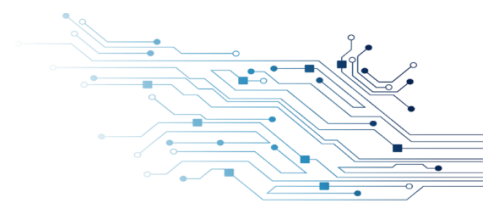




題目：Multiplexer(8-bit input & 16-bit output)





1. 實驗程式碼

```
module LAB3 (KEY, SW, LEDR, HEX3, HEX2, HEX1, HEX0);
    input [3:0] KEY;
    input [9:0] SW;
    output [9:9] LEDR;
    output [6:0] HEX3, HEX2, HEX1, HEX0;
    reg [7:0] A, B, C, D;
    wire [15:0] resultAB, resultCD, sum;
    reg [15:0] num_display;

    //sum = (A x B) + (C x D)
    mult AB(.dataa(A), .datab(B), .result(resultAB));
    mult CD(.dataa(C), .datab(D), .result(resultCD));
    add ABCD(.dataa(resultAB), .datab(resultCD), .cout(LEDR), .result(sum));
    //display {A,B}, {C,D} or sum
    four_digit_seg display(num_display, HEX3, HEX2, HEX1, HEX0);

    //KEY[1] is manual input clock, KEY[0] is active-low asynchronous reset
    always @(posedge KEY[1] or negedge KEY[0]) begin
        if(!KEY[0]) begin
            A = 0;
            B = 0;
            C = 0;
            D = 0;
        end
        else begin
            //SW[9] is write enable
            if(SW[9] == 1) begin
                //SW[8] decides A&B or C&D
                if(SW[8] == 1) begin
                    //KEY[2] decides A&C or B&D
                    if(KEY[2] == 1) begin
                        A = SW[7:0];
                    end
                    else begin
                        B = SW[7:0];
                    end
                end
                else begin
                    if(KEY[2] == 1) begin
                        C = SW[7:0];
                    end
                    else begin
                        D = SW[7:0];
                    end
                end
            end
        end
    end
end
```

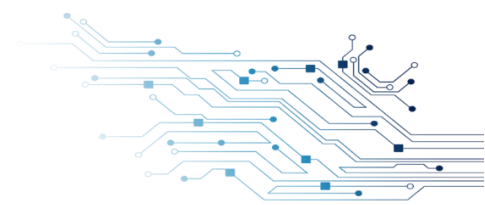




```
//KEY[3]&SW[8] decides display
always @(*) begin
    if(KEY[3] == 0) begin
        num_display = sum;
    end
    else begin
        if(SW[8] == 1) begin
            num_display = {A, B};
        end
        else begin
            num_display = {C, D};
        end
    end
end
endmodule
```

```
//bch to seg
module seg_decoder (bch, seg);
    input [3:0] bch;
    output [6:0] seg;
    reg [6:0] seg;

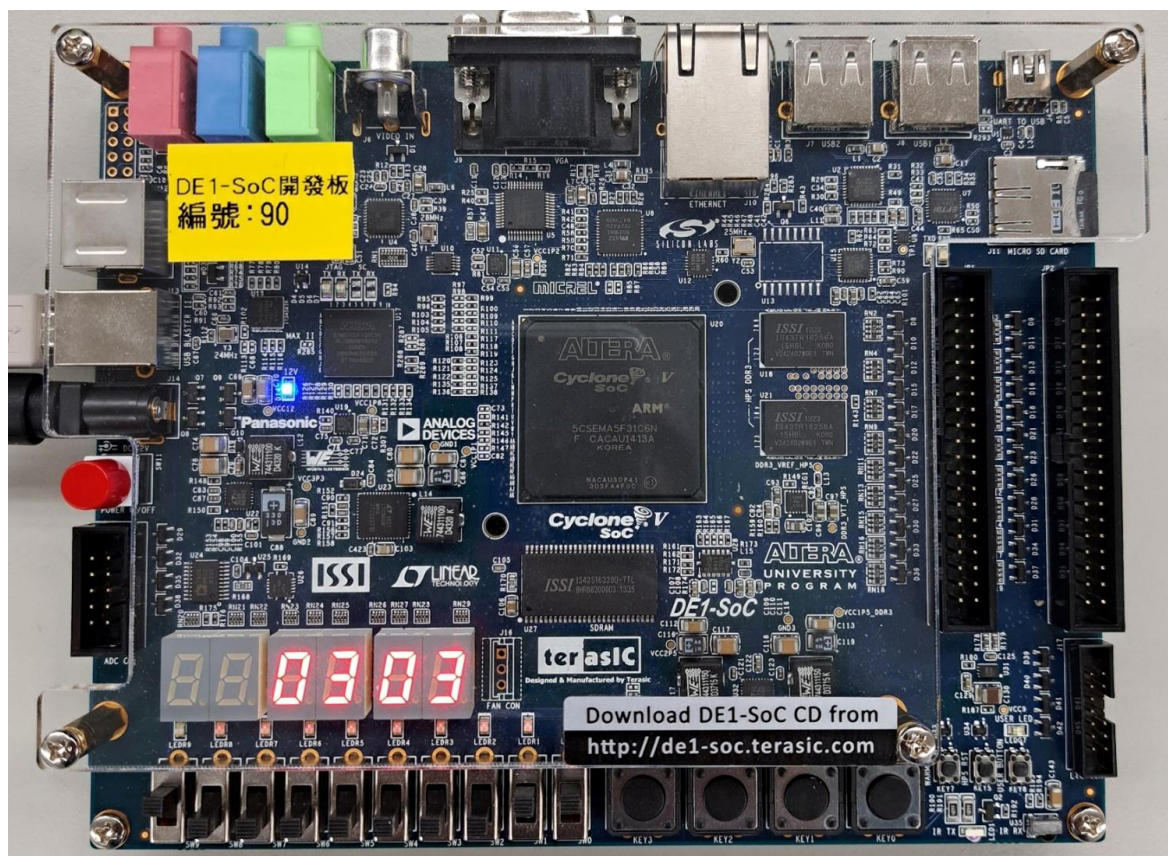
    always @(bch) begin
        case (bch)
            0 : seg = 7'b1000000;
            1 : seg = 7'b1111001;
            2 : seg = 7'b0100100;
            3 : seg = 7'b0110000;
            4 : seg = 7'b0011001;
            5 : seg = 7'b0010010;
            6 : seg = 7'b0000010;
            7 : seg = 7'b1111000;
            8 : seg = 7'b0000000;
            9 : seg = 7'b0010000;
            10 : seg = 7'b0001000; //A
            11 : seg = 7'b0000011; //b
            12 : seg = 7'b1000110; //C
            13 : seg = 7'b0100001; //d
            14 : seg = 7'b0000110; //E
            15 : seg = 7'b0001110; //F
            //lights out
            default : seg = 7'b1111111;
        endcase
    end
endmodule
```

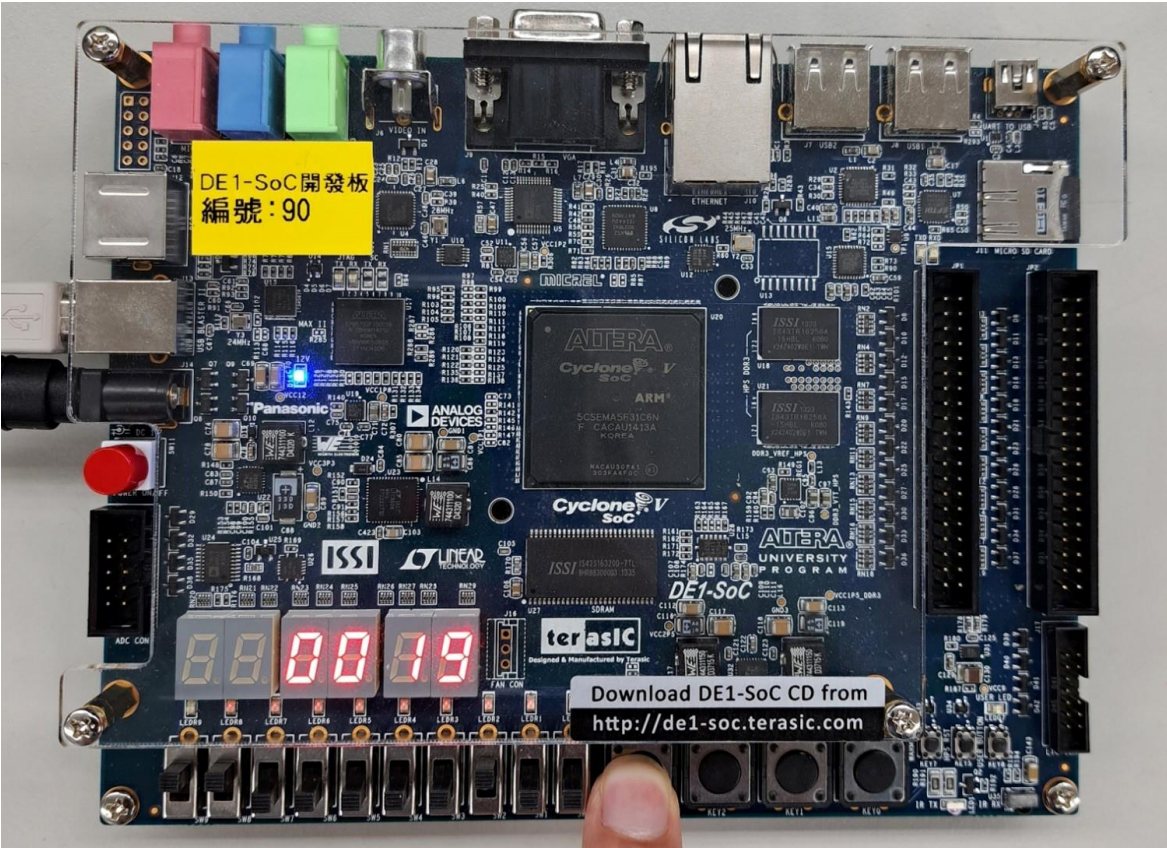
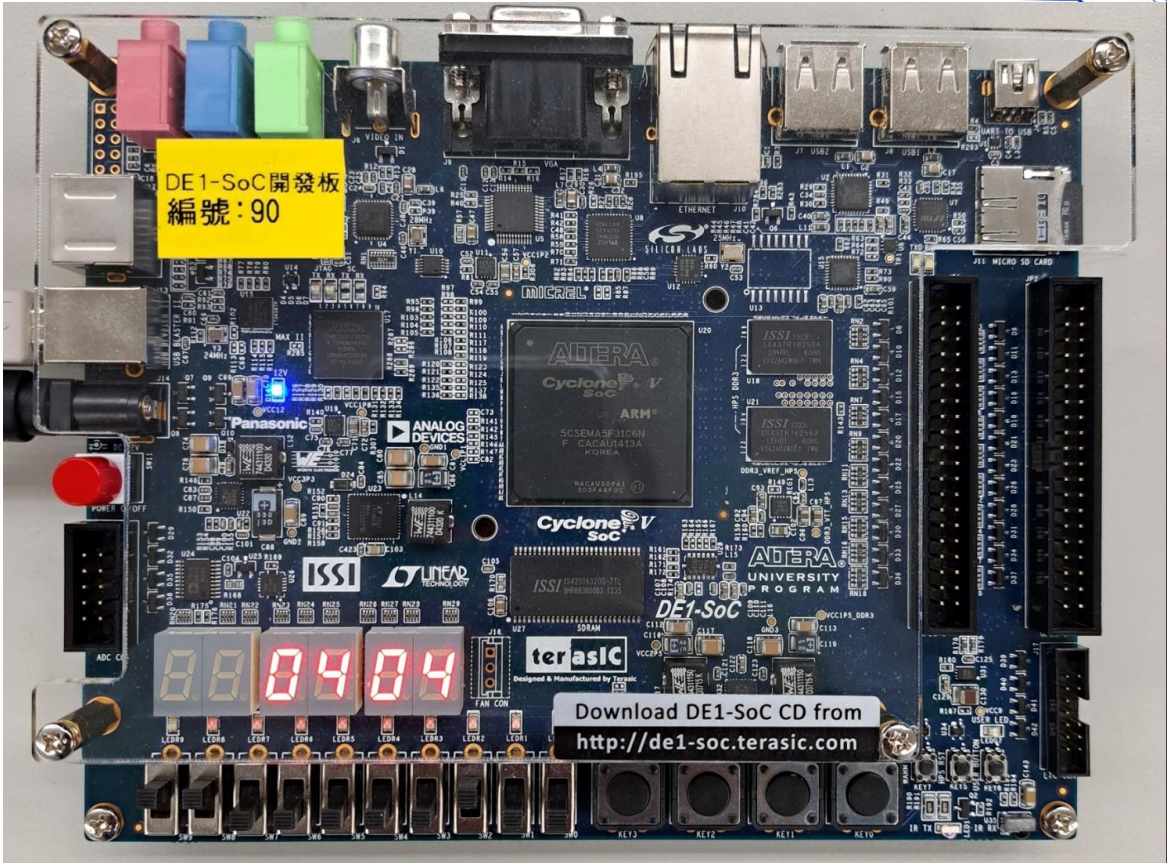


```
//display four digit in seg
module four_digit_seg (num, seg3, seg2, seg1, seg0);
    input [15:0] num;
    output [6:0] seg3, seg2, seg1, seg0;

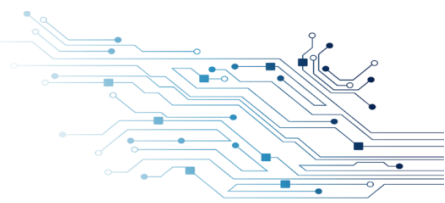
    seg_decoder de3(num[15:12], seg3);
    seg_decoder de2(num[11:8], seg2);
    seg_decoder de1(num[7:4], seg1);
    seg_decoder de0(num[3:0], seg0);
endmodule
```

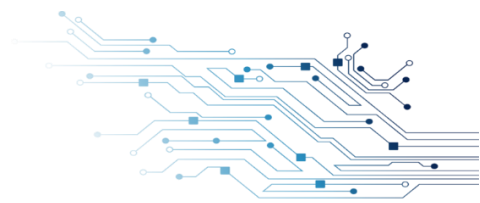
2. 實驗結果

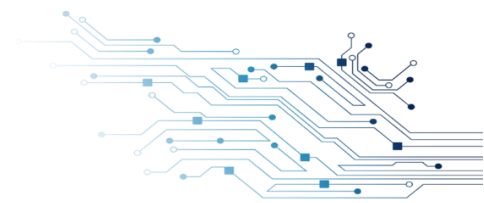
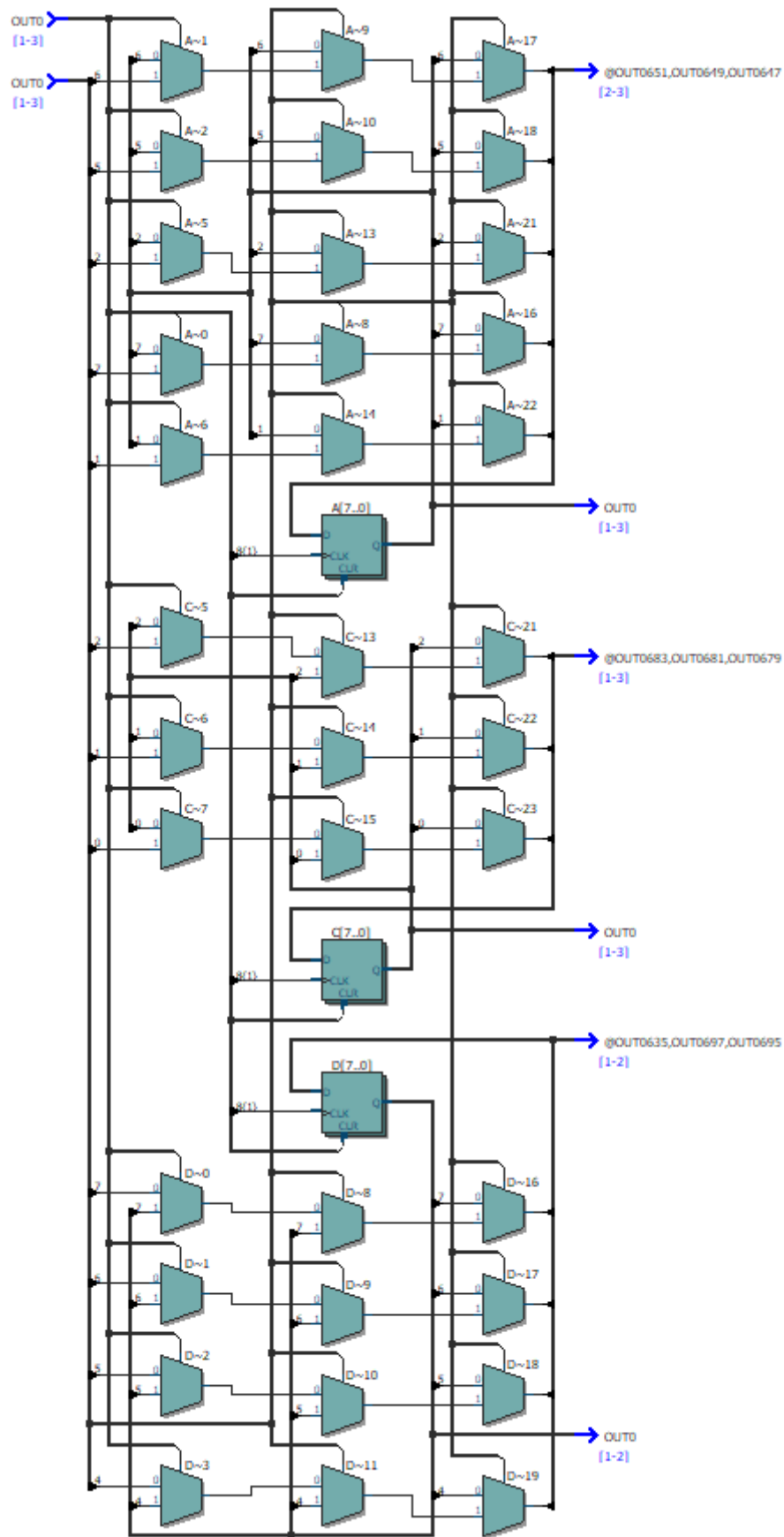


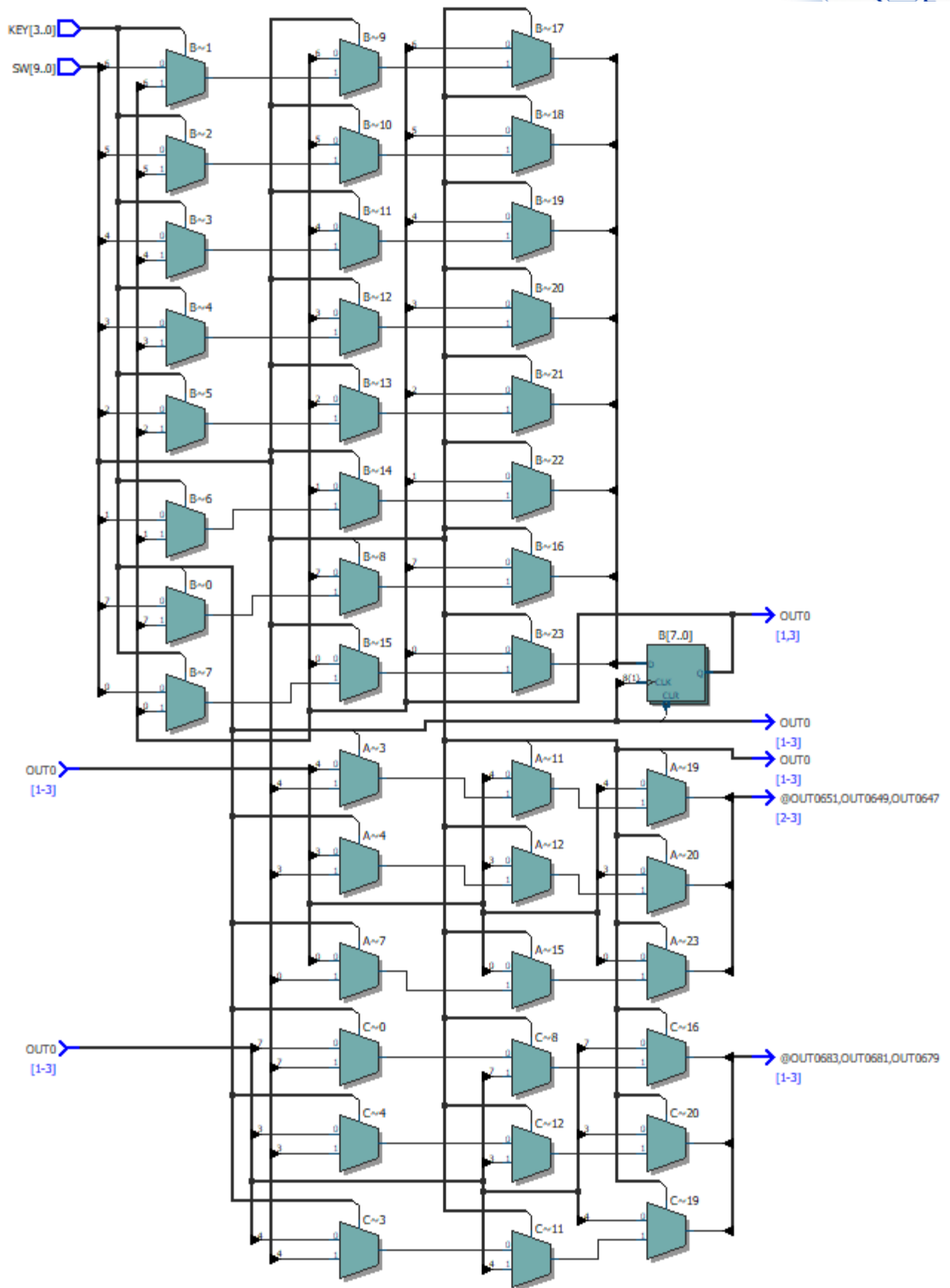


3. RTL 佈局










4. 問題與討論


不能在 always block 裡面初始化 module，因為 flip-flop 裡面不能有其他電路。解決辦法是



在 `always block` 外面初始化該 `module`，然後在 `always block` 裡面去判斷訊號要到哪一個 `module`。在此次練習中，我將 AB 合併和 CD 合併作為 `flip-flop` 的一些輸入，輸出再拉線給 `four_digit_seg` 顯示，不需要在 `flip-flop` 裡面直接顯示，而是做好最後的結果才輸出。

- ✚ `always block` 裡面盡量不要寫多個 `if`，因為不能確定是並行還是順序執行，甚至執行順序都不確定。在此次練習中因為這樣合成了運作不正常的電路，將其分成不同的 `always block` 就能正常運行。
- ✚ 宣告任意變數時，一定會確認有多少位寬，之後使用該變數就不須描述位寬範圍，除非使用特定範圍才要描述。
- ✚ `always block` 如果訊號不是 `edge`，那麼建議都用*，雖然不影響合成的電路，但模擬的波形圖不同(因為觸發訊號不同)，這樣不方便確認是否正確。
- ✚ quartus 13.1 用 megawizard 去 instantiate IP；quartus 20.1 用 catalog IP 去 instantiate IP，使用別人寫的 `module` 可節省時間。

