1 Implement Key Board

1.1 Press 0/1/2/3/4/5/6/7/8/9 and show them in the seven-segment display. When a new number is pressed, the previous number is refreshed and over written.

1.2 Press a/s/m (addition/subtraction/multiplication) and show them in the seven-segment display as your own defined A/S/M pattern. When you press "Enter", refresh (turn off) the seven-segment display.

Design Specification:

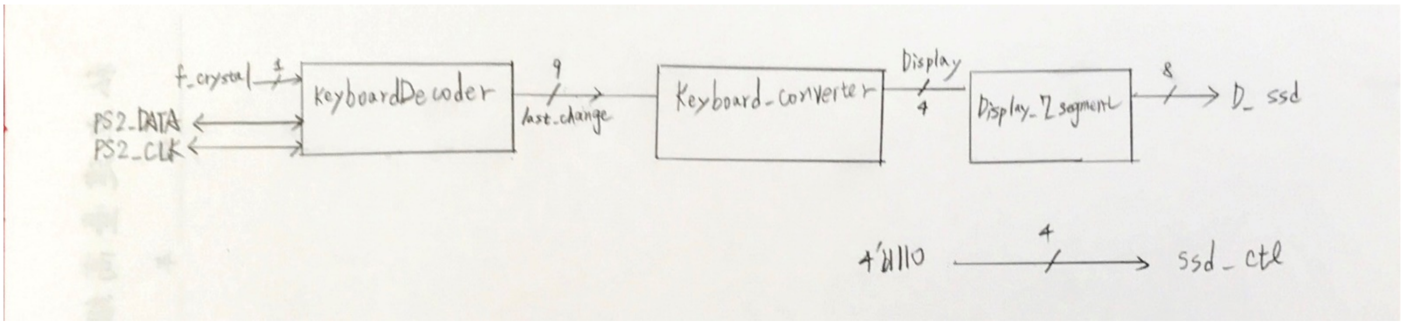
Input: f_crystal, rst

Output: [7:0]D_ssd

Output: [3:0]ssd_ctl

Inout: PS2_CLK, PS2_DATA

Block diagram:



Design Implementation:

I/O pin assignment:

f_crystal—W5

rst—V17

PS2_CLK—C17

PS2_DATA—B17

ssd_ctl[3]—W4

ssd_ctl[2]—V4

ssd_ctl[1]—U4

ssd_ctl[0]—U2

D_ssd[7]—W7

D_ssd[6] — W6

D_ssd[5] — U8

D_ssd[4] — V8

D_ssd[3] — U5

D_ssd[2] — V5

D_ssd[1] — U7

D_ssd[0] — V7

Using the signal of last_change from KeyboardDecoder, I am able to know the which number key is pressed by the user. And because it's 9-bit hexadecimal number, I convert it 4-bit decimal number in Keyboard_converter module, where it's basically a table showing the relation of input and output. For example, when input of Keyboard_converter is 9'h3D, output would be 4'd7.

As for the a/s/m, I still use Keyboard_converter module to make them 4-bit decimal number. Hence, I also need to define how to show them on 7-segment display inside the Display_7_segment module. For example, when Display (in the block diagram above) is 4’d10, 7-segment display would show a.

Discussion:

In this experiment, its requirement is relatively easy compared to other experiments in this lab. I only take the signal of last_change and convert it to a 4-bit decimal number so that I can use the Display_7_segment module I already completed in previous lab to show the result. Also, when I was doing this experiment, I wanted to try to use the extended keys on the right-hand side of the keyboard. I thought the MSB of last_change would be 1 when using the extend keys. However, it’s not. Still, MSB of last_change is still 0 and only the make code is different. It helped me understand the use of extended key in the following experiments.

2 Implement a single digit decimal adder using the keyboard as the input and display the results on the 14-segment display (The first two digit are the addend/augend, and the last two digits are the sum).

Design Specification:

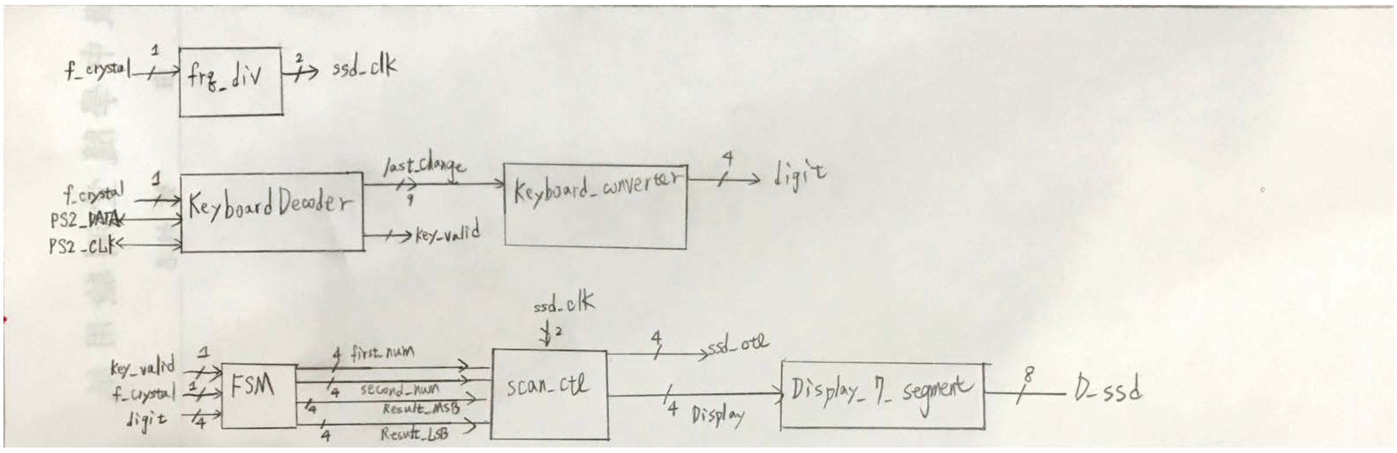
Input: f_crystal, rst

Output: [7:0]D_ssd

Output: [3:0]ssd_ctl

Inout: PS2_CLK, PS2_DATA

Block Diagram:



Design Implementation:

I/O pin assignment:

f_crystal—W5

rst—V17

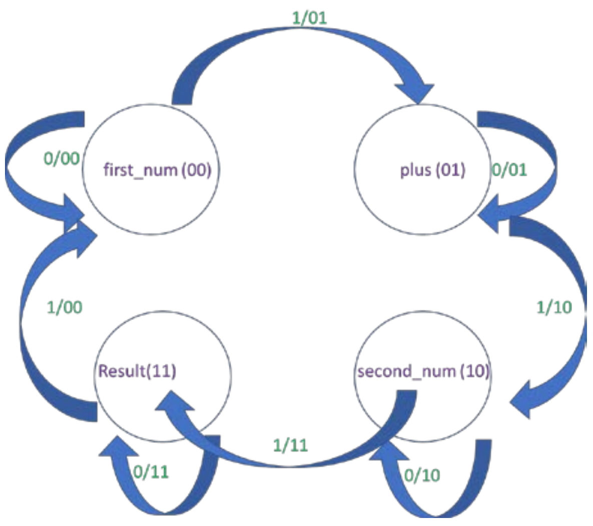
PS2_CLK—C17

PS2_DATA—B17

ssd_ctl[3]—W4

ssd_ctl[2]—V4
ssd_ctl[1]—U4
ssd_ctl[0]—U2
D_ssd[7]—W7
D_ssd[6] — W6
D_ssd[5] — U8
D_ssd[4] — V8
D_ssd[3] — U5
D_ssd[2] — V5
D_ssd[1] — U7
D_ssd[0] — V7

State diagram of FSM



In this experiment, as we need to show different numbers on 7-segment display, I added the frequency divider (frq_div module in block diagram) back. As same as experiment #1, I designed a Keyboard_converter to convert input from keyboard into a 4-bit number.

The most essential function in this experiment is to key in two single digit number and show their added result. Therefore, I designed a FSM with four states (2 bits to represent state). When signal of key_valid from KeyboardDecoder is 1'b1, the state changes as drawn in the state diagram above. It will change from input of first number, plus, input of second number to Result sequentially.

Eventually, I showed the number on the 7-segment display. When the input is not given by the user, it shows zero on 7-segment display.

Discussion:

Because I only used signal of key_valid to change the state of FSM, FSM will change its state twice as key_valid is 1'b1 when keyboard is pressed or released. I didn't notice this at first, so the result was weird. Therefore, I will show the calculated result at the first_num state in the state diagram when the user doesn't type anything as when input the second number, FSM change its state twice from second_num to first_num in the state diagram directly. I knew I am supposed to use key_valid and key_down together to reduce the use of state. Not until I completed the experiment #2, #3, I knew how to use key_down and key_valid correctly.

3 Implement a two-digit decimal adder/subtractor/multiplier using the right-hand-side keyboard (inside the red block). You don't need to show all inputs and outputs at the same time in the 7-segment display. You just need to show inputs when they are pressed and show the results after "Enter" is pressed.

Design Specification:

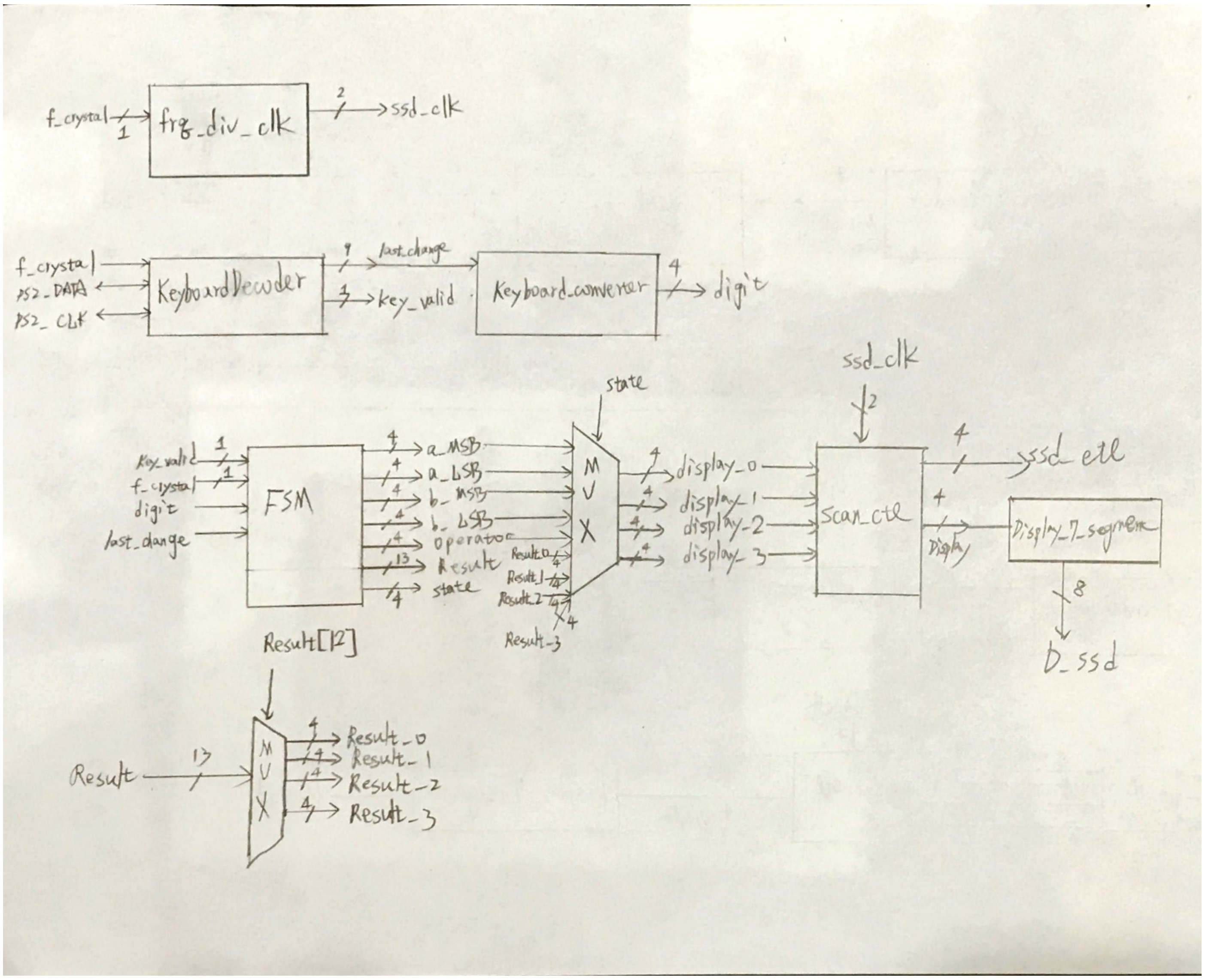
Input: f_crystal, rst

Output: [7:0]D_ssd

Output: [3:0]ssd_ctl

Inout: PS2_CLK, PS2_DATA

Block Diagram:



Design Implementation:

I/O pin assignment:

f_crystal—W5

rst—V17

PS2_CLK—C17

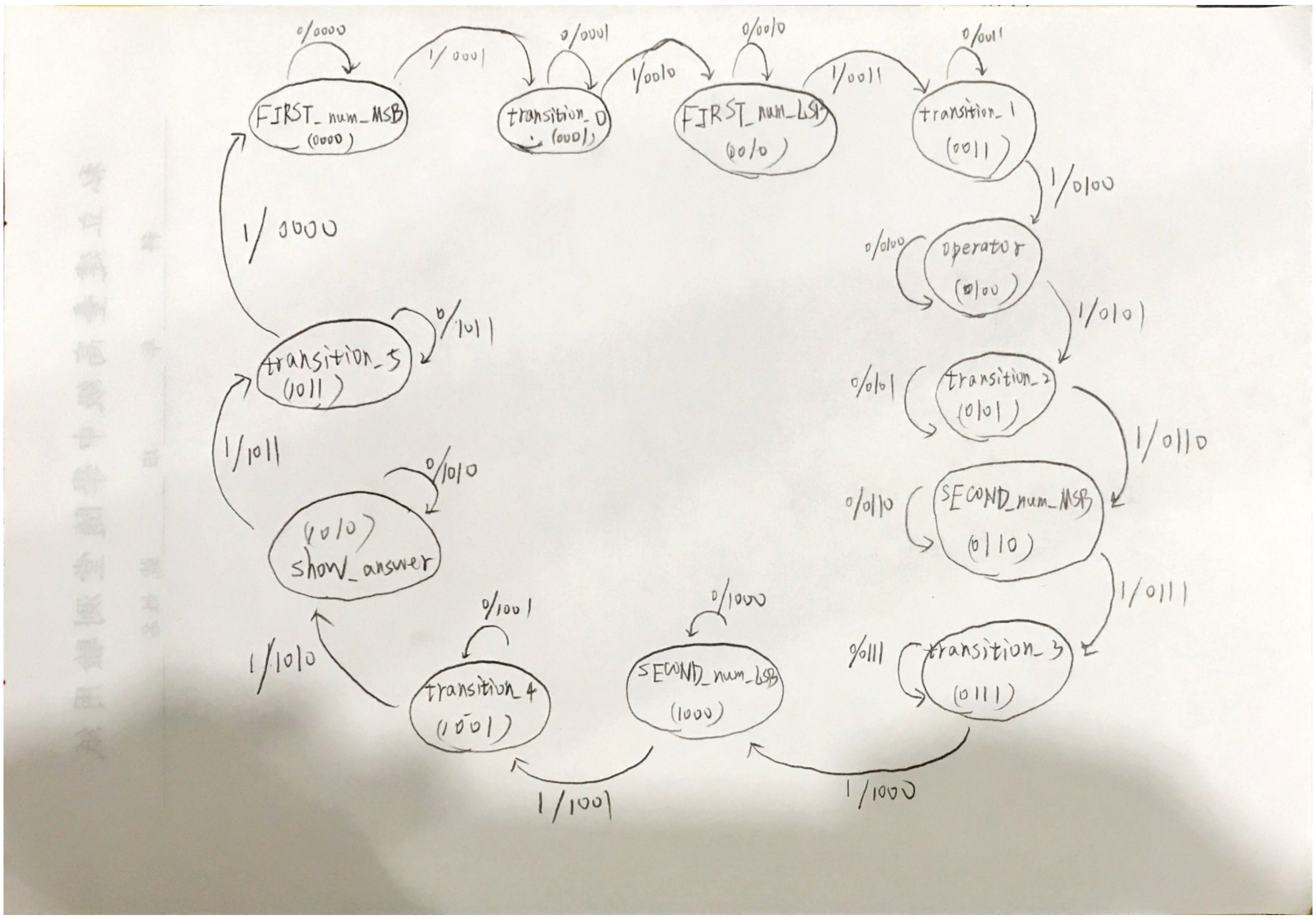
PS2_DATA—B17

ssd_ctl[3]—W4

ssd_ctl[2]—V4

ssd_ctl[1]—U4
ssd_ctl[0]—U2
D_ssd[7]—W7
D_ssd[6] — W6
D_ssd[5] — U8
D_ssd[4] — V8
D_ssd[3] — U5
D_ssd[2] — V5
D_ssd[1] — U7
D_ssd[0] — V7

State diagram of FSM:



This experiment is more complicated than the experiment #2 since it's double digits and three operation to implement.

Frequency divider, KeyboardDecoder and Keyboard_converter work the same role as the experiment #2. So, I start to explain my design with FSM.

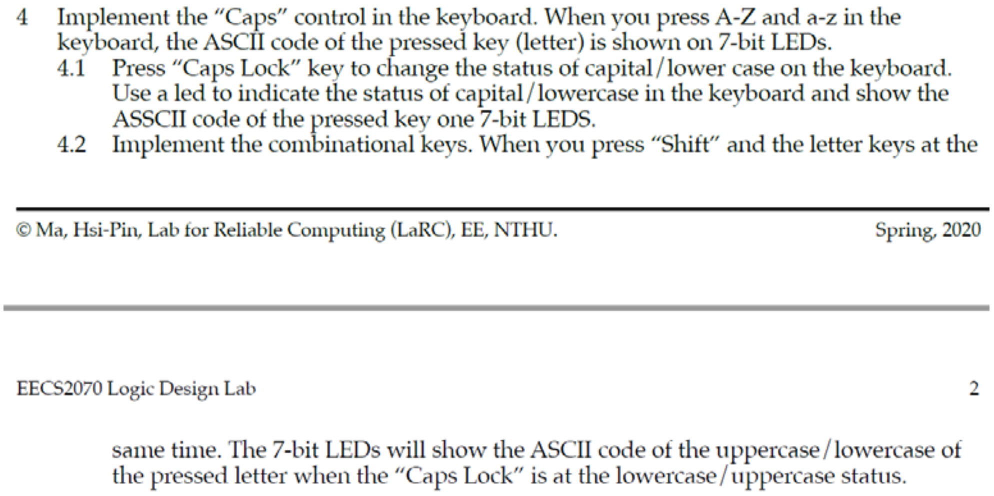
I need 12 states in my FSM as I only use signal of key_valid to decider if the FSM would change its state or not. As shown in the state diagram, there are many transition states, which is basically useless. For other states, FSM would take values from keyboard and save them to corresponded registers. For example, in the state of FIRST_num_LSB, FSM would save the LSB of first number to its corresponded registers. Overall, FSM will take the value of first number, operator, second number sequentially. And finally, it calculates the result according to operator.

For calculating the result, the maximum would be 99*99 = 9801 so that I used a 13-bit register for result. Also, the result could be negative as there is 'subtraction' operation to choose. Therefore, I used the MSB of result to decide the result is positive of negative. If it's negative, I used a temporary register to store 0 – result (becomes positive) and calculated each digit of (0 – result). After calculating, I will show '-' on 7-segment display to indicate it's negative. If it's positive, I won't change anything. I only calculate each digit.

Eventually, according to the different state of FSM, I will show its corresponded number on the 7-segment display.

Discussion:

Since I didn't know how to use key_down to change the state of FSM when I was doing this experiment, I spent a lot of time writing FSM. Also, it's way more complicated than it should be.
But luckily, the architecture of FSM is not too hard to design, I still completed it correctly.


4 Implement the "Caps" control in the keyboard. When you press A-Z and a-z in the keyboard, the ASCII code of the pressed key (letter) is shown on 7-bit LEDs.
4.1 Press "Caps Lock" key to change the status of capital/lower case on the keyboard. Use a led to indicate the status of capital/lowercase in the keyboard and show the ASSCII code of the pressed key one 7-bit LEDS.
4.2 Implement the combinational keys. When you press "Shift" and the letter keys at the

© Ma, Hsi-Pin, Lab for Reliable Computing (LaRC), EE, NTHU. Spring, 2020

EECS2070 Logic Design Lab 2

same time. The 7-bit LEDs will show the ASCII code of the uppercase/lowercase of the pressed letter when the "Caps Lock" is at the lowercase/uppercase status.

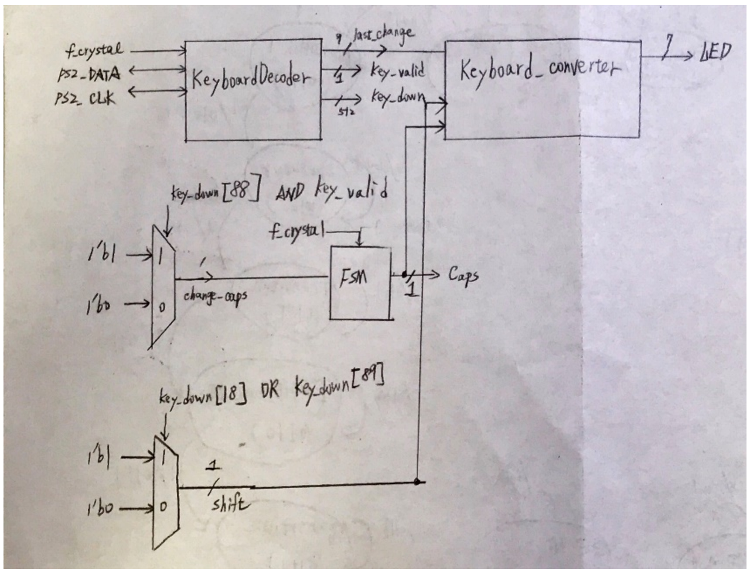
Design Specification:

Input: f_cyrstal, rst

Output: Caps, [6:0]LED

inout: PS2_CLK, PS2_DATA

Block Diagram:



Design Implementation:

I/O pin assignment:

f_crystal—W5
rst—V17
PS2_CLK—C17
PS2_DATA—B17
Caps—L1
LED[6]—U14
LED[5]—U15
LED[4]—W18
LED[3]—V19
LED[2]—U19
LED[1]—E19
LED[0]—U16

In the experiment, we will implement Caps control in the keyboard. So, I used a FSM to control the "Caps Lock" signal. When the "caps lock" is pressed (key_down [88] = 1'b1) and key_valid is 1'b1 at the same time, FSM would change its change to the ~Caps (state of FSM, which only has 2 possible values).

As for "shift" control, when key_down[18] or key_down[89] is 1'b1 (shift is pressed), signal of shift becomes 1'b1.

Then, inside Keyboard_Decoder module, when (Caps == 1'b1 && shift == 1'b0) OR (Caps == 1'b0 && shift == 1'b1), it represents uppercase of the character. Otherwise, it's will show lowercase when pressed.

Discussion:

When I was implementing FSM, I found "caps lock" is not very sensitive when it's pressed. (i.e. it sometimes couldn't change its state immediately) I think it's because I only used key_down to control it. When I used key_down && key_valid to decide if the Caps (state of FSM) needs to be changed, "caps lock" becomes a lot more sensitive. I haven't figured out the reason as the Verilog code of KeyboardDecoder is too complex.

Conclusion for Lab9:

In this Lab, I have learned how to use keyboard to interact with FPGA board. Using the sample code given by professor, I am able to know which keys are pressed currently (key_down) and which key is just pressed or released (last_change). Plus, a FSM can help us know which state we are so that we are able to preform addition, subtraction, multiplication through the input from keyboard.

Reference:

Sample code given by professor.