CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Modeling and Simulation of MESI Cache Coherency Protocol

A graduate project submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Engineering

By

Kambla Kethana Rao

May 2023

The graduate project of Kambla Kethana Rao is approved:

Dr. Brad Jackson	Date
Dr. Vijay J Bhatt	Date
Dr. Shahnam Mirzaei, Chair	Date

California State University, Northridge

Acknowledgements

I want to start by expressing my sincere gratitude and respect to Dr. Shahnam Mirzaei for overseeing my project during my time at California State University, Northridge, and for his crucial contribution to this project. I want to thank him for providing me with the chance to work with him and for introducing me to the field of computer architecture. I would not have been able to finish this project without his priceless guidance and help. I owe him a tremendous debt of gratitude for his unceasing support and invaluable counsel in all facets of my academic life.

I would also like to thank Dr. Brad Jackson and Dr. Vijay J Bhatt for supporting and guiding me to complete my graduate project. I am grateful of Dr. Geng for all the support and guidance she has provided me throughout my masters. I appreciate the helpful assistance of the Computer Engineering Department's teachers and secretarial staff.

Lastly, I would like to thank my family for their constant support throughout my life.

Table of Contents

List of Figures

List of Tables

Abstract

Modeling and Simulation of MESI Cache Coherency Protocol on FPGA

By

Kambla Kethana Rao

Master of Science in Computer Engineering

To make multi-core processors faster and more dependable, the number of cores is steadily raised. Many problems arise with this increase and they need to be resolved. In this project, I worked on resolving one of the problems called the cache coherence. The concurrent functioning of multiple processors and the potential for separate caches to contain different versions of the same memory block lead to a cache coherence problem. To prevent the use of out-of-date values, the data which is shared by different cores must have the latest value. To address the issue of cache coherency and maintain data consistency across all caches and memory, cache coherence protocols are used. In this project I have worked on MESI protocol. Questasim is used as the simulator and Verilog and system Verilog languages are used for the implementation. To place the data into the cache and to replace them certain policies must be followed for efficiency. In this project, set associative is used for both the instructions cache and for the data cache and True- LRU is used as the replacement policy. Cache statistics were generated for hit rate/miss rate count. Designed and Simulated a split L1 cache backed by shared L2 cache Verilog, (Questasim). Implemented MESI protocol for cache coherency in a and various testcases were generated were by changing the number of sets, block size and number of ways and the cache statistics for hit rate/miss rate count where compared for each test case. Graphs were generated by comparing each test case to show the changing of the hit ratio with the changes in the size of the cache, number of sets, number of ways and the size of blocks. Verified using Individual trace tiles.

Chapter 1 Introduction

1.1 Introduction

Since computers have become such an integral part of modern life, their performance is vital. Earlier, the idea was that a faster computer would result from raising processor frequency. . To attain great performance, however, designers must pay attention to other factors like expanding the number of cores when the market for increasing CPU frequency has reached saturation. Cache was added for quicker memory access as more cores were being used. As the caches have become essential parts of contemporary processors. By reducing the number of access to the main memory , they dramatically improve the performance of the system. Most of microprocessors contain numerous layers of cache to mask the growing gap between CPU and memory speed. In the era of multi-core systems, we have two architectural choices: a distributed architecture or a shared memory approach . According to the distributed paradigm, each core needs to have its own private memory. Different cores communicate with one another by using a message passing mechanism. The shared memory model, which assumes that all cores access a single, common memory, is the more popular of the two options. Multiple cached copies may exist at any given time in a shared memory system. Therefore, for such a system to function as intended, coherence and uniformity are crucial.

1.2 Cache Coherency

Data consistency across caches in a shared memory system is known as cache coherence. A portion of the data existing in the main memory is stored in each processor's cache memory in a multi-processor system.

Cache coherence makes sure that every cache has the most recent version of the data when different processors try to access the same piece of information. This is required because modifications made to a specific data item by one processor must be visible to all other

processors, ensuring that each processor sees a consistent view of memory.exchange words with one another. The shared memory model, which assumes that all cores access a single, common memory, is the more popular of the two options. Multiple cached copies may exist at any given time in a shared memory system. Therefore, for such a system to function as intended, coherence and uniformity are crucial. The fundamental concept is that each processor should broadcast its action to every other processor, and each processor should modify the shared block's state as necessary. Distributed caches are coordinated by coherence protocols, giving processors a consistent picture of the memory.

1.3 Cache Coherency Protocol

The basic approaches were to Update or to Invalidate. When a write action is carried out in the update protocol, a broadcast message is sent to others, who then update their cache block. When a processor performs an invalidation protocol write action, the other processors receive a broadcast invalidation message and clear their blocks as necessary. With all of the aforementioned methods, there are disadvantages. The broadcasting of "update" messages in the Update method is useless if a data block is only stored in one cache. Similar to this, if the core performs update and read operations on its own, broadcasting an invalidate message in the "Invalidate" method is pointless. The correct update protocol was required. Directory-based method created “by Censier and Feautrier in IEEE Toc1978” and Snoopy-based technique created by “Goodman12 in 1983”. The two most researched cache coherence algorithms in use today are these two.

1.3.1 Snoop Based Cache Protocol

The protocol for preserving cache coherency in symmetric multiprocessing environments is also known as a bus-snooping algorithm. To check if they have a copy of the requested data block, all caches on the bus use a snooping system to monitor (or snoop) the bus. For each

physical memory block it holds, each cache keeps a copy of the sharing data. Multiple copies of a document can frequently be viewed in a multiprocessing environment without any problems with coherence, but in order to write, a processor needs to have exclusive access to the bus.

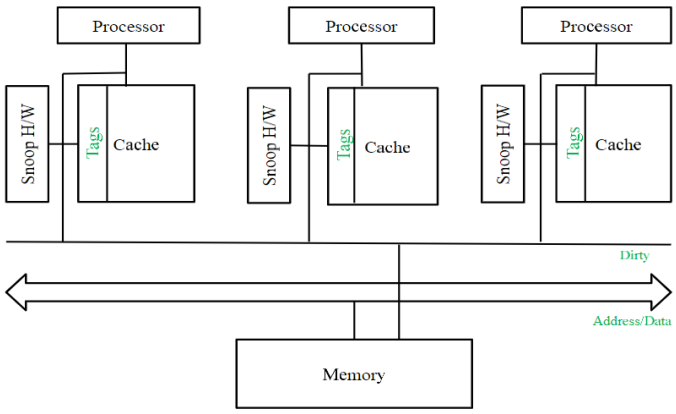

Figure 1.1 Snoop Based Cache Protocol

There are two main types of snooping protocol:

1.3.1.1 Write-invalidate

The processor that is writing data invalidates copies in the caches of every other processor in the system before making modifications to its local duplicate. The local computer tells all other caches to look for a copy of the invalidated file by sending an invalidation signal over the bus. Once the cache copies have been made invalid, the local computer's data can be updated until another processor requests it.

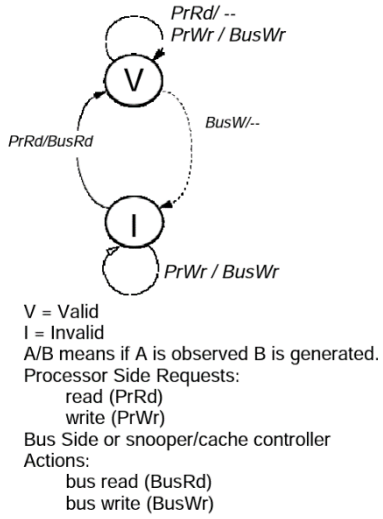

Figure 1.2 Write Invalidate

1.3.1.2 Write-update

With this protocol, whenever a processor modifies the data in its cache, all other cached versions are instantly updated as well. The new data block is sent via the broadcast method to every cache that has copies. The new data is sent over the bus by the data writer processor. All caches that have copies of the content are then updated. This scheme does not only create one local duplicate for writes, in opposition to write-invalidate.

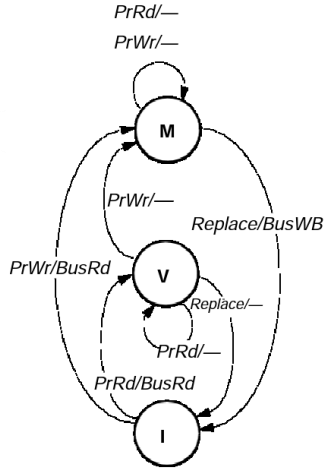

Figure 1.3 Write update

1.3.2 Directory-based coherence

Directory-based cache is a type of cache coherence protocol used in multi-processor systems. In this protocol, a central directory maintains information about the location and state of data blocks stored in different caches throughout the system. Each cache block has a corresponding entry in the directory, which indicates which processor(s) hold a copy of the block, and whether it is in a modified, shared, or invalid state. When a processor needs to access a block that is not present in its own cache, it first checks the directory to determine its location and state. If another processor has a copy of the block, the requesting processor can either read the data directly from the other processor's cache or request a copy of the block from the other cache. When a processor modifies a block in its cache, it updates the directory to reflect the new state of the block. The directory can then inform other processors holding a copy of the block to invalidate their copy or request a copy of the modified block to be written back to main memory. Directory-based cache coherence can improve performance compared to other coherence protocols in large-scale multi-processor systems as it reduces the amount of traffic on the interconnect network by centralizing the coherence information. However, the directory itself can become a bottleneck in highly parallel systems with many processors, and the overhead of maintaining directory consistency can impact overall system performance.

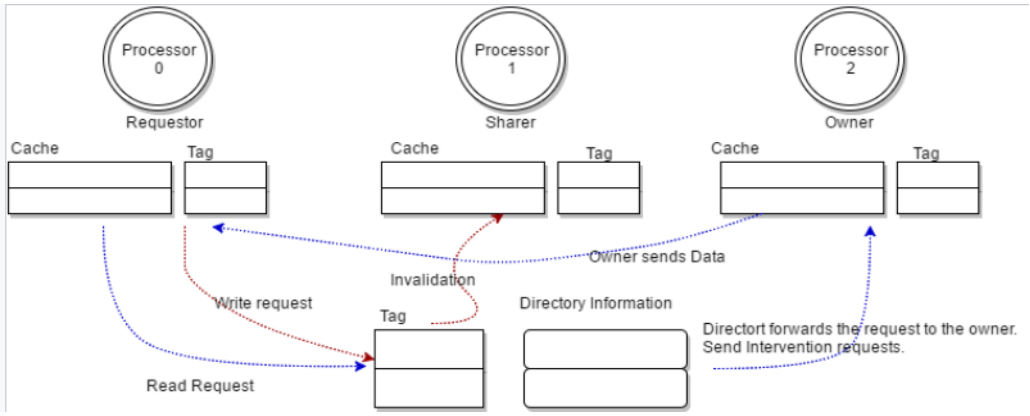

Figure 1.4 Directory-based coherence scheme

The various nodes in the shared memory system that adhere to the directory-based coherence protocol are listed below.

The node in the memory block known as the Requestor Node is the one asking the processor to perform a write or a read. The Director Node is the node that stores the state information for each cache block in the system, and the Requestor sends its queries there. The owner node is the node that currently controls the cache block's most recent state. The directory might not always contain the most recent information.Sharer nodes are any number of nodes that are sharing a copy of the cache block.

Other coherence protocols, such as snooping-based coherence, which involve broadcasting of cache access requests to all processors and consequently cause a substantial amount of traffic on the interconnect and lower system performance, are more scalable than directory-based coherence.

1.4. Snoop vs Directory

Bus-based systems are limited by the fact that all nodes are simultaneously using a common bus, which prevents them from performing well. Bus networks can perform well for a modest number of nodes. While the number of nodes is increasing, some issues could arise in this respect. Particularly given that only one node may use the bus at once, which will adversely affect the system's total performance. On the other hand, directory-based systems won't experience this slowdown, which would limit the system's ability to scale. Since the bus structure itself can arrange all of the system's traffic and guarantee the smooth passage of all signals that is passed through. it serves as a traffic organizer.

1.5 Requirements for Proper coherence protocol

There are a few key requirements that must be met for any coherent strategy to be effective. For example, a protocol should be bandwidth-efficient, bus-based communication should be avoided, and transmission delay from cache to cache should be as low as possible. The initial move from cache to cache is crucial because we prefer to move relevant data from cache to cache rather than through main memory. Approaches based on snoopy enable low-latency cache-to-cache transfer since they broadcast messages on the bus. In contrast, directory-based protocols add an extra clock cycle by delaying sending requests to the destination cache until they have received an acknowledgement. The second step is to steer clear of bus-like designs because they limit the system's ability to accommodate more cores. Transmission is point-to-point in a directory-based system, enabling the integration of many cores. Thirdly, bandwidth reduction has a finite amount of effectiveness. Reduced bandwidth is necessary to prevent interconnect contention since it has an impact on system performance.

1.5.1 Evolution of cache coherence protocols

For sustaining coherence, a number of models and protocols have been developed, including the write-once, Dragon protocols, Synapse, Berkeley and Firefly.

1.5.2 MESI PROTOCOL

The MESI protocol is a cache coherence protocol used in multi-processor systems. It is an acronym for Modified, Exclusive, Shared, and Invalid, which are the four different states that a cache line can be in.

1. Modified (M): When it is in the Modified state, it means that the line has been modified by the processor, and the data in the cache line is the most up-to-date version. Any changes made to the cache line will be written back to main memory when the cache line is evicted.
2. Exclusive (E): When it is in the Exclusive state, it means that the line is present only in

the cache of a single processor, and the data in the cache line is the most up-to-date version. No other processor can have a copy of this cache line.

3. Shared (S): When a cache line is in the Shared state, it means that the line is present in multiple processor caches, and all the copies contain the same data. Any processor can read the data in the cache line, but if one processor modifies the data, it must transition the cache line to the Modified state.
4. Invalid (I): When a cache line is in the Invalid state, it means that the line is not present in the cache of any processor, and accessing this line will cause a cache miss.

The MESI protocol uses a set of rules to ensure cache coherence, such as:

- If a processor wants to read or write to a cache line in the Exclusive or Shared state, it must first gain ownership of the line by requesting it from other processors.
- If a processor wants to write to a cache line in the Shared state, it must first transition the line to the Modified state and invalidate all other copies of the line in other processor caches.

Overall, the MESI protocol ensures that all processors see a consistent view of memory by coordinating cache coherence between the caches of different processors in the system.

States	**Processors Load**	**Processors Store**	**Processors Eviction**	**Incoming Read Req.**	**Incoming Write Req**
Modified	Hit	Hit	Write-back and Change to Invalid	Send data Write-back and Change to Shared	Send data and Change to Invalid
Exclusive	Hit	Change to Modified	Silent Evict and Change to Invalid	Send data Write-back and Change to Shared	Send data Hit and Change to Invalid
Shared		Change to Modified	Silent Evict and Change to Invalid	None	Change to Invalid
Invalid	Hit	Change to Modified	None	None	None

Table 1- MESI Protocol

Chapter 2 Cache Memory

2.1 Cache

Cache memory, usually referred to as cache, is a separate memory system that temporarily retains recently used information or instructions so that a central processing unit (CPU) may process them more quickly. The cache of a computer augments and increases the primary memory. Both the cache and the main memory are internal random-access memories. (RAMs). Only the most frequently utilized data or program code from the main memory is moved to the cache. Finding data in the cache and transferring it to the CPU for processing takes less time due to its reduced capacity. The locality of reference is the principle used by the cache memory, which states that programs tend to access the same data and instructions repeatedly, and therefore it is beneficial to keep this data and instructions in a small, fast memory to reduce access time. There are several levels of cache memory in modern computer systems, with each level having a larger capacity and a longer access time than the previous level. The cache levels include L1 (level 1) cache, L2 (level 2) cache, and sometimes L3 (level 3) cache.

However, a device's cache memory may not always be able to store all of the necessary data because it is often limited in relation to its RAM and CPU processing capabilities. Whichever event actually takes place will determine whether a "cache hit" or "cache miss" occurs.

2.2 Operations performed on local caches:

1. Write hit
2. Read miss
3. Read hit
4. Miss

1. Write hit: When a processor modifies a piece of data that is already present in its local cache, it is called a write hit. In this case, the processor can directly update the data in its local cache without accessing the main memory. The modified data is kept in the

cache until the cache line is evicted or until the processor explicitly writes the data back to the main memory.

2. Read miss: When a processor needs to read a piece of data that is not present in its local cache, it is called a read miss. In this case, the processor must retrieve the data from the main memory and load it into its local cache. If other processors have a copy of the same data in their local caches, the cache coherence protocol is used to ensure that all processors see a consistent view of memory.
3. Read hit: When a processor needs to read a piece of data that is already present in its local cache, it is called a read hit. In this case, the processor can directly access the data in its local cache without accessing the main memory. This operation is faster than a read miss because the data is already present in the cache.
4. Miss: A cache miss can refer to either a read miss or a write miss, which occurs when a processor needs to modify or read data that is not present in its local cache. In this case, the processor must retrieve the data from the main memory and load it into its local cache. If other processors have a copy of the same data in their local caches, the cache coherence protocol is used to ensure that all processors see a consistent view of memory. Cache misses are slower than cache hits because they require accessing the main memory, which is slower than accessing the cache.

2.3 Cache writing policies

Cache writing policy refers to the way in which a cache handles writes to data that is already present in the cache. There are two main cache writing policies: write-through and write-back.

Write-through: In a write-through cache, every write operation to the cache is also written to the main memory. When a processor writes to a cache line, the data is first written to the cache, and then immediately written to the main memory to maintain cache coherency. However, this

policy can cause a significant slowdown in write operations because every write to the cache requires an additional write to the main memory.

Write-back: In a write-back cache, writes to the cache are only written to the main memory when the cache line is evicted or explicitly flushed. When a processor writes to a cache line, the modified data is kept in the cache until the cache line is evicted or until the processor explicitly writes the data back to the main memory. This policy can improve performance by reducing the number of writes to the main memory. However, it can also introduce the risk of data inconsistency if a cache line is modified in one cache but not updated in another cache.

In addition to these two main cache writing policies, there are also hybrid policies, such as write-combining, where multiple writes to the same cache line are combined into a single write to the main memory to improve performance. The choice of cache writing policy depends on the specific application and the trade-off between performance and consistency.

2.4 Types of Cache Memory

Cache memory is divided into two categories depending on its physical location and proximity to the device's CPU.

1.Primary Cache Memory: The primary cache memory, also known as the main cache memory, is the SRAM located on the same die as the CPU, which is as close as it can be installed. This is the type generally used in the storage and retrieval of information between the CPU and the RAM.

2.Secondary Cache Memory: The secondary cache memory is the same hardware as the primary cache memory. However, it's placed further away from the CPU, ensuring the existence of a backup SRAM that can be reached by the CPU whenever needed.

2.5 Different Level of Cache Memory

Modern computer systems have more than one piece of cache memory, and these caches vary in size and proximity to the processor cores and, therefore, in speed. These are known as cache levels.

The smallest and fastest cache memory is known as Level 1 cache, or L1 cache, and the next is the L2 cache, then L3. Most systems now have an L3 cache. Since the introduction of its Skylake chips, Intel has added L4 cache memory to some of its processors as well. However, it's not as common.

2.5.1 Level 1 Cache

Level 1 (L1) Cache: This cache memory is located on the same chip as the CPU, also known as the microprocessor. It is the smallest and fastest cache memory, with a capacity of a few kilobytes to a few hundred kilobytes. The L1 cache is divided into separate instruction and data caches, each with its own set of control logic. The L1 cache operates at the same clock speed as the CPU and provides low-latency access to the most frequently used data and instructions.

The size of the L1 cache varies depending on the CPU architecture, but it is typically very small, with sizes ranging from 8 KB to 512 KB. Because of its small size, the L1 cache is very expensive to manufacture and is therefore used sparingly. The main advantage of L1 cache is its speed. Because it is located on the CPU itself, it can be accessed very quickly, with access times measured in nanoseconds. This makes it ideal for storing frequently accessed data and instructions that the CPU needs to access quickly. L1 split cache refers to a cache memory hierarchy design where the first level cache (L1 cache) is divided into separate instruction and data caches. In this design, the processor has two independent caches that store instructions and data respectively, rather than a single cache that stores both.

2.5.2 Level 2 Cache

A form of memory called Level 2 (L2) cache is incorporated into a computer's processor chip to help speed up access to frequently accessed data. It is a short chunk of memory that serves as a buffer to hold frequently accessed data and is located between the CPU and the main memory (RAM). Because L2 cache is quicker than main memory, the processor can retrieve data more quickly, enhancing the computer's performance. L2 cache can be a separate chip on the same package as the CPU or it can be integrated into the processor itself, unlike L1 cache, which is normally built directly into the processor core. Depending on the processor model and maker, the L2 cache size and performance can change.

2.5.3 Level 3 Cache

Level 3 (L3) cache is a type of memory that is also built into some processor chips, typically in high-end processors used in servers and workstations. It is a larger cache memory than L2 cache, which is intended to further reduce the latency of accessing data from the main memory. L3 cache is similar in function to L2 cache, acting as a buffer between the processor and main memory to store frequently accessed data. However, it is typically larger in size than L2 cache and can be shared between multiple processor cores, allowing it to be accessed more efficiently by multiple cores. Like L2 cache, the size and speed of L3 cache can vary depending on the processor model and manufacturer. L3 cache is also important for maintaining consistency across multiple processor cores. In multi-core processors, multiple cores may be accessing the same data in main memory simultaneously. Without a shared L3 cache to coordinate access to this data, the different cores could end up with different versions of the same data, leading to inconsistent results and slower performance. However, L3 cache tends to be larger than L2 cache, and its presence can have a significant impact on the performance of multi-core processors.

2.6 Cache Mapping

There must be a mechanism for locating the required data because cache memory are extremely quick and keep expanding to meet the needs of software computing processes. Otherwise, the CPU can find that it spends more time looking up the appropriate instruction in memory than processing it. The address of the data or instruction that the CPU wishes to read from RAM is known. It has to check the memory cache to verify if a reference to that RAM memory address and the related data or instruction are present. There are many strategies for moving the information and commands from RAM to cache memory, and they frequently give priority to some factors over others. For example, lowering the search speed affects accuracy and the chance of a cache hit. Meanwhile, increasing the likelihood of a cache hit also lengthens the typical search times. Depending on the various levels of compromise in speed and accuracy, there are three types of cache mapping techniques. They are

1.Direct cache mapping

2.Assosciative mapping

3.Set assosicative mapping

2.6.1 Set-Associative Cache Mapping

In this project, I have used the set- assosicative mapping. set assosicative cache mapping

Set-associative cache mapping is a caching technique that combines the benefits of direct-mapped and fully-associative cache mapping techniques. It is a compromise between the two methods and is widely used in modern processors.

In set-associative cache mapping, the cache is divided into a number of sets, and each set contains multiple cache lines. Each cache line in a set is associated with a tag that indicates the memory address of the data stored in that cache line. When a CPU needs to access data, the memory address is first divided into three fields: the tag field, the set field, and the word field.

The tag field is used to identify the memory block being accessed, the set field is used to identify the set in which the memory block is located, and the word field identifies the byte within the block that is being accessed. The CPU then searches the cache for the requested data by comparing the tag value with the tags in the cache lines in the corresponding set.

If the requested data is found in the cache, it is called a cache hit and the data is returned to the CPU. If the requested data is not found in the cache, it is called a cache miss and the CPU retrieves the data from the main memory and stores it in the cache.

The advantage of set-associative cache mapping over direct-mapped cache mapping is that it reduces the likelihood of cache conflicts, where two or more memory blocks map to the same cache line. Set-associative mapping provides more flexibility and can store more data than direct-mapped caches while still being efficient in terms of hardware complexity and cache access speed

2.7 Cache replacement policies

Our cache's capacity is limited. Particularly in caching setups that make use of pricey and high-performance storage. In other words, we are forced to evict certain objects while keeping others. Algorithms for replacing caches do this. They make the decisions regarding what belongs there and what has to go. Different cache replacement policies have different trade-offs between cache hit rate, complexity, and overhead. The choice of the cache replacement policy depends on the specific application and the design goals of the system. Few of the policies are FIFO, LFU and random replacement policy. In FIFO, the first cache line that was loaded into the cache is the first to be replaced. As the name suggests, this policy randomly selects a cache line to be replaced. It does not consider the access history or any other factor, making it a simple and easy-to-implement policy. I used the LRU policy.

2.7.1 LRU algorithm

The Least Recently Used (LRU) policy is a cache eviction algorithm used in computer operating systems and other computer systems to manage memory. It works on the principle that the data items that are least recently used should be evicted first. When the cache becomes full and there is a need to store a new item, the LRU policy checks the dirty bits of all the items in the cache. It identifies the item that has been accessed the least recently and evicts it from the cache to make space for the new item. The LRU policy is based on the assumption that data items that were accessed recently are more likely to be accessed again in the near future than the items that were accessed a long time ago. Therefore, by evicting the least recently used items, the cache can make room for more frequently accessed items, thus improving cache hit rate and overall system performance. To implement the LRU algorithm, a data structure called a doubly linked list is typically used. Each node in the linked list represents a cache item, and the linked list is ordered from most recently used to least recently used. When an item is accessed, it is moved to the front of the linked list. When a new item needs to be added to the cache, the least recently used item is removed from the back of the linked list. The LRU algorithm is popular because it is simple to implement and often provides good cache performance. However, it can be slow if the cache size is very large, and it may not perform well in certain access patterns. Other cache eviction algorithms, such as LFU (Least Frequently Used) and MRU (Most Recently Used), are also used in practice depending on the specific use case.

2.8 Calculations

The following formulas have been used.

Size of single cache line= Size of single frame in RAM= Size of single page in Processor.

Number of blocks = Size of cache / number of words in block.

Number of sets = Number of blocks in cache

/number of blocks in set

Tag bits = Number of bits in Main memory address – Sum of other field.

Number of sets = 2 power index bits

Block Size = 2 power offset

Number of frames/set = 2 power tag bits

Cache capacity = Block size in byte * (blocks per set) * (number of sets)

Cache size = Block size * set * way

Cache size= Number of lines in cache * size of line

Tag bits = (address bits) – (index bits) – (block offset bits)

Number of frames in RAM = Number of sets * Number of frames/set

Chapter 3

System Architecture

3.1 Design

This project functionally simulates a split instruction/data L1 cache for a 32-bit processor in a system with multiple processors. The system employs MESI protocol to ensure cache coherence. The instruction cache is 2-way set associative, consists of 16K sets, and has 64-byte lines. The data cache is 4-way set associative, consists of 16K sets, and has 64-byte lines. Both caches employ LRU replacement policy and are backed by an L2 cache (which is modeled as a stub in this simulation). Snoop protocol is used. Statistics regarding the number of reads, writes, hits, and misses are generated, as well as a hit percentage rate. This simulation has a single-cycle interface between a processor and L1, and between L1 and L2. All processor reads and writes are a single byte

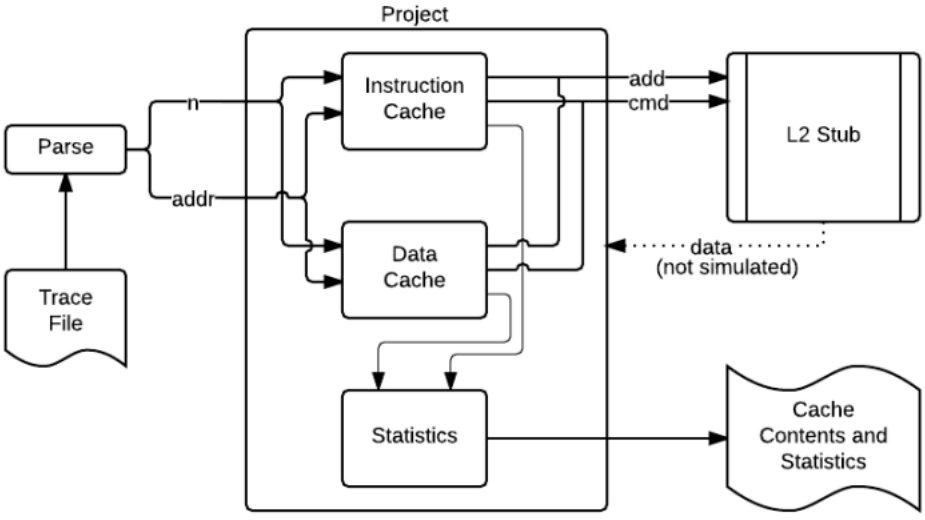

Figure 3.1 Model of code

Implemented MESI protocol for cache coherency in a 2-way set associative L1 instruction cache and 4-way set associative data cache with write-back using write-allocate, used True-

LRU as eviction policy. Generated cache statistics for hit rate/miss rate count.

In the course of designing this L1 cache, I have made a few assumptions regarding the CPU:

• The cache hierarchy is inclusive. By making the L2 cache support an inclusive policy, the synchronization logic between the L1 and L2 cache is greatly simplified.

• The data cache is write-through. The L1 and L2 caches together are required to support memory writebacks. However, because the cache design also needs to support MESI, I decided to implement the L1 data cache as a simple write-through cache. Because the cache hierarchy uses an inclusive policy, evictions forced by MESI in the L2 cache will force an eviction in the L1 cache.

• Cache contents (actual data) are irrelevant for this simulation. Thus, all byte offsets are ignored. Because we only stub out the processor, the next level cache, and our cache eviction policy is based entirely on memory addresses, there is no need to examine data values.

The 32-bit addresses from the processor are broken down into the following fields:

31:20 = 12-bit tag

19:6 = 14-bit index

5:0 = 6-bit offset

The following method for interface with L2 is defined. A 26-bit address specifies a 64-byte cache line, and must be supplied with one of the following 2-bit cmd_out commands:

00 No Operation (ignore any input on address lines)

01 Read from L2

10 Write to L2

11 Read with intent to modify. To run, the simulation requires a trace file formatted using the protocol specified in the project description. A print command will output human-readable cache contents and statistics.

3.2 Explanation of the each module of the code.

3.2.1 Data module

The size of the sets, ways, tags and index is defined initially. Commands for the data cache, mesi parameters, and parameters for the L2 cache are defined. In the invalidate block, for every way in the cache, the LRU bits are set, the MESI is set to invalid and the tag bits are set to zero. For all the ways, if the incoming tag matches the current tag, the LRU bits are set. If the current tag in is shared or Exclusive state, the current tag is set to invalid. If the current tag is in modified state, then it is written back to L2 and then put in the invalid state. For the read block, the total numbers of reads are incremented first for the statistics. If the current tag state is in modified, shared or exclusive then, the hit count is increased. If there was no hit, it is checked if the current tag is in invalid state. If there was no hit, for all the ways in the set. it is checked to see if it is in invalid. New LRU bits are calculated based on the invalid way. If there was no invalid way, evict the LRU way. In the write module, firstly the write count is increased, then the ways within the set is searched, if there is a hit, the LRU is updated and the hit counter is increment. Then the current tag is put under exclusive state. If the current tag is in exclusive or modified state, then it is put in modified state first and hit count is incremented. If it is in shared state, the hit count is incremented and put in exclusive state. If there is no hit, it is checked if the given tag is in invalid state and the miss count is incremented. If it is not in invalid state, then an LRU is evicted. In the snoop module, if current index is in the modified or exclusive, then it is put in the shared state and the LRU bits are updated.

3.2.2 Instruction module

The size of the sets, ways, tags and index is defined initially. Parameters for MESI protocol and instruction cache are defined. First reset is defined, always at the positive edge, for all the sets in the cache, the LRU bits are set to zero. Then, for every way in the cache, the MESI is

set to invalid and the tag bits are set to zero. In the next block, the read count is increased first for the statistics. For all the ways in the sets,if the tag of the current index matches any tag in the shared or exclusive , then the hit counter is incremented, else the miss counter is incremented. If there was no hit,for all the ways in the set. it is checked to see if it is in invalid. New LRU bits are calculated based on the invalid way. If there was no invalid way, evict the LRU way. Till then the current index is kept under Shared.

3.2.3 LRU module

Dirty bits are activated for each set depending on the lrubits given from the cache modules.

3.2.4 Testbench

No actual data is given from the trace files. Only a command such as read or write is given along with the address. The input file is given in the testbench which opens and reads the files. If no clock signal is found, then it is set to reset.

3.2.5 Statistics module

A statistics module is written to calculate the data reads, data writes, data hits, data miss and the hit ratios and the miss ratio of the data cache and the instruction cache. A conditional operator is used to set the value to zero if the denominator of the hit ratio values to zero, so that the value does not become undefined.

3.2.6 Top level module

In digital design, the top module is the highest level module in a design hierarchy. It is the module that contains all other modules and sub-modules within a design. The top module is typically the first module instantiated in a design and represents the entire system or device being designed. A top module is defined to accumulate all of the modules. It connects the L1 and L2 module with the data and instruction cache. It takes the values given from the testbench

and gives it to the statistics module to compute the output. The commands from the trace files are defined in this module.

3.2.7 Trace file

A trace file is a type of computer file that records the sequence of events or actions that occur during the execution of a program or system. It is often used in debugging or performance analysis of software applications or computer systems. A trace file typically contains a detailed log of the system's activities, including the time and date of each event, the type of event that occurred, any associated error messages or warnings, and other relevant information. This data can be used to analyze system behavior, identify performance bottlenecks, or track down bugs and other issues. Trace files can be generated by various types of software, including operating systems, network monitoring tools, and application profiling software. They are often saved in a plain text format or a specialized binary format, depending on the software that generated them. The data is irrelevant in this project. Hence the input file has only 2 values, the command and the address.

Chapter 4 Testing and Results

4.1 Questa Sim

Questa Sim is used as the simulator. The code was written in Notepad++ files and uploaded onto Questa Sim, where they were compiled, simulated and run to receive the output. Questa Sim is used in large multi-million gate designs, and is supported on Microsoft Windows and Linux, in 32-bit and 64-bit architectures.

Firstly new project is created and the files are added to it.

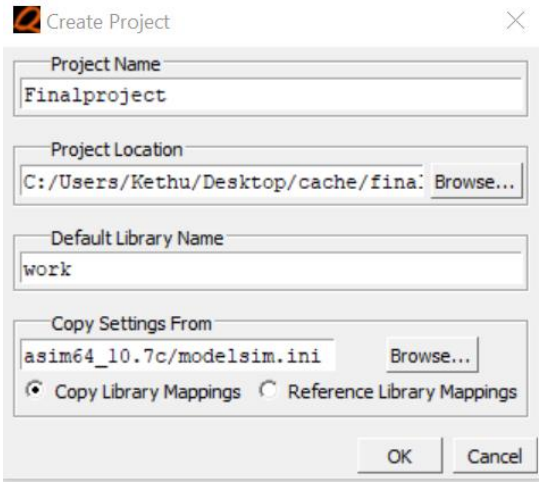

Figure 4.1 New project

Then all the files are compiled and if no errors are found, the simulation is started.

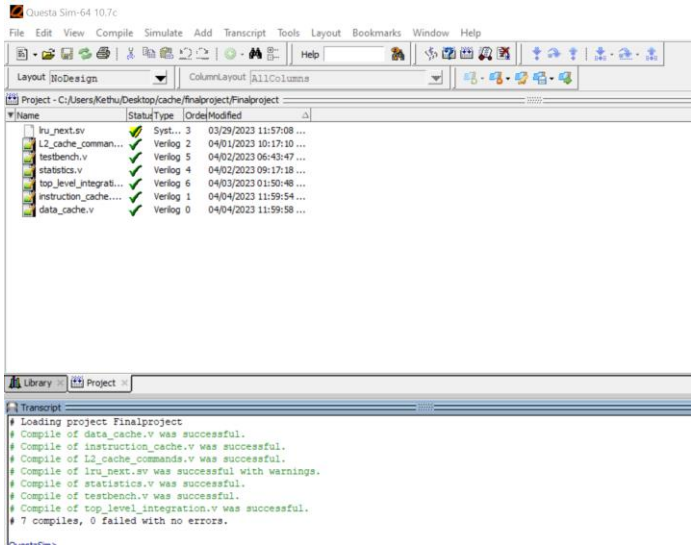

Figure 4.2 Compilation

In the simulation, the testbench is selected and all the files are run. Then the statistics output is received.

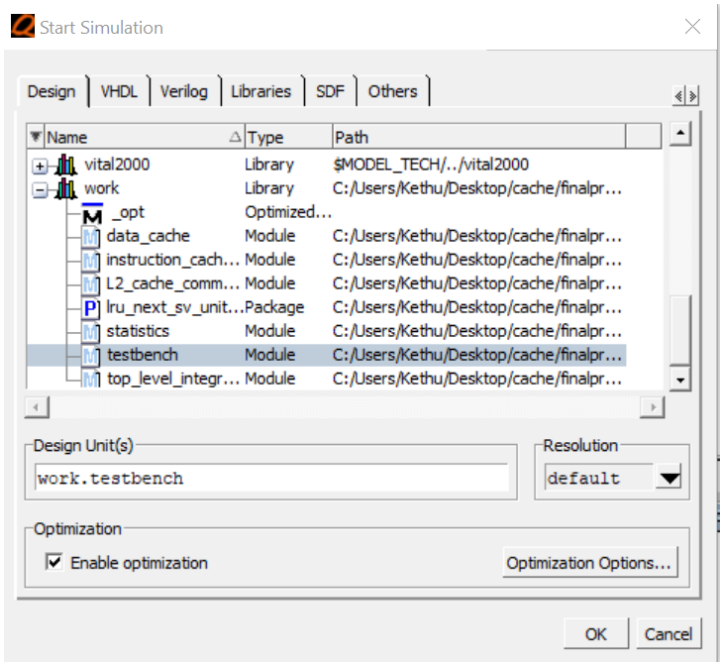

Figure 4.3 Simulation

4.2 Test Vectors

```
2 408ed4
0 10019d94
2 408ed8
1 10019d88
2 408edc
0 10013220
2 408ee0
2 408ee4
1 100230b8
2 408ee8
0 10013220
2 408eec
2 408ef0
2 408ef4
1 10013220
```

Figure 4.4 Test Vectors

As the data is irrelevant, the test vectors are only, the command and the address.Here,

Read = 4'd0

Write = 4'd1

Instruction Fetch = 4'd2

4.4 Results

4.4.1 Testcase 1

A 4-way set associative instruction cache with 16K sets and a 4-way set associative data cache with 16K sets and cache size of 4MB was designed and the following outputs were obtained. The following are the detailed specifications

Index bits- 14

Tag bits- 12

Sets- 16k

Way-4

Cache size -4MB

Block size- 2*6

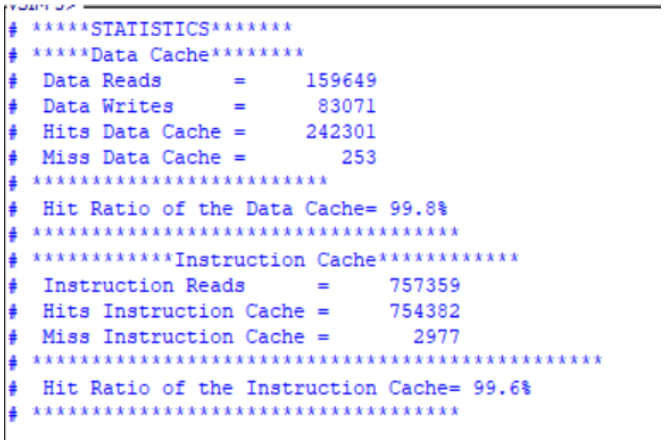

Figure 4.5 Result 1

4.4.2 Testcase 2

A 4-way set associative instruction cache with 4K sets and a 4-way set associative data cache with 4K sets and cache size of 4MB was designed and the following outputs were obtained. The following are the detailed specifications

Index bits- 12

Tag bits- 12

Sets- 4k

Way-4

Cache size -4MB

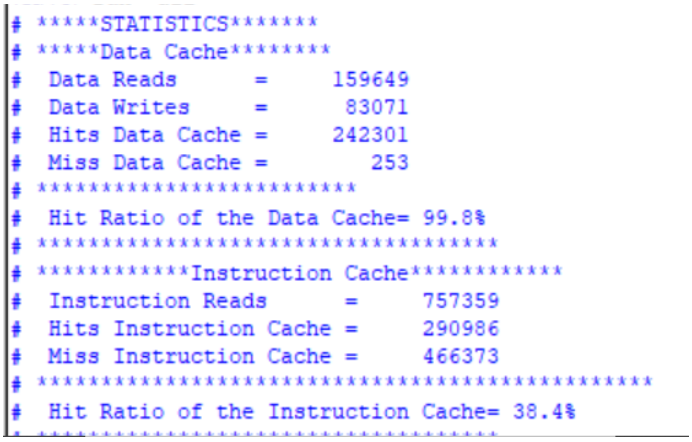

Figure 4.6 Result 2

4.4.3 Testcase 3

A 4-way set associative instruction cache with 1K sets and a 4-way set associative data cache with 1K sets and cache size of 4MB was designed and the following outputs were obtained. The following are the detailed specifications

Index bits- 10

Tag bits- 12

Sets- 1k

Way-4

Cache size -4mb

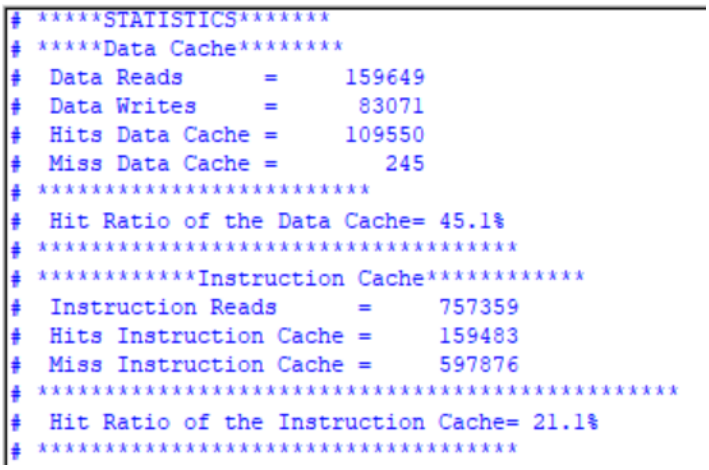

Figure 4.6 Result 3

4.4.4 Testcase 4

A 4-way set associative instruction cache with 4K sets and a 4-way set associative data cache with 4K sets and cache size of 4MB was designed and the following outputs were obtained. The following are the detailed specifications

A split cache with following specifications was developed

Cache size- 16mb

Number of sets-4k

Tag -10

Way -4

Index- 12

Block size- 2*10

```
VSIM 11> run -all
# *****STATISTICS*******
# *****Data Cache********
#  Data Reads      =     159649
#  Data Writes     =      83071
#  Hits Data Cache =     242301
#  Miss Data Cache =        253
# ************************
#  Hit Ratio of the Data Cache= 99.8%
# ***********************************
# ************Instruction Cache************
#  Instruction Reads      =     757359
#  Hits Instruction Cache =     290986
#  Miss Instruction Cache =     466373
# ************************************************
#  Hit Ratio of the Instruction Cache= 38.4%
# ***********************************
```

Figure 4.7 Result 4

4.4.5 Testcase 5

A 2-way set associative instruction cache with 8K sets and a 2-way set associative data cache with 8K sets and cache size of 16MB was designed and the following outputs were obtained. The following are the detailed specifications

Cache size- 16mb

Number of sets-8k

Tag -9

Way -2

Index- 13

Block size- 2*10

```
VSIM 14> run -all
# *****STATISTICS*******
# *****Data Cache********
#  Data Reads      =     159649
#  Data Writes     =      83071
#  Hits Data Cache =     242271
#  Miss Data Cache =        252
# ************************
#  Hit Ratio of the Data Cache= 99.8%
# ***********************************
# ************Instruction Cache************
#  Instruction Reads      =     757359
#  Hits Instruction Cache =     685798
#  Miss Instruction Cache =      71561
# ************************************************
#  Hit Ratio of the Instruction Cache= 90.6%
# ***********************************
```

Figure 4.8 Result 5

4.4.6 Testcase 6

A 8-way set associative instruction cache with 2K sets and a 8-way set associative data cache

with 2K sets and cache size of 16MB was designed and the following outputs were obtained. The following are the detailed specifications

Cache size- 16mb

Number of sets-2k

Tag -11

Way -8

Index- 11

Block size- 2 10

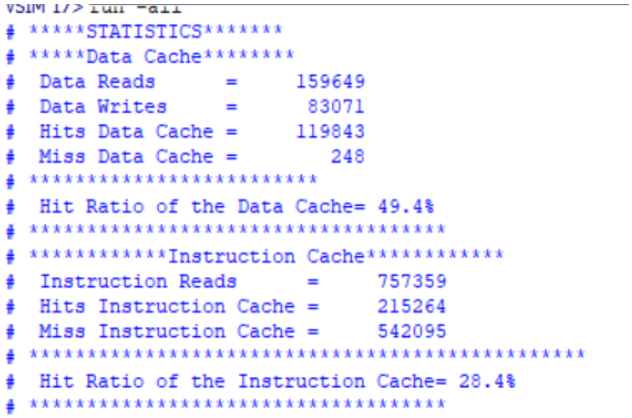

Figure 4.9 Result 6

4.4.7 Testcase 7

A 2-way set associative instruction cache with 8K sets and a 2-way set associative data cache with 8K sets and cache size of 64MB was designed and the following outputs were obtained. The following are the detailed specifications

Cache size- 64mb

Number of sets-8k

Tag -7

Way -2

Index- 13

Block size- 2*12

```
VSIM 22> run -all
# *****STATISTICS*******
# *****Data Cache********
#  Data Reads      =     159649
#  Data Writes     =      83071
#  Hits Data Cache =     242271
#  Miss Data Cache =        252
# ************************
#  Hit Ratio of the Data Cache= 99.8%
# ************************************
# ************Instruction Cache************
#  Instruction Reads      =     757359
#  Hits Instruction Cache =     685798
#  Miss Instruction Cache =      71561
# ***********************************************
#  Hit Ratio of the Instruction Cache= 90.6%
# ***********************************
```

Figure 4.10 Result 7

4.4.8 Testcase 8

A 4-way set associative instruction cache with 4K sets and a 4-way set associative data cache with 4K sets and cache size of 64MB was designed and the following outputs were obtained. The following are the detailed specifications

Cache size- 64mb

Number of sets-4k

Tag -8

Way -4

Index- 12

Block size- 2*12

```
VSIM 25> run -all
# *****STATISTICS*******
# *****Data Cache********
#  Data Reads      =     159649
#  Data Writes     =      83071
#  Hits Data Cache =     242301
#  Miss Data Cache =        253
# ************************
#  Hit Ratio of the Data Cache= 99.8%
# ************************************
# ************Instruction Cache************
#  Instruction Reads      =     757359
#  Hits Instruction Cache =     290986
#  Miss Instruction Cache =     466373
# ***********************************************
#  Hit Ratio of the Instruction Cache= 38.4%
# ***********************************
```

Figure 4.11 Result 8

4.4.9 Testcase 9

A 8-way set associative instruction cache with 2K sets and a 8-way set associative data cache

with 2K sets and cache size of 64MB was designed and the following outputs were obtained. The following are the detailed specifications

Cache size- 64mb

Number of sets-2k

Tag -9

Way -8

Index- 11

Block size- 2 12

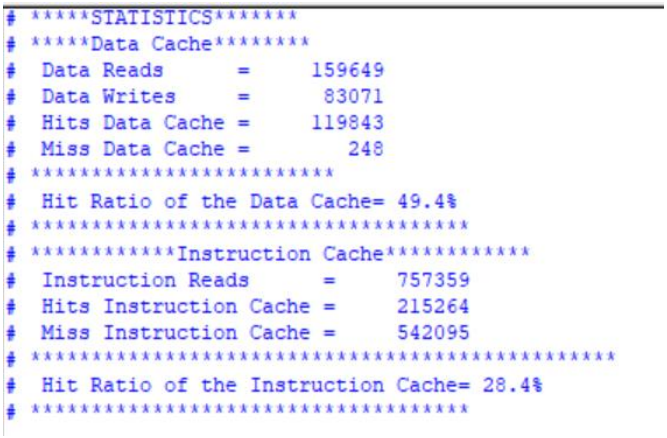

Figure 4.12 Result 9

Cache size- 4mb Tag bits- 12 Index bits – 14 Number of sets – 16k Block size – 2*6 Data Hit ratio- 99.8 Instruction hit ratio-99.6 Way 4	Cache size- 4mb Tag bits- 12 Index bits – 12 Number of sets – 4k Block size – 2*8 Data Hit ratio – 99.6 Instruction hit ratio- 38.4 Ways - 4	Cache size- 4mb Tag bits- 12 Index bits – 10 Number of sets – 1k Block size – 2*4 Data Hit ratio- 45 Instruction hit ratio- 21 Ways -4

Cache size- 16mb Tag bits- 9 Index bits – 13 Number of sets – 8k Block size – 2*10 Way-2 Data Hit ratio- 99.8 Instruction hit ratio- 90	Cache size- 16mb Tag bits- 10 Index bits – 12 Number of sets –4k Block size – 2*10 Way-4 Data Hit ratio – 99.6 Instruction hit ratio- 38	Cache size- 16mb Tag bits- 11 Index bits – 11 Number of sets – 2k Block size – 2*10 Way-8 Data Hit ratio- 49 Instruction hit ratio- 28
Cache size- 64mb Tag bits- 7 Index bits – 13 Number of sets – 8k Block size – 2*12 Way-2 Data Hit ratio- 99.8 Instruction hit ratio-90.6	Cache size- 64mb Tag bits- 8 Index bits – 12 Number of sets – 4k Block size – 2*12 Way-4 Data Hit ratio- 99.6 Instruction hit ratio- 38.4	Cache size- 64mb Tag bits- 9 Index bits – 11 Number of sets – 2k Block size – 2*12 Way-8 Data Hit ratio- 49 Instruction hit ratio- 28

Table – 4.1 Results

4.5 Graphs Results

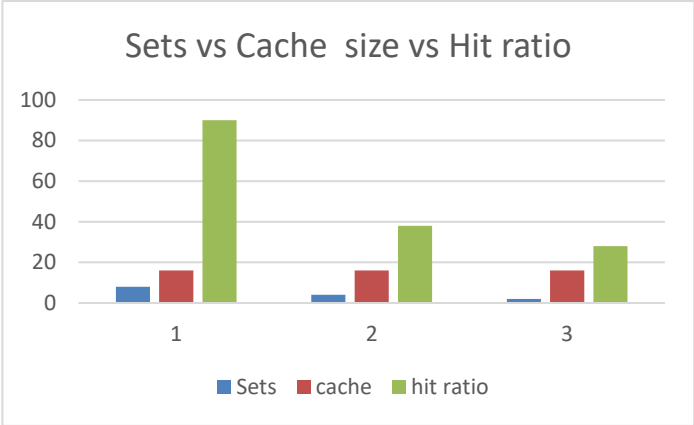

Figure 4.13 Graph 1

Sets vs Hit ratio vs cache

Cache size – constant (16mb)

Sets – varying (in the graph, the scale for sets are assumed as 1000s)

Hit ratio – varying (the scale is assumed as %)

Firstly, the cache size is kept constant at 16mb and the sets are constantly changed and with the decrease in the number of sets, the hit ratio decreases. It can be seen that the hit ratio decreases with decrease in the number of sets.

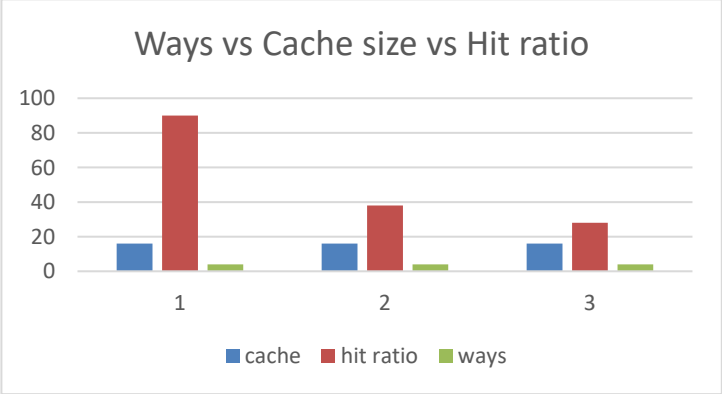

Figure 4.14 Graph 2

Ways vs Hit ratio vs Cache

Cache is constant at 16mb

Ways – varying

Hit ratio – varying (the scale is assumed as %)

Ways are constantly changed and with the decrease in the number of ways, the hit ratio increases.

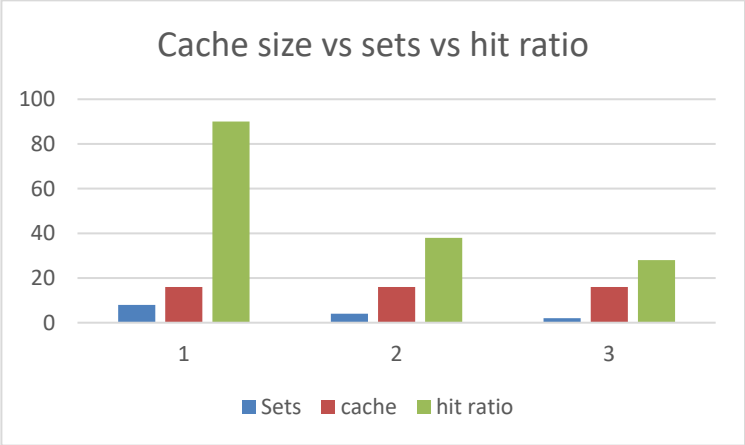

Figure 4.15 Graph 3

Different cache values vs same sets vs hit ratio

Cache size – constant (16mb)

Sets – varying (in the graph, the scale for sets are assumed as 1000s)

Hit ratio – varying (the scale is assumed as %)

The hit ratios are mainly effected by the changes in the sets, but where unaffected by the changes in the size of cache

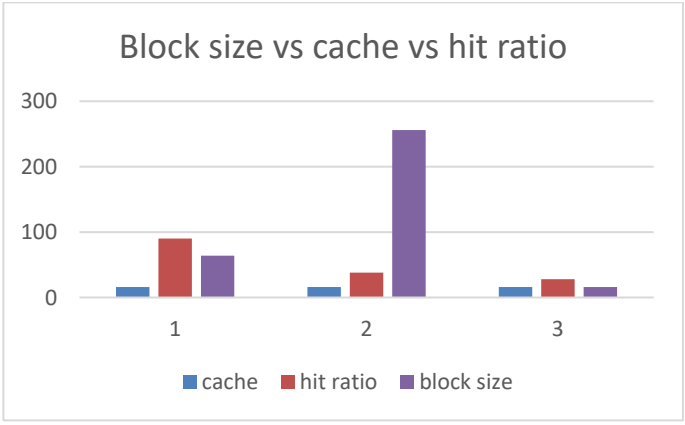

Figure 4.16 Graph 4

Block size vs cache vs hit ratio

Block size – (varying from 64 to 256 to 16)

Cache size -constant (the scale is assumed to be in mb)

Hit ratio – varying (the scale is assumed in percentages)

The size of the block does not effect the hit ratio as we have not given any data here.

I have tested the hit ratios with the cache size kept constant and the block values changing in the first scenario and block values kept constant and the cache size kept constant

Hence the hit ratios mainly depend upon the size of the sets or the ways.

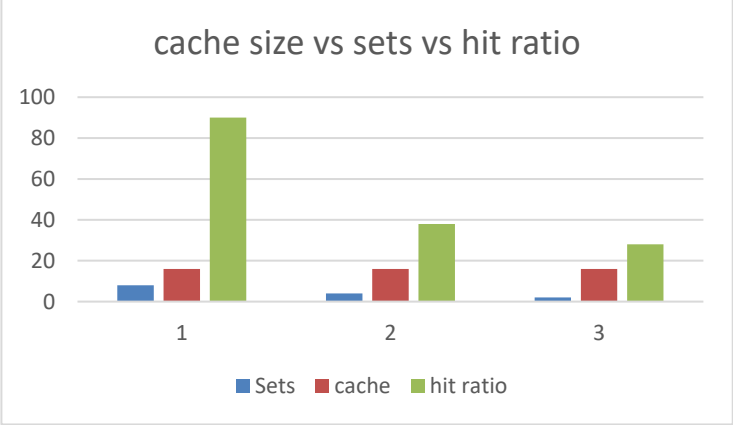

Figure 4.17 Graph 5

Cache size constant vs sets changing vs hit ratio of instruction cache.

Cache size – constant (16mb)

Sets – varying (the scale for sets is assumed to be in 1000s and the values range from 2k,4k to 8k)

Hit ratios – varying (the hit ratio for instruction cache is varying from 90% ,40% to 30%) .

It can be seen that the hit ratio decreases with decrease in the number of sets.

4.6 Calculations

Cache size = block size * number of lines

Number of lines = number of sets * way

If the number of sets = 2^a, then ‘a’ index bits are required

If block size = 2^b, then ‘b’ offset bits are required

Index bits + tag bits + offset bits = 32 bits microprocessor(here)

Number of lines = cache size / block size

For the 16mb cache size,

16mb = 2^{24}

Cache size = block size * number of lines

$2^{24} = 2^{10}$ * number of lines (assuming constant block size)

Number of lines = number of sets * ways

Number of lines = 2^{14}

2^{14} = number of sets * ways (starting from the highest way)

2^{14} = number of sets * 2^3

Number of sets = 2^{11} = 2k sets

Hence the index bits= 11

As the block size is 2^{10} , the offset bits are 10

As Index bits + tag bits + offset bits = 32 bits microprocessor

Tag bits = 11

Hence, the final specifications are

Cache size- 16mb

Tag bits- 11

Index bits – 11

Number of sets – 2k

Block size – 2*10

Way-8

Data Hit ratio- 49

Instruction hit ratio- 28

For the 64mb cache size,

64mb = 2^{26}

Cache size = block size * number of lines

$2^{26} = 2^{12}$ * number of lines (assuming constant block size)

Number of lines = number of sets * ways

Number of lines = 2^{14}

2^{14} = number of sets * ways (starting from the highest way)

2^{14} = number of sets * 2^{3}

Number of sets = 2^{11} = 2k sets

Hence the index bits= 11

As the block size is 2^{12}, the offset bits are 12

As Index bits + tag bits + offset bits = 32 bits microprocessor

Tag bits = 9

Hence the final specifications are

Cache size- 64mb

Tag bits- 9

Index bits – 11

Number of sets – 2k

Block size – 2*12

Way-8

Data Hit ratio- 49

Instruction hit ratio- 28

For the 4mb cache size,

4mb = 2^{22}

Cache size = block size * number of lines

$2^{22} = 2^6$ * number of lines (assuming a block size)

Number of lines = number of sets * ways

Number of lines = 2^{14}

2^{14} = number of sets * ways (starting from the highest way)

2^{14} = number of sets * 2^2

Number of sets = 2^{14} = 16k sets

Hence the index bits= 14

As the block size is 2^6 , the offset bits are 6

As Index bits + tag bits + offset bits = 32 bits microprocessor

Tag bits = 12

Hence the final specifications are

Cache size- 4mb

Tag bits- 12

Index bits – 14

Number of sets – 16k

Block size – 2*6

Way-4

Data Hit ratio- 99

Instruction hit ratio- 99

Chapter 5 Conclusion and Future Scope

5.1 Conclusion

Hence, a split instruction/data L1 cache for a 32-bit processor in a system was simulated. MESI protocol was implemented to ensure cache coherence. Different test cases were defined with decreasing number of sets. There was a decrease in the hit ratio , with the decrease in the number of sets. In general, having more sets in a cache can lead to better performance. This is because with more sets, the cache can store more blocks and reduce the likelihood of conflicts, which occur when multiple memory blocks map to the same cache set. On the other hand, having fewer sets can reduce the complexity and power consumption of the cache, as well as the cost of the hardware. However, this comes at the expense of cache capacity and potentially higher miss rates due to conflicts. Therefore, the choice of the number of sets in a cache involves a trade-off between capacity, performance, complexity, power consumption, and cost. Hence 16K sets in each of cache is considered to be optimal. I have tested with changing blocks and ways, but what impacted the hit ratio mainly was the number of sets. LRU replacement policies was employed on both the caches and Snoop protocol is used and statistics regarding the number of reads, writes, hits, and misses as well as a hit percentage rate are generated.

5.2 Future Scope

The following are few of the future scopes

- As I have tested the MESI protocol, MOESI and MOESIF protocols can also be tested by adding the additional owned and forward cases. A comparison can be done amongst the 3 different protocols.
- The trace files only consisted of commands and address values. Changes can be made to added actual data values and then check how it effects the hit ratio, when data values of different sizes are included.
- LRU was used in this project, MRU and random replacement policies can be used and the hit ratio can be compared.
- For this project a 32 bit processor was considered, changes can be made to test for a 64 bit processor and change the cache size and sets accordingly.

REFERENCES

[01] Attada Sravanthi1, Ch. Rajasekhara Rao2, K. Krishnam Raju3, L. Rambabu4, Implementation of MESI protocol using Verilog

[02] Formal Verification of a mesi-based cache implementation: a thesis by Venkateshwara Kottapalli.

[03] Performance comparison of cache coherence protocol on multi-core architecture by A Tiwari

[04] Manjush P. V., "Split L1 Cache Simulation," Project Report, M. S. in Digital Systems Design, Vellore Institute of Technology, 2014.

[05] "MESI-ISC: MESI cache coherency protocol with an integrated shared cache,"

[06] Nguyen, Anh. "Cache Coherence Simulation." University of Colorado Denver, CSCI 5593: Computer Architecture, 2017.

[07] Gujar, K., & Solanki, U. (2019). Automatic Toll Tax System Using RFID. International Research Journal of Engineering and Technology, 6(6), 1545-1548.

[08] Marathe, R. (2019). L1 Cache Simulator using MESI Protocol