Lab 1 Report

Graded

Group

EMANUEL ZAVALZA
KAI WANG
View or edit group

Total Points

40 / 40 pts

Question 1

Lab Report (for Workshop 2 following template from the syllabus)

12 / 12 pts

✔ **– 0 pts** Correct

– 4 pts Missing Section Design Description

– 2 pts Missing Section Simulation documentation

– 12 pts Lab Report for WS2 entirely missing

Question 2

Workshop 2 Part 2

4 / 4 pts

✔ **– 0 pts** Correct

– 2 pts Partially missing

– 4 pts Entirely missing

Question 3

Workshop 1 Clock Divider

8 / 8 pts

✔ **– 0 pts** Correct

Question 4

Workshop 1 Debouncing

8 / 8 pts

✔ **– 0 pts** Correct

– 2 pts part 2 partially incorrect, should be "No"

Question 5

Workshop 1 Register File

8 / 8 pts

✔ **– 0 pts** Correct

– 2 pts Part 1 should be Sequential Logic

– 2 pts Part 2 should be Comb logic

Question assigned to the following page: 2

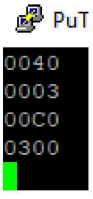
Lab 1

Translated Commands to Binary:

First reset all register values to 0 with right push button

command	instruction	register	reg/constant	reg/constant	final
PUSH R0 0x4	00	00	0100		00000100
PUSH R0 0x0	00	00	0000		00000000
PUSH R1 0x3	00	01	0011		00010011
MULT R0 R1 R2	10	00	01	10	10000110
ADD R2 R0 R3	01	10	00	11	01100011
MULT R1 R3 R3	10	01	11	11	10011111
SEND R0	11	00	X	X	1100
SEND R1	11	01	X	X	1101
SEND R2	11	10	X	X	1110
SEND R3	11	11	X	X	1111

Results from Putty:



Questions assigned to the following page: 2 and 1

CS152A Lab 1 Workshop 2

Introduction and Requirements

In Lab 1 Workshop 2, We took an existing sequencer testbench and module and extended the Verilog code to add some functionality and fully understand how it works. We used the Xilinx Vivaldo ISE to run the provided code on our Digilent Basys3 FPGA board first, interacting with the module by sending instructions via physical slide switches and buttons on the FPGA. We translated sequencer instructions from plaintext assembly-style language into binary instructions, then ran it. Then, we began workshop 2, where we extended the existing testbench to create more user friendly sequencer output and loaded custom instructions instead of the static hardcoded defaults. We made the simulation output more readable by modifying model_uart.v to store the output until a carriage return (and line feed) was received to print line by line instead of byte by byte. Next, we extended the testbench tb.v to take binary instructions from a new file called “seq.code” containing the number of instructions followed by the actual binary instructions up to 1024 lines and executed them in sequence, replacing the static hardcoded previous instruction. We then programmed seq.code with the binary instructions to output the first 10 fibonacci numbers, hardcoding the first two then having the sequencer calculate the rest.

Questions assigned to the following page: 2 and 1

Design Description

For the first part of workshop 2, “Nicer UART Output”, we first analyzed the existing code in model_uart.v. This file contained all the necessary details to implement the UART protocol, originally printing out a byte every time the edge of input RX triggered an always block and data input could be read. We modified just this module to instead collect bytes in a temporary storage register “reg[7:0] bytes[4:0];” until a certain number of bytes were received as tracked by an index variable, at which point we knew given the rest of the modules a full line of data including the carriage return and line feed was collected in bytes and should be printed. We then printed the bytes in the order they were received. This did not have any side effects interacting with other modules at all, only changing the display output.

```
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;
     index = index + 1;
     if (index == 6) begin
         $display("PRINTING BYTES");
         $display ("%s Received bytes %s%s%s%s", name, bytes[0],bytes[1],bytes[2],bytes[3]);
         index = 0;
     end else if (index <= 4)
         bytes[index - 1] = rxData;
```

For the second part of workshop 2, we extended the testbench tb.v to take binary instructions from a new file called “seq.code”, replacing the static hardcoded previous instructions. Based on the spec, we created seq.code with one line containing the number of instructions followed by one binary instruction per line up to 1024 lines to be executed in sequence. We did so by reading through tb.v to identify the user tasks that would actually tell the FPGA to do tasks, namely “tskRunPUSH, tskRunMULT, tskRunSEND, and tskRunADD” and commented out the hardcoded instructions using said tasks. We then added in code to read seq.code using $readmemb and decode instructions, converting them into task/function calls using array slices to determine the arguments.

Questions assigned to the following page: 2 and 1

```
        //load commands from seq.code instead
//        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);

        //read seq file for instruction number
        $readmemb("seq.code", instructions);
        instructions_length = instructions[0];
        //start decoding commands
        for (i = 1; i < instructions_length + 1; i = i + 1) begin

           if(seq_op_push == instructions[i][7:6])
               tskRunPUSH(instructions[i][5:4], instructions[i][3:0]);
           else if(seq_op_add == instructions[i][7:6])
               tskRunADD(instructions[i][5:4], instructions[i][3:2], instructions[i][1:0]);
           else if(seq_op_mult == instructions[i][7:6])
               tskRunMULT(instructions[i][5:4], instructions[i][3:2], instructions[i][1:0]);
           else if(seq_op_send == instructions[i][7:6])
               tskRunSEND(instructions[i][5:4]);

        end
```

For the final part of workshop 2, we programmed seq.code with the binary instructions to output the first 10 fibonacci numbers, hardcoding the first two then having the sequencer calculate the rest. We first began with writing out the instructions in pseudocode, then converting them into binary using the table given in the lab 1 instruction manual. Afterward, we wrote the number of lines and the binary instructions into a new seq.code file and ran a simulation to test if the testbench would perform the instructions from seq.code and output 10 fibonacci numbers as expected. We confirmed that it was working by reading the display output and checking that the numbers were correct.

Questions assigned to the following page: 2 and 1

Master_state_machine.v | ne | seq.code

```
26 lines
PUSH R0 0x0
PUSH R0 0x0
PUSH R0 0x0
PUSH R0 0x0
SEND R0
PUSH R1 0x0
PUSH R1 0x0
PUSH R1 0x0
PUSH R1 0x1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
ADD R0 R2 R1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
ADD R0 R2 R1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
```

```
11010
00000000
00000000
00000000
00000000
11000000
00010000
00010000
00010000
00010001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000
```

Questions assigned to the following page: 2 and 1

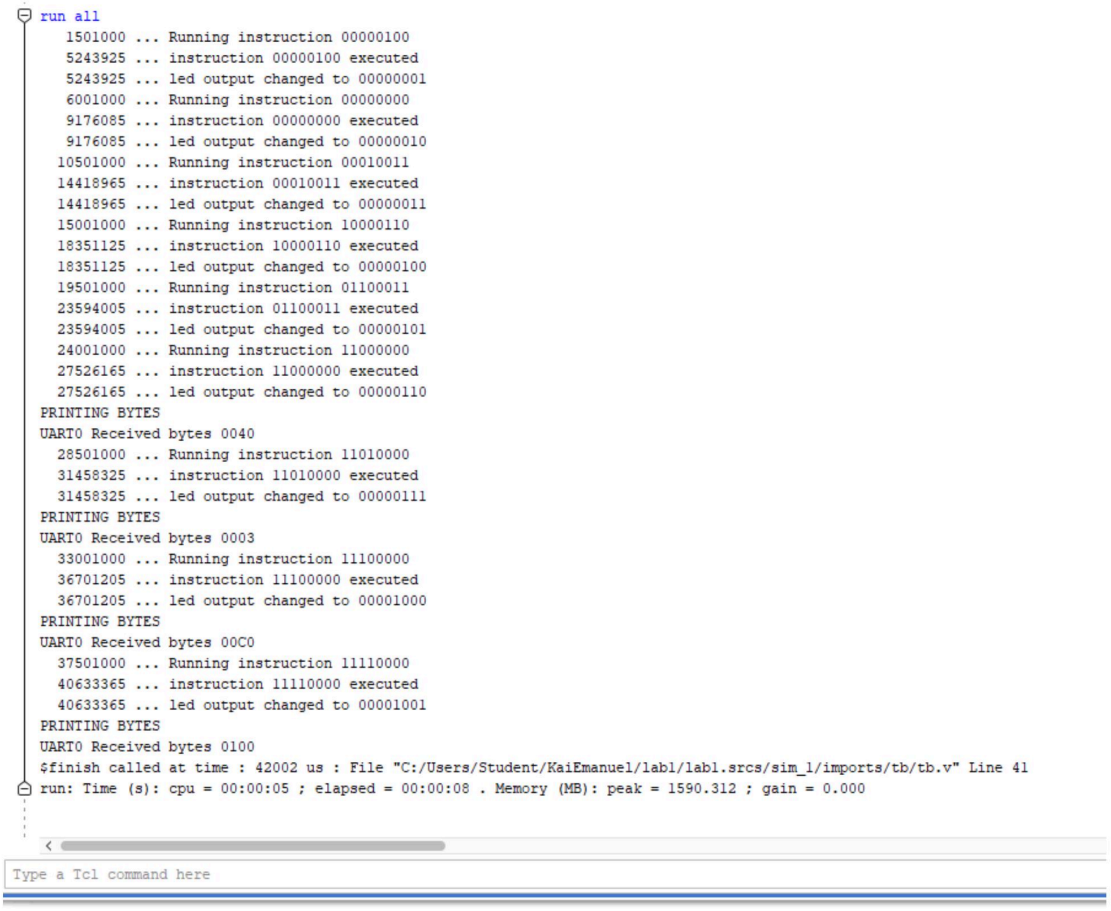
Simulation Documentation

Nicer UART Output:

```
run all
   1501000 ... Running instruction 00000100
   5243925 ... instruction 00000100 executed
   5243925 ... led output changed to 00000001
   6001000 ... Running instruction 00000000
   9176085 ... instruction 00000000 executed
   9176085 ... led output changed to 00000010
  10501000 ... Running instruction 00010011
  14418965 ... instruction 00010011 executed
  14418965 ... led output changed to 00000011
  15001000 ... Running instruction 10000110
  18351125 ... instruction 10000110 executed
  18351125 ... led output changed to 00000100
  19501000 ... Running instruction 01100011
  23594005 ... instruction 01100011 executed
  23594005 ... led output changed to 00000101
  24001000 ... Running instruction 11000000
  27526165 ... instruction 11000000 executed
  27526165 ... led output changed to 00000110
PRINTING BYTES
UART0 Received bytes 0040
  28501000 ... Running instruction 11010000
  31458325 ... instruction 11010000 executed
  31458325 ... led output changed to 00000111
PRINTING BYTES
UART0 Received bytes 0003
  33001000 ... Running instruction 11100000
  36701205 ... instruction 11100000 executed
  36701205 ... led output changed to 00001000
PRINTING BYTES
UART0 Received bytes 00C0
  37501000 ... Running instruction 11110000
  40633365 ... instruction 11110000 executed
  40633365 ... led output changed to 00001001
PRINTING BYTES
UART0 Received bytes 0100
```

Questions assigned to the following page: 2 and 1

$finish called at time : 42002 us : File "C:/Users/Student/KaiEmanuel/lab1/lab1.srcs/sim_1/imports/tb/tb.v" Line 41
run: Time (s): cpu = 00:00:05 ; elapsed = 00:00:08 . Memory (MB): peak = 1590.312 ; gain = 0.000



An Easier Way to Load Sequencer Program

1. Identify the part of the tb.v where the instructions are sent to the UUT.
   They are on lines 31-39 of tb.v.

2. Which user tasks are called in this process?
   tskRunPUSH, tskRunMULT, tskRunSEND, and tskRunADD
   The instructions are sent using these tasks, with the input arguments hardcoded.

Questions assigned to the following page: 2 and 1

For loading instructions from seq.code we got the exact same output as before

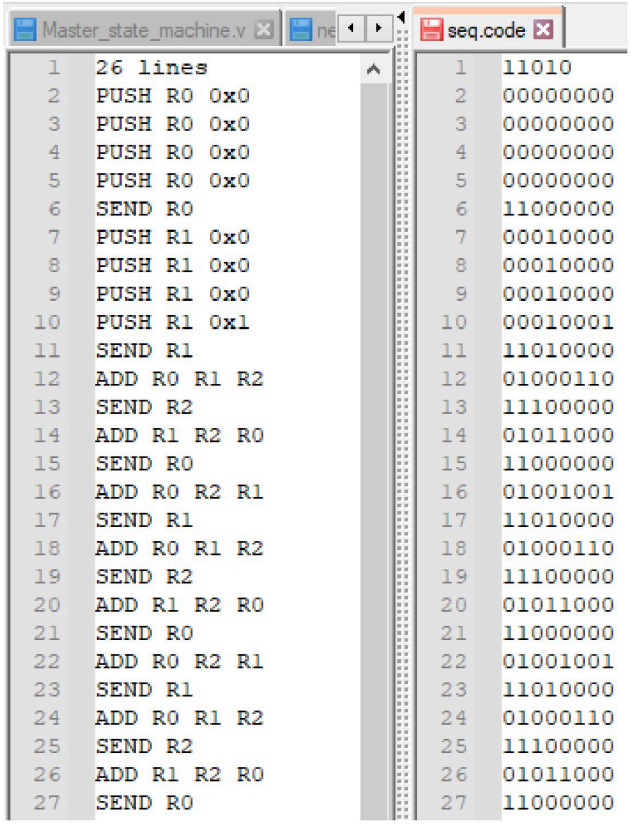
```
   1501000 ... Running instruction 00000100
   5243925 ... instruction 00000100 executed
   5243925 ... led output changed to 00000001
   6001000 ... Running instruction 00000000
   9176085 ... instruction 00000000 executed
   9176085 ... led output changed to 00000010
  10501000 ... Running instruction 00010011
  14418965 ... instruction 00010011 executed
  14418965 ... led output changed to 00000011
  15001000 ... Running instruction 10000110
  18351125 ... instruction 10000110 executed
  18351125 ... led output changed to 00000100
  19501000 ... Running instruction 01100011
  23594005 ... instruction 01100011 executed
  23594005 ... led output changed to 00000101
  24001000 ... Running instruction 11000000
  27526165 ... instruction 11000000 executed
  27526165 ... led output changed to 00000110
PRINTING BYTES
UART0 Received bytes 0040
  28501000 ... Running instruction 11010000
  31458325 ... instruction 11010000 executed
  31458325 ... led output changed to 00000111
PRINTING BYTES
UART0 Received bytes 0003
  33001000 ... Running instruction 11100000
  36701205 ... instruction 11100000 executed
  36701205 ... led output changed to 00001000
PRINTING BYTES
UART0 Received bytes 00C0
  37501000 ... Running instruction 11110000
  40633365 ... instruction 11110000 executed
  40633365 ... led output changed to 00001001
PRINTING BYTES
UART0 Received bytes 0100
$finish called at time : 42002 us : File "C:/Users/Student/KaiEmanuel/lab1/lab1.srcs/sim_1/imports/tb/tb.v" Line 64
run: Time (s): cpu = 00:00:06 ; elapsed = 00:00:08 . Memory (MB): peak = 888.453 ; gain = 0.000
```

UART0 Received bytes 0040
UART0 Received bytes 0003
UART0 Received bytes 00C0
UART0 Received bytes 0100

Fibonacci Numbers

We wrote pseudocode, then translated it into sequencer binary instructions in order to print the first 10 numbers of the Fibonacci series from the UART output. We hardcoded and UART outputted the first 2 numbers of the sequence (0 and 1) then used the sequencer to add the previous 2 numbers and print out the result repeatedly until we had 10 total numbers.

Questions assigned to the following page: 2 and 1



Master_state_machine.v

```
26 lines
PUSH R0 0x0
PUSH R0 0x0
PUSH R0 0x0
PUSH R0 0x0
SEND R0
PUSH R1 0x0
PUSH R1 0x0
PUSH R1 0x0
PUSH R1 0x1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
ADD R0 R2 R1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
ADD R0 R2 R1
SEND R1
ADD R0 R1 R2
SEND R2
ADD R1 R2 R0
SEND R0
```

seq.code

```
11010
00000000
00000000
00000000
00000000
11000000
00010000
00010000
00010000
00010001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000
01001001
11010000
01000110
11100000
01011000
11000000
```

Output:

run all

1501000 ... Running instruction 00000000
5243925 ... instruction 00000000 executed
5243925 ... led output changed to 00000001
6001000 ... Running instruction 00000000
9176085 ... instruction 00000000 executed
9176085 ... led output changed to 00000010
10501000 ... Running instruction 00000000
14418965 ... instruction 00000000 executed
14418965 ... led output changed to 00000011
15001000 ... Running instruction 00000000
18351125 ... instruction 00000000 executed
18351125 ... led output changed to 00000100
19501000 ... Running instruction 11000000
23594005 ... instruction 11000000 executed
23594005 ... led output changed to 00000101

Questions assigned to the following page: 2 and 1

PRINTING BYTES
UART0 Received bytes 0000
24001000 ... Running instruction 00010000
27526165 ... instruction 00010000 executed
27526165 ... led output changed to 00000110
28501000 ... Running instruction 00010000
31458325 ... instruction 00010000 executed
31458325 ... led output changed to 00000111
33001000 ... Running instruction 00010000
36701205 ... instruction 00010000 executed
36701205 ... led output changed to 00001000
37501000 ... Running instruction 00010001
40633365 ... instruction 00010001 executed
40633365 ... led output changed to 00001001
42001000 ... Running instruction 11010000
45876245 ... instruction 11010000 executed
45876245 ... led output changed to 00001010
PRINTING BYTES
UART0 Received bytes 0001
46501000 ... Running instruction 01000110
49808405 ... instruction 01000110 executed
49808405 ... led output changed to 00001011
51001000 ... Running instruction 11100000
55051285 ... instruction 11100000 executed
55051285 ... led output changed to 00001100
PRINTING BYTES
UART0 Received bytes 0001
55501000 ... Running instruction 01011000
58983445 ... instruction 01011000 executed
58983445 ... led output changed to 00001101
60001000 ... Running instruction 11000000
62915605 ... instruction 11000000 executed
62915605 ... led output changed to 00001110
PRINTING BYTES
UART0 Received bytes 0002
64501000 ... Running instruction 01001001
68158485 ... instruction 01001001 executed
68158485 ... led output changed to 00001111
69001000 ... Running instruction 11010000
72090645 ... instruction 11010000 executed

Questions assigned to the following page: 2 and 1

72090645 ... led output changed to 00010000
PRINTING BYTES
UART0 Received bytes 0003
73501000 ... Running instruction 01000110
77333525 ... instruction 01000110 executed
77333525 ... led output changed to 00010001
78001000 ... Running instruction 11100000
81265685 ... instruction 11100000 executed
81265685 ... led output changed to 00010010
PRINTING BYTES
UART0 Received bytes 0005
82501000 ... Running instruction 01011000
86508565 ... instruction 01011000 executed
86508565 ... led output changed to 00010011
87001000 ... Running instruction 11000000
90440725 ... instruction 11000000 executed
90440725 ... led output changed to 00010100
PRINTING BYTES
UART0 Received bytes 0008
91501000 ... Running instruction 01001001
94372885 ... instruction 01001001 executed
94372885 ... led output changed to 00010101
96001000 ... Running instruction 11010000
99615765 ... instruction 11010000 executed
99615765 ... led output changed to 00010110
PRINTING BYTES
UART0 Received bytes 000D
100501000 ... Running instruction 01000110
103547925 ... instruction 01000110 executed
103547925 ... led output changed to 00010111
105001000 ... Running instruction 11100000
108790805 ... instruction 11100000 executed
108790805 ... led output changed to 00011000
PRINTING BYTES
UART0 Received bytes 0015
109501000 ... Running instruction 01011000
112722965 ... instruction 01011000 executed
112722965 ... led output changed to 00011001
114001000 ... Running instruction 11000000
117965845 ... instruction 11000000 executed

Questions assigned to the following page: 2 and 1

117965845 ... led output changed to 00011010
PRINTING BYTES
UART0 Received bytes 0022
$finish called at time : 118502 us : File
"C:/Users/Student/KaiEmanuel/lab1/lab1.srcs/sim_1/imports/tb/tb.v" Line 69
run: Time (s): cpu = 00:00:15 ; elapsed = 00:00:21 . Memory (MB): peak = 899.586 ; gain =
0.000

The numbers were output in each “UART0 Received bytes xxxx” line in hexadecimal form:
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022

We tested each part of the workshop according to the lab specifications, and encountered no errors with the simulation in this process. All relevant output was saved here in this documentation section.

Conclusion

Over the course of Lab 1 Workshop 2, we were able to improve the viewing experience by outputting section of bytes on the same line and facilitating manual input by converting the input to be passed in through a text file to make it less error-prone than enter instructions by hand on the FPGA board. After these changes were made, our team was able to easily encode a set of input instructions to output a set of fibonacci numbers. Overall, this was a great workshop experience and gave us great insight into Verilog and UART.

Question assigned to the following page: 3

CS152A Lab 1 Workshop 1

In the previous session you were given a tutorial of FPGA design and implementation, and went through the whole process using Xilinx's toolchain. In this session, you will explore the example's simulation process to learn more about behavioral simulation, debugging, and design.

Answer the following questions as much as you can and include the answers in your lab report. **You'll have to simulate with the original source code to answer the questions**.

Clock Dividers

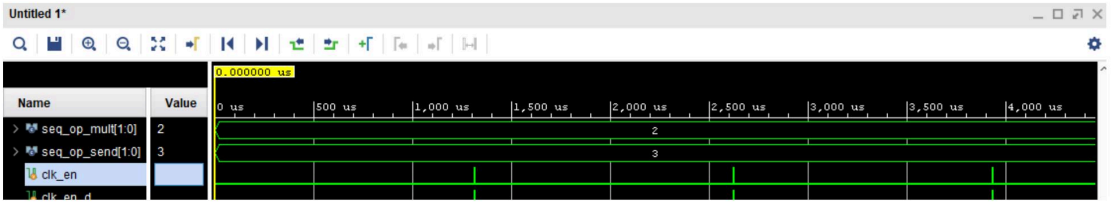
In the basys3.v file, there is a signal/reg named clk_en. The clk_en represents a clock that is much slower than the master clock clk. Read the section of code that's relevant to clk_en and try to understand how the clock divider is implemented.

1. Add clk_en to the simulation's waveform tab and then run the simulation again. Use the cursor to find the periodicity of this signal (you can select the signal and use arrow keys to reach the exact edges). Capture a waveform picture that shows two occurrences of clk_en, and include it in the lab report. Indicate the exact period of the signal in the report.

Period = 2,622.455000 us - 1,311.735000 us = **1,310.720000 us**

Period ~= 0.00131 seconds ~= 1/763 seconds

Frequency = 1 / period = 1 / (1310.72 us) = 762.9 ~= 763 Hz



2. A duty cycle is the percentage of one period in which a signal or system is active: where D is the duty cycle, T is the interval where the signal is high, and p is the period. What is the exact duty cycle of clk_en signal?

Signal is high for 1,311.745000 us - 1,311.735000 us = 0.01 us

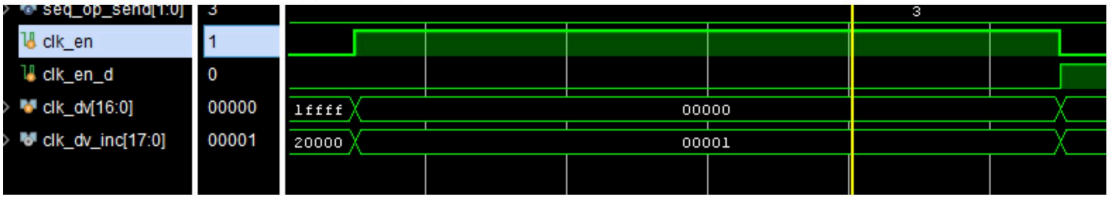
Duty cycle = signal_high_time / period = 0.01 us / 1,310.720000 us = 7.62939453 * 10^-6 s

Duty cycle = 0.000762939453%

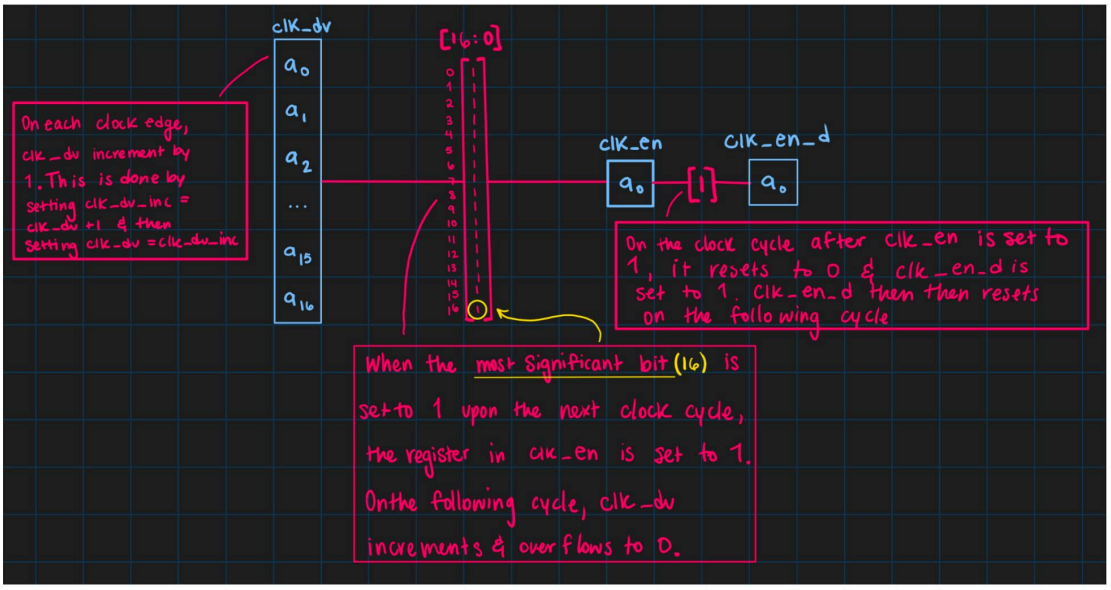
3. What is the value of clk_dv signal during the clock cycle that clk_en is high?

Questions assigned to the following page: 4 and 3

Clk_dv is 0 when clk_en is high



4. Draw a simple schematic/diagram of signals clk_dv, clk_en, and clk_en_d signals. It should be a translation of the corresponding Verilog code.



Debouncing

Now move on to the signal inst_vld. Read the relevant code and use the simulation as your aid, answer the following questions in your lab report.

1. What is the purpose of clk_en_d signal when used in expression ~step_d[0] & step_d[1] & clk_en_d? Why don't we use clk_en?

Clk_en_d signal is just clk_en delayed by one clock cycle. For the expression "~step_d[0] & step_d[1] & clk_end_d]," clk_en_d is used to correctly calculate if the instruction is provided. Step_d is altered based on value of btnS, which is if the button is pressed that clock cycle. The bit math expression makes sure the button is pressed and released and the state of clk_en used for that must be delayed since it won't

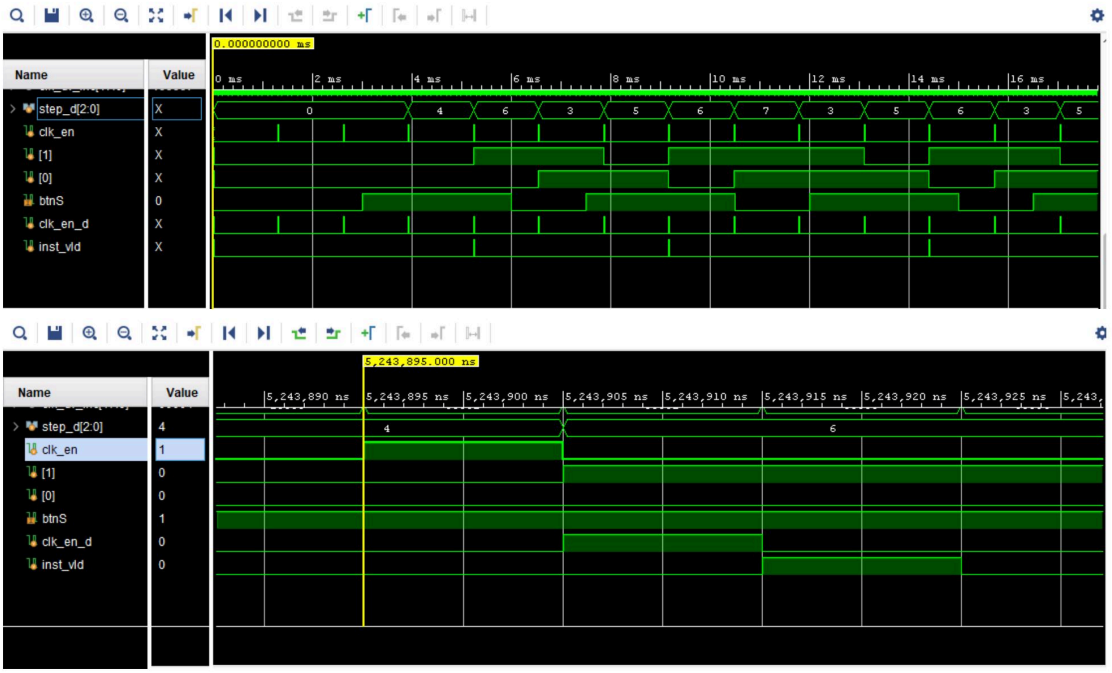
Question assigned to the following page: 4

change states during evaluation. The button bouncing rapidly will then not create valid instruction since the transitions from high to low are not stable. If clk_en was used instead step_d might change during evaluation, so evaluation should be delayed by one clock cycle.

2. Instead of clk_en <= clk_dv_inc[17], can we do clk_en <= clk_dv[16], making the duty cycle of clk_en 50%? Why?

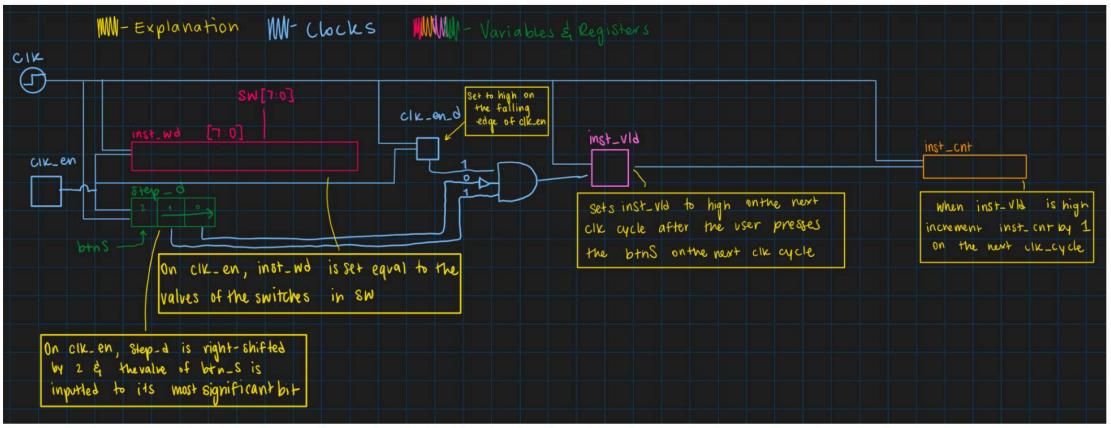
The duty cycle does not become 50%. The period is halved since the divider is using one less bit and thus divides in half the time, but this only doubles the duty cycle to approximately 0.00001525878%. The period is halved but the time the signal is high remains the same, so duty cycle doubles. To make the duty cycle 50% we'd have to use the clk signal directly, which is high half the time and low half the time.

3. Include waveform captures that clearly show the timing relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld.



4. Draw a simple schematic/diagram of the signals above (clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld). It should be a translation of the corresponding Verilog code.

Questions assigned to the following page: 4 and 5



Register File

The sequencer's register file is located in a file called seq_rf.v. It stores the values of the four registers. Take a look at the source code and see if you can understand how it is implemented. Answer the following questions in the lab report.

1. Find the line of code where a register is written a non-zero value. Is this sequential logic or combinatorial logic?

The line of code is:

else if (i_wstb)

    rf[i_wsel] <= i_wdata;

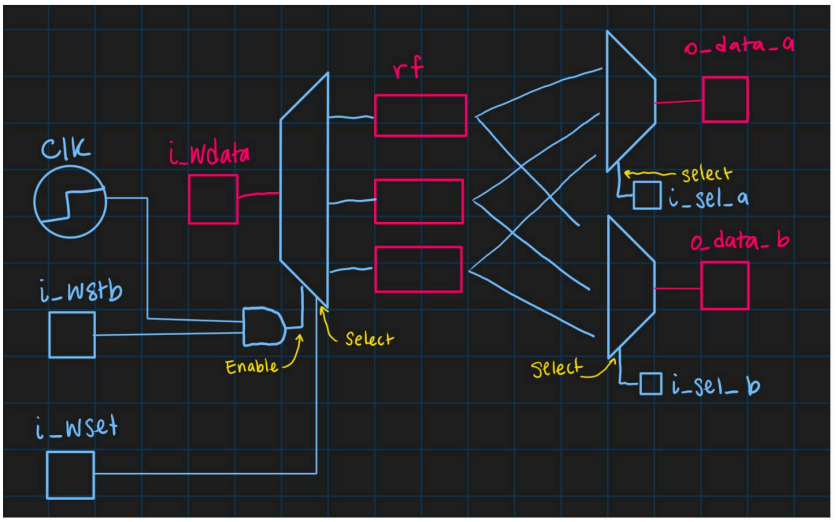
If the input signal i_wstb says to write to a register on this clock cycle, the register file indicated by input i_wsel is set equal to i_wdata. This is sequential logic because it sets registers based on a clock cycle

2. Find the lines of code where the register values are read out from the register file. Is this sequential or combinational logic? If you were to manually implement the readout logic, what kind of logic elements would you use?

This is combinatorial logic. To reimplement the readout logic, use a mux to select the appropriate register to read, then save the value to o_data_a or o_data_b respectively.

3. Draw a circuit diagram of the register file block. It should be a translation of the corresponding Verilog code.

Question assigned to the following page: 5



4. Capture a waveform that shows the first time register 3 is written with a non-zero value.