ECE 385

Spring 2024

Experiment #4

Lab 4

Yuxuan Li   yuxuan43

Yanghonghui Chen   yc47

Section XC

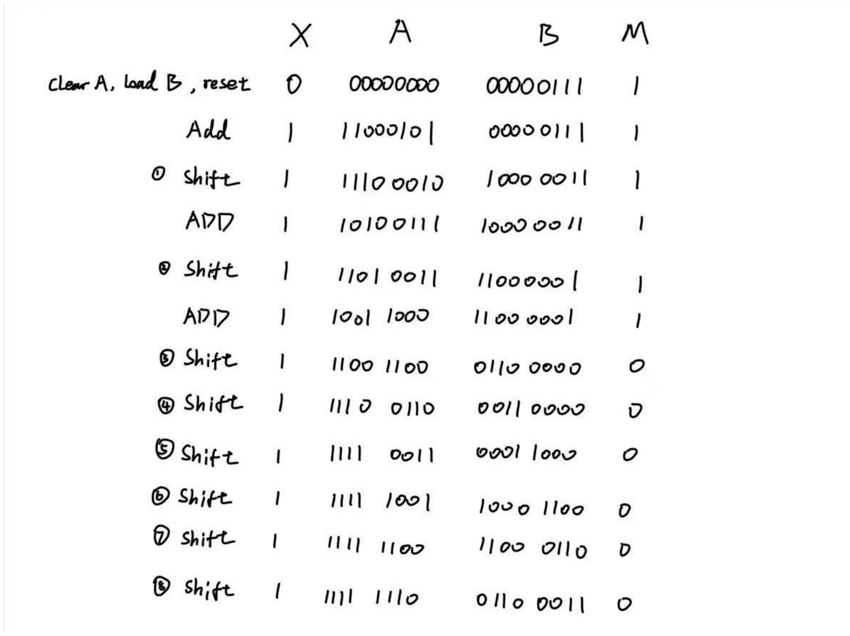
TA: Shixin Chen

1. Introduction

In this lab section, we implemented a 16-bit 2's complement multiplier that operates on 8-bit component's multiplication. Two multipliers A and B are loaded into two shift registers, and there's a signal X and a determinator signal M. We used a 9-bit adder to do add and subtract computation, step computation result of adding and shifting make up the partial sum and shifting to implement multiplication. The general process of 8-bit multiplication consists of seven possible add, eight shift and one possible subtraction.

2. Answer to Pre-lab Question

*Rework the multiplication example on page 4.2 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.*



	X	A	B	M
Clear A, Load B, reset	0	00000000	00000111	1
Add	1	11000101	00000111	1
① shift	1	1110 0010	1000 0011	1
ADD	1	10100111	1000 0011	1
② Shift	1	1101 0011	11000001	1
ADD	1	1001 1000	1100 0001	1
③ Shift	1	1100 1100	0110 0000	0
④ Shift	1	1110 0110	0011 0000	0
⑤ Shift	1	1111 0011	0001 1000	0
⑥ Shift	1	1111 1001	1000 1100	0
⑦ shift	1	1111 1100	1100 0110	0
⑧ shift	1	1111 1110	0110 0011	0

3. Summary of operation

The multiplier do multiplication by dividing it into serval different adding calculations of multiplier and multiplicand. Since both multiplier and multiplicand are 8-bit 2's complement binary numbers, the process can be seperated into two steps, adding and shifting. At the begginning of A * B computation, we first clear X and A signal, load B into a shift register, and let the signal M be the least significant bit of B. Then we add the value of A which is passed in by switches into the other register, while passing A's most significant bit into signal X.

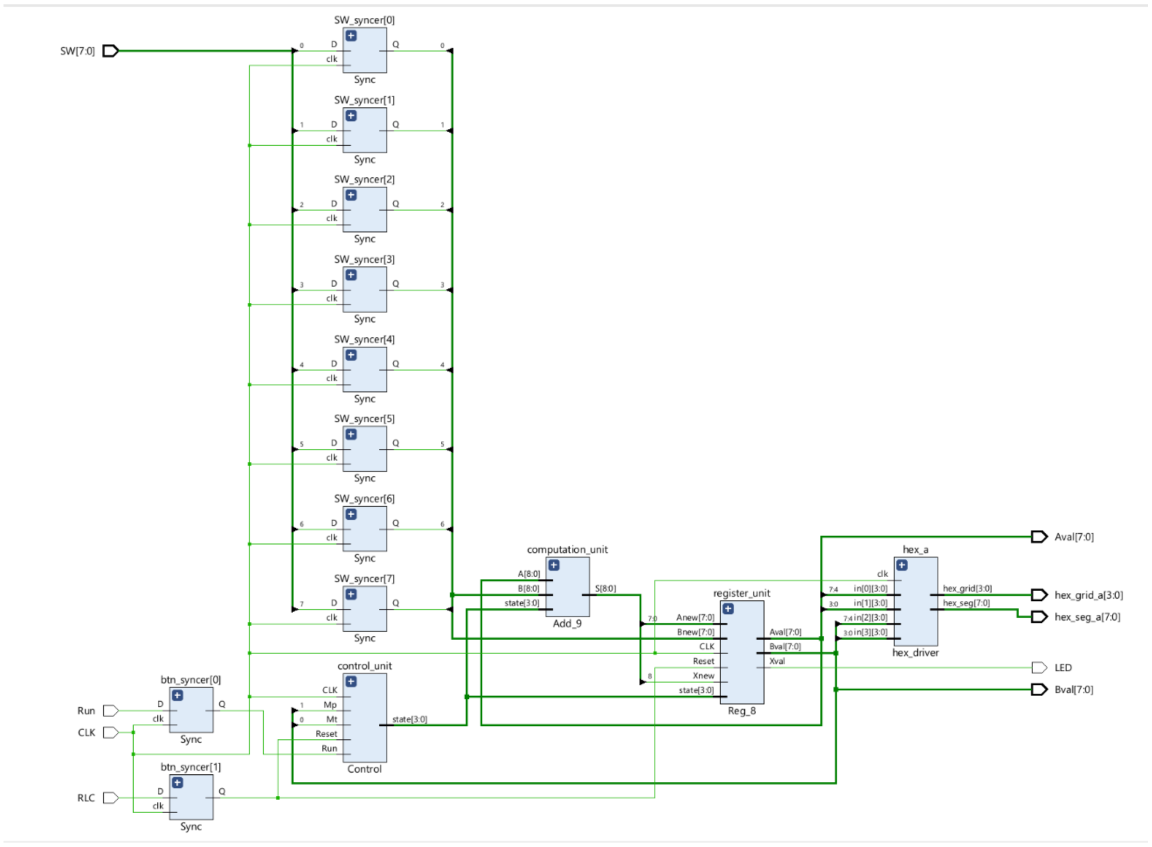
After these initializing steps, we begins calculation and use signal M to determine whether to do addition or not. We first do a shift operation to XAB, shifting X to A's msb and shifting A's lsb out into B's mst. This shifting process is done by shift register. Then we update the value of X and M. If signal M is 1, we do an extra addition calculation that add the switches' value into A, thus updating

A. If signal M is 0, then we only do shifting of XAB but do not add any component. We repeat these two steps(shifting XAB and possibly adding SW and A) until the 8th shift, during which we should use M to judge whether this 2's complement calculation's result should be negative number. If M is 1 this time, we should subtract value of switches from value in register A, then do the shifting last time for XAB. If M is 0, then just simply do shifting. After 8 times of shifting, the final result in AB(seem as a whole 16-bit value) will be the result of this 2's complement calculation.

Need to mention that, we use 9-bit adder for addition process since we extend A and B to 9-bit when doing calculation, considering overflow could happens. In this case, X is able to store the sign of A and conserve the sign of A adding SW, while solving the problem of overflow.

The multiplier can also do consecutive computation, while clearing XA and updatting the new value in A register and X. Previoously calculated value will remains unchanged if it doesn't extend 8-bit, and by changing SW that passsed in we can decide the next value of A. But if previous value can't be just stored in B, part that stored in A will lost and causing a wrong result for new operation.

4. Top Level RTL Diagram



5. Written Description of SystemVerilog Modules

(1) module full_adder

input: A, B, Cin
output: S, Cout
Description: The 1-bit full adder that takes in the inputs A, B, and Cin and adds them, returning sum result s and Cout signal c. This is done through the logical expression $s = A \oplus B \oplus Cin$, and Cout = (A&B) | (A&Cin) | (B&Cin). This module is based on fulladder module from previous lab, in which we used it to implement different types of adders.

(2) module Adder4
input: [3:0] A, B, Cin
output [3:0] S, Cout
Description: The 4-bit ripple adder implemented by serial design of 1-bit full adders. Take two 4-bit value input and carry-in bit, take 4-bit value S and carry-out bit as output. This module is copied from Adder4 module from previous lab.

(3) module Add_9
input: [8:0] A, B; [3:0] state
output: [8:0] S;
Description: A modified version 9-bit Adder, which can not only do adding, but also can use the input 'state' signal to determine whether to do adding or subtracting operation. In this module, we used to Adder4 units and a fulladder that connected in series to perform 9-bit operation. As for subtracting, we uses LSB of state to indicate control signal M, and use B XOR {9{state[0]}} to gain the value that should be passed into calculation. Return the result in S.

(4) module Reg_8
input: [7:0] Anew, Bnew; [3:0] state; Xnew, Reset, CLK,
output: [7:0] Aval, Bval; Xval
Description: The 8-bit shift register unit that do shifting and loading new value into XAB based on control signal [3:0] state and Reset. If Reset is 1, then clear X and A manually while loading Bnew into B. Otherwise, decisions are made based on state. If state is 1000, it will clear X and A while keeping B unchanged. If state is 0100, it will shift XAB for one bit. If state is 0010, it will do add/subtract operation and load the result from Add_9 into register.

(5) module Control
input: Reset, run, CLK; Mp, Mt
output: [3:0] state
Description: The finite state machine that determine state transform, and output control signal state for doing operation in register and adder units. We designed 19 states in control unit. If the Reset signal is 1, it should goes to Halt state. State machine transfer when receiving Run signal, it should go from Halt state to Clear state, in which the register will clear A and load B. Next state of Clear in determined by signal Mt, which should be LSB of B. State machine will go through branches of Shift and Add after this. There're seven adding states and eight shifting states, and the transfer is determined by Mp at the state of previous Shift. It remains same for every branch until Shift7. Shift7's next state will be Sub or Shift8, also based on Mp, but Sub is to do subtraction, not adding. After

Shift8 which is the last shift, it will goes to Done state, in which machine transfer to Halt if there isn’t a Run signal, and to Clear state and run again if there’s a run signal.

(6) module Multiplier_toplevel

input: CLK, RLC, RUN, [7:0] SW,

output: LED; [7:0] hex_seg_a, [3:0] hex_grid_a, [7:0] hex_seg_b, [3:0] hex_grid_b; [7:0] Aval, Bval

Description: The overall top level control module that connect physical IO port of FPGA board and clock signal to circuit. It takes and loads the value of switches adjusted by user, and connecting other modules together while passing control signal ‘state’ between register unit, calculation unit and control unit. Also, it takes register’s output value and show them on LED so that user can see the value changes.

(7) module hex_driver

input: CLK, Reset, [3:0] in[4]

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

Description: This module is copied from lab3, since its functionality fits the task in lab4 and we don’t need to do further improvement. It takes output bits of registers and 3-bits into Hex number and display on LED unit of the FPGA. The input value of module is 4 four-bit signal, totally make up to 16 bits. The module converts upper 4bits and lower 4bits in register value to two hex displayer respectively. The displayed numbers are in Hex format, and user can see value of register A & B separately.

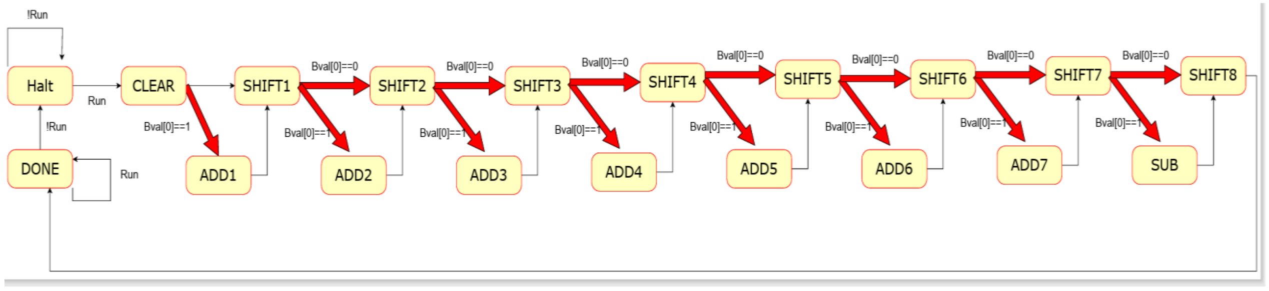
(8) module Sync

input: CLK, D

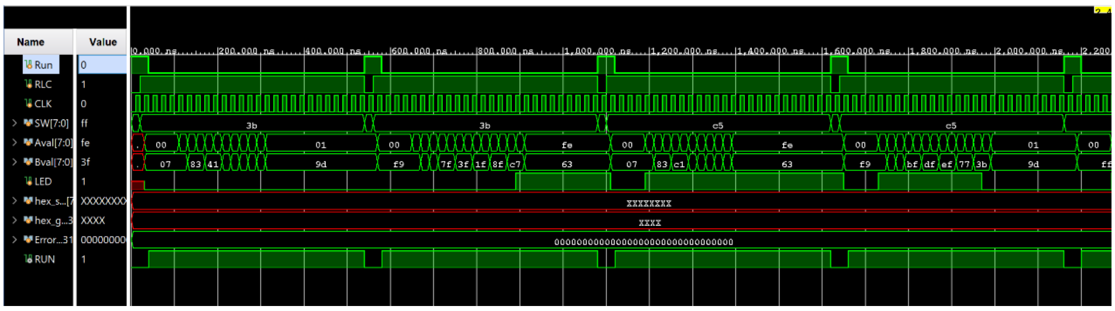
output: D

Description: This module is modified from lab3’s provided file. It serves as a synchronizer for pushbutton and switches inputs. This module’s logic makes sure to change buttons’ value only when CLK signal is at a positive edge. Thus, we simplified lab3’s design and only left an always_ff unit, which is enough for this functionality.

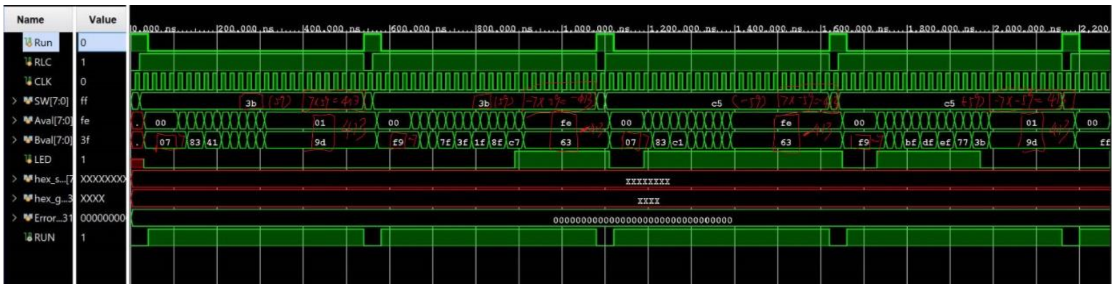
6. State Diagram for Control Units



7. Simulation Graph



Annotated Version:



8. Answer to Post-lab Questions

(1) *Refer to the Design Resources and Statistics in IVT and complete the following design statistics table.*

LUT	*71*
DSP	*0*
Memory (BRAM)	*0*
Flip-Flop	*76*
Frequency	*178.83MHz*
Static Power	*0.076W*
Dynamic Power	*0.013W*
Latches	*0*
Total Power	*0.083W*

(2) *What is the purpose of the X register? When does the X register get set/cleared?*

X stores the MSB of A register's value, it's used to determine the shift-in bit's value of register A. X is set only when the final add/subtract computation is completed, and it's cleared when finite state machine goes through the CLEAR state or RESET state which clear the value stored in X and A.

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

Then the MSB calculation could possibly be wrong due to the overflow of 7-bit components. The MSB will be the shift-in input for register A and will affect the final result's accuracy. A 9-bit adder can assure that shift in value for A is prepared and correct right after the calculation process is completed, and without it the overflow result from adding two 8-bit component could be lost.

(4) *What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?*

It only works if the product of previous multiplications can be truncated to an 8-bit value without changing its value. If the product of previous multiplications is over 128 or smaller than -127, the extended part over 8-bit will be lost after rerun the multiplier, and the result will be incorrect.

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

Pencil-and -paper method is more logical to human language and more intuitive for human understanding since it can see all partial product during calculation. It could work as a good method to justify whether your result generated by multiplier is correct. The disadvantage is that keep adding partial product's results means that you should prepare extra registers to store the values in the calculating process. That's redundant for hardware since number of registers is limited.

9. Bug encountered

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

. When I tried to do the simulation after finishing the first version of codes, we found that the A_value stays 0 and there are no intermediate state transitions. Later, I rechecked the toplevel file and I found that I should use previously provided file to do synchronization instead of simply using the "module Sync (

input logic clk, D,

```
        output logic Q);
        always_ff @ (posedge clk) begin
            Q <= D;
        end
endmodule
```
endmodule". So, I may use the previously provided file for synchronization.

(b) Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab? manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

. The lab manual is well assembled. For example, the explanation about the control unit is clear "The circuit should stop once the multiplication is done and the correct result should be displayed by outputting AB on the hex displays. Another (consecutive) multiply operation can be triggered by releasing the Run button and pressing it again. ". I could clearly know how to design the state machine.

10.Conclusion

- We implemented an 8-Bit Multiplier in SystemVerilog in this week's lab. The main concept of this lab containing methods to taking 2's complement multiplication algorithm into bit-wise logic processing consists of shifting and binary adding. We also learned how to implement a SystemVerilog project from nothing, and use self-designed logic transformation graph to implement a finite state machine and write its control unit.