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 defined 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 later 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. 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 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.

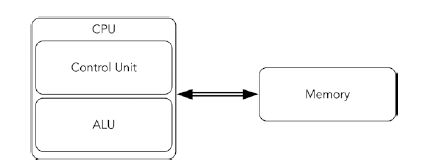


Figure 1: Simple von Neumann architecture (not exactly what we made here) (Zhirkov 2017, 4)

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. An adder module was added to test for signed and unsigned operation overflow and underflow. 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 we made had a saturation flag, which is often used in digital signal processing.

The control unit was developed 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 destination. Restricting operands, as we did here, to registers simplifies the control unit (Dandamudi 2005, 43). Many iterations of the control unit happened. At first were incrementing PC counter incorrectly, by doing so both in the fetch and init 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.

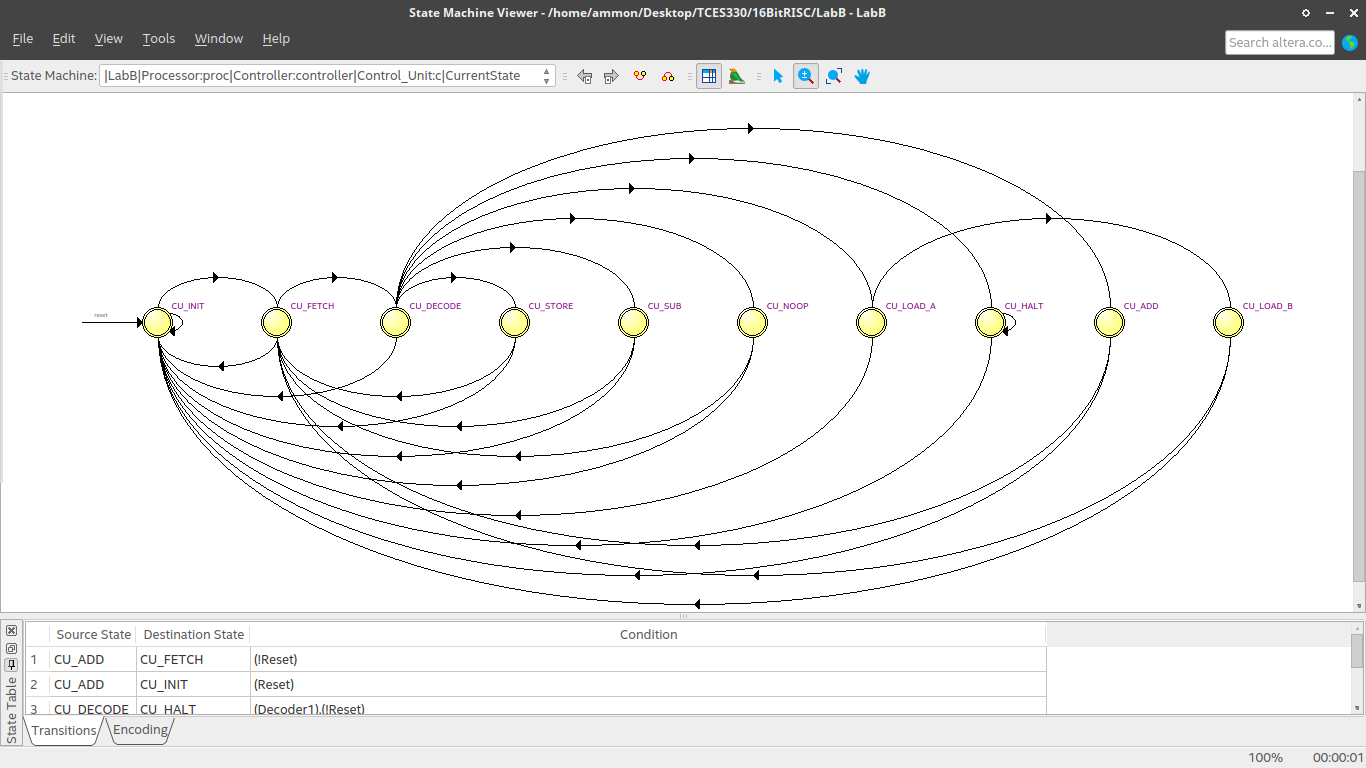


Figure 2: The finite state machine for the Control Unit

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 16-1 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 state the control signal must be set which will set the data path.

