CS M152A - Lab 2

Jair Hinojosa, Kyle Romero
904771873, 204747283
Section 5
TA: Fan Hin Hung
Date Performed: 1/30

1.0 - Introduction

The purpose of this lab was to implement various components of a simple UART sequencer. During the first part of this lab, we altered the provided source code to add five new features to a provided module. During the second part of this lab, we answered theoretical questions, drew implementation diagrams, and captured simulation waveforms regarding topics relevant to this lab and the design of digital systems.

1.1 - Background

A sequencer is a programmable logic device that can be used to manipulate and store sequences of information. The provided source code is an adder/multiplier sequencer that interprets instructions and performs operations on constants stored in four 16-bit registers. The instructions are entered by using the switches and buttons on the Nexys 3 FPGA board. The contents of a register can be displayed on a desktop monitor using the board's UART output.

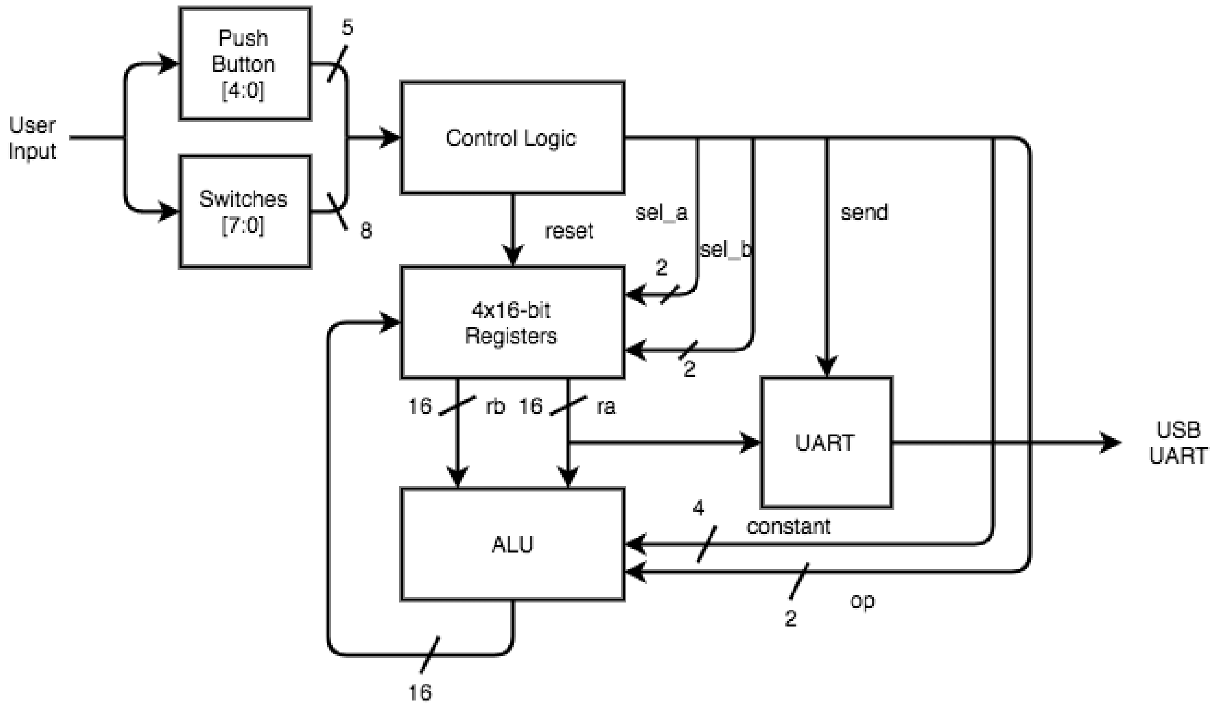


Figure 1.1.1 - The high-level modules that make up the provided sequencer.

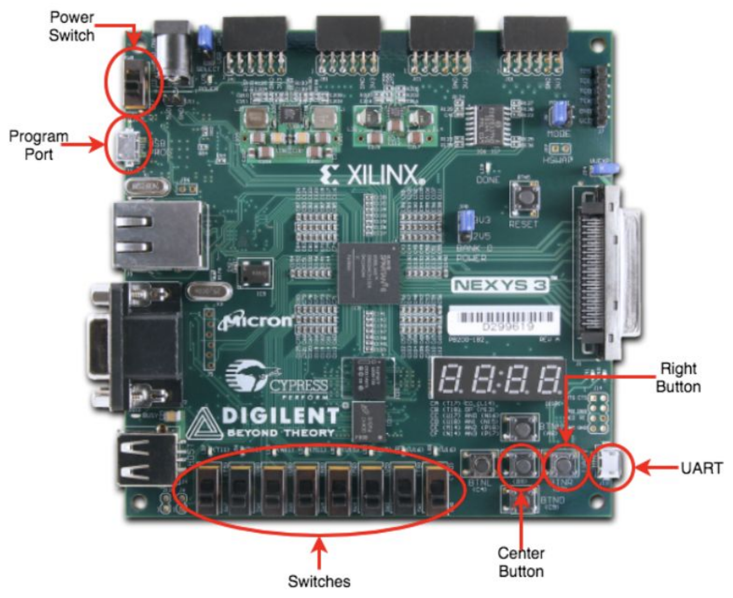


Figure 1.1.2 - The components of the Nexys 3 Board used in this lab.

The instructions designate an operation to be performed on one or multiple registers or binary constants. Instructions are entered using the board's switches and are executed by pressing the center button on the board. The values stored in each register can be reset to zero using the right button. Finally, the LEDs above the switches display a binary counter of the number of instructions executed since the last reset.

00 - PUSH	2-bit value X	4-bit Constant

Figure 1.1.3 - Each PUSH instruction begins with the op code 00. The operation left-shifts register X by 4 bits and pushes the new constant into the four least significant bits.

01 - ADD	2-bit value X	2-bit value Y	2-bit value Z

Figure 1.1.4 - Each ADD instruction begins with the op code 01. The operation performs unsigned integer addition on register X and register Y and stores the result in register Z.

10 - MULT	2-bit value X	2-bit value Y	2-bit value Z

Figure 1.1.5 - Each MULT instruction begins with the op code 10. The operation performs unsigned integer multiplication on register X and register Y and stores the result in register Z.

11 - SEND	2-bit value X	Don't Care Bits

Figure 1.1.6 - Each SEND instruction begins with the op code 11. The 16-bit hexadecimal value stored in register X is converted to ASCII, and sent to the UART for display.

1.2 - Design Requirements

The first part of this lab contains five components:

- Implementing the multiply operation
- Nicer UART output during simulation
- Nicer UART output during implementation
- Allowing users to input instructions via a file
- Executing instructions for the Fibonacci sequence from a file

Part 1: Missing Multiply Operation

A properly implemented multiply operation will successfully perform unsigned integer multiplication on the values stored in the registers specified in each instruction. If the multiplication results in overflow, we should only store the sixteen least significant bits. To implement this operation, the specification suggests looking at the addition operation for inspiration. Finally, we note that the completion of this task is marked by the successful execution of an instruction sequence provided in the testbench.

Part 2: Nicer UART Output During Simulation

Currently, the testbench outputs one byte at a time during simulation. We want to modify this behavior to make the output easier to read. We want the contents of each register to be output together on one line. For example, we would like to output “0004\n” instead of “0\n”, “0\n”, “0\n”, and “4\n”. The specification restricts us to only modifying “model_uart.v” and to only using the $display function.

Part 3: Even Nicer UART Output During Implementation

Currently, the UART output does not specify which register it is displaying the contents of. We would like to modify this behavior to make the output more readable. For example, we want our UART output to be “R0:0003” instead of “0003” if we SEND R0 and the value stored in the register is the constant 0x0003.

Part 4: An Easier Way to Load Sequencer Program

Typing out every instruction by hand in our testbench is time consuming. Therefore, we would like to alter our testbench to read instructions from a file named “seq.code” that is at most 1024 lines long. The first line of the file will contain the number of instructions and subsequent lines will hold 8-bit instructions in binary. The specification notes that we should first load the file’s data into an array using $readmemb or $fopen/$fscanf and then execute each instruction.

Part 5: Fibonacci Numbers

This portion of the lab simply asks us to use the feature added in **Part 4** to output the first ten numbers of the Fibonacci Sequence. We were responsible for coding these instructions into the file named “seq.code”.

2.0 - Design Documentation

Since each part of the lab is independent from the rest, we will document our design of each portion of the lab separately below.

2.1 - Missing Multiply Operation

To implement the multiply operation we modified some of the existing source files to mimic the functionality of the add operation. We had to modify the ALU to accommodate the multiply op code and ensure that the register values were correctly forwarded to the seq_mult module. We added the seq_mult module which was essentially a copy of the addition module with the addition sign changed to a multiplication sign.

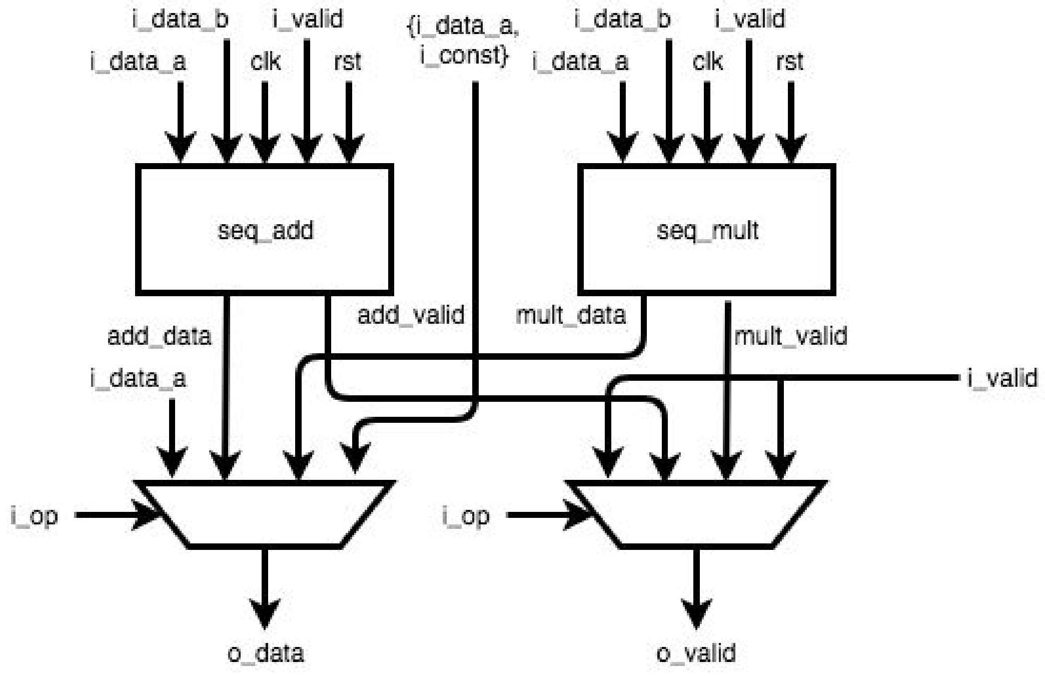


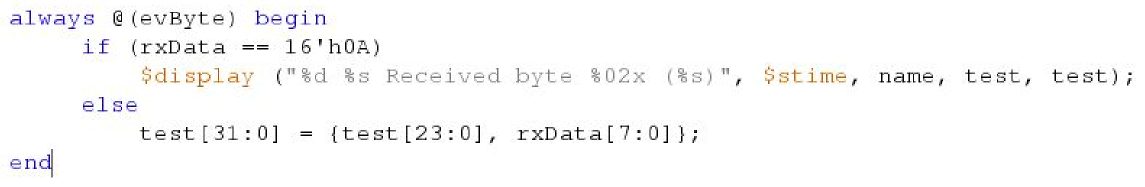
Figure 2.1.1 - A high level diagram of our ALU implementation.

Notes: The multiplexers select for the output from the adder/multiplier or pass on inputs depending on the op code, i_op. These values get passed on to other parts of our sequencer that handle the other instructions such as SEND and PUSH.

2.2 - Nicer UART Output

In order to output register contents on only one line, our implementation utilizes events. The event evByte is signaled each time one full byte is received from the UART output.

Our code examines this signal and either stores the output in a buffer or uses the $display function to output the register values when a line feed is detected.



```
always @(evByte) begin
     if (rxData == 16'h0A)
          $display ("%d %s Received byte %02x (%s)", $stime, name, test, test);
     else
          test[31:0] = {test[23:0], rxData[7:0]};
end
```

Figure 2.2.1 - The Verilog testbench code that is executed each time a byte is taken from the UART output. Note, we $display when an LF (0x0A) is detected and buffer otherwise.

2.3 - Even Nicer UART Output

In order to tell which register number we are outputting, we modified the files "uart_top.v" and "nexys3.v" to pass this information between them when a new instruction is received (inst_vld is high). Then, in the file "uart_top.v" we modified the existing state machine to output the register number in front of the register contents when sending output to UART.

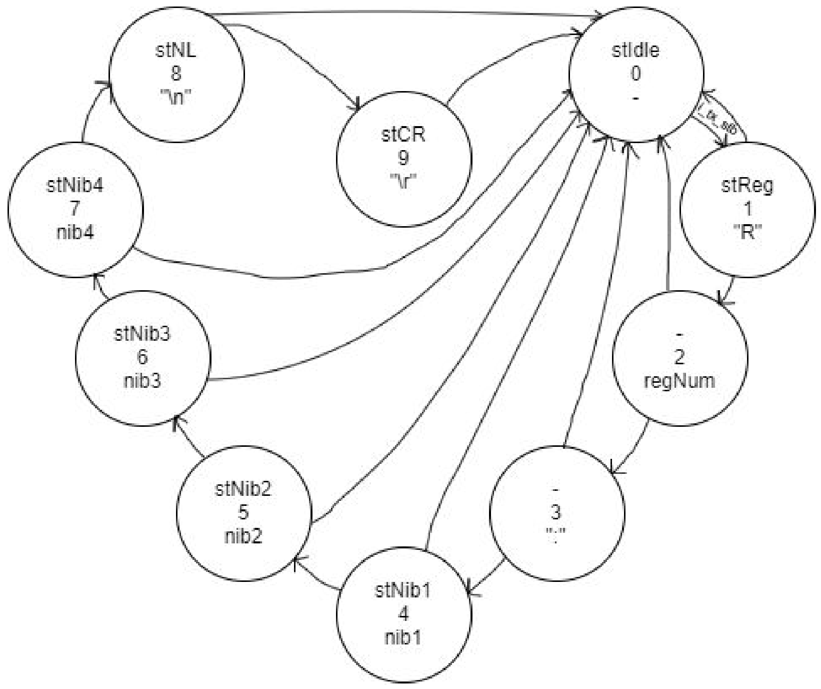
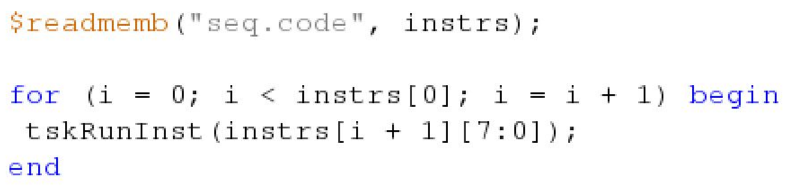


Figure 2.3.1 - The state machine diagram that implements our new output format. Note: The current state only moves from stIdle to stReg if the signal i_tx_stb is high. The lines returning to stIdle represent the transitions when the rst is high.

2.4 - An Easier Way to Load Sequencer Program

To allow the loading of instructions from a file, we utilized the function $readmemb to load the contents of our file, "seq.code", into an array. This required us to create an array of size 1024 that holds our instruction count and 8-bit instructions. Then, we are able to iterate through this array and execute each instruction.



```
$readmemb("seq.code", instrs);

for (i = 0; i < instrs[0]; i = i + 1) begin
 tskRunInst(instrs[i + 1][7:0]);
end
```

Figure 2.4.1 - The Verilog testbench code that reads from our array and executes instructions.

2.5 - Fibonacci Numbers

We generated each of these instructions manually in Sublime using the encodings from the lab manual. Then, we transferred the instructions into a file named “seq.code” that could be loaded into our program because of the features added in **Part 2.4**.

3.0 - Simulation Documentation

Since each part of the lab is independent from the rest, we will provide simulation documentation for each component.

3.1 - Missing Multiply Operation

As suggested in the manual for the first part of this lab, we tested our implementation for general functionality by comparing our testbench output to the values obtained from the Warm Up exercise in the lab manual. Furthermore, we identified some possible problem test cases that we used to stress test our multiplier.

Values [Hex Values]	Possible Issue
5 * 0 [0005 * 0000]	Multiplying by zero
5 * 1 [0005 * 0001]	Multiplying by one
65535 * 2 [FFFF * 0002]	Overflow, ensure least 16-bits retained

Figure 3.1.1 - Possible problem test cases that we used to test our multiplier.

3.2 - Nicer UART Output

We tested our implementation of this feature by running our testbench and ensuring that each register was pushed in one line, rather than in four lines. We then proceeded to add a few more instructions to be absolutely certain that this feature functioned as intended.

3.3 - Even Nicer UART Output

We tested our implementation of this feature by running our code on the Nexys 3 board and ensuring that the correct register number was output during each UART output. We developed several test cases in order to ensure this feature was implemented as intended.

Instruction	Expected Output
11000000	R0:[value in R0]
11010000	R1:[value in R1]
11100000	R2:[value in R2]
11110000	R3:[value in R3]

Figure 3.3.1 - The test cases that test for nicer UART output.

3.4 - An Easier Way to Load Sequencer Program

We were able to test this portion of our lab by ensuring that the instructions were properly loaded into an array and then executed. We tested the instructions from the Warm Up exercise in the lab manual and some additional instructions for general functionality. Additionally, we checked for edge case errors by generating a “seq.code” file that was exactly 1024 lines long. Further testing was performed in **Section 3.5** of this lab.

3.5 - Fibonacci Numbers

The testing of this part of the lab is fairly self-evident. We simply checked manually that our generated “seq.code” file correctly output the first ten numbers of the Fibonacci sequence.

4.0 - Conclusion

In summary, each component of this lab held its own unique design challenges and specifications. First, we implemented the multiply operation by mimicking the functionality of the addition operation and ensuring that our program could handle multiply instructions. Then, we created nicer UART output by taking advantage of events in the testbench and by modifying a state machine in our implementation. Finally, we were able to provide an easier way of loading instructions into our sequencer by utilizing the $readmemb function.

The main issue we ran into while working through this lab was understanding the UART protocol and understanding how the supplied code was used to implement the sequencer on the Xilinx board. Through reading documentation of the UART protocol, we were able to overcome this issue and complete the lab more easily. Another issue came later when we were tasked with loading the sequencer program from a file. Although we had some experience with Verilog, neither of us had experience with reading from a file. After doing some research on the subject, we found the code to input from a file was simpler than we had expected.

Due to the disjoint nature of this lab we found it very difficult to write an organized report for the project. One suggestion we have for the future is to require only the appendix portion from students instead of a report and an appendix.

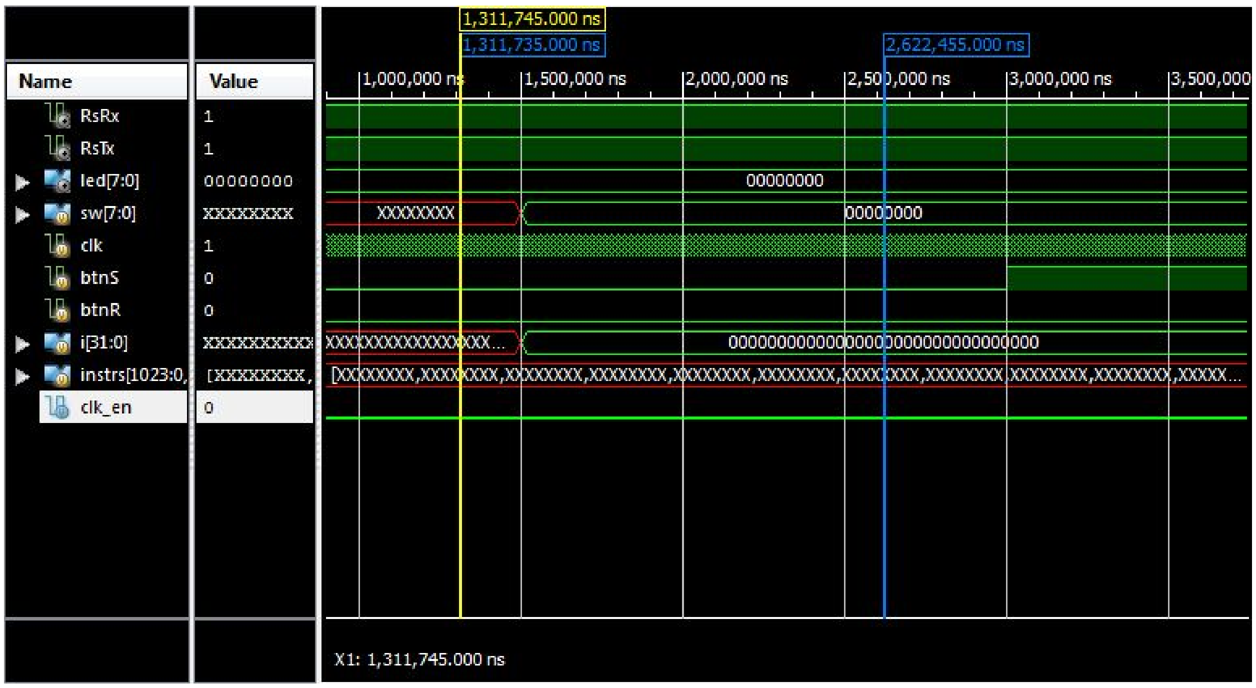
5.0 - Appendix

The following are our solutions to the questions in CSM152A Lab 2 Workshop 2.

5.1 - Clock Dividers

Question 1: The exact period of this clock cycle can be found by consulting the waveform and performing the following calculation:

$$2,622,455.000\ ns\ -\ 1,311,735.000\ ns\ =\ 1,310,720.000\ ns$$

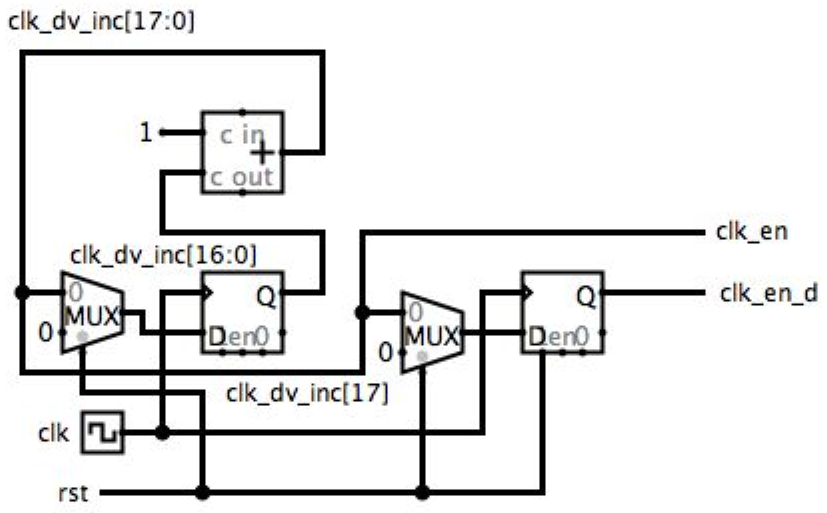


Question 2: The exact duty cycle of the clk_en signal can be found using the following calculation, in addition to consulting the above waveform and our answer to #1:

$$D\ =\ \frac{10.000\ ns}{1,310,720.000\ ns}\ *\ 100\%\ = 0.00076294\%$$

Question 3: By adding the clk_dv signal to our waveform, we were able to observe that it takes on the value of 0 (for 17 bits) while the value of clk_en is high.

Question 4: Here is our Logisim translation of the Verilog code that shows the relationships between clk_div, clk_en, and clk_en_d.

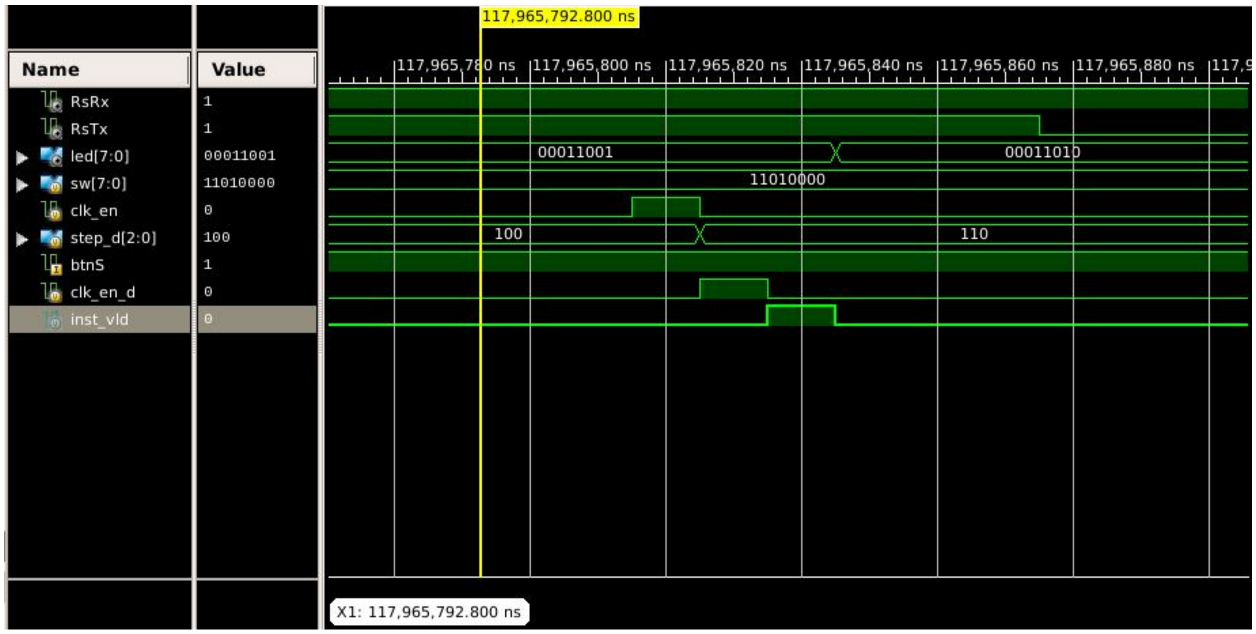


5.2 - Debouncing

Question 1: The signal clk_en_d is delayed by one clock cycle from clk_en and allows for proper sample handling. The signal step_d can be thought of as two registers that hold the two most recent sampled values of the button press. The signal is_btnS_posedge will only contain the proper value for a positive edge of a button press if we have sampled for at least two clock cycles. Therefore, we need the delayed value of the clock to be in this line. Basically, substituting clk_en for clk_en_d will break the sequencer by causing the signal inst_valid to behave incorrectly.

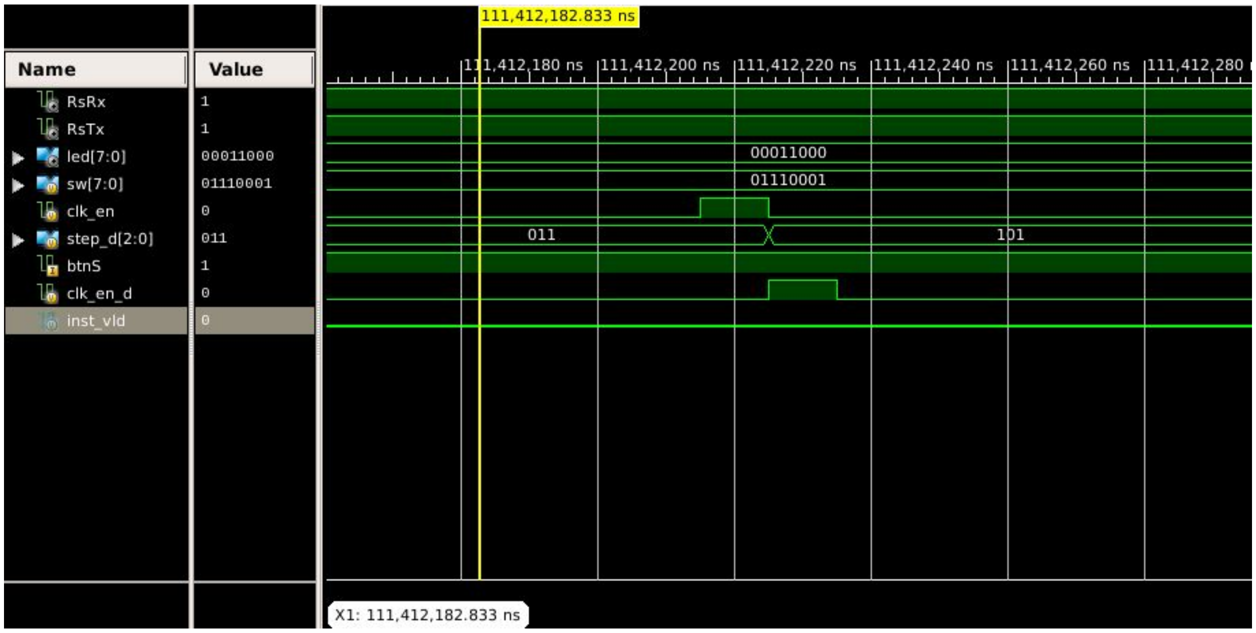
Question 2: We should be able to do this. Moving this bit down will essentially cause sampling to happen half as often which would only be an issue if we have abnormal noise in a certain button press. Assuming using clk_dv_inc[16] provides sufficient downsampling, it is unlikely that this change will result in any significant errors.

Question 3: The following waveforms capture the relationship between the signals clk_en, step_d[1:0], btnS, clk_en_d, and inst_vld.



Name	Value
RsRx	1
RsTx	1
led[7:0]	00011001
sw[7:0]	11010000
clk_en	0
step_d[2:0]	100
btnS	1
clk_en_d	0
inst_vld	0

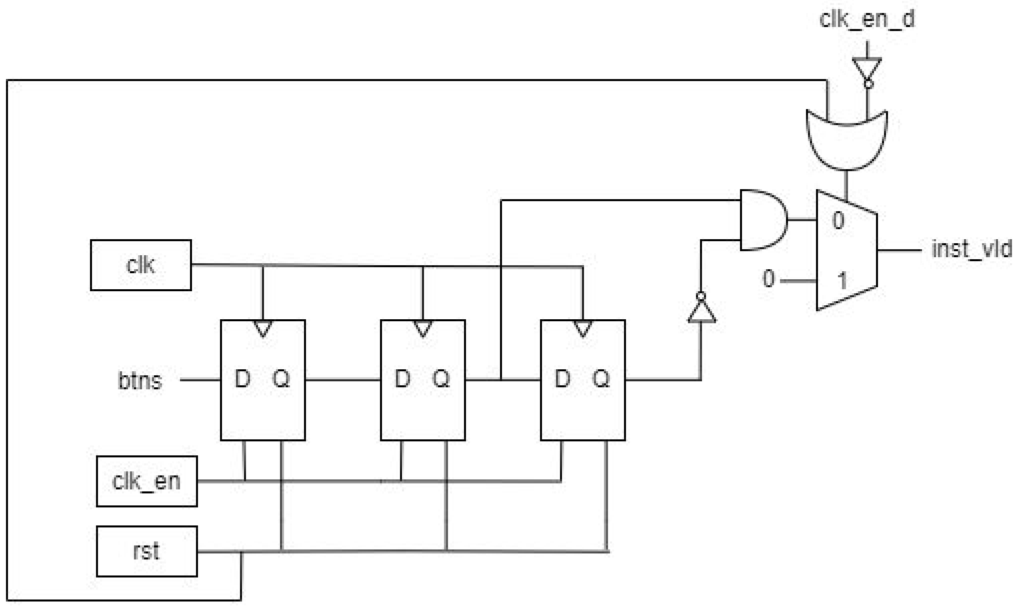
In the above waveform, we can see that btnS is high and the steps can be seen to have a positive edge (110) in the lowest two bits. This results in inst_vld being set to high and we have successful debouncing on this positive edge.



Name	Value
RsRx	1
RsTx	1
led[7:0]	00011000
sw[7:0]	01110001
clk_en	0
step_d[2:0]	011
btnS	1
clk_en_d	0
inst_vld	0

In the above waveform, we see a transition of step_d that does not result in a inst_vld being set high. The positive edge in step_d[2:0] has permeated to the correct bits for the signal to be passed through to inst_vld. In this case, inst_vld should switch to high on the next positive edge of clk_en_d.

Question 4: The following is a circuit diagram of the signals mentioned in this section

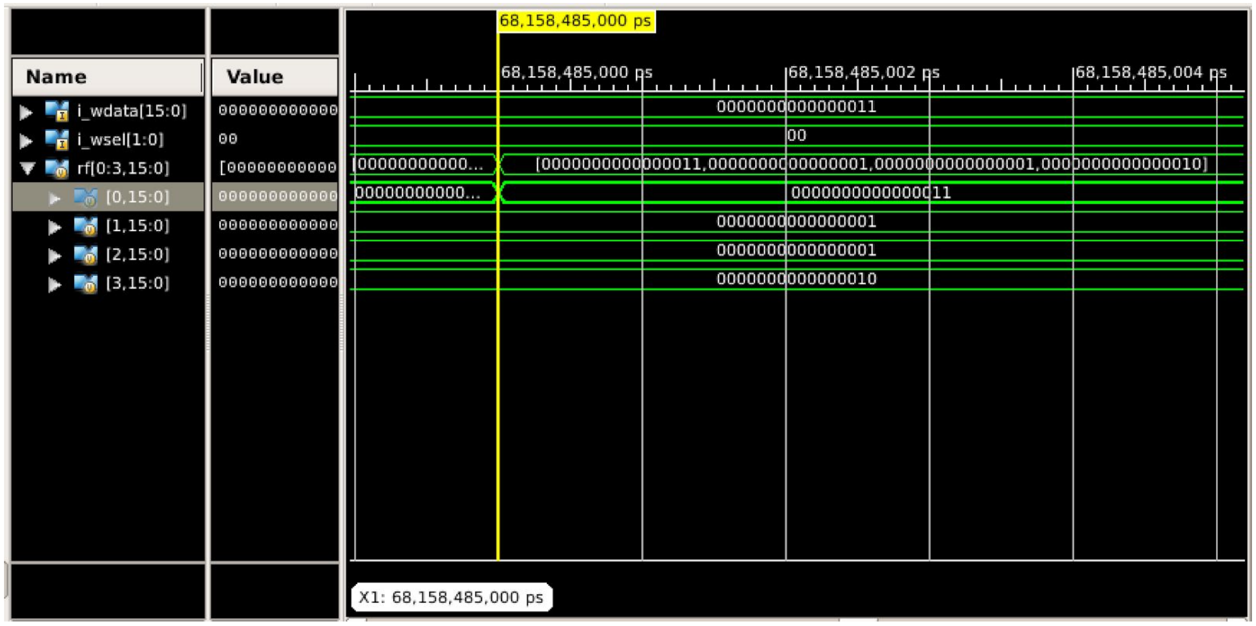


5.3 - Register File

Question 1: In file “seq_rf.v”, we see that a register is written with a non-zero value on line 33 (rf[i_wsel] <= i_wdata). The use of a non-blocking statement indicates that this line uses sequential logic. Moreover, we may also conclude that this is sequential logic due to the fact that this line is in an always block which is synchronized with the clock signal.

Question 2: The register values are written out on lines 35 and 36 which contain the code `assign o_data_a = rf[i_sel_a]` and `assign o_data_b = rf[i_sel_b]` respectively. These lines use combinatorial logic as evidenced by the blocking statements, meaning they are not dependent on any timing. Rather than using these blocking statements, we would also be able to manually implement the readout logic through the use of multiplexers with i_sel_a and i_sel_b as the select bits. Then we would simply assign the o_data_a and o_data_b to the output of the multiplexers.

Question 3: The following is a waveform showing the first time register zero is written with a nonzero value during execution of our “seq.code” that displays the first ten Fibonacci numbers.



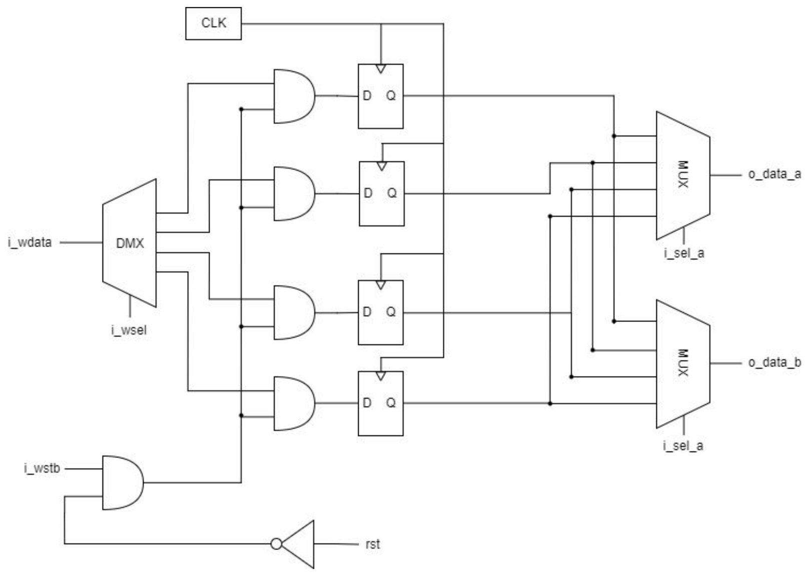
```
68,158,485,000 ps

Name              Value
i_wdata[15:0]     000000000000   0000000000000011
i_wsel[1:0]       00             00
rf[0:3,15:0]      [00000000000   [00000000000...  [0000000000000011,0000000000000001,0000000000000001,0000000000000010]
  [0,15:0]        000000000000   00000000000...   0000000000000011
  [1,15:0]        000000000000   0000000000000001
  [2,15:0]        000000000000   0000000000000001
  [3,15:0]        000000000000   0000000000000010

68,158,485,000 ps   68,158,485,002 ps   68,158,485,004 ps
X1: 68,158,485,000 ps
```

The first instruction that stores a nonzero value in register zero is an add instruction of registers two and three. This results in the Fibonacci number 3 being stored in register 0.

Question 4: The following is a circuit diagram of our register file block.


CLK
i_wdata
DMX
i_wsel
D Q
D Q
D Q
D Q
MUX
o_data_a
i_sel_a
MUX
o_data_b
i_sel_a
i_wstb
rst