ECE385

Fall 2022

Experiment 4

An 8-bit Multiplier in System Verilog

Lai Shu Xian
Section NL
TA: Nick Lu

Basic functionality of multiplier circuit

The circuit of this lab takes in two inputs both of which can be up to an 8-bit hexadecimal number. The circuit performs an 8-bit multiplies 8-bit operation on the two inputs to produce a 16-bit output. The product is displayed on the display segments of our FPGA, and subsequent attempts to run multiply will perform the multiplication of the lower 8 bits of latest product with the current switches' configuration.

Prelab

1. Stepwise demonstration of how the algorithm runs in an example 11000101 * 00000111.

Initial values X = 0, A = 00000000, B = 00000111, switches is set to 11000101.

Function	X	A	B	B zeroth bit (LSB)	Next step
ClearA_ LoadB_Reset	0	00000000	00000111	1	LSB = 1, perform A + S and store in A
Add	1	11000101	00000111	1	1-bit arithmetic right shift on XAB.
Shift	1	11100010	10000011	1	LSB = 1, Perform A+S and store in A.
Add	1	10100111	10000011	1	1-bit arithmetic right shift on XAB.
Shift	1	11010011	11000001	1	LSB = 1, Perform A+S and store in A.
Add	1	10011000	11000001	1	1-bit arithmetic right shift on XAB.
Shift	1	11001100	01100000	0	LSB = 0, only do 1-bit right shift.
Shift	1	11100110	00110000	0	LSB = 0, Only do 1-bit right shift.
Shift	1	11110011	00011000	0	LSB = 0, only do 1-bit right shift
Shift	1	11111001	10001100	0	LSB = 0, only do 1-bit right shift.
Shift	1	11111100	11000110	0	LSB = 0, only do 1-bit right shift.
Shift	1	11111110	01100011	1	8th shift is done. Stop. 16-bit product in AB.

0b11000101 (-59 in decimal) * 0b00000111 (7 in decimal) = 1111,1110,0110,0011 (-413 in decimal) and it agrees with what our algorithm has as the final product in AB.

Written description and diagrams of multiplier circuit

a. Summary of operation

We call the number to be multiplied by the multiplicand, whereas the number we multiply is call the multiplier. There are two buttons on the FPGA board, the upper button is for

ClearA_LoadB_Reset, the lower button is for Run. The 8 switches are used to set the value of our two 8-bit inputs.

In the beginning of a single multiplication, we start by setting the switches value to the value of our multiplier. Then, by pressing the ClearA_LoadB_Reset button, we clear registers A and X and simultaneously load the value of the switches into register B. Register B now has the value of the number we want to multiply.

We then tune the switches to set the value for our multiplicand. Once we done that, we can hit the Run button to multiply A and B and the result will be displayed on 4 display segments of the FPGA displays (reading left to right of the displays will be our result).

If subsequent multiplication is desired, we can repeatedly press-and-release the Run button, and this will continuously take the lower 8-bits of the latest product (the 16-bits currently displayed on board) and multiply by the current switches value and display the result again.

If we press the ClearA_LoadB_Reset button, we restart the whole process as if we just started to want to multiply.

b. Top Level Block Diagram

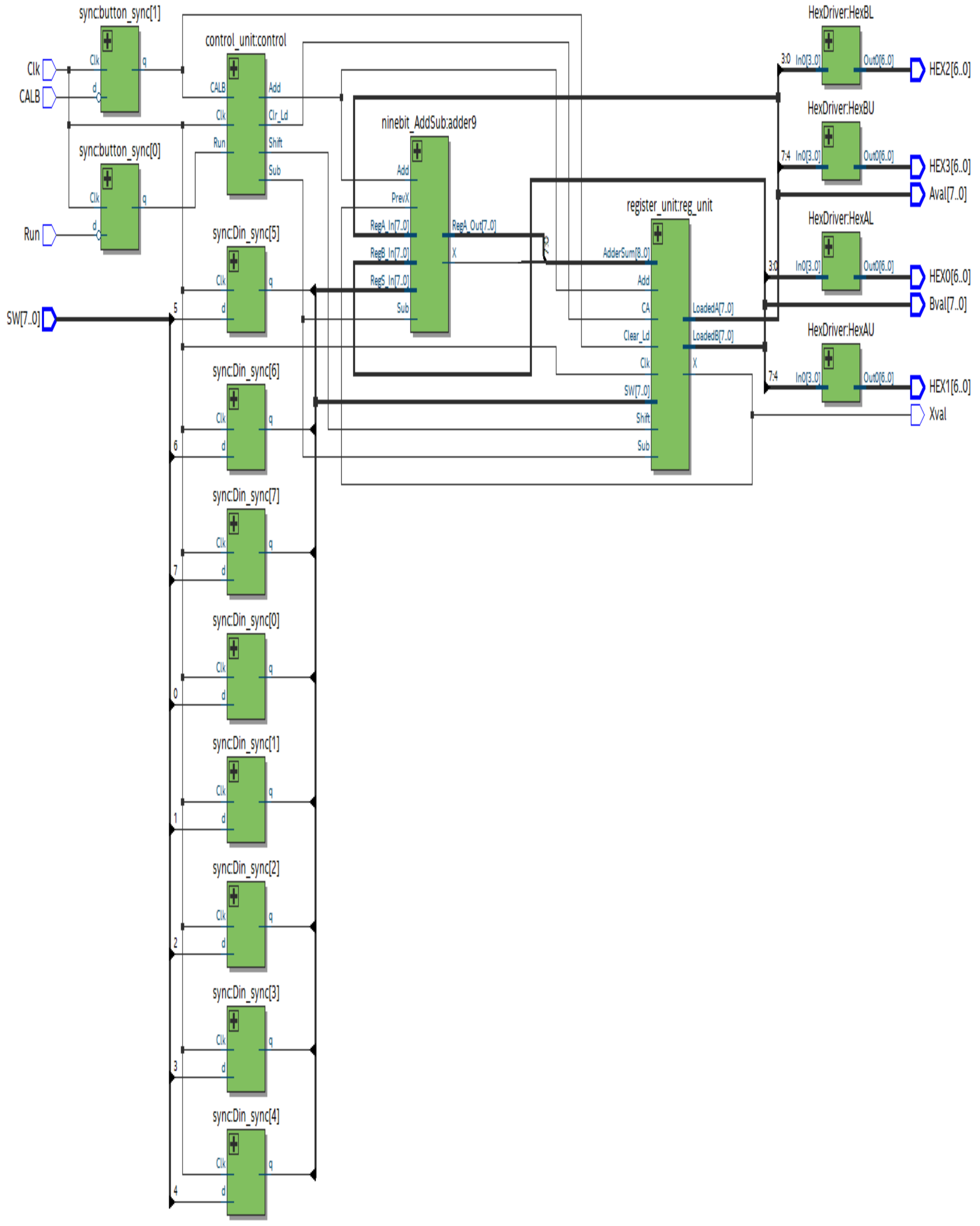


Figure 1 shows the top-level block diagram of the 8x8 multiplier generated by Quartus.

c. Written Description of Modules

Module: single_rippleadder.sv Inputs: A, B, Cin Outputs: S, CarryOut	Description: Takes in A, B, Cin to perform a single bit addition and generate S that represents output, CarryOut if there is overflow. Purpose: Basic block to construct ripple adder for addition of longer bits number.
Module: fourbit_ripple.sv Inputs: [3:0] A, [3:0] B, Cin Outputs:[3:0] S, Cout	Description: Performs addition on two 4-bit numbers with a given initial carry in. Sum is generated in output S as well as Cout to indicate overflow. Purpose: Building block for longer chain addition, highly reusable for n-bit addition where n is a multiple of 4. Later to use to combine with another five-bit ripple adder to form a nine bit adder block.
Module: fivebit_ripple.sv Inputs: [4:0] A, [4:0] B, Cin Outputs: [4:0] S, Cout	Description: Performs addition on two 5-bit numbers with a given initial carry in. Sum is generated in output S as well as Cout to indicate overflow. Purpose: Building block for longer chain addition, highly reusable for n-bit addition where n is a multiple of 5. Later to use to combine with another four-bit ripple adder to form a nine-bit adder block.
Module: CSA_select.sv Inputs: cout1, cout2, prev_cout Outputs: next_cout	Description: Performs combinational logic on the two carry outs of a slice of select adder and the rippled carry forward to generate a correct carry forward for the next addition unit Purpose: Glue logic for carry select adder between blocks of single bit adder to generate the next carry in for subsequent adder.
Module: CSA_mux.sv Inputs: [4:0] adder0, [4:0] adder1, select Outputs: [4:0]	Description: Takes in the two different outputs from the adder blocks as well as the carry in generated by previous block to

	determine which output correctly represent the final sum. Purpose: Functions as a 2-1 mux for CSA
Module: select_adder.sv Inputs: [8:0] A, [8:0] B, Cin Outputs: S, Cout	Description: Takes in two 9-bit inputs and perform an addition operation on them given an initial carry-in Cin. The sum of the addition will be outputted in S and Cout will indicate whether we have an overflow. Purpose: Used as the ALU for the nine-bit adder.
Module: register_unit.sv Inputs: Clk, CA, Clear_Ld, Shift, Add, Sub, [7:0] SW, [8:0] AdderSum Outputs: A_out, [7:0] LoadedA, [7:0] LoadedB	Description: Takes in a clock, several control signals (CA, Clear_Ld, Shift, Add, Sub) and an 8-bit number based on the switches, and the nine-bit addition output from the nine-bit adder. Depending on the control signals, registers A and B will be cleared, loaded or shifted accordingly. A_out is the output of the discarded A bit if A is being right shifted (in other words, the least significant bit of A). LoadedA and LoadedB are the values that has been succesfully loaded into the registers A, B respectively. Purpose: Acts as a wrapper module to combine all the involved registers A, B and X.
Module: reg8_A.sv Inputs: Clk, CA, Reset_CALB, Shift, Shift_In, Add, Sub, [7:0] Din Outputs: Shift_Out, [7:0] Data_out	Description: Takes in a clock and some control signals. On rising clock edge, based on the control signals perform a load, a clear or a shift accordingly on the register content. If a load is desired, then Din serves as the value being loaded. If shift is desired, then Shift_In is the value being right-shifted in. Data_Out will output the eventual content of the register after the register contents update.

	Purpose: Acts as register A in the block diagram that at the end of a multiply operation, store the higher 8 bits of the product.
Module: reg8_B.sv Inputs: Clk, Shift, Shift_In, Load, [7:0] Data_In Outputs: [7:0] Data_Out	Description: Takes in a clock and control signals to load or to shift (but no clear). On the rising clock edge based on the control signal perform the corresponding action to update register. The updated register value will be in output Data_Out. Purpose: Acts as register B in the block diagram that stores the multiplier in the beginning of a multiply, then store the result of the lower 8 bits of a product when multiplication completes.
Module: ninebit_AddSub.sv Inputs: [7:0] RegA_In, [7:0] RegS_In, [7:0] RegB_In, Add, Sub Outputs: X, [7:0] RegA_Out	Description: Inputs Add and Sub serve as control signal to determine whether we are performing value of register A plus value of register S or subtracting them. RegB_In as an input also serves as control because the least significant bit in B tells use whether we need to perform an add. If it is 1, add. Otherwise do nothing or equivalently, add 0. The output of the adder unit will be A+S, A-S or A+0, and RegA_Out will have the eight lower bits of that addition while the ninth bit will be in output X. Purpose: Performs the addition/subtraction as needed depends on register B LSB as per algorithm. The output is then fed the loading value for the register unit.
Module: control_unit.sv Inputs: Clk, Run, CALB Outputs: ClrLD, Shift, Add, Sub	Description: Takes in the global clock and inputs from the buttons. If Run is pressed, triggers the full cycle of the multiplication algorithm. If CALB is pressed, Register A and X will be cleared, and register B will be

	loaded and the state machine is reset to its initial state. Purpose: Moore state machine implementation of the control path for multiplier. Based on which states we have, produce the corresponding output signals to control the data path.
Module: HexDriver.sv Inputs: [3:0] In0 Outputs: [6:0] Out0	Description: Takes in a 4-bit binary number and outputs its corresponding hexadecimal value on a 7-segment LED display setup using switch case statements. Purpose: To light up the correct LED segments to show a 4-bit binary number as a single hex number on the LED display segment.
Module: testbench.sv Inputs: N/A Outputs: N/A	Description: N/A Purpose: a self-written test file to be run on ModelSim to assist in debugging and verification process of design.
Module: eightbit_multiplier.sv Inputs: Clk, ClearA_LoadB, Run, [7:0] SW Outputs: [6:0] HEX0-3, [7:0] Aval, [7:0] Bval, Xval	Description: Takes in two buttons (ClearA_LoadB and Run) and 8 switches as input. Details about how these switches and buttons sequence relate to the multiplication operation is explained in the beginning of the report. Outputs the 16-bit product of A x B onto the hexadecimal display segments, and the 16-bit value is stored from left to right in Aval, Bval respectively. Xval is stored but does not have significance in terms of the product. Purpose: Serves as top-level design entity file of the 8-bit multiplier. Is instantiated within the testbench during simulation and programmed onto the FPGA board when synthesizing. HEX0-3 serve as display values on the board to show the result. Aval Bval are used for simulation debugging purposes.

d. State Diagram for Control Unit

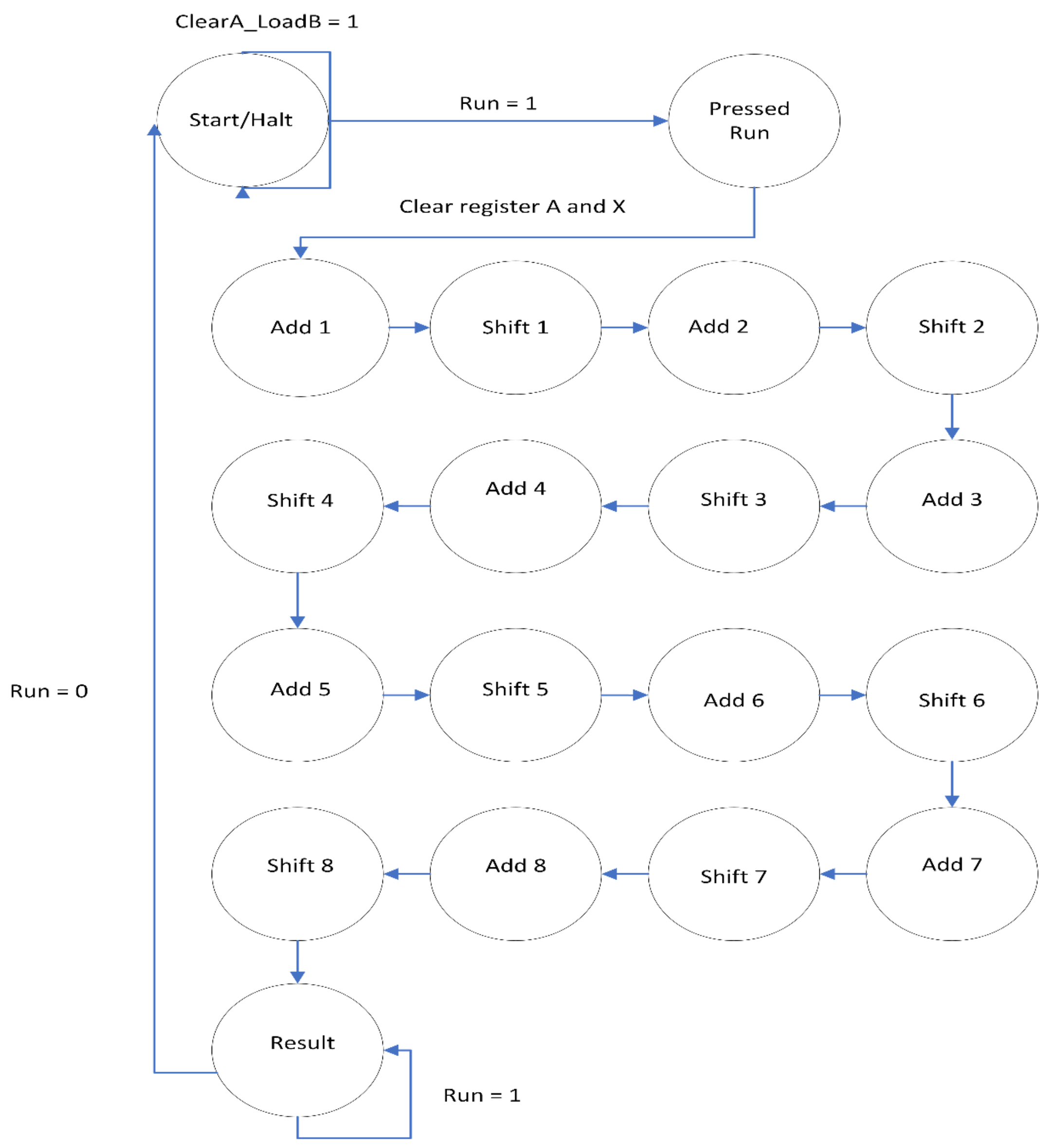


Figure 2 shows the Moore state machine diagram that I used to implement the multiplier. The Start/Halt state is the default starting state in the FSM.

In the table below, we will list down the outputs (Clear, Shift, Add and Subtract) corresponding to each state. The labelled arcs of our state machine diagram are the conditions that leads to the transition. Run and ClearA_LoadB represents button pressed when 1 and button not pressed when 0. The outputs will serve as control signals to our data path.

States/Output	Clear	Shift	Add	Subtract
Start/Halt	0	0	0	0
Pressed Run	1	0	0	0
Add1, Add2, Add3 … Add7	0	0	1	0
Shift1, Shift2, Shift3 … Shift8	0	1	0	0
Add8	0	0	0	1
Result	0	0	0	0

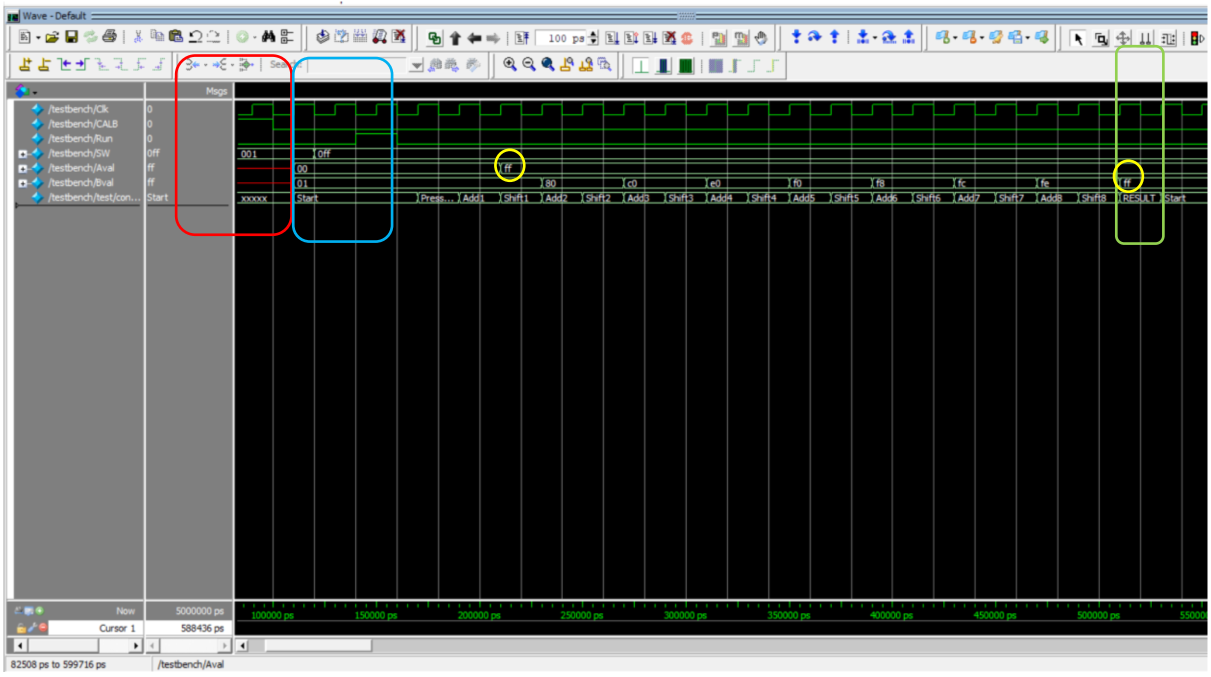
Note that some of the Add/Sub states are dummy states in the sense that they can add (or subtract) 0 if operation is unnecessary. The logic is handled within the data path, not the state machine.

Annotated Simulation waveform

To ensure the correctness of our implementation, we constructed a comprehensive testbench that tested all possible multiplication pairs (--, ++, +-, -+). They are tested with no reset in between.

The 4 snapshots below show the four simulation outputs, displayed numbers are in hex.

Note: zoom-in is necessary to see the values of the picture clearly.

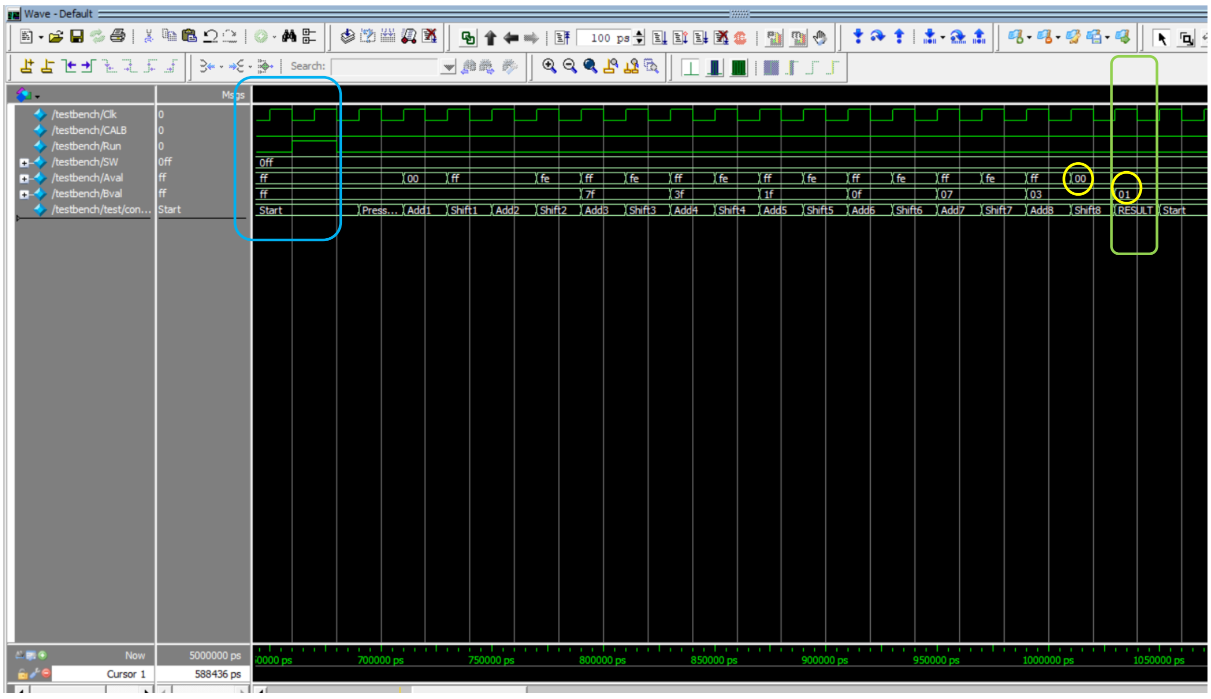


Simulation 1 shows -1x1 *operation.*

Red box shows we set the switches (SW) to 0x001 and hit the ClearA LoadB button. We can see that at the beginning of blue box Aval is cleared is Bval is set to 0x01.

Blue box then shows we set switches to 0xff and hit the Run button. The algorithm then begins as shown by the bottom signals of alternating states.

Green box shows the state where we have the result ready in register A and register B. Those values are also circled in yellow as emphasis. The top yellow circle is register A; bottom yellow circle is register B. They combine to give 0xffff (decimal -1), which is the expected answer.

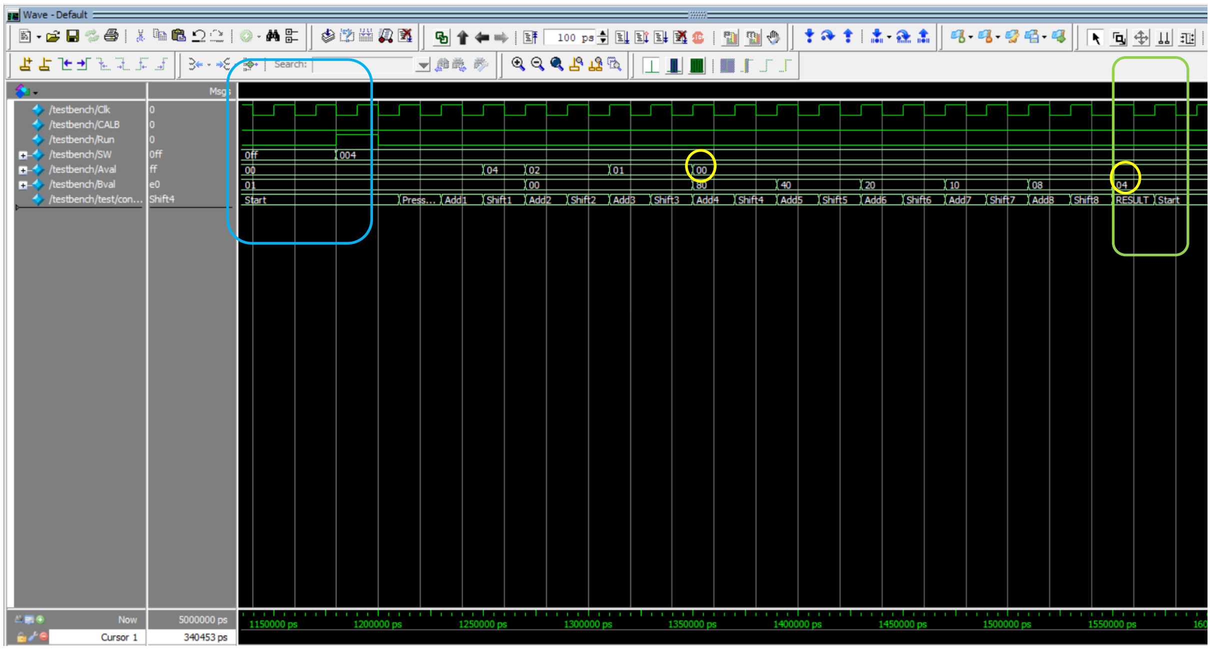


Simulation shows -1x-1 *operation.*

Blue box shows we set switches to 0xff and hit the Run button. Bval is 0xff before we hit the Run button, this means we are performing 0xff multiply 0xff.

The algorithm then begins as shown by the bottom signals of alternating states.

Green box shows the state where we have the result ready in register A and register B. Those values are also circled in yellow as emphasis. The top yellow circle is register A; bottom yellow circle is register B. They combine to give 0x0001 (decimal 1), which is the expected answer.

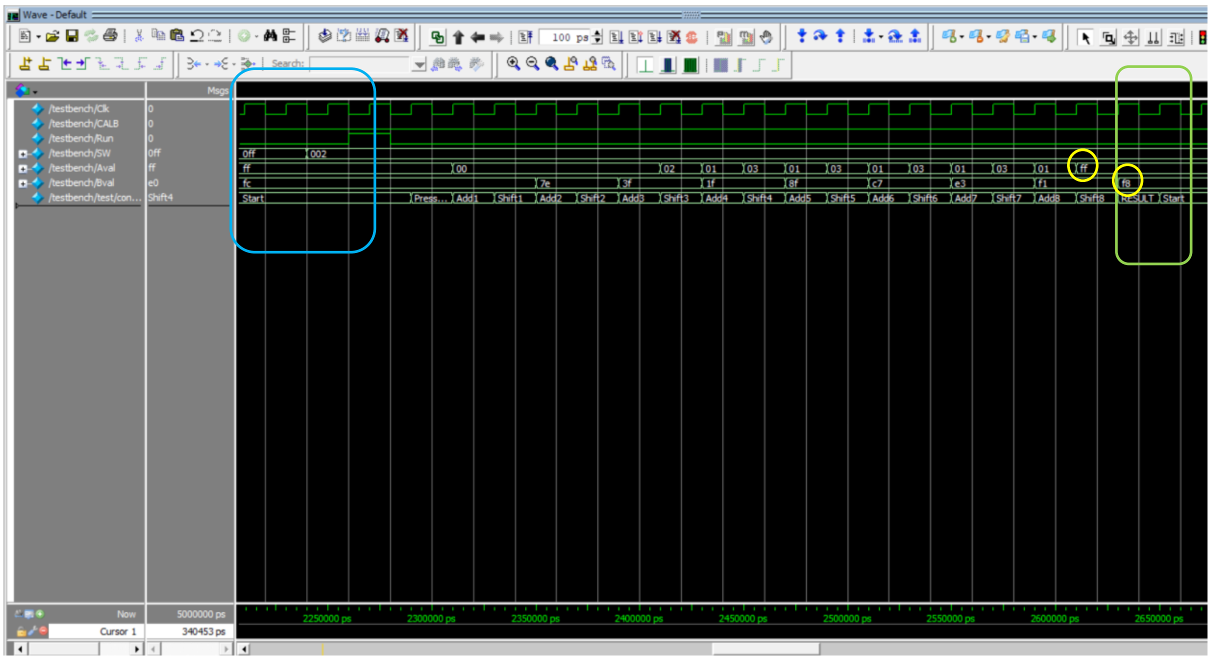


Simulation shows 4x1 *operation.*

Blue box shows we set switches to 0x04 and hit the Run button. Bval is 0x01 before we hit the Run button, this means we are performing 0x04 multiply 0x01.

The algorithm then begins as shown by the bottom signals of alternating states.

Green box shows the state where we have the result ready in register A and register B. Those values are also circled in yellow as emphasis. The top yellow circle is register A; bottom yellow circle is register B. They combine to give 0x0004 (decimal 4), which is the expected answer.



Simulation shows 2x-4 *operation.*

Blue box shows we set switches to 0x02 and hit the Run button. Bval is 0xfc before we hit the Run button, this means we are performing 0x02 multiply 0xfc.

The algorithm then begins as shown by the bottom signals of alternating states.

Green box shows the state where we have the result ready in register A and register B. Those values are also circled in yellow as emphasis. The top yellow circle is register A; bottom yellow circle is register B. They combine to give 0xfff8 (decimal -8), which is the expected answer.

Post-lab Questions

1. Design and Resource Statistics

LUT	117
DSP	0
Memory (BRAM)	0
Flip-flop	46
Frequency	75.37MHz
Static Power	89.97mW
Dynamic Power	1.54mW
Total Power	103.66mW

Possible Optimizations to decrease total gate count/increase maximum frequency

One of the main time-efficiency waste comes from current implementation is due to the usage of Moore FSM instead of a Mealy FSM. Since there is a counter-like structure in the add-shift loop of the algorithm, there should be a Mealy FSM that can simplify the states, which can cut down the total gate count and increase frequency.

I used the carry select adder implementation for my 9-bit adder module, this could be replaced with a carry ripple adder design to cut down the total gate count since I will not need the glue logic and half the number of adder blocks. However, this will come at a price of sacrificing the operating maximum frequency because ripple adder is slower than select adder.

Conceptual Post-lab questions

2.1 What is the purpose of the X register?

- The purpose of the X register is to maintain (or remember) the sign bit for our multiplicand.

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

- We will mess up the logic (sign bit in particular) of this algorithm because the X register is supposed to represent the sign bit. If we do an 8-bit addition and use the carry out, when we perform the addition of two numbers with MSB = 1, then the carry out will be 1 because of overflow. However, what we want is that the MSB be flipped in the sum because negative-negative in multiplication gives positive.

Similarly, we can lose the sign bit if we do an addition with a number like x80 + x00. The carry out will be zero, and now it seems like we are doing a positive-positive multiplication.

<u>2.3 Limitations for continuous multiplications? Under what circumstances will the algorithm fail?</u>

- One limitation for continuous multiplication is we can never take the full result of our product and multiply by the next number. We always only take the lower 8-bit of the result and multiply it by the current value of switches. An example of such instance is if we have the current product to be 0x7FFF. Continuous multiplication technically should be 0x7fff * (whatever the switches value is). Suppose switches' value are 0x7f (decimal 127), then we expect the result of the continuous multiply to give 0x3F7F81. But the result we obtain will only be 0xFF81 (-127 in decimal).

- Basically, the circumstance for continuous multiplication to fail is when the current product consists of more than 8 bits, then we are losing some of our current product.

<u>2.4 Advantages and disadvantages of the implemented algorithm over the pencil-and-paper method?</u>

- The advantages of this algorithm it saves the arithmetic effort of multiplying as well as skipping unnecessary additions. In the pencil and paper method, we multiply in each iteration and perform a wide addition to get the result. In this algorithm, we replace the multiplications by bit-shifting, which is less computing extensive.

- The disadvantage of this method is it could be more complicated to explain conceptually to another person why it works.

Conclusion

Functionality wise, my circuit completely fulfils the lab requirement in the sense that the circuit can perform single as well as subsequent multiplication.

I thought the timeframe given for this lab compounded by the size of the class could be an issue for some groups/individuals. This fact can be easily confirmed by the staff or the students who went to office hours to seek help – the queue was horrible. It is right to say that it is not a conceptually hard lab, and no complex algorithm needs to be implemented. However, the challenges mainly came from not having solid foundation and exposure to coding, designing, and debugging in System Verilog. I appreciate that the process forces us to be independent and I understand certain skills can have a steep learning curve at the start, but I still think there are things that can be done better so that students are better prepared. But again, hoping for ECE courses to ever care about those lagging behind is like hoping for China to become a democratic country.