Project #2

Simulation of CPU, Cache, Bus, and Memory Datapath

By: Jeff Grindel – A20237004

Subhi Beidas – A20245234

Instructor: Dr. Suresh Borkar

ECE 585-01

Due Date: 4/11/2013

**II. Executive Summary**

This report goes over what project the group was assigned with, how the group went about designing and implementing the project by making certain assumptions, backing up those assumptions with correct output data. The report finishes with a longer conclusion and all the related code attached.

**III. Introduction**

# Goal

The goal of this project is to build a VHDL simulation of a 32-bit version of the MIPS processor. The developed architecture is to include a CPU, cache, bus, and memory modules that emphasize the simulation of hit/miss and data-path cache scenarios for different types of instructions.

# Theoretical Background

**Cache:**In modern computing systems, accessing data directly from disk or memory is relatively time consuming or *expensive.* This could pose an issue if a set variables stored in memory are extensively required by the computing unit since accessing the respective variables could be time consuming. To deal with this, caches are utilized to minimize this overhead. Caches acts a less expensive intermediate storage block that stores blocks of memory that have been access, been accessed or are going to be accessed be the program.  If the requested data is in fact present contained in the cache a ***cache hit*** is identified, this request can be served by simply reading the cache, which is comparatively faster than accessing the memory. Otherwise, if the data block is not present a ***cache mis*s** is identified and the data has to be fetched from its respective location in memory. For efficiency, multiple caches are usually implemented to perform difference functions. Widely implemented abstractions usually involve the separation of cache into two main functions, *Instruction Cache* *and Data Cache.*

**Instruction Cache:** A stores instruction which helps in reducing the cost of going to memory to fetch instructions. In some other cases it also has other functions, such as branch prediction information. Instructions are only read/brought to the iCache and cannot be modified. So when the I-cache is full and a block of instructions is to be placed into the cache, it can usually over-wite anywhere in the cache.

**Data Cache** is a intermediate storage component that contains the application data that is going to be utilized by the processor. Data is loaded from memory into the data cache. The element needed is then loaded from the cache line into a register and the instruction using this value can operate on it.

Along with different cache functions, there also exist different algorithms or *writing policies* that dictate the interaction between the cache and the memory. In this project’s case the chosen writing policy implemented was the ***Write Back with write Allocate*** policy which involves the following:

- Write to main memory whenever a write-hit is performed to the cache

- If a write misses, allocate a line in the cache for the data written.

# Group Specific Specifications

**Instructions**

The MIPS processor is to support the three instruction formats of R, I, and J, along with store word and load word. A table was provided in the project specifications that included all the instructions to be designed



As seen in the table, there is a custom set of instructions that are to be implemented, which is chosen based on the last digit of the student ID’. In this groups case, the 4th set (BNE LUI) was chosen. Another group specific specification was the writing policy, which in our case was set to Write Through with Write Allocate.

**IV. Design**

The group decided after lots of discussion to split all of the required pieces of structure into separate modules and then combine them into an overall dataflow. This allowed for each individual component to be thoroughly tested. The overall dataflow can be seen below as well as in the appendix.



Figure X: Datapath design of cache/memory system

Based on the figure above, the black lines are mostly data lines, and the orange lines are control inputs for specific block. There are six main blocks where a majority of the functionality takes place.

The ICache block is 256-Byte word-addressable cache (implemented as a 64x32 array, 2048 bits) The cache is directly mapped from memory so the following formula was used to calculate the index of the cache.

(1)

This block takes in the Instruction Address (Program Counter) from the test bench program, as well as a flag, IHC, to test the different functionality of either an Icache Hit or Icache Miss. Since we implemented a write through policy, on a cache hit we take the word value from cache and update it to memory. Upon a instruction miss in cache, the data will be fetched from memory. Since the Write Allocate strategy was implemented, a block of memory was put back in the cache to update it. The data is then correct and outputted. The mux on the top is to select either from the memory (ICache Hit) or from the cache after a block update (ICache Miss). The access time for the ICache operation is just 1 cycle (10 ns).

The Bus block is simply to model the bus delay. Since the specifications of the bandwidth of the bus stated, 32 words/cycle, the assumption was made that the bus will delay a full cycle when 32 words or less are put on the bus. Since the most that will put across the bus is a memory block (8 words) this assumption works.

The memory Block is 1024 Bytes of Byte-addressable storage (implemented as a 1024x8 array). The memory then has addresses from 0x0 to 0x3FF. When there is an ICache Hit, a word is brought into the memory to perform a word update, when there is an ICache Miss the Block of memory at the inputted address is outputted and write allocated back to cache to update the values in cache. After the correct instruction is retrieved from either the cache or memory, the Op-Code is inputted into the CPU Block. The memory has a port access time of 5 cycles/word for reads, and an additional memory read time of 3 cycles/word for a total of 8 cycles/word for reading. The memory has a port access time of 3 cycles/word for writes, and an additional memory write time of 4 cycles/word for a total of 7 cycles/word for writing.

The CPU is a simple block, based on the Op-code inputted from the instruction cache/memory process it will determine a couple output signals. The instructions that needed to be implemented are shown below:

|  |  |
| --- | --- |
| Common Name | Register Names |
| lw $s1, 200 ($t3) | lw R17,200(R11) |
| sw $s3, 100 ($t4) | sw R19,100(R12) |
| add $s3, $t3, $t2 | add R19,R11,R10 |
| beq $s5, $t6, 400 | beq R21,R14,0[400-PC] |
| bne $s5, $t6, 500 | bne R21,R14,0[500-PC] |
| lui $s6, 40 | lui R22,40 |

There are two types of instructions implemented ALU/Branch Instructions and Memory Access functions. The last four on the list above are the ALU/Branch instructions. These only require the use of register values. The Op-codes are determined to match specific instructions. For the beq and bne instructions in the CPU the ALU\_DONE flag will be set. For the other two, the register value will be updated with the proper value (add, lui). The Initial values of the register can be seen below.

|  |  |  |  |
| --- | --- | --- | --- |
| Register Number | Initial value | Register Number | Initial value |
| Reg 8 | 0x00000008 | Reg 17 | 0x00000017 |
| Reg 9 | 0x00000009 | Reg 18 | 0x00000018 |
| Reg 10 | 0x00000010 | Reg 19 | 0x00000019 |
| Reg 11 | 0x00000011 | Reg 20 | 0x00000020 |
| Reg 12 | 0x00000012 | Reg 21 | 0x00000021 |
| Reg 13 | 0x00000013 | Reg 22 | 0x00000022 |
| Reg 14 | 0x00000014 | Reg 23 | 0x00000023 |
| Reg 15 | 0x00000015 | Reg 24 | 0x00000024 |
| Reg 16 | 0x00000016 | Reg 25 | 0x00000025 |

This will give the following results for the Add and LUI.

|  |  |
| --- | --- |
| Register Names | Results |
| add R19,R11,R10 | R19 <= 0x00000021 |
| lui R22,40 | R22<= 0x00280022 |

These two instructions, like the branch instructions will output the ALU\_DONE flag to signal that this branch is complete. For Load Word the ALU calculates the Data Address, sets the R\_W flag to 1 to signal a read, D\_Type flag to signal for the data memory access, and finally the Register number where the data in the memory will be loaded into the register. For Store Word the ALU calculates the Data Address, sets the R\_W falg to 0 to signal a write, D\_Type flag to signal for the data memory access, and the data in Register 19 (which is initially 0x19).

The DCache datapath will only be run through when there is a data memory access. This is only for the instructions Load word and Store Word, and signaled from the CPU as the D\_Type signal. The DCache block is 128-Byte word-addressable cache (implemented as a 32x32 array). The cache is directly mapped from memory so the following formula was used to calculate the index of the cache.

(2)

Once the correct value of the block is reached on a read hit (DCache Hit and Load Word), it will update the word in memory and output that data at the memory address, finally updating the value in the register file. On a read miss (DCache Miss and Load word), it will find the correct value in memory, and write allocate a memory block back to the DCache, updating 8 words in the DCache. On a write hit (DCache Hit and Store Word), the DCache will write Data value output from the ALU to the DCache, then it will be updated in memory. On a write miss (DCache Miss and Store Word), the DCache will store the correct value in memory, and then write allocate a block of memory back to the DCache.

**V. Implementation**

To show the implementation of the cache/memory system, code snippets will be shown in respect to their programs. Most everything related specific values in the cache and memory were hardcoded; it made the process of debugging a lot simpler. In order to run through the entire memory sub systems the testbench was written to initialize the Instruction addresses and the Cache Hit types.

The ICache is 256-Bytes of word-addressable memory, this was achieved by using an array. It has inputs of an Instruction Address, the IHC flag, and a Block Input for write allocate. It outputs the data at the instruction address. An array was utilized to define the memory, and all the values were initialized to 0’s.

**type** i\_array\_type **is** **array** **(**0 **to** 63**)** **of** std\_logic\_vector**(**31 **downto** 0**);**

**signal** I\_Cache **:** i\_array\_type **:=** **((others** **=>** **(others=>**'0'**)));**

The memory is a direct map to the cache, equation 1 was used to index into the instruction cache array. In this component, if the IHC was asserted, modeling a hit, then the Op-codes would be defined at the address specified. And then the data would be outputted after 1 cycle time to model its access time. If the IHC was not asserted then it would check if a block was being inputted. Upon a miss, one goes to the memory and retrieves the block of memory and write allocates it back into the cache.

The DCache is 128-Bytes of word-addressable memory; this was achieved by using an array. It has inputs of the Data Address, the DHC flag, Data input, block input, a read/write signal. It also has an input to determine if the CPU found a ALU instruction or a memory instruction, this is done using an ALU\_Done flag. The Data Cache outputs LW and SW done flags as well as the data at the memory address. Like the ICache, an array was used to simulate the array. This can be seen below.

**type** array\_type **is** **array** **(**0 **to** 31**)** **of** std\_logic\_vector**(**31 **downto** 0**);**

**signal** D\_Cache **:** array\_type **:=** **((others** **=>** **(others=>**'0'**)));**

The memory is a direct map to the cache, equation 2 was used to index into the data cache array. In this component the code will only be ran if it is a memory instruction (Ie. ALU\_Done is set to 0). This ensures that no data is corrupted in the D-cache dataflow. The Read/Write signal is used to determine if the instruction was a load or store. When R/W is asserted a load instruction is being implemented. On a D-Cache hit, a fake value of x”77777777” is loaded into cache and then outputted to update memory. On a miss, it will go to memory and fetch a block of memory and write allocate it back into the cache. When the Read/Write signal is not asserted, meaning a store instruction is being implemented, on a D-Cache hit; the inputted data value will be stored to the cache and then outputted to update memory. On a D-Cache miss, it goes to memory to get a block to write allocate back to the D-Cache.

The memory module was modeled using an array. It is 1024 Bytes of word addressable. It has all the same inputs, the cache flags, R/W flag, and the address and the data that will update the word upon a hit. It then out puts the data at the memory and/or a block of memory. The array initialization can be seen below.

**type** array\_type **is** **array** **(**0 **to** 1023**)** **of** std\_logic\_vector**(**7 **downto** 0**);**

**signal** memory **:** array\_type **:=** **((others** **=>** **(others=>**'0'**)));**

When there is an instruction hit, the data will be brought in from the cache for the write through strategy. In the following definition C\_Type define which cache is being used, 0 for Instruction Cache, and 1 for Data Cache. This is done by the following code.

**if** **(**IHC **=** "1" and C\_type **/=** "1"**)** **then**

memory**(**mem\_blk**)** **<=** Data\_In**(**31 **downto** 24**)** **after** **(**cycle\_time **\*** write\_access**)** **+** **(**cycle\_time **\*** write\_add**);**

memory**(**mem\_blk**+**1**)<=** Data\_In**(**23 **downto** 16**)** **after** **(**cycle\_time **\*** write\_access**)** **+** **(**cycle\_time **\*** write\_add**);**

memory**(**mem\_blk**+**2**)** **<=** Data\_In**(**15 **downto** 8**)** **after** **(**cycle\_time **\*** write\_access**)** **+** **(**cycle\_time **\*** write\_add**);**

memory**(**mem\_blk**+**3**)** **<=** Data\_In**(**7 **downto** 0**)** **after** **(**cycle\_time **\*** write\_access**)** **+** **(**cycle\_time **\*** write\_add**);**

Since in memory has specified access times for read and write. In this case for the write through, it will take 3 Cycles/word for port access then an additional access time of 4 Cycles/word. This leads to 7 cycles/word. For this branch it will take 7 cycles to complete the writing to the memory. This strategy of delaying is used throughout all of the modules to model delay in the circuit. When there is a miss in either the data cache or the instruction cache this module will output a block of memory. It is specified that a block is 8 words, so when there is a read or write the delay times will be multiplied by 8 to account for the 8 separate words.

One of the simpler modules was the modeling the bus. The bus will just output whatever is inputted. Within the module, there is availability to input all the signals that have been described before, however the most important ones are a data word, and a block (8 words). These would just delay one cycle time and then outputted.

The CPU model is a highly simplified version of a typical CPU. In our case, since we are mainly modeling the cache/memory design, we simplified the CPU quite a bit. The CPU will just take in the Op-Code from the instruction cache/memory system. From the Op-code it determines a few key outputs, if it is an ALU instruction, it will set the ALU\_Done flag to 1 to signify the ALU operation is done. For the Memory functions, load and store, it will output the respective R/W signal, and the Data Address that was hardcoded (this was done since it was assumed that Register values were initialized and were constant at the each of the instruction). For the Load instruction the following code is used.

**if** **(**OPC **=** x"8D7100C8"**)** **then** --Load instruction

**report** "In load"**;**

ALU\_DONE**<=**"0"**;**

DAddr**<=**x"000000D9"**;** --200 + $11

R\_W**<=**"1"**;**

Data**<=**"1"**;**

Data\_reg**<=**x"00000000"**;**

Reg\_Num **<=** "10001"**;** --R17

One of the last modules that was created was the Register Data component. This simply initializes the register values, and inputs register number from CPU, and data input. For the Add and LUI this is where the Register Values will show being updated. And when the load instruction is inputted, it will load the data value into the specific register.

Once all of the modules were constructed and tested individually, they were combined into two major modules. In order to make the combination of the entire dataflow simple, the modules was split in two. The first one constructed was the ICache\_Dataflow that constructed the top half of the Figure showed in the Overall Data flow. This included the ICache, Bus, Memory, a Mux, and finally the CPU. The second one constructed was the DCache\_Dataflow that constructed the bottom half of the Figure shown in the Overall Data flow. This included the DCache, Bus, Memory, a Mux. These two modules were then constructed together along with the register file to obtain the final Testbench program, which we called Cache\_Memory\_Dataflow. This simple inputs the Instruction address, and the hit flags. It will then out the Done flags, and then finally the data out. When running the program, the register file can also be shown to show the register value changes.

**VI. Results**

**Load**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | Opcode |
| 0x200 | lw $s1, 200 ($t3) | 0x8D7100C8 |

The instruction load word is meant to load the word stored at 200($11) and store it in Reg 17.

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

**Store**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | Opcode |
| 0x288 | lw $s1, 200 ($t3) | 0xAD930064 |

The instruction Store word is meant to store the word at Reg S1 in the memory location designated by 100($12) .

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

**Add**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | Opcode |
| 0x010 | lw $s1, 200 ($t3) | 0x016A9020 |

Adds two registers(Reg[19] and Reg[11]) and stores the value in 19. In our module this is an ALU operation and hence should assert the ALU\_DONE signal.

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

**Branch if equal**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | Opcode |
| 0x018 | **beq $s5, $t6, 400** | 0x12AE0178 |

The instruction branch if equal will branch to the branch address if the value in Reg 21 and Reg 14 are equal. In our simulation this instruction does not entail accessing the data cache

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

From the waveform above, It can be noted that LW\_DONE signal is asserted when the instruction is done running at 170ns. Since the iCache was a miss it can be noted that no block was brought back from the memory to chace and the output went straight to the mux and inputted to the CPU

**Branch if not equal**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | 0xOpcode |
| 0x020 | **bne $s5, $t6, 500** | 0x16AE01D4 |

The instruction branch if not equal will branch to the branch address if the value in s5 21 and t6 14 are not equal.

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

**Load upper immediate**

|  |  |  |
| --- | --- | --- |
| Instruction Address | Instruction stored | Opcode |
| 0x028 | **lui $s6, 40** | 0x3C160028 |

The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes. The immediate value 40 is shifted 16 bits and stored in register s6

**Test when iCache miss and dCache miss**

**Test when iCache miss and dCache hit**

**Test when iCache hit and dCache miss**

**Test when iCache hit and dCache hit**

**VI. Discussion**

* A discussion on how you tried to optimize your design
* A discussion on any improvements or additional features made to your design
* A discussion on what does not work correctly in your design

**VII. Conclusions**

In conclusion, the developed code was successful in meeting the specifications. With hardcoding everything, the group knew what values would appear, and what each instruction would do. Knowing it all made everything a little bit easier to code. The report shows how each component was set up and how it all came together at the end in the final test bench program. The lab served as a good practical simulation for the expected workflow and ultimately provided the team with a more hands-on insight in respect to the operation of caches.

**Attachments**

Each individual program’s code will be attached, the individual test benches for the components won’t be necessary, since the overall test bench works without issue.