ECE 385
Fall 2021
Experiment #4

An 8-bit Multiplier in SystemVerilog

Michael Stoens (mstoens2)
Ralph Balita (rbalita2)
Lab Section: AB3

1) Introduction: Summary of the basic functionality of the multiplier circuit

The purpose of experiment #4 is to design and implement an 8-bit 2's complement multiplier. The multiplier uses an add-shift algorithm to multiply two numbers, similar to the "pen-and-pencil" method of multiplication. The 2's complement number's most significant bit indicates the sign of both the multiplier and the multiplicand. There will be a total of 8 shifts and up to 8 additions depending on whether the LSB of B, notated as "M" is a 1 or 0. Before the 8th and last shift, the machine will look at the LSB and determine if we need to subtract the multiplier to obtain our correct result. The final result of the multiplication is stored as a 16 bit sign extended value in the A and B registers. The user can then perform continuous multiplication or enter new values to begin a new computation.

2) Pre-Lab Question:

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

Multiplication Example of 11000101 * 00000111					
Function	X	A	B	M	Comments for the next step
ClearA, LoadB, Reset	0	00000000	*00000111*	1	Multiplicand is added to A in next step (INS)
ADD	0	11000101	*00000111*	1	00000000+11000101 = 11000101 -> A
SHIFT *after ADD	0	11100010	*10000011*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
ADD	0	10100111	*10000011*	1	11000101+11000101 = 10100111 -> A
SHIFT *after ADD	0	11010011	*11000001*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
ADD	0	10011000	*11000001*	1	11010011+11000101 = 10011000 -> A
SHIFT *after ADD	0	11001100	*01100000*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	11100110	*00110000*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	11110011	*00011000*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	11111001	*10001100*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
SHIFT *M=0	0	11111100	*11000110*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
SHIFT *M=0	0	11111110	*11100011*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
SHIFT *M=0	0	11111111	*01110001*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A

		0	1		
Result		11111110	01100011		Result of operation = 16’b SEXT in A and B

Note: The result of the multiplication operation is a sign extended value held in both A and B

Multiplication Example of 00000111 * 11000101					
Function	**X**	**A**	**B**	**M**	**Comments for the next step**
ClearA, LoadB, Reset	0	00000000	*11000101*	1	Multiplicand is added to A in next step (INS)
ADD	0	00000111	*11000101*	1	00000000 + 00000111 = 00000111
SHIFT *after ADD	0	00000011	*11100010*	0	Right shift 1 from A. Left Shift 1 to B. SEXT A
SHIFT *M=0	0	00000001	*11110001*	1	Right shift 1 from A. Left Shift 1 to B. SEXT A
ADD	0	00001000	*11110001*	1	00000001 + 00000111 = 00001000
SHIFT *after ADD	0	00000100	*01111000*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	00000010	*00111100*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	00000001	*00011110*	0	Right shift 0 from A. Left Shift 0 to B. SEXT A
SHIFT *M=0	0	00000000	*10001111*	1	Right shift 1 from A. Left Shift 1 to B. SEXT A
ADD	0	00000111	*10001111*	1	00000000 + 00000111 = 00000111
SHIFT *after ADD	0	00000011	*11000111*	1	Right shift 1 from A. Left Shift 1 to B. SEXT A
SUB	1	11111100	*11000111*	1	Right shift 1 from A. Left Shift 1 to B. SEXT A
SHIFT *after SUB	1	11111110	*01100011*	1	Right shift 0 from A. Left Shift 0 to B. SEXT A
RESULT		11111110	01100011		Result of operation = 16'b SEXT in A and B

Note: The result of the multiplication operation is a sign extended value held in both A and B

Notice that the results of both 00000111 * 11000101 and 11000101 * 00000111 are the same

3) Written description and Diagrams of of the multiplier circuit

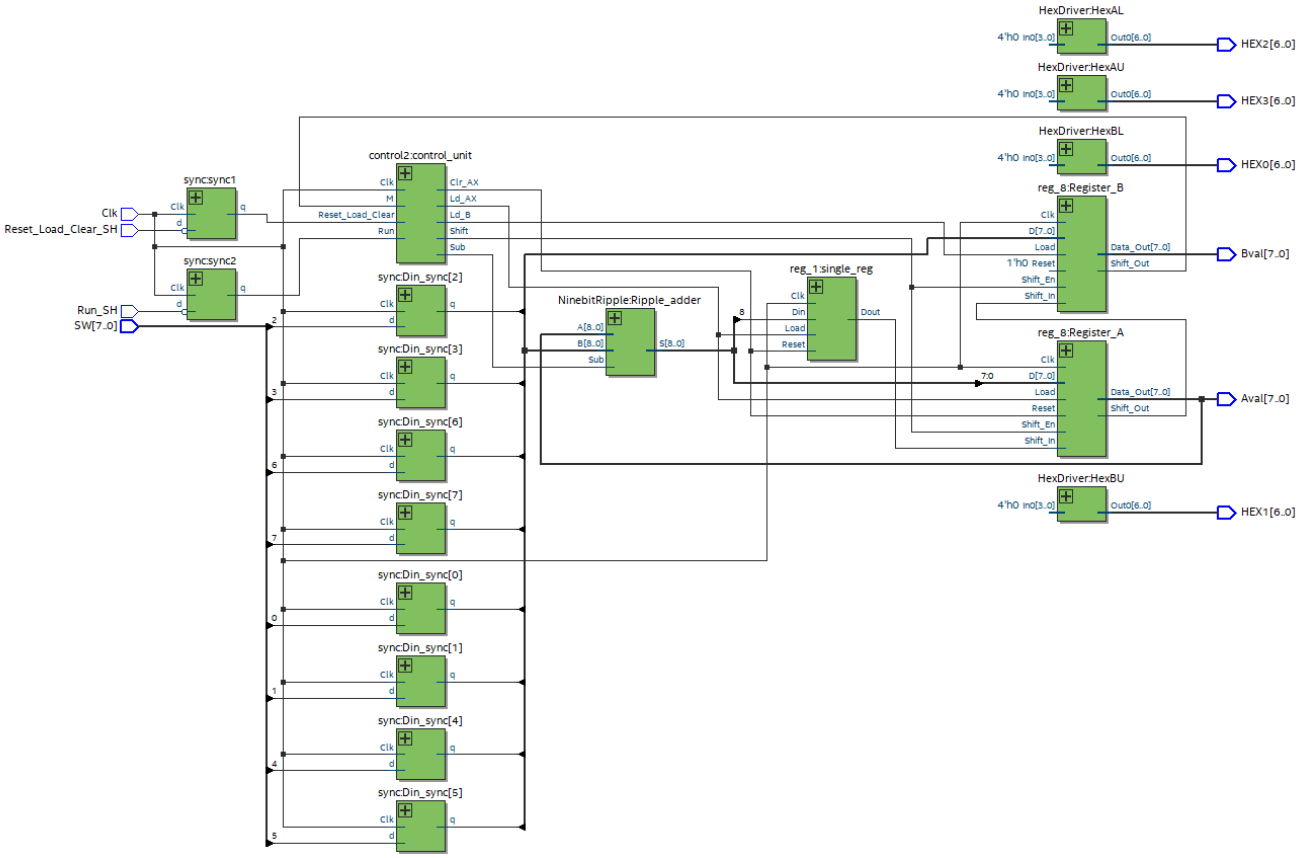
Summary of operation:

Loading Operands. The multiplier uses an add-shift algorithm to multiply two numbers, similar to the "pen-and-pencil" method of multiplication. To implement this algorithm, we began with creating a finite state machine to implement our control signals. Our finite state machine begins in the "Start" state. Within the "Start" state, registers A and X are cleared, and we are able to load values to register B using the switches of the DE-10. Once we have set the value of register B, which will be used as our multiplier, we press the Reset_Load_Clear button to advance to the "Load" state. Within the load state, we once again set the value of the switches to our desired value, this time for our multiplicand. Once we have entered the two values we want to multiply, we press the "Run" button to begin the multiplication.

Computing Result. The finite state machine then progresses through a sequence of thirteen alternating add and shift states. The first state in the sequence is Add1(A1). In this state and all other "Add" states, the value of the switches (SW) is added to the value of register A if the least significant bit of register A, notated as M, is a binary 1. If M is a binary 0, the FSM proceeds directly to the next state without loading the value from the adder into register A. After each "Add" state is a "Shift" state, where each of the registers (A, B and X) performs a right shift. The state machine then proceeds to the next "Add" state and repeats until the seventh "Shift" state is reached. Following the seventh shift, the state machine will proceed to the "Subtract" if M is a binary 1 (Since register B has been shifted 7 times the value of the LSB was the original MSB, meaning a binary 1 would indicate that the value was negative). Following the optional subtraction, the state machine proceeds through the eighth and final shift to complete the cycle.

Storing Result. After the sequence of Shift and Add states have finished, the state machine enters the Hold state and the result of the operation is then stored in both A and B as a 16-bit signed extended value. From the Hold State, the user can then perform continuous multiplication by pressing the Run button, or return to the State state to enter new values by pressing the Reset_Load_Clear button.

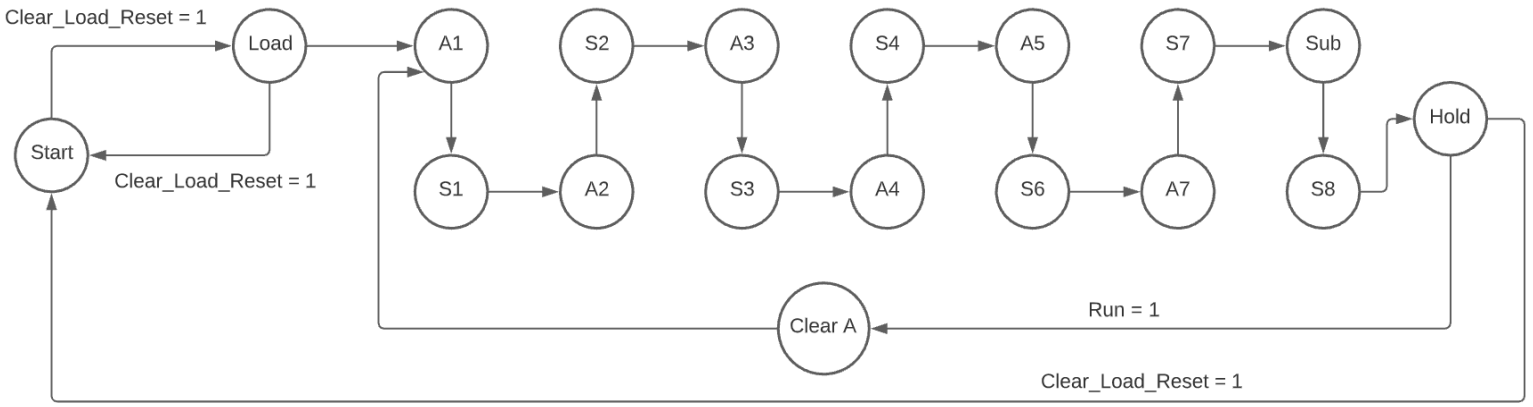
Top Level Block Diagram:



List all modules used

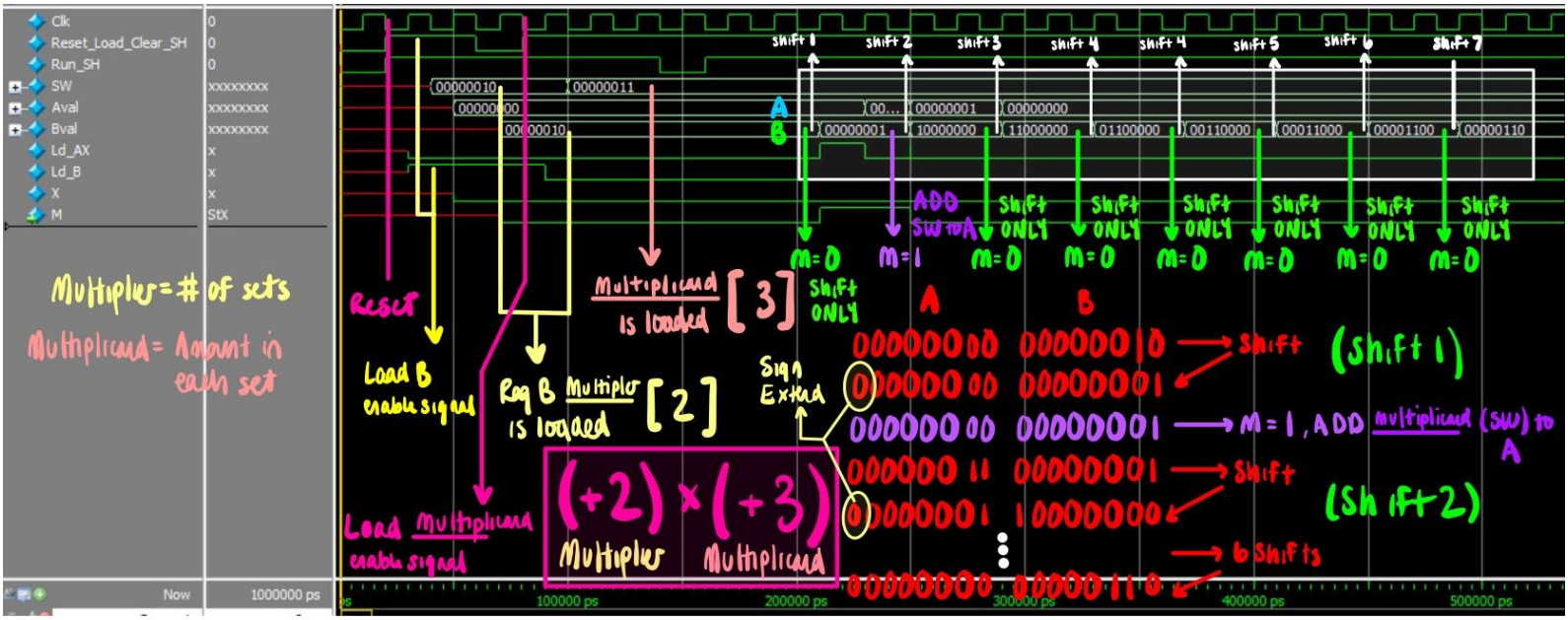
Written description of all .SV modules				
Module	**Inputs**	**Outputs**	**Description**	**Purpose**
Syncronizers.sv	Clk, d	q	Synchronizes asynchronous input signals to the Clk signal	Synchronizes the pushbutton signals to the Clk for use with the FPGA
control2.sv	Clk, Reset_Load_Clear M, Run,	Clr_AX, LD_AX, Ld_B, Shift, Sub	This module implements a twenty state finite state machine to control the multiplier	Controls the next state logic and signals for adding, shifting and loading the registers.
HexDriver.sv	[3:0] In0	[6:0] Out0	This is used to control the HEX display. Each Input has a hard coded output to display a number on the HEX Display	Translates binary values to signals needed to display those values on the HEX displays of the DE-10
reg_8.sv	Clk, Reset, Load, Shift_In, Shift_En, [7:0]D	Shift_Out [7:0]Data_Out	8 bit shift register. Automatically resets when Reset = 1. Loads data when Load = 1. Executes a right shift when Shift_EN = 1.	Used to implement shift registers A and B.
reg_1.sv	Clk, Reset, Load, Din	Dout	1 bit registers. Automatically resets when Reset = 1. Loads data when Load = 1.	Used to implement register X
Processor.sv	Clk, Reset_Load_Clear_SH Run_SH, [7:0] SW	[7:0] Aval,Bval [6:0] HEX1, HEX2, HEX3, HEX4	Top level module for the 8 bit multiplier.	Implements all of the other modules and connects all signals.
9bitRipple.sv	[8:0] A, B. Sub	[8:0] S. Cout	9 bit bit ripple adder. Performs addition of two 9 bit binary values.	Used to add SW and the contents of register A in order to multiply. Register A and SW are sign-extended to 9 bits before addition.
testbench.sv	None	None	Testbench used to set values and implement control signals for simulation.	Used to debug and verify if the program is functioning correctly.

State Diagram for the Control Unit:

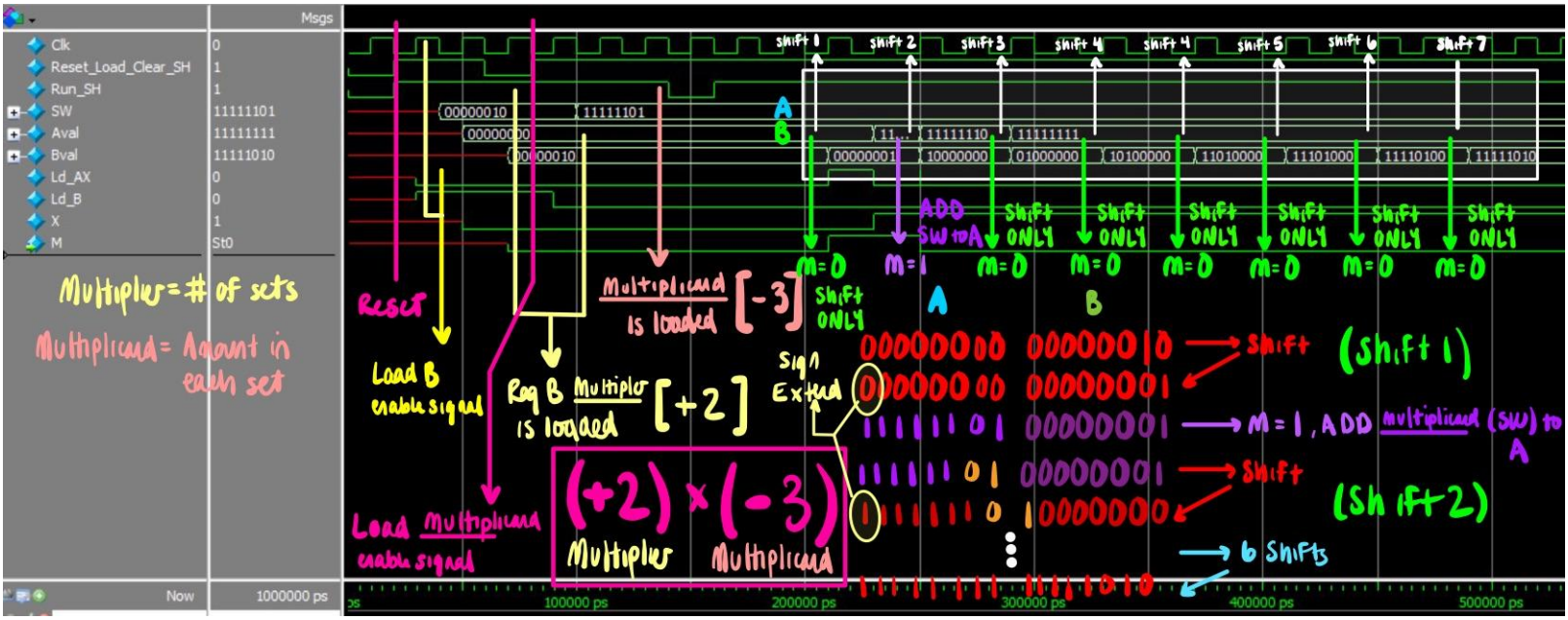


4) Annotated pre-lab simulation waveforms

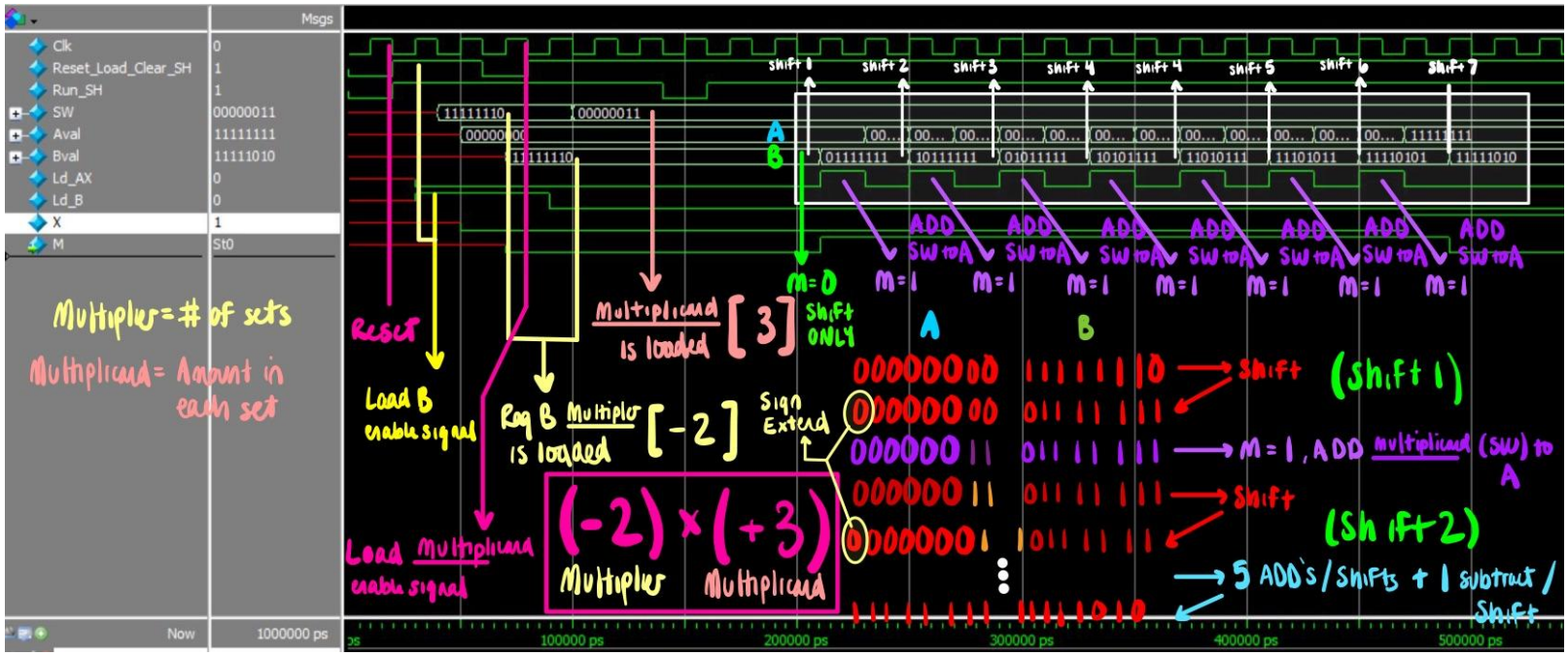
positive × positive



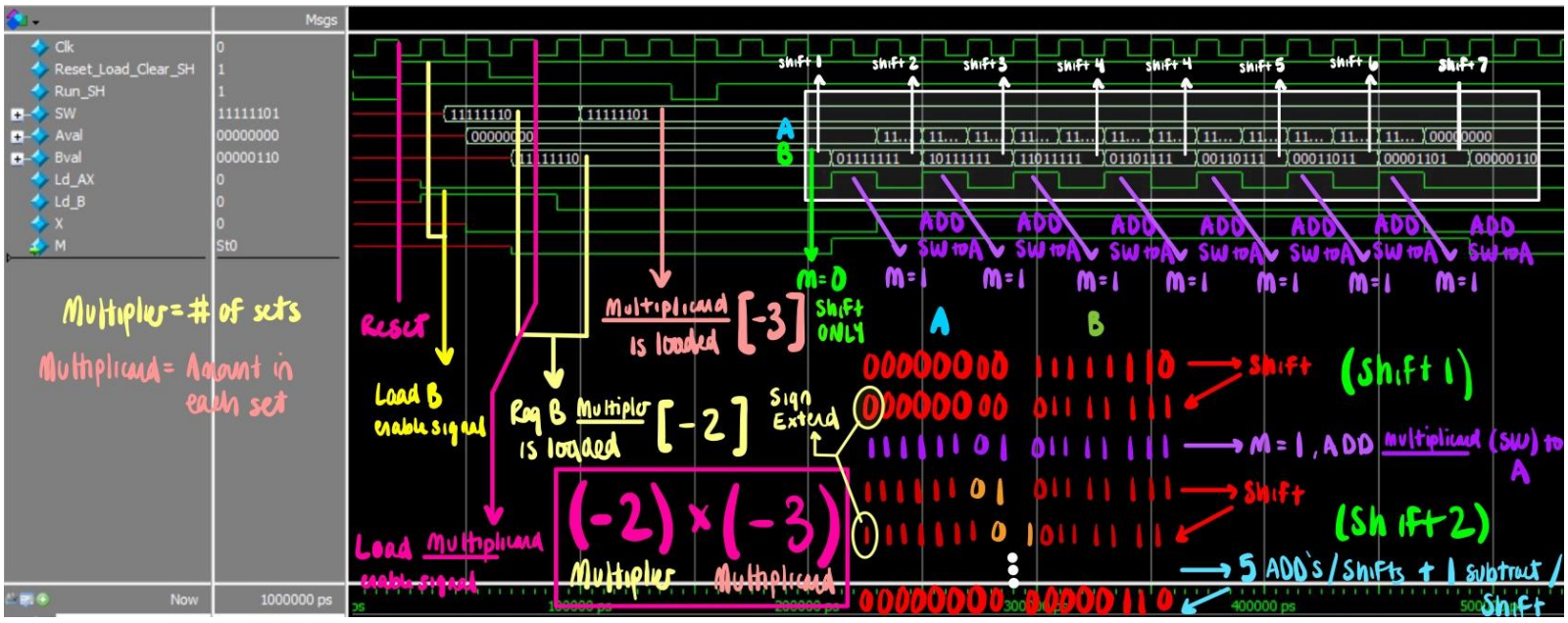
positive × negative



negative × positive



negative × negative



5) Answers to post-lab questions

Design Statistics	
LUT (look up tables)	529
DSP (Digital signal processors	0
Memory(BRAM)	1,088
Flip Flop	595
Frequency	109.0MHz
Static Power	63.47mW
Dynamic Power	1.21mW
Total Power	81.43mW

The component of our multiplier that has the most room for optimization is our control unit. When designing our finite state machine, we discussed several different ideas of how we could minimize the number of states used. We began trying to design a state machine that would implement single Add and Shift states instead of separate states for each of the eight shifts/adds, but ultimately abandoned the design due to the complexity involved with keeping track of how many times we had shifted/added. Another design for our control unit implemented optionally transitioning to each add state based on the M signal instead of always transitioning to the Add state and generating the Load A signal based on M. We fully implemented this design but could not get it to function properly, so we proceeded to redesign it to always transition through the Add state. We believe this implementation would be more efficient since we would not need to transition through every state on each execution of the multiplier

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

The purpose of the X register is to hold the sign bit of the result of the sum of register A and the Switches. The sign bit is held in the X register and then shifted into register A. This ensures that during each shift state register A is always shifting in the correct value. The X register is cleared at the same time as the A register, which occurs at the beginning of each multiplication.

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

The reason the input of the X register must be the 9th bit instead of the carry out of an 8 bit adder is because the carry out bit is not always equal to the sign bit of the resultant sum. By sign extending the 8 bit inputs to 9 bits before adding them, the most significant bit of the 9 bit sum will always hold the correct sign bit. We can then use this value to shift into the A register during the shift states. If we were to use the carry out bit of an 8 bit register, we would not always shift the correct value into the A register, and therefore would not get correct results for our multiplication.

What are the limitations of continuous multiplications? Under what circumstances, Will the implemented algorithm fail?

Continuous multiplications are limited to the number of bits that are available to hold the result. In this lab, we take two 8-bit 2's complement numbers, and shift 8 times to output a 16-bit sign extended value. With each multiplication, we have to clear registers A and X before continuing to the next multiplication. This means that any value of the previous result that is stored in register A will be lost. Since the data held in registers A and X is erased with each multiplication, our continuous multiplication is limited to values whose product is less than 8 bits so they are only stored in register B.

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

One of the main differences between the add-shift algorithm and the pencil and paper method is that in the add-shift method we shift the product to the right and add it to the sum with each step instead of shifting to the left and computing one large sum at the end. By shifting the product to the right and adding with each step, this provides the advantage of only requiring an 8 bit adder to compute the sum instead of the 16 bit adder that would be required for the pencil and paper method. The downside of the add-shift method is the added complexity required to implement option addition (based on the least significant bit) before each shift state. An advantage of the pencil and paper method is it is easy to understand the implementation since it is such a familiar process.

6) Conclusion

Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

The design of the lab was straightforward and understandable, but hard to implement at the start. Since we were given a blank slate, we understood the mechanics of each module but did not know how to best indicate if we were making progress. Our group did not understand how to efficiently use ModelSim to debug our modules. We spent many hours troubleshooting our design but ultimately could not get it to function properly. After the deadline of the experiment demo, we learned how to utilize modelsim to trace down errors in our code. We found that we had an error in the sign extension of registers A and the Switches before they were sent to the adder. This resulted in “Dont care” values being shifted through our registers. After fixing our error in the sign extension, our multiplier functioned without issue.

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.

Debugging SystemVerilog code, when starting from a blank template, is challenging. Debugging SystemVerilog Code, prior to this lab, meant that we would take a variable that we thought was a problem and map it into the Top-Level Entity as an output. We would then compile the code and run ModelSim to check its waveform. Overtime, the code got messy and cluttered, and it was difficult to see what our original code would look like. In a way, our way of debugging SV code made it easy for us to lose track of where pins were connected for the Processor Unit. What we realized after the lab was that we did not know ModelSim’s fullest debugging capabilities. After using a lot of trial and error, we realized that ModelSim is the perfect tool to trace bugs in our code.

One important thing to consider for next semester is to give a guide as to how to use ModelSim, particularly adding waveforms for variables (instead of adding them as outputs for our Top-Level) and also using the ModelSim terminal to compile, restart, and run our new code (instead of waiting tirelessly for our code to compile on Quartus before opening ModelSim).