Lab 1 Report

UCLA Samueli School of Engineering
CS M152A Spring ‘24
Lab 5 TA Chenda Duan

Group 6
Tae Hwan Kim 506043010
Brad Lowe 905581956

Introduction

I. What is UART?

UART, or Universal Asynchronous Receiver/Transmitter, is functionality for a field-programmable gate array (FPGA) that allows it to communicate with a PuTTY window. This allows the values stored in its registers and its outputs to be seen on a window of the computer that the FPGA is connected to.

II. What were the inputs/outputs, how did you monitor behavior?

The inputs to our FPGA came from the buttons and switches on it. For example, we can use the 16 switches to input binary numbers and then press a button to send that number, often representing an instruction, to the FPGA where it will be executed with the code fed into it through Vivado. The outputs are whatever shows up on the PuTTY window, as well as the LEDs that light up on the FPGA itself. Outputs come from running the instructions we submit, with the SEND instruction in particular allowing the output to be sent to PuTTY.

III. What does the testbench do and how does it help monitor code behavior?

The testbench is a Verilog program that is used to test the rest of our code. It holds a hardcoded (or in the case of 4B, programmable) sequence of instructions that will be run through the actual code, with logs showing the outputs at regular intervals. This helps us monitor the behavior because we can run the testbench to see the output of the logs anytime we make changes to the code.

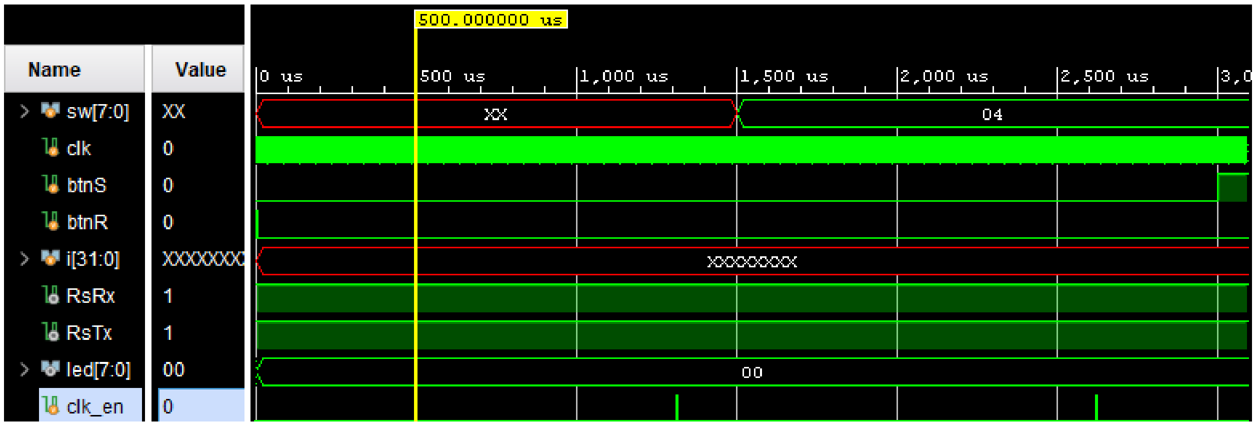
Part 1: Understanding Instruction Encoding & Part 2: Operating the UART

Demo confirmed by TA Chenda, in which we translated the instructions into binary and input them on our FPGA.

Part 3: Observing Testbench Results

Section 3A: Clock Division

Screenshot: *simulation waveforms showing 2 consecutive cycles of the clk_en signal*



I. What is the period of clk_en?

```
Period: t3 - t2 = t2 - t1 = 1,310.72 us = 1,310,720 ns
```

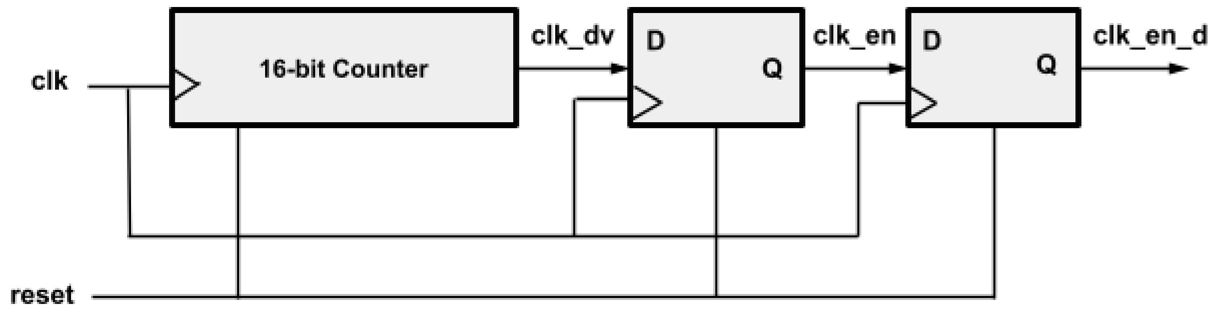
II. What is the exact duty cycle of the clk_en signal?
A duty cycle is the percentage of one period in which a signal or system is active: D=T/P×100%, where D is the duty cycle, T is the interval where the signal is high, and P is the period.

```
T = 1,311.745 us - 1,311.735 us = 0.010 us
P = 1,310.72 us
D = 0.010 / 1,310.72 * 100% = 0.00076%
```

III. What is the value of the clk_dv signal during the clock cycle that clk_en is high?
Reference the code to explain your answer.

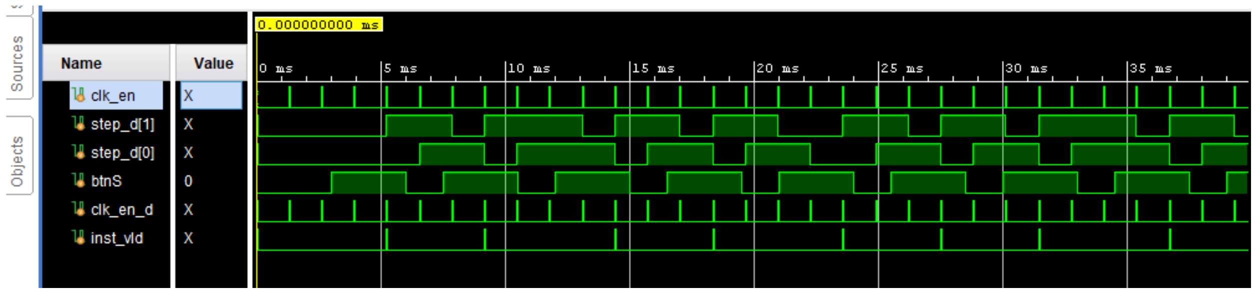
`clk_dv` is 0 during the clock cycle that `clk_en` is high.
The code, "`clk_en <= clk_dv_inc[17]`", implies when a 17-bit `clk_dv` register overflows, `clk_en` goes high and simultaneously `clk_dv` is reset to 0 for the next counting cycle.

IV. Draw the diagram: circuit involving the signals clk_dv, clk_en, and clk_en_d.
It should be a translation of the corresponding Verilog code.



Section 3B: Debouncing

Screenshot: *waveform that clearly shows the timing relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld*



Name	Value
clk_en	X
step_d[1]	X
step_d[0]	X
btnS	0
clk_en_d	X
inst_vld	X

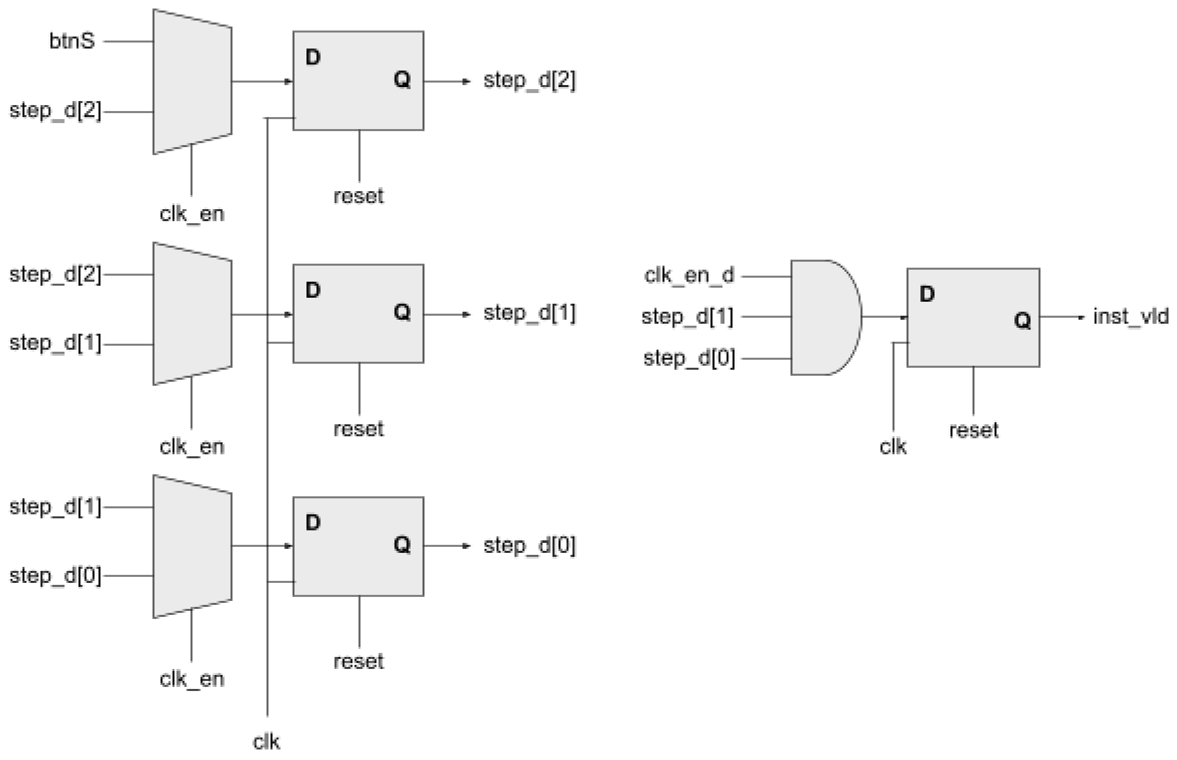
I. What is the purpose of the clk_en_d signal when used in expression ~step_d[0] & step_d[1] & clk_en_d? Why don't we use clk_en?

`clk_en_d` provides a brief delay to make sure that any changes in the button press state (step_d) are fully registered before acting on them, helping to keep things in sync.

II. Instead of clk_en <= clk_dv_inc[17], can we do clk_en <= clk_dv[16]? Why?

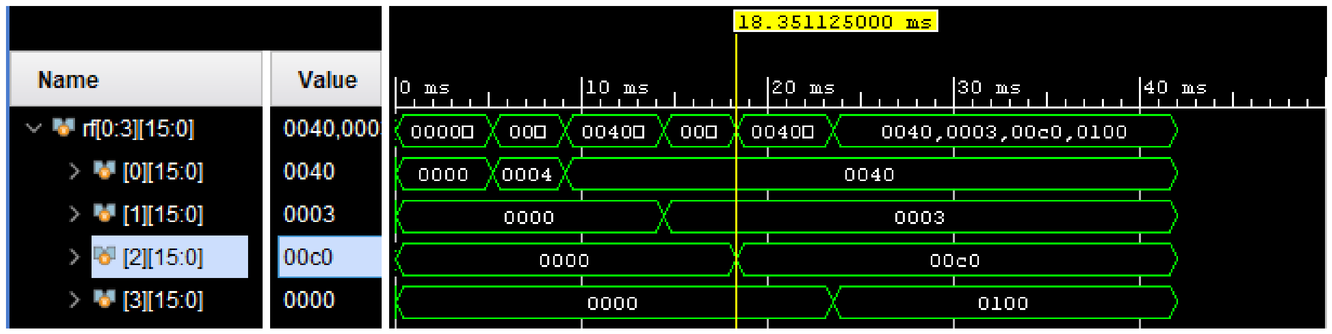
We generally can't directly replace `clk_en <= clk_dv_inc[17]` with `clk_en <= clk_dv[16]` without altering the behavior significantly. Using `clk_dv_inc[17]` is intended to capture an overflow, which happens once every 2^{17} cycles for a 17-bit counter, thus toggling the `clk_en` signal less frequently. This setup is used to generate a lower frequency timing signal. However, switching to `clk_dv[16]`, which toggles every 2^{16} cycles, would double the frequency of `clk_en` as it halves the period of the toggle compared to using the overflow bit.

III. Draw the diagram: circuit involving the signals 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.*



Section 3C: Register file

Screenshot: *a waveform for the first time register with index 2 (3rd) is written with a non-zero value***.**



I. Find the line of code where a register is written with a non-zero value and copy it in your report.

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

II. Find the lines of code where the register values are read out from the register file and include it in your report.

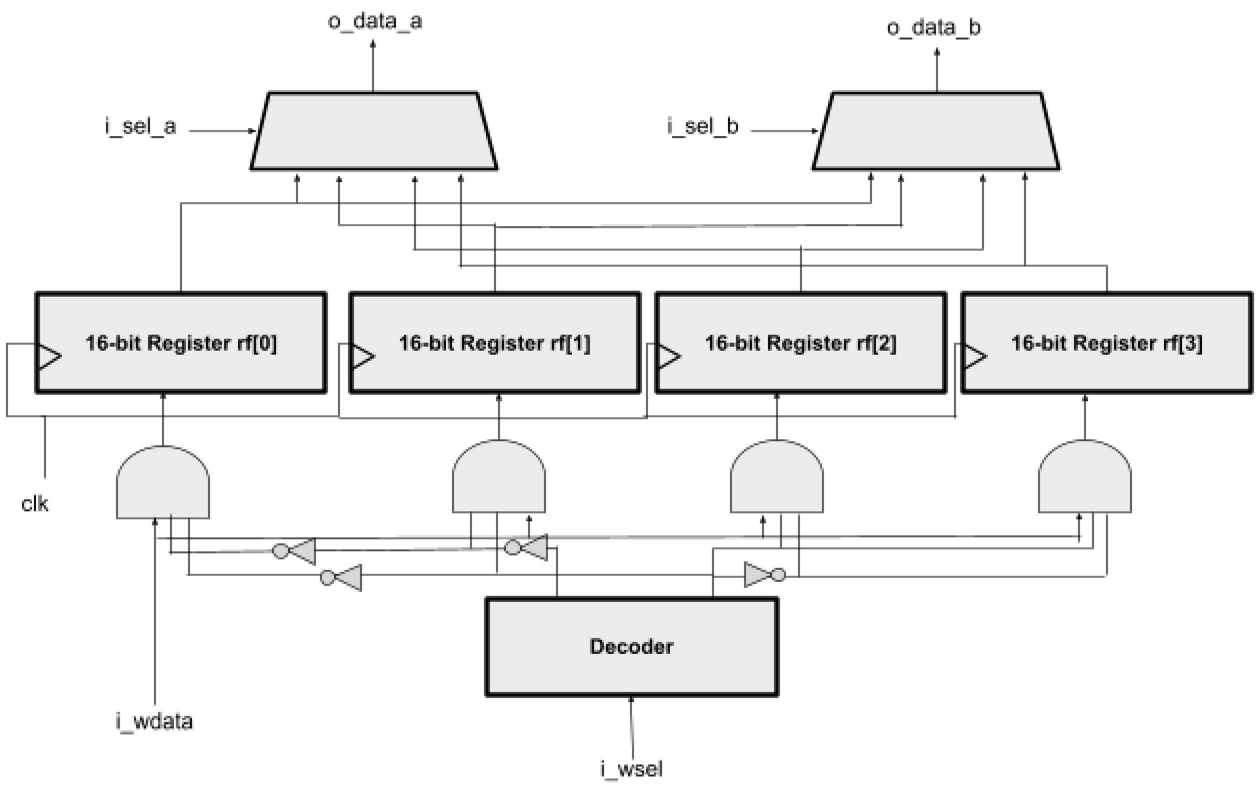
```
    assign o_data_a = rf[i_sel_a];
    assign o_data_b = rf[i_sel_b];
```

III. Is this sequential or combinational logic? If you were to manually implement the readout logic, what kind of logic elements would you use?

The line `rf[i_wsel] <= i_wdata;` is a sequential logic because it involves a register and is clocked, which means the action of writing to the register depends on the state of the clock and potentially previous states.

The lines `assign o_data_a = rf[i_sel_a];` and `assign o_data_b = rf[i_sel_b];` are combinational logic, as the output at any moment is determined by the current values of the selection signals `i_sel_a` and `i_sel_b`.

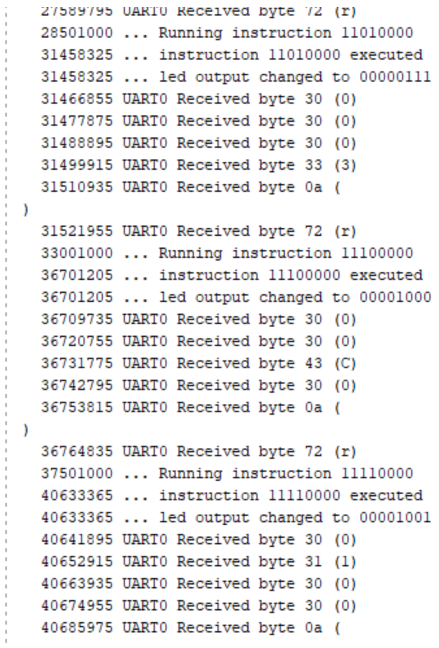
IV. Diagram: the register file block.
It should be a translation of the corresponding Verilog code.



Part 4: Modifying the Testbench

Section 4A: Pretty Console

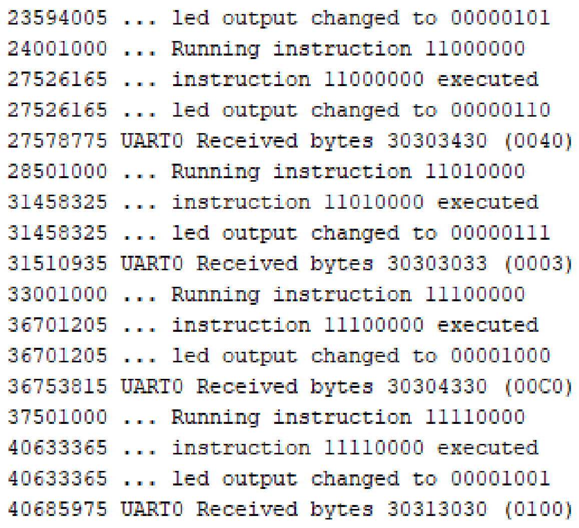
Console screenshot of the initial problem:



```
  27589795 UART0 Received byte 72 (r)
  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 72 (r)
  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 72 (r)
  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 (
```

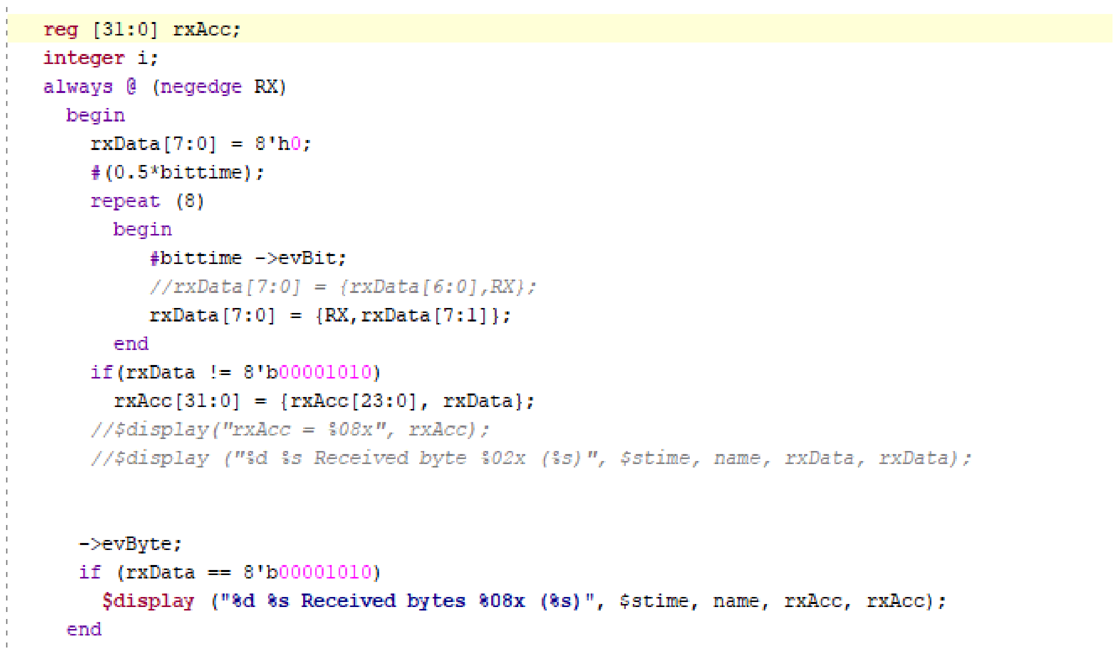
Looking at the above screenshot, we can see that there is an entire message printed out for every single byte received, even when they are part of the same word. It would be much easier to read this if each group of messages was combined into only a single log.

Our solution & console screenshot:



```
23594005 ... led output changed to 00000101
24001000 ... Running instruction 11000000
27526165 ... instruction 11000000 executed
27526165 ... led output changed to 00000110
27578775 UART0 Received bytes 30303430 (0040)
28501000 ... Running instruction 11010000
31458325 ... instruction 11010000 executed
31458325 ... led output changed to 00000111
31510935 UART0 Received bytes 30303033 (0003)
33001000 ... Running instruction 11100000
36701205 ... instruction 11100000 executed
36701205 ... led output changed to 00001000
36753815 UART0 Received bytes 30304330 (00C0)
37501000 ... Running instruction 11110000
40633365 ... instruction 11110000 executed
40633365 ... led output changed to 00001001
40685975 UART0 Received bytes 30313030 (0100)
```

Now that we made our changes, it is clear that the logs are much more readable, and it is far more clear when one word ends and another begins. They also take up less space on the screen so we can see more of the logs at once.

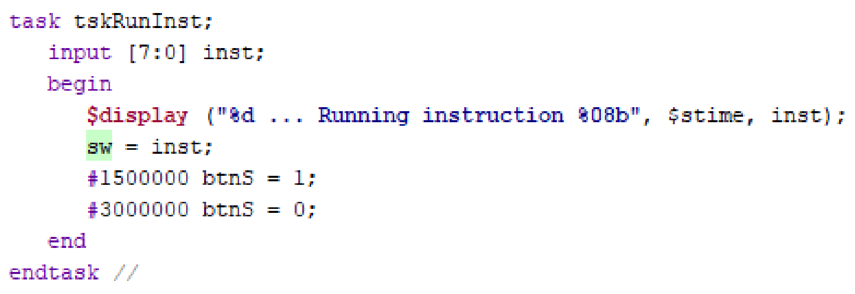


```
reg [31:0] rxAcc;
integer i;
always @ (negedge RX)
  begin
    rxData[7:0] = 8'h0;
    #(0.5*bittime);
    repeat (8)
      begin
         #bittime ->evBit;
         //rxData[7:0] = {rxData[6:0],RX};
         rxData[7:0] = {RX,rxData[7:1]};
      end
    if(rxData != 8'b00001010)
      rxAcc[31:0] = {rxAcc[23:0], rxData};
    //$display("rxAcc = %08x", rxAcc);
    //$display ("%d %s Received byte %02x (%s)", $stime, name, rxData, rxData);


   ->evByte;
   if (rxData == 8'b00001010)
     $display ("%d %s Received bytes %08x (%s)", $stime, name, rxAcc, rxAcc);
  end
```

Shown above is the code we changed in order to fix this issue. We added a new 32 bit register that can store up to 4 characters, and only print the contents when we see a \n character, or ASCII code 00001010 in binary. The demo for this section consisted of showing the code and the output to the TA.

Section 4B: Load Different

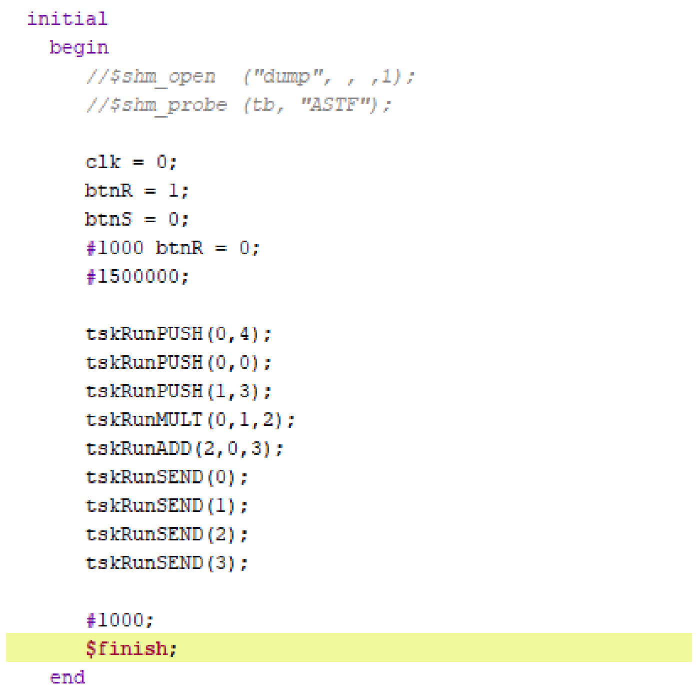
I. Find the core, most general function that is feeding instructions into the basys3 module.



```
task tskRunInst;
   input [7:0] inst;
   begin
      $display ("%d ... Running instruction %08b", $stime, inst);
      sw = inst;
      #1500000 btnS = 1;
      #3000000 btnS = 0;
   end
endtask //
```

`tskRunInst` is the most general function because all of the specific instructions (PUSH, MULT, ADD, AND SEND) use it in order to send which instruction will be run, as well as the arguments, to the basys3 module.

II. Which instructions *(e.g. ADD R0 R1 R2)* are fed into the UUT?



```
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
```

Specific instructions are specified using the tskRunPUSH/ADD/MULT/SEND functions. Each of these functions are themselves calls to tskRunInst, with specific information about the instruction (opcode, number of registers, if there is an immediate) hardcoded to reduce the size of the general function. The instructions shown above are as follows:

```
PUSH R0 0x4
PUSH R0 0x0
PUSH R1 0x3
MULT R0 R1 R2
ADD R2 R0 R3
SEND R0
SEND R1
SEND R2
SEND R3
```

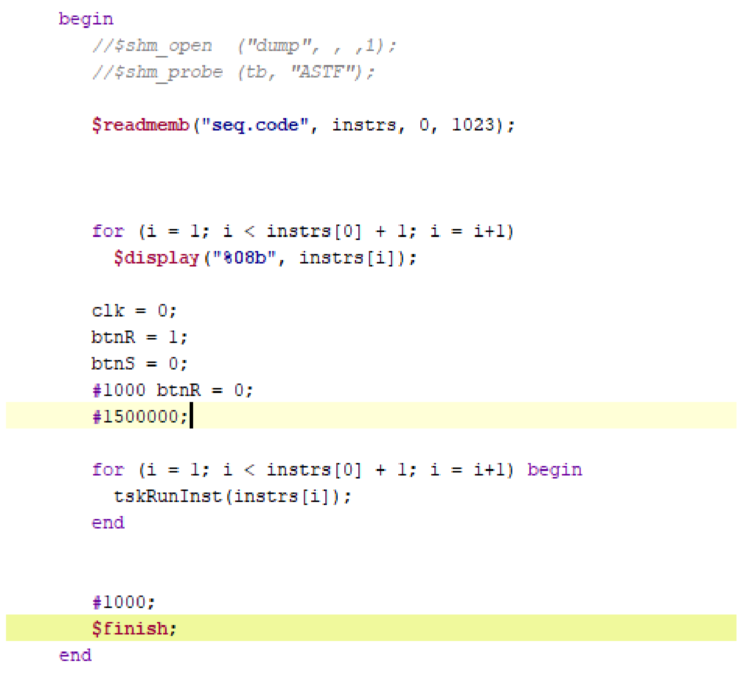
These are the same instructions we used earlier in Parts 1 and 2.

III. Modify the testbench such that it loads seq.code into an array, and executes every instruction in the file. *Explain the disadvantages of the initial system and modified testbench*.

The initial testbench system hardcoded the instructions shown above into the testing process, meaning there is no way to test other sequences of instructions.

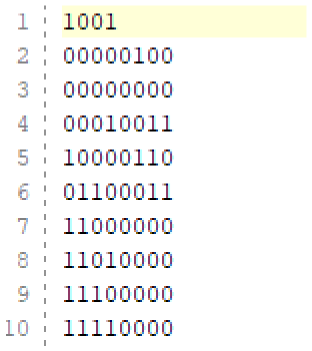
Our solution was to use the $readmemb function to read a file named seq.code in the same directory as the testbench, fill out a 2d register array with the information in the file, and run the testbench using this user-generated sequence of instructions instead. This way, the sequence is easily changeable and the user can test more sequences easily. This approach automates the process of testing different sequences so that we don't manually input each time.

New code:



```
begin
   //$shm_open  ("dump", , ,1);
   //$shm_probe (tb, "ASTF");

   $readmemb("seq.code", instrs, 0, 1023);



   for (i = 1; i < instrs[0] + 1; i = i+1)
     $display("%08b", instrs[i]);

   clk = 0;
   btnR = 1;
   btnS = 0;
   #1000 btnR = 0;
   #1500000;

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


   #1000;
   $finish;
end
```

seq.code file contents:

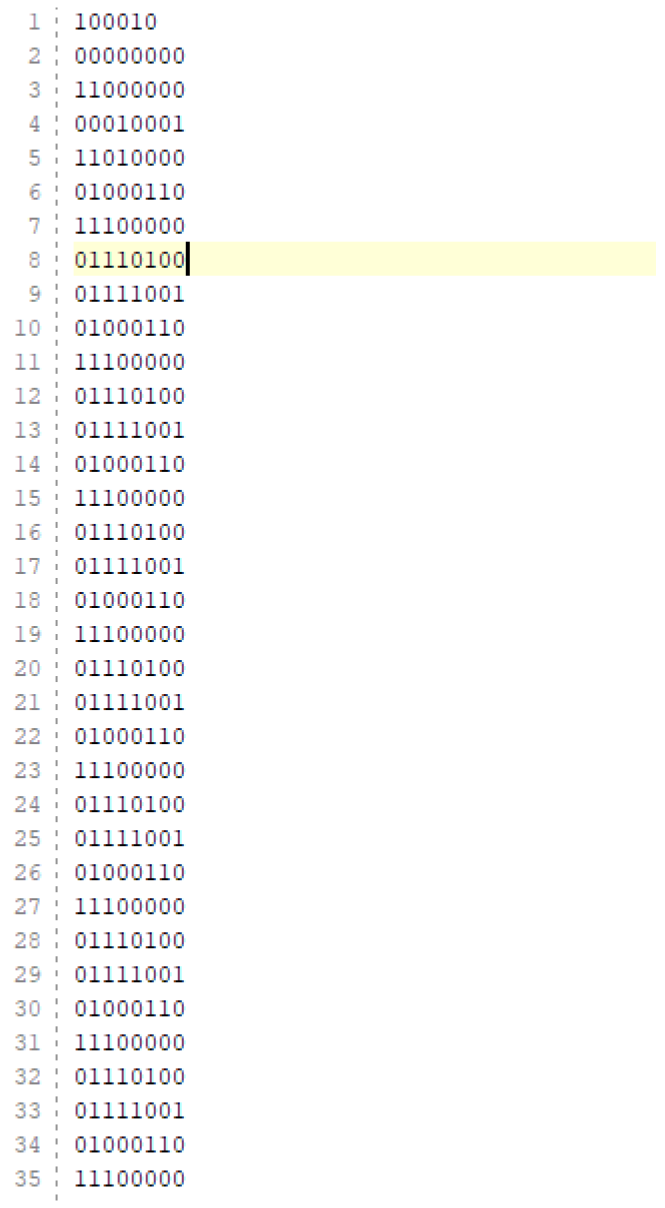


```
1001
00000100
00000000
00010011
10000110
01100011
11000000
11010000
11100000
11110000
```

The demo for this section consisted of showing TA Chenda our output in the console, and the new code and seq.code files we wrote.

Section 4C: Fibonacci

I. Based on the 2 previous changes, write new contents for seq.code such that it prints out the first 10 Fibonacci numbers. *Explain seq.code file.*



```
100010
00000000
11000000
00010001
11010000
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
01110100
01111001
01000110
11100000
```

For the Fibonacci sequence, we began by pushing 0 into R0 and 1 into R1 and printing out the values using Send. We then shifted the value of R1 into R0 and R2 into R1 by adding them with 0, which is stored in R3, and storing it in the correct register. Then, we used the Add Instruction to add the two previous Fibonacci numbers in R0 and R1 and placed the result in R2. We used the Send instruction to print the result, repeating until 10 numbers were ouput.

Conclusion

In this lab, we worked with instruction encoding for a miniature assembly language, translated instructions into binary, operated an FPGA to run these instructions and saw the output on a PuTTY window, observed and analyzed the results and waveforms of the testbench, and made changes to the testbench code to make its output more readable and its input easier to change.

Some things we learned by working in this lab are how to generally use an FPGA, Verilog, PuTTY, and Vivado, how to understand instruction encoding, and how clock signals work and what their waveforms look like.

Specifically, in Part 3, we learned how to analyze waveforms to understand what signal frequencies stand for and their timing relationships. We figured out that the clk_en_d signal introduces a delay so that button press states are accurately registered, which is debouncing. In addition, we worked with a register file and learned how data is written and read from registers, distinguishing between sequential and combinational logic.

In Part 4, we learned how data is concatenated and assigned. With this knowledge, we modified the existing code so that the system works in the way we wanted. We dealt with several instructions and used the “readmemb” function to manipulate the code and generalize the system, and then generated seed data for testing purposes, achieving an overall good approach to test the data more easily and efficiently.

This lab was relatively problem-free, other than learning to use Vivado and Verilog, and not knowing when to use PuTTY and the FPGA for our outputs. The diagrams for Part 3 were also difficult to produce. Other than that, it went smoothly.