The University of Illinois at Urbana-Champaign

ECE 385

LAB 4

LAB REPORT

by Shubham Gupta & Devul Nahar (sg49 & danahar2)

2/25/2022

Introduction:

The goal of this lab is to create an 8-bit 2's complement multiplier circuit. The circuit takes two 8 bit 2's complement values from register B (multiplier) as well as the switches (multiplicand) and loads the final 16-bit output value into registers X (1-bit), A (8-bit), and B (8-bit). Note that register X also works as a sign extension to the value stored in register A as well as the final output itself. The control unit for this lab has 4 inputs: Run, ClearA_LoadB, Reset, and Clock. In its most simplified form, this circuit is supposed to load value from the switches into register B, whilst simultaneously clearing register A and X (using the ClearA_LoadB signal). Once you loaded register B, you set the switch to your multiplicand, and then you do addition based on the least significant bit of register B. Every cycle of addition, you shift registers A and B to the right. This entire circuit works synchronously (using the clock).

Pre-Lab Question:

Initial values: X = 0, A = 0000 0000, B = 0000 0111, and S = 1100 0101.

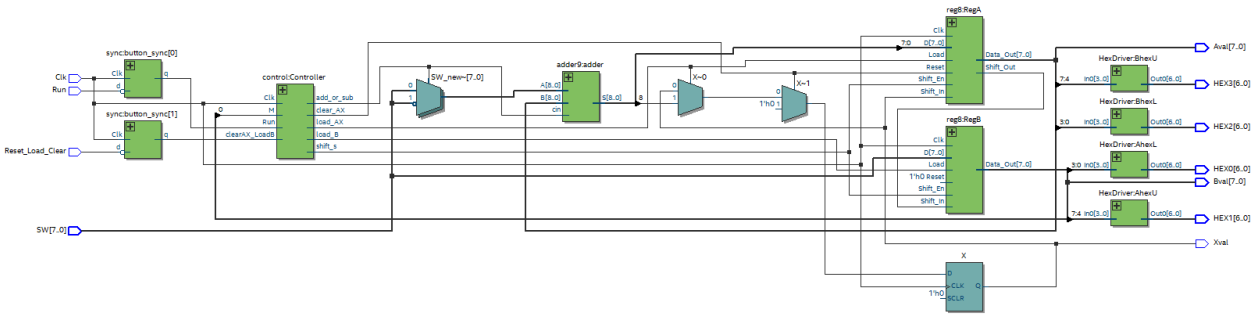
Function	X	A	B	M	Comments for the next step
Clear A, Load B, Reset	0	0000 0000	*00000111*	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	1100 0101	*00000111*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1110 0010	1 *0000011*	1	Add S to A since M = 1.
ADD	1	1010 0111	1 *0000011*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1101 0011	11 *000001*	1	Add S to A since M = 1.
ADD	1	1001 1000	11 *000001*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1100 1100	011 *00000*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1110 0110	0011 *0000*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 0011	0001 1*000*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1001	1000 11*00*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1100	1100 011*0*	0	Do not subtract S from A since M = 0. Shift XAB.
SHIFT	1	1111 1110	0110 0011	0	8th shift done. Stop. 16-bit Product in AB.

Description and Diagram of Multiplier:

a. **Summary of operation:** As soon as our multiplier is powered, the user must enter the data for register B with the help of the switches on the board. User then must press the Clear_Load_Reset button on the FPGA to load the values of switches in register B. As soon as the user presses Clear_Load_Reset, the hex display (lower 8 bits) shows the value that is loaded into register B. Then the user must enter the data for register A via the switches on the FPGA, upon which the user must press the Continue button. As soon as Continue is pressed, the FSM begins to shift through the bits in XAB and runs through the required states as per the state diagram in part d. For every clock cycle, the FSM generates the required control signals through which the multiplication algorithm is implemented. The control signals are mainly for the shift, add, or subtract action which is fed to various components like the full adder, shift registers, and register X. Main idea of the FSM is to carry out 8 shifts of registers AB. Before the shifts are carried out, we check for the least significant bit of register B (M). If M is 1 and it is not the last shift (8^{th} shift) we add S (from switches) to register A (initialized to 0s at the start of the multiplication) and set the sign extension bit X after the addition. Else, if M is 1 and it is the last shift (8^{th} shift) we subtract S from A and set X. If M is 0, we simply skip addition or subtraction and carry out the shift. Note, that after every shift we increment our counter to keep track of shifts and update M. Once the FSM halts, the answer is displayed as 16-bits in the lower four hex displays. The lower two displays show the content of register B and the two after it shows the contents of register A. For continuous multiplication, the new value should be set via the switches and the user must press Continue again to compute the answer. To clear all values and reset the multiplier, the

user must press Clear_Load_Reset button. This is the basic operation cycle of our multiplier.

b. Top Level Diagram:



c. Description of Modules:

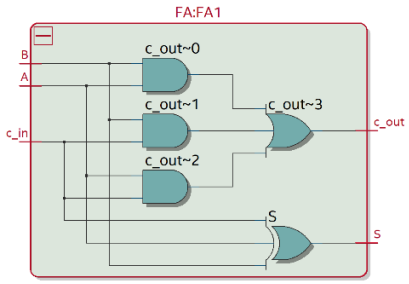
i. Module: adder1.sv

Inputs: A, B, c_in

Outputs: S, c_out

Description: It is the implementation of a 1-bit full adder, that takes bits A, B, and c_in to generate a sum S and c_out.

Purpose: It is used as the fundamental budling block in the 9-bit ripple carry adder implementation for this lab.



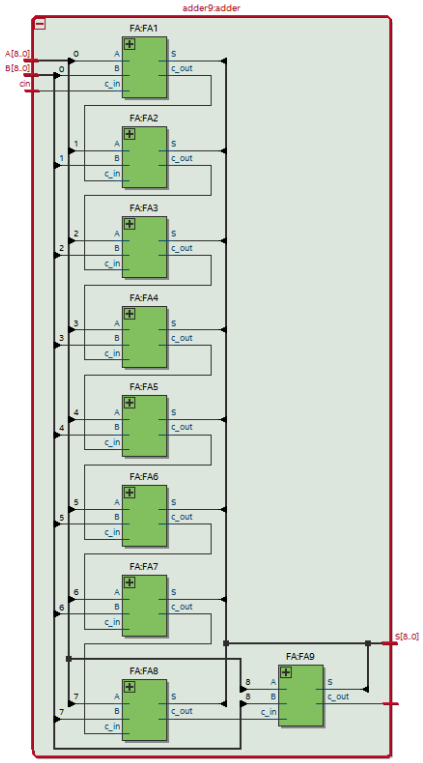
ii. Module: adder9.sv

Inputs: [8:0] A, B, cin

Outputs: [8:0] S, cout

Description: It is a 9-bit ripple carry adder, it is used to add values of A and S. It is also used as a 2's complement subtractor by inverting S and having the cin as 1. It is made up of nine 1-bit full adders.

Purpose: It is very important for the multiplier as it is used to add values of A and S. It is also used as a 2's complement subtractor by inverting S and having the cin as 1.



iii. **Module:** reg8.sv

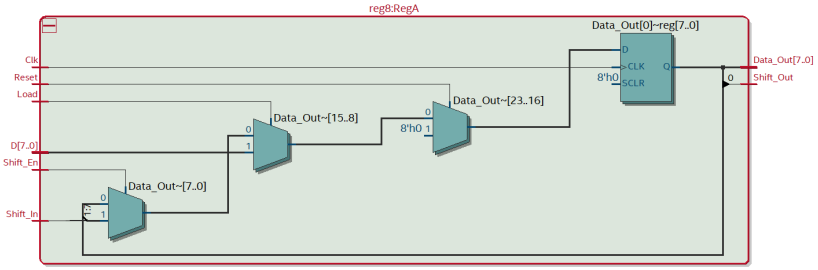
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D

Outputs: Shift_Out, [7:0] Data_Out

Description: It is a synchronous 8-bit shift register. The module is taken from previous labs.

Purpose: It is a crucial component of the multiplier as it stores 8-bit values for A

and B. It implements our ability to perform the right shift. With the help of data out, we can set bits like X and M.



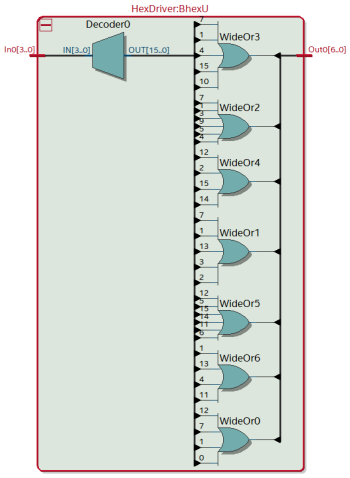
iv. **Module:** HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: We use a UNIQUE switch case to map the binary values to the hexadecimal display. Given to us from previous labs.

Purpose: It maps the binary values to the hexadecimal LED display. Used for displaying values stored in registers A and B. For user interface on the FPGA mainly.



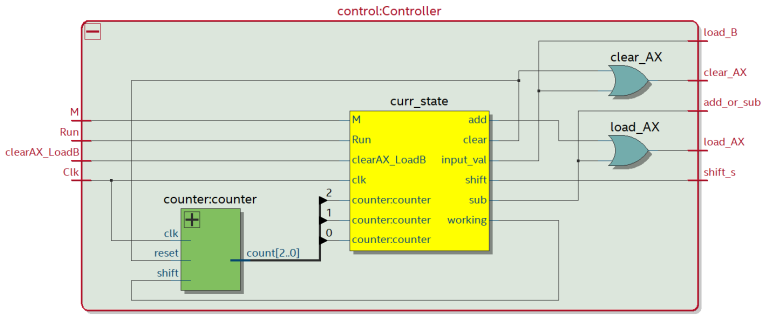
v. **Module:** control.sv

Inputs: Clk, clearAX_LoadB, Run, M

Outputs: shift_s, add_or_sub, clear_AX, load_B, load_AX

Description: This module implements our state machine and generates the control signals according to which shift, addition, or subtraction is carried out. We have used a mealy state machine and is implemented via a counter that updates after every shift. Based on the current count and state, the output signals and next state are decided. We have used case statement to implement this FSM.

Purpose: The control unit is the heart of the multiplier. It generates appropriate signals to carry out operations like shift, add, or subtract for the adder and shift registers. Moreover, based on clear signals, we clear XA, or load XA. This is crucial for our implementation to work correctly.



vi. **Module:** multiplier.sv

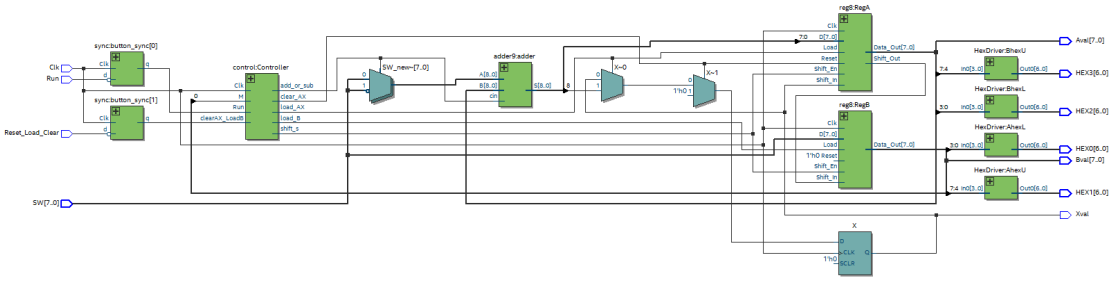
Inputs: Clk, Reset_Load_Clear, Run, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3

Outputs: Xval, [7:0] Aval, [7:0] Bval

Description: It is the top-level entity in our design. We have implemented all I/O connections in this module. The control unit is connected to the 8-bit shift registers, register X, and the 9-bit full adder. It also connects the outputs of the shift registers to the hex drivers to provide user interface. Implementation of

synchronizers is also done here.

Purpose: It is the top-level entity of our design. Its main purpose is to facilitate I/O connections between the control unit, full adder, hex drivers, and the shift registers. It acts as the medium for us to interact with the multiplier and initializes our FSM. It also instantiates all the required modules.



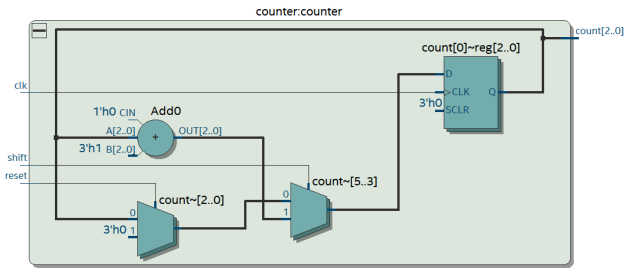
vii. **Module:** counter.sv

Inputs: clk, shift, reset

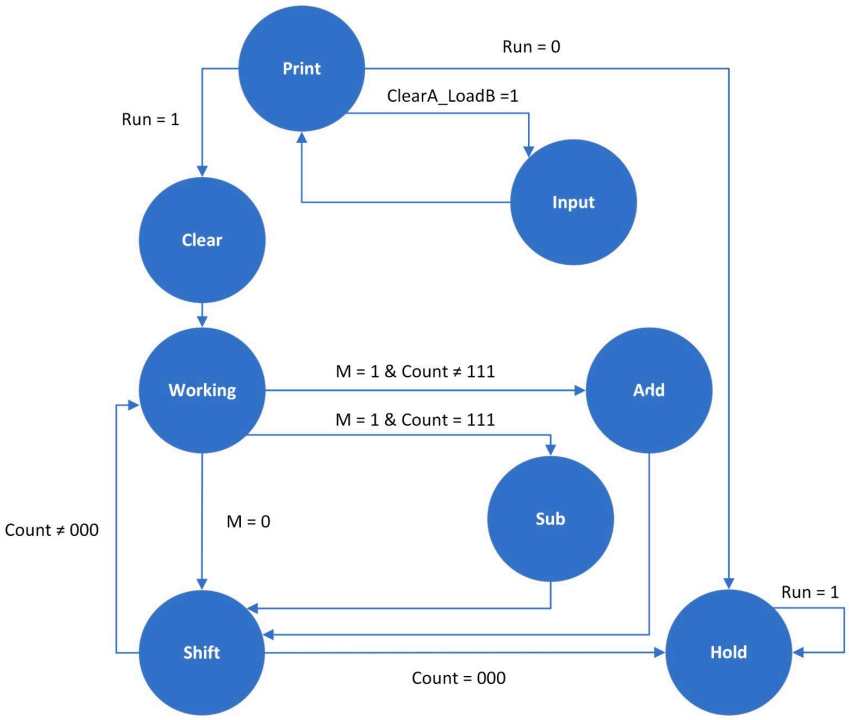
Outputs: [2:0] count

Description: It is a 3-bit synchronous counter that is implemented using always_ff statement. We step the counter by 1 when it is enabled. We reset it to 0 whenever reset signal is high.

Purpose: Its primary purpose is to keep track of number of shifts that have taken place, upon resetting to 0, we know that 8 shifts are performed. Counter also tells us when to perform subtraction based on the count.

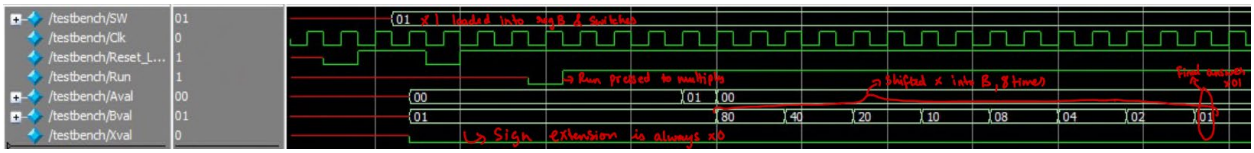


d. State Diagram for Control Unit:

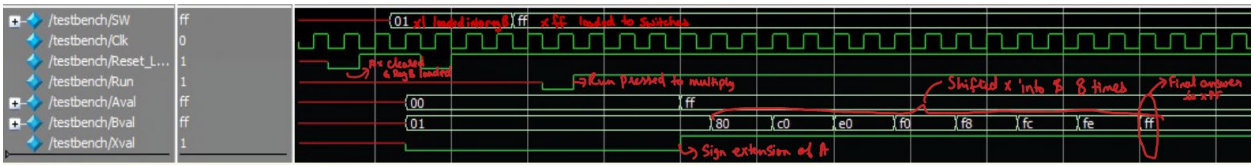


Annotated Waveform Simulations:

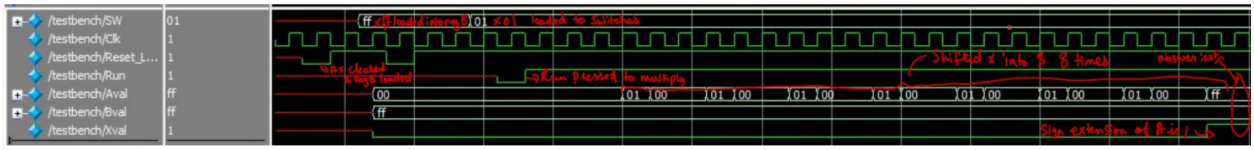
Positive * Positive:



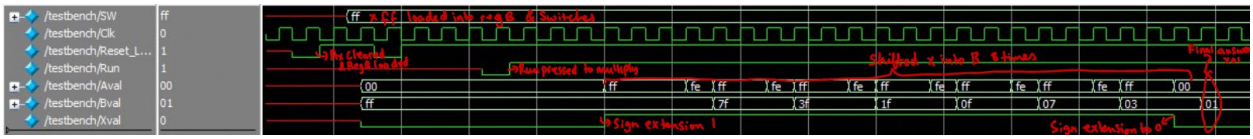
Positive * Negative:



Negative * Positive:



Negative * Negative:



Explanations:

As described in the introduction of this lab, in order to multiply two numbers, we first load values onto register B (by pressing Reset_Load_Clear). Note, that both the buttons Reset_Load_Clear and Run are active low, this means that when the signal goes low, that is the same as a logical 1. After register B has been loaded and registers A and X have been cleared, we set the switches which acts as the multiplicand. Finally, we press run which does the multiplication and loads it into registers X, A, and B (by shifting in X 8 times into A). Also, note that the sign extension bit X changes whenever we do the addition/subtraction and the answer changes sign. Note that the simulation shows the results x01 * x01 = x01, x01 * xFF = -FF, xFF * x01 = xFF, xFF * xFF = x01.

Post Lab Questions:

LUT	91
DSP	0
Memory (BRAM)	0
Flip-Flop	30
Frequency	69.97 MHz
Static Power	89.97 mW
Dynamic Power	1.52 mW
Total Power	103.74 mW

Possible Optimization:

One way in which we can optimize our design to run faster is by using a CLA adder instead of a carry ripple adder. As you can see from annotated simulation, the additions using requires the greatest number of clock cycles. Connecting back to the previous lab, where we saw that the CLA is the most time efficient, we would expect the overall efficiency of the multiplier to increase. However, by using a CLA the number of LUTs will increase as compared to the ripple carry adder, however this is a space vs time efficiency problem that is left to be solved by the designer as per the requirement of the application.

What is the purpose of the X register? When does the X register get set/cleared?

The purpose of X register is:

- To keep track of the sign, it is a sign extension bit.
- To keep track of which value gets shifted into most significant bit of register A.

X is cleared when Reset button is pressed and whenever a new multiplication is carried out, may it be continuous or non-continuous multiplication. X is set to most significant bit of A whenever an ADD or SUBTRACT is carried out.

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

In that case the sign extension bit would be incorrect. Since the carry out bit of an 8-bit adder would show the overflow and not actually the sign of the 8-bit number, it would not be 2's complement. The 9^{th} bit from the adder is important to correctly sign extend an 8-bit addition.

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

Before we carry out any multiplication cycle, we always clear XA and set them to 0. Our multiplier is meant to operate only on 8-bit operands. When we carry out continuous multiplications, the answer is stored in AB, but A is cleared for the next continuous multiplication. Therefore, if the answer of a multiplication exceeds 8-bit (register A holds a part of the answer) then the algorithm would fail because A would be cleared, and multiplication would be carried out between S and lower 8-bits (register B). But we want the answer by multiplying AB with S not just B with S. Hence the algorithm would fail.

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

Advantages:

- It is more space efficient as the pen-and-paper method implementation. Pen-and-paper implementation requires multiple registers to store the values of each line.
- 16-bit adder to add the values in the pen-and-paper method in contrast to 9-bit adder in our implementation.

Disadvantages:

- Difficult to understand, i.e., less intuitive design.
- Pen-and-paper implementation would give the answer in maximum 8 clock cycles (1 cycle to parallelly compute 16-bit answers in 8 registers + 7 cycles to sum up all the lines). Meanwhile our implementation would require 16 clock cycles in worst case (8 shifts + 7 adds + 1 subtract).

Conclusion:

In this lab we have implemented a 8 bit 2's complement multiplier that computes a 16 bit answer followed by a sign extension bit X. As soon as Continue is pressed, the FSM begins to shift through the bits in XAB and runs through the required states as per the state diagram in part d. For every clock cycle, the FSM generates the required control signals through which the multiplication algorithm is implemented. The control signals are mainly for the shift, add, or subtract action which are fed to various components like the full adder, shift registers and register X. Main idea of the FSM is to carry out 8 shifts of registers AB. Before the shifts are carried out, we check for the least significant bit of register B (M). If M is 1 and it is not the last shift (8^{th} shift) we add S (from switches) to register A (initialized to 0s at the start of the multiplication) and set the sign extension bit X after the addition. Else, if M is 1 and it is the last shift (8^{th} shift) we subtract S from A and set X. If M is 0, we simply skip addition or subtraction and carry out the shift. Note, that after every shift we increment our counter to keep track of shifts and update M. Once the FSM halts, the answer is displayed as 16-bits in the lower four hex displays. The lower two displays show the content of register B and the two after it show the contents of register A.

The lab resources were very helpful and self-explanatory. We did not have any issues while understanding and implementing the lab and design. The only part that was difficult to understand was making the test bench. Though the lab provides a 10-minute video explaining how a test bench works, we felt it lacked a lot of technicalities and was rushed. We believe that explaining the test bench in a more comprehensive manner would be of great help and help build basic understanding of simulation.