CS M152A Lab 2

Michael Wu, Haoyu Yun, Jennie Zheng

Introduction and Requirement

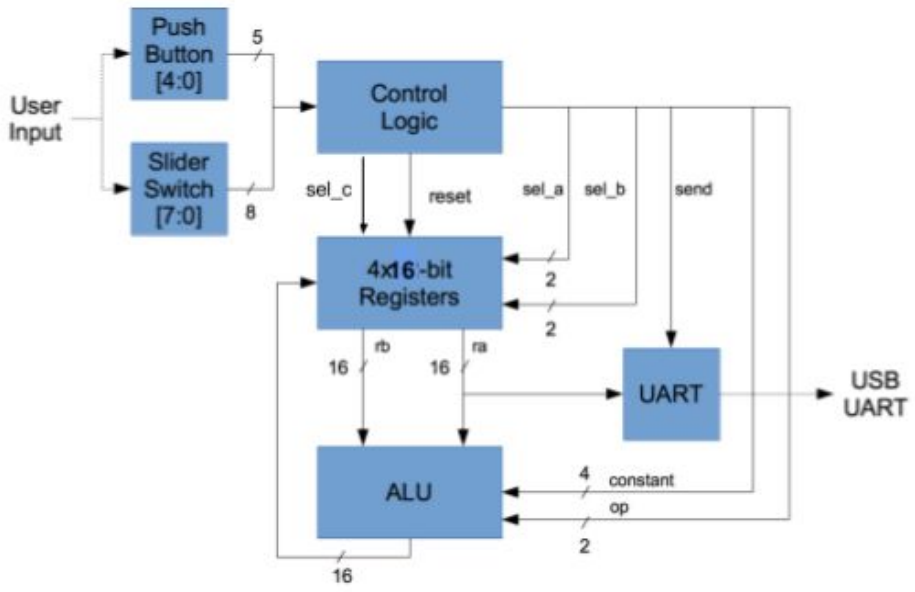
We build a sequencer to interpret and execute 8-bit instructions encoded by 8 switches (1 -- up, 0 -- down).

These instructions allows the user to PUSH into one of four registers an initial value up to 15, ADD or MULT values to be stored in the registers, and SEND a register's value to be printed through the UART.

The first two bits specifies the type of operation and the remaining bits specify 2-bit register identities or 4-bit constants (in the case of PUSH).

The instruction may be entered with the center button, and all variables may be reset with the right button. A sequence of 8 LEDs track the number of instructions executed since the previous reset (mod 256).

We added multiple features to improve the functionality output of our sequencer. These include the Multiply module, improved output from the UART, and the ability in the testbench to load instructions from a text file. We demonstrate this ability by writing a text file with a set of instructions which causes the first 10 Fibonacci numbers to be printed.



Project Diagram

Design Description

We implement several additional functionalities to the given sequencer template.

Multiply Operation

The Multiply module (in the **seq_mul.v** file) allowed us to implement the MULT instruction, analogous to the ADD instruction in behavior and implementation. When the MULT operation (10) is specified in bits 7 and 6 of the instruction, the value of two source registers are multiplied together and stored in a third destination register (specified by bits 1 and 0).

UART Output

The UART output is modified so that a register's value (in binary digits) are given in 1 line. A register is used to store all digits, and previous digits are shifted to the left as new digits are added, until a '\n' signals the end of the value. The **model_uart.v** file was modified and the key additions are shown in the code below.

```
31      always @ (negedge RX)
32        begin
33           rxData[7:0] = 8'h0;
34           #(0.5*bittime);
35           repeat (8)
36             begin
37                #bittime ->evBit;
38                //rxData[7:0] = {rxData[6:0],RX};
39                rxData[7:0] = {RX,rxData[7:1]};
40             end
41           ->evByte;
42           if(rxData==8'h0a) begin
43             $display ("%s", fullData);
44             fullData = 32'b0;
45           end
46           else begin
47             fullData = fullData<<8;
48             fullData[7:0] = rxData;
49           end
50        end
51
```

We additionally modified the UART output to indicate the register number when its value is printed (e..g "R2:0010"). We modified **uart_top.v** to add three additional states in the outputting process to accommodate for the three additional characters of output. We modified **nexys.v** to include the register's number among the outputs of the **uart_top** module.

Instruction Loading

We added the **readLine** task in the testbench to read and process lines of instructions from a text file (named seq.code), running the existing tskRunPUSH, tskRunADD, tskRunMULT, and tskRunSEND tasks accordingly.

```
69      task readLine;
70          input [7:0] line;
71             // Travers through the insn lines
72          begin
73          if (line[7] == 0 && line[6] == 0)
74              tskRunPUSH(line[5:4], line[3:0]);
75          else if (line[7] == 0 && line[6] == 1)
76              tskRunADD(line[5:4], line[3:2], line[1:0]);
77          else if (line[7] == 1 && line[6] == 0)
78              tskRunMULT(line[5:4], line[3:2], line[1:0]);
79          else
80              tskRunSEND(line[5:4]);
81           end
82       endtask
```

We demonstrated its functionality through a text file of instructions. When all instructions are processed, the first 10 Fibonacci numbers are printed through the UART.

Simulation Documentation

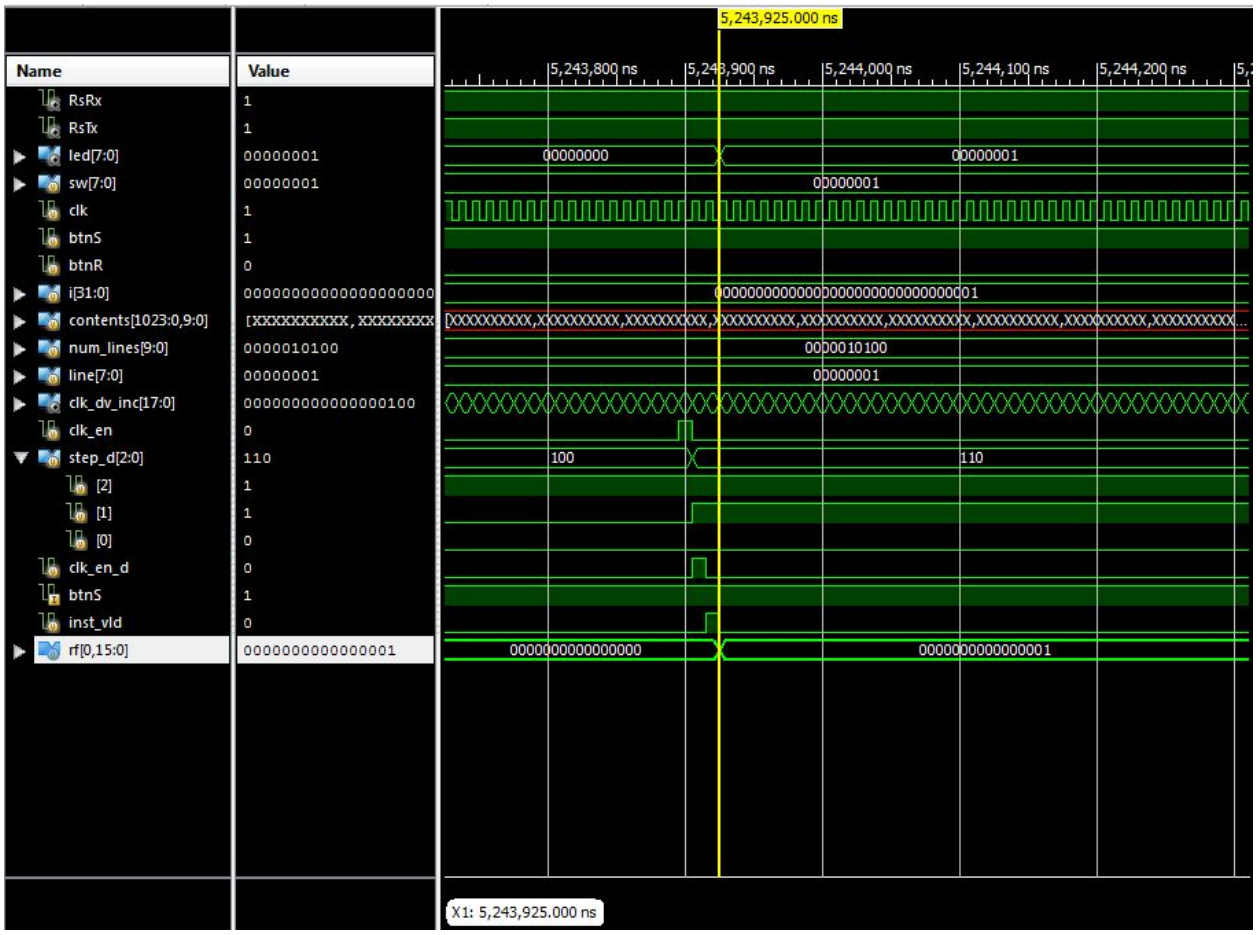
Against our TA's advice, we tested most of our modules using the FPGA board rather than using test benches. We realized, after wasting a lot of time waiting for the software to build and synthesize our code, that test benches was a much more time-efficient test strategy.

We decided to test our code after every major design step in order to ensure that the code worked properly. We implemented the simple multiply operation module without any bugs since it was simply a matter of copying the add operation.

We took a while, however, to fully understand the UART states' code and implement the Nicer UART output step. First, we faced a bug in which the code was only iterating through six states rather than nine states because we did not properly initialize the state variable. Later on, we experienced the problem where we were overwriting our first value and not writing into the other values, because we forgot to shift our registers properly in uart_top.

Finally, when implementing instruction loading, we faced numerous compile errors when attempting to use a double array. Eventually, we looked at online examples of double arrays and figured out the proper way to address them.

In this lab, we realized that most of our bugs were from lack of understanding of either Verilog syntax or Verilog documentation. We can often save time by spending more time reading on Verilog rather than rushing to write out the code. Furthermore, we learned that Git version control is invaluable for us to track our progress and find bugs. Whenever we accidentally introduce a program-breaking bug, our best bet at finding the bug is to check out what we've changed since our last git commit.



Waveform of successful ALU program.

Conclusion

We improved a sequential circuit which functioned as a simple ALU. First, we implemented a multiply instruction to complement the already existing add instruction. Furthermore, we improved the UART by first writing the register binary output in one line and then adding a register number before the register value. Lastly, we programmed the sequential circuit to load instructions from a specified text file (seq.code) before read input interactively from switches and buttons.

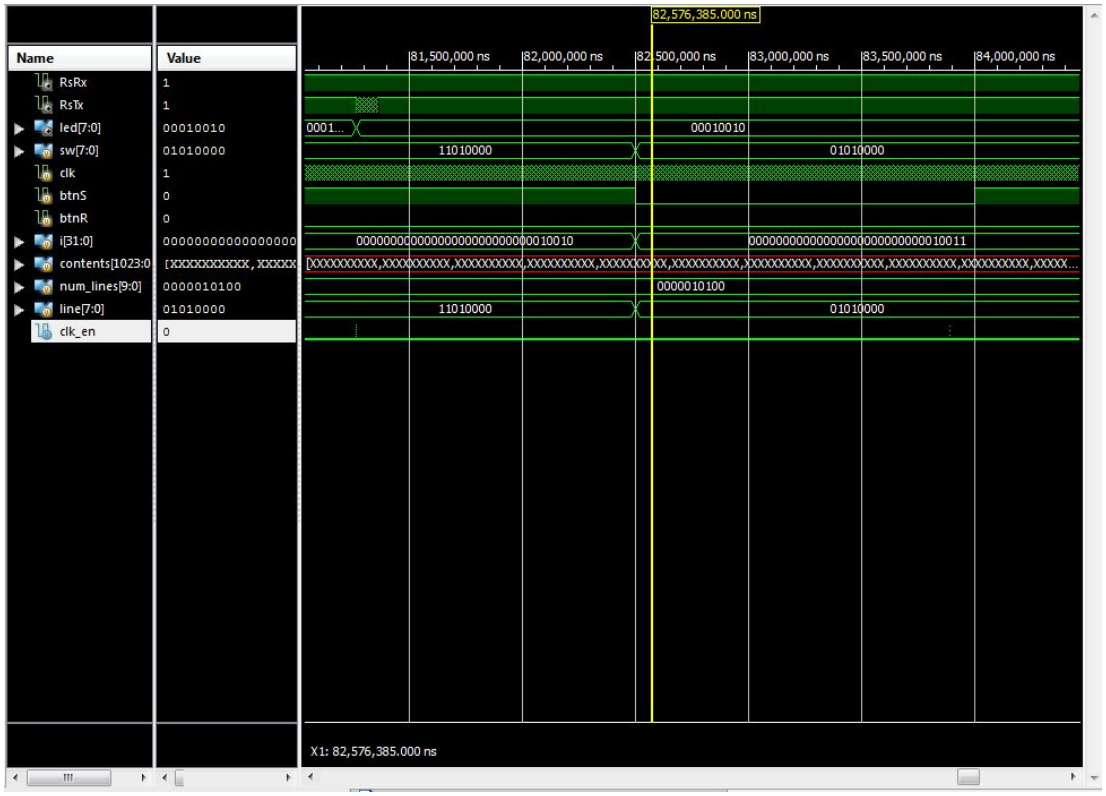
We wrote the circuit in Verilog. We split the task up into modules, which we combined to generate our ALU and registers system.

The primary difficulties were in familiarizing ourselves with the skeleton code, the UART protocol, as well as with Verilog file I/O system. After learning these pieces, it was simply a matter of testing and debugging our code until it worked. We were satisfied with our modular design of the circuit.

Workshop 2 Questions

Implementation [Clock Dividers]

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.



$Period_{clk\_en}$ = 2^17 * $Period_{clk}$= 131072 clock cycles of master clock clk
= 1310720ns = 1.3ms
(Note that $Period_{clk}$= 10ns.)

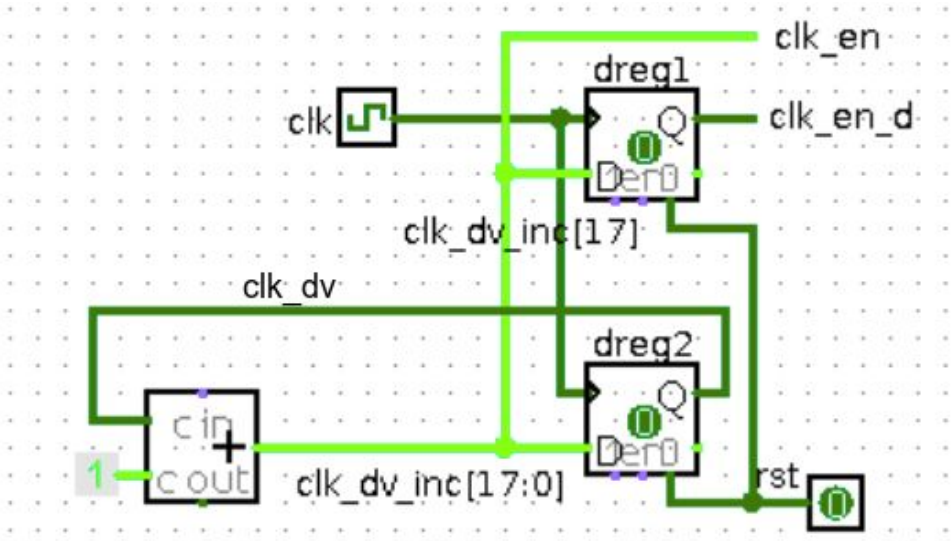
2. A duty cycle is the percentage of one period in which a signal or system is active: D = ×100% , where D is the duty cycle, T is the interval where the P T signal is high, and p is the period. What is the exact duty cycle of clk_en signal?

T = 1 clock cycle
P is stated above
Duty Cycle is (T/P)*100% = **(100 / 2^17) %**

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

   When the signal is high, clk_dv = **17b'00000000000000001**

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, such as the following. NOTE: for other lab reports you don't have to draw diagram with this much detail. A diagram outlining high level module interactions is sufficient.



Implementation [Debouncing]

1. What would be the impact if we use clk_en instead of clk_en_d in the statement at line 102 of nexys3.v?

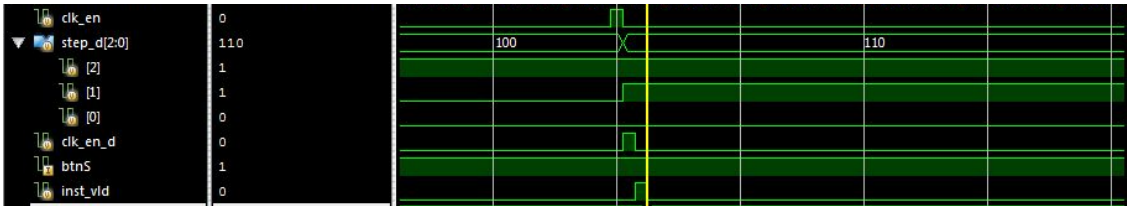
clk_en_d is delayed by one clock cycle. If we used clk_en, the values of step would not be set properly and the incorrect value may be stored in inst_vld via is_btnS_posedge.
Notice in the waveform that when step_d changes, clk_en is set to 0 while the delayed clk_en_d stays at 1 for the next cycle. In other words, step_d and thus is_btnS_posedge would be different values when clk_en is 1 compared to when clk_en_d is 1, and incorrect values would be stored to inst_vld.

2. Instead of clk_en <= clk_dv_inc[17] (line 76), can we instead use a 50% duty cycle clk_en by assigning the first bit of clk_dv to clk_en (clk_en <= clk_dv[16])? Why?

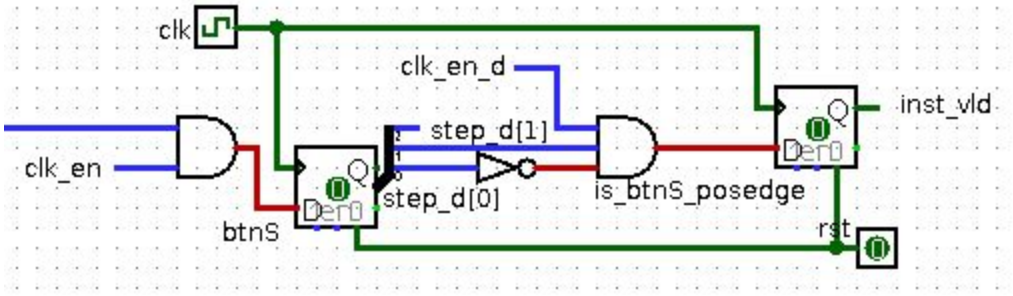
Yes, you can. Clk_en increments by 1 every master clock cycle. Therefore, its most significant bit will be 1 exactly half of the time. Therefore, you can create a 50% duty cycle clk_en by assigning the first bit of clk_dv.

In order to use clk_en with the same period but a 50% duty cycle, you'll have to modify the Verilog code in order to detect the positive edge of clk_en rather than detect that clk_en is high.

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.



[Implementation] 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.

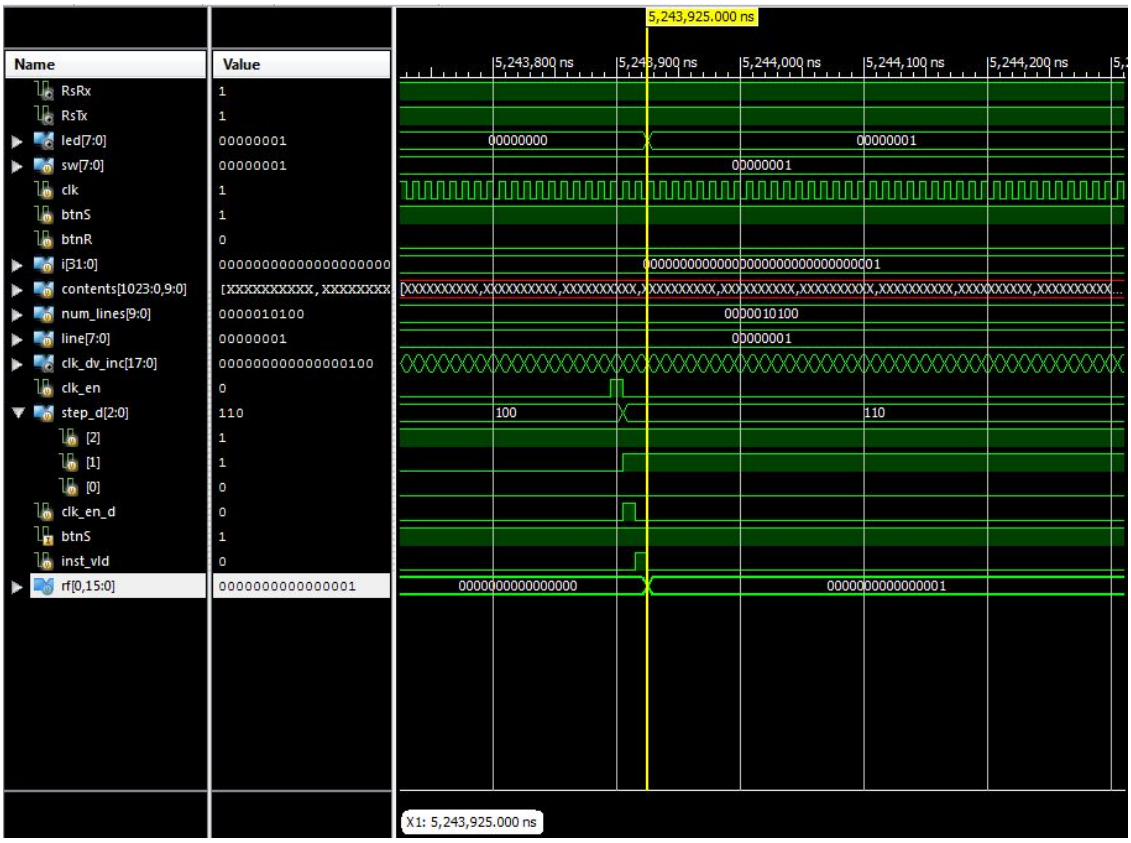
1. Find the line of code where a register is written a non-zero value. Is this sequential logic or combinatorial logic?

In line 33, rf is written to a non-zero value.
This is sequential logic because it stores the data within an always @ (posedge clock) segment.

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?

In lines 35-36, rf is read out into output variables o_data_a and o_data_b.
It's combinational logic because it uses unconditional assign statements.

3. Capture a waveform that shows the first time register 0 is written with a non-zero value.



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