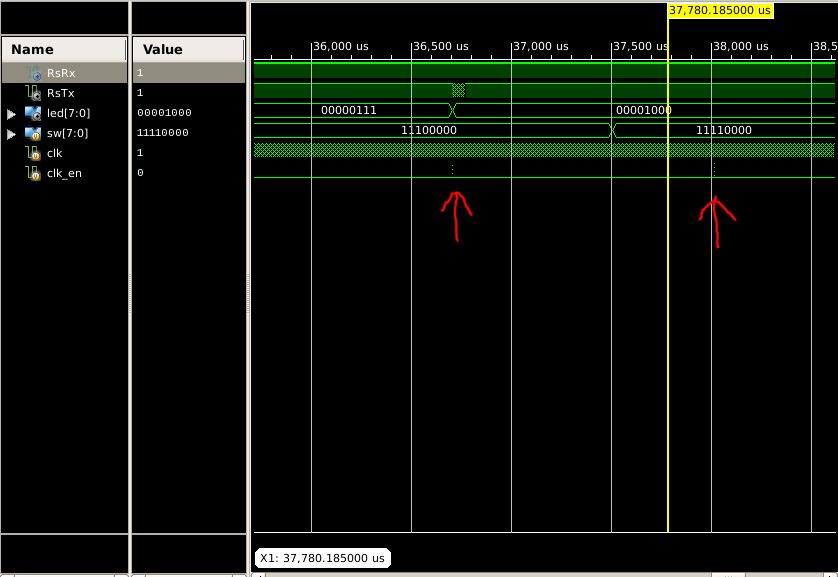
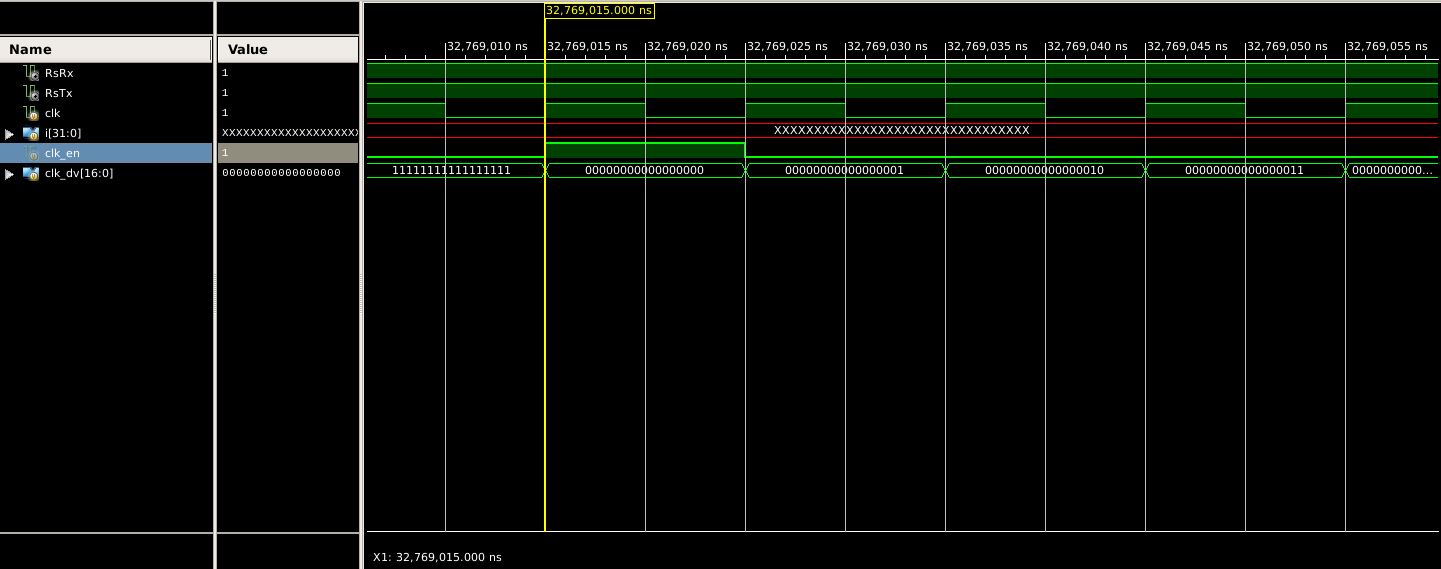
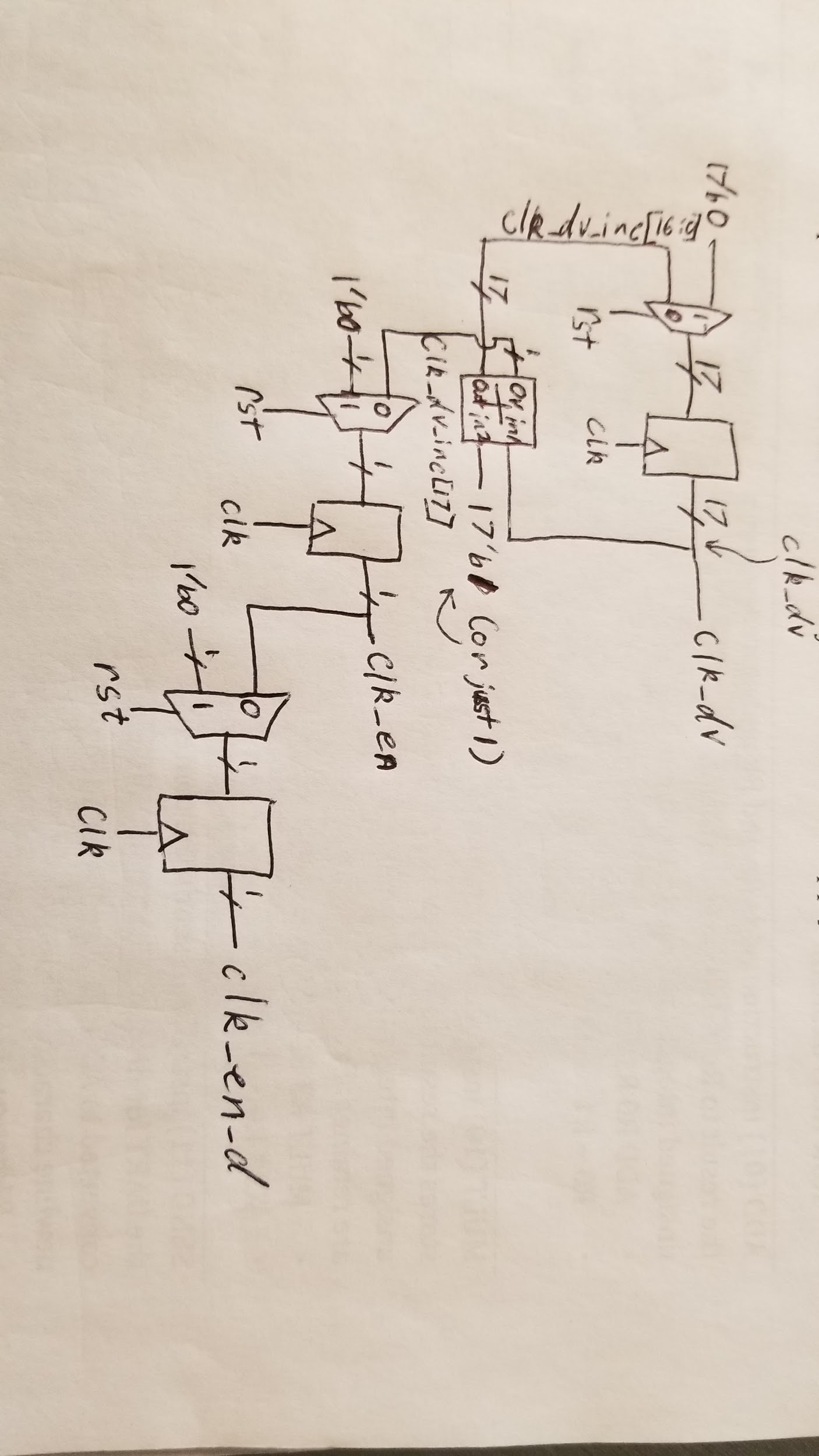
**Lab Report 1**

Alexander Graening, Xilai Zhang, Joseph Miller

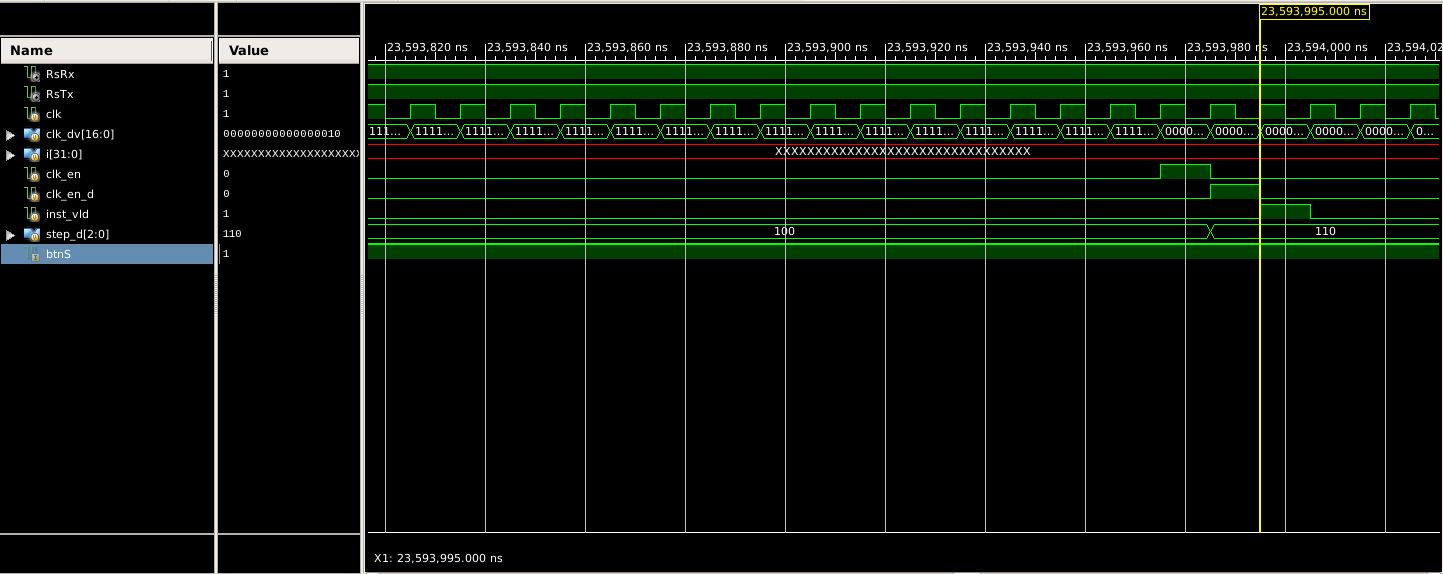
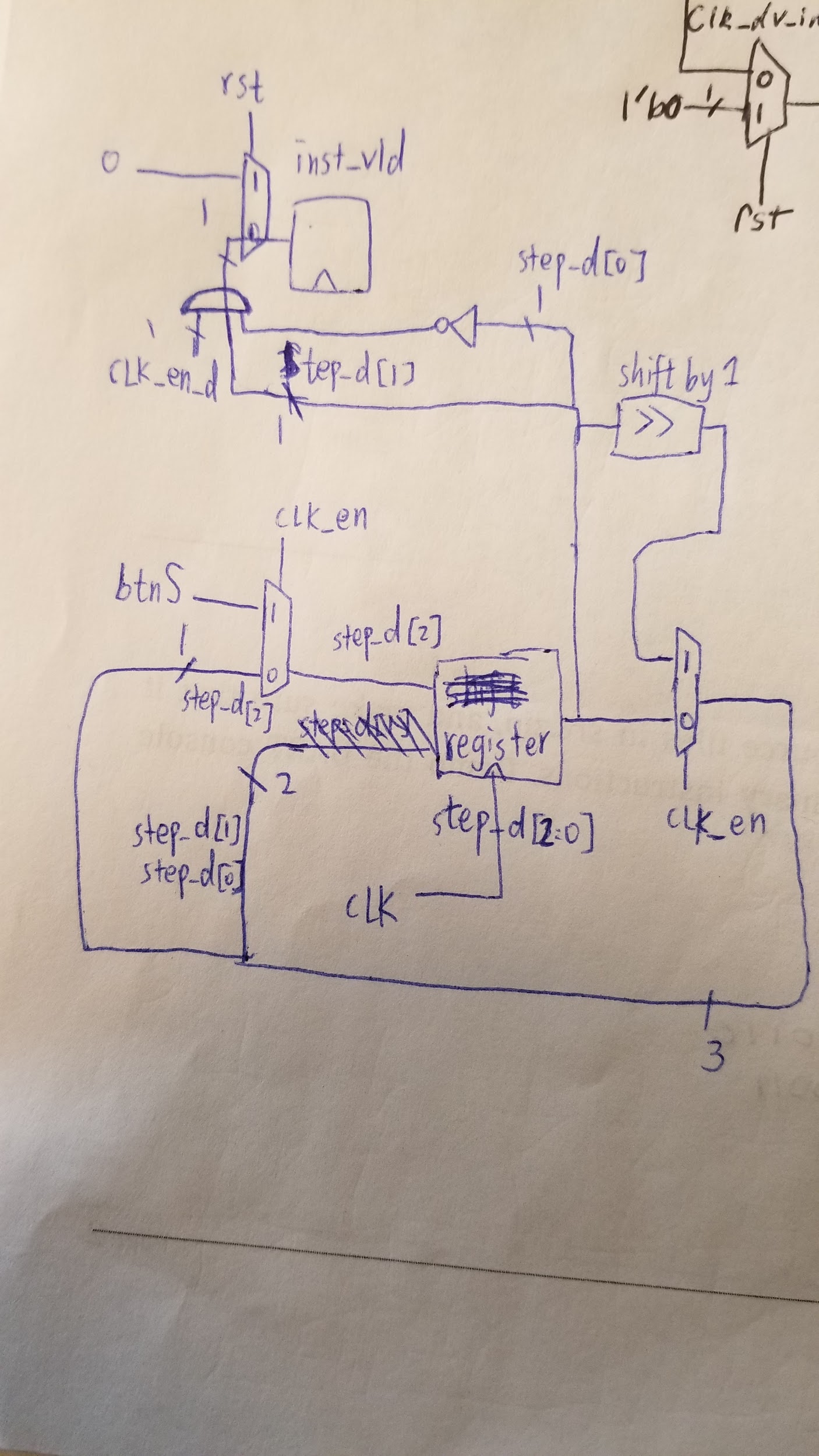
**Workshop 1:**

**Clock Dividers**

1. After adding clk\_en to the waveform we used the cursor to measure the differences in time between successive posedge clk\_en. Measuring three time differences gave the same result of 1.31072 mili-seconds period. (For example: 41.944055ms - 40.633335ms = 1.31072ms) **Figure 1:** Signal clk\_en Showing two pulses
2. Using a similar method to measure the time between pulse edges, we calculated a duty cycle of 0.0000076 or 0.00076%. (T = 1.311745ms - 1.311735ms = 0.00001ms, P = 1.31072ms, D = (T/P)\*100% = 0.00076%)
3. When clc\_en is high, clk\_dv = 17’b0. clk\_dv counts up to 16’b1111111111111111, then clk\_en gets clk\_dv[17] and clk\_dv is assigned to the less significant zero bits.**Figure 2:** Signal clk\_dv Showing High Pulse for clk\_en.

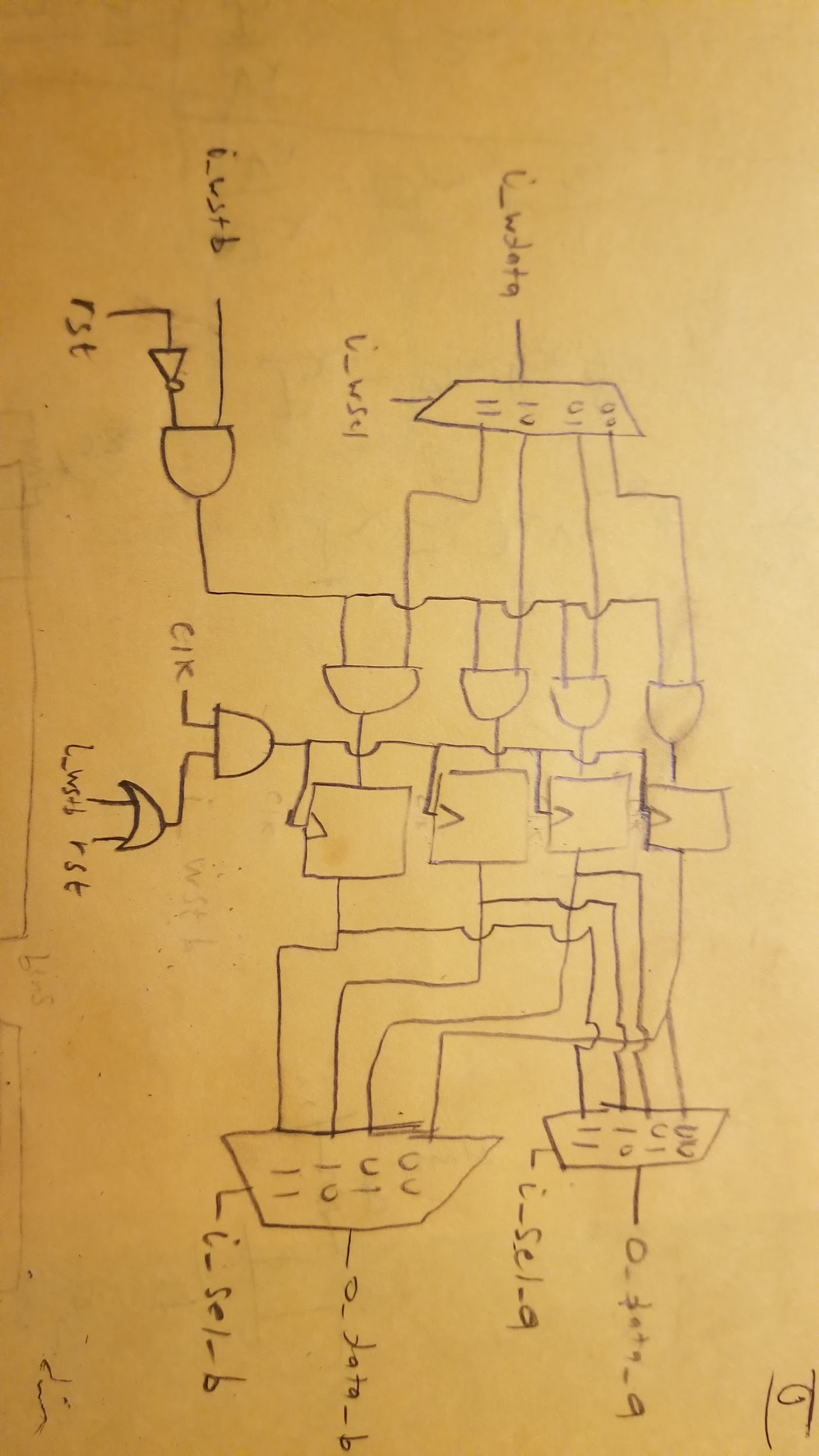
**Figure 3:** Schematic Diagram Translation of Clock Divider Verilog. The three multiplexers handle reset logic by loading 0 into each of the registers if rst is pressed to clear all the values. Note the outputs of the adder block are on the left and the inputs are on the right in this diagram. Essentially, the value clk\_dv is incremented each clock cycle by adding 17’b1 and the overflow is stored in the register for clk\_en. The value of clk\_en is stored in the clk\_en\_d which acts as an indicator of the clock division pulse the pulse clock cycle after the pulse.

**Debouncing**

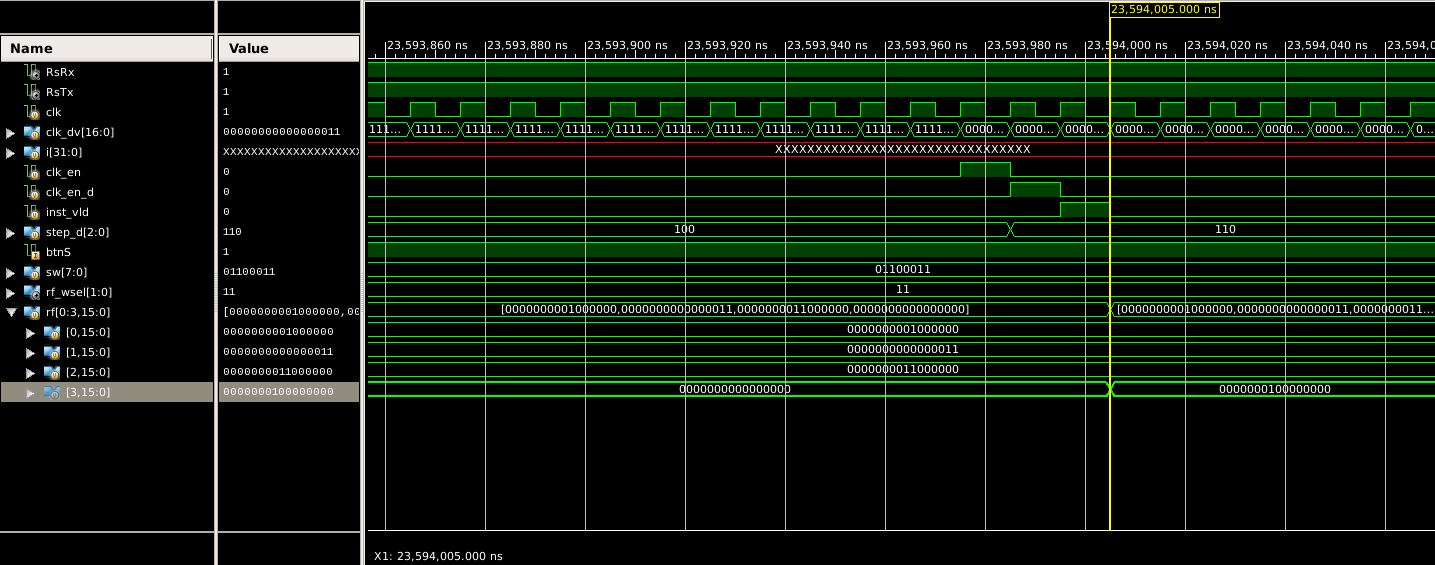
1. The code uses clk\_en\_d to remember clk\_en for a second step of the state machine used to capture a button press. The rising edge of clk\_en triggers measurement of the btnS and storage in step\_d buffer. clk\_en\_d is recorded clk\_en that triggered the shift of the bits [2:1] into position [1:0] in step\_d. Since the instruction can only be calculated as valid after the shift, clk\_en\_d is necessary and clk\_en would miss the shift.
2. Only changing the single line of code “clk\_en <= clk\_dv[17];” into “clk\_en <= clk\_dv[16];” would not work, since this would result in a signal that is high 50% of the time and low the other 50%. (This is different from the single short pulse used in the original code.)
3. **Figure 4:** Debouncing Waveform Showing Timing for Capturing Rising Edge of Button
4.  **Figure 5:** Schematic for Debouncing： step\_d is a 3 bit register that may update its value on clock cycle edge. If clk\_en signal is high, btnS is taken in as step\_d[2], the original step\_d[2] and step\_d[1] are shifted by 1 and used as new step\_d[1] and step\_d[0]. Otherwise step\_d remain unchanged. Inst\_vld is the AND result of clk\_en\_d, step\_d[1] and ~step\_d[0], and the reason for this is described in part 1. When reset is high, inst\_vld is set to 0. (Note: The black pen in the upper right corner is part of the previous diagram, not this circuit.)

**Register File**

1. In line 33, the selected register (rf[i\_wsel]) gets i\_data with a non-blocking assignment. This indicates sequential logic. In addition, this is located within an always block triggered at a clock cycle which serves as an additional hint that this piece of code is intended to be sequential.
2. Lines 35 and 36 use blocking assignments (combinatorial logic) to assign the outputs. If we were implementing this in a block diagram without using Verilog, we would use multiplexers to select the correct register value to connect to the output based on the select signal.



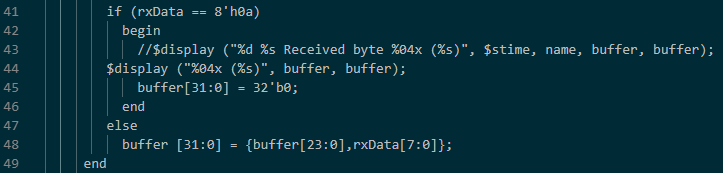
**Figure 6:** Schematic for register file. In this diagram, the the mux on the left is selected from to choose which register to denf i\_wdata to. The parameter i\_wsel is the value being selected on. This data is then anded with the output from (i\_wstb & rst) to determine if the data should pass through or be reset to 0. The data then passes to one of the 4 registers that will capture and hold onto the value. The clock signal driving these registers is controlled by the clk and (i\_wstd | rst). The registers are to only be updated if either of those signals are high and the clock signal is high. The output of the registers then feeds to two different muxes. I\_sel\_a is used to determine which register to pull from to assign o\_data\_a and i\_sel\_b is used to select which register to assign o\_data\_b.

1. **Figure 7:** First Time REgister 3 is Updated to a Nonzero Value. This is due to the instruction for R3 = R0 + R2.

**Workshop 2:**

**Nicer UART Output:**

The original output from the provided testbench presented the data one byte at a time rather than being a single line of output. Since this is pretty difficult to read, we were tasked with modifying the code to display the code in a more readable format where all the bytes were on a single line. To achieve this, we created a 4 byte long register and began checking received bytes for the newline character ‘\n’. If we didn’t receive a newline, the byte would be concatenated onto the new larger register and the process would repeat until receipt of the newline.



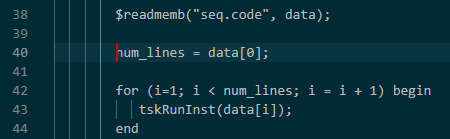
**Figure 8: Code Snippet Showing Modifications to model\_uart.v file**

**An Easier Way to Load Sequencer Program:**

1. The part of the testbench responsible of sending signals to the UUT is located in the tskRunInst task located at the bottom of the file. It is in these lines of code that the switch register is properly modified with the input received from the hard coded instructions present in lines 30-38.

2. The user tasks called in this process are tskRunPUSH, tskRunMULT, tskRunADD, and tskRunSEND. Each of these tasks of course call tskRunInst which is responsible for modifying the switch register for the UUT.

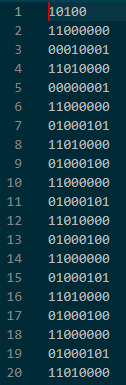
Originally, this file was hardcoded to run a specific set of instructions, rather than reading them in from a separate file. We improved on this design by utilizing the $readmemb function in verilog to pull data from a file seq.code and load it into a register. Once the data was loaded, the first line was read to determine the number of instructions present in the registers, and the rest of the instructions were executed by using the tskRunInst task which set the switch register to the proper instruction to be carried out

.

**Figure 9: Code snippet showing modifications to tb.v**

**Fibbonacci Numbers:**

The final part of this lab tasked us with writing a binary version of the fibonacci sequence named seq.code to be loaded into the testbench. It was a simple 20 line program that followed a basic set of logic. Send R0 since it is reset to 0 by default. Add one to register R0 then add R0 and R1 before sending R1. Then the two registers would be added to each other and stored alternatingly between R0 and R1. After a register was written to, that register would be sent. This process is repeated until all 10 numbers of the fibonacci sequence were printed.



**Figure 10: Fibonacci program used.**