CS M152A

Lab 2: Sequencer

Bryan Wong

805 111 517

Teammates: Austin Zhang, Natasha Sarkar

Date: February 3rd, 2020

TA: Srishti Majumdar

Section: Lab 2

1. Introduction and Requirements

In this lab, we were tasked with running a small scale FPGA sequencer project, understanding it, and completing 7 modification tasks on it. We learned styles, formats, and structures for coding in Verilog, as well as useful techniques like clock dividers and debouncers. The baseline sequencer project we were given included four 16-bit general purpose registers that could store values, perform add instructions, and send values to a UART output. In addition, we implemented several features, including a multiply instruction, a dedicated SEND button, a nicer UART output, an easier way to load instructions into the sequencer using a text file input, and writing a series of instructions to write the first 10 Fibonacci numbers.

2. Design Description

From a high-level perspective, the sequencer works by allowing users to send encoded instructions using 8 switches and 2 push buttons that are on the Nexys3 FPGA board. The 8 switches are configured to signal an 8-bit instruction, and the center push button causes the instruction to be executed. Depending on the instruction, the control logic will then send signals to the four registers, ALU, and the UART output. These components carry out various operations such as PUSH, ADD, MULT, and SEND. SEND instructions can be viewed using the USB UART on a Putty terminal output. There is also a dedicated SEND push button that acts as if a regular SEND instruction was invoked, and a RESET button clears all registers.

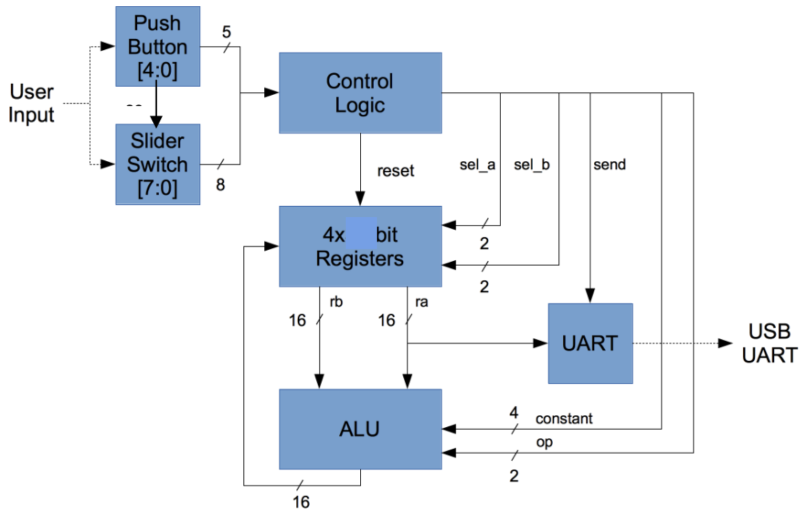


Figure 1: Sequencer Diagram. FPGA switch and button inputs allow us to perform a variety of instructions using four 16-bit registers, an ALU, and a UART output.

The first 2 left-most switches (sw[7:6]) are used to represent the instruction Opcode, which tells the control logic which type of instruction is being completed. Upon receiving the Opcode, the control logic will send the corresponding signals to the registers, ALU, and UART output.

If a PUSH instruction is invoked (Opcode 00), 2 ‘Ra’ bits are used to select one of four destination registers. The ALU left shifts the contents of that register by 4 bits and “pushes” the 4-bit constant indicated in the instruction into the lower 4 bits. This value is then written back to the same destination register.

If an ADD instruction is invoked (Opcode 01), the remaining 6 bits are used to signify 3 registers. The unsigned integer values at registers ‘Ra’ and ‘Rb’ are added in the ALU and written back to register ‘Rc’. This process is the same for the MULT command (Opcode 10), with unsigned integer multiplication performed in the ALU instead.

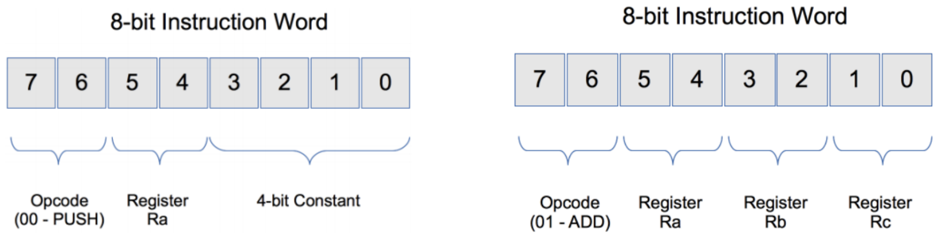


Figure 2: PUSH and ADD Instruction Format. Opcodes of 00 and 01 signal PUSH and ADD, respectively. MULT takes the same form as ADD, but with Opcode 10 instead.

If a SEND instruction is invoked (Opcode 11), the next 2 ‘Ra’ bits are used to select a register whose contents are printed to the USB UART output. The control logic also uses a ‘send’ bit to signal this to the UART module. The 4 least significant bits are not used at all and do not influence the instruction.

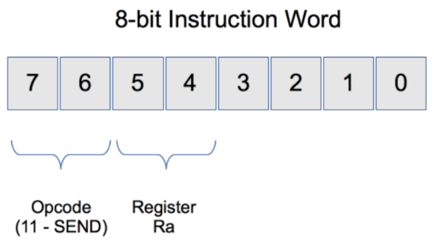


Figure 3: SEND Instruction Format. An Opcode of 11 signals a SEND instruction. The least significant 4 bits are ignored and do not influence the instruction.

Finally, there is a dedicated RESET push button. When pressed, the control logic sends a ‘reset’ signal that clears all four 16-bit registers such that they contain ‘0’.

Implementation Tasks

In order to implement the multiply instruction, we saw that the Opcode for multiplication was defined in the 'seq_definitions.v' file. We created a new multiplication module 'seq_mult.v' styled after the pre-existing 'seq_add.v' module. We then called this module from 'seq_alu.v', adding variables and cases so that multiplication could be selected as an option from the ALU.

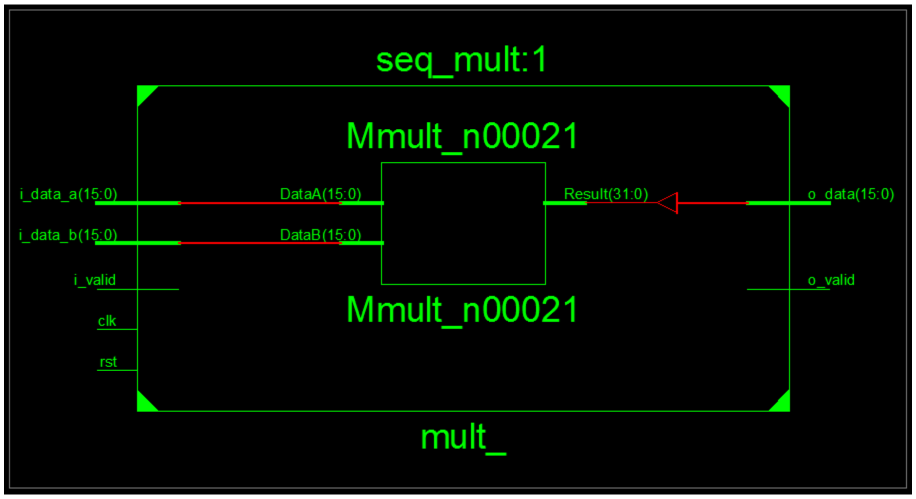


Figure 4: seq_mult Module. Our sequential multiplication module takes in two operands, 'i_data_a' and 'i_data_b', and outputs the result as well as an 'o_valid' bit.

In order to implement the dedicated SEND button, we added an option in 'seq.v' for register file contents to be transmitted if a button press was detected. The 'nexys3.v' top module, after receiving these contents, will send them to the UART USB output. We also implemented button debouncing in the top module using a clock divider to sample the button at a lower rate and only registering a button press on a positive edge.

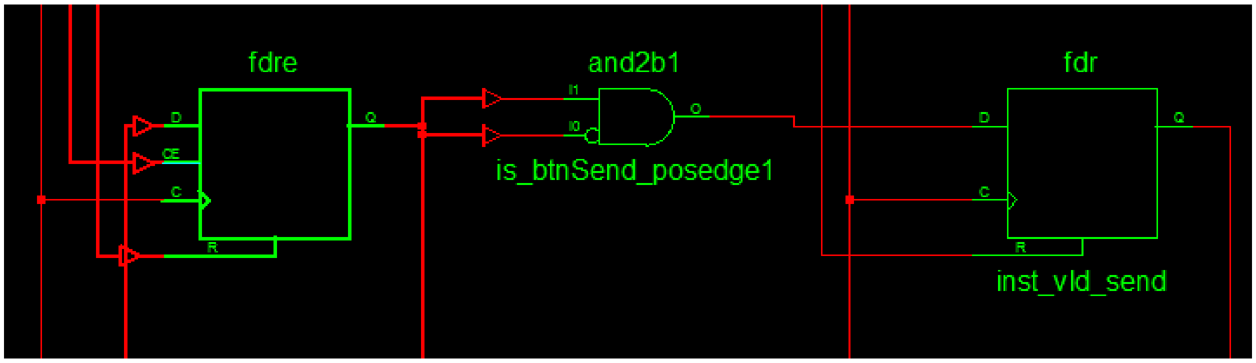
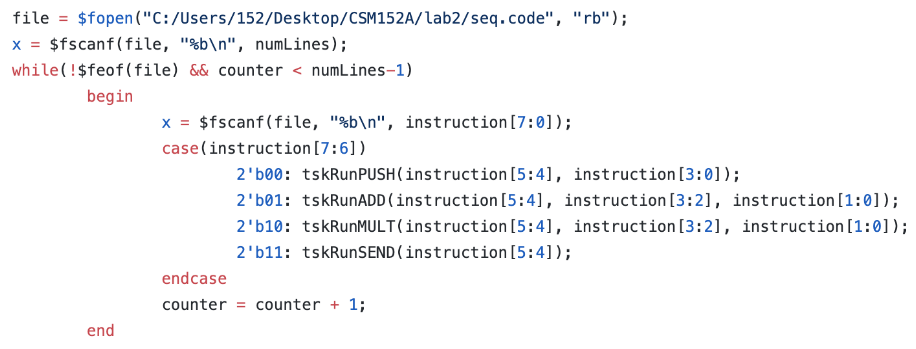


Figure 5: SEND Button Debouncing Schematic. The is_btnSend_posedge is set to 1 when seeing a change in button state from 0 to 1. Flip flops are used to sample the button at a slower rate, to avoid registering multiple button presses.

To add a nicer UART output, we modified our ‘model_uart.v’ testbench file. We added a 40-bit buffer to store up to 5 bytes of data at a time. If a carriage return is received, signalling the end of the data, the program will print “Received Sequence (XXXX)”, where XXXX is the 4-byte contents of a register. Otherwise, the buffer will continue to shift itself to take in the next byte of input data.

In order to print the register number with its data, we also made modifications to the ‘uart_top.v’ module. This module functions by using a state machine to switch from idle, to printing data, to printing newlines and carriage returns. We added additional states to print ‘R’, ‘:’, and the register number while not disrupting the existing UART output. We also added an additional input for the register number, so that the top module can feed this information to the UART module.

Finally, we implemented a way to process text files in ‘tb.v’ and use them as a more convenient way to execute a series of instructions. The testbench will open the ‘seq.code’ file, which includes a first line indicating the instruction count (up to 1024), followed by a series of encoded instructions on each line. We used a counter to process only the number of lines specified in the first line of the file, invoking the corresponding tskRun function for each instruction. If the end of file is encountered or the counter reaches the specified limit, the instruction reading process is terminated.



```
file = $fopen("C:/Users/152/Desktop/CSM152A/lab2/seq.code", "rb");
x = $fscanf(file, "%b\n", numLines);
while(!$feof(file) && counter < numLines-1)
        begin
                x = $fscanf(file, "%b\n", instruction[7:0]);
                case(instruction[7:6])
                        2'b00: tskRunPUSH(instruction[5:4], instruction[3:0]);
                        2'b01: tskRunADD(instruction[5:4], instruction[3:2], instruction[1:0]);
                        2'b10: tskRunMULT(instruction[5:4], instruction[3:2], instruction[1:0]);
                        2'b11: tskRunSEND(instruction[5:4]);
                endcase
                counter = counter + 1;
        end
```

Figure 6: Instruction File Processing. The program reads the file line by line, using the instruction opcode to choose a tskRun function to call with corresponding arguments.

3. Simulation

To test the baseline sequencer before we made any modifications, we ran the 'tb.v' on iSim with the following preloaded instructions:

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

This resulted in the proper output, with R0 containing 64, R1 containing 3, R2 containing 192, and R3 containing 256. The simulation waveforms can also be viewed in the following figure:

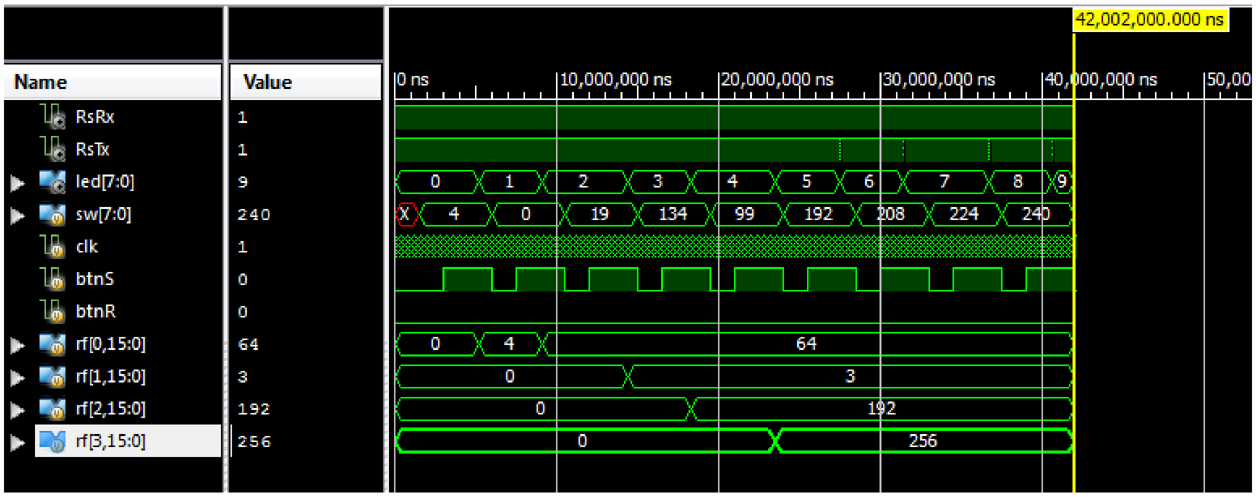


Figure 7: Test Simulation Waveforms. Each of the 4 register files' contents are shown.

We also made sure to test each of the commands that were implemented: PUSH, ADD, MULT, and SEND on each of the four available registers. We also made sure to test for overflow when performing the unsigned integer ADD and MULT instructions to see that they were properly truncated. We also made improvements to the testbench's UART output by enabling it to print out the register's number followed by its contents in an easy-to-read 4-byte format. We also tested the file input function by giving the program a variety of seq.code test cases. For example, we tested the first line signalling less, equal, and more lines than in actuality. To test this function out, we also wrote a seq.code file that contained the instructions for printing out the first 10 Fibonacci numbers using PUSH, ADD, and SEND instructions.

4. Conclusion

In this lab, we were able to understand a prewritten sequencer module and pick up some of the best practices in terms of Verilog styling, formatting, and structures. This was especially put to the test when we were tasked with making several modifications to the modules and testbench to improve the functionality of the sequencer. These implementation tasks proved quite challenging, as they involved understanding how various modules interact with each other in order to achieve the desired result. One task that stood out as especially difficult was implementing the register number in the UART display, which required an understanding of the internal state machine inside 'uart_top.v' as well as how it interacts with the top module. After struggling with this task and receiving help from the TAs during office hours, we were finally able to complete it.

Ultimately, the process of figuring out how the sequencer module worked in order to create our implementations was a highly effective learning experience. We are now much more familiar with how modules should interact with each other in Verilog, and also what types of formatting practices are preferred. We also gained a great deal of experience in debugging, which will hopefully smooth out the process for future labs.

Workshop Answers

Clock Dividers

1. The period of clk_en is 1,310,720 nanoseconds, as depicted in the simulation waveform below:

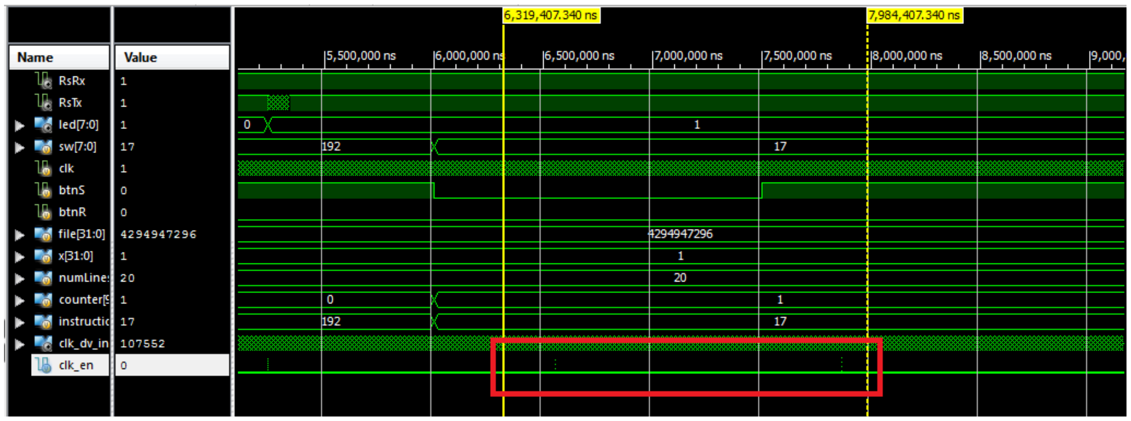


Figure 8: A simulation waveform showing two occurrences of clk_en.

2. The clk_en signal is active for 10 nanoseconds, so the duty cycle can be calculated as follows:

$$D = \frac{T}{P} = \frac{10}{1{,}310{,}720} * 100\% = 0.000762939\%$$

   The duty cycle of clk_en is 0.000762939%.

3. The clk_dv signal is 0 when clk_en is high.

4. The following figure is the schematic translation for the Verilog code involving the clk_dv, clk_en, and clk_en_d signals:

```
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
```

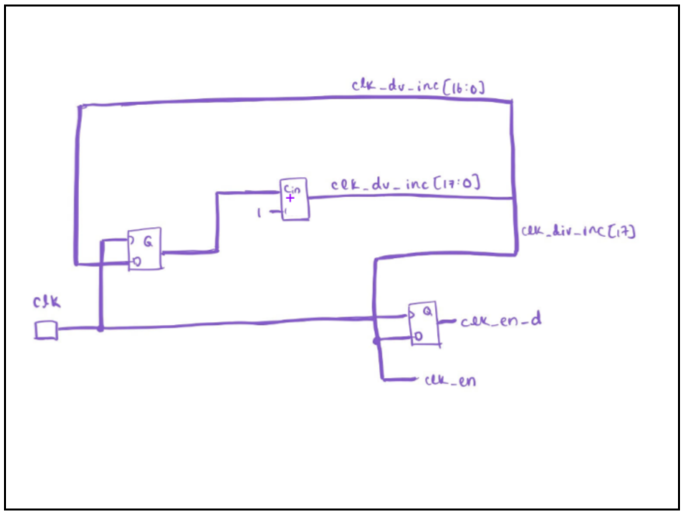


Figure 9: Clock Divider Schematic Translation. clk_dv_inc acts as a 17 bit counter, which triggers clk_en once each cycle. clk_en_d buffers this by 1 additional cycle.

Debouncing

1. If we used clk_en instead of clk_en_d in the statement at line 102 of ‘nexys3.v’, we would effectively remove the debouncing element of our code. This could result in multiple button presses being incorrectly registered, which could cause unwanted instructions to be executed.

2. We can use the line ‘clk_en <= clk_dv[16]’ to effectively create a 50% duty cycle clk_en. This is still allowed, since the period remains the same and we are only looking for the rising edge of the clk_en signal to invoke an instruction.

3. The following waveform capture shows the timing relationship between clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld:

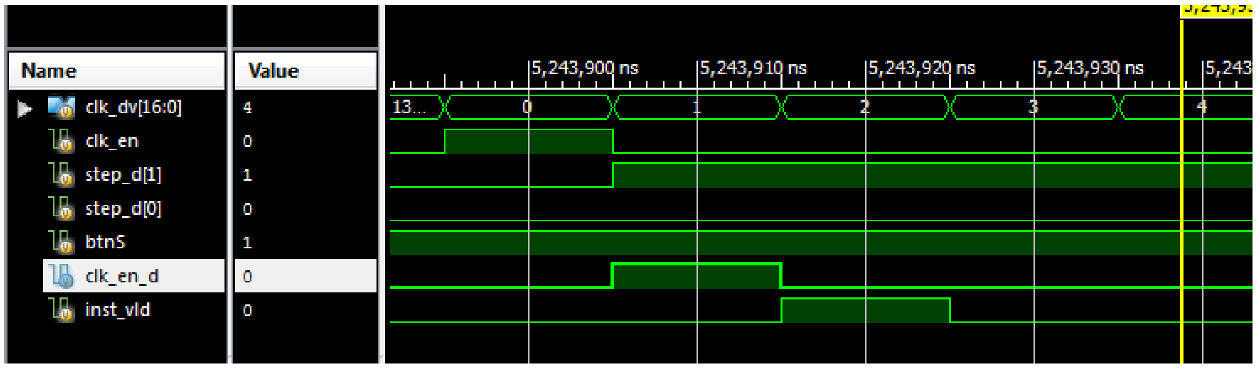


Figure 10: Clock Timing Simulation Waveforms.

4. A schematic showing the interaction of the aforementioned signals can be found below:

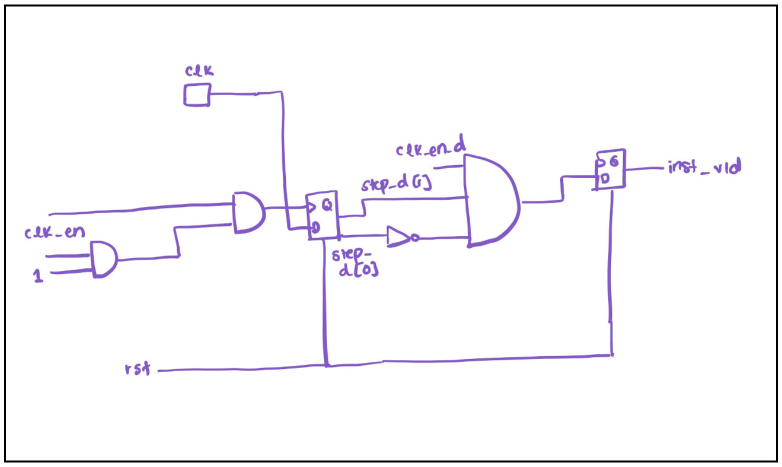


Figure 11: Clock Timings Schematic Translation. If clk_en is high and a positive edge is seen on the state of the button, the inst_vld bit is set to 1, allowing an instruction to be executed.

Register File

1. The line where a register is written a non-zero value is ‘rf[i_wsel] <= i_wdata;’. The ‘i_wdata’ input is passed in from the results of the ALU, which in this case is an example of sequential logic since it is inputted on the positive edge of the clock signal.

2. The register values are read out from the register file in the lines of code ‘assign o_data_a = rf[i_sel_a];’ and ‘assign o_data_b = rf[i_sel_b];’. This is an example of combinatorial logic, since there is no delay or waiting for a clock signal. If I were to manually implement the readout logic, I would use multiplexers with ‘i_sel_a’ and ‘i_sel_b’ as select bits to pick the contents of two of the four registers.

3. The following figure shows the first instance of register 0 being written with a non-zero value:

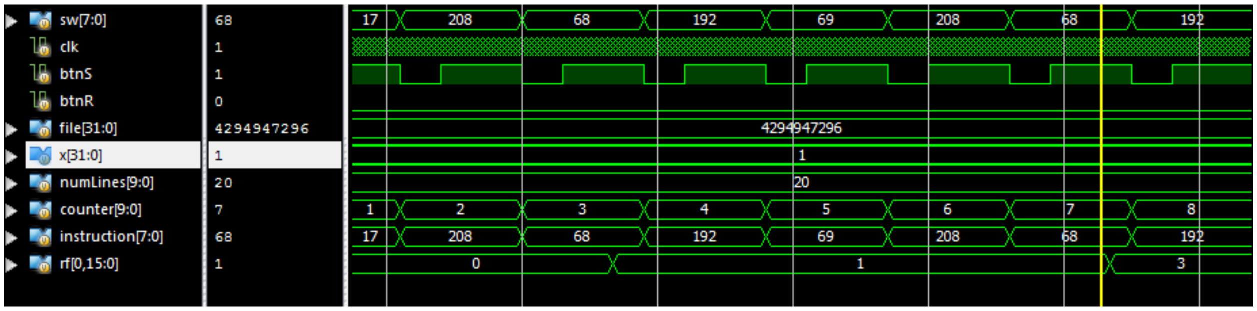


Figure 12: Register 0 Simulation Waveform. Register 0 (shown in the bottom row) is written with the value 1 following an ADD R0 R1 R0 instruction.

4. The following figure is a circuit diagram translation of the register file block:

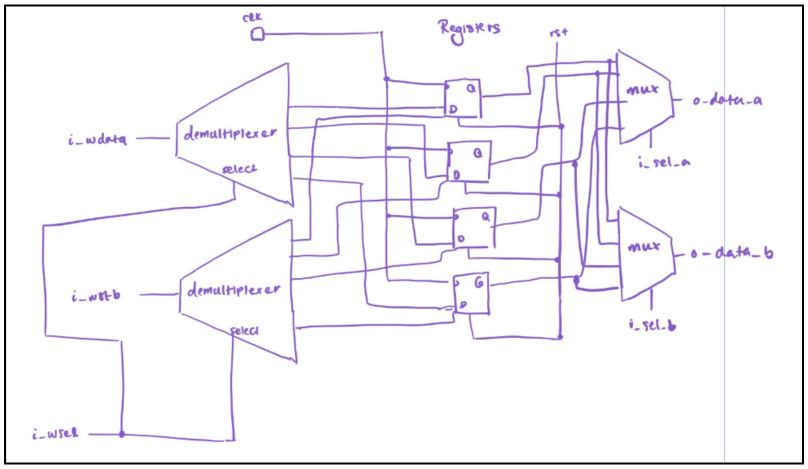


Figure 13: Register File Schematic Translation. 2 bits each of ‘i_sel_a’ and ‘i_sel_b’ are used as multiplexer selects to read the contents of two registers to ‘o_data_a’ and ‘o_data_b’. If the ‘i_wstb’ write enable bit is on, we can also write ‘i_wdata’ to a register selected by ‘i_wsel’ using demultiplexers.