16-bit (not) RISC CPU

# Ammon Dodson

# Jake McKenzie

***Introduction:***

**Any computation can be split into sub-computations. Those sub-computations may be split even further. We call any computation that cannot be split any further: an instruction** (Mutlu 2015)**. Processing of instructions consists of components that transform data signals and components that determine what those transformations should be. This is done by establishing a *data path* and *control unit*** (Thimmannagari 2005, 5-6)***.***

**In this laboratory assignment we will create a processor which will allow any given *programmer* the ability to execute a set of instructions. This will be done by implementing the instruction set given to us in the design specifications and implementing components correctly to create a processor that will produce the desired output of the programmer. The processor made for this assignment, while being reduced, does not strictly follow all the RISC conventions as it does not have any pipelining of any kind.**

Experimental Design:

For this lab RTL code was first developed to match the design specifications. Processing the instruction is handled by a state machine where the current state is the programmer-visible state and next-state is the instruction execution specification (Wang 2017, 105-111). Many issues arose in the design and test of this RTL circuit, especially in the ALU and control unit which will be talked about further on in the laboratory assignment. First the data path then the control unit were designed as these were agreed upon to be the most important modules for our project. When designing a computer how do we turn an english mnemonic such as LOAD 0x0B or SUB 0x0B 0x1B? We do this by defining which part of an address goes to an OP-Code and which part goes to an address. Anything within the allowable states that we have can be chosen, they need not be sequential like we defined them in this assignment.

An ALU had already been designed for a passion solo project so we adapted it to fit the needs of this assignment. If we look at the simplest von Neumann model in figure 1, we see that the control unit, ALU, and datapath (included in the ALU and path between memory and CPU here) play the central role central role in the CPU. We did not make a strictly von Neumann model CPU because our data memory is separate from the instruction memory, which is some criteria von Neumann defined originally. This was because all computers at this time had no modular structure to them and were purpose built.

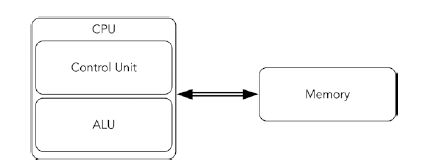


Figure 1: Simple von Neumann architecture (not exactly what we made here) (Zhirkov 2017, 4)

The ALU was previously developed with an add, subtract, and, or, xor, shift left, shift right, roll left, roll right but only the operations asked of us were used. Unused operations were not included. The adder module was built separately to ensure correct behavior for signed and unsigned overflow and underflow checking. Heavy use of explicit casting was needed in the testbench to ensure that our values were being treated exactly the way we wanted them to. Signed overflow was one of the more difficult parts of our testing for this assignment.

After we completed this testbench we ensured that the ALU was working correctly. Conditional flags for correctness are often used in many processors, ARM based processors have a negative, zero, carry, overflow and saturation flag built into each instruction (Upton, et al. 2016, 134). While we made an overflow flag for this processor, the UART receiver we made previously had a saturating counter, which is often used in digital signal processing.

The control unit was developed in the form of a lookup table. The machine word opcode is used as an index into that table, and the different control signals are sent to their respective destinations. Restricting operands, as we did here, to registers simplifies the control unit (Dandamudi 2005, 43). Many iterations of the control unit happened. At first, we were incrementing PC counter incorrectly, by doing so both in the fetch and decode state. After fixing this we ran into an assortment of timing issues. The control unit’s job is to ensure that the rest of the processor is performing the correct instructions.

The data path needs control signals which are driven by the control unit. To get the timing correct and to remove latching warnings, we had to make multiple always, always\_ff and always\_comb blocks to split up latching assignments from combinational assignments to reduce warnings.

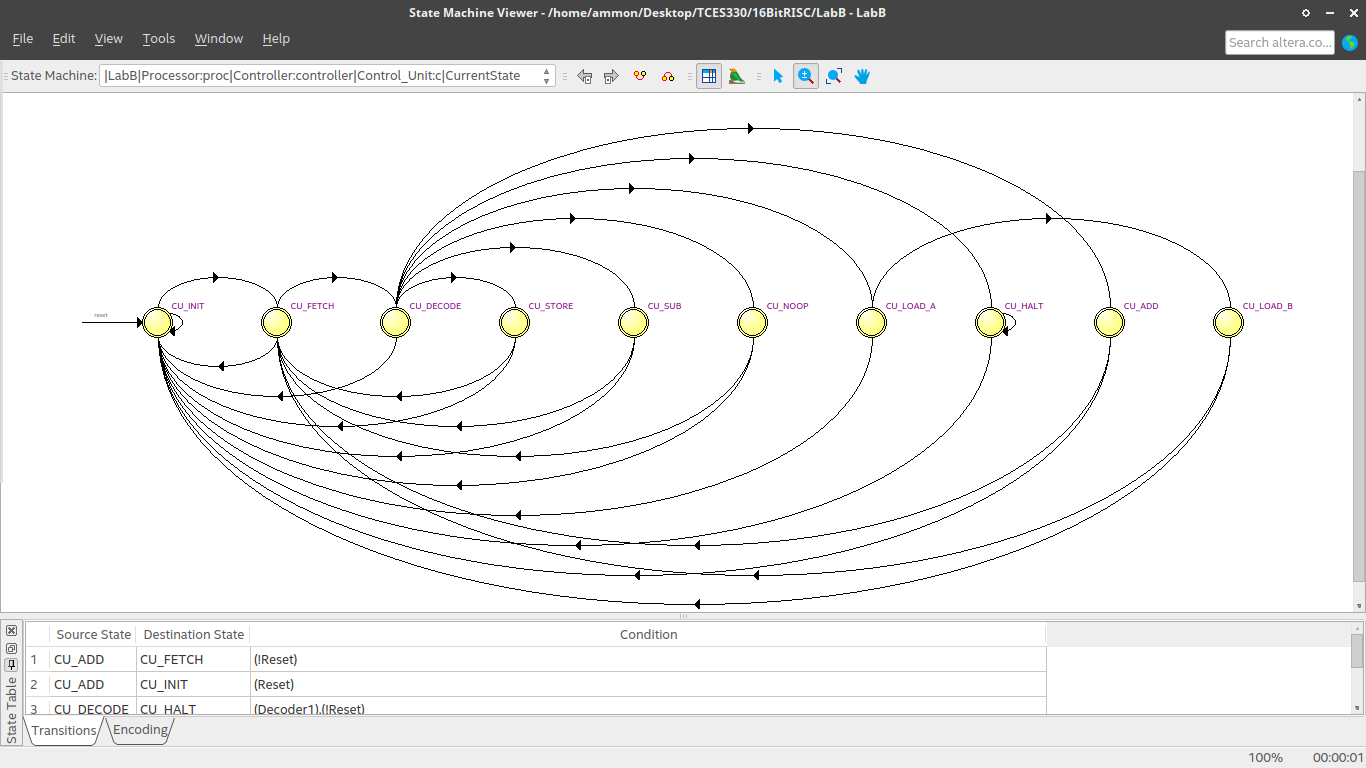


Figure 2: The finite state machine for the Control Unit

There’s not much of a difference between instructions and memory, they’re all just bits in the end (Landley and Dionne 2016). It’s the orientation those bits take that matters and the datapath drives the conversation the memory has with the instructions. Many difficulties arose when designing the data path. We failed to instantiate the LPM ROM memory file for the datapath for quite a while, a week in fact, and we still aren’t sure how we got it to work. The datapath instantiates a 16 wide 2-1 multiplexer and instantiates the ram module, register file module and ALU module. The datapath is setup for the generation of necessary control signals.

These control signals when generated and given to the respective components at a given instant, that is a state duration will achieve place transfer of the program counter contents onto a bus. In each clock cycle the control signals must be correctly set to set the data path.

Bringing the design all together was quite an undertaking. If you come from a traditional software background world, writing in SystemVerilog can seem like deciphering hieroglyphics from different eras and we felt constantly handicapped by the tools in quartus and modelsim, especially the LPM in quartus. If we look at Figure 3 on the next page we see that we obtain the same overall circuit as the design specifications. Errors are eventualities, either from something we wrote or from an unexpected condition in the environment. There are two ways to handle errors: A system can enter a state of graceful degradation where the software does the best it can or it can fail loudly and immediately (White 2011, 70).

We decided to go with the former as all errors we received were errors that were non-critical according the experts on quartus forums, even though the compiler in quartus told the errors were critical.

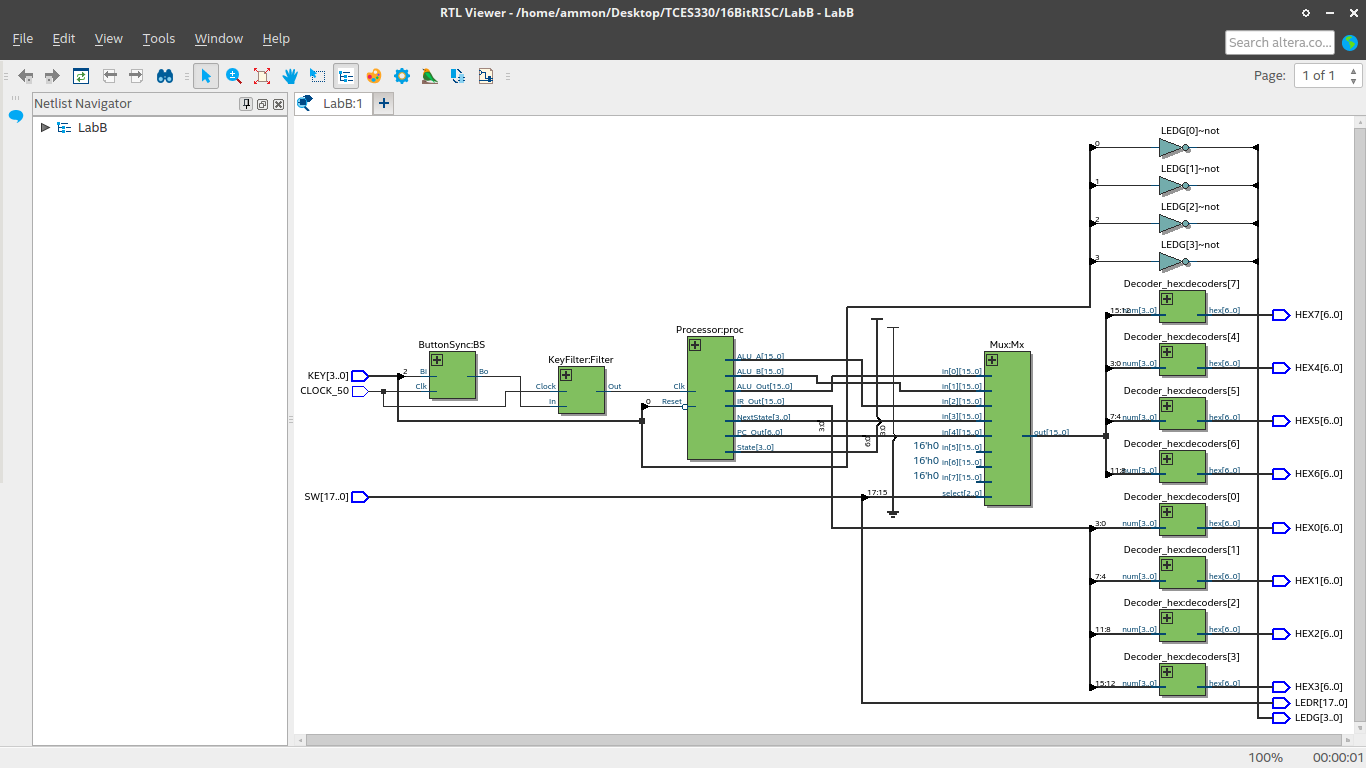


Figure 3: Top level RTL generation for the processor

Simulations:

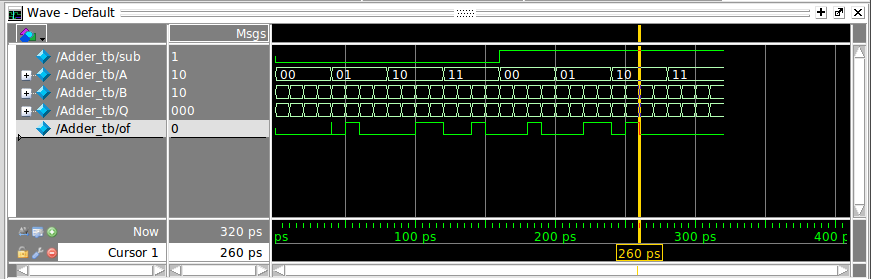
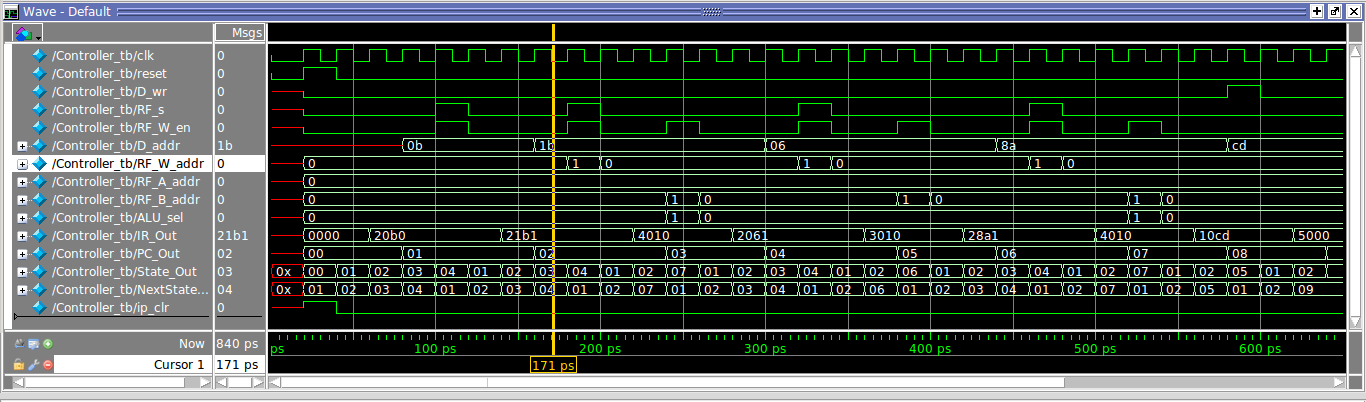
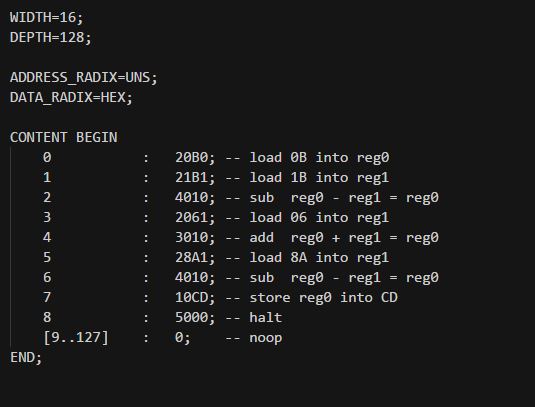


Figure 4: Adder timing diagram we created to check for signed and unsigned bit overflow and underflow (Allen n.d.).

Timing diagrams show the relationship between transitions, where some show transitions on different signals or the same signal (White 2011, 46). For testing for signed and unsigned bit overflow and underflow we reduced the bit length to two. The ALU does not know about signed or unsigned numbers, it simply does the binary arithmetic we ask it to. This timing diagram show results that we expect of the adder.



Figure 5: Timing diagram for the Controller

Above is the timing diagram of the controller module, showing each instruction being loaded an executed as instructed. By looking at IR\_Out we see that the controller performs as we expected, as can be seen from the mif file. At first when testing the controller, we ran into issues of the program counter being delayed one extra clock cycle. This was fixed by changing our finite state machine operated in the control unit module.

Figure : Instructions from mif file

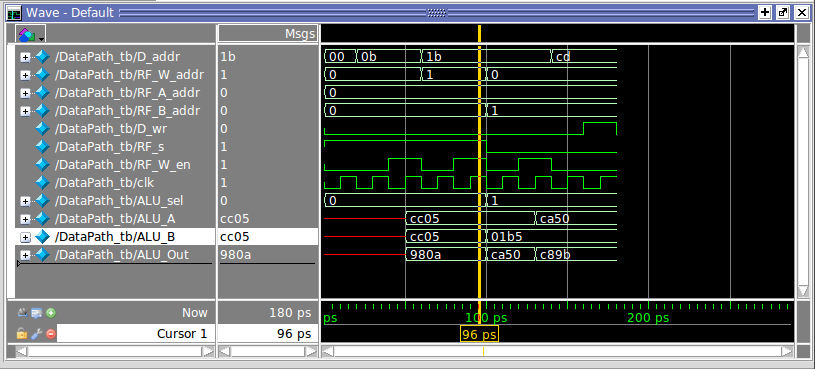
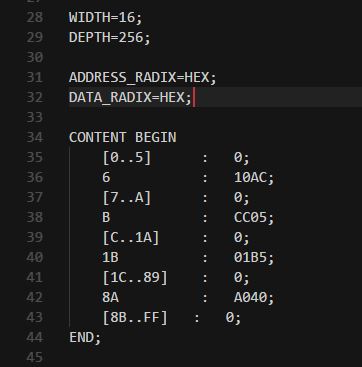


Figure : datapath timing diagram & memory mif file

The datapath worked as expected also, developing this testbench required considerably unorthodox strategies. Looking at memory data within modelsim showed initially that we were storing each element off by one. This was fixed by simplifying the ALU and changing how we called modules inside of the datapath along with other miscellaneous changes.

Writing the testbench for the datapath took a stretch of thinking as the module is so simple that it’s hard to write a test for such a simple system. The control unit was tested by simply traveling between states. We assign IR to be a specific value then go through each state obtaining what we expect.

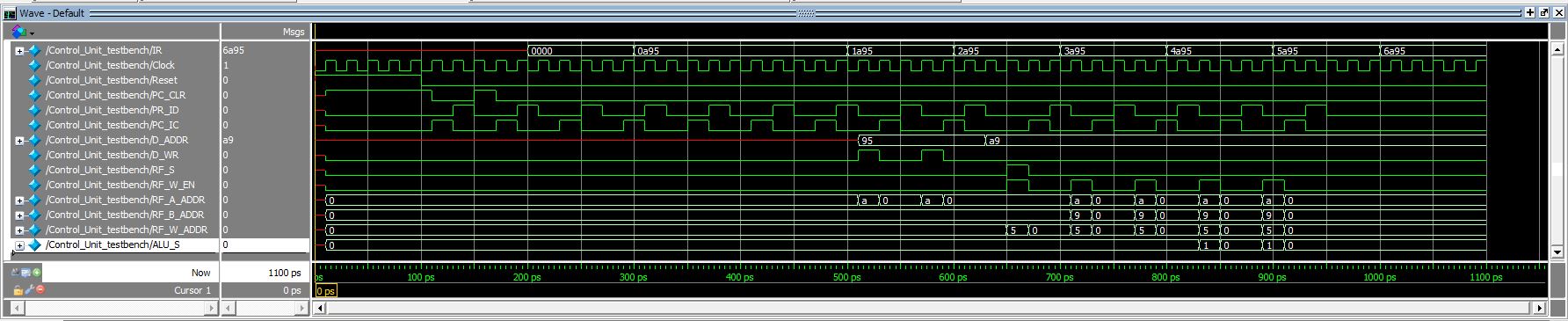


Figure 8: control unit (finite state machine) timing diagram

Writing the testbench was also not a simple affair even though the ALU itself was quite simple. We checked for signed and unsigned bit under and overflow and as such had to write a testbench that tested for such. Extensive use of casting was used, which we were new to in SystemVerilog. When developing processors for small embedded devices type conversions are a waste of processor time and as such conversion between signed and unsigned should be avoided as much as possible (White 2011, 240). The ALU testbench output exactly what we wanted it to.

For the two main operations asked of us in this assignment, addition and subtraction, the two’s complement addition is performed. The sum and difference are stored in the destination operand. The contents of the source operand are not affected. For testing purposes, we kept the bit width to two.

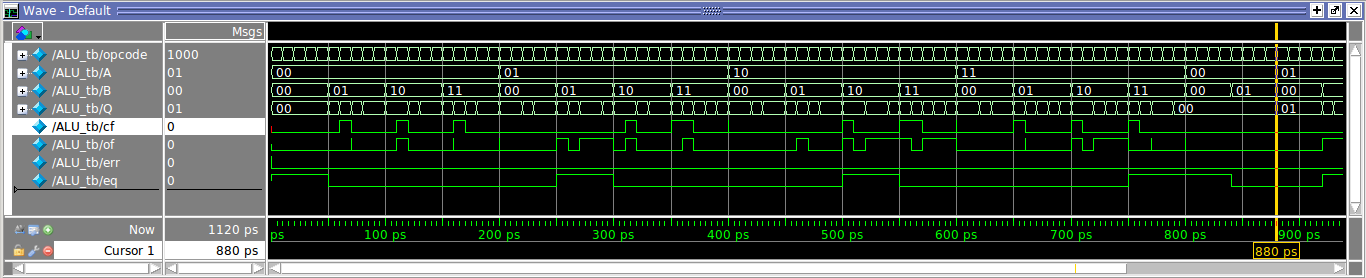


Figure 9: ALU timing diagram

We developed more processes for our ALU than we were asked to, including a barrel shifter. Barrel shifters are unique to ARM processors, and it is used to generate shift operations and immediate operands. The ARM processor has no shift instructions (Wang 2017, 13). Instead it uses a barrel shifter, which performs shifts in conjunction with other instructions (Wang 2017, 13). Our barrel shifter works in a similar way to that of what is inside the ARM processor.

Our integers are half the size of that in ARM processors, but it operates on the same principle. We do not use the instruction stream as data which presents problems if we were ever to use immediate values in the instructions (Wang 2017, 13).

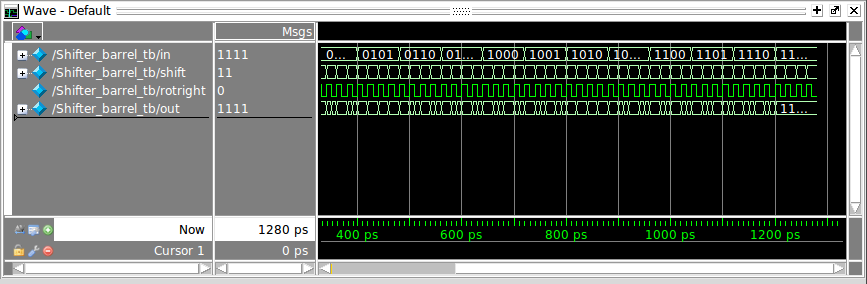


Figure 10: timing diagram for the barrel shifter

Experimental Results:

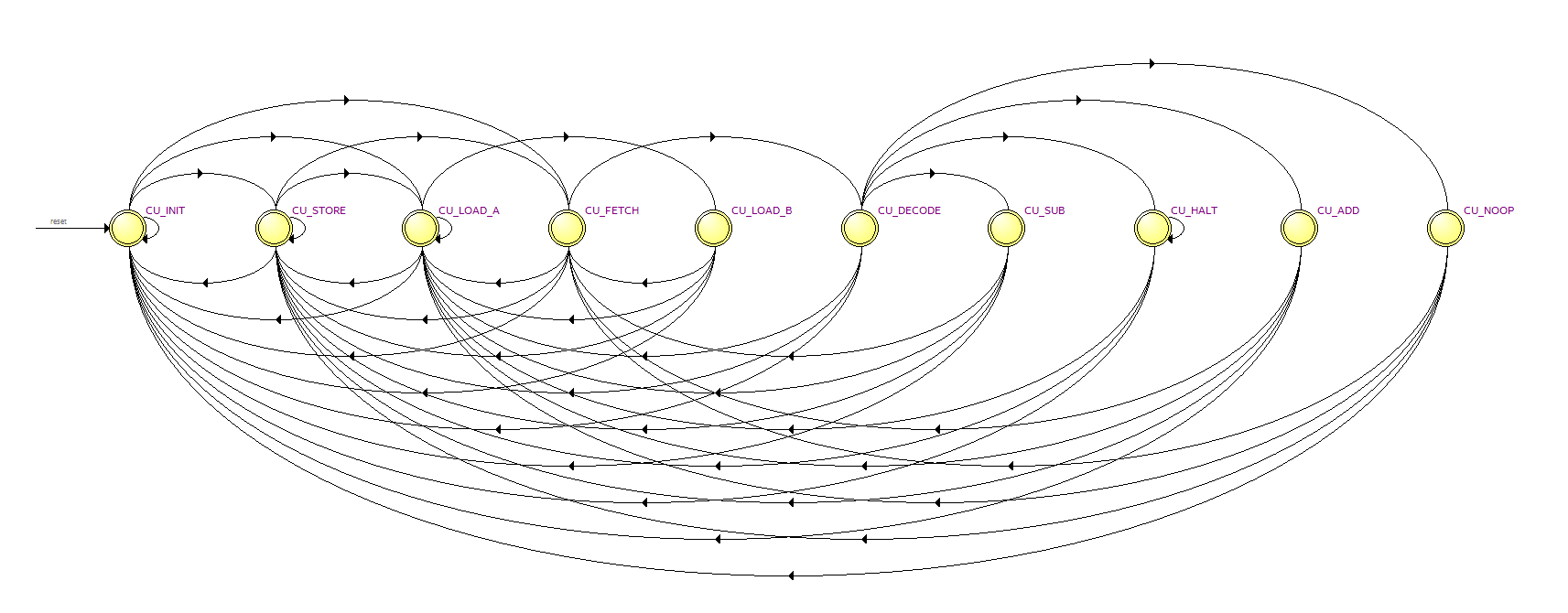


Figure 11: Weird results when we compile the same code on a windows machine (this is incorrect)

Figure 11 shows the result of compiling our project and looking at the FSM viewer for the control unit in quartus on Windows. When compiling, the same code, in quartus and looking at the FSM in quartus on a linux box we obtained the correct FSM. This is a problem when relying on other’s tools and dealing with black boxes.

We know the code is the same because we use git for source control. The same issues arose when dealing with the LPM library. When instantiating a module that quartus creates and it works the first time you don’t put much thought into it. As soon as an error does arise debugging that error becomes quite challenging. We spent a week alone, off and on, figuring out which library to call inorder to use the LPM RAM and ROM module correctly. Problems when using the LPM library appear to be specific to every use case, which is fine, albeit unfortunate as it becomes hard to debug your code by looking for help that others asked for. We believe that we fixed the latter problem by making a library call, but it is unknown as to the source of our solution as we were trying many things hoping one would work. The former problem was never fixed and is just a reality we accepted as to the limitations of the software as it would correctly in modelsim and on the FPGA when we send it to the board.

As for the results on the board, the CPU performs exceptionally, just as it did in the testbenches individually.

Conclusions:

We built a fully functioning CPU, with no delays, that meets all design specifications. No jump instruction was added because the CPU would have needed to have been re-written from the start to include it, due to the specifications for grading so we opted to not include the extra credit. Building small purpose-built CPUs comes up when designing embedded systems more often than one might think, and the experience gained from doing this assignment was invaluable. Personally, we explored many of the operations not yet exposed to us in class when researching what goes into modern CPUs such as barrel shifters and rotations. This gave us a much deeper appreciation of these operations and the design considerations.

Premature optimization is the root of all evil and products should work before they are tuned (White 2011, 249). We wrote each testbench and made sure each lower module was working before moving on. While this design process took a long time, when we finally got to the end everything just seemed to work. One of the reasons we decided not to include a jump wasn’t that it was difficult to implement a jump but because we didn’t do it at the start. Introducing complexity to a product right before you ship it is asking for disaster (White 2011, 249).

We got to build this computer ourselves, which means we got to define what the language is and what these commands are. Within reason we can come up with any operation we want, and call them whatever we want, if the operations correspond to operations that can be performed by a computer. There is a lot of freedom in what commands we can create. We built something that sequences through a set of instructions that we defined automatically. That is a very powerful idea and one that will stick with us for projects to come.

# Works Cited

Allen, Ian. n.d. *The CARRY flag and OVERFLOW flag in binary arithmetic.* http://teaching.idallen.com/dat2343/10f/notes/040\_overflow.txt.

Dandamudi, Sivarama P. 2005. *Guide to RISC Processors for Programmers and Engineers.* Ottawa, Ontario: Springer Science.

Landley, Rob, and Jeff Dionne. 2016. *Building a CPU from Scratch: jcore Design Walkthrough by Rob Landley & Jeff Dionne.* June 3. Accessed June 6, 2018. https://www.youtube.com/watch?v=lZGHbMS882w.

Mutlu, Onur. 2015. "Computer Architecture Lecture 5: Intro to Microarchitecture: Single-Cycle." Pittsburgh, Penssylvania: Carnegie Mellon.

Thimmannagari, Chandra. 2005. *CPU Design: Answers to Frequently Asked Questions.* New York: Springer.

Upton, Eben, Jeff Duntemann, Ralph Roberts, Tim Mamtora, and Ben Everard. 2016. *Learning Computer Architecture with Rasberry Pi.* Cambridge: Wiley.

Wang, KC. 2017. *Embedded and Real-Time Operating Systems.* Pullman: Springer.

White, Elecia. 2011. *Making Embedded Systems: Design Patterns for Great Software.* Aptos, California: O'Reilly Media.

Zhirkov, Igor. 2017. *Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture.* Saint Petersburg: APRESS.