Computer Organization 2022

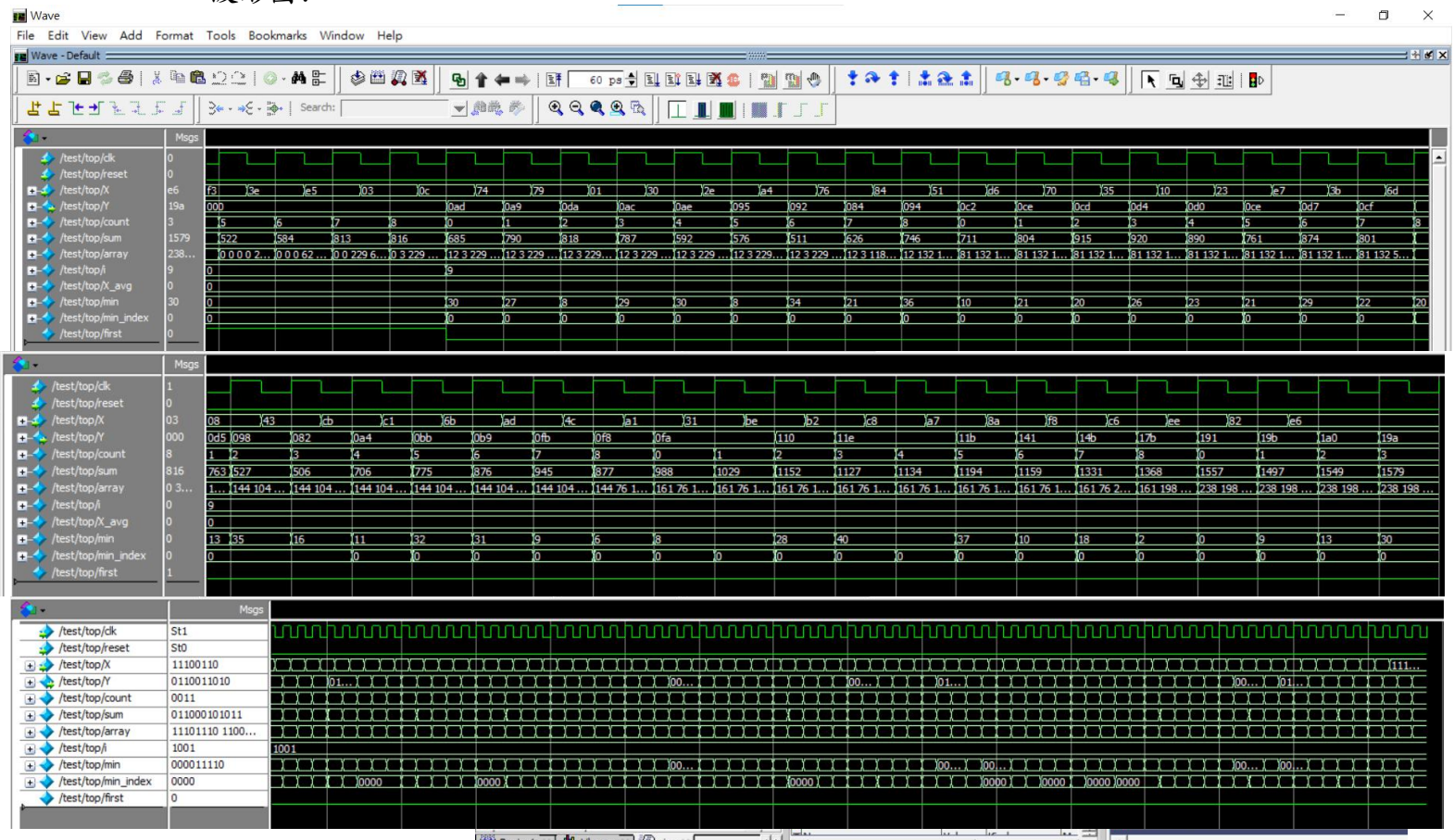
HOMEWORK 1

系級: 資訊工程系 113　學號: F74109016　姓名: 葉惟欣

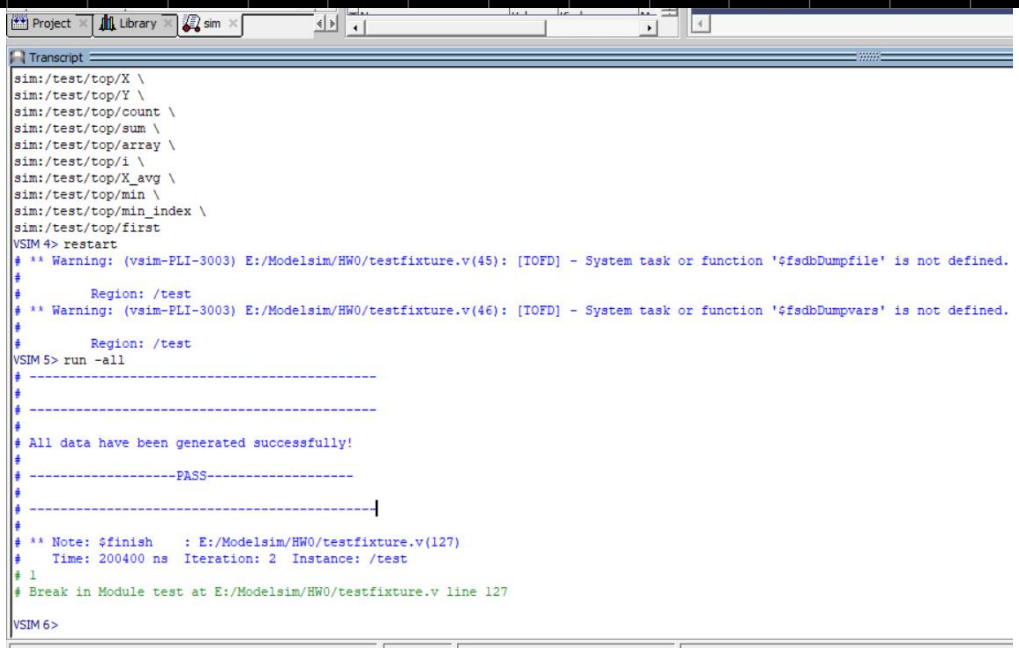
實驗結果圖:

(波形圖及模擬完成截圖)

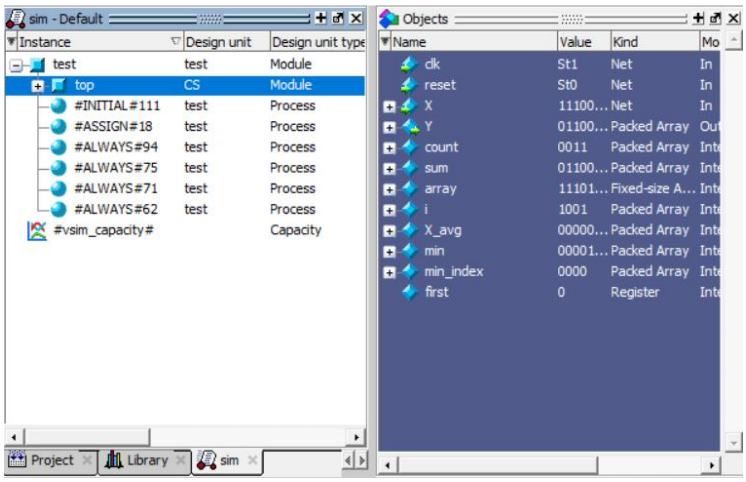
波形圖:



模擬完成截圖:



Signal 為 test 裡面的 top，CS.v 使用到的變數。



程式運作流程:

(簡單說明波形變化的意義)

有兩個 always 一個為循序(sequential)電路，在 clk 為正緣時或 reset 為正緣時觸發。另一個為組合(combination)電路，在每次 count 有變化時此電路都會被觸發。

循序電路 reset 時變數初始化，此時 first 為 true(程式碼:first <= 1'b1;)。如果不是 reset 時，則將 input X 的值存到 array 陣列中，且將 sum 加上新的值。Array 的 index(變數名稱為 count)同時也被加一，將 input X 依序放到每個陣列元素中。此時的 count 變化會觸發組合電路。

在組合電路中，第一個 if 條件判斷(count==9)時，則將 count 重新設為 0(程式碼:count <= 4'b0000;)，代表所有的陣列元素都被放滿了，要從頭開始放變數。再來，又當 first 為 true 時，將 first 設為 false(程式碼:first <= 1'b0;)，代表現在已經不是第一輪。可以做第二個 if 條件判斷(first == 1'b0)。

組合電路中，第二個條件判斷，這裡是在做找尋最接近的值，變數 min 為平均數(sum / 4'b1001)跟陣列元素的差。For 迴圈裏面第一個 if 是在判斷陣列元素是否小於平均值，小於才有機會成為最接近的值。第二個 if 是在判斷是否比當前的最小值小，小的話就將最小值與最接近平均之陣列元素的 index 更新。也因此剛開始將最小值設為非常大。

為了讓這裡的執行順序為一行一行下來，因此在組合電路中都用 = 作為賦值(assign)。而在循序電路中，為了同時執行，因此用 <= 作為賦值(assign)。

Y = #0.6 (sum + 4'b1001 * array[min_index])/ 4'b1000;

此為延時給值。因為 test 檔的 holdtime 為 0.5。

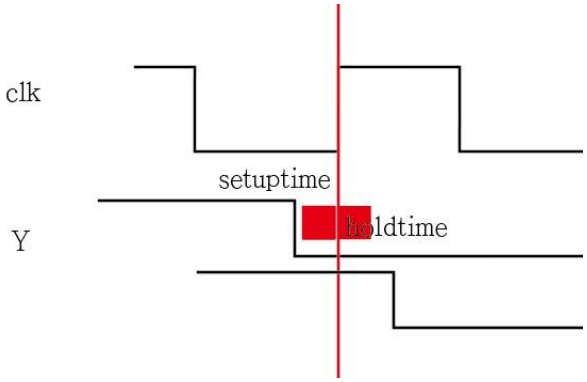
如果不在 holdtime 後給值。則 Y 給的值則是 unknown 如下圖，因此要將 Y 的波形圖延後成第三個波形圖。

Setup time:值必須維持定值，因為要在 clk

```
specify
$setup(Y, posedge pclk, 0.5, flag1);
$hold(posedge pclk, Y, 0.5, flag2);
endspecify
```

來的時候給正確的值。

Hold time:則是必須穩定剛剛進來的值，不要讓新的值覆蓋到先前的值，不然 flipflop 的值會影響當前的輸出。



將算出來的值給 Y 作為 output 後，則將 sum 減到目前的 index 的值(目前 index 的值為剛剛 X 新的 input 放進來後加一，所以相當於是最舊的值)。在 sum 減掉舊的值後，新的值又加入 sum 並覆蓋掉剛剛舊的值。

程式碼

```
`timescale 1ns/10ps
module CS(Y,X, reset, clk);
        input clk, reset;
        input [7:0] X;
        output reg      [9:0] Y;
        reg unsigned [3:0] count;
        reg unsigned [11:0] sum;
        reg unsigned [7:0] array [8:0];
        reg unsigned [3:0] i;
        reg unsigned [8:0] min;
        reg unsigned [3:0] min_index;
        reg first;
        initial
                begin
                        count <= 4'b0000;
                        sum <= 12'b000000000;
                        array[0] <= 8'b00000000;
                        array[1] <= 8'b00000000;
                        array[2] <= 8'b00000000;
                        array[3] <= 8'b00000000;
                        array[4] <= 8'b00000000;
                        array[5] <= 8'b00000000;
                        array[6] <= 8'b00000000;
                        array[7] <= 8'b00000000;
                        array[8] <= 8'b00000000;
                        min <= 9'b000000000;
                        min_index <= 4'b0000;
                        first <= 1'b1;
                        Y <= 10'b0000000000;
                        i<=4'd0;
                end

      always@(posedge clk or posedge reset)
                begin
                        if(reset)
                                begin
                                        sum <= 12'b000000000;
                                        count <= 4'b0000;
                                        Y <= 10'b0000000000;
                                        min_index <= 4'b0000;
                                        first <= 1'b1;
                                end
                        else
                                begin
                                        array[count] <= X;
                                        sum <= sum + X;
                                        count <= count + 4'b0001;
                                end
                end
```

```
always@(count)
        begin
                if(count == 4'b1001)
                        begin
                                count <= 4'b0000;
                                if(first == 1'b1)
                                        begin
                                                first <= 1'b0;
                                        end
                        end
                if(first == 1'b0)
                        begin
                                i = 4'b0000;
                                min = 9'b011111111;
                                min_index = 4'b0000;
                                for(i = 4'b0000;i < 4'b1001;i=i+4'b0001)
                                begin
                                        if((sum / 4'b1001) >= array[i])                 // 平均值 > 陣列元素時才跑if的body
                                        begin
                                                if(min > (sum / 4'b1001 - array[i]))    //目前的最小值 > (平均值 - 陣列元素)
                                                begin
                                                        min = (sum / 4'b1001 - array[i]); //最小值 = (平均值 - 陣列元素)
                                                        min_index = i;                  //最小值的index = 此陣列元素的index
                                                end
                                        end
                                end
                                Y = #0.6 (sum + 4'b1001 * array[min_index])/ 4'b1000;   // (sum + 9 * 最小值) / 8
                                min_index = 4'b0000;
                                sum = sum - array[count];
                        end
        end
endmodule
```

心得

（請寫下完成本次作業的心得、學到哪些東西、困難點的部分。）

這次的作業一開始在操作 Modelsim 時遇到一些困難，像是起初都是用 test 去跑波形圖，後來才用 test 裡的 top 去跑波形圖。

在寫 verilog 遇到最大的困難是 debug 的部分，原本不知道 for 迴圈是一次開相對應的 register，而不是像 C/C++是一條一條執行下去，跟自己想的差距很大。也因此原本我都沒有特別注意，nonblocking 與 blocking 的差別，認為 verilog 都是如自己想像的一行一行執行下去。但在寫完此次作業後，我才了解，原來程式如果要設計成循序執行時要用 blocking，此時的賦值用 =。而如果程式要設計成同步執行，則是用 nonblocking，此時的賦值用<=。

程式碼第一塊就是循序電路，在 clk 為正緣或是 reset 為正源時，則執行右邊的語句。這些語句適用 nonblcoking，寫的一開始，我是寫成 array[count] <= X;

sum <= sum + array[count];

後來才發現這跟我想的不太一樣，因為 array[count] 跟 sum 加的其實是還沒改變前的值，因為 nonblocking 為同時執行。之後就改成現在的樣子。

```
always@(posedge clk or posedge reset)
        begin
                if(reset)
                        begin
                                sum <= 12'b000000000;
                                count <= 4'b0000;
                                Y <= 10'b0000000000;
                                min_index <= 4'b0000;
                                first <= 1'b1;
                        end
                else
                        begin
                                array[count] <= X;
                                sum <= sum + X;
                                count <= count + 4'b0001;
                        end
end
```

原本的我把所有程式碼都寫成 nonblocking ，後來了解到像是 for 迴圈、有先後順序的數學運算、找最接近的值後再與原本的 array 裡的元素做相加，這些有先後步驟的 statement 都需要另外再寫在 blocking 裡，而一個 always

裡面就只能是 blocking 或是 nonblocking，不能兩者同時存在。

而第二個 always 的部分，我原本都是在時脈為正緣後觸發，後來思考過後發現，應該要等 input 已經存進去 sum 與 array 裡，count 也要變動後才開始來找最接近的值，因此把他拉出來由 count 改變時觸發。

在 debug 中我遇到最困難的部分就是有 hold time 的問題，後來我去看測知的 verilog 才發現原來資料要穩定 0.5 個時間單位，因此我在讀值的時候應該要在大於 0.6 個時間單位後讀值才不會有資料不穩定的問題。

```
specify
$setup(Y, posedge pclk, 0.5, flag1);
$hold(posedge pclk, Y, 0.5, flag2);
endspecify
```

經過這次作業我了解很多原先只是理論的東西現在真正的被用出來，像是 setup time 與 hold Time，循序電路，組合電路，nonblocking 與 Blocking 的觀念都透過這次作業更加強化了。希望下一次寫作業時能夠流暢的完成作業。