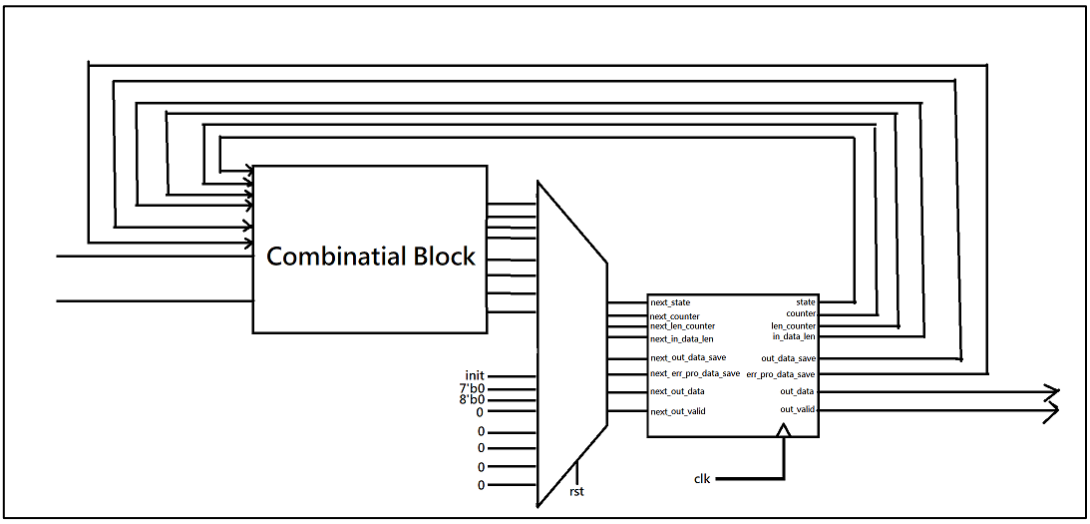
Lab 2

學號:109062110 姓名:祝語辰

A. Lab Implementation

a. Lab2_1:

1、 Block diagram:



我的整個設計的主要結構就是向上圖一樣，兩個 input，兩個 output，大部分的計算都是在 combinatial block 中完成。

2、 Kernel code explanation:

```
    end
    encrypt : begin
        if (len_counter != in_data_len) begin
            next_output_data_save[len_counter] = output_data_save[len_counter] + counter;
        end
    end
    protect:begin
        if (len_counter != in_data_len) begin
            for(i=0, j=0; i<12; i=i+1)begin
                case(i)
                    0:next_err_pro_data_save[len_counter][0] = output_data_save[len_counter][6] ^ output_data_save[len_counter][4] ^ output_data_save[len_counter][3] ^ output_data_save[len_counter][1] ^ output_data_save[len_counter][0];
                    1:next_err_pro_data_save[len_counter][1] = output_data_save[len_counter][6] ^ output_data_save[len_counter][5] ^ output_data_save[len_counter][3] ^ output_data_save[len_counter][2] ^ output_data_save[len_counter][0];
                    3:next_err_pro_data_save[len_counter][3] = output_data_save[len_counter][7] ^ output_data_save[len_counter][3] ^ output_data_save[len_counter][2] ^ output_data_save[len_counter][1];
                    7:next_err_pro_data_save[len_counter][7] = output_data_save[len_counter][7] ^ output_data_save[len_counter][6] ^ output_data_save[len_counter][5] ^ output_data_save[len_counter][4];
                    default:begin
                        next_err_pro_data_save[len_counter][i] = output_data_save[len_counter][j];
                        j=j+1;
                    end
                endcase
            end

        end
    end
```

以上是我 lab2_1 的核心 code。

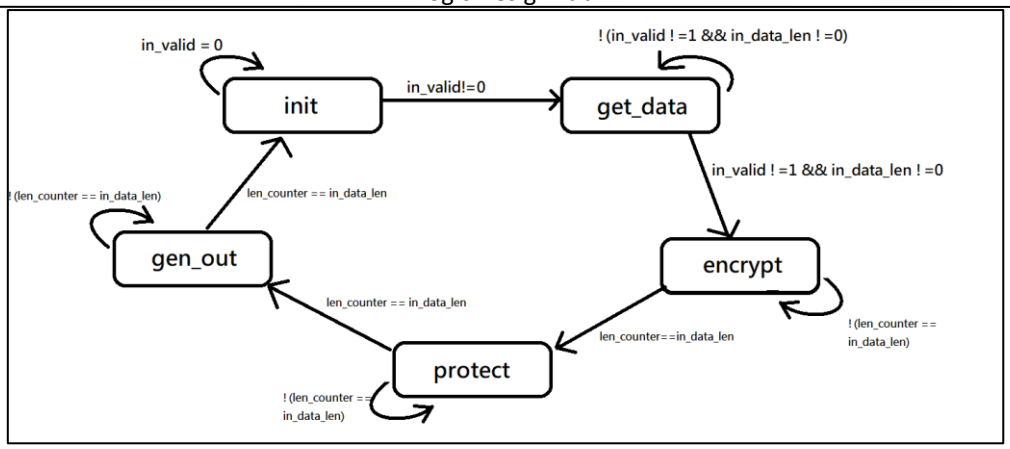
主要就是在這裡處理加密的部分。

在 encrpt state 時，會使用 len_counter 以及 counter 來做加密。

因為 counter 限制是 0 到 127，如果直接使用他來記錄當前處理到哪個 input Data 時會出錯，所以適用兩個 counter 來解決。

然後再 err protect 的部分，則時直接用列舉的方式來處理。

3、 FSM:



因為一次會有很多的 input，所以會有 in_valid 的訊號來確認當前輸入的值是否是有效的。所以當 invalid 訊號從無效到有效，就代表開始輸入 data 了，這是後就會進到 get_data state 中。

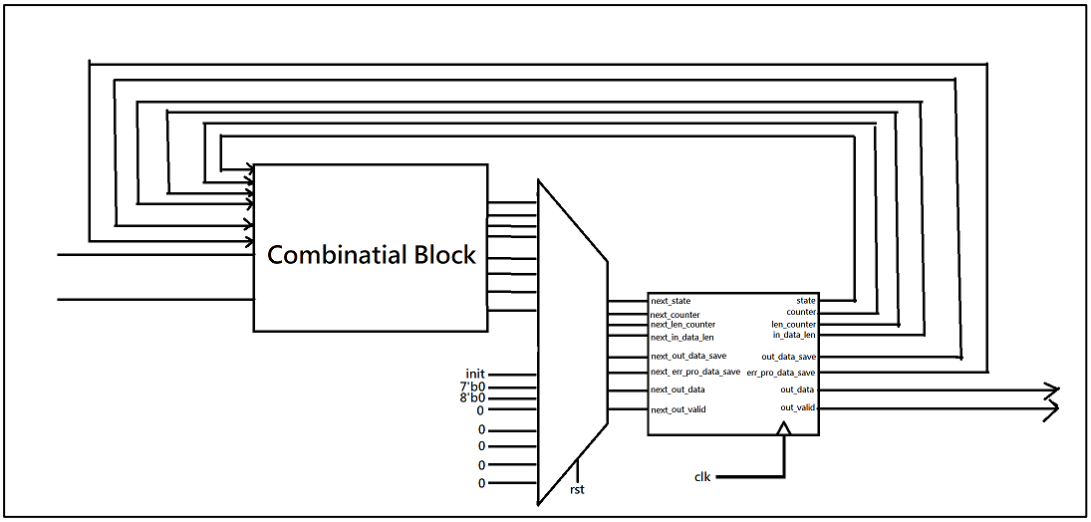
在離開 get_data 時，同樣也 in_valid 的值也會變動，從有效輸入到無效輸入。而且因為設計的關係，至少會有一個有效的輸入值，所以在離開 get_data 時，in_valid 會變成 0 且 in_data_len 的值不為 0，此時就會進入到 encrypt state 中。

在 encrypt /protect/ gen_out state，我們會依序對每一個 data 做加密、err 保護處理，以及輸出，所以在各個 state 中，使用 len_counter 來記錄，一旦 len_counter 等於 in_data_len，就代表處理完成，進入到下一個 state。

b. Lab2_2:

1、 Block diagram:

Lab2_2 的設計和 lab2_1 類似，差別就在變數名稱不同，以及 kernal code 的部分。故在這裡的 block diagram 也放同樣的。



2、 Kernel code explanation:

```
detect_err : begin
    if (len_counter != one_bit_err_in_data_len) begin
        next_parity[len_counter][0] = data_save[len_counter][10] ^ data_save[len_counter][8] ^ data_save[len_counter][6] ^ data_save[len_counter][4] ^ data_save[len_counter][2] ^ data_save[len_counter][0];
        next_parity[len_counter][1] = data_save[len_counter][10] ^ data_save[len_counter][9] ^ data_save[len_counter][6] ^ data_save[len_counter][5] ^ data_save[len_counter][2] ^ data_save[len_counter][1];
        next_parity[len_counter][2] = data_save[len_counter][11] ^ data_save[len_counter][6] ^ data_save[len_counter][5] ^ data_save[len_counter][4] ^ data_save[len_counter][3];
        next_parity[len_counter][3] = data_save[len_counter][11] ^ data_save[len_counter][10] ^ data_save[len_counter][9] ^ data_save[len_counter][8] ^ data_save[len_counter][7];
    end
end
fix_err : begin
    if (len_counter != one_bit_err_in_data_len && parity[len_counter] != 0) begin
        if(data_save[len_counter][parity[len_counter]-1] == 0)begin
            next_data_save[len_counter][parity[len_counter]-1] = 1;
        end
        else begin
            next_data_save[len_counter][parity[len_counter]-1] = 0;
        end
    end
end
decrypt:begin
    if (len_counter != one_bit_err_in_data_len) begin
        next_plaintext_save[len_counter] = {data_save[len_counter][11:8],data_save[len_counter][6:4],data_save[len_counter][2]};
    end
end
```

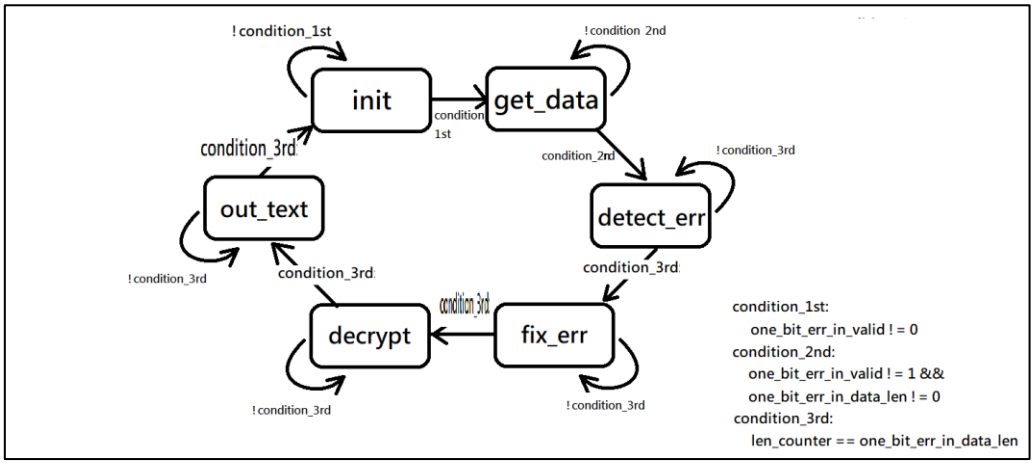
以上是我 lab2_2 的核心 code。

主要就是在這裡處理解密的部分。

邏輯上和 lab2_1 相似，差別在於，我使用了 parity 來存取 err detect 後出來的結果。

因為該結果(存在 parity 中)所表示的位置是從 1 開始標號的，但在我的 code 中，儲存是從 0 開始的，因此這個事後要做微調。

3、 FSM:



這裡的 FSM 和 lab2_1 類似。因為一次會有很多的 input，所以會有 one_bit_err_in_valid 的訊號來確認當前輸入的值是否是有效的。所以當 one_bit_err_in_valid 訊號從無效到有效，就代表開始輸入 data 了，這是後就會進到 get_data state 中。

在離開 get_data 時，同樣也 one_bit_err_in_valid 的值也會變動，從有效輸入到無效輸入。而且因為設計的關係，至少會有一個有效的輸入值，所以在離開 get_data 時，one_bit_err_in_valid 會變成 0 且 one_bit_err_in_data_len 的值不為 0，此時就會進入到 encrypt state 中。

在 detect _err/fix_err /decrypt/ out_text state，我們會依序對每一個 data 做 g 是否有 err 的檢測，處理 err、解碼，以及輸出明文，所以在各個 state 中，使用 len_counter 來記錄，一旦 len_counter 等於 one_bit_err_in_data_len，就代表處理完成，進入到下一個 state。

B. Questions and Discussions

1. In this lab, our reset signal is a synchronous reset. What if it is an asynchronous reset? And how to modify your design to implement an asynchronous reset?

在現在的 design 中，我即使 reset 的信號被拉起，也必須要等到 positive clock edge 才會做 reset 該做的事。但如果是 asynchronous reset，reset 的信號會被加入到 always block 的 sensitive list。這個時候，該 always block 的內容，除了在 clk edge 會執行外，在 reset 訊號有變動時，也會執行。這時候，就可以非同步的進行 reset。

- synchronous reset:

```
always@(posedge clk)begin
    if(rst)begin
        …
end
```

- asynchronous reset:

```
always@(posedge clk or posedge reset)begin
    if(rst)begin
        …
end
```

2. If you are not allowed to use for-loop to initialize your memory, what should you do instead?

在數量不多的時候，用列舉的方式來完成。

C. Problem Encountered

碰到的最大問題，就是 coding style 的部分，常常寫出一個邏輯上看起來沒問題，但實際上跑起來卻 crash 的情形。因為這一次的 lab 中，有比較多內部的訊號要處理，外加是要用 FSM 所以一開始嘗試的時候問題比較大，自己嘗試從無到有開始寫時，碰到了許多困難。但幸好有助教提供的 template，把作業的難度下降了許多，並且在也使我的 coding style 有所改善。在日後的 lab 也可以做為參考。

D. Suggestions

無