**ECE 385**

Fall 2019

Experiment #5

An 8-Bit Multiplier in SystemVerilog

Jacqueline Jiang and JiHo Han

AB6 / Wednesday 12:00 - 2:50 p.m.

Nicholas Cebry and Yujie Zhou

# Introduction

## Summary of the Basic Functionality of the Multiplier Circuit

The main function of this circuit is multiplying two 8-bit 2’s complement numbers on the FPGA board to get a 16-bit product. Since this circuit uses 2’s complement, this circuit is capable of multiplying both negative and positive numbers. Our multiplier uses the add-shift algorithm to multiply its operands. This circuit supports consecutive multiplication, and it also has switches allowing the user to input their own operands.

# Pre-Lab Question

## Function Table

The table in Figure 1 demonstrates the add-shift algorithm by multiplying 7 with -59.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | **X** | **A** | **B** | **M** | **Next** | **Comments of Next Step** |
| Clear A/Load B | 0 | 0000 0000 | | 0000 0111 | 1 | ADD | add S to A, then shift |
| ADD | 1 | 1100 0101 | | 0000 0111 | 1 | SHIFT | shifting after add |
| SHIFT | 1 | 1110 0010 | 1 | 000 0011 | 1 | ADD | add S to A, then shift |
| ADD | 1 | 1010 0111 | 1 | 000 0011 | 1 | SHIFT | shifting after add |
| SHIFT | 1 | 1101 0011 | 11 | 00 0001 | 1 | ADD | add S to A, then shift |
| ADD | 1 | 1001 1000 | 11 | 00 0001 | 1 | SHIFT | shifting after add |
| SHIFT | 1 | 1100 1100 | 011 | 0 0000 | 0 | SHIFT | shift, don't add |
| SHIFT | 1 | 1110 0110 | 0011 | 0000 | 0 | SHIFT | shift, don't add |
| SHIFT | 1 | 1111 0011 | 0 0011 | 000 | 0 | SHIFT | shift, don't add |
| SHIFT | 1 | 1111 1001 | 10 0011 | 00 | 0 | SHIFT | shift, don't add |
| SHIFT | 1 | 1111 1100 | 110 0011 | 0 | 0 | SHIFT | shift, don't add |
| SHIFT | 1 | 1111 1110 | 0110 0011 | | 0 | NOP | done, stop |

*Figure 1: Function Table of Operation 00000111 \* 11000101 (7 \* -59)*

## 

# Written Description and Diagram of Multiplier Circuit

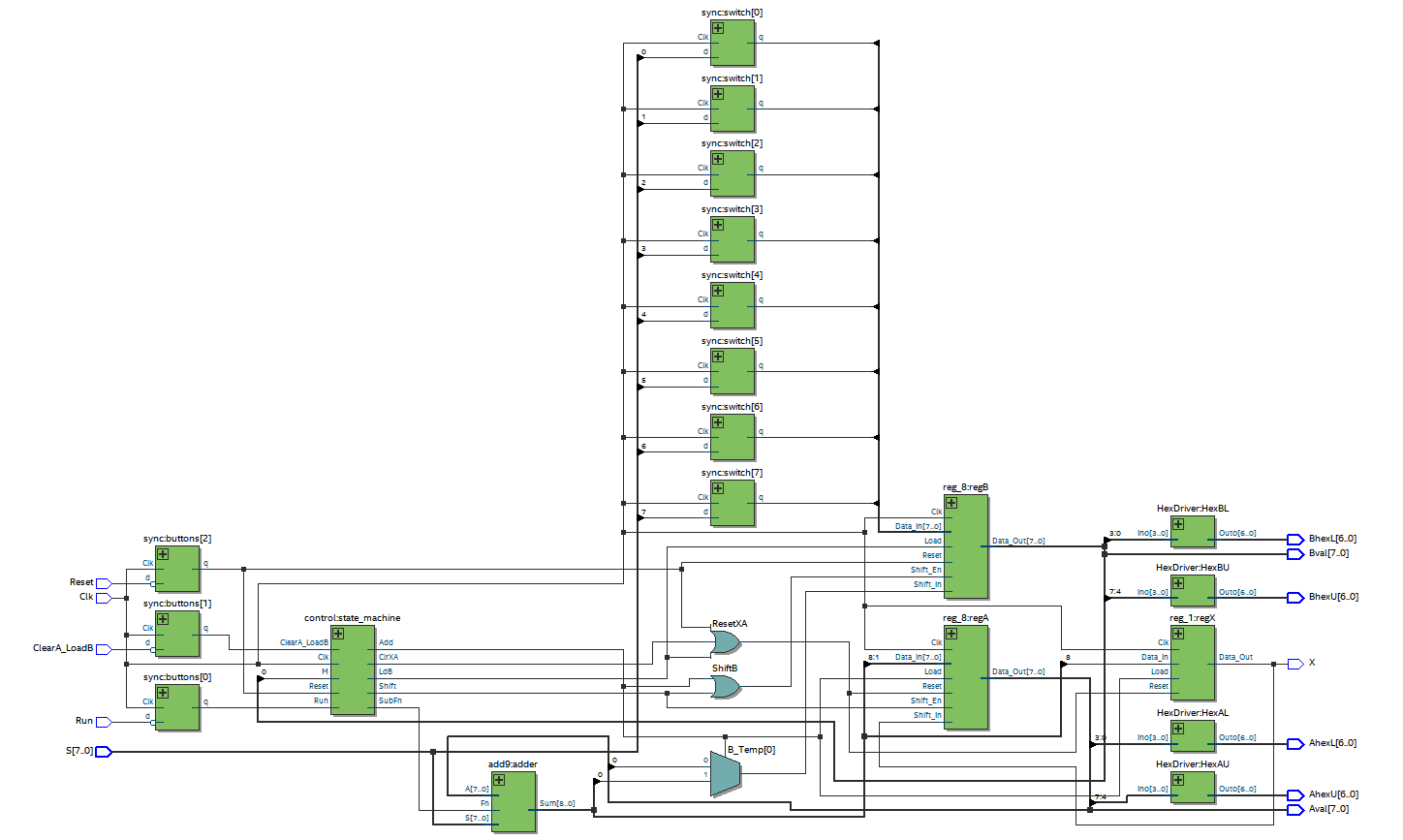
## Summary of Operation

The total multiplier circuit can be seen in Figure 2. The circuit has a Reset button that allows the user to reset the circuit back to its initial state. The circuit also has synchronizers on its switches and push buttons to synchronize these asynchronous inputs with the rest of the circuit. In order to multiply, first, the user must input the multiplier through the switches. Then, the ClearA\_LoadB button should be pressed to load the multiplier into register B while clearing registers X and A at the same time. After register B is loaded, the user should input the multiplicand through the switches. When the user is ready, the Run button should be pressed to begin the operation.

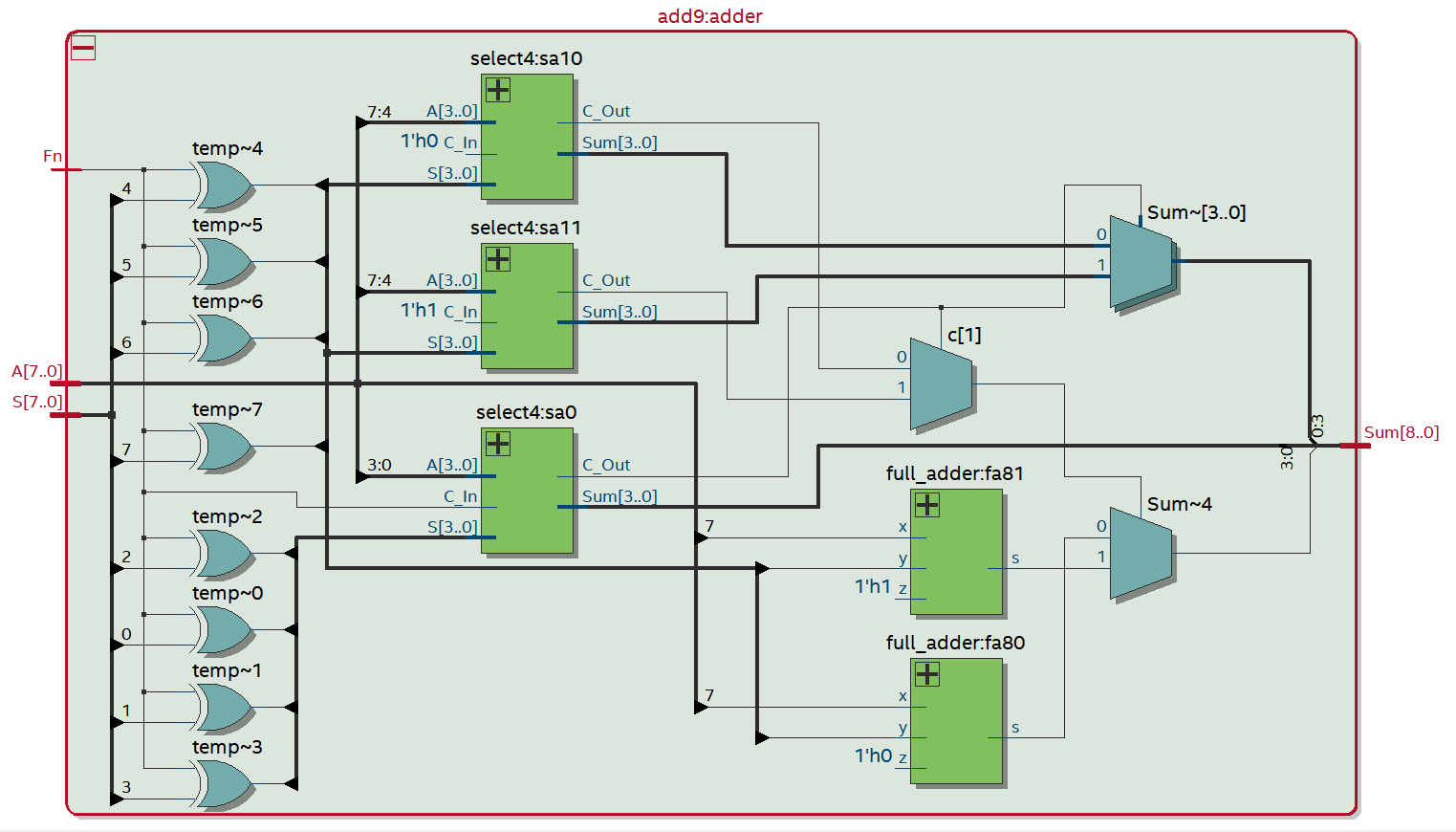
The multiply operation is a combination of adding and shifting. Whether the circuit should add and shift or just shift depends on M, which represents the least significant bit in register B. If M is equal to 1, then the circuit will perform an add-shift operation. The add operation is executed by a 9-bit adder (Fig. 3). This 9-bit adder will sign-extend the multiplicand from the switches and register A to 9 bits and then add them together using the carry-select method, which was implemented using full adders. The most significant bit of the sum is loaded into register X, and the rest of the sum is stored into register A. The carry-out bit of the sum is discarded. It is important to note that register X holds the sign bit of the product, so if the current output is negative, X will hold the value 1, and if the current output is positive, X will hold the value 0. After the add operation, the register X, A, and B are arithmetically shifted right by one bit with register X preserving its value until the next add/subtract operation. That is the value in register X will be shifted into register A, and the least significant bit of A is shifted into B. The least significant bit of B, otherwise known as M, is shifted out of the circuit, and the new value of M is used. On the other hand, if M is equal to 0, the adding operation is skipped and registers X, A, and B are similarly right shifted. The edge case of this algorithm is the most significant bit of the multiplier. If this bit is 1, the circuit will subtract instead of adding to account for the negative multiplier. This is also done through the 9-bit adder, which has a Fn input to allow the user to choose between adding or subtracting.

Regardless of the value of M, the circuit will always shift registers X, A, and B. As such the final product can be found by combining the values in registers A and B. That is, register A will hold the upper 8 bits of the product and register B will hold the lower 8 bits of the product. Register X represents whether the final answer is positive (if X is 0) or negative (if X is 1). This final product is displayed in the Hexadecimal display of the board. All states and control signals within the circuit are set by a finite state machine (Fig. 4).

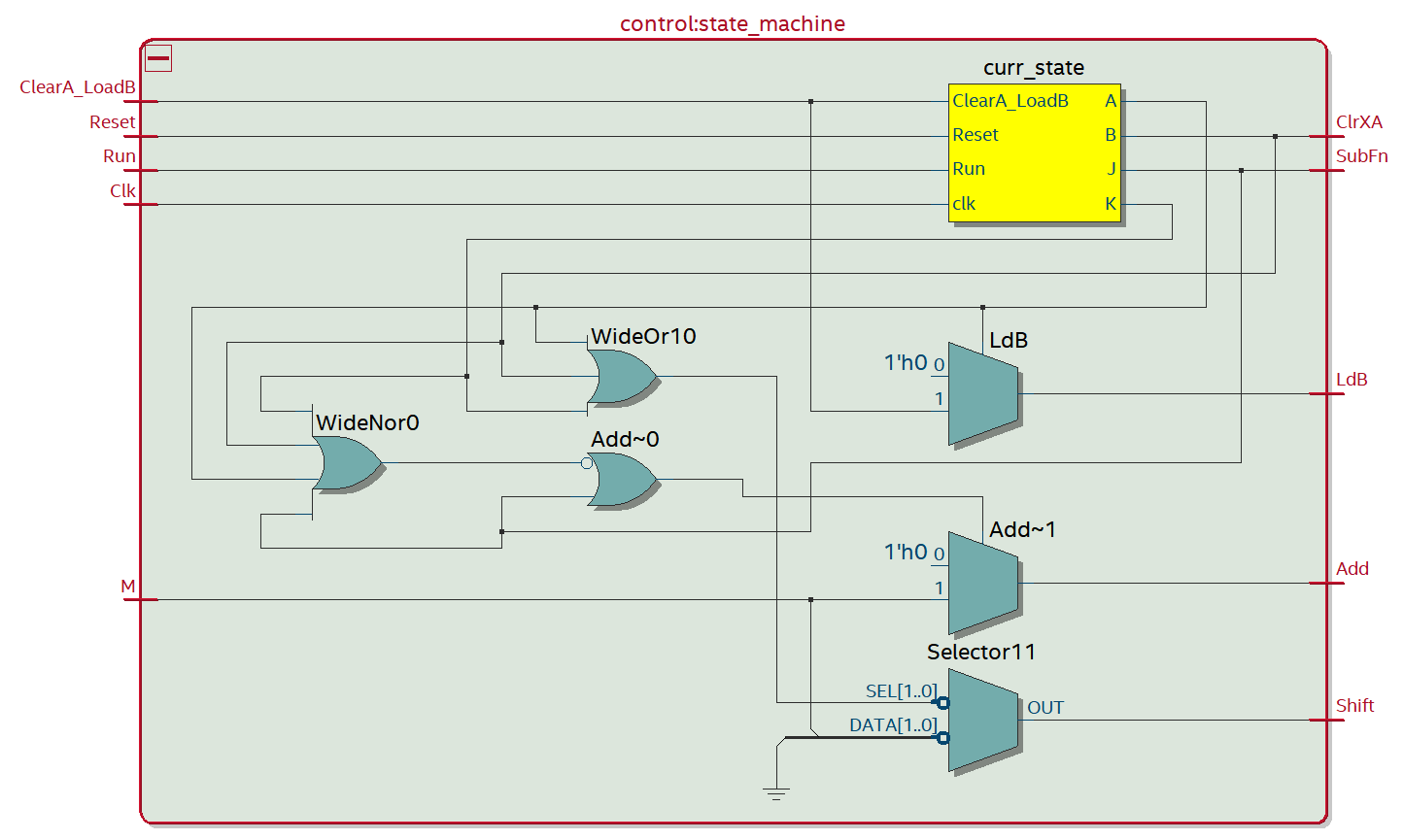
## Block Diagrams of Multiplier Circuit



*Figure 2: Top Level Block Diagram of the Multiplier*



*Figure 3: Block Diagram of the 9-bit Adder*



*Figure 4: Block Diagram of the Control Unit*

## 

## 

## Written Description of .sv Modules

Module: lab5\_top from *lab5\_top.sv*

Inputs: Clk, Reset, Run, ClearA\_LoadB, [7:0]S

Outputs: [6:0]AhexU, [6:0]AhexL, [6:0]BhexU, [6:0]BhexL, [7:0]Aval, [7:0]Bval, X

Description: This is a top-level entity that connects the LEDs and buttons from the board to the rest of the multiplier circuit.

Purpose: This module is used as a top-level wrapper to connect the other modules to the inputs and outputs of the FPGA board.

Module: full\_adder from *full\_adder.sv*

Inputs: x, y, z

Outputs: s, c

Description: This is a 1-bit adder that takes in two operands (x and y) and a carry-in bit (z) and outputs the sum (s) and carry-out bit (c).

Purpose: This module is used to add two 1-bit values together. Specifically, it is used in the select4 module and the add9 module to add values together.

Module: select4 from *add9.sv*

Inputs: [3:0]A, [3:0]S, C\_In

Outputs: [3:0]S, C\_Out

Description: This is a carry-select unit that will add 4-bit inputs together. It will first calculate all possible carry-bit combinations for every bit after bit 0 and then select the correct carry bit and sum afterwards.

Purpose: This module is used to add 4-bit inputs together using the carry-select method. It was created to help parallelize the load for our 9-bit adder.

Module: add9 from *add9.sv*

Inputs: [7:0]A, [7:0]S, Fn

Outputs: [8:0]Sum

Description: This is a 9-bit carry-select adder. It takes in two 8-bit operands, sign-extends them to 9 bits, and adds them together to produce a 9-bit sum using the carry-select method. This 9-bit adder also doubles as a 9-bit subtractor when Fn is set to a logic one.

Purpose: This module is used to create an adder/subtractor to add the switch input to register A whenever an addition operation was required in the add-shift method of multiplication.

Module: reg\_1 from *registers.sv*

Inputs: Clk, Reset, Load, Data\_In

Outputs: Data\_Out

Description: This is a positive-edge triggered 1-bit register that is capable of synchronously resetting and parallel loading. When Reset is high, the register will be cleared with zeroes on the positive edge of Clk. When Load is high, the register will be loaded with Data\_In on the positive edge of Clk,

Purpose: This module is used to create a 1-bit register to hold X in the multiplier circuit.

Module: reg\_8 from *registers.sv*

Inputs: Clk, Reset, Load, Shift\_In, Shift\_En, [7:0]Data\_In

Outputs: [7:0]Data\_Out

Description: This is a positive-edge triggered 8-bit register that is capable of synchronously resetting, parallel loading, and right shifting. When Reset is high, the register will be cleared with zeroes on the positive edge of Clk. When Load is high, the register will be loaded with Data\_In on the positive edge of Clk, When Shift\_En is high, the register will shift right by 1 bit on the positive edge of Clk.

Purpose: This module is used to create 8-bit registers to hold the values of A and B in the multiplier circuit.

Module: control from *control.sv*

Inputs: Reset, Clk, Run, ClearA\_LoadB, M

Outputs: LdB, ClrXA, Shift, Add, SubFn

Description: This is a hybrid of a Mealy and Moore finite state machine and is used to control the current and next states of the circuit. The next state depends mostly on the current states, but it occasionally will depend on the inputs (like Run and Reset) to the circuit. It includes a total of 11 states, and it will assign the appropriate control signals for the given state. The state changes with the positive edge of Clk.

Purpose: This module is used to create a finite state machine. It intakes the inputs from the FPGA board and sends out the appropriate control signals for each state so that our multiplier knows what to do at any given point in time.

Module: sync from *Synchronizers.sv*

Inputs: Clk, d

Outputs: q

Description: This is a positive-edge triggered 1-bit flip-flop that is used to synchronize the inputs with the rest of the circuit. On the positive edge of Clk, data from d is loaded into q.

Purpose: This module is used to synchronize the inputs from the FPGA board to the logic inside the circuit by using flip-flops synced up to the clock.

Module: HexDriver from *HexDriver.sv*

Inputs: [3:0]In0

Outputs: [6:0]Out0

Description: This is a converter that takes in a 4-bit hexadecimal number and transforms it into an on-board seven-segment display output.

Purpose: This module transforms a hexadecimal input into the corresponding output signal for the seven-segment display.

Module: testbench from *testbench.sv*

Inputs: None

Outputs: None

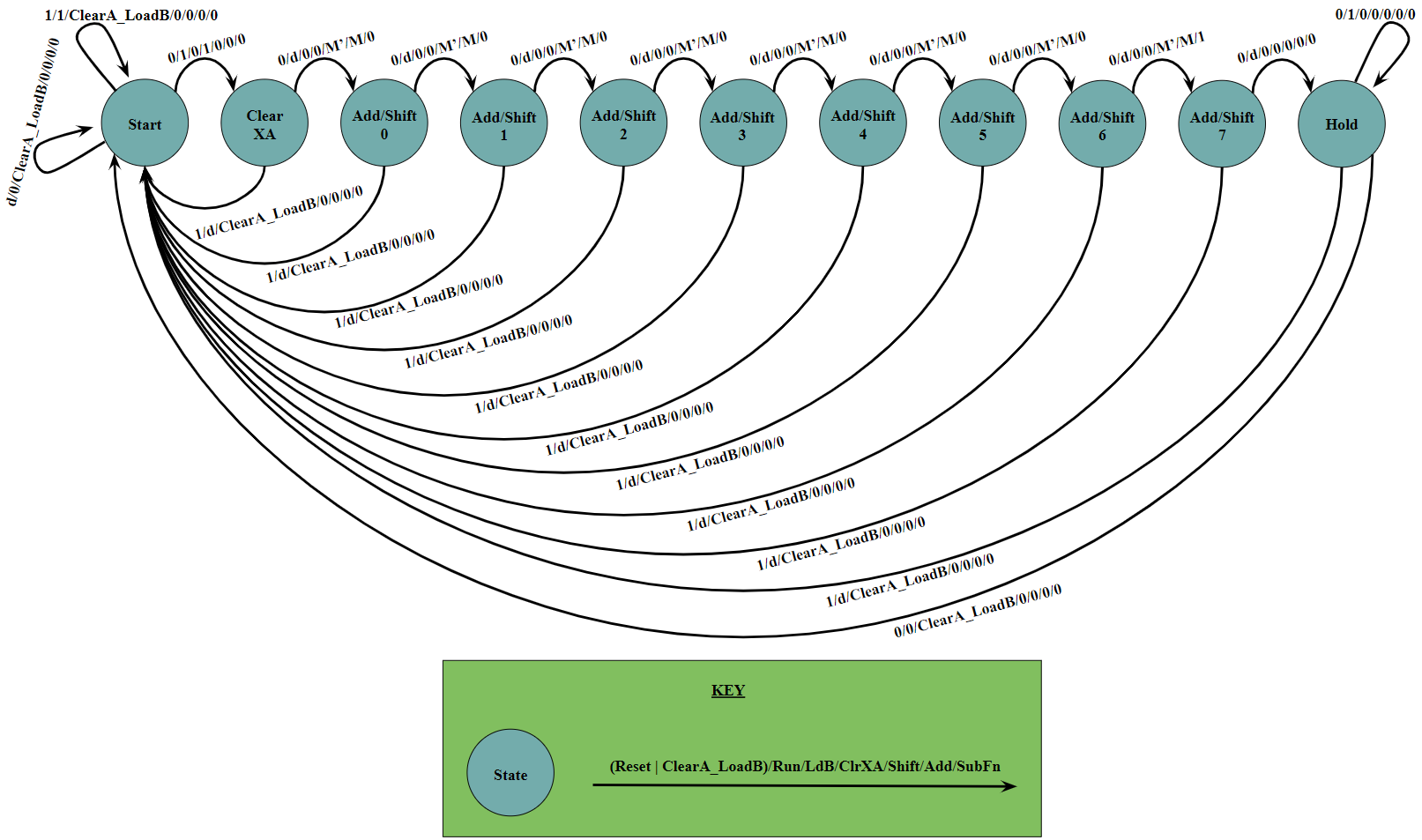
Description: This is a testbench used to simulate the waveforms of the multiplier circuit. It tests -\*+, +\*-, +\*+, and -\*- operations as well as consecutive multiply operations (without ClearA\_Load\_B or Reset). It will keep track of the amount of wrong products that we find and report it at the end of the simulation.

Purpose: This module is used to test the written code to see if it is able to multiply consecutively with both positive and negative operands.

## State Diagram for Control Unit

The state diagram in Figure 5 represents the states within the control unit. In this lab, we had a total of 11 states. The very first state is the Start state, which is the state that our finite state machine will always return to at the end of each operation and if either Reset or ClearA\_Load B is pressed. We chose this design because we wanted Reset to restart the whole circuit no matter where it is at, and we wanted to allow the user to clear registers X and A and load in a new multiplier in register B whenever they wanted by pushing the ClearA\_LoadB button. The next state, Clear XA, is used to clear registers X and A to ensure that our multiplier can execute consecutive multiply operations without having to press ClearA\_LoadB or Reset. The next 8 states are the add-shift operations executed on each bit of the operand. Whether the circuit will add-shift or just shift depends on M as mentioned in the operation of the circuit. Finally, the last state, Hold, forces the circuit to wait until the user lets go of the Run button before progressing, which makes sure that the multiply operation is performed only once on each cycle.

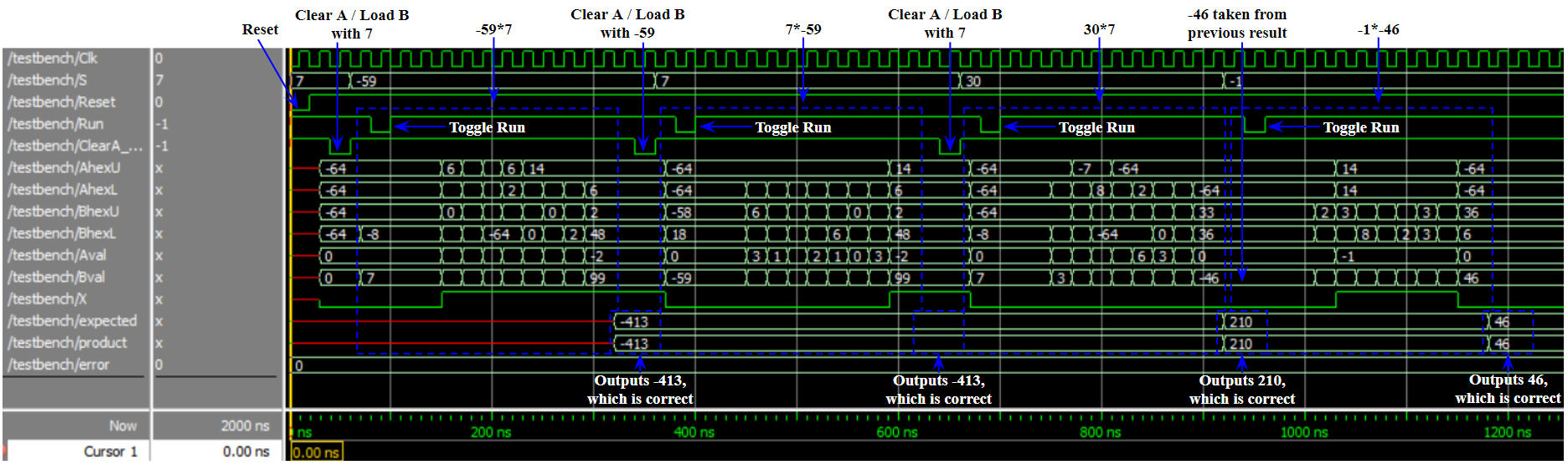
The transition labels on the arrows represent the inputs and outputs of the circuit. The *d* stands for don’t cares, which means we do not care about the value of the signal. *M*, as stated earlier, represents the least significant bit of register B, and its inverse is represented by *M’*. *ClearA\_LoadB* represents the inverted signal of ClearA\_LoadB from the push buttons.



*Figure 5: Finite State Diagram of Multiplier Circuit*

# Annotated Pre-Lab Simulation Waveforms

Figure 6 shows the simulated waveforms for 4 multiply operations. All numbers in the waveforms are represented in decimal to improve readability. The waveform “product” is the combination of the signals X, Aval, and Bval so that we can more easily examine the final product of the operation. The waveform “expected” is the expected value of the operation. The first multiply operation tested the negative times positive combination (-59 \* 7), and our circuit successfully outputted -413. The second multiply operation tested the positive times negative combination (7 \* -59), to which our circuit also successfully outputted -413. The third multiply operation multiplied two positive operands (30 \* 7), and our circuit gave the correct product 210. The fourth tested two negative operands along with consecutive multiplications by using the value in register B (-46) and multiplying it with -1. Our circuit outputted 46, which is expected.



*Figure 6: Simulation of 4 Different Multiplication Operations*

# Answers to Post-Lab Questions

## Design Statistics Table and Ideas for Improving the Design

Figure 7 below shows the design statistics of our multiplier circuit.

Two main different ways exist for improving the design; one is to change the components implemented, and the other is to change the algorithm embedded in the design. There are many binary multiplication algorithms, and therefore there are multiple ways of improving the implemented design. However, every design will have its pros and cons since each one has a different balance between performance and cost.

In terms of changing the components implemented, switching from the 9-bit carry-select adder to a different type of adder could be useful, depending on which aspect we would like to prioritize. Currently, our carry-select adder, though the fastest method out of the three we covered in class, requires the most power and a lot more logic gates to implement. As such, if we were looking to optimize the power and area of our circuit at the cost of speed, using a carry-ripple adder would be ideal because it will reduce power consumption and the number of logic gates even though it will run at a lower frequency. We could also use the more balanced carry-lookahead adder which would reduce power consumption and could possibly reduce the number of logic gates used while still outputting a better frequency than the carry-ripple adder. However, this option will need more investigation in whether a 9-bit carry-lookahead adder is more efficient than a 9-bit carry-select adder in terms of speed versus area.

If we wish to change the algorithm used to increase the maximum frequency, one possible algorithm out of many is the Toom-Cook multiplication algorithm, which is an excellent choice for multiplication of numbers with excessively long bits. Normally, other algorithms will require (n\*m) numbers of smaller multiplication operations to produce the final product, where n and m are a number of bits of the multiplicand and multiplier. However, since the Toom-Cook algorithm utilizes the divide-and-conquer algorithm, the number of multiplication operations required is drastically reduced at the cost of using much more logic gates. This is especially good for digital systems that handle excessively large numbers. For instance, in order to multiply two 2048-bit numbers, the implemented algorithm will require numbers of multiplication, whereas Toom-Cook will only need 177147 numbers of multiplication, which is approximately 24 times fewer multiplications.

|  |  |
| --- | --- |
| **LUT** | 101 |
| **DSP** | 0 |
| **Memory (BRAM)** | 0 |
| **Flip-Flop** | 39 |
| **Frequency (MHz)** | 238.55 |
| **Static Power (mW)** | 98.52 |
| **Dynamic Power (mW)** | 0 |
| **Total Power (mW)** | 147.8 |

*Figure 7: Design Statistics Table*

## Purpose of the Register X

The X primarily represents the most significant bit of A after each add-shift operation, which is ultimately the sign of the current value. Just like 2’s complement, 0 means positive, whereas 1 means negative. The purpose of the register X is to hold the sign bit value to make the shifting process easier, and it allows the user to quickly detect if the final product of the multiply operation is positive or negative. Outside of clearing and resetting, the register X is only changed by the add/subtract operation in the algorithm.

## Limitation of Continuous Multiplication

The biggest limit of continuous multiplication is in its size. Our multiplier can only multiply 8-bit operands. As such, if the product of the previous multiplication operation needs more than 8 bits to be represented, the following multiplication operation will result in an incorrect product because more bits than the ones available are needed to represent the previous product as a multiplier. An example of this can be seen in the simulation waveforms in Figure 6. The very last multiplication operation is an example of continuous multiplication because neither Reset nor ClearA\_Load B was pressed. The previous product, 210, was transformed into -46 in the last multiplication operation because the number 210 required 9 bits to be represented in 2’s complement, which was 1 bit more than the maximum size of the multiplier. As such, the operation was carried out with -46 as the multiplier instead of 210.

## Advantages and Disadvantages of the Add-Shift Method Compared to Paper-and-Pencil Method

The biggest advantage of the add-shift method over paper-and-pencil is in the reduction of resources required. When using the paper-and-pencil method, we need a register for each addition operation, which will drastically increase the size of the total circuitry. On the other hand, the add-shift method only requires 2 registers and a flip-flop for the most significant bit. Moreover, our implemented multiplier reuses registers by saving half of the product in the register that used to contain the multiplier. This makes the algorithm much more efficient while reducing the resources consumed. This is convenient in implementing bigger systems as it automatically saves the running product of the multiplication with each shift. Additionally, it is useful for consecutive multiplication when the product can be represented in less than 8 bits because the user will not have to load in the multiplier.

However, this feature is not always convenient as the add-shift method requires much more time to produce the output compared to the paper-and-pencil method. The paper-and-pencil method can be parallelized because each add operation can be calculated separately from each other since this operation only depends on the multiplicand and multiplier. On the other hand, the add-shift method requires that the previous add-shift operation finishes completely before the next add-shift operation can begin since it relies on the running product of the entire multiplication operation.

# 

# Conclusion

## Functionality, Faults in Design and its Solution

The design of this lab is to multiply two 8-bit 2’s complement numbers together to produce a 16-bit 2’s complement product using SystemVerilog to design the circuit. In the end, our design worked properly as intended, and produced the correct products for all combinations of positive and negative 2’s complement numbers. Additionally, our multiplier used the add-shift method and supported consecutive multiplications.

We ran into very few bugs in the process of design our adder. Other than the common small mistakes like missing the occasional semicolon and such, which were easily caught by the Quartus compiler, the main bug that we had was not accounting for consecutive multiplication. In our first implementation, we forgot to account for this feature, so our registers X and A were not cleared before we multiplied consecutively. This created issues when we needed to add/subtract within our circuit since there were garbage values cluttering our registers. However, because we wrote a testbench to test for this feature, we were easily able to identify this issue as we saw that register A was not zeroed out when our consecutive multiplication operation began. We fixed this bug by adding an additional state into our state machine to clear registers X and A before we began to add and shift.

One thing that could be improved from the lab manual is the given pencil-and paper multiplication demonstration on page 5.1. The addition operations were badly aligned which made it a bit more difficult to see the transition from the pencil-and-paper method to the add-shift method shown in the table on page 5.2. One thing that we did appreciate about the lab manual is the pre-lab question to rework the function table of the multiplication example presented, using 7 as the multiplier and -59 as the multiplicand. This greatly improved our understanding of how the add-shift method actually worked.