Lab Report:
AWorking with R- and I-type Instructions

Nisha Patel

November 22, 2016

Abstract

In this report, I am discussing about finishing off our Register File and creating the R-Instruction and I-Instruction datapath.

1 Introduction

For this lab, we are continuing to develop a working CPU. Using the components and skills learned in the previous lab, we're going to finish off our Register File and work on handling our simple instruction set. 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

Hooking the Pieces Together
We have so far been building more and more complex pieces from our simple building blocks. We are going to continue this trend for this lab. The most complex piece we will need is the Register File.
We will have 8 nibble registers contained in the register file. You will need two selectors to determine which registers will be read, and a decoder to decide which register to write to. Remember, you should require a falling edge clock to actually perform a write to the requested register.

Your Register File will be connect to your nibble ALU, both supplying data from registers for processing, and allowing the ALU results to be stored in it.The only "new" code should be the creation of a clock - this will allow you to focus on ensuring that your CPU follows the course ALU codes.

Initializing Signal Values
As we move on to more complex circuits, we will occasionally want to initialize signals to certain values (it will help prevent 'u' for undefined values). In your entity definitions, you may want to set it to something like '0', while in testbenches we may set it to something more interesting...You can do this via an initializer.

Clocks in TB
Our testbenches are starting to get very complex, in order to take some of the burden off of our shoulders, we can work on automating the clock. I know some of you may have already decided to take this approach on your own, but I'd like to get us all down to a full clock cycle of 4ns (signal changes every 2ns). Some sample code:

```
signal clk : std_logic := '0';
constant clk_period : time := 4 ns;
--declare any other signals needed
BEGIN
--port map for unit you're testing goes here
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;  --for 2 ns signal is '0'.
        clk <= '1';
        wait for clk_period/2;  --for next 2 ns signal is '1'.
   end process;
--rest of test bench goes here
```

Figure 1: This is a sample code

Working with the code For this lab, we should be able to start pushing through instructions following the course ALU codes see V2
Things to note:

- The first bit will determine whether the R command (0) or an I version (1) should be run.
- You will need to split off your inputs into multiple branches - sometimes you will have an immediate, and sometimes a register and a shift amount.
- Your Rd, Rn, and ALU op code should stay at the same bit positions, so should not need splitting of lines.

3 Experimental Setup

Parts List

- A selector (3 bit and 1 bit selectors needed)
- A 3-bit Decoder - for inside the register file
- Nibble ALU
- 4 bit Register
- Register File with 8 Registers

4 Experiments

We're continuing our work on building our way up to a working CPU that can handle most (if not all!) of the LEGv8 instruction set. We will be developing our register file, connecting it to the ALU, and running some

simple instructions through the whole thing. We are going to build R-Instruction Datapath and I-Instruction Datapath off of register file.

4.1 Experiment 1: Building your register file

In this experiment, we are using MUXs, decoders, and nibble registers built previously. We will be combining everything in order to build out 8 register register file. Register file expects to both read and write in a single clock cycle - we will start by reading from the register file, and then writing back on the falling clock edge.

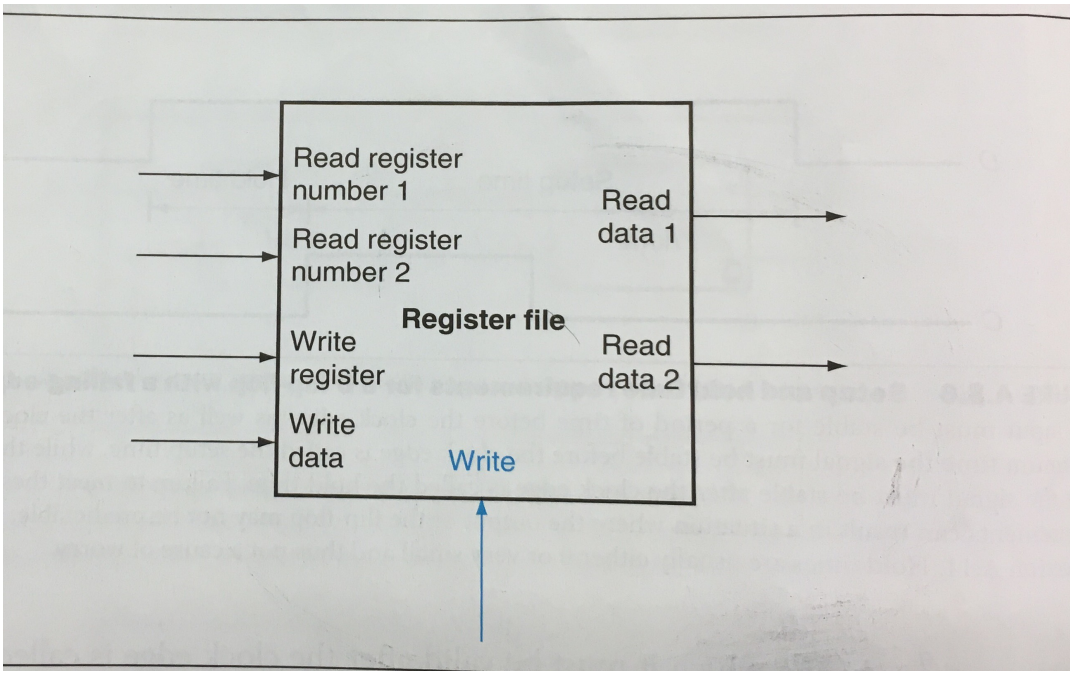

Figure 2: This is a blackbox logic diagram of a register file

FIGURE A.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

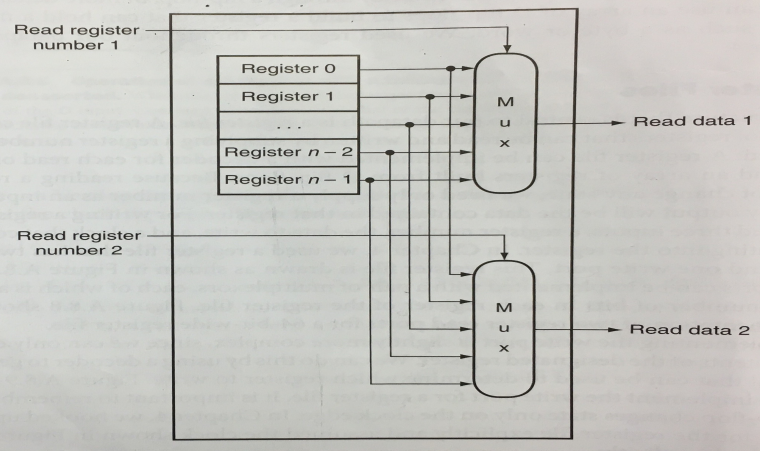

FIGURE A.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of *n*-to-1 multiplexors, each 64 bits wide. The register read number signal is used as the multiplexor selector signal. Figure A.8.9 shows how the write port is implemented.

Figure 3: This is a blackbox logic diagram of a read section

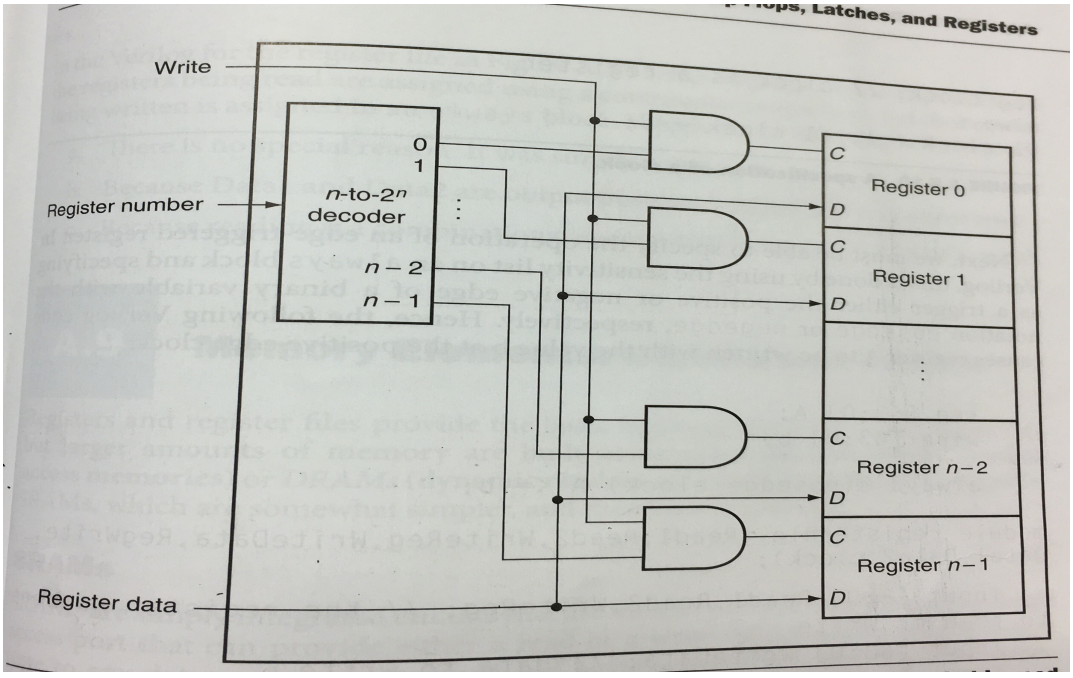

Figure 4: This is a blackbox logic diagram of a write file

4.1.1 Experiment 1 Conclusions

First of all, we are building small parts and then putting them together as needed. I started with the write section and then I build read section. At last, I merged them together. I choose iterative approach just to be sure that each individual part worked. I used lot of temporary variables for the nibble register outputs, temps for all and gates, temps for decoder and temps that combined two muxs together in order to make it a 8 bit. Truly speaking, my register entity looks like a crap because I have so many temporary variables, but it works. I randomly chose appropriate inputs for the testbench. I have not tested every possible inputs because it is insane to do that so I tested fifteen inputs to make sure it works.

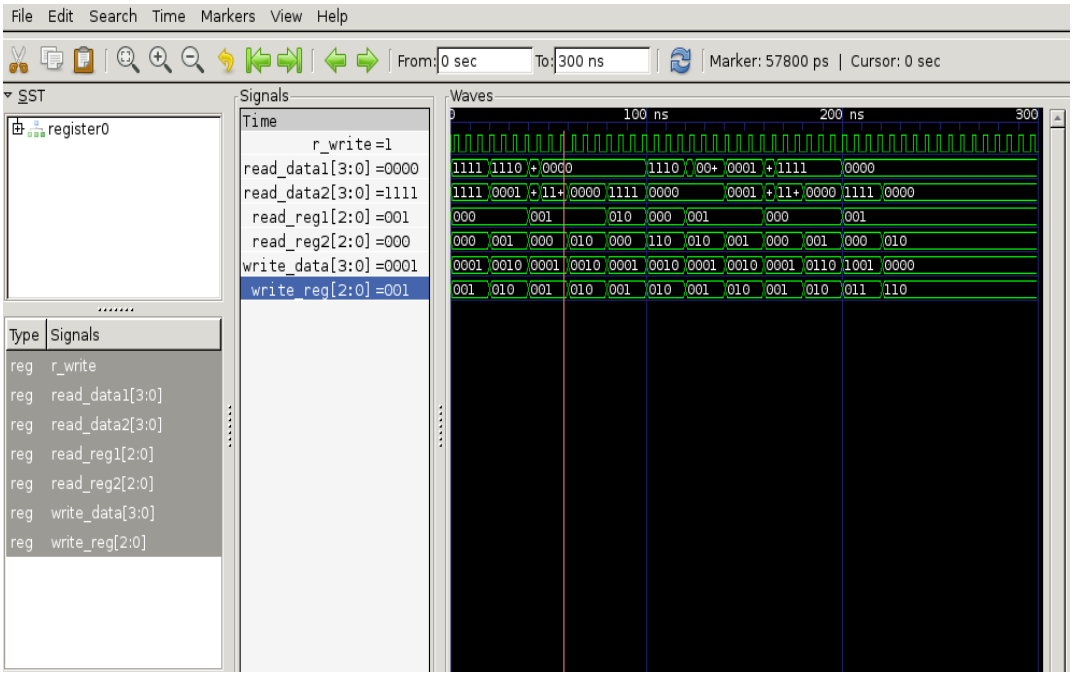

Figure 5: This is a gtkwave of a register file

4.2 Experiment 2: Creating the R-Instruction Datapath

Once your Register File is complete, we will be working to connect it to the ALU. Unfortunately, there is no good diagram in the textbook for this simple connection. The closest we're going to get is figure 4.10 on p 269. This also includes load and store instructions (hence the Data memory object). If you remove the sign extend, the data memory, and the MemtoReg MUX, and the ALU Src, this figure most closely approximates what we are going for.

Components needed:

- Register File
- ALU

Inputs:

- 16-bit instruction
- Clock

Outputs:

- ALU result

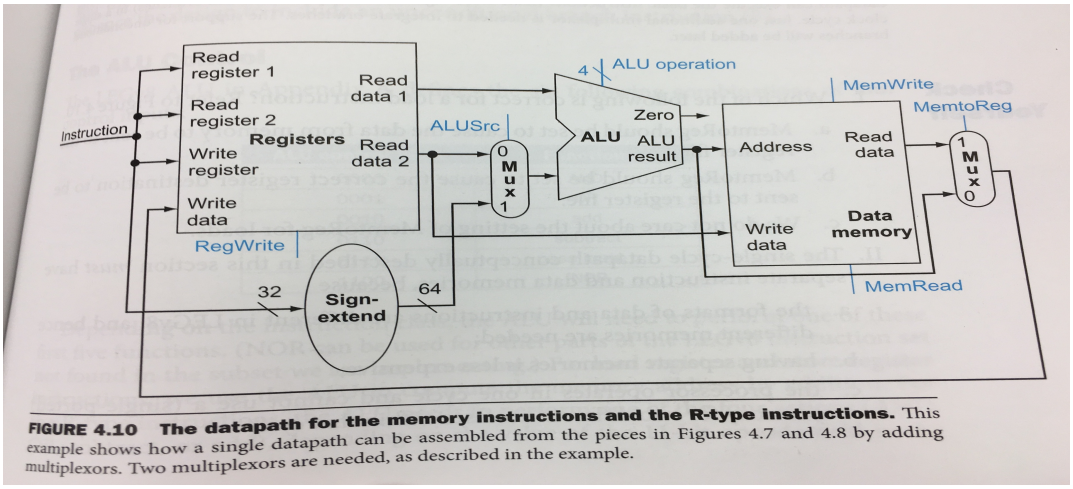

FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

Figure 6: This is a logic diagram of a Instruction Datapath

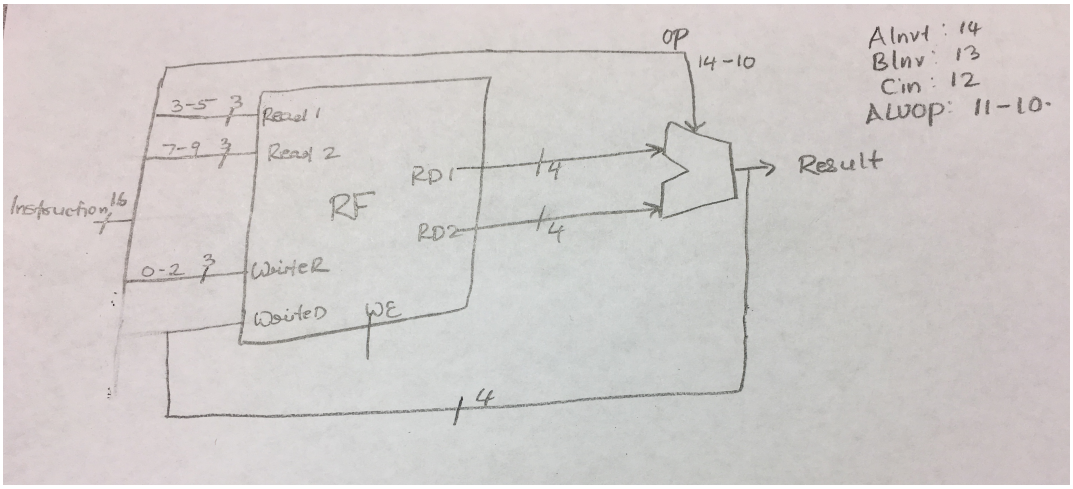

Figure 7: This is a logic diagram of a R-Instruction Datapath that we drew in class

4.2.1 Experiment 2 Conclusions

In this experiment, following are the subsets of the inputs that I have used: Total 16 bit instructions that is broken down into individual bit that goes into there appropriate location. For example, (read1 (3-5), read2 (7-9),writeR (0-2), OP(14-10 (ainverse(14), binverse(13), cin(12), ALU op (10-11) and a clock bit). Furthermore, we are ignoring bit number 15 and 6 for this experiment because we are not using them. I randomly chose appropriate inputs for the testbench. I have not tested every possible inputs because it is insane to do that so I tested fifteen inputs to make sure it works.

Yes I chose to initialize the writeData and set it equals to the zeros. I created a temp variable that holds the initialized signal and passed it into the register, then I override it after I get the result from the ALU. I decided to initialized the writeData because it takes the output from the ALU as it's input and we can't get

the output until we finish the cycle. Everything was most challenging for this lab. Specifically, combining things together according to there appropriate bit location.
I guess I am, maybe....ready for the I-Instructions. I got the R-Instruction, so I only have to add a MUX and move a few things around to build the I-Instructions - I am ready!

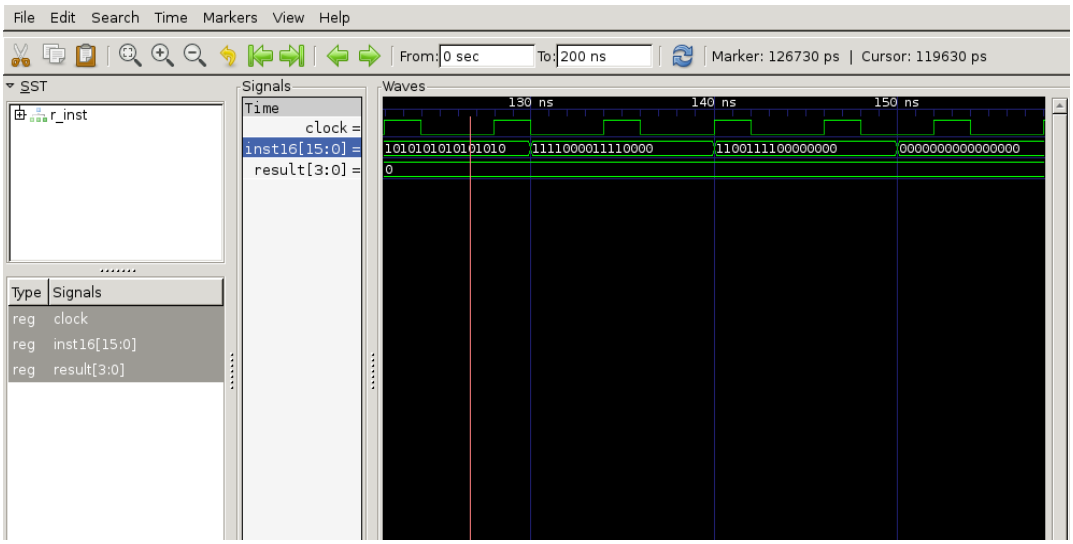

Figure 8: This is gtkwave of the R-Instruction Datapath

4.3 Experiment 3: Creating the I-Instruction Datapath

In this experiment we are inserting the extra components, MUX and ALUSrc, into the previous experiment. Since we are only storing Nibbles of data, and our instruction format supports immediates a full nibble in size, we will not need a sign extension (Still working from Fig 4.10), though we will need the ALUSrc MUX.

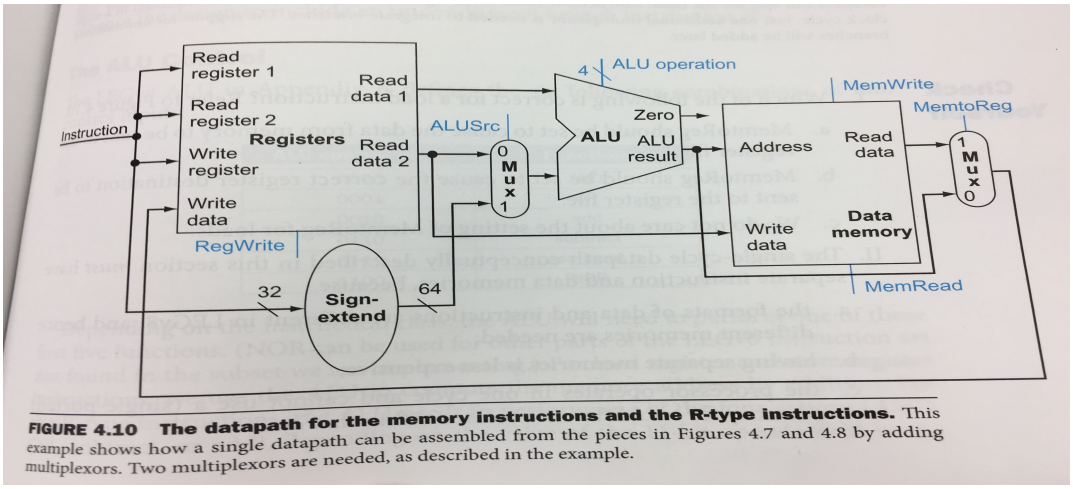

FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

Figure 9: This is a logic diagram of a Instruction Datapath from the book

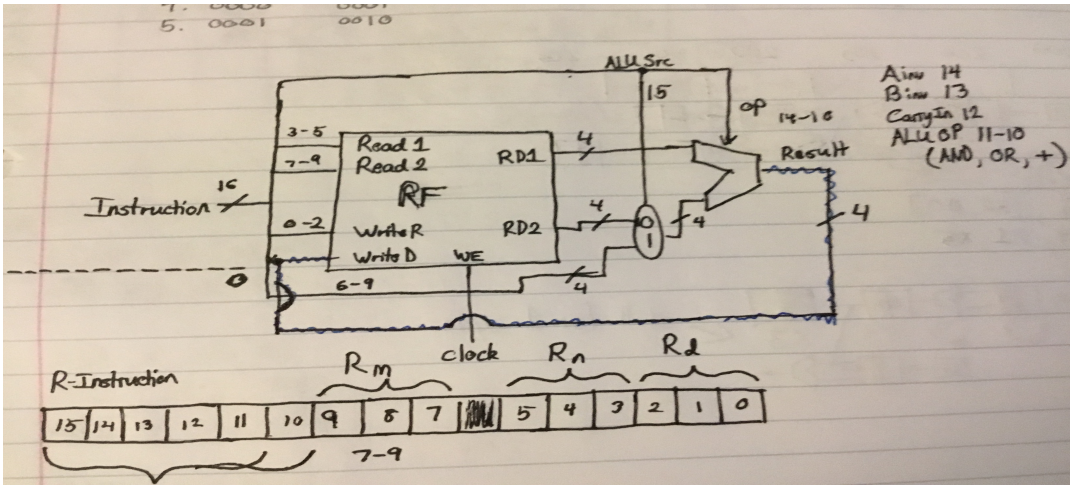

Figure 10: This is a logic diagram of a R-Instruction Datapath that we drew in class

4.3.1 Experiment 3 Conclusions

I randomly chose appropriate inputs for the testbench. I have not tested every possible inputs because it is insane to do that so I tested fifteen inputs to make sure it works. I chose to initialize the writeData and set it equals to the zeros. I created a temp variable that holds the initialized signal and passed it into the register, then I override it after I get the result from the ALU. I decided to initialized the writeData because it takes the output from the ALU as it's input and we can't get the output until we finish the cycle. Everything was most challenging. I was confused about the ALUSrc bit until I figured out that I need a 1-bit MUX. Yes, timing is the biggest issue that I have faced. Also, the clock was acting weird for some reason, but I think it has something to do with the timing.

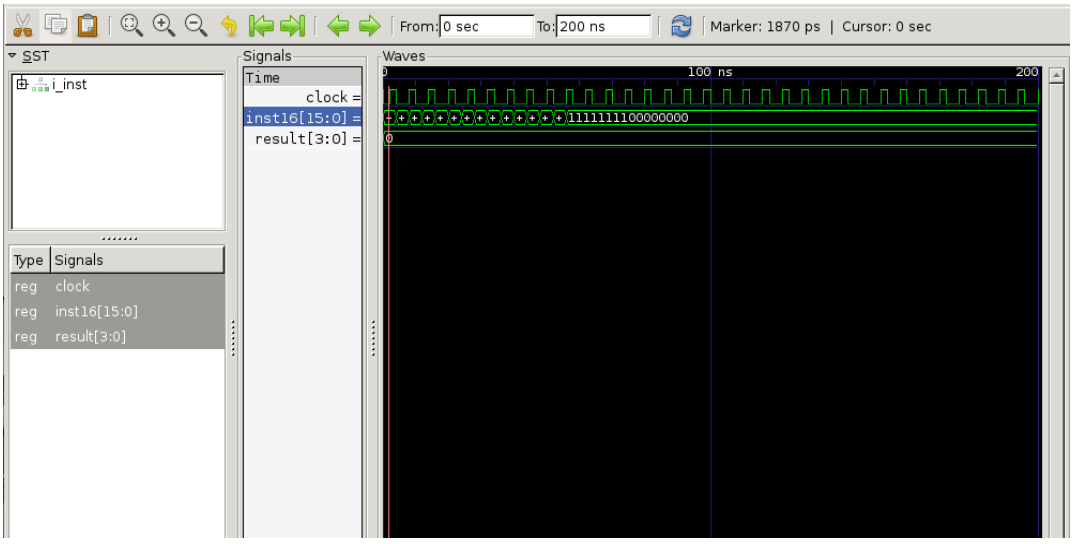

Figure 11: This is gtkwave of the I-Instruction Datapath

5 Conclusion

In short, first we created 8 register register file. Second, we created R-Instruction Datapath using our register file. Third, we created I-Instruction Datapath using our R-Instruction and adding a few other components.

5.1 Challenges

The entire lab was most challenging.
Note: There are some issues with my .vhld files, and I could not figure out therefore I am submitting what I have. I think it'd be better than nothing.