ECE 385

SP 2022

EXPERIMENT #4

Multiplier

Tianyu (Joe) Yu, Yulun (Ben) Wu

2/28/2022

Curtis Yu

Introduction:

In this lab, we designed and implemented a multiplier that can support single and consecutive multiplication with positive or negative multipliers represented in 2's complement binary number. The multipliers are typed into the FPGA program with 8 switches, and the result will be stored and displayed in 2 registers.
11000101

Prelab Question:

Calculate 1000101 *00000111 with method in lab manual

Function	X	A	B	M	Comments for next step
Clear A Load B	0	00000000	00000111	1	ADD
ADD	1	11000101	00000111	1	SHIFT
SHIFT	1	11100010	1 0000011	1	ADD
ADD	1	10100111	1 0000011	1	SHIFT
SHIFT	1	11010011	11 000001	1	ADD
ADD	1	10011000	11 000001	1	SHIFT
SHIFT	1	11001100	011 00000	0	SHIFT
SHIFT	1	11100110	0011 0000	0	SHIFT
SHIFT	1	11110011	00011 000	0	SHIFT
SHIFT	1	11111001	100011 00	0	SHIFT
SHIFT	1	11111100	1100011 0	0	SHIFT
SHIFT	1	11111110	01100011	1	8th shift done

Table 1. Calculation Process of Pre-lab Question

Description and Diagram of Multiplier Circuit

Summary of Operation

Step 1: Connect the FPGA to the computer and initialize the program.

Step 2: Choose the first multiplier, a 2's complement binary number, and flip the corresponding digits with the Switch 0 to 7 on the FPGA board.

Step 3: Click the Key 0 button, which functions as a Clear-A-Set-B. Now the values on the switches will be loaded to register B and displayed as a 2 digits hex value on FPGA.

Step 4: Choose the second multiplier, and express it as 2's complement binary number on the switches.

Step 5: Click the Execute button, now the FPGA board will use the designed program to multiply the number within register B and the number currently expressed by switches. The result will be displayed as a 4 digits hex value on the FPGA board.

Step 6: If consecutive calculations are needed, the user will now flip the switches to write another multiplier, then click the Execute button. The result will be the product of the first calculation's result and the value expressed on the switch, and it will be displayed again as 4 digits hex value and. If no longer execution is needed, clicking the Reset button will clear the Register A and load B with a new value as a multiplier in the next round of calculation.

Top Level Block Diagram

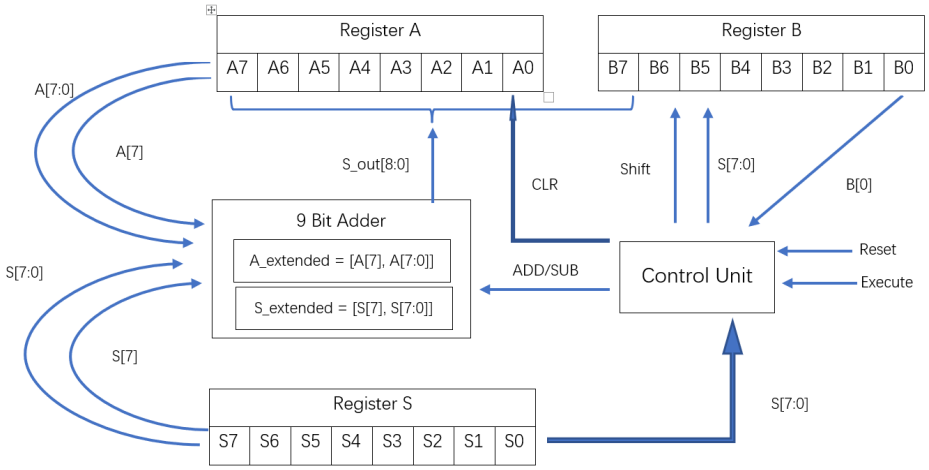

Figure 2. Top Level Operation Block Diagram

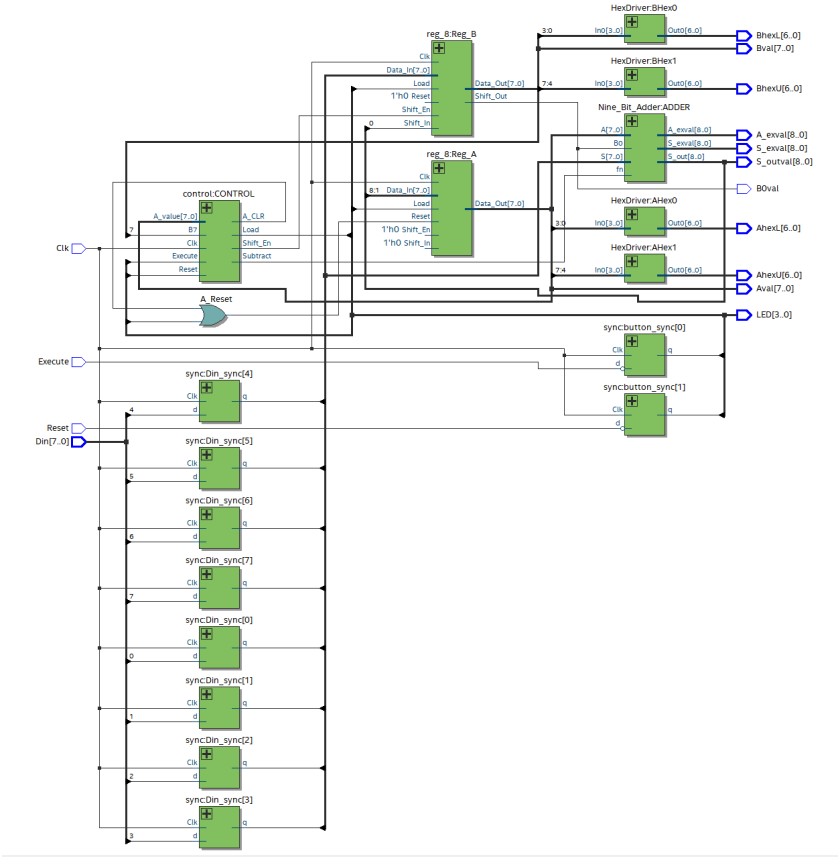

Figure 3. Generated RTL Diagram

Add Text for 3.b

Written Description of .sv Modules

1. **MODULE: full_adder (in ADDER9.sv)**

   Inputs: x,y,z
   Outputs: s, c
   Description: This module is a basic full adder we designed in previous class, adding x and y with z as carry in bit. The output s is the result of this bit, and c is the carry out bit.
   Purpose: This is a sub-module for our 9 bit adder, which is the central computational unit for this lab.

2. **MODULE: ADDER9.sv**

   Inputs: [8:0] S_ex, [8:0] A_ex, fn
   Outputs: [8:0] S_out,
   Description: This module works as a 9 bit ripple carry adder. It takes input from sign-extended S (the first multiplier) and A (the current value in register A, with sign-extended bit X as the most significant bit), with carry in bit fn, which changes value in the last shift-add state of our logic design. The fn value will be come 1 when we need to do subtraction operation, and A_ex value will go through XOR gate in the higher level module (9_bit_adder) before entering ADDER9 module.
   Purpose: This is still a higher level submodule for the computational unit, which completes the task of 9-bit ripple carry adder.

3. **MODULE: 9_bit_adder.sv**

   Inputs: [7:0] S, [7:0] A, fn, B0
   Outputs: [8:0] S_out,
   Description: This is the major computational unit of our project. It first takes 8 bit values S and A, and it then extends them within the module to local variables S_ex0 and A_ex. In order to fulfill subtraction operation in the correct time, there will be two local variables, B0_array and inverter, each of which consists of 8 bits of B0 value, the last bit of B register, and fn, the function selection signal received from the control unit. S_ex0 will become S_ex by first AND with B0_array, and XOR with the inverter next, and it will be sign extended to 9 bits at last. The purpose is to add 0 to A when B0 is 0, and do subtraction when fn is 1. Then S_ex, A_ex, and fn will be feeded into ADDER9 module, to produce the output S_out.

Purpose: This is the major computational unit, it serves to provide the result of add or subtraction in each calculation shift/add state of our design.

4. **MODULE: reg_8.sv**

Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] Data_In
Outputs: Shift_Out, [7:0] Data_Out
Description: This is a positive-edge triggered 8-bit register with synchronous input Reset Load, and Shift_En. The two types of load/shift is designed to fit the functionality of register A and B. When Load is high, data is parallel loaded from 8-bits Data_In, and when Shift_En is high, the last 7 bit will be shifted from previous bit, and the most significant bit will be loaded with value Shift_In. Data_Out will be the result of load or shift, and Shift_Out is the least significant bit of Data_Out.
Purpose: This module is used to create the registers with a special designed function (Shift_En, Shift_Out) that stores and does basic load and shift operations for register A and B in our design.

5. **MODULE: control.sv**

Inputs: Clk, Reset, Execute
Outputs: Shift_En, Load, Subtract, A_CLR
Description: This module is the control unit which works as a Moore machine with 11 states (from A to K), and each state has specific outputs of Shift_En, Load, Subtract, and A_CLR. Shift_En is specially designed for register B's input with the same name, which always lets register B read a new Shift_In bit and right shift other bits when a new digit of shift/add operation takes place. Load is for the register A, which updates value in every state of the cycle. Subtract is designed to tell the 9_bit_adder module the correct time to take subtraction operation instead of addition, and it only becomes one in eighth shift/add state. A_CLR is designed for giving register A a signal to clear its content in every next calculation cycle. It's very significant because it stabilizes the performance of consecutive calculations. The 11 states each correspond to different functions. State A is a reset/prepare state, allowing number input from switches to register B with Reset button. Once the Execute button is pressed, it will come to state B all the way to J. B will raise A_CLR value to clear A's value, and B-I will do a normal shift/add operation with output 1/1/0/0. State J will raise the Subtract signal for the subtraction operation in the last state of every calculation cycle (output: 1/1/1/0), and state K is a halt state, which will directly back to A once Execute is 0. It's simply preventing multiple calculation cycles if the user doesn't release the Execute button.

Purpose: This module is used to create the registers with a special designed function (Shift_En, Shift_Out) that stores and does basic load and shift operations for register A and B in our design.

6. **MODULE: Synchronizers.sv**

Inputs: Clk, d
Outputs: q
Description: This module aims to bring asynchronous signal, Execute and Reset in this project, into synchronous signal into the program. It applies "always_ff" to feed the received signal from button to variables of program in every clock edge.
Purpose: This module is used to create the registers with a special designed function (Shift_En, Shift_Out) that stores and does basic load and shift operations for register A and B in our design.

7. **MODULE: HexDriver.sv**

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: It takes four binary bits input from In0 and converts to 7 bits for Hex-display.

Purpose: This module converts 4bit binary numbers from registers into hexadecimal so we can display using LEDs.

8. **MODULE: Multiplier.sv**

Inputs: Clk, Reset, Execute, [7:0] Din
Outputs: [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU
Description: This is the top level module that calls all modules mentioned above. It serves as a main "breadboard" to connect the inputs and outputs of each module with each other and prepare the final output, which is going to be shown through Hex displayer which is driven by the HexDriver module.
Purpose: This module is the central module that gathers all sub-modules and connect them together, generating the output through HexDriver.

State Diagram for Control Unit

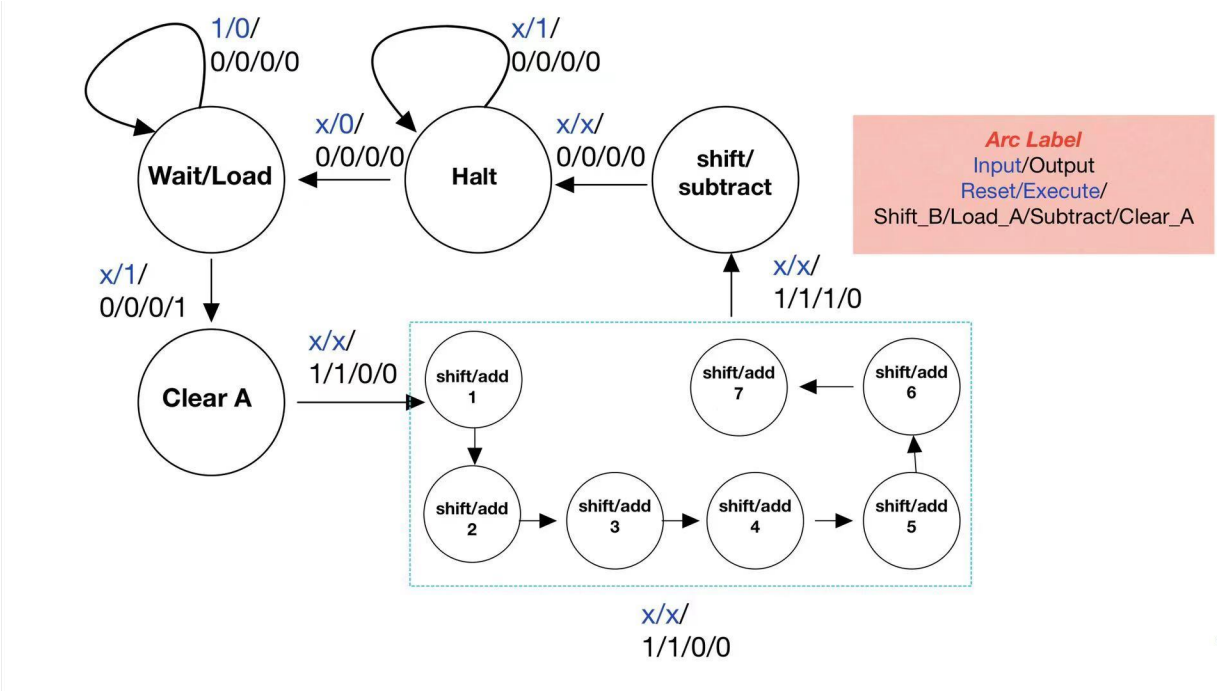

Figure 4. State Diagram for Control Unit

The state in the upper-left corner of the diagram is state A, and the following states are B, C … K. here are 11 states, and each state's inputs and outputs are labeled in the diagram.

Annotated Pre-lab Simulation Waveforms

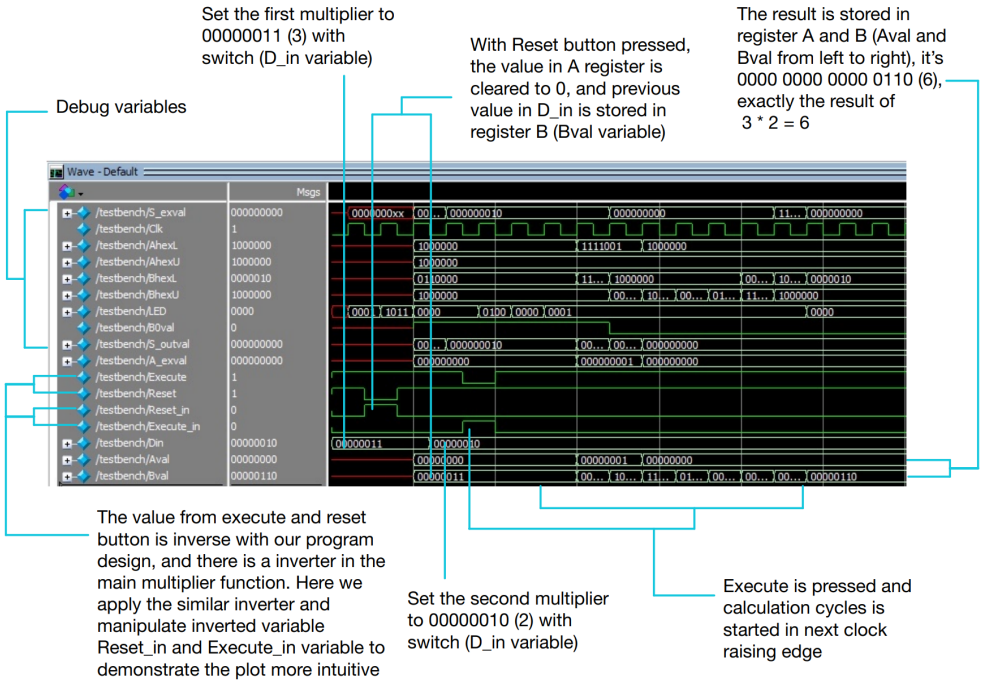

Figure 5. Simulation of "+ × +" Multiplication

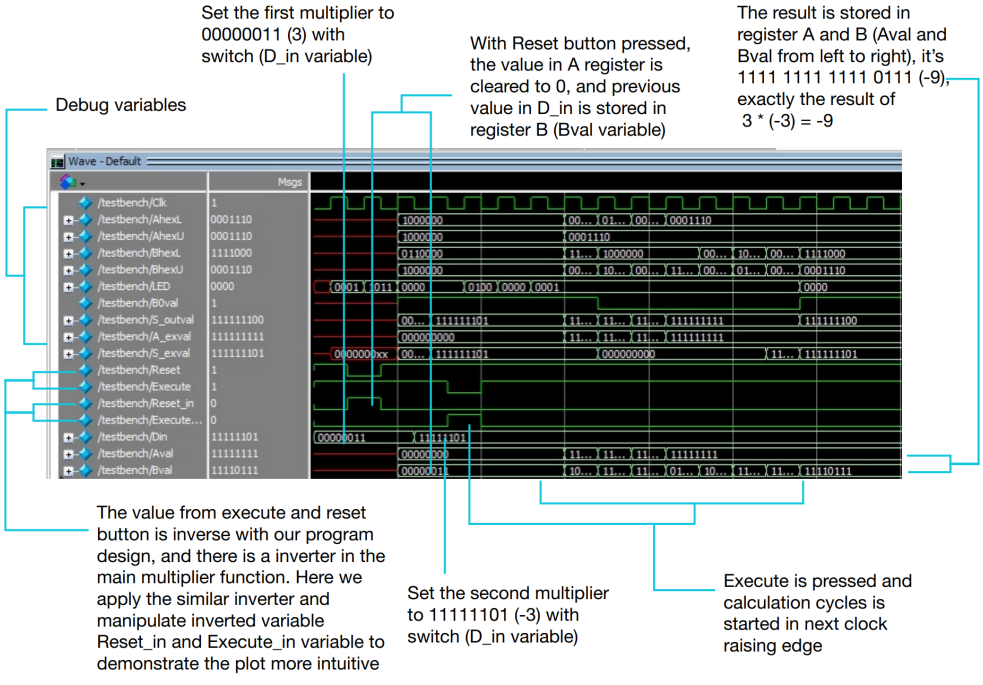

Figure 6. Simulation of "+ × -" Multiplication

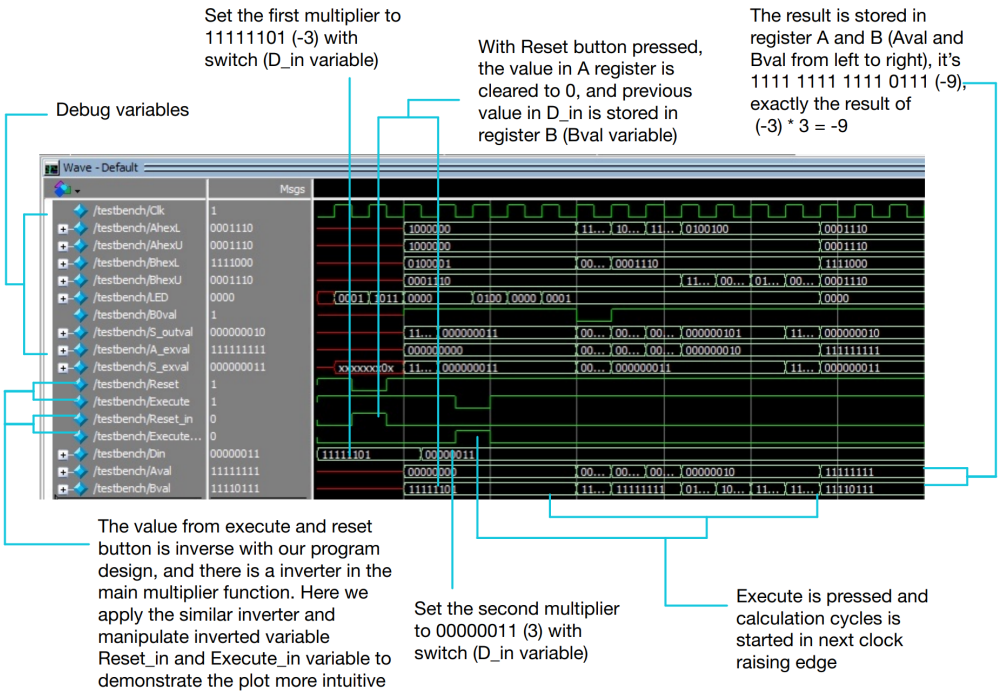

Figure 7. Simulation of "- × +" Multiplication

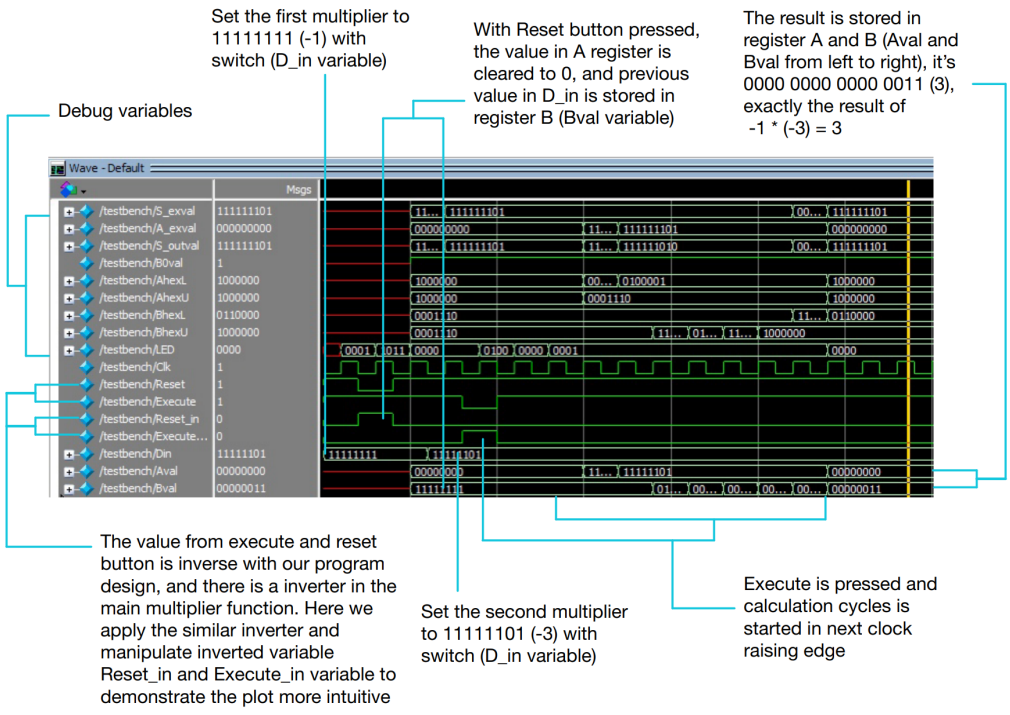

Figure 8. Simulation of "- × -" Multiplication

Answers to Post-lab Questions

Design Statistics Table

LUT	563
DSP	0
Memory (BRAM)	64,512/483,840 (13%)
Flip-Flop	591
Frequency	62.48 MHz
Static Power	90.00 mW
Dynamic Power	6.26 mW
Total Power	112.28 mW

Table 2. Post Lab Summary Table

Idea to optimize:

1. In this lab, we applied ripple-carry adder as the adder module throughout the multiplication process. It's more handy and stable for completing this lab demo. However, in order to increase maximum frequency and reduce power consumption, we can also utilize the look-ahead adder or select adder we applied in the previous lab, to reduce the gate delay or number of gates.
2. We created a separated state to clear register A for consecutive multiplication. There might be better design for this purpose that can reduce the gate number and increase the frequency.

Questions:

1. **What is the purpose of the X register? When does the X register get set/cleared?**
    a. The purpose of the X register is to sign-extend the value in register A, so that the sign of register A will not change after a right shift. The X register also prevents the loss of information in overflow situations. In our design, we didn't implement the X register, instead, we load the 9th to 6th bit from the adder directly from the adder. Since we don't have the X as a register, the value of X is update everytime we load a new set of value into register A

2. **What would happen if you used the carry out of an 8-bit adder instead of output of a 9-bit adder for X?**
    a. The carry out bit of an 8-bit adder does not represent the sign of the result, but it is just a detection for overflow. So if we use the carry out of an 8-bit adder instead of output of a 9-bit adder for X, the machine will calculate the wrong result when the 8th carry-out bit and register X have different values.
3. **What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?**
    a. Since our design is for 8 bits x 8 bits multiplier, we truncate the most significant 8 bits of the 16 bits result when doing continuous multiplications. Therefore, if the result from the previous multiply operation exceeds 8 bits, it will lose some bits in the next multiply cycle and continuous multiplications will not work in this case.
4. **What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?**
    a. One advantage is that the multiplication algorithm's design takes up less space since it requires 2 registers. Also, the multiplication algorithm's design can be faster by skipping the add state when M is 0.

Conclusion

In this lab, we successfully accomplished the design of the multiplier, and the functionality is completed throughout the demo process. I think it's REALLY IMPORTANT to inform the student that the signal from FPGA's buttons are inversely setted, that when we push down the button, the signal sending is 0, and when we release it, the signal becomes 1. This is not a significant lesson that students need to learn, but without the clarification, students may need to spend much more time on figuring out why the simulation and synchronizer don't act like they expect. Besides, I think it's important to re-emphasize that both shift and add operation needs to be done within a single clock cycle. I believe that this is the most restrictive requirement in this lab, but I think it's only emphasized in the lecture and lecture slides, so it's a good idea to clarify this requirement in the lab instruction file more. That's all, thanks!