CS M152A

Lab 2 Report

Sequencer Lab

Authors:

Aaron Yoo, 004745379

Chayanis Techalertumpai, 404798097

Christian Rodriguez, 804789345

Date: 11/03/2019

TA: Logan Kuo

Introduction and Requirements

In Lab 2, we were expected to run a small scale FPGA project and make modifications to it. We were responsible for implementing an adder/multiplier sequencer. The sequencer contained four 16-bit purpose registers and can perform addition or multiplication instructions using the registers as operands. We were responsible for completing 6 tasks in total in order to finish the lab.

The 6 tasks we had to implement include a warm up task, implementing a multiply operation, a separate send button, a nicer UART output, an easier way to load the sequencer program, and outputting the first ten numbers of the Fibonacci sequence.

Our sequencer has an ALU unit which made it possible to implement a multiply operation. It was similar to the add operation, with a slight change to the code. The separate send button and nicer UART output is related to helping the programmer with readability. Once these instructions are implemented, the PUTTY output can now tell the programmer which register holds a certain value. The reason for implementing an easier way to load the input to the sequencer program in a faster and more modular way. Successfully implementing this meant that we can write inputs to our sequencer in another file and quickly load it onto our sequencer.

Design Description

At first, we had to learn how to use the FPGA board in order to implement our sequencer. We mainly used the slider switches and the buttons in order to send instructions to our sequencer. There were 8 switch positions that represented a single 8-bit instruction, a center button that executed the instruction and the right button which was used to reset values of all registers to 0. Lastly, there were 8 LED indicators which specifically showed the number of instructions executed since the last reset.

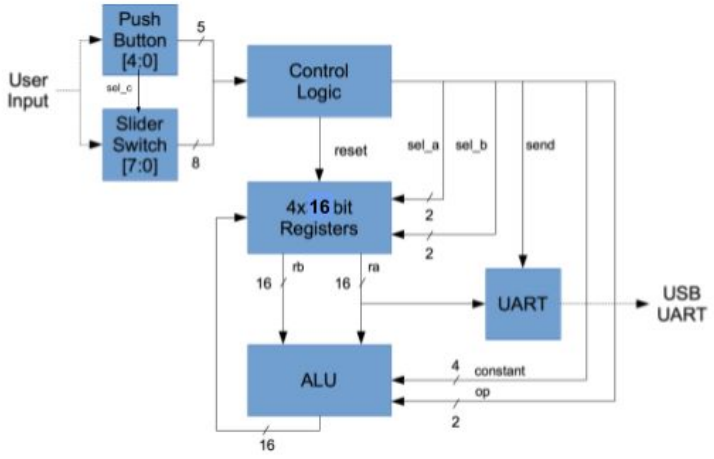


Figure 1: Project Design Diagram

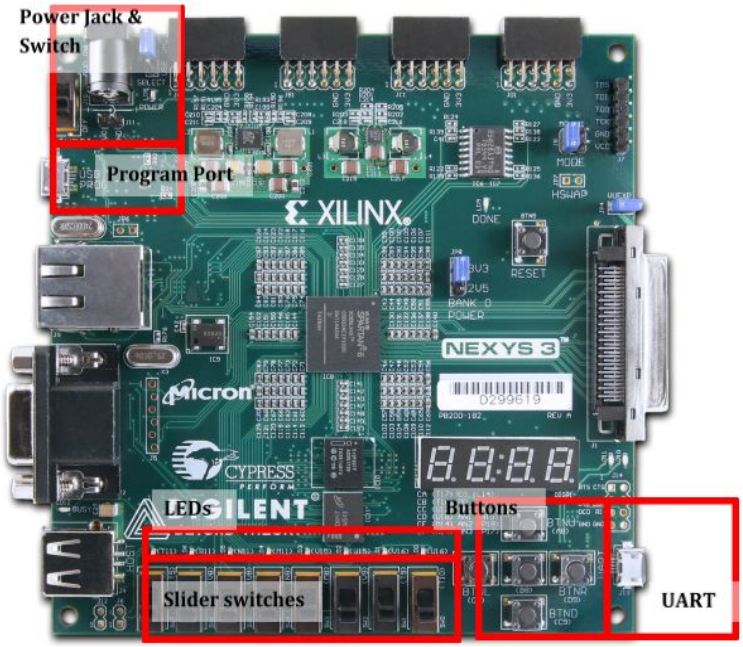


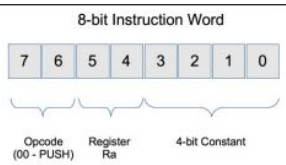
Figure 2: Nexys 3 Board

The sequencer instruction specifications helped us understand how the project worked:

Sequencer Instructions

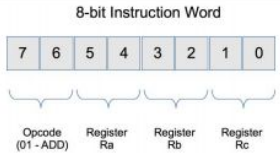
PUSH (00) instruction left-shifts the target register by 4 bits and "pushes" the new constant in. Example:

- R0 starts with value 0x55AA
- PUSH R0 0xB
- Now R0 is 0x5AAB



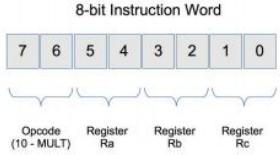
ADD (01) instruction adds Ra and Rb and stores the result to Rc. Addition is performed in unsigned integer arithmetic. Example:

- ADD R0 R1 R3
- R0 + R1 -> R3



MULT (10) instruction multiplies Ra and Rb and stores the result to Rc. Inputs are assumed to be unsigned integers. Only the lower 16-bit results are retained. Example:

- MULT R3 R0 R2
- R3 * R0 -> R2



SEND (11) instruction sends the content of Ra to the UART for display. The 16-bit value is converted to ASCII-HEX and appended with a newline character. Example:

- R1 has a value of 0x1234
- SEND R1 causes the UART to send "1234\n"

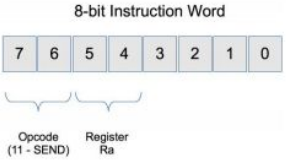


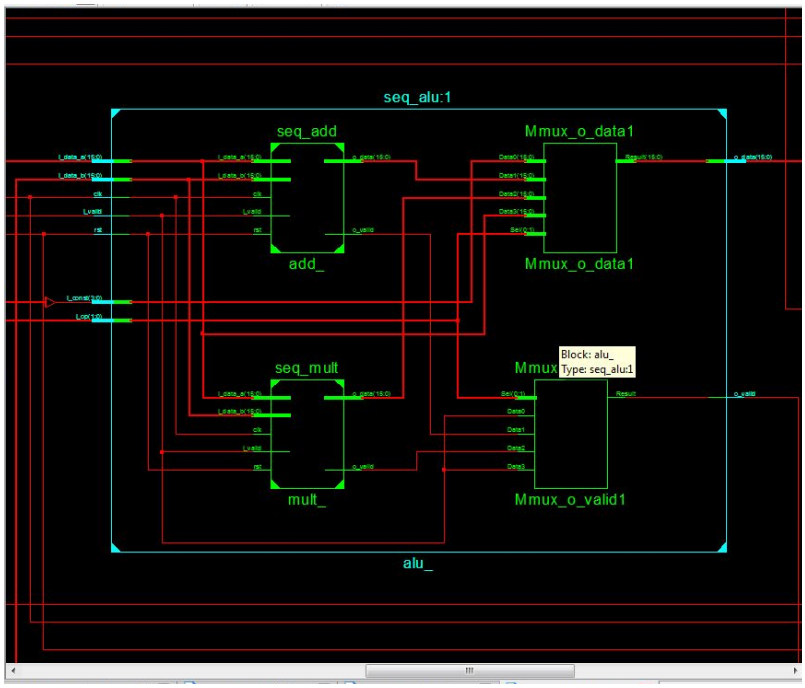
Figure 3: Sequencer Instructions

Task 1: Warm Up Task

Our first task was to build the project with the source files Logan provided for us (src.zip). We had to make sure that the project worked correctly, which was the main purpose for this part of the lab. After successfully building the project, we demoed the correct UART console output to Logan.

Task 2: Missing Multiply Operation

Our second task was to implement a functioning multiply operation. The current version of the code that we had was missing the operation so we have to implement it ourselves. Our implementation made sure that our operation would successfully perform unsigned integer multiplication and store it in the correct registers. We handled multiplication overflow by storing the 16 least significant bits into the register we specified.

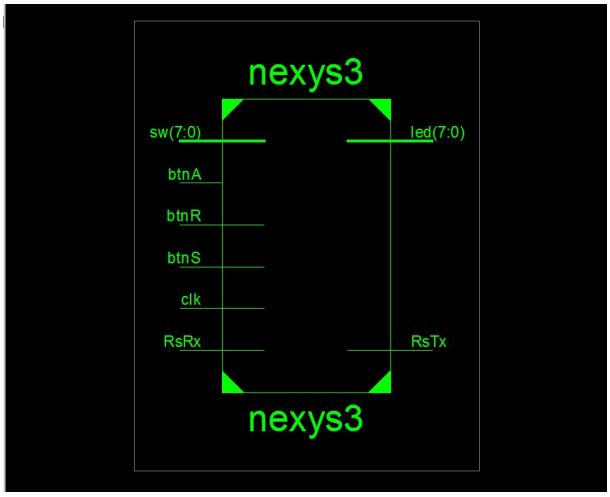



```
module seq_mult(
   // Outputs
   o_data, o_valid,
   // Inputs
   i_data_a, i_data_b, i_valid, clk, rst
   );

`include "seq_definitions.v"

   output [alu_width-1:0] o_data;
   output                 o_valid;

   input  [alu_width-1:0] i_data_a;
   input  [alu_width-1:0] i_data_b;
   input                  i_valid;

   input                  clk;
   input                  rst;

   assign o_valid = i_valid;
   assign o_data  = i_data_a * i_data_b;

endmodule // seq_mult
```

To implement our multiplier, we duplicated all of the inputs that would normally go the seq_add module (which was provided for us). The only real change that we had to make was to make o_data = i_data_a * i_data_b. This allows us to output a multiplication instead of an addition.

Task 3: A Separate SEND Button

Our third task was to implement a separate send button. Originally, we only have an execute button that is used to execute all types of instructions (push, add, mult, and send) input by using the slider switches. We were tasked to implement a send button which acts as a conditional statement to prevent error. The send button will only execute the instruction if and only if the send instruction is input using the slider switches. If the send button is pressed while the switches are encoding a non-send instruction, then the send button does nothing.




```
// ========================================================================
// Instruction Stepping Control / Debouncing
// ========================================================================

always @ (posedge clk)
  if (rst)
    begin
       inst_wd[7:0] <= 0;
       step_d[2:0]  <= 0;
    end
  else if (clk_en) // Down sampling
    begin
       inst_wd[7:0] <= sw[7:0];
       step_d[2:0]  <= {btnS, step_d[2:1]};
    end

// Detecting posedge of btnS
wire is_btnS_posedge;
assign is_btnS_posedge = ~ step_d[0] & step_d[1];
always @ (posedge clk)
  if (rst)
    inst_vld <= 1'b0;
  else if (clk_en_d)
    inst_vld <= is_btnS_posedge;
 else
   inst_vld <= 0;

always @ (posedge clk)
  if (rst)
    inst_cnt <= 0;
  else if (inst_vld)
    inst_cnt <= inst_cnt + 1;

assign led[7:0] = inst_cnt[7:0];
```

```
module nexys3 (/*AUTOARG*/
   // Outputs
   RsTx, led,
   // Inputs
   RsRx, sw, btnS, btnR, btnA, clk
   );
```

To implement the SEND button, we enable another button connection in the ucf file and add our new button called btnS into our nexys3 module. The code in nexys3.v basically performs signal processing so that our btnS variable receives signal from the physical button. We also added the sequencer module which has the ALU that processes the instructions.

Task 4: Nicer UART Output

Our fourth task was to implement a nicer print instruction. Currently, the testbench outputs only the value of our registers, but not the register number. We had to modify the uart_top.v and nexys3.v file in order to change the outputs to include the register number. The modified output would now be “R0:0003” instead of just “0003”.

```
always @ (posedge clk)
  if (rst)
    state <= stIdle;
  else
    case (state)
      stIdle:
        if (i_tx_stb)
          begin
             state   <= stR;
             tx_data <= i_tx_data;
          end
      stCR:
        if (~tfifo_full) state <= stIdle;
      default:
        if (~tfifo_full)
          begin
             state   <= state + 1;
             if (state > 3) begin  // We don't want to destroy the data while printing out the prefix
               tx_data <= {tx_data,4'b0000};
             end
          end
    endcase // case (state)
```

```
parameter stIdle = 0;
parameter stR = 1;
parameter stReg = 2;
parameter stColon = 3;
parameter stNib1 = 4;
parameter stNL   = uart_num_nib+3+1;
parameter stCR   = uart_num_nib+3+2;
```

Our task was to add a prefix to the uart output of the form “R0:”. In order to do this we added states to the state machine. The states that we added are stR, stReg, stColon. We edited the case statement that manages state so that the R, Reg, and Colon states, are visited before the nib states. Lastly, we don’t flush tx_data unless the state > 3. This prevents us from loosing output data

Task 5: An Easier Way to Load Sequencer Program

Our fifth task was to load the sequencer program directly from a text file. The purpose for this is to optimize the loading process of our sequencer. We had to modify our testbench such that it loads seq.code into an array and make it execute every instruction on the file. It is important to note that the name of the file has to be “seq.code,” can only be up to 1024 lines long, the first line of the file contains a binary number that indicates how many instructions there are in the file, and the remaining n-1 lines contained a single binary instruction.

```
$readmemb("seq.code", mem);
for (i = 1; i <= mem[0]; i = i + 1) begin
    // switch on the opcode
    case(mem[i][7:6])
        2'b00: tskRunPUSH(mem[i][5:4], mem[i][3:0]);
        2'b01: tskRunADD(mem[i][5:4], mem[i][3:2], mem[i][1:0]);
        2'b10: tskRunMULT(mem[i][5:4], mem[i][3:2], mem[i][1:0]);
        2'b11: tskRunSEND(mem[i][5:4]);
    endcase
end
```

Our task was to implement an easier way to load programs. Basically, we needed to read the raw bytes from a text file. After reading these ascii encoded bytes using readmemb. The first line contains the number of instructions, so it is used to control our loop. Then, we just need to decode each instruction by switching on the first two bytes and then passing in the correct arguments to the other instructions.

Task 6: Fibonacci Numbers

Our final task was to output the first ten numbers of the Fibonacci series from the UART. For this final task, we used our completed result from task 5 (put lines of instructions in seq.code) in order to print out the first ten numbers of the Fibonacci sequence.

```
// Fibonacci Program
tskRunPUSH(0, 1);
tskRunPUSH(1, 1);
tskRunSEND(0);  // 1
tskRunSEND(1);  // 1

tskRunADD(0, 1, 2);
tskRunSEND(2);  // 2

tskRunADD(1, 2, 0);
tskRunSEND(0);  // 3

tskRunADD(2, 0, 1);
tskRunSEND(1);  // 5

tskRunADD(0, 1, 2);
tskRunSEND(2);  // 8

tskRunADD(1, 2, 0);
tskRunSEND(0);  // 13

tskRunADD(2, 0, 1);  // 21
tskRunSEND(1);

tskRunADD(0, 1, 2);  // 34
tskRunSEND(2);

tskRunADD(1, 2, 0);  // 55
tskRunSEND(0);
```

The above code shows a program we wrote that outputs the first ten Fibonacci numbers. The command “tskRunPUSH(0,1) and tskRunPUSH(1,1)” puts the decimal value 1 in register 0 and register 1. The command “tskRunSEND(0) and tskRunSEND(1)” prints out the value of register 0 and 1 respectively. We need at least 3 registers in order to print out the Fibonacci numbers because we need two registers to be the operands and one to be the sum. The “tskRunSEND(0, 1, 2)” instruction essentially puts the value of register 0 and register 1 to register 2. We then recycle this instruction and use the “tskRunSEND” instruction to output the Fibonacci numbers.

Simulation documentation

Output: Missing Multiply Operation

The code below shows the testbench code for the demo program:

```
// Demo Program
tskRunPUSH(0,4);
tskRunPUSH(0,0);
tskRunPUSH(1,3);
```

```
tskRunMULT(0,1,2);
tskRunADD(2,0,3);
tskRunSEND(0);
tskRunSEND(1);
tskRunSEND(2);
tskRunSEND(3);
```

The code below shows the output for our multiply instruction:

```
       5 ... led output changed to 00000000
 1501000 ... Running instruction 00000100
 5243925 ... instruction 00000100 executed
 5243925 ... led output changed to 00000001
 6001000 ... Running instruction 00000000
 9176085 ... instruction 00000000 executed
 9176085 ... led output changed to 00000010
10501000 ... Running instruction 00010011
14418965 ... instruction 00010011 executed
14418965 ... led output changed to 00000011
15001000 ... Running instruction 10000110
18351125 ... instruction 10000110 executed  // multiply occurs here
18351125 ... led output changed to 00000100
19501000 ... Running instruction 01100011
23594005 ... instruction 01100011 executed
23594005 ... led output changed to 00000101
24001000 ... Running instruction 11000000
27526165 ... instruction 11000000 executed
27526165 ... led output changed to 00000110
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 34 (4)
27600815 UART0 Received byte 30 (0)
28501000 ... Running instruction 11010000
31458325 ... instruction 11010000 executed
31458325 ... led output changed to 00000111
31466855 UART0 Received byte 52 (R)
31477875 UART0 Received byte 31 (1)
31488895 UART0 Received byte 3a (:)
31499915 UART0 Received byte 30 (0)
31510935 UART0 Received byte 30 (0)
31521955 UART0 Received byte 30 (0)
31532975 UART0 Received byte 33 (3)
33001000 ... Running instruction 11100000
36701205 ... instruction 11100000 executed
36701205 ... led output changed to 00001000
```

```
36709735 UART0 Received byte 52 (R)
36720755 UART0 Received byte 32 (2)
36731775 UART0 Received byte 3a (:)
36742795 UART0 Received byte 30 (0)
36753815 UART0 Received byte 30 (0)
36764835 UART0 Received byte 43 (C)
36775855 UART0 Received byte 30 (0)
37501000 ... Running instruction 11110000
40633365 ... instruction 11110000 executed
40633365 ... led output changed to 00001001
40641895 UART0 Received byte 52 (R)
40652915 UART0 Received byte 33 (3)
40663935 UART0 Received byte 3a (:)
40674955 UART0 Received byte 30 (0)
40685975 UART0 Received byte 31 (1)
40696995 UART0 Received byte 30 (0)
40708015 UART0 Received byte 30 (0)
```

Output: Nicer UART and Fibonacci numbers

```
      5 ... led output changed to 00000000
 1501000 ... Running instruction 00000001
 5243925 ... instruction 00000001 executed
 5243925 ... led output changed to 00000001
 6001000 ... Running instruction 00010001
 9176085 ... instruction 00010001 executed
 9176085 ... led output changed to 00000010
10501000 ... Running instruction 11000000
14418965 ... instruction 11000000 executed
14418965 ... led output changed to 00000011
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)
15001000 ... Running instruction 11010000
18351125 ... instruction 11010000 executed
18351125 ... led output changed to 00000100
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)
19501000 ... Running instruction 01000110
```

```
23594005 ... instruction 01000110 executed
23594005 ... led output changed to 00000101
24001000 ... Running instruction 11100000
27526165 ... instruction 11100000 executed
27526165 ... led output changed to 00000110
27534695 UART0 Received byte 52 (R)
27545715 UART0 Received byte 32 (2)
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)
28501000 ... Running instruction 01011000
31458325 ... instruction 01011000 executed
31458325 ... led output changed to 00000111
33001000 ... Running instruction 11000000
36701205 ... instruction 11000000 executed
36701205 ... led output changed to 00001000
36709735 UART0 Received byte 52 (R)
36720755 UART0 Received byte 30 (0)
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)
37501000 ... Running instruction 01100001
40633365 ... instruction 01100001 executed
40633365 ... led output changed to 00001001
42001000 ... Running instruction 11010000
45876245 ... instruction 11010000 executed
45876245 ... led output changed to 00001010
45884775 UART0 Received byte 52 (R)
45895795 UART0 Received byte 31 (1)
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)
46501000 ... Running instruction 01000110
49808405 ... instruction 01000110 executed
49808405 ... led output changed to 00001011
51001000 ... Running instruction 11100000
55051285 ... instruction 11100000 executed
55051285 ... led output changed to 00001100
55059815 UART0 Received byte 52 (R)
55070835 UART0 Received byte 32 (2)
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)
55501000 ... Running instruction 01011000
```

Output: An Easier Way to Load Sequencer Program

```
      $readmemb("seq.code", mem);
      for (i = 1; i <= mem[0]; i = i + 1) begin
          // switch on the opcode
          case(mem[i][7:6])
             2'b00: tskRunPUSH(mem[i][5:4], mem[i][3:0]);
             2'b01: tskRunADD(mem[i][5:4], mem[i][3:2], mem[i][1:0]);
             2'b10: tskRunMULT(mem[i][5:4], mem[i][3:2], mem[i][1:0]);
             2'b11: tskRunSEND(mem[i][5:4]);
          endcase
      end
```

The command readmemb puts the code in the file “seq.code” into the `mem` array. We loop through the array and then we switch on the first 2 bits which determines the instruction (PUSH, ADD, MULT, or SEND). Our case statement is a mini instruction decoder and handles each operation separately.

Output: A Separate Send Button

The code below is our logic for our new SEND button. It is essentially mimicking the original execute button but instead, it is placed on another button and only works IF AND ONLY IF the sequencer is instructed to do the send instruction AND the new button is pressed.

```
// ============================================================================
// Instruction Stepping Control / Debouncing
// ============================================================================

   always @ (posedge clk)
     if (rst)
       begin
          inst_wd[7:0] <= 0;
          step_d[2:0]  <= 0;
       end
     else if (clk_en) // Down sampling
       begin
          inst_wd[7:0] <= sw[7:0];
          step_d[2:0]  <= {btnS, step_d[2:1]};
       end

// Detecting posedge of btnS
   wire is_btnS_posedge;
   assign is_btnS_posedge = ~ step_d[0] & step_d[1];
   always @ (posedge clk)
```

```
  if (rst)
    inst_vld <= 1'b0;
  else if (clk_en_d)
    inst_vld <= is_btnS_posedge;
 else
   inst_vld <= 0;

always @ (posedge clk)
  if (rst)
    inst_cnt <= 0;
  else if (inst_vld)
    inst_cnt <= inst_cnt + 1;

assign led[7:0] = inst_cnt[7:0];
```

The code below binds our newly programmed SEND button to our sequencer. inst_vld determines whether or not our new SEND button is pressed and inst_wd transfers the instructions to our sequencer.

```
// ============================================================================
// Sequencer
// ============================================================================

   seq seq_ (// Outputs
             .o_tx_data                 (seq_tx_data[seq_dp_width-1:0]),
             .o_tx_valid                (seq_tx_valid),
             .o_print_reg               (print_reg),
             // Inputs
             .i_tx_busy                 (uart_tx_busy),
             .i_inst                    (inst_wd[seq_in_width-1:0]),
             .i_inst_valid              (inst_vld),
             /*AUTOINST*/
             // Inputs
             .clk                       (clk),
             .rst                       (rst));
```

Conclusion

In conclusion, lab 2 introduced us to a very exciting small scale FPGA project that helped us solidify our understanding of how software and hardware design come as one. We implemented a multiply operation in order to have our sequencer execute multiplication instructions. We also implemented our own SEND button. Then, we created a nicer UART output, which conveyed the register number of the value that we were printing out. In addition, we found an easier way

to load the sequencer program and outputted the first ten numbers of the Fibonacci sequence using our new loading module.

We faced many difficulties regarding how Verilog worked with the FPGA board. This was our first encounter with software implementation with an FPGA board. Understanding how each file in the source code was linked to one another was a big challenge for our group. However, once we figured how each file linked to another, we managed to successfully finish our tasks. We learned how to deal with these problems by searching through verilog documentation and online resources. One problem we came across was the implementation of the separate SEND button. We tried implementing the separate SEND button, but once we thought we finished, it completely broke the simulations of our other completed tasks. We decided to skip this task first and then come back to it later once we finished the other tasks.

The lab spec was really thorough and provided great explanations on what we needed to do but we thought that more guidance on the high level linkage of source files would help tremendously. The lab did tell us, for the most part, which files to modify but it was a little challenging figuring out how all of the files connected with one another. Overall, the lab was still very fun, as it taught us how to incorporate the FPGA board with Verilog software in order to implement a sequencer.