Lab 5: Cache Controller Implementation in Verilog HDL

Due **5/1** (midnight)

Instructor: Bingzhe Li
TA: Ricardo Hernandez

1 Introduction

In Lab 3/4, you implemented versions of the ARMv4 machine. In this lab, you will try to recognize the importance of getting data to the processor by implementing a cache controller along with your pipelined processor. Caches although difficult tasks to comprehend really just do "desk duty" on access to main memory. They are meant to be smart about what to load/store and what to optimize to alleviate issues with the memory wall [1].

This lab will be one of the last labs for the semester. Therefore, it is important you put your best foot forward in completing this lab as it will count significantly towards your lab grade. I encourage you spend more time than usual on this lab (as you also have longer than usual to work on it) and maybe try one or more of the bonus opportunities listed near the end of this document. They are a great way to make up points you may have lost on previous labs.

For this lab we will again be using Verilog HDL to model the hardware. We will not require the implementation of using the DSDB and FPGA for testing this time. Those implementation will be the extra credits.

This laboratory, although simpler than the last two, is more about understanding how and why caches perform during a memory access. For full credit, you should achieve the following in this lab:

- Implement a direct-mapped cache controller with write-through write allocate capabilities in Verilog, which should be implemented in the **control.sv** file.
- Ensure that your implementation is correct by simulating it and verifying that the cache is either hitting correctly within the cache or misses and loads the correct value back into the cache along with passing the correct value to the processor.
- The key to making your cache work is making sure you have a Finite State Machine allocated within your cache controller and making sure all the states work. This can be difficult as states make debugging and verification harder (see hints later).
- ~~Implement your design on the National Instruments Elvis III and DSDB board. Compare your implementation in terms of execution with the pipelined microarchitecture from the previous lab.~~

2 Lab Resources

As usual, the source code for this lab is provided on Canvas, which you should have access to if you are reading this document. You will be making a Finite State Machine in this lab. If you need a refresher on those, look back at lab 1 where we created a basic one from a diagram.

3 Cache

This design is a simple cache and is really meant to help you understand caching better. The cache is a write-through cache; this means when writing to a memory location, the memory **and** cache are both written to make sure the data is consistent. The organization of the cache is shown in Figure 1.

The more difficult part in understanding this cache is what happens on a write miss. Write misses are confusing because the data is not in the cache, so why should I really care about it. However, how you handle a write miss is important to keep the data consistent. A write miss can be handled in several ways. Most caches utilize two options:

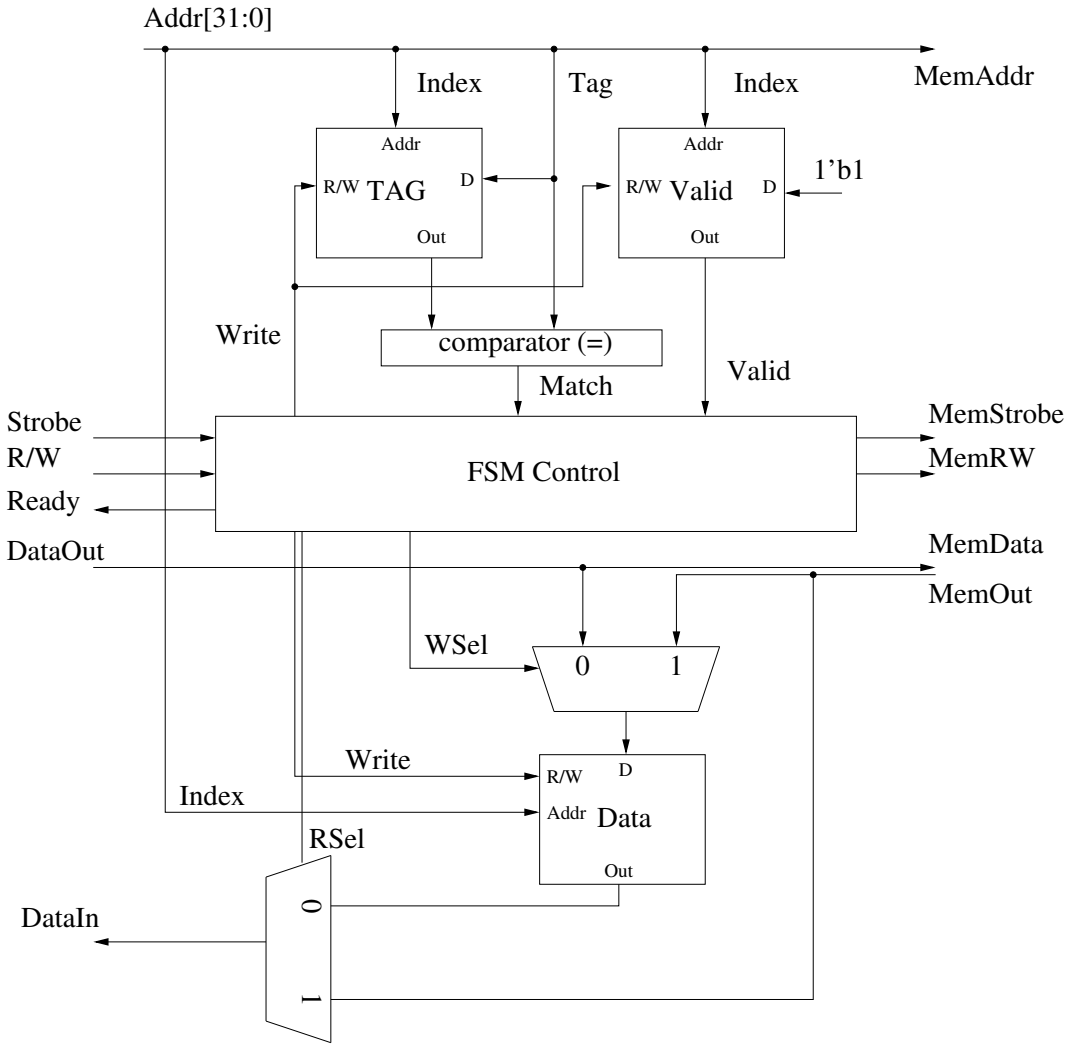

Figure 1: Direct-Mapped Cache Architecture (clocks not shown).

1. Write allocate
2. No-Write allocate

The basic difference between the two is: for write allocate the data is written to both cache and memory, whereas a no-write allocation only writes to memory while leaving the cache alone. Most write-through caches utilize no-write allocate since subsequent writes really have no advantage, so it is easier to just update memory and not the cache, as well. Although not very efficient, this lab utilizes a write allocate, but a super easy fix to make this a no-write allocate, the more common option, is to modify the FSM in Section 4. A block diagram of a write-through cache with no-write allocation is shown in Figure 2. To give a reasonable size to your cache, please use a 4KB direct-mapped cache. This will be 18 bits for the tag, 12 bits for the index or set and 2 bits for the byte offset. This is shown visually in Figure 3.

4 Finite State Machine (FSM)

The key element on the cache controller is the Finite State Machine (FSM). The FSM makes sure the cache works and, if needed, alerts the main memory for any synchronization or replacement. To make things easier, the FSM should be a Moore type FSM (i.e., the output is based on present state only).

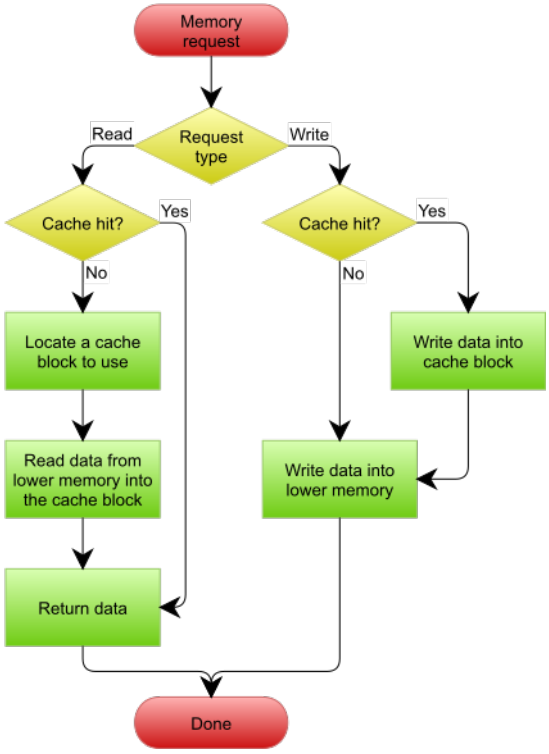

Figure 2: Write-Through No-Write Allocate Cache [Wikimedia Commons [2]].

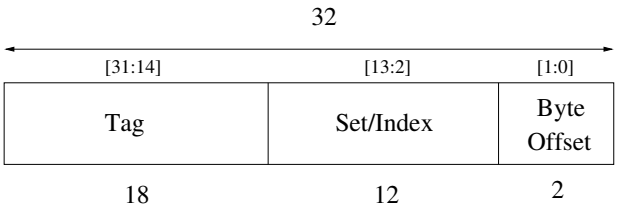

Figure 3: Bit Breakdown for Cache Design.

The FSM is quite simple; it can be accessed for a Read or Write. Any access, either read or write, starts with a Strobe. The Strobe is an asynchronous input and necessary to initiate the access since it is not known how long the memory will take in most instances, similar to our other labs. For testing this lab, we will use a counter to wait for main memory to complete. These are typically called wait states and most memories have them. The counter will start counting once it enters one of the states to access main memory. It will then count down until it hits 0 and then move to the final state. For this laboratory, please use 100 or `0x64` cycles as the number of wait states. To make things easier, the counter is hard-coded to this value, however, the FSM in Section 4 can be modified to output this value.

The FSM has several inputs and these are usually tied to the cache hardware as shown in Figure 1. The following are the inputs utilized for the FSM:

- Reset - reset the FSM
- Strobe - initiate access
- R/W - indicates a Read (1) or Write (0)
- Match (M) - from comparator indicating the tag is in the cache based on the index.
- Valid (V) - the data in the cache is a valid item (i.e., its tags match the address by the instruction)

- CtrSig - signal from the counter that it has counted down to 0

The following are the output utilized for the FSM:

- LdCtr - this is a signal to tell the counter to reset its value to its wait state. Most memories have different wait states for read and write, but it is assumed in this laboratory that they are equal.
- RdyEn - this signal is utilized to indicate the cache is ready on a Read Hit
- Rdy - this is the `Ready` signal indicating the cache access is complete. You should use this signal within your microarchitecture to stop stalling the pipeline.
- Write (W) - this is an indication of a miss and therefore, a write is necessary to update the cache.
- MStrobe - this is a strobe for main memory indicating a memory access is needed.
- MRW - this is the R/W signal for main memory utilizing similar encoding as the processors signaling 1: Read 0: Write
- WSel - Selects data from main memory on a `ReadMiss`
- RSel - Selects data from main memory on a WriteMiss/WriteHit

FSMs can be difficult because it is hard to see each transition. Consequently, the state transition table is shown in Table 1. In Table 1, the `WriteMiss` and `WriteHit` states are identical and can be consolodated for optimization. In addition, these states are sometimes useful to keep counters of hits and misses that most current architectures record although this is not used by most users. As noted in Table 1 the `Reset` signal is not shown and should reset the FSM to the `Idle` state. You can use any encoding you wish for its implementation.

Present State	Strobe	R/W	M	V	CtrSig	Next State	LdCtr	RdyEn	Rdy	W	MStrobe	MRW	WSel	RSel
Idle	0	x	x	x	x	Idle	1	0	0	0	0	0	0	0
Idle	1	1	x	x	x	Write	1	0	0	0	0	0	0	0
Idle	1	0	x	x	x	Read	1	0	0	0	0	0	0	0
Read	x	x	0	0	x	ReadMiss	1	1	0	0	0	0	0	0
Read	x	x	0	1	x	ReadMiss	1	1	0	0	0	0	0	0
Read	x	x	1	0	x	ReadMiss	1	1	0	0	0	0	0	0
Read	x	x	1	1	x	Idle	1	1	0	0	0	0	0	0
ReadMiss	x	x	x	x	x	ReadMem	1	0	0	0	1	0	0	0
ReadMem	x	x	x	x	1	ReadData	0	0	0	0	0	0	0	0
ReadMem	x	x	x	x	0	ReadMem	0	0	0	0	0	0	0	0
ReadData	x	x	x	x	x	Idle	0	0	1	1	0	1	1	0
Write	x	x	0	0	x	WriteMiss	1	0	0	0	0	0	0	0
Write	x	x	0	1	x	WriteMiss	1	0	0	0	0	0	0	0
Write	x	x	1	0	x	WriteMiss	1	0	0	0	0	0	0	0
Write	x	x	1	1	x	WriteHit	1	0	0	0	0	0	0	0
WriteMiss	x	x	x	x	x	WriteMem	1	0	0	0	1	1	0	0
WriteHit	x	x	x	x	x	WriteMem	1	0	0	0	1	1	0	0
WriteMem	x	x	x	x	1	WriteData	0	0	0	0	0	1	0	0
WriteMem	x	x	x	x	0	WriteMem	0	0	0	0	0	1	0	0
WriteData	x	x	x	x	x	Idle	0	0	1	1	0	1	0	1

Table 1: State Transition Table for FSM within Cache Controller (Note: to improve visibility Reset is not shown; it is implied that a Reset will cause the FSM to go to Idle)

5 Microarchitecture Changes

Very few changes are needed in the ARMv4 core itself. You can, however, make changes to for additional credit if you would like.

The Hazard Detection Unit (HDU) is already integrated to handle stalls and many of the registers have been converted over to accept these stalls. You need to make sure the `Ready` signal is correctly used in the Microarchitecture and that all the proper registers have been connected to the HDU otherwise, even with a functioning cache, the design will not pass the testbench.

The Strobe signal indicates that you want to access the data memory, as with previous labs. Again, remember the controller is early within the pipeline stage and much of the control is passed down the pipeline stages to make it simpler. Therefore, you will also have to make sure this Strobe signal reaches all necessary stages of the pipeline.

One of the more difficult things to understand is how to handle the stalls. Any `LDR/STR` mnemonic (instruction) should theoretically stall the pipeline as it waits for the cache to either give the data or get the data from memory. The logic should handle `Strobe` and `Ready` signals:

```
assign StallF = ldrStallD | PCWrPendingF | Strobe&!Ready;
```

However, you will see that there may be an issue with how the stall is handled as it the pipeline needs to handle the stall as soon as it sees it (i.e., through the controller). If you remember that a memory instruction does not always utilize an address directly; it sometimes calculates an effective address (e.g., `mem[12+84] = 7`). Therefore, if you stall the address as soon as you see the memory instruction, it may not work as it computes this effective address (i.e., `12+84=96`) during the `Execute` or `EX` stage. It is easier to initially try to use the stall during the `MEM` stage - there are ways to optimize this, but you should think about the design ideas mentioned in class.

6 Getting Started & Tips

1. Much of the code has been given to you, so its up to you to understand how Caches work and integrate it into the pipelined architecture.

2. For this laboratory, you only have to implement the cache on the current pipelined architecture. That is, you do not have to add any additional instructions since `LDR/STR` are already implemented. Yes, this architecure is missing instructions similar to the start of lab3 and lab4, but we won't be using those extra instructions for testing in this lab.

3. It is highly advisable to create the FSM first and add this first and use the sample FSM in the `fsm` directory. Test this FSM separately before trying to simulate the cache.

4. Review the material on caches from the lecture notes and the textbook [1].

5. It would be highly advisable to dump out the cache contents using the dump command that you might have used in labs 3/4. This way you can tell if the cache is working. You should add this to the bottom of your DO file similar to the following command:

   ```
   mem save -outfile memory.dat -wordsperline 1 /testbench/dut/ram/RAM
   mem save -outfile register.dat -wordsperline 1 /testbench/dut/arm/dp/rf/rf
   mem save -outfile valid.dat -wordsperline 1 /testbench/dut/cache/ValidRam/RAM
   mem save -outfile tag.dat -wordsperline 1 /testbench/dut/cache/TagRam/RAM
   mem save -outfile data.dat -wordsperline 1 /testbench/dut/cache/DataRam/RAM
   ```

6. FSMs are tricky to debug as its not always known which state you are in. Therefore, it is highly advisable to output the `NEXT_STATE` and `CURRENT_STATE` signals that house the states on the wave window. A sample FSM is given to you with a DO file to see this on a wave window. This is located in the `fsm` diretory of your repository - please use this FSM as a template for your cache controller.

7. The FSM can easily be modified to record the Hit and Miss which might be useful if you want to add a set-associate cache. However, right now it does not record this information.

6.1 Tips

- You are pretty much not going to finish this assignment if you start late - guaranteed. So, start early and apportion your time appropriately.
- Read this handout in detail. Ask questions to the TAs or me using Canvas.
- When you encounter a technical problem, please sift through your logs file and look any error specifically. Many times the error tells you the problem.
- Be cautious about how Verilog handles signed/unsigned numbers.
- Memory accesses must be aligned. If you are accessing a byte, any address is allowed. But, if you are accessing a 4-byte word, the two lowest bits of the address must be zero.

7 Handin

You should commit all your code to your GitHub repository and upload **a copy of your code** (compressed Zip file including all code) and **your lab report (PDF file)** to Canvas. Your code should be readable, well-documented, **compilable** and **executable**. In addition, please turn in additional test cases that you used in the `inputs/` directory. If you feel the need to describe any additional aspects of your design in detail, please include these in a separate README.

8 Extra Credit

Again, there are many possibilities for extra credit, but you should only attempt this if you get the baseline project done and you have extra time. I mentioned some of these in class and they are documented here for your edification.

- Modify FSM to handle no-write allocation
- 2/4-way set associate cache.
- Write-back no-write policy (will require a dirty bit (D) - more RAM)
- Add a write buffer (FIFO) - see FIFO directory in the repository for a working version of a FIFO as discussed in class.
- Add block size for spatial locality.
- Multi-level cache (change wait-state for 2nd level to something other than 1 (e.g., 10 cycles to mimic a real L2 cache)
- Integration of results with dineroIV (`http://pages.cs.wisc.edu/~markhill/DineroIV/`)
- Software-timing model to mimic cache and/or memory from Lab 1 code
- Perform an analysis with and without the cache to show that it is benefiting your archtecture (make sure your include your AMAT).
- Cool video to describe what your group is doing

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2016.

[2] Wikipedia contributors, "Cache (computing)," 2011. [Online; accessed 01-Apr-2018].