University of Illinois at Urbana-Champaign

ECE385
Digital Systems Laboratory

Lab4: An 8-Bit Multiplier in SystemVerilog

TA: Abigail Wezelis, Harris Mohamed

Yan Miao, Guangxun Zhai
yanmiao2, gzhai5

February 2021

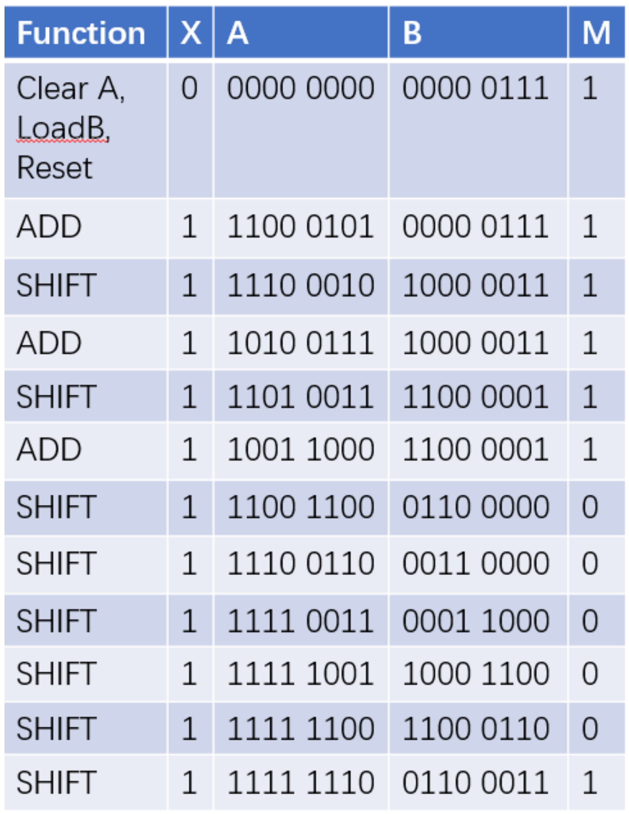
Contents

1 Introduction

In this lab, we aim at writing the SystemVerilog code about implementing a 8-bit multiplier. Inputs into the multiplier should be two 8-bit 2's compliment numbers. We used the shift-add algorithm to to fulfill our design.

2 Pre-lab Question

Rework the multiplication example on page 5.2 of the lab manual, as in compute 00000111 * 11000101 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
Clear A, LoadB, Reset	0	0000 0000	0000 0111	1
ADD	1	1100 0101	0000 0111	1
SHIFT	1	1110 0010	1000 0011	1
ADD	1	1010 0111	1000 0011	1
SHIFT	1	1101 0011	1100 0001	1
ADD	1	1001 1000	1100 0001	1
SHIFT	1	1100 1100	0110 0000	0
SHIFT	1	1110 0110	0011 0000	0
SHIFT	1	1111 0011	0001 1000	0
SHIFT	1	1111 1001	1000 1100	0
SHIFT	1	1111 1100	1100 0110	0
SHIFT	1	1111 1110	0110 0011	1

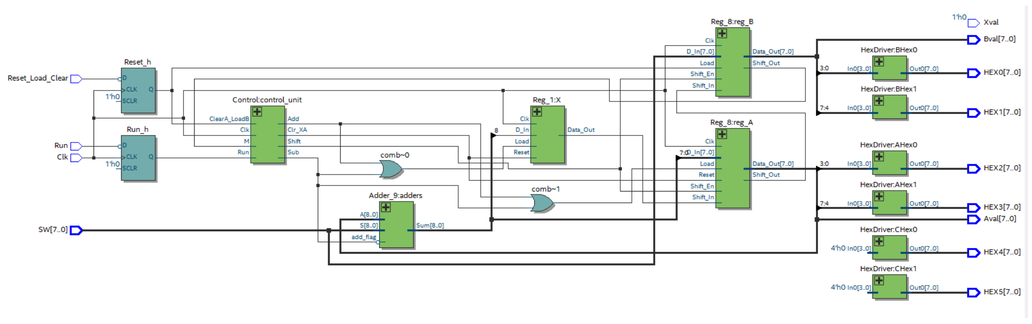
0000 0111 * 1100 0101

3 Written Description and Diagrams of Multiplier Circuit

3.1 Summary of Operation

We have 8 switches to decide our inputs and 2 buttons for Load and Reset. When we want to load a value into the register, we first adjust the switch to satisfy our input and press the Load button one time to load the number; the value being loaded will be displayed on our HEX1 and HEX0 as well as being stored in the register B. And if we want to perform the multiplication, we could set the switches to the 8-bit binary number we are going to multiply and then press the Run button; the result would be shown on HEX3, HEX2, HEX1 and HEX0(as well as register A and register B). Moreover, we made sure that the multiplier will only execute once even if we pressed Run for more than one cycle(E.g. holding the button). And reloading value to register B is always available in all states.

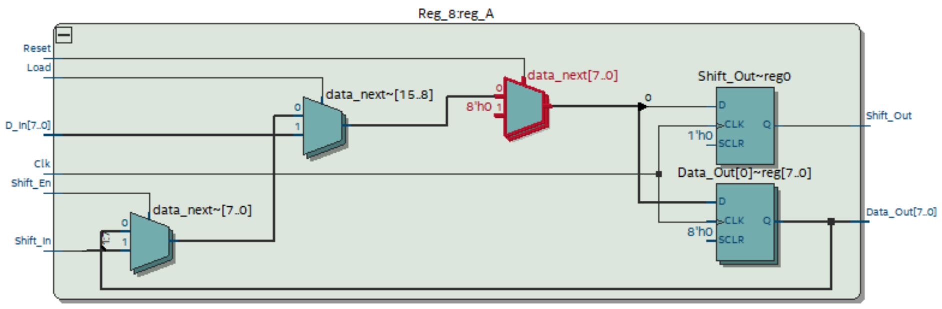
3.2 Top Level Block Diagram



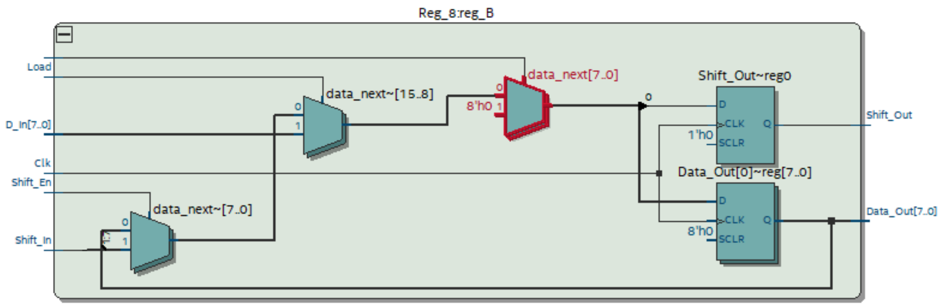
RTL View of Whole Project

3.3 Description of all .SV modules

1. **Module: Shift_Reg_8**
   - Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D_In
   - Outputs: Shift_Out, [7:0] Data_Out
   - Description: This is an 8-bit shift register. In Lab4, we initialized two instances of this shift register as register A and register B. The shift-in port(MSB of register B will be connected to the shift-out port(LSB) of register A, so when the control unit determines that a shift should be performed, both registers can shift all together.
   - Purpose: save the computational result of the multiplication.



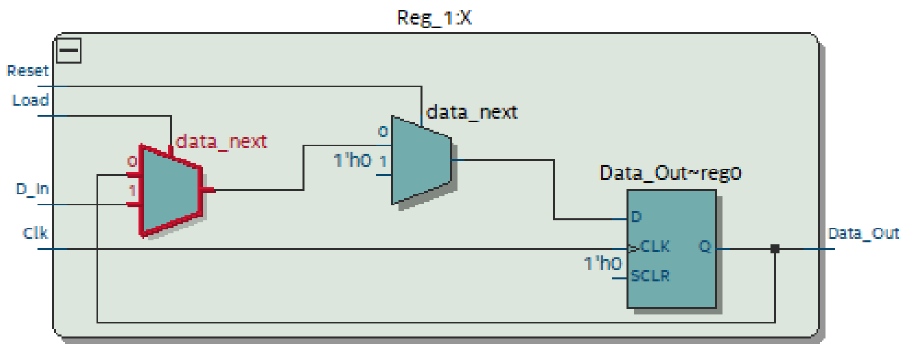
Reg_A RTL



Reg_B RTL

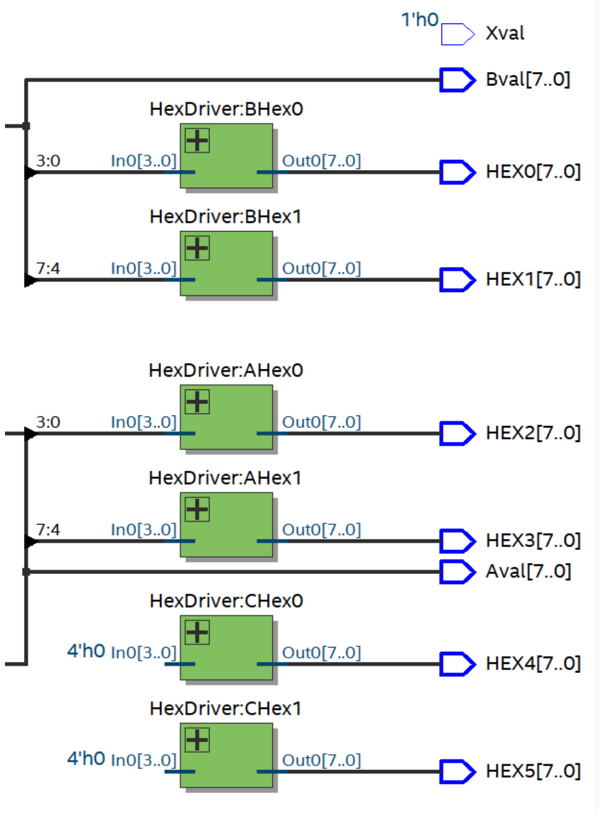
2. **Module: Reg_1**

   - Inputs: Clk, Reset, Load, D_In
   - Outputs: Data_Out
   - Description: this one-bit register takes the value of X, which is the MSB of register A to preserve its value during the arithmetic shift. It gets set whenever the adder performs a addition or subtraction.
   - Purpose: preserve the sign of value in register A in the case of arithmetic shift.



Reg_1 RTL

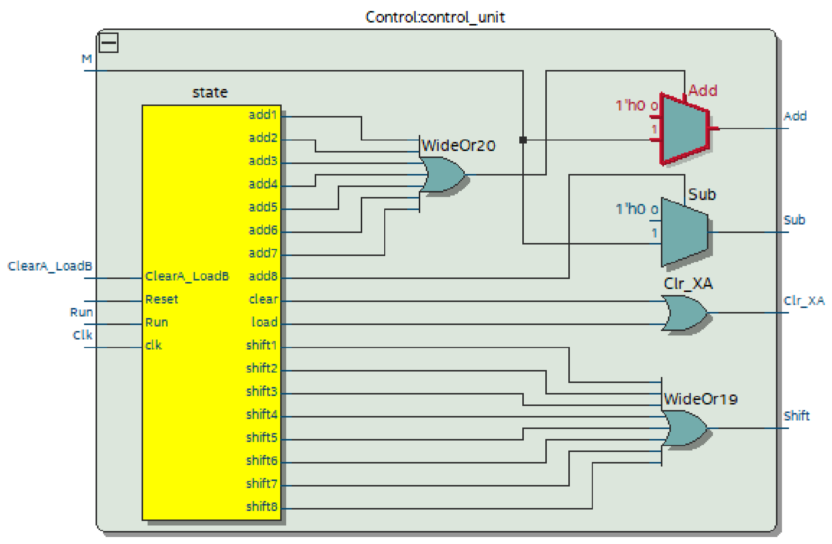
3. **Module: Multiply_top_level**
    - Inputs: Clk, Reset_Load_Clear, Run, [7:0] SW
    - Outputs: [7:0] Aval, Bval, Xval, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5
    - Description: This is the top-level file that defines the whole procedure of from how we load the switch values, how we do the operations, control run, and also how to print our output.
    - Purpose: This module serves for connecting other modules to make them work together.
4. **Module: HexDriver**
    - Inputs: [3:0] In0
    - Outputs: [7:0] Out0
    - Description:Each hex value has its unique corresponding 8-bit binary value. (8 bit because we modify the HEX driver to not display decimal point by adding another bit)
    - Purpose: Hex module is used to output the hex value on LED using the 8-segment method.



Hex_Driver RTL

5. **Module: Control**

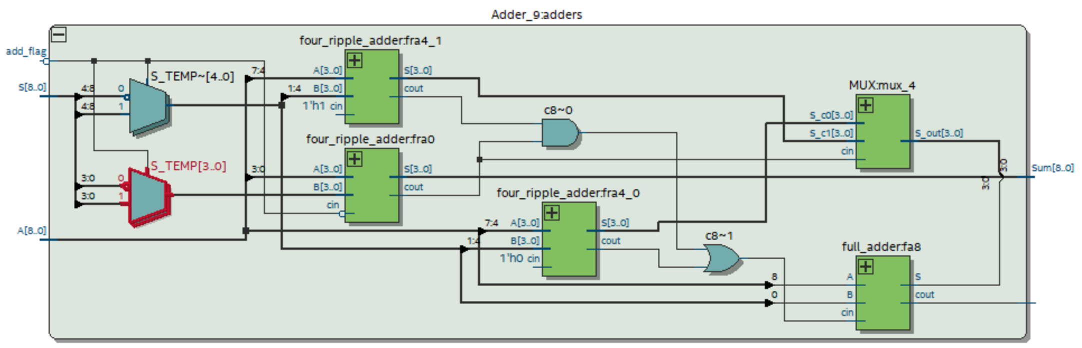
   - Inputs: Clk, Reset, Run, ClearA_LoadB, M
   - Outputs: Shift, Add, Sub, Clr_XA
   - Description: In this lab, we use start, load, waiting, halt, hold, clear, 8 add, and 8 shift states which are totally 22 states to implement the control unit. Add and shift are the basic steps to do multiplications. Halt state is to prevent continuous run if not releasing the button. Waiting state is the state between load and run. This is a Mealy Machine since the output(Add/Subtract) depends on both the state(Add8) and the input(M).
   - Purpose: This module controls the running states of our major operations and also how we interrupt or reset when we finish one operation on board.



Control RTL

6. **Module: Adder_9**

   - Inputs: [8:0] A, S, add_flag
   - Outputs: [8:0] Sum, cout
   - Description: This module serves to perform the operation of add and subtraction. Input add_flag will determine which operation we are going to do in this adder. We totally use three 4-bit ripple_adders, one 1-bit full_adder and a MUX.
   - Purpose: Do the operation of add or minus for the multiplication.



Adder_9 RTL

7. **Module: MUX**

   - Inputs: [3:0] S_c0, [3:0] S-c1, cin
   - Outputs: [3:0]S_out
   - Description: This module is a 2-to-1 MUX, which has 2 inputs, one select bit, and one output. When the select bit is 0, MUX will transmit one of its input values to its output and when the select bit is 1. it will output another input value.
   - tPurpose: A multiplexer makes it possible for several input signals to share one device or resource.

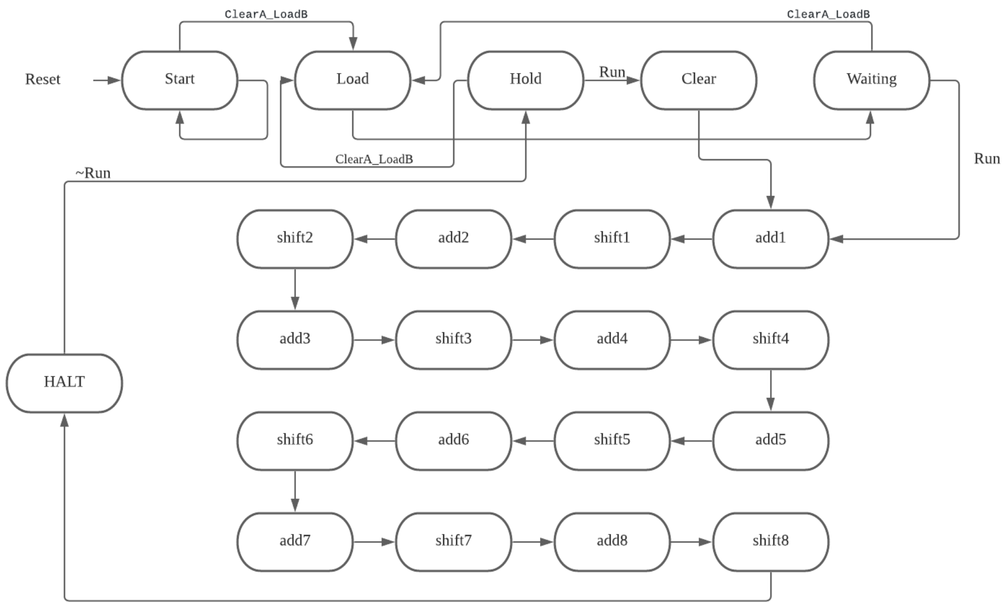
8. **Module: four_ripple_adder**

   - Inputs: [3:0] A, [3:0] B, cin
   - Outputs: [3:0]S, cout
   - Description: this module is just a tiny part of the 16-bit-ripple-adder. It does exactly the same function and have same weakness as the 16-bit-ripple-adder.
   - Purpose: Ripple-adder can do addition of longer bits of input numbers with easy design but ineffective speed.

9. **full_adder**

   - Inputs: A, B, cin
   - Outputs: S, cout
   - Description: we use the given logic of addition between inputs and out outputs
   - and assign the correct value to S and c_out.
   - Purpose: Because full-adder is included in any of the three complicated n-bit adders, this module is used as a basic file for doing adding calculation of A/B.

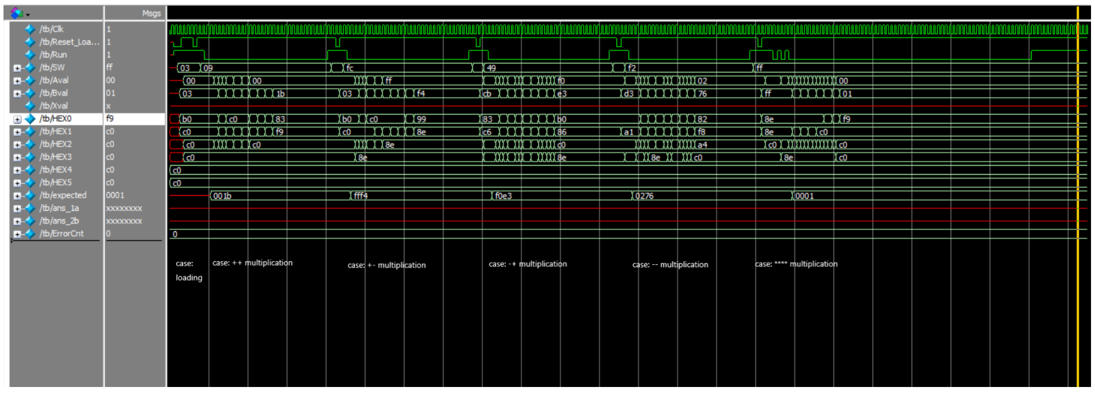
3.4 State Diagram for Control Unit



State Diagram

Each states can go to the start state when Reset = 1.

4 Annotated Pre-lab Simulation Waveforms



We wrote 6 cases in our testbentch to test the functionality of our code. Case 1 is just loading switch value into B, and the result of B and switch are both 8'd03. Case 2-5 we tested all the multiplication operations, including ++, +-, -+, –. And Case 6 is continuous multiplication. After each case, we compare the result of our design in simulation with the expected pre-calculated result; if there is an discrepancy, we would increase the err_cnt variable. As you can see, the err_cnt in the above simulation is 0, which indicates our design is functioning properly.

5 Answers to Post-Lab Questions

(a) **Refer to the Design Resources and Statistics in IQT and complete the following design statistics table. Come up with a few ideas on how you might optimize your design to decrease the total gate count and/or to increase maximum frequency by changing your code for the design.**

LUT	102
DSP	0
Memory (BRAM)	0
Flip-Flop	41
Frequency	264.13MHz
Static Power	89.94mW
Dynamic Power	0mW
Total Power	98.75mW

Design Analysis

There are several ways to optimize our current design. First, we could try using fewer states by introducing 2 counters. Since many of the states are doing the same thing, we could replace 8 Add/Shift states with 1 state with the help of counters, and use the counter value to check how many add/shift we already performed. If we have less states, intuitively we will have a shorter critical path and thus leads to faster execution time. Second, we could try to remove the reset signal(that is given by the provided file), since in our current FSM, the reset signal will always lead us back to the starting state, which creates more state transitions. In our implementation, the reset signal has no use since its function is already covered by our Clear_Load signal.

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

X register saves the value of the most significant bit of Register A, and it serves as the sign bit of our multiplication in this lab. The reason we have it is because we want to preserve the sign when we do an arithmetic shift of register A. We set the X register to the MSB of A whenever an addition and a subtraction is performed. We clear the value of X whenever Reset_Load_Clear is pressed and right before continuous multiplication is performed.

(c) **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 reason why we used a 9-bit adder is because we want to clearly specify and preserve the sign of the result. If we use the carry bit of the 8-bit adder, the carry bit is not necessarily the same as the sign of the addition result. Therefore, the result of multiplication could be wrong because we are shifting with the wrong value as the sign of A might be changed.

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

Continuous multiplication only works when the product is within range [-128, 127], which means that the product can be represented with 8-bit 2's compliment and only fits in register B(visually speaking, the output only takes HEX1 and HEX0). On the other hand, continuous multiplication will fail in the product needs to be represented with more than 8 bits(also using register A); this fails because we need to clear register A before continuous multiplication, therefore some data in register got lost and we would get the wrong product.

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

One advantage of our implementation over the pencil-and-paper algorithm is that we only need a 9-bit adder while the pencil-and-paper needs a 16-bit adder(they shift all the values and do addition, so the longest value has 16 bits). The disadvantage of the algorithm lies in that it's less intuitive than the pencil-and-paper one. We need an extra register to preserve the MSB and conceptually it took longer to understand the algorithm.

6 Conclusion

6.1 Functionality of Our Design

Our design uses 22 states in total, of which 16 of them are shift and add. The others are starting and waiting states, that prevents multiple execution when holding down the Run key.

One of the bugs we had is exactly related to this: when we pressed the Run key, the values displayed on HEX changes rapidly, that is solved by adding another release_button state.

Another bug we encountered is that the negative-positive multiplication behaves strangely, that happened because we forgot the "add one" part when converting a 2's compliment number (flip + add one). But we solved everything and finished all the functionality required in the document.

We also had the bug of not being able to reload the value into Register B after one successful load; with the help of signalTap, we discovered that this is caused by a missing state transition back to load state from the waiting state and can be solved by adding a simple if statement.

6.2 Suggestions to Lab-Manual and Given Document

We always felt the knowledge we learned from lectures are pretty useful for us to understand and do the labs. However, lectures are always delivered after we finish the labs so that we spend extra time on learning these new labs and might change some of our designs after lectures. As a result, we hope we can somehow accelerate the current process of our lectures so that we can learn stuff before we start the labs.