higher level and data of Ring0 is available at Ring3 so it might lead to live-lock deadlock situation. Hence to eliminate these issues we connect lower order ring in a star methodology with the higher/global ring.

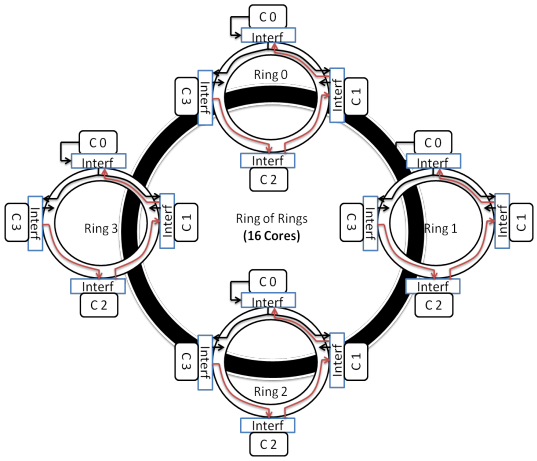


Figure 7.2: Ring of Ring Diagram: Implementation of R2

7.2.3 Mesh of Rings

This methodology can be implemented for processors more than 32 in number. The rings are connected in form of mesh structure, each mesh node is connected in form of ring with clients. The methodology preferred for higher throughput is priority and time delay is trade off.

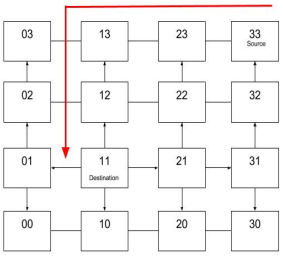


Figure 7.3: Mesh of Ring Diagram: Implementation of Hybrid

Bibliography

[1] REENA E. Implementing Cache Coherence through Manager Client Pairing. Master's thesis, INDIAN INSTITUTE OF TECHNOLOGY MADRAS, 2016.

[2] Thomas M. Conte Jesse G.Beu, Michael C. Rosier. Manager-Client Pairing: A Framework for Implementing Coherence Hierarchie , journal = MICRO-44 Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, year = 2011.

[3] Michael R. Marty and Mark D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. *ACM Symposium on Microarchitecture*, 2006.

[4] Xiangyao Yo Kevin Kai Wei Chang Greg Nazario Reetuparana Das Gabriel H.Loh Rachata Ausavarungnirun, Chris Fallin and Onur Mutlu. A case for hierarchical rings with deflection routing: An energy-efficient on-chip communication substrate. *Elsevie - Parallel Computing*, 2016.