ECE 385

Spring 2023

Experiment 4

An 8-Bit Multiplier in System Verilog

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

This lab intends to be a project of designing an 8-bit multiplier on FPGA board using System Verilog without any provided modules. The 8-bit multiplier will first load the 8-bit multiplicand to register B and by looking at the least significant bit (M) of B, the state machine will then determine whether to add the multiplier values, which are on the switches, to register A or shift both registers A and B. After the first 7th shift, the value on the switch will then be subtracted from register A if M is 1. Not to mention, during the subtraction and subtraction, both operands will be sign-extended to 9-bit, and the most significant bit will be stored in register X. Then, all the register will be shifted for the last time, and the result on register A and B will be the final 16-bit result.

Prelab Problem

Compute 11000101 * 00000111 in a table:

Function	X	A	B	M	Comments for the next step
Clear A, Load B, Reset	0	00000000	00000111	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	11000101	00000111	1	Shift XAB by one bit after ADD complete
SHIFT	1	11100010	10000011	1	Since M = 1, multiplicand (available from switches S) will be added to A
ADD	1	10100111	10000011	1	Shift XAB by one bit after ADD complete
SHIFT	1	11010011	11000001	1	Since M = 1, multiplicand (available from switches S) will be added to A
ADD	1	10011000	11000001	1	Shift XAB by one bit after ADD complete
SHIFT	1	11001100	01100000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11100110	00110000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11110011	00011000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111001	10001100	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111100	11000110	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	11111110	01100011	0	8th shift done. Stop. 16-bit Product in AB.

Operation of the 8-Bit Multiplier

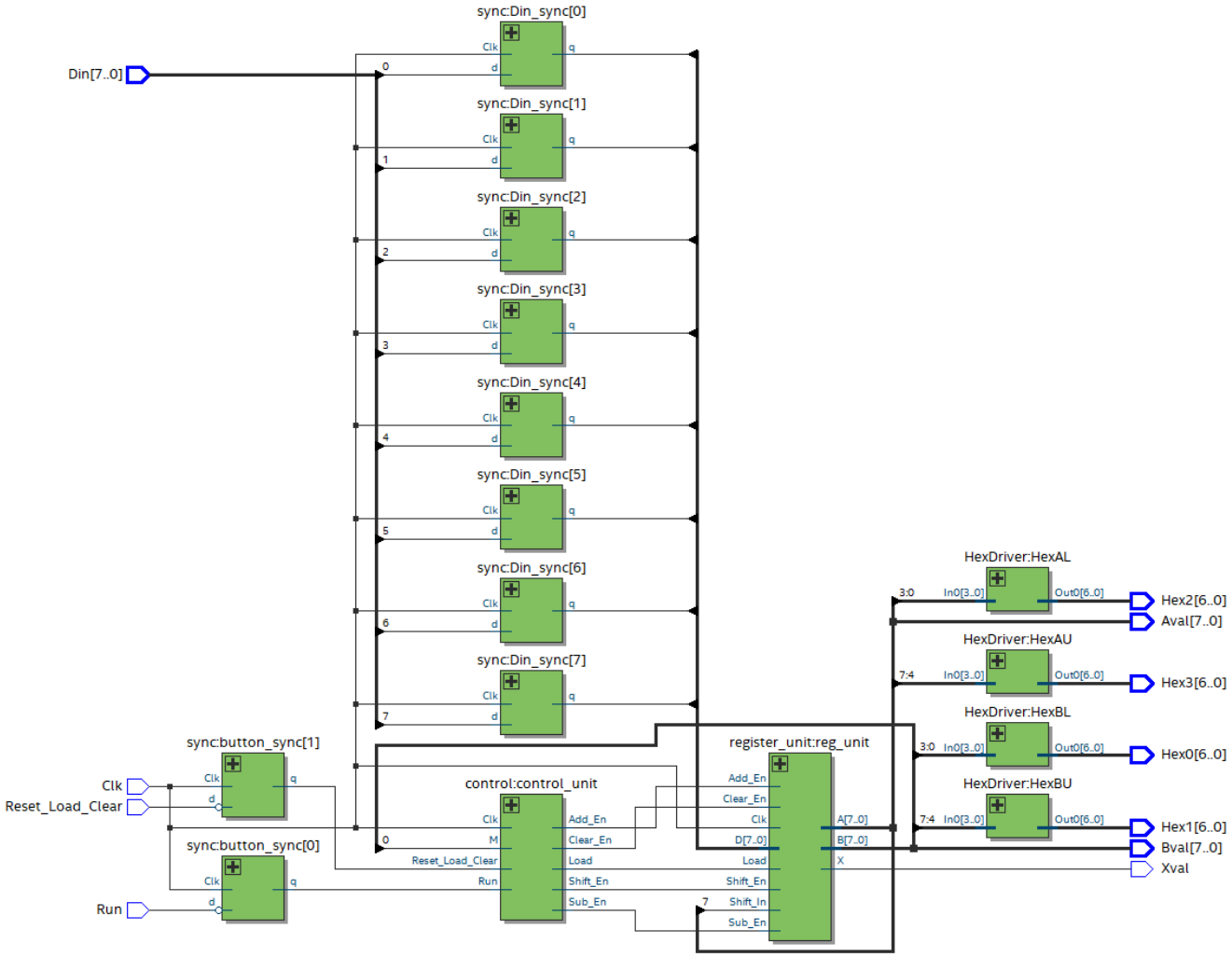


Figure I. 8-Bit Multiplier Top-Level Block Diagram

In order to conduct a full 8-bit multiplier operation, the multiplier first needs to be loaded into register B by utilizing the switches to set the multiplier value and pressing the reset_load_clear button on our FPGA, or simulating the act in our testbench. The pressing of the reset_load_clear button also simultaneously clears the contents of the X and A registers. Once the multiplier is loaded into register B with the X and A registers clear, we can then set the switch to represent the value of the multiplicand and press the run button. This will trigger one complete cycle/operation. During this operation, we may perform up to three types of operations on the bits in registers X, A, and B: shift, add, and subtract. In total, one complete 8-bit multiplier operation requires 8 single bit right shifts of registers X, A, and B in the order XAB. In other words, the bit stored in register X is shifted into the most significant bit of register A, and the least significant bit of register A is shifted into the most significant bit of register B, with the

least significant bit of register B shifted out and discarded. The add operation signals for the sign-extended 9-bit multiplicand/S on the switch to be added to the sign-extended 9-bit contents of register A, while the subtract operation signals for the sign-extended 9-bit multiplicand/S on the switch to be subtracted from the sign-extended 9-bit contents of register A, with the result stored in XA. The signal that determines whether a shift, add, or subtract operation should be conducted is the least significant bit of the multiplier/register B, which is denoted M. When M is a 1, the add operation is conducted, followed by a shift operation. Otherwise, if M is a 0, the shift operation is conducted without the add operation. The only possible scenario where we might conduct the subtract operation is after the 7th shift operation. If M is 1 after the 7th shift operation, then we will conduct a subtract operation instead of the normal add operation, followed by the final 8th shift operation, completing the computation. The 16-bit product will be stored in AB, or the registers A and B.

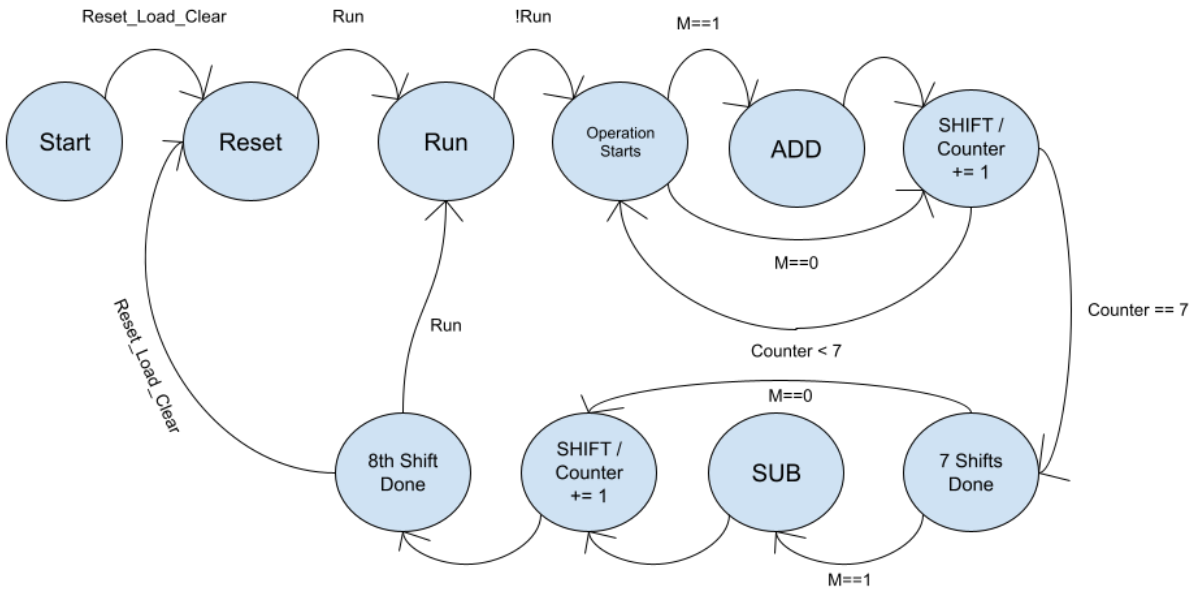


Figure II. Control Unit State Diagram

If we observe the control unit state diagram above, we see that from the start state, pressing the reset_load_clear button enters the reset state, where we clear the contents of registers X and A, and load the contents on the input switches into register B. Hitting the run button from the reset state transitions us into the run state, releasing the button then signals for the operation to begin. This additional state prevents us from computing several cycles of the operation from a single button press. Once the operation starts, we either enter the add or shift/counter++ state depending on the M signal as described in the previous section. We will loop back to check the least significant bit of register A as long as counter value is less than 7, indicating that less than 7 right shifts have been performed. Once 7 right shifts have been performed, then we will either conduct a subtract operation if M = 1, or jump straight to the last shift operation. Once the 8th shift operation is completed, we will enter a halt state where we can either go back to the reset state through the press of the reset_load_clear button, or enter another computation cycle with the new multiplier being the contents of register B, and the contents of registers X and A cleared.

SV Modules Summary

Module: reg_8.sv
Inputs: [7:0] D, Clk, Clear_En, Shift_In, Load, Shift_En, Add_En, Sub_En
Outputs: [7:0] Data_out, Shift_Out, X
Description: This is a positive-edge triggered 8-bit register with a synchronous clear and synchronous load. When the Load is high, data is loaded from Din into the register on the positive edge of Clk. Otherwise, if the Clear_En is high, the register is set with a value of 0. Not to mention, if Shift_En is high, the register will perform a right shift with Shift_In as the most significant bit and Shift_Out as the previous least significant bit. Last, the Add_En and Sub_En will determine whether to perform an addition or subtraction with the value in the register and D.
Purpose: This module is used to create the registers that store multiplication results in A and B and to make the register shift or do operation when necessary.

Module: register_unit.sv
Inputs: [7:0] D, Clk, Clear_En, Shift_In, Load, Shift_En, Add_En, Sub_En
Outputs: [7:0] A, [7:0] B, X
Description: This is a register unit where registers A and B are created and loaded with all the input signals Clear_En, Shift_In, Shift_En, Add_En, and Sub_En in order to perform the corresponding operation to the registers and store the sign bit X.
Purpose: This module is used as the register unit to create registers A and B and store X.

Module: control.sv
Inputs: Clk, Reset_Load_Clear, Run, M
Outputs: Shift_En, Add_En, Sub_En, Clear_En, Load
Description: This is the positive-edge triggered flip-flop that has 3 states, which include a start state, a waiting state, operation states, and a halt state. When the Reset_Load_Clear signal is high, the flip-flop will return to the start state in which only the Clear_En and Load signal is high. Otherwise, if the Run signal is high during the start state, the flip-flop will then proceed to the waiting state which has low output signals, and until the Run signal is low again, it will proceed to the operation state. During the operation states, the input M signal will be used to determine whether to do addition and then shift or only shift. If M is 1, then it will proceed to a state which has a high Add_En signal and go to the shifting state which has a high Shift_En signal afterward. On the other hand, if M is 0, it will only proceed to the shifting state. The process will continue until 7 shiftings are done. Then, if M is 1, the state machine will then proceed to the state that has a high Sub_En signal and then proceeds to the shifting state. Otherwise, it will only go to the shifting state. After all 8 shiftings are done, then it will proceed to the halt state. In the halt state, the output signal is low, and it will only return to the start state when the Reset_Load_Clear signal is high. Or if the Run signal is back to high again, the state machine will do another multiplication by returning to the operation states.
Purpose: This module is used as a simple state machine to make sure the Run switch input will only run the multiplication once until the Run switch is back to low. Also, it is used to tell the system what operations to do by observing the input signal M.

Module: full_adder.sv
Inputs: A, B, cin

Outputs: s, cout
Description: This is a full adder module that takes in the inputs A, B, and cin and then produce the output s, which is A^B^cin, and output cout, which is (A&B)|(A&cin)|(B&cin).
Purpose: This module is used as the routing unit to determine which input values have to be stored in the shift in bits(A_In and B_In) of A and B.

Module: adder9.sv
Inputs: [7:0] A, [7:0] S
Outputs: [7:0] A_new, X
Description: This is a 9-bit full adder module that takes in 8-bit inputs A and S and has a series of 9 full adders with 0 connecting to the carry-in bit of the first full adder and the carry-out bit of the first full adder connecting to the carry-in of the second full adder and so on to produce a 9-bit output, which has the most significant bit as X and rest of 8-bit values as A_new. More importantly, the 8-bit inputs A and S are sign extended to 9-bit in order to put in 9 full adders.
Purpose: This module is used to combine 9 full adders to create a 9-bit full adder and to sign extend the 8-bit input during the operation.

Module: subtracter9.sv
Inputs: [7:0] A, [7:0] S
Outputs: [7:0] A_new, X
Description: This is a 9-bit full subtracter module that takes in 8-bit inputs A and S and has a series of 9 full adders with 1 connecting to the carry-in bit of the first full adder and the carry-out bit of the first full adder connecting to the carry-in of the second full adder and so on to produce a 9-bit output, which has the most significant bit as X and rest of 8-bit values as A_new. More importantly, the 8-bit inputs A and S are sign extended to 9-bit in order to put in 9 full adders and S is negated in order to do the subtraction.
Purpose: This module is used to combine 9 full adders to create a 9-bit subtracter with S being negated and to sign extend the 8-bit input during the operation.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the Hex driver that takes in 4-bit value In0 from the register in order to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.
Purpose: This module is used to show the value in the adder in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

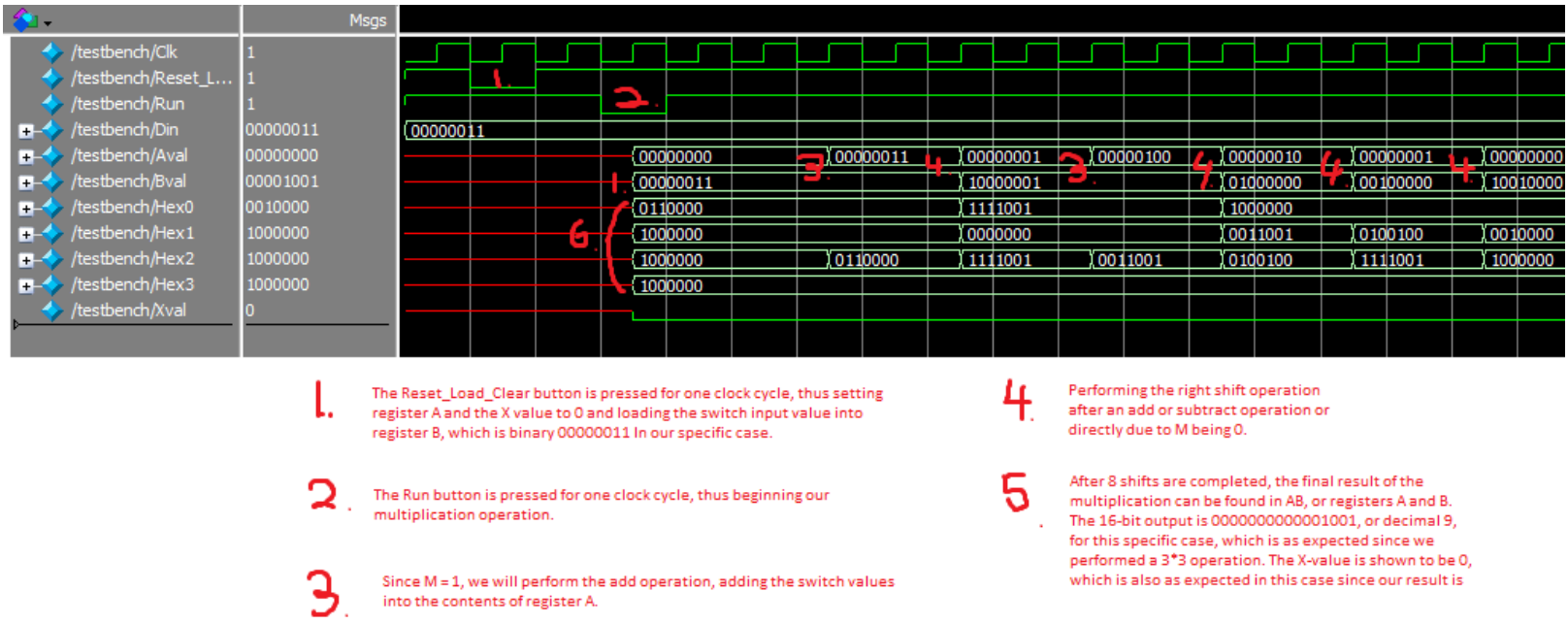
Module: Synchronizers.sv
Inputs: clk, Reset, d
Outputs: q
Description: These are the synchronizers that use positive-edge-triggered flip-flops to set the output value q to the input value d at every rising edge of the clock. During the rising edge of Reset signal, the synchronizer with reset to 0 will turn q to 0 while the synchronizer with reset to 1 will turn q to 1.

Purpose: This module is used as the synchronizer for bringing asynchronous signals into the FPGA board.

Module: multiplier_top_level.sv
Inputs: Clk, Reset_Load_Clear, Run, [7:0] Din
Outputs: [7:0] Aval, [7:0] Bval, Xval, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3
Description: This is the entire processor where each input values are used to initiate all the modules that are created and described above and organize all the modules into a single 8-bit multiplier with the 8-bit input Din being loaded as multiplicand into register B and multiplier and then perform a multiplication to produce a 17-bit output, which is the combination of Xval, Aval, and Bval
Purpose: This module is used to combine all the modules into a single multiplier.

Annotated Simulation Trace

Sign +*+ (3*3) Simulation



1. The Reset_Load_Clear button is pressed for one clock cycle, thus setting register A and the X value to 0 and loading the switch input value into register B, which is binary 00000011 In our specific case.

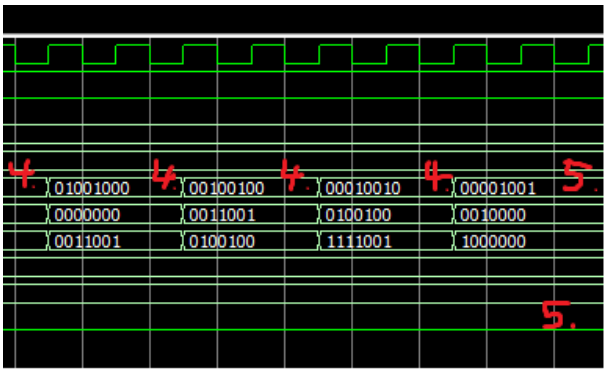
2. The Run button is pressed for one clock cycle, thus beginning our multiplication operation.

3. Since M = 1, we will perform the add operation, adding the switch values into the contents of register A.

4. Performing the right shift operation after an add or subtract operation or directly due to M being 0.

5. After 8 shifts are completed, the final result of the multiplication can be found in AB, or registers A and B. The 16-bit output is 0000000000001001, or decimal 9, for this specific case, which is as expected since we performed a 3*3 operation. The X-value is shown to be 0, which is also as expected in this case since our result is

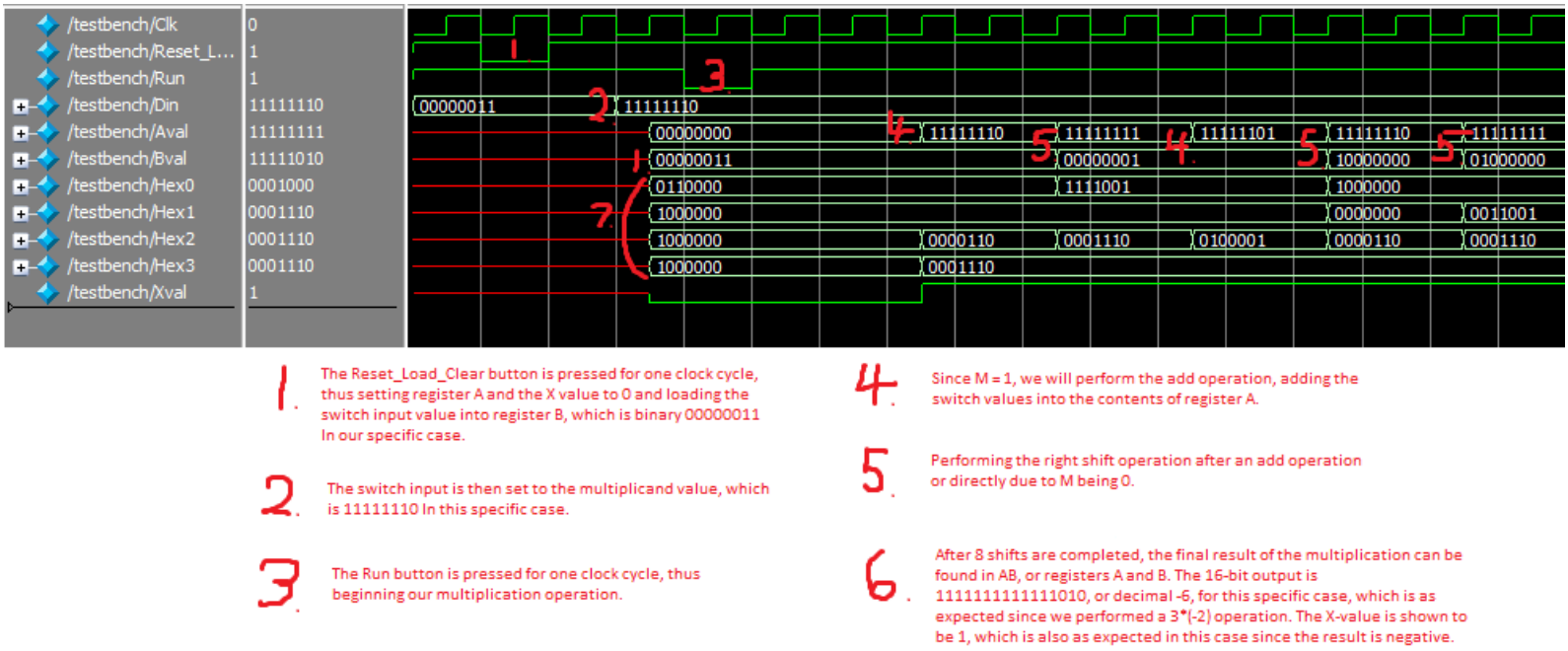
Figure III. +*+ (3*3) Operation Annotated Simulation Trace Part 1.



6. The HEX displays displaying the contents of registers A and B onto our FPGA, updated as needed between clock cycles.

Figure IV. +*+ (3*3) Operation Annotated Simulation Trace Part 2.

Sign +*- (3*(-2)) Simulation



1. The Reset_Load_Clear button is pressed for one clock cycle, thus setting register A and the X value to 0 and loading the switch input value into register B, which is binary 00000011 In our specific case.

2. The switch input is then set to the multiplicand value, which is 11111110 In this specific case.

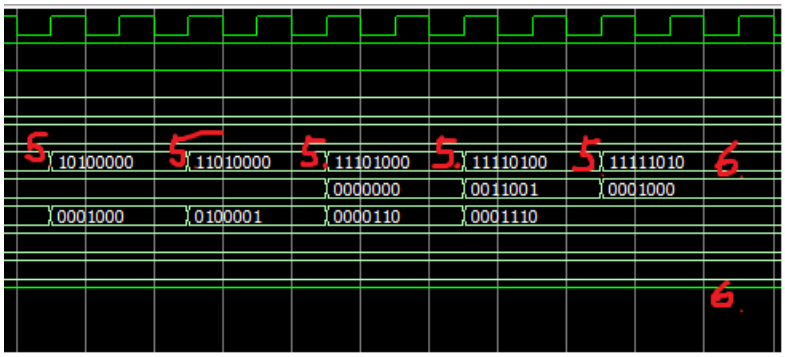
3. The Run button is pressed for one clock cycle, thus beginning our multiplication operation.

4. Since M = 1, we will perform the add operation, adding the switch values into the contents of register A.

5. Performing the right shift operation after an add operation or directly due to M being 0.

6. After 8 shifts are completed, the final result of the multiplication can be found in AB, or registers A and B. The 16-bit output is 1111111111111010, or decimal -6, for this specific case, which is as expected since we performed a 3*(-2) operation. The X-value is shown to be 1, which is also as expected in this case since the result is negative.

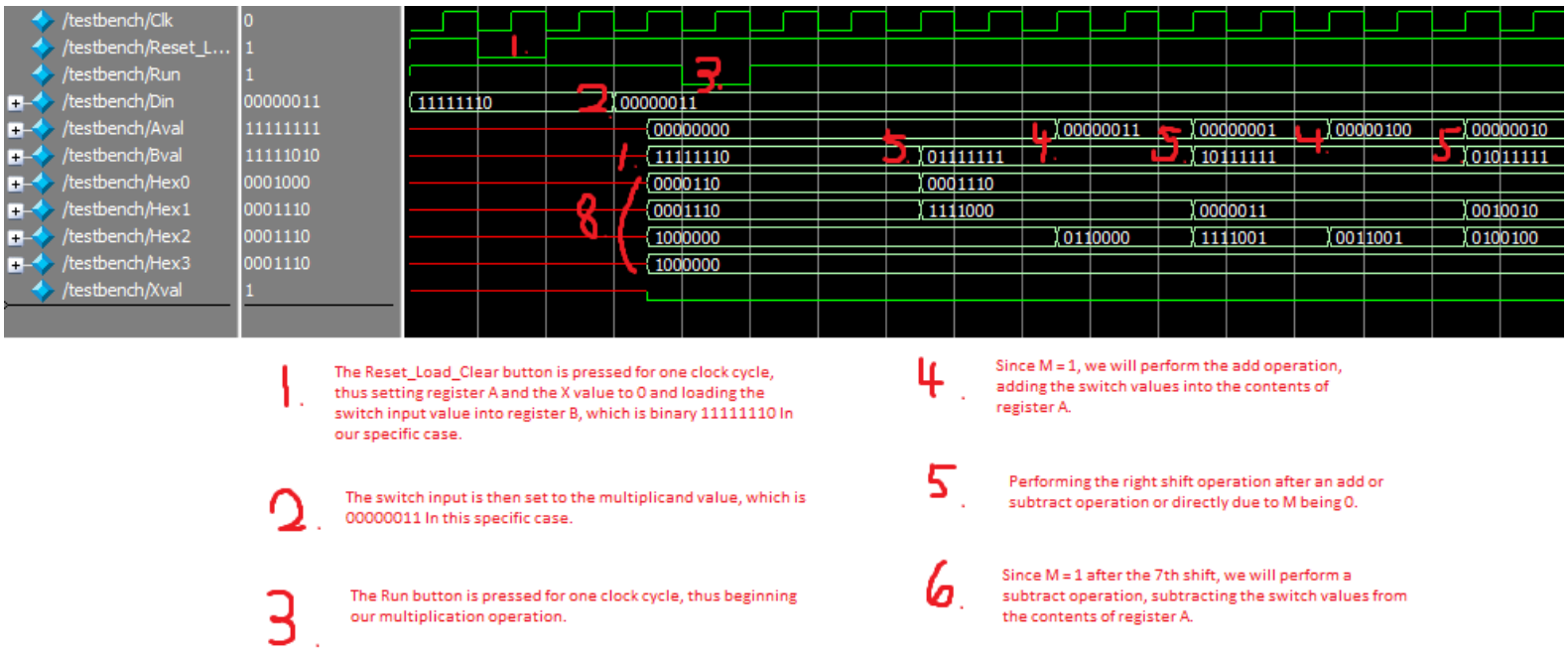
Figure V. +*- (3*(-2)) Operation Annotated Simulation Trace Part 1.



7. The HEX displays displaying the contents of registers A and B onto our FPGA, updated as needed between clock cycles.

Figure VI. +*- (3*(-2)) Operation Annotated Simulation Trace Part 2.

Sign -*+ ((-2)*3) Simulation



1. The Reset_Load_Clear button is pressed for one clock cycle, thus setting register A and the X value to 0 and loading the switch input value into register B, which is binary 11111110 In our specific case.

2. The switch input is then set to the multiplicand value, which is 00000011 In this specific case.

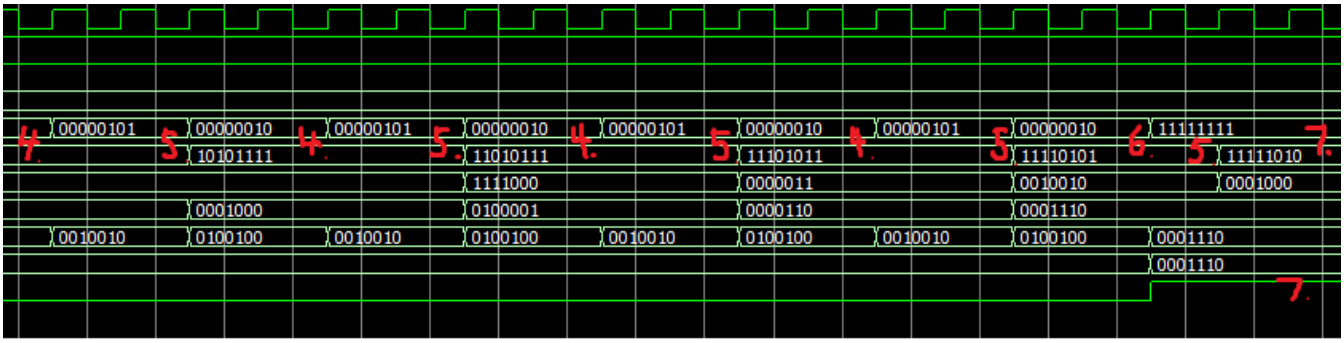
3. The Run button is pressed for one clock cycle, thus beginning our multiplication operation.

4. Since M = 1, we will perform the add operation, adding the switch values into the contents of register A.

5. Performing the right shift operation after an add or subtract operation or directly due to M being 0.

6. Since M = 1 after the 7th shift, we will perform a subtract operation, subtracting the switch values from the contents of register A.

Figure VII. -*+ ((-2)*3) Operation Annotated Simulation Trace Part 1.

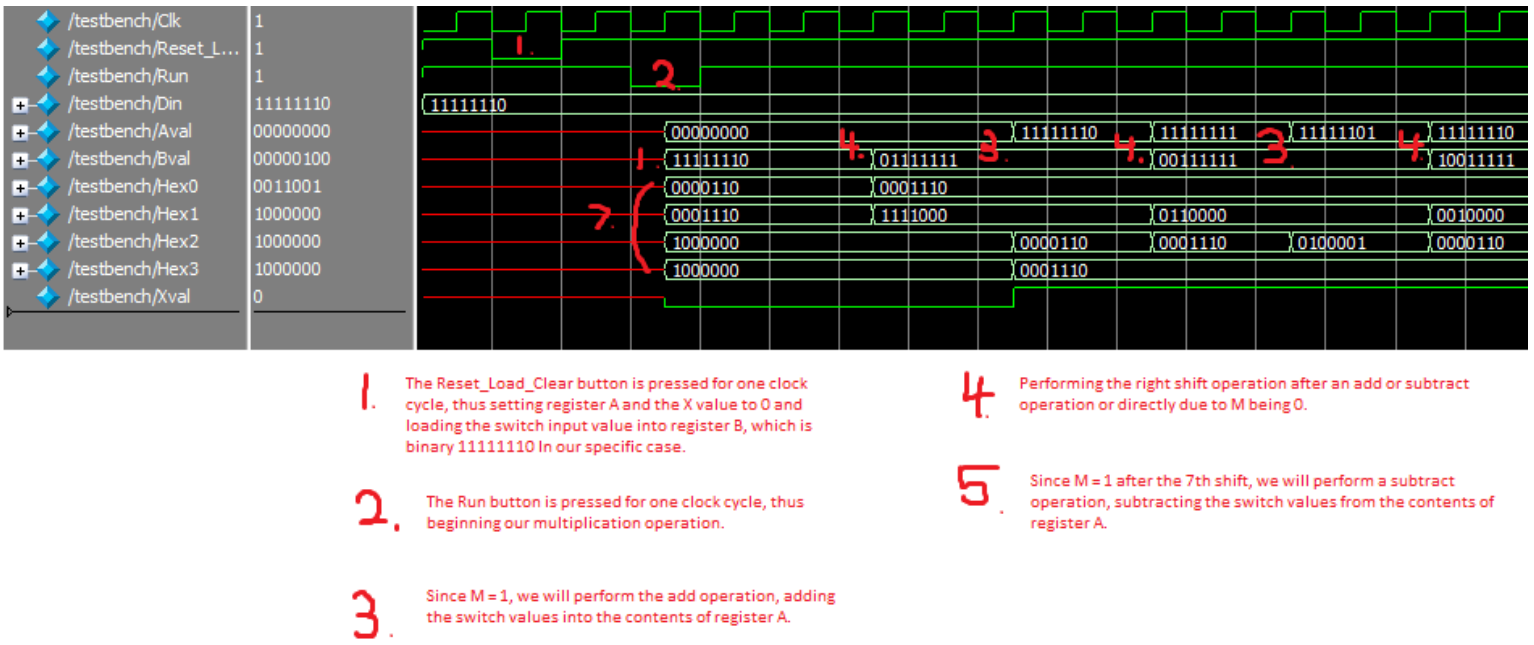


7. After 8 shifts are completed, the final result of the multiplication can be found in AB, or registers A and B. The 16-bit output is 1111111111111010, or decimal -6, for this specific case, which is as expected since we performed a (-2)*3 operation. The X-value is shown to be 1, which is also as expected in this case since our result is negative

8. The HEX displays displaying the contents of registers A and B onto our FPGA, updated as needed between clock cycles.

Figure VIII. -*+ ((-2)*3) Operation Annotated Simulation Trace Part 2.

Sign -*- ((-2)*(-2)) Simulation



1. The Reset_Load_Clear button is pressed for one clock cycle, thus setting register A and the X value to 0 and loading the switch input value into register B, which is binary 11111110 In our specific case.

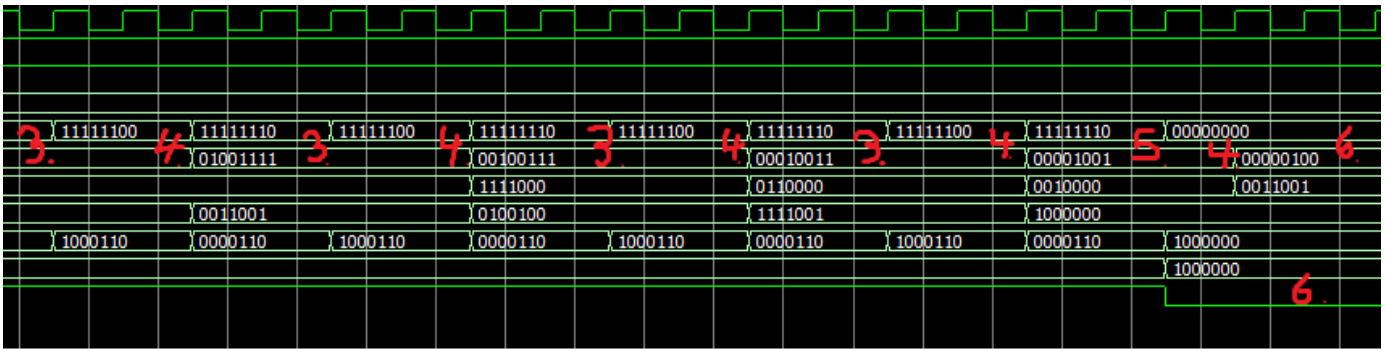
2. The Run button is pressed for one clock cycle, thus beginning our multiplication operation.

3. Since M = 1, we will perform the add operation, adding the switch values into the contents of register A.

4. Performing the right shift operation after an add or subtract operation or directly due to M being 0.

5. Since M = 1 after the 7th shift, we will perform a subtract operation, subtracting the switch values from the contents of register A.

Figure IX. -*- ((-2)*(-2)) Operation Annotated Simulation Trace Part 1.



6. After 8 shifts are completed, the final result of the multiplication can be found in AB, or registers A and B. The 16-bit output is 0000000000000100, or decimal 4, for this specific case, which is as expected since we performed a (-2)*(-2) operation. The X-value is shown to be 0, which is also as expected in this case since our result is positive.

7. The HEX displays displaying the contents of registers A and B onto our FPGA, updated as needed between clock cycles.

Figure X. -*- ((-2)*(-2)) Operation Annotated Simulation Trace Part 2.

Post-Lab Questions

1. Fill in the table shown in 5.6 with your design's statistics.

	Multiplier
LUTs	138
DSP	0
Memory (BRAM)	0/1,677,312
Flip-Flop	62
Max Frequency	77.2 MHz
Static Power	89.97 mW
Dynamic Power	2.63 mW
Total Power	105.51 mW

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

One of the ideas that we have to optimize the design to decrease the total gate count is to combine the adder and subtracter modules into one simple module that incorporates an additional signal which tells the module whether to perform a subtraction or addition. This would save us from having two separate modules, one for addition and one for subtraction like we had in our original design. Thus, the total number of gates used will decrease. On the other hand, our original implementation involved listing out every single possible state that we could transition to, without the utilization of a counter, which we modified for our lab report state diagram above. Thus, our original design required over 30 states, as each of the 8 sub-operations required a potential add or subtract state as well as a shift state. This dramatically increased the complexity of our design, increasing the total power required to perform a single multiplier operation and negatively impacting the maximum frequency. The change we made to the control unit state diagram above illustrates our refined design to increase the maximum frequency by incorporating the use of a counter. For counter less than 7, we will repeat the same potential add and shift states (forming a loop). Then, once counter hits 7, indicating the completion of 7 shifts, we will then perform the final potential subtraction operation followed by the 8th and final shift. This implementation should drastically decrease the complexity of our circuit and thus decrease our power consumption and increase maximum frequency.

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?

The purpose of the X register is to store the sign bit of the calculation. When the Reset_Load_Clear is high, X is cleared to 0. Otherwise, during the shifting process, X remains the same as it is an arithmetic right shift. Also, X is set during the addition or subtraction process, when values of the register A and the value on the switches, S, are extended to 9-bit and either added together or subtracted (A - S). The 9-bit result is then stored in XA, or registers X and A.

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

X then would not be able to reflect the correct sign bit of the operation. For example, if we add 8'b11111111 to 8'b00000001, then we would get a carry-out of 1 and an 8-bit value of 0, which contradicts the expected result of 0 as -1 + 1 = 0. On the other hand, if we sign extended them to 9-bit, then the result of 9'b111111111 plus 9'b000000001 would be 9'b000000000, which is the same as our expected result.

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

The main limitation of continuous multiplication is the register size. For our specific design, we utilized only 8-bit registers. This means we will run into issues with continuous multiplication once we try to perform another continuous multiplication operation on a result/new multiplier that exceeds the bounds [-128, 127]. Since continuous multiplication involves clearing registers X and A and relying solely on the contents of register B to represent the new multiplier in the next multiplication operation, if AB is not simply the sign-extended value of the 8-bits inside B, then our implemented algorithm will fail, because our multiplier will no longer be correct once cut down to 8-bits. 8-bit 2's complement can only represent numbers ranging from -128 to 127, which means that if our result of a previous calculation exceeds that range, then performing another multiplication operation will lead to our algorithm failing.

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

Starting with disadvantages, the implemented multiplication algorithm is slightly more complex and confusing than the pencil-and-paper method, particularly for operations involving small multiplier or multiplicand values. The pencil-and-paper method is more in-line with the traditional method of multiplication that we are familiar with. For small positive numbers, the pencil-and-paper method could also be quite faster than the implemented algorithm as it does not require N complete shifts like for N-bit multiplication operations. On the other hand, the main advantages of the implemented multiplication algorithm would be highlighted if the operation involved large multiplicand/multiplier and/or negative values. In these cases, the pencil-and-paper method would likely be more time-consuming and more prone to errors. This is because we would likely have to add or subtract a significant number of binary digits, which

could get really messy if there is a lot of carry-over to the next significant bit. In this case, performing addition and/or subtraction operations one at a time then shifting would be more efficient and less prone to errors.

Conclusions

Through this lab, we learned about the multiplier operation for 8-bit 2's complement numbers. We learned about the contrasting methods of implementation, namely one involving counters to simplify the number of states involved in our operation, which could get overwhelming with additional bits, and one involving listing out every single possible potential state. The former would be much easier to scale and serve as a template for larger bit multiplication operations.

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

As mentioned above, our original design approach was far more complex than necessary as we did not incorporate the use of a counter to track the number of shift states completed. Although functionally speaking, our design worked fine, it was very inefficient and complex. Therefore, in our lab report, we modified our design andn incorporated a counter to help us track the number of shifts and potential add operations we have completed before shifting to a potential subtract state and the final shift state. This helped us significantly reduce the complexity of our design and improve functionality. In addition, one design error that we ran into was the computation of several cycles with a single run button press on the FPGA, thus resulting in results different to those anticipated. As we later discovered, this error is based on the fact that a single button press, not matter how fast it appears to the naked eye, may continuously send the run signal for a significant amount of time such that our design completes several continuous multiplication operations. To work around this issue, we added an extra state that we enter once we press the run button, which will not transition into the actual computation cycle until we release 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.

No, we did not find anything particularly ambiguous or incorrect in the lab manual or given material.