ECE 385

Spring 22

Experiment #4

Introduction to SystemVerilog, FPGA, CAD, and 8-bit Multipliers

Siddarth Iyer, Geitanksha Tandon

TA: Ruihao Yao (RY)

1. Introduction:

a. Summarize the basic functionality of the multiplier circuit.

In this experiment, we designed a multiplier in SystemVerilog. This multiplied two 8-bit 2's Complement numbers using an add-shift algorithm. This multiplier was displayed on the De10-Lite FPGA board. We use the add-shift system being very similar to the method done by hand on pen-and-paper, however, it uses the MSB of the 2's complement number to determine whether the last step of the add-shift system is an addition or a subtraction. We store one value in our B register, then save the value to be multiplied in the switches. The result gets added and stored in the register combination of A and B, which slowly gets shifted to the right before another add (or subtract) is performed.

00000111 (7, Multiplicand) x 11000101 (-59, Multiplier)

```
        00000111         7 (multiplicand)
       x 11000101     x (-)59 (multiplier)
        00000111      (-)413
       +00000000x
       +00000111xx
      +00000000xxx
      +00000000xxxx
     +00000000xxxxx
     +00000111xxxxxx
    -00000111xxxxxxx  Subtract (or Add 2's comp of 00000111)
    111111001100011 (2's comp of result=0000000110011101=413)
```

Figure: Multiplication of 7 x -59, done by the method we plan to implement in our multiplier.

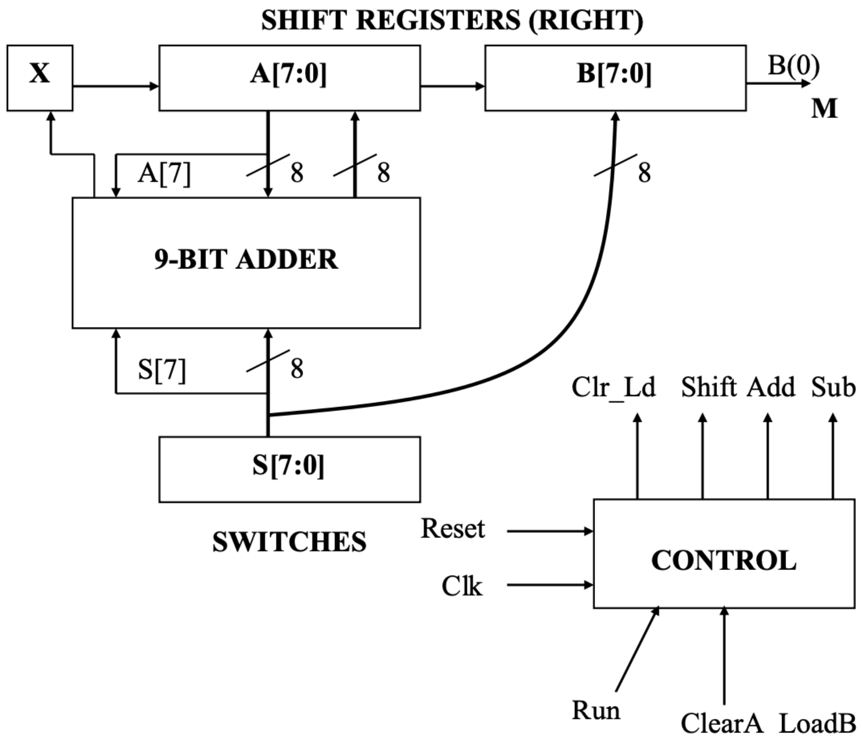


Figure: The multiplier block diagram partially set up.

2. Pre-lab question:

*a. Rework the multiplication example on page 5.2 of the lab manual, as in compute 11000101 * 00000111 in a table like the example. Note that the order of the multiplicand and multiplier are reversed from the example.*

Initial Values: X = 0; A = 0000 00000; B = 0000 0111 (**Multiplier**: B is achieved by using the button ClearA_LoadB which loads initial value into B). S = 1100 0101 (**Multiplicand**); M = B[0] (Least Significant bit of B).

Function	X	A	B	M	Comments for the next step
A - Clear, B - Load, Reset	0	0000 0000	0000 0111	1	M = 1 means that multiplicand added to A.
ADD	1	**1** 100 0101	0000 0111	1	Add complete, shift XAB.
SHIFT	1	**1** 110 0010	**1** 000 0011	1	M=1, ADD S to A.
ADD	1	**1** 010 0111	**1** 000 0011	1	Add complete, shift XAB.
SHIFT	1	**1** 101 0011	**1** 100 0001	1	M=1, ADD S to A.
ADD	1	**1** 001 1000	**1** 100 0001	1	Add complete, shift XAB.
SHIFT	1	**1** 100 1100	**0** 110 0000	0	M = 0, SHIFT.
SHIFT	1	**1** 110 0110	**0** 011 0000	0	M = 0, SHIFT.
SHIFT	1	**1** 111 0011	**0** 001 1000	0	M = 0, SHIFT.
SHIFT	1	**1** 111 1001	**1** 000 1100	0	M = 0, SHIFT.
SHIFT	1	**1** 111 1100	**1** 100 0110	0	M = 0, SHIFT.
SHIFT	1	**1** 111 1110	**0** 110 0011	0	Done

3. Written description and diagrams of multiplier circuit

a. Summary of operation:

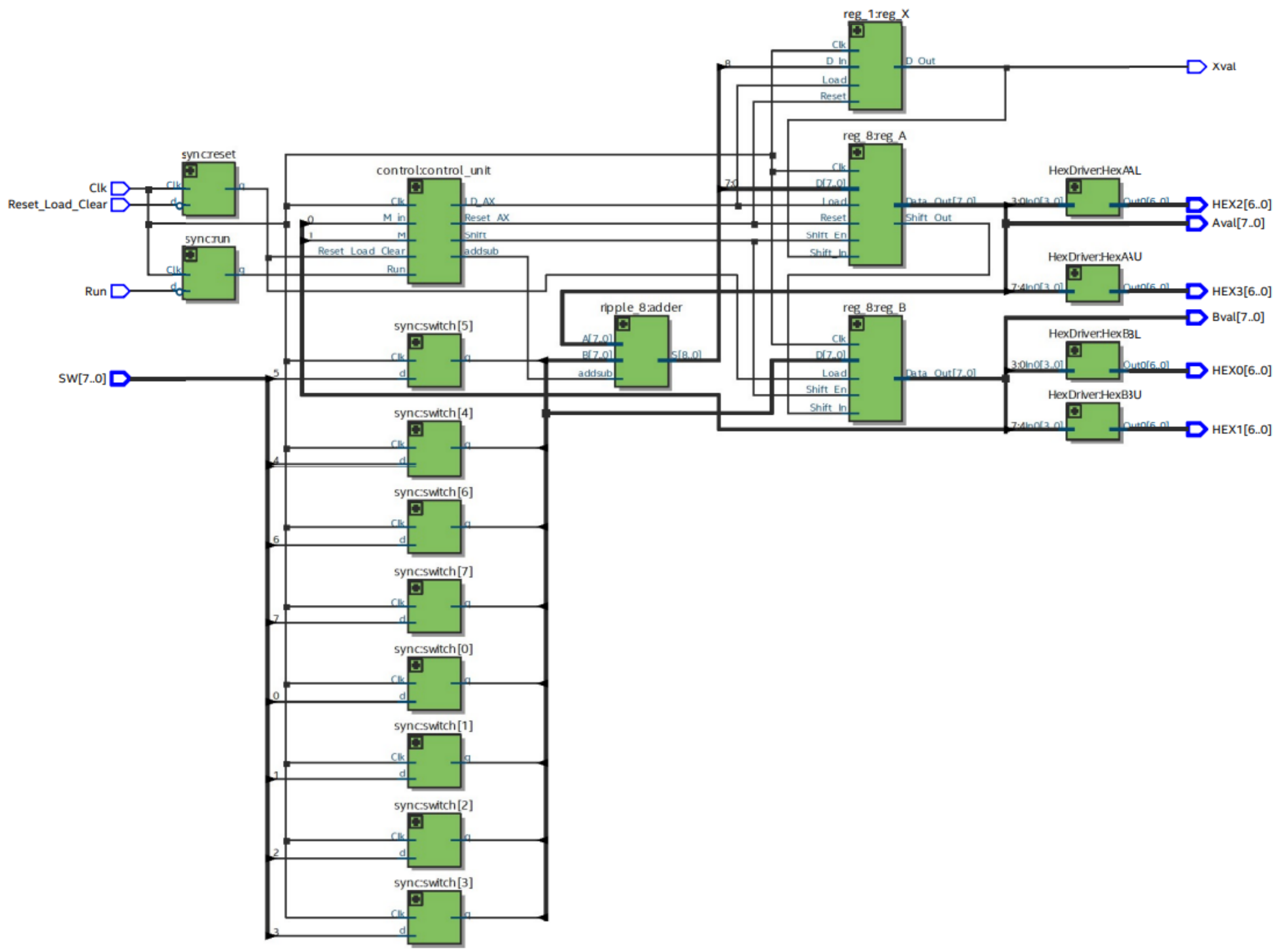
i. Explain in words how operands are loaded, how the multiplier computes its result, how the result is stored, etc.

The 4 main components of our multiplier circuit are the X register, A register, B register, and the Switches (which act as a register). The A register helps us get our 16-bit multiplied value, the B register initially stores the Multiplier, and the S register (given by switches) acts as our Multiplicand.

- The first step to execute a multiplication is loading the value of the multiplier into register B, clearing the value of register A to start a fresh multiplication, and resetting our state machine. We do that by setting the multiplier value on the switches, then clicking the ClearA_LoadB_Reset button.
- The second step is to click the run button. In this situation, we begin our multiplication. We take the LSB of the B register. If it is 0, then the shifting state takes place. In the shifting state, the A MSB gets the value from the X value register, the LSB of A becomes the MSB of B register, and all the values in A< B get shifted to the right by one bit. If it is a 1, then the value in the switches register gets added to the value in A and stored back in the register A. Then, the shift state is executed and performs the same function as mentioned above.
- This add-shift state takes place 7 times. On the 8th time, depending on the value of the X register, the last addition state either becomes a subtraction state if Xval is 1, otherwise it becomes an addition state, and then gets shifted. Once the entire addition (or subtraction, as possible in the last addition state) is done, then the Registers A, B together display a 16-bit value result.

b. Top Level Block Diagram.

i. This can be generated from the RTL viewer. Please only include the top-level diagram and not the RTL view of every module.



c. Written Description of .sv Modules

i. List all modules used in a format shown in the appendix of this document.

ii. You may insert expanded RTL diagrams of each individual module here if it is legible.

Module Descriptions:

Module: Control.sv

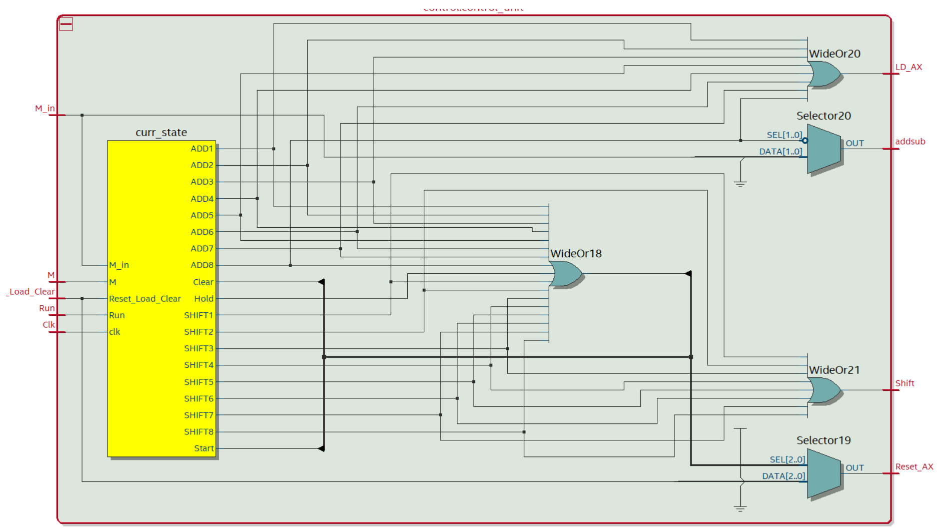
Inputs: Clk, Reset_Load_Clear, Run, M, M_in

Output: Reset_AX, Shift, addsub, LD_AX

Description: Describes an FSM with 21 states as follows:
- Clear State, which is the first state of the FSM when started, where the A, X values are reset for a new multiplication.
- Start State, which resets the values of A, X if Reset_Load_Clear is pressed to run a continuous multiplication.
- Hold State, which holds the result of the multiplication without resetting any values to be able to view the result of the multiplication.
- Shift 1,2,3,4,5,6,7,8, which shift the value of X into MSB of A, shift all bits of A to the right by one bit, shift the LSB of the A into the MSB of B, and shift all bits of B by one.
- Add 1,2,3,4,5,6,7, which load the 9-bit adder value of A, S into the A, X respectively.
- Add 8, which adds the value of the multiplicand if M is 0, otherwise subtracts the value of the multiplicand if M is 1.

The values of A, X are reset when LD_AX is pressed. When the run is pressed, the FSM transitions to an add state if the M is 1, otherwise shifts the entire sum of XAB to the right by one. This is done continuously for around 9 states of shift, and then the multiplication result is stored in the A, B registers as a 16 bit value.

Purpose: Describes an FSM used to implement multiplication.



Module: full_adder.sv

Inputs: a, b, cin

Outputs: s, cout

Description: This is a full adder that takes in a, b, cin as inputs, sums them together using combinational logic, and outputs the sum s and cout.

Purpose: This module is used to design a Full Adder which is instantiated to create a 9-bit carry ripple adder.

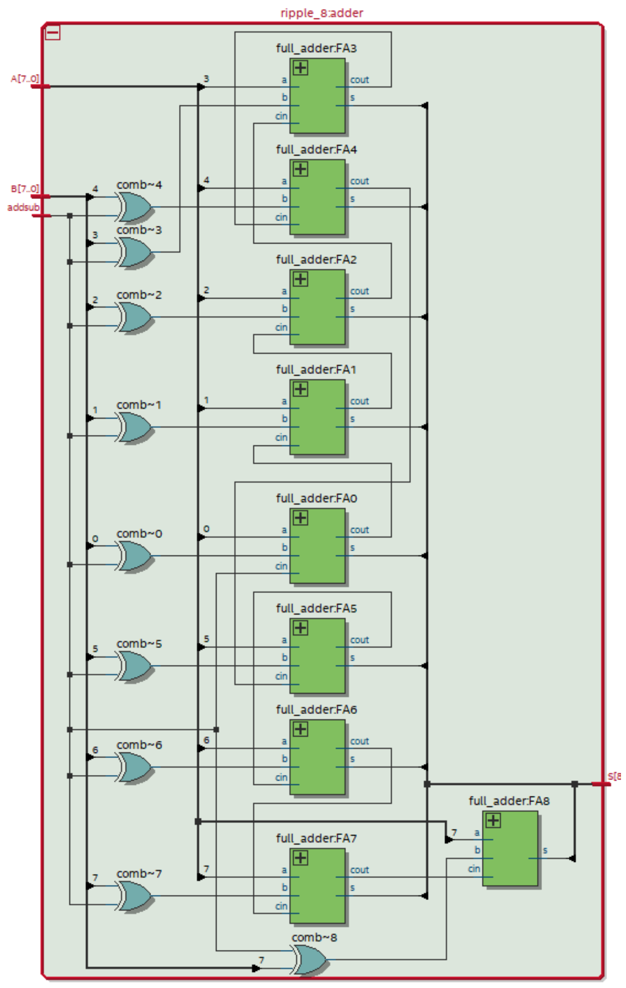
Module: adder9.sv (ripple_8)

Inputs: [7:0] A, B, addsub

Outputs: [8:0] S

Description: This is a carry ripple adder that takes in A, B and addsub as inputs, then either sums them together or subtracts B from A, depending on the addsub input. Its outputs are 9 bits so the carry out can be accounted for as an extra-bit of output.

Purpose: This module is used to implement the addition of the 8 bits of A, S, thereby resulting in the final value of X, A and B to implement multiplication.



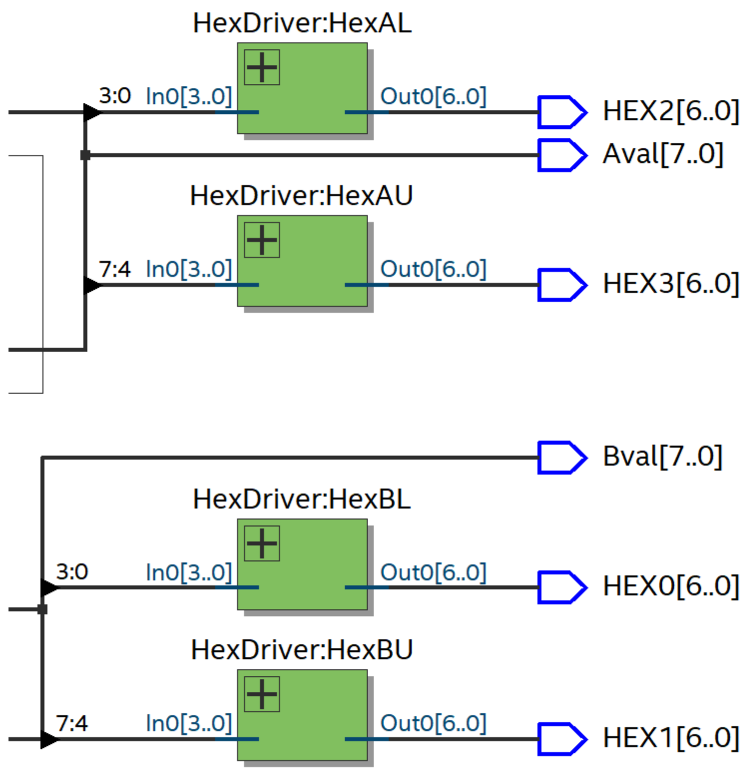
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This makes use of a unique case switch to set the 7 bits of Out0 to the corresponding 4 bit In0, to display the 4 bit hex number on the 7-segment LED.

Purpose: This module maps a hex number to its corresponding value so it can be displayed on a 7-segment LED.

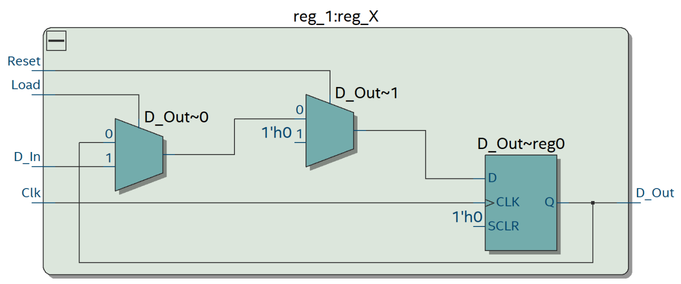


Module: reg_1.sv

Inputs: Clk, Reset, Load, D_In
Outputs: D_Out

Description: This is a positive-edged 1-bit parallel load register. When Reset is high, the register asynchronously resets and sets D_Out to 0. When Load is high, D_In loads into Data_Out.

Purpose: This module is used to create a 1 bit register to store the 1-bit X value in our multiplier.

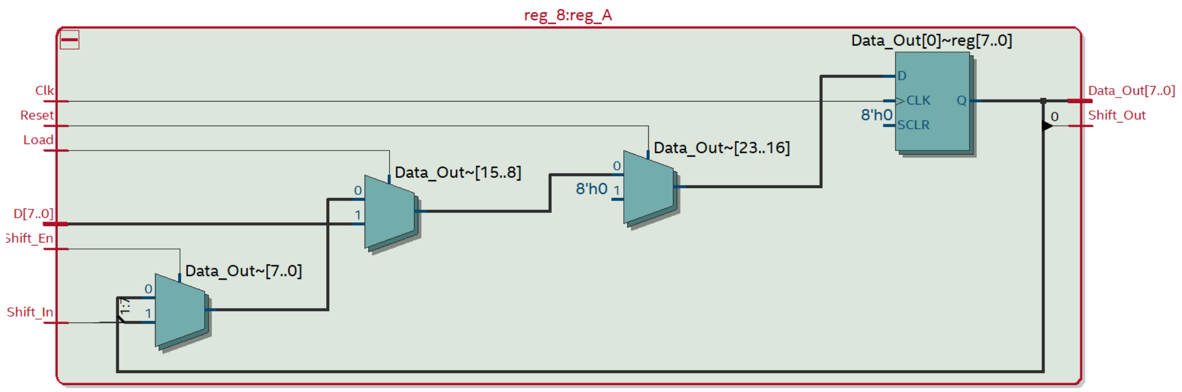


Module: reg_8.sv

Inputs: Clk, Reset, Load, [16:0] Shift_In, Shift_En, D
Outputs: [16:0] Data_Out, Shift_Out

Description: This is a positive-edged 8-bit parallel load register. When Reset is high, the register asynchronously resets and sets Data_Out to 0. When Load is high, D loads into Data_Out. If the Shift_En signal is high, then the Data_Out gets in the concatenation of the Shift_In bit, and the top 6 bits of Data_Out value from bits [7:1]. It also sets the value of Shift_Out to the LSB of the Data_Out (Data_Out[0]).

Purpose: This module is used to create an 8-bit register to either reset a value of a register, or to shift the bits one bit to the right when there is a shift signal inputted as high. This helps create the A, B registers for the multiplier we implement, with capabilities to shift to the right in the various shift states.



Module: Processor.sv

Inputs: Clk, Reset_Load_Clear, Run, [7:0] SW

Outputs: Xval, [7:0] Aval, Bval, [6:0] Hex 3,2,1,0.

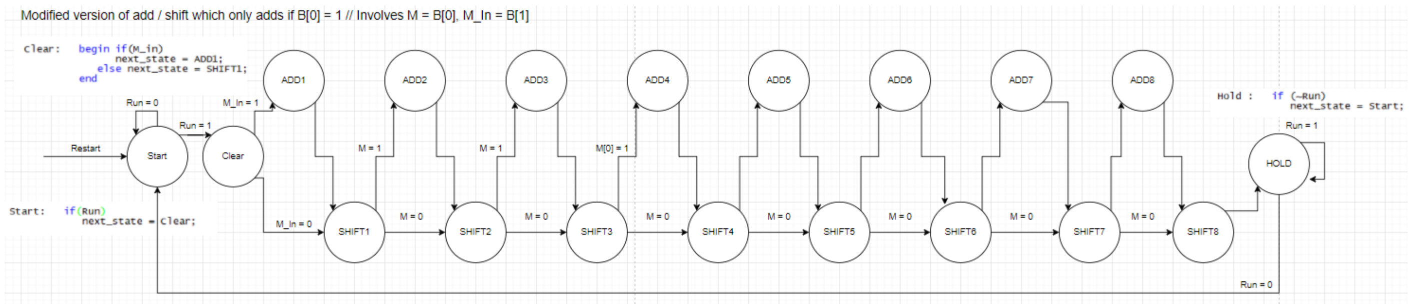
Description: This module is the top level module that instantiates all the modules in the program, helping execute the actual working multiplier. It implements the FSM in the control state, instantiates the registers A, B, X, SW, and creates the structure of the 9 bit ripple adder. This then uses the values we get from the registers to display on the hex drivers, so we can observe the result of the multiplication.

Purpose: This module helps us instantiate all modules and execute a function of the working multiplier, helping us actually create the FSM that we implemented and use the multiplier.

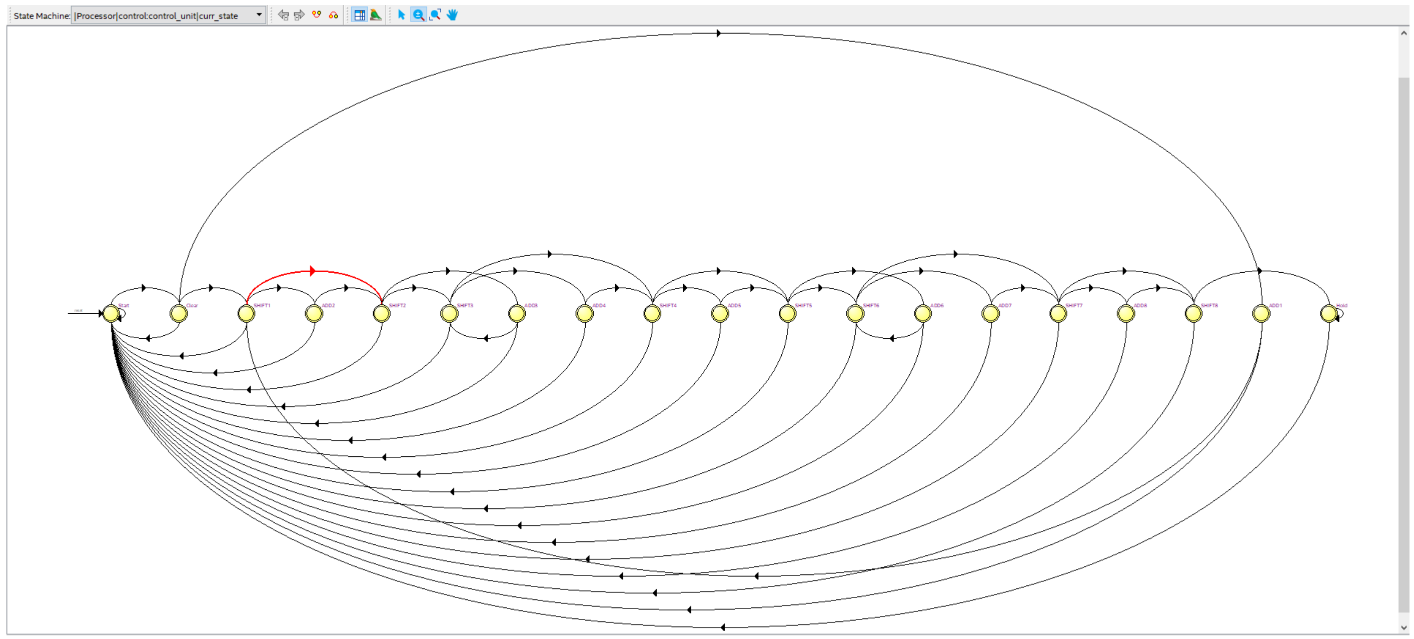
---

d. State Diagram for Control Unit:

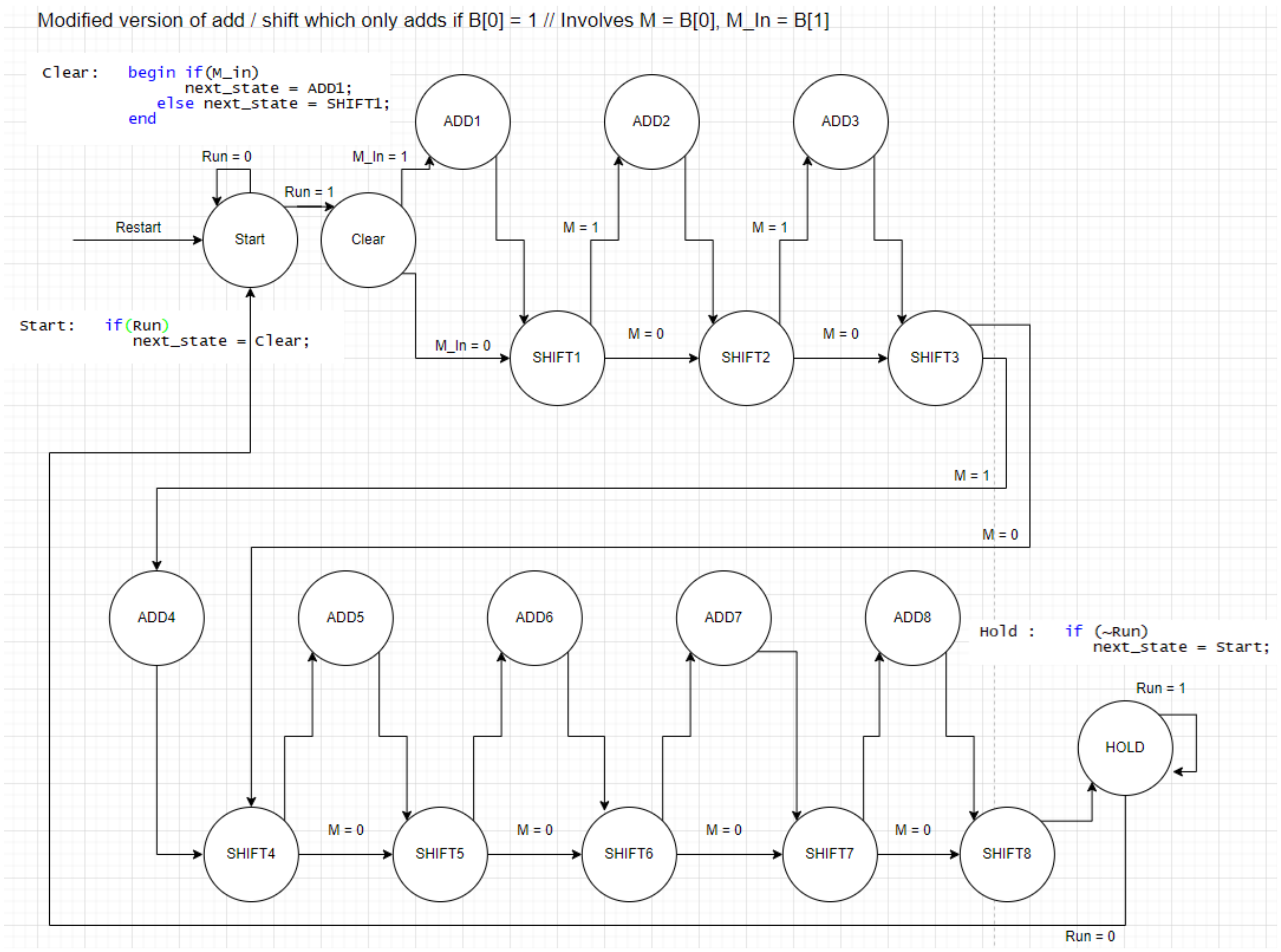
3 Diagrams are given below. The first one is the RTL State machine given by Quartus, the second is the one generated but with less clarity, the third one is the final state machine with a slightly different arrangement for clarity.


Modified version of add / shift which only adds if B[0] = 1 // Involves M = B[0], M_In = B[1]
Clear: begin if(M_in) next_state = ADD1; else next_state = SHIFT1; end
Start: if(Run) next_state = Clear;
Hold : if (~Run) next_state = Start;
Restart
Start
Clear
Run = 0
Run = 1
M_In = 1
M_In = 0
ADD1
ADD2
ADD3
ADD4
ADD5
ADD6
ADD7
ADD8
SHIFT1
SHIFT2
SHIFT3
SHIFT4
SHIFT5
SHIFT6
SHIFT7
SHIFT8
M = 1
M[0] = 1
M = 0
HOLD
Run = 1
Run = 0

State diagram: (Zoomed-in version given later).



RTL-Generated state machine.


Modified version of add / shift which only adds if B[0] = 1 // Involves M = B[0], M_In = B[1]

```
Clear:   begin if(M_in)
                    next_state = ADD1;
              else next_state = SHIFT1;
         end
```

```
Start:   if(Run)
            next_state = Clear;
```

```
Hold :   if (~Run)
            next_state = Start;
```

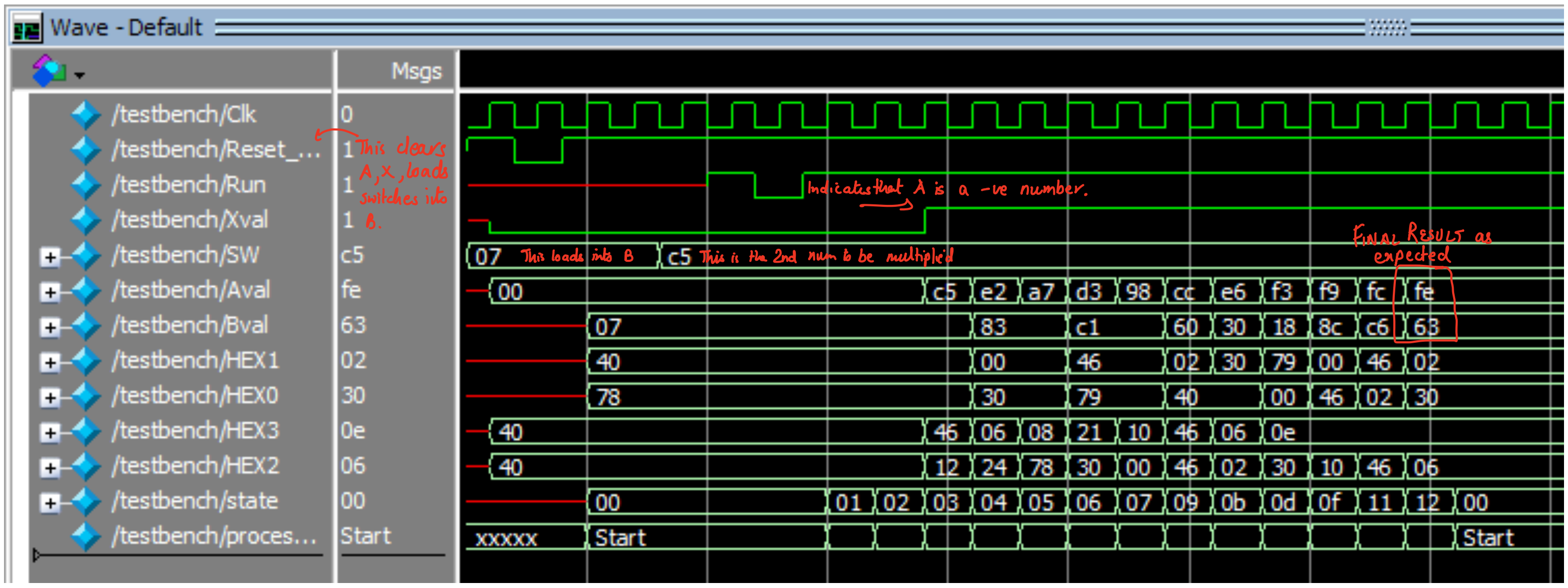
<u>State diagram made on diagrams.net.</u>

<u>Figures: The Finite state machine diagram for Control Unit, version 1 matches RTL Diagram, version 2 has more clarity.</u>

4. Annotated pre-lab simulation waveforms.

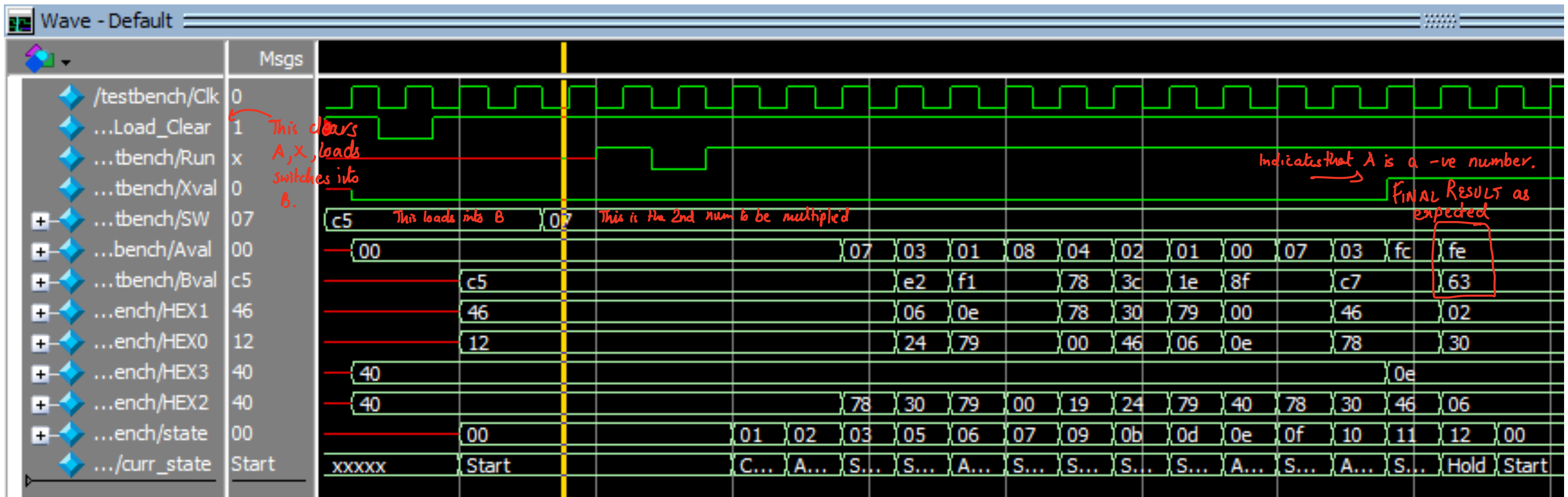
a. Must show 4 operations where operands have signs (++), (+*-), (-*+) and (-*-):*
b. Waveform must have notes that clearly show the operands as well as the result, etc.

```
//Test 1 (+-) : B = x07 (+7), S = xC5 (-59), Result = xFE63 (-413)

SW = 8'b00000111;
Reset_Load_Clear = 1;
#2 Reset_Load_Clear = 0;
#2 Reset_Load_Clear = 1;
#4 SW = 8'b11000101;
#2   Run = 1;
#2   Run = 0;
#2   Run = 1;
```



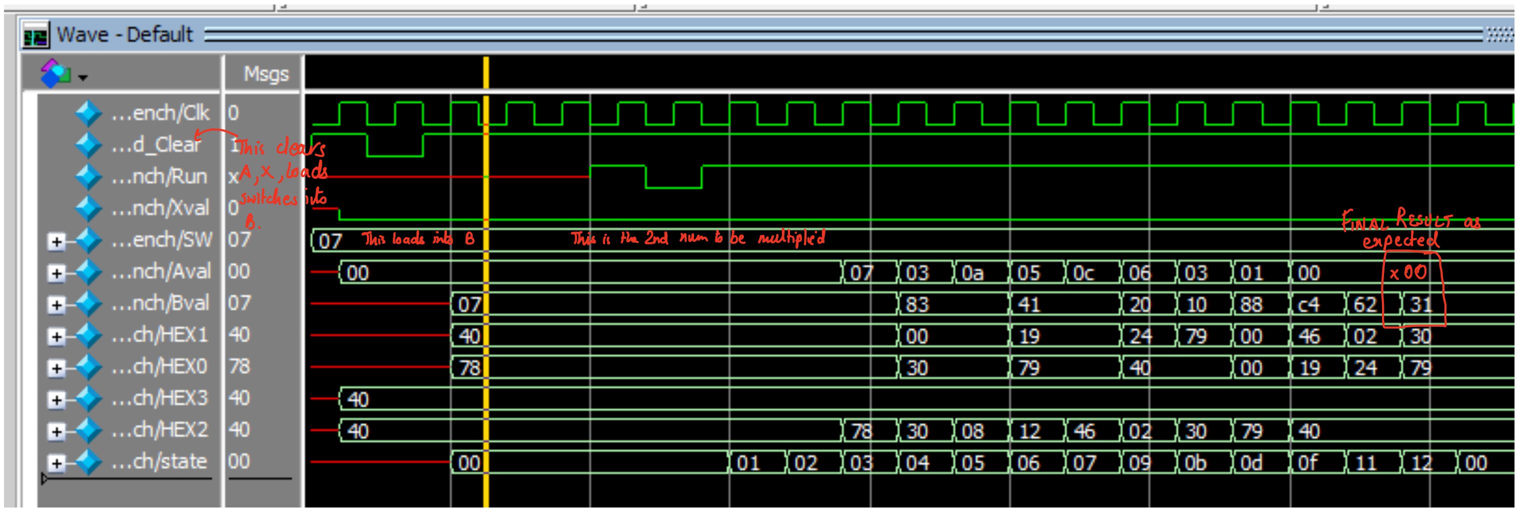
Wave - Default
Msgs
/testbench/Clk 0
/testbench/Reset_... 1
/testbench/Run 1
/testbench/Xval 1
/testbench/SW c5
/testbench/Aval fe
/testbench/Bval 63
/testbench/HEX1 02
/testbench/HEX0 30
/testbench/HEX3 0e
/testbench/HEX2 06
/testbench/state 00
/testbench/proces... Start
This clears A, X, loads switches into B.
Indicates that A is a -ve number.
07 This loads into B
c5 This is the 2nd num to be multiplied
Final Result as expected
SW: 07 c5
Aval: 00 c5 e2 a7 d3 98 cc e6 f3 f9 fc fe
Bval: 07 83 c1 60 30 18 8c c6 63
HEX1: 40 00 46 02 30 79 00 46 02
HEX0: 78 30 79 40 00 46 02 30
HEX3: 40 46 06 08 21 10 46 06 0e
HEX2: 40 12 24 78 30 00 46 02 30 10 46 06
state: 00 01 02 03 04 05 06 07 09 0b 0d 0f 11 12 00
process: xxxxx Start Start

```
// Test 2 (-+) : B = xC5 (-59), S = x07 (+7), Result = xFE63 (-413)

SW = 8'b11000101;
Reset_Load_Clear = 1;
#2 Reset_Load_Clear = 0;
#2 Reset_Load_Clear = 1;
#4 SW = 8'b00000111;
#2   Run = 1;
#2   Run = 0;
#2   Run = 1;
```

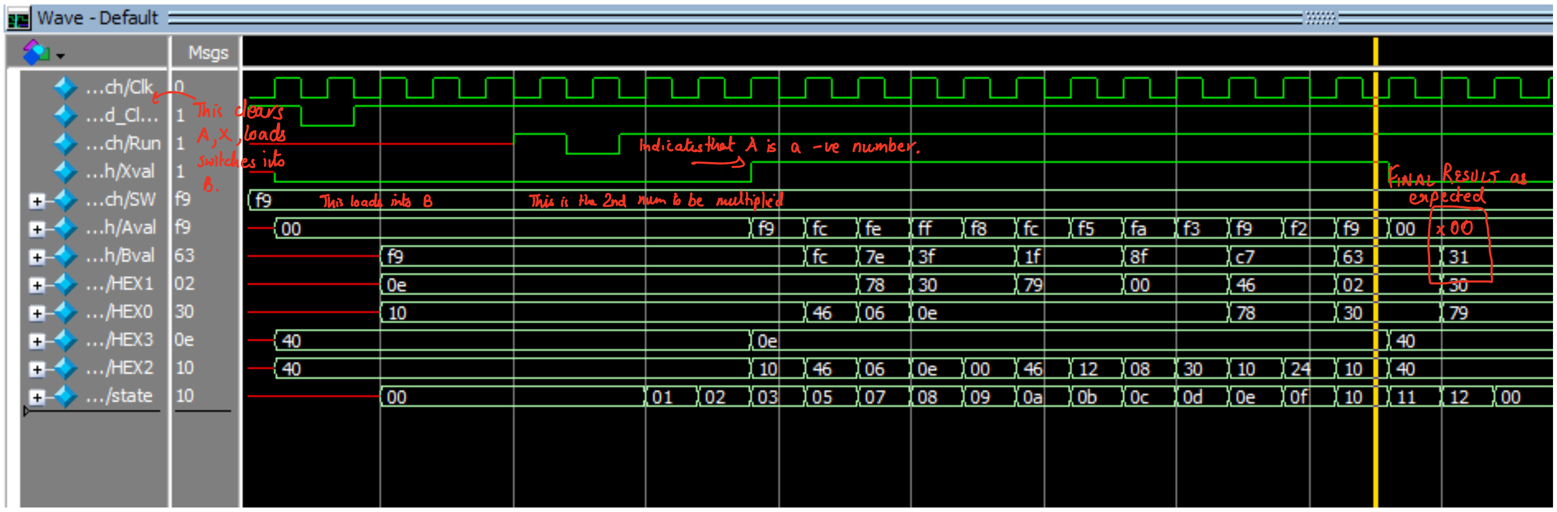


Wave - Default
Msgs
/testbench/Clk 0
...Load_Clear 1
...tbench/Run x
...tbench/Xval 0
...tbench/SW 07
...bench/Aval 00
...tbench/Bval c5
...ench/HEX1 46
...ench/HEX0 12
...ench/HEX3 40
...ench/HEX2 40
...ench/state 00
.../curr_state Start
This clears A, X, loads switches into B.
Indicates that A is a -ve number.
c5 This loads into B
07 This is the 2nd num to be multiplied
Final Result as expected
Aval: 00 07 03 01 08 04 02 01 00 07 03 fc fe
Bval: c5 e2 f1 78 3c 1e 8f c7 63
HEX1: 46 06 0e 78 30 79 00 46 02
HEX0: 12 24 79 00 46 06 0e 78 30
HEX3: 40 0e
HEX2: 40 78 30 79 00 19 24 79 40 78 30 46 06
state: 00 01 02 03 05 06 07 09 0b 0d 0e 0f 10 11 12 00
curr_state: xxxxx Start C... A... S... S... A... S... S... S... S... A... S... A... S... Hold Start

```
// Test 3 (++) : B = x07 (+7), S = x07 (+7), Result = x0031 (+49)

SW = 8'b00000111;
Reset_Load_Clear = 1;
#2 Reset_Load_Clear = 0;
#2 Reset_Load_Clear = 1;
#4 SW = 8'b00000111;
#2   Run = 1;
#2   Run = 0;
#2   Run = 1;
```



```
// Test 4 (++) : B = xF9 (-7), S = xF9 (-7), Result = x0031 (+49)

SW = 8'b11111001;
Reset_Load_Clear = 1;
#2 Reset_Load_Clear = 0;
#2 Reset_Load_Clear = 1;
#4 SW = 8'b11111001;
#2   Run = 1;
#2   Run = 0;
#2   Run = 1;
```



5. Post-Lab Questions:

1.) Refer to the Design Resources and Statistics in IQT and complete the following design statistics table.

LUT	94
DSP	0
Memory (BRAM)	0
FLip-Flop	46
Frequency	80.31 MHz
Static Power	89.97 mW
Dynamic Power	1.56 mW
Total Power	104.50 mW

In our structure, we used 19 total states for adding and then shifting at the worst case. We realized that we could optimize our circuit by avoiding some of the adding states (which add a zero when M = 0) and so we decided to implement that in our circuit.

Q1. What is the purpose of the X register? When does the X register get set/cleared?

- The X register is the sign extended bit value of register A. It becomes a value of 0 when the MSB of register A is 0 (A[7] = 0 -> X = 0). When the MSB of register A is 1, then the X register gets set to 1 (A[7] = 1 -> X = 1).

Q2. What would happen if you used the carry out of an 8-bit adder instead of output of a 9-bit adder for X?

- If we used the carry out bit of an 8-bit adder, then the values would not represent the sign extension of the A register, and would only consider the overflow of the addition. By using a 9 bit adder, we can ensure that the X register holds the actual sign extended bit of the A register.

Q3. What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?

- The limitation of continuous multiplication is that eventually it is very likely that we will each a situation where the result of the multiplication is larger than the value that can be obtained from just 16 bits. In those situations, there are one of two things that could happen, either the value when truncated can fit into the valu of 16 bits without loss of information (like -1 in Hex: 32’hFFFFFFFF is the same as 16’hFFFF), and so this would be an acceptable situation, or the second option is that when truncated, the values are not equal to each other (32’h0003FFFF is different from 16’hFFFF) and this would cause an erroneous result in our calculations.

Q4. What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

- Advantage of pen-and-paper method over multiplication algorithm: In this method, all the values to be added can be calculated in a singular moment / state, and then it would only take one state to add all the bits together to obtain the result. Hence, the actual calculation would only take 2 states and would be extremely time-optimized as the number of multiplication calculations increase. In addition, in the pen-and-paper method we have no concerns about overflow because there is no technical limitation on paper for us to write our answer.
- Advantages of the multiplication algorithm over the pen and paper method: In the pen-and-paper method, we would need a huge 16 bit adder, and also would need several 8-16 bit registers to store the temporary values that are calculated. This makes the implementation of our algorithm extremely well space-optimized, as it only needs a single 1-bit register, and 2(3 if we include the switch register) registers to get the same output.

---

6. Conclusion

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.
b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

In our design, we tried to implement the optimized version of the Adder, in which we skipped the add state entirely if the MSB of Register A (A[0]) = 1. In this situation we had to look ahead to perform an add (or subtract) only if the bit 1 value of Register A (A[1]) = 1. However, we did not handle the situation of the looking ahead correctly and so we had a slight error when we attempted to multiply 2 negative integers together. We then fixed this by altering some of the M values to a new variable called M_in which stores the 01st bit of the register A (M_in = A[1]).
I believe that the notes on using testbenches for modelsim simulations, and the employment of signaltap are not detailed enough for us to use them appropriately when necessary. The implementation of the packed arrays for the hex drivers was also confusing to understand, and required intense debugging during office hours.
