Lab Report:
ALUs and Registers

Nisha Patel

November 6, 2016

Abstract

In this report, I am discussing about creating a sequential circuits. We are continuing to develop a working CPU.

1 Introduction

For this lab, we are continuing to develop a working CPU. Using the components and skills learned in the previous lab, we are going to create an ALU and start in on a Register File. As before, we will be working with GHDL to create the components that will go into it. You will need to build and test using GHDL, and observing the outputs via gtkwave to ensure it is behaving appropriately.

2 Background

Previously, we were working exclusively with combinatorial circuits. These are nice and asynchronous - timing doesn't matter, and there is no such thing as state. Now that we're moving on to sequential circuits, things get interesting. We use keyword inout to specify that it acts as both input and output. We can analyze, build the executable, and run it, but when we get an error.
error: simulation stopped by –stop-delta
This is because the circuit will not enter a stable state (this is expected). One interesting thing to note: we used the keyword inout to specify that it acts as both input and output.The problem is that we are running a simulation (things don't quite work out how they do in the real world). So, we need to fudge things a bit.
Things to note:

- Everything is in a process: this means that it will happen in order, not asynchronously
- GHDL is case insensitive.

3 Experimental Setup

Parts List

- A selector (2 bits for selection, for now)
- A 3-bit Decoder (for now)- you should be able to reuse the one from lab 5
- 4-1 bit ALUs
- 4 bit Register
- Register File with 8 Registers

4 Experiments

We're going to work on building our way up to a working CPU that can handle most (if not all!) of the LEGv8 instruction set. For now, we are focusing solely on building up an ALU and working towards a register file.

4.1 Experiment 1: Building your selector(or MUX)

In the first experiment, we will be building our selector (or MUX). We have seen these several times in class so far, and we will need to support 3 operations: AND, OR, ADD (also subtracts, but that will be implemented separately). A selector is going to be of 2 bits. It takes four bits input and selects only 2 bits.

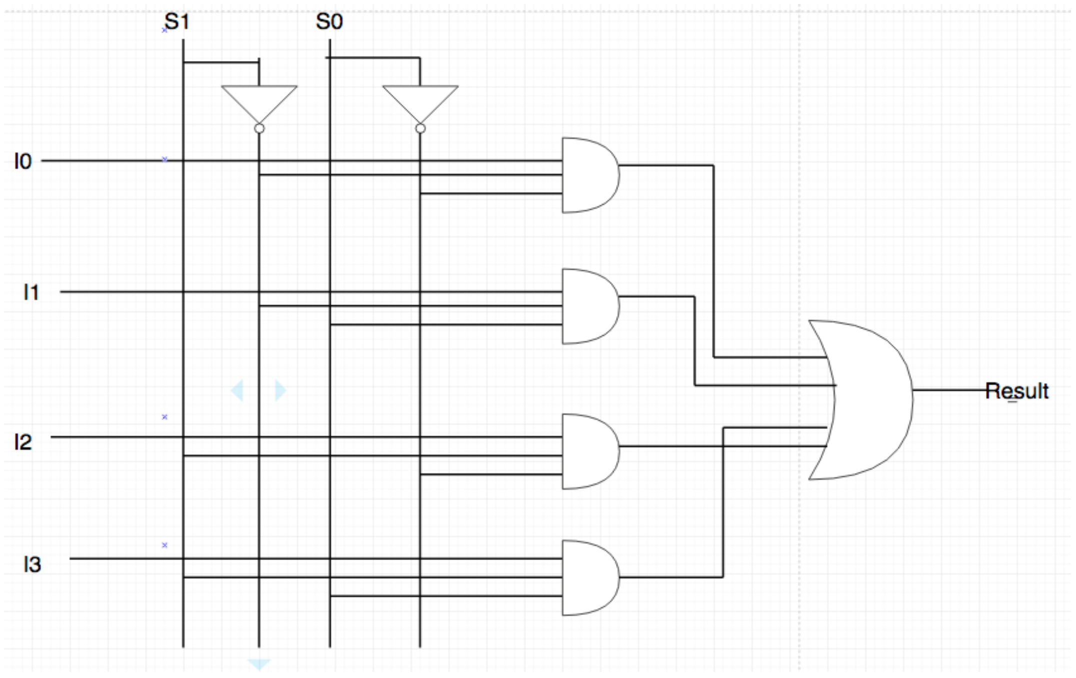


Figure 1: This is a logic gate of a 2-bit selector

4.1.1 Experiment 1 Conclusions

In class, we show a blackbox approach first and then we saw the logic gate for mux. I think what I build for this experiment is quite similar as we saw in class. It takes 4 inputs and the mux selects two out of four and gave the final output. I choose to used and, or and not gates to implement my selector. First I started building the first and gate and then second and so one.

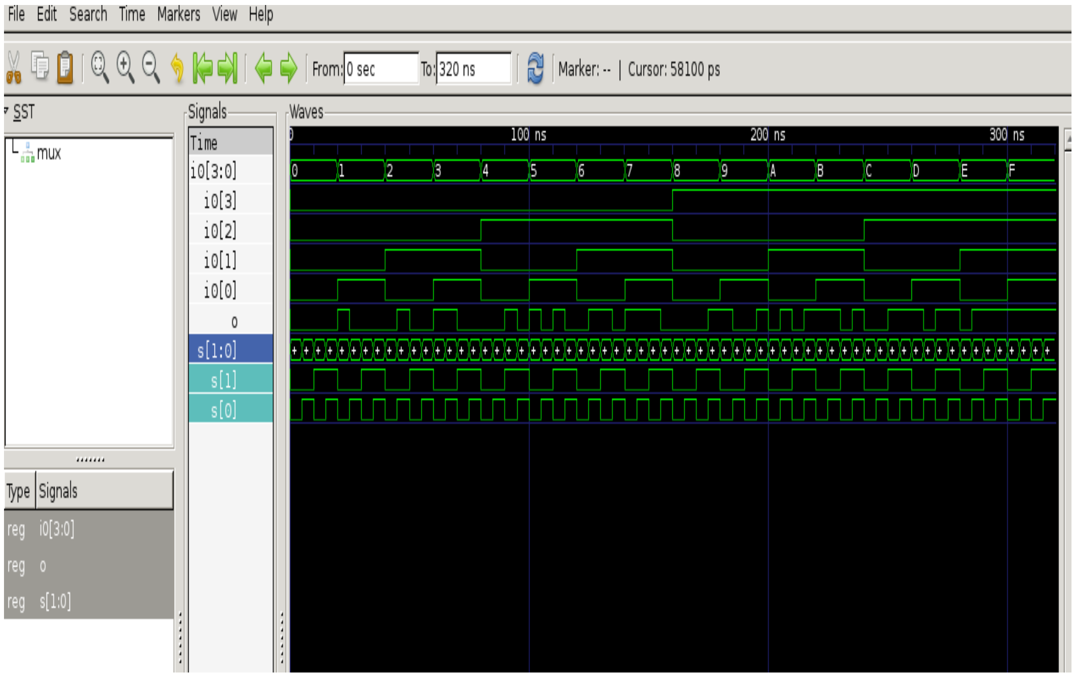


Figure 2: This is a gtkwave of a 2-bit selector

4.2 Experiment 2: 1-Bit ALUs

In this experiment, we are creating a 1-bit ALUs using our mux and full adder (that we created for lab 5). We are using p A-32 figure from our book. It is the most advanced one we looked at so far. Your inputs should include:

- The 2 bits you will be working on (let's call them A and B in here)
- Anot (a flag to invert A)
- Bnot (a flag to invert B)
- Carry in
- 2 bits for MUX (to determine what task will be accomplished)

Your outputs should include:

- Carry out
- The MUX output

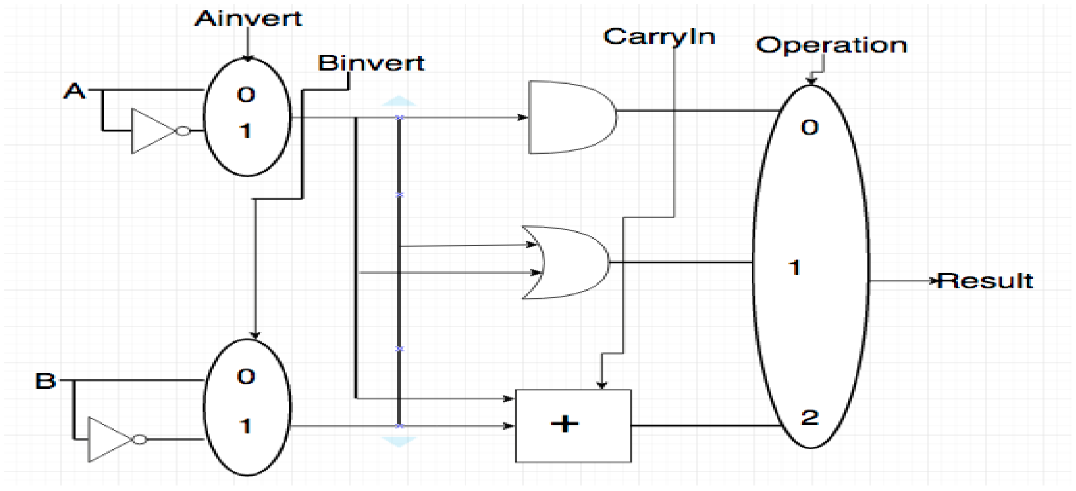
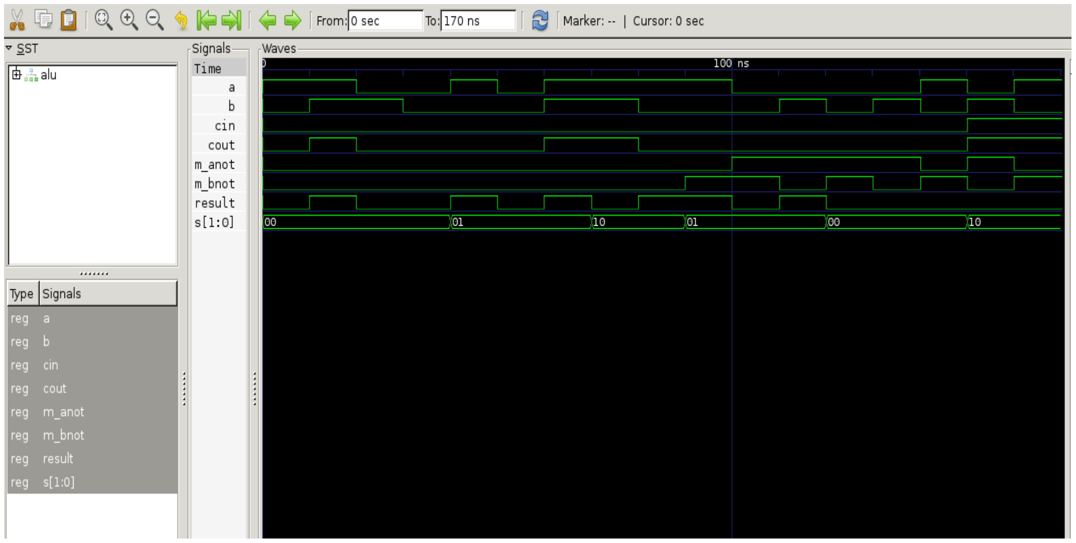


Figure 3: This is a logic diagram of a 1-bit ALU

4.2.1 Experiment 2 Conclusions

I only followed the example from the book although the example from the book was bit confusing. At first, I tried to create separate testbenches for the separate behaviors. For instance, I create a test bench for a and b inverts. Then I instantiate them into the 1 bit ALU-No Luck. Second, I created bit ALU from the book, then I added that with the full adder. When I tried to merge bit ALU and the fulladder with the actual 1-bit ALU, thing got messy- No Luck Yet. Lastly, decided to follow the figure from the book. First I started building smaller parts and then build upon it. Finally, I created 1-bit ALU as you wanted.

I have 165 lines in my testbench that is where I tested my inputs. However, I haven't tested all inputs because they were so many and it is impossible to test all inputs. I only tested possible cases. I think I need at least 30 to 40 more lines to for full test coverage. I think I need n*4 lines to fully test 4-bit ALU because it will take 4 bit input and there are sixteen possible cases for each test case.



Type	Signals
reg	a
reg	b
reg	cin
reg	cout
reg	m_anot
reg	m_bnot
reg	result
reg	s[1:0]

Figure 4: This is gtkwave of 1 Bit ALU

4.3 Experiment 3: 4-bit ALU

In this experiment, we are creating a Nibble ALU. For this experiment you get to build on your ALU from before. Tie together 4 1-bit ALUs to create a Nibble ALU. To make sure that everything is tied together properly, perform some spot testing with a testbench (you don't need to exhaustively test everything (unless you want to!), do a subset to prove to yourself that it's working).

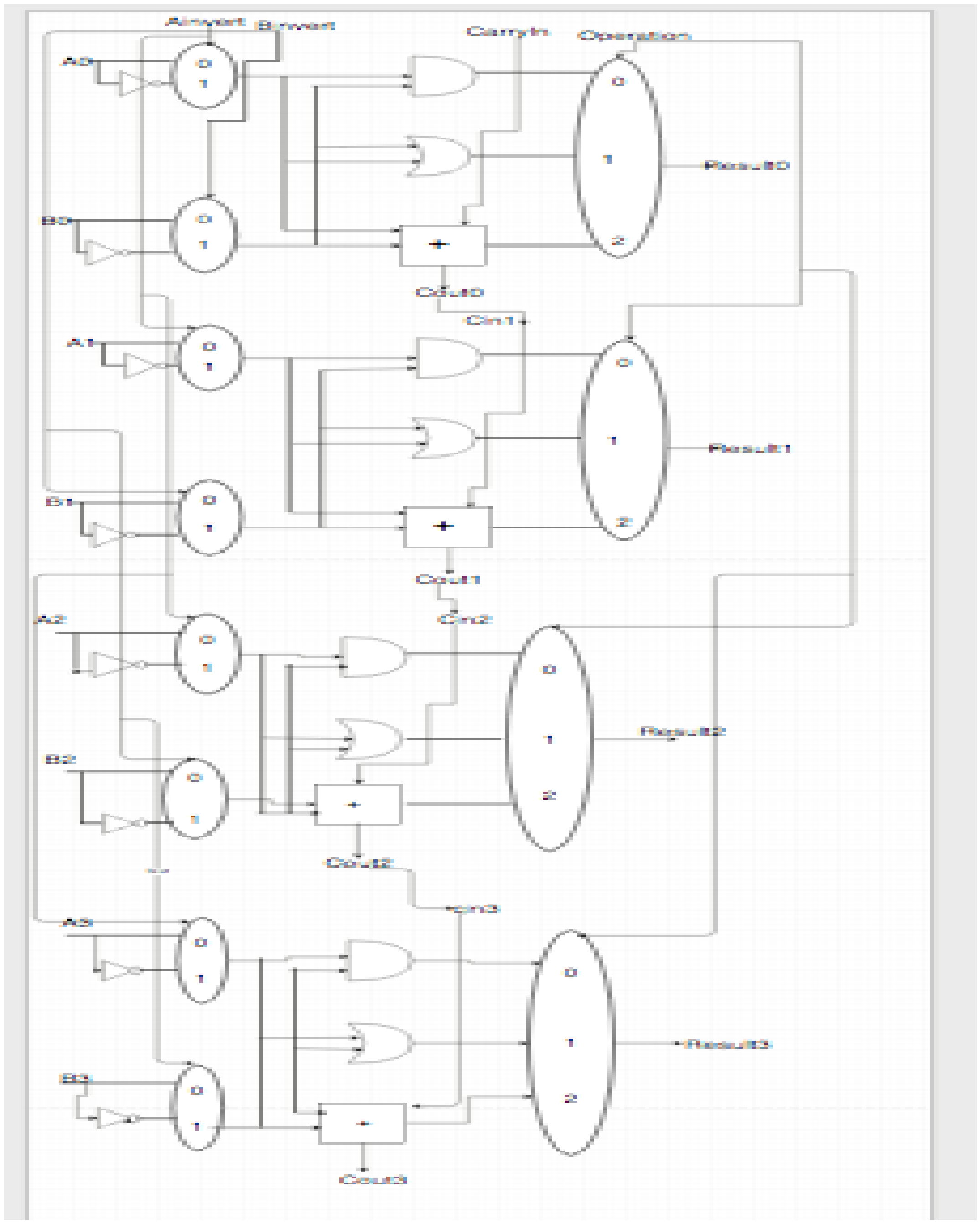


Figure 5: This is a logic diagram of a Nibble ALU

4.3.1 Experiment 3 Conclusions

NO, I have not tested all input combinations. I only tested a subset of it. I randomly picked 4-bit numbers as the inputs for a and b. I tested them where a and b were same and opposite. The most challenging aspect of hooking up the ALU was when I instantiate them in the entity. Coming on to the testbench, the most challenging part was to decide when to invert a or b. I was very confused. I tested one case at a time out of 6 (and, or, nor, nand, add, and subtract). I followed the truth table to double check my result and yet it was still confusing. Yes, timing is becoming a problem now. If I give less time, it will mess up our result or else if I have more time, it will take forever to process. When I created a .vcd file, it was taking forever to finish the process therefore I have to force the stop time. I used following statement: Ghdl -r executablename –stop-time=125ns –vcd=name.vcd. Note: I am sorry about the blurry logic diagram. It was hard to fit 4-bit ALU on a page so...

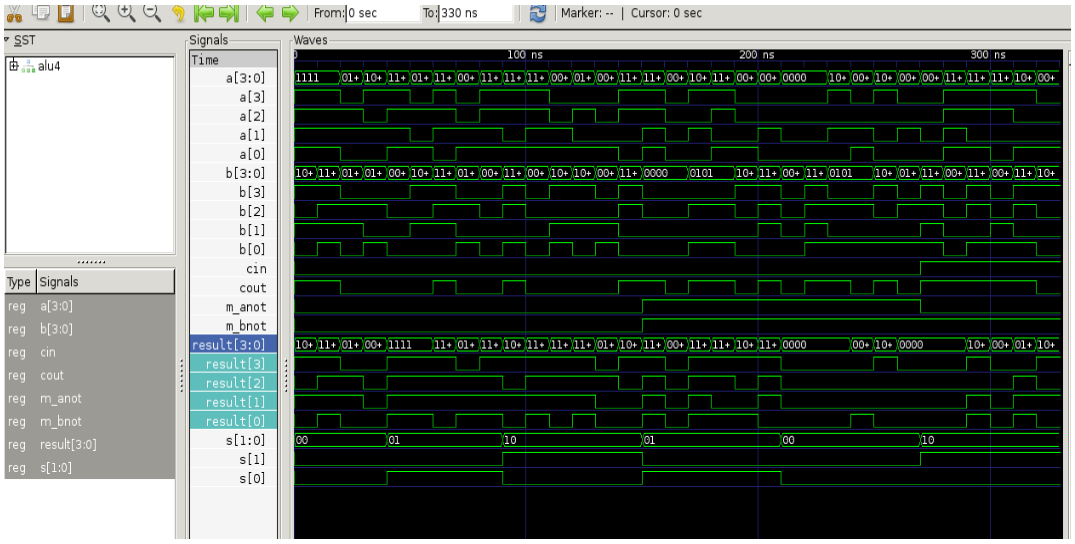


Figure 6: This is gtkwave of 4 Bit ALU

4.4 Experiment 4: D Flip Flop

After all of the work building up the ALU, it's time to start working towards a Register File. The most basic building block of this is our D Flip-Flop. Of course, you will most likely wish to start with a D Latch as a building block for the flip-flop, and possibly even using an SR Latch as a building block for that. Feel free to use the SRLatch code I have provided in this lab as a basic building block (you can also just jump right in if you'd prefer). Make sure to create a falling edge flip-flop (just to make sure we are all on the same page later on).

After creating your D Flip-Flop, make sure to fully testbench all inputs. Remember, this is a sequential circuit. That means you have to (for example) test the case where D is 0, Clock is 0, and Q was 1, as well as the case where D is 0, Clock is 0, and Q was 0.

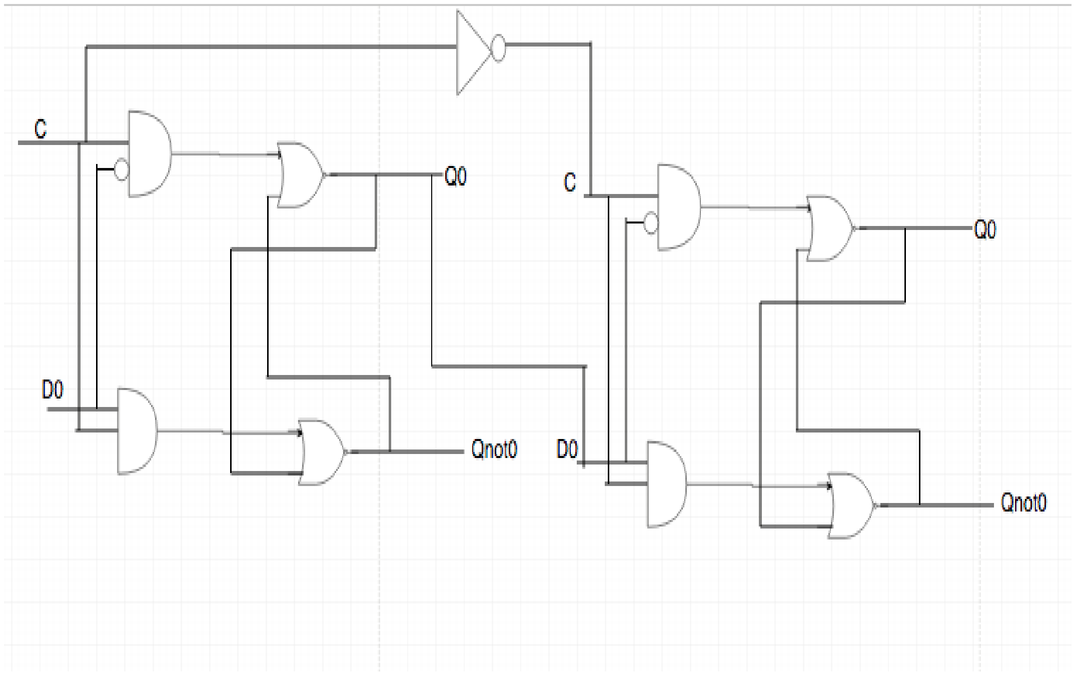


Figure 7: This is a logic diagram of a D Flip Flop

4.4.1 Experiment 4 Conclusion

First I created SRLatch and instantiate it into my DLatch. Then, combined two Dlatches in order to create a D Flip Flop. However, for some reason my q was undefined and I was unable to find what is wrong with my q. Therefore I chose a different route. I used clock'event for the falling edge triggered. Inside the process, it will wait until the clock event and the clock equals to zero to set the value of q. When the clock and clock even are 0 then only the falling edge triggered and set q equals to d. When the execution reaches the wait, it stops until the condition is true. I already said what difficulties I face implementing D Flip Flop

4.5 Experiment 5: Nibble Register

For this experiment, you will be using your D Flip-Flop to create a Nibble Register. Your Register will take in 4 bits and a clock signal, and will output 4 bits. Make sure all data is written in parallel (on the falling edge). To make sure that everything is tied together properly, perform some spot testing with a testbench (you don't need to exhaustively test everything (unless you want to!), do a subset to prove to yourself that it's working).

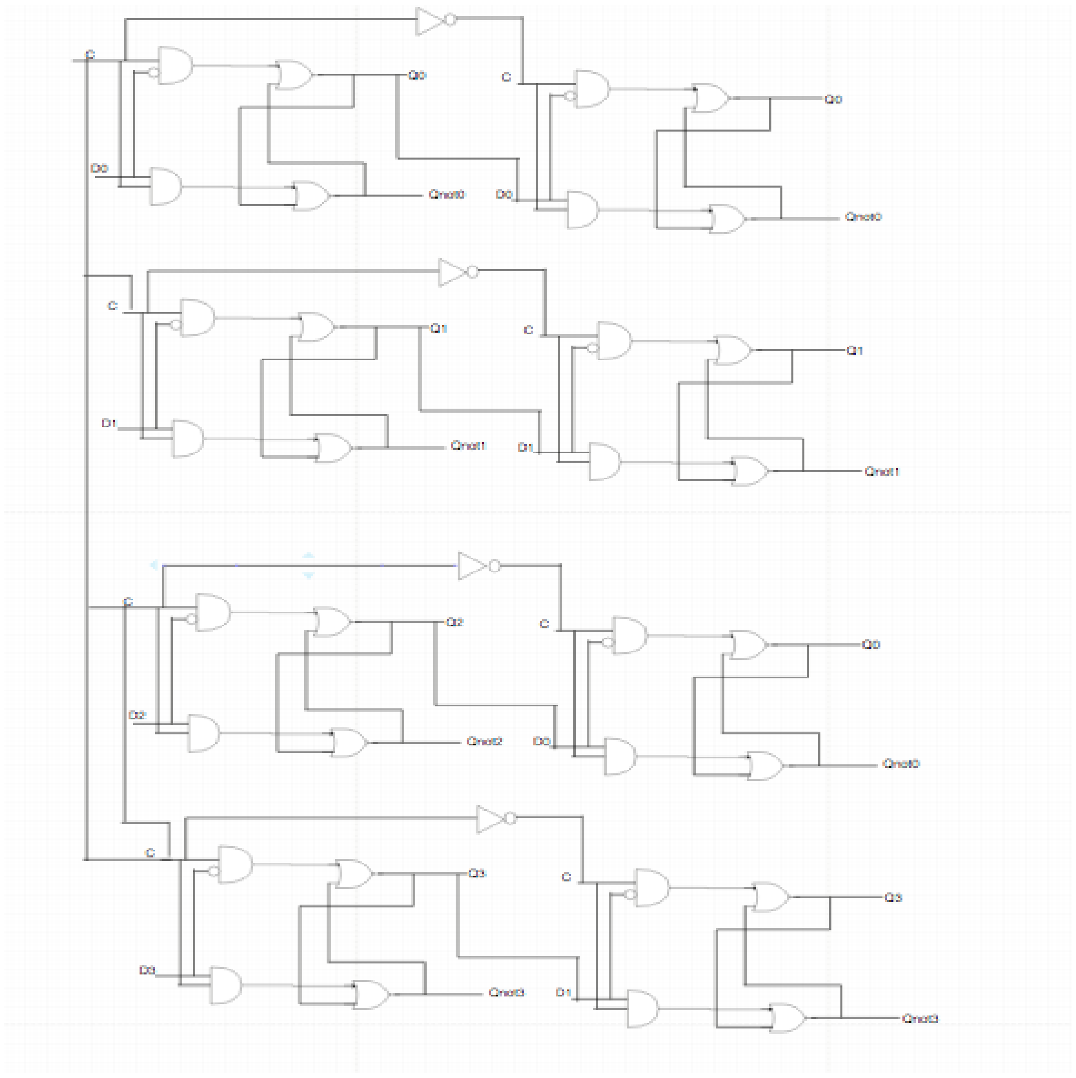


Figure 8: This is a logic diagram of a Nibble Register

4.5.1 Experiment 5 Conclusion

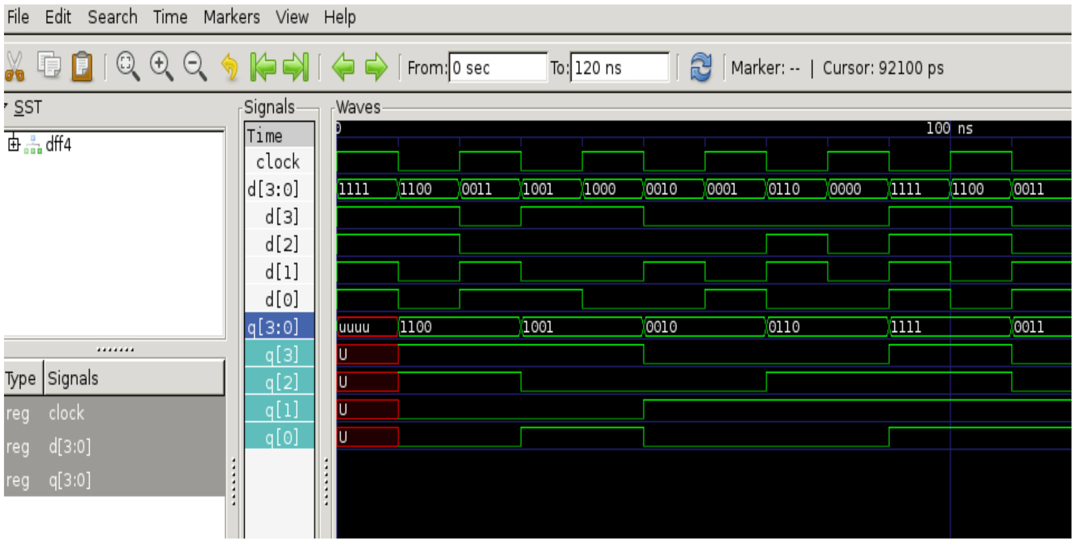


Figure 9: This is gtkwave of a Nibble Register

NO, I have not tested all input combinations. I only tested a subset of it. I randomly picked 4-bit numbers as the inputs for d. I tested thirteen different inputs twice. Overall, I tested twenty-six input combinations. After building nibble ALU, this was easier. However, I was freaking out when I saw my first q is undefined and the rest are defined. After talking to you, it makes sense why the first q is undefined. Yes, I am very excited for a full register file. I am enjoying coding in vhdl and creating gtkwaves. It is so much fun.

5 Conclusion

In short, first we created 2 bit selector. Second, we created 1 bit ALU using our selector and a full adder. Third, we created a nibble ALU using out 1-bit ALU. Fourth, we created our first sequential circuit-D Flip Flop. Last, we created a nibble register using our D Flip Flop. Now, I am looking forward to create a full register file.

5.1 Challenges

The most challenging part in this lab was finding what went wrong with the 1-bit ALU output. I was getting wrong output with AND and NAND gate when I tested with a='1" and b='0'. The problem was with my mux. After fixing it, all my outputs were correct.