Josh Wildey
EC 551
3/12/2018

Lab 1: ALU and Microprocessor Design and Implementation

Abstract:

The goal of this lab is to create a simple microprocessor (including an ALU) realized on a Spartan-6 FPGA to execute the simplified assembly code (with extensions). An external assembler, written in python, was used convert programs written in assembly into 'list' files that were read in by the Xilinx tools to then be converted into block RAM using a Von Neuman style architecture. Upon startup the microprocessor will start at the base address of 31 and execute instructions as dictated by the assembly program. 7-segment displays and the VGA port can be used to display statuses of registers, program counters, memory locations.

Software Design:

The overall module architecture is show in the figure below for what was able get implemented. An ALU Module was started that would have also been connected to the lab1 module but never finished. The software was designed as a single cycle CPU that would take multiple clock cycles to execute an instruction with the hope that once the single cycle was working, it could be modified in such a way to be a pipelined CPU.

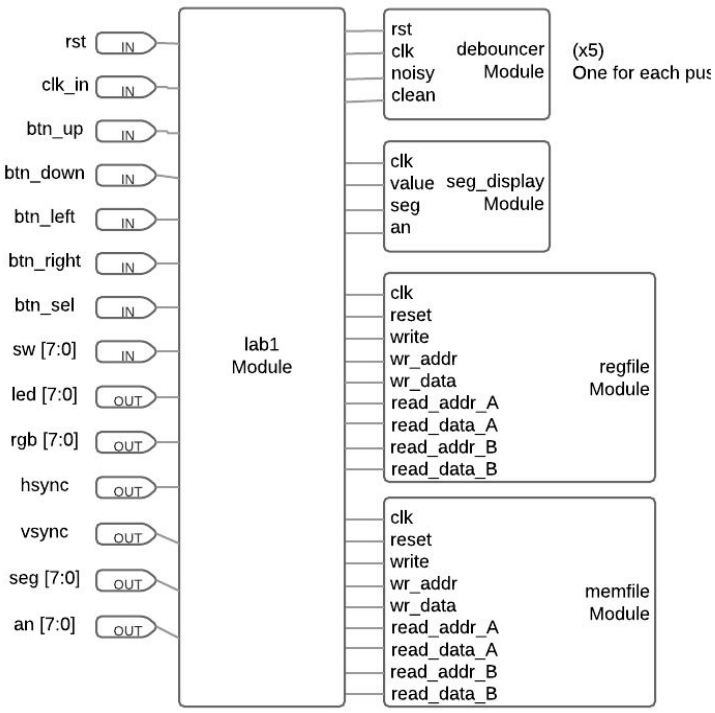

Figure 1: Overall Final Implemented Module Architecture

The top level module (lab1 module) of this design has all the inputs and outputs defined to interface with all the push buttons, switches, leds, 7 segment displays, VGA output, and the Micron Cellular RAM. The VGA and Cellular RAM were intended to be used but not implemented in the final design.

There is a Cellular RAM module written that partially works, but has issues with addressing which was not figured out in time for the final submission. This module has a few influences in the code base even though it was not used. The first of which is the clock determination. The clock used in the top level module was divided down from 100 MHz to 1 MHz for easier math if clock manipulation was necessary and also to allow for a memory read/write to the Cellular RAM within one (1) clock cycle. The Cellular RAM utilizes the full 100 MHz clock to achieve the specific timing requirements for asynchronous reading/writing. The timing diagrams generated by the simulation and can be seen in the diagrams below.

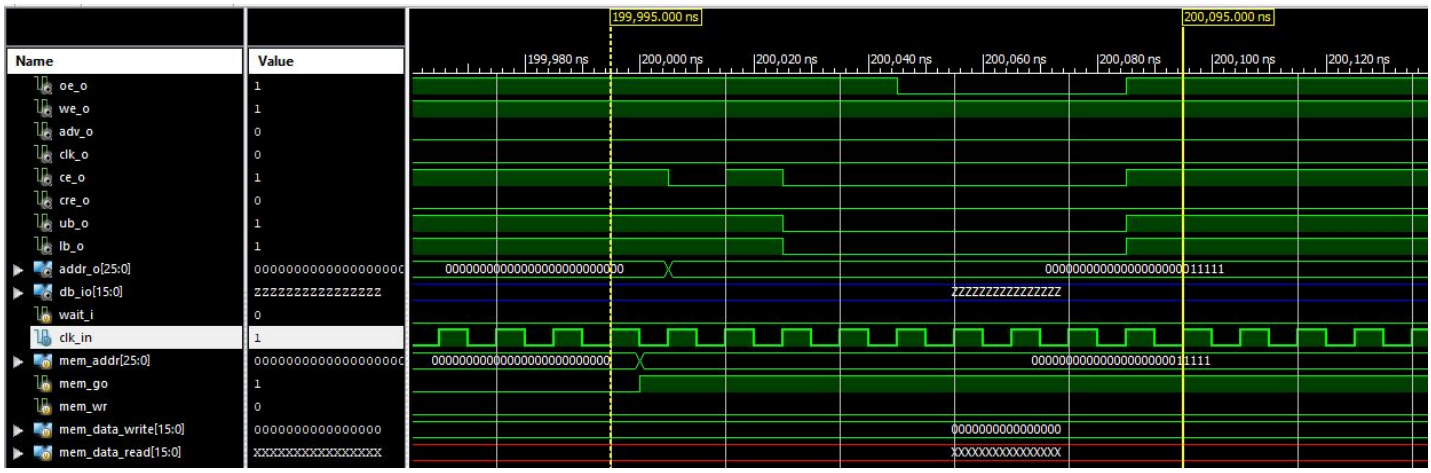

Figure 2: Cellular Memory Asynchronous Read

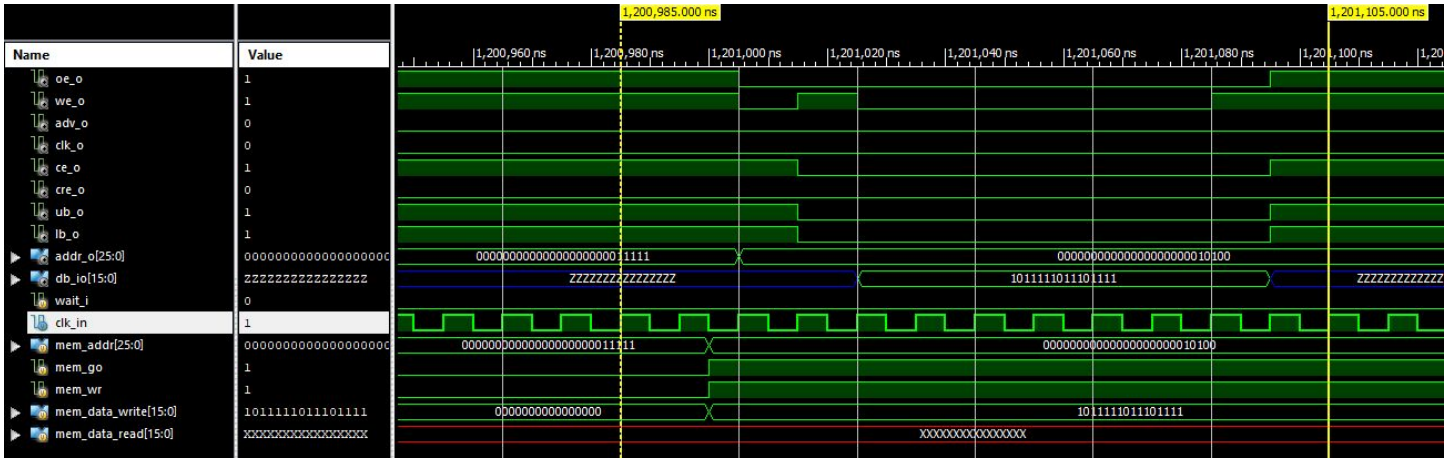

Figure 3: Cellular Memory Asynchronous Write

The lab1 module has 2 parameters that can be modifed. One is the address after a reset happens, which as of right now defaults to 31 due to the python script that generates the

memory (to be covered in more detail below). The other is the time to stay in the initialize state. This time is another requirement by the Cellular RAM because it takes approximately 151 μs to power up and initialize, so the initialize time is defaulted to 200 μs.

There are a number of registers at the top level to keep track of the state, the previous state, the program counter (PC), a status register (CC), OP code and its operands. There are all self explanatory and are the main variables that are used in the state machine. The main state machine for the microprocessor is found on line 351 of *lab1.v*. It operates in an always loop on the positive edge of the 1MHz clock and checks for button presses while simultaneously running through the state machine. Details of the state machine are detailed below in Figure 4.

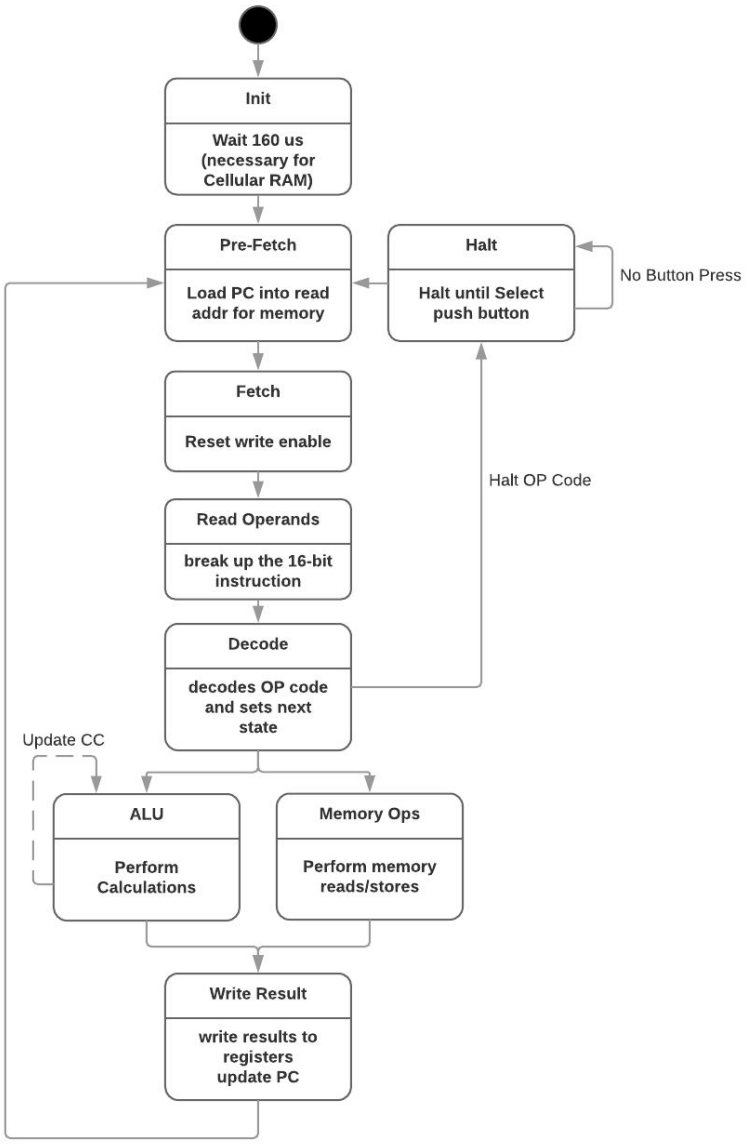

Figure 4: Main Microprocessor State Machine

Within the state machine, tasks are called to perform operations associated with the current state. The tasks were written to keep the state machine legible and modular. The implemented tasks for the final submitted product were for reading/writing to the register file, reading/writing to the memory file, and decoding the current instruction and determining the next state.

The memory and register files both have the same interface and operate in similar fashion. The only difference is that the memory file has an initial block that uses the *$readmemh* command to load the '.list' file. The assembler used to generate the '.list' file is a modified version obtained from the sx86-emulator from the EC327 Git repository[3]. It was modified to take a command line input for a name of a file (which has the written assembly code for the program to be run) and it generates the '.list' file for memory addresses 0-60. The memory file was written to use a 12-bit address field to accommodate the instruction set and the full 4096 address possibilities, but the implementation only uses 60 to cut down on synthesis and creating the bitfile. The other difference between the register and memory files is that the register file only has a 3-bit wide address field since this implementation only requires 6 general purpose registers.

The 7-segment display module can display both register values and memory values. The left push button is used to display registers, and the right push button is used to display the memory location of the current instruction. The left and right push buttons switch a bit controlling a MUX determining which value to display. The Up and Down push buttons were programmed to choose which register (0-5) to display though it did not appear to work during the demo. The LEDs displayed the binary encoding of the register address to display.

Of the OP codes that were required to be implemented in this lab, only a few were successfully completed and tested. Those OP codes were Halt, and a couple move operations. The jump commands and arithmetic commands had partially written code that was in place to test state machine transitions, but never fully tested. The implemented instructions took 4 clock cycles to complete. The way the code was structured allowed for modification to be a pipelined CPU instead of a single cycle which was the original intent. The jump instructions and arithmetic instructions would likely use less and more clock cycles, respectively, due to their more simplistic and complex natures.

A timing diagram can be found below in Figure 5 that demonstrates the successful transitioning through the state machine and performing register and memory manipulation. The program used to run this is listed below.

```
mov R0,1  ;put the number 0 into register R0
;halt     ;halts execution until push button is pressed
mov R1,2  ;put the number 2 into register R1
;halt     ;halts execution until push button is pressed
add R0,R1 ;add the contents of R1 to R0; R0new = R0old + R1 = 3
;halt     ;halts execution until push button is pressed
add R0,R1 ;add the contents of R1 to R0; R0new = R0old + R1 = 5
;halt     ;halts execution until push button is pressed
```

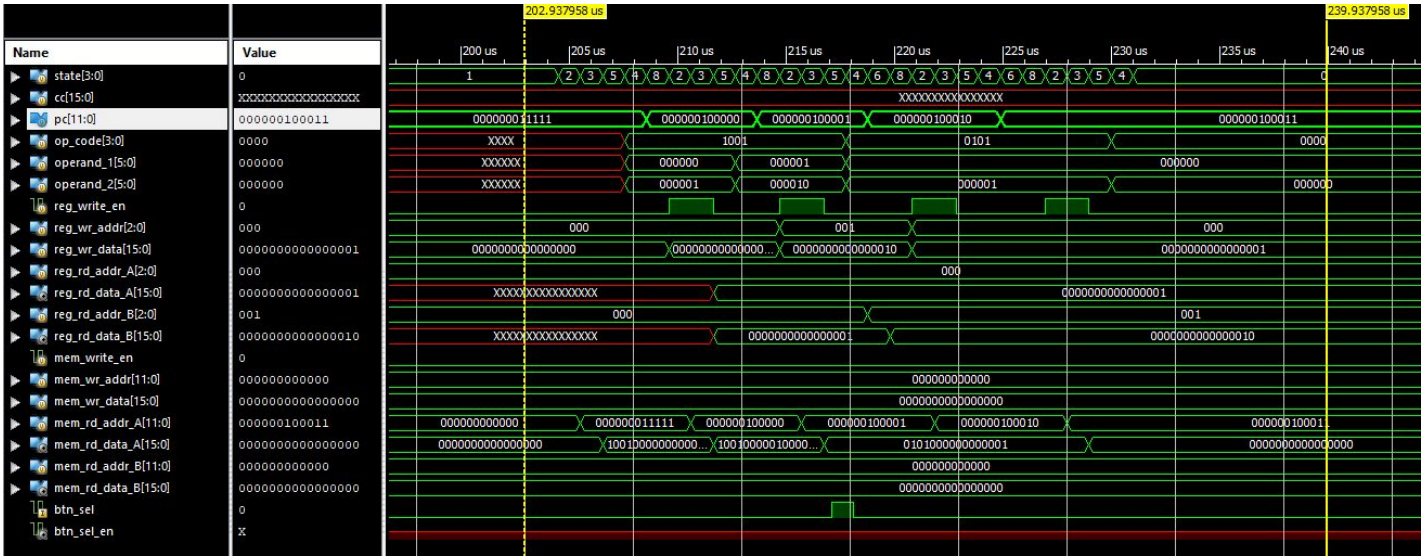

Figure 5: Main Microprocessor Timing Diagram

Assembler:

Python 2.7 is required to use the assembler. To run the modified version of the assembler the following command needs to be run:

```
python sx86-assembler.py <filename.asm>
```

This python script will generate a '.list' file that needs to be placed in the same directory as the source verilog files so that the memfile.v can read it in and initialize the memory.

Conclusion:

The final submitted solution contains a partially implemented microprocessor. A working state machine is implemented to cycle through a select few OP codes with other paths untested. Stubs are written in a modular sense for implementations of the other OP codes however due to (personal) timing constraints remain unfinished. The 7 segment display was used to display both register values and memory values (that were specified by the program counter).

References

1. Nexys3 Datasheet https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys3/documentation/Nexys3_rm.pdf
2. Micron Cellular RAM Datasheet http://www.stm32circle.com/resources/Datasheets/16mb_burst_cr1_0_p23z.pdf
3. Sx86-emulator https://github.com/cwoodall/sx86-emulator