CS M152A Lab 1 Report

Puvali Chatterjee
Lab Partner: Vivian Ha

Introduction & Requirements

UART, which stands for Universal Asynchronous Receiver/Transmitter, is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Embedded systems, microcontrollers, and computers mostly use UART as a form of device-to-device hardware communication protocol. The transmitting UART is connected to a controlling data bus that sends data in a parallel form. From this, the data is transmitted on the transmission wire serially, bit by bit, to the receiving UART, which then converts the serial data into parallel for the receiving device. The baud rate is the rate at which information is transferred to a communication channel.

In this lab, we used UART to transmit data from the USB-UART bridge in a Nexys3 FPGA board to a PC-based PuTTY terminal. On the Nexys3 board, we operated an adder/multiplier sequencer using switches and buttons. The sequencer has four 16-bit general purpose registers, and it can perform instructions using registers as operands. The results are stored into register files. The sequencer can perform 4 instructions:

1. PUSH (00): left-shift the target register by 4 bits and push the new constant
2. ADD (01): add Ra and Rb and store the result in Rc
3. MULT (10): multiply Ra and Rb and store the result in Rc
4. SEND (11): send the content of target register to the UART for display

Workshop 2: After testing out the sequencer, we modified the model_uart.v file to suppress the per-byte console output and instead output one line at a time after a carriage return character is received. Then, we modified the testbench (tb.v) to make it load and execute a sequence of instructions from a file seq.code. The instructions in seq.code print the first 10 numbers of the Fibonacci series.

Workshop 1: To do workshop 1, we used the original testbench tb.v (with static instructions). First, we explored the clock divider functionality in nexys3.v and simulated clk_en, clk_dv and clk_en_d. Then, we simulated clk_en, step_d[1:0], clk_en_d, inst_vld and btnS and explored how signal debouncing was implemented in the code. Finally, we studied seq_rf.v to glean how the four 16-bit registers in the sequencer were implemented.

Nicer UART

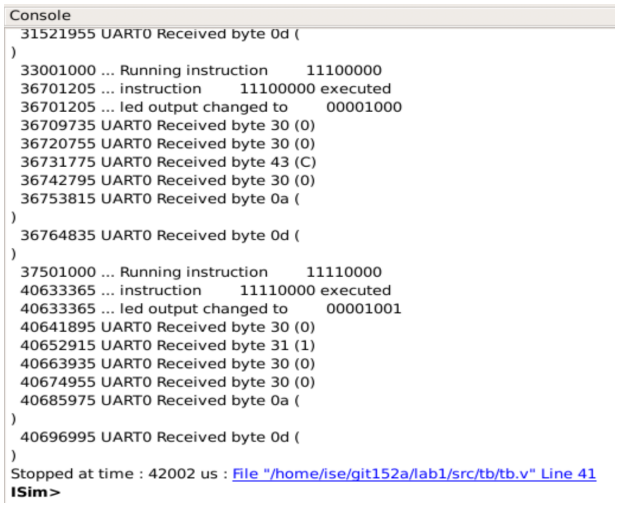
Original:

```
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);
  end
```

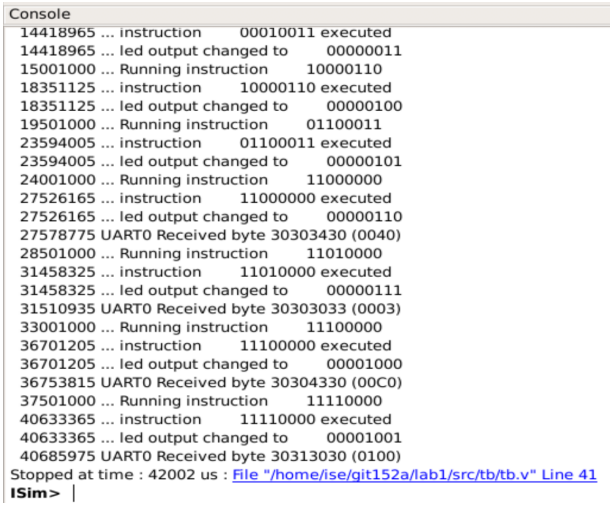
Modified:

```
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;
      //10 is ascii for line feed
      if (rxData == 10)
              $display("%d %s Received byte %02x (%s)", $stime, name, rxDataLine, rxDataLine);
      //13 is ascii for carriage return
      else if (rxData != 13)
              rxDataLine[31:0] = {rxDataLine, rxData};
end
```

Here, RX is the input to the model_UART and rxData is a 2-byte reg. To suppress the per-byte output, we first declared a 4-byte reg rxDataLine. Then, after concatenating every new input RX to rxData (at every negative edge of RX), we checked for input being a new line (ASCII decimal value 10) or carriage return (ASCII decimal value 13). At every carriage return, we concatenated rxData to rxDataLine. At every new line, we printed the contents of rxDataLine to the console.



Original



"Nicer" UART

An Easier Way to Load the Sequencer Program

---

1. Identify the part of the tb.v where the instructions are sent to the UUT.

The sequencer testbench sends a static (hardcoded) set of instructions to the UUT in the initial block as follows:

```
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);

     #1000;
     $finish;
  end
```

2. Which user tasks are called in this process?

The user tasks called here are tskRunPUSH, tskRunMULT, tskRunADD and tskRunSEND. Each task calls tskRunInst within itself to actually run the instruction.

We modified tb.v to load a sequence of instructions from a text file “seq.code” (maximum 1024 lines, first line is the number of instructions). To do so, we used the built-in Verilog system task $readmemb as follows:

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

for (i = 0; i < insns[0]; i = i + 1) begin
   tskRunInst(insns[i + 1]);
end

#1000;
$finish;
```

Here, insns is declared as reg [7:0] insns [1023:0] since each instruction is 8 bits long and the file seq.code has 1024 lines at most.

$readmemb loads seq.code and the first line of seq.code is recorded as the number of instructions. A for loop iterates through each instruction and calls tskRunInst to execute it.

Annotated screenshot of seq.code:

```
11010      26 Instructions
00000000
00000000
00000000   } Left shift R0 4 times and push 0
00000000
11000000   Send R0 —> 0
00010000
00010000
00010000   } Left shift R1 4 times and push 1
00010001
11010000   Send R1 —> 1
01000110   Add R0 and R1 and store in R2
11100000   Send R2 —> 1
01011011   Add R1 and R2 and store in R3
11110000   Send R3 —> 2
01101100   Add R2 and R3 and store in R0
11000000   Send R0 —> 3
01110001   Add R3 and R0 and store in R1
11010000   Send R1 —> 5
01000110   Add R0 and R1 and store in R2
11100000   Send R2 —> 8
01011011   Add R1 and R2 and store in R3
11110000   Send R3 —> 13
01101100   Add R2 and R3 and store in R0
11000000   Send R0 —> 21
01110001   Add R3 and R0 and store in R1
11010000   Send R1 —> 34
```

Console output (with nicer UART):

```
Console
ISim>
# 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
 23646615 UART0 Received byte 30303030 (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
 45928855 UART0 Received byte 30303031 (0001)
 46501000 ... Running instruction       01000110
 49808405 ... instruction       01000110 executed
 49808405 ... led output changed to       00001011
 51001000 ... Running instruction       11100000
 55051285 ... instruction       11100000 executed
```

(1)

```
 55051285 ... led output changed to       00001100
 55103895 UART0 Received byte 30303031 (0001)
 55501000 ... Running instruction       01011011
 58983445 ... instruction       01011011 executed
 58983445 ... led output changed to       00001101
 60001000 ... Running instruction       11110000
 62915605 ... instruction       11110000 executed
 62915605 ... led output changed to       00001110
 62968215 UART0 Received byte 30303032 (0002)
 64501000 ... Running instruction       01101100
 68158485 ... instruction       01101100 executed
 68158485 ... led output changed to       00001111
 69001000 ... Running instruction       11000000
 72090645 ... instruction       11000000 executed
 72090645 ... led output changed to       00010000
 72143255 UART0 Received byte 30303033 (0003)
 73501000 ... Running instruction       01110001
 77333525 ... instruction       01110001 executed
 77333525 ... led output changed to       00010001
 78001000 ... Running instruction       11010000
 81265685 ... instruction       11010000 executed
 81265685 ... led output changed to       00010010
 81318295 UART0 Received byte 30303035 (0005)
 82501000 ... Running instruction       01000110
 86508565 ... instruction       01000110 executed
 86508565 ... led output changed to       00010011
 87001000 ... Running instruction       11100000
 90440725 ... instruction       11100000 executed
 90440725 ... led output changed to       00010100
 90493335 UART0 Received byte 30303038 (0008)
 91501000 ... Running instruction       01011011
 94372885 ... instruction       01011011 executed
 94372885 ... led output changed to       00010101
 96001000 ... Running instruction       11110000
 99615765 ... instruction       11110000 executed
 99615765 ... led output changed to       00010110
 99668375 UART0 Received byte 30303044 (000D)
100501000 ... Running instruction       01101100
103547925 ... instruction       01101100 executed
```

(2)

```
103547925 ... led output changed to       00010111
105001000 ... Running instruction       11000000
108790805 ... instruction       11000000 executed
108790805 ... led output changed to       00011000
108843415 UART0 Received byte 30303135 (0015)
109501000 ... Running instruction       01110001
112722965 ... instruction       01110001 executed
112722965 ... led output changed to       00011001
114001000 ... Running instruction       11010000
117965845 ... instruction       11010000 executed
117965845 ... led output changed to       00011010
118018455 UART0 Received byte 30303232 (0022)
Stopped at time : 118502 us : File "/home/ise/git152a/l
ISim>
```

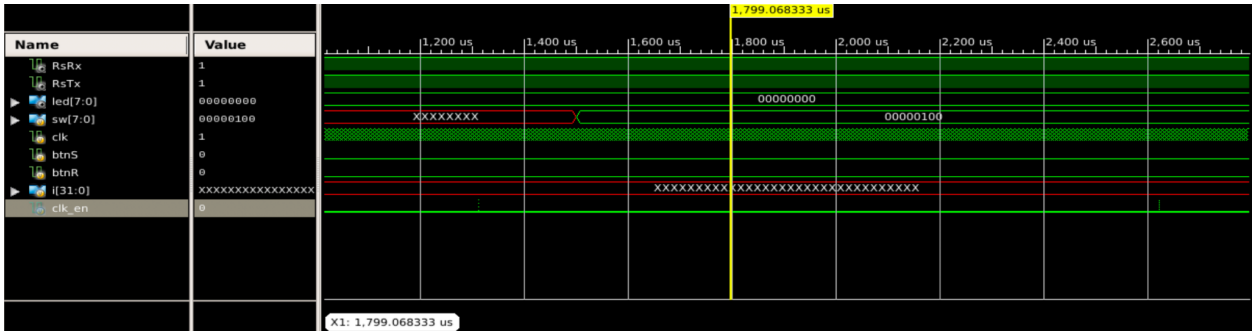
(3)

As shown, the output reports each time the UART receives a byte, and the received bytes are the first 10 Fibonacci numbers in hexadecimal representation: 0, 1, 1, 2, 3, 5, 8, D (d13), 15 (d21) and 22 (d34).

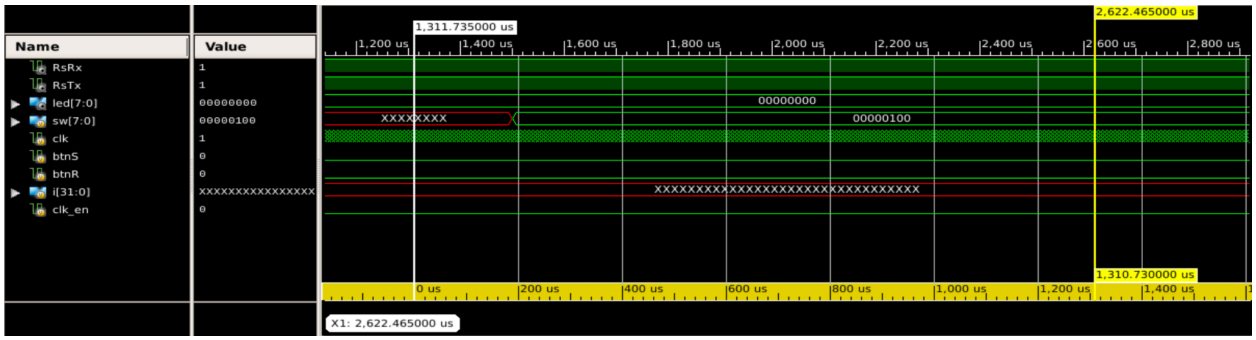
Clock Dividers

---

1. 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.

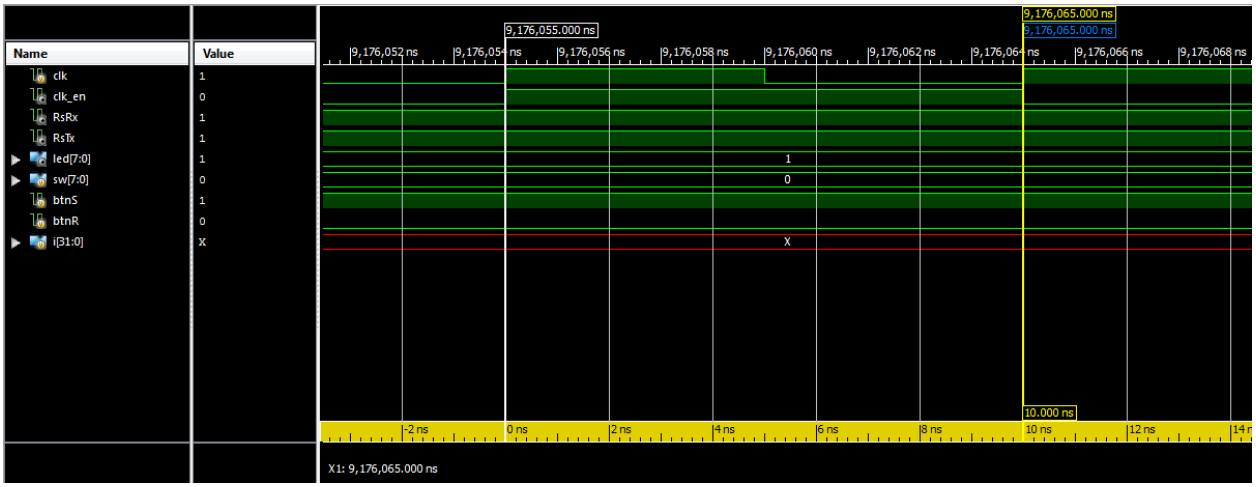


The above waveform shows the first 2 occurrences of clk_en.



The distances between the two markers set at the two occurrences of clk_en is shown to be 1,310.73 µs = **1,310,730 ns** ← period of clk_en.

2. What is the exact duty cycle of clk_en signal?

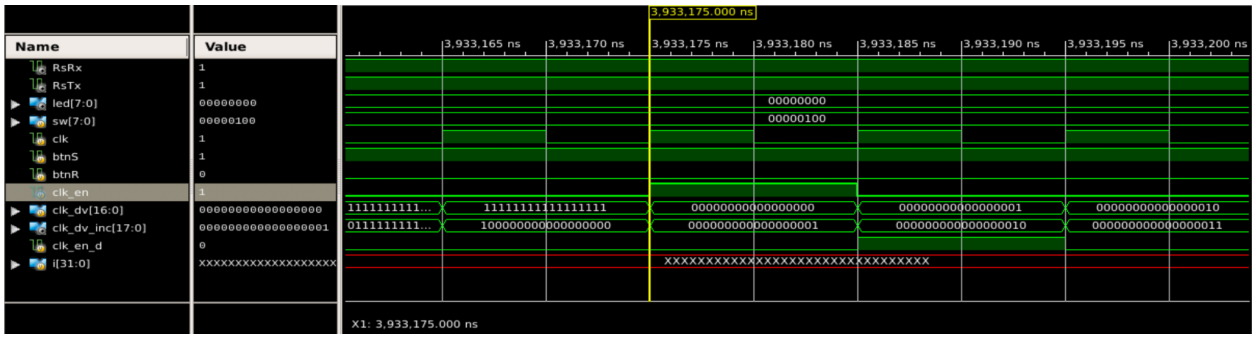


As shown in the above waveform, clk_en is high for 10 ns. So:

p = 1,310,730 ns, T = 10 ns

$D = \frac{T}{P} \times 100\% = \frac{10}{1{,}310{,}730} \times 100\% = (7.63 \times 10^{-4})\%$ or **0.000763%** ← duty cycle of clk_en

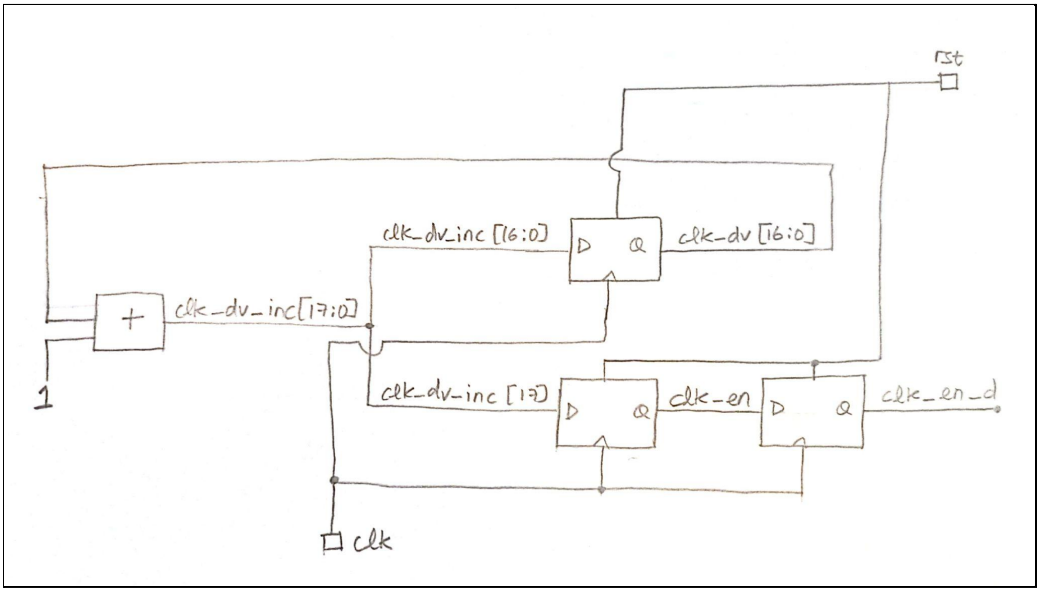
3. What is the value of the clk_dv signal during the clock cycle that clk_en is high?



clk_dv is always 0 when clk_en is high because in the nexys3.v code (shown below), clk_en gets the most significant bit of clk_dv_inc while clk_dv gets the 17 remaining bits. When clk_dv_inc is 18'b10_0000_0000_0000_0000, clk_en gets 1'b1 so it becomes high and clk_dv gets 17'b0 so it is low. When clk_dv becomes low, clk_dv_inc becomes 18'b1 (since clk_dv_inc <= clk_dv + 1) so clk_en goes back to low at the next posedge of clk.

```
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
```

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

---

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?

The given expression appears in the following code block in nexys3.v:

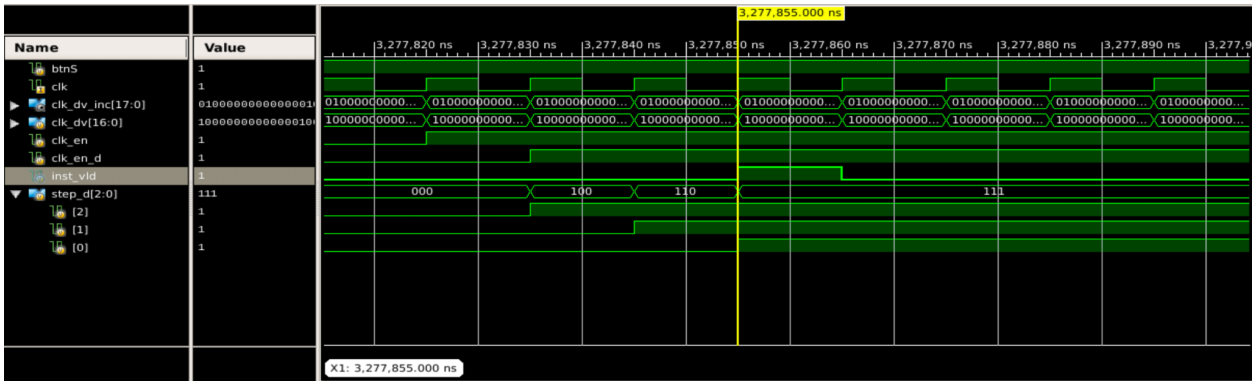
```
always @ (posedge clk)
  if (rst)
    begin
       inst_wd[7:0] <= 0;
       step_d[2:0]  <= 0;
    end
  else if (clk_en)
    begin
       inst_wd[7:0] <= sw[7:0];
       step_d[2:0]  <= {btnS, step_d[2:1]};
    end

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];
```

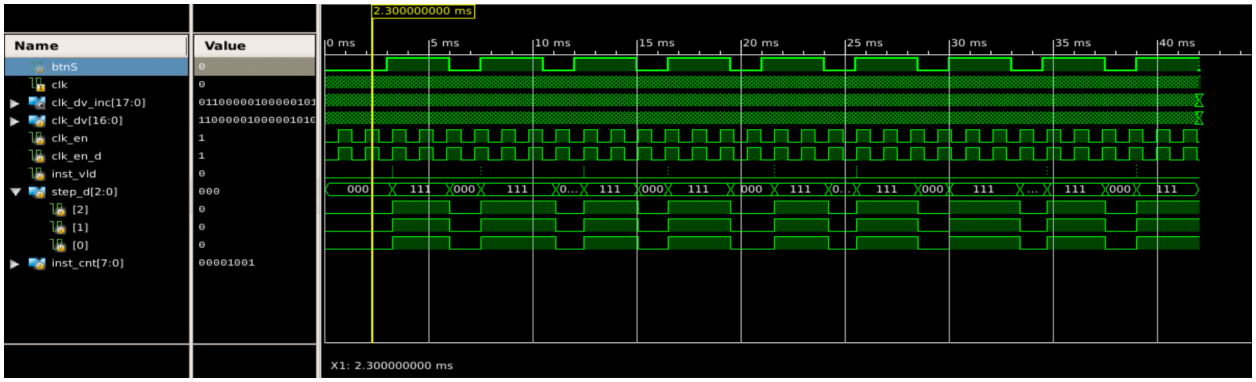
The purpose of debouncing is to generate a *single* inst_vld pulse with the period of clk every time btnS turns high. clk_en_d is a delayed clk_en i.e. it becomes high when clk_en goes from high to low. clk_en_d is used in the expression to set inst_vld because step_d is changed when clk_en is high, and clk_en goes back to low after step_d is changed. If clk_en was used instead of clk_en_d, inst_vld would always evaluate to 0 because clk_en is never high at the same time as ~step_d[0] and step_d[1]. This is demonstrated in the waveforms in part 3.

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?

When we do clk_en <= clk_dv_inc[17], clk_en is high only once during the period of clk_dv_inc, which is why its duty cycle is $1/2^{17}$ = 0.000763%. Setting clk_en <= clk_dv[16] would make clk_en's duty cycle 50%.

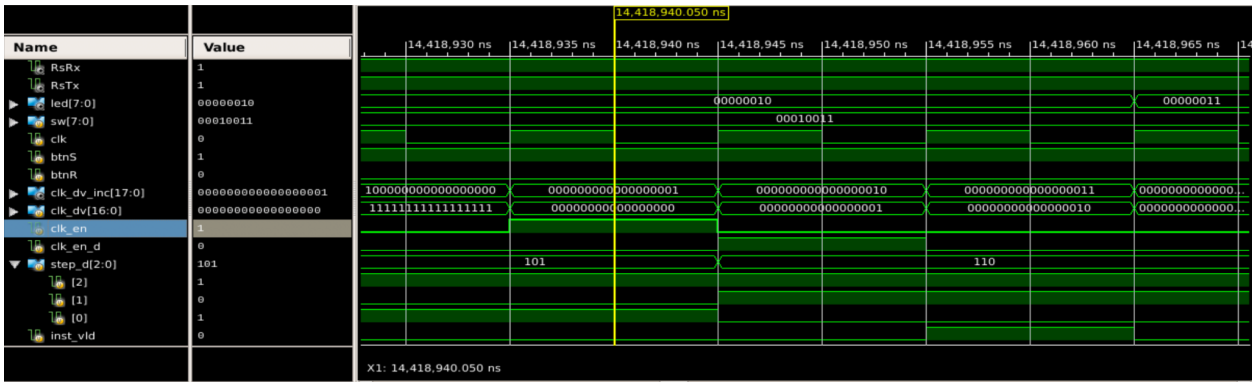
Waveforms:

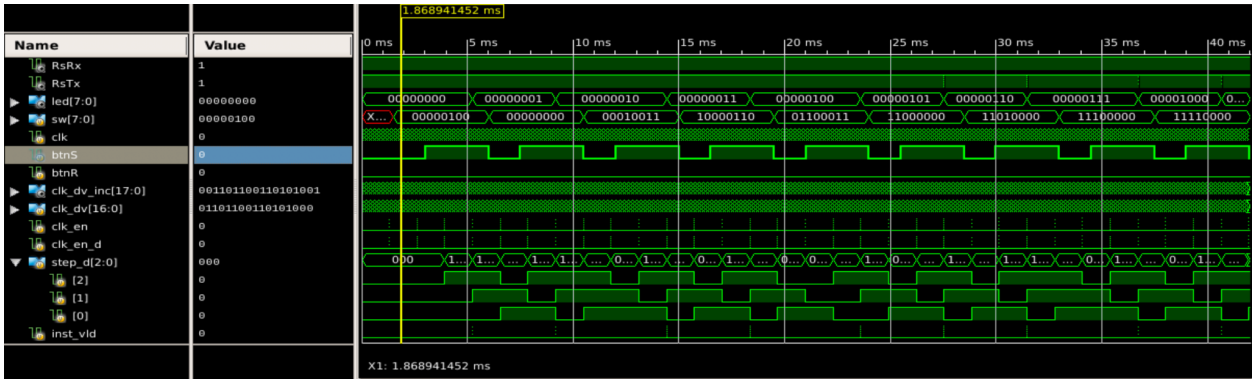




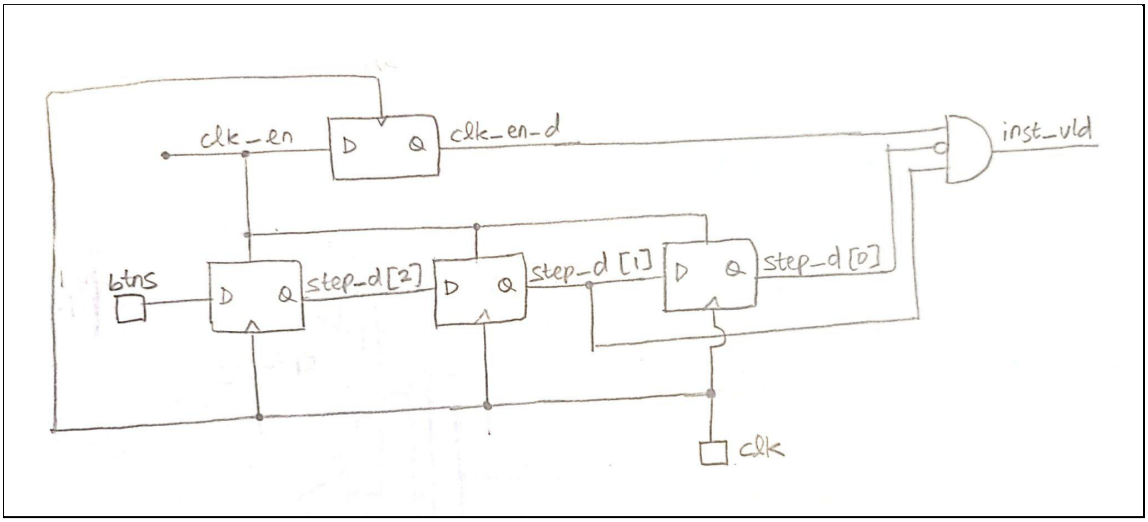
There is still only one inst_vld pulse with the period of clk for each time btnS becomes high. This is because inst_vld is still triggered by the same conditions and it is also reset to 0 at the posedge of clk. Therefore, we can do clk_en <= clk_dv[16].

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. It should be a translation of the corresponding Verilog code.



Register File

---

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

A register is written a non-zero value in the line:

```
rf[i_wsel] <= i_wdata;
```

This is sequential logic because it is in an edge-sensitive always block that is invoked at posedge clk. Also, use of the non-blocking assignment operator indicates sequential logic.

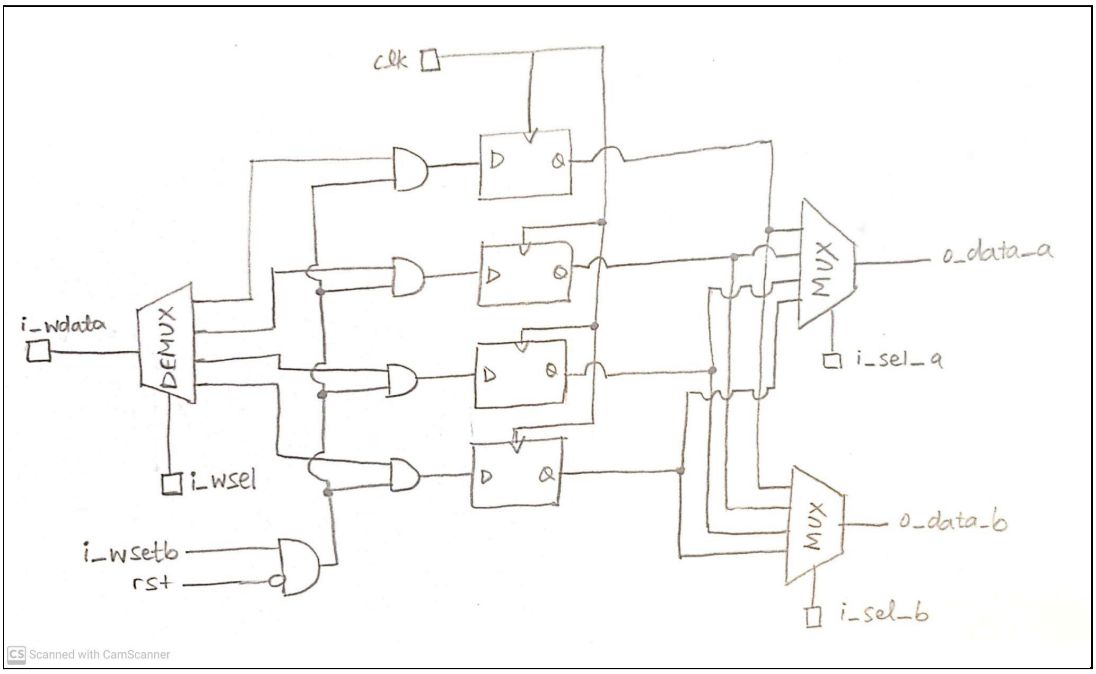
2. Find the lines of code where the register values are read out from the register file. Is this sequential or combinatorial logic? If you were to manually implement the readout logic, what kind of logic elements would you use?

Register values are read out from the register file in the lines:

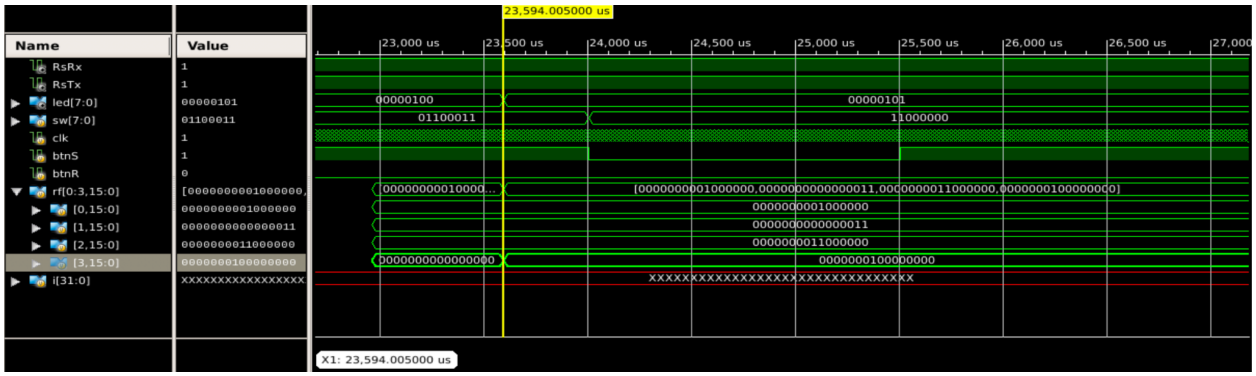
```
assign o_data_a = rf[i_sel_a];
assign o_data_b = rf[i_sel_b];
```

This is combinatorial logic because of the blocking assignment operators. These lines are also not in any edge-sensitive always block. To manually implement this readout logic, I would use multiplexers with i_sel_a and i_sel_b as the selectors to read out the values from the register file and assign them to o_data_a and o_data_b.

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



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



Conclusion

In this lab, we operated an adder/multiplier sequencer on a Nexys3 FPGA board and transmitted its output via USB-UART to a PC-based PuTTY terminal. We modified a model uart and testbench for simulation to suppress the per-byte output and to read in instructions for the sequencer from a text file instead of executing static hardcoded instructions. We also studied the provided source code to understand the implementation of clock dividers, debouncing and a register file.

We did not encounter any significant issues while modifying model_uart.v or tb.v for workshop 2, but I did face some problems while doing workshop 1, especially while answering the questions and

translating the Verilog code into schematics of the signals involved in clock dividers, debouncing and register files. I also had some difficulty understanding how the various signals were being manipulated to generate the debouncing pulse int_vld, so I referred to an online resource (listed in references section) to learn more about how and why button debouncing is done in Verilog. The best solution to these issues was to read the source code closely and draw diagrams to be able to visualize the logic.

Another minor issue I had was while trying to add the reg “rf” (in seq_rf.v) to the waveform since it is an array and cannot be declared as an output port. I solved this by looking up the Xilinx ISE manual.

One way this lab could be improved is by providing more preliminary information on UART protocol, clock dividers and debouncing before the lab to help students be more prepared before encountering these for the first time in class.

References

---

1. https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

2. https://www.fpga4student.com/2017/04/simple-debouncing-verilog-code-for.html