ECE 385

Fall 2022

Lab 4

An 8-bit Multiplier in SystemVerilog

Zhuoyuan Li, Shihua Cheng

Tianhao Yu

Introduction

An 8-bit multiplier is built using SystemVerilog for this lab. The multiplier circuit accepts a multiplicand input SW[7:0] from the switches. It accepts a control signal from a button to clear internal registers and load value from the switches. After the value is loaded, the multiplier is provided from the switches. Then, it performs the multiplication of the loaded value and the new value from the switches on another different button press, and stores the 16-bit result into two 8-bit registers. The 16-bit result is displayed on the LEDs. Both the input and the output are signed integers. It does exactly one multiplication on one button press, and it can do consecutive multiplications.

Pre-lab question

Following is a table showing the procedure of 11000101 (S) * 00000111 (B).

Function	X	A	B	M	Comments for the next step
Clear A, Load B, Reset	0	0000 0000	*00000111*	1	Since M=1, S will be added to A.
ADD	1	1100 0101	*00000111*	1	Shift XAB by one bit after ADD
SHIFT	1	1110 0010	1 *0000011*	1	Add S to A since M=1.
ADD	1	1010 0111	1 *0000011*	1	Shift XAB by one bit after ADD
SHIFT	1	1101 0011	11 *000001*	1	Add S to A since M=1.
ADD	1	1001 1000	11 *000001*	1	Shift XAB by one bit after ADD
SHIFT	1	1100 1100	011 *00000*	0	Do not add since M=0. Shift XAB.
SHIFT	1	1110 0110	0011 *0000*	0	Do not add since M=0. Shift XAB.
SHIFT	1	1111 0011	00011 *000*	0	Do not add since M=0. Shift XAB.
SHIFT	1	1111 1001	100011 *00*	0	Do not add since M=0. Shift XAB.
SHIFT	1	1111 1100	1100011 *0*	0	Do not SUB since M=0. Shift XAB.
SHIFT	1	1111 1110	01100011	1	8th shift done. Stop. 16-bit product in AB.

Written description and diagrams of multiplier circuit

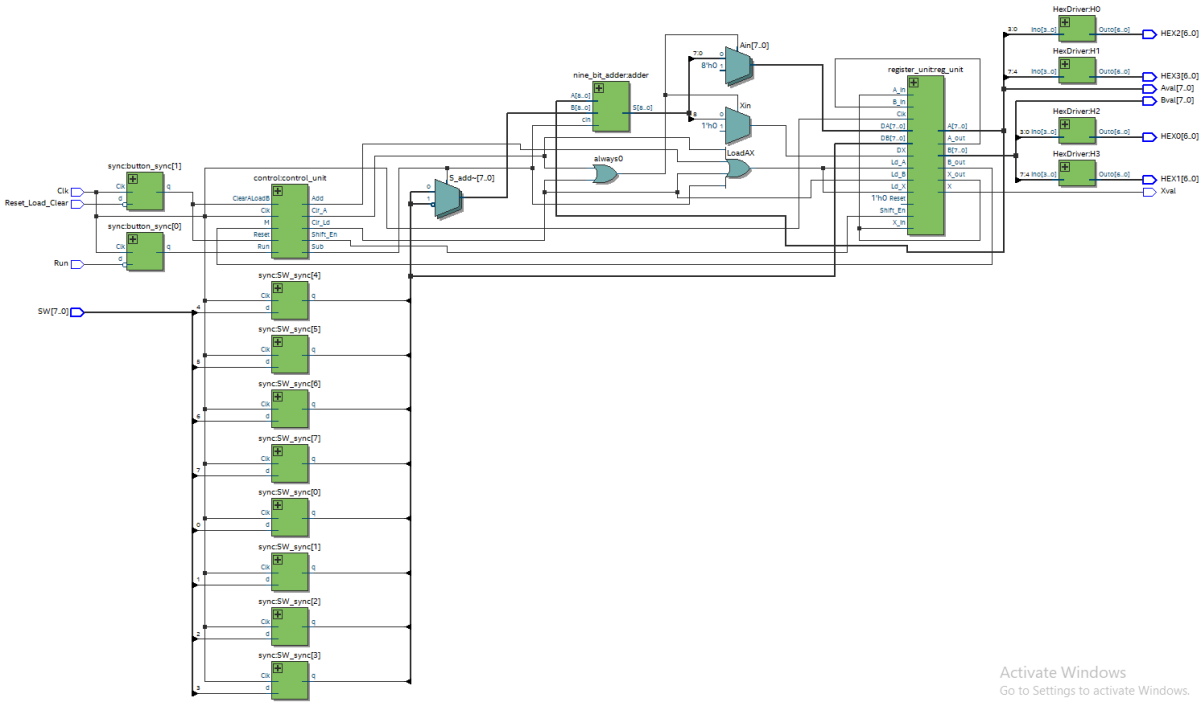
Summary of operation

The operands are loaded in series. The 8-bit multiplicand is loaded by adjusting the lower 8 switches on the board to provide the value and then pressing Key 0 (Reset_Clear_Load) on the board. At this point, the program has been reset and the value of the multiplicand has been loaded to an 8-bit register (Register B). Then, adjust the switches again to provide the multiplier. Press Key 1 (Run) button to calculate the 16-bit multiplication result. During the execution, the switches should not be changed (which is unlikely since the execution is fast), because the multiplier is not stored in any internal registers but is read from the switches. Two 8-bit registers (A, B) and one 1-bit register (X) are used for calculation. The final 16-bit result will be stored with its higher 8 bits in Register A and lower 8 bits in Register B. “Run” can be pressed multiple times to perform consecutive multiplication. The value will be passed to the on-board LEDs through 4 HexDrivers to visualize the result.

The details of the calculation after the two operands are loaded and “Run” is pressed is explained below. Register B (8-bit) stores the first operand. Register A (8-bit) and the second operand from the switches (8-bit) are connected to a 9-bit adder as inputs after sign-extension. The highest bit of the result of the 9-bit adder is connected to a 1-bit register, X, to keep track of the sign of the result. The remaining 8 bits are passed to Register A. A finite state machine is used to control the load signals of register A and X to perform an A <= A + SW, as well as to arithmetically right shift the value XAB. Register A and X will only load the result from the adder if the least significant bit, M, of register B is a 1. Then the 17-bit value, XAB, will perform an arithmetic right shift. This process will be repeated for 7 times under the control of the FSM, and then the adder will do the subtraction A <= A - SW by changing the carry in of the adder to 1 and inverting all the SW bits. A and X will load this value only if M is 1. Afterwards, the final (8th) arithmetic right shift of XAB is performed, and the final result is stored in register A and B. A contains the higher 8 bits, and B contains the lower 8 bits. Therefore, the control unit has 8 shift states that do exactly 8 arithmetic right shifts, 7 Add states that do A <= A + SW only when M=1, and 1 Sub state that does A <= A - SW only when M=1 before the final shift state. Also, it has an initial and a final state where all control signals are set to zero. The initial state will go to the first Add state only when the signal “Run” is high, and the final state will go back to the initial state only when the signal “Run” is low. This is to avoid multiple rounds of calculations during one button press.

Top Level Block Diagram

The top level block diagram of the multiplier is shown on the next page.



Written Description of .sv Modules

Module: control

Inputs: Clk, Reset, ClearALoadB, Run, M

Outputs: Clr_Ld, Shift_En, Add, Sub, Clr_A

Description: This is the control unit consisting a FSM to control which state of the multiplication is on and output the corresponding control signals including Clr_Ld, clear register A and load register B by switch, Shift_En (enable shift), Add(an addition operation between register A and switch), Sub(a subtraction between register A and switch), and Clr_A(clear register A without loading register B). The transition between each state happens on the rising edge of Clk. Several states require input signals like Run in order to transit to the next state. It returns to the original state if reset is inputted.

Purpose: The control module is used to control which state of the multiplication is on and output the corresponding control signals for each state.

Module: nine_bit_adder

Inputs: [8:0] A, B, cin,

Outputs: [8:0] S, cout

Description: This is the 9 bit adder which adds or subtracts sign extended register A and switch. It consists of two 4-bit carry ripple adders and a full adder. When a subtraction is necessary, cin = 1, and the switch will be negated; otherwise cin will be 0. The output will be a 9 bit result which will later be loaded into A and X in the processor unit.

Purpose: Conduct addition or subtraction of register A and register switch.

Module: ripple_adder_4

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

Outputs: cout, [3:0] S

Description: Just a simple 4 bit ripple adder which consists part of the 9 bit adder module.

Purpose: This 4 bit ripple adder consists part of the 9 bit adder.

Module: full_adder

Inputs: cin, A, B

Outputs: cout, S

Description: Just a simple 1 bit full adder which consists part of the 9 bit adder module and the 4 bit ripple_adder..

Purpose: This full adder consists part of the 9 bit adder and the 4 bit ripple_adder.

Module: register_unit

Inputs: Clk, Reset, X_In, A_In, B_In, Ld_X, Ld_A, Ld_B, Shift_En, [7:0] DA, [7:0] DB, DX,

Outputs: A_out, B_out, X_out, [7:0] A, [7:0] B, X;

Description: The register unit consists of two positive-edge triggered synchronous 8-bit registers for A and B with a 1 bit register X. It outputs the least significant bit of A, B and X as A_out, B_out, X_out. When conduction addition or subtraction, its output is [7:0] A. It will shift the registers once while shift_En is 1, so that A_out will become B_In, X_out will become A_In, and X_In will remain itself. During the loading process, Ld_X, Ld_A and Ld_B are three signals which control which register should conduct such loading, and [7:0] DA, [7:0] DB, DX are the actual data each register should load.

Purpose: This module is created to store A, B and X and operate their load operation and shift operation.

Module: processor

Inputs: Clk, Reset_Load_Clear, Run, [7:0] SW,

Outputs: Xval, [7:0] Aval, Bval, [6:0] HEX1, HEX0, HEX3, HEX2

Description: top level entity that creates instances of other modules and connects them together.

Purpose: Serve as the top level entity that connects other modules.

Module: reg_8.sv

Inputs: [7:0] Din, Clk, Load, Reset

Outputs: [7:0] Dout

Description: This is a positive-edge triggered 8-bit register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on the positive edge of Clk.

Purpose: This module is used to create the registers that store operands A and B in the adder circuit.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: HexDriver converts a 4-bit unsigned integer (0-F) to hexadecimal form that is mapped to the LED on the FPGA board.

Purpose: The content of the register can be visualized on the FPGA board.

Module: sync

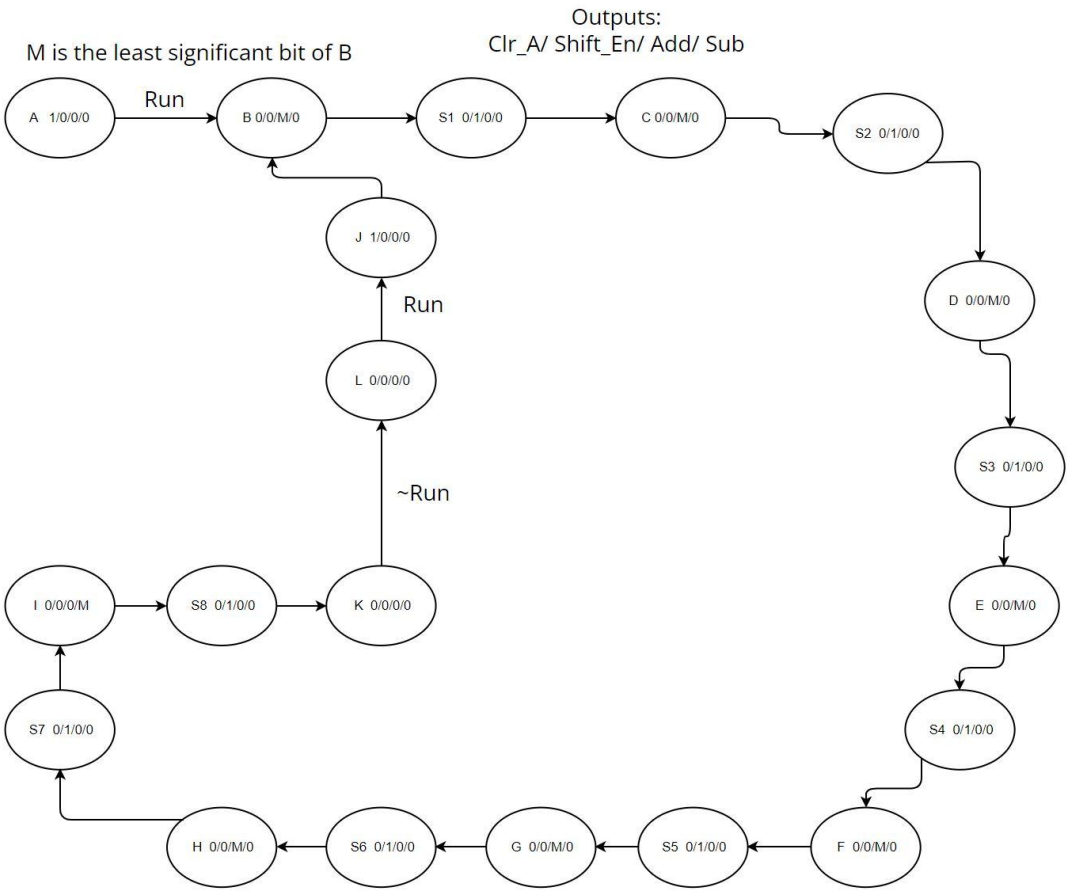
Inputs: Clk, d,

Outputs: logic q

Description: a module that synchronizes the asynchronized switch, button, and clock, so that, for example, after the run button is pressed, the run signal will become high at the next rising edge of the Clk.

Purpose: synchronize the switch, clock, and buttons.

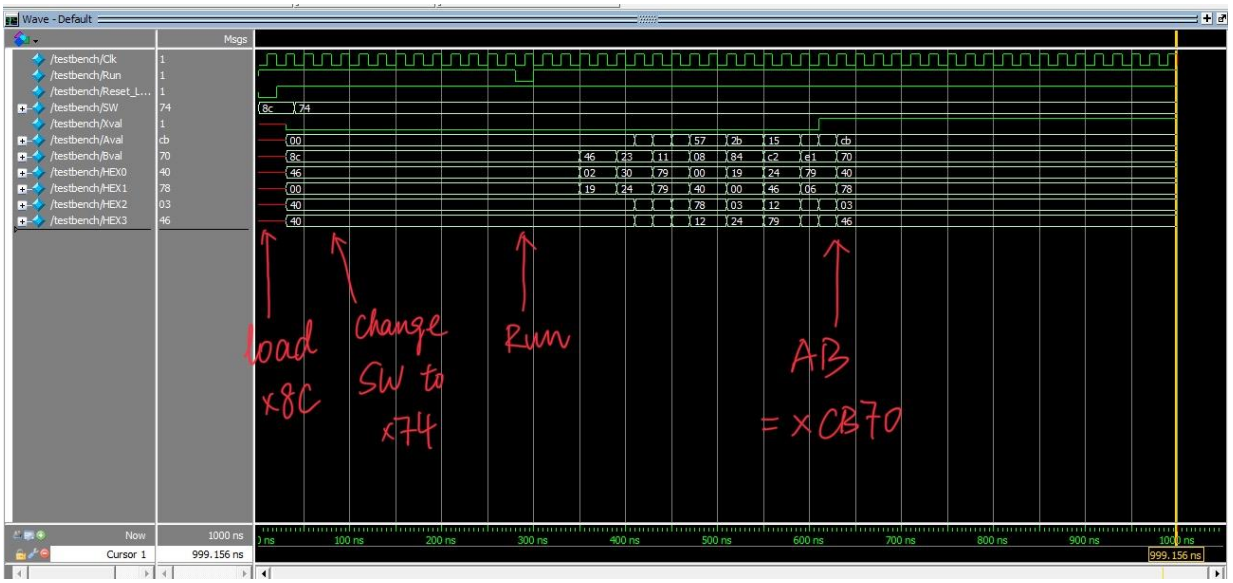
State Diagram for Control Unit



Annotated pre-lab simulation waveforms

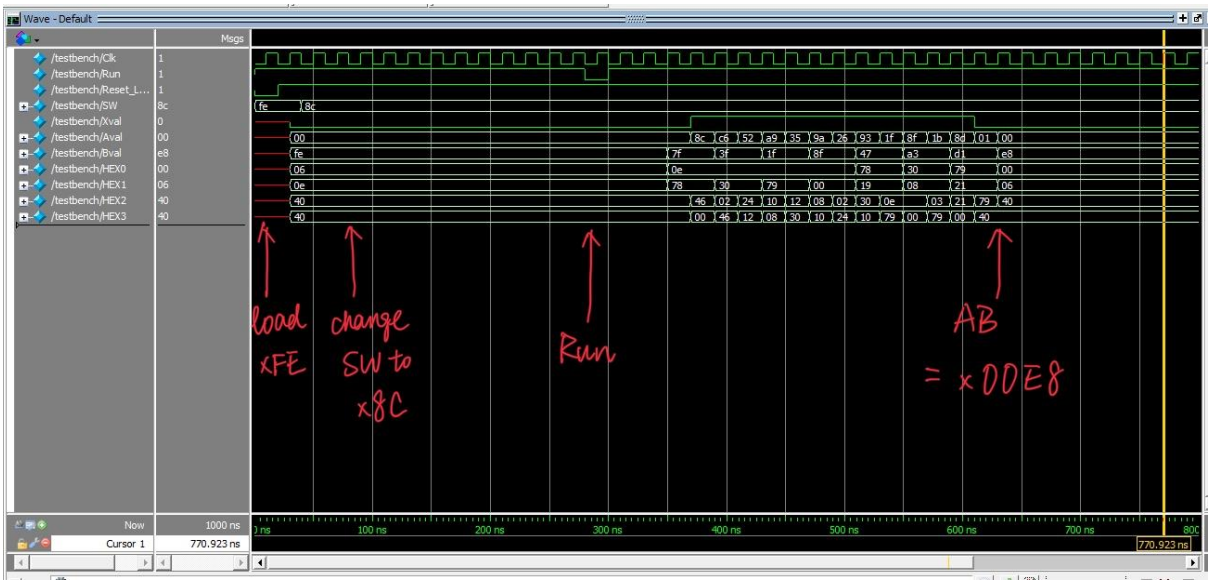
8’h74 * 8’h8C: (116 * -116 = -13456 = 16’hCB70)

8’h8C * 8’h74: (-116 * 116 = -13456 = 16’hCB70)



8’h74 * 8’h47: (116 * 71 = 8236 = 16’h202C)

8’hFE * 8’h8C: (-2 * -116 = 232 = 16’h00E8)



Answers to post-lab questions

Following is the design resources & statistics table of the circuit in Lab 4.

	8-bit Multiplier
LUT	553
DSP	0
BRAM	4480 bits
Flip-Flop	697
Max Frequency	79.34MHz
Static Power	90.00mW
Dynamic Power	7.31mW
Total Power	110.32mW

In order to further optimize the area of the circuit, one possible improvement is to redesign the 1-bit X register. In our design, the 1-bit register is designed to have the exact same functionalities as the 8-bit registers. In fact, the shift functionality is the same as parallel load for the 1-bit register since there is only 1 bit. We can just use a latch for the X register to save some area.

To increase the max frequency, we can use a faster adder with lower delay. In our design, we used a 9-bit ripple adder. We can use a 9-bit carry select adder which is much faster than the ripple adder. A good configuration would be 4-5 grouping. A 4-bit ripple adder and two 5-bit ripple adders are needed, a 2:1 MUX is used to let the carry out of the 4-bit ripple adder choose the output of the 5-bit ripple adders. This will slightly increase the area but also increase the max frequency of the circuit.

The purpose of the X register is to store the sign bit of the result from adder (A + SW). It never changes during shift states because the shifts are arithmetic. It is cleared at the beginning of the calculation. It can only be changed by the output of the adder. If the least significant bit of B is 1, then the output of the adder gets loaded to X and A. If the highest bit of the output is 1, then X is set to 1. Otherwise, it is set to 0. It is necessary to use a register instead of directly connecting the highest bit of the 9-bit adder to the shift-in bit of A, because after the adder output is loaded to A, the adder value is immediately updated to (A + SW), so the previous 9th bit immediately disappears. Therefore, it is necessary to store it in X.

If we use the carry out of a 8-bit adder instead of a 9-bit adder, then the sign bit of A + SW will be incorrect. The carry out can be zero even if the addition result is negative (highest bit is 1), and during next shift, the whole result will be incorrect since a zero (carry out) is shifted in instead of a one given that AB should be negative. For example, during the start of the calculation, A = 8'h00, S = 8'hff, B = 8'h11, then the first state should be ADD. The addition A <= A + S is equal to 8'hff with a carry out equal to zero, so the highest bit will be zero

instead of one after one shift. This is not true since the result in A should be a negative number (highest bit is 1). Using a 9-bit adder avoids this problem because the operands are first sign extended.

The limitation of continuous multiplications is that the multiplier treats inputs as 8-bit signed integers, and outputs as 16-bit signed integers. In continuous multiplications, the output is then used as an input, but no conversion is made from 16-bit signed integer to 8-bit signed integer. For example, when the output of a multiplication is 0x0080, it should be interpreted as 128 in decimal, since it is a 16-bit signed integer. However, when it is used again as an input in the next multiplication, it will be interpreted as an 8-bit signed integer, so it will be interpreted as -64 in decimal. In such circumstances, the implemented algorithm will fail.

As an advantage, the implemented multiplication algorithm is more accurate compared to the pencil-and-paper method which may come up with inaccurate answers if miscalculation occurs, such as when the multiplier is a negative number. For example, when the multiplier is 8'hff (-1), the pencil-and-paper method will treat it as 15 in decimal and output the result 15 times multiplicand instead of -1 times multiplicand. As a disadvantage, the implemented multiplication algorithm, when conducting consecutive multiplication, may encounter overflow when the result of last multiplication cannot be represented by 8 bits, a case which will not happen in pencil and paper method. For example, when the output of one execution is 0x0080 (128 in decimal), our implementation will incorrectly use it as -64 for the next run, but the pencil-and-paper method will not.

Conclusion

In conclusion, the 8-bit multiplier that we designed in SystemVerilog worked as expected both in simulation and on the FPGA board. It successfully performs the multiplication of two signed 8-bit integers and outputs the signed 16-bit integer result as desired. The lab manual and instructions were clear and helpful.