ECE 385

Spring 2021

Experiment #4

8-bit Multiplier

Quanrui Bai, Jarod Partlo

ABJ

Xinbo Wu, Kevin Xia

Introduction

This experiment is intended to implement an 8-bit multiplier. It will take two 8-bit 2's complement integers as input and output a 16-bit result that will be shown on the FPGA board's hex display. There are two keys to control the circuit. Reset_Load_Clear will reset the registers and load the value from switches to register B as multiplicand, and then clear the value in register X and A. Run will start the computation cycle multiplying the number in register B by the multiplier indicated on the switches.

Prelab Questions

Function	X	A	B	M	Comments for next step
ClrA,LdB,Reset	0	00000000	00000111	1	M=1, add multiplicand to A
ADD	1	11000101	00000111	1	Shift XAB by one bit
SHIFT	1	11100010	10000011	1	Add S to A, Since M =1
ADD	1	10100111	10000011	1	Shift XAB by one bit
SHIFT	1	11010011	11000001	1	Add S to A, Since M =1
ADD	1	10011000	11000001	1	Shift XAB by one bit
SHIFT	1	11001100	01100000	0	Shift XAB by one bit
SHIFT	1	11100110	00110000	0	Shift XAB by one bit
SHIFT	1	11110011	00011000	0	Shift XAB by one bit
SHIFT	1	11111001	10001100	0	Shift XAB by one bit
SHIFT	1	11111100	11000110	0	Shift XAB by one bit
SHIFT	1	11111110	01100011	1	8th shift done. 16-bit product in AB

Table-1 Rework of the example for reversed multiplicand and multiplier

As the table shows, we reworked the example of the lab manual by swapping the multiplicand and multiplier, and the result was the same: -413.

Written description and diagrams of multiplier circuit

a. Summary of Operation

First, we reset, which also clears the registers X and A. Then the value indicated in the switches will be loaded into shift register B as multiplier (one of the operands). Another shift register, register A, will store the more significant portion of the result after each iteration. The second operand is controlled by what is on the switches after register B is loaded. Every shift register stores an 8-bit 2's complement integer.

During execution, the multiplier's computation cycle uses an "add and shift" algorithm. There will be a logic variable M storing the current least significant bit of register B. When the LSB of register B is 1, we will add the value of the multiplicand to register A once, such that M is the signal to determine whether we need to do addition (or in a later step, subtraction). Every time we add the multiplicand to register A, the data in registers X, A, and B will shift right by 1 bit (ordered as XAB). Register X is an additional bit that will hold the extra bit of A to allow 9-bit addition and maintain the bit to be shifted in. Register X is the sign-extension bit for addition since every operand is 2's complement. If we do not consider X, the most significant bit may be lost during the computation cycle, and then whether the value is negative and positive may be lost.

A critical part of the computation cycle is that when the process reaches its eighth shift iteration, we need to subtract rather than add if M is 1 to arrive at the correct sign of the final value.

b. Top Level Block Diagram

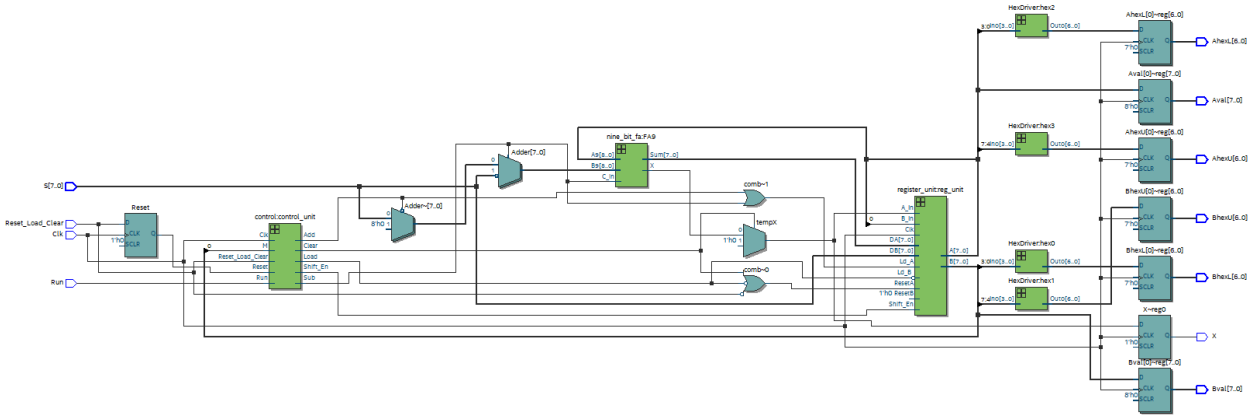

Figure -1 Top Level Block Diagram from Quartus Prime

c. Written Description of SystemVerilog Modules

Module: multiplier.sv
Inputs: [7:0] S, Clk, Reset_Load_Clear, Run
Outputs: [7:0] Aval, [7:0] Bval, [6:0]AhexU, [6:0]AhexL, [6:0]BhexU, [6:0]BhexL,X
Description: This is the top-level design entity which will initialize many registers.
Purpose: This will wire up our modules, and connect to LED display, switches, and two keys.

Module: full_adder.sv
Inputs: [3:0]A4, [3:0]B4, C_in
Outputs: [3:0] Sum, C_out
Description: This is the 4-bit adders with four 1-bit full adders. This adder will wait the Carry in from last adder, so the propagation time is very slow.
Purpose: This module will do the 4-bit addition, and then it will be composed of 9-bit adders for multiplication

Module: nine_bit_fa.sv
Inputs: [8:0] A9, [8:0] B9, C_in
Outputs: [8:0] Sum, X
Description: This is the 9-bit adder which is composed of two 4-bit adders and one 1-bit full adder. The nineth bit will be stored in register X.
Purpose: This module will do 9-bit addition in order to get the result for the multiplication.

Module: reg_8.sv
Inputs: [7:0] D, Clk, Reset, Shift_In, Load, Shift_En
Outputs: [7:0] Data_Out, Shift_Out
Description: This module is for shift register.
Purpose: This module is to accept the shift enable, shift in, and reset logic to control the register whether to shift, load, or reset.

Module: register_unit.sv
Inputs: [7:0] DA, [7:0]DB, Clk, ResetA, ResetB, A_In, B_In, Ld_A, Ld_B, Shift_En,
Outputs: [7:0] A, [7:0]B, A_out, B_out
Description: This module is composed of two reg_8 module to do the shift register.
Purpose: It will work as two 8-bit shift registers.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module has hardcoded values that convert 4bit binary numbers from registers into hexadecimal
Purpose: It will support the display hexadecimal onto FPGA board.

Module: control.sv
Inputs: Clk, Reset, Run, Reset_Load_Clear, M
Outputs: Load, Shift_En, Add, Sub, Clear
Description: This module is to control the states of multiplier, which contains hold state, 8 shift states, and 8 add states.
Purpose: It will assign outputs based on the state, controlling the reset, load, and clear functionality.

d. State Diagram for Control Uni

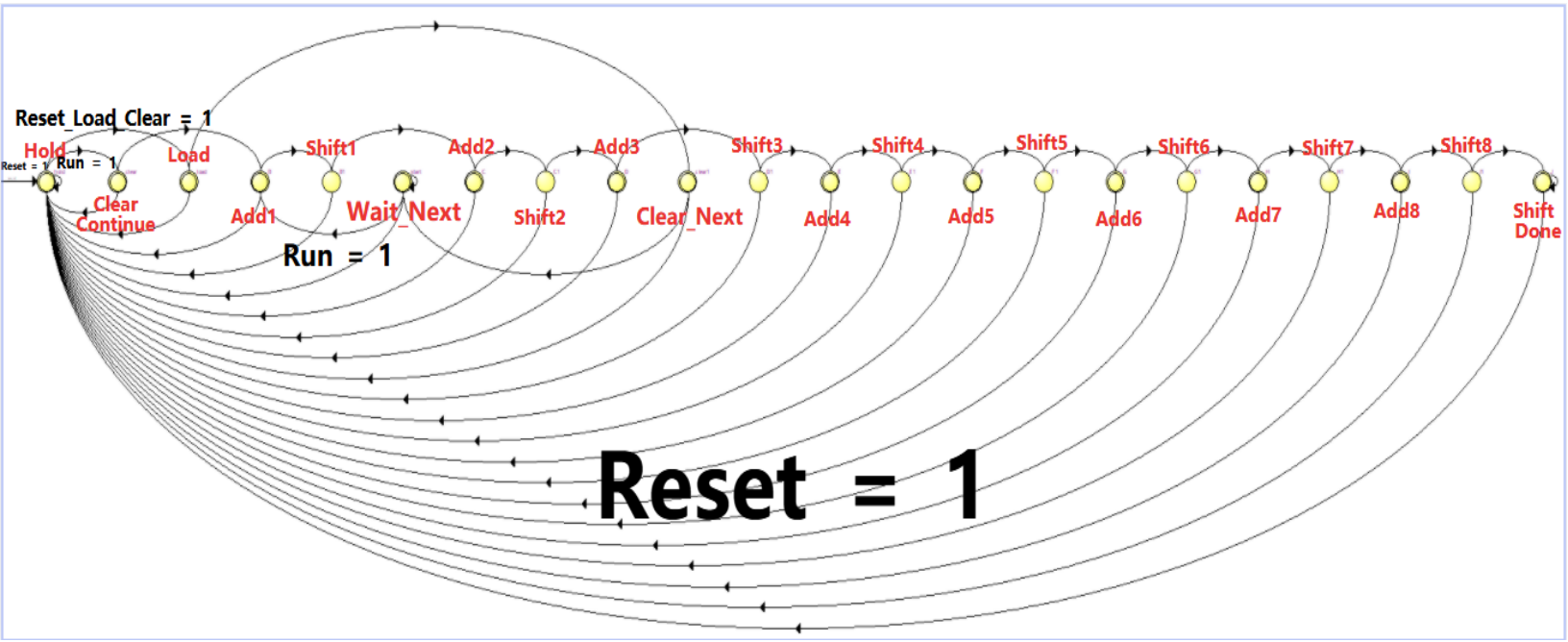

Figure- 2 State Diagram for Multiplier Control Unit

Red Texts(state)

Black Texts(input)

This is our state machine diagram. All red texts are the states' name, and the black texts are the input needed for next state. Other transitions without black texts will not care the input, and it will directly go to next state.

We have eight **Add** states, and eight **Shift** states, and each **Add** state will be followed by one **Shift** state without any inputs. After **Shift_Done**, we will go back to **Hold** state to display our result on Hex display. If we press **Run** =1at **Hold**, we will do consecutive multiplication. If we press **Reset_Load_Clear** = 1 at Hold, we will do the next multiplication, and wait for **Run** signal at **Wait_Next**.

For all transitions going to **Hold** state at the bottom of the diagram, they will process when the input of **Reset** = 1.

Annotated pre-lab simulation waveforms

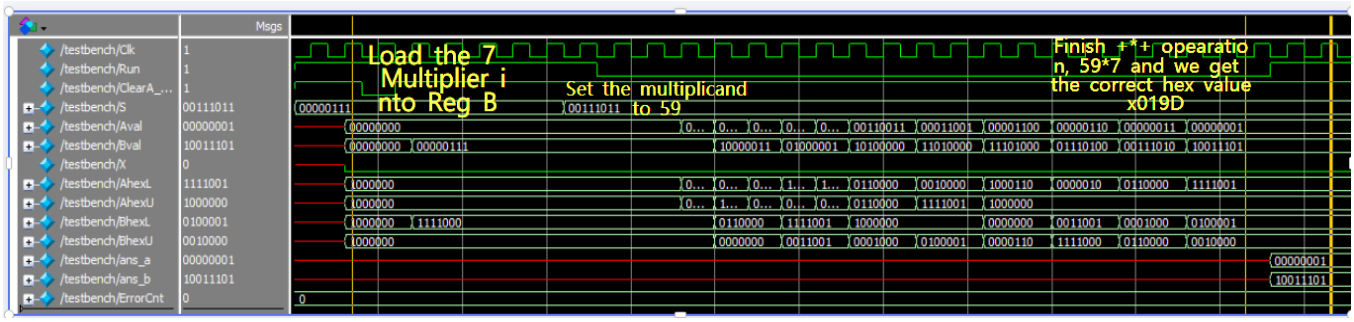

Figure -3 The Simulation Waveform of +*+ operation

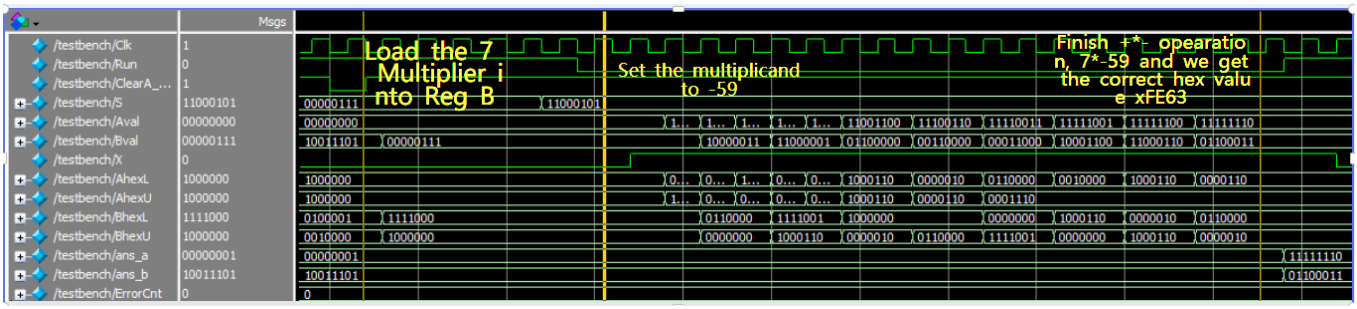

Figure -4 The Simulation Waveform of +*- operation

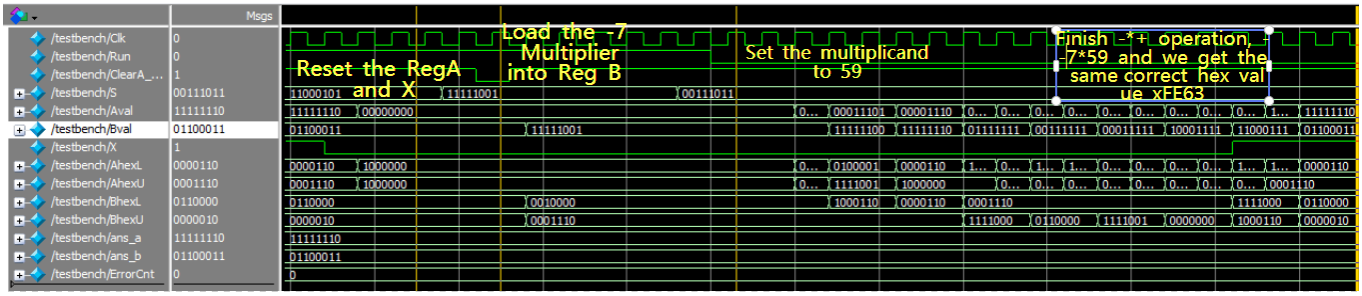

Figure -5 The Simulation Waveform of -*+ operation

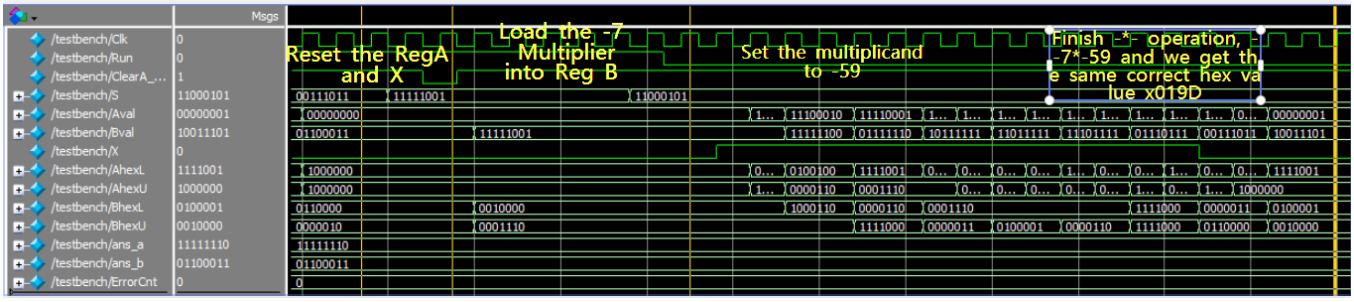

Figure -6 The Simulation Waveform of -*- operation

Post-Lab Questions

1.

LUT	109
DSP	0
Memory(BRAM)	0
Flip-Flop	84
Frequency(Mhz)	206.53
Static Power(mW)	89.94
Dynamic Power(mW)	0
Total Power(mW)	98.66

Table 2: Design Resources and Statistics

To optimize our design, I think the most efficient change we can do is to decrease the LUT or Flip-Flog by improving our control unit. In our design, we build the state diagram for multiplier by eight add states and eight shift states (totally 16). Actually, we can combine these states into eight states, and use a counter to specify the eighth shift, so we can easily determine which shift needs to check the M logic parameter to do addition/subtraction. Another way is to use more efficient adders, such as Carry Look-Ahead Adder or Carry Select Adder.

2.

Q1: The purpose of the X register is to hold onto the last carry-out bit, as it will be needed to shift into the most significant bit of the A register. Register X is essentially a sign-extension of the current value as values are stored as 2's complement. The X register gets cleared with each reset or load and can be set during an addition/subtraction operation in any of the addition/subtraction states.

Q2: If we only have 8-bit addition, we will not have an X register and the most significant bit will be lost. Then we cannot determine the proper sign of the result. Also, we may not able to retain the previous X. Without X, if the carry-out from the previous add iteration was already 1, the carry-out value would remain 1 when another 1 was carried out.

Q3: The limitations of this adder circuit include bit storage limitations. With enough continuous multiplications or with multiplication of large enough operands, the adder can eventually reach an overflow condition. The algorithm fails when doing continuous multiplications if the result of one of the additions to the XA registers exceeds 9 bits. Also, sometimes the continuous multiplication will lead the result and the multiplier sharing same shift register, so the result may be incorrect.

Q4: This multiplier operates much more quickly than the pencil-and-paper method and requires maintaining less data at once so it will use fewer registers but is more complex. However, the pencil-and-paper method can handle arbitrarily large numbers by keeping track of the intermediate results of repeated additions.

Conclusion

In this lab, we implemented an 8-bit multiplier by using the add & shift algorithm. We included a sign-extension register X to specify whether the value was positive or negative and hold onto the bit that needed to be shifting into register A. We designed the state machine in the control unit to determine whether we need to do perform addition, subtraction, or a bitwise shift according to the value of the least significant bit of Register B. Our design could not find the correct result in some edge cases, such as 8-bit times 8-bit with overflow. This error may be because the design fails to consider storing some X values during the multiplication process. When overflow occurs, the value in the register X is incorrect.

We don’t recommend changes for this lab. The lab was very challenging but a valuable experience.