Matthew Lumantas

Baldomero Vargas

EE120A

Lab 8 - Project 7, Processor

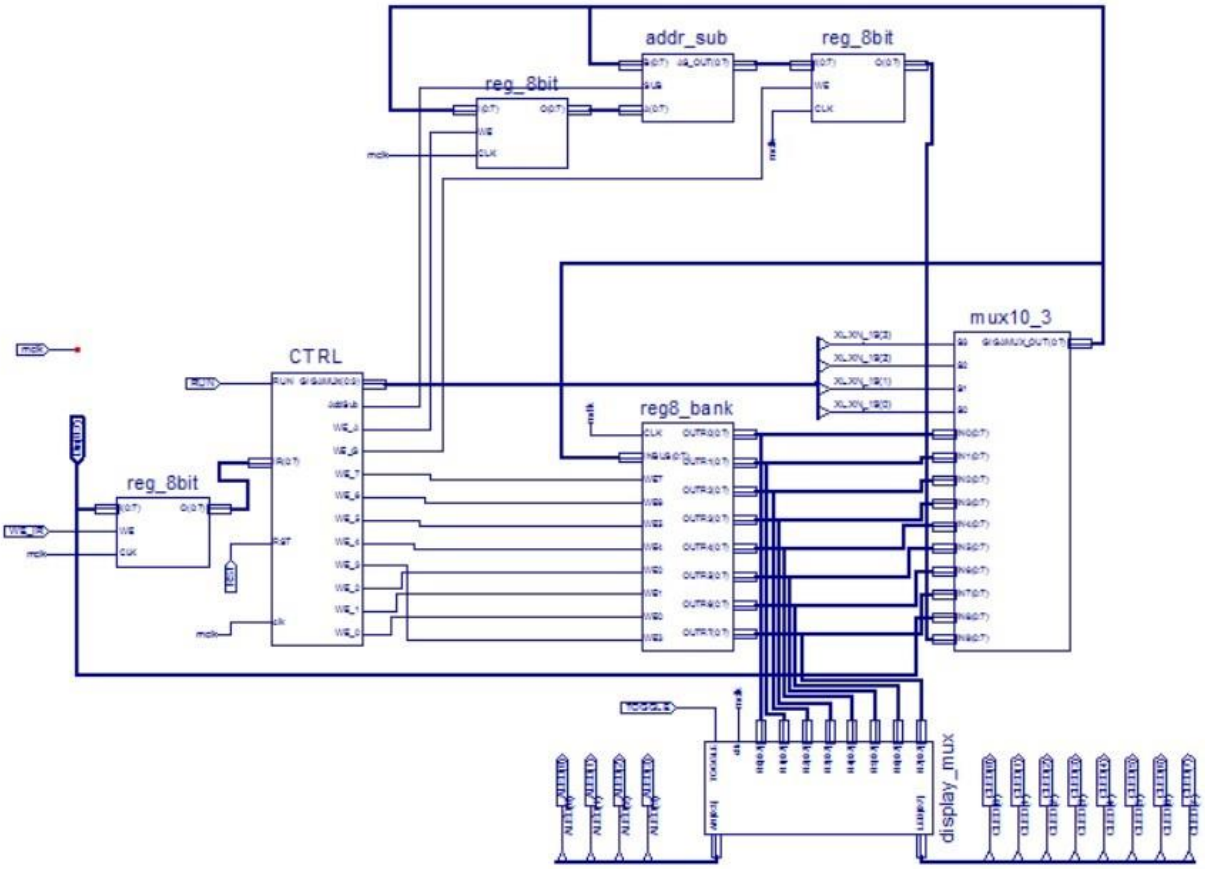
Introduction

The aim of this project is to design and implement all elements of a Von Neumann model computer onto the BASYS2 FPGA unit, with the exception of a dedicated memory component. A Von Neumann computer consists of an input device, output device, a processor consisting of a control unit and arithmetic / logic unit (ALU), and a memory unit. The specifications consisted

of the implementation of four different instructions, a register bank of eight registers labeled R_0 to R_7, and an ALU that can add and subtract. The four instructions are as follows:

Instruction	RTL Function
mov R_x, R_y	R_x <- [R_y]
movi R_x, D	R_x <- D
add R_x, R_y	R_x <- [R_x] + [R_y]
sub R_x, R_y	R_x <- [R_x] - [R_y]

The primary concept introduced with this lab is the creation and implementation of a high-level state machine. This allows us to sketch out the state machine for the datapath level components that we can then use to create gate level designs and finite state machines for the processor.



Schematic 1: Processor Overview Analysis

We decided to create an eight-bit processor due to the specifications: The four required instructions can be encoded using two bits, and two different selections of eight different

registers use three and three bits, neatly taking up the total eight bits. Input is handled using the eight switches on the BASYS2 for data and instruction input and the buttons are used as display toggle, reset, instruction load, and instruction execute. The seven segment display is used to output the contents of each possible register, cycling through the register bank using the display toggle button. The control unit and ALU are implemented by schematic to be loaded onto the BASYS2 board.

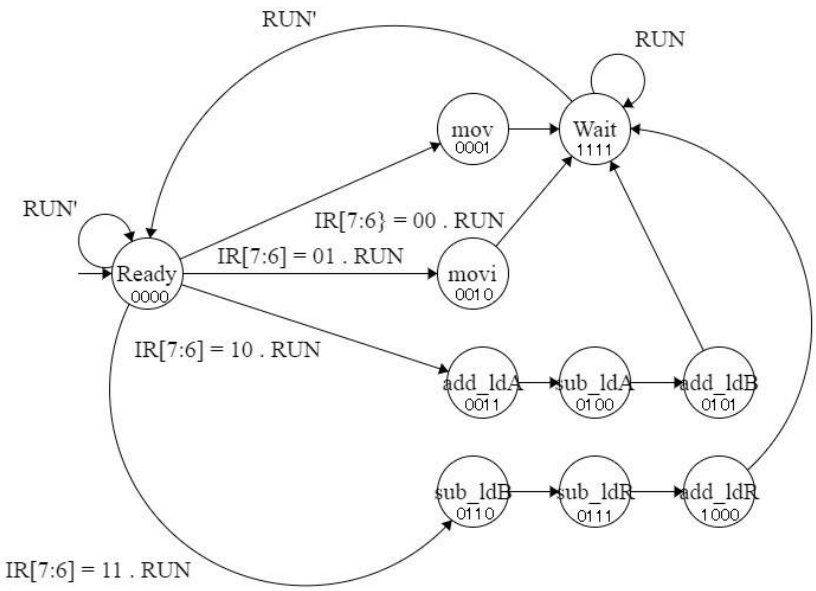


Figure 1: High-level State Machine for Eight-bit Processor

Figure 1 is a simplified image of the HLSM we used in the processor without the signals asserted at each state. Table 1 is the truth table for this machine with the control signals. Each of the four instructions are shown, with the "Wait" state acting as a falling-edge trigger to only allow the processor to execute the instruction once for every press of the run button. If the

instructions were to directly return to the "Ready" state then the processor would run the instruction thousands of times within the press-depress of the run button.

Run	IR[6:7]	IR[3:5]	IR[0:2]	C[0:3]	MuxEN	Mux[0:3]	WER	Rdec[0:2]	ALU	WEA	WEG	N[0:3]
0	xx	xxx	xxx	0000	0	xxx	0	xxx	x	0	0	0000
1	00	xxx	xxx	0000	1	00	1	IR[3:5]	x	0	0	0001
1	01	xxx	xxx	0000	1	10	1	IR[3:5]	x	0	0	0010
1	10	xxx	xxx	0000	1	01	0	xxx	x	1	0	0011
1	11	xxx	xxx	0000	1	01	0	xxx	x	1	0	0110
x	xx	xxx	xxx	0001	0	xxx	0	xxx	x	0	0	1111
x	xx	xxx	xxx	0010	0	xxx	0	xxx	x	0	0	1111
x	xx	xxx	xxx	0011	1	00	0	xxx	0	0	1	0100
x	xx	xxx	xxx	0100	1	11	1	IR[3:5]	x	0	0	0101
x	xx	xxx	xxx	0101	0	xxx	0	xxx	x	0	0	1111
x	xx	xxx	xxx	0110	1	00	0	xxx	1	0	1	0111
x	xx	xxx	xxx	0111	1	11	1	IR[3:5]	x	0	0	1000
x	xx	xxx	xxx	1000	0	xxx	0	xxx	x	0	0	1111
1	xx	xxx	xxx	1111	0	xxx	0	xxx	x	0	0	1111
0	xx	xxx	xxx	1111	0	xxx	0	xxx	x	0	0	0000

00 = zext[IR[0:2]]
01 = zext[IR[3:5]]
10 = Din
11 = Gout

Table 1: Processor Control Truth Table

Equations

MuxEN = Run.C3'.C2'.C1'.C0 + C3'.C1.C0 + C3'.C2.C0'

Muxmux1 = Run.IR[7]'.IR[6].C3'.C2'.C1'.C0' + C3'.C2.C1'.C0' + C3'.C2.C1.C0

Muxmux0 = Run.IR[7].C3'.C2'.C1'.C0' + C3'.C2.C1'.C0' + C3'.C2.C1.C0

WER = Run.IR[7]'.C3'.C2'.C1'.C0' + C3'.C2.C1'.C0' + C3'.C2.C1.C0

ALU = C3'.C2.C1.C0'

WEA = Run.IR[7].C3'.C2'.C1'.C0'

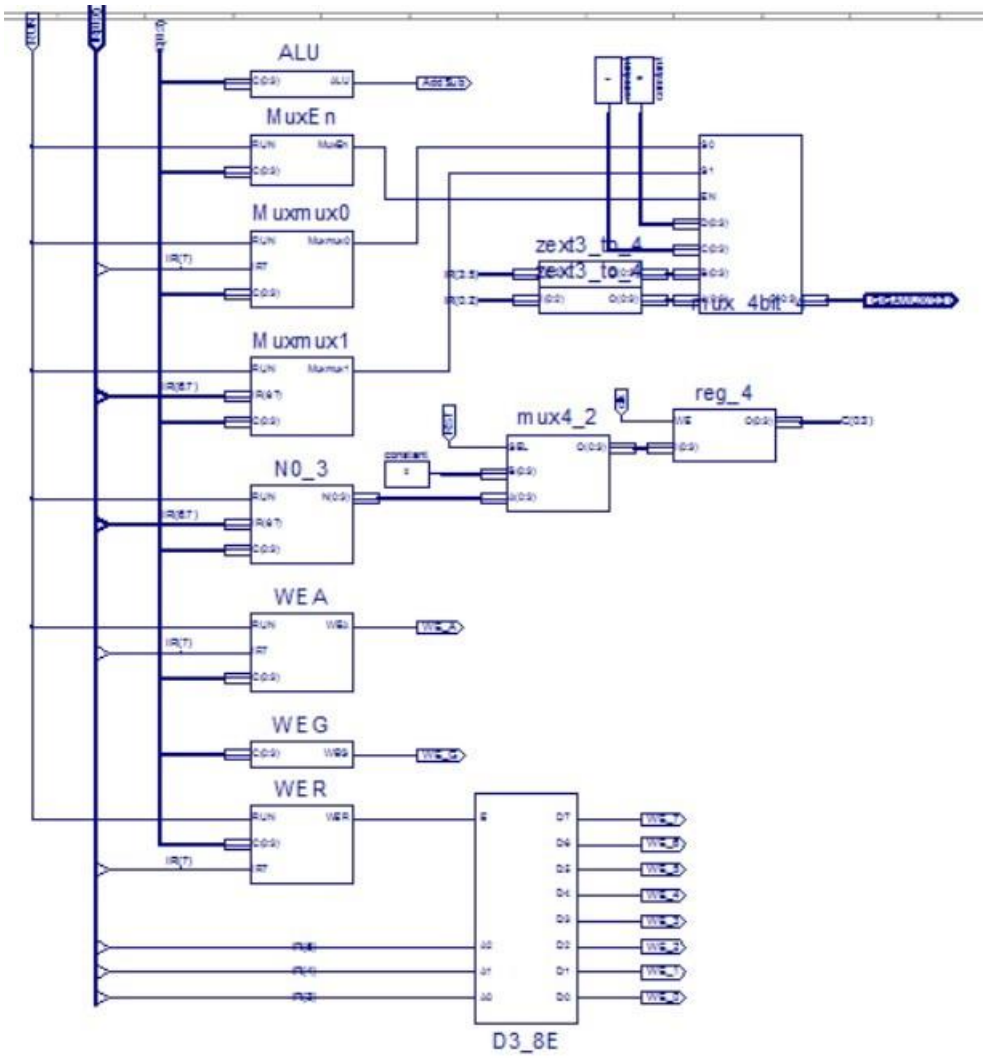
WEG = C3'.C2'.C1.C0 + C3'.C2.C1.C0'

N_3 = C3'.C1'.C0 + C3'.C2'.C1.C0' + C3'.C2.C0 + C3.C2'.C1'.C0' + Run.C3.C2.C1.C0

N_2 = Run.IR[7].IR[6].C3'.C2'.C1'.C0' + C3'.C2'.C0 + C3'.C1.C0' + C3'.C2.C1' + C3.C2'.C1'.C0'
+ Run.C3.C2.C1.C0

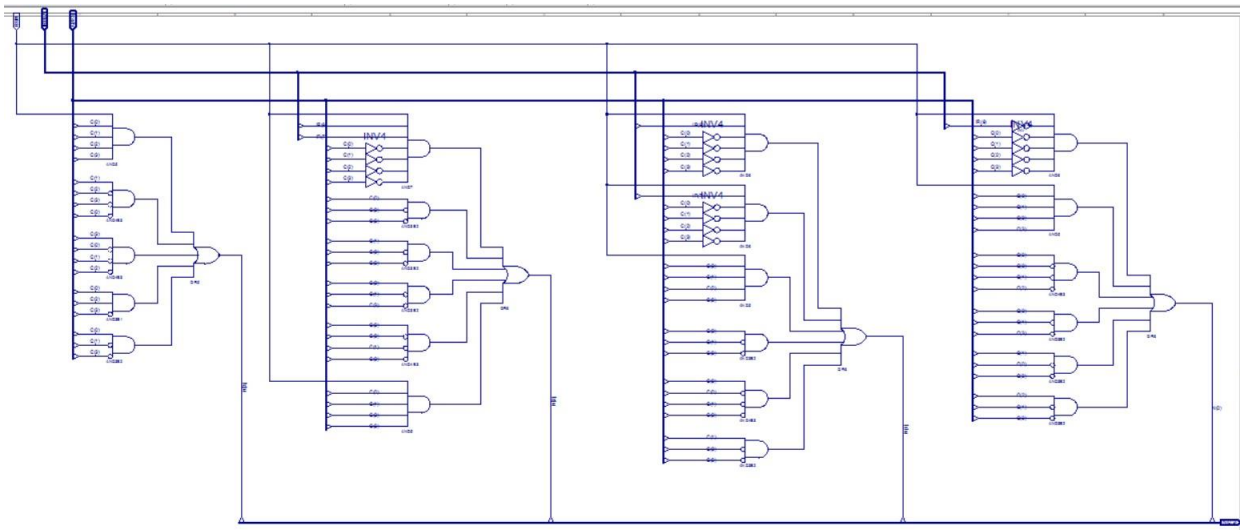
N_1 = Run.IR[6].C3'.C2'.C1'.C0' + Run.IR[7].C3'.C2'.C1'.C0' + C3'.C1.C0' + C3'.C1'.C0 +
C3.C2'.C1'.C0' + Run.C3.C2.C1.C0

N_0 = Run.IR[6]'.C3'.C2'.C1'.C0' + C3'.C1'.C0 + C3'.C1.C0' + C3'.C2.C1' + C3.C2'.C1'.C0' +
Run.C3.C2.C1.C0

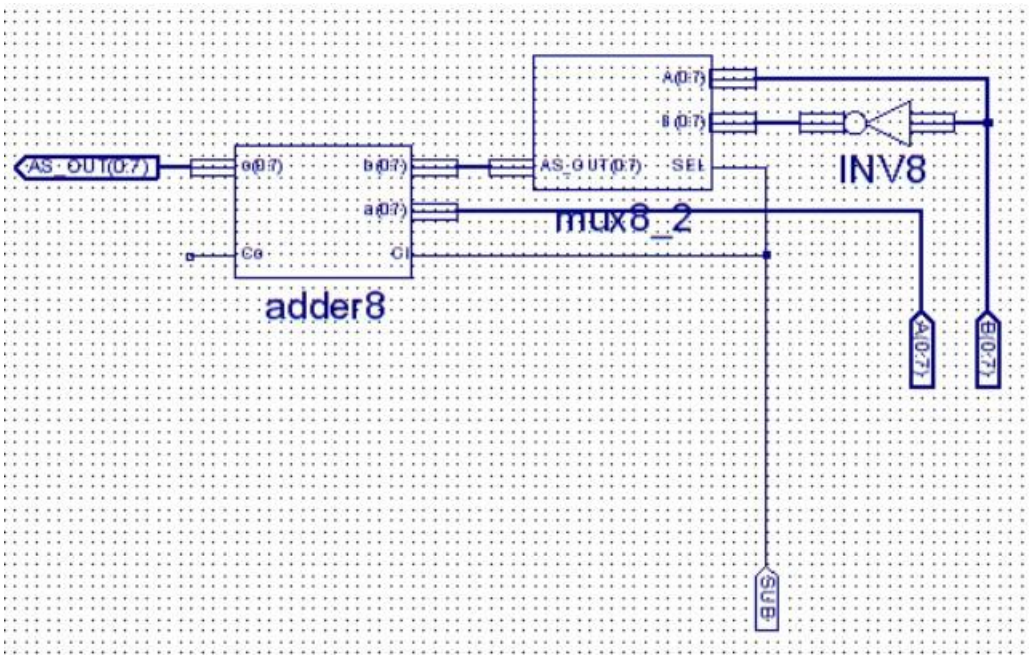


Schematic 2: Control Unit

Above the different equations are shown as black boxes; these are then connected to some additional logic before being sent back up to the datapath level design. The top right logic determines the signal sent to the large multiplexer at the datapath level. This mux selects the input select lines for the higher mux. The middle right logic is the memory register for the state machine with a reset to zero selection at the mux. The bottom right logic is the write enable logic for the processor. Because the Rx component of the instruction are the only possible write destinations we can directly connect the decoder to the instruction word and have enable logic to allow us to write as needed.



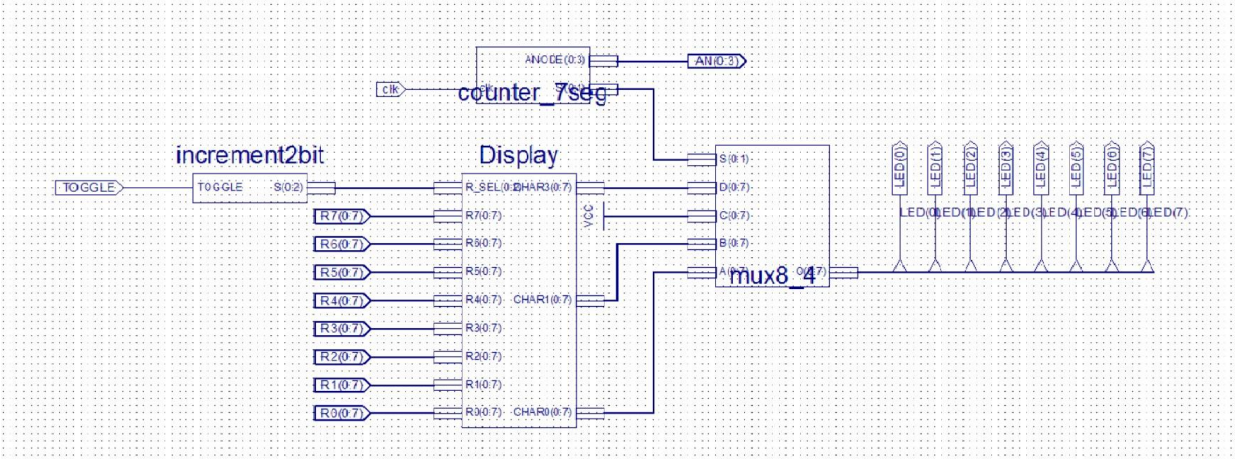
Schematic 3: State Control



Schematic 4: ALU

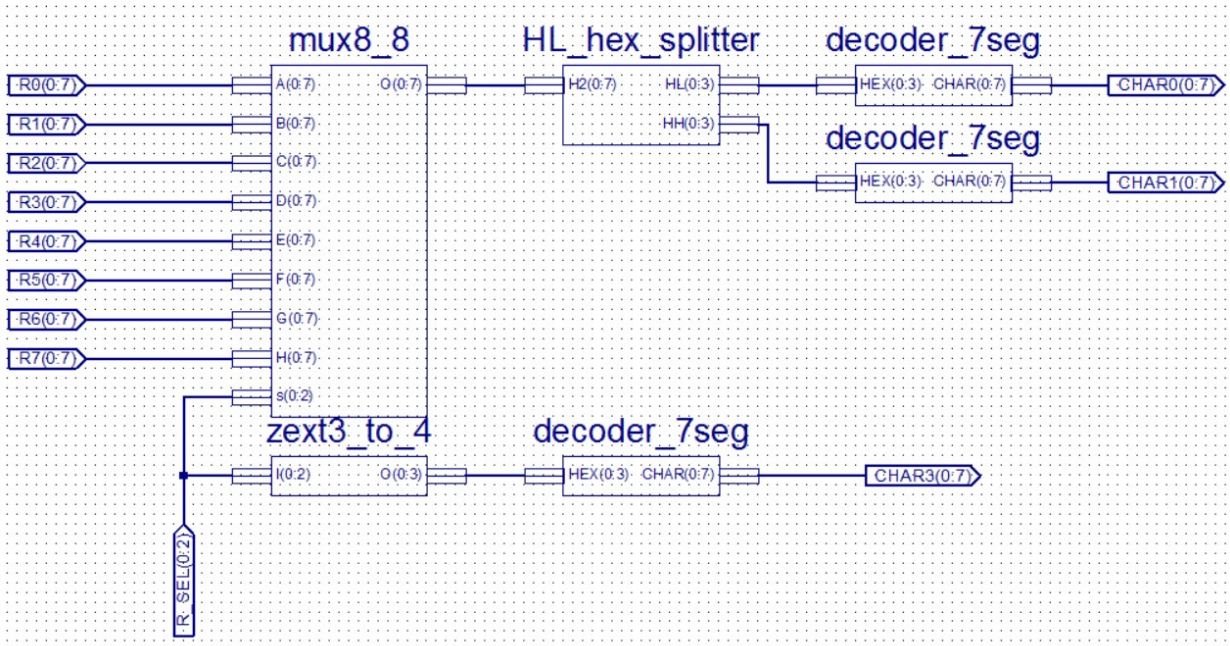
The ALU we implemented is quite simple, as it can only perform addition and subtraction. Because we are working with two's complement signed numbers subtraction is performed by inverting one input and asserting the carry-in at the adder. As seen in the datapath level schematic, the ALU also uses two registers: one located at one of the inputs to the adder and one at the output of the adder. The input register allows us to perform arithmetic with two numbers using a single bus. The first number is placed on the bus and stored, then the second

number is placed on the bus. One of the ALU's inputs comes off the register and the other directly off the bus. Thus, we can work with two numbers. The ALU output register allows us to read the output of the ALU only when we need to. While the register is not write enabled the ALU can output whatever values it may happen to calculate without affecting the processor's operation.



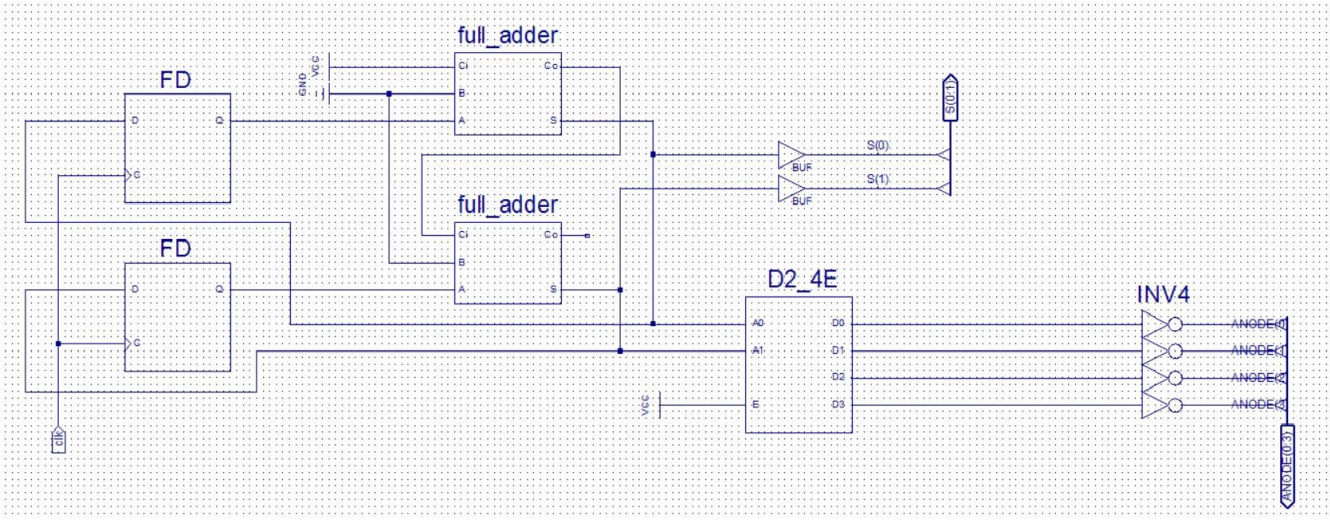
Schematic 5: Seven Segment Display Logic

This schematic details the operation of the register display. One seven-segment display shows the current register that is being viewed and two other displays show the 8-bit value of the register in two hex characters. The "increment2bit" component is actually a 3-bit incrementer: Every time "TOGGLE" is asserted the output increases by one. This value is the register that we wish to output to the display. The "Display" component contains logic to output the register and its value, which will be viewed more in-depth later. The "counter_7seg" and mux control the display multiplexing that we first implemented in Lab 4, Part 3.



Schematic 6: Register Display Output

This design takes in values from each of the eight registers in the processor register bank, splits the values into a most significant hexadecimal character and least significant hexadecimal value. It also takes in the selector value for the registers and converts that into a hex value. These three four-bit values are each connected to a seven-segment character "decoder" that lights up the necessary segments to display that value.



Schematic 7: Display Counter

In order to display different characters on the seven-segment package, the characters must be connected to a time multiplexer. This is almost exactly the same as the component used in Lab 4 to multiplex the displays. Every clock pulse the selector increments by one, selecting a single particular seven-segment to be lit and outputting a selection to connect a particular character output from the "Display" logic.

Processor UCF

```
NET "mclk" LOC = "B2" | CLOCK_DEDICATED_ROUTE = FALSE;

NET "Din(0)" LOC = "P11";
NET "Din(1)" LOC = "L3";
NET "Din(2)" LOC = "K3";
NET "Din(3)" LOC = "B4";
NET "Din(4)" LOC = "G3";
NET "Din(5)" LOC = "F3";
NET "Din(6)" LOC = "E2";
NET "Din(7)" LOC = "N3";

NET "RUN" LOC = "G12";
NET "WE_IR" LOC = "C11";
NET "RST" LOC = "M4";
NET "TOGGLE" LOC = "A7";

NET "CLED(0)" LOC = "L14";
NET "CLED(1)" LOC = "H12";
```

```
NET "CLED(2)" LOC = "N14";
NET "CLED(3)" LOC = "N11";
NET "CLED(4)" LOC = "P12";
NET "CLED(5)" LOC = "L13";
NET "CLED(6)" LOC = "M12";
NET "CLED(7)" LOC = "N13";

NET "ALED(3)" LOC = "K14";
NET "ALED(2)" LOC = "M13";
NET "ALED(1)" LOC = "J12";
NET "ALED(0)" LOC = "F12";
```

Discussion

Specification

The system works entirely as requested in the specification. We can load in immediate values from the input switches to the register, which then can be displayed on the sevensegment. We can copy values from register to register. Finally, we are also able to correctly add and subtract values from two registers. We encountered a number of problems implementing the schematic. Designing the system itself at the datapath level was relatively easy compared to the implementation at the gate level.

Problems and Issues

First, our control signals were not being asserted at the correct state. We traced this to making many mistakes simply transferring boolean equations into the schematic: connecting wrong wires together and incorrectly optimized equations. We solved this by testing each signal individually and comparing it to the desired operation described in the truth table, adjusting the logic for any incorrect signal.

Secondly, we countered glitches in the seven-segment display. The display would not completely multiplex each character solely to its assigned display. The characters would "ghost" between all four displays, preventing being able to read the values. This was a result of the BASYS2's internal oscillator's frequency of 50 MHz, which was extremely fast for our system. The problem was solved in two ways: By creating a rudimentary clock divider using adders that divided the clock by 128, and by changing the master clock from the on-board oscillator to an external function generator set at 200 KHz.

The third, and last, major problem was incorrect operation of addition and subtraction. The board would incorrectly add or subtract values and get odd results almost completely at random. This was traced again to the speed of the on-board oscillator. Slowing down the clock speed completely fixed the issue. We theorize that the speed of the clock was too fast for our likely largely inefficient method of implementation for the entire processor.

Possible Improvements

Because of our choice of eight-bit instructions, it would be nearly impossible to add any more instructions without drastically redesigning the processor. Moving to even one more bit in each instruction for a total of nine would allow us to implement another four instructions.

The register display component does not account for two's complement signed numbers. It will output the eight-bit value contained in a register as unsigned. This would require a not-insignificant amount of redesign of the display character logic but because we have an unused seven-segment display we could use it to display a negative sign.

The "RESET" button does not actually do anything significant. An early (and fairly simple to implement) improvement to the processor would zero out all the registers when the reset button is pressed. All this would entail would be a mux before the register that loads integer 0 to the register and an extra OR gate to the write enable.

Conclusion

This project was undoubtedly the most difficult specification we have had to implement. In hindsight, however, the design was not very complicated. We had doubts about if the timing of the signals from the state machine were correct, but that was the only uncertainty we had in terms of design. Nearly all of the time spent working on this project was spent debugging. Most of our problems were simply annoyances of having to use the Xilinx ISE design program or having to deal with a very fast clock. We decided to forgo using simulated test benches because we felt that it would be faster and more simple to add a few I/O markers and upload the component to the board. We both have a very strong foundation from the CS61 class which greatly helped our processor design. By breaking down each complex black-box into more simple components revealed that a processor could be made solely with the components and ideas from previous labs in this class: muxes, registers, counters, adders, decoders, and basic combinational logic.