1. Implement key board using the left-hand-side keyboard (inside the black blocks).

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 overwritten.

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.

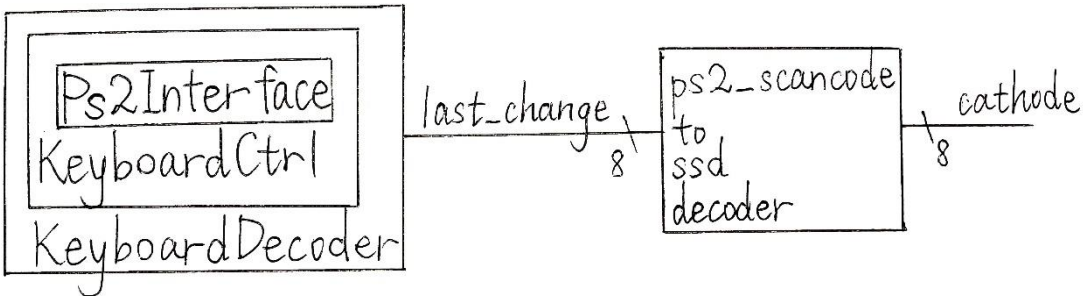
1. Specification

```
module LAB9_1_top(
  output [7:0] cathode,         // seven segment display

  inout PS2_DATA,               // (PIC24) USB HID Host
  inout PS2_CLK,                // (PIC24) USB HID Host

  input rst_n,                  // active low reset
  input clk                     // 100MHz global clock
);
```

2. Block Diagram



根據老師的課堂講解，KeyboardDecoder.v 的其中一個 output last_change 的訊號包含了被按下的按鍵的 scan-code。因此，只要將 last_change 的訊號連接到一個解碼器，將 scan-code 轉換成 SSD 即可。

3. Finite state machine

無

4. Implement

本題要讓鍵盤上的數字鍵被按下後，顯示對應的數字在七段顯示器上。檢查老師提供的 KeyboardDecoder.v，發現可以使用其 output last_change 來判斷是哪一個按鍵被按下。因為當一個按鍵被按下之後，last_change 的值就會更新成那個按鍵的 scan-code，所以只要將 last_change 連接到一個 scan-code 轉 SSD 的解碼

器即可。下圖是 LAB9_1_TOP.v 的內容。

```
15  KeyboardDecoder KeyboardDecoder(
16    .key_down,
17    .last_change,
18    .key_valid,
19    .PS2_DATA,
20    .PS2_CLK,
21    .rst(~rst_n),
22    .clk
23  );
24
25  ps2_ssd_decode ps2_ssd_decode(
26    .ssd(cathode),
27    .ps2(last_change[7:0])
28  );
```

其中，解碼器 ps2_ssd_decode.v 的內容是 case statement。將數字 0 到 9 與 enter 等可能的情況一一列出，構成 look-up table。其 code 就只是把所有可能的情況列出來而已，詳參.v 檔。

5. Conclusion

第一題很簡單，沒有遇到困難。我猜想老師會這麼安排第一題，旨是為了要確認每一位同學都能順利的引入老師提供的 IP，並確保正常運作。

在操作的過程中，可以學習如何使用別人寫好的 IP。透過本題，可以了解 KeyboardDecoder.v 的 last_change 訊號代表的意義，並練習使用該訊號來取得被按下的按鍵等資訊。

2. Implement a single digit decimal adder using the left-hand-side keyboard (inside the black blocks). Use the key board as the input and display the results on the 7-segment display (The first two digits are the addend/augend, and the last two digits are the sum).

1. Specification

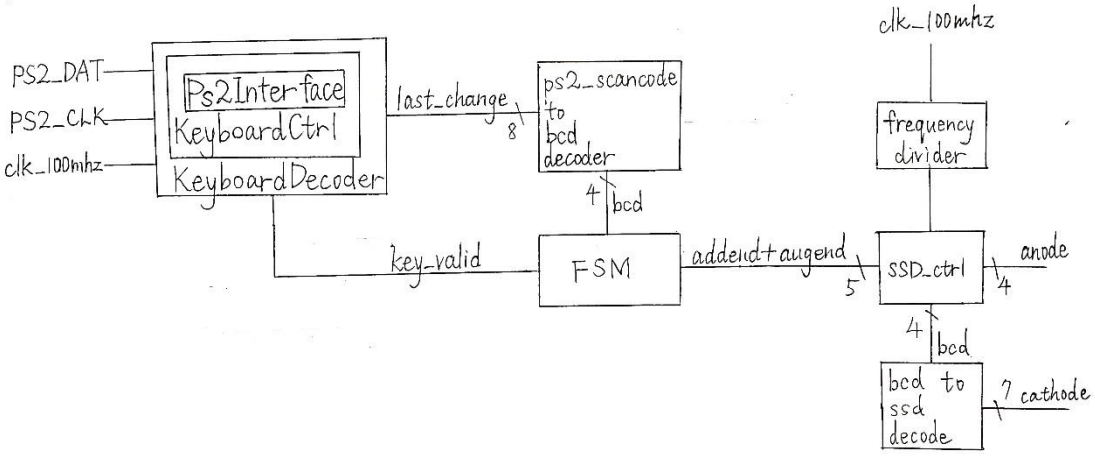
```
module LAB9_2_top(
  output [7:0] cathode,          // seven segment display
  output [3:0] anode,            // seven segment display

  inout PS2_DATA,                // (PIC24) USB HID Host
  inout PS2_CLK,                 // (PIC24) USB HID Host

  input rst_n,                   // active low reset
  input clk                      // 100MHz global clock
);
```

較第一題只多了 anode，讓七段顯示器看起來能同時顯示四個不同的數字。

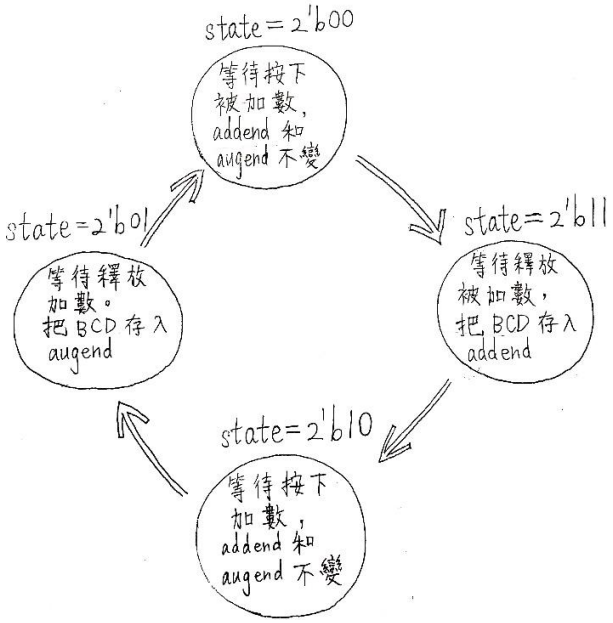
2. Block Diagram



從 block diabram 可以看出，本題建立在上一題的基礎之上。上一題，last_change 直接使用 look-up table 轉換成七段顯示器 cathode 的樣式。但在本題，由於要將輸入的數字相加，因此需要先轉換成可以被加法器理解的 BCD 形式。addend 與 augend 透過加法器相加後，再轉換成七段顯示器 cathode 的樣式。FSM 的 CLK 連接 key_valid 作為時脈。其功能主要為判斷當下的 last_change 在轉換成 BCD 之後，應當存入 addend、還是存入 augend。

3. Finite state machine

本題的 FSM 有四個 state，可以表示成{被加數|加數, 按下|釋放}。如下圖。



此 FSM 的 state transition 為：按下被加數→釋放被加數→按下加數→釋放加數。可以使用一個二位元下數計數器來描述。

```
56  reg [1:0] state; // {addend_or_augeng, key_pressed}
57
58  always @(posedge key_valid or negedge rst_n)
59    if (~rst_n) state <= 'b0;
60    else        state <= state - 'b1;
```

每當按下或釋放一個按鍵，key_valid 都會維持一個 clk 週期的 1'b1。因此，使用 key_valid 的 falling edge 作為 Flip-flop 的 CLK，可以確保所有 D 都是已經更新後且穩定的資料。由於按下或釋放按鍵過了 1/100M 秒之後都會收到一次 key_valid 的 falling edge，所以只有在 state 從奇數變為偶數時才需要改變 augend 或 addend 的值。每按一次鍵盤，就依照順序將按下的數字（scan-code 已經過 look-up table 轉換成 BCD）存入加數 addend、被加數 augend。

```
65  always @(negedge key_valid or negedge rst_n) begin
66    if (~rst_n) begin
67      addend <= 'b0;
68      augend <= 'b0;
69    end else begin
70      case (state[1])
71        1'b1: begin
72          addend <= last_change_bcd;
73          augend <= augend;
74        end
75        1'b0: begin
76          addend <= addend;
77          augend <= last_change_bcd;
78        end
79        default: begin
80          addend <= 'b0;
81          augend <= 'b0;
82        end
83      endcase
84    end
85  end
```

4. Implement

接續上文的 FSM，緊接著的是用來控制顯示器的模組。此段描述的是：將存放在 FSM 裡的 addend 與 augend 相加，再傳送到 ssd_ctrl.v，以在七段顯示器上顯示和。

```
87  ssd_ctrl ssd_ctrl(
88    .cathode,
89    .anode,
90    .bcd0((addend + augend) % 5'd10),
91    .bcd1((addend + augend) / 5'd10),
92    .bcd2(augend),
93    .bcd3(addend),
94    .clk,
95    .rst_n
96  );
```

ssd_ctrl.v 的架構跟前幾次 LAB 中所使用的一樣，由除頻器、MUX、解碼器等組成，功能是將四個 BCD 轉換成 SSD 的樣式之後，以適當的頻率切換四個 anode 的 enable，在四個七段顯示器上顯示出對應的 cathode 樣式。ssd_ctrl.v 的 code 跟上一次 LAB 的幾乎一模一樣，詳參.v 檔。

5. Conclusion

由於在本 LAB 中所使用的 KeyboardDecoder.v 等模組由老師提供，並非我自己親手架構，所以如果只靠老師在課堂上的講解內容，來理解各個 output 的含意與用法，是困難而吃力的。本題遭遇的最大困難在於，每一次按下或釋放鍵盤按鍵，key_valid 都會形成一個時長為 1/100M 秒鐘的脈衝訊號。因此，使用者按下並釋放一個按鍵的過程中，總共會產生兩個脈衝。在本題，只有第一個（因為按下按鍵而產生的）脈衝是有用的，第二個脈衝毫無用處，可以直接忽略。因此，本題如果要在建構 FSM 時，使用 key_valid 來做為 Flip-flop 的時脈 CLK 的話，就必須要在「儲存 addend」與「儲存 augend」這兩個動作之間，多插入一個狀態，用來等待（跳過而不做任何事情）該按鍵被釋放。

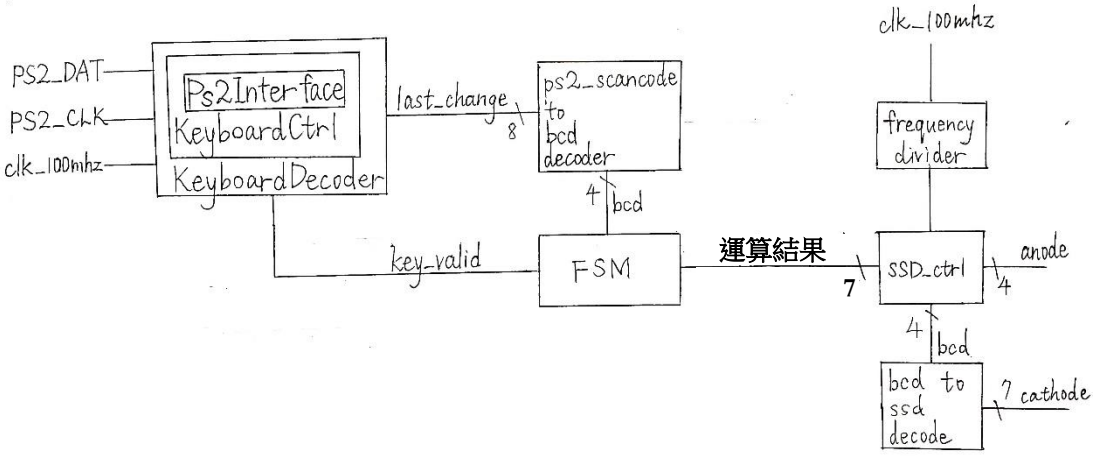
透過本題，可以熟悉 KeyboardDecoder.v 的 key_valid 訊號所代表的意涵，並練習使用 key_valid 來做為 FSM 中 Flip-flop 的 CLK。練習利用 FSM 的當下狀態來解讀數字的意涵，用 state transisiton 來將輸入一一存入加數與被加數中。結合前幾次 LAB 所建構出來的七段顯示器控制模組，反映使用者的輸入，並即時顯示兩個數字的和。

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.

1. Specification

同第二題。

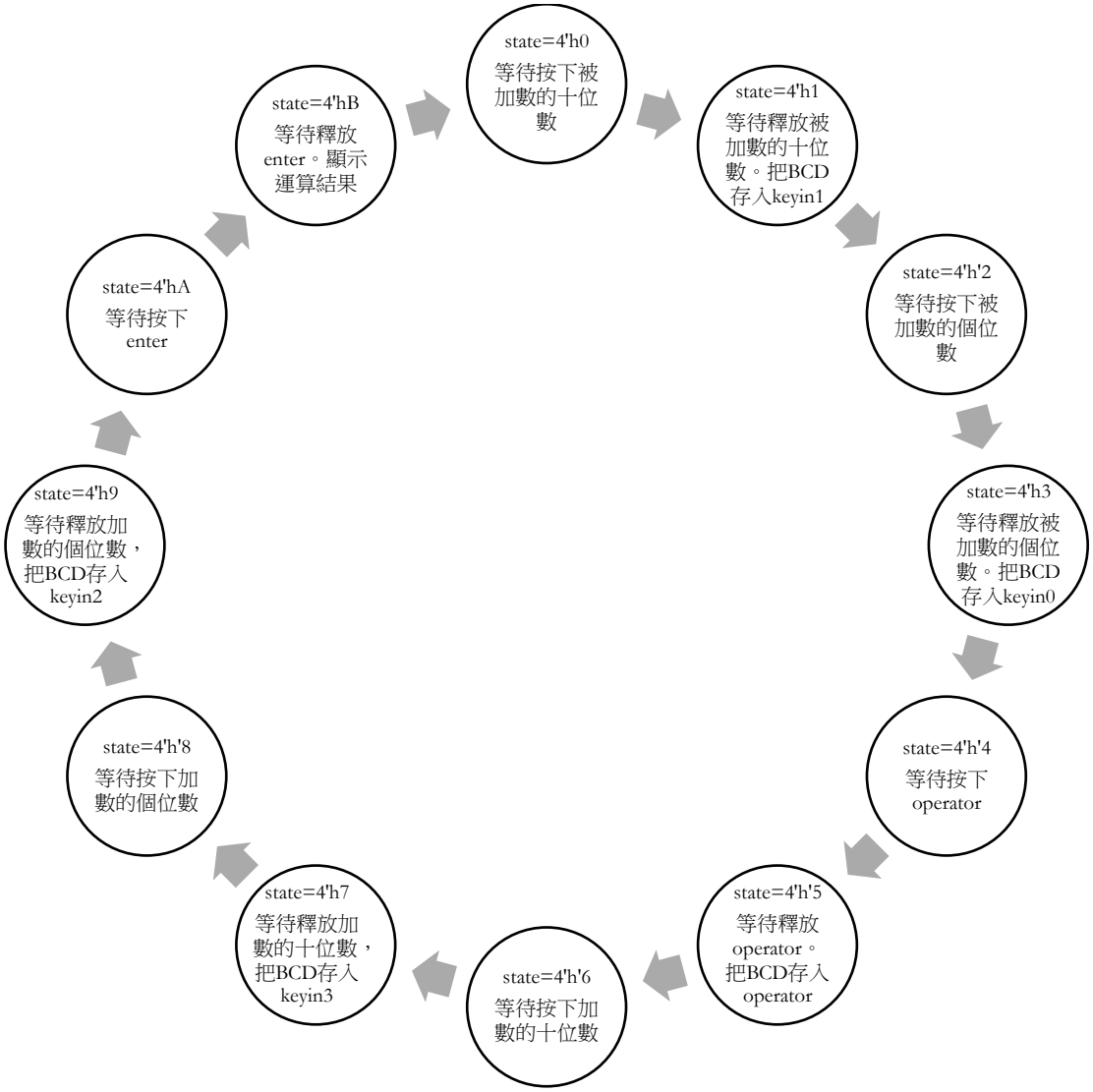
2. Block Diagram



運算結果的 bit width 從一位數（4 bits，0~9）改成兩位數（7 bits，00~99），其餘同第二題。由於要輸入 operator 與 enter，所以 ps2-scancode-to-bcd decoder 除了要把數字 0~9 的 scan-code 轉成 bcd 之外，我把 a（加法）、s（減法）、m（乘法）、enter，這四個按鍵的 scancode 轉換成 4'hA、4'hB、4'hC、4'hD。因為在原本的 BCD 中，4'hA、4'hB、4'hC、4'hD 是 unused，所以這麼做並不會影響到數字 0~9 的運作。本題的 FSM 有較大更動，上一題只需要四個 state，依序存入 addend、augend；本題則需要 12 個 state，依序存入 addend、augend 的十位數、個位數，以及 operator。

3. Finite state machine

本題的 FSM 有 12 個狀態（state = 4'h0 ~ 4'hB）。其中，偶數的 state 都是等待按下按鍵，奇數的 state 都是等待釋放按鍵。因此，從偶數的 state 變成奇數的 state 的時候，需要將新的 BCD 存入對應的 addend、augend 的十位數、個位數（reg 的名稱是 keyin0 ~ keyin3）、operator。此外，雖然沒有在下圖中列出，但是為了要即顯示使用者輸入的數字，所以從偶數的 state 變成奇數的 state 的時候，也要將新的 BCD 顯示在七段顯示器上（存入 display0 ~ display3），按下 enter 時則顯示運算結果。



keyin0 ~ keyin3 儲存被加數和加數的十位數和個位數，operator 則儲存要執行的運算。

```
88  // store each keystroke in order
89  reg [3:0] keyin0;
90  reg [3:0] keyin1;
91  reg [3:0] keyin2;
92  reg [3:0] keyin3;
93  reg [3:0] operator;
```

從偶數的 state 變成奇數的 state 時，會將 BCD 存入 display0 ~ display3，以即時反應使用者的輸入。在 state 從 4'hA 變成 4'hB 時，則將運算結果 answer 存入 display0 ~ display3，使七段顯示器顯示以這些資訊所計算出來的結果，如下。

```
122 reg [3:0] display0;
123 reg [3:0] display1;
124 reg [3:0] display2;
125 reg [3:0] display3;
126 wire [6:0] numberA;
127 wire [6:0] numberB;
128 wire [6:0] answer;
129
130 assign numberA = keyin0 * 10 + keyin1;
131 assign numberB = keyin2 * 10 + keyin3;
132 assign answer = (operator == ADDITION)       ? (numberA + numberB) :
133                 (operator == SUBTRACTION)    ? (numberA - numberB) :
134                 (operator == MULTIPLICATION) ? (numberA * numberB) : ('b0);
```

FSM 中，Flip-flop 以 key_valid 當作 CLK，state transition 可以用一個四位元上數計數器來描述，如下。

```
78  // update flow (state 0 ~ A)
79  reg [3:0] state;
80
81  always @(posedge key_valid or negedge rst_n) begin
82    if (~rst_n) state <= 'b0;
83    else        state <= (state < 4'hB)
84                       ? (state + 1'b1)
85                       : ('b0);
86  end
```

至於每一個 state 要做麼事情，另外用一個 always block 來描述。先描述重設時該做的事情，如下。

```
136 always @(negedge key_valid or negedge rst_n) begin
137   if (~rst_n) begin
138     keyin0   <=  'b0;
139     keyin1   <=  'b0;
140     keyin2   <=  'b0;
141     keyin3   <=  'b0;
142     operator <=  'b0;
143     display0 <= 4'hF;
144     display1 <= 4'hF;
145     display2 <= 4'hF;
146     display3 <= 4'hF;
147   end else begin
```

接下來一一列出所有 state 要做的事情。先依序將使用者的 input 存入被加數的十位數與個位數，如下。

```
147   end else begin
148     case (state)
149       4'h1: begin
150         keyin0   <= last_change_bcd;
151         display0 <= last_change_bcd;
152         display1 <= 4'hF;
153       end
154       4'h3: begin
155         keyin1   <= last_change_bcd;
156         display0 <= last_change_bcd;
157         display1 <= display0;
158       end
```

然後將使用者的 input 存入 operator，如下。

```
159       4'h5: begin
160         operator <= last_change_bcd;
161         display0 <= last_change_bcd;
162         display1 <= 4'hF;
163       end
```

隨後再存入加數的十位數與個位數，如下。

```
164       4'h7: begin
165         keyin2   <= last_change_bcd;
166         display0 <= last_change_bcd;
167         display1 <= 4'hF;
168       end
169       4'h9: begin
170         keyin3   <= last_change_bcd;
171         display0 <= last_change_bcd;
172         display1 <= display0;
173       end
```

上述狀態除了將使用者的 input 存入被加數或加數的十位數或個位數之外，同時也將 BCD 存入 display0 ~ display3，以即時反應使用者的輸入於顯示器上。而按下 enter 時，則將上述數字的運算結果顯示於顯示器上，如下。

```
173       end
174       4'hB: begin
175         display0 <= answer % 10;
176         display1 <= answer / 10;
177       end
```

以上就是所有可能需要執行的動作。

上述沒有提及的那些偶數的 state 則不做任何事情，如下。

```
178         default: begin
179           keyin0   <= keyin0;
180           keyin1   <= keyin1;
181           keyin2   <= keyin2;
182           keyin3   <= keyin3;
183           operator <= operator;
184           display0 <= display0;
185           display1 <= display1;
186           display2 <= display2;
187           display3 <= display3;
188         end
189       endcase
190     end
191   end
```

4. Implement

首先，把 KeyboardDecoder.v 的 last_change 用 look-up table 轉換成 BCD，只要把所有 0~9 都列出來即可。同時，我特別將加減乘、enter 編碼成 4'hA、4'hB、4'hC、4'hD，如下。

```
34  reg [8:0] last_change_bcd;
35  localparam ADDITION       = 4'hA;
36  localparam SUBTRACTION    = 4'hB;
37  localparam MULTIPLICATION = 4'hC;
38  localparam ENTER          = 4'hD;
```

```
64      // operator
65      8'h1C:   last_change_bcd = ADDITION;
66      8'h1B:   last_change_bcd = SUBTRACTION;
67      8'h3A:   last_change_bcd = MULTIPLICATION;
68      // enter
69      8'h5A:   last_change_bcd = ENTER;
```

經過 FSM 之後，只要再連接七段顯示器的控制模組即可，其作法為：先用除頻器產生適當的頻率（1/2^20），再用 MUX 依序 enable 四個 anode，並將對應的 display0 ~ display3 傳送到 BCD to SSD decoder。控制七段顯示器模組的部分跟上一次 LAB 一樣，只是 reg 的名稱換掉了而已，詳參.v 檔。

5. Conclusion

這部分的 FSM 複雜了一點，我一開始還把控制 keyin 與 display 的描述拆開來寫，結果發現沒有比較簡單易懂，反而還讓我搞混十位數和個位數的輸入順序（例如，state1 的時候，我誤將 BCD 存入了 keyin0，而在 state3 的時候，將 BCD 存入 keyin1，結果讓我的兩個數字的十位數跟個位數剛好倒過來，而且完全無法從顯示器察覺到異狀，因為 display 是接對的）。

本題可以學習到較複雜的 FSM 建構方法。我學期初時完全沒有想到我居然可以實作出一台計算機，我覺得我進步了許多。

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 on 7-bit LEDs.
4.2 Implement the combinational keys. When you press "Shift" and the letter keys at the same time, 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.

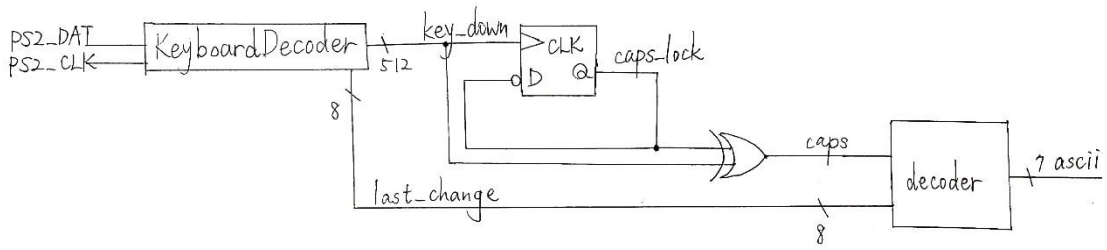
1. Specification

```
module LAB9_4_top(
  output           caps,       // capital/lowercase
  output reg [6:0] ascii,      // pressed key

  inout PS2_DATA,              // (PIC24) USB HID Host
  inout PS2_CLK,               // (PIC24) USB HID Host

  input rst_n,                 // active low reset
  input clk                    // 100MHz global clock
);
```

2. Block Diagram



3. Finite state machine

無

4. Implement

接好 KeyboardDecoder.v。本題只需要使用其中的 key_down（用來判斷 capslock 與 shift）、與 last_change（用來判斷按下的英文字母是 a ~ z 中的哪一個）。

```
12  wire [511:0] key_down;
13  wire [8:0]   last_change;
14  wire         key_valid;
15
16  KeyboardDecoder KeyboardDecoder(
17    .key_down,
18    .last_change,
19    .key_valid,
20    .PS2_DATA,
21    .PS2_CLK,
22    .rst(~rst_n),
23    .clk
24  );
```

判斷應轉換成大寫還是小寫。用 T Flip-flop 來儲存當下的 caps_lock，再跟 shift 做 XOR，就可以知道要大寫還是小寫。其中 9'h0_12 與 9'h0_59 分別為左邊和右邊的 shift；9'h0_58 則是鍵盤上的 caps Lock 按鍵 scan-code。

```
26  reg caps_lock;
27
28  always @(negedge key_down[9'h0_58] or negedge rst_n) begin
29    if (~rst_n) caps_lock <= 0;
30    else        caps_lock <= ~caps_lock;
31  end
32
33  assign caps = caps_lock ^ (key_down[9'h0_12] | key_down[9'h0_59]);
```

最後，將判斷好的大小寫訊號 caps 傳送到 MUX，就可以產生對應的 ASCII code。內容只是把所有大寫和小寫的字母 scan-code 一一列出而已，詳參.v 檔。

5. Conclusion

這題相對第三題來說單純很多，完全不需要使用到 FSM 就可以完成（其實理論上用來儲存 caps 的 Flip-flop 可以算是一種只有兩種 state 的 FSM）。除了老師提供的 KeyboardDecoder.v 之外，都是應用之前 LAB 的觀念就好。