Lab 2: Sequencer

CS M152A Lab 3
TA: Gu, Hongxiang
May 6, 2019

Nathan Marsh, [illegible]
Denise Wang, [illegible]

Introduction

The goal of this lab is to understand and modify a small scale FPGA project. It contains seven total tasks with some being implementation portions and others simulation portions.

The tasks that make up the implementation portion are:

- Warm Up Task: Translate a “program” into binary instructions.
- Missing Multiply Operation: Create the Multiply operation using the coded Add operation.
- A Separate SEND Button: Change the initial design to use a separate send button that is dedicated to the send functionality.
- Nicer UART Output: Implement a more intuitive way to print the content of a particular register.

The tasks that make up the simulation portion are:

- An Easier Way to Load Sequencer Program: Change the static set of instructions. Instead, load the instructions from a text file.
- Fibonacci Numbers: Design a sequence of instructions such that the first 10 numbers of the Fibonacci series are printed from the UART.

These tasks allow for familiarization of style, techniques, formats, and structures of code, etc. that would be useful for future design projects.

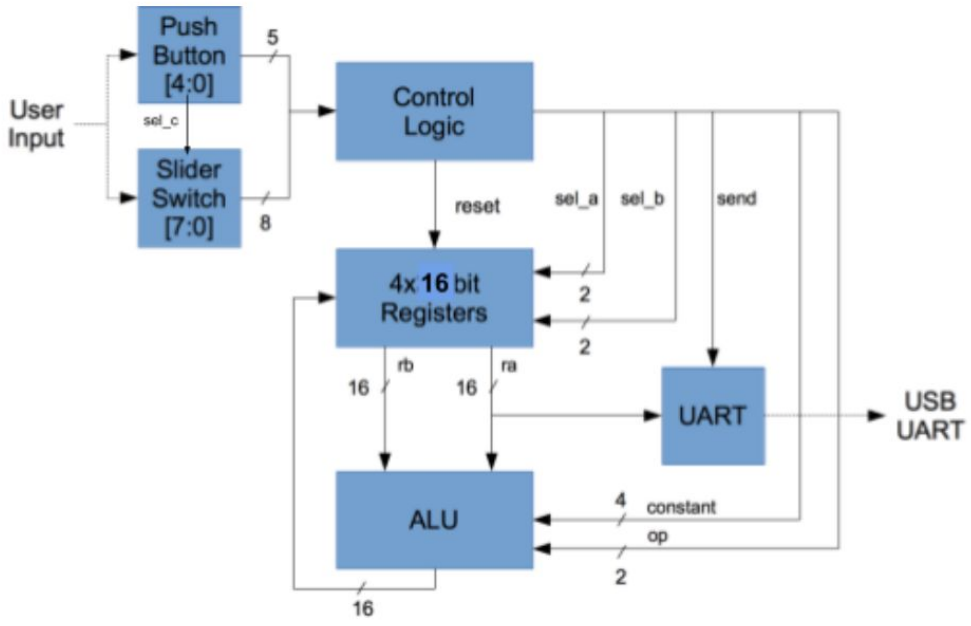


Figure 2.1- Project Diagram. The adder/multiplier sequencer has four 16-bit general purpose registers and can perform add or multiply instructions using registers as operands. The results are stored into register files.

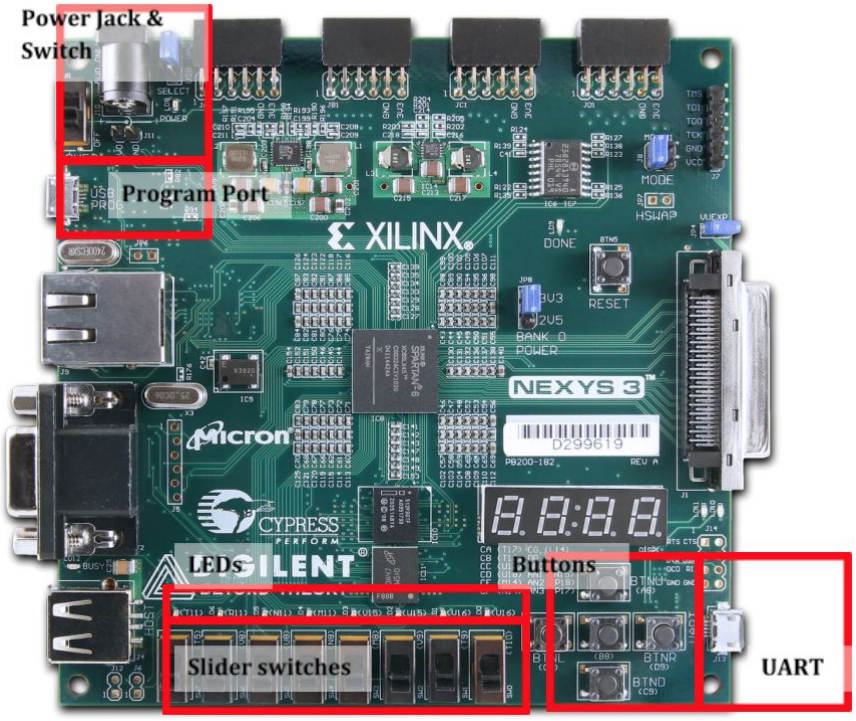


Figure 2.2- Nexys 3 Board. The sequencer is operated using the switches and push buttons. The eight slider switches represent a single 8-bit instruction (up: 1, down: 0), the center button enters and executes the instruction, and the right center button resets values of all registers to 0.

Warm Up Task

Using the Nexys 3 Board, the first task is to translate the following “program” into binary instructions and output the correct result on the UART console. The components used throughout this task includes the switches and center and right button. Each of the instructions are inputted using the switches, executed using the center button, and reset using the right button.

Push (00)- Left shifts target register by four bits and “pushes” new constants in.

7	6	5	4	3	2	1	0
Opcode- 00 Push		Register RA		4 Bit Constant			

Add (01)- Adds (unsigned integer arithmetic) values of registers RA and RB and stores the result to register RC.

7	6	5	4	3	2	1	0
Opcode- 01 Add		Register RA		Register RB		Register RC	

Mult (10)- Multiplies values of registers RA and RB and stores the result to register RC. The inputs are assumed to be unsigned integers and only the lower 16-bit results are retained.

7	6	5	4	3	2	1	0
Opcode- 10 Mult		Register RA		Register RB		Register RC	

Send (11)- Sends the content of register RA to the UART for display. The 16-bit value is converted to ASCII-HEX and appended with a newline character.

7	6	5	4	3	2	1	0
Opcode- 11 Send		Register RA					

Program:

PUSH R0 0x4: Left shifts register R0 by four bits and “pushes” 4 in it.

0	0	0	0	0	1	0	0

PUSH R0 0x0: Left shifts register Ro by four bits and “pushes” 0 in it.

0	0	0	0	0	0	0	0

PUSH R1 0x3: Left shifts register R1 by four bits and “pushes” 3 in it.

0	0	0	1	0	0	1	1

ADD R0 R2 R2: Adds register R0 and R2 and stores the resulting value in register R2.

0	1	0	0	1	0	1	0

ADD R0 R2 R2: Adds register R0 and R2 and stores the resulting value in register R2.

0	1	0	0	1	0	1	0

ADD R0 R2 R2: Adds register R0 and R2 and stores the resulting value in register R2.

0	1	0	0	1	0	1	0

ADD R2 R0 R3: Adds register R2 and R0 and stores the resulting value in register R3.

0	1	1	0	0	0	1	1

SEND R0

0	0	0	0	0	0	0	0

SEND R1

0	1	0	0	0	0	0	0

SEND R2

1	0	0	0	0	0	0	0

SEND R3

1	1	0	0	0	0	0	0

Figure 2.3- UART Console Output. The resulting outputs of the Warm Up Task.

Missing Multiply Operation

The next task is to implement the multiply operation missing from the code. The multiply operation executes unsigned integer multiplication of the register values in each of the instructions. From the add operation in the seq_add.v file, the multiplication operation can be implemented by modifying the instances of addition to multiplication. Specifically, instead of adding inputs i_data_a and i_data_b, multiply the two and assign the result into o_data. We needed to make a seq_mult.v file that was exactly the same as the add file but had multiplication instead of addition.

Test Case	Result	Pass/No Pass
0003*0001	Multiplication of two hex values	Pass
0003*0000	Multiplication of a hex value and zero	Pass
0003*0001	Multiplication of a hex value and one	Pass
0003*FFFF	Overflow, sixteen least significant bits	Pass

Separate SEND Button

To make a separate send button work, we modified nexys3.ucf, nexys3,v and seq.v files. In the nexys3.ucf, we uncommented on of the buttons in the file to get the physical functionality activated. In nexys3.v, we wrote the code to check if the button was pushed using button debouncing. We implemented our button exactly like the original button by creating separate registers and always @ blocks to mimic the implementation given. We also modified the incrementor on instruction count to

increment if it if either button was pressed. In seq.v, we modified the o_tx_valid signal to be asserted if the send signal was asserted. We implemented the task in this way because it seemed like the most logical way to do it without writing a lot of extra or different code. We tested this signal by opening our Putty terminal and seeing if pushing the button gave proper outputs.

Nicer UART Output

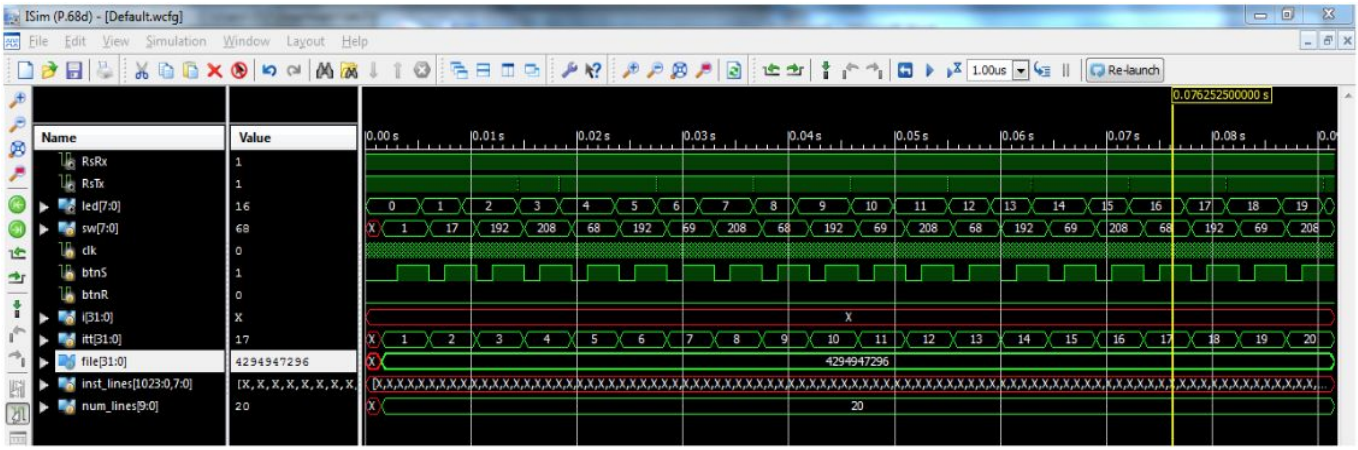
To get a nicer UART output we had to modify uart_top.v and nexys.v files. In uart_top.v, we needed to alter a finite state machine to get a better output format. Originally, the state machine printed the register and then a new line and carriage return (‘\n’ and ‘\r’) after the register was printed. In our modifications we introduced new states that printed R, the register number and colon before outputting the register value and the new line characters. In order to get the register number to be printed we needed to modify nexys3.v. In nexys3.v we got the register value from the sequencer module and fed it into the uart module to solve our problem of not having the register number in uart_top.v. We chose this approach because in class we were told to alter the finite state machine for this part. When testing the nicer uart implementation, we saw on the Putty terminal that we were not getting the new line functionality. We figured out that the state variable was overflowing because it had fewer bits for the old finite state machine. Once we changed our state incrementer to have five bits this part worked perfectly.

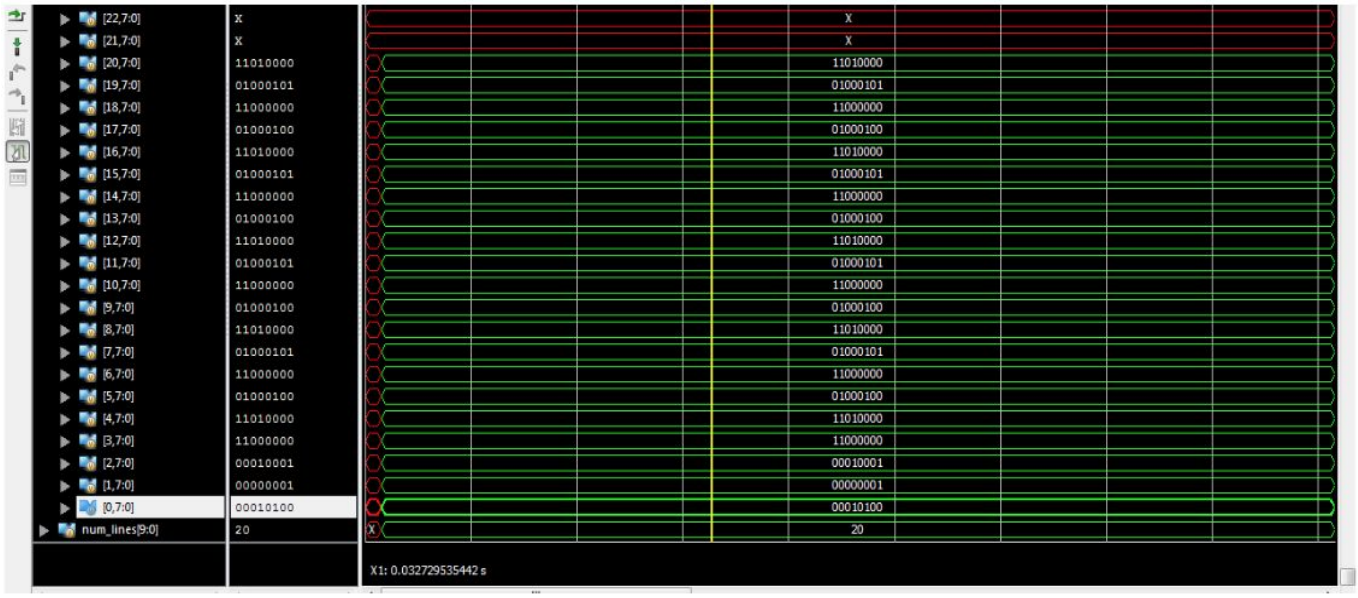
An Easier Way to Load Sequencer Programs

To complete this task, we were required write a test bench that loads a file called seq.code and executes the included binary instructions. To implement this task, we looked at the functions already specified in tb.v. Included were functions that sent push, multiply, add, and send instructions. We commented out the calls to these functions because we were leading from the file seq.code not sending the instructions through the test bench. We noticed there was a tskRunInst that ran instructions when given binary input. This function worked perfectly because our seq.code file is supposed to have binary instructions to execute on each line following the first instruction count line. To extract the lines from the file, we used the functions fopen to open the file and readmemb to load the load the file into a 1024 long array of 8-bit numbers. Then, we extracted the first line to get the number of instructions. Finally, we looped from 1 to the number of instructions, and in each loop we ran tskRunInst on each part of the 1024 length array which is equivalent to running through all of the instructions. We chose this approach because this method was somewhat hinted at in the specification. We tested our modifications after completing part 7 because we needed to write a seq.code file to test if we could read from seq.code.

Fibonacci Numbers

In this task we had to write instructions in the seq.code file in order for the UART to print the first ten Fibonacci numbers. In order to do this task, we used two registers R0 and R1 and started by loading the registers with the value one. Then, we printed those two values since ones are the first two values of the Fibonacci numbers. Next, we added R0 to R1 and stored it in R0, then printed R0. Since the next Fibonacci number require the sum of the previous registers, we just added the contents of R0 and R1 at each stage and alternated the destination register, then printed out the value of the destination register. We chose this approach because it seemed like it used the least number of registers and completed in the least number of instructions. We tested this task by running the test bench and checking the values sent to the UART and converting the numbers to decimal to see if they matched the actual Fibonacci numbers.





14427495 UART0 Received byte 52 (R)
14438515 UART0 Received byte 30 (0)
14449535 UART0 Received byte 3a (:)
14460555 UART0 Received byte 30 (0)
14471575 UART0 Received byte 30 (0)
14482595 UART0 Received byte 30 (0)
14493615 UART0 Received byte 31 (1)

18359655 UART0 Received byte 52 (R)
18370675 UART0 Received byte 31 (1)
18381695 UART0 Received byte 3a (:)
18392715 UART0 Received byte 30 (0)
18403735 UART0 Received byte 30 (0)
18414755 UART0 Received byte 30 (0)
18425775 UART0 Received byte 31 (1)

27534695 UART0 Received byte 52 (R)
27545715 UART0 Received byte 30 (0)
27556735 UART0 Received byte 3a (:)
27567755 UART0 Received byte 30 (0)
27578775 UART0 Received byte 30 (0)
27589795 UART0 Received byte 30 (0)
27600815 UART0 Received byte 32 (2)

36709735 UART0 Received byte 52 (R)
36720755 UART0 Received byte 31 (1)
36731775 UART0 Received byte 3a (:)
36742795 UART0 Received byte 30 (0)
36753815 UART0 Received byte 30 (0)
36764835 UART0 Received byte 30 (0)
36775855 UART0 Received byte 33 (3)

45884775 UART0 Received byte 52 (R)
45895795 UART0 Received byte 30 (0)
45906815 UART0 Received byte 3a (:)
45917835 UART0 Received byte 30 (0)
45928855 UART0 Received byte 30 (0)
45939875 UART0 Received byte 30 (0)
45950895 UART0 Received byte 35 (5)

55059815 UART0 Received byte 52 (R)
55070835 UART0 Received byte 31 (1)
55081855 UART0 Received byte 3a (:)
55092875 UART0 Received byte 30 (0)
55103895 UART0 Received byte 30 (0)
55114915 UART0 Received byte 30 (0)
55125935 UART0 Received byte 38 (8)

62924135 UART0 Received byte 52 (R)
62935155 UART0 Received byte 30 (0)
62946175 UART0 Received byte 3a (:)
62957195 UART0 Received byte 30 (0)
62968215 UART0 Received byte 30 (0)
62979235 UART0 Received byte 30 (0)
62990255 UART0 Received byte 44 (D)

72099175 UART0 Received byte 52 (R)
72110195 UART0 Received byte 31 (1)
72121215 UART0 Received byte 3a (:)
72132235 UART0 Received byte 30 (0)
72143255 UART0 Received byte 30 (0)
72154275 UART0 Received byte 31 (1)
72165295 UART0 Received byte 35 (5)

81274215 UART0 Received byte 52 (R)
81285235 UART0 Received byte 30 (0)
81296255 UART0 Received byte 3a (:)
81307275 UART0 Received byte 30 (0)
81318295 UART0 Received byte 30 (0)
81329315 UART0 Received byte 32 (2)
81340335 UART0 Received byte 32 (2)

90449255 UART0 Received byte 52 (R)
90460275 UART0 Received byte 31 (1)
90471295 UART0 Received byte 3a (:)
90482315 UART0 Received byte 30 (0)
90493335 UART0 Received byte 30 (0)
90504355 UART0 Received byte 33 (3)
90515375 UART0 Received byte 37 (7)

Figure 2.3- The first 10 numbers of the Fibonacci series that are printed from the UART.

Conclusion

Each part of this lab contained 7 tasks that had us modify code and add new elements to different source files. For the first task, we had to output the correct result on the UART console given a set of binary instructions using the slider switches and buttons on the Nexys 3 Board. For the second task, we were to implement the multiplication operation properly multiplies stored register values. For the separate send button task, we had to change the design from the initial normal instruction to use a separate send button that is dedicated to the send functionality. The nicer UART output had us implement a more intuitive way to print the content of a particular register, which was done by modifying the uart_top.v and nexys.v files. For the easier way to load sequencer program, we were to change the static set of instructions to loading

instructions from a text file, which was done by writing a test bench that loads the file seq.code and executes the included binary instructions. For the last task, we designed a sequence of instructions so that the first 10 numbers of the Fibonacci series are printed from the UART. The problem that we encountered throughout this lab was not getting the new line functionality on the putty terminal in the nicer UART output task. However, this error was solved by changing the state incrementer to holding five bits, because initially there was an overflow in the state variable since there were fewer bits for the old state machine. Overall, each component of the lab familiarized us with different techniques that would allow for a better understanding in future design projects.