**Introduction**

The aim of this lab was to build a sequencer. A sequencer is a simpler version of the processor that we have installed in most electronics today. The sequencer is able to perform 4 different tasks. It can push a value to a register (memory), add values in two registers, multiply values in two registers and display the value of a register. Each of these tasks has a different binary encoded operation code that tells the FPGA what operation to perform with the current registers. A base code was provided to us and the aim of the lab was to modify this to perform certain tasks.

The first task was to implement the multiplication operation. The next step involved buffering the output to Putty and displaying it when a carriage return is received from the Verilog code. The output operations are only to output the value of a specific register to the console (Putty if running on FPGA), or the Xilinx console if we’re simulating the code. Our third task was to display what register we were outputting in addition to the value of the register since it provides some more meaningful debugging data. Up till now, while simulating we had been manually calling functions to perform operations on the sequencer. The next task was to get it to read operation codes from a text file. The final task was to generate the first 10 numbers of the Fibonacci series using operation codes and feeding that into the sequencer.

**Design Description**

The topmost sequencer module, seq.v, creates an instance of the alu.v module and passes in the inputs to this alu module. It then outputs this to the register file if it is a multiplication, addition or push operation through the seq\_rf module. The seq\_rf module has a 1-bit signal seq\_valid\_in, that is set to true any time the input from to the alu is valid.

Implementing the Multiplication module

The multiplication module was based off the addition module that was already implemented in the files given. The way the addition module works is by getting two 16-bit inputs and then doing a simple addition operation and then outputting it. The output is set to valid if the input is considered valid. Similarly, the multiplication module inputs two 16 bit numbers, does a simple multiplication using the Verilog \* operator, and outputs this result truncated to a 16 bit number. The output is considered valid if the input is valid. For this task, a change had to be made in seq.v though, involving the seq\_valid\_in signal. Any time that the operation code is a push, add or multiplication, the seq\_valid\_in signal needs to be set to true so that the input can be passed to the alu. The only change we made to this module is to add the code that would set this signal to true for multiplication operations.

Nicer UART Output

The aim of this task was to buffer the output of the SEND operations. Initially what was happening that the value of each register was sent 4 bits at a time. We needed to change this so that the data is sent to the console when a carriage return (\r) was received. The uart\_top.v module implements the buffering by using a Finite State Machine. The Finite State machine for this task has 7 different states. These states distributed as 1 idle state, 4 states for receiving output values, and 2 states for carriage return and new line characters.

(Maybe draw out the FSM by hand on paper and insert image here?)

FILL BY ANDREW

Even Nicer UART Output

The aim of this task was to add the register number to the output when a send operation is received. To do this, we modified the finite state machine to have 3 additional states, to send 3 characters R, the register number, and a colon. Furthermore, the register number is passed in as input from the operation code as a parameter to the uart\_top module.

The FSM works as follows. If the current state of the FSM is the idle state and it receives an input, the current state is changed to stSendR, which adds the letter “R” to the FIFO output. The next state is to send the register as an ASCII value to the FIFO. The bit value of the register, received as an input, is converted to its ASCII representation using the fnNib2ASCII function. After sending this register value, we switch to a state where we send the colon character to the FIFO. Finally, we need to output the register value, which follows the same steps as described in the above section. The state is changed to stNib1, which is the state to read the first 4 bits. The input is then stored in tx\_data. These 4 bits are converted to ASCII hex-representation inside an always block using the given function, fnNib2ASCII and sent to the FIFO output. This process is repeated for the next 3 4-bit inputs. The states are changed continuously to different Nib states. After receiving these 16 bits of input, the FSM changes to the new line state (stNL). A new-line is appended to the FIFO, and the state is again incremented to the stCR; the carriage return state. In this state, a carriage return character is appended to the FIFO. The FSM waits until the FIFO is not completely full, at which point it goes back to the idle state and the entire process is repeated again on receiving the next SEND operation.

(May be include in Simulation section?)

An Easier Way to Load Sequencer Program

Up until now, during simulation, the sequencer was running static instructions. These operations were being executed by calling the specific functions tskRunPUSH, tskRunADD, tskRunMULT and tskRunSEND. These instructions modify the value of the sw register and call the tskRunInst function. For this task, we added code that reads the instructions from a text file and executes the instructions by setting the sw register automatically, and directly calls the tskRunInst instead of going through specific functions. This is done using the Verilog readmemb function, which reads each input line by line. The number of instructions is parsed on the first line, and then the tskRunInst function is repeatedly called.

Maybe answer questions in workshop 1 here?

Fibonacci Sequence

The Fibonacci sequence is a recursive sequence that starts with the numbers 0 and 1 and each of the following numbers is a sum of the previous two numbers. To simulate this

FILL BY ANDREW

**Simulation and Documentation**

**Conclusion**

The aim of this lab was to reverse engineer Verilog code to understand what the given code was doing and incorporate some additional features into this code. The tasks included adding a multiplication operation to the ALU, buffering and displaying the output in a nicer format, and reading instructions from a text-file instead of using static instructions. Each of the tasks involved understanding what the different modules were doing and writing code that would add these features.

One challenge we encountered was with implementing the multiplication operation. Initially we just modified the seq\_mult module, which produced incorrect results. On examining waveforms, we found that the input valid parameter to the ALU was not being set to true. To correct this, we modified seq.v so that the input was considered valid, and was actually written to the register file if we had a multiplication operation. We faced a couple of challenges with “even nicer output.” It took us a while to come up with a way to send the register number. To send the register number, we decoded the operation code and added it as a parameter into the uart\_top module. Additionally, we were displaying the characters “R” and “:” through model\_uart. This was giving the correct result when we simulated the design but didn’t work when we used PuTTY. To correct this, we added more states to the FSM in model\_uart and outputted the characters “R” and “:” through the FIFO. This method worked for both the simulation and implementation.

Overall, this lab was challenging compared to the previous one. It was good practice of reverse engineering code, a practice that is definitely required for working in the industry.

**Workshop 2**

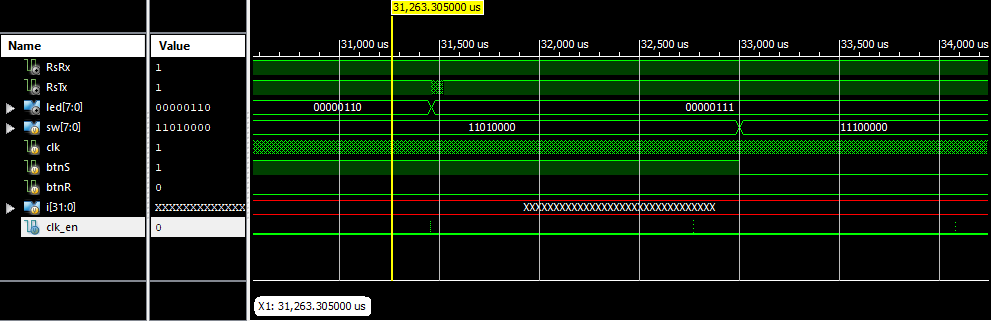
Clock Dividers

1.

Value 1: 32,769.025 us

Value 2: 34,079.745 us

Time Period = Value 2 – Value 1 = 1,310.72 us



2.

Signal High start at time: 31,458.295 us

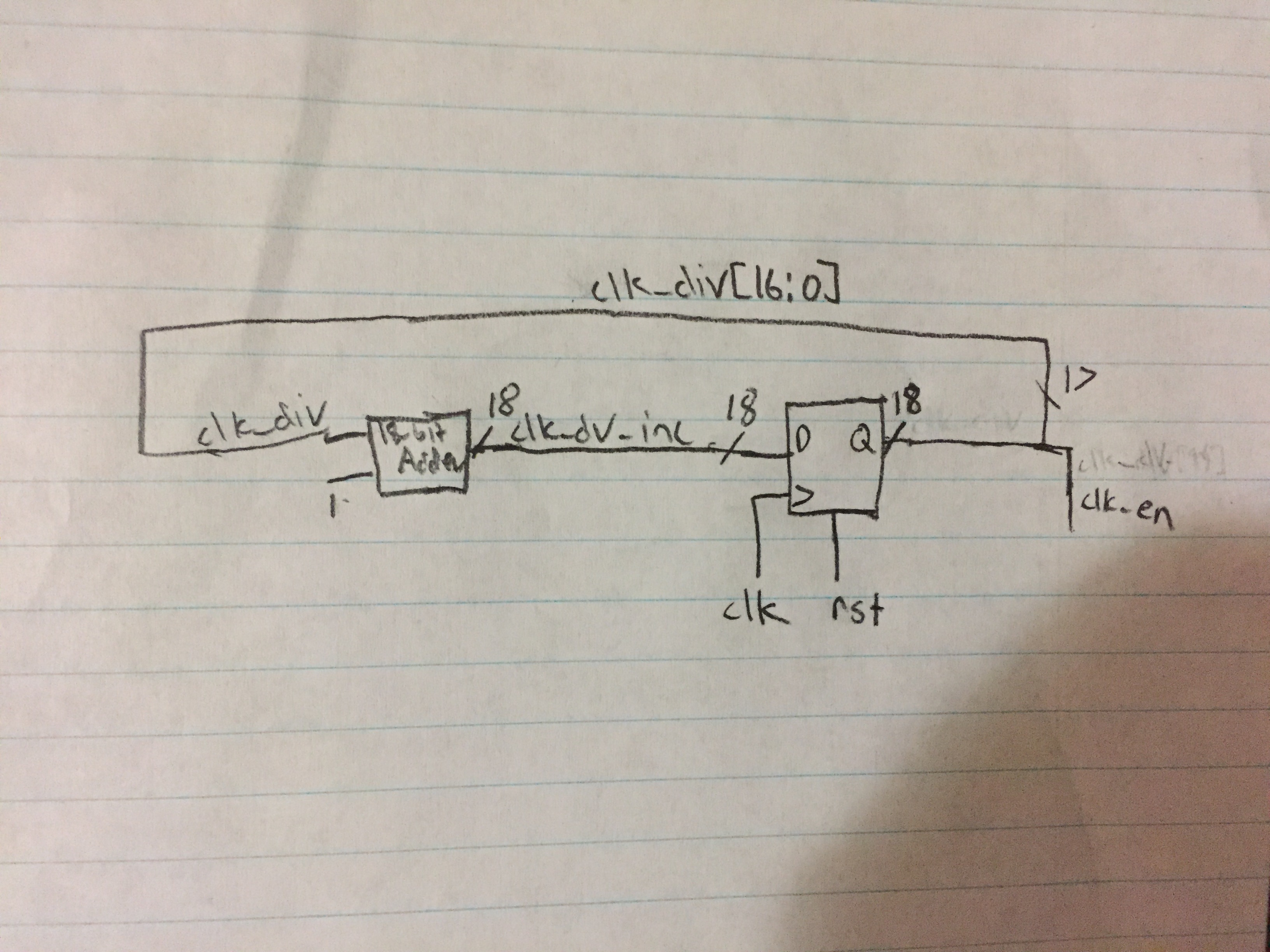
Signal High end at time: 31,458.305 us

T = 0.010 us

Duty Cycle = (0.01 / 1310.72) \* 100

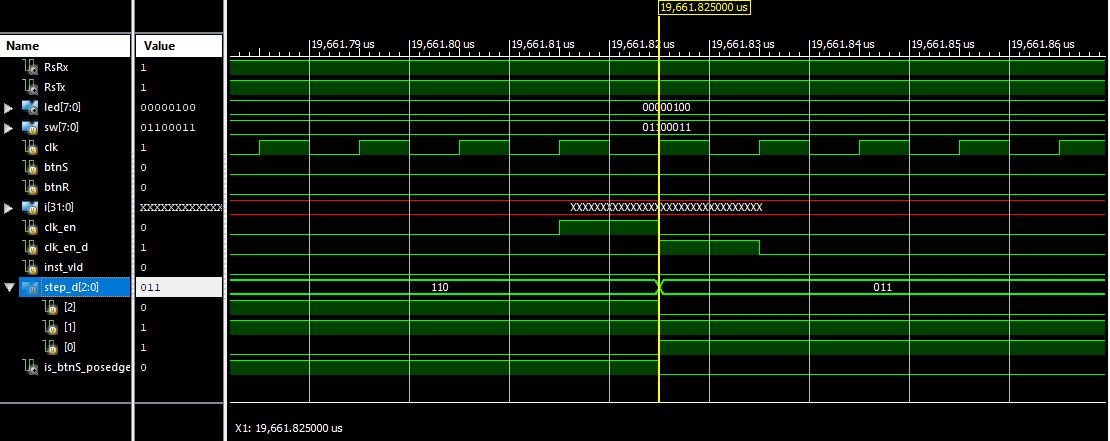
3. The value of clk\_dv is 0 when clk\_en is high

4.



Debouncing

1. If we use clk\_en instead of clk\_en\_d, the inst\_valid signal will be sampled one clock-cycle too early giving an incorrect result. This is due to the fact that we want to make sure that there is a positive transition of btnS. We avoid metastability using edge detection. We check the step\_d signal, which clocks in the transitions of button S and detects the positive edge.
2. If we change clk\_en to be the 17th bit instead of the 18th bit of clk\_div\_inc, then our clock\_en will be functioning at a higher frequency. Due to this, we are sampling our signals at a higher frequency, making the sequencer more susceptible to noise. Although the sequencer is more susceptible to noise, upon running the simulation we found that it still gave the correct output.



Register File

1. A non-zero value is written to the register at line 33 of seq\_rf.v

The line of code is: rf[i\_wsel] <= i\_wdata;

This is sequential logic since its using a clocked always block with non-blocking assignments.

1. The registers are read at line 35 and 36 of seq\_rf.v

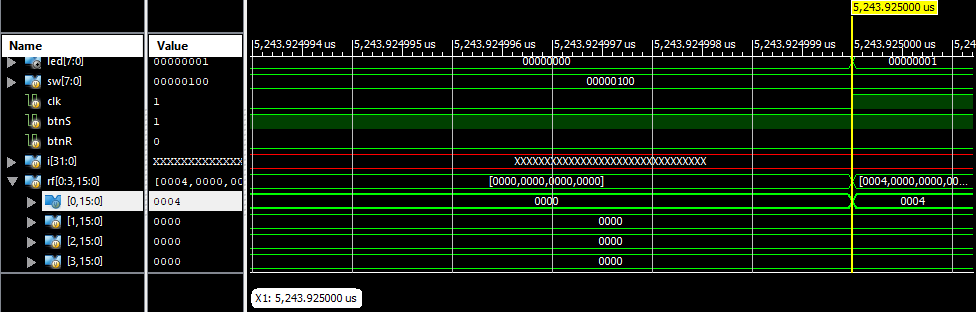
The lines of code that do this are:

assign o\_data\_a = rf[i\_sel\_a];

assign o\_data\_b = rf[i\_sel\_b];

Since the code is using assign statements, it is combinational logic. If we were manually implementing this logic, we’d use multiplexers with i\_sel\_a or i\_sel\_b and rf array as inputs.

3.



4.