Shikha Mody (604922996), Brendon Ng (304925492), Armaan Singh (404932331)

COM SCI M152A Lab 5

TA: Logan Kuo

**Lab 2 Report**

**Sequencer**

**Introduction**

In this lab, we were tasked with implementing an adder/multiplier sequencer as an FPGA project. The sequencer is designed to use four 16-bit registers, and perform addition and multiplication operations between them. The contents of the registers should also be able to be modified and read at any time. To operate the sequencer, the user inputs an instruction using an array of 8 switches on the board. Then, the user presses a push-button to execute the instruction, this increments the instruction counter as well as updates the register values according to the instruction. This lab is to be done both as a simulation, as well as implemented on the FPGA board. Since the majority of the code was given to us, the main purpose of this lab was to understand how verilog code is implemented on the board, how to modify that code, and how the board can be interacted with when running a program. The overview of the design requirements as provided in the Lab 2 specifications is shown below:

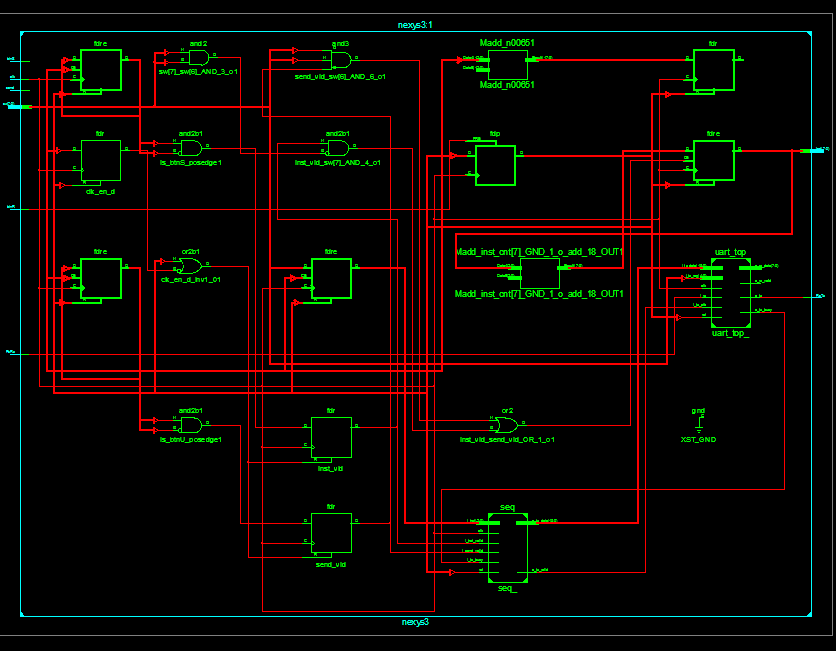
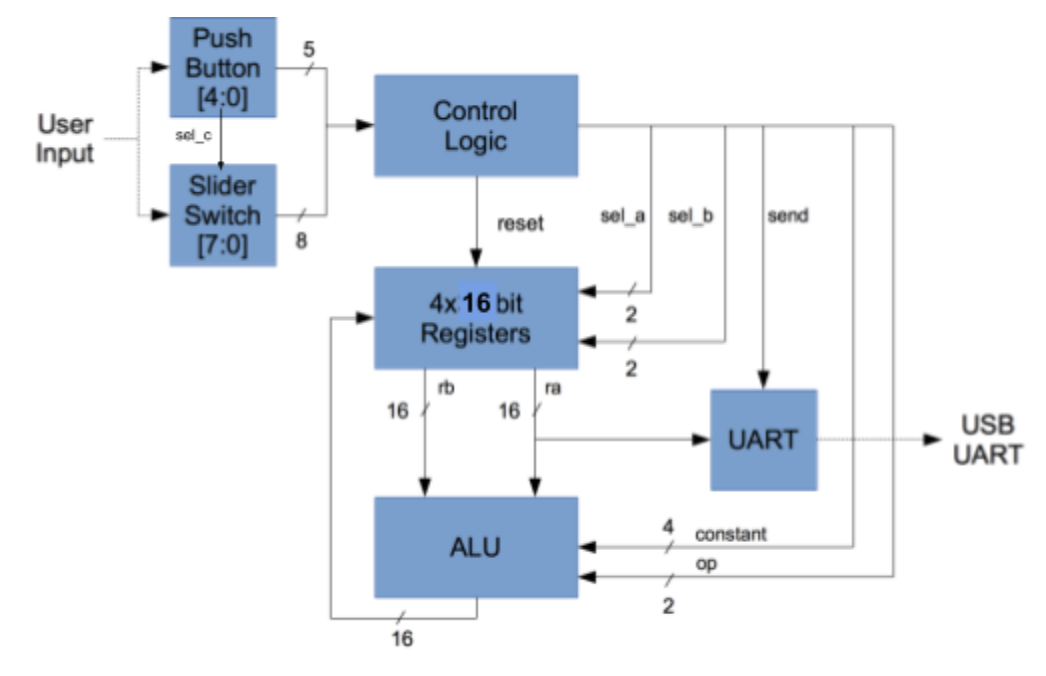
* One 8-bit input: sw[7:0]
* Three debounced push-buttons
  + BTNR - Used to reset the register values to zero
  + SEND - Used to push a register value over UART
  + BTNS - Used to execute a sequencer instruction
* Output SEND instructions over UART

Essentially, the 8-bit input from the switches is decoded and the board executes the instruction, accessing and modifying the registers accordingly. In the lab, we were required to implement the multiplication instruction and the SEND button’s functionality. In addition, we modified the UART output to improve readability by printing the register name before it’s value. In the testbench, we also added the ability to load instruction from a text file when simulating the project, and we created a sample script to send the first 10 fibonacci numbers to UART.

*This background information was adapted from the Lab 2 specification.*

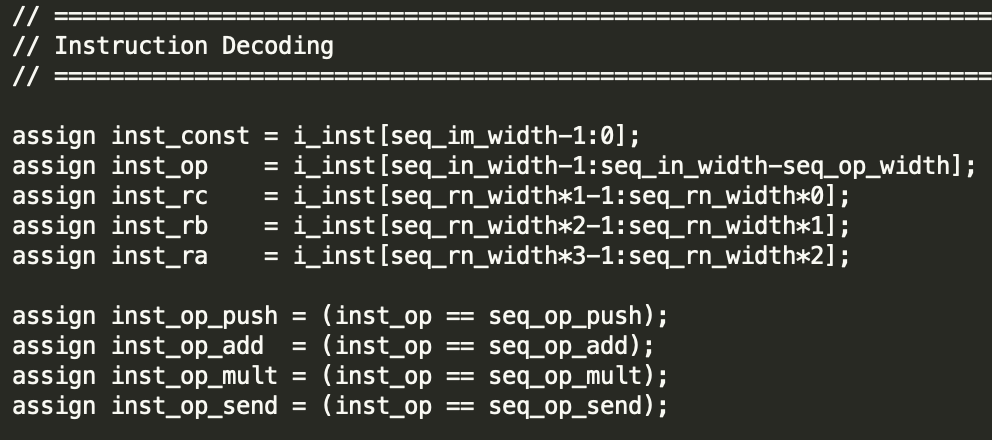
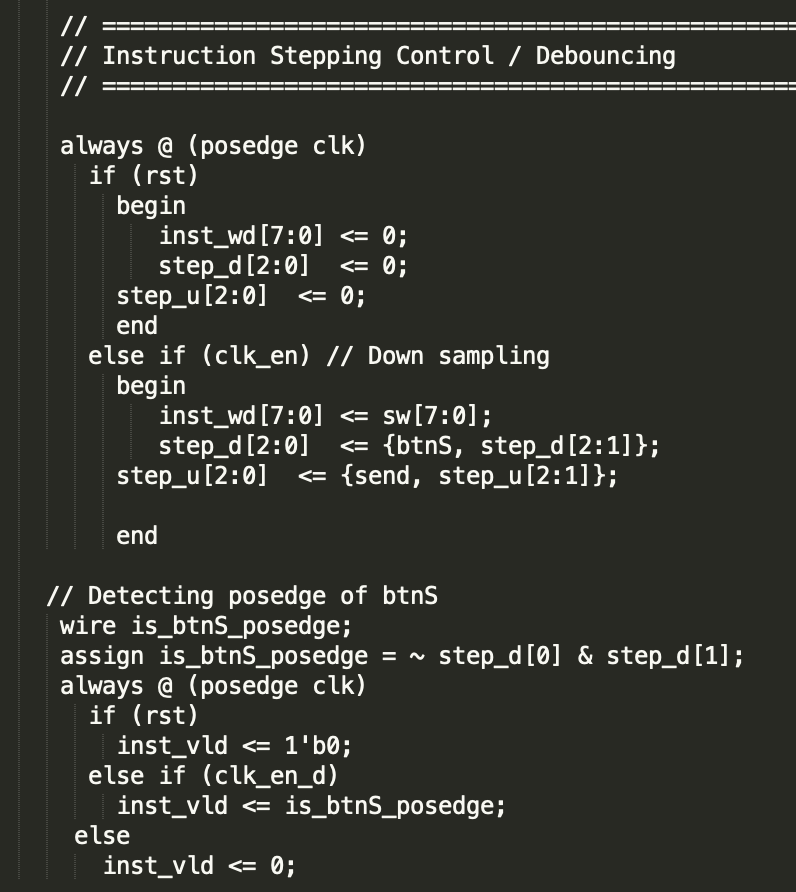
**Design Description**

The sequencer is implemented with a modular design, taking user input from the switches and buttons in nexys3.v, sending the instruction code to our sequencer where the signal is decoded and the correct instruction is executed, and the result is stored in the register file. Upon a send instruction, the contents of the specified register are sent to the UART and outputted to the Putty console.



*Diagram of Sequencer architecture from specification (left), and module equivalent (right)*

In nexys.v, signals from the 8 switches are inputted as well as the button inputs for execute, send, and reset. Here, we use debouncing to ensure proper receiving of button inputs. We modelled it after the debouncing that was implemented for the reset button. Essentially, debouncing allows us to “desensitize” the button so that the button stays at a consistent value when it changes state instead of changing very rapidly at every clock cycle (sampling). This added stability to each button press. When an execution button press is detected, we set the variable `inst\_vld` to 1. When a send button press is detected, `send\_vld` is set to 1. Both of these valid booleans are passed into the sequencer to indicate to it whether to execute an instruction or whether to execute a send. We check to see if switches sw[7] and sw[6] are flipped to 1 when a send button is pressed and we check to make sure they are not both 1's when an instruction is executed before we increment our instruction count. The LED outputs are assigned to the 8-bit binary value of our instruction count and outputted.



*Code Snippets: Button signal debouncing (left), instruction decoding (right)*

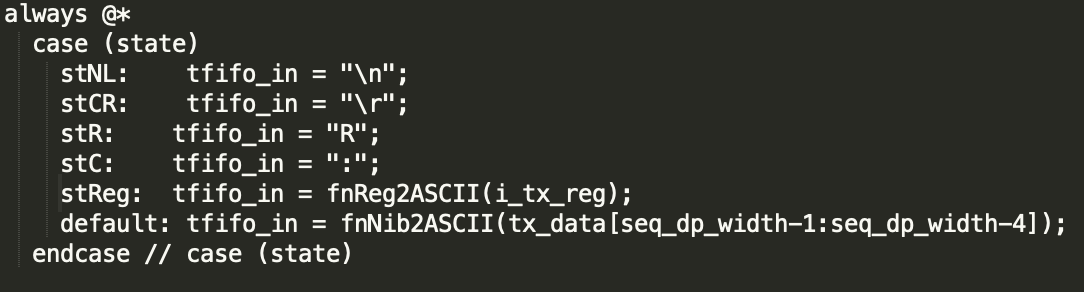
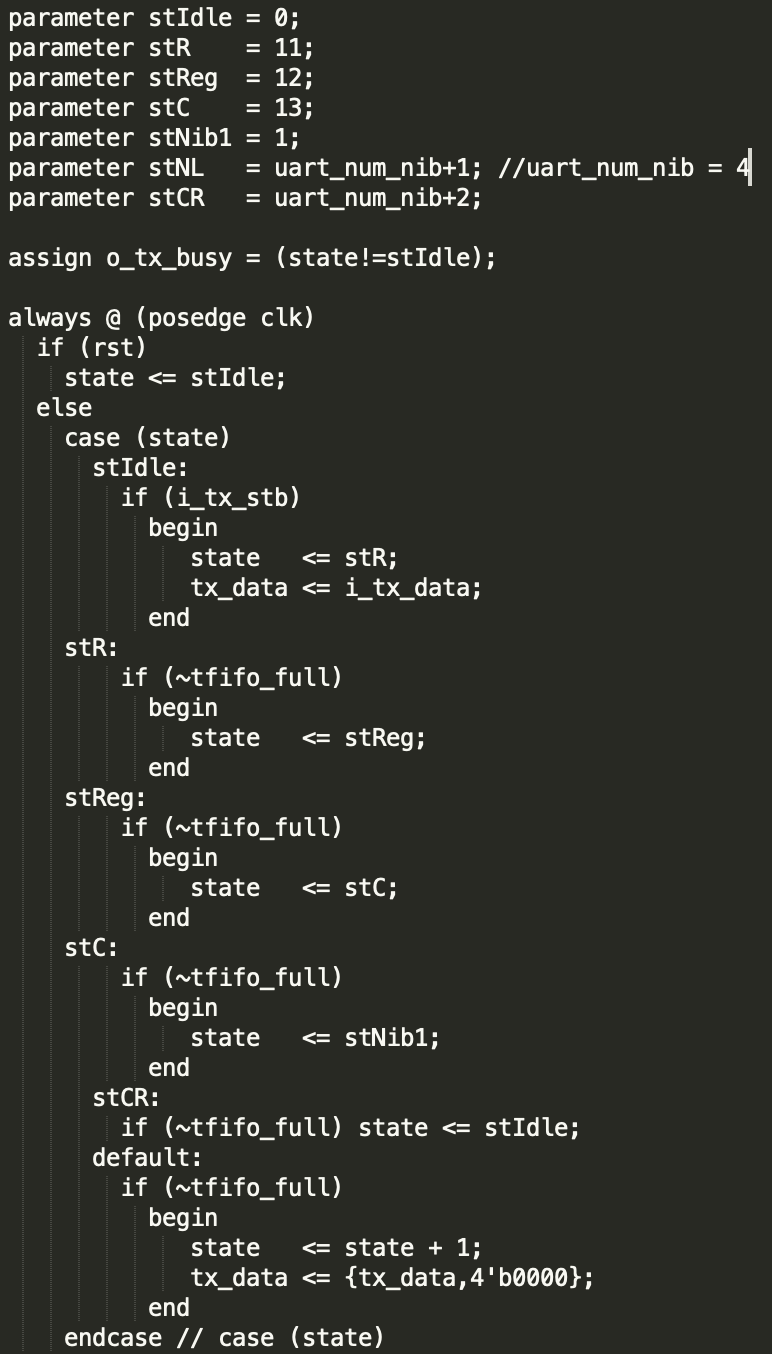
The nexys.v module calls the seq.v sequencer module which takes inputs of the instruction code and whether an instruction or send has been called. The seq module takes the instruction code and decodes it by slicing it into its respective parts. The two most significant bits are assigned to the op code, the next six bits are assigned to the three different register specifier variables, the four least significant bits are also assigned to constant field and is used only if we are performing a load. Only one of the variables `inst\_op\_push`, `inst\_op\_add`, `inst\_op\_mult`, or `inst\_op\_send` is set to one depending on the op code and whether instr\_valid or send\_valid is one, indicating which operation is to be performed. The registers specified, the instruction code, and the constant data are passed into the ALU module for operation. The ALU module sets its output data to the result of the operation and that data is inputted into the register file module where it is stored depending on which registers were selected.

The ALU module takes in as input operation data and the op code. The ALU calls both add and multiplication modules and retrieves the output of both in two different regs. Depending on the op code, it sets the module output to be stored into the register file to be one of the results from the add, multiply, or inputted constant data (for push function).  


*Code Snippets: In seq\_alu.v, both add and mult modules are called and output is set*

*according to op code.*

The UART module and output is essentially a state machine. It is called by nexys.v and is given input of the data being sent and the values of switches sw[5] and sw[4] indicating which register is being sent. The uart\_top.v module sends one ASCII character at a time to the Putty output console. The module will output a different character depending on which state it is in. If the state is stR it will output "R", if it is in state stReg it will output an ASCII number 0-2 depending on the values of switches sw[5] and sw[4], if it is in state stC it outputs a colon, and the other 4 states correspond to the different bytes of the hexadecimal value to be outputted. The state machine starts in an idle state and when something is to be sent, it transitions into the stR state and "R" is printed, then transitions to the stReg to output the register number, then transitions to stC to output the colon. From stC, it goes into state stNib1 which has the state number 1. From here the state number corresponds to which digit of the output data we are sending. The digit is translated into ASCII, outputted, and the state number increases by 1. When the state number reaches 5, it is now in stNL where a new line is outputted. The state number then goes to 6 which is stCR where a carriage return is outputted to successfully complete the single output and move to a new line, returning to stIdle to wait for the next send instruction.



*Code Snippets: (Left) Verilog implementation of state machine*

*(Right) ASCII output depending on state*

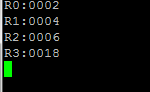
*State Diagram: Representation of the UART module state machine*

Other aspects of the design of this lab included how we read a file and stored its contents. We used the $readmemb function to read a file, and stored the contents in a two dimensional array according to the dimensions of the specification (i.e. 1024 lines). This allowed simplified instruction loading for simulation. To execute the instructions, we looped through the memory array and executed each instruction with a counter variable, similar in functionality to a for loop. Since the first line of the file represented how many instructions were contained within it, this was our stopping condition.

**Simulation Documentation**

Because this lab was broken into relatively small parts, we either simulated our implementation and used the produced waveforms, or implemented it onto our FPGA board in order to interactively test it. Simulation and other testing documentation will be explained. This lab was broken into many different parts that built on top of each other, so by testing later parts we were also simultaneously testing earlier parts and ensuring that they were correct. This way, we could complete all parts of the lab and demonstrate them accordingly.

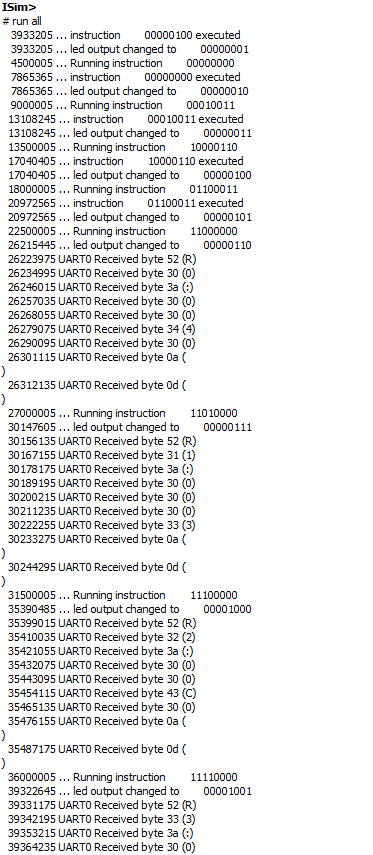
The first part of the lab allowed us to get familiar with the structure of the modules and code that was provided, so we started with the first steps of the lab to ensure that the existing functionality worked properly: PUSH, ADD, and SEND (as seen in the “Warm Up Task” section of the lab specifications). Once we implemented the multiply operation, we tested all of these functionalities. The Putty output is shown below:



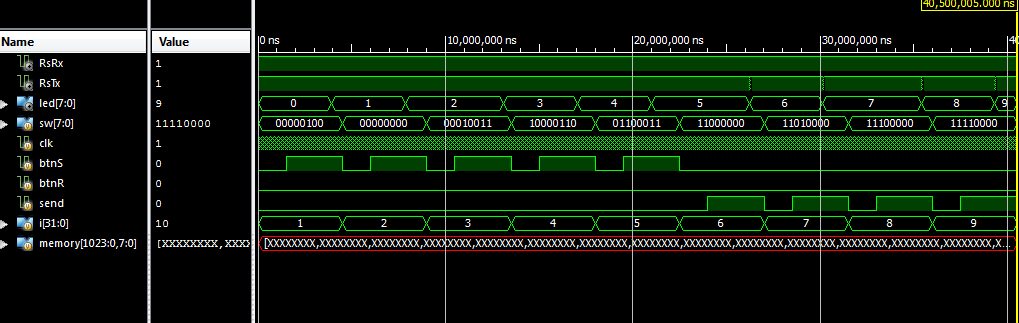
The image above represents the following instructions:

1. Store 2 into R0 ⇒ PUSH R0 0x2
2. R1 = R0 + R0 ⇒ ADD R0 R0 R1
3. R2 = R0 + R1 ⇒ ADD R0 R1 R2
4. R3 = R1 \* R2 ⇒ MULT R1 R2 R3
   1. The result of this is 0018 (hexadecimal). In decimal, it is 24 (4\*6=24).

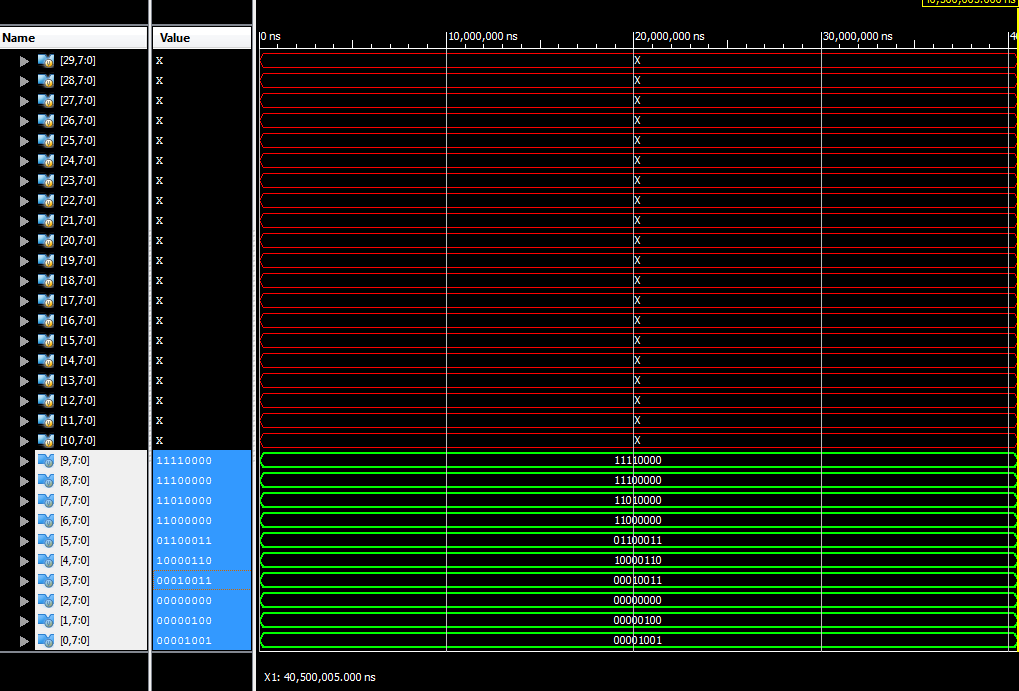
After each of these steps, we “sent” the output to the Putty console, which is shown above. Based on the values we stored into each register (R0-R3), we could see that our multiplication was functioning properly. Also, this test used a separate SEND button to send values to the Putty console, another requirement of the lab. The register labels also functioned correctly. Because many of the output modifications built upon one another’s functionality, we could see if many parts were working with just one test. This particular image shows the operation (add, multiply), send button, and “Nicer UART” output functionalities.

The main parts where utilized the waveform simulation was for the last two portions of the lab: loading a file of instructions and demonstrating Fibonacci sequence output. We first tested the $readmemb and execution functionalities with example instructions given in the Lab 2 Specification. This file is included as “sampleSeq.code.” Hence, we could verify that the correct steps were being executed in the correct order. See the figure on the right for a snippet of console output of the sample instructions:

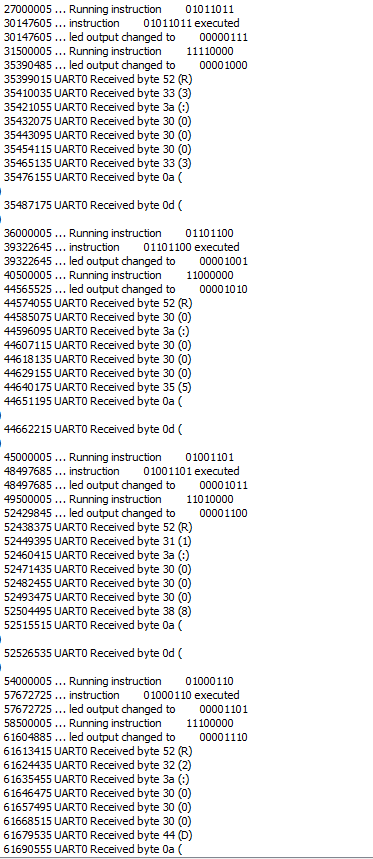
The console output helped us trace the execution of instructions. It corresponded to the sequence’s meaning (i.e. doing the correct operation and affecting the correct registers), which we determined by tracing through each sample instruction. In addition, we examined the waveforms for this sample sequence:



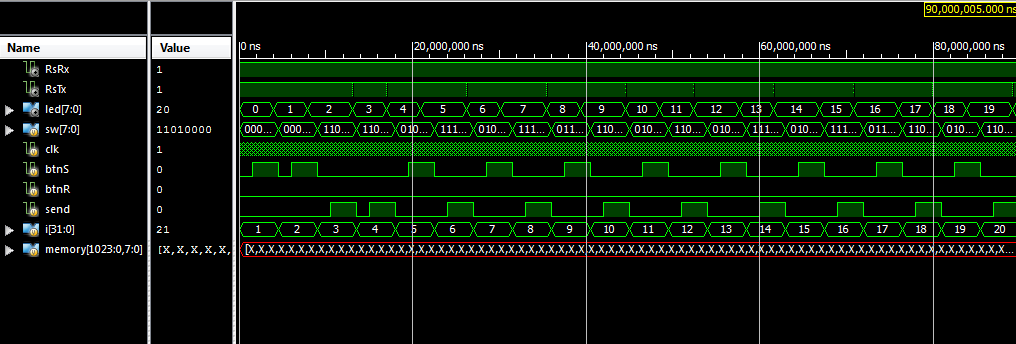
This shows the instructions as sw[7:0], instruction count in led[7:0], and memory content of the stored lines of instructions from the file. For clarity, we changed the radix representation to binary for the switches (instructions). The reason why memory (array that stores contents of file) has many X’s in it is because we allocated the “memory” array to have 1024 rows, corresponding to the 1024 possible lines of instructions as the lab specification stated. See below for a magnified view of the rows in the memory register, with particular focus on the instructions that were indeed present in the file (9 total):



Because the rest of the file was empty, no other rows in the memory array were used, corresponding to ‘X.’ (We did not initialize it in the test bench–if we had, all other values would be 0, or whatever value they were initialized to).

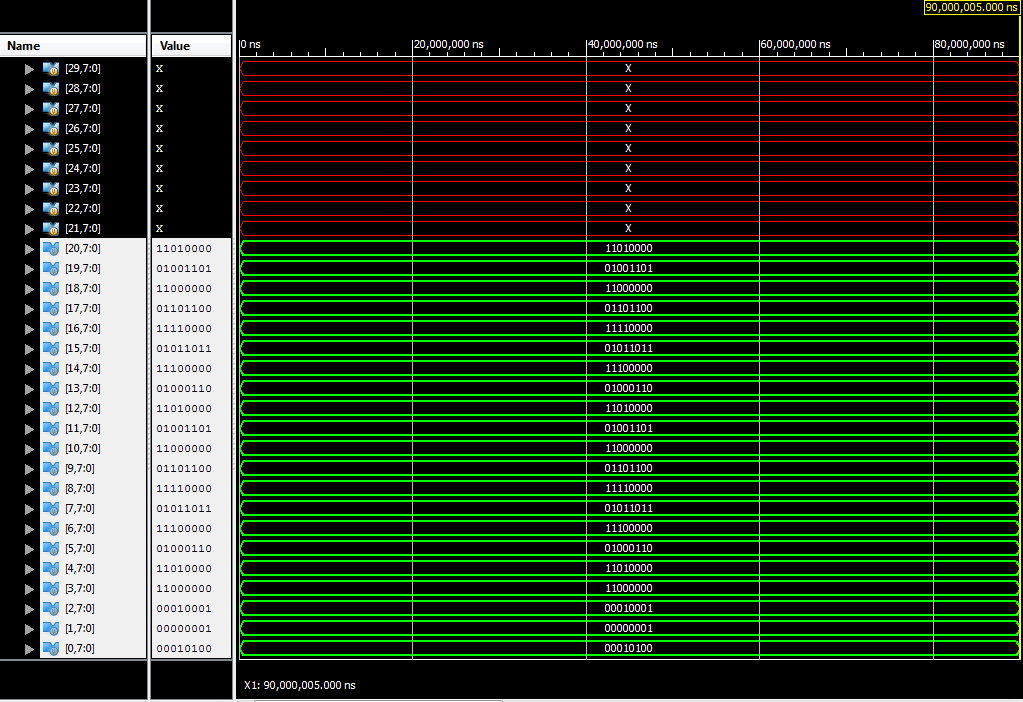
Then, we used this new functionality to output a series of the first 10 Fibonacci sequence numbers. We translated these instructions to the proper format (with ADD, SEND, and PUSH) so that we could put them into a file (“seq.code”) which would then be read by our program. There were 20 total instructions, corresponding to 21 lines in the seq.code file (20 instructions, first line had the number 20 in binary for instruction count). Below is a snippet of console output from running these instructions:

The first 10 Fibonacci sequence numbers (in decimal format) are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. These correspond to: 1, 1, 2, 3, 5, 8, D, 15, 22, and 37 in hexadecimal, which is what we demonstrated. Lastly, see below for a waveform diagram of the Fibonacci sequence output. Once again, the memory array shows ‘X,’ but the magnified version shows the correct number of lines and instructions:

**

*Above: Waveform diagram of Fibonacci sequence with all instructions*

*Below: Magnified version of memory array*



As a result, by both simulating and implementing each section of the lab, we were able to isolate bugs and get the correct results. For the specific instructions of our Fibonacci sequence, see the seq.code file.

**Conclusion**

In summary, since the modules for this lab were for the most part complete, our design mainly centered around implementing key features within these larger projects to allow for more intuitive and easier use of the sequencer. When simulating, we also followed this modular approach while completing each of the tasks within the lab. While designing any of the modifications, it became important to simulate that module and analyze how the waveform would respond to certain instructions being run. We mainly ran into difficulties when implementing the UART improvements where our program would output only the least significant 4 bits of the register. By using the simulation to analyze our state machine, we were able to better understand the problem and eventually fix it. Generally, this lab could be improved by having the modifications to the test bench done earlier in the lab. Being able to edit and load a text file with sample code into the simulation would have allowed for more rapid and streamlined testing throughout the project. Regardless, the lab was helpful in showing us how a larger Verilog project with multiple interconnected modules is implemented.