# Lab Report:
# Building our Base

Nisha Patel

October 21, 2016

**Abstract**

***In this report, I am discussing about components needed to create a simple 4-bit ALU. We are creating a decoder, half adder, full adder, ripple-carry adder and a 4-bit shifter.***

## 1 Introduction

For this lab, we are developing our components needed to create a simple 4-bit ALU. You 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

GHDL is a Hardware Description Language (the G currently stands for nothing). It follows the IEEE 1076 VHDL standard, so will be compatible with almost everything you can find out in the wild. As this is a free implementation, it is what we will be sticking with for the rest of the semester. We will be working with 2 basic GHDL commands:
ghdl -a *.vhdl.
The above will create an analysis of a design file (in VHDL terms). You can then make an executable file with the elaborate option:
(ghdl -e executable_name) You can then run your executable with the -r option, or (on linux), you should have an executable of executableName sitting in your current directory. (i.e, ghdl -r executable_name –vcd=filename.vcd).

To test the gtkwave you have an X session going (sit at the machine, or set up X forwading) -X is needed on the client side.
(ssh -X 10.30.1.101)

## 3 Experimental Setup

**Parts List**

- a decoder (3 inputs)
- 3 full adders and a half adder
- 4-bit shifter

# 4 Experiments

We are going to work with the components we?ve built in class to test and eventually run a simple ALU. We do not have working memory yet, so we are going to be working with immediate values, and recording the output (can not store it anywhere either).

## 4.1 Experiment 1: Building your decoder

In the first experiment, we will be creating 3-bit decoder that takes 3 inputs and gives 8 outputs. We are using NOT and AND gates.

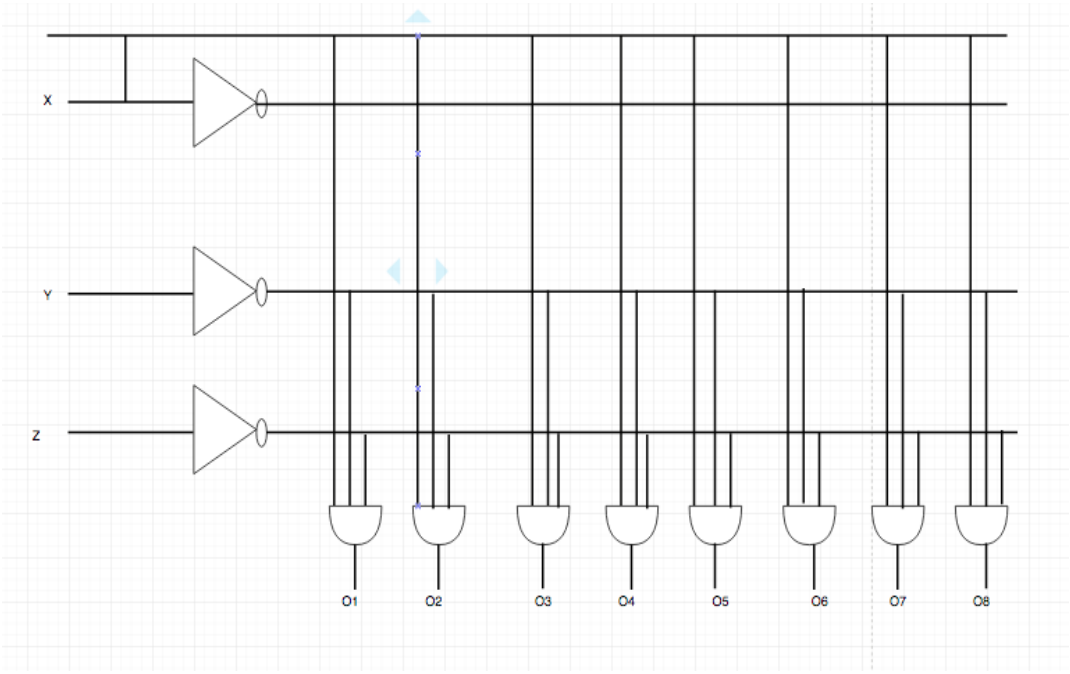


Figure 1: This is a logic gate of a 3-bit decoder

### 4.1.1 Experiment 1 Conclusions

In class, we worked a 2-bit decoder that takes 2 inputs and gives 4 outputs. In this experiment, the only difference is that it takes 3 inputs and gives 8 outputs. The procedure is still the same no matter how many bits you increase. I used NOT and AND gates to build my decoder. I used 3 NOT gates and 8 AND gates for 3-bit decoder. I think building a decoder using NOT and AND gates is very hard or most likely not possible because we will be using so many wires, transistors and resistors. I really do not think it is a good idea to build the decoder on the breadboard unless we use those ICs.

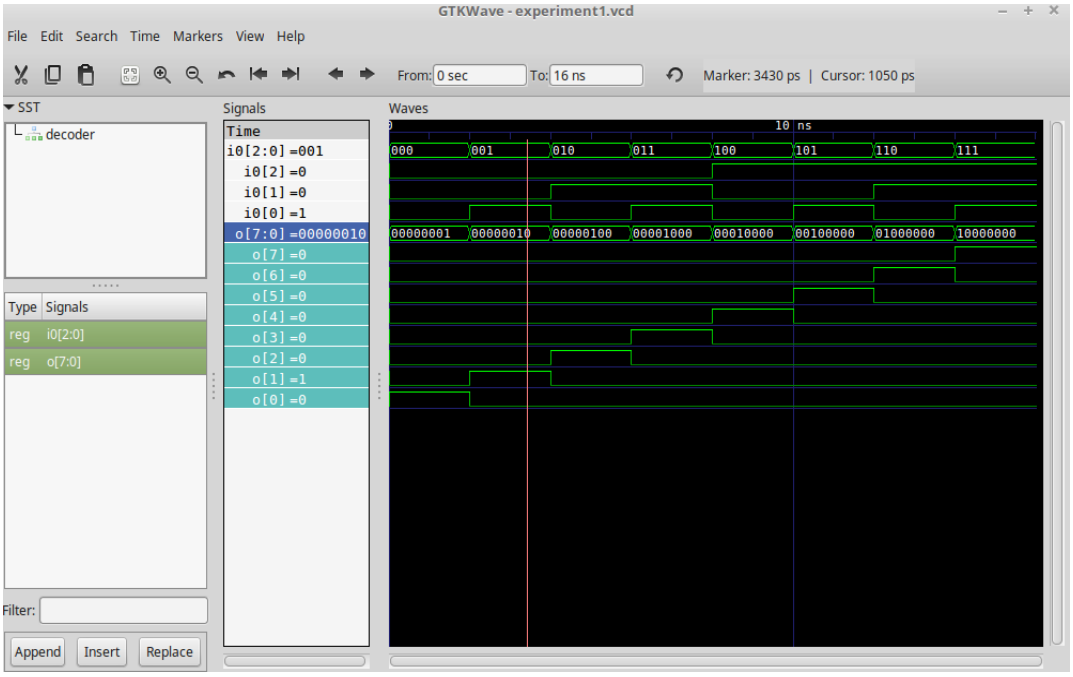


Figure 2: This is a gtkwave of a 3-bit decoder

## 4.2 Experiment 2 Individual Adders

In this experiment, we are creating a half adder and a full adder using logic gates in VHDL. In the previous lab, we have seen the difficulties of creating these adders. Creating the adders using VHDL is very simple and easy.

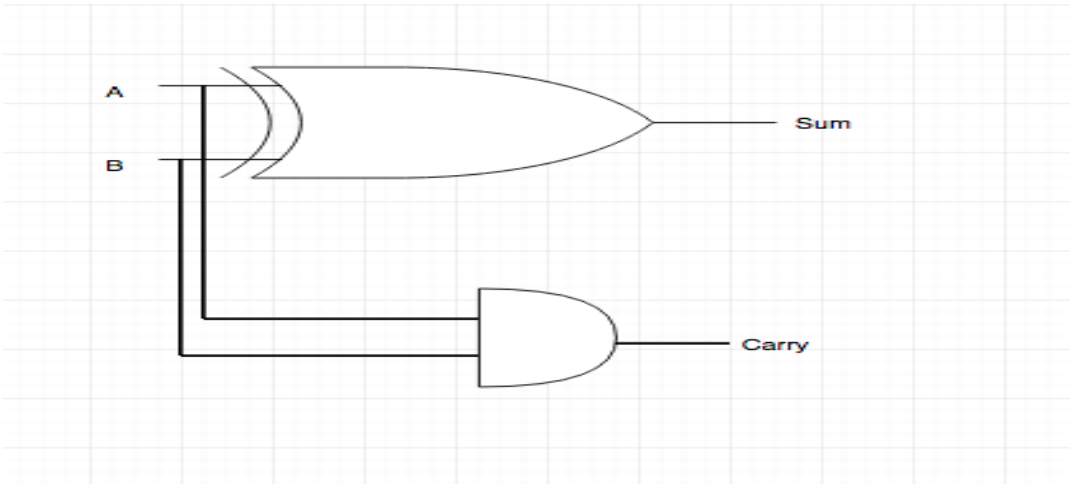


Figure 3: This is a logic diagram of a half adder

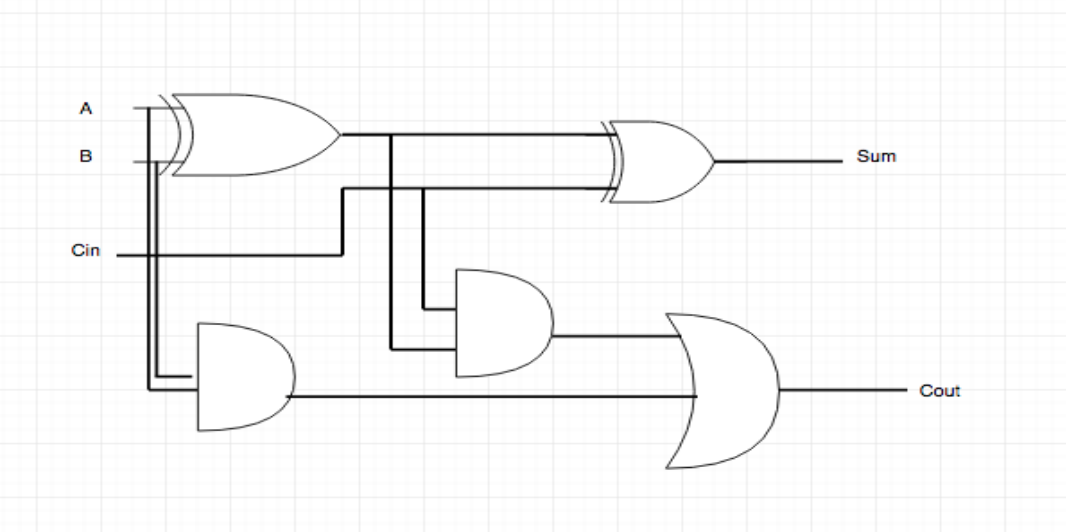


Figure 4: This is a logic diagram of a full adder

### 4.2.1 Experiment 2 Conclusions

We created half adder and full adder for this experiment. I used a XOR and a AND gates for the half adder. In addition, I used two XOR, two AND and a OR gates for the full adder. I build my full adder from the half adder. I added two half adders and OR the result together in order to create a full adder. I choose this particular approach because it seems easier to implement in VHDL code and it looks cleaner. It is way easier to create these components in VHDL vs on a breadboard. I think breadboard gets messier as you build more complex circuits on it and it is hard to follow what is going on your breadboard. For instance, in the previous lab we build half adder using six transistors which was a huge mess. On the other hand, if we look at the half adder in VHDL code, it looks simple and neat. Personally speaking, I am enjoying VHDL.

I found a better example to create the adders using logic vectors and I used it. By using logic vectors, the code looks more simpler and shorter. s

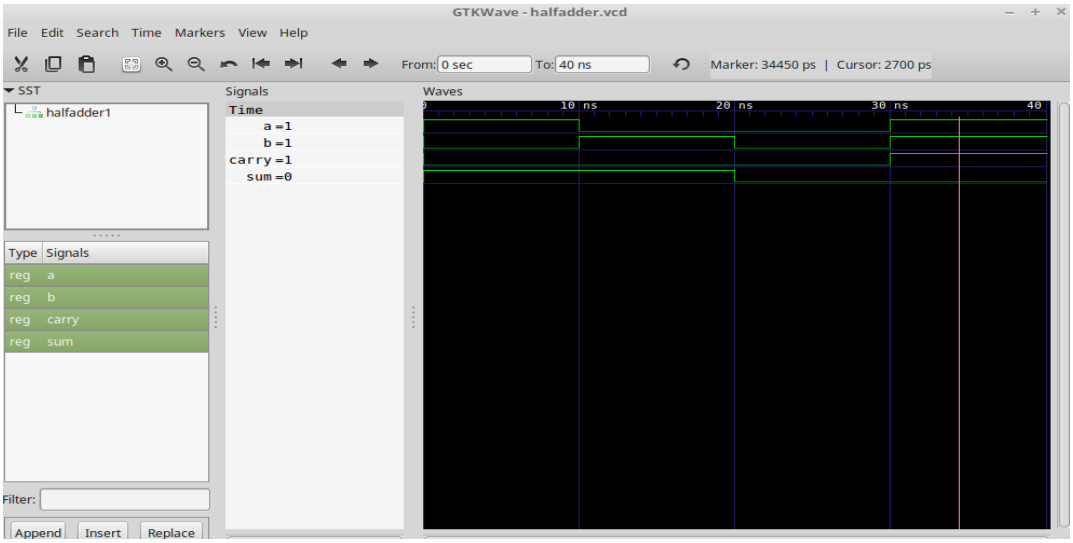


Figure 5: This is gtkwave of the half adder in VHDL code

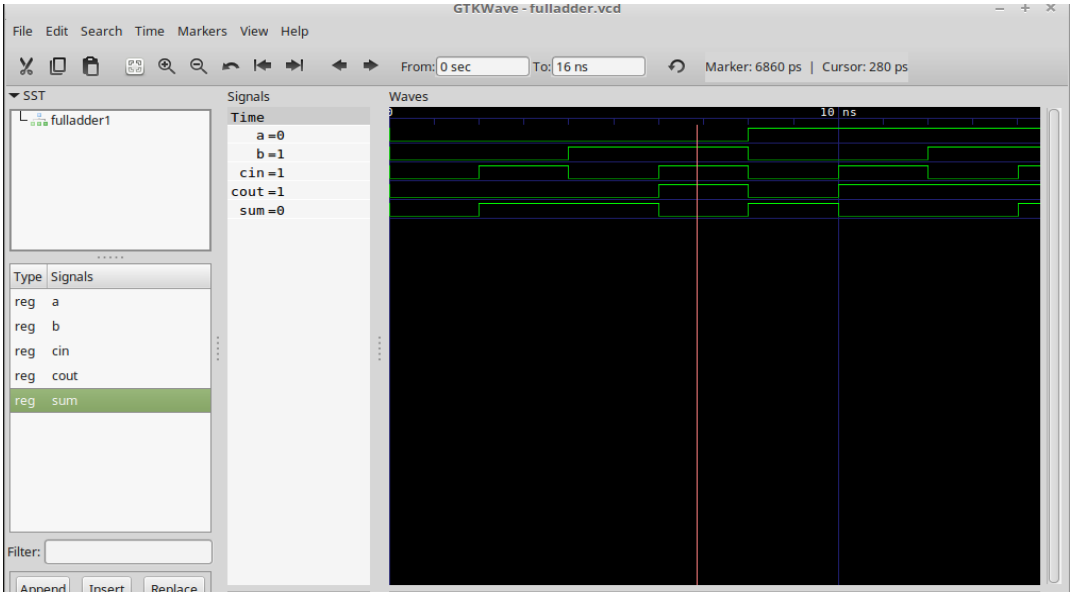


Figure 6: This is gtkwave of the full adder in VHDL code

## 4.3 Experiment 3 Ripple-carry Adder

In this experiment, we are creating a nibble adder. For this experiment we get to build on our adders from before. Tie together a half adder with 3 full adders - this will allow us to add together a full nibble.

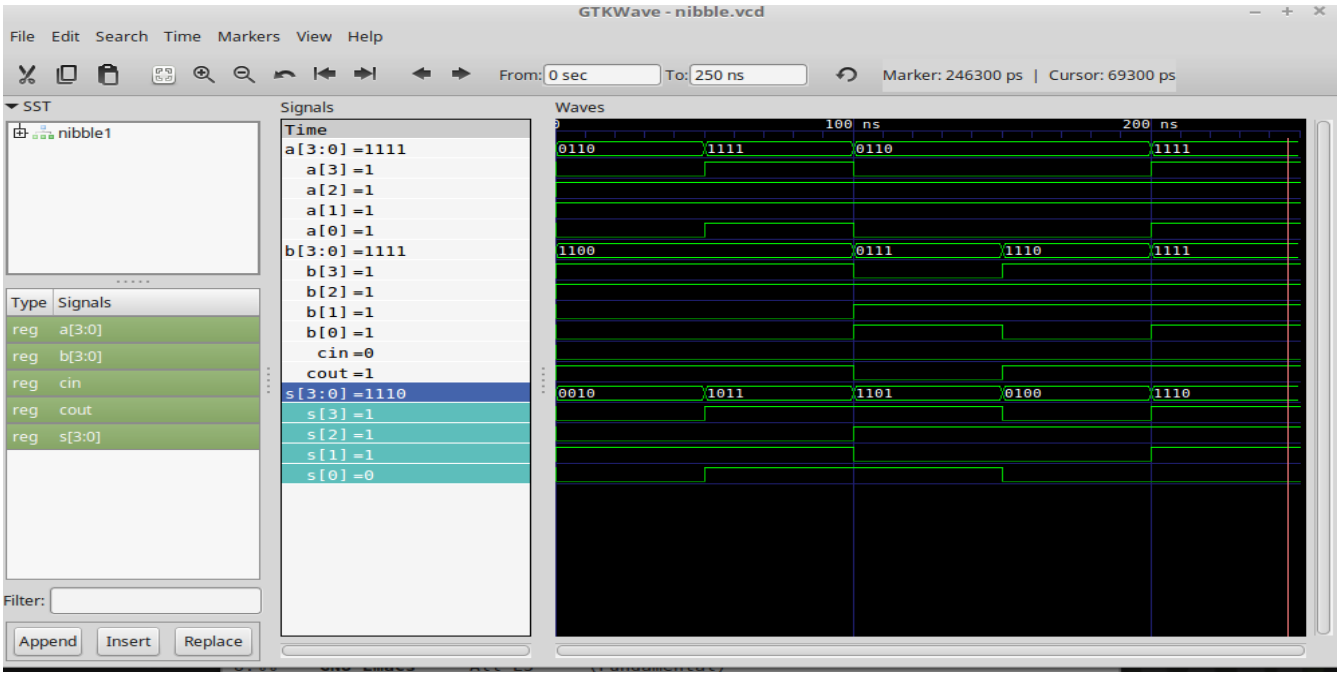


Figure 7: This is a gtkwave of the nibble adder in VHDL code

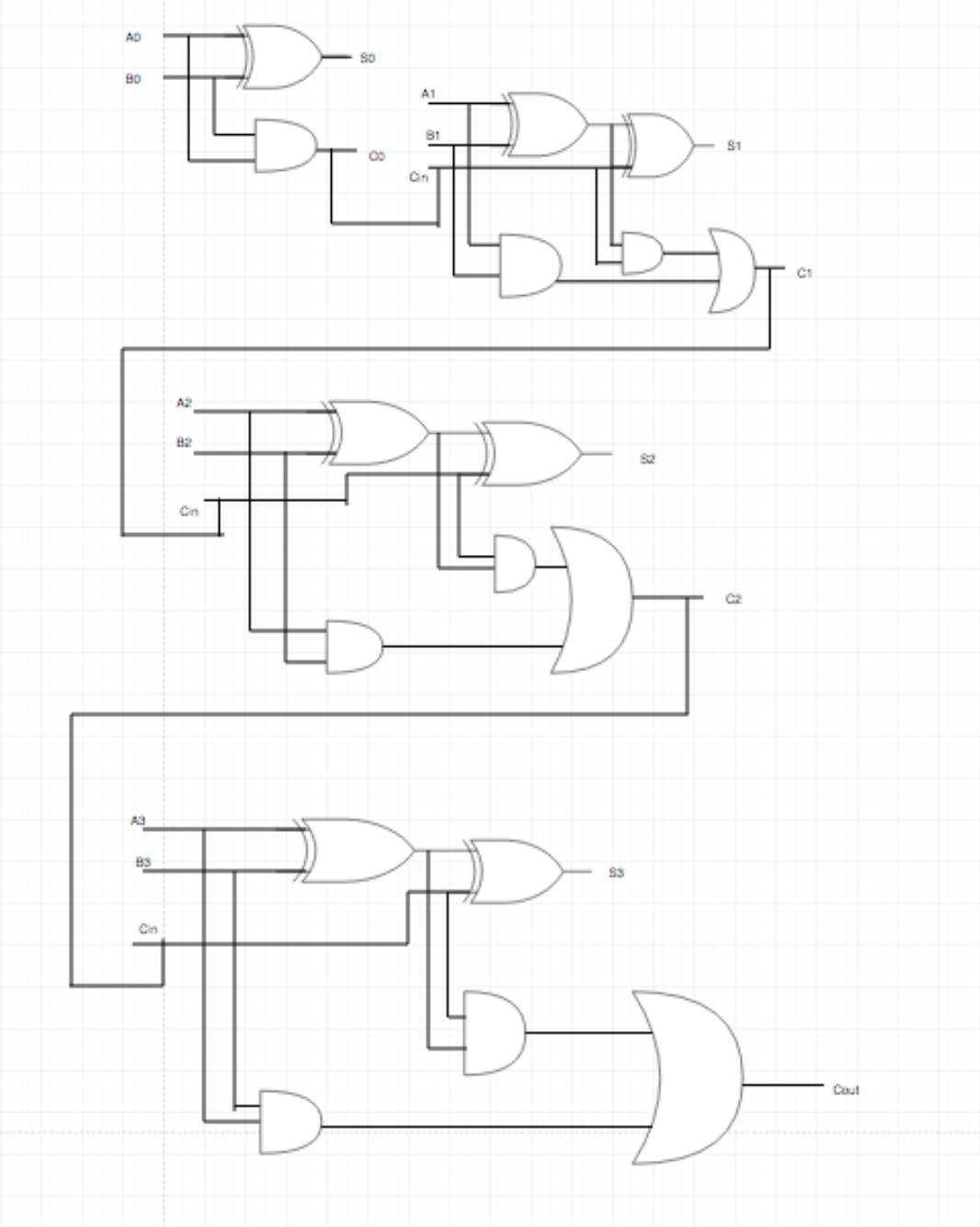


Figure 8: This is a logic diagram of a nibble adder

#### 4.3.1 Experiment 3 Conclusions

I chose std_logic_vector approach for this experiment. No I am not getting sick of creating test benches yet. In fact, I really like creating test bench by using standard logic vector approach.

I used instantiation and do port map for the adders. For example, HA : entity work.halfadder port map(a =>a(0), b =>b(0), sum =>s(0),cout =>c1); Here is where we use previous half adder where we port map the component of the half adder. Furthermore, **work** is the current working directory where all your projects

files will be complied and a **halfdder** is the component name as defined in the second experiment.

Originally, the wait time I have is 50 ns and I tested with 5ns, 200ns, and 500ns, but nothing changes. It is weird that nothing is changing. The only thing I can think of is if I decrease the time (very less time), then I the end time will change. I tracked down the end times for different ns and here is what I found.
For instance: end time for 1 ns is 500000
end time for 50 ns is 250000000
end time for 200 ns is 1000000000

If I want to create a byte nibble adder, I would used a half adders and seven full adders. For 64-bit adder, I would used a half adder and 63 full adders (Oh my god, so may adders).The simple logic behind adding the bits is as you add a bit, you add a full adder. I think adding full adders for the long run is a very bad idea. We can make it easier by using four 16-bit LCU adders to make it work as 64-bit adder by only using four adders. Otherwise, we can also use two 32-bit ALU adders. I think either option will make it easier and simpler.

## 4.4 Experiment 4: Bit Shifter

In this experiment, we are creating a 4-bit shifter that we discussed in class. It takes four bit input plus s that can be either 0 or 1 and four bit output. The bit shifter use NOT, AND ,and OR gates.

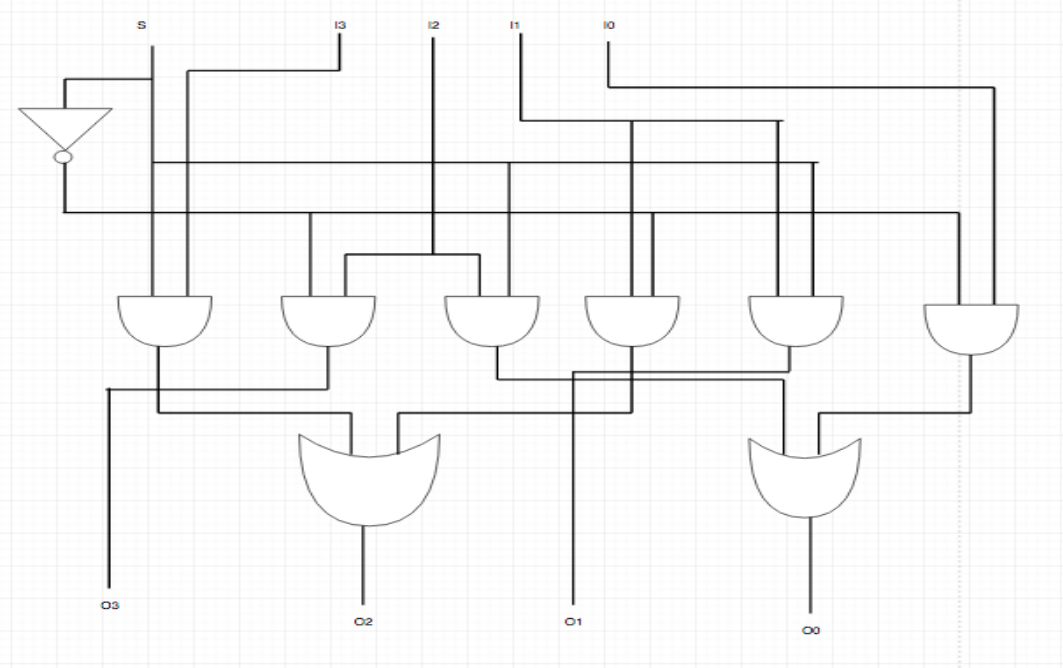


Figure 9: This is a logic diagram of a 4-bit shifter

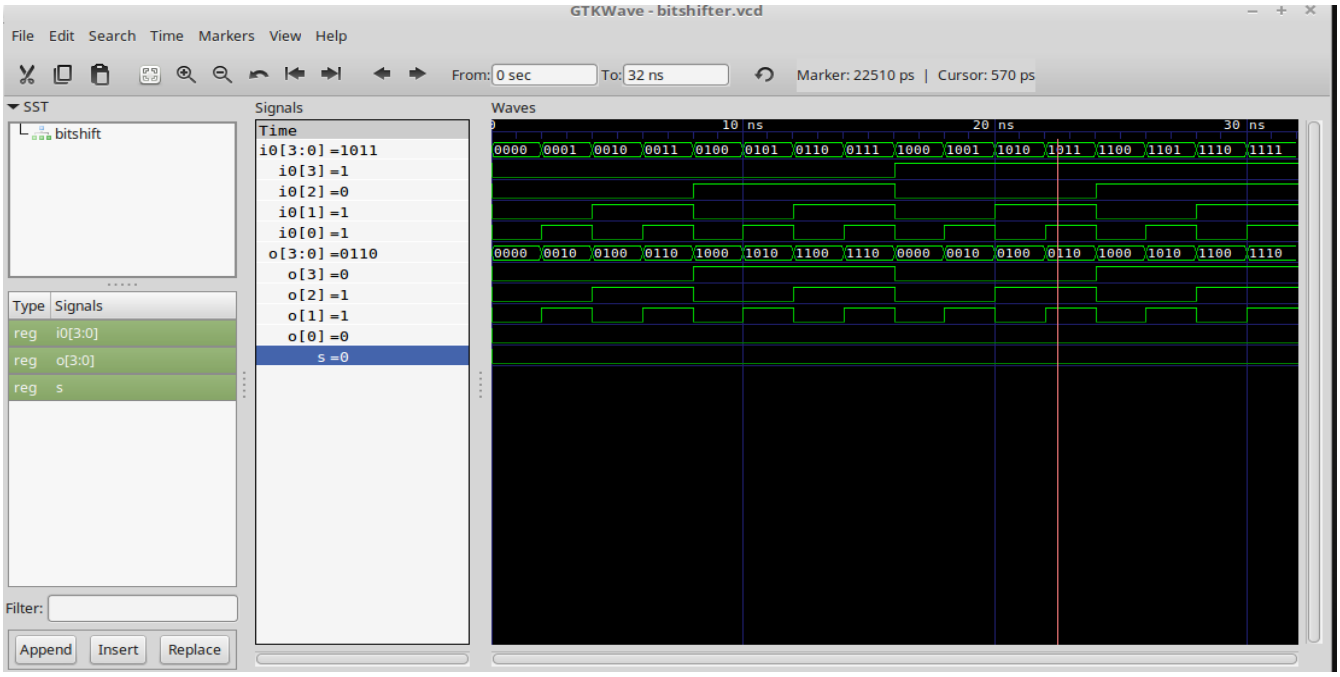


Figure 10: This is a gtkwave of the 4-bit shifter when s =0 in VHDL code

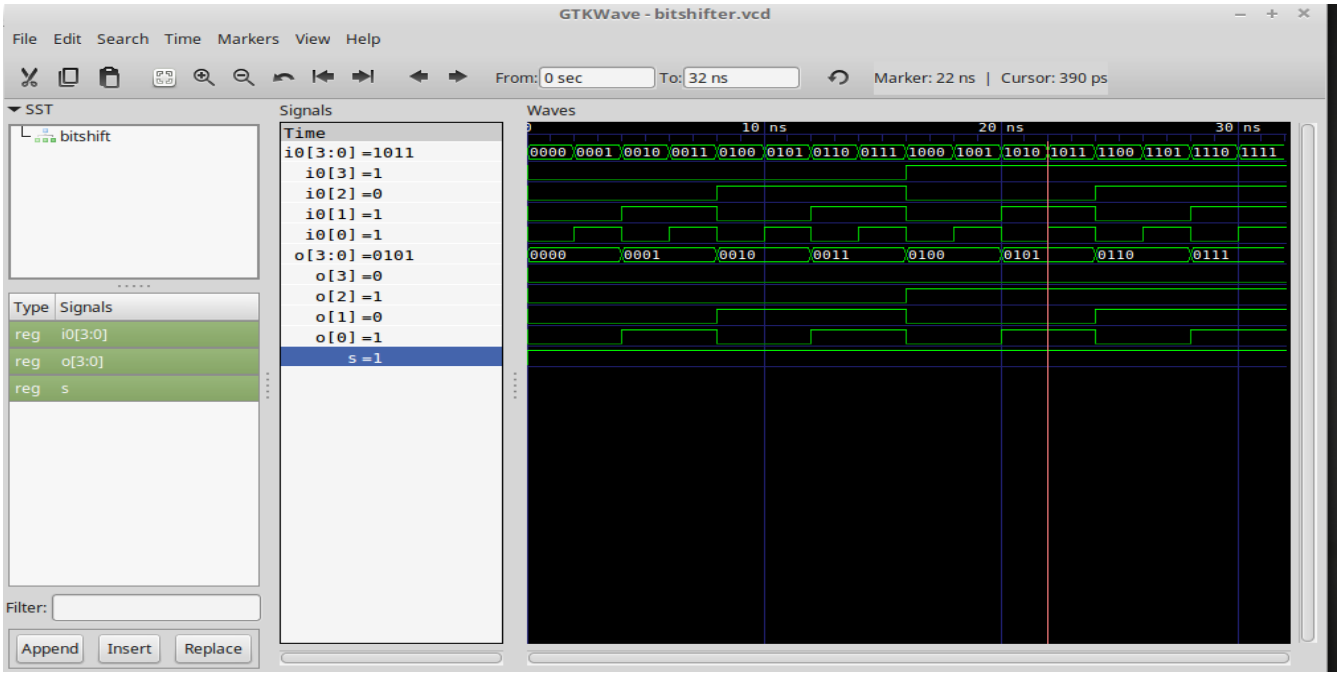


Figure 11: This is a gtkwave of the 4-bit shifter when s =1 in VHDL code

#### 4.4.1 Experiment 4 Conclusion

Yes I can envision tying all these components together to make up ALU. That is why we are creating small components for the ALU. I am not sure about how to modify it to make it a signed shift, but I think we can change the AND gate and make it an OR gate instead. I assume that if we put the input of one and put it as an output for the second one, then I think we can handle multiple shifts. Finally chain them together.

# 5 Conclusion

In short, first we created 3-bit decoder. Second, we created a half adder and a full adder. Third, we created ripple carry adder using previously created half adders and the full adders. Last, we created a 4 -bit shifter. Overall, it was fun building our base in VHDL.

## 5.1 Challenges

The most challenging part for this lab is to create a nibble adder. Second, drawing logic gates in dia was terrible, so I used draw.io which was way easier and simpler than dia. It took me only an hour to draw all the logic diagrams. Last, using X-forwading from my computer was bit sketchy