Jaci Reichenberger, A20131719

Stephen Potter, A11341625

ECEN 4243: Computer Architecture

1 May 2021

Lab 5: Cache Controller Implementation in Verilog HDL

Section 1: Introduction

This lab focused on the importance of getting data to the processor through implementation of a cache controller. The cache is important to machines as it is in charge of what data to load/store and it assists with the memory wall. This was done by creating an FSM machine to control the states of the cache. The FSM was created in the control.sv file, then compiled and debugged until the results stated that the simulation was successful. Thus, the desired results were reached after the FSM table was understood, and that table was converted into SystemVerilog code.

Section 2: Baseline Design

The baseline design for this lab was to create an FSM machine for a cache controller. The state transition table (Figure 1) was given, and an FSM diagram (Figure 2) was created based on the table. There are 10 states within this FSM, which are dictated by five inputs: Strobe, R/W, M, V, and CtrSig. The Strobe initiates access to either read or write when set to high. R/W indicates whether the instruction is Read(high) or Write(low). M, which stands for Match, is from the comparator, which indicates that the tag is in the cache based on the index. V, which stands for Valid, indicates that the data in the cache is a valid item. CtrSig is a signal from the counter that it has counted down to zero. The 10 states that the FSM can be in are as follows: Idle, Read, ReadMiss, ReadMem, ReadData, Write, WriteMiss, WriteHit, WriteMem, and WriteData. The output of the various states are these variables: LdCtr, RdyEn, Rdy, Write (W), MStrobe, MRW, WSel, and RSel. LdCtr is a signal that tells the counter to reset its value to the wait state. RdyEn is a signal that is utilized to indicate that the cache is ready on a Read Hit. Rdy is the Ready signal that says the cache access is complete and will tell the microarchitecture to stop the pipeline stalling. Write (W) is an indication of a miss, so a write will be needed to update the cache. MStrobe is a strobe for main memory indicating a memory access is needed. MRW is the R/W signal for the main memory, which is a Read for high and a Write for low. WSel selects data from main memory on a ReadMiss, and RSel selects data from main memory on a WriteMiss or a WriteHit. The output for each state can be seen in Figure 1.

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

Figure 1: State Transition Table of Cache Controller FSM

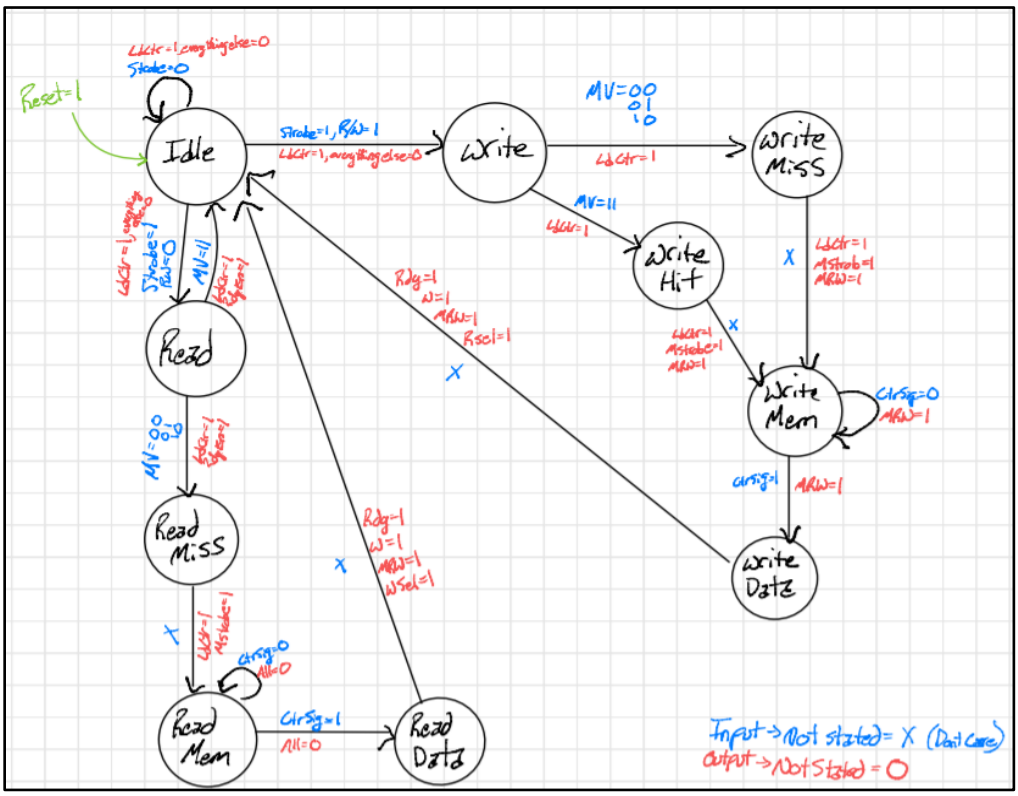

Figure 2: FSM Diagram

Section 3: Design

The design for this lab was based on a direct mapped write-through cache. It is meant to be the simplest implementation of a cache. However, what benefits it offers in simplicity, it lacks in speed and efficiency. When using the write-through method, the processor has to write into the cache and the memory in order to stay consistent. This takes extra time - up to 100 clock cycles - which hinders the performance of the caching scheme, and ultimately the CPU. This lab was meant to show the basics of using a cache by using one of the simplest methods to implement. The design begins with initiating a CacheControl module, which includes input and output wires to represent all the connections to the other crucial parts of the cache system. Next, the parameters are set to assign the binary values that correspond to the states of the machine and allow for the case selection. Within the case selection, there is a check to see if the reset bit is active; if not, the FSM moves on to check the state of the machine. The program then checks to see if any of the conditions are met or need to be met. These include Idle, Read, ReadMiss, ReadMem, ReadData, Write, WriteMiss, WriteHit, WriteMem, and WriteData. Each of these different cases are active depending on the way that the cache and memory reads.

The first state is the Idle state. This is the start point for all reads and writes. Therefore, if the input has Strobe set high, and R/W set high, then the FSM will go to the next state, Write. If Strobe is set to high, and R/W set to low, the next state will be a Read. If the Strobe is set to low, then the FSM will stay in the Idle state. Within the Read state, if M and V are both set to high, then the FSM will go back to Idle, otherwise the next state will be ReadMiss. Within the ReadMiss state, it does not matter what the input is, as it will automatically go to the ReadMem state. Within the ReadMem state, if CtrSig is high, the next state will be ReadData; however, if CtrSig is low, it will stay within the ReadMem state. The ReadData state will change the output, then reset the FSM back to Idle. The Write state acts similar to the Read state, but when M and V are high, it will go to WriteHit, otherwise the next state will

be WriteMiss. Both WriteMiss and WriteHit will go directly to WriteMem after changing the output. WriteMem acts just the same as ReadMem, but instead of going to ReadData on CtrSig set to high, it will go to WriteData. And finally, the state is again set back to Idle to run through the loop again. One more thing kept in mind is the Reset. When the Reset is high, the FSM will go back to Idle, regardless of which state it was in at the time. This concludes the two main loops associated within the control.sv file that controls the cache operation.

Section 4: Testing Strategy

Testing was done similarly to previous labs and was achieved by using an input file to run some SystemVerilog along with other source files included in the lab source code. Much of this code was prewritten, and it was only required to fill in the control.sv file and compile and run the simulation. The first step that was done in the testing process was to test the FSM sample file given to the students. This file was a working bare bones skeleton that could be used to base the design on for the control.sv file. Running this code showed how the program should operate if the proper logic was followed. The sample FSM code revealed that the simulation should stall after reaching the proper values read from its memory, which are CURRENT_STATE and NEXT_STATE, and which should have the values that the simulation was able to reach within the simulation time. The FSM sample code is assigned values of zero to seven for the different active states, and the Idle state is assigned the value eight. When cycling through the states checking to see if the current state is less than or equal to the next state, the simulation advances through the numbers sequentially. The waveforms of this simulation are seen in Figure 3.

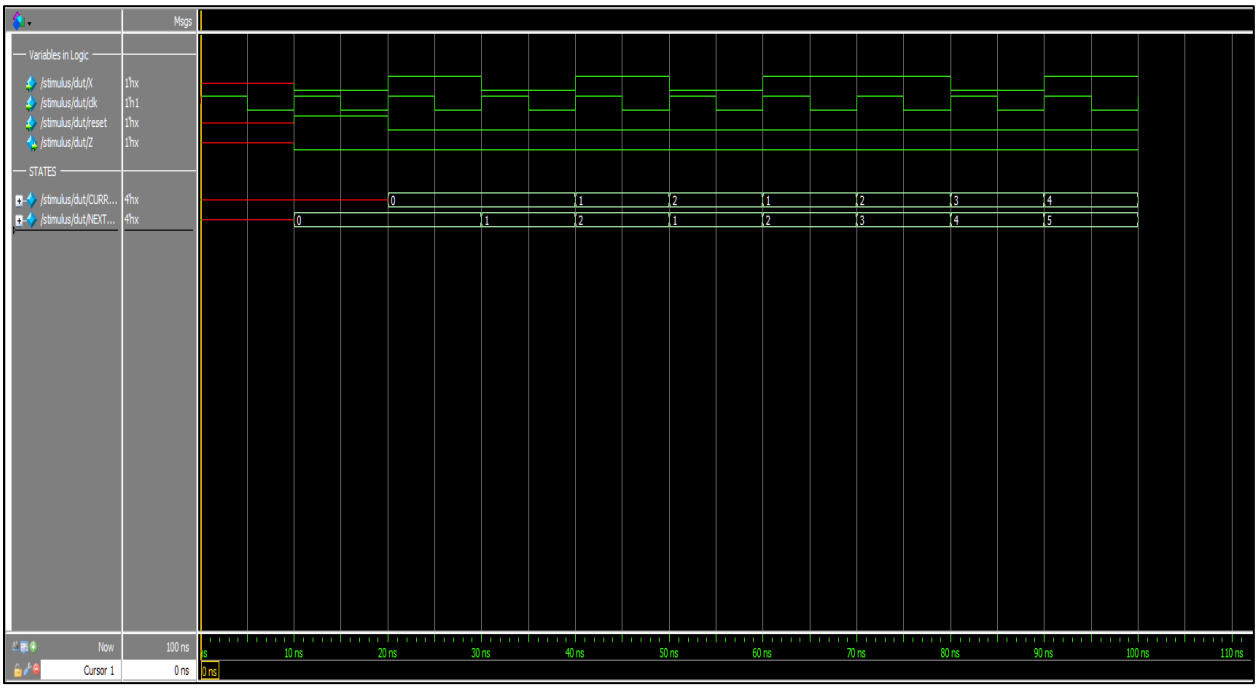

Figure 3: Waveforms of the Sample FSM

Testing the FSM source code gave insight for how the control.sv should be written and implemented in the arm_pipelined simulator. The next step in the process was to implement this into the arm_pipelined simulator source code with some minor adjustments. The FSM that was to be designed in the control.sv file needed to get its source data and behavior from separate files that were part of the higher-level source code. These other files were the testbench (tb) file and the actual memory file for the simulation. This allowed the simulation to pull data from real

memory addresses and the tb file told the simulation how to test with this data. The arm simulation was then run with the proper source files adjusted and compiled. When the simulation waveforms were observed, the results were shown to be correct. The simulator should, as mentioned above, stall at the proper values if those values are reached. Those values should be in the signals DataAdr and WriteData with the values 100 and 7 respectively. This is shown below, highlighted by the red box, in the waveform in Figure 4. Also shown in Figure 5, it can be seen that the simulation stalled at line 30 of the tb.sv file, which is right after the "Simulation succeeded" print statement. Also shown in Figure 5 is the terminal transcript window, that also shows a "Simulation succeeded" message highlighted in blue. With this, the simulation testing was concluded and the simulation and testing complete.

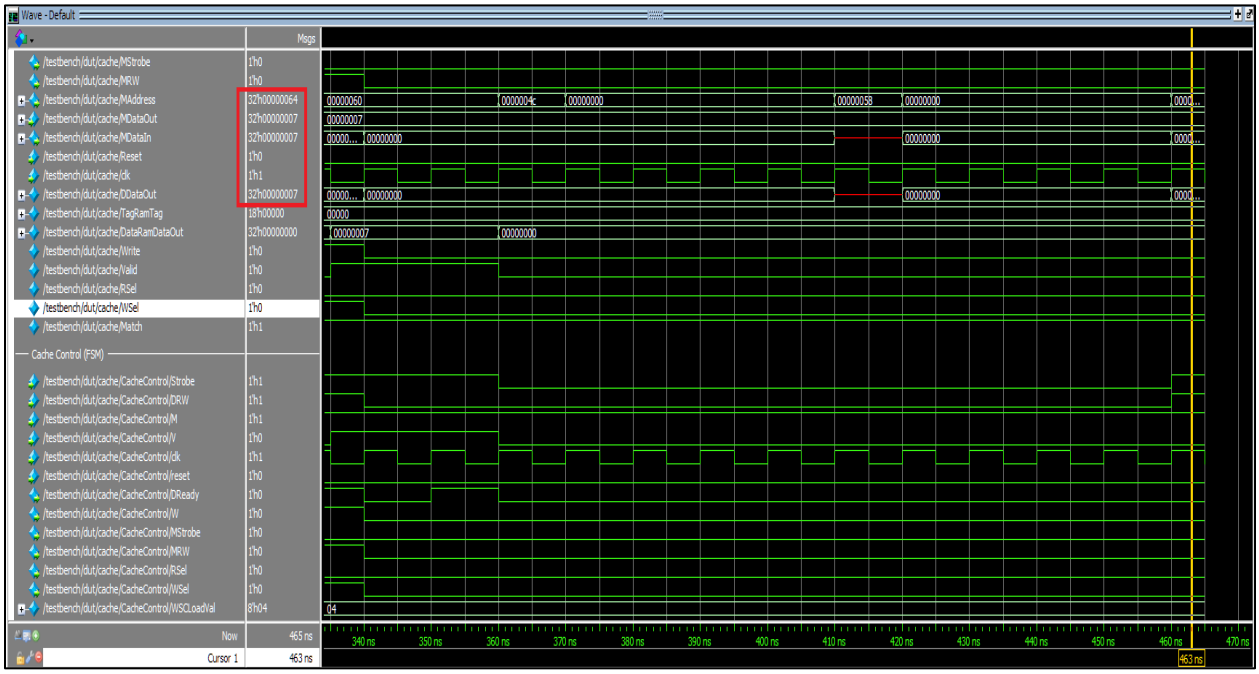

Figure 4: Final Waveform Results of the arm_pipelined Simulation

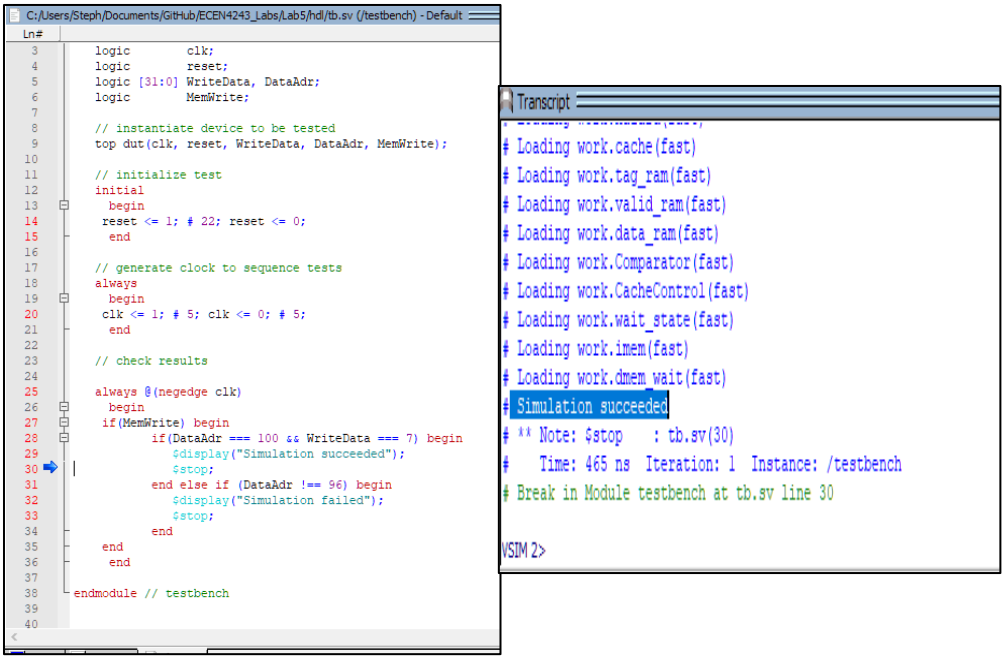

Figure 5: Testbench Code of Successful Simulation (Left), and Terminal Showing a Successful Simulation (Right)

Section 5: Evaluation

Through multiple iterations of the FSM code, and a full understanding of how the FSM of the cache controller works, this lab was able to be completed successfully. Through use of the given testbench, the simulation showed that it was successfully completed, thus proving that the built FSM was designed correctly. Therefore, the results obtained from this lab helped to show how a simple cache controller can work with a pipelined structure. Figuring out how exactly the FSM worked was critical in this lab. Not only does the FSM dictate what is in the cache at a given time, it also tells the memory to write to the cache if needed. Though it has shown not to be the most efficient cache, the cache that was created got across the basics of how one would be implemented in computer architecture.

All in all, this lab was beneficial in that it showed how to create and implement a basic cache controller. This lab was successful with a creation of the FSM in control.sv. With the aid of the State Transition Table, and a drawn diagram for the FSM, the logic for the FSM was fully understood, thus making it fairly simple to implement for the SystemVerilog code using case statements. In sum, this lab successfully taught how to create a simple cache for an ARMv4 Pipelined Architecture.