Implementation of Tilelink Protocol for Single Core Processor

A Project Report

submitted by

ADAPA JANARDHANA SWAMY

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.
April 2016

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementation of Tilelink Protocol for Single Core Processor**, submitted by **Adapa Janardhana Swamy**, 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 carried out by him under my 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.

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

Place: Chennai

Date: **6th May, 2016**

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.

First and foremost, I offer my earnest gratitude to my guide, Dr. V. Kamakoti whose knowledge and dedication has inspired me to work efficiently on the project and I thank him for motivating me, and allowing me freedom and flexibility while working on the project.

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 in my project work.

My special thanks and deepest gratitude to Rahul Bodduna and Abhinaya Agrawal who have been very supportive. They have enriched the project experience with their active participation and invaluable suggestions.

ABSTRACT

With the increasing number of cores per processor, the complexity of designing and implementing the caches and their coherence is also increasing. So, this project is based on the "Tilelink" protocol which will abstract the cache coherence transactions between various levels of caches in the memory hierarchy. This helps in replicating the same implementation for different coherence domains in a large memory hierarchy. Tilelink protocol is based on Manager-Client Pairing (MCP), MCP defines a clear communication interface between users of data (clients) and the mechanisms that monitor coherence of these users (managers). MCP provides the encapsulation within each tier of the hierarchical protocol so each component coherence protocol can be considered in isolation. MCP can be applied in a divide-and-conquer manner to partition a manycore processor into arbitrarily deep hierarchies.

This thesis describes the design and implementation of "Tilelink protocol for single-core processor". This work involves implementing this Tilelink protocol between L1ICache, L1DCache and L2ICache, L2DCache respectively of the I-class processor of Shakti processor series, which is based on RISC-V ISA. The code for the entire project is written in a HDL namely Bluespec System Verilog (BSV). TLM data structures and interfaces of BSV are used in implementing this protocol.

KEYWORDS: Tilelink, Cache, Cache Coherence, Single Core Processor, Manager Client Pairing

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BSV	Bluespec System Verilog
MCP	Manager-Client pairing
TLM	Transaction Level Modelling
FIFO	First In First Out
RISC	Reduced Instruction Set Computer
BRAM	Block Random Access Memory
HDL	Hardware Description Language
CPU	Central Processing Unit

CHAPTER 1

Introduction

1.1 Overview

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 research of The SHAKTI Processor project. It aims to build 6 variants of processor (e.g.C-Class, I-Class, M-Class,etc.) based on the RISC-V ISA. My project targets I-Class processor which is 64-bit, 1-4 core and 5-8 stage out of order processor. This processor aims at 200-1Ghz industrial control / general purpose 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). This project describes the design and implementation of "Tilelink protocol of single-core processor". This work involves implementing this Tilelink protocol between L1ICache, L1DCache and L2ICache, L2DCache respectively of the I-class processor which is based on RISC-V ISA. Tilelink protocol is customised for Instruction caches and Data caches in this project.

1.2 Organisation of thesis

Chapter 2 deals with architecture of Cache and Cache coherence which is the basis for the rest of the thesis. It also gives some insight about the Bluespec System

Verilog, its key features, TLM module of BSV.

Chapter 3 describes the architecture of Manager-Client Pairing, this chapter discusses a standardized coherence communication interface between managers and clients and permissions checking algorithm of MCP.

Chapter 4 discusses the specifications of Tilelink protocol - their channels, transaction flow of the signals and the metadata of the clients and managers.

Chapter 5 gives us implementation perspective of the project. It explains in detail how each block of the interface is implemented in Bluespec System Verilog(BSV).

Chapter 6 contains a conclusion and description on the future work.

CHAPTER 2

Background

2.1 Bluespec System Verilog

The design of the blocks and their testing is written in Bluespec SystemVerilog (BSV). BSV is a high level Hardware Description Language. It expresses synthesizable behavior with rules, a rule can be viewed as a declarative assertion expressing a potential atomic state transition. The BSV compiler produces efficient RTL code that manages all the potential interactions between rules by inserting appropriate arbitration and scheduling logic, logic that would otherwise have to be designed and coded manually. BSV connects the modules by interfaces and methods. It also provides predefined library elements like FIFOs, BRAMs etc. which are modeled using BSV methods.

It has powerful static type checking which removes potential human errors which can't be detected at the stage of compilation normally but can be detected now during the compilation. BSV also has more general type parameterization (polymorphism) due to which modules and functions can be parameterised by other modules and functions, this enables the designer to reuse designs and glue them together in much more flexible ways. BSV's static elaboration helps to arrive at the design much faster than the other HDLs. The BSV compiler also can generate the synthesizable verilog code of the written bluespec code which can be used later for synthesis purposes.

BSV has an inbuilt package called TLM (Transaction Level Modeling) which is used in this thesis and explained in the next part of this chapter.

2.1.1 TLM

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. In addition, since the specific signaling details of each bus protocol are encapsulated in pre-designed transactors, users are not required to learn, re-implement, and re-verify existing standard protocols.

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. 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. Table 2.1 describes the components of a RequestDescriptor and the valid values for each of its members.

Table 2.2 describes the components of a TLMResponse and the valid values for its members.

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 subinterfaces as the requests and responses. The TLMSendIFC interface generates (Get) requests

Table 2.1: Request Descriptor

Member Name	Valid Values
command	READ, WRITE, UNKNOWN
mode	REGULAR, DEBUG, CONTROL, UNKNOWN
addr	Bit#(addr_size)
data	Bit#(data_size)
burst_length	UInt#(uint_size)
byte_enable	Bit#(TDiv#(data_size, 8))
burst_mode	INCR, WRAP, CNST, UNKNOWN
burst_size	Bit#(TLog#(TDiv#(data_size, 8)))
prty	UInt#(uint_size)
lock	True, False
thread_id	Bit#(id_size)
transaction_id	Bit#(id_size)
export_id	Bit#(id_size)
custom	cstm_type

Table 2.2: TLMResponse

Member Name	Valid Values
command	READ, WRITE, UNKNOWN
data	Bit#(data_size)
status	SUCCESS, ERROR, NO RESPONSE, UNKNOWN
prty	UInt#(uint_size)
thread_id	Bit#(id_size)
transaction_id	Bit#(id_size)
export_id	Bit#(id_size)
custom	cstm_type

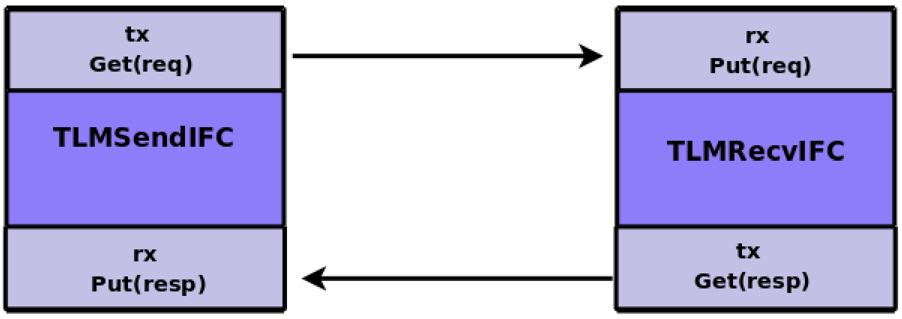


Figure 2.1: Connecting TLM Send And Receive Interfaces

and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests and generates (Get) responses. These TLMSendIFC and TLMRecvIFC can be connected by mkConnection in the Connectable package of BSV.

The Data Structures Request Descriptor in TLMRequest, TLMResponse, and the interfaces provided TLMSendIFC and TLMRecvIFC are used extensively in this thesis.

2.2 Cache

Cache is a component used by central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Data is transferred between memory and cache in blocks of fixed size, called cache lines. When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

CPU may have two independent caches, instruction and data caches, an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store. These caches can be organized as a hierarchy of more cache levels (L1, L2, etc.).

A L1 cache is a memory cache that is directly built into the processor, which is used for storing the processor's recently accessed information, thus it is also called the primary cache. L1 cache is the fastest cache memory, since it is already built within the chip with a zero wait-state interface, making it the most expensive cache among the CPU caches. However, it has limited size. It is used to store data that was accessed by the processor recently, it is the first cache to be accessed and processed when the processor itself performs a computer instruction. It is implemented with the use of static random access memory (SRAM), it makes use of two transistors per bit. The two transistors form a circuit known as a 'flip-flop' since it has two states it can flip between; the second transistor manages the output of the first transistor. For as long as power is supplied to the circuit, it can hold data without external assistance. The L2 cache feeds the L1 cache, which feeds the processor. It uses the same control logic as Level 1 cache and is also implemented in SRAM. L2 cache is slower than L1 but much larger than it.

If the cache is fully associative, it means that any block of RAM data can be stored in any block of cache. The advantage of such a system is that the hit rate is very high, but the search time is extremely long — the CPU has to look through its entire cache to find out if the data is present before searching main memory. At the opposite end of the spectrum we have direct-mapped caches. A direct-mapped cache is a cache where each cache block can contain one and only one block of main memory. This type of cache can be searched extremely quickly, but since it maps 1:1 to memory locations, it has a low hit rate. In between these two extremes are n-way associative caches. A 2-way associative cache means that each main memory block can map to one of two cache blocks. An eight-way associative cache means that each block of main memory could be in one of eight cache blocks.

Larger caches have better hit rates but longer latency. To address this trade off, many computers use multiple levels of cache, with small fast caches backed up by larger, slower caches. Multi-level caches generally operate by checking the fastest, level 1 (L1) cache first; if it hits, the processor proceeds at high speed. If that smaller cache misses, the next fastest cache (level 2, L2) is checked, and so on, before accessing external memory.

2.2.1 Cache coherence

Cache coherence is the consistency of shared resource data that ends up stored in multiple local caches. Coherence defines the behavior of reads and writes to the same memory location. In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

The coherence of caches is obtained if the following conditions are met:

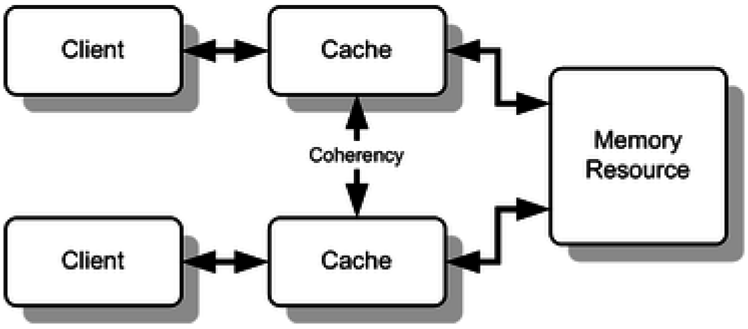


Figure 2.2: multiples caches sharing the same memory source

i) In a read made by a processor P to a location X that follows a write by the same processor P to X, with no writes to X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P. This condition is related with the program order preservation, and this must be achieved even in single-core architectures.

ii) A read made by a processor P1 to location X that happens after a write by another processor P2 to X must return the written value made by P2 if no other writes to X made by any processor occur between the two accesses and the read and write are sufficiently separated. This condition defines the concept of coherent view of memory. If processors can read the same old value after the write made by P2, we can say that the memory is incoherent.

iii) Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, from any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

Cache Coherency protocol is the protocol that maintains memory coherence according to a specific consistency model. Various coherency protocols are: MSI, MESI, MOSI, MOESI. In the protocols every cache line is assigned to one of the following states:

a) Modified: The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Shared state.

b) Exclusive: The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.

c) Owned: This cache is one of several with a valid copy of the cache line, but has the exclusive right to make changes to it. It must broadcast those changes to all other caches sharing the line. The introduction of owned state allows dirty sharing of data, i.e., a modified cache block can be moved around various caches without updating main memory. The cache line may be changed to the Modified state after invalidating all shared copies, or changed to the Shared state by writing the modifications back to main memory. Owned cache lines must respond to a snoop request with data.

d) Shared: This line is one of several copies in the system. This cache does not have permission to modify the copy. Other processors in the system may hold copies of the data in the Shared state, as well. The cache line may not be written, but may be changed to the Exclusive or Modified state after invalidating all shared copies. It may also be changed to the Invalid state at any time.

e) Invalid: Indicates that this cache line is invalid.

CHAPTER 3

Manager-Client pairing

Manager-Client Pairing enables coherence hierarchies to be constructed and evaluated quickly without the high design-cost previously associated with hierarchical composition. Manager-Client defines a standardized coherence communication interface and permissions checking algorithm, MCP defines a clear communication interface between users of data (clients) and the mechanisms that monitor coherence of these users (managers). MCP provides encapsulation within each tier of the hierarchical protocol so each component coherence protocol can be considered in isolation. MCP can be applied in a divide-and-conquer manner to partition a many core processor into arbitrarily deep hierarchies.

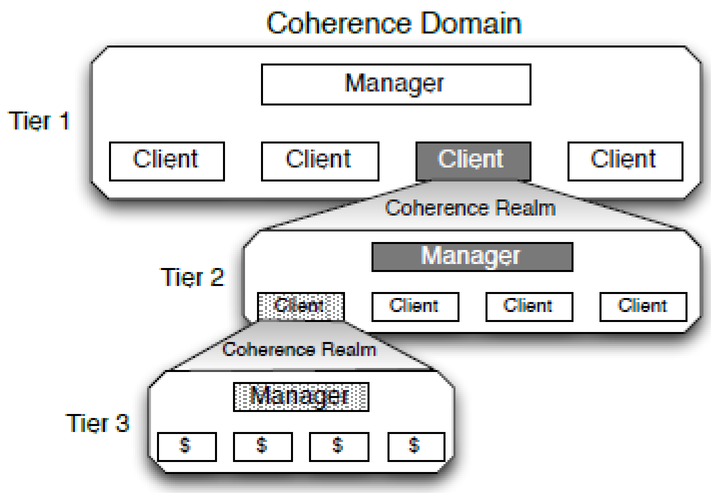
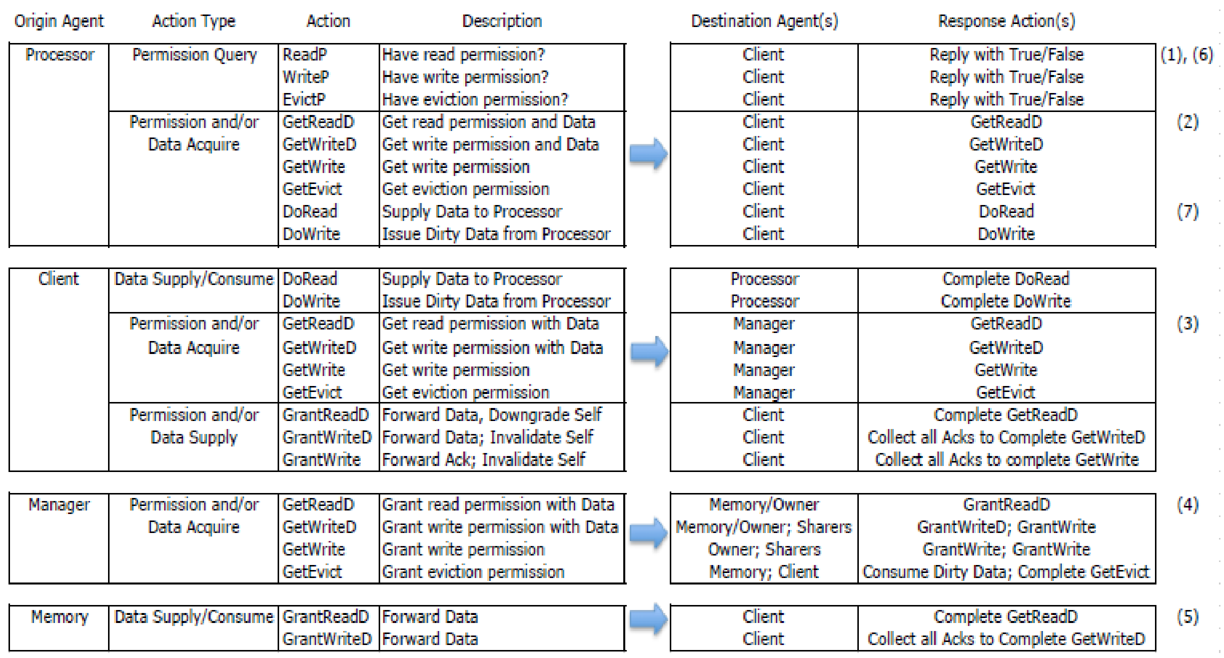


Figure 3.1: Coherence hierarchy labeled with MCP terminology



Origin Agent	Action Type	Action	Description	Destination Agent(s)	Response Action(s)	
Processor	Permission Query	ReadP	Have read permission?	Client	Reply with True/False	(1), (6)
		WriteP	Have write permission?	Client	Reply with True/False	
		EvictP	Have eviction permission?	Client	Reply with True/False	
	Permission and/or Data Acquire	GetReadD	Get read permission and Data	Client	GetReadD	(2)
		GetWriteD	Get write permission and Data	Client	GetWriteD	
		GetWrite	Get write permission	Client	GetWrite	
		GetEvict	Get eviction permission	Client	GetEvict	
		DoRead	Supply Data to Processor	Client	DoRead	(7)
		DoWrite	Issue Dirty Data from Processor	Client	DoWrite	
Client	Data Supply/Consume	DoRead	Supply Data to Processor	Processor	Complete DoRead	
		DoWrite	Issue Dirty Data from Processor	Processor	Complete DoWrite	
	Permission and/or Data Acquire	GetReadD	Get read permission with Data	Manager	GetReadD	(3)
		GetWriteD	Get write permission with Data	Manager	GetWriteD	
		GetWrite	Get write permission	Manager	GetWrite	
		GetEvict	Get eviction permission	Manager	GetEvict	
	Permission and/or Data Supply	GrantReadD	Forward Data, Downgrade Self	Client	Complete GetReadD	
		GrantWriteD	Forward Data; Invalidate Self	Client	Collect all Acks to Complete GetWriteD	
		GrantWrite	Forward Ack; Invalidate Self	Client	Collect all Acks to complete GetWrite	
Manager	Permission and/or Data Acquire	GetReadD	Grant read permission with Data	Memory/Owner	GrantReadD	(4)
		GetWriteD	Grant write permission with Data	Memory/Owner; Sharers	GrantWriteD; GrantWrite	
		GetWrite	Grant write permission	Owner; Sharers	GrantWrite; GrantWrite	
		GetEvict	Grant eviction permission	Memory; Client	Consume Dirty Data; Complete GetEvict	
Memory	Data Supply/Consume	GrantReadD	Forward Data	Client	Complete GetReadD	(5)
		GrantWriteD	Forward Data	Client	Collect all Acks to Complete GetWriteD	

Figure 3.2: Base functions for standardized communication between processors, clients and managers

3.1 Defining Base functions

In a shared memory machine, the cache coherence protocol is responsible for enforcing a consistent view of memory across all caches within a coherence domain. This includes defining the mechanisms that control acquisition and holding of read permissions, write permissions, the respective restrictions on each, and how updates to data are propagated through the system. The responsibilities of this effort can be divided between two kinds of agents: managers that manage permission propagation and clients that hold these permissions. Figure 3.2 summarizes and enumerates a comprehensive list of the base functions required for communication between processors, clients, managers and memory in a flat protocol.

An example of how a read sequence would operate on an invalid block-

i) First a Processor issues a ReadP to its client. This client replies with 'false'.

ii) Processor takes another action, GetReadD. This results in the client executing its GetReadD action, which in turn will cause the Manager to execute its GetReadD action.

iii) The Manager GetReadD action is a forward request. Assuming there is no client owner, Memory is regarded as the owner of the data and asked to execute its GrantReadD action. This results in Memory supplying Data to the client, completing the GetReadD.

iv) Upon completion, the processor can retry its ReadP action, which the client will respond with 'true'.

v) The processor can safely execute its DoRead action for which the client will supply data.

3.2 Coherence Hierarchy Construction

The relationship between processor and client agents has similarities to that between manager agents and memory. If manager agents were given the ability to issue permissions-query upwards like processors do towards their client, then replacing the implementation details of the coherence protocol with a black box yields a self-similar upper and lower interface. Not only does this insight enable recursion through a simple interface definition, but also allows encapsulation of the coherence protocols used in the hierarchy, reducing design complexity. Manager queries are accomplished by pairing the manager agent of each coherence realm in a tier with a client in the next higher-up tier in the hierarchy (or an all-permission client if there is no higher tier, e.g., memory). Since there is one logical manager agent per coherence realm, this allows the client to represent the permissions of the entire realm and all tiers beneath this realm. Permission/Data acquisition and supply are also possible due to the pairing of managers with clients in the next tier. The manager requires no details regarding the operation of the higher coherence protocol provided, it can defer that responsibility to it's paired client. By systematically asking the paired client for either read or write permission, the client can take part in its native coherence scheme until it has completed the request. This is much like how a processor is unaware of how coherence in the caches are implemented; it simply asks if it has permissions and receives data.

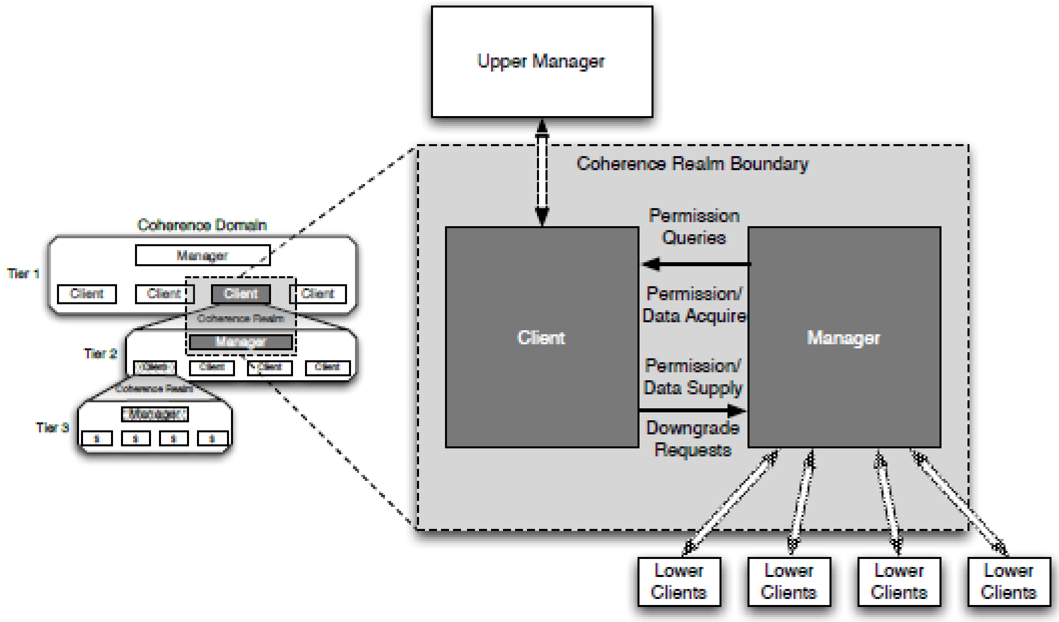


Figure 3.3: Manager-Client Pairing and associated interfaces to preserve encapsulation

3.3 Permission Hierarchy Algorithm

the client agent must behave as a gateway for the manager of the coherence realm, restricting what permissions can be awarded, and taking action when permissions must be upgraded in the coherence realm before the manager can begin request resolution. The manager agents now must consult the gateway client before allocating permission, which in turn may recursively send another permission request to another manager-client pair. Figure 3.4 demonstrates the Manager- Client Pairing algorithm for processing permission acquires.

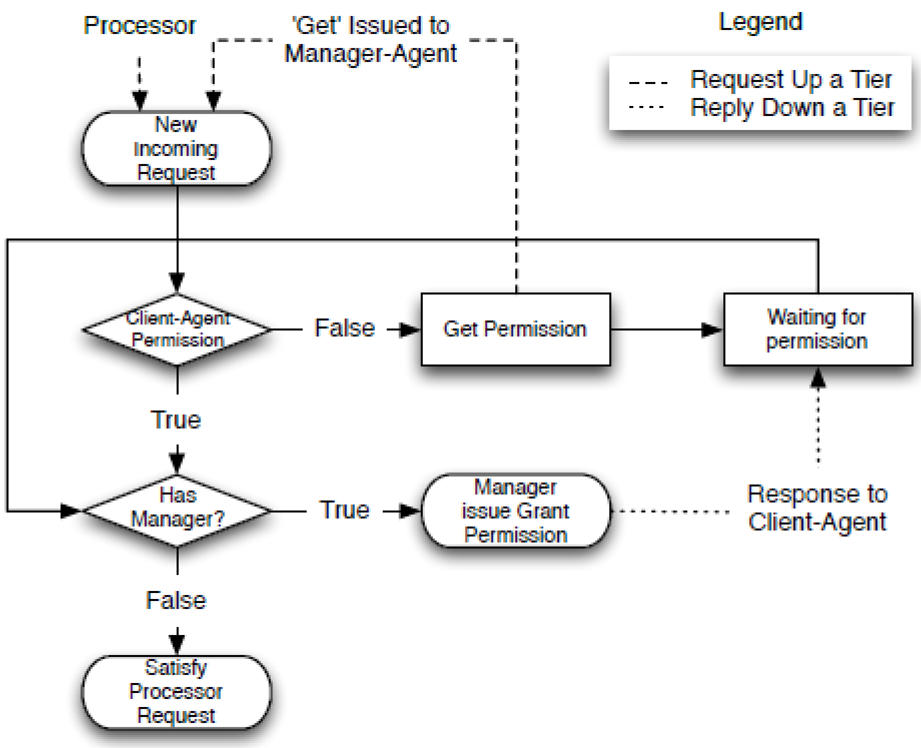


Figure 3.4: Coherence hierarchy permission checking algorithm

CHAPTER 4

Tilelink

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. It assumes Manager-Client pairing Architecture.

The participating agents in this coherence protocol are:

i) clients requesting access to cache blocks.

ii) managers overseeing the propagation of cache block permissions and data.

A client may be a cache, a DMA engine, etc. A manager may be an outer-level cache controller, a directory, or a broadcast medium such as a bus. In a multi-level memory hierarchy, a particular cache controller can function as both a client (wrt. caches further out in the hierarchy) and a manager (wrt. caches closer to the processors).

4.1 Channels

TileLink defines five independent transaction channels. Channels may contain both metadata and data components. The channels are:

a) **Acquire**: Initiates a transaction to acquire access to a cache block with proper permissions. Also used to write data without caching it.

Table 4.1: Acquire signals

Signal Name	Description
addr_block	Physical address of the cache block, with block offset removed
client_xact_id	Client's id for the transaction
data	Client-sent data, used for Put transactions
a_type	Type of the transaction defined by coherence protocol

b) **Probe**: Queries a client to determine whether it has a cache block or revoke its permissions on that cache block.

Table 4.2: Probe signals

Signal Name	Description
addr_block	Physical address of the cache block, with block offset removed
p_type	Transaction type, defined by coherence protocol

c) **Release**: Acknowledgement of probe receipt, releasing permissions on the line along with any dirty data. Also used to voluntarily write back data.

Table 4.3: Release signals

Signal Name	Description
addr_block	Physical address of the cache block, with block offset removed
client_xact_id	Client's id for the transaction
data	Client-sent data, used for Put transactions
r_type	Transaction type, defined by coherence protocol

d) **Grant**: Provides data or permissions to the original requestor granting, access to the cache block. Also used to acknowledge voluntary Releases.

e) **Finish**: Final acknowledgement of transaction completion from requestor, used for transaction ordering.

These channels may be multiplexed over the same physical link, but to avoid deadlock TileLink specifies a priority amongst the channels that must be strictly enforced. The prioritization of channels is Finish >> Grant >> Release >> Probe >> Acquire. Preventing messages of a lower priority from blocking messages of a higher priority from being sent or received is necessary to avoid deadlock. When

Table 4.4: Grant signals

Signal Name	Description
client_xact_id	Client's id for the transaction
manager_xact_id	Manager's id for the transaction, passed to Finish
data	Client-sent data, used for Put transactions
g_type	Transaction type, defined by coherence protocol

Table 4.5: Finish signals

Signal Name	Description
manager_xact_id	Manager's id for the transaction, passed to Finish

running on networks that provide guaranteed ordering of messages between any client/manager pair, the Finish acknowledgment of a Grant can be omitted.

4.2 Transaction Flow

There are two types of transaction that can occur on a cache block managed by TileLink:

i) The first type enables clients to acquire a cache block: A client sends an Acquire to a manager The manager sends any necessary Probes to clients The manager waits to receive a Release for every Probe that was sent The manager communicates with backing memory if required Having obtained the required data or permissions, the manager responds to the original requestor with a Grant Upon receiving a Grant, the original client responds to the manager with a Finish to complete the transaction.

ii) The second type of transaction is supports clients voluntarily releasing a cache block: A client sends a Release to a manager, specifying that it is voluntary The manager communicates with backing memory if required The manager acknowledges completion of the transaction using a Grant.

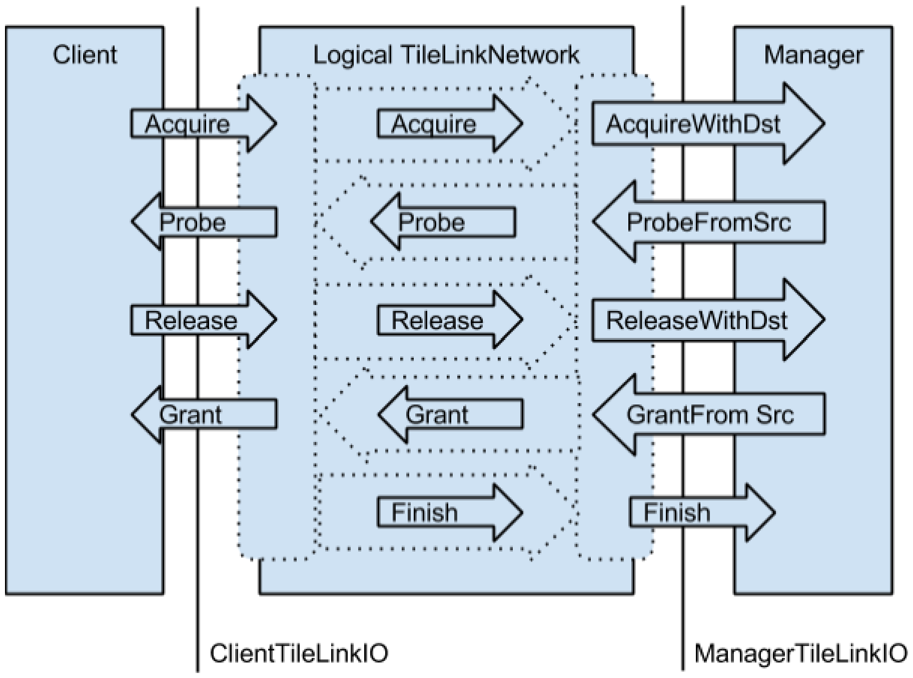


Figure 4.1: Tilelink Architecture

4.3 Metadata

Metadata are the opaque sets of bits which are processed and mutated by the coherence policy. Metadata are divided into client-side and manager-side classes, and any particular cache controller can store either or both types.

i) **ClientMetadata**:

ClientMetadata is a set of bits that abstracts the "state" of a certain cache block, w.r.t. the permissions available on that block inside this particular client cache controller. The metadata may also store other information about the cache block, for example whether it has been dirtied by a store operation. There are three types of calls that can be made against this metadata:

a) **Permissions checks**: Boolean functions answer questions about the permissions on a cache line.

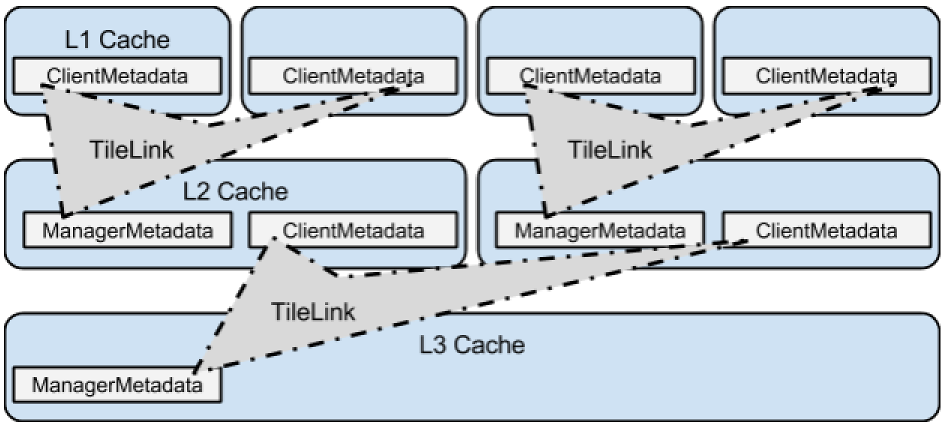


Figure 4.2: Metadata Hierarchy

b) **Message creations**: Functions return TileLink channel bundles based on the combination of current metadata state and particular memory operations.

c) **Metadata updates**: Functions return new ClientMetadata objects whose internal state has been updated based on a particular coherence event or message received.

ii) **ManagerMetadata**:

ManagerMetadata is a set of bits that abstracts the "state" of certain cache block, w.r.t. the existence of copies of that block in all the client caches managed by this manager cache controller.

a) **Permissions checks**: Boolean functions answer questions about the permissions on a cache line

b) **Message creations**: Functions return TileLink channel bundles to use as responses to Clients based on the combination of current metadata state and particular TileLink messages

c) **Metadata updates**: Functions return new ManagerMetadata objects whose internal state has been updated based on a particular coherence event or message

CHAPTER 5

Implementation

This chapter discusses the architecture of Tilelink protocol in single core I-class processor of Shakti series, and how it is implemented in Bluespec SystemVerilog. Let us discuss first about the architecture of the processor.

5.1 Architecture

CPU contains the processor, L1Cache, L1ICache, L2DCache, and L2ICache as represented in Figure 5.1. Here, SOC contains the processor and the L1, L2 caches, the L1ICache and L2ICache are connected by Tilelink_icache, and L1DCache and L2DCache are connected by the Tilelink_dcache. The tilelink_icache and tilelink_dcache are the variants of Tilelink protocol which are customised for their respective cache functions.

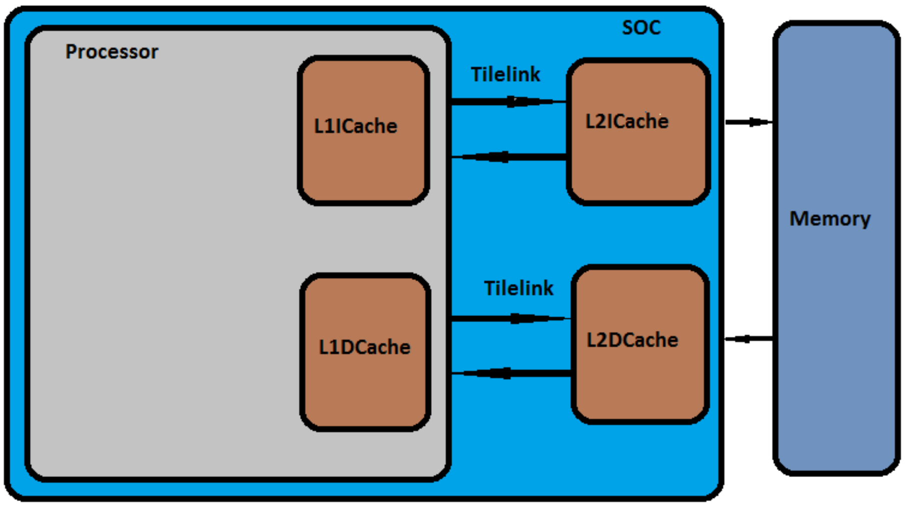


Figure 5.1: SOC Architecture

5.2 Tilelink Implementation

As represented in Figure 4.1, Tilelink contains a client, manager and Logical Tilelink Network modules. But as shown in Figure 5.2, Tilelink protocol for single core requires only two channels- Acquire and Grant. As there is only one L1Cache and one L2Cache, there is no need for channel Probe, Release, and in this implementation of the cache in I-class processor, the ordering of messages is maintained, so there is no need of Finish channel also. Here, To_L2 and From_L2 are the methods from L1Cache and From_L1 and To_L1 are the methods from L2Cache. The communication between Client module and Logical Tilelink Network module is made by TLM interfaces provided by TLM module in BSV, TLMSendIFC on the Client side interface and TLMRecvIFC on Logical Tilelink Network side. Similarly, the communication between manager module and Logical Tilelink Network module is made by TLMSendIFC on the Logical Tilelink Network side interface and TLMRecvIFC on Manager side.

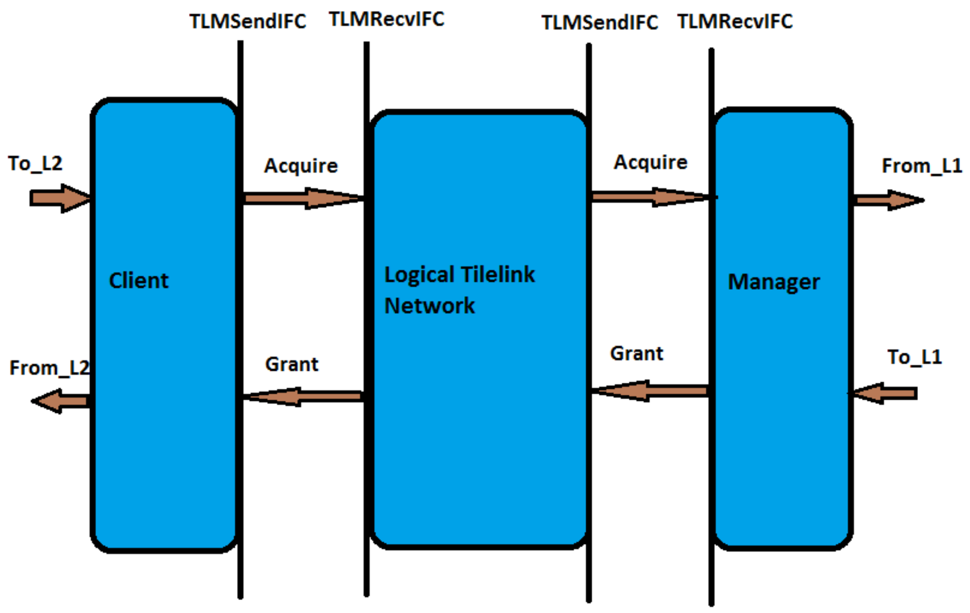


Figure 5.2: Tilelink Implementation

5.2.1 Client

The implementation of the Client module in the above Tilelink protocol is shown in Figure 5.3. The struct from the To_L2 method coming from L1Cache is converted into a acquire signal of tilelink protocol specifications, as a RequestDescriptor of TLMRequest which is provided by TLM module of BSV. The converted TLMRequest is enqueued into a fifo_send_acquire, a FIFO of BSV. The grant signal in fifo_recv_grant created using TLMResponse is converted into the required struct of the method From_L2. The FIFOs - fifo_send_acquire and fifo_recv_grant are connected to the TLMSendIFC using toGet() and toPut() respectively, the inbuilt functions provided by BSV.

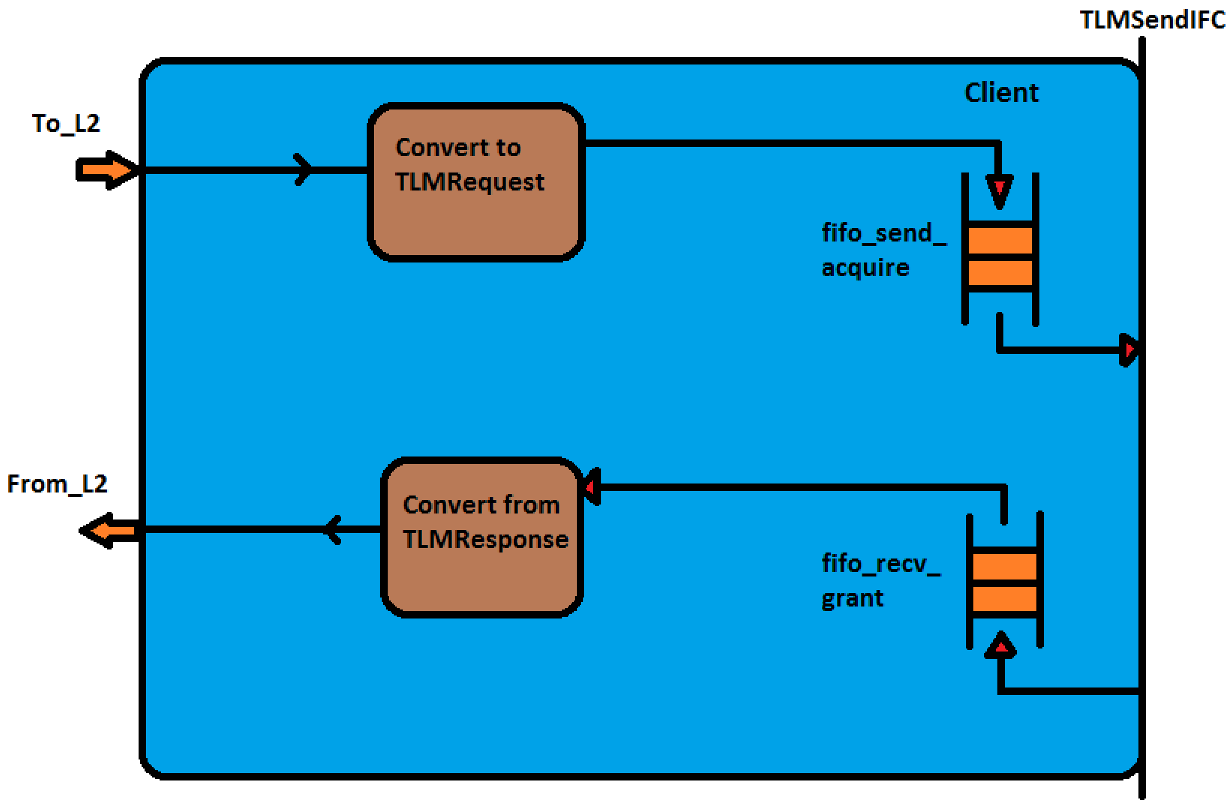


Figure 5.3: Client Implementation

5.2.2 Manager

The implementaion of the manager module in the above Tilelink protocol is shown in Figure 5.4. The acquire signal in fifo_recv_acquire created using TLMRequest is converted into the required struct of the method From_L1. The struct from the To_L1 method coming from L2Cache is converted into a grant signal of tilelink protocol specifications, as a TLMResponse which is provided by TLM module of BSV. The converted TLMResponse is enqueued into a fifo_send_grant, a FIFO of BSV. The FIFOs - fifo_recv_acquire and fifo_send_grant are connected to the TLMRecvIFC using toPut() and toGet() respectively, the inbuilt functions provided by BSV.

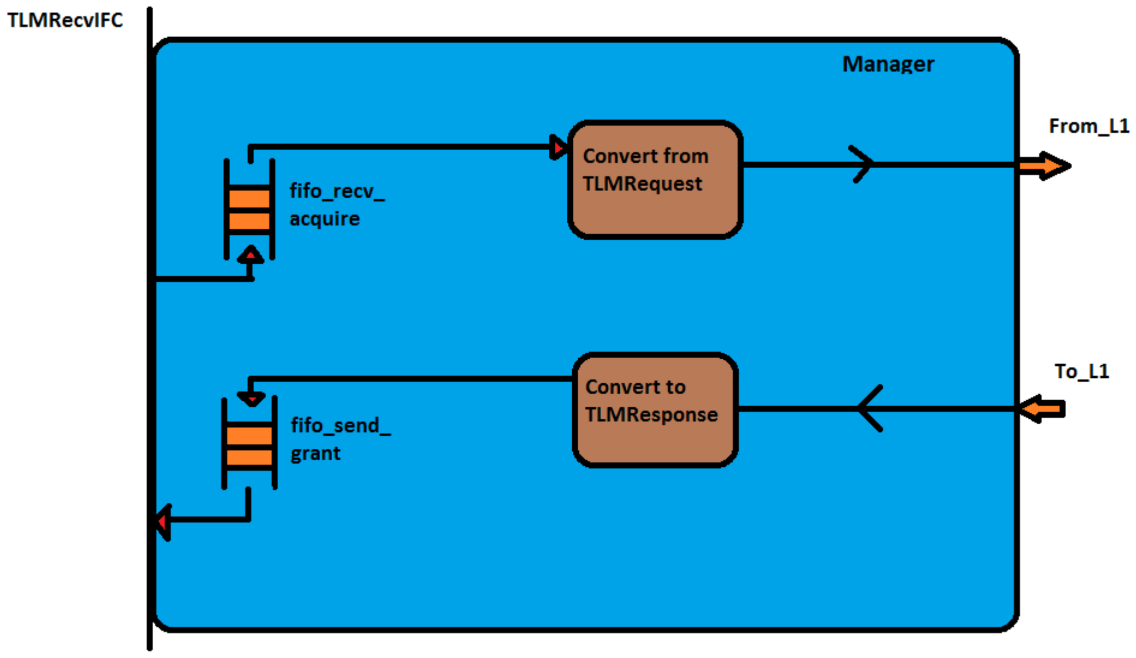


Figure 5.4: Manager Implementation

5.2.3 Logical Tilelink Network

The implementaion of the Logical Tilelink Network module in the above Tilelink protocol is shown in Figure 5.5. The FIFOs - fifo_recv_acquire and fifo_send_grant are connected to the TLMRecvIFC using toPut() and toGet() respectively, and the FIFOs - fifo_send_acquire and fifo_recv_grant are connected to the TLMSendIFC using toGet() and toPut() respectively. The acquire signal in fifo_recv_acquire in the form of TLMRequest is enqueued into the fifo_send_acquire by the rule rule_acquire. Similarly, the grant signal in fifo_recv_grant in the form of TLMResponse is enqueued into the fifo_send_grant by the rule rule_grant. The rules - rule_acquire and rule_grant should not fire in parallel and the rule_grant should be given priority over rule_acquire as the channel Grant has more priority than channel Acquire in the Tilelink protocol, to avoid deadlock. This priority of the rules is maintained by the scheduling attributes - descending_urgency, and mutually_exclusive, which are provided by BSV.

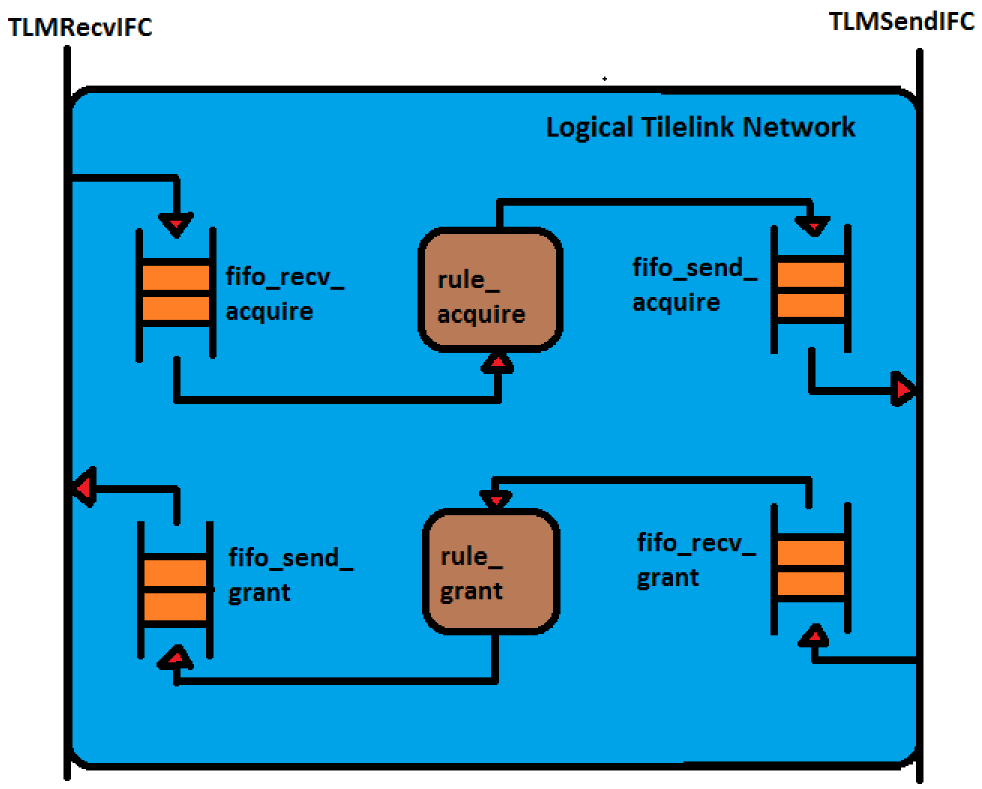


Figure 5.5: Logical Tilelink Network Implementation

These modules have been connected in SOC by mkConnection of Connectable class, which is provided by BSV, the resultant SOC is then connected to memory.

CHAPTER 6

Conclusion and Future work

The Tilelink protocol is implemented on both Instruction caches and Data caches. It has been tested with RISC-V instruction set, and it's working perfectly. This design is realized in Bluespec System Verilog (BSV) which provides module and configuration flexibility. The design is fully synthesizable by the verilog code generated by the Bluespec compiler. Sizes of the FIFOs used in the implementation are random numbers that are adequate enough for testing purposes.

This thesis shows the implementation of the Tilelink protocol in the single-core processor, it can be easily extended to the multi-core processor with the given Tilelink specifications in chapter-4 and with a similar implementation and extending the modules specified in the chapter-5. For implementing Tilelink protocol in multi-core processor, the Client Metadata should be added to the client module, and the Manager Metadata should be added to the Manager module. All the five channels (or four excluding the Finish) are needed for implementing multi-core, and the priority is maintained by the given scheduling attributes. When one of these channels make a hit with either the Client Metadata or Manager Metadata, the metadata should be updated and any new messages should be created if required by the corresponding Cache Coherence policy implemented in that domain.

REFERENCES

[1] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte, “Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies”, December 2011.

[2] https://github.com/ucb-bar/uncore, Tilelink 0.3.3 specification, 2015

[3] Bluespec,Inc.Bluespec System Verilog Reference Guide, Revision 30 July 2014.