ECE 385

Fall 2019

Experiment #5

An 8-Bit Multiplier in SystemVerilog

Pouya Akbarzadeh and Patrick Stach

Section: ABC

Lab TA: Vikram Anjur, Yuhong Li

Index

5.1 Introduction

5.1.1 Goal of the Lab

5.1.2 Summary of Operation

5.2 - Diagrams

5.2.1 Top Level Block Diagram

5.2.2 Finite State Machine (State Diagram for Control Unit)

5.3 Modules

5.3.1 Processor

5.3.2 9-Bit Adder

5.3.3 Control Unit

5.3.4 Hex Driver

5.3.5 Full Adder

5.3.6 Eight Shift Register

5.3.7 Sync

5.3.8 Register

5.3.9 Test Bench

5.4 Pre-Lab Questions

5.4.1 (A)

5.4.2 (B)

5.4.2.1

5.4.2.2

5.4.2.3

5.5 Post Lab Questions

5.5.1

5.5.2

5.6 Conclusion

5.1 Introduction

5.1.1 The Goal of the Lab

The goal of this lab was to learn more about SystemVerilog by designing and coding a multiplier for two 8-bit 2's complement numbers and afterwards run it on the FPGA board.

5.1.2 Summary of Operation

The core of our operation is the control unit. Based on the algorithm, we end up right shifting a total of eight times- but at every step before we shift, we will determine we will just shift or add and then shift. We look at the LSB- B[0], or "M" to determine whether we need to add before shifting. If M =1, we will add and shift, and if M=0, we will just shift. Our state machine will automatically transition between the states based on this M value, i.e: if we are at INIT and M=1, we will go to ADD1 then SHIFT1, and if M=0 we will just go to SHIFT1. The INIT state is responsible for clearing XA and will automatically be entered in after run is pressed in the START state.

To clarify, a caveat of the algorithm is that the last ADD state is actually a SUBTRACT, so if M=1 we subtract then shift, or if M=0 we just shift.

We also have an END state that it will stay in if RUN is still pressed after the multiplication is complete. When RUN is released, it will go back to the START state until RUN is pressed again.

At each state, we have various control signals that the FSM will be outputting: ADD at the "ADD" or "SUB" states to load XA from the output of the adder, SUB to determine whether we need to use XOR gates to invert one of the signals to subtract, Shift_En at the SHIFT states to enable the shift mode in the shift registers.

The "X" register is important in preserving the sign extend of XA when shifting. It gets updated at every add operation, and we need it to store the sign so that when A shifts right that we preserve the sign of A when it shifts right.

As for user input: RESET will clear X, A and B to 0 (will also interrupt a multiplication cycle to go back to the START or END based on if RUN is still pressed) , CLR_A_LD_B will clear X and A while loading the switch values into B. Then the value the switches are set to right before RUN is pressed will be the value you will be putting in the adder.

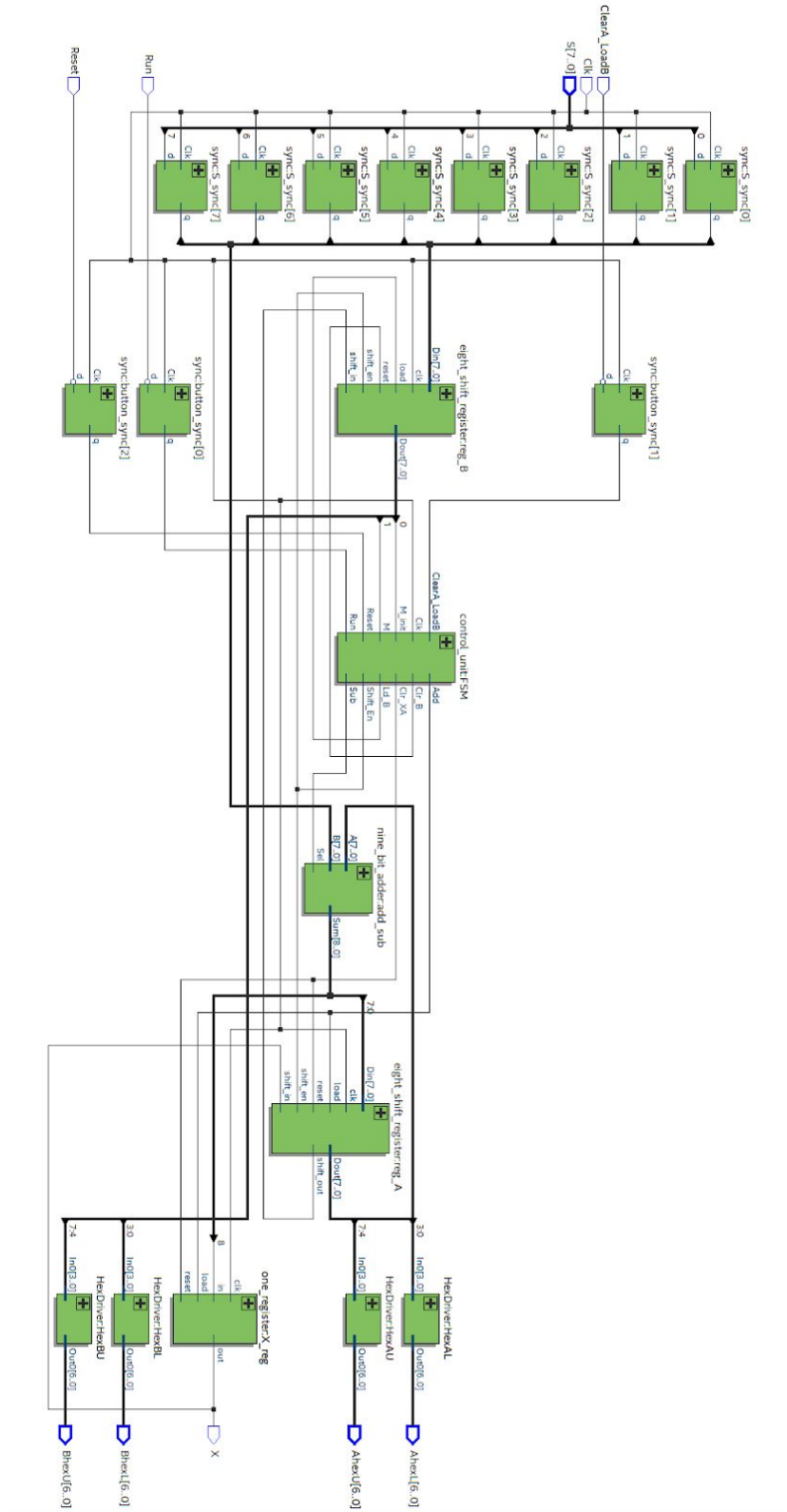
These control signals are also generated in the FSM with combinational logic and outputted in the LD and CLR pins respectively to clear or load a register.

The limitations of continuous multiplications is that in order for our algorithm to work, we need to clear XA at the start of the multiplication cycle. XAB is large enough to store a 17-bit answer as a result of multiplication, so if we try to consecutively multiply a number that "overflowed" into XA, we will not get the correct answer since the bits in XA will get reset to 0 and we will thus be using the wrong multiplier. Our algorithm will only work if the product that we will use in consecutive multiplication can be completely contained in B.

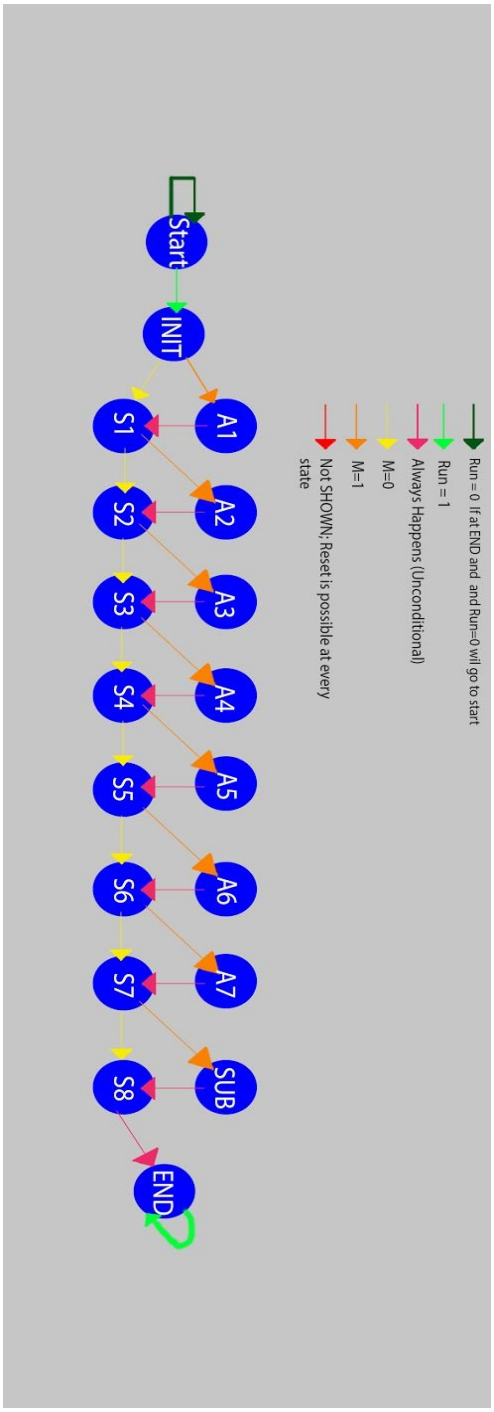
The advantage of using the paper-pencil method is that it is easier to understand conceptually and can be done relatively fast if we don't need to process a lot of multiplications or use very big numbers. However, the big disadvantage is that we cannot practically implement this in a digital system since the paper-pencil method is using decimal while the shift-add algorithm is in binary.

5.2 - Circuits and Explanation

5.2.1 Top Level Block Diagram



5.2.2 Finite State Machine (State Diagram for Control Unit)



5.3 Modules

5.3.1 Processor

```
module Processor
(
    input logic Clk, // Clock
    input logic Run, // Executes operation
    input logic ClearA_LoadB,
    input logic Reset, // Reset (i.e registers to zero, etc)
    input logic[7:0] S,
    output  logic[6:0] AhexL, AhexU, BhexL, BhexU,
    output logic X
    );

    //local logic variables go here
    logic Reset_SH, ClearA_LoadB_SH, Run_SH; // Sync HIGH for push buttons
    logic Shift_En, Add, Sub, Clr_XA, Clr_B, Ld_B; // Logic for control unit
    logic [7:0] A, B; // Registers A, B
    logic [8:0] Sum;  // Output of Add/Sub module
    logic [7:0] S_S;  // Sync LOW for switches
    logic A_shift_out;


    one_register X_reg ( .in(Sum[8]),
                         .clk(Clk),
                         .reset(Clr_XA),
                         .load (Add),
                         .out(X));


    nine_bit_adder add_sub (      .A(A),
                            .B(S_S),
                            .sel(Sub),
                            .Sum(Sum));

    eight_shift_register reg_A (  .Din(Sum[7:0]),
                            .clk(Clk),
                            .reset(Clr_XA),
                            .load(Add),
                             .shift_en(Shift_En),
                            .shift_in(X),
                            .shift_out(A_shift_out),
                            .Dout(A));

    eight_shift_register reg_B  (   .Din(S_S),
                            .clk(Clk),
                            .reset(Clr_B),
                            .load(Ld_B),
                            .shift_en(Shift_En),
                            .shift_in(A_shift_out),
                            .Dout(B));

    control_unit FSM (.Reset(Reset_SH),
                      .Clk(Clk),
                      .Run(Run_SH),
                      .ClearA_LoadB(ClearA_LoadB_SH),
                      .M(B[1]),
                      .M_init(B[0]),
                      .Shift_En(Shift_En),
                      .Sub(Sub),
                      .Clr_XA(Clr_XA),
                      .Clr_B(Clr_B),
                      .Ld_B(Ld_B),
                      .Add(Add));

    HexDriver HexAL (.In0(A[3:0]),.Out0(AhexL) );
    HexDriver HexBL (.In0(B[3:0]), .Out0(BhexL) );
    HexDriver HexAU (.In0(A[7:4]), .Out0(AhexU) );
    HexDriver HexBU (.In0(B[7:4]), .Out0(BhexU) );



    sync button_sync[2:0] (Clk, {~Reset, ~ClearA_LoadB, ~Run}, {Reset_SH, ClearA_LoadB_SH, Run_SH});
    sync S_sync[7:0] (Clk, S, S_S);

endmodule
```

Inputs: clk, Run, ClearA_LoadB Reset, [7:0] s
Outputs: x, [6:0] AhexL, AhexU, BhexL, BhexU
Description: The processor is the top level. The processor instantiates other modules and allows for the circuit to work, it also reads the data input from switches and outputs it on the display. The processor can be seen as the heart of the circuitry meaning it connects everything and brings the circuit to life.
Purpose: Practically it's where all the wiring goes. And it allows us to run and test everything.

5.3.2 9-Bit Adder

```
module nine_bit_adder
(
    input logic[7:0] A,B,
    input Sel,
    // Sel = 0 => Output = A + B
    // Sel = 1 => Output = A - B
    output logic[8:0] Sum
);
    logic c0, c1, c2, c3, c4, c5, c6, c7;
    full_adder fa0(.A(A[0]), .B(Sel^B[0]), .Cin(Sel),.Sum(Sum[0]), .Cout(c0));
    full_adder fa1(.A(A[1]), .B(Sel^B[1]), .Cin(c0), .Sum(Sum[1]), .Cout(c1));
    full_adder fa2(.A(A[2]), .B(Sel^B[2]), .Cin(c1), .Sum(Sum[2]), .Cout(c2));
    full_adder fa3(.A(A[3]), .B(Sel^B[3]), .Cin(c2), .Sum(Sum[3]), .Cout(c3));
    full_adder fa4(.A(A[4]), .B(Sel^B[4]), .Cin(c3), .Sum(Sum[4]), .Cout(c4));
    full_adder fa5(.A(A[5]), .B(Sel^B[5]), .Cin(c4), .Sum(Sum[5]), .Cout(c5));
    full_adder fa6(.A(A[6]), .B(Sel^B[6]), .Cin(c5), .Sum(Sum[6]), .Cout(c6));
    full_adder fa7(.A(A[7]), .B(Sel^B[7]), .Cin(c6), .Sum(Sum[7]), .Cout(c7));
    full_adder fa8(.A(A[7]), .B(Sel^B[7]), .Cin(c7), .Sum(Sum[8]));

endmodule
```

Inputs: [7:0] A, B, Sel
Outputs: [8:0] Sum
Description: While the name is adder. This module has the ability to also subtract. .B(Sel^B[*bit-number*]) performs XOR operation on the select bit and the bit at hand. This logic operation allows for subtraction to take place where needed.
Purpose: Made to do the adding and subtracting needed for the shift-add algorithm.

5.3.3 Control Unit

```
1   module control_unit
2   (
3       input logic Reset, Clk, Run, ClearA_LoadB, M, M_init,
4       output logic Shift_En, Clr_XA, Clr_B, Ld_B, Sub, Add);
5       logic initialize;
6
7       enum logic [4:0]
8       {  START, END, INIT,
9          S1, S2, S3, S4, S5, S6, S7, S8,
10         A1, A2, A3, A4, A5, A6, A7, SUB} curr_state, next_state;
11
12      assign Clr_XA = Reset + ClearA_LoadB+ initialize;
13      assign Clr_B = Reset;
14      assign Ld_B = ClearA_LoadB;
15
16      always_ff @ (posedge Clk)
17      begin
18
19         if (Reset)
20            if (Run)
21               curr_state <= END;
22            else
23               curr_state <= START;
24         else
25            curr_state <= next_state;
26      end
```

***Not the entire code*
Inputs: Reset, Clk, Run, ClearA_LoadB, M, M_init
Outputs: Shift_En, Clr_XA, Clr_B, Ld_B, Sub, Add
Description: This is file/segment of the project is a finite state machine implementation. Each state has its own output and next level logic. If we were going to continue with the analogy of processor being the heart it is fair to consider the control unit the brain. For the diagram of the control unit look at section 5.2.2.
Purpose: This part of the circuit chooses if we are adding, shifting, subtracting, or in a hold state.

5.3.4 Hex Driver

```
module HexDriver (input  logic[3:0]  In0,
                  output logic [6:0]  Out0);

    always_comb
    begin
       unique case (In0)
           4'b0000   : Out0 = 7'b1000000; // '0'
           4'b0001   : Out0 = 7'b1111001; // '1'
           4'b0010   : Out0 = 7'b0100100; // '2'
           4'b0011   : Out0 = 7'b0110000; // '3'
           4'b0100   : Out0 = 7'b0011001; // '4'
           4'b0101   : Out0 = 7'b0010010; // '5'
           4'b0110   : Out0 = 7'b0000010; // '6'
           4'b0111   : Out0 = 7'b1111000; // '7'
           4'b1000   : Out0 = 7'b0000000; // '8'
           4'b1001   : Out0 = 7'b0010000; // '9'
           4'b1010   : Out0 = 7'b0001000; // 'A'
           4'b1011   : Out0 = 7'b0000011; // 'B'
           4'b1100   : Out0 = 7'b1000110; // 'C'
           4'b1101   : Out0 = 7'b0100001; // 'D'
           4'b1110   : Out0 = 7'b0000110; // 'E'
           4'b1111   : Out0 = 7'b0001110; // 'F'
           default   : Out0 = 7'bxxxxxxx; // undefined output
       endcase
    end

endmodule
```

Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: The HexDriver was provided to us during Experiment 4 and was simply recycled for this experiment. The HexDriver translates a binary input representing a number into the specific segments of a 7-segment display that would need to be able to display that number in Hex
Purpose: Used for displaying the final answer on the FPGA board.

5.3.5 Full Adder

```
module full_adder
(
    input A,B,Cin,
    output Sum, Cout);

    assign Sum = A ^ B ^Cin;
    assign Cout = (A&B) | (B&Cin) | (A&Cin);

endmodule
```

Inputs: A, B, Cin
Outputs: Sum, Cout
Description: The Full-Adder is the basic block which is again recycled from a previous lab. This module performs a bitwise add between A and B and also the carry in (Cin). It outputs the sum and carryout (Cout) of the operation. It could be considered the foundation of this entire lab.
Purpose: Find the Sum and the CarryOut of the calculation. Used to build other adders.

5.3.6 Eight Shift Register

```
module eight_shift_register

   (  input logic [7:0] Din,
      input clk, reset, load, shift_en, shift_in,
      output logic shift_out,
      output logic [7:0] Dout );

      always_ff @ (posedge clk)
      begin
         if (reset)
            Dout <= 8'h0;
         else if (load)
            Dout <= Din;
         else if (shift_en)
            Dout <= {shift_in, Dout[7:1]};
      end

      assign shift_out = Dout[0];

endmodule
```

Inputs: [7:0] Din, Clk, rest, load, shift_en, shift_in
Outputs: shift_out, [7:0] Dout
Description: It is an 8-bit shift register that is positive edge triggered. It has reset bit which will clear the register and a load bit which will enable loading data to the register
Purpose: Used for A and B to store and shift data

5.3.7 Sync

```
module sync (
   input  logic clk, d,
   output logic q
);

   always_ff @ (posedge clk)
   begin
      q <= d;
   end

endmodule
```

Inputs: clk, d
Outputs: q

Description: Synchronizes asynchronous inputs such as buttons with the clock
Purpose: Allows everything to be synchronous and work together.

5.3.8 One Bit Register

```
module one_register

   (  input logic in,
      input logic clk, reset, load,
      output logic out );
      always_ff @ (posedge clk)
      begin
         if (reset)
            out <= 0;
         else if (load)
            out <= in;
      end

endmodule
```

Inputs: in, clk, reset, load
Outputs: out
Description: It is a 1 bit register, to store 1 bit of information. Gets updated on positive edge of clock based on reset or load select bit.
Purpose: To store the 'X' value. It maintains its value for sign extend when you are not adding and shifting.

5.3.9 Test Bench

```
//////////////////////////////////////////////////////////
// TEST CASE 3: -12*9
//Toggle Reset
#40 Reset=0;
#2 Reset=1;

// Set switches to +9
 S= 8'b0001001;

// Load +9 into B
#2 ClearA_LoadB=0;
#2 ClearA_LoadB=1;

// Set switches to -12
#2 S= 8'b11110100;

//Run -12*9
#2 Run=0;
#2 Run=1;
```

**One of many test cases (Used for description)
Inputs: NONE
Outputs: NONE
Description: First at "A" RESET is pressed and the next clock cycle X, A, and B are cleared. At "B" CLR_A_LD_B is pressed, which in the next clock cycle clears X and A, and loads 9 from the switches into B. Before we press RUN, we change the switches to equal -12.
Purpose: Test out different cases such as ++/+-/-+/-- and test other features such as reset, and clear.

5.4 Pre-Lab Questions

5.4.1 (A)

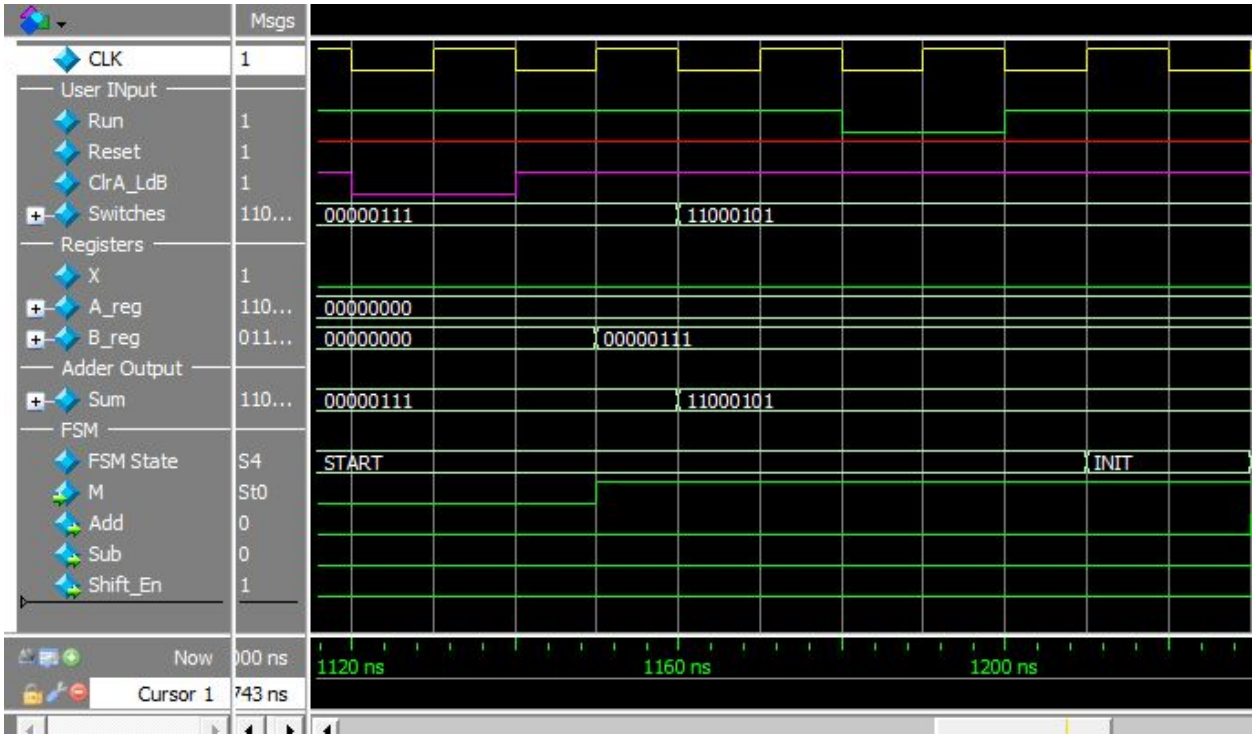
Rework the 8-bit multiplication example presented in the table form at the beginning of this assignment. Use Multiplier B = 7, and Multiplicand S = -59. Note that this is different than the case when B = -59 and S = 7.

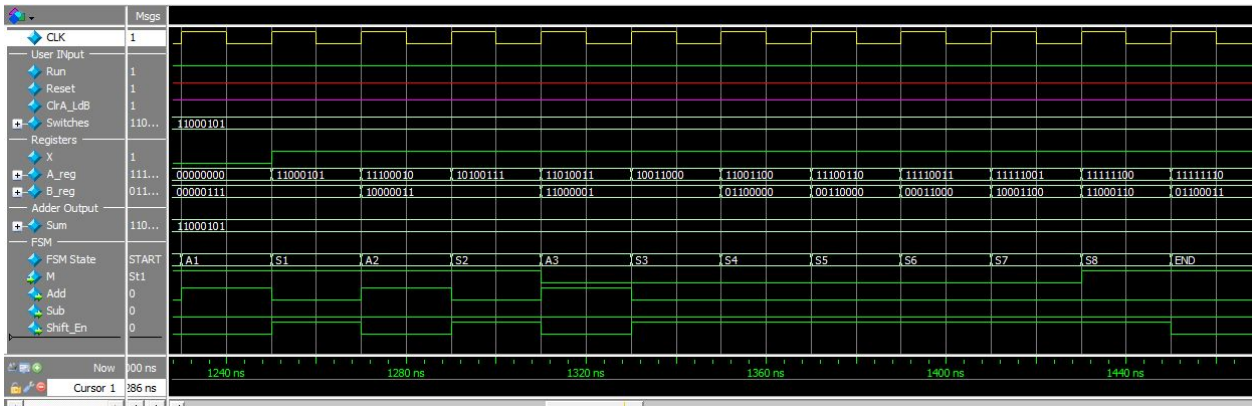
-59 = 11000101

+7 = 00000111

Function	X	A	B	M	Comments for Next Step
ClearA_LDB	0	0000 0000	0000 0111	1	Initialize step clears X and A before operation (not needed here but needed if using consecutive multiplication
INIT	0	0000 0000	0000 0111	1	M = 1 so we add then shift
A1	1	1100 0101	0000 0111	1	Added, now we shift next
S1	1	1110 0010	1000 0011	1	M=1 so we add then shift
A2	1	1010 0111	1000 0011	1	Added, now we shift next
S2	1	1101 0011	1100 0001	1	M=1 so we add then shift
A3	1	1001 1000	1100 0001	1	Added, now we shift next
S3	1	1100 0011	0110 0000	0	M=0 so we just shift
S4	1	1110 0110	0011 0000	0	M=0 so we just shift
S5	1	1111 0011	0001 1000	0	M=0 so we just shift
S6	1	1111 1001	1000 1100	0	M=0 so we just shift
S7	1	1111 1100	1100 0110	0	M=0 so we just shift
S8	1	1111 1110	0110 0011	1	Halt since we the last shift
END	1	1111 1110	0110 0011	1	DONE

***Verified with Modelsim below:**





5.4.2 (B)

Design, document, and implement the 8-bit multiplier in SystemVerilog.

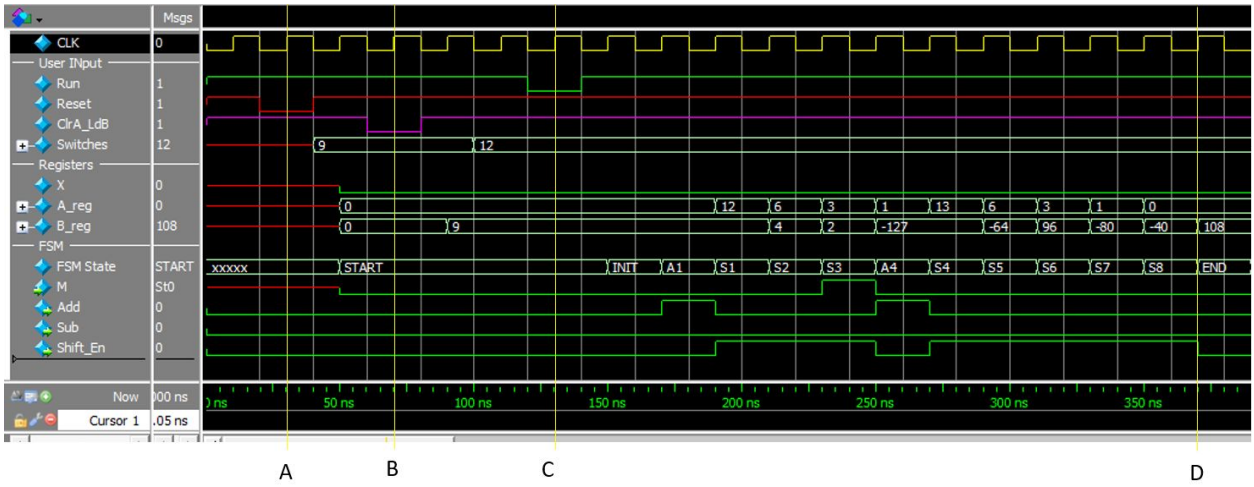
(DONE)

5.4.2.1 Your code for the 8-bit multiplier. You can bring the code to the lab using a USB storage device, FTP, or any other method. **(DONE)**

5.4.2.2 Block diagram of your design, with components, ports, and interconnections labeled. **(DONE)**

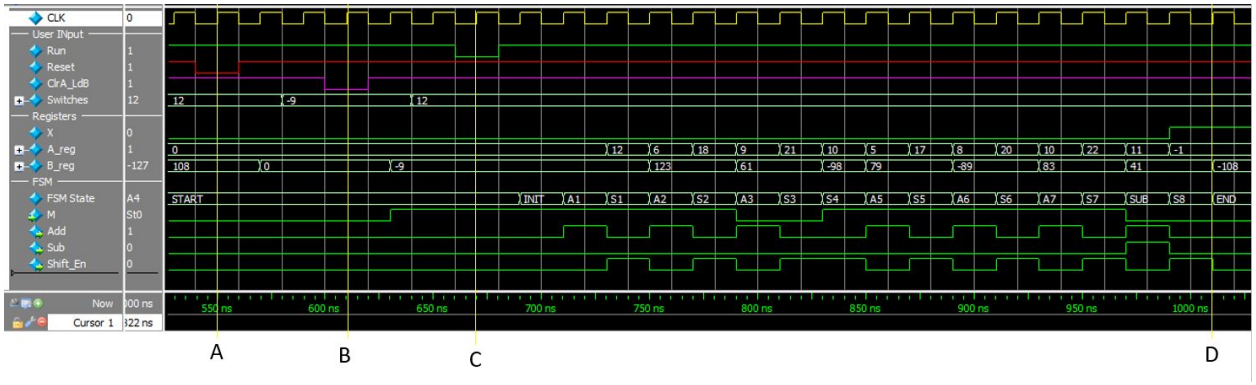
5.4.2.3 A simulation of your design showing at least one full multiplication. You should set the radix of the Switches, Aval, and Bval signals to signed decimal for readability. (Radix is set by right-clicking on a signal and selecting Properties.) **(DONE)**

Case 1: 12*9



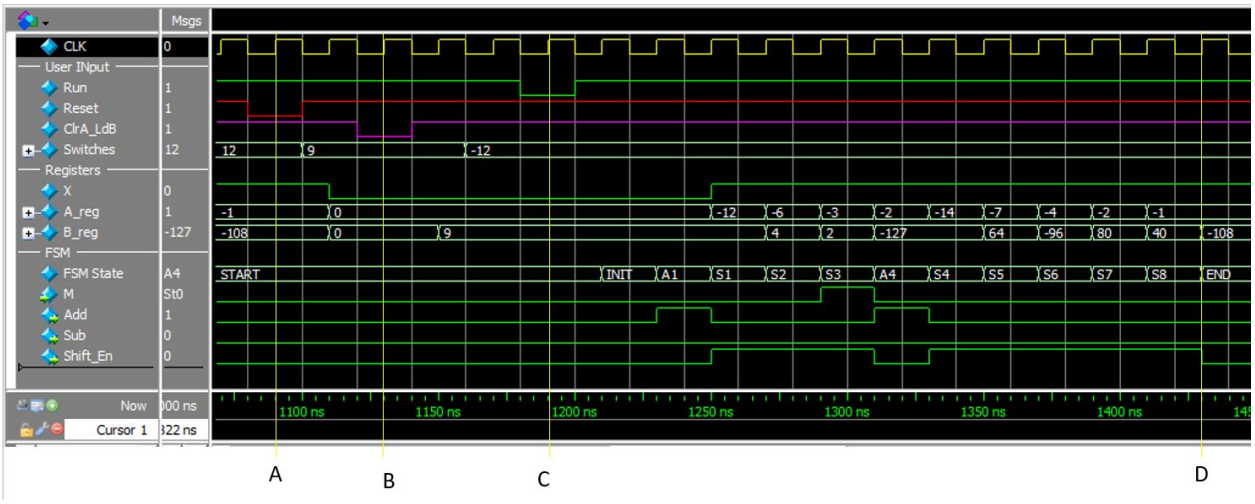
First at “A” RESET is pressed and the next clock cycle X, A, and B are cleared. At “B” CLR_A_LD_B is pressed, which in the next clock cycle clears X and A, and loads 9 from the switches into B. Before we press RUN, we change the switches to equal 12. When RUN is pressed, at “C” the next clock cycle the FSM will enter in the INIT state, clearing X and A. The FSM then automatically transitions between the states such as adding/shifting based on the LSB of XAB, M. Then based on the state we will have different output control values (Shift_En, Add, Sub). At “D” the multiplication is finished and we halt, entering the “END” state with a resulting value in XAB of 108.

Case 2: 12*-9



First at “A” RESET is pressed and the next clock cycle X, A, and B are cleared. At “B” CLR_A_LD_B is pressed, which in the next clock cycle clears X and A, and loads -9 from the switches into B. Before we press RUN, we change the switches to equal 12. When RUN is pressed, at “C” the next clock cycle the FSM will enter in the INIT state, clearing X and A. The FSM then automatically transitions between the states such as adding/shifting based on the LSB of XAB, M. Then based on the state we will have different output control values (Shift_En, Add, Sub). At “D” the multiplication is finished and we halt, entering the “END” state with a resulting value in XAB of -108.

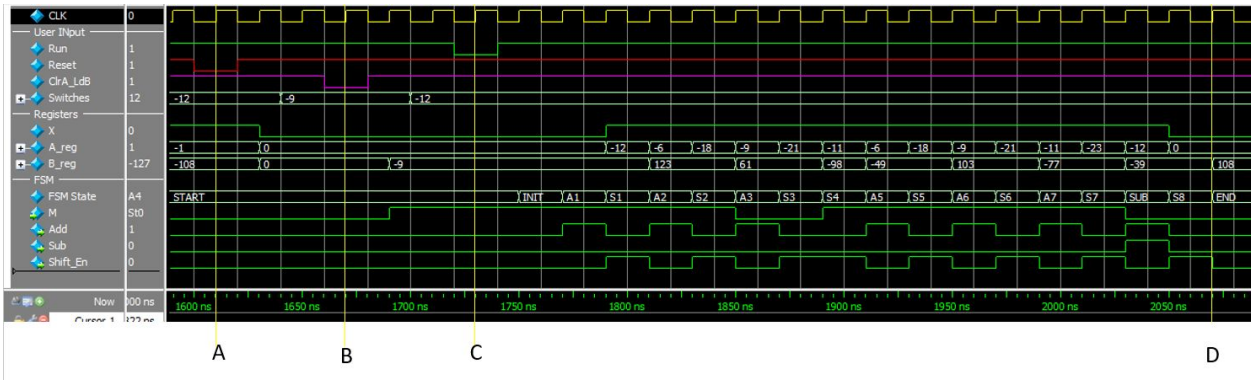
Case 3: -12* 9



First at “A” RESET is pressed and the next clock cycle X, A, and B are cleared. At “B” CLR_A_LD_B is pressed, which in the next clock cycle clears X and A, and loads -9 from the switches into B. Before we press RUN, we change the switches to equal 12. When RUN is pressed, at “C” the next clock cycle the FSM will enter in the INIT state, clearing X and A. The FSM then automatically transitions between the states such as adding/shifting based on the LSB

of XAB, M. Then based on the state we will have different output control values (Shift_En, Add, Sub). At “D” the multiplication is finished and we halt, entering the “END” state with a resulting value in XAB of -108.

Case 4: -12*-9



First at “A” RESET is pressed and the next clock cycle X, A, and B are cleared. At “B” CLR_A_LD_B is pressed, which in the next clock cycle clears X and A, and loads -9 from the switches into B. Before we press RUN, we change the switches to equal -12. When RUN is pressed, at “C” the next clock cycle the FSM will enter in the INIT state, clearing X and A. The FSM then automatically transitions between the states such as adding/shifting based on the LSB of XAB, M. Then based on the state we will have different output control values (Shift_En, Add, Sub). At “D” the multiplication is finished and we halt, entering the “END” state with a resulting value in XAB of 108.

5.5 Post Lab Questions

5.5.1

Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

LUT	96
DSP	N/A
Memory (BRAM)	0
Flip-Flop	47
Frequency	73.08 Mhz
Static Power	98.49mW
Dynamic Power	3.25mW
Total Power	140.18mW

Come up with a few ideas on how you might optimize your design to decrease the total gate count and/or to increase the maximum frequency by changing your code for the design.

One thing that added to the complexity of the FSM is that we allowed resets in the middle of the calculation. Had we not allowed resets in the middle of the calculation this would have removed extra transitions (over 18 total) which most definitely would have simplified the logic. It also likely wouldn't have caused any design issue: with such a high clock speed it is extremely unlikely that a user would be able to press reset in the miniscule amount of time it is actually in the ADD/SHIFT states.

Another thing that we could have done to optimize the design was by using less states. We had 8 total SHIFT states and 7 total ADD states + one SUB state, where in each state of the same type we had the exact same output control signals (i.e: every SHIFT state had SHIFT_EN =1, etc). Theoretically, we probably could have had one ADD state, and one SHIFT state, with a counter to keep track of the current cycle we are on to know when to end, and when the ADD state should actually be a SUB state (on last "add"). However, this would have been much more difficult to troubleshoot and would introduce even more caveats since a counter is synchronous and would have to propogate and would introduce delay that we would need to factor into our logic.

5.5.2

Make sure your lab report answers at least the following questions:

• What is the purpose of the X register. When does the X register get set/cleared?

See 5.1.2

• What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?

See 5.1.2

• What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

See 5.1.2

5.6 Conclusion

In conclusion, our implementation of this lab worked as intended. We did run into some mistakes of our implementation, however. The first issue was that we did not have an INIT state at first. This was an issue because we did not clear X and A at the beginning of the multiplication cycle.

When we fixed this, we still had another issue, since our design is synchronous, we had to factor in the issue of taking an extra clock cycle for the correct operations and values updating. For example, if you are in one one of the SHIFT states, we do not actually perform the shift until right at the end of the cycle.

Specifically the issue this was causing was that we were not transitioning into the correct states because the "M" value we were using to make decisions for state transitions had to be shifted in and was thus delayed. We got around this by making M be B[1], since by the time we actually needed to access that it would be B[0] when it shifted.

This caused an issue in the particular case where we go straight from INIT to add, the first add case. Because there were no previous shifts before this ADD state, the M value we would need to use should be B[0], not B[1].

These issues were spotted with the help of the modularity of the system, being able to unit test modules in SystemVerilog/ModelSim with testbenches and determining the exact place where something went wrong. ModelSim was extremely helpful in troubleshooting because not only would we be able to see the values at any time of our control bits, but we could symbolically see what the current state was by using the radix for states in an FSM.

By being able to work out all our quirks and issues in software, by the time we fixed the simulations it was just a matter of uploading it to the board and having it work no problem.