CS M152A Lab 1 Report

Hanna Co

October 20, 2021

1 Introduction and Requirement

The purpose of this lab was to familiarize students with smaller FPGA programs. We were given a small scale FPGA project, and were asked to modify and run it. This lab is meant to familiarize us with Verilog code and get some exposure to digital logic.

2. Design Requirement

a. Nicer UART

For workshop two, the requirements simply asked us to first modify a testbench program to output one line at a time, rather than one byte at a time. To do this, we simply created a register that could store four bytes, and we stored input bytes into this register. We would continue storing bytes until a carriage return ('\r') is received.

```
   always @ (negedge RX)
     begin
        rxData[7:0] = 8'h0;
        #(0.5*bittime);
        repeat (8)
          begin
             #bittime ->evBit;
             //rxData[7:0] = {rxData[6:0],RX};
             rxData[7:0] = {RX,rxData[7:1]};
          end
        ->evByte;
        //$display ("%d %s Received byte %02x (%s)", $stime,
name, rxData, rxData);
        if (rxData == 16'h0D) begin
            $display ("%d %s Received line %s", $stime, name,
rxDataLine);
        end
        else if(rxData != 16'h0A)  begin
            rxDataLine [31:0] = {rxDataLine [23:0],
rxData[7:0]};
        end
```

```
end
```

b. An Easier Way to Load Sequencer Program

The next part of this workshop asked us to identify the part of tb.v where instructions are sent to the UUT. Below is an image of this portion of the code.

```
tskRunPUSH(0,4);
tskRunPUSH(0,0);
tskRunPUSH(1,3);
tskRunMULT(0,1,2);
tskRunADD(2,0,3);
tskRunSEND(0);
tskRunSEND(1);
tskRunSEND(2);
tskRunSEND(3);
```

As we can see, the user tasks being called in this process are PUSH, MULT, ADD, and SEND. However, we were given an easier way to run this program: we wrote these instructions in binary, and placed them into a file called seq.code. Then, rather than writing every instruction we wanted to execute, we simply loaded in the seq.code file, parsed it line by line to execute our program.

c. Fibonacci Numbers

The last part of the lab asked us to write a binary sequence of instructions to print out the first ten fibonacci numbers.

00010100
00000000
00010001
11000000
11010000
01000110
11100000
01100100
11000000
01001001
11010000
01000110
11100000

01100100
11000000
01001001
11010000
01000110
11100000
01100100
11000000
Our Fibonacci Sequence Code

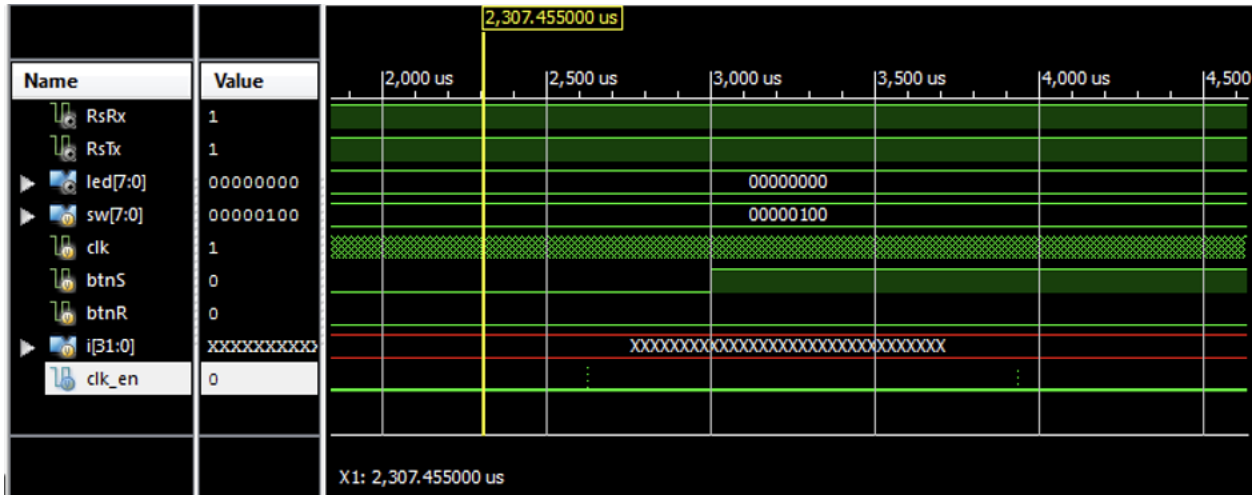
The first line is simply how many lines of instructions there are, and the rest of the file is the actual instructions for calculating and printing the fibonacci numbers.

3 Simulation Documentation

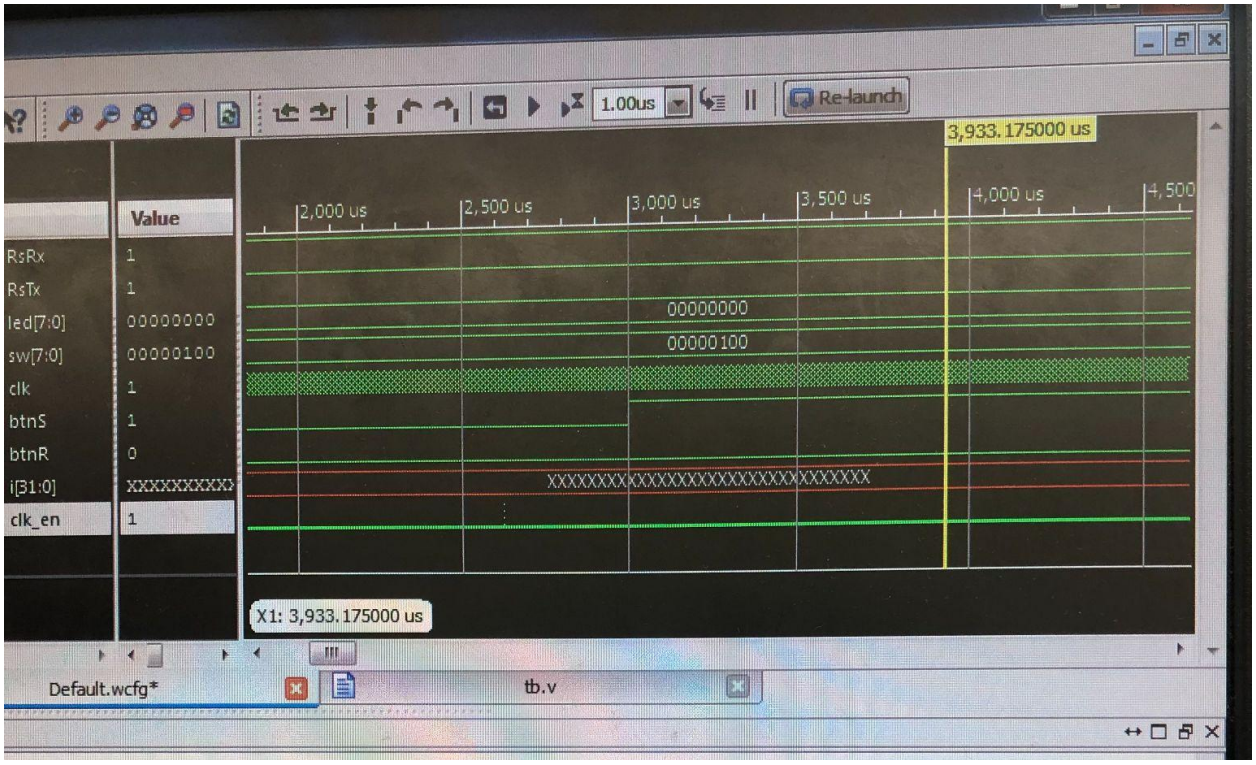
The second part of this lab was a bit more involved. Workshop two involved simulating the provided programs, and analyzing that and the source code to understand the program's logic.

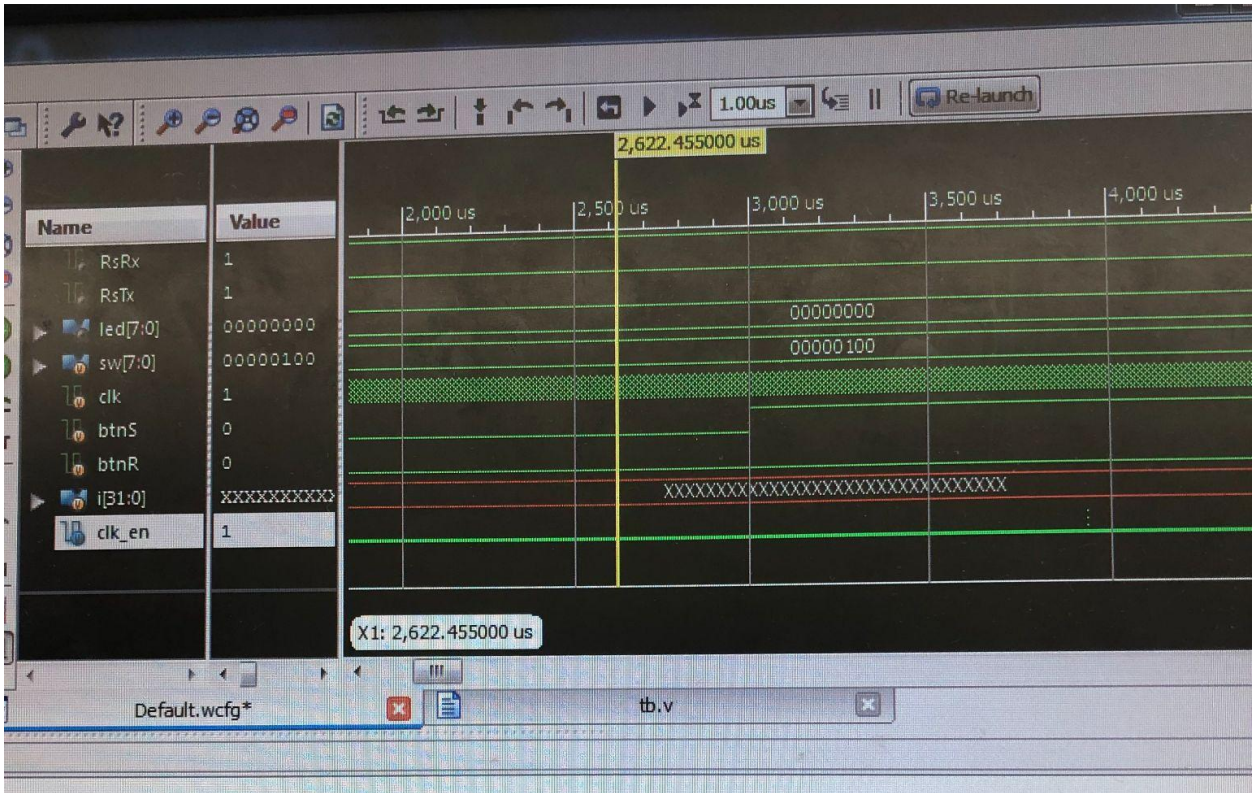
a. Clock Dividers

We simulated our program, and captured a screenshot of two cycles of clk_en; in other words, two instances where clk_en was high. The image is included below.



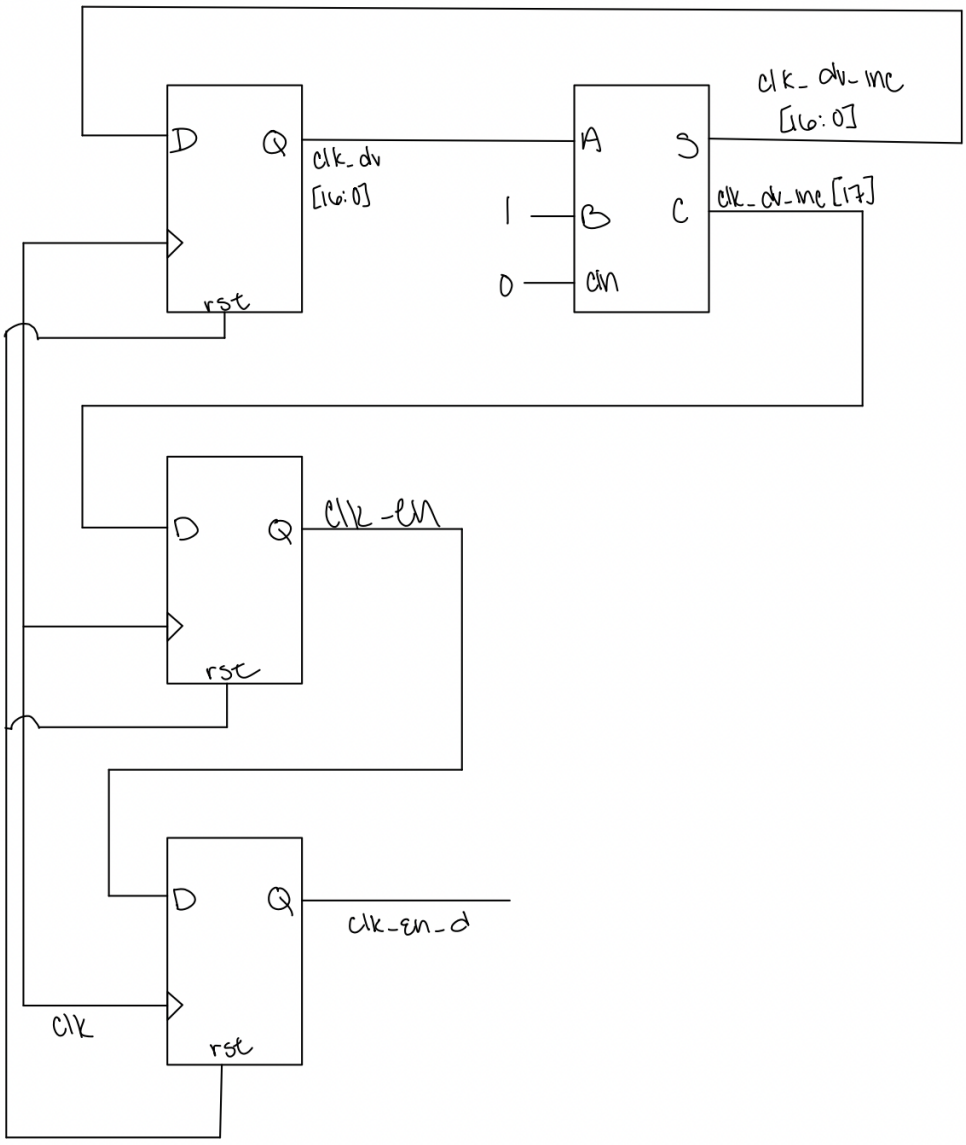
Using the simulator, we were able to get the exact time between the two cycles of clk_en. Screen captures of this are included below.





From this, we concluded that the period is 1310.72 microseconds. From this, we can also calculate the duty cycle, which is the interval where the signal is high, divided by the period, multiplied by 100%. The interval where the signal is high is 3933.185 microseconds - 3933.175

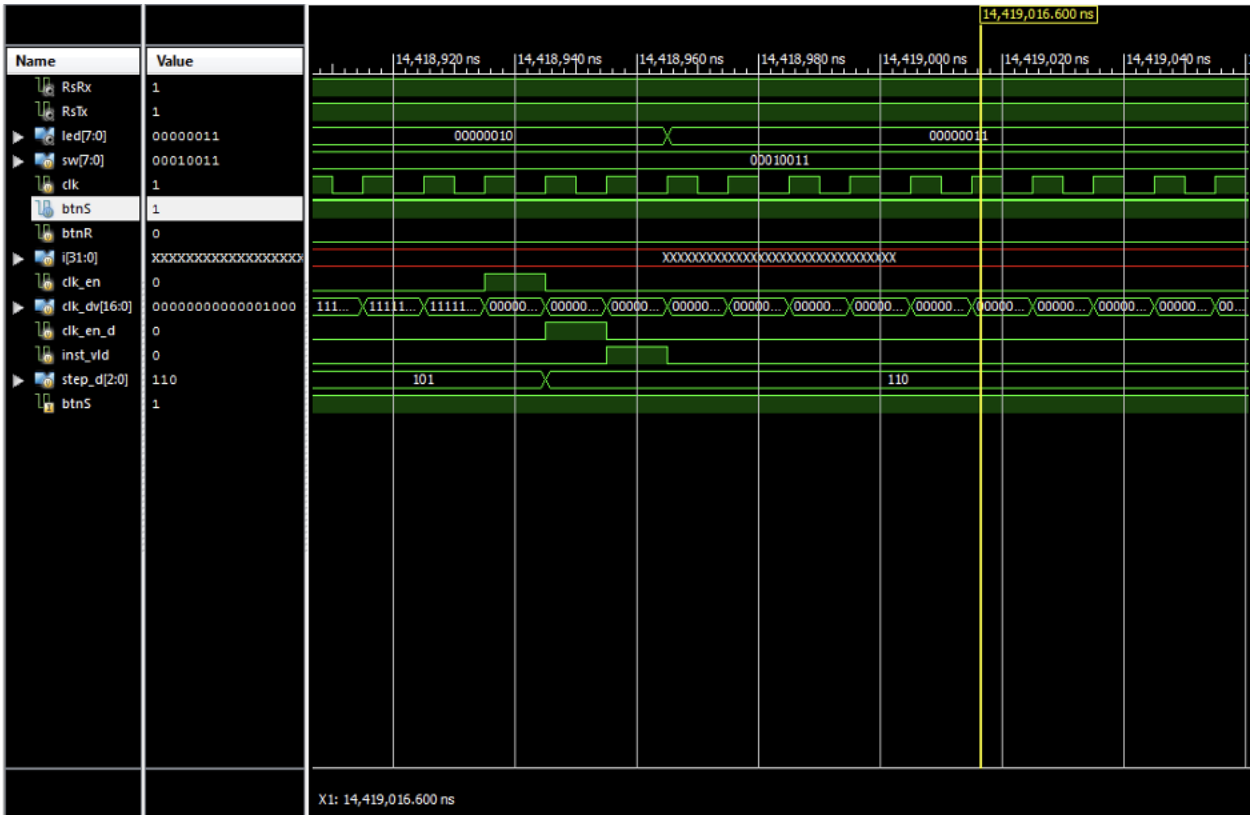
microseconds = 0.01 microseconds. Therefore, the duty cycle is 0.00762%. When clk_en is high, clk_dv is zero. Below I have included a simple diagram of the clk_dv, clk_en, and clk_dn_d signals.



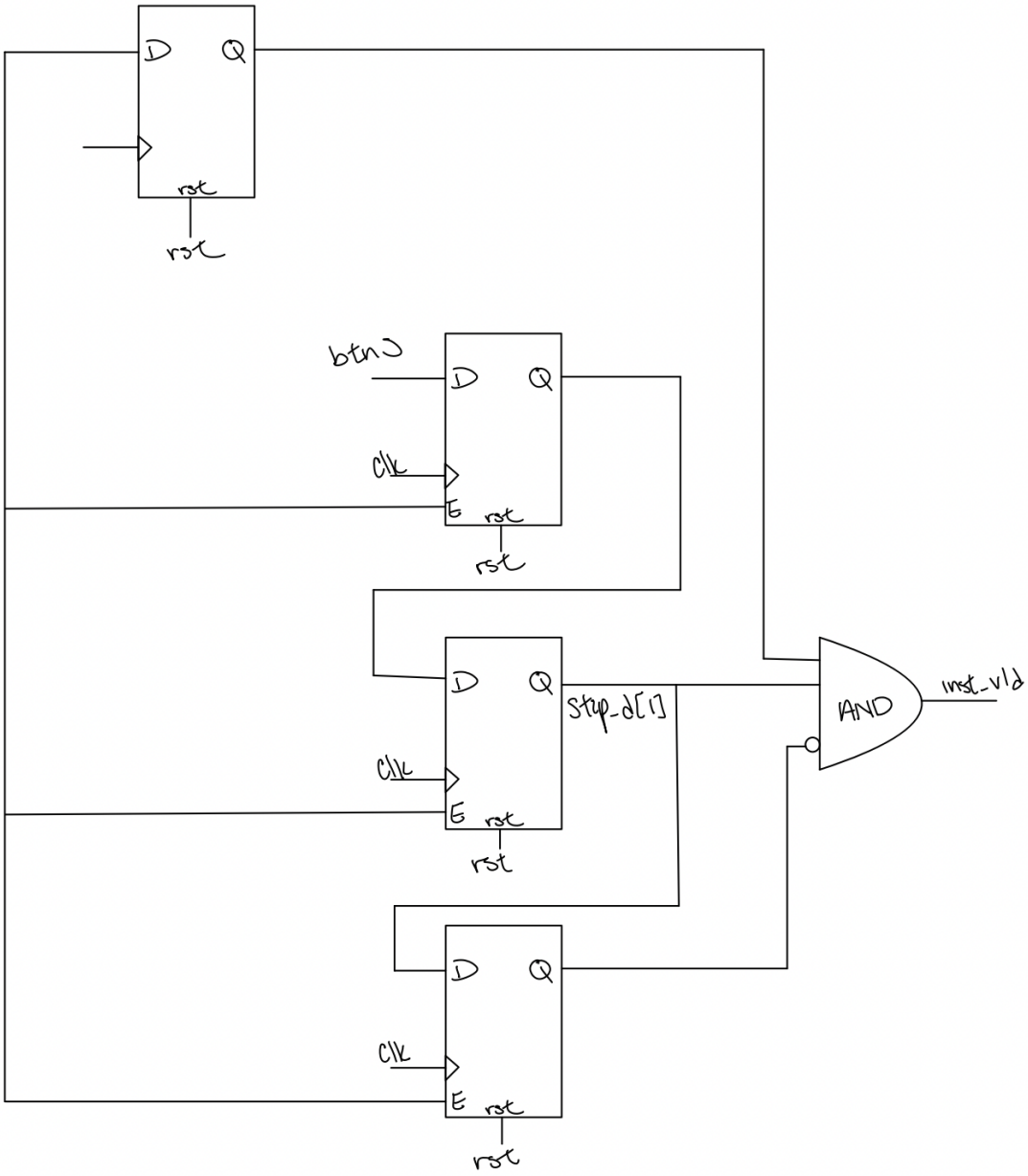
b. Debouncing

If clk_en_d was just high, then we want inst_vld to be high. The purpose of the clk_en_d signal is because we need to check if the button was just pressed. This is because the button press has not yet been processed, so we have clk_en_d, which is one cycle behind clk_en, in order to process the button press. If we change `clk_en <= clk_dv_inc[17]` to `clk_en <= clk_dv[16]`, the duty cycle will become 50%. This is because we are using binary digits, so

the value of each bit differs by a factor of two. So instead of overflowing every 2^17 bits, we overflow every 2^16 bits, which is half of our original duty cycle. The following waveform shows the relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld.



The diagram below is a simple schematic of the above signals.

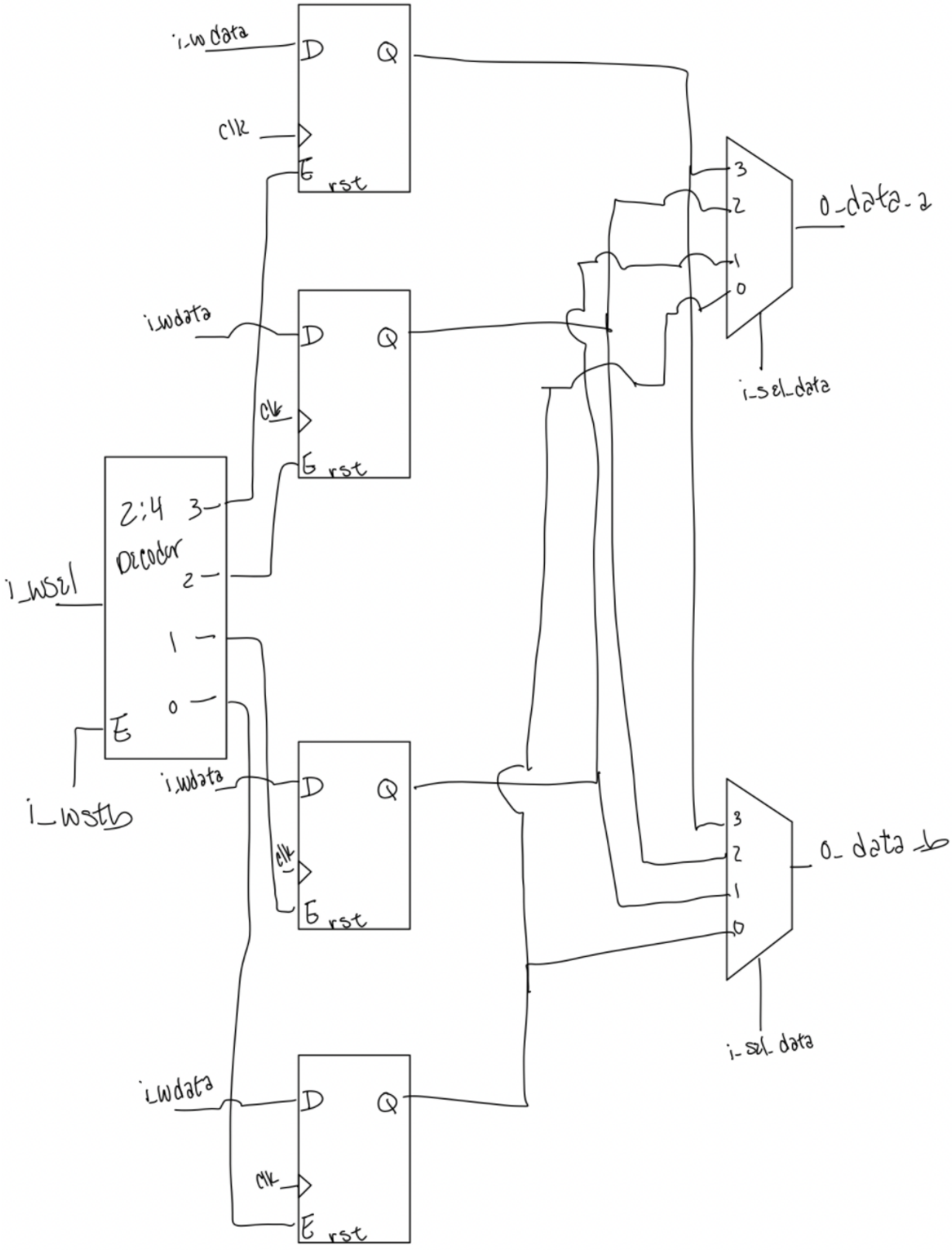


c. Register File

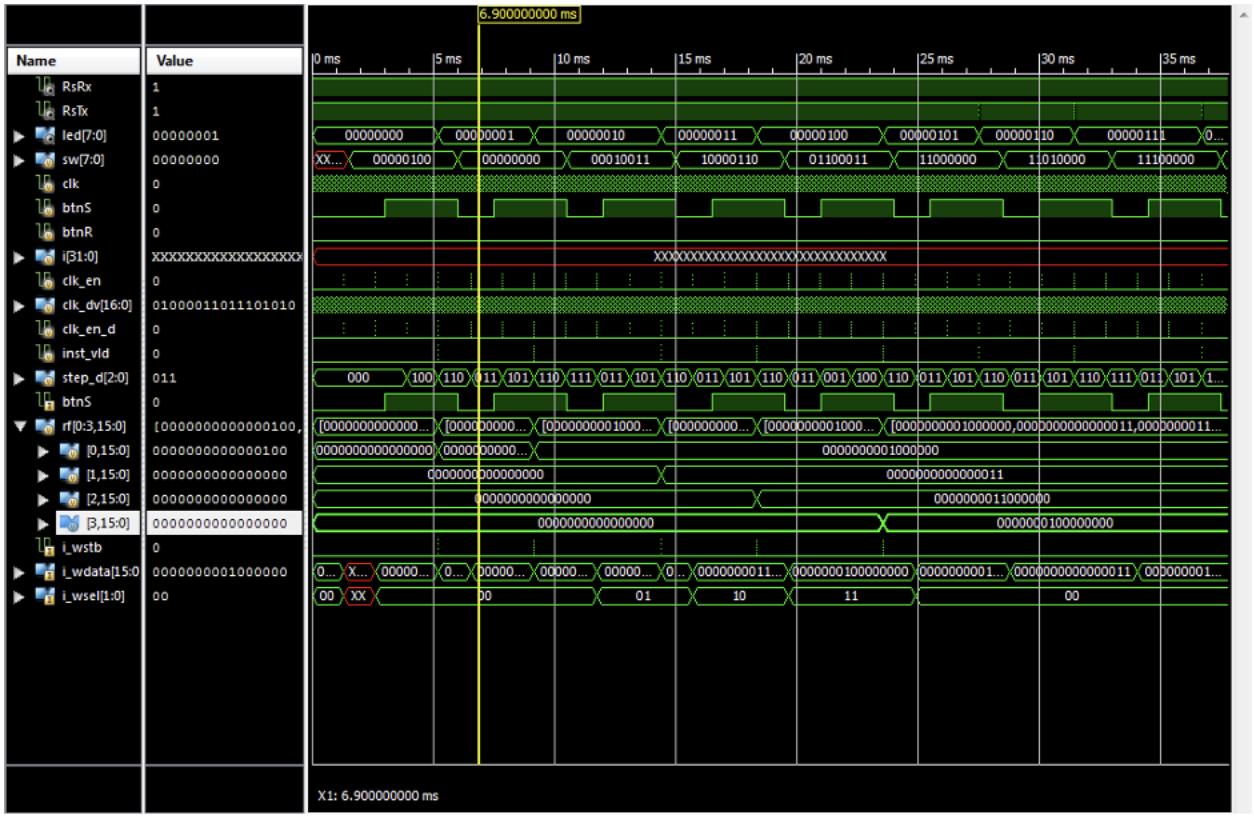
```
module seq_rf (/*AUTOARG*/
   // Outputs
   o_data_a, o_data_b,
   // Inputs
   i_sel_a, i_sel_b, i_wstb, i_wdata, i_wsel, clk, rst
   );
```

```
`include "seq_definitions.v"

   output [alu_width-1:0]   o_data_a;
   output [alu_width-1:0]   o_data_b;

   input [seq_rn_width-1:0] i_sel_a;
   input [seq_rn_width-1:0] i_sel_b;

   input                    i_wstb;
   input [alu_width-1:0]    i_wdata;
   input [seq_rn_width-1:0] i_wsel;

   input                    clk;
   input                    rst;

   reg [alu_width-1:0]      rf [0:seq_num_regs-1];
   integer                  i;

   always @ (posedge clk)
     if (rst)
       begin
          for (i=0;i<seq_num_regs;i=i+1)
            rf[i] <= 0;
       end
     else if (i_wstb)
       rf[i_wsel] <= i_wdata;

   assign o_data_a = rf[i_sel_a];
   assign o_data_b = rf[i_sel_b];

endmodule // seq_rf
```

Line 33, `rf[i_wsel] <= i_wdata;` is where a register is written with a non-zero value. This is sequential logic, because it's time-dependent, updating on the positive edge. Lines 35 and 35, `assign o_data_a = rf[i_sel_a]; assign o_data_b = rf[i_sel_b];` are where register values are read out from the register file. This is combinatorial logic, because it's independent of timing. If I were to manually implement this readout logic, I would use multiplexers, to select which register to read data out from. The diagram below shows the register file block.


i_wdata
D
Q
clk
E
rst
i_wdata
D
Q
clk
E
rst
i_wdata
D
Q
clk
E
rst
i_wdata
D
Q
clk
E
rst
2:4
Decoder
3
2
1
0
E
i_wsel
i_wstb
3
2
1
0
o_data_a
i_sel_data
3
2
1
0
o_data_b
i_sel_data

The image below is a waveform capture of the first time register 3 is written with a non-zero value.



4 Conclusion

In conclusion, our nicer uart output was implemented by simply creating a larger register, and checking the received bit's value. We were able to load in instructions from a sequencer file by reading in the file line by line and executing the binary instructions. Using instructions provided to us, we wrote binary instructions for calculating and printing the first ten fibonacci numbers.

A couple problems we encountered were involving the detection of the carriage return. Initially, we had written the carriage return character as '\r', but we had trouble detecting it, so we changed it to its ASCII representation. Additionally, our solution was not very general, as our register could only hold four bytes, so if we receive more than four bytes of input before a carriage return, then we will have lost some of the input.

Some suggestions I have for improving the lab would be to have more resources for students who have less experience with digital logic, as well as clearer instructions and expectations.