Cache Coherence Protocols in Multi-Processor

Azilah Saparon, and Fatin Najihah Bt Razlan

***Abstract*—**This paper describes the cache coherence protocols in multiprocessors. A cache coherence protocol ensures the data consistency of the system. Typical modern microprocessors are currently built with multicore architecture that will involve data transfers between from one cache to another. By applying cache coherence protocols to each of the caches, the coherency problem can be solved. With this resolution, simulations of the applied cache coherence protocols can be each presented to walk-through the coherency processes. This simulation is developed based on Verilog Coding and implemented using Xilinx Software. Using the same software, test benches was constructed to verify the functionality for each of the protocols. The cache coherence protocols consist of read operations and writes operations of the cache.

***Keywords*—** cache coherence protocols, snooping, MSI, MESI, MEOSI, memory architecture, direct-mapped cache.

I. Introduction

AS Moore's Law [2] predicts, hardware is becoming progressively smaller and execution times quicker. The current hardware world is dominated by the multi-cores or many-cores [3]. As the trend shifts from single-core to multi-core processors for tuning up the performance, system architectures have several alternatives on one of the most important system resources-the cache. In multiprocessor architectures caching plays a very important role and it is actually the key to the performance of the processor. In all-cached architectures, any information is transferred into the primary cache first before being used. Unfortunately, the presence of writable shared information in caches introduces the problem of Cache Coherence. The cache coherence problem arises from the possibility that more than one cache of the system may maintain a copy of the same memory block. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion. Therefore, some basic protocols are adapted in order to eliminate the problem of cache coherency in the memory system such as Snooping Protocol and Directory Protocol. Accordingly, different multiprocessors based systems implement different cache coherence protocols that have given birth to different protocol verification logics.

Azilah Saparon is with the Faculty of Electrical Engineering, Universiti Teknologi MARA, Shah Alam 40450, Selangor, Malaysia (e-mail: azilah574@salam.uitm.edu.my).

Fatin Najihah Razalan, was a student at Universiti Teknologi MARA. She is now with Intel Corporation, Jalan Sultan Azlan Shah, Kawasan Perindustrian Bayan Lepas, 11900 Bayan Lepas, Pulau Pinang, Malaysia(e-mail: fatinnajihah_razlan@yahoo.co.uk).

This paper studies the cache coherence mechanism in multiprocessor environment by designing a simple cache and memory to serve as a platform to implement the cache coherent protocols which also focuses only in write invalidate type of Snooping Protocols such as MSI, MESI, and MOESI. With the simulations obtained, the changes between states of each of the coherence protocols can be witnessed and analyzed for educational purpose and to further the development of the invented protocols.

No doubt that multicore and many core systems rely on inter-core communication via shared memory. In such systems it is necessary to make sure that data consumed by all the cores is up to date. Cache coherence protocols help ensures this [4]. In other words, the correct operation of these applications thus depends on the correctness of the cache coherence transactions. However, verifying the correctness of these transactions is not insignificant since even simple coherence protocols have multiple states [5]. This research focuses on the study of cache coherence in CMPs (Chip Multiprocessors).

Section II, III, and IV describes the architecture of the cache, cache coherence protocol, and architecture of the memory involved in this project respectively. Section V shows the implementation and results of the simulations. The last section VI concludes all the findings.

II. The Cache

A cache is a small size and high speed memory that stores data temporarily from some frequently used addresses which is in the main memory. It is used in modern CPUs to shorten the time cycle required by the processor to fetch the requested data thus effectively improving the CPU performance.

A. Architecture

Fig.1 shows a simple direct-mapped cache with four-word block size purposely designed for this project. The cache constructed has 8 sets, each consist of four 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 34 bit address which consists of bit 0 to 1 *byte offset*, bit 2 to 4 *block offset*, bit 5 to 7 *set* and the rest bits for the *tag*. The *byte offset* is used as function of the cache for it to be able to read variable data for example CPU requesting only 1 byte or 2 byte data and the *word offset* on the other hand, used to select any words inside the cache. The *tag* bit is for representing the tag address requested by the CPU and to be compared to the tag addresses inside the cache.

http://dx.doi.org/10.15242/IIE.E1014020

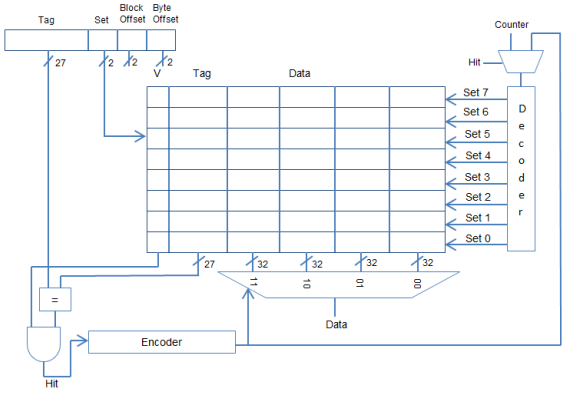

Fig.1 Direct-mapped cached

B. Write Operation

The write operation of the cache started off with comparing between the tags. If there is a hit, the multiplexor will select inputs from the encoder used by the data multiplexor. On the contrary, if there is no hit, the multiplexor will select inputs from the counter module where it will supply the multiplexor which also functioned as set select multiplexor for the decoder to enable any empty set for the new data to be written. The sets are selected based on the set select address for example, set select address for 000 is for set 0, set select address for 001 is for set 1, and so on.

C. Read Operation

For the read operation, the process begins with comparing the CPU tag address with the tag addresses inside cache using a comparator. It will be a hit if the CPU's tag address match to one of the tag address inside the cache. Unfortunately, that alone will not be able to give the data because the tag register has to be valid too in order to get the data from the cache because the valid bit indicates whether the set holds meaningful data. Therefore, if the valid bit is 0, the content is meaningless. If a hit with valid bit is detected, the signal is asserted into the encoder to guide the data multiplexor to select the data from the sets. There should be only one hit at a time since every set should hold different data inside them. No hit meaning no data. The data output obtained from the data multiplexor can be chosen whether it should provide 1 byte, 2 byte, 3 byte and 4 byte depending on the CPU request.

III. Math

The main memory is relatively larger in size and slower in terms of the speed compared to the cache. Any access towards the main memory takes longer average time because of its large size.

A. Architecture

The main memory used for this project is 4Kbyte. The architecture is much simpler compared to cache where contains data with their addresses and does not need any states compared to the cache and the protocols. The data is accessed by the cache through a 32-bit address line. The size of data for the main memory is in the same size with the data from the cache since the point here is just to demonstrate the transitions between the cache and the main memory with the cache protocols.

B. Operation

The main memory also has simpler operation compare to the cache. Its function is just to store the data with their addresses. Whenever a *miss* in *invalid state* of a cache occur, the cache will have to access the main memory and write it contents to the cache. To be able to write to the main memory, *write* enable signal will be asserted and the data is stored referring to the address given. The address is word aligned and the data is 4 byte on each set. On the other hand, for *read* purpose, the main memory will receive the input from the cache and provide the data according to the address requested as the output.

IV. The Cache Coherence Protocols

The cache coherence snooping protocol is one of the techniques of maintaining the coherency between the caches in multi-processor environment using hardware. The protocols implemented inside of the cache rely on a shared bus between the processors for coherence. For this project, three protocols are brought to test which are the MSI, MESI and MEOSI protocols. Each of the protocols is tested with the designed cache together with the memory to demonstrate their function. The input signals for the protocols consist of *read hit (RH)*, *write hit (WH)*, *snoop hit on read (SHR)*, *snoop hit on write (SHW)*, *read miss shared (RMS)*, and *read miss exclusive (RME)*. The *read hit* and *write hit* represents if the CPU's request feedback a *hit* while reading and writing to the cache respectively. *Snoop hit on read* and *snoop hit on write* represents the signals for the *snooping* of the bus process which results in *hit* from another cache during read or write operation. The *read miss in shared* and *read miss in exclusive* represents the signal that results from detecting the state of the other cache from an *invalid* state of the local cache.

A. MSI

MSI protocol is the simplest invalidation-based protocols. It consists of 3 states only which is *Modified (M), Shared (S) and Invalid (I)*. Fig.2 shows the State Diagram of MSI protocol. The *Invalid* state is where the local cache does not have a valid copy. The *Shared* state means that the local cache and other cache possibly have a valid copy with respect to main memory. The local cache is in *modified* state if it is the only cache that has a valid copy.

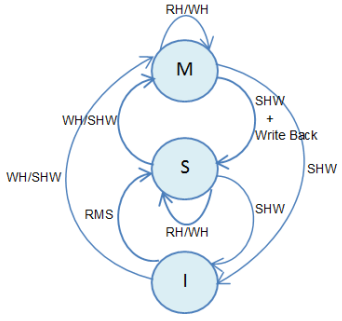

Fig. 2 State Diagram of MSI Protocol

B. MESI

This protocol is an advance to the MSI protocol which added a state of *exclusive (E)* to reduce the number of bus messages sent out for *invalid* to *modified* transition. The *exclusive* state is where the local cache has a valid copy including the main memory while no other cache does. This gives flexibility to the local cache to modify itself without needs for snooping from other caches. Fig.3 shows the state transition of the MESI protocol.

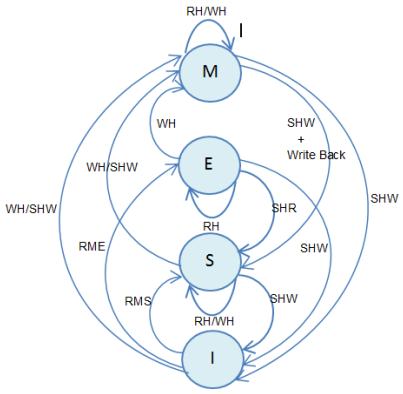

Fig.3 State Diagram of MESI Protocol

C. MEOSI

The further extension of MESI protocol is MEOSI protocol where it reduces the number of bus messages sent out for *Invalid* to *Modified* transition while still allowing multiple sharers. This protocol includes another state called *Owned* which the local cache has a valid copy and at the same time, other caches and the main memory may have a valid copy too. The owner can avoid lengthy memory access. There should be only one cache can be in the *Owned* state and the data can be shared among the caches. Fig.4 illustrates the state transition diagram of the MEOSI.

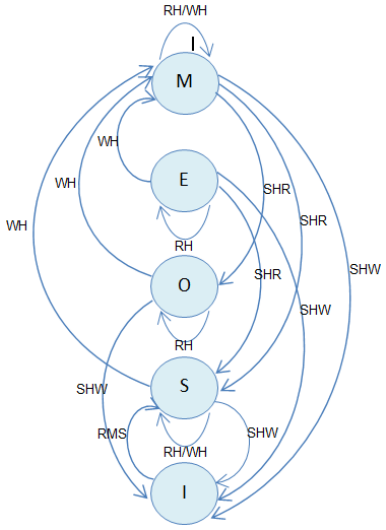

Fig.4 State Diagram of MEOSI Protocol

D. Operation

The coherence protocols consist of the *Fetch Data*, *Read Data*, *Write Memory*, *Write Cache*, *Read Cache* states combined with and some of the protocol states such as *Invalid*, *Modified*, *Shared*, *Exclusive* and *Owned* according to the protocol implemented for the designed cache.

i. *Fetch Data:* Here, all the input parameters will be initialized and reset conditions are being set. This state will be determined whether it is a write or read operation. A busy flag provided for the purpose of preserving the process of a read or write operation is also being kept down at this state.

ii. *Read Cache:* The data requested by the CPU is checked whether there is a *hit* in the cache. If there is a *hit*, the cache will supply the CPU with the requested data. If it is a *miss*, the cache will set the signal *Read Miss (RM)* to high and it will go to the *Invalid* state.

iii. *Invalid:* In this state indicates the cache is not up to date and the cache will send out signal to the bus to look at other cache. If the copy of the data exists in the other cache, the signal is named *Read Miss Shared (RMS)* will be asserted and it will enter the *Shared* state. On the other hand, if there is no cache copies exist, the signal *Read Miss Exclusive (RMS)* will be asserted and it will enter the *Exclusive* state. It will enter the *Modified* state if a *Write Hit* or a *Snoop Hit on Write* is asserted.

iv. *Shared:* From this state, can enter the *Modified* state only if a *Write Hit* or *Snoop Hit on Write* is asserted. This state represents if the local cache has a copied data that another processor also has.

v. *Exclusive:* This state is where the local cache is the sole owner of the data and it is free to modify the data without a bus message. If a *Snoop Hit on Read (SHR)* is asserted, the state will be changed to *Shared* state and if a *Snoop Hit on Write* is asserted, the state will change to *Invalid* state.

vi. *Modified:* This is a state where the *Shared* data has been modified. . It will change its state to either *Shared* or *Invalid* state if there is the signal *Snoop Hit on Write* asserted.

vii. *Read Memory:* In this state, the data from the memory is being read and the next state for the read memory would be the *Write Cache* state.

viii. *Write Cache:* Data either from other cache or the memory is written into the local cache in this state. It will use the data stored in the temporary register. If the state results from a *miss*, it will return back to *Read Cache* state or else it will go back to *Fetch Data* state for the next instruction.

ix. *Write Memory:* This state resulted from the *write enable* signal asserted from the *Fetch Data* state. Next state of this state would be *Write Cache* state.

V. Result and Analysis

In this section, it will be divided into three sections. All of the sections describe about the testing the functionality of the cache, memory, and the cache coherence protocols. The whole designs were simulated using ISim software provided by Xilinx ISE Design Suit.

A. Testing the Cache

The testing for the cache involved the basic function of direct-mapped cache with 4-word block size with the basic read and write operations. This part used the most plenty coding compare to the other designs. In order to implement the cache with the protocols, it must be properly designed so that it will demonstrate the flow of the coherency process. The problem when designing this cache arises when determining how to enable any empty set and read the *hit* data from the cache according to the set. The problem was solved by adding multiplexors that holds the *enable* bit and *hit* bit according to the sets. A direct-mapped type of cache was chosen as the design since it is the simplest cache compared to other design cache such as *set-associative* and *fully-associative* cache. This is also because the direct-mapped design has the highest miss rate thus suitable to implement the coherent protocols. A simple counter is used as a replacement method rather than using LRU where it will change to the next set of cache after filling in the data.

i. *Write Test*: The write test begin with inserting inputs shown in Table 1. The inputs consist of the tag address, set, and the data to be written into the cache.

TABLE I
INPUT DATA FOR WRITE TEST

Data in	*Tag*	*Set*
1111111	1111111111_000000000_11111111	000
f0f0f0f0	1000000000_000000000_11111111	010
aabbccdd	1010101010_000000000_11111111	010

Using the inputs above, the write operation for this cache shows the correct destination written into the sets in the cache.

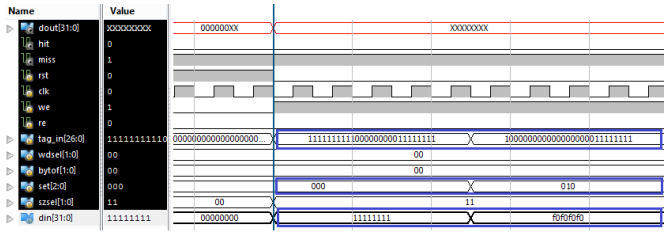

Fig.5 Results of write operation and its destination

Based on the waveform of Fig.5, the *tag address 1111111111_000000000_11111111* was successfully written the data 11111111 in set 0 (3'b000). The data f0f0f0f0 also successfully written in set 2 (3'b010) with *the tag address 1000000000_000000000_11111111*. Since the objective for this test is just to test on the write operation, the use of *byte offset* bit and *word select* bit was not being manipulated for this operation.

This write test operation also should include the functionality of the re-write policy where the data can be replaced on the same set by a different data inserted. Fig.6 shows the simulation of re-write policy. From the waveform it can be seen that a new data was successfully written into the same set on the cache which proves the functionality of the policy. Below shows the data *aabbccdd* was successfully written into set 2(3'b010) with a *tag address* of *1010101010_000000000_11111111* replacing the old data *f0f0f0f0* with the *tag address 1010101010_000000000_11111111*.

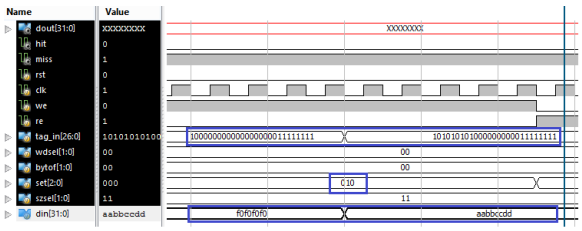

Fig.6 Results of re-write policy

ii. *Read Test:* For this test, the cache is verified whether it can fetch the same data that is stored during the previous write operation test. This test is continuity from the write test where the same input is reused to check whether this cache can supply the correct output data. For this test, the the first *tag address* will be compared to the *tag* of the cache is *1111111111_000000000_11111111*. The first and second attempt to read the cache does not gives any data since there are no *tag address* included thus it cannot compare the *tags.* Fig.7 shows the data from *set 0* (3'b000) with *wdsel* of 0 and *wdsel* of 1 gives back the data 00001111. This is because the data written earlier is only in *word 0* and 1 of the *set 0*. The input for the data size is as Table II below. Notice that the *dout* . Since the maximum size of the output is 4 byte, the output will result in 4 byte data

TABLE II
SIZE OF DATA

Data Size	szsel
1 byte	00
2 byte	01
3 byte	10
4 byte	11

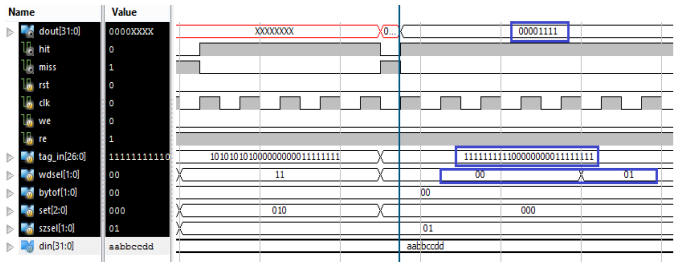

Fig.7 Results of read test

B. Testing Main Memory

The testing of the main memory involves simple function of the read and write operation. Table III below shows the input data supplied to the memory for the simulation.

TABLE III
INPUT DATA FOR MAIN MEMORY

Address	*Data In*	*Array Occupied*
00000000	*f0f0f0f0*	RAM[0,31:0]
00000004	*aabbccdd*	RAM[1,31:0]

Referring to the Fig. 8, when the signal *write enable (we)* is set to high, it enables the data to be written into the memory. Based on the table, when the input *address, a* is *00000000*, the data to be written, *wd* is *f0f0f0f0. S*ince the data is word aligned, the *data* will be occupied in the first array of the memory. The second *data aabbccdd* will be placed in the second array of the memory which is *00000000*. During the read operation, the signal *we* were set to low to enable the memory to be read. To test if the memory is being correctly written, the read operation was tested right after the write operation. Simulation below shows that when the *data* requested from the *address, a* of *00000000* will result in *f0f0f0f0.* When the *data* requested from the *address, a* of *00000004* it will result in *aabbccdd. This shows that the* main memory can perform the correct read and write operation.

Fig. 8 Read and write operation

C. Testing the cache coherence protocols

During testing the cache coherence protocols, it does not require much time since the operation of the protocols is quite simple although there are a lot of states in it. This test was conducted using a different approach from other tests. The test bench used to verify the function of the protocols was constructed in a different way by making as if there are two processors (also means two cache) involve with the protocols. The *processor* is named as *PA* for *processor A* and *PB* for *processor B.* The protocol was made as if it is implemented to both of the *cache* in *processor A (CacheInSectorA)* and *cache* in *processor B (CacheInSectorB).* Both of the *caches* are enable by the *processors* simultaneously with the current state of the *local cache* and the other *cache.* The states of both of the *caches* are determined by the input signal asserted to the protocol such as *RMS, RME, RMO,* which represents the signal *Read Miss detecting Shared, Read Miss detecting Exclusive,* and *Read Miss detecting Owned* respectively are responsible for the change in states from an *Invalid* state. The *Write Hit (WH), Read Hit (RH), Snoop Hit on Read (SHR), and Snoop Hit on Write (SHW)* are responsible for the change into other state for the *Modified, Shared, Exclusive* and *Owned* states. Figure 9, Figure 10 and Figure 11 shows the waveform for the MSI, MESI and MEOSI protocols to demonstrate the changing between the states.

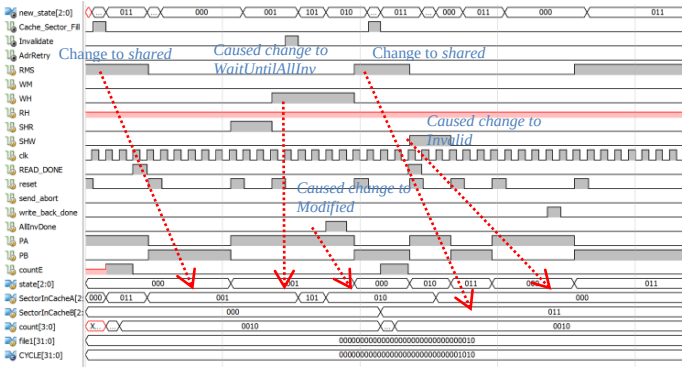

Fig. 9 Simulation results for MSI protocol

Figure 9 shows the simulations of the waveform for MSI protocol. This process starts with *Processor A (PA)* starts on *Invalid (3'b000)* state to *cache fill (3'b011)* and change to *shared (3'b001)* state after aquiring the data from memory. The *cache in sector A* later changes to *waits until all cache invalidate* when a *write hit* is asserted. The state then change to *Modified (3'b010).* The *cache in sector B* stays *Invalid (3'b000)* until a signal *snoop hit on write* signal is asserted where it will effect the *cache sector in* A to change to *Invalid (3'b000)* to write cache line back to memory.That is when *cache sector (3'b011) in B* will start the process of *cache fill.*

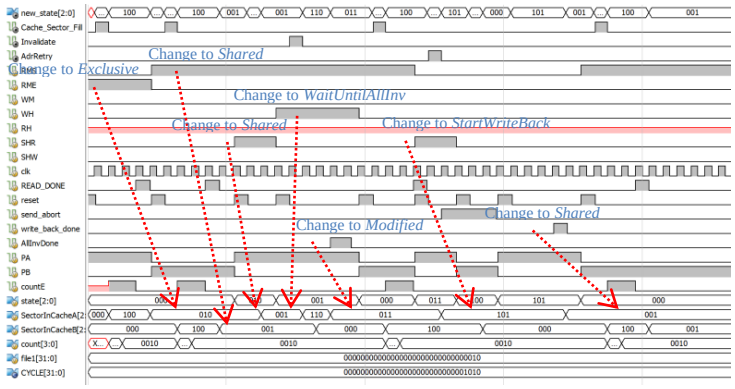

Fig. 10 Simulation results for MESI protocol

Figure 10 shows a situation when the *Processor A (PA)* is on, the *present state* of the sector is *Invalid.* Theoretically, the local processor will put the requested *address* on the bus and start reading from the memory. The *state* of *cache in sector A* changed from *Invalid* to *cache fill (3'b100)* and then *Exclusive (3'b010)* for having the only copy of data. After a certain delay, the *Processor B (PB)* will declare *Snoop Hit on Read* operation by the *cache in sector A.* The *cache in sector A* changed to *Shared* state and it will then declare a *Write Hit* and the *cache in sector B* is now on *cache filled state (3'b100)* and later changed into *Shared* state for having a copy of data from other cache. After the *cache in sector B* finished the process of snooping, the *cache in sector A* state change from *Shared* to *Modified (3'b011)* as its data was being modified after it start to *write back* by enabling the *write back done* signal and after the process completed, the signal will be disabled again. During that process, the state of *cache in sector B* become *Invalid* and then change to *cache fill* while

cache in sector A become *Shared.* The *sector in cache B* tries to read the *address again* from the *cache in sector A.*

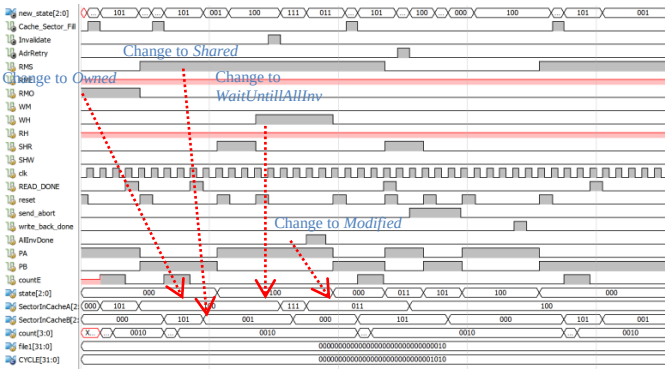

Fig.11 Simulation results for MEOSI protocol

Same goes to Fig.11 where the test bench used is quite the same only that here, the *Processor A* is tested in the added "*Owned*" state. *Sector in cache A* like always, starts with *Invalid (3'b000)* state and went into *cache fill (3'b101)*. After sending the address to the bus and reading from the memory, the *sector in cache A* changed to *Modified* state for having the valid copy of the data but at the same time, other cache could have the valid data too. The *Processor B* on the other hand, starts at a *shared (3'b001)* state. After a while, *Processor B* again declared of having *Snoop Hit on Read* when the *Processor A* turned on which then leads changing states of *Owned* into *wait until all invalid (3'b111)* state and later changed to *Modified (3'b011)* state. The *Processor B* again sends the signal *Snoop Hit on Read* which then results in the change state of *sector in cache A* to be in *Owned (3'b100)*. From the simulation, it can be verified that the changing between the states are successfully followed the state transition diagram showed in the earlier section of this paper.

VI. CONCLUSION

Based on the simulation that had been done, it can be concluded that the design for each components are working properly. Unfortunately the design has not yet implemented in real life system such as Spartan board. Further development of this project can be improvised in terms of the functionality and the flow of the design.

ACKNOWLEDGEMENT

The author would like to acknowledge with gratitude, UiTM Research Management Institute (RMI) and Faculty of Electrical Engineering, UiTM for supporting this work under Science Fund code (01-01-01-SF0436).

REFERENCES

[1] P. A. Archibald and J. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. ACM Transactions on Computer Systems", 1986, 273--298. http://dx.doi.org/10.1145/6513.6514

[2] G. E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, vol. 38, April 1965.

[3] Xin Lai, Cong Liu, Zhiying Wang and Quanyou Feng, "A Cache Coherence Protocol Using Distributed Data dependence Violation Checking in TLS", in IEEE Conference Publications for Second International Conference on Intelligent System Design and Engineering Application, 2012.

[4] Hennessy.J and Patterson. D, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 3rd edition, 2003.

[5] Austin, T.M.; , "DIVA: a reliable substrate for deep submicron microarchitecture design," Microarchitecture, 1999. MICRO-32. Proceedings.32nd Annual International Symposium on , pp.196-207, 1999.

[6] Fernandez-Pascual, R. et al.; , "A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on , pp.157-168, 10-14 Feb. 2007.

[7] Meixner, A.; Sorin, D.J.; , "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on , pp.145-156, 10-14 Feb. 2007.

[8] Borodin, D.; Juurlink, B.H.H.; , "A Low-Cost Cache Coherence Verification Method for Snooping Systems," Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on , pp.219-227, 3-5 Sept. 2008.

[9] Hennessy, J.; and Patterson, D.; , "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 3 edition, 2003.

[10] Meng Zhang; et al.; , "Fractal Coherence: Scalably Verifiable Cache Coherence," Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on , pp.471-482, 4-8 Dec. 2010.