California State University, Northridge
ECE 524L - Advanced FPGA Design
Fall 2022

Lab 2: RAM and Buffer of FPGA

September 12,2022
Instructor: Saba Janamian

Troy Israel

Table of Contents

Introduction

The purpose of this lab was to familiarize myself with designing RAM and ROM in FPGA using VHDL while also learning how to implement a Lookup Table (LUT) in VHDL. Performing these tasks shows us that we can infer memory structures inside the FPGA. Essentially, this lab teaches us to create small to intermediate scale memory for certain types of applications. We will also be designing a First In First Out (FIFO) data transfer in this lab.

Procedure

Part 1:

In Part 1 of this lab, I am going to design a ROM based signed-magnitude adder. This adder is going to take 2 inputs and each of the 2 inputs will be 8 bits and later on 16 bits. Using python code that will create 256 entries for the 8 bit input and over 65,000 entries for the 16 bit. These entries are for a dataset that will be used in designing the Lookup Table (LUT). The LUT will be implemented by reading in a text file, (two text files, 1 for the 8 bit and the other for 16 bit). In VHDL, I will use the textio library to read in the data generated by the python code. In the python code, it will handle sign bits as well as handling the magnitude portion of each data point.

In the test bench, I will create a simulation that will read from either LUT and will output a stream of data. It will show the result of each data element as either 8 bit value or 16 bit value. In this case each of the outputs will be an actual address that will be taken from the LUT.

Part 2:

In Part 2 of this lab, I will be designing a ROM based temperature conversion. It will implement two different LUTs. One LUT will contain Fahrenheit to Celisus values and the other LUT will contain the opposite conversion. Similar to part 1, this portion of the lab will also do the same file reading. The two temperature LUTs will be made from python code. Inside the python code, it will perform calculations and write the results into text files. Using VHDL, it will read those two text files which will be the LUTs. There will be a special signal made that when set to either 0 or 1 it will make the VHDL code read from a specific LUT. This signal will come more into play in the testbench rather than the actual top module. But the top module will be set up so that based around that specific bit, it will give access to the right LUT.

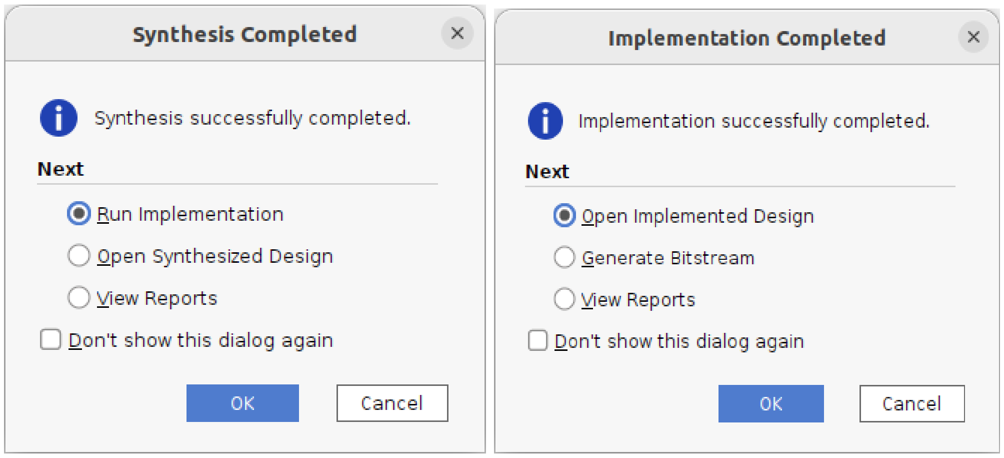
Part 5:

In part 5, I will create a FIFO buffer with extended status. Given to us was three templates, a FIFO top module, a FIFO controller and register file. From the templates we have two statuses, empty and full. In this lab, I need to add three more statuses. First a word counter which will track how many words are in the FIFO. Initially this should be set 0. The next two are extensions of the empty and full processes that were given in the template. One of the extra statuses is a checker called almost empty which is supposed to let the user know that FIFO is almost empty. Let the user know around 25% or less. And the other one is the opposite, almost full. Let the user know that at 75% the FIFO is close to filling up. The test bench will further implement the actual processes, whereas the control file will set up the logic for the three extra statuses.

Test Strategy

Part 1

For Part 1, the majority of the code was provided to us as a template. The primary thing that I needed to set up by myself was the text files from canvas. Since I am using a linux machine, I needed the command line to transfer the text files from the Downloads folder to the Project directory. I specifically placed them into a data folder just above the source files. Once I made sure the file reading in the VHDL code had the correct directory address for the text files, I could then synthesize the project and later use the simulation test bench. When doing part 1, I also need to run the synthesize tool twice because of the fact that this part asked for two variations, one for 8-bit and the other for 16-bit and I used the generic statement in VHDL to make for easier changes.


Synthesis Completed

Synthesis successfully completed.

Next

[✓ Run Implementation]
Open Synthesized Design
View Reports

Don't show this dialog again

Implementation Completed

Implementation successfully completed.

Next

[✓ Open Implemented Design]
Generate Bitstream
View Reports

Don't show this dialog again

Fig 2.1 For both 8-bit and 16-bit Designs were both synthesizable and passed the implementation test.

Part 2

For part 2, similar to part 1, the parts of the code were given as templates. Since this part of the lab also implemented LUTs to do the temperature conversion, I needed to make sure that the text files generated from the python code would be included in the data file inside the project. One of the differences between this part and the previous part of this lab was that I needed to make a signal that would check if the data coming in was in Fahrenheit or in Celsius. So if set to 0 the data stream is celsius and if 1 the data stream is fahrenheit.

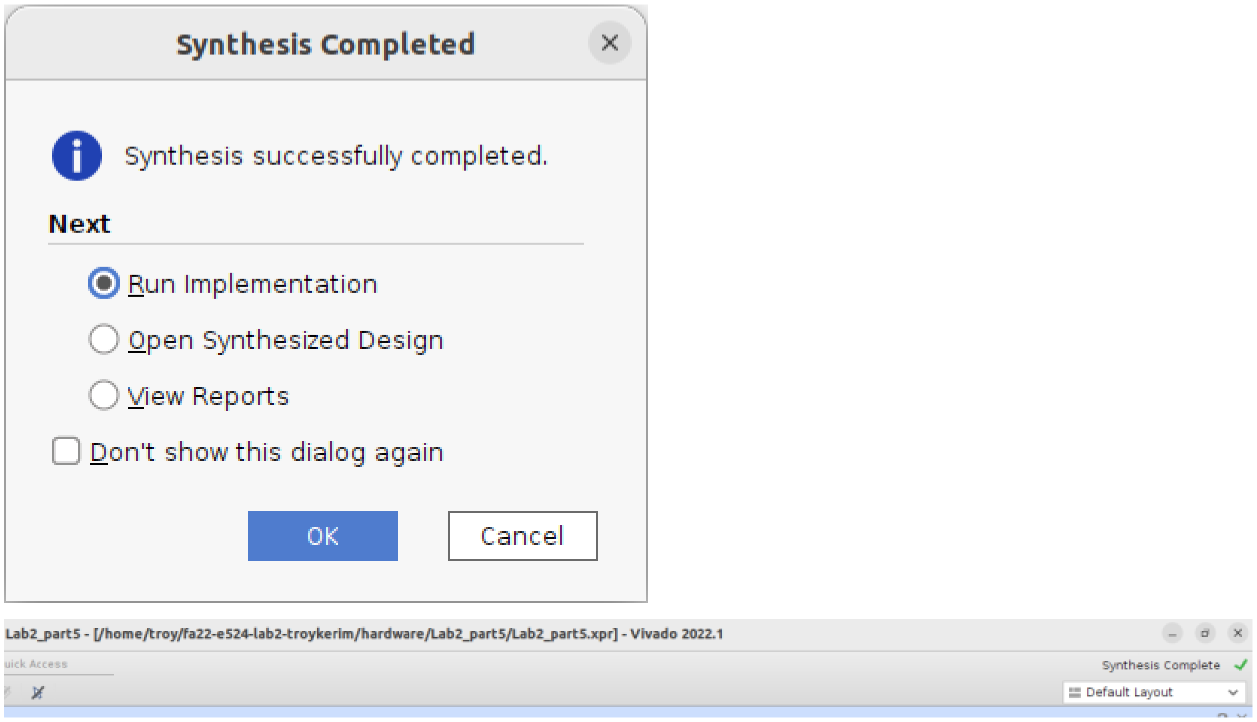
Part 5:

This part of the lab I realized that my approach to this part was greatly overestimated. I was overthinking how to design the FIFO with extended status. This occurred before we went to this part in lab class. I realized that I was making too many signals and ports. I did get the three ports we needed but instead I kept making more ports thinking that I needed them.

However, there were some parts of this design that was shown in lab class that I wouldn't have known how to do, like creating and implementing a function to handle the issue of where the write pointer and read pointer would be located. As shown in the lab 2 slides, the FIFO operates in a circular pattern, 8 bits for our lab, but at some point if you keep writing the tail can be ahead of the head pointer. To fix this problem we simply needed to create a constant that is the same size as the address. This part again was shown to me in class and I did not know how to create it.

I had also had problems with creating the actual test bench. While I knew to create a constant for the clock pulse and make its own process block and at half the clock pulse give it 1 and then 0. I did not know how to handle reset all that well. But after seeing it in class I now have a better understanding of how to set up reset for the simulation as well as understanding that all our modules have a reset and it's important to be aware of that. I tend to overlook that point. I had a hard time with the data setup and when to read and write. For whatever reason I over thought this part of the test bench. I didn't consider the fact that I needed to first set wr and rd to 1 whenever I wanted to perform either operation.

After seeing how it was done in class, I relearned how to create a function and how this function works to correct the head pointer and tail pointer to make sure that we get the correct subtraction when we want to determine our word count. I also learned how to utilize constants and subtypes better and how they can be beneficial in the development of the code. It helps with not having to keep redefining types over and over again.


Synthesis Completed

Synthesis successfully completed.

Next

[✓ Run Implementation]
Open Synthesized Design
View Reports

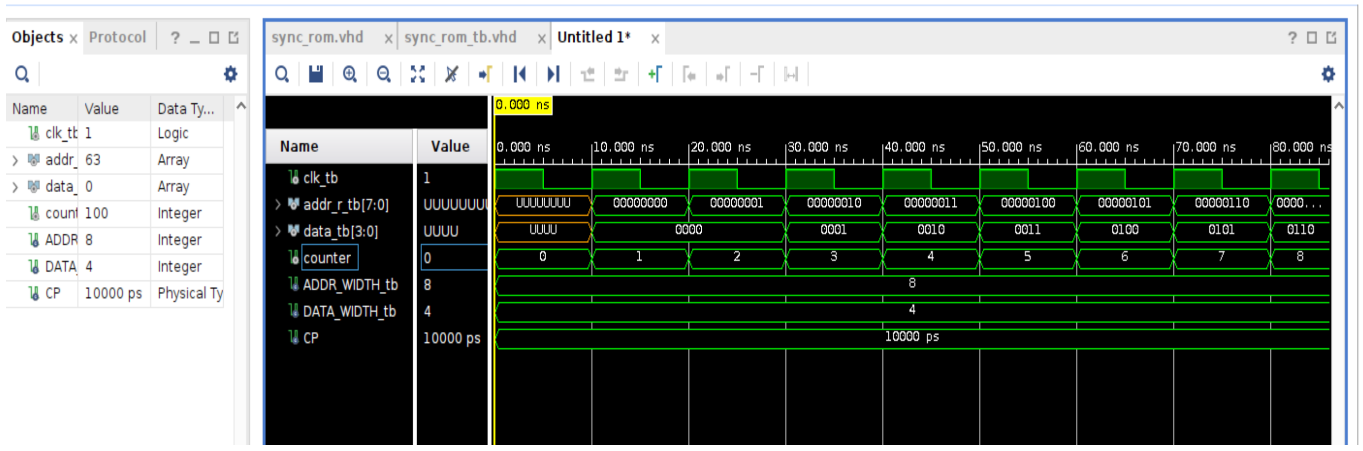
Don't show this dialog again

Lab2_part5 - [/home/troy/fa22-e524-lab2-troykerim/hardware/Lab2_part5/Lab2_part5.xpr] - Vivado 2022.1

Fig 2. Showing that Part 5 was synthesizable

Results

Part 1:



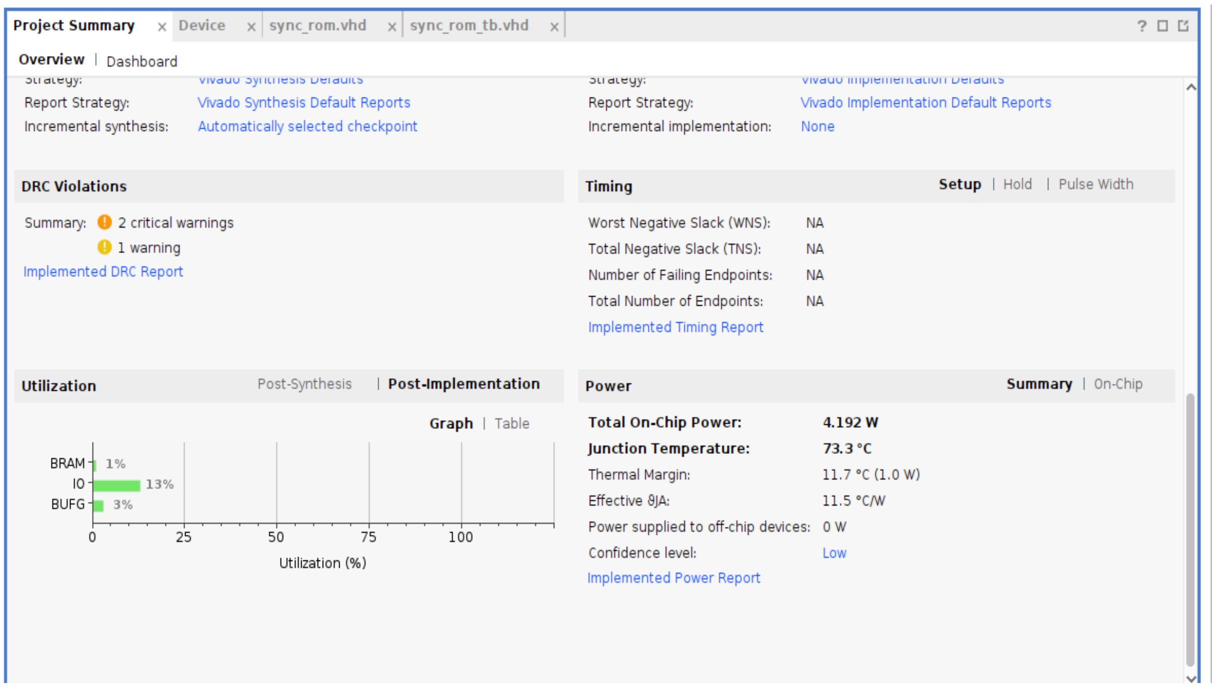
Name	Value	Data Ty...
clk_tb	1	Logic
addr_	63	Array
data_	0	Array
count	100	Integer
ADDR	8	Integer
DATA	4	Integer
CP	10000 ps	Physical Ty

```
0.000 ns
Name           Value      0.000 ns  10.000 ns  20.000 ns  30.000 ns  40.000 ns  50.000 ns  60.000 ns  70.000 ns  80.000 ns
clk_tb         1
addr_r_tb[7:0] UUUUUUUU   UUUUUUUU  00000000   00000001   00000010   00000011   00000100   00000101   00000110   0000...
data_tb[3:0]   UUUU       UUUU      0000                  0001       0010       0011       0100       0101       0110
counter        0          0         1          2          3          4          5          6          7          8
ADDR_WIDTH_tb  8                                                     8
DATA_WIDTH_tb  4                                                     4
CP             10000 ps                                              10000 ps
```

Fig 2.2 Part 1 8-Bit Waveform

The LUT made from the text file technically contains an address and this address you need to interpret in a specific way. To interpret the data, for example the 8 bit values, like 00010001. The first 4 bits, 0001, will be A, and the next 4 will be B. A and B both have an MSB which in this example 0.

Inside the simulation waveform, the result is pushed back 1 clock cycle because of latency. The Data out is giving a 4 bit result of A and B. A and B are combined as 8 bit addresses.



Project Summary | Device | sync_rom.vhd | sync_rom_tb.vhd

Overview | Dashboard

Strategy:	Vivado Synthesis Defaults	Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Synthesis Default Reports	Report Strategy:	Vivado Implementation Default Reports
Incremental synthesis:	Automatically selected checkpoint	Incremental implementation:	None

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing — Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	NA
Total Negative Slack (TNS):	NA
Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA

Implemented Timing Report

Utilization — Post-Synthesis | Post-Implementation — Graph | Table

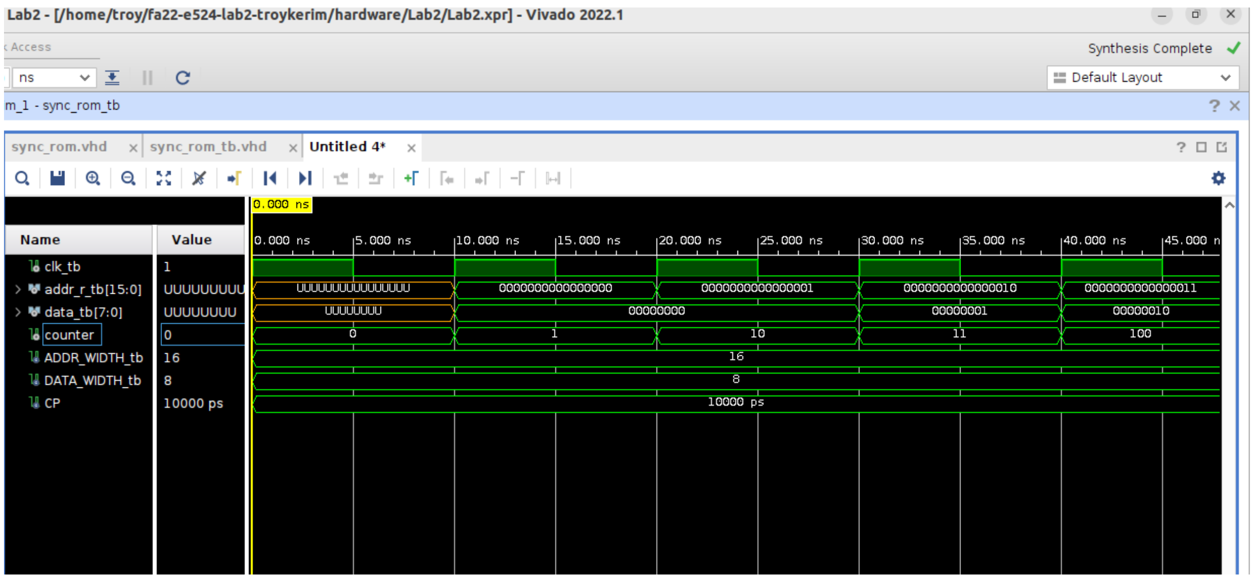
BRAM	1%
IO	13%
BUFG	3%

Power — Summary | On-Chip

Total On-Chip Power:	**4.192 W**
Junction Temperature:	**73.3 °C**
Thermal Margin:	11.7 °C (1.0 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Fig 2.3 Part 1 8-Bit Usage



Lab2 - [/home/troy/fa22-e524-lab2-troykerim/hardware/Lab2/Lab2.xpr] - Vivado 2022.1

Synthesis Complete

m_1 - sync_rom_tb

sync_rom.vhd | sync_rom_tb.vhd | Untitled 4*

Name	Value
clk_tb	1
addr_r_tb[15:0]	UUUUUUUUU
data_tb[7:0]	UUUUUUUU
counter	0
ADDR_WIDTH_tb	16
DATA_WIDTH_tb	8
CP	10000 ps

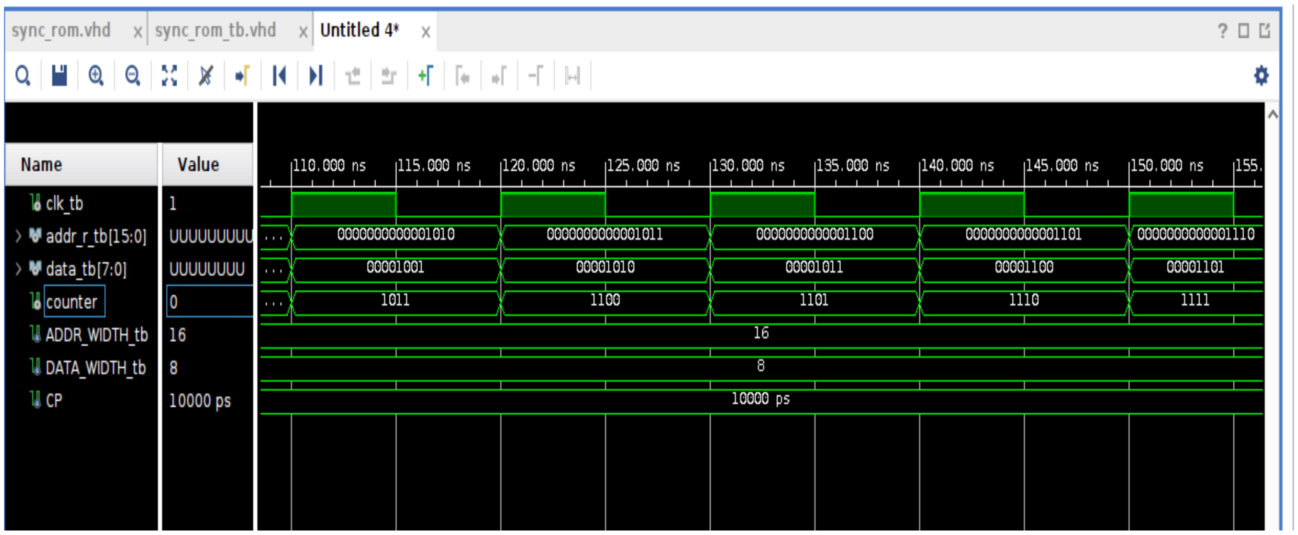


Fig 2.4 Part 1 16-bit Waveforms

Part 2:

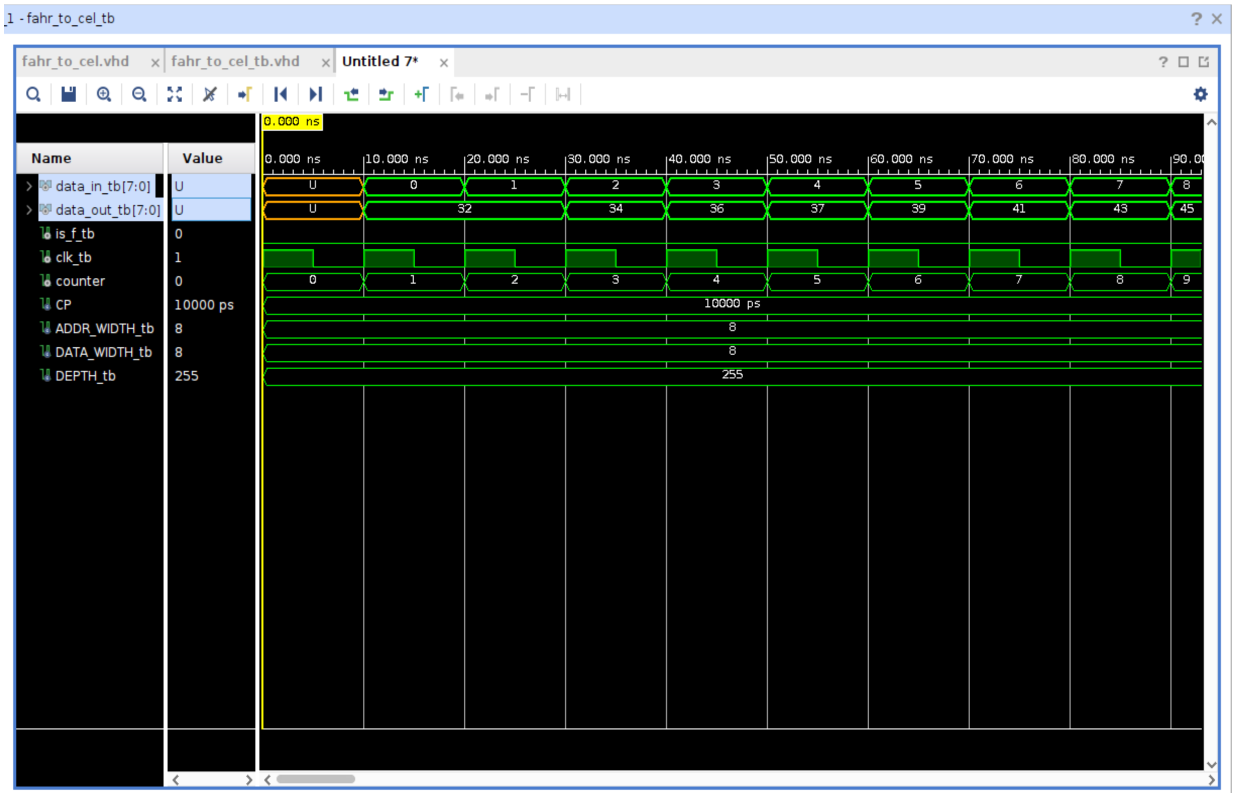
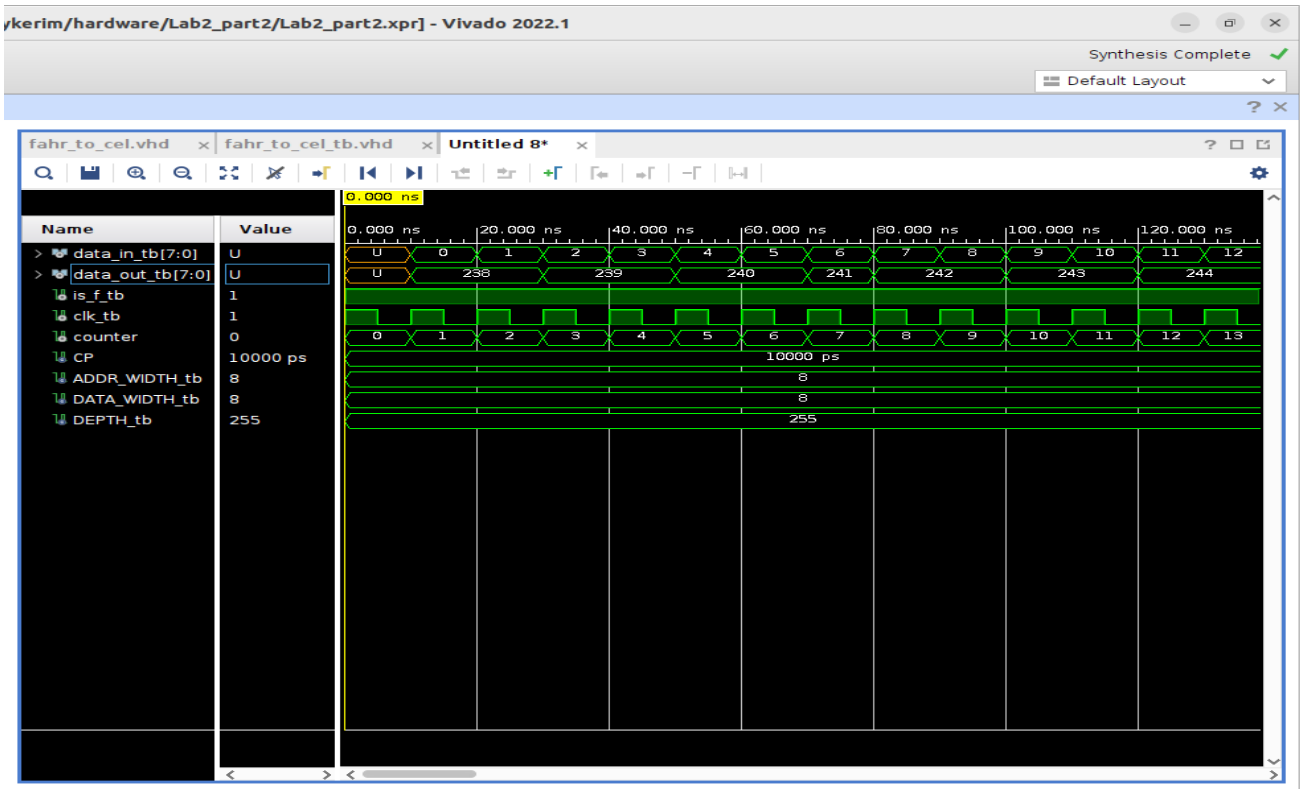


Fig 2.5 Part 2 Celsius to Fahrenheit waveform

In the above waveform, it shows celsius values being converted into fahrenheit values. Again, the actual input data and output values are 1 clock cycle apart. Meaning at the rising edge of the clock, data from the LUT is read and then is displayed in the waveforms. And at the next clock cycle the converted value is then displayed in the waveforms. Also, inside the testbench, the bit that checks if the data input is fahrenheit or celsius is currently set to 0 when performing the celsius to fahrenheit conversion.



Name	Value
data_in_tb[7:0]	U
data_out_tb[7:0]	U
is_f_tb	1
clk_tb	1
counter	0
CP	10000 ps
ADDR_WIDTH_tb	8
DATA_WIDTH_tb	8
DEPTH_tb	255

```
data_in_tb:  U 0 1 2 3 4 5 6 7 8 9 10 11 12
data_out_tb: U 238 239 240 241 242 243 244
counter:     0 1 2 3 4 5 6 7 8 9 10 11 12 13
CP:          10000 ps
ADDR_WIDTH_tb: 8
DATA_WIDTH_tb: 8
DEPTH_tb:    255
```

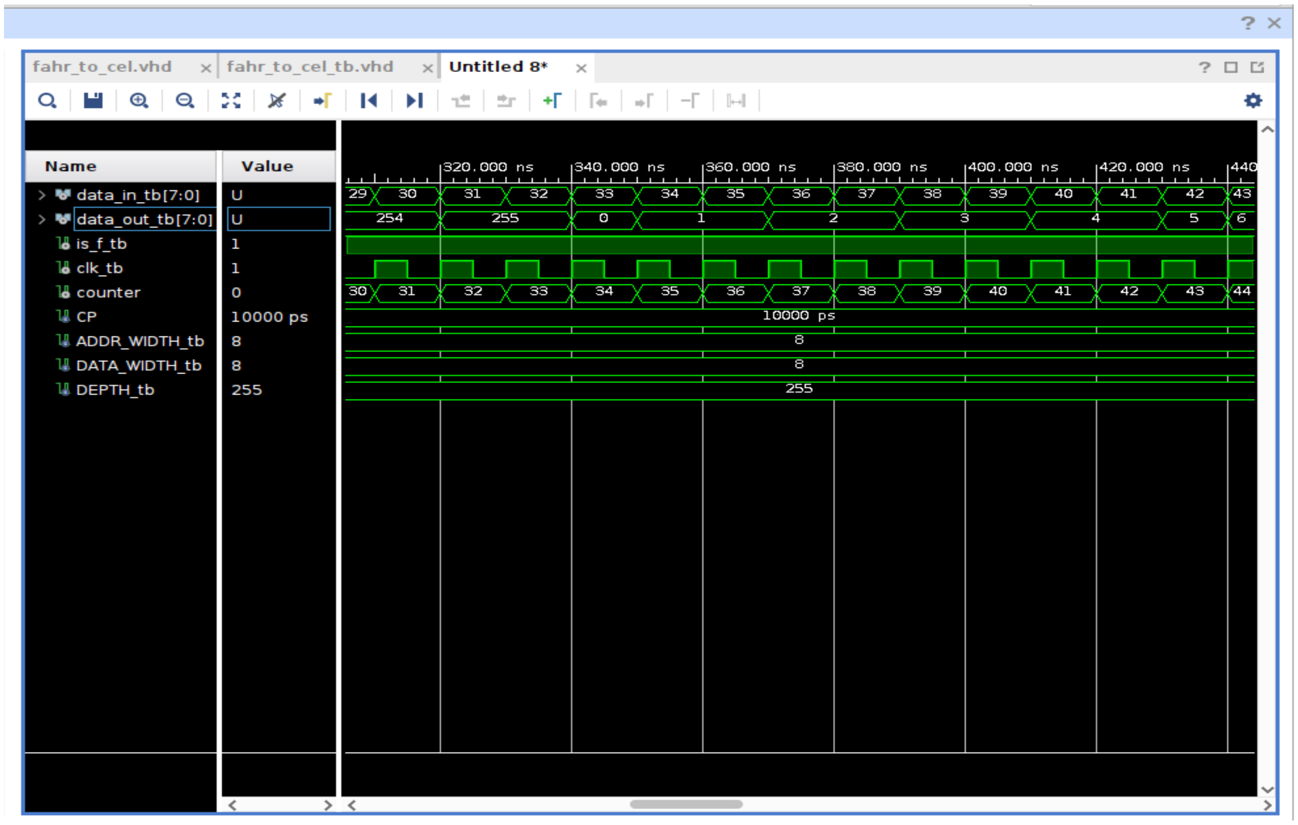


Fig 2.6 Fahrenheit to Celsius Waveform

The above two screenshots show Fahrenheit values and their celsius counterparts. There is a bit of an issue initially. Fahrenheit values from 0 to 31 are not defined in my code, this is because I am using code that was given to us in a template but at the same time it was my responsibility to make sure that the waveforms don't print out any garbage data. After 32, the waveforms begin to show the correct values. Since we do not have float values, a lot of the data gets truncated down. An possible solution to fix the garbage data would be to hardcode the test bench so it starts at 32 instead of 0. And another solution would be to adjust the actual dataset file so the negative temperature values of celsius would be read as well.

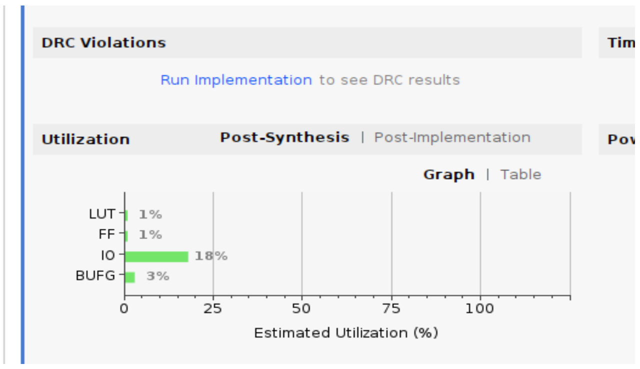


Fig 2.7 Fahrenheit to Celsius Usage

Part 5:

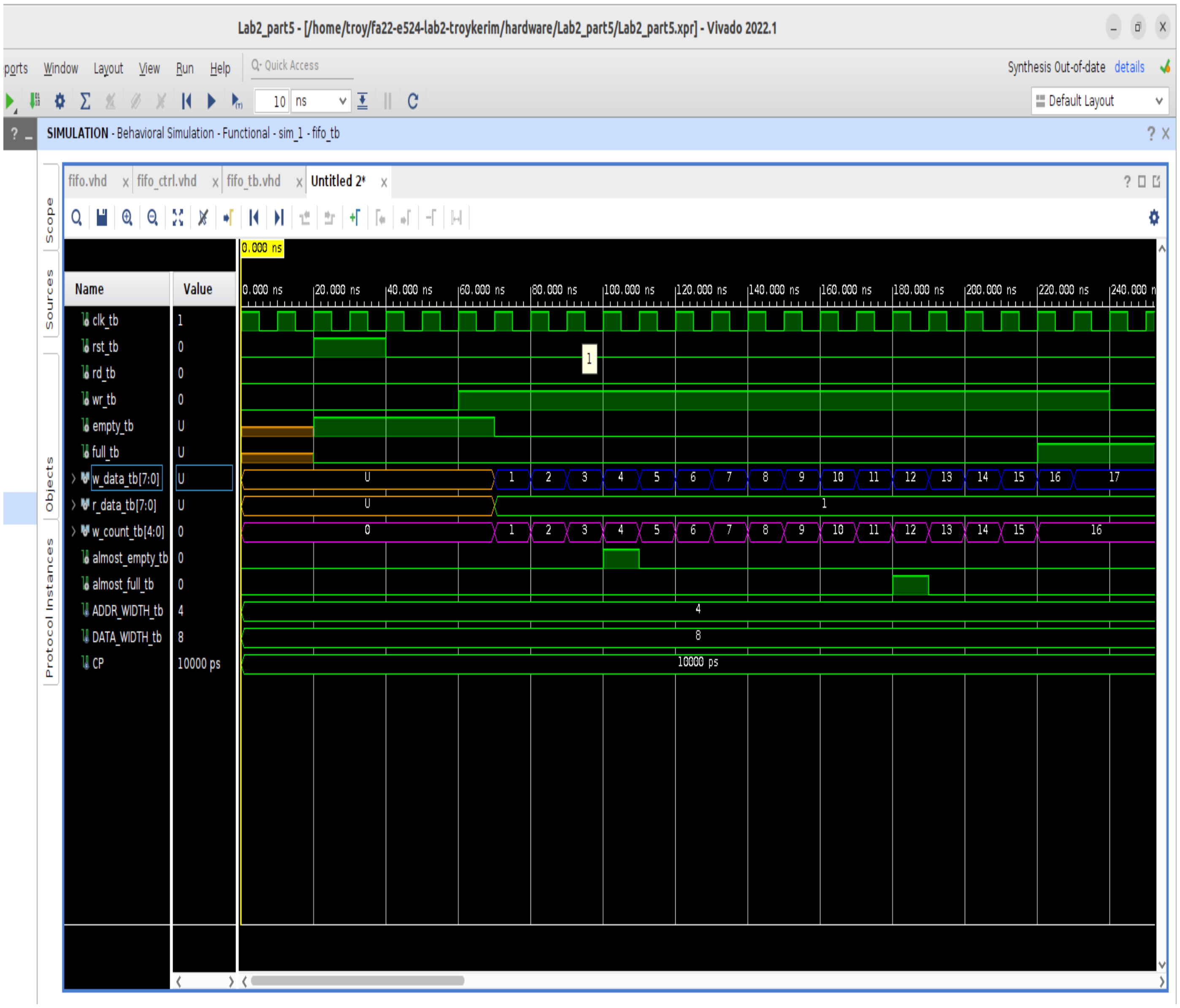
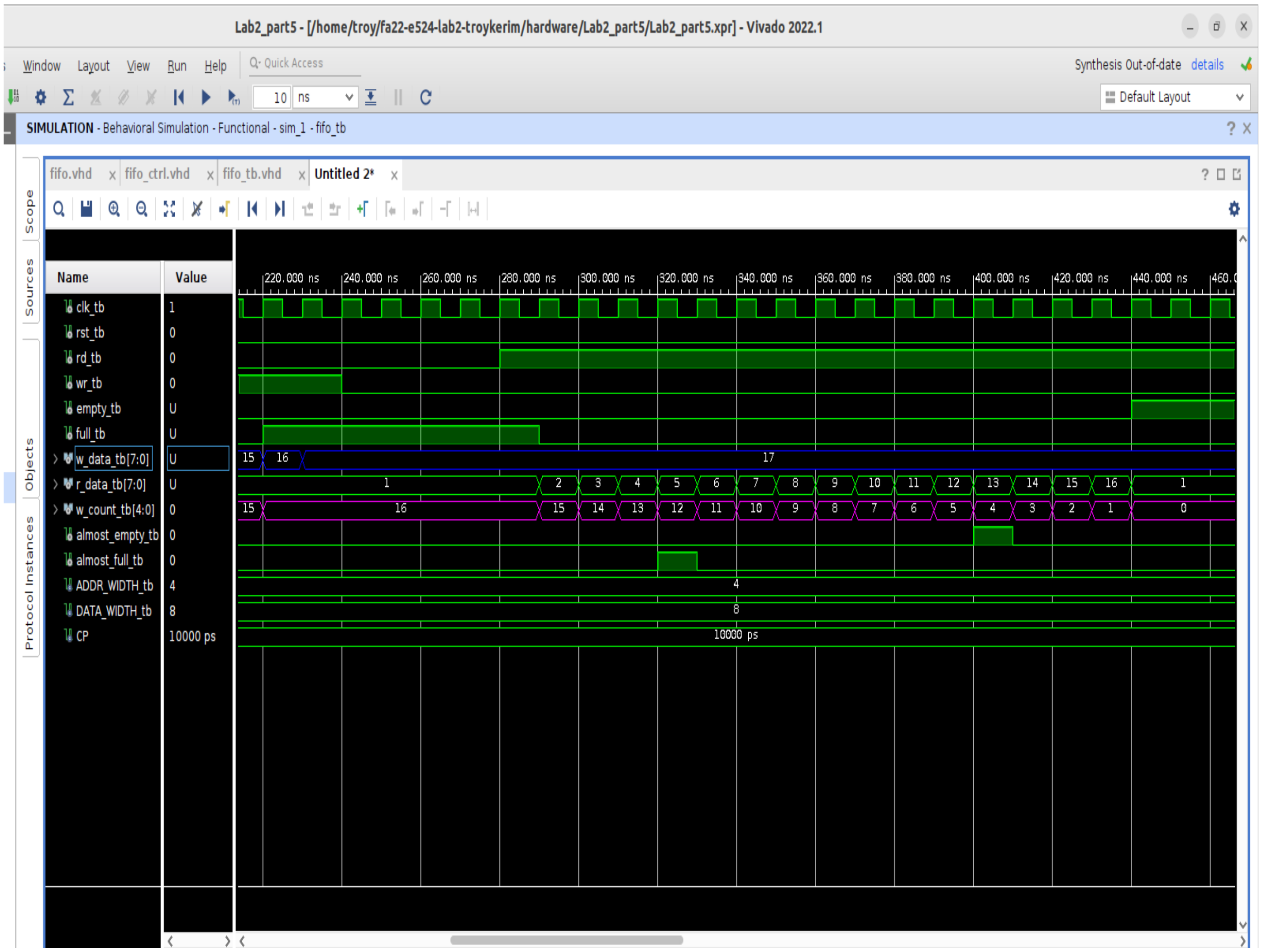


Fig 2. Part 5 FIFO Extended Status Write Operation

The screenshot above shows the write operation. Initially, the empty_tb is set to 1 because the FIFO is empty. Its a flag to let us know the overall status of the FIFO similar to the full_tb which will show the opposite later in the simulation. In order to start writing to the FIFO the wr_tb needs to be set to 1. While wr_tb is set 1 we can write into the FIFO and for about 16 clock cycles we can write in data. At the 4 data points the almost_empty_tb is set to 1 rather early. This occurs because I did not make logic for it to basically only happen during the read operation. This is something that should in theory be better suited to be set 1 during read operation because we are removing data from the FIFO. At the data point 12, almost_full_tb is set to 1 to let us know that the FIFO is almost at max capacity. After data 16, 17 follows and while it shows up in the simulation it did not get stored into the FIFO.


Lab2_part5 - [/home/troy/fa22-e524-lab2-troykerim/hardware/Lab2_part5/Lab2_part5.xpr] - Vivado 2022.1
Window Layout View Run Help
Quick Access
Synthesis Out-of-date details
10 ns
Default Layout
SIMULATION - Behavioral Simulation - Functional - sim_1 - fifo_tb
fifo.vhd fifo_ctrl.vhd fifo_tb.vhd Untitled 2*
Scope Sources Objects Protocol Instances
220.000 ns 240.000 ns 260.000 ns 280.000 ns 300.000 ns 320.000 ns 340.000 ns 360.000 ns 380.000 ns 400.000 ns 420.000 ns 440.000 ns

Name	Value
clk_tb	1
rst_tb	0
rd_tb	0
wr_tb	0
empty_tb	U
full_tb	U
w_data_tb[7:0]	U
r_data_tb[7:0]	U
w_count_tb[4:0]	0
almost_empty_tb	0
almost_full_tb	0
ADDR_WIDTH_tb	4
DATA_WIDTH_tb	8
CP	10000 ps

Fig 2. Part 5 FIFO Extended Status Read Operation

Above is the read operation, which is pretty much the same process as the write operation but just performing the opposite task. Because FIFO is first in first out, the first data read after we set wr_tb to 0 and rd_tb to 1 is the value 1 and then followed by 2, 3 and etc. Similar to write operation, the almost_full_tb is set to 1 because there is no logic that is set up to make sure that it only turns on during write operation. Once we read all the data the empty_tb is set to 1 again.

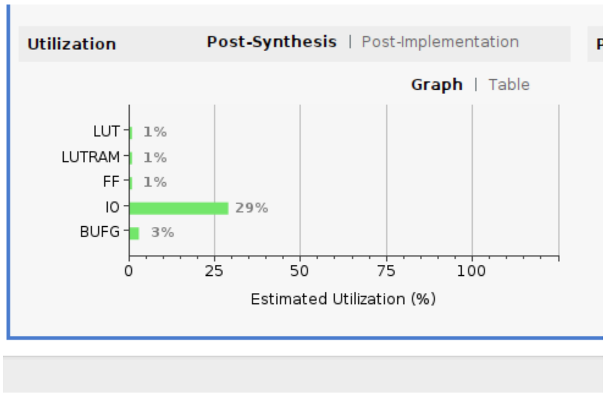


Fig 2.9 Part 5 Utilization Chart

Implemented Designs

The next series of screenshots show RTL schematics and Synthesized schematics.

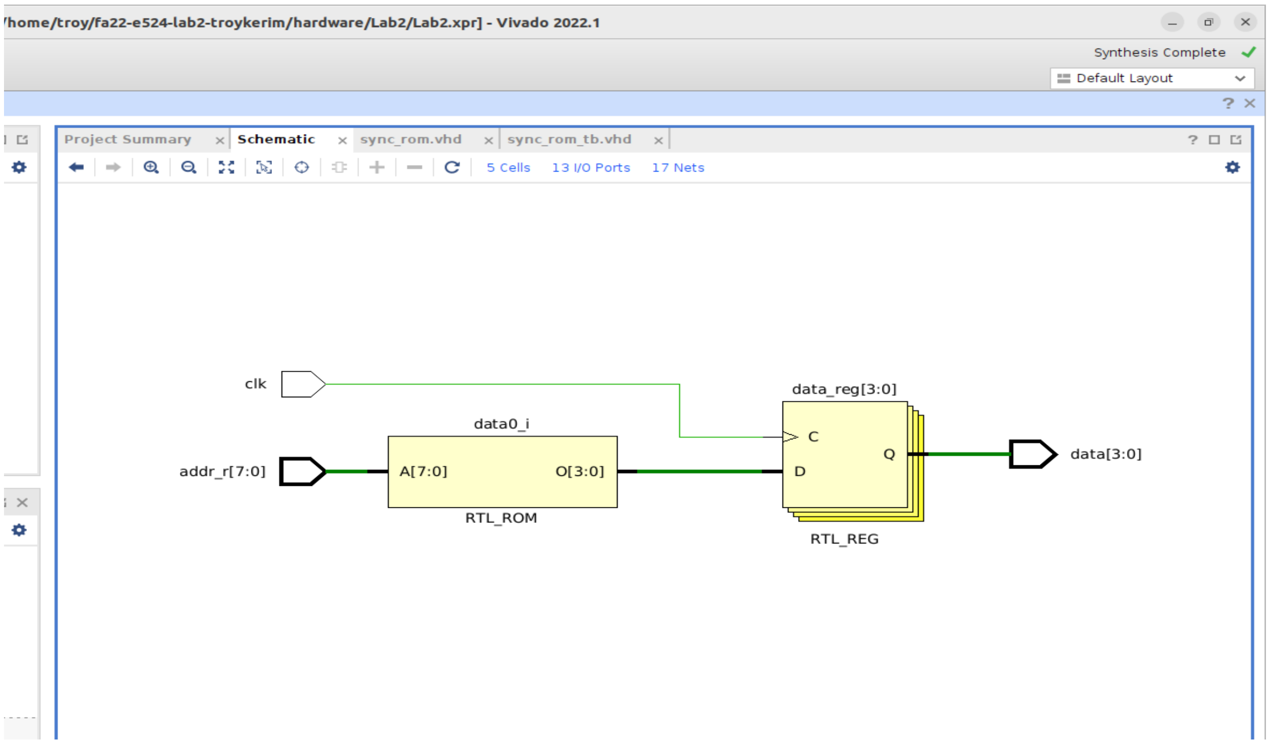


Fig 2.10 Part 1 8-bit Schematic

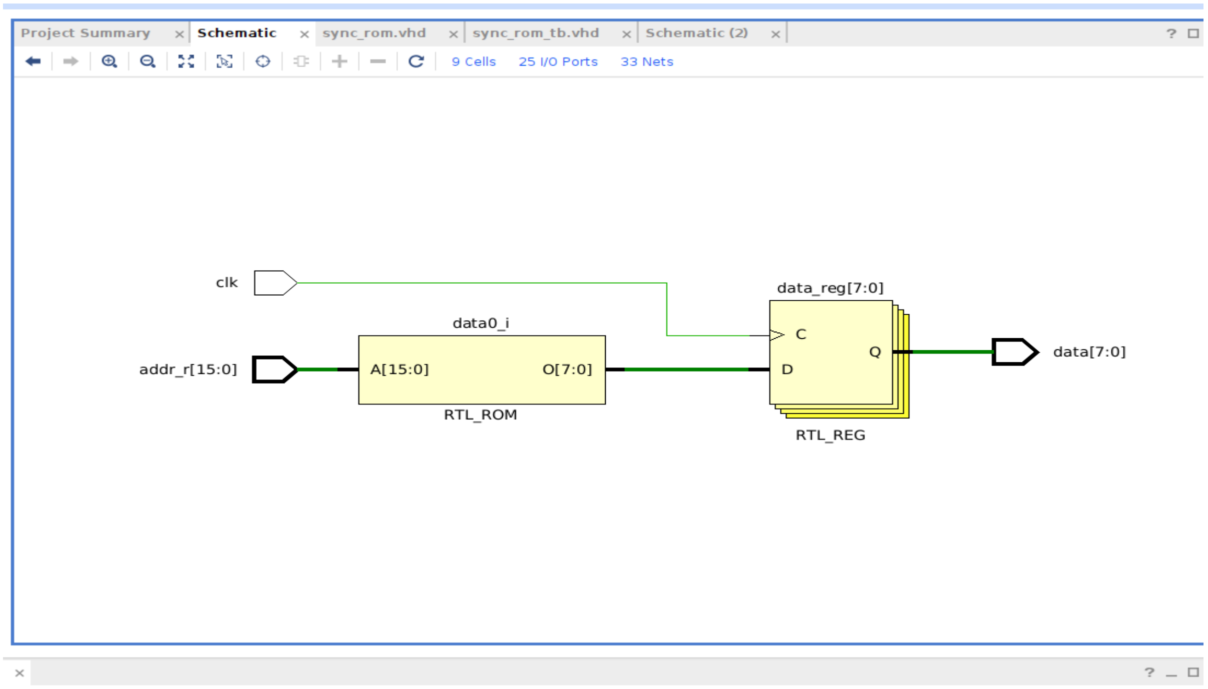


Fig 2.11 Part 1 16-Bit schematic

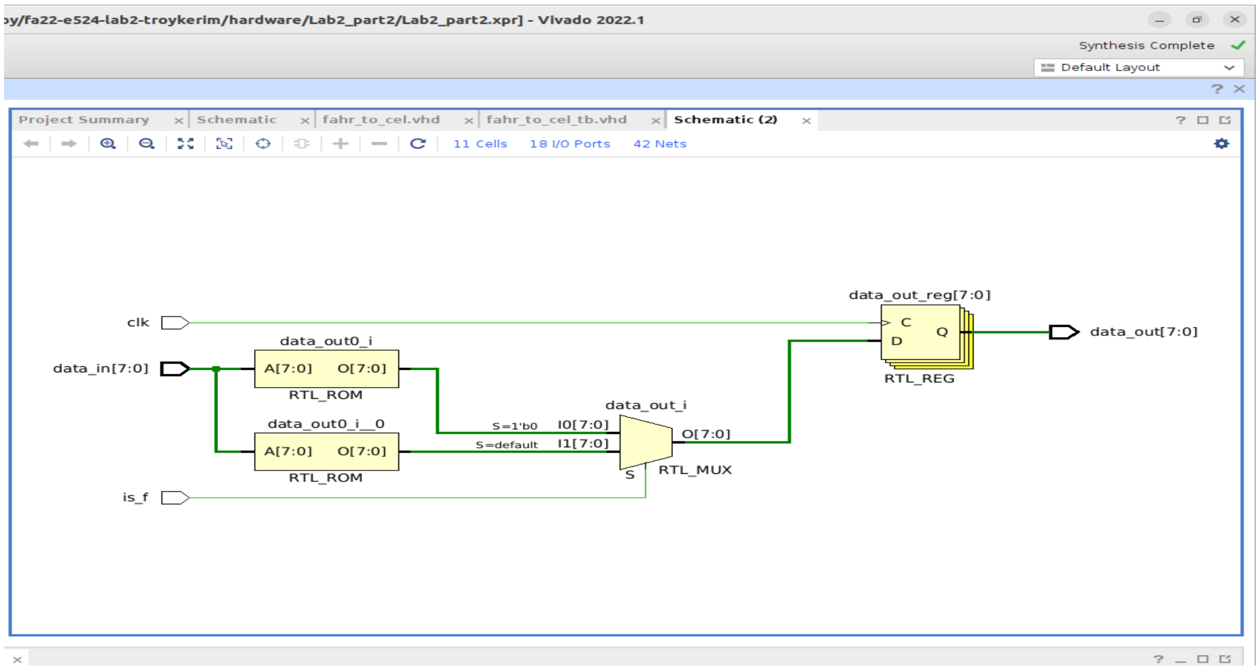


Fig 2.12 Part 2 Fahrenheit to Celsius Converter Schematic

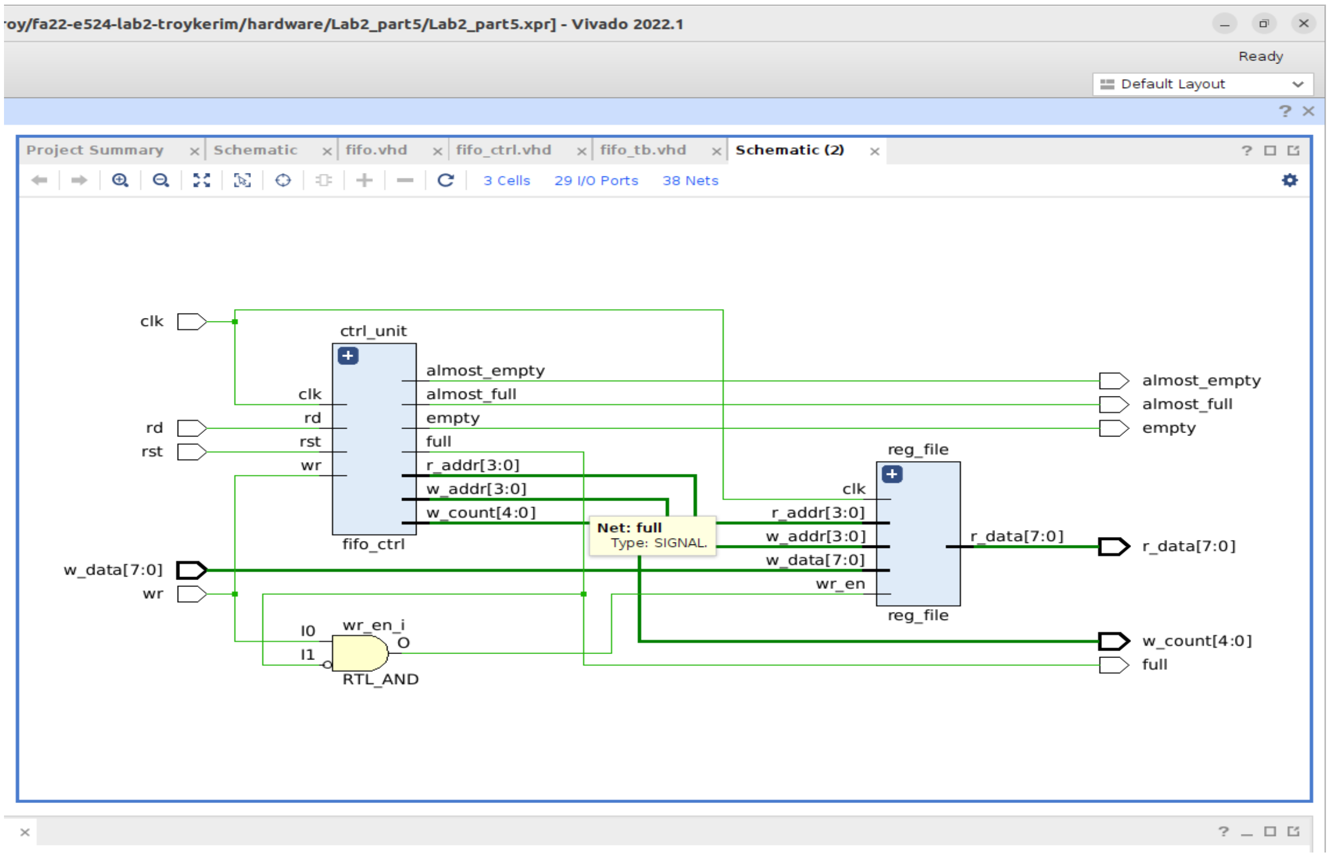


Fig 2.13 Part 5 FIFO extended status Schematic

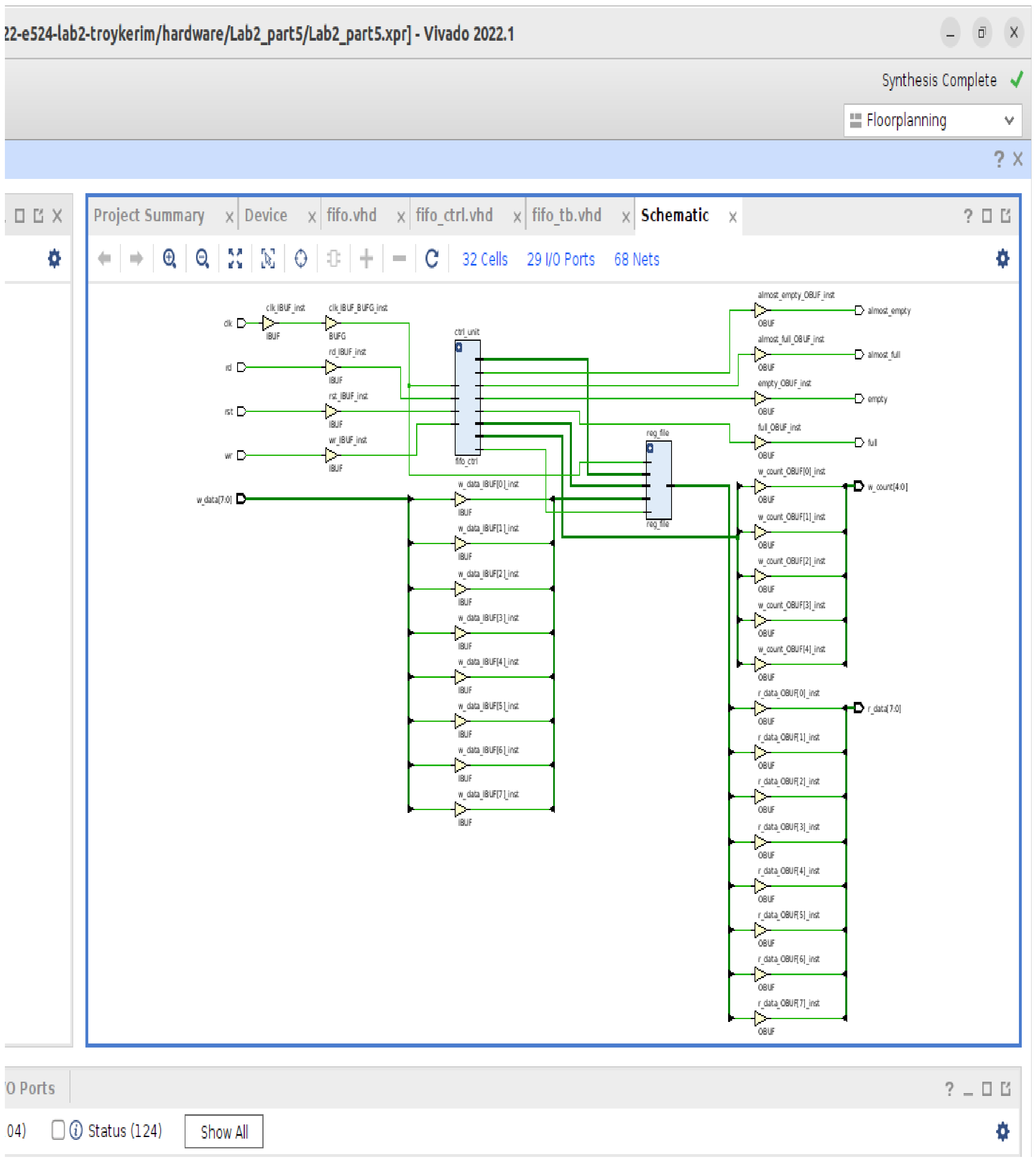


Fig 2.14 Part 5 FIFO extended Status Synthesized Schematic

Conclusion:
In this lab we learned to create Lookup tables. A topic that is both very interesting and very important in the field of FPGA design. We also learn how to read and write, specifically write, from a text file. This is something that in the previous class we did not really cover all that well. I struggled primarily with the text file creation because we were originally supposed to create the code that generates the text files, specifically for part 1. Part 2 I did not have a bad time making the files for that. Until part 5 was shown in class I had a hard time understanding the concepts of how to implement the FIFO with more statuses. I was overthinking parts of it especially when it came to port and signal creation. However, I did not really know how to set up my reset in the testbench, how to configure the data values like how it was shown in class and I did not know how to create or really implement a function all that well. But after reviewing what was shown in class I do feel much better off now with all those concepts that I was comfortable with beforehand.

Appendix:
https://github.com/csun-ece/fa22-e524-lab2-troykerim/