CS M152A: Introductory Digital Design Laboratory

Lab #1

Sequencer

Name: Andrew Juarez, Atibhav Mittal

UID: 504-572-352, 804-598-987

Lab Section: 4

Due Date: November 2, 2017

1.1: Introduction and Requirement

The aim of this lab was to build a sequencer. A sequencer is a simpler version of the processor that we have installed in most electronics today. The sequencer can perform 4 different tasks. It can push a value to a register (memory), add values in two registers, multiply values in two registers and display the value of a register. Each of these tasks has a different binary encoded operation code that tells the FPGA what operation to perform with the current registers. A base code was provided to us and the aim of the lab was to modify this to perform certain tasks.

The first task was to implement the multiplication operation. The next step involved buffering the output to Putty and displaying it when a carriage return is received from the Verilog code. The output operations are only to output the value of a specific register to the console (Putty if running on FPGA), or the Xilinx console if we're simulating the code. Our third task was to display what register we were outputting in addition to the value of the register since it provides some more meaningful debugging data. Up till now, while simulating we had been manually calling functions to perform operations on the sequencer. The next task was to get it to read operation codes from a text file. The final task was to generate the first 10 numbers of the Fibonacci series using operation codes and feeding that into the sequencer.

1.2: Design Description

The overall design contains a top level module that instantiates the register file, ALU, control logic, and UART controller and the relationships are shown in the following diagram.

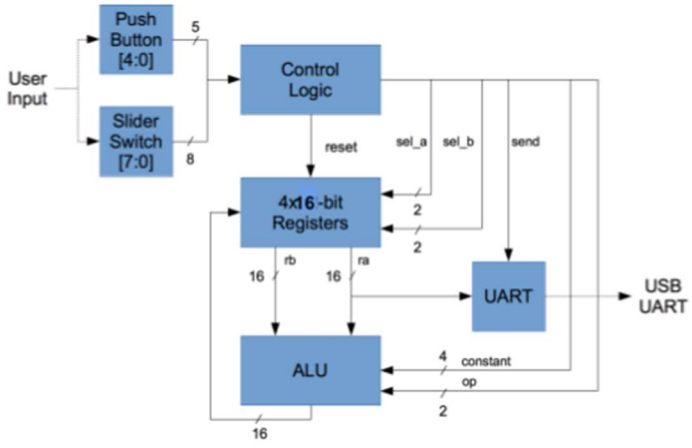


Figure 1: Project Diagram: The inputs to the design are the push buttons, slide switches, and the output will be to PuTTY and LEDS. However, the LEDS are not shown in this diagram. [1]

The topmost sequencer module, seq.v, creates an instance of the alu.v module and passes in the inputs to this alu module. It then outputs this to the register file if it is a multiplication, addition or push operation through the seq_rf module. The seq_rf module has a 1-bit signal seq_valid_in, that is set to true any time the input from to the alu is valid.

Missing Multiply Operation

The multiplication module was based off the addition module that was already implemented in the files given. The way the addition module works is by getting two 16-bit inputs and then doing a simple addition operation and then outputting it. The output is set to valid if the input is considered valid. Similarly, the multiplication module inputs two 16 bit numbers, does a simple multiplication using the Verilog * operator, and outputs this result truncated to a 16-bit number. The output is considered valid if the input is valid. For this task, a change had to be made in seq.v though, involving the seq_valid_in signal. Any time that the operation code is a push, add or multiplication, the seq_valid_in signal needs to be set to true so that the input can be passed to the alu. The only change we made to this module is to add the code that would set this signal to true for multiplication operations.

Nicer UART Output

The goal of this task was to suppress the per-byte address output. Instead, we needed to display four bytes per line, where the four bytes represent the value of the register being read. To accomplish this task, the code from model_uart was studied.

```
always @ (negedge RX)
    begin
      rxData[7:0] = 8'h0;
      #(0.5*bittime);
      repeat (8)
        begin
          #bittime ->evBit;
    rxData[7:0] = {RX,rxData[7:1]};
        end
      ->evByte;
$display ("%d %s Received byte %02x (%s)", $stime, name, rxData, rxData);
```

The repeat(8) loop reads 1 byte at a time. Then, the 1-byte value is displayed. Now to display four bytes at a time, a 32-bit register was added to store the four-byte value. After each repeat loop, 1 byte was added to the 32-bit register. After four repeats, the next characters received were the newline character and the carriage-return character. During these characters, the bytes were not added to the 32-bit register, preserving the correct register value. Once the newline character was received, the value of the register was displayed.

Even Nicer UART Output

The aim of this task was to add the register number to the output when a send operation is received. To do this, we modified the finite state machine to have 3 additional states, to send 3 characters R, the register number, and a colon. The uart_top.v module uses a Finite State Machine machine for this task has 7 different states. These states distributed as 1 idle state, 4 states for receiving output values, and 2 states for carriage return and new line characters. Furthermore, the register number is passed in as input from the operation code as a parameter to the uart_top module.

The FSM works as follows. If the current state of the FSM is the idle state and it receives an input, the current state is changed to stSendR, which adds the letter “R” to the FIFO output. The next state is to send the register as an ASCII value to the FIFO. The bit value of the register,

received as an input, is converted to its ASCII representation using the fnNib2ASCII function. After sending this register value, we switch to a state where we send the colon character to the FIFO. Finally, we need to output the register value, which follows the same steps as described in the above section. The state is changed to stNib1, which is the state to read the first 4 bits. The input is then stored in tx_data. These 4 bits are converted to ASCII hex-representation inside an always block using the given function, fnNib2ASCII and sent to the FIFO output. This process is repeated for the next 3 4-bit inputs. The states are changed continuously to different Nib states. After receiving these 16 bits of input, the FSM changes to the new line state (stNL). A new-line is appended to the FIFO, and the state is again incremented to the stCR; the carriage return state. In this state, a carriage return character is appended to the FIFO. The FSM waits until the FIFO is not completely full, at which point it goes back to the idle state and the entire process is repeated on receiving the next SEND operation.

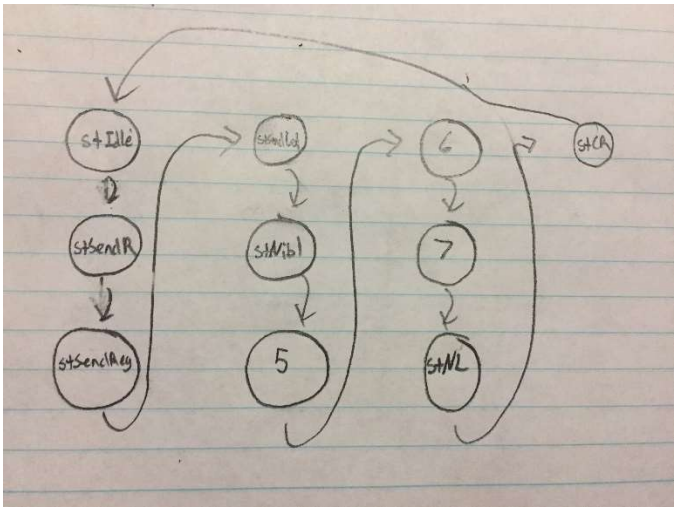


Figure 2: Finite State Machine: The diagram shows the finite state machine implemented in the uart_top module. The added states to the previous state machine are stSendR, stSendReg, and stSendCol.

An Easier Way to Load Sequencer Program

Up until now, during simulation, the sequencer was running static instructions. These operations were being executed by calling the specific functions tskRunPUSH, tskRunADD, tskRunMULT and tskRunSEND. These instructions modify the value of the sw register and call the tskRunInst function. For this task, we added code that reads the instructions from a text file and executes the instructions by setting the sw register automatically, and directly calls the tskRunInst instead of going through specific functions. This is done using the Verilog readmemb function, which reads each input line by line. The number of instructions is parsed on the first line, and then the tskRunInst function is repeatedly called.

Lab Manual Questions:

1. The instructions are sent in the task tskRunInst. In this task, sw is given the value of inst. The instantiated module uut gets the value of sw because it is an input port of it.

2. If it is a push instruction, tskRunPUSH formulates the instruction
and invokes tskRunInst.

If it is an add instruction, tskRunADD formulates the instruction and invokes tskRunInst.
If it is a mult instruction, tskRunMULT formulates the instruction and invokes tskRunInst
If it is a send instruction, tskRunSEND formulates the instruction and invokes tskRunInst.

Fibonacci Numbers

The Fibonacci sequence is a recursive sequence that starts with the numbers 0 and 1 and each of the following numbers is a sum of the previous two numbers. Out of the four instructions the sequencer provides to the user, the PUSH, ADD, and SEND instructions were used. Initially, one needs to push the value of 0 into register 0 and send the value to the display. Next, one needs to push the value of 1 into register 1 and the value to the display. Now that the two initial values are loaded into registers 0 and 1, the two registers need to be added and the result needs to be displayed. To accomplish this, register 0 and register 1 were added and the result was stored in register 0. The value of register 0 was displayed. Then, register 0 and register 1 were added and the result was stored in register 1. The value of register 1 was displayed. This continued until the 10^{th} number in the Fibonacci Sequence was displayed. After coming up with the algorithm, the instructions were converted to their bit representations and written in a text file.

1.3: Simulation and Documentation

Missing Multiply Operation

To test the implementation of the multiply operation, the test bench given to us was used. The expectation was that the output should be the same as the output from the warm up task. The following snipped it the output received after implementing the multiplication operation.

```
 27534695 UART0 Received byte 30 (0)
 27545715 UART0 Received byte 30 (0)
 27556735 UART0 Received byte 34 (4)
 27567755 UART0 Received byte 30 (0)
 27578775 UART0 Received byte 0a (
)
 27589795 UART0 Received byte 0d (
)
 28501000 ... Running instruction       11010000
 31458325 ... instruction       11010000 executed
 31458325 ... led output changed to       00000111
 31466855 UART0 Received byte 30 (0)
 31477875 UART0 Received byte 30 (0)
 31488895 UART0 Received byte 30 (0)
 31499915 UART0 Received byte 33 (3)
 31510935 UART0 Received byte 0a (
)
 31521955 UART0 Received byte 0d (
)
 33001000 ... Running instruction       11100000
 36701205 ... instruction       11100000 executed
 36701205 ... led output changed to       00001000
 36709735 UART0 Received byte 30 (0)
 36720755 UART0 Received byte 30 (0)
 36731775 UART0 Received byte 43 (C)
 36742795 UART0 Received byte 30 (0)
 36753815 UART0 Received byte 0a (
)
 36764835 UART0 Received byte 0d (
)
 37501000 ... Running instruction       11110000
 40633365 ... instruction       11110000 executed
 40633365 ... led output changed to       00001001
 40641895 UART0 Received byte 30 (0)
 40652915 UART0 Received byte 31 (1)
 40663935 UART0 Received byte 30 (0)
 40674955 UART0 Received byte 30 (0)
 40685975 UART0 Received byte 0a (
)
 40696995 UART0 Received byte 0d (
)
```

Figure 3: Output for Simulation of Warm-Up Task with Implementation of Multiplication Operation

Nicer UART Output and Even Nicer UART Output:

To test the implementation of Nicer UART Output, the same test bench was used. However, the output was expected to have a different format. The format should have four bytes on each line.

To test the implementation of Even Nicer UART Output, the given test bench was used again. In this task, the format should be the ASCII character 'R', the register number, the ASCII character ':', then the four bytes representing the register value. After running the test bench, the following output was given in iSim.

```
R0:0040
  28501000 ... Running instruction       11010000
  31458325 ... instruction       11010000 executed
  31458325 ... led output changed to       00000111
R1:0003
  33001000 ... Running instruction       11100000
  36701205 ... instruction       11100000 executed
  36701205 ... led output changed to       00001000
R2:00C0
  37501000 ... Running instruction       11110000
  40633365 ... instruction       11110000 executed
  40633365 ... led output changed to       00001001
R3:0100
```

Figure 4: Output for Simulation of Nicer UART Output/Even Nicer UART Output

The figure shows the output for Even Nicer UART output. The Nicer UART Output had the same result; however, it did not include the 'R' character, the register number, and the ':' character.

.

An Easier Way to Load Sequencer Program

To test the implementation of this task, the test bench was run with the expectation of displaying the same exact output as the Even Nicer UART Output. We added a text file that contained the instructions. The output did not change after using readmemb function to read from the text file. Hence, the sequencer was reading and executing the instructions correctly.

Fibonacci Numbers

To test the implementation of Fibonacci numbers, iSim was used to display the output. The expected output was the following sequence of numbers: 0, 1, 1, 2, 3, 5, 8, D, and 15 in hexadecimal. The output from the the simulation is shown in the figure below.

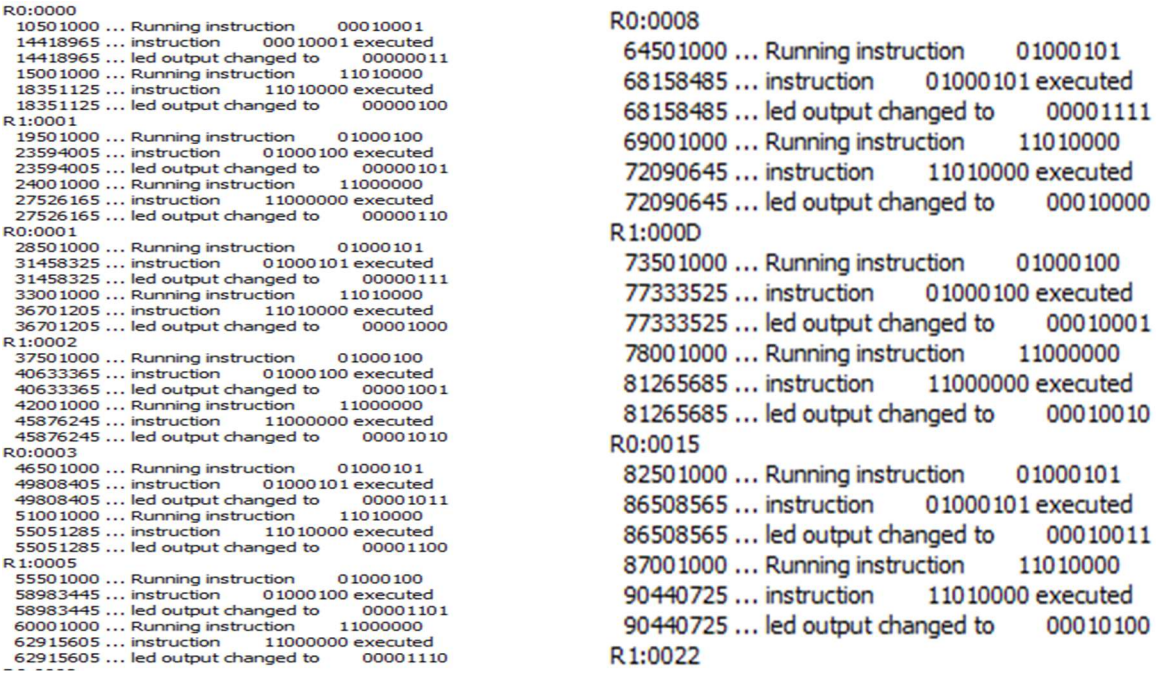


Figure 5: Output for Fibonacci Numbers Simulation

1.4: Conclusion

The aim of this lab was to reverse engineer Verilog code to understand what the given code was doing and incorporate some additional features into this code. The tasks included adding a multiplication operation to the ALU, buffering and displaying the output in a nicer format, and reading instructions from a text-file instead of using static instructions. Each of the tasks involved understanding what the different modules were doing and writing code that would add these features.

One challenge we encountered was with implementing the multiplication operation. Initially we just modified the seq_mult module, which produced incorrect results. On examining waveforms, we found that the input valid parameter to the ALU was not being set to true. To correct this, we modified seq.v so that the input was considered valid, and was actually written to the register file if we had a multiplication operation. We faced a couple of challenges with "even nicer output." It took us a while to come up with a way to send the register number. To send the register number, we decoded the operation code and added it as a parameter into the uart_top module. Additionally, we were displaying the characters "R" and ":" through model_uart. This was giving the correct result when we simulated the design but didn't work when we used PuTTY. To correct this, we added more states to the FSM in model_uart and outputted the characters "R" and ":" through the FIFO. This method worked for both the simulation and implementation.

Overall, this lab was challenging compared to the previous one. It was good practice of reverse engineering code, a practice that is definitely required for working in the industry.

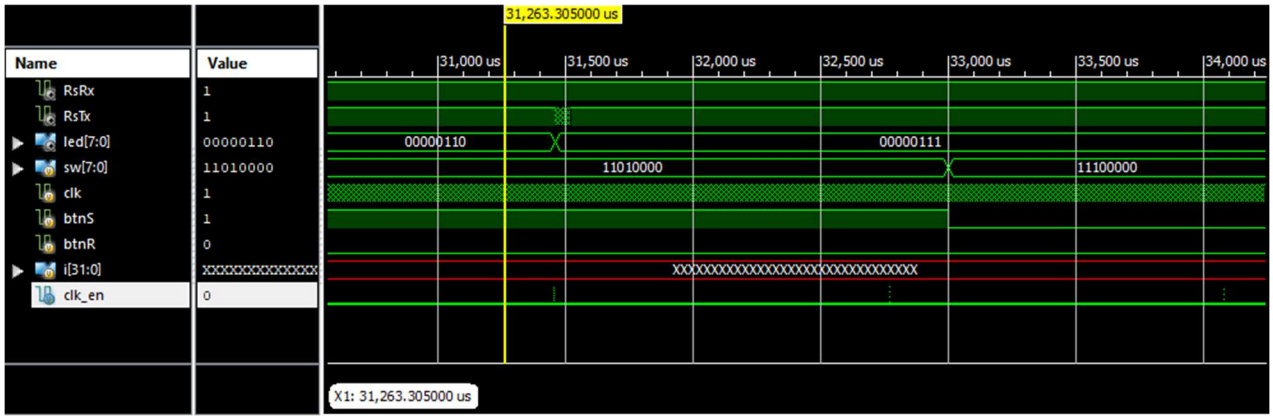
1.5: Appendix

Clock Dividers

1.

Value 1: 32,769.025 us
Value 2: 34,079.745 us

Time Period = Value 2 – Value 1 = 1,310.72 us



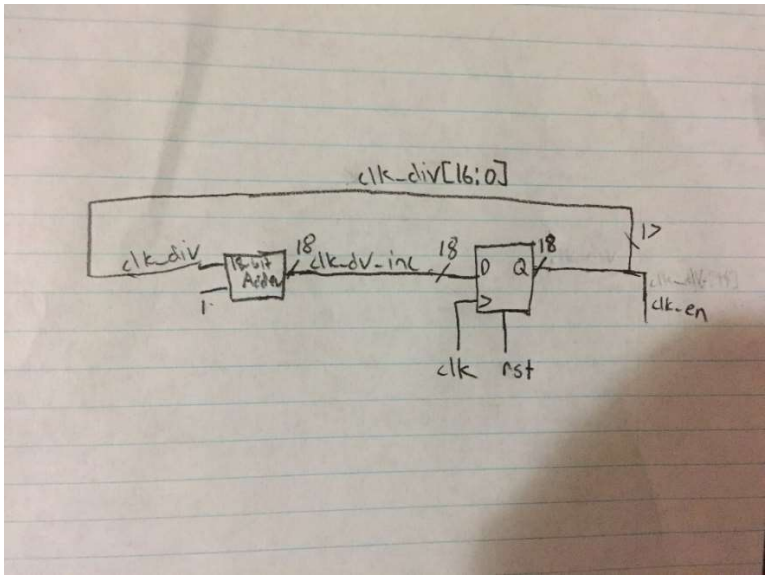
2.

Signal High start at time: 31,458.295 us
Signal High end at time: 31,458.305 us

T = 0.010 us
Duty Cycle = (0.01 / 1310.72) * 100

3. The value of clk_dv is 0 when clk_en is high

4.

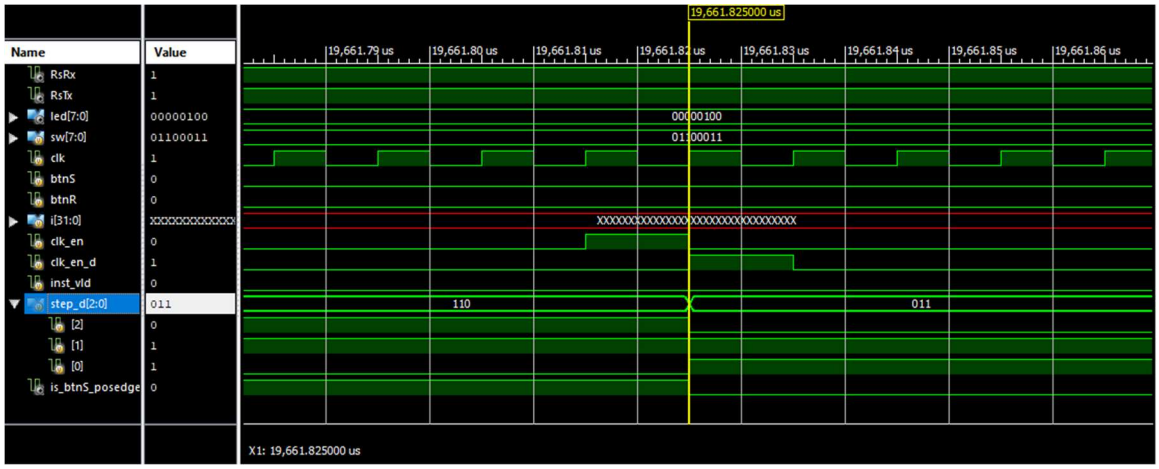


Debouncing

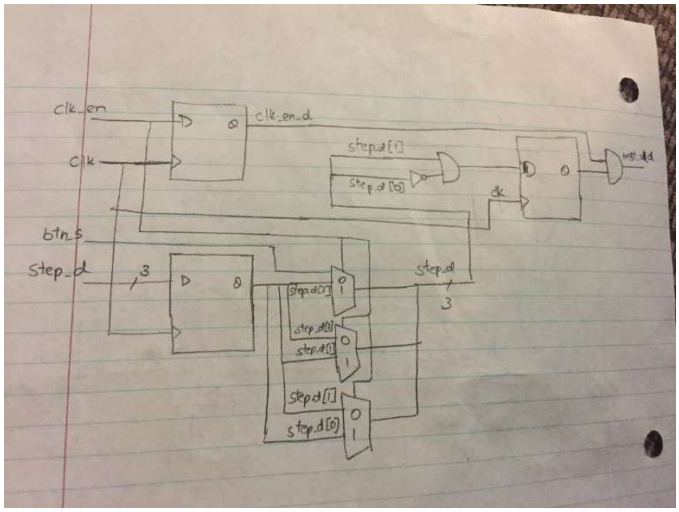
1. If we use clk_en instead of clk_en_d, the inst_valid signal will be sampled one clock-cycle too early giving an incorrect result. This is due to the fact that we want to make sure that there is a positive transition of btnS. We avoid metastability using edge detection. We check the step_d signal, which clocks in the transitions of button S and detects the positive edge.

2. If we change clk_en to be the 17th bit instead of the 18th bit of clk_div_inc, then our clock_en will be functioning at a higher frequency. Due to this, we are sampling our signals at a higher frequency, making the sequencer more susceptible to noise. Although the sequencer is more susceptible to noise, upon running the simulation we found that it still gave the correct output.

3.



4.



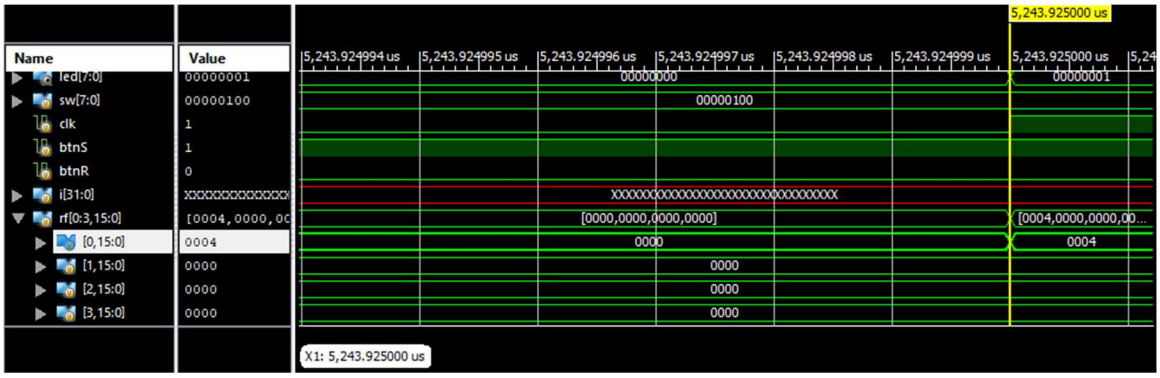
Register File

1. A non-zero value is written to the register at line 33 of seq_rf.v
   The line of code is: rf[i_wsel] <= i_wdata;

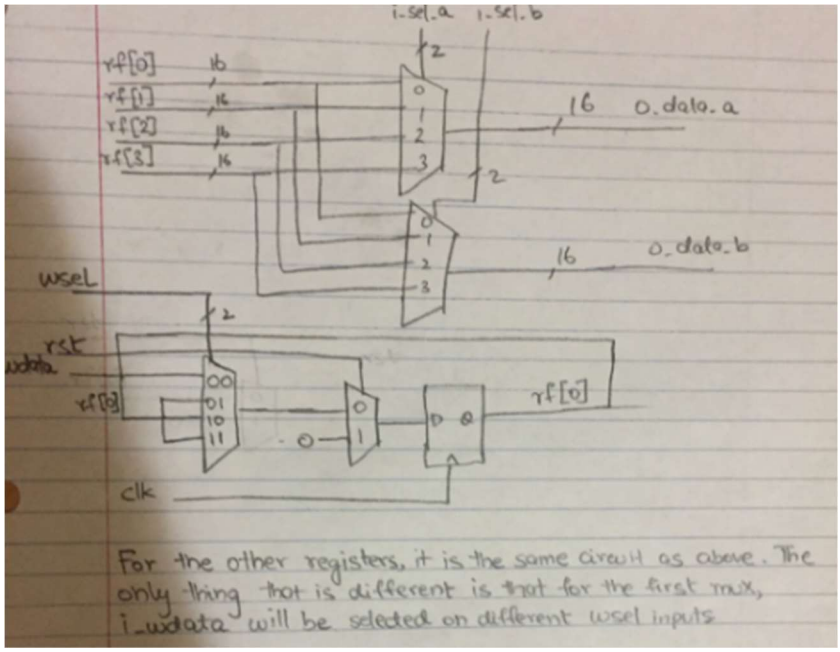
   This is sequential logic since its using a clocked always block with non-blocking assignments.

2. The registers are read at line 35 and 36 of seq_rf.v
   The lines of code that do this are:
   assign o_data_a = rf[i_sel_a];
   assign o_data_b = rf[i_sel_b];
   Since the code is using assign statements, it is combinational logic. If we were manually implementing this logic, we'd use multiplexers with i_sel_a or i_sel_b and rf array as inputs.

3.



4.



Bibliography

1. Potkonjak, M. Computer Science M152A: Introductory Digital Design Laboratory Lab Manual. (Univ. California Los Angeles, Los Angeles, California).