ECE 385

Fall 2020

Experiment #5

An 8-Bit Multiplier in System Verilog

Name: Xiao Shuhong & Lu Yicheng
Lab Section: D225

1. Introduction

1.1. Summarize the basic functionality

for this lab, we design an 8-bit multiplier which accept two 8-bit 2's complement numbers and give their result in the form of 16 bits. single multiplication and consecutive multiplication can be realized using this circuit.

The whole circuit can be divided into serval part: register part, control part and I/O part. Main control input signals are reset, run and clearA_loadB. For reset being press, we clear all the contains in register and init our circuit; for run button, it should be press after both multiplicand and multiplier are loaded, every time we press run, it will only do one execution; for clearA_loadB, we use it to load our multiplier from switch.

A successful operation can be described as follow:

first, do reset to clear data from last multiplication; second, set multiplier B in switch and press clearA_loadB to load it into register B; then, set multiplicand using switch, press run to see result. If you want to do consecutive multiplication, you can just change the switch which represent A and press run again, otherwise, you can press reset to start a new multiplication or stop.

2. Pre-lab question

2.1. Rework the multiplication example in pre-lab question

Function	X	A	B	M	comments
Clear A, Load B	0	0000 0000	0000 0111	1	Since M=1, add S to A
ADD	1	1100 0101	0000 0111	1	Shift XAB by one bit
SHIFT	1	1110 0010	1000 0011	1	Add S to A
ADD	1	1010 0111	1000 0011	1	Shift XAB by one bit
SHIFT	1	1101 0011	1100 0001	1	Add S to A
ADD	1	1001 1000	1100 0001	1	Shift XAB by one bit
SHIFT	1	1100 1100	0110 0000	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	0001 1000	0	Do not add since M=0, shift XAB
SHIFT	1	1111 1001	1000 1100	0	Do not add since M=0, shift XAB
SHIFT	1	1111 1100	1100 0110	0	Do not add since M=0, shift XAB
SHIFT	1	1111 1110	0110 0011	0	All bit done, stop, 16-bit Product in AB

3. Written description and diagrams of multiplier circuit

3.1. Summary of operation

For an 8-bit multiplication, we need two 2’s complement 8 bits binary number as our multiplicand(A) and multiplier(B).

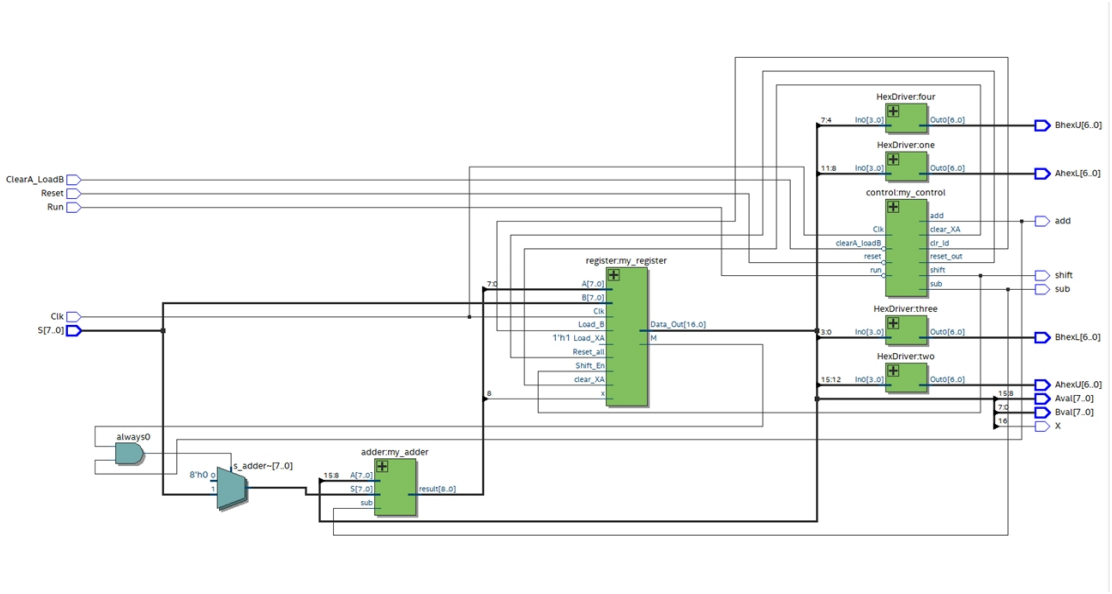
For the operands load, we first set our multiplier B in the switch, then press ClearA_LoadB button to load B from switch to our register B. Then, for multiplicand A, we just set it in the switch. That is all to finish our operands setting.

For computing, we use a state machine to control the specific operation. Except ClearA_LoadB, we have other three kinds of operations: ADD, SHIFT and SUB. Based on a algorithm very similar to paper-and -pencil method, we check the least significant bit in register B, we named it as M, if M=1, we will do ADD or SUB if 7 shift have already done, then follow a SHIFT. Otherwise M=0, we just do SHIFT. Then ADD operation first sign-extend number in register A (we call it A below) and number in switch (we call it switch below) to 9 bits, then add then and store the result into register X and A, with X another 1 bit register. For SHIFT operation, we see XAB as one whole part and do arithmetically right-shift and store the result back to XAB. After 7 SHIFT happens, if M=1, we will do SUB, similar as before we fist sign-extend A and S to 9 bits, then subtract S from A, store result back to XA.

After all the above operation finish, our result of B multiply A is store in register AB as a 16-bit 2’s complement number.

One more thing to mention here is we modified the adder from last lab to count 7 SHIFT operations and decide whether to use SUB or not.

3.2. Top Level Block Diagram



3.3. Written description of .sv Modules

3.3.1 list all modules and RTL views

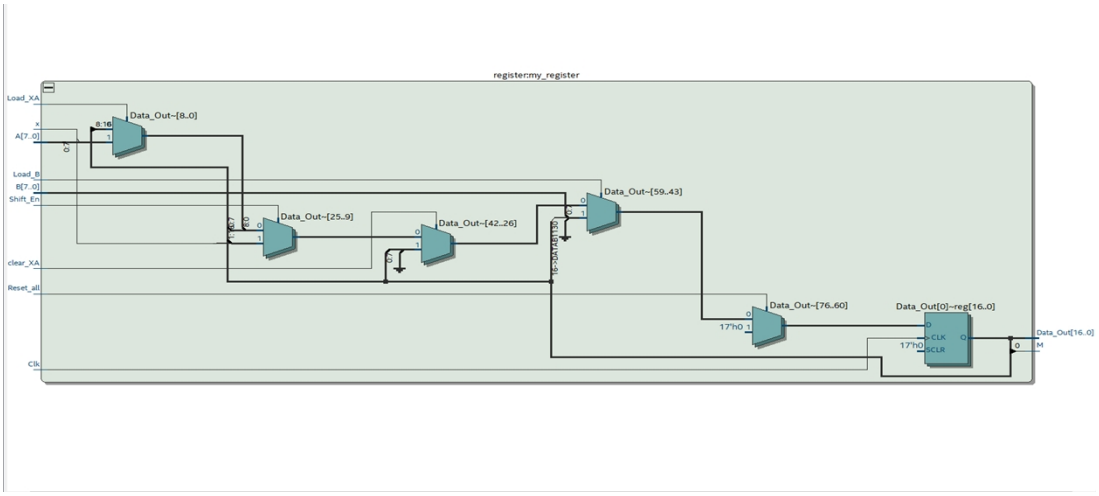
Module: register.sv

Inputs: [7:0]A, [7:0]B, x, Clk, Reset_all, clear_XA, Load_B, Load_XA, Shift_En.

Outputs: [16:0] Data_Out, M.

Description: This module is our register unit contains register A, B and X. The change of register data during operations: Reset, ClearA_LoadB and SHIFT are declared here.

Purpose: This module is the register unit of lab5, storing the value of XAB.



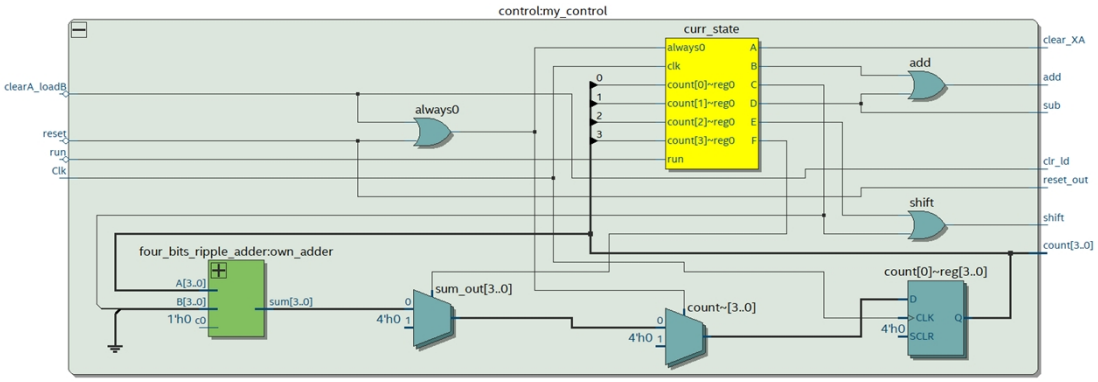
Module: control.sv

Inputs: run, clearA_loadB, Clk, reset.

Outputs: [3:0] cout, add, sub, shift, clear_XA, clr_ld, reset_out.

Description: This module is our control unit achieved as a state machine, 7 states are designed to represent wait, add, shift, 8th bit sub, 8th bit shift, hold1, hold2.

Purpose: control.sv is used to decide whether to add, shift or subtract during a multiplication and send the corresponding add, shift, clear_XA, subtract signal to register unit.



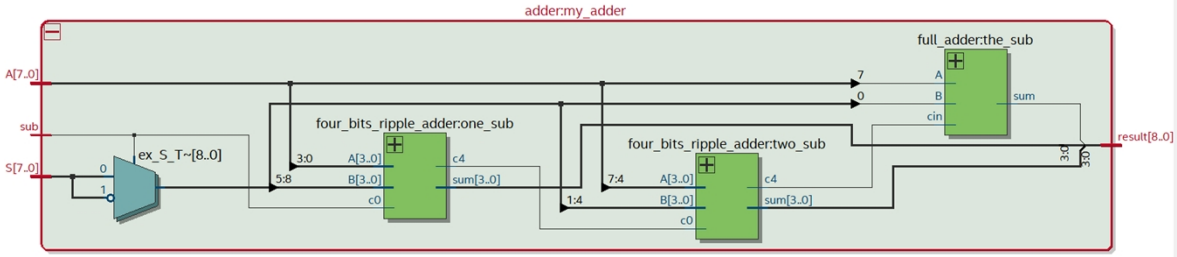
Module: adder.sv

Inputs: [7:0] A, [7:0] S, sub.

Outputs: [8:0] result.

Description: This is 8-bit adder accept two 8-bit 2's complement binary number and give their 9-bit sum or difference based on signal sub. It is generated from single bit full adder, which is very similar to what we do in lab 4.

Purpose: This adder is used in the state machine in our control unit, worked as a counter to count 7 shifts, which help us to control whether we may use sub instead of add.



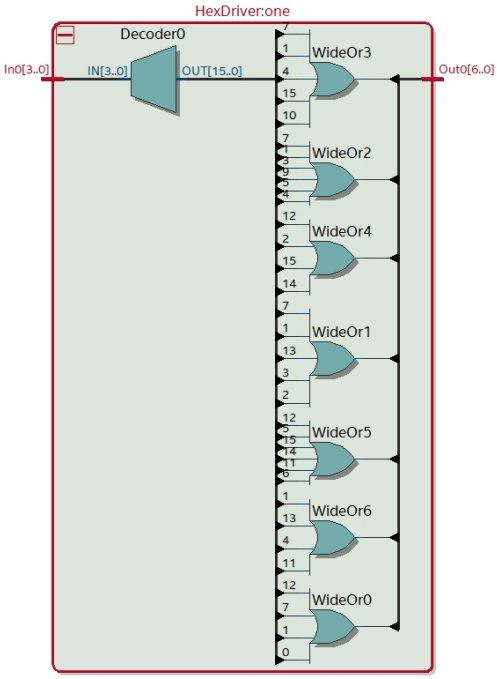
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0.

Description: This module achieves a mux to exchange output of our circuit to 7-bit display signal.

Purpose: This module helps to light the 7 bits segment display on the FPGA board.



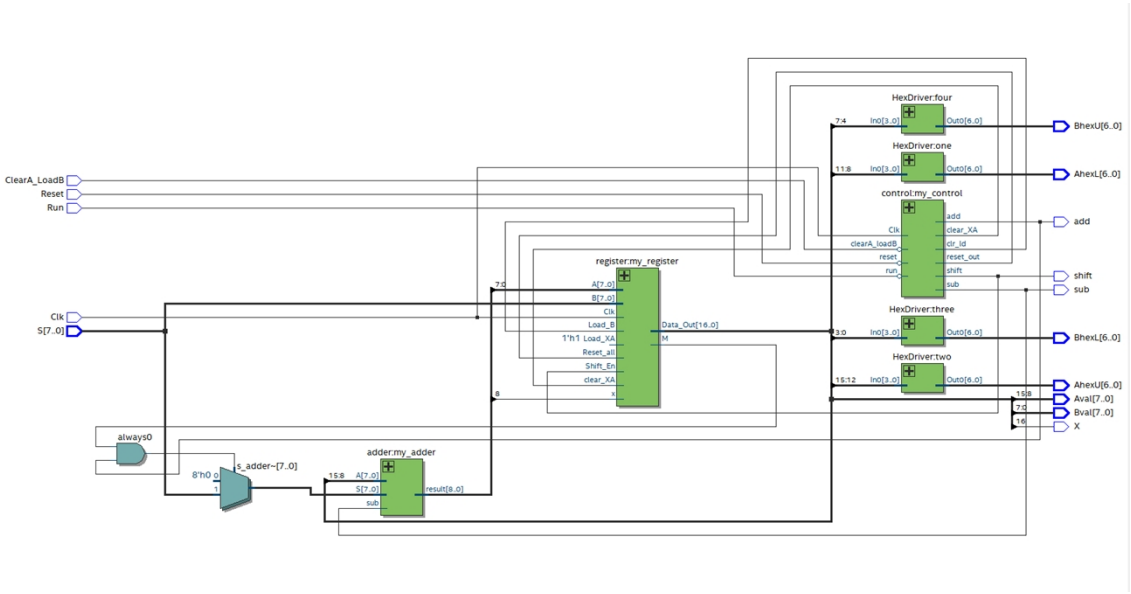
Module: top_level.sv

Inputs: [7:0] S, Clk, Reset, Run, ClearA_LoadB.

Outputs: [7:0] Aval, [7:0] Bval, [6:0] AhexU, [6:0] AhexL, [6:0] BhexU, [6:0] BhexL, x, add, sub, shift.

Description: This is the top level of our lab5, contains register unit part, control unit part.

Purpose: This module connects each part of our lab5.



3.4. State Diagram for Control Unit

7 states are designed to represent wait, add, shift, 8th bit sub, 8th bit shift, hold1, hold2.

wait is the start state, if signal run=0, we just inner loop it. we do clear XA here for both single multiplication and consecutive multiplication.

when run=1, we go to state add, here we do add, after one clock cycle, add is finished and we go to next state shift.

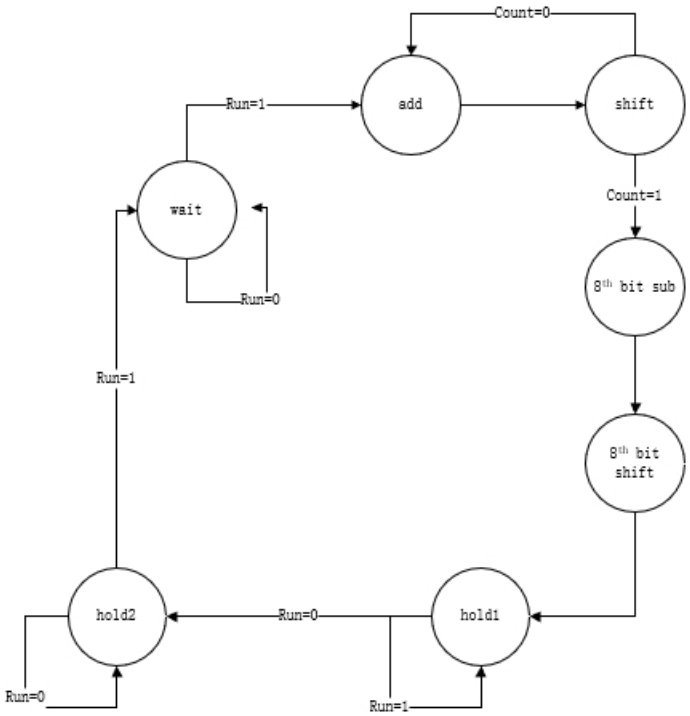
At state shift, we do shift operation. Here we apply an adder as counter to count how many times we have done shift, if less than seven, we go back to state add then shift again and increase the adder by one; if adder is already 7, which means what left is only the 8th bit, so we go to state 8th bit sub.

For state 8th bit sub, we do sub, then after one clock cycle sub is finish, we go to 8th bit shift.

State 8th bit shift actually do the same operation as shift state. Then after one clock cycle and shift is finished, we go to hold 1 state.

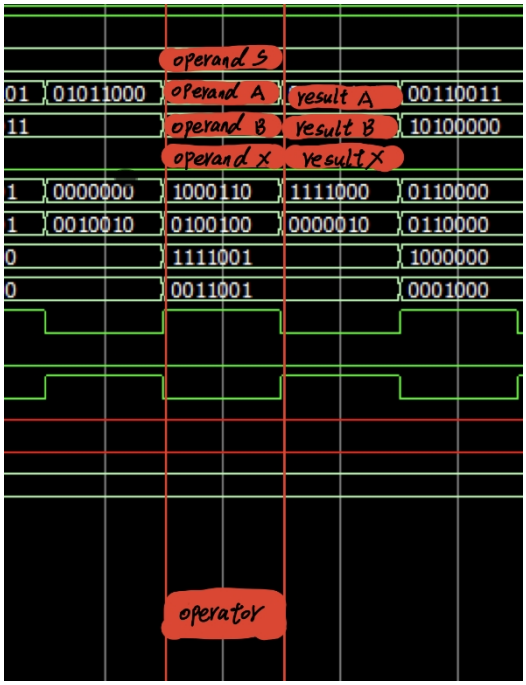
Hold1 state do nothing but just use to keep the result display. And after one clock cycle it goes to state hold 2 which still keep the result display and we clear our counter for consecutive multiplication. If run=0, just inner loop and keep result, otherwise we go to state wait and it will clear_XA for next multiplication, as the time we press run to be 1 must

larger than single clock, so it will stay at wait state one clock cycle and go to state add as we expect.



4. Annotated pre-lab simulation waveform

To see operands, operators and results in the waveform, please see the picture below:



There are four types of operators:

IF operators == add :

Operand S + Operand A = {Result X , Result A}

IF operators == sub :

Operand A - Operand S = {Result X , Result A}

IF operators == shift :

{Operand X, Operand A, Operand B}

ARUTHMETICALLY SHIFT

{Result X, Result A , Result B }

IF operators==holds:

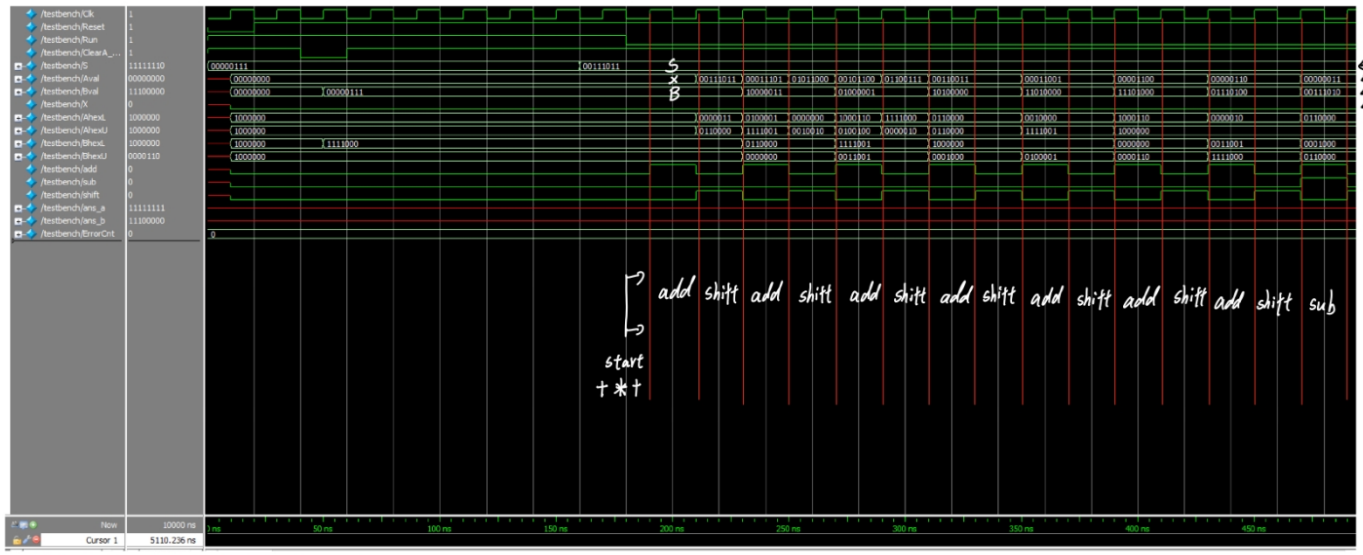
Meaning { Operand A, Operand B} is now become **final result.**

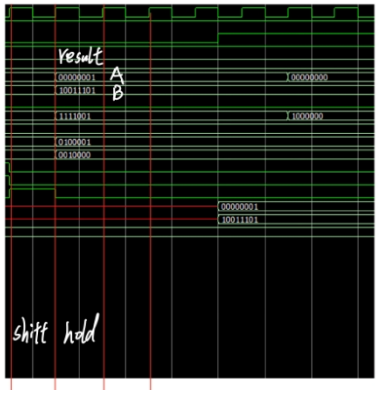
In this way, we can clearly show the operands as well as the result in the waveform.

Note: When add signal==1,sub==0, we do add operation.

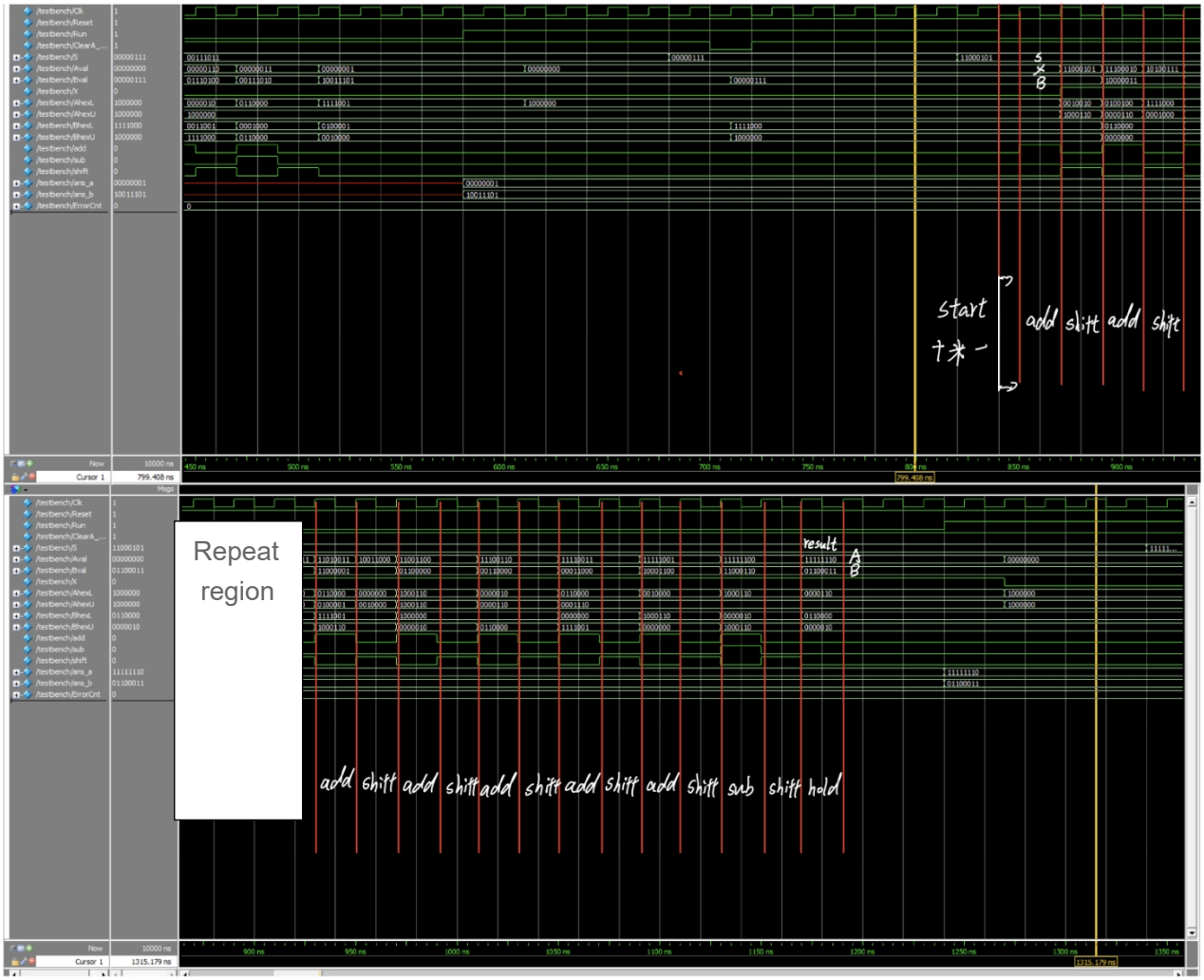
When add signal==1,sub==1, we do sub operation

Example1: 7 * 59 = 413

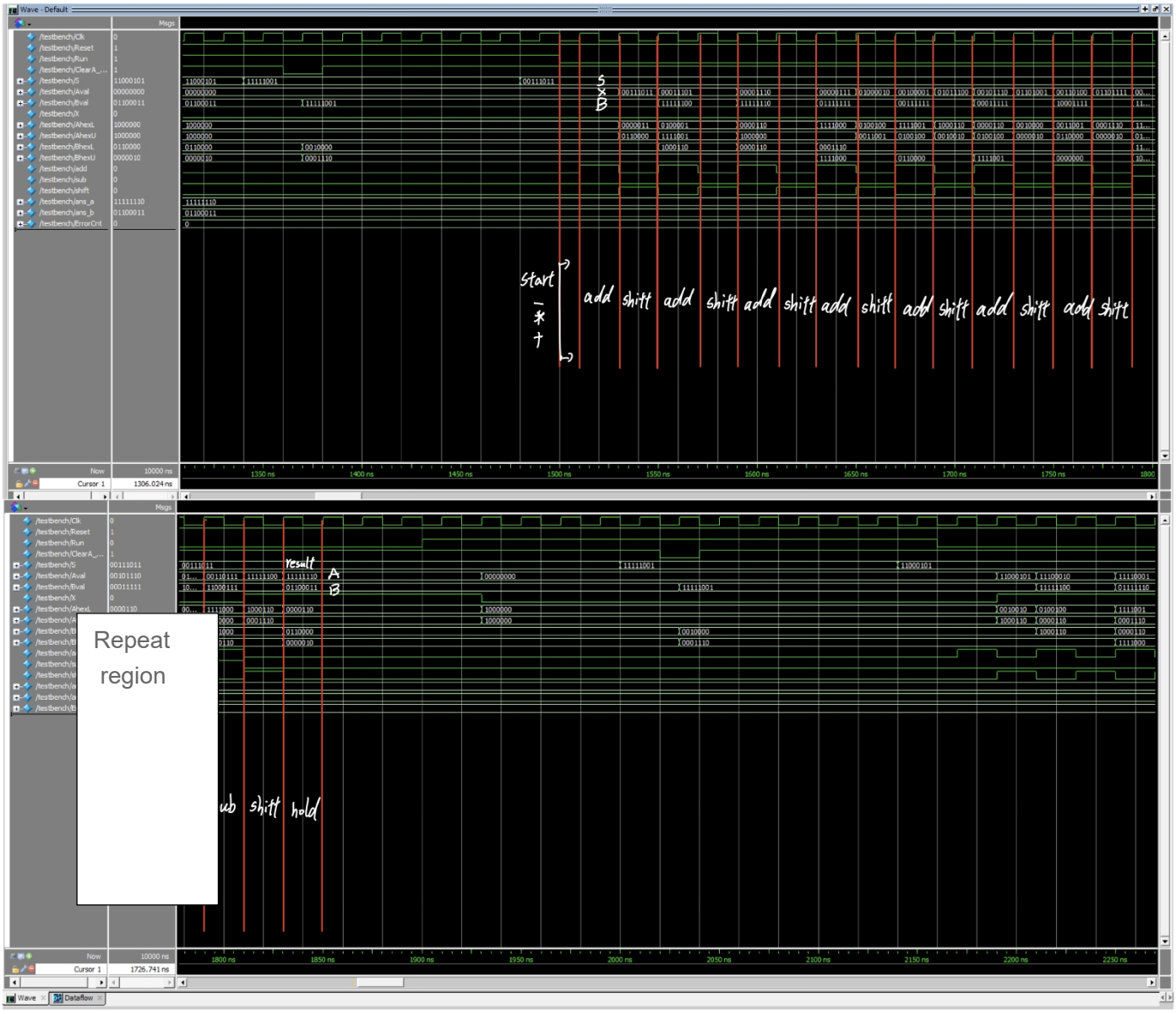




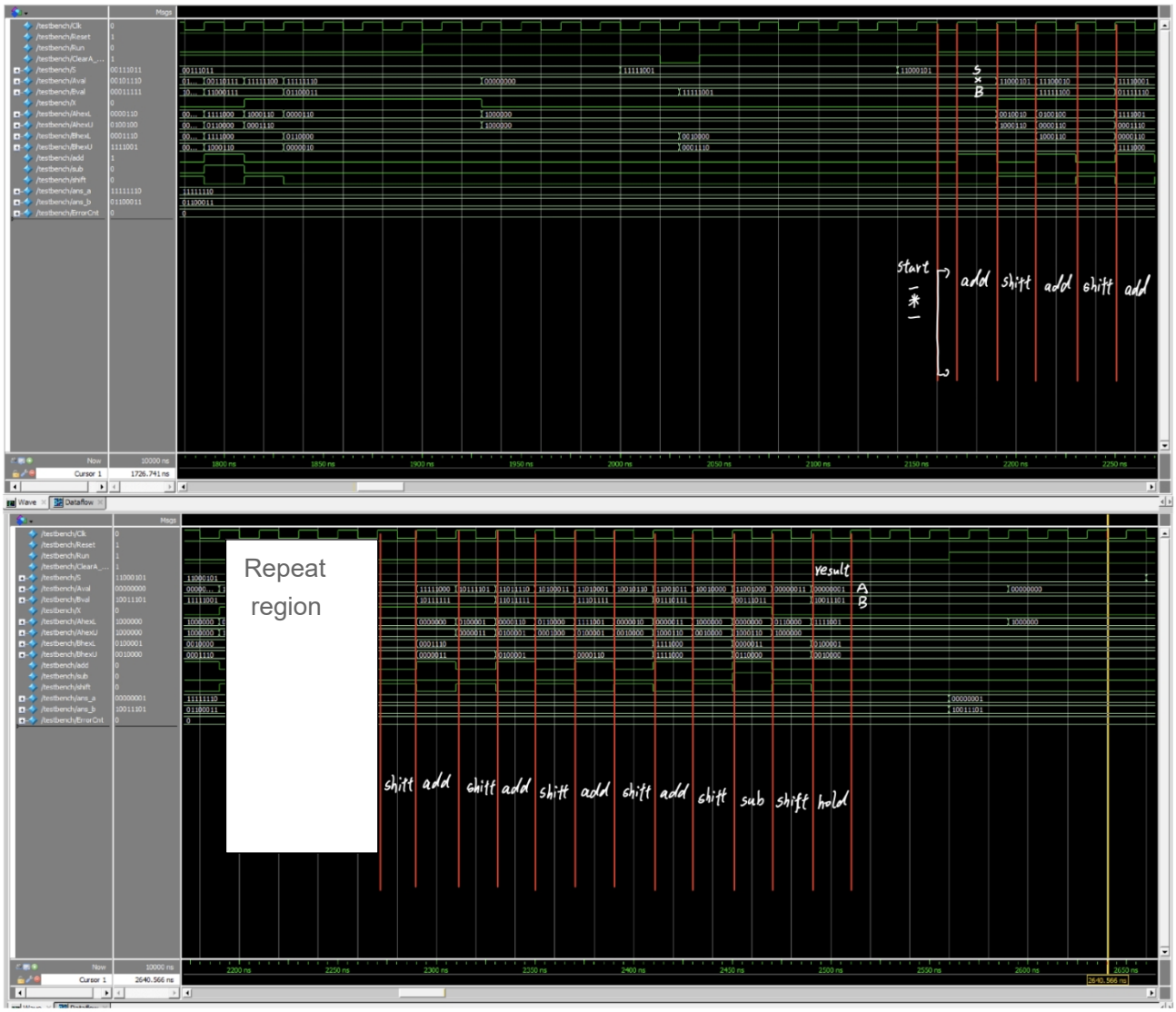
Example2: 7 * -59 = -413



Example3: -7 * 59 = -413



Example4: -7 * -59 = 413



5. Answer to post-lab questions

5.1

LUT	64+28=92
DSP	0
BRAM	0
FLIP-FLOP	28
Frequency	65.74 MHz
Static Power	98.51 mW
Dynamic Power	2.19 mW
Total Power	144.89 mW

How we decrease the Total gate count:

1, When design the state machine, we use a 4 bits counter to count the number of add operation we have taken. In this way, we only need 7 states. If we use lots of states to represent the 7 add operations, 1 sub operation, 8 shift operations, then we need more combinational logic to transfer from one state to another state.

How to increase the maximum frequency:

1, We use two four bits ripple adders and one full adder to construct the 9 bits adder. We can use CSA to replace the ripple adder sine CSA is much faster than the ripple adder.

5.2

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

When we add two 8 bits numbers, we can get at most a 9 bits number.

For example:

0111 1111+ 0111 1111=0 1111 1110(if we do not use X, then, will get a negative number by adding two positive numbers)

1000 0000+1000 0000=1 0000 0000(if we do not use X, we will get 0)

X get set/cleared at:

1, When we have already finished the multiplication and want to continue doing multiplication, we have to set X==0 first.

2, if reset button is pressed, we have to clear X.

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

Since we have to clear XA every time we start a new multiplication operation, we have to pay attention to the correctness of B.

the result should be in range:

$$-2^7 <= \text{result} <= 2^7-1$$

Result out of this range is correct but cannot do any further “continuous multiplications”.

Reason:

If result> 2^7-1, then bits in A will set to zero when doing the next multiplication. Also, if the 8th bits =1 and bits in A=0, then after clear XA, the operand B will be a negative number instead of positive number.

If -2^7> result, the 8th bits will be zero, so after clear XA, the operand B will be a positive number instead of negative number.

What are the advantages and disadvantage of the implemented multiplication algorithm over the pencil-and-paper?

Advantages:

1, save memory space. Our implemented multiplication algorithm only needs to hold XAR registers (ignore flip-flops in CONTROL and ADDER). However, we have to list eight auxiliary number if solved by pencil and paper.

Disadvantage:

1, In our design, we still have to wait for one cycle even there is no need to add. We can just skip this if solved by pencil and paper.

2, In our design, shift operation takes almost half of the executing time. However, if solved by pencil and paper, since we list all eight-auxiliary number, there is no need to shift.

6. Conclusion

6.1. Discuss functionality of design

Just follow the instruction from lab manual, we can successfully finish this lab. In state machine part, we think a repeated add and shift state for seven times is a little bit dummy so we decided to use a counter to help us, as we are asked not to use the “+”, we finally have the adder from last lab to reduce the code length of our state machine, and it works well.

One worth taking bugs happened during our lab is, when we do consecutive multiplication, when the result is large than 8 bit(we means that register XA begin to store some none-zero bit), if I just press run button, the result show wrong, but if I press and not loose the key, we get the right answer. The reason is we have to do clear XA before the next multiplication and we just wrongly designed our state machine so it clear_XA immediately after one clock cycle which is too short for us to see and go to the wait state. The solution is adding one more state between the two states to do nothing but just hold our result.

6.2. Comments for lab manual and other materials

The lab manual is quite clear. However, we have difficult in understanding the multiplication algorithm when we read the manual. It seems better if the manual can take some paragraph to explain why the multiplication algorithm is correct.