

ECE385

Fall 2023

Experiment #4

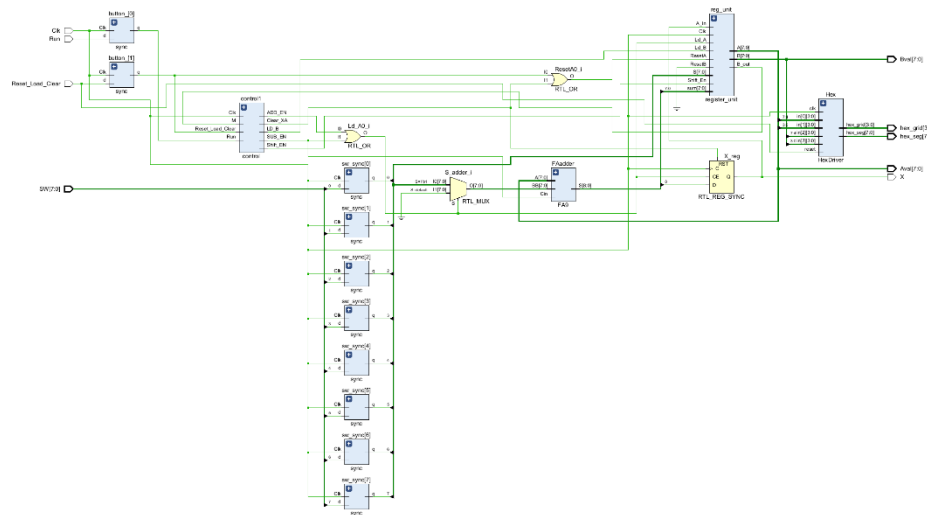
Lab 4

Haoyu Huang Eitan Tse
Lab section: ZY 10/Day & Time: 10.2
Your TA's Name: Yang Zhou

1.Introduction

We implement a two's-complement, 16-bit multiplier operating 8-bit multiplications. We use two 8-bit registers A and B which can shift bits, one-bit signal X, one-bit signal M and a nine-bit adder which can compute add and subtract computation. They can get the partial sum of the result and shift to get the result. Besides, the multiplier can do consecutive multiply in some cases. In general, the algorithm consists of seven adds, eight shifts, and a subtract step to compute the final product.

Top Level Block Diagram



2. Written Description of Modules

Module: toplevel (toplevel.sv)

input Clk, Reset_Load_Clear, Run, [7:0] SW,
output [7:0] hex_seg, [3:0] hex_grid, [7:0] Aval, Bval, X

Description:

Top level module for the lab. Module takes in the Switches, Clk, Reset, and Run signals and runs the multiplier circuit. The module instantiates reg_unit. The module then declares some temporary variables to hold values, this includes S_adder, Clr_Ld, Clr_XA, ADD, SUB, shift_en, M. The module then has an if statement. If we need to do sub or add, then S_adder will become value of SW, otherwise SW is equal to 0. We have a block to update X signal (most significant bit of A). We clear X when Clr_XA is 1 and assign it to sign of A when ADD | SUB is high. Control unit will control Clr_Ld, Clr_XA, ADD, SUB, shift_en. Clr_XA will reset the value of register A and X. ADD and SUB signal indicate which computation we will implement (or only shift). Shift_en indicates whether to shift register. For register unit, since we need to clear XA after every computation we need to Reset register A. While we can use Reset_Clear_Run button to Load B but not reset. Besides, register unit will also shift out the last bit of regB. Finally, we instantiate Hexdriver to display our output.

Purpose: Top Level for the lab. Used to instantiate all other modules and serve as the main circuit.

Module: reg_8 (reg_8.sv)

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

Output [7:0] Data_Out, Shift_out

Description: The module implements the 8-bit register in block diagram. The module is a positive edge clock-triggered 8-bit register. When the reset signal is high, the register is initialized with zeros. When the load signal is high, D [7:0] is loaded into the register.

Purpose: The 8-bit register store A and B value. It will be instantiated in register unit. The sum value will be loaded into register A.

Module: register_unit (register_unit.sv)

input Clk, ResetA, ResetB, A_In, Ld_A, Ld_B, Shift_En, [7:0] sum, S

output B_out, [7:0] A, [7:0] B

Description: The module takes in Clk, ResetA, ResetB, A_In, Ld_A, Ld_B, Shift_En, [7:0] sum, S. Register A will store the value of the sum. The register will store the value of switch and shift out signal M. The register will do shift operation when Shift_En is high.

Purpose: We decided to interface with the two registers A and B through a register unit module. We instantiate two 8-bit registers.

Module: FA9 (FA9.sv)

input [7:0] A, BB, Cin,

output [8:0] S, Cout

Description: The module can do 9 bits. It can calculate the sum of positive and negative numbers. It consists of two FA_4 modules and one fulladder module. FA_4 will calculate the ripple sum and fulladder will compute extending bit. When Cin is one, it means that we need to do subtracting. We derive 2's complement of B by following formula, $B = (\{BB[7], BB\} \wedge \{8\{Cin\}\})$.

Purpose: The module can do add and subtract computation. It essentially uses 2's complement to calculate sum of A and B

Module: FA_4 (fulladder.sv--lab3)

Inputs: [3:0] A, [3:0] B, Cin

Outputs: Cout, [3:0] sum

Description: This module is the 4-bit ripple adder that takes in the inputs A, B, and cin. The module will instantiate four 1-bit full adders arranged in a ripple-carry adder configuration. It will generate output sum and Cout.

Purpose: It is used to implement FA9. The purpose of this module is to build a four-bit adder for use.

Module: fulladder (fulladder.sv)

Inputs: A, B, Cin

Outputs: sum, Cout

Description: This module is the 1-bit ripple adder that takes in the inputs A, B, and Cin and adds them. Then it will generate the outputs sum and Cout. This is done through the logical expression $s = A \oplus B \oplus \text{Cin}$, and $\text{Cout} = (a \& b) \mid (a \& \text{c_in}) \mid (b \& \text{c_in})$. Below is a truth table that displays how these functions are just adding these inputs.

Purpose: The purpose of this module is to build a one-bit adder. It is used to count extend bit in FA9.

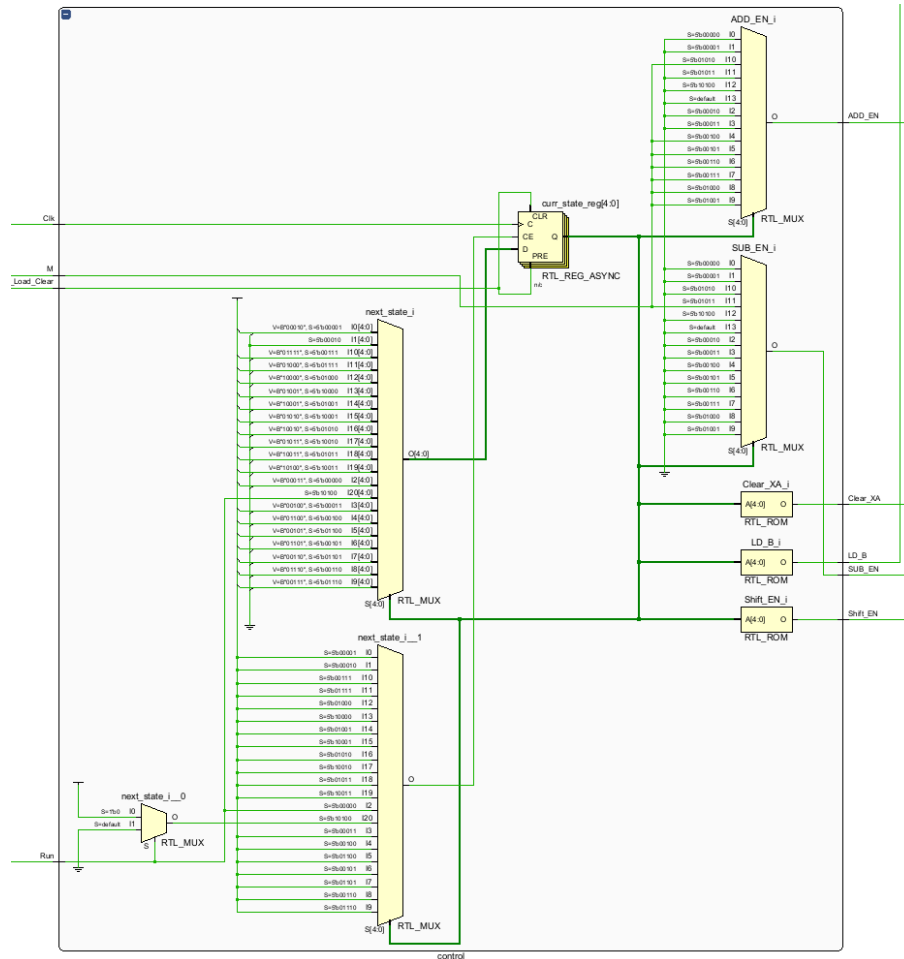
Module: control (control.sv)

input Clk, Reset_Load_Clear, Run, M

output Shift_EN, ADD_EN, SUB_EN, Clear_XA, LD_B

Description: We have to start state A1 which will clear XA and store value to register B. Then we halt and wait for run. If the Run signal is high, we will go to the computing state. ADD_EN will depend on the last bit of B (M). We have formula $\text{ADD_EN} = M$. SUB_EN is always 0. If M is 0, we will not do add operation. Otherwise, we will use an adder to calculate it. Then we have shift state to shift one bit. We repeat the same procedure 7 times. When we reach the last bit, $\text{ADD_EN} = 0$. We have formula $\text{SUB_EN} = M$. Then we do shift and reach the halt state, which only returns to the halt state if $\text{Run} = 0$.

Purpose: The control module essentially has 21 states. Except start state and end state which need Reset_Load_Clear state, I divide the rest state to shift and sum. When we finish the whole cycle. If we want to do consecutive computation, it will not go back to the start state. However, it will go the reset state that only clear value of register A and X.



Module: Hexdriver(Hexdriver,sv)

Inputs: Clk, Reset, [3:0] in [4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: The module takes output bits of registers in the register unit. It will change 3-bits into Hex number and display on the FPGA. The input is array type (each 4-bits) which can take in 16 bits in total. It can convert upper 4bits and lower 4bits of register to each hex display respectively. Output is 8-bits hex_seg and 4-bits hex_grid. Hex_seg is used to display numbers in LED. Every LED has 7 lights. Number can be expressed in 7 bits in LED. For example, 0 can be expressed as 11000000. Hex_seg [7] is constant 1 in this case. Since FPGA has multiple HEX displays, hex_grid will instruct the board to use specific displays.

Purpose: The module is used to display Hex numbers on FPGA boards, allowing the user to see the contents of registers A and B.

Module: sync (Synchronizers.sv)

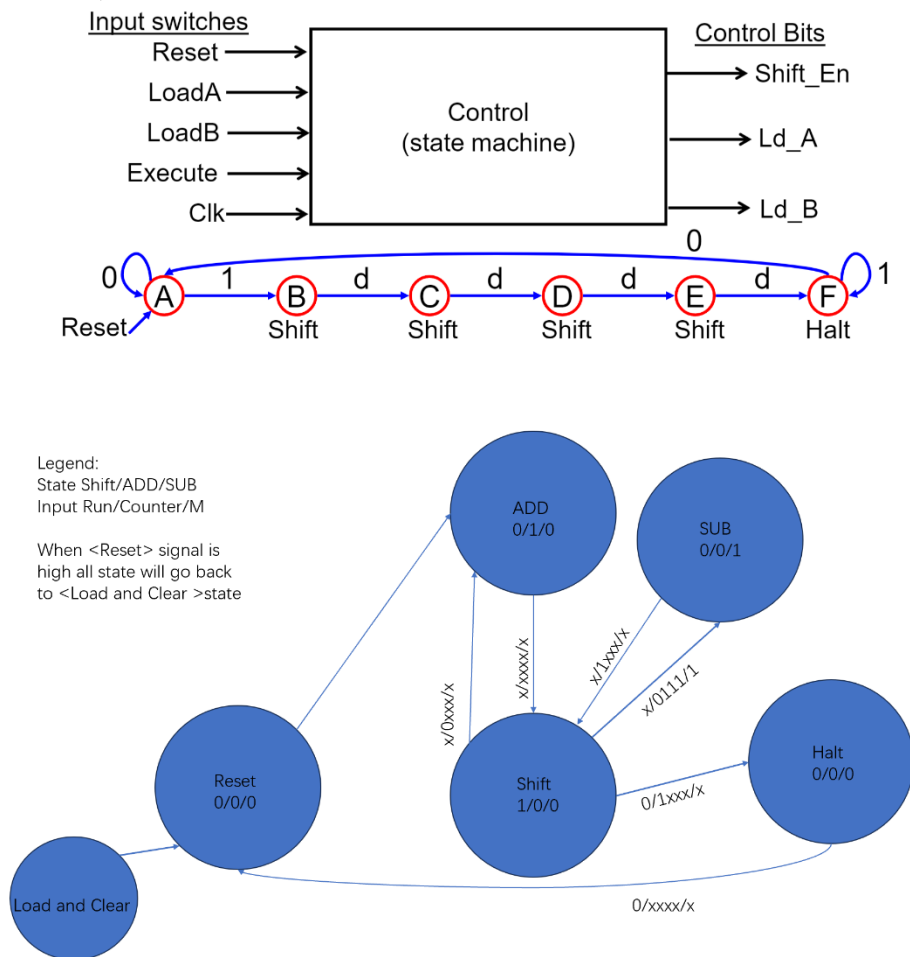
input Clk, d

output q

Description: The module takes in CLK and d. It will use always_ff block to assign d to q.

Purpose: These are synchronizers required for bringing asynchronous signals into the FPGA synchronizer with no reset (for switches/buttons)


**State Diagram for control unit:
(From lecture)**



Our state diagram may be updated by skipping the add state when the M is high, which can reduce the total number of the state.

Answers to Pre-Lab Questions

*Pre-Lab Question -Rework the multiplication example on page 5.2 of the lab manual, as in compute $11000101 * 00000111$ in a table like the example. Note that the order of the multiplicand and multiplier are reversed from the example.*



Function	X	A	B	M	Next Step
Reset	0	0000 0000	0000 0111	1	ADD SW to A
ADD	1	1100 0101	0000 0111	1	Shift XAB
Shift	1	1110 0010	1000 0011	1	ADD SW to A
ADD	1	1010 0111	1000 0011	1	Shift XAB
Shift	1	1101 0011	1100 0001	1	ADD SW to A
ADD	1	1001 1000	1100 0001	1	Shift XAB
Shift	1	1100 1100	0110 0000	0	Shift 5 times: the last 5 bits are 0
Shift 5 times	1	1111 1110	0110 0011	1	No need to ADD

Written Description: Summary of Operation Explain in words how operands are loaded, how the multiplier computes its result, how the result is stored, etc.

The multiplier works by adding the multiplicand, multiplier number of times. Besides we need to shift register to get the correct result. We are essentially adding A and B. The procedure is first adding the SW into A whenever the last bit of B (M) is 1 which indicate 1 times A. Then we need to store the sign of A into X. The module then does right shift XAB and then looks at the M again, and if it's 1, then it adds SW into A and shifts again. This is repeated for the first 7 bits of B. When it comes to the 8th shift, if M is 1, then the circuit subtracts SW from A. This is done by first inverting all the bits of SW and then adding 1. I have implemented this algorithm in the adder. So, we can do subtract computation directly. We load it to A to get the result which is stored in AB. Since B has shifted 8 times, B value has been shifted out.

Register X has two main uses. Since we extend A and B to 9 bits when we do calculation. X can store the sign of A, and conserve the sign of

$A+SW$. It can also solve the problem of overflow. If we don't have such a bit, two positive numbers may cause negative numbers. This solves the overflow problem of the circuit. In general, the circuit works by looking at the M , and adding (or ignore) the SW into A and then right shifting XAB , and when reached the last bit, it subtracts if the M is 1. We use 2's complement to add them.

Consecutive computation. The multiplier can do consecutive computation if the result has same value if it is truncated to 8 bits. For example (-1 and 1). Since we will do 2's complement multiplication, -1 and 1 remain the same. I implement the consecutive computation by clear register A and X every computation cycle. The previous value will be stored in register B , and we can do the next computation by changing the value of SW .

3. Annotated Simulation Waveforms

Multiply 7 and -59 together



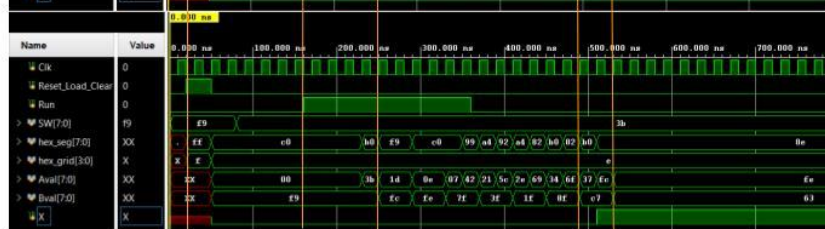
Multiply -7 and -59 together



Multiply 7 and 59 together



Multiply -7 and 59 together



Clear XA, load B Start operation with multiplier in switches Begin shifting/shifting and adding Final shift, set X and decide whether to ADD/SUB for final operation Operation complete, 16-bit result in AB

4. Answers to Post-Lab Questions

Resource Utilization

LUT	69
DSP	0
Memory (BRAM)	0
Flip-Flop	73
Latches	0
Frequency	176.991MHz
Static Power	0.070W
Dynamic Power	0.011W
Total Power	0.081W

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

The X register retains the MSB of the A+S operation. Typically it is only set after the final ADD operation is completed, where it is determined if the 8th bit of B is 0 or 1. If it is 1, it indicates a negative number and that the final operation should be a subtraction. Then, X can be set appropriately so the correct value will be shifted in on the final shift operation. On the other hand, if two very larger operands are added in A+S, then it can also retain the correct MSB bit for shifting in.

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

The MSB calculations would be incorrect. Considering the operation conducted is an add then shift or just a shift, it is very important to have the shift-in bit calculated correctly. While the immediate carryout of adding A and S together could be considered the MSB, it wouldn't be accurate as soon as the shift is conducted. Having the MSB be calculated by 9-bit adders and sign-extended inputs ensures accuracy and the correct value is shifted in.

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

It only works as long as the product from the prior multiplication can be truncated to an 8-bit value without changing its value. This means the value has to lie between -128 and 127. If it is outside of that, then the result of the next multiplication to be carried out is incorrect.

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

The pencil-and-paper method lets us see all the partial products and allows us to verify our work more easily. However, keeping track of all of these partial

products is unnecessary because its addition will always be correct, since keeping track of them means redundant hardware. The implemented multiplication algorithm skips that, only requiring a way to store the current multiplicand and thus being lighter on resource usage than it would be to store everything using the pen-and-paper method.

5. Conclusions

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

I encountered some problems with consecutive computation. I go to the reset state first. Then I add one more state to clear the value of X and register A before each calculation cycle.

I think the lab is pretty good. The only thing is that it is not compatible with lectures. The professor teaches how to implement two-always state later.

Overall, the lab is a great continuation of the last lab. We implement many modules including state machine. It is a good practice to use 2-always block to implement state machine. We also modified the ripple adder in the previous lab to make it calculate the sum of negative and positive numbers. It is also meaningful to go through the multiply algorithm. It is our first time implementing the whole project. It is good practice to consider all parts of the project and design the module.