ECE 385

Fall 2023

Experiment 4

Designing an 8-bit multiplier in SystemVerilog

Cher Rui Tan, Adithya Balaji
Section: Friday 11.30am
TA: Gene Lee

Introduction

In this lab, we designed and built an 8-bit multiplier that is able to perform multiplication on two 8-bit strings, using an add-shift algorithm based on the handwritten method of multiplication. To perform the multiplication, use SW[8:0] to Load one value, and then set the switches to another desired number and press the run button to perform the multiplication operation.

Pre-lab questions

a. Rework the multiplication example on page 5.2 of the lab manual, as in compute 11000101 * 00000111 in a table like the example.
The table below reworks the multiplication example in reverse order.

Function	X	A	B	M	Comments for next step
Clear A, LoadB, Reset	0	00000000	00000111	1	Add S to A since M = 1
ADD	1	11000101	00000111	1	Right shift by 1 bit
SHIFT	1	11100010	10000011	1	Add S to A since M = 1
ADD	1	10100111	10000011	1	Right shift by 1 bit
SHIFT	1	11010011	11000001	1	Add S to A since M = 1
ADD	0	10011000	11000001	1	Right shift by 1 bit
SHIFT	1	11001100	01100000	0	M = 0, right shift by 1 bit
SHIFT	1	11100110	00110000	0	M = 0, right shift by 1 bit
SHIFT	1	11110011	00011000	0	M = 0, right shift by 1 bit
SHIFT	1	11111001	10001100	0	M = 0, right shift by 1 bit
SHIFT	1	11111100	11000110	0	M = 0, right shift by 1 bit
SHIFT	1	**11111110**	**01100011**	0	Done, 16-bit product stored in AB

Figure 1: Multiplication example

Written description and diagrams of multiplier circuit

Summary of operation

Loading: Before performing the multiplication, the multiplier is set using the switches 8 to 1, and this is loaded into Register B by pressing the reset-clear button. Subsequently, we set the multiplicand we want by using the switches and pressing the run button to perform the multiplication computation.

Multiplication: As described above, an add-shift algorithm is used to multiply B and S. This involves 21 states, with the 16 alternating add and shift states. In the Add state, the lowest significant bit of B is checked by storing this bit in a register M and checking whether it is 1 or 0. If 1, we add the 9-bit sign extended version of A and S, and if M is 0, we do not do anything to the value in the A register. In the shift state, we simply perform an arithmetic right shift on the 17 bit string XAB. There is a register to store the value of X, which is the most significant bit of A + S for this operation, to be used during the right shift to ensure no loss of data. This keeps repeating until the 2nd last state, where a subtract operation is performed instead of adding if the least significant bit, which is the original most significant bit of S, is 1. This is because the numbers are represented using two's-complement, which means we need to take into account that the most significant bit of the multiplicand being 1 would lead to an inverted result. After the final shift state after this, we store the final result in AB to be displayed on the Hex displays.

Post Multiplication: After procedure is completed, the FSM transitions into state Q where it waits for the Run signal to be low. This prevents the process from running multiple times while the Run button is held down. The state machine then transitions into the Hold state where we are able to observe the display before moving to the Clear state when the Run button is pressed again. In the Clear state, A is cleared right before the next multiplication of 8 bit 2's complement numbers.

b. Top Level Block Diagram

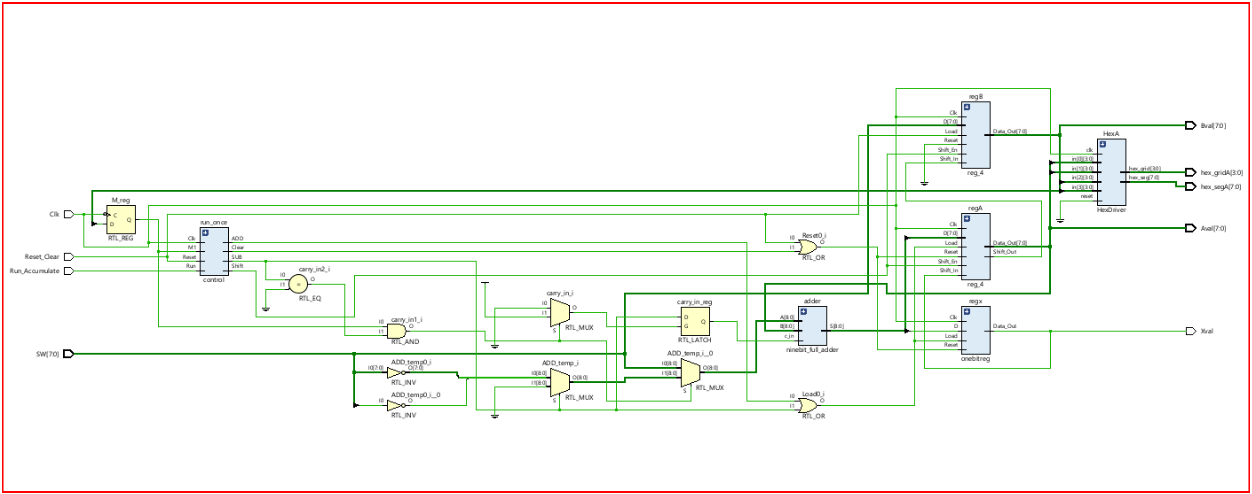

Figure 2: RTL Block Diagram

Written Description of .sv Modules

Module: toplevelmultiplier.sv:

Inputs:	input Clk, Reset_Clear, Run_Accumulate,
	input [7:0]SW,
Outputs:	output logic [7:0] Aval, Bval,
	output logic Xval,
	output logic [7:0] hex_segA,
	output logic [3:0] hex_gridA

Description:Top level module for multiplier which instantiates all the other modules needed to perform the operation, as well as declares temporary variables to be used in these other modules. Additionally, handles the logic for outputting the result on the hex displays by instantiating a HexDriver module. Handles logic for what to Add based on the M bit value.

Purpose: Module encompassing the entire logic process to implement the multiplier.

Module: control.sv:

Inputs:	input logic Clk, Reset, Run, M1,
Outputs:	output logic Clear, Load, Shift, ADD, SUB);

Description: Implements a 21 state FSM with 16 states A to P relevant to the 16 alternating Add, shift states, Q being a temporary hold state where every signal is forced to 0 and waits till the run signal goes back to 0, a Hold state where we are able to observe the the result before Run is pressed and finally the Clear state before any add or shift happens. Additionally, a default case is written in the FSM, as well as the logic for reset being 1 written in an always_ff block where it remains in the hold state while reset is being pressed and moving to the next state when released.

Purpose: To control the FSM for the multiplication operation

Module: fourbit_full_adder.sv:

Inputs:	input logic [3:0] A, B,
	input logic c_in

Outputs: output logic [3:0] S,
output logic c_out

Description:

Top level unit for each four bit adder, which calls on 4 individual full adders to compute the necessary results of S and c_out for one group of 4 bits of A added to one group of 4 bits of B.

Purpose: To instantiate 4 adders to be able to implement a 4 bit adder to compute the result for 4 bits of A and 4 bits of B.

Module: full_adder.sv:

Inputs: input logic x, y, z,

Outputs: output logic s, c

Description:

One full adder that performs an add operation on 1 bit of A with 1 bit of B, and determines the carryout from the values in X, Y and Z, which are the two bits and carryin.

Purpose: To act as a single-bit full adder to be used and called back in the higher level 4bit_full_adder module.

Module: reg_4.sv:

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

Outputs: output logic Shift_Out,
output logic [7:0] Data_Out;

Description: A default module to create an 8 bit register which runs on the positive edge of the clock and implements the logic for reset, load and shift_en. Data out is set to 0 when reset is high, data out is set to the input when load is high, and data out consists of the X bit concatenated with the 8 bit data out to produce a 9 bit string. Also handles the logic of assigning the X bit to the MSB of the 8 bit string in A.

Purpose: A register unit that is instantiated in upper level modules to be able to handle the loading and outputting of the bit strings after each multiplication operation is performed. Two copies of this are made to handle register A as well as B.

Module: onebitreg.sv:

Inputs:	input logic Clk, Reset, Load,
	input logic D,
Outputs:	output logic Data_Out

Description: A default one bit register module is created to run on the positive edge of the clock. Instantiated in the top level to hold and assign the value of the X bit to a variable XVal.

Purpose: To be used to store and output the value of the MSB X.

Module: hex.sv:

Inputs:	input Clk, Reset_Clear, Run_Accumulate,
	input [7:0]SW,
Outputs:	output logic [7:0] Aval, Bval,
	output logic Xval,
	output logic [7:0] hex_segA,
	output logic [3:0] hex_gridA

Description: Implements a 4-digit hexadecimal display driver by using a counter to cycle through 4 digits, and selecting the appropriate pattern based on the input nibbles. Also contains a module within for nibble to hex.

Purpose: To display hexadecimal numbers on the 4-digit 7-segment display

Module: sync.sv:

Inputs:	input Clk, Reset_Clear, Run_Accumulate,
	input [7:0]SW,
Outputs:	output logic [7:0] Aval, Bval,
	output logic Xval,
	output logic [7:0] hex_segA,
	output logic [3:0] hex_gridA

Description: Takes asynchronous signals and holds them in a flip flop allowing for them to be implemented on a clock signal. This results in the signals being converted from asynchronous to synchronous signals.

Purpose: To eliminate asynchronous components such that the entire circuit remains synchronous to reduce the chance of static hazards arising from timing issues and ensure proper signal handling in FPGA designs.

<u>State Diagram for Control Unit</u>

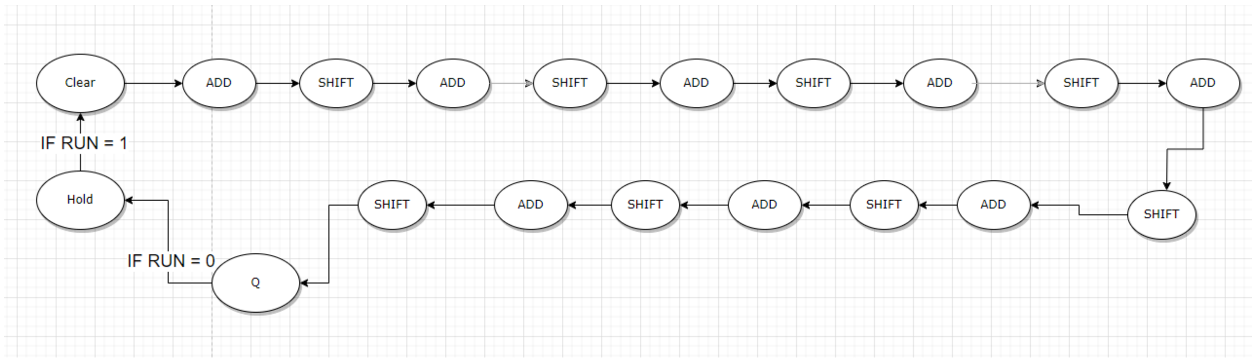

Figure 3 : FSM Diagram

<u>Annotated pre-lab simulation waveforms.</u>

The operation for x02 multiplied by x02 (2 x 2 = 4) is done below:

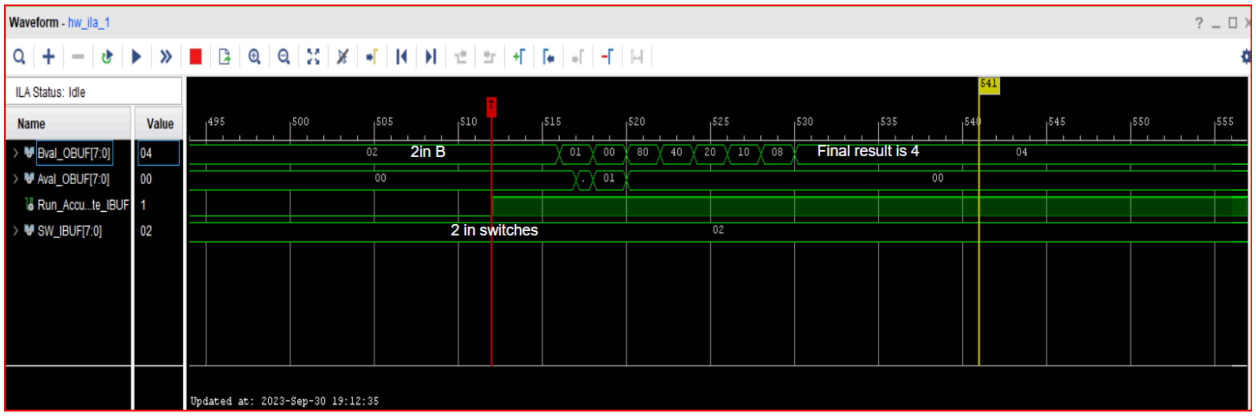

Figure 4: Annotated sim waveform

The operation for x02 multiplied by xFE (2 x (-2) = -4) is done below:

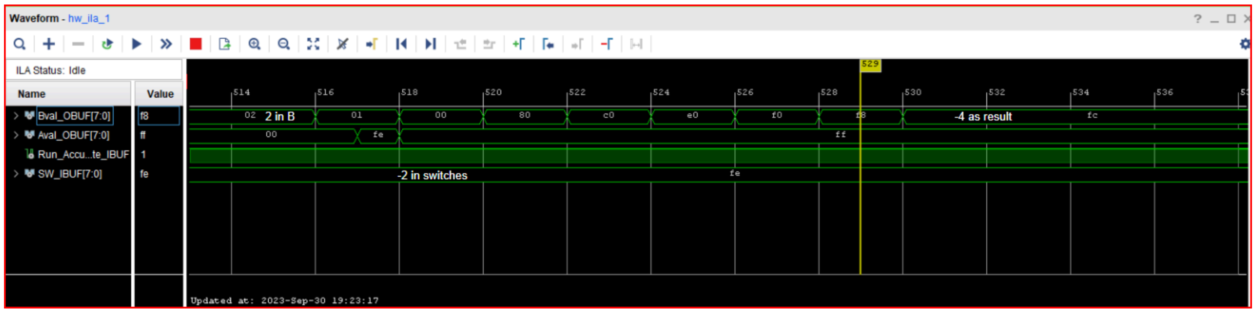

Figure 5: Annotated sim waveform

The operation for xFE multiplied by x02 ((-2) x 2 = -4) is done below:

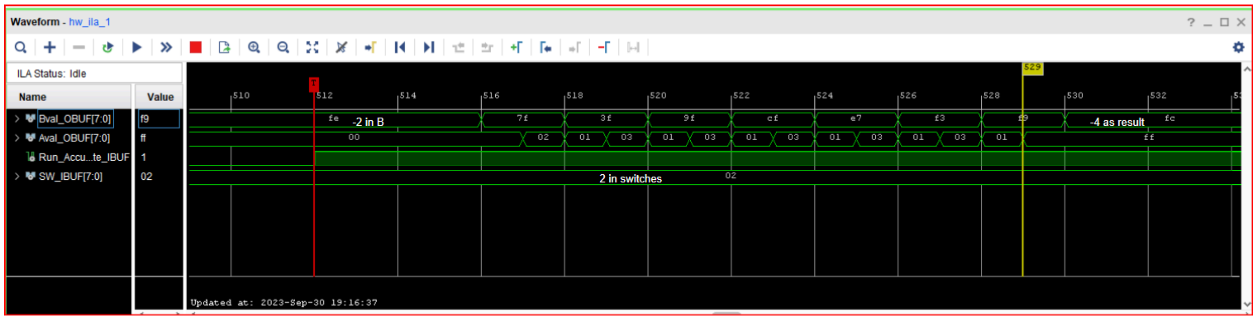

Figure 6: Annotated sim waveform

The operation for xFF multiplied by xFF ((-1) x (-1) = 1) is done below:

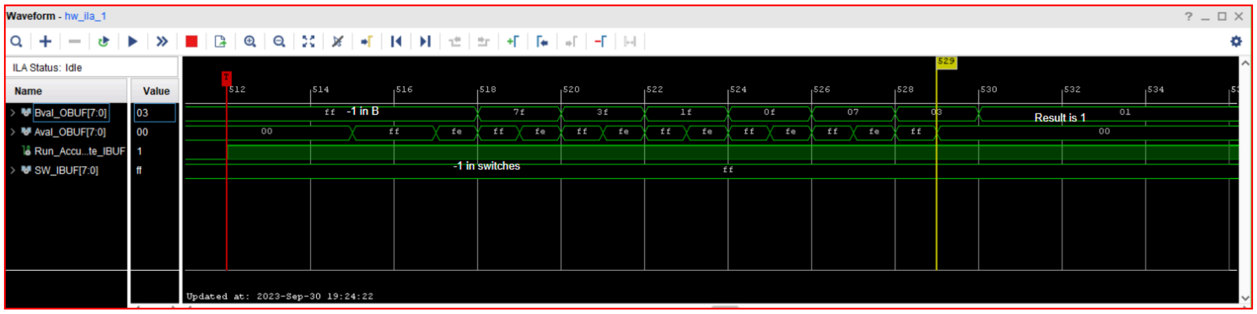

Figure 7: Annotated sim waveform

Answers to post-lab questions

1) FIlled in design and statistics table

Note that the WNS is 5.677, and the frequency is calculated by 1/(10 - wns).

LUT	1126
DSP	0
Memory (DRAM)	0
Flip-flop	1844

Latches*	0
Frequency	0.231 Hz
Static Power	0.006W
Dynamic Power	0.072W
Total Power	0.077W

Question: 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:

One way to increase the maximum frequency of this multiplier is using our knowledge from the previous lab to implement a CSA to perform the add operation on 8 bits in register A, instead of the CRA that is used in this code.

To decrease the total gate count, implement a serial addition algorithm, by using one adder to add the multiplicand as many times as the multiplier. We use this algorithm in ECE120 to implement a multiplier in LC3. This being a much simpler but less time efficient process.

2) Make sure your lab report answers at least the following questions
1) • What is the purpose of the X register? When does the X register get set/cleared? • 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 X register is used to take the MSB of the 8 bit string A, B, and assign it to a variable XVal, so that we have a means of storing XVal to be used in the arithmetic right shift after every ADD state. The X register gets set when the MSB of the 8 bit string AB is 1 after the ADD state, and X register gets cleared when the MSB is 0.

If we used the carry-out of the 8-bit adder instead of the output of the 9-bit adder, there will be cases where the MSB is 0 and the carryout is 1, or the MSB is 1 and there is no carryout, which will lead to an incorrect sign extension and present the wrong X value, and subsequently this would affect the shift states where the wrong value is shifted into the new AB string.

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

As the result of successive multiplications the value in B will be larger and larger. There will come a point where the implemented algorithm will fail if we keep multiplying the already large B value with an A value that is large as well. This result needs more than 16 bits to represent.

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

The biggest advantage of the implemented algorithm compared to the paper method would be the speed of processing, as we use two register (9 bit and 8 bit) to compute the additions during the process, compared to the paper method of calculating all the multiplicands one by one, and then adding them up at the end of the process. This involves Carry bits which are extra logic to take into account.

However, the disadvantage is that the implemented algorithm is more complex to implement and requires more gates and registers (as well as FSM states) resulting in more LUTs being used in the FPGA. Additionally, the implemented algorithm fails for continuous multiplication due to the 16 bit limit.

Conclusion

We experienced several bugs that we had to fix. Firstly, we did not initially have a default case in our FSM, which led to some of our states malfunctioning. However, adding in this default case fixed our FSM. Secondly, we had a few inferred latches show up due to some if statements which did not comprehensively include every case, since we did not have an else statement. However, we fixed this by making sure all our if statements had all the cases covered.

In conclusion, we managed to build a 8 bit multiplier using the add-shift algorithm using an Moore machine FSM, and we found that the multiplication works for all test cases, but fails after many clicks where the result is too large for 8 bits. Thus, a potential future experiment we can conduct is to

In conclusion, we implemented a 16 bit adder in 3 different ways, using CRA, CLA and CSA. We observed that the CLA and CSA are both significantly faster than the CRA, but also use up more gates and are more complex to implement. For a potential future experiment, we might want to figure out a better way to implement the CSA than the 4x4 design that we used in this lab, which clearly introduces more delays than we need with alternative designs.