ECE 385

Spring 2024

Lab 4

Ziheng Li (zihengl5)

Xin Yang (xiny9)

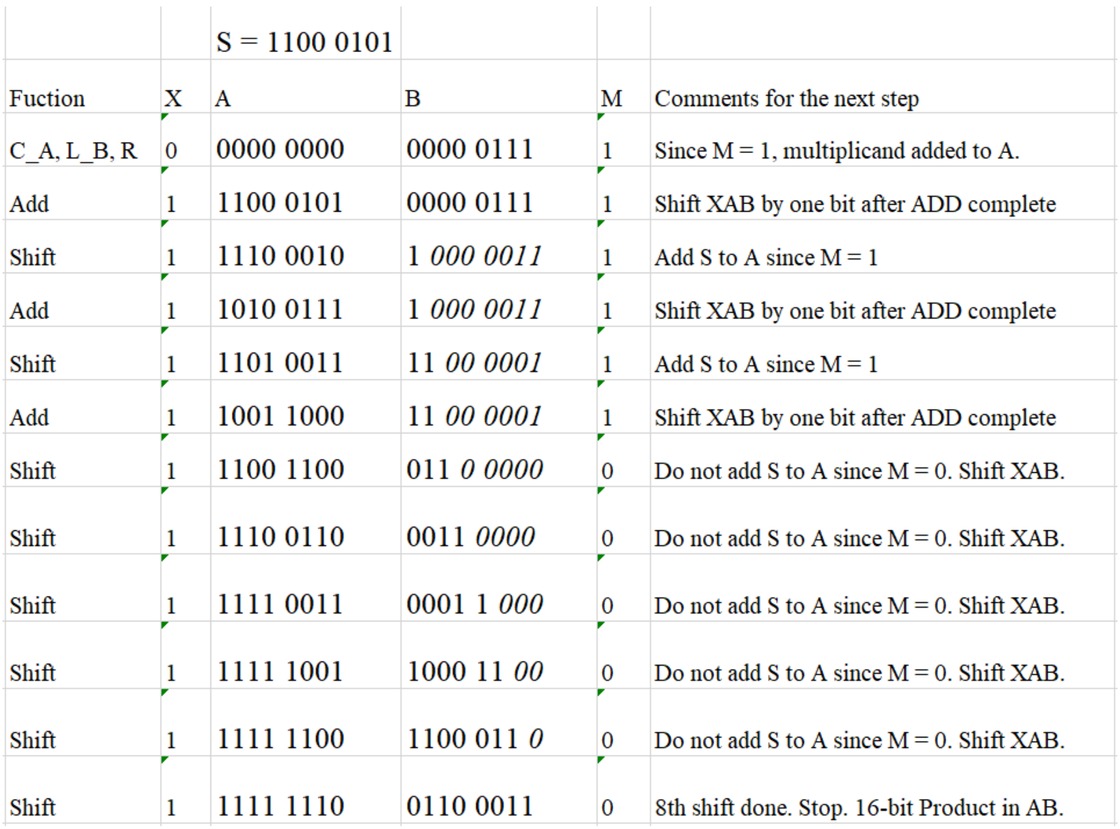
Section AL1

Prof. Zuofu Chen

Introduction

In this experiment, we design a multiplier in SystemVerilog for 8-bit 2's complement numbers by only using logical operation. This experiment utilize the add-shift algorithm, similar to the traditional pencil-and-paper multiplication method, with a specific emphasis on handling 2's complement numbers based on their sign bits. This hands-on experiment aims to deepen understanding of digital logic design and its applications in computer engineering, particularly in how arithmetic operations worked at the hardware level.

Pre-lab question



		S = 1100 0101			
Fuction	X	A	B	M	Comments for the next step
C_A, L_B, R	0	0000 0000	0000 0111	1	Since M = 1, multiplicand added to A.
Add	1	1100 0101	0000 0111	1	Shift XAB by one bit after ADD complete
Shift	1	1110 0010	1 *000 0011*	1	Add S to A since M = 1
Add	1	1010 0111	1 *000 0011*	1	Shift XAB by one bit after ADD complete
Shift	1	1101 0011	11 *00 0001*	1	Add S to A since M = 1
Add	1	1001 1000	11 *00 0001*	1	Shift XAB by one bit after ADD complete
Shift	1	1100 1100	011 *0 0000*	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	1110 0110	0011 *0000*	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	1111 0011	0001 1 *000*	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	1111 1001	1000 11 *00*	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	1111 1100	1100 011 *0*	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	1111 1110	0110 0011	0	8th shift done. Stop. 16-bit Product in AB.

Fig.1 Annotated multiplication process

Summary of operation

The operands are loaded into the circuit through a two-step process. First, we set the switch to the number for multiplier and press Reset_Load_Clear to load the value into register B. The Reset_Load_Clear button also clears the X and A registers. Next, the switches are set to represent the multiplicand, and the Run button is pressed to compute the result. Once the Run signal is triggered, the circuit completes the multiply operation autonomously, regardless of the status of the Run signal. The operation uses an add-shift algorithm, similar to the pencil-and-paper method, but adapted for handling 2’s complement numbers. The algorithm involves sign extending the A values to 9 bits, summing them, and storing the result in XA. The circuit performs right shifts and add/subtract operations based on the multiplier's LSB (M) and the number of shifts performed. After the multiplication process finished, the result will stored in AB and shows on the hex display. Moreover, it is capable of doing consecutive multiplication by modifying the switch and pressing Run again. the circuit will clean XA and multiply B values with switches value.

Top Level Block Diagram

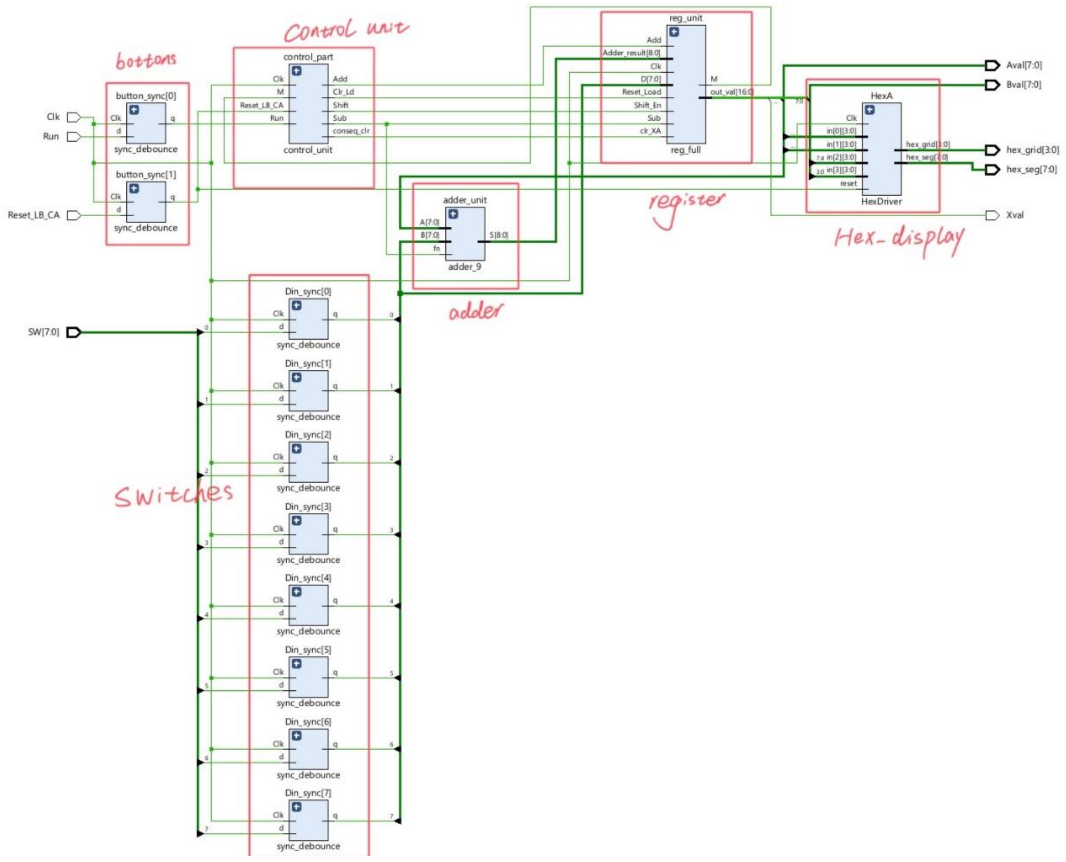


Fig.2 Top level block diagram

Written Description of .sv modules

Module: full_adder.sv
Inputs: x, y, z
Outputs: s, c
Description: Basic 1-bit full-adder module.
Purpose: Produce binary summation s of x and y with carry-in z and output carry-out bit c.

Module: adder_9.sv
Inputs: [7:0] A, [7:0] B, fn
Outputs: [8:0] S
Description: This module implements a 9-bit adder using full_adder.
Purpose: Produce binary summation of sign extended A and B with a XOR select/carry-in fn.

Module: full_adder_9_bits.sv
Inputs: [8:0] A, [8:0] B, c_in
Outputs: [8:0] S
Description: First, B was XORed with the function selector fn (0 for add, 1 for subtract) to perform add/subtract. Then both A and XORed-B were extended into 9-bit and pass in to full adder with fn as carry in, as shown below.
Purpose: Produce binary results for A + B or A – B, depends on fn value.

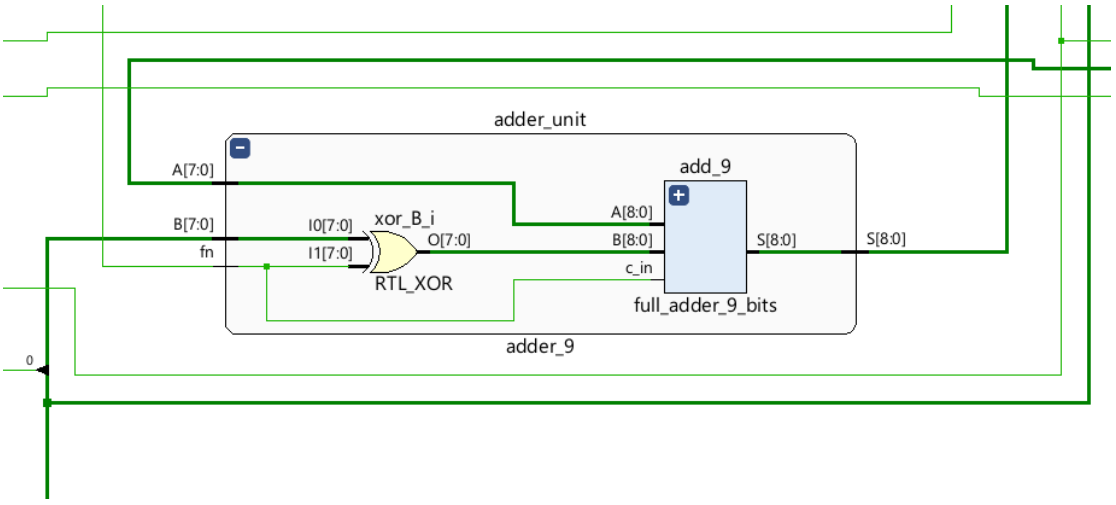


Fig.3 Adder unit diagram

Module: control_unit.sv
Inputs: Clk, Reset_LB_CA, Run, M
Outputs: Clr_Ld, Shift, Add, Sub, conseq_clr
Description: This is the control unit for the whole circuit. This module defines the FSM and state transition logic with 2 blocks of always statements.
Purpose: Output the 5 signals to control the operation of other modules. See **Control Unit** for details

Module: HexDriver.sv
Inputs: clk, reset, in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The module uses a loop to instantiate four nibble_to_hex converters, mapping each of the 4-bit inputs to their corresponding 7-segment display patterns. Then choose the hex_grid to determine which of the four LEDs is active.
Purpose: Convert binary into hexadecimal and then display them on the LEDs.

Module: reg_full.sv
Inputs: Clk, Reset_Load, Shift_En, Add, Sub, [7:0] D, [8:0] Adder_result, clr_XA
Outputs: M, [16:0] out_val
Description: This module implements a 17-bit register. When **Shift** is high, the register will right shift 1 bit. When **Add** or **Sub** is high, the register [16:8] will store **Adder_result**. When **Reset_Load** is high, the [16:8] will be clear and [7:0] will store the input **D**.
Purpose: Stores XAB value.

Module: Synchronizers.sv
Inputs: Clk, d
Outputs: q
Description: This module takes the signal d from the fpga button and switch. And outputs q for other modules to use. Firstly, it flops the input d twice. And then change the button when 2^COUNTER_WIDTH stable input cycles are recorded.
Purpose: It is used as a debouncer and a synchronizer for push buttons and switches.

Module: top_level.sv
Inputs: Clk, Reset_LB_CA, Run, [7:0] SW
Outputs: Xval, [7:0] Aval, [7:0] Bval, [7:0] hex_seg, [3:0] hex_grid
Description: This is the top-level module for the multiplier. All three inputs will goes into control unit, and control unit will output corresponding signal to register unit. The result of register unit will show up on hex display and LED.
Purpose: This module is used as a combination of all other modules in the project for an 8-bit multiplier.

Control Unit

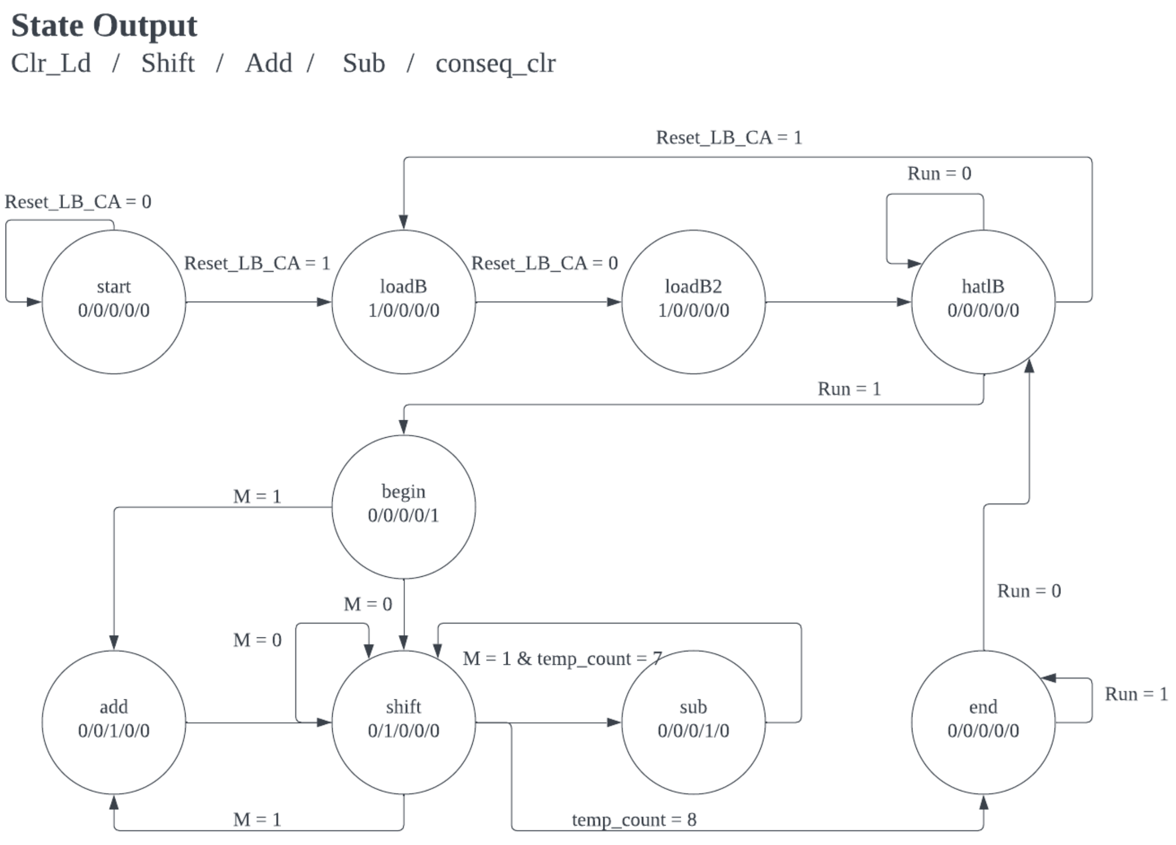


Fig.4 State transaction

We utilize a Moore machine in this lab. For each state, it have 5 outputs, Clr_Ld, Shift, Add, Sub, and conseq_clr. In order to make sure the state only produce 8 shifts in total, we also employee a counter inside each state.

s_start: This state is the beginning state, whenever the multiplier is powered, this is the first state reached. If Reset is high, it will go to s_loadB state. All outputs for this state is 0.

s_loadB: This state aim to load switches value into register B. Clr_Ld signal will be high for this state. If Reset is 0, it will go to s_loadB2 state.

s_loadB2: This state produce the same output as s_loadB. The purpose for this state is to maintain Clr_Ld signal high for two clock cycles. The next state is s_haltB. (See Bug encountered for details)

s_haltB: At this point, register B should have multiplier value. It will goes to s_begin if Run is 1, self-looping otherwise. All outputs for this state is 0.

s_begin: This is the beginning state for actual multiplying process. If last digit of B is 1, it goes to s_add, otherwise, it goes to s_shift. Conseq_clr signal is high for this state to clear XA value before next consecutive multiply.

s_shift: Shifting state will goes to s_end if the counter reaches 8. If last digits (M) of B is 1, it will goes to s_add or s_sub (if counter value is 7). If M is 0, it will keep shifting. The counter will increment at this state.

s_add: Adding state will produce add signal to adder to perform adding function. After each adding, it will goes to shift state.

s_sub: Subtracting state will produce subtract signal to adder to perform subtraction, it goes to shift state no matter what.

s_end: Since the Run signal will be high for a long time (compare to clock speed), this state make sure next multiplication does not happen until we press Run again. If Run back to 0, it goes to s_haltB to perform another new/consecutive multiplication.

Simulation waveforms

+ * +

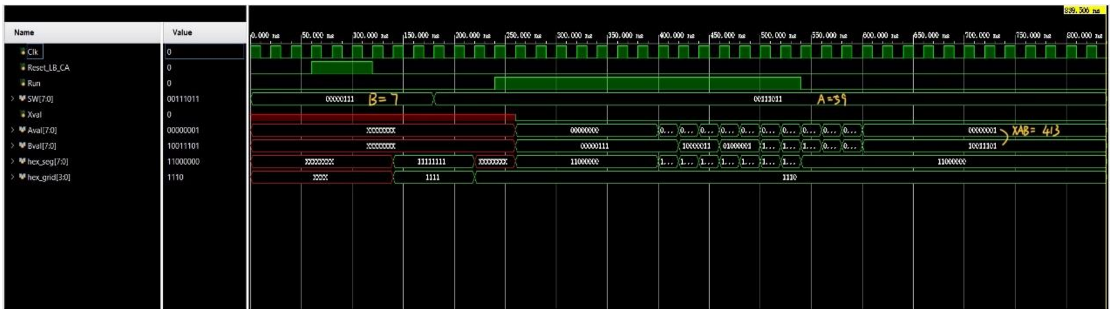


Fig.5 Simulation for +*+

This positive * positive situation. First, we set the switches into 00000111 (7), and set Rest_LB_CA to high. We can see the value was loaded into Bval. Then the switches was set to 00111011 (59), and value was loaded into Aval. Finally, we press Run, and results are presented in XAB as 0 00000001 10011101 (413).

+ * -

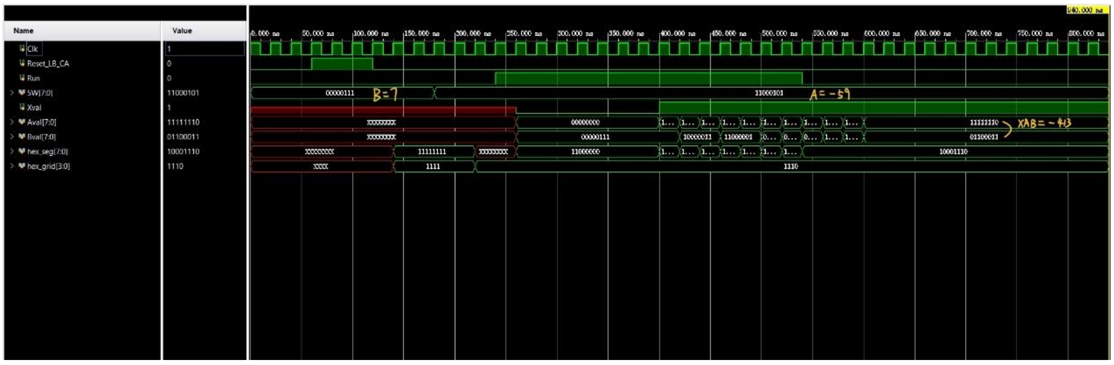


Fig.6 Simulation for +*-

This positive * negative situation. First, we set the switches into 00000111 (7), and set Rest_LB_CA to high. We can see the value was loaded into Bval. Then the switches was set to 11000101 (-59), and value was loaded into Aval. Finally, we press Run, and results are presented in XAB as 1 11111110 01100011 (-413).

- * +

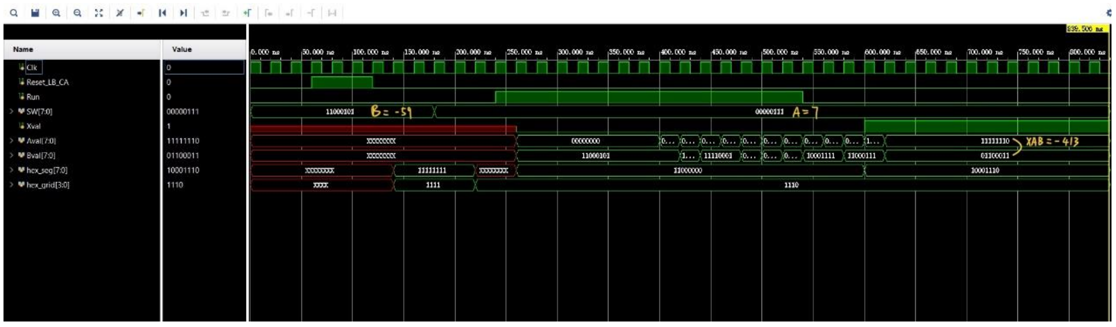


Fig.7 Simulation for -*+

This negative * positive situation. First, we set the switches into 11000101 (-59), and set Rest_LB_CA to high. We can see the value was loaded into Bval. Then the switches was set to 00000111 (7), and value was loaded into Aval. Finally, we press Run, and results are presented in XAB as 1 11111110 01100011 (-413).

- * -

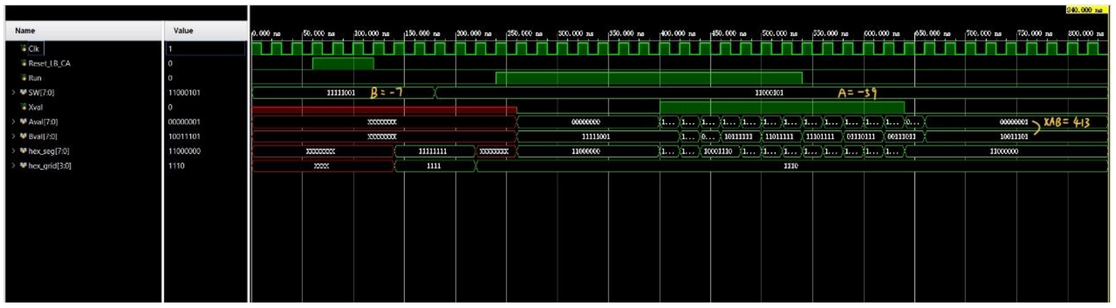
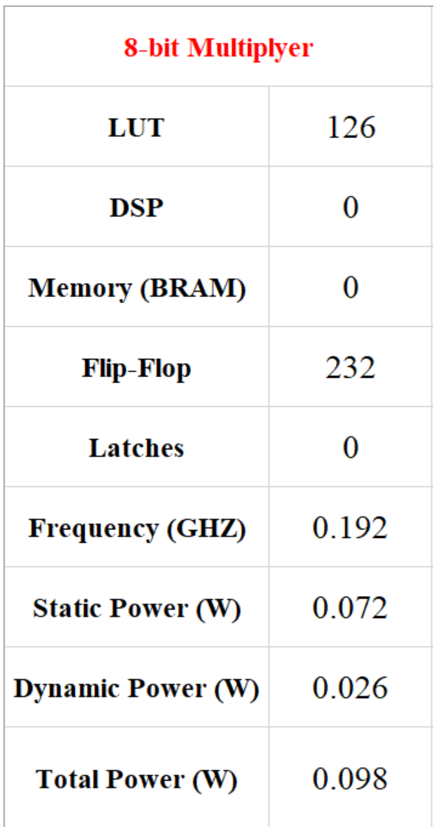


Fig.8 Simulation for -*-

This negative * negative situation. First, we set the switches into 11111001 (-7), and set Rest_LB_CA to high. We can see the value was loaded into Bval. Then the switches was set to 11000101 (-59), and value was loaded into Aval. Finally, we press Run, and results are presented in XAB as 0 00000001 10011101 (413).

Post Lab Question

Refer to the Design Resources and Statistics in IVT and complete the following design statistics table.



8-bit Multiplyer	
LUT	126
DSP	0
Memory (BRAM)	0
Flip-Flop	232
Latches	0
Frequency (GHZ)	0.192
Static Power (W)	0.072
Dynamic Power (W)	0.026
Total Power (W)	0.098

Fig.9 Statistic 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.

In this design, we use the CRA for the adder unit. We can then improve the maximum frequency by changing this to the CSA algorithm. By separately computing sums and carry-outs and then selecting the correct one with latter carry in, adders in CSA can pre-calculate the result without waiting propagation delay.

Moreover, we might be able to merge some of our states in the control Unit, which will also simplify the intermediate logic and hence improve the speed.

What is the purpose of the X register? When does the X register get set/cleared?

The purpose of the X register is to hold the value of the sign of the result (most significant bit). When we shift the XAB value, the new X will hold its old value, and thus keeping the result's sign. X will be set whenever we have an addition of negative number or subtraction of a positive number. The X register is cleared at the beginning of each multiplication operation.

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

In this case, the sign of the intended result might be lost. For example, if we are adding 10000000 (-128) with 00000001 (1). Using the original X method, A will be extended to 110000000, and B will be XORed with fn and extended to 000000001. Adding them together, the result should be 11000001 (-127). However, if we use the carry out of the 8-bit adder, A value will be 10000000, B value will be 00000001, there is no carry out for this addition, and hence the result became 010000001 (129), which is incorrect. In summary, using the carry out cannot preserve the sign information and cause incorrect output.

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

The continuous multiplication will work if the 16-bit result could be truncated to 8-bit without changing its value. In other words this only works if the product is within the range [-128, 127].

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

The advantages of the implemented multiplication are higher computation speed compared to the pencil-and -paper method. In addition, we only need two 8-bit register and one single bit register to store the result. If we want to use the pencil-paper method, we have to keep track of all intermediate results, which requires much more registers. However, the disadvantage is that it is not as intuitive as the previous method since it contains several logic transitions including sign extension and shifting.

Bugs Encountered

During the design process of FSM in the control unit, we defined 1 state to loadB originally. However, the output of the debounced switch value could not be loaded correctly into the register within 1 clock cycle. Then we added this loadB2 state, which provides an additional clock cycle to make sure the value correctly loaded into registerB.

Conclusion

Our designed multiplier circuit successfully performs the multiplication and consecutive multiplication operation for two 8-bit 2's complement numbers using logical operations. Through the implementation of the add-shift algorithm in SystemVerilog, we were able to achieve accurate results for 4 types of input combinations. The circuit operates within the specified timing constraints and efficiently utilizes FPGA resources. One thing during this developing process is that we find that consecutive multiplication will fail if we do not clear the value of XA before the second operation.