109511207 電機 13 蔡宗儒

(一) 如何設計作業

使用兩個 Sequential Logic，一個為計數器，數 15 個 Cycle 後重置，用來得知目前是第幾個 Cycle；另一個為累加器，得知目前的總和為多少。計算的部分由 Combinational Logic 去計算 cnt_nxt 與 sum_nxt 為多少，並傳回兩個 Flip-Flop。out_valid 和 out 則都藉由 cnt 是否等於 15 做判斷，是的話 out_valid 輸出 High，out 輸出用 sum 做計算後的結果(1-9 or 15)；否則 out_valid 輸出 Low，out 輸出 0，架構圖如下圖 1。

(二) 架構圖

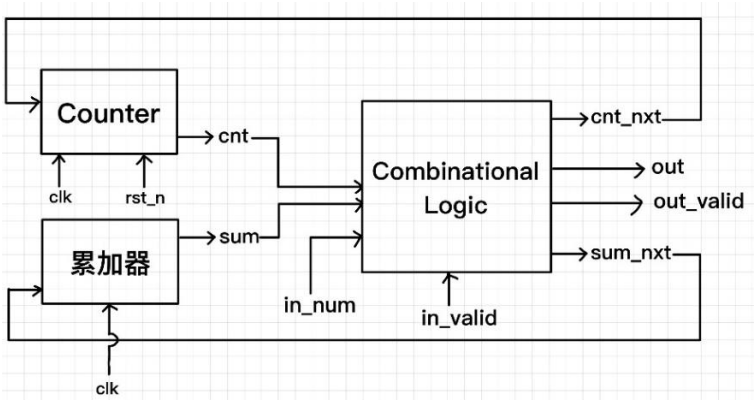

圖 1

(三) 遇到的困難與解決方法

1. 如何減少計算量

這次作業有需要乘 2 和取餘數，所以如何減少不必要的運算是一大重點。乘 2 方面其實可以很簡單的解決，因為不管 in_num 為多少，每次計算都會有分別對應要加的數值(ex. in_num = 5 時，sum 要加 1；in_num = 7 時，sum 要加 5)，所以我設了一個 wire 為 in_temp，並使用 case 來存每次要加的數值(根據為第奇數個還是第偶數個 Cycle 來判斷)，如下圖 2。

```
always_comb begin
    if(cnt[0] == 1)
        in_temp = in_num;
    else begin
        case (in_num)
            0 : in_temp = 0;
            1 : in_temp = 2;
            2 : in_temp = 4;
            3 : in_temp = 6;
            4 : in_temp = 8;
            5 : in_temp = 1;
            6 : in_temp = 3;
            7 : in_temp = 5;
            8 : in_temp = 7;
            9 : in_temp = 9;
            default : in_temp = in_num;
        endcase
    end
end
```

圖 2

取餘數方面我想了很久，一開始是用最直覺的%10，但是用%會需要很大的面積，後來想到其實可以在每次計算 sum_nxt 前判斷是否會大於等於 10，會的話將值扣掉 10 即可，這樣不只簡單，也節省了大量的面積，如下圖 3。

```
assign sum_nxt = (sum + in_temp >= 10) ? (sum + in_temp - 10) : (sum + in_temp);
```

圖 3

2. 跟同學討論時發現有些人在 case 內直接算好 sum_nxt，這樣 Code 少很多行，也可以少變數，但為什麼面積反而翻倍(Code 如下圖 4)？

這是因為 Verilog 不能用軟體語言來想，並不是越少行 Code 或是變數越少，面積就越小。圖 4 的方法在硬體上其實就是在 Case 產生的多工器內做了很多判斷跟運算，然而這樣等於 Case 0~9 都需要把他們各自的電路出來，才能符合 0~9 各自的需求，但這樣要把 0~9 的多工器跟加/減法器都做出來，會導致面積大幅上升。但如果拆開來用圖 2 以及圖 3 的方式做，在硬體上表示 Case 產生的多工器只要讓變數 in_temp 變成其他定值就好，再將 in_temp 拉出去做加/減法的運算，這樣就不用在內部做複雜的計算。如此一來只需要一個加/減法器做最後的計算，面積的大量差距也是因為這個原因，所以如果可以的話盡量在 Case 外做運算，Case 內則不要做太多運算。

```
always_comb begin
    if(cnt[0] == 1)
        sum_nxt = (sum + in_num >= 10) ? (sum + in_num - 10) : (sum + in_num);
    else begin
        case (in_num)
            0 : sum_nxt = sum;
            1 : sum_nxt = (sum + 2 >= 10) ? (sum - 8) : (sum + 2);
            2 : sum_nxt = (sum + 4 >= 10) ? (sum - 6) : (sum + 4);
            3 : sum_nxt = (sum + 6 >= 10) ? (sum - 4) : (sum + 6);
            4 : sum_nxt = (sum + 8 >= 10) ? (sum - 2) : (sum + 8);
            5 : sum_nxt = (sum + 1 >= 10) ? (sum - 9) : (sum + 1);
            6 : sum_nxt = (sum + 3 >= 10) ? (sum - 7) : (sum + 3);
            7 : sum_nxt = (sum + 5 >= 10) ? (sum - 5) : (sum + 5);
            8 : sum_nxt = (sum + 7 >= 10) ? (sum - 3) : (sum + 7);
            9 : sum_nxt = (sum + 9 >= 10) ? (sum - 1) : (sum + 9);
            default : sum_nxt = (sum + in_num >= 10) ? (sum + in_num - 10) : (sum + in_num);
        endcase
    end
end
```

圖 4

(四) 心得

我覺得打 Verilog 真的滿有趣的，很多軟體的想法在這邊都不適用，才會有明明 Code 更精簡，面積卻倍增的狀況發生，這個時候應該在心裡把電路想出來，就會知道差異發生在哪裡了，然後把 Sequential Logic 跟 Combinational Logic 分開真的很重要，因為有發現沒有分開可能會導致面積變大(我認為應該是因為有些電路有分開有些沒有，會導致有些線路要繞比較長)，而且分開也能夠讓邏輯更為清楚，我覺得對於剛開始學 Verilog 的初學者，這些基本習慣是很重要的。另外我也學會了一些精簡 Code 的方式，雖然對縮小面積沒有幫助，但是能夠看起來更精簡。像是(cnt == 15) 可以寫成 (&cnt) (如下圖 5)。

```
assign out_valid = (&cnt ) ? 1 : 0;
```

圖 5

(五) Area & Delay

```
Total cell area:          1483.574409
Total area:                 undefined
```

圖 6

```
data required time                                  9.84
data arrival time                                  -8.86
-------------------------------------------------------------
slack (MET)                                         0.98
```

圖 7