ECE 385

Fall 2019

Experiment #5

A 8-Bit Multiplier in SystemVerilog

Xinglong Sun Churan He
ABD T 8AM
Mihir Iyer

Introduction

a. Basic Functionality

In this experiment, we design and build a 8-bit multiplier with SystemVerilog on an FPGA board. The multiplier is capable with 2's complement signed number multiply calculation, and it utilizes bit-shifting method to find the 16-bit binary result. The circuit contains Control Unit (with a state machine to control different components), Adder Unit (doing actual calculation), Register Unit (storing value for calculation and result) and Hex Driver Unit (showing the value on display). For the upper level operation, user need to reset the system, load the first number to Register B, switch to the second number, press the Run button, and the multiply result of these two numbers would be shown on the display.

Pre-lab question

a. A concrete example

Let's see a concrete example of how to perform multiplication using the add-shift method. Specifically, we will multiple the contents of register B and the switches S, leaving the result in registers AB. Variable X indicates the sign of the result, and M is the least significant bit of register B.

Initial Values: X = 0, A = 00000000, B = 00000111(7 in decimal), S = 11000101(-59 in decimal). The initial values are achieved by the ClearA_LoadB signal which brings our finite state machine to a Clr_Ld state.

Function	X	A	B	M	Next Step
Clr_Ld	0	00000000	00000111	1	Since M=1, S will be added to A
ADD	1	11000101	00000111	1	Shift XAB right by one bit
SHIFT	1	11100010	10000011	1	Since M=1, S will be added to A
ADD	1	10100111	10000011	1	Shift XAB right by one bit
SHIFT	1	11010011	11000001	1	Since M=1, S will be added to A
ADD	1	10011000	11000001	1	Shift XAB right by one bit
SHIFT	1	11001100	01100000	0	Don't ADD since M=1, Shift XAB
SHIFT	1	11100110	00110000	0	Don't ADD since M=1, Shift XAB
SHIFT	1	11110011	00011000	0	Don't ADD since M=1, Shift XAB

SHIFT	1	11111001	10001100	0	Don't ADD since M=1, Shift XAB
SHIFT	1	11111100	11000110	0	Don't ADD since M=1, Shift XAB
SHIFT	1	11111110	01100011	1	8th shift done. Stop. Result in AB

Written description and diagrams of multiplier circuit

a. Summary of Operation

How the operands are loaded

There are two operands in one multiply operation. The first operand would be stored in Register B, and it's controlled by the ClearA_LoadB button. The second operand isn't directly stored at any place, but need to be kept on the switch input. If the last digit of Register B (M) is 1, the adder would add/subtract the contend in Register A with the switch input, and the result would be feedback to the Register A.

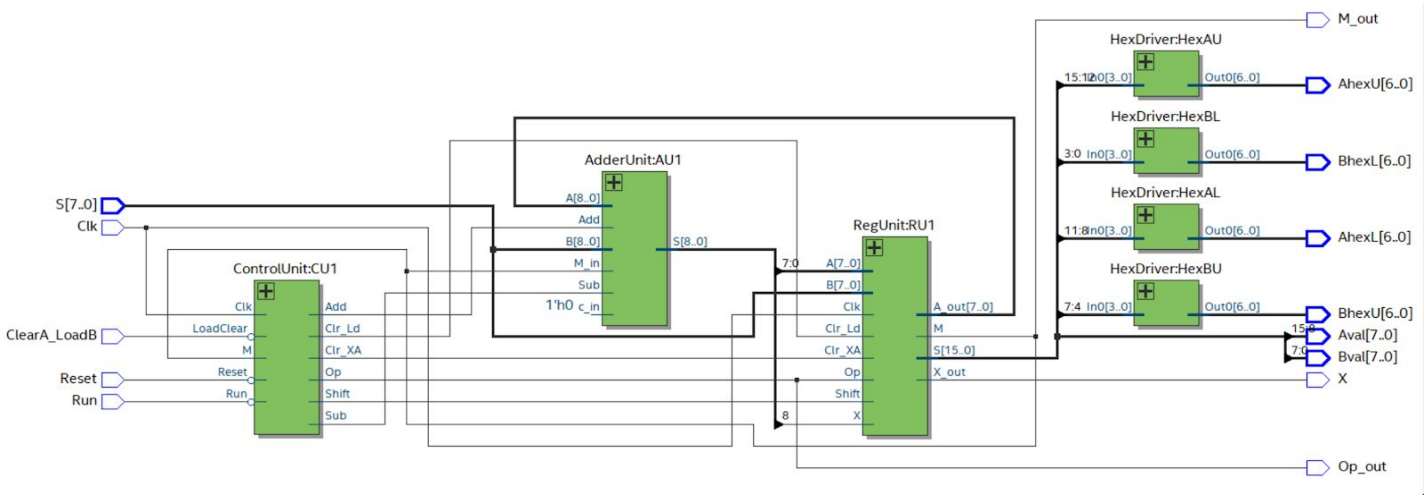
How the multiplier computes its result

The multiplier using bit-shifting addition to computes its result. Register A and Register B forms the Register Unit. After calculating one bit, the content in the Register Unit would perform an arithmetic right shift for the next bit. The content within Register A would be added/subtracted with the switch input value depending on current last digit in Register B (M). The multiply is complete when the arithmetic right shift performs 8 times (original Register B data disappears) .

How the result is stored

The adder would add/subtract the current Register A value with the switch input value, and feed the value back to Register A. The data in Register A is updated by the adder. With the arithmetic right shift, the last bit in Register A would be the first bit in Register B. When this arithmetic right shift performs 8 times, and addition/subtraction performs the number of '1' in the original Register B value of times, the multiply process is finished, and the result is the current 16-bit value stored in Register A and Register B.

b. Block Diagram



c. Modules Descriptions

As shown in the above diagram, there are three units in our design : Adder Unit, Register Unit, and Control Unit. In terms of ".SV" modules, we created 7 modules : Multiplier.sv, ShiftReg.sv, bitsFullAdder.sv, AdderUnit.sv, RegUnit.sv, ControlUnit.sv, and HexDriver.sv.

Module: Multiplier.sv
Inputs: Clk, Reset, Run, ClearA_LoadB,
[7:0] S
Outputs:
X, M_out, Op_out,
[7:0] Aval,Bval,
[6:0] AhexL,AhexU,BhexL,BhexU
Description: This is the top-level unit in our design. It's a positive-edge triggered 8-bit multiplier with synchronous reset. After hitting the Reset button which brings our circuit into the Reset state, user can load switch values(S) into Register B by hitting ClearA_LoadB button. When Run is high, the circuit is going to perform the actual Multiply operation, which generates the product of values of RegisterB and Switches.
Purpose: It wraps up the three units(Adder, Reg, and Control) mentioned above, and display the results on LEDs by outputting corresponding hex values.

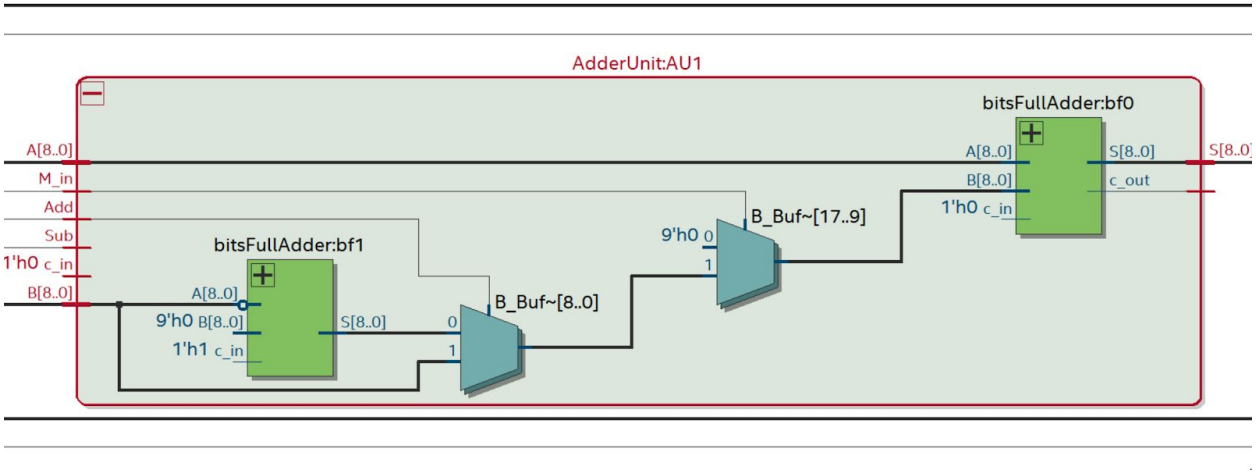
Module: bitsFullAdder.sv
Inputs: c_in,[8:0] A,B,
Outputs: [8:0] S,
c_out
Description: This is a 8-bit full adder, which computes the sum of A and B with c_in as carry-in value. More specifically, we use the naive Ripple Carry Design to connect 8 1-bit full adder serially together.
Purpose: This is a fundamental building block used in later parts of the circuit.
Our Adder Unit is going to be based on this 8-bit full adder.

Module: AdderUnit.sv
Inputs: c_in, Add, M_in,
[8:0] A,B
Outputs: [8:0] S,
c_out
Description: This 8-bit AdderUnit is based mainly on the 8-bit full adder with a few more logics. The logic can be shown clearly by the following table:

M_in	Add	S

0	x	A
1	1	A+B
1	0	A-B

A-B is achieved by first obtaining the two's complement negated value of B. We use another adder to perform (~B+1) to generate the value of -B.
Purpose: AdderUnit is going to perform the actual computation(Add and Subtraction) needed in computing the product. M_in and Add are fed in by the Control Unit.



AdderUnit Block Diagram

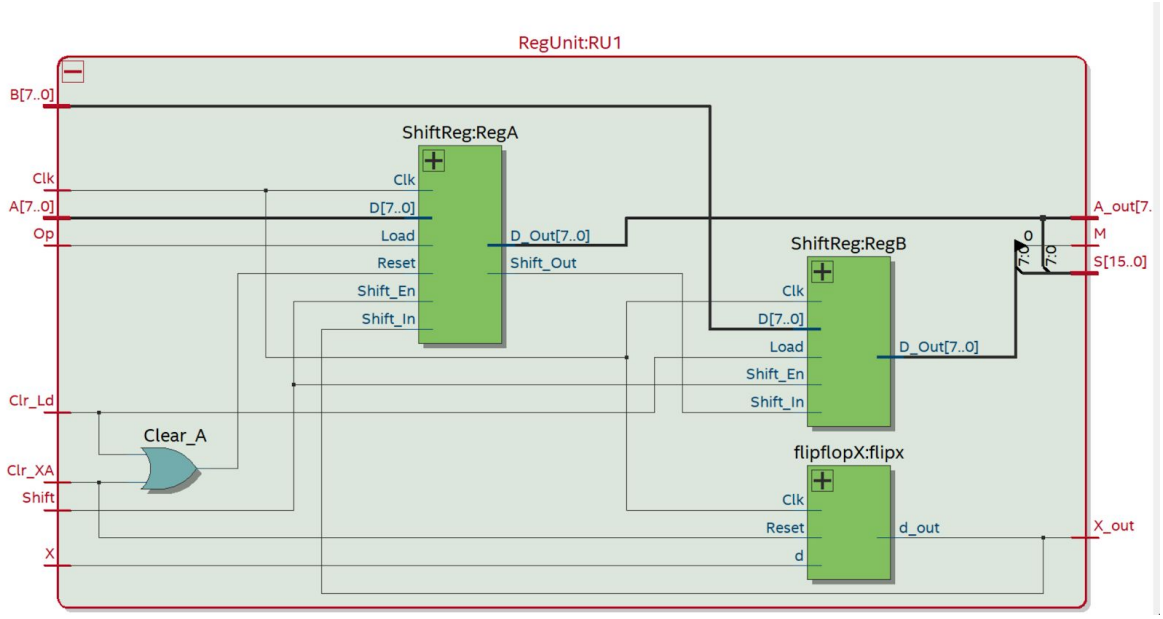
Module: ShiftReg.sv
Inputs: Clk, Reset, Load, Shift_In, Shift_En,
[7:0] D
Outputs: Shift_Out,
[7:0] D_Out
Description: This is a 8-bit shift register with synchronous Reset and synchronous load. When Load is high, data is loaded from D into the register on the positive edge of Clk. When Shift_En is high, register is going to perform a right shift taking into the Shift_In.
Purpose: This is a fundamental building block used in later parts of the circuit.
It's going to be a major component in the Register Unit.

Module: RegUnit.sv
Inputs: Shift, Clr_Ld, Clr_XA, Clk, X, Op,
[7:0] A, B,

Outputs: M,X_out
[15:0] S,
[7:0] A_out,

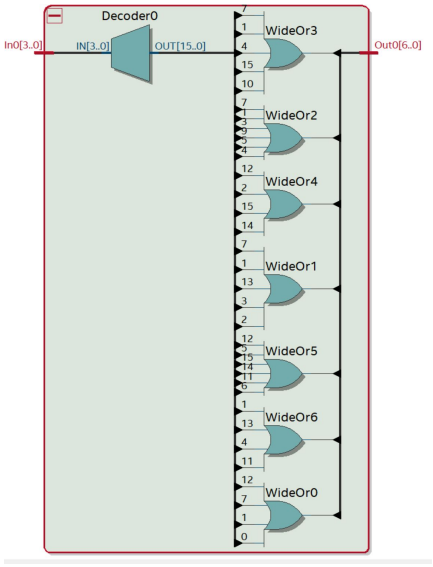
Description: The RegUnit contains two basic 8-bit shift registers described above with simple additional logic and manipulation. When Clr_Ld is high, value in Register A is going to be cleared and Register B is going to be loaded with the switch values. When Clr_XA is high, values in Register A and X are going to be cleared. When Op is high, Register A is going to be fed into the computation result. When shift is high, all of X, Register A, and Register B are going to perform an arithmetic right-shift (as a 17-bit register).

Purpose: RegUnit keeps track of the results of our multiplication in the add-shift algorithm. It stores the operands and performs the necessary shift. In the end, the product will be stored in Register AB(16-bit).



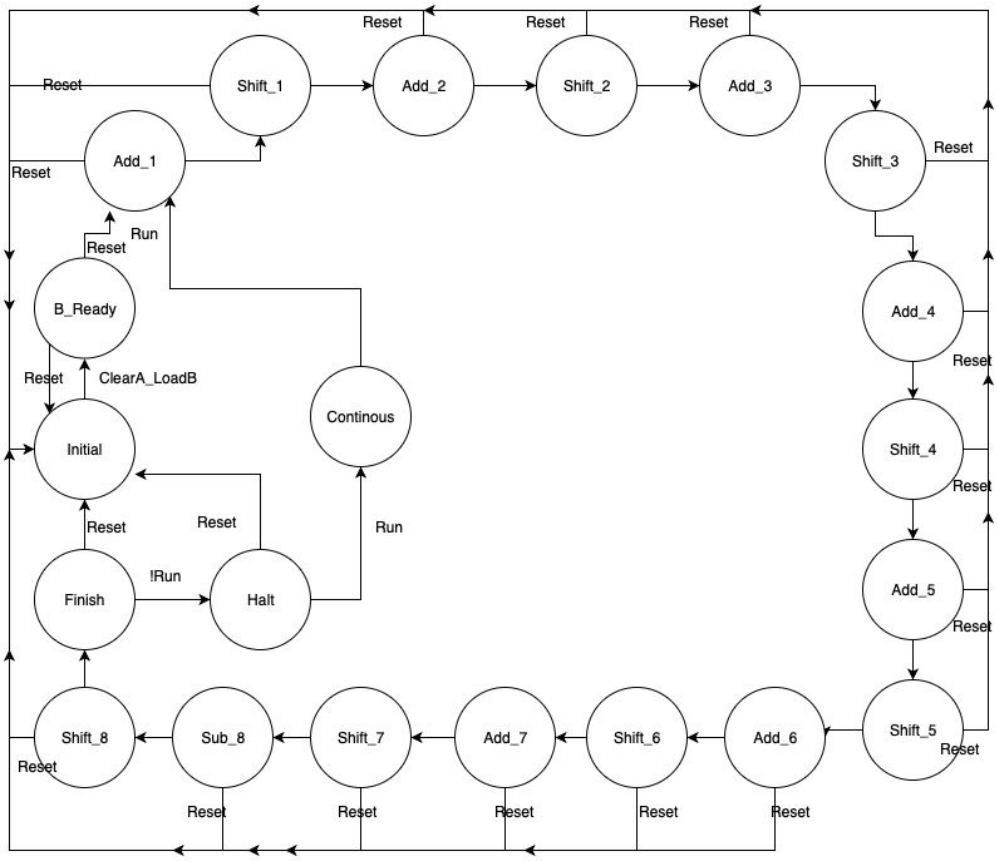
Module: ControlUnit.sv
Inputs: Reset, Clk, Run, M, LoadClear,
Outputs: Shift, Add, Op,Clr_Ld, Clr_XA
Description: The ControlUnit is basically a Moore Finite State machine with 22 different states. It contains 16 operation states(ADD, SHIFT, and SUB) and 6 additional states like RESET, INIT, HALT, etc. How we design and implement this FSM is shown in details in the following subsection.
Purpose: The ControlUnit, as its name suggests, controls the actions performed by the AdderUnit and RegUnit. More specifically, it controls when we should load the values, do the actual ADD and SUB, right shift the value, and halt the state machine.

Module: HexDrivert.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: The HexDriver is basically a decoder that maps a 4-bit value to a 7-bit one.
Purpose: HexDriver can display our binary values in decimal on the LED display of the FPGA board.

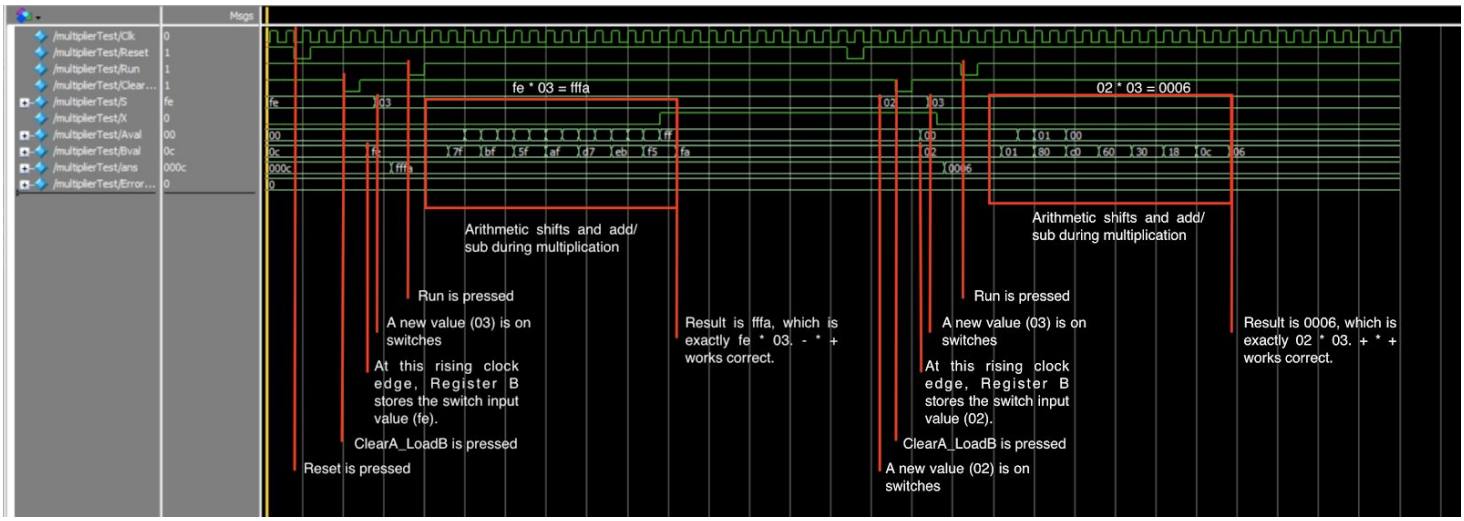


HexDriver Block Diagram

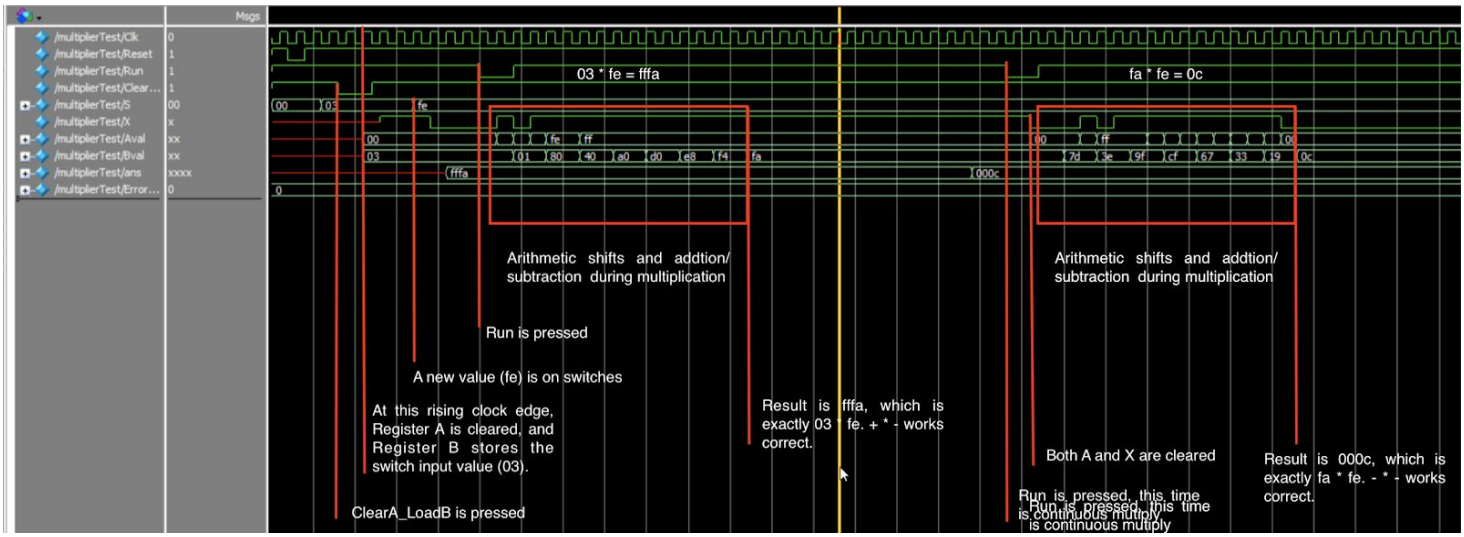
d. State Machine Diagram



Annotated pre-lab simulation waveforms



*03 * fe = fffa (+ * -), fa*fe = 000c (-*-)* —— *3*-2 = -6, -6*-2 = 12*



*fe * 03 = fffa (- * +), 02*03 = 0006 (+*+)* ——*-2*3 = -6, 2*3 = 6*

Answers to post-lab questions

a. Design's statistics

LUT	110
DSP	0

Memory(BRAM)	0
Flip-Flop	39
Frequency	72.38 MHz
Static Power	98.51mW
Dynamic Power	2.12mW
Total Power	144.58mW

b. Purpose of the X register

The X register is mainly for Register A value's sign extension (when performing arithmetic right shift). The value of the X register is calculated at the same time with Register A, thus it is set every time when the addition/subtraction performs, and its value is the sum of most significant bit of Register A and the switch input with Register A calculation's carry-out. When we start a new cycle of calculation, pressing the ClearA_LoadB button or pressing the Run button for continuous multiplication, the data stored in the X register would be cleared.

c. Limitations of continuous multiplications

Our multiplier can only perform at most 8-bit * 8-bit = 16-bit calculation. In continuous multiplication, if the previous result exceeds the space of Register B (8-bit), which means Register A is containing one or more bits of the first operands in the next multiplication, hitting the Run button would clear the content in Register A, so these bits cannot be used in calculation anymore. Thus, the limitation of continuous multiplication is that the previous result needs to be less or equal to 8-bit, otherwise the result would not be valid.

d. Advantages and disadvantages compared to the pencil-and-paper method

The simple add-shift algorithm we use is very similar to the pencil-and-paper method of multiplication except the final step for 2's Complement numbers depends on the sign bit. Pencil-and-paper method is easier for humans to understand and conduct. However, for computers, add-shift algorithm is more feasible. The computation is done through a series of multiplications and shifts which could be completed very easily by digital computers. Given numbers being stored in computers in 2's complement binary, add-shift method is more straightforward. Besides Add-Shift, we also learn that there are other multiplication algorithms that could be performed efficiently by FPGA, such as Carry Save Adder Multiplier, Booth Multiplier, and Array Multiplier. Another naive and straightforward multiplication method is to repeatedly add the multiplicand and reduces the multiplier by '1' in each cycle. This method, though very easy to come up with, performs very poorly when the operands get large.

e. Possible Improvements

To improve the maximum frequency, we can utilize the Carry Select Adder instead of the conventional Carry Ripple Adder which we currently use. A 9 * 1-bit Carry Select Adder would significantly improve the speed of addition/subtraction. Since we would have a maximum 8 times of addition/subtraction, the increased speed of each addition/subtraction would reflect on the improvement of the overall multiplication speed and maximum operation frequency.

To reduce the gate count, we can redesign our state machine. Currently, we are using the Moore State machine. Our design is not optimized, and it has 22 different states. The excessive number of states and state transitions would lead to a large number in total gate count. If we implement the state machine in the Mealy State machine, which output depending both of its state and input, we can optimize all addition/subtraction states into one arithmetic state, as well as optimize all shift states into one shift state. This optimization can significantly reduce the number of states, which in turn would reduce the gate count.

Conclusion

Our multiplier works appropriately with all combinations of operands (++, +-, -+, --). Initially, our design of the state machine had a problem. As in a shift state, the control unit would determine the next state is add/sub or shift depending on the last bit (M) of shifted Register B. This was a design problem, because the next state had been determined before the register B was actually shifted. Thus the M value we used was not valid for determining the next state. To solve this problem, we redesign the state transition procedure, add/sub state would always followed by shift state, but in the add/sub state, one of the operands would depend on the last bit of the shifted Register B (M). If M is 0, the operand is also 0, so the add/sub state is this case did nothing to the value in Register A. Other parts of this lab are simple to us. We think the lab manual provided is really helpful to us, especially the provided detailed example, which helped us understand how the multiplier should work. The information provided in the lab manual is correct, and doesn’t need to be modified.