Lab Report 1

Russell Jew

Jeffrey Jiang

For the first part of our lab, we were only asked to implement a circuit on the FPGA that we were given. Most of the code was already given to us and we only had to follow the procedures The instances of the design processes of a circuit in our implementation of the Verilog circuit that we created is as described as follows.

1. We didn't have any design because the design of the FPGA was given to us.
2. Created a project and added the source code and implementation files to the project. The code that was provided was modular and hierarchically organized.
3. We used ISIM by clicking on "Simulate Behavioral Model." It brought us to a window where we could observe the waveform by selecting input waves. We added some signals to the waveform display and ran the simulation, watching the output in the console and waveform viewer. By running the simulation, we were able to determine whether our verilog code was producing the correct output.
4. Since we eliminated the bugs in the program, we used the synthesis tool in ISE to create a netlist that will be used to connect cells on our FPGA.
5. While ISE was synthesizing our code, there is also a portion that goes into the technology schematic which allows for device specific libraries to further enhance the netlist.
6. After we generate our list of connections (netlist), the "Implement Design" feature maps the components that we have in our netlist to a physical location on the board. This is the mapping component of the FPGA design.
7. Now that the board has all the components in place, and the connections that are needed, the final step is to route the components using a wire to connect components for communications.
8. Verilog generated the bitstream using the placing, routing, synthesizing, that it did that is readable by the FPGA. Then we could play around with the slide switches and the buttons to create our own "programs" by changing the values of the bits on the FPGA and outputted the values to the Putty console.

For the second part of the lab, we are only needed to understand the encoding scheme of the Verilog code that we were given. From the instructions provided in the lab manual, the translated "program" in binary is as follows.

```
00 00 01 00
00 00 00 00
00 01 00 11
10 00 01 10
01 10 00 11
```

```
11 00 00 00
11 01 00 00
11 10 00 00
11 11 00 00
```

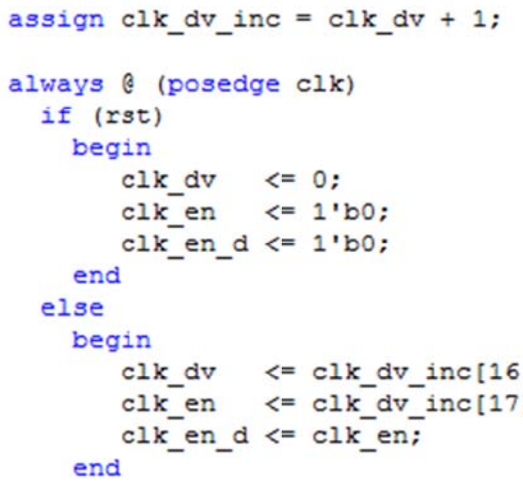
In the third part of this lab, we were instructed to display the first 10 Fibonacci numbers. The following "program" will print out all the Fibonacci numbers. This "program" pushes 1 to two registers and then successively adds two registers in a rotating fashion and prints it out after adding. This allows for us to reuse registers and continue rotating the registers.

PUSH R0 0x1	00000001
SEND R0	11000000
PUSH R1 0x1	00010001
SEND R1	11010000
ADD R0 R1 R2	01000110
SEND R2	11100000
ADD R1 R2 R3	01011011
SEND R3	11110000
ADD R2 R3 R0	01101100
SEND R0	11000000
ADD R3 R0 R1	01110001
SEND R1	11010000
ADD R0 R1 R2	01000110
SEND R2	11100000
ADD R1 R2 R3	01011011
SEND R3	11110000
ADD R2 R3 R0	01101100
SEND R0	11000000
ADD R3 R0 R1	01110001
SEND R1	11010000

On one hand, we can use the switches to compute and display all of these Fibonacci using these instructions. However, we will also translate this code directly into binary in Workshop 2 to run on the testbench. The binary that we used to match these instructions is shown beside each instruction that we use.

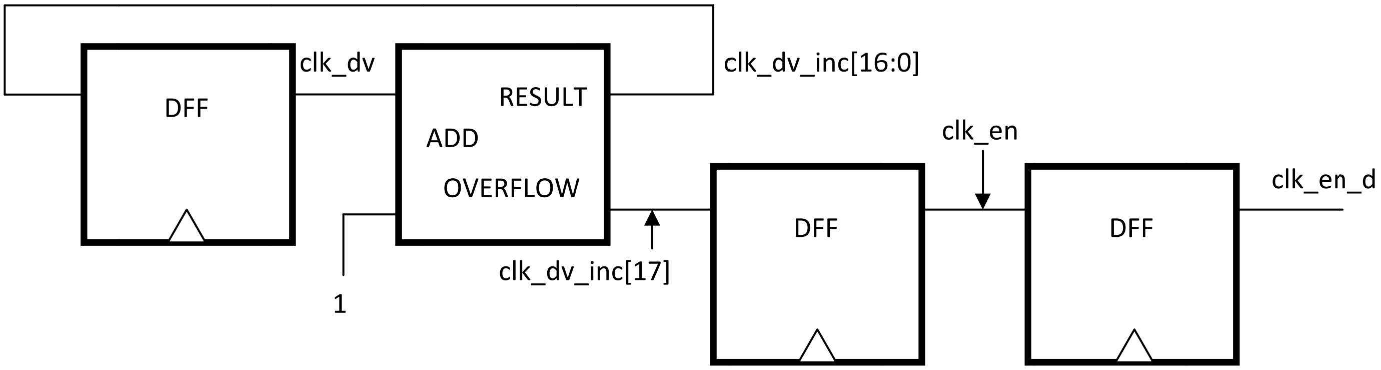
Workshop 1 deals with the analysis of various components within the Verilog code and familiarizing ourselves with the iSim tool in Verilog. This part of the lab is the largest chunk of the lab report and deals mainly with three variables used in the Verilog code, clock enable, instruction valid, and register files.

Clock Enable:



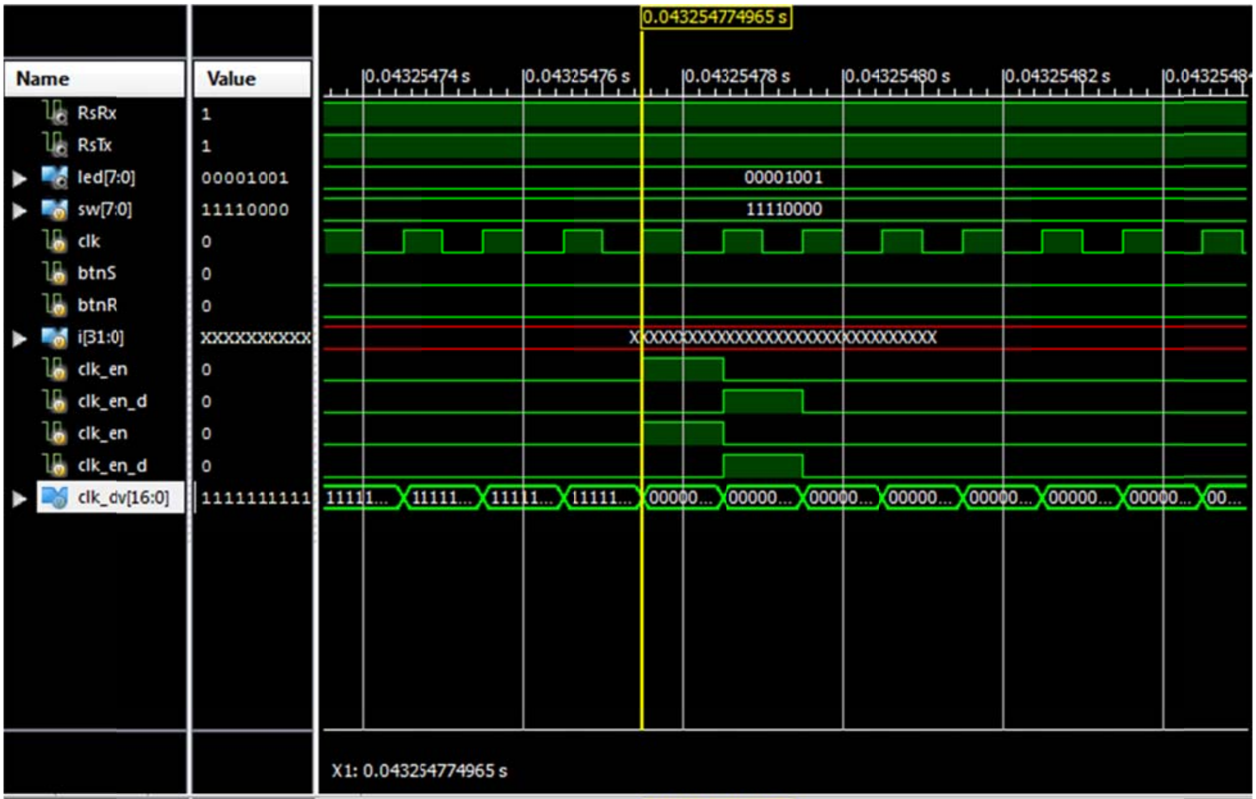
```
assign clk_dv_inc = clk_dv + 1;

always @ (posedge clk)
  if (rst)
    begin
       clk_dv   <= 0;
       clk_en   <= 1'b0;
       clk_en_d <= 1'b0;
    end
  else
    begin
       clk_dv   <= clk_dv_inc[16:0];
       clk_en   <= clk_dv_inc[17];
       clk_en_d <= clk_en;
    end
```

First of all, we deal with clock enable. There are two variables that relate to clock enable in the Verilog code: clk_en and clk_en_d. These two values are used essentially as a clock divider. The idea behind the code is that there is a variable called clk_dv that is incremented per clock tick. This means that clk_en is only high for a single clock tick, when clk_dv overflows to take on the value of 2^{17} (clk_dv will have a value of 0). Essentially, this means that clk_en is essentially another repetitive clock cycle that occurs every 2^{17} clock cycles. In this case, clk_en_d is just a delayed version of the clk_en signal. The reason for this is for other components in the circuit, but the purpose of this variable is to provide an enable that is clocked at a specific rate, dividing the actual clock signal by 2^{17}. The following pictures show two consecutive instances where these values become nonzero. The resulting period is around 1.31 ms, which is for most purposes still a quick enough for any button pressing purposes that we have. The circuit for this code is the following:



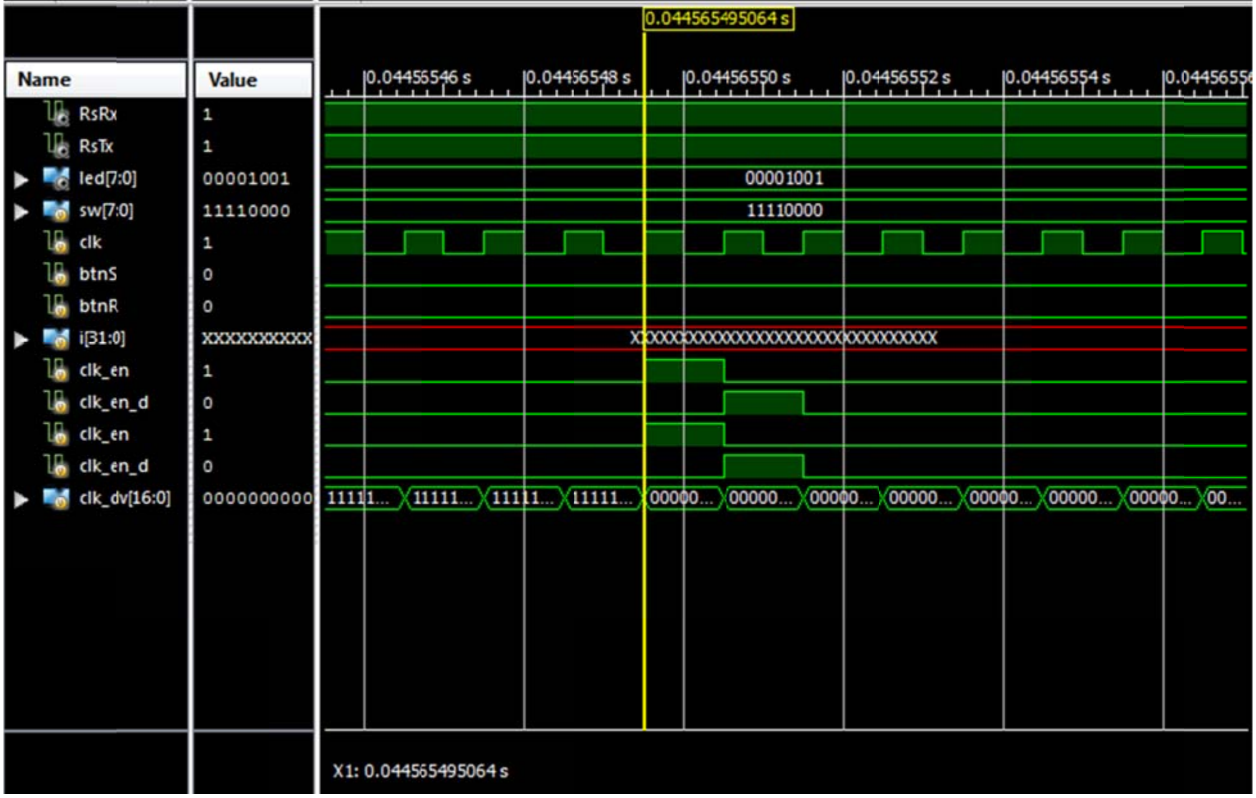
The clock signal is built in at every spot where the triangle is in the DFF. Here, we see that the signals are delayed in the way that it is.

Clock enable pictures



Name	Value
RsRx	1
RsTx	1
led[7:0]	00001001
sw[7:0]	11110000
clk	0
btnS	0
btnR	0
i[31:0]	XXXXXXXXXX
clk_en	0
clk_en_d	0
clk_en	0
clk_en_d	0
clk_dv[16:0]	1111111111

X1: 0.043254774965 s

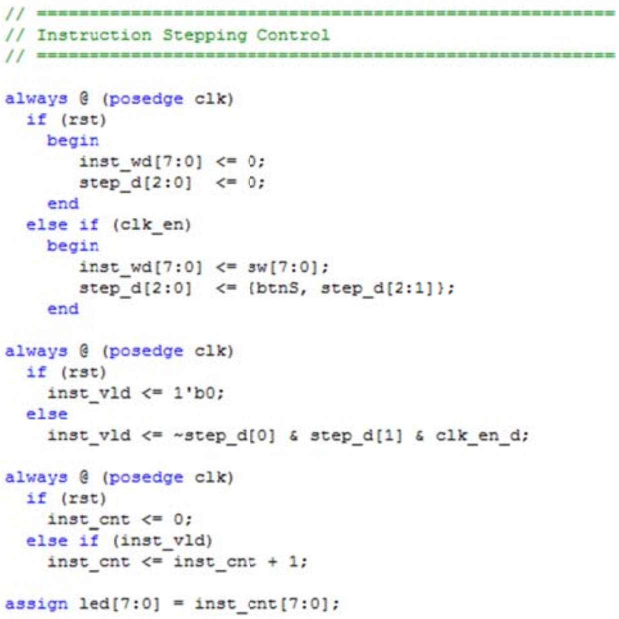


Name	Value
RsRx	1
RsTx	1
led[7:0]	00001001
sw[7:0]	11110000
clk	1
btnS	0
btnR	0
i[31:0]	XXXXXXXXXX
clk_en	1
clk_en_d	0
clk_en	1
clk_en_d	0
clk_dv[16:0]	0000000000

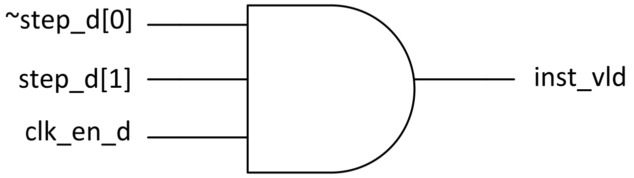
X1: 0.044565495064 s

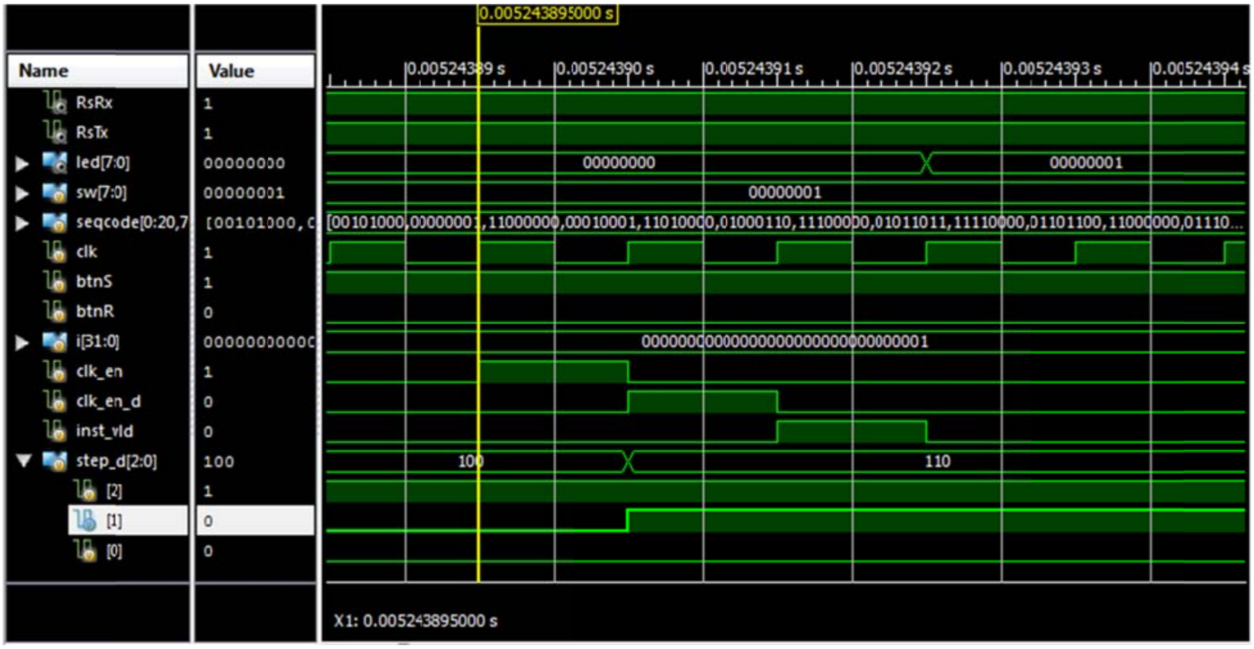
Instruction Valid:

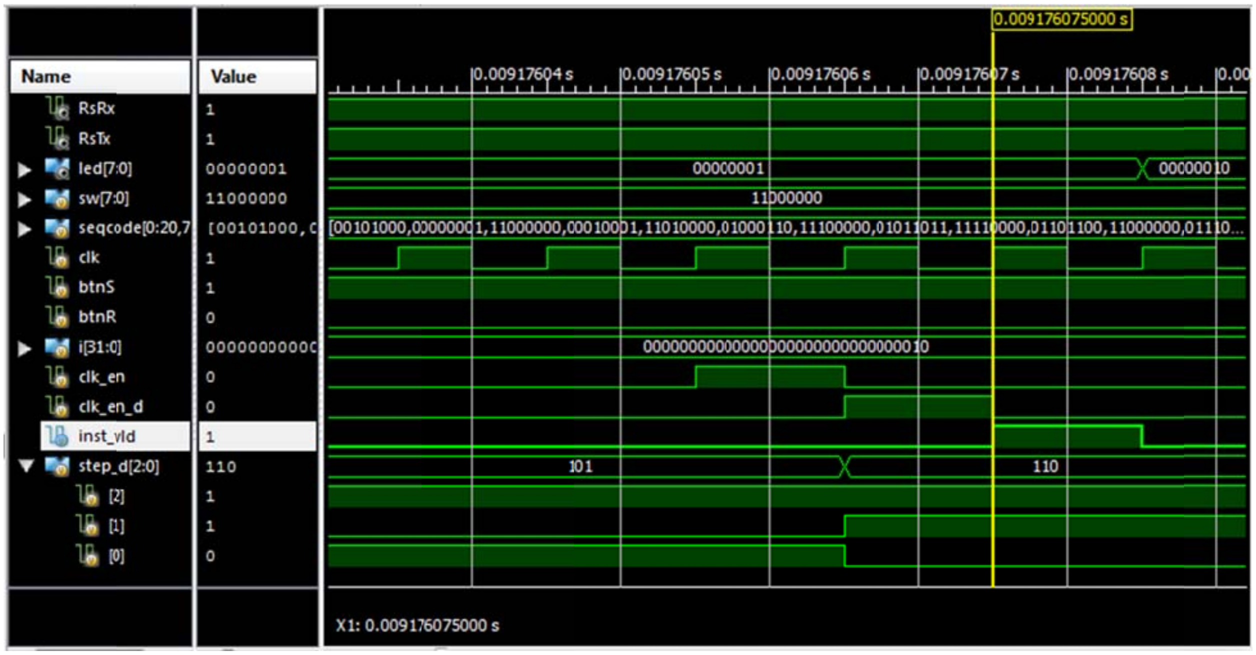
Instruction valid is essentially the signal that allows for instructions to run and creates the instruction incrementing implementation. Notice that clk_en_d becomes the positive edge of the new "clock" signal. We want to use clk_en_d because we do not want to use such a fast clock for the purposes of this part of the lab. We use clk_en_d because this matches up with the step_d[2] change (since the clk_en signal does not match up with the change of the step_d signal due to the fact that step_d changes the clock cycle after clk_en). Just as a note, step_d represents the state of the button in the last three clk_en cycles. Using the Fibonacci code running (not the sample code), the first two instances of inst_vld being nonzero are shown in the two pictures. This shows that the time interval is 3.93 ms. From the previous analysis, this is a period of about two clk_en_d cycles. The schematic for instruction valid is more simplistic. It is just a logical combination of the three wires we already know.



```
// ==========================================================
// Instruction Stepping Control
// ==========================================================

always @ (posedge clk)
  if (rst)
    begin
       inst_wd[7:0] <= 0;
       step_d[2:0]  <= 0;
    end
  else if (clk_en)
    begin
       inst_wd[7:0] <= sw[7:0];
       step_d[2:0]  <= {btnS, step_d[2:1]};
    end

always @ (posedge clk)
  if (rst)
    inst_vld <= 1'b0;
  else
    inst_vld <= ~step_d[0] & step_d[1] & clk_en_d;

always @ (posedge clk)
  if (rst)
    inst_cnt <= 0;
  else if (inst_vld)
    inst_cnt <= inst_cnt + 1;

assign led[7:0] = inst_cnt[7:0];
```

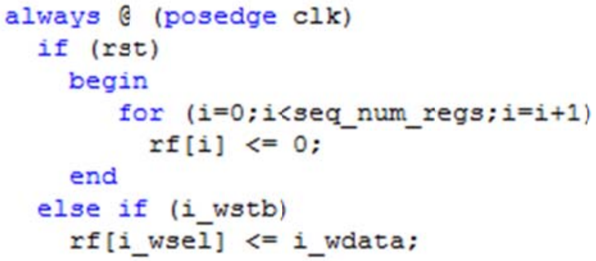






Instruction Valid Pictures

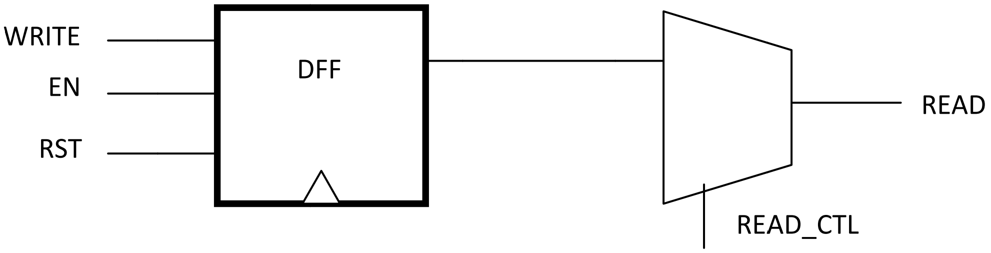
Register File:

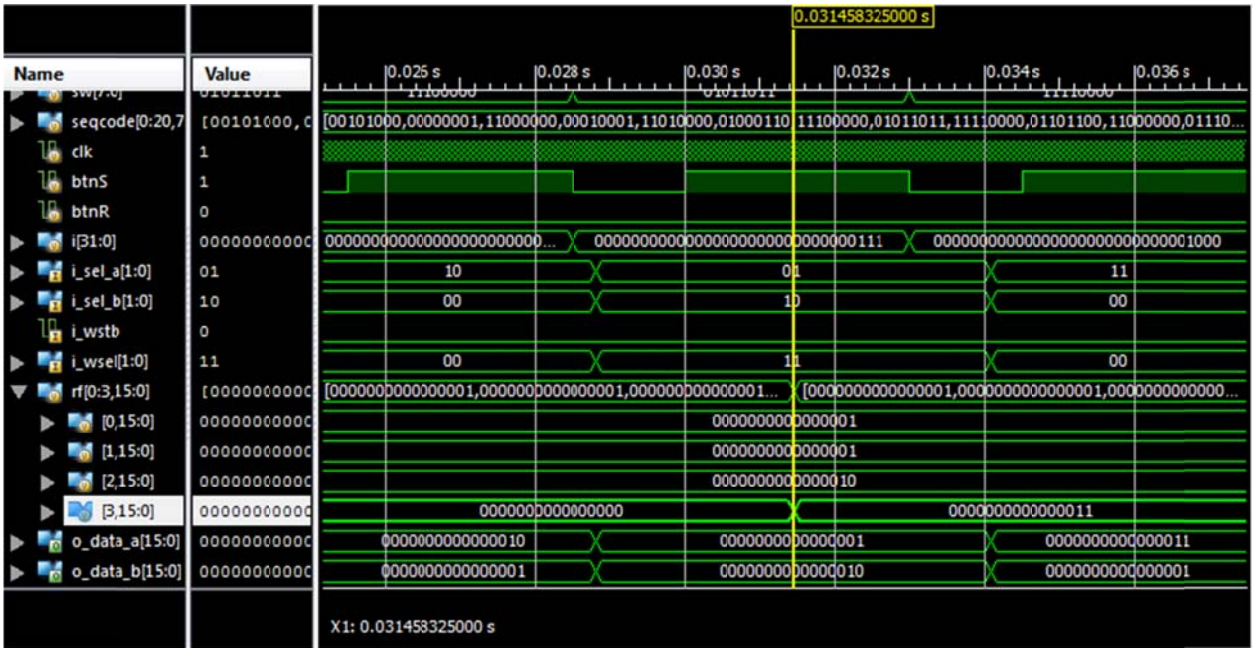


```
always @ (posedge clk)
  if (rst)
    begin
      for (i=0;i<seq_num_regs;i=i+1)
        rf[i] <= 0;
    end
  else if (i_wstb)
    rf[i_wsel] <= i_wdata;
```

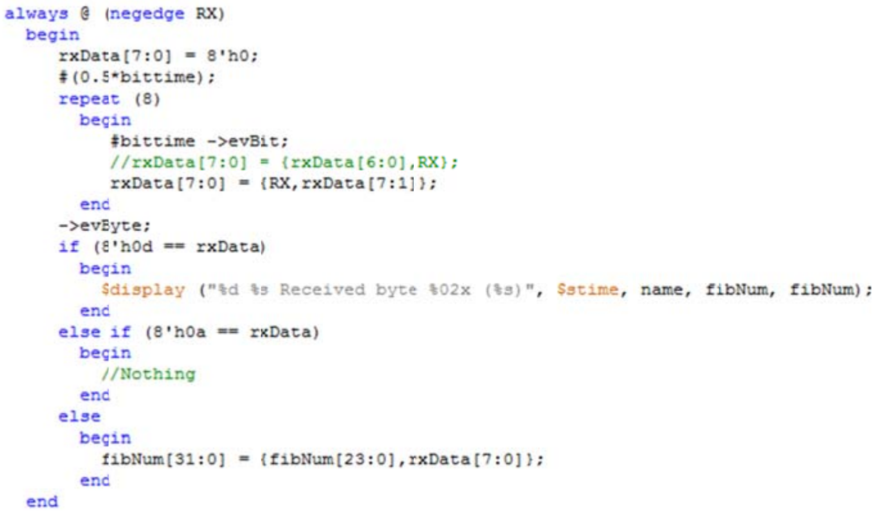
The line of code dealing with the setting of register values to nonzero values is the final line in the picture to the right. The entire picture deals with the setting of these registers. This is sequential logic because at each clock cycle there may be a different value set to the register file, but there cannot be any change in the register files in between clock cycles. The setting of the register files itself, however, is combinational logic and shows the connect between the two forms of logic.

The lines of code dealing with writing out the register files are the following two lines, where the query register is written out, as well as the instruction code in the test bench. If I were to manually create readout logic, a possible implementation would be to use a MUX, connected to different DFFs, which contains the information, selected using a select pin. The circuit represents a single register.



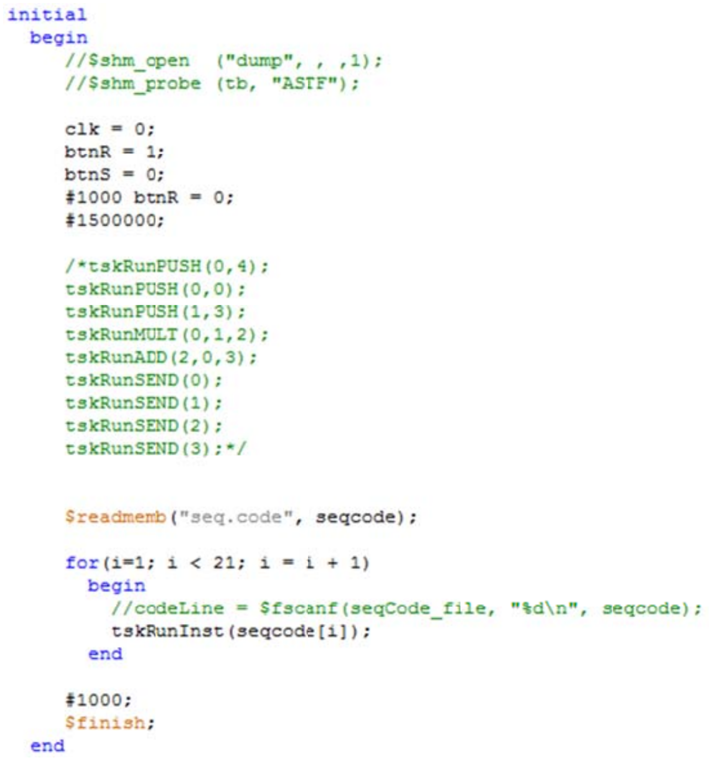


For workshop 2, we essentially want to make the iSim component more user-friendly and able to read instructions from computer file input, instead of mechanical input. This required two steps, making the UART more usable, and allowing for file input. This is show in the picture to the right.



```
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;
     if (8'h0d == rxData)
       begin
         $display ("%d %s Received byte %02x (%s)", $stime, name, fibNum, fibNum);
       end
     else if (8'h0a == rxData)
       begin
         //Nothing
       end
     else
       begin
         fibNum[31:0] = {fibNum[23:0],rxData[7:0]};
       end
  end
```

In tb.v, the only real instruction that sends out signals to the UUT is the tskRunInst() function, which accepts an 8 bit value as its input. Every other user task that is created are just for user-friendly usage, such as the tskRunPUSH() or the tskRunADD() methods (which are commented out in the read file code. These are defined just after the initial block in the tb.v code and most of the other tasks are just constructing the 8 bit signal to pass into tskRunInst(). Therefore, based on our text file version of this, we are able to just send our text file instructions straight into the task that was defined and run the code. This is, therefore, able to successfully run our Fibonacci demo completely on the computer.



```
initial
  begin
     //$shm_open  ("dump", , ,1);
     //$shm_probe (tb, "ASTF");

     clk = 0;
     btnR = 1;
     btnS = 0;
     #1000 btnR = 0;
     #1500000;

     /*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);*/


     $readmemb("seq.code", seqcode);

     for(i=1; i < 21; i = i + 1)
       begin
         //codeLine = $fscanf(seqCode_file, "%d\n", seqcode);
         tskRunInst(seqcode[i]);
       end

     #1000;
     $finish;
  end
```