MULTI-SIZED OUTPUT CACHE CONTROLLERS

Mohd Naqib Bin Johari
Faculty of Electrical Engineering
Universiti Teknology MARA
benbon88@gmail.com

***Abstract* – This thesis describes the design of a Multi-sized Output Cache Controller that will handle 2Kbyte 16 ways with 4 word block size cache. A cache controller is a device that used to sequences the read and write of the cache storage array [1]. Most of modern microprocessor is designed with multiple core architecture that will lead to massive traffic of cache data transfer. By taking the advantage of using temporal locality and spatial locality to the cache, the problem can be solved. With this solution, a controller that capable to handle huge amount of way and block size need to be designed. It also should have the capability overcome the cache coherence. This design will be implemented using Xilinx software. It was developed base on Verilog coding. Using the same software, a test bench was constructed to test the functionality of the controller. This cache controller consists of four stages, from request to read data. It had the capability to read and write to different agent on various output data size from 1byte till 16 byte.**

***Index Terms* – cache controller design, cache design, memory architecture, set-associative cache.**

I. INTRODUCTION

Throughout the last decades, the technology of digital electronic have become more advance. As the times goes on, this advancement have made the computer and other electronic hardware such as mobile phone, PDA and many electronics gadget become smaller, faster and cheaper to produce. Most of these devices are using microprocessor as their brain to control their operation. Nowadays, making a faster microprocessor is the main concern. One of the important components inside the microprocessor is the cache controller. As the microprocessor speed vastly increases, designing a much faster cache become very important [2][3].
One of the ways to improve the cache controller is by executing a pipelined cache controller [4]. However, this solution will increase the complexity of the circuit. Multicore architecture was introduced to increase the processing speed and had been widely use over the world due to its high performance [6]. As the multicore system allow to process multiple applications simultaneously, the cache controller was introduced to overcome the problem existed during all the cores sharing cache memory on a single dye [6]. However, the cache controller needs to be fast enough to deals with this massive data transfer between cache, memory and the processor. Increasing the size of the cache can increase the cache performance. But there is a trade-off, where the caches access time increases as the size increasing [7]. Nevertheless, most of caches earn a lot of benefit from larger cache size [8]. This cache controller is designed to have a capability to read and write data with three different agents. With the capability to handle the large size or space of the cache will help increase the performance as they reduce the miss penalty [7]. It will also able to overcome the cache coherence problem [5].

A. Problem statement

Applying multi-ways and multi block to the cache are the common method to reduce the miss penalty. However, it will lead to the complexity of the cache controller design. Besides that, the multi-core configurations that provide by most of microprocessor today also contribute to the complexness of the controller which leads to the increasing of the price and power consumption. Thus, simpler controller will be needed. To improve the processing performance, it also needs to handle the multiple output size.

II. THE CACHE

The cache is fastest memory available on the market due to its small sizes and architecture. It is the highest part in the memory hierarchy tree. But it is also the most expensive among them all.

A. The Architecture

Figure 1 below shows the architecture of the cache designed for this particular project. The cache implemented here has a memory of 2 kbyte. It is 16 ways set associative cache with a block size of 16 byte. This mean that the data corresponding with 16 different tag with the same line. Each way has the capability to hold 128-bit on each set. There are 8 set for each of ways.

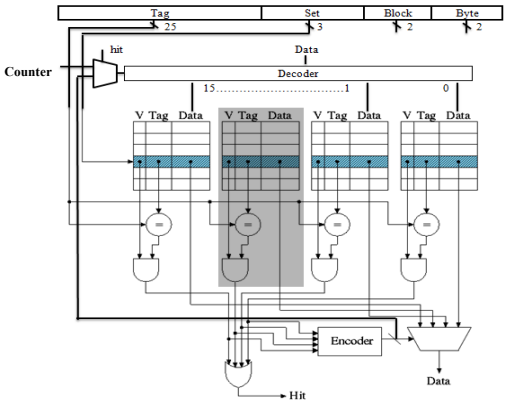

Fig. 1 The block diagram for the cache architecture.

Since the cache needs to provide variable data from 1-bit till 16-bit, the byte offset for the address is used. The bits 0 to 1 are called *byte select address*. It is used when the CPU requesting only 1 byte or 2 byte data. The bits 2 to 4 are *block select address*. These bits are used to select the any word inside the caches. The bits 5 to 7 are called *set select address*. These bits are used to select the set in every ways inside the cache. The rest of the bits are *tag addresses* for the tag register. Each set in each way has its individual tag addresses which used to identify each data inside the cache.

B. Write Operation

To write the cache, first the way will selected by a special replacement algorithm. Since the main objective of this thesis is to build a cache controller, the counter will be used as the replacement method instead of LRU (*least recently used*) replacement policy. The counter will feed into a *way select multiplexor* which selected by a *hit* bits before feed into a decoder. If there is a hit, the multiplexor will select the input from hit encoder (the same selector as the *data multiplexor* used). If it is not, the multiplexor will select the input from the counter module. The decoder will enable any empty way or any way with the same input tag register base on decision made by the *way select multiplexor*. The set of each ways is selected based on *set select address*. For instance, if the *set select address* is *001*, set number one will be selected.

C. Read Operation

To read the cache, all the ways will be enabled. The *set select address* will decide which set will be used. Base on the *tag address* provided by the CPU, it will be compared using comparator inside each way. If there is the same tag register inside the cache, it will be a hit. But the tag register must be also valid, if it is not, means there is no data for that register and no hit. The *data multiplexor* will select the way which asserted the hit. These multiplexor select the data via the data provide by the *hit encoder*. There should be only one hit at a time since one each way should be consists different register. If there is no hit, no data will provided. If there is a hit, the output will be taken from the data from *data multiplexor*. But this is happen when the CPU request the 16 byte data output. Remember that the controller should provide 1 byte, 2 byte, 4 byte, 8 byte and 16 byte. Whenever the CPU request an 8 byte data output, the data from *data multiplexor* will feed into another multiplexor called *8 byte multiplexor*. This multiplexor will select only 8 byte of the address given. If the CPU request a 4 byte data output, the data from *8 multiplexor* will feed into another multiplexor called 4 *byte multiplexor*. If the CPU request a 2 byte data output, the data from *4 multiplexor* will feed into another multiplexor called 2 *byte multiplexor*. If the CPU request a 1 byte data output, the data from *2 multiplexor* will feed into another multiplexor called 1 *byte multiplexor*. The final data will provide trough the output.

III. THE MAIN MEMORY

The main memory is slower compare to the cache due to its huge sizes and architecture. It is the lowest part in the memory hierarchy tree.

A. The Architecture

The total main memory has 4096 bytes of data. The data is arranged one byte after another according to their address. Each address corresponds to a single byte of data. Unlike the cache, there are no states associated with the data. Data is accessed by the cache through a 32 bit address line.

The main memory use the different size of data compare to cache. In this project queues are designed to make up for this size mismatch. It will convert 8 byte input to two 4 byte before feed it to the main memory by queues it together. This will be explained later in the cache controller chapter.

B. Operation

The memory has simpler architecture compare to the cache since the data inside only consist of 4 byte on each sets, the address is word aligned. Whenever the controller wanted to write data inside the memory, the *write enable* will be asserted. And the data will be stored with correspond to the address given. For instance, if the address is *0x00000000*, the data will be stored in the first sets of the memories. If the address is *0x00000004*, the data will be stored in the second sets of the memories and so on.

To read the data, there is no need to enable anything to the memories. Just feed the *address input* with an address, the output will provide the data which correspond to the address.

IV. THE CACHE CONTROLLER

Cache Controller is a device that control the data transfer between cache, main memory and microprocessor. When the microprocessor sends an address to request data, the cache controller will check the data inside cache. If the data are available, the cache controller will send the data to processor. If the data are not present in the cache, the cache controller will fetch the data from the main memories and send to the microprocessor as well as the cache [9].

A. The Architecture

This cache will have four stages which is fetch data, read cache and main memory, write main memory and cache and provide data to the processor. All of these stages are designed using FSM (*Finite-state machine*). These stages were divided into few states. Figure 2 bellow shows the main state diagram for the cache.

As mentioned before, this controller needs to write 8 byte fill data path as well as read data size of provide 1 byte, 2 byte, 4 byte, 8 byte and 16 byte. This is where the main challenge come from which is to aligned variable data exchange between the memory, cache and CPU. This will be explained more detail in the operation section.

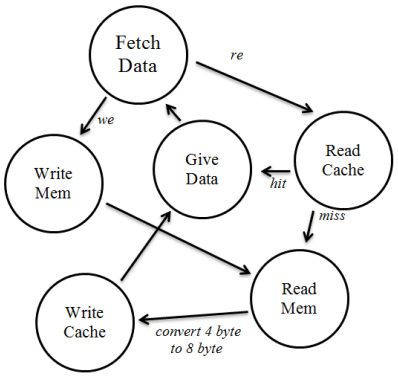

Fig. 2 The state diagram for the cache controller.

This controller also needs to deal with 3 different agents which is the processor. Since only one level of cache will be used, all the processor will share the same cache. To overcome the data transfer clash, the controller will provide the *busy* flag. This flag will asserted whenever the cache or memory were being accessed. Thus, the processor only can request the data if the *busy* flag is not asserted. This might make it slower but it will save more space and cost compare to the multi-level cache.

B. Operation

The controller consists of a few states; *Fetch Data*, *Read Memory, Write Memory, Write Cache, Read Cache and Give Data*. All this state just the main state where there are a few more states with in these states.

1) Fetch Data: At this state, all the controlling output and temporary register whether will be initialize, reset or emptied by the controller. As a data fetcher, it will check whether it is read or write operation. Within this stage, the controller will keep the busy flag down until it goes to other stage.

2) Read Cache & Give data: At this state, the cache will be checked. If there is a hit, the data found in the cache will be sent to the CPU. If there is no hit of miss, it will go to the next state which is *Read Memory* state.

3) Read Memory: In this state, there are another three states. These states are used to convert four 4 byte data memory to a single 16 byte data as mentioned in the memory chapter. In each of these states, it will do a word align address increment for requesting data from the memory. All four of the 4 byte data collected will be stored in a single 16 byte temporary register and pass it to the *Write Cache* states. This acts like a parallel to single data converter where is multiple smaller size data are combined together to form one big data size.

4) Write Cache: At this states, it will use the previously stored data in the temporary register and write it to the cache. If the state is access due to the cache miss, it will return back to the *Read Cache & Give data* state. Else, it will go back to *Fetch Data* state waiting for the next instruction.

5) Write Memory: Writing the memory will require another extra stage as the input data size is 8 byte. As the *Read Memory* stage, it needs to convert the data size first but into smaller size. First, half of the data (lower part) will be stored in the memory. Then, there will be an increment to the address with one word to store another half of the data (upper part) into the memory. This It will convert 8 byte input to two 4 byte where is the single line data broke into smaller multiple outputs to feed the smaller input size memory. The next state will be back to *Write Cache* state.

V. Result and Analysis

In this chapter, it will be divided into four sections. The first section will describe about the synthesizing and implementing the design. The second and the third section are to test the functionality of the design foundation which is the main memory and the cache. The last section will be the controller itself. The entire cache controller design (including the whole cache and the memory) were simulated using ISim software provided by the Xilinx software.

A. Synthesizing And Implementing The Design

Before simulate the cache controller, the design was synthesized first. This process was executed using the tools provided by the Xilinx software. Table I below shows the synthesis report of the *Device Utilization Summary* based from this design.

TABLE I. Synthesis Report

Device Utilization Summary			
Logic Utilization	***Used***	***Available***	***Utilization***
Number of Slices	3667	4656	78%
Number of Slice Flip Flops	2444	9312	26%
Number of 4 input LUTs	7202	9312	77%
Number of bonded IOBs	233	232	100%
Number of GCLKs	9	24	37%

Base on the result given, the design cannot meet the spec that been demanded. The development board reference that been used for this simulation was Xilinx Spartan 3E XC3S500E – FG320. This due to the number of the number of *Bond Input Output Blocks* (IOBs) used just a little bit exceeded the maximum capacity of the device package. The design was using a lot of input and output which cannot be supported by the package. This was cause by the spec required for the design. Not including the other input and output, the data output alone required 128 ports while the package only support for 232 ports only. However, the other logic utilization were still can be supported by the selected development board. As been observed, the number of slice only utilizes 78% of the board. Each slice contains a number of *Lookup Tables* (LUT's), flip-flops, and a carry logic element which make up the logic of the design before mapping. After mapping, all of the LUT's and flip-flops are

packed into slices. The other logic utilization also below the maximum limit of the board where number of Slice Flip Flop, 4 Input LUTS and *Global Clocks* (GLKs) only utilize 26%, 77%, and 37% respectively. From this result, by simply remove one of the outputs or inputs might allow it to implement the design to the FPGA. Unfortunately, during implementing the design, the result shows the different story. The Table II below shows some of the *Device Utilization Summary* results after the design were implemented.

TABLE II. IMPLEMENTING DESIGN REPORT

Device Utilization Summary			
Logic Utilization	***Used***	***Available***	***Utilization***
Number of SLICEMs	2672	2328	114%
Number of occupied Slices	4654	4656	99%
- containing only related logic	4654	4654	100%
- containing unrelated logic	0	4654	0%

These results were the additional result provided after implementing the design. The other results remain the same as previous result in Table I. From the observation, it tends to give more error when the design was implemented. This due to the number of the number of SLICEM used was exceeded the maximum capacity of the device package by 114%. Thus, the design was cannot be implemented as an error was occur. SLICEM (M = memory) can be configured to implement distributed memory or shift registers instead. Since this design consist of 4k byte of main memory and 2k byte of cache memory. It will take a lot of SLICEM to implement. All of this problem can be overcome by using the latest and more advance package such as Spartan6, Virtex6 and Virtex7 which provide more slices with their package.

The only reason that the design need to be implemented is that one of the given requirement demand the controller to be test its functionality using the logic analyzer. This can be done by implement it on the FPGA and check output on the FPGA using the logic analyzer. This will test the design can be work correctly in real life. Because of this limitation, it cannot be implemented on the FPGA.

There were many other problems that had been faced to achieve the requirement given. For most of it, it can be solved. One of it was, instead of using multiplexor, it was replaced with a decoder to select the way that should be used to write the data to the cache. This helps to reduce some logic utilization of the design. But there is one problem that was not able to be solved. There is a function that will enable the user to clear all the data from the memory. This is different from reset function as the memory should still be available as the reset function executed. Figure 3 shows the coding that had been designed to execute the clear function. If this function was implemented to the design, the number of slice will be utilizing the development board for more than 350%.

By assuming this weird behavioral was caused by the counter that been used to clear the data set by set, another coding that not using the *For Loop Statement* was been designed. It was replaced by specifying every set to be clear without any loop. But the result was remaining the same. Because of this matter, the clear function was not included to the design.

```
//CLEAR MEMORY, TAG REG & VALID BIT//
      if (clr)
            begin
               for(i=0; i<8; i=i+1)
                  begin
                     way[i]      <= 128'hxx;
                     val[i]      <= 1'bx;
                     tag[i]      <= 25'bxx;
                  end
            end
```

Fig. 3 Coding for clear function.

B. *Testing The Main Memory*

The testing will cover the basic function of the main memory which on how it would read and writes data to the memory. The data from Table III below shows the input data for the simulation and where it should be occupied. Noted that this memory was word aligned, making the next address will be *0x04* instead of *0x01*.

TABLE III. MEMORY INPUT DATA

Address	***Input Data***	***Array Occupied***
00000000	20202020	RAM[0,31:0]
00000004	22222222	RAM[1,31:0]

Figure 4 shows the waveform of the memory simulation. As seen from the Figure 4, when the *write enable* (*we)* is at high state, it will allow the memory to be written with the data provide from the input source which is *write data* (*wd*). It will locate the data precisely according to the provided input *address* (*a*). When *a* is *00000000* and *wd* is *20202020*, the first array of the memory will fill with *20202020*. This memory does not have any read enable function which mean that the data can be access any time which leads to faster operation. So, if *we* is low and *a* is *00000000*, the output which is *read data* (*wd*) will provide the stored data at address *00000000* which was *20202020*.

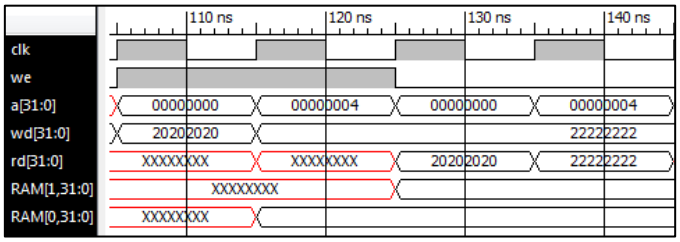

Fig. 4 Waveform of the memory read and writes simulation.

C. *Testing The Cache*

The testing will cover the basic function of a cache which write policy of a 16 way set associative cache and a basic cache read hit or miss test. Since the design consist lot of cache coding, this stage is very crucial as this where to determine the design was functioning correctly. A lot of development and troubleshooting were done at this stage. During the early stage of designing the cache, there is a problem on how to decide which way need to be selected to write and how to replace the old data when the address was updated with new data. Using a LRU is not an option as it would take a lot more time to design it. Besides, the main objective was designing a cache controller where the cache just the foundation of the design. A simple counter was used instead of using LRU. What it did is it will change to the next way after one way was filling with data. However, from the data analysis, even it had the same tag and set, it tend to write at the next way instead replacing the old data. This is where the decision has been made to add an additional multiplexor to choose the input for the decoder to select the way as been mentioned before in the cache chapter.

1) Write Test: The write test will follow data provided in the Table IV. As been explained before in cache architecture section, the tag and set were determined by the data address.

TABLE IV. CACHE INPUT DATA

Address	*Input Data*	*Tag*	*Set Occupied*
00000000	22222222 20202020	00000000	0
00000008	f3f3f3f3 f2f2f2f2	00000000	0
00000080	aaaaaaaa aaaaaaaa	00000001	0

Using the provided data, the cache was been simulated to write the data into the correct destination in the cache. Figure 5 shows the waveform obtained from *way0*.

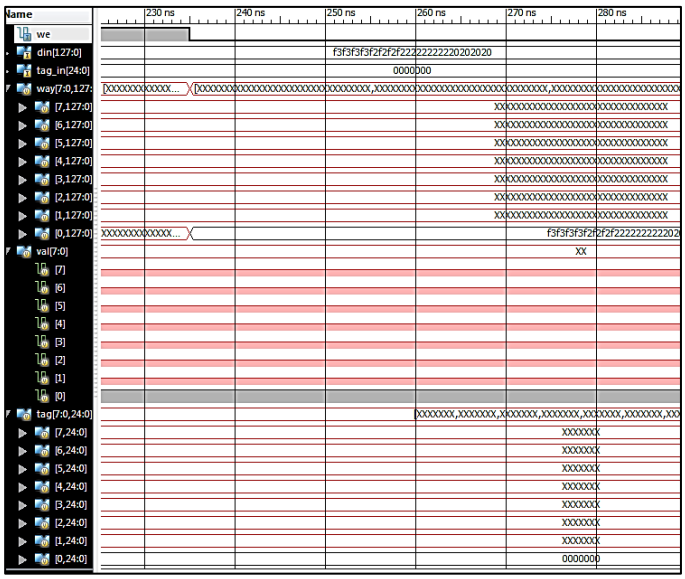

Fig. 5 Waveform of the cache writes simulation for *way0*.

As can be seen from the Figure 5, the data for address *0x00* and *0x08* were occupying the correct location. As the first data, it will select the first way which is *way0*. Remember that this 16 byte aligned, so both of this address data will share the same set and way as they have the same set number (*000*) and tag register (*25'b00*). Other than that, the valid flag for the first set was also asserted. The other locations were remaining empty. For the last data (*0x80*) it also located at the correct way which is the next way (*way1*) and set *000*.

To test whether the re-write policy working properly, the last data address in Table IV which is *0x80* had been replaced with the same address as previous data which is *0x08*. What should happen is it will replace the old data with the new one at the exact location in the cache instead of going to the next way as show in the Figure 6 below. This clarifies that this cache can replace the old data properly. The old data which is *f3f3f3f3_f2f2f2f2_22222222_20202020* was successfully replaced with the new data which is *aaaaaaaa_aaaaaaaa 22222222_20202020*. The lower part was remaining the same because only address *0x08* data was change while the address *0x00* still using the old data.

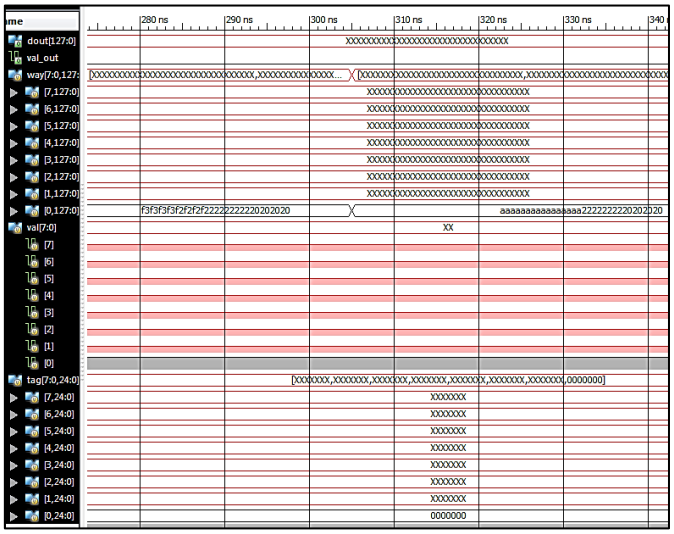

Fig. 6 Waveform of the cache re-writes simulation in *way0*.

2) Read Test: On this test, the cache will be checked whether it manage to fetch the same data that been stored during the previous test or not. For this test it will continue from the previous test. The functionality of this design output size selector was also going to be tested. To select the size, the *size select* (*szsel*) will be asserted according to Table V.

TABLE V. CACHE SIZE SELECT

Data Size	*szsel*
1 byte	001
2 byte	010
4 byte	011
8 byte	100
16 byte	101

Since the maximum size of the output is 16byte, all the output will be in 16 byte size format. What the data size do just choose the desired data with respect of the selected size. For example, if the stored data is *aaaaaaaa_aaaaaaaa 22222222_20202020* and the selected data is the second word with the size of 4 byte, the output will be *00000000 00000000_00000000_ 22222222* instead of *22222222*. The result from Figure 7 confirms that the cache can be read properly. It did provide the correct output as it supposed to. From the output waveform (see the second output from Figure 7), been observed that if the third word from cache set with a size of 16 byte was chosen, it will give the whole sets data including the first and the second word. Note that the cache was 16 byte aligned, thus the if the CPU choose for 16 byte output, it will only look at the address bit 4 and onward and the bit 0 till bit 3 will be neglected.

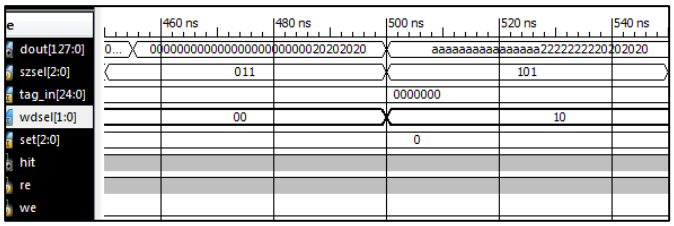

Fig. 7 Waveform of the cache read simulation.

D. Testing The Cache Controller

During the early stage of designing the controller, lots on test had been ran to do a lot of correction. The main problems come from the glitch which caused by the clock mismatch. Most of the data did not arrive at the same data as the other. This is hard to troubleshoot as there is so many ports and register need to check. Although not all the data arrive at the same time, it manages to get the correct output. Basically this test is almost the same with the cache test but with lesser input to deal with. During the cache test, the address were converted manually during the testbench which mean that all the tag register, set, word select and byte select manually asserted rather than asserted a single address. Differ from previous test, only the address need to be included and the controller will automatically convert it to those register. It also will handle the cache and the main memory at the same time. On this test, both read and write test for both cache and main memory will be executed at the same time. It will use the same data and address as the previous test. The Figure 8 shows the output of the executed simulation.

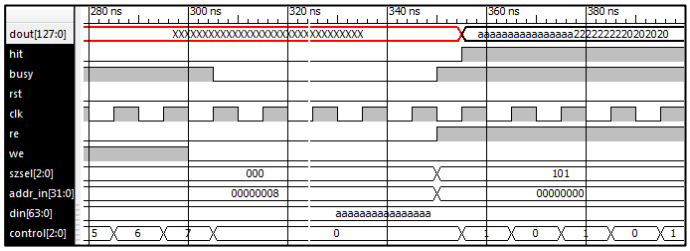

Fig. 8 Waveform of the cache controller simulation.

This result is just only a part of the total output as there is too many data of result to include in the figure. However, as observed from the Figure 8, it managed to call the same data with the same address as been used on the previous test. All the result was exactly the same. The FSM of the controller also work as it should. Inside of the cache and the main memory also had been checked and it did provide the same waveform as had been tested before on both cache and the main memory. Other than that, the busy flag also work without any hassle.

VI. CONCLUSION

Base from all the simulation that had been done, it can be conclude that the design was successfully functioning. Furthermore, the utilization problem might be solved and the design can be implemented in to real life if there is higher spec development board package around to work with.

ACKNOWLEDGMENT

First and foremost, my special thanks to Puan Siti Lailatul Binti Mohd Hassan who serving concurrently as my VLSI lecturer and supervisor for her remarkable guidance, support and advice to carry out my final year project. Many thanks to my Digital Design lecture Dr. Azilah Binti Saparon for providing me input, suggestion and knowledge throughout the study. Also, special thanks go to my beloved mother and my family for their encouragement and support during doing this project.

REFERENCES

[1] David E. Culler, Jaswinder Pal Singh, Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Gulf Professional Publishing, 1999, pp. 381.

[2] Roy W. Badeau, "A 100-MHz Macropipelined VAX Microprocessor", in *IEEE Journal of Solid-State Circuits,* Vol. 27, No. 11, November, 1992.

[3] Daniel W. Dobberpuhl, "A 200-MHz 64-bit Dual Issue CMOS Microprocessor", *in IEEE Journal of Solid-State Circuits*, Vol. 27, No. 11, November, 1992.

[4] Apoorv Srivastava, "1900-MHz CMOS 4-Kbyte Pipelined cache", *in IEEE Journal of Solid-State Circuits*, 1995, pp.1053.

[5] Vipin S. Bhure, Praveen R. Chakole, "Design of Cache controller for Multi-core Processor System", *in International Journal of Electronics and Computer Science Engineering*, pp.520.

[6] Alokika Dash, Peter Petrov, "Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering", *IEEE Journal of Solid-State Circuits*, 2006.

[7] Kunle Olukotun, "Multilevel Optimization of Pipelined Caches", *IEEE Journal of Solid-State Circuits*, October 1997.

[8] David Money Harris, Sarah L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann, March 2007, pp. 476.

[9] Badri Ram, *Advanced Microprocessors and Interfacing*, Tata McGraw-Hill Education, Sept 2001, pp.289.