HW3 電路設計說明

第14組

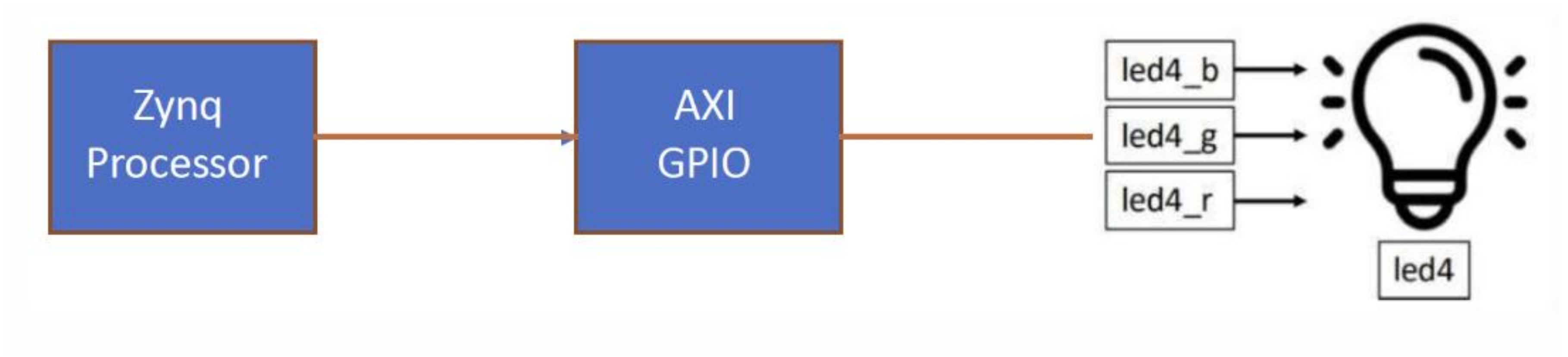
黃偉峰 E34106010

陳識博 E94106096

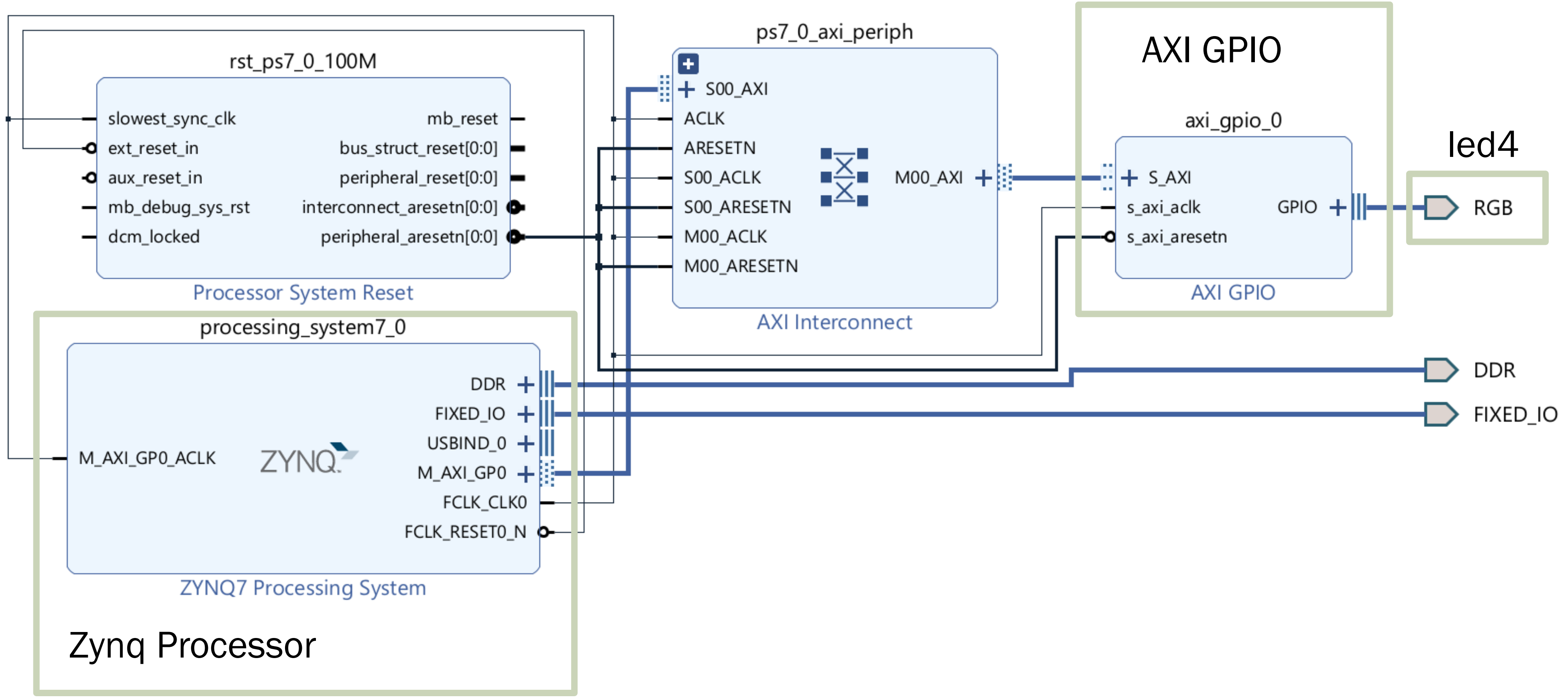
黃芊 F74104040

Problem 1 – RGB LED

Block Diagram

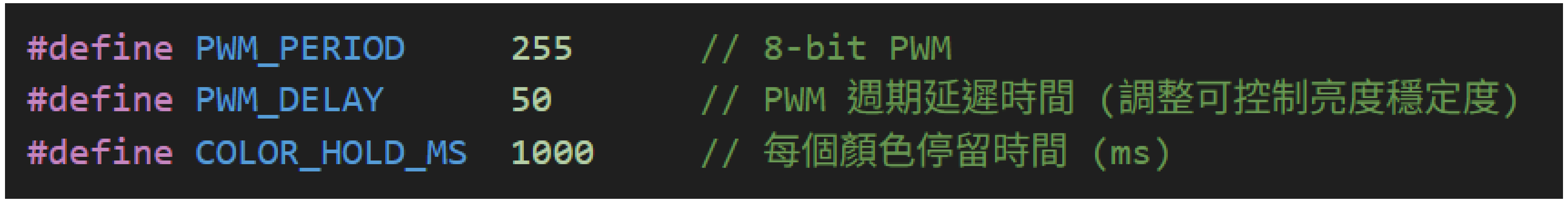


Block Diagram (Our Design)



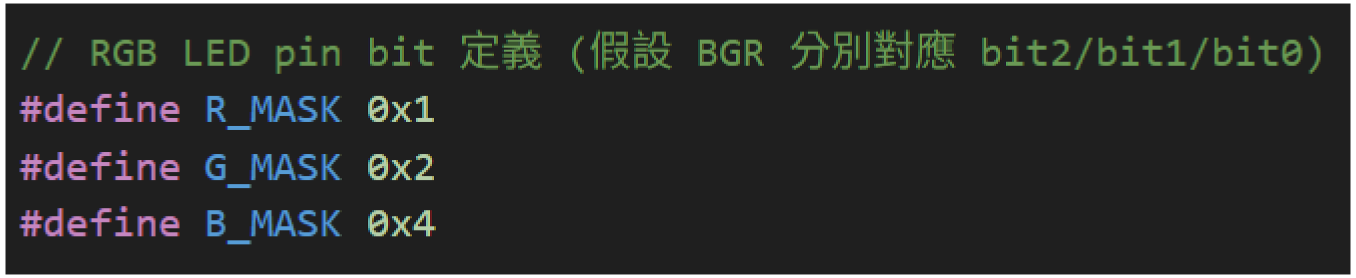
Software 設計

變數定義



```
#define PWM_PERIOD     255      // 8-bit PWM
#define PWM_DELAY      50       // PWM 週期延遲時間 (調整可控制亮度穩定度)
#define COLOR_HOLD_MS  1000     // 每個顏色停留時間 (ms)
```

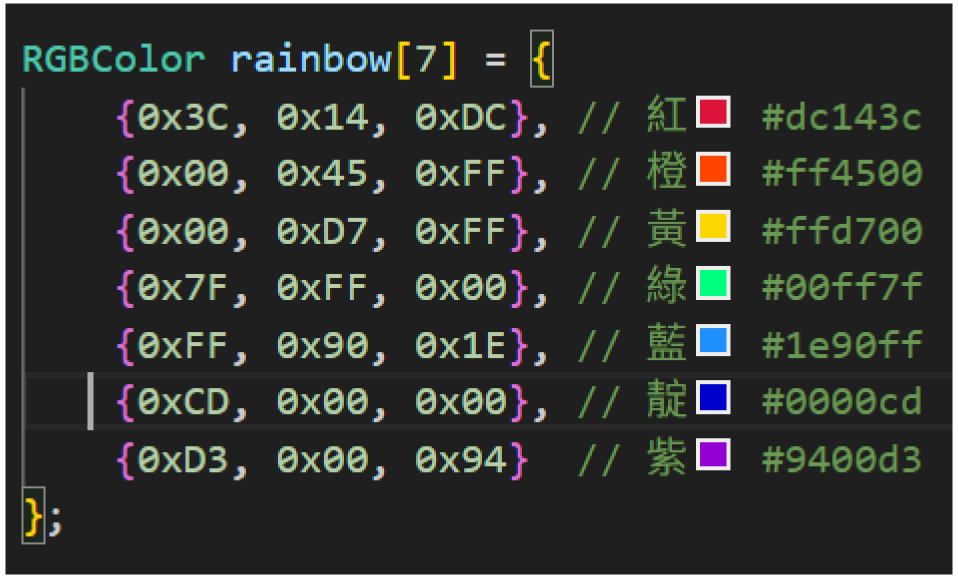
- PWM_PERIOD = 255：模擬 8-bit PWM（0~255）解析度。
- PWM_DELAY：每個 PWM 比較點之間延遲多少時間（影響 PWM 頻率)。
- COLOR_HOLD_MS = 1000：每個顏色停留 1秒。



```
// RGB LED pin bit 定義 (假設 BGR 分別對應 bit2/bit1/bit0)
#define R_MASK 0x1
#define G_MASK 0x2
#define B_MASK 0x4
```

- 由X_MASK決定R、G、B為high的時間

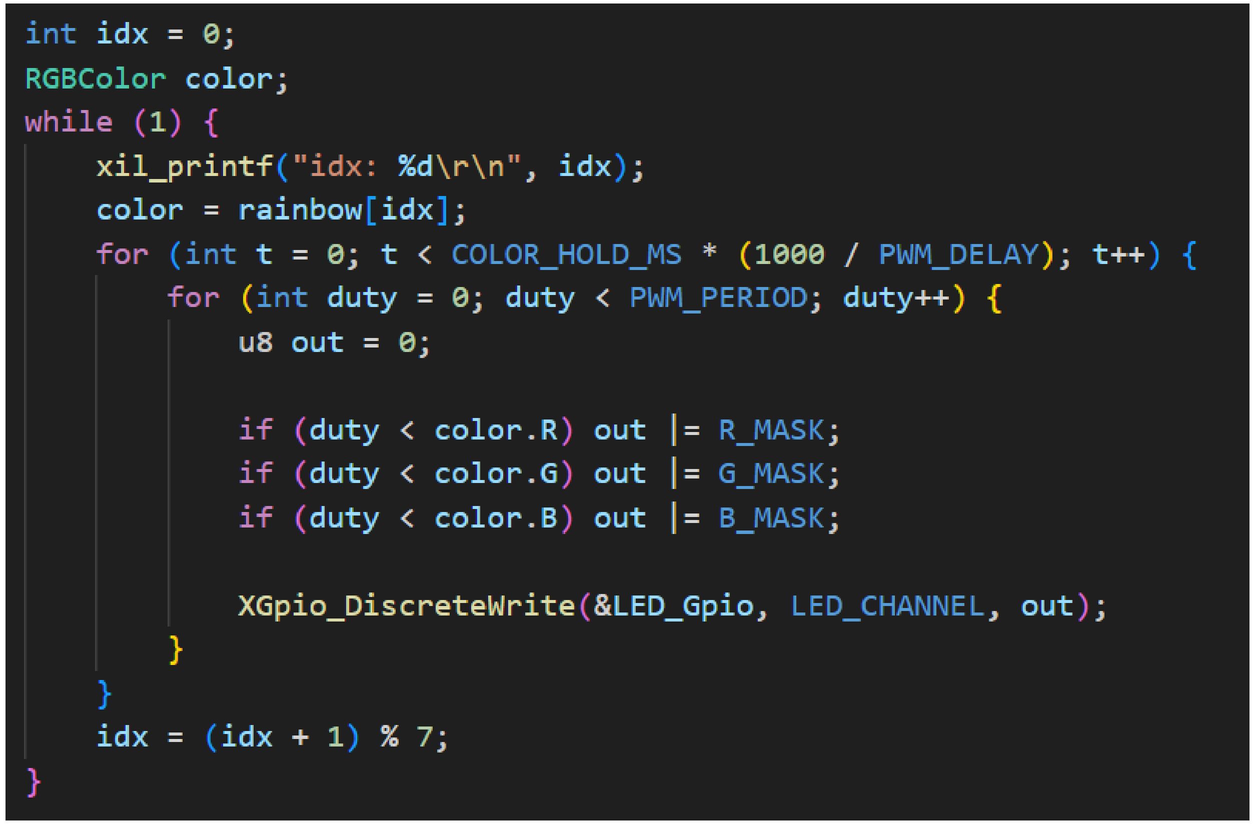
變數定義



```
RGBColor rainbow[7] = {
    {0x3C, 0x14, 0xDC}, // 紅🟥 #dc143c
    {0x00, 0x45, 0xFF}, // 橙🟧 #ff4500
    {0x00, 0xD7, 0xFF}, // 黃🟨 #ffd700
    {0x7F, 0xFF, 0x00}, // 綠🟩 #00ff7f
    {0xFF, 0x90, 0x1E}, // 藍🟦 #1e90ff
    {0xCD, 0x00, 0x00}, // 靛🟦 #0000cd
    {0xD3, 0x00, 0x94}  // 紫🟪 #9400d3
};
```

七種顏色(紅、橙、黃、綠、藍、靛、紫)的定義。

主迴圈: 顯示彩虹七色



```
int idx = 0;
RGBColor color;
while (1) {
    xil_printf("idx: %d\r\n", idx);
    color = rainbow[idx];
    for (int t = 0; t < COLOR_HOLD_MS * (1000 / PWM_DELAY); t++) {
        for (int duty = 0; duty < PWM_PERIOD; duty++) {
            u8 out = 0;

            if (duty < color.R) out |= R_MASK;
            if (duty < color.G) out |= G_MASK;
            if (duty < color.B) out |= B_MASK;

            XGpio_DiscreteWrite(&LED_Gpio, LED_CHANNEL, out);
        }
    }
    idx = (idx + 1) % 7;
}
```

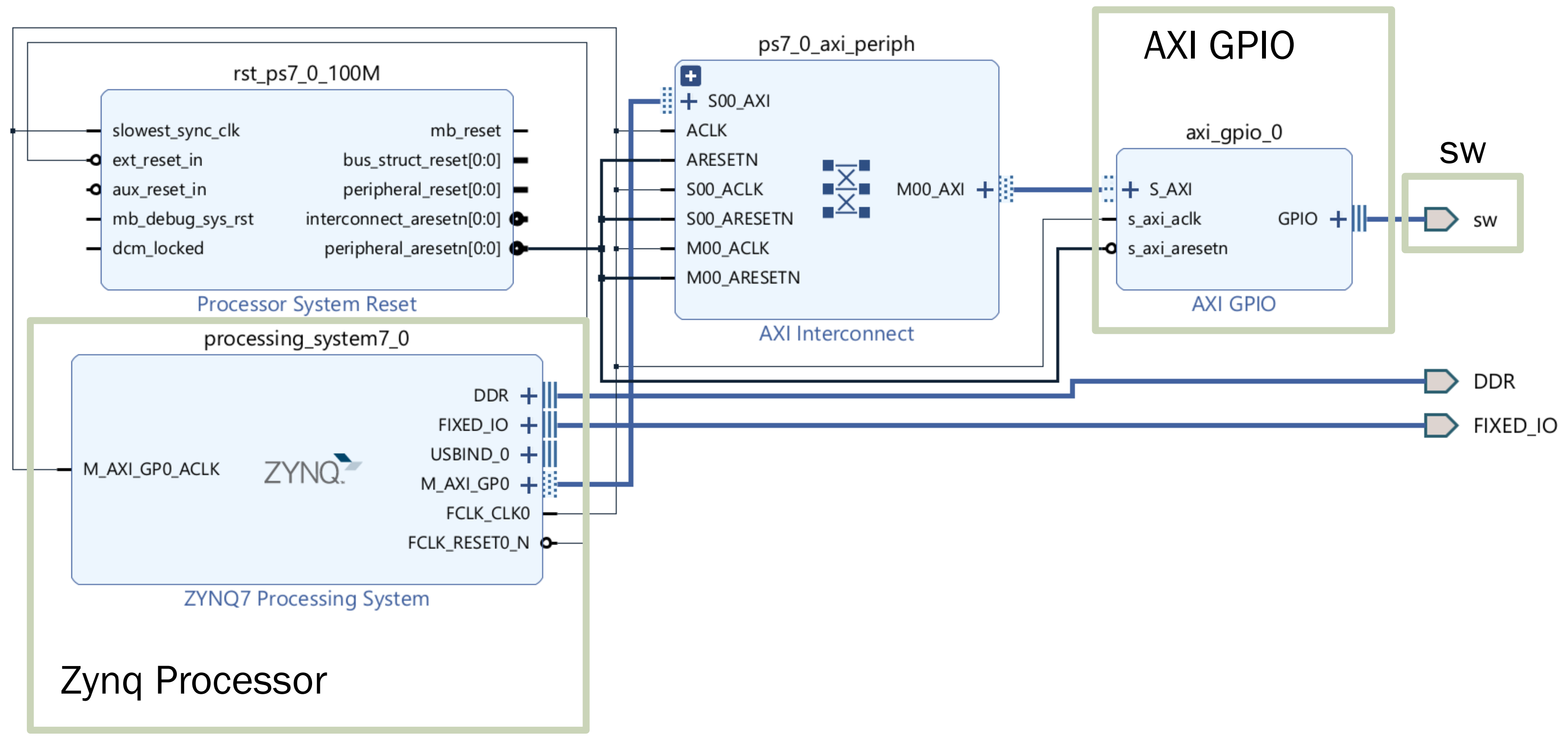
- 目前主要顏色: color = rainbow[idx]
- For loop: 創造不同顏色RGB的波型(out)
- 持續時間約1秒
- 最後換下一個顏色(mod 7)

Youtube 連結

- Problem1

Problem 2 – Sorting

Block Diagram (Our Design)



排序演算法介紹 - Merge sort

Merge sort

Merge Sort是一種經典的Divide and Conquer演算法。它的核心想法是：
「把大問題（長陣列）分成小問題（短陣列），各自排好後再合併成完整的有序陣列。」

演算法流程:

1. **分割** (Divide)：
   - 把陣列從中間一分為二，分成左右兩半。
2. **遞迴排序** (Recursion)：
   - 分別對左半部、右半部重複進行Merge Sort。
3. **合併** (Merge)：
   - 將兩個已經排序好的子陣列「合併」成一個排序好的大陣列。

舉例

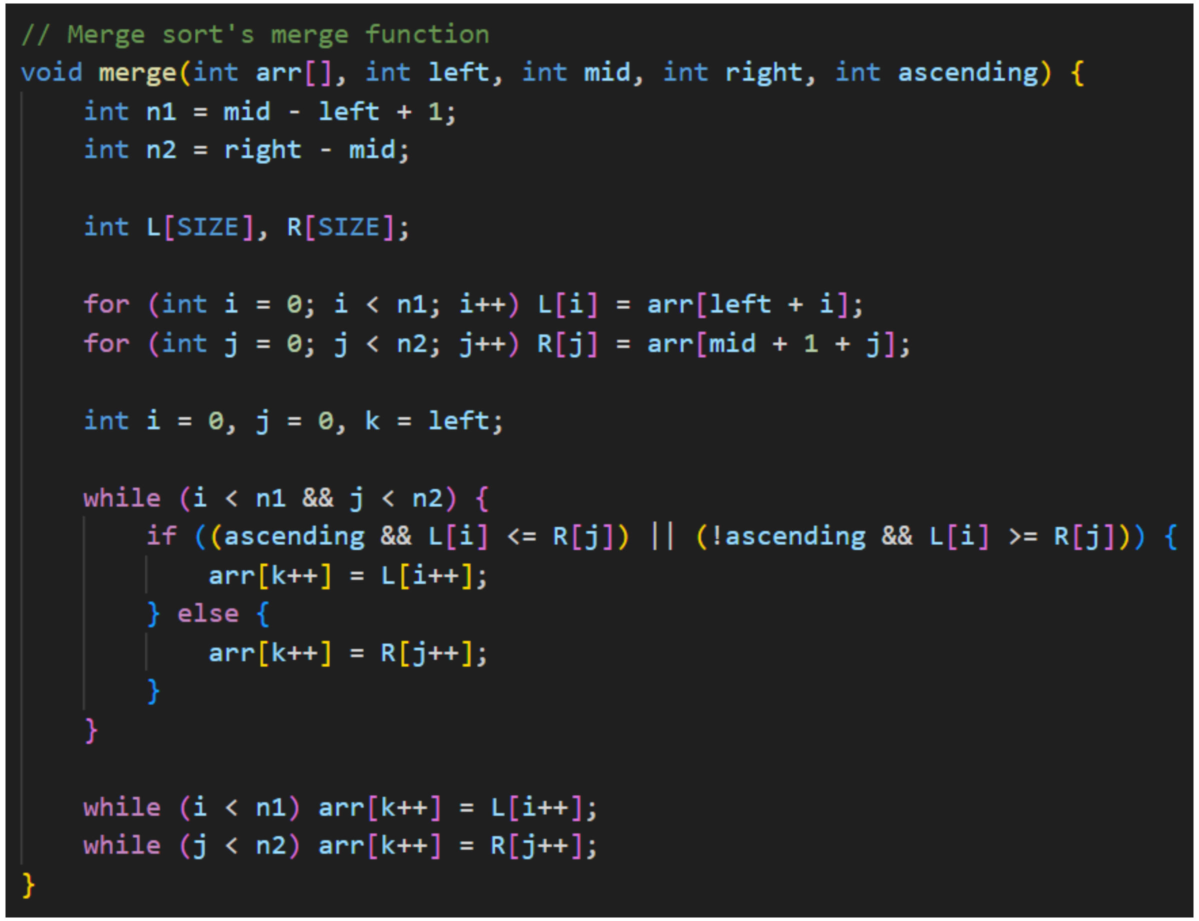
假設要排序：[8, 3, 5, 4, 7, 6, 1, 2]

分割過程:
[8,3,5,4] [7,6,1,2]
[8,3] [5,4] [7,6] [1,2]
[8] [3] [5] [4] [7] [6] [1] [2]

每兩個排好後合併：
[3,8] [4,5] [6,7] [1,2]
[3,4,5,8] [1,2,6,7]
[1,2,3,4,5,6,7,8]

Software 設計

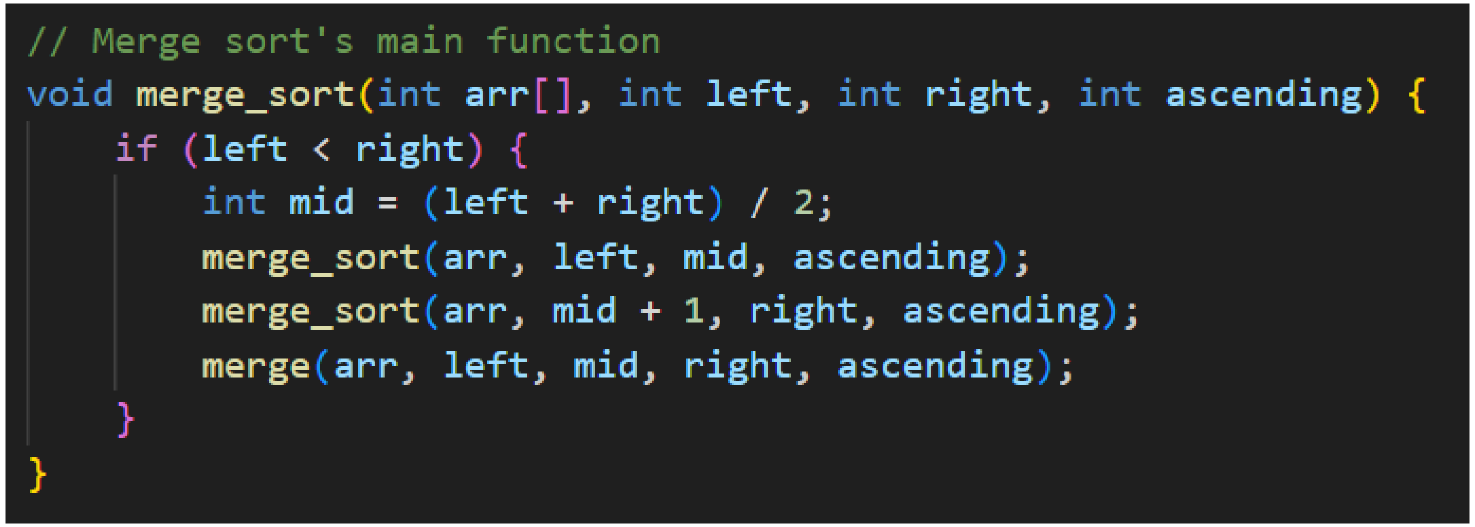
Algorithm – merge sort



```c
// Merge sort's merge function
void merge(int arr[], int left, int mid, int right, int ascending) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[SIZE], R[SIZE];

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if ((ascending && L[i] <= R[j]) || (!ascending && L[i] >= R[j])) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

- 把兩個子陣列[left..mid]和[mid+1..right]合併成一個排序好的陣列。
- ascending參數是決定要由大到小排列還是由小到大排列。
- 用兩個臨時陣列L、R來幫忙合併。

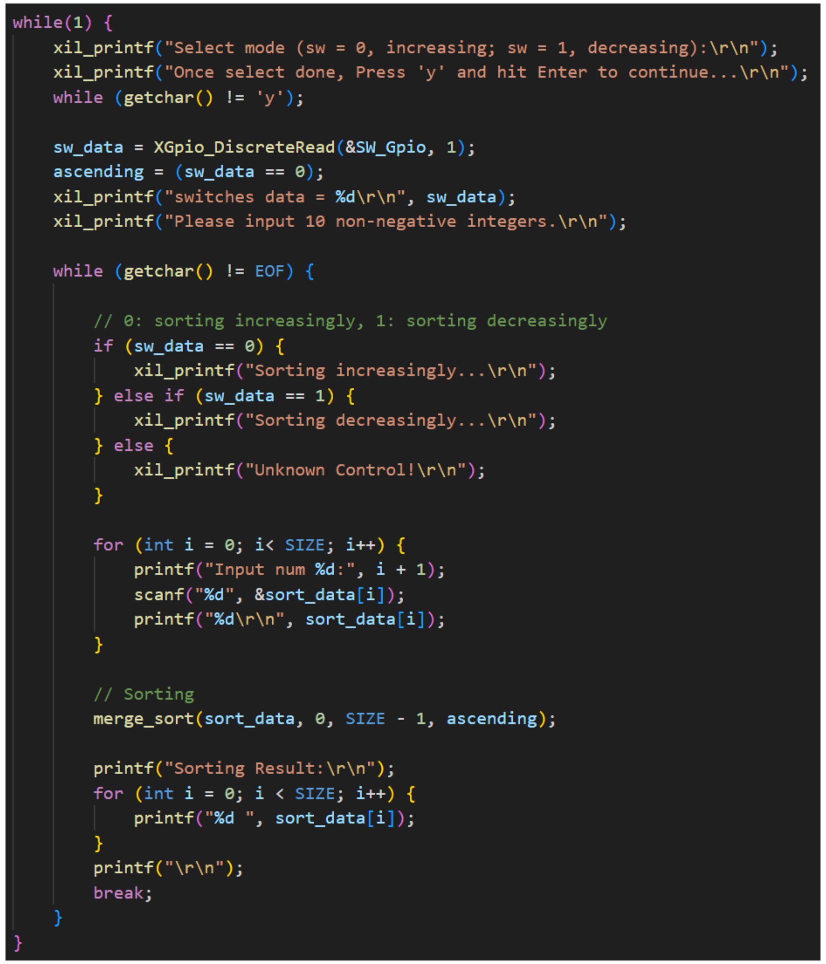
Algorithm – merge sort



```
// Merge sort's main function
void merge_sort(int arr[], int left, int right, int ascending) {
    if (left < right) {
        int mid = (left + right) / 2;
        merge_sort(arr, left, mid, ascending);
        merge_sort(arr, mid + 1, right, ascending);
        merge(arr, left, mid, right, ascending);
    }
}
```

- 用Divide-and-Conquer的方法
- 把陣列分成兩半，各自排序，最後merge起來。
- 同樣用ascending參數判斷如何排序。

主函式

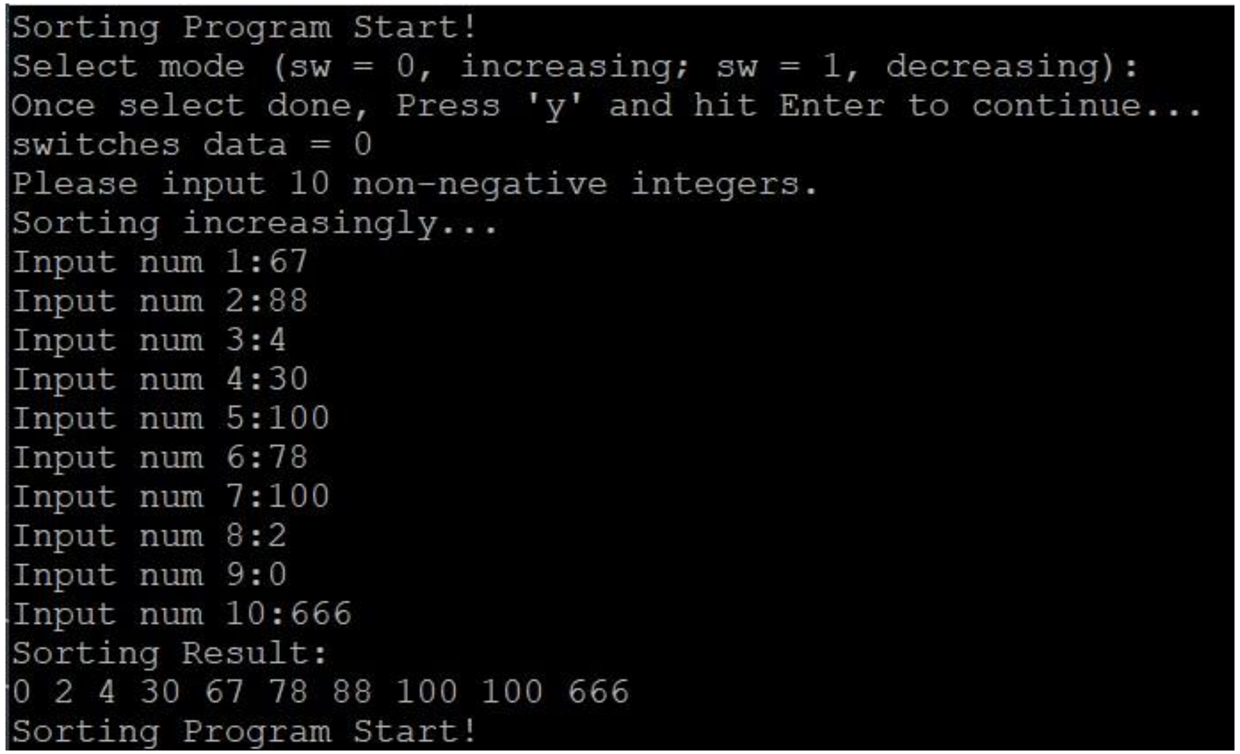


```c
while(1) {
    xil_printf("Select mode (sw = 0, increasing; sw = 1, decreasing):\r\n");
    xil_printf("Once select done, Press 'y' and hit Enter to continue...\r\n");
    while (getchar() != 'y');

    sw_data = XGpio_DiscreteRead(&SW_Gpio, 1);
    ascending = (sw_data == 0);
    xil_printf("switches data = %d\r\n", sw_data);
    xil_printf("Please input 10 non-negative integers.\r\n");

    while (getchar() != EOF) {

        // 0: sorting increasingly, 1: sorting decreasingly
        if (sw_data == 0) {
            xil_printf("Sorting increasingly...\r\n");
        } else if (sw_data == 1) {
            xil_printf("Sorting decreasingly...\r\n");
        } else {
            xil_printf("Unknown Control!\r\n");
        }

        for (int i = 0; i< SIZE; i++) {
            printf("Input num %d:", i + 1);
            scanf("%d", &sort_data[i]);
            printf("%d\r\n", sort_data[i]);
        }

        // Sorting
        merge_sort(sort_data, 0, SIZE - 1, ascending);

        printf("Sorting Result:\r\n");
        for (int i = 0; i < SIZE; i++) {
            printf("%d ", sort_data[i]);
        }
        printf("\r\n");
        break;
    }
}
```

- 先調整switch，確定排序模式，再按’y’和Enter確定。
- 接下來出入10個非負整數
- 輸入完後，開始merge sort
- 最後將結果印在終端機上。

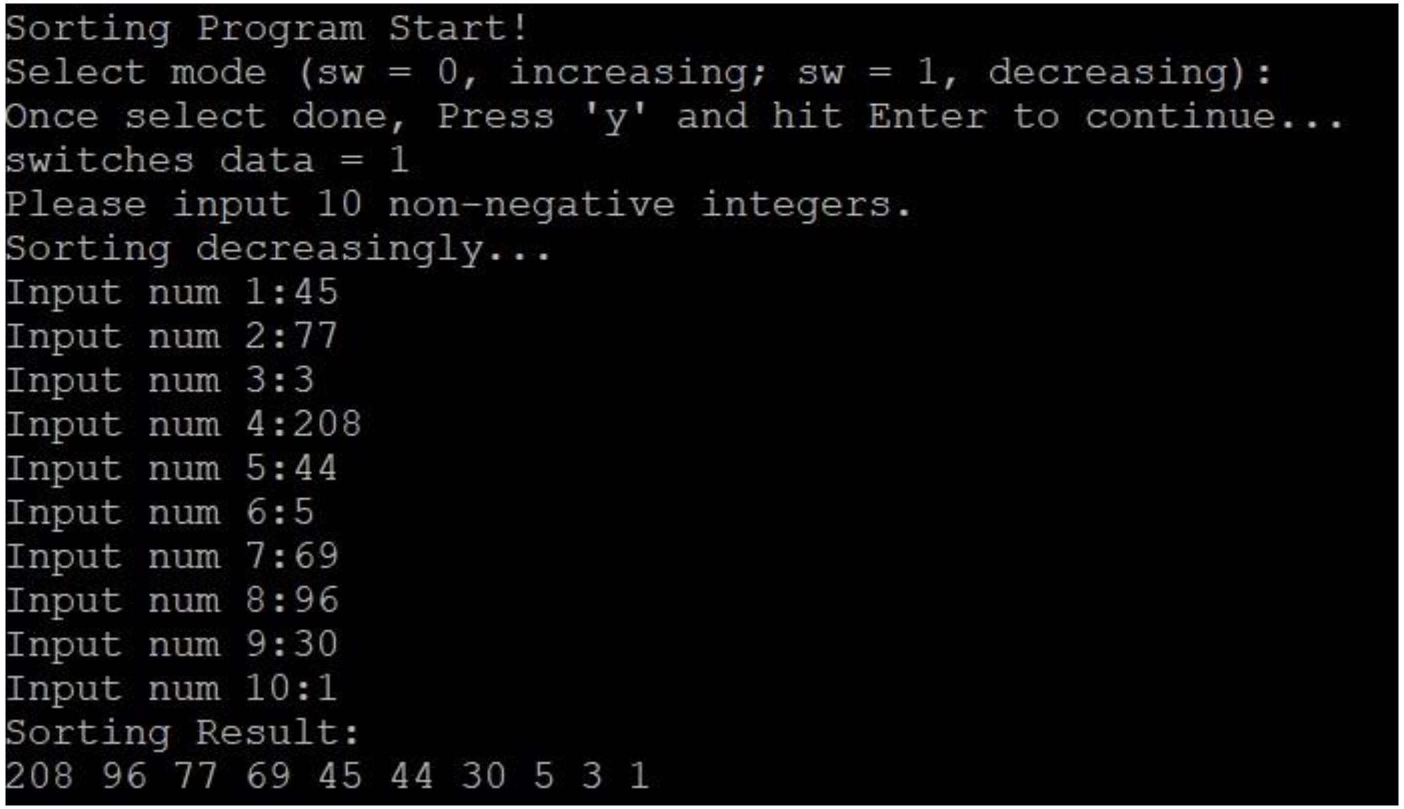
(重複以上步驟)

結果 – 由小到大排序



```
Sorting Program Start!
Select mode (sw = 0, increasing; sw = 1, decreasing):
Once select done, Press 'y' and hit Enter to continue...
switches data = 0
Please input 10 non-negative integers.
Sorting increasingly...
Input num 1:67
Input num 2:88
Input num 3:4
Input num 4:30
Input num 5:100
Input num 6:78
Input num 7:100
Input num 8:2
Input num 9:0
Input num 10:666
Sorting Result:
0 2 4 30 67 78 88 100 100 666
Sorting Program Start!
```

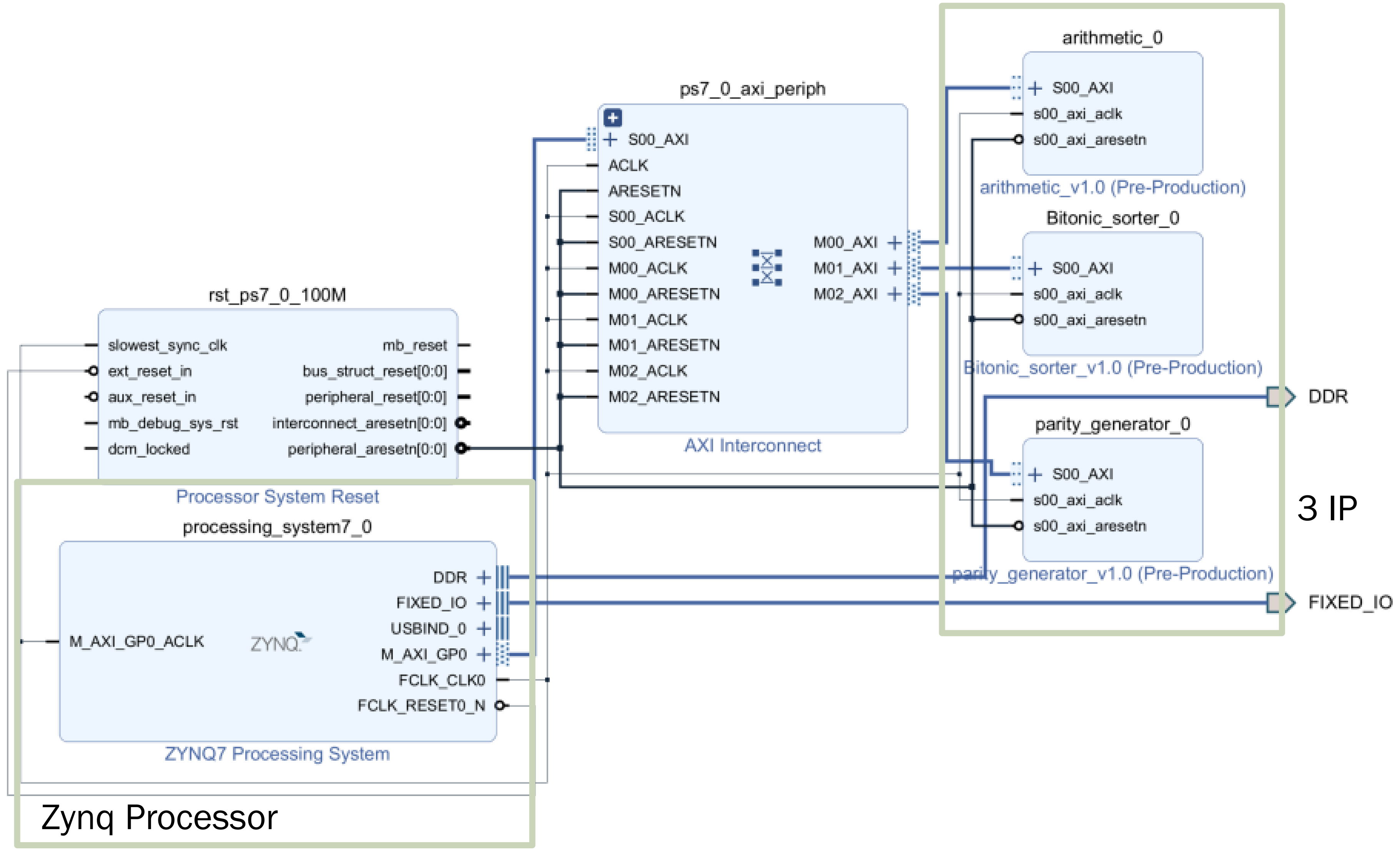
結果 – 由大到小排序



```
Sorting Program Start!
Select mode (sw = 0, increasing; sw = 1, decreasing):
Once select done, Press 'y' and hit Enter to continue...
switches data = 1
Please input 10 non-negative integers.
Sorting decreasingly...
Input num 1:45
Input num 2:77
Input num 3:3
Input num 4:208
Input num 5:44
Input num 6:5
Input num 7:69
Input num 8:96
Input num 9:30
Input num 10:1
Sorting Result:
208 96 77 69 45 44 30 5 3 1
```

Problem 3

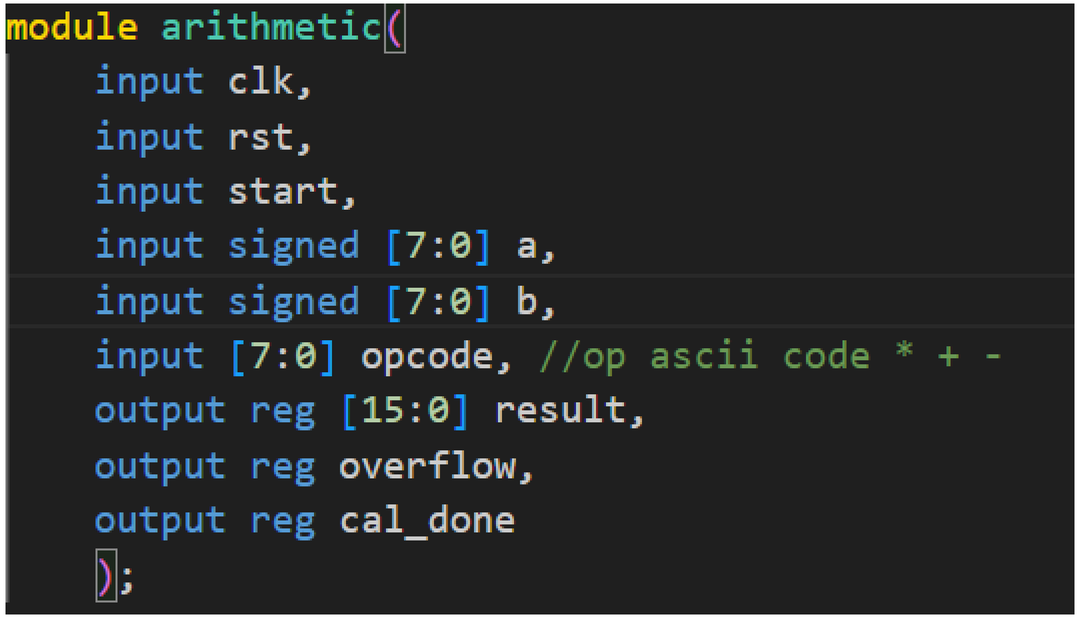
Block Diagram (Our Design)



Hardware 設計

3-1 Arithmetic 設計

3-1 Arithmetic



```
module arithmetic(
    input clk,
    input rst,
    input start,
    input signed [7:0] a,
    input signed [7:0] b,
    input [7:0] opcode, //op ascii code * + -
    output reg [15:0] result,
    output reg overflow,
    output reg cal_done
    );
```

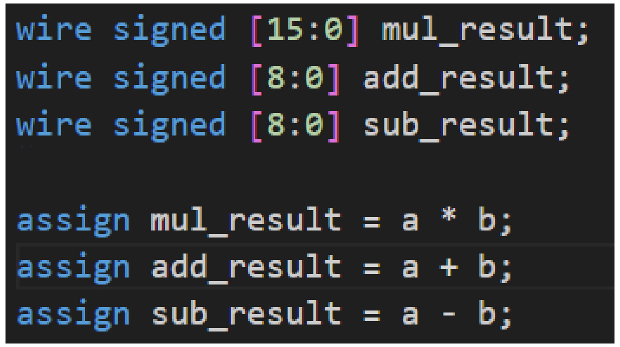
使用 start 以及 cal_done 訊號確保輸出結果正確

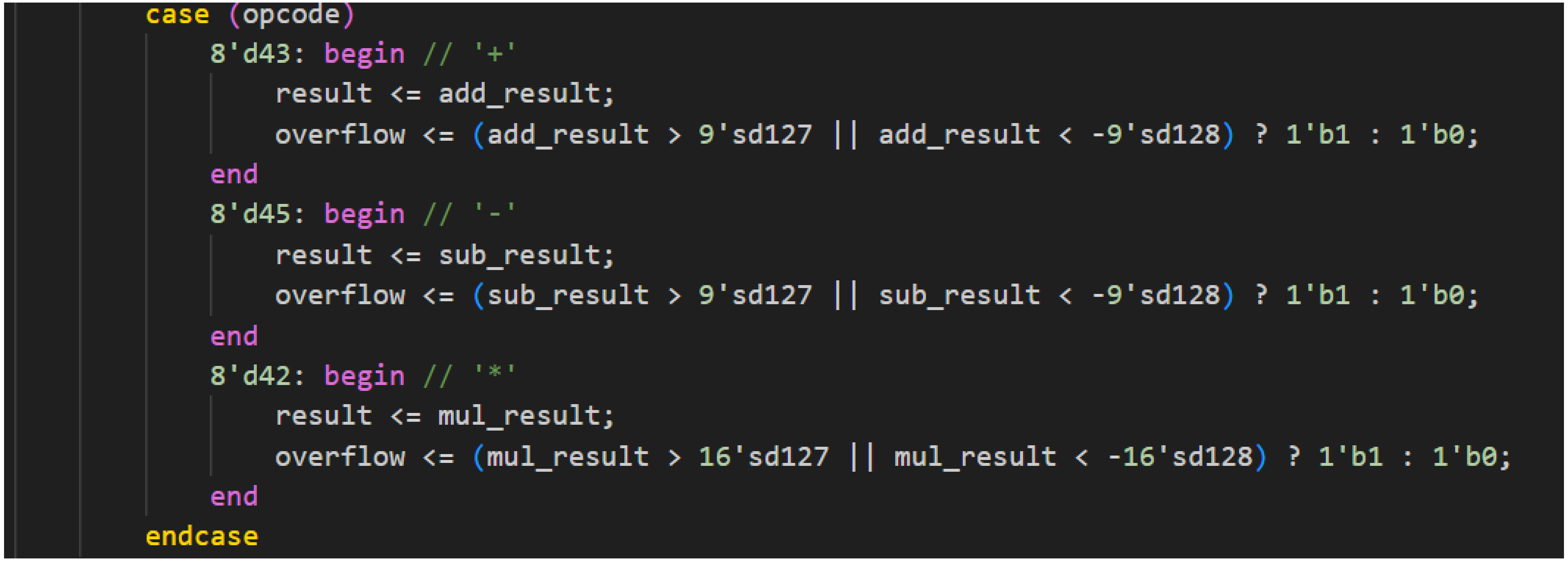
a b 為兩個輸入的 8 bits 有號數運算元

opcode 表示要進行的運算

輸出為 16 bits 有號數 result 以及 overflow 訊號檢查計算結過是否發生overflow

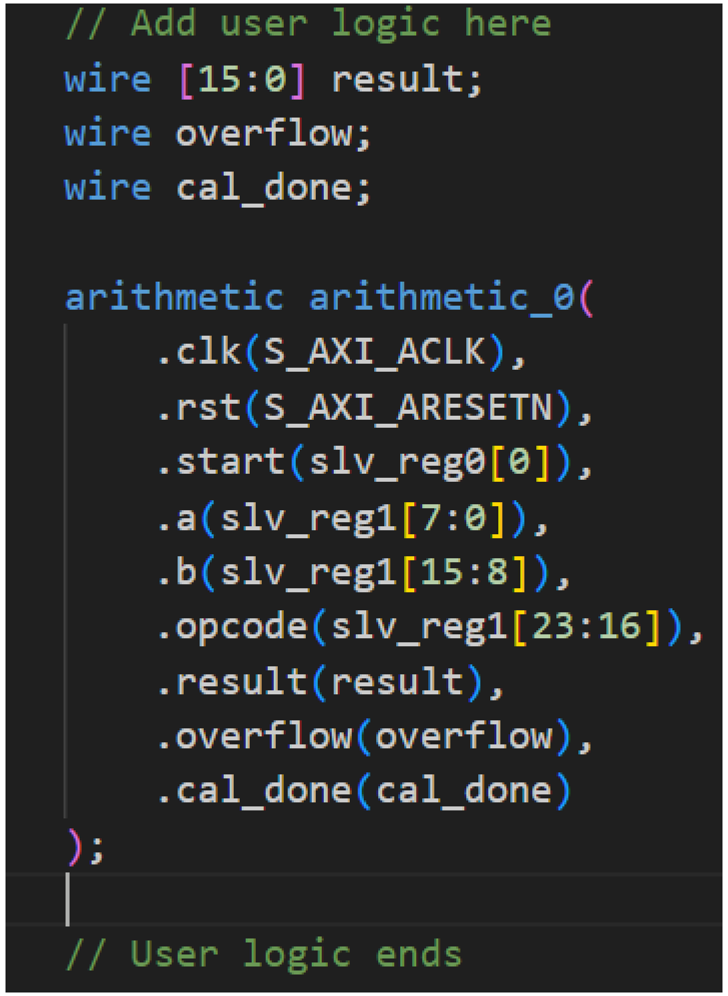
3-1 Arithmetic


```
wire signed [15:0] mul_result;
wire signed [8:0] add_result;
wire signed [8:0] sub_result;

assign mul_result = a * b;
assign add_result = a + b;
assign sub_result = a - b;
```


```
case (opcode)
    8'd43: begin // '+'
        result <= add_result;
        overflow <= (add_result > 9'sd127 || add_result < -9'sd128) ? 1'b1 : 1'b0;
    end
    8'd45: begin // '-'
        result <= sub_result;
        overflow <= (sub_result > 9'sd127 || sub_result < -9'sd128) ? 1'b1 : 1'b0;
    end
    8'd42: begin // '*'
        result <= mul_result;
        overflow <= (mul_result > 16'sd127 || mul_result < -16'sd128) ? 1'b1 : 1'b0;
    end
endcase
```

根據輸入 opcode 決定輸出 result 及檢查是否發生 overflow

3-1 Arithmetic - 連接AXI IP

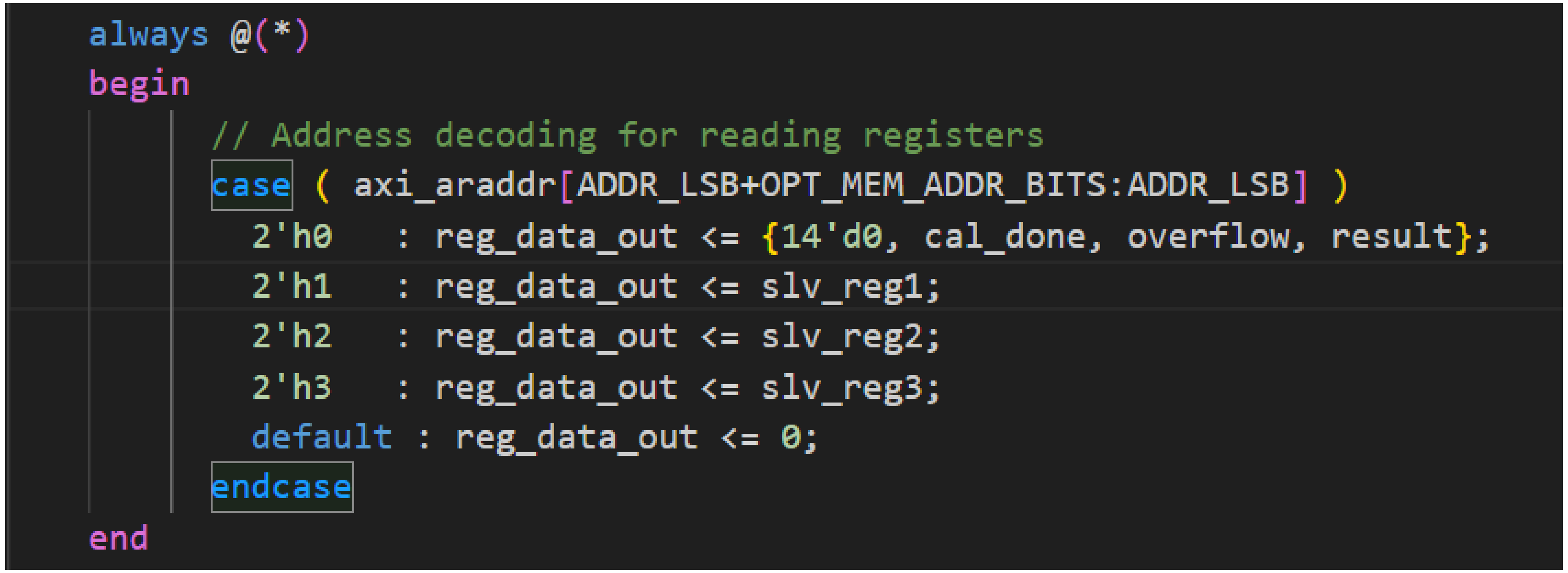


```
// Add user logic here
wire [15:0] result;
wire overflow;
wire cal_done;

arithmetic arithmetic_0(
    .clk(S_AXI_ACLK),
    .rst(S_AXI_ARESETN),
    .start(slv_reg0[0]),
    .a(slv_reg1[7:0]),
    .b(slv_reg1[15:8]),
    .opcode(slv_reg1[23:16]),
    .result(result),
    .overflow(overflow),
    .cal_done(cal_done)
);

// User logic ends
```

使用 slv_reg0[0] 作為計算start訊號

slv_reg1[7:0] 作為運算元 a
slv_reg1[15:8] 作為運算元 b
slv_reg1[23:16]作為 opcode

3-1 Arithmetic - 連接AXI IP



```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0   : reg_data_out <= {14'd0, cal_done, overflow, result};
        2'h1   : reg_data_out <= slv_reg1;
        2'h2   : reg_data_out <= slv_reg2;
        2'h3   : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
      endcase
end
```

輸出資料回到PS端

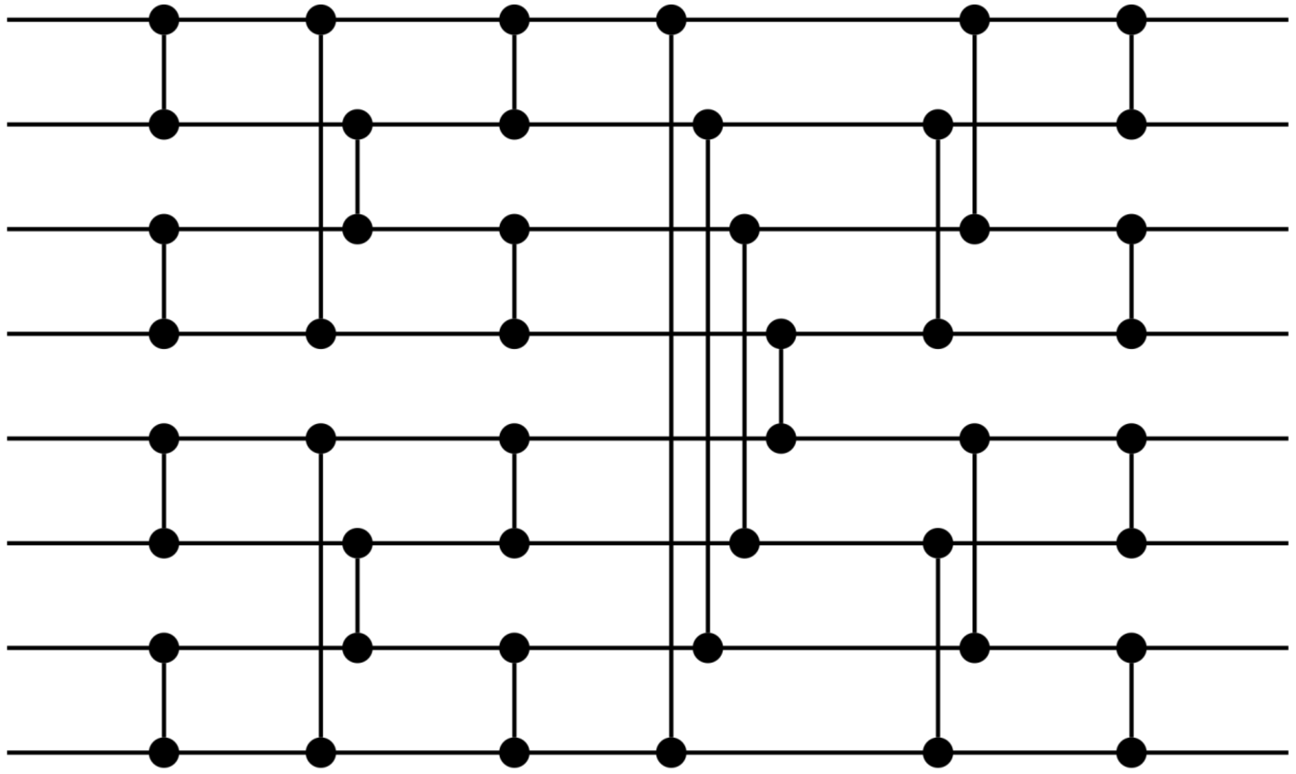
3-2 Sorting設計

3-2 Sorting

使用演算法:Bitonic Sort

數字位元數:4bits

數列長度:固定為8筆



3-2 Sorting - Bitonic Sort 介紹

一、定義

Bitonic Sort 是一種基於比較的排序演算法，最早由 K. Batcher 於 1968 年提出。其設計目的在於支援 **並行處理（Parallel Processing）**，尤其適用於硬體電路或具有高度平行度的運算平台。Bitonic Sort 的核心運作方式為：透過將資料轉換為 **Bitonic 序列（Bitonic Sequence）**，再進行合併排序（Bitonic Merge），以達到全序排列。

二、演算法特性

•**時間複雜度**：為 O(log^{2}n)，雖然在一般序列長度下並不優於快速排序，但在支援並行處理的環境中可大幅提升效能。

•**演算法類型**：屬於比較排序（Comparison-based Sort）與排序網路（Sorting Network）的一種。

•**資料長度要求**：通常需為 2k2^k2k（k 為正整數），以利遞迴劃分與合併操作。

3-2 Sorting - Bitonic Sort 介紹

三、硬體設計上的優勢
Bitonic Sort 的演算法特性，使其非常適合在硬體層面上實作，原因包括：

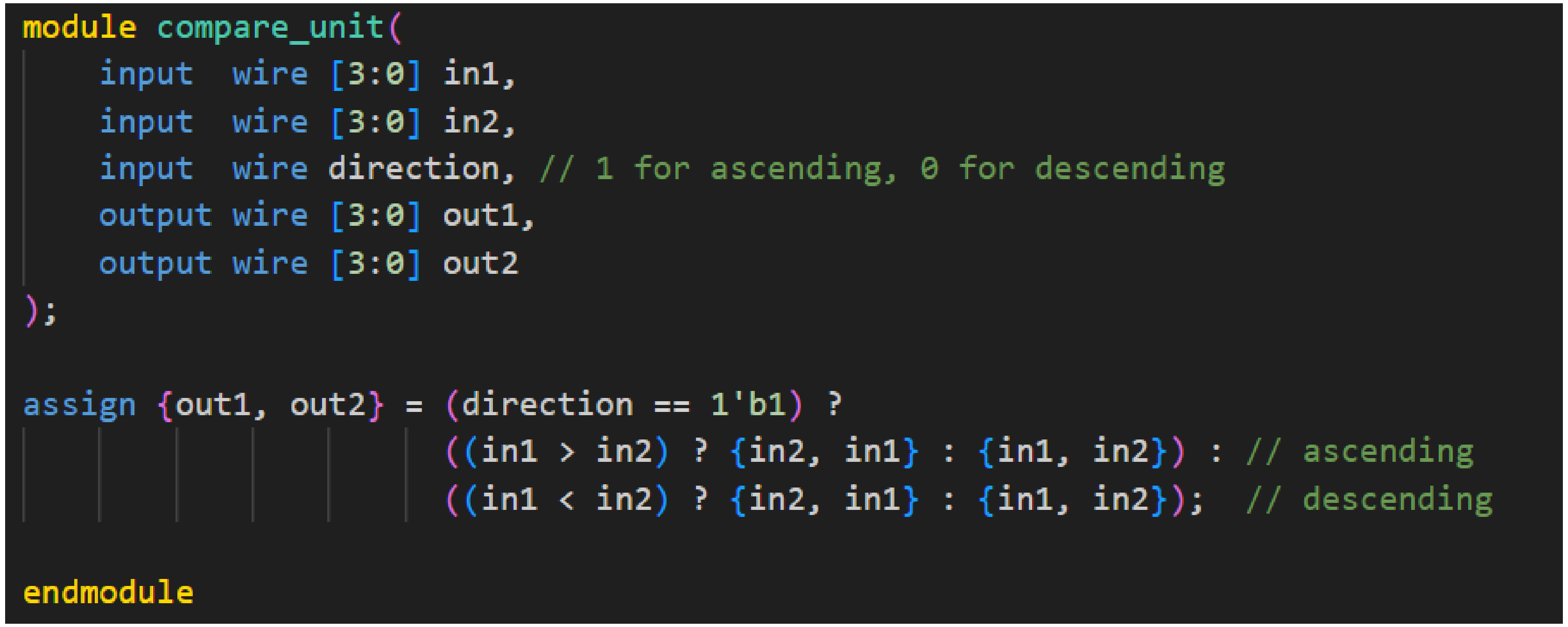
1.固定架構與控制流程
Bitonic Sort 不依賴資料內容改變流程，所有比較與交換操作皆可預先規劃。這使其可以設計成具有高度規律性與可重複性的 **硬體排序網路（Sorting Network）**。

2.高度並行性
多個比較與交換操作可於同一個時鐘週期（clock cycle）中同時執行，特別適合在 GPU、FPGA、ASIC 等平台上實現。這種並行能力可使排序速度隨硬體資源擴展而大幅提升。

3.硬體資源可預測性
所需的比較器數量與連線拓樸皆可在設計初期確定，便於進行資源配置與延遲預估。這點對硬體電路設計（如 RTL 設計、VLSI 規劃）特別重要。

4.低延遲特性
在硬體上使用 Bitonic Sorting Network，可實現 pipeline 化操作，在高頻運作下具備低延遲與高吞吐量的優勢，廣泛應用於網路封包排序、資料流處理等領域。

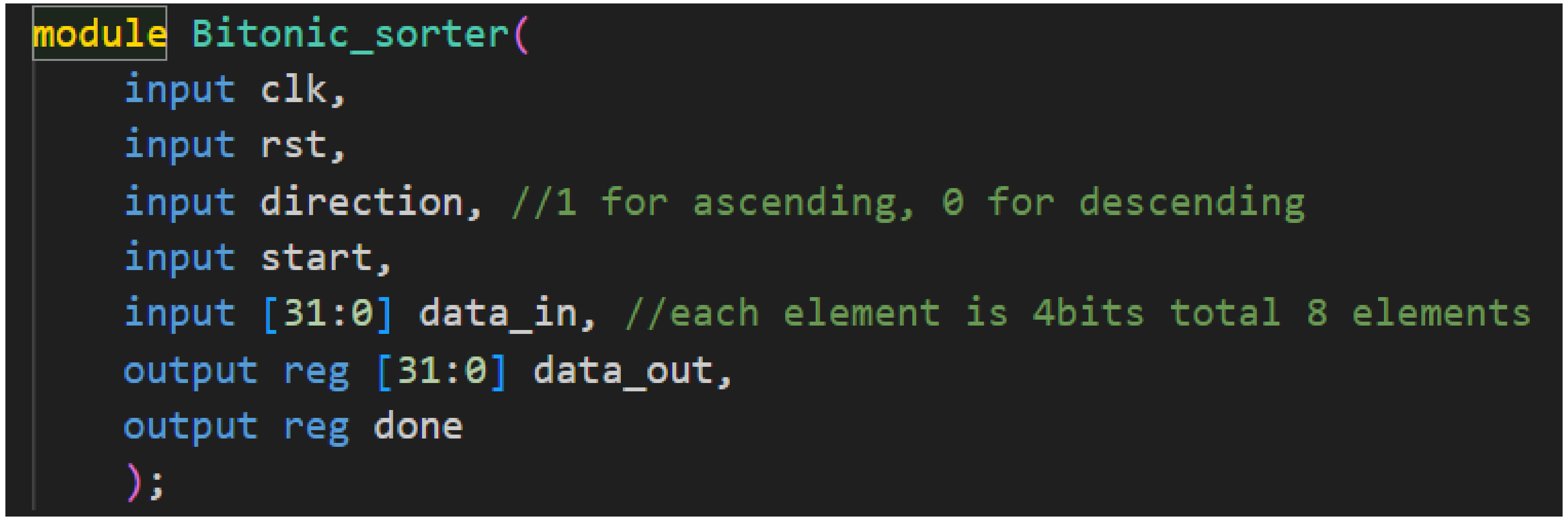
3-2 Sorting – Compare Unit



```
module compare_unit(
    input  wire [3:0] in1,
    input  wire [3:0] in2,
    input  wire direction, // 1 for ascending, 0 for descending
    output wire [3:0] out1,
    output wire [3:0] out2
);

assign {out1, out2} = (direction == 1'b1) ?
                      ((in1 > in2) ? {in2, in1} : {in1, in2}) : // ascending
                      ((in1 < in2) ? {in2, in1} : {in1, in2});  // descending

endmodule
```

先設計兩個 4bits inputs 的比較單元

3-2 Sorting



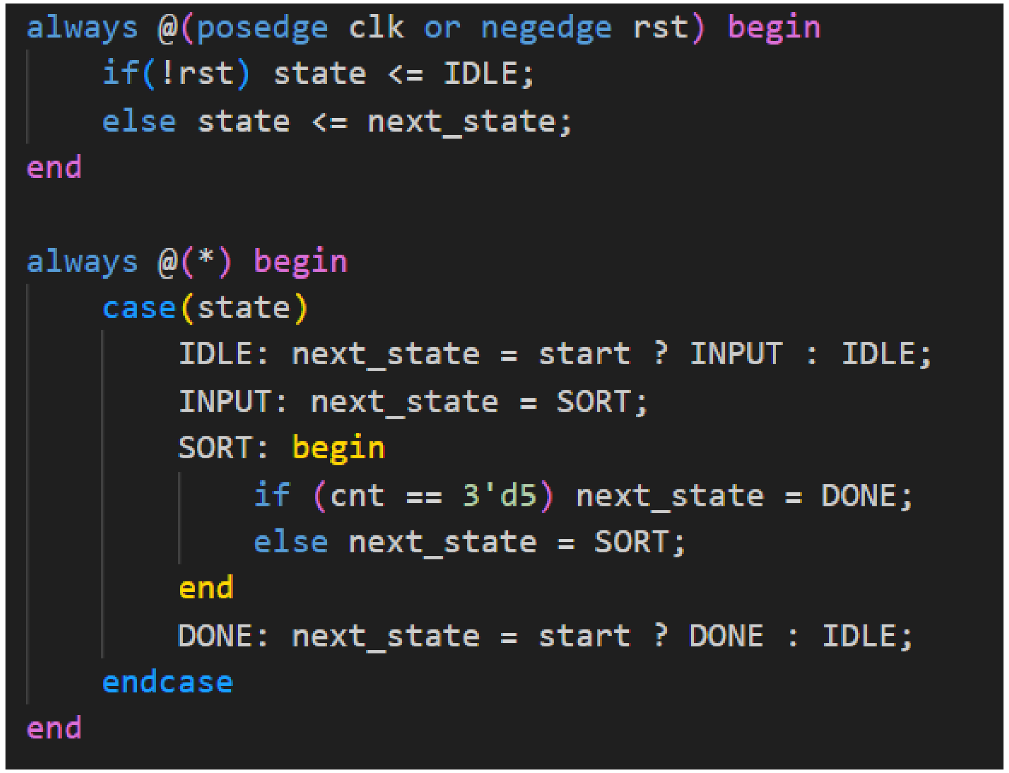
```
module Bitonic_sorter(
    input clk,
    input rst,
    input direction, //1 for ascending, 0 for descending
    input start,
    input [31:0] data_in, //each element is 4bits total 8 elements
    output reg [31:0] data_out,
    output reg done
    );
```

8 個 4bits 輸入透過 32bits 的data_in輸入

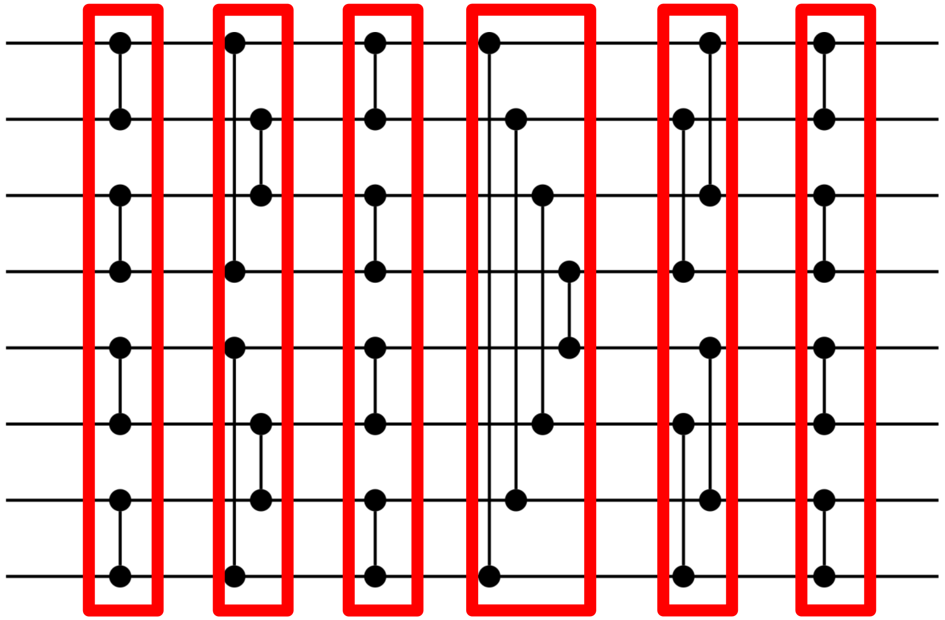
我們的設計可以選擇排序遞增或遞減

data_out 輸出排序好的序列

3-2 Sorting



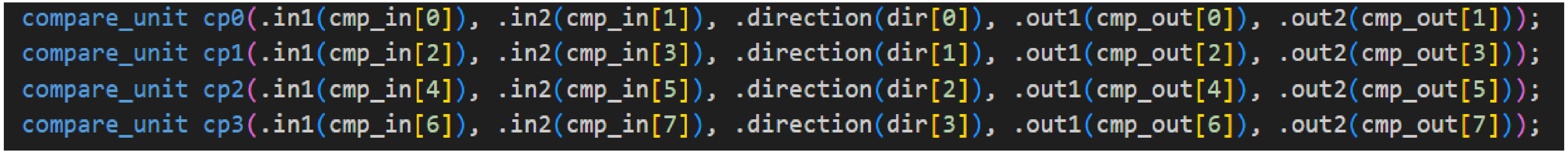
```
always @(posedge clk or negedge rst) begin
    if(!rst) state <= IDLE;
    else state <= next_state;
end

always @(*) begin
    case(state)
        IDLE: next_state = start ? INPUT : IDLE;
        INPUT: next_state = SORT;
        SORT: begin
            if (cnt == 3'd5) next_state = DONE;
            else next_state = SORT;
        end
        DONE: next_state = start ? DONE : IDLE;
    endcase
end
```



FSM 設計

將sortin網路拆分多步驟完成，因此
我們的設計僅需要四個比較單元

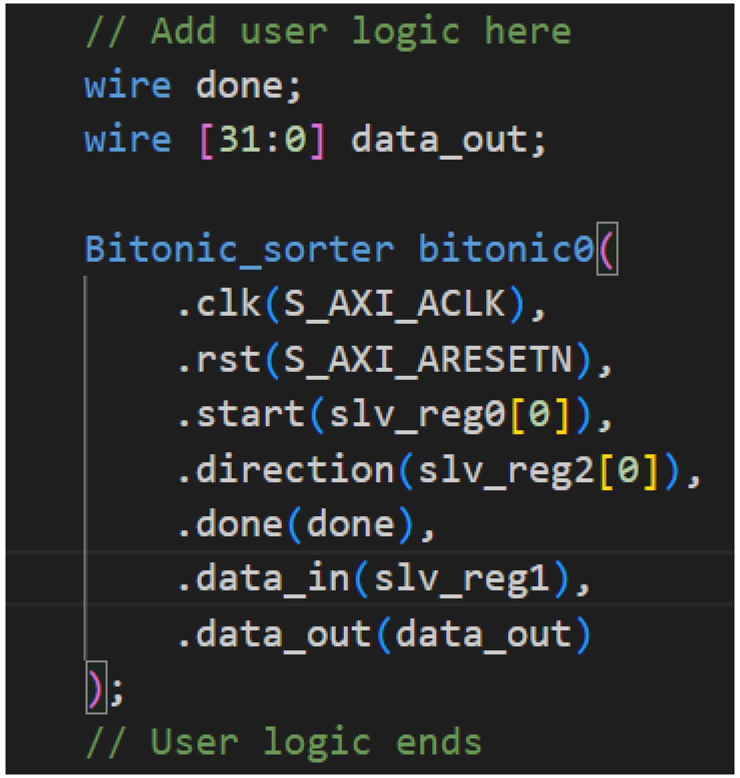
3-2 Sorting



```
compare_unit cp0(.in1(cmp_in[0]), .in2(cmp_in[1]), .direction(dir[0]), .out1(cmp_out[0]), .out2(cmp_out[1]));
compare_unit cp1(.in1(cmp_in[2]), .in2(cmp_in[3]), .direction(dir[1]), .out1(cmp_out[2]), .out2(cmp_out[3]));
compare_unit cp2(.in1(cmp_in[4]), .in2(cmp_in[5]), .direction(dir[2]), .out1(cmp_out[4]), .out2(cmp_out[5]));
compare_unit cp3(.in1(cmp_in[6]), .in2(cmp_in[7]), .direction(dir[3]), .out1(cmp_out[6]), .out2(cmp_out[7]));
```

只使用四個比較單元

3-2 Sorting - 連接AXI IP

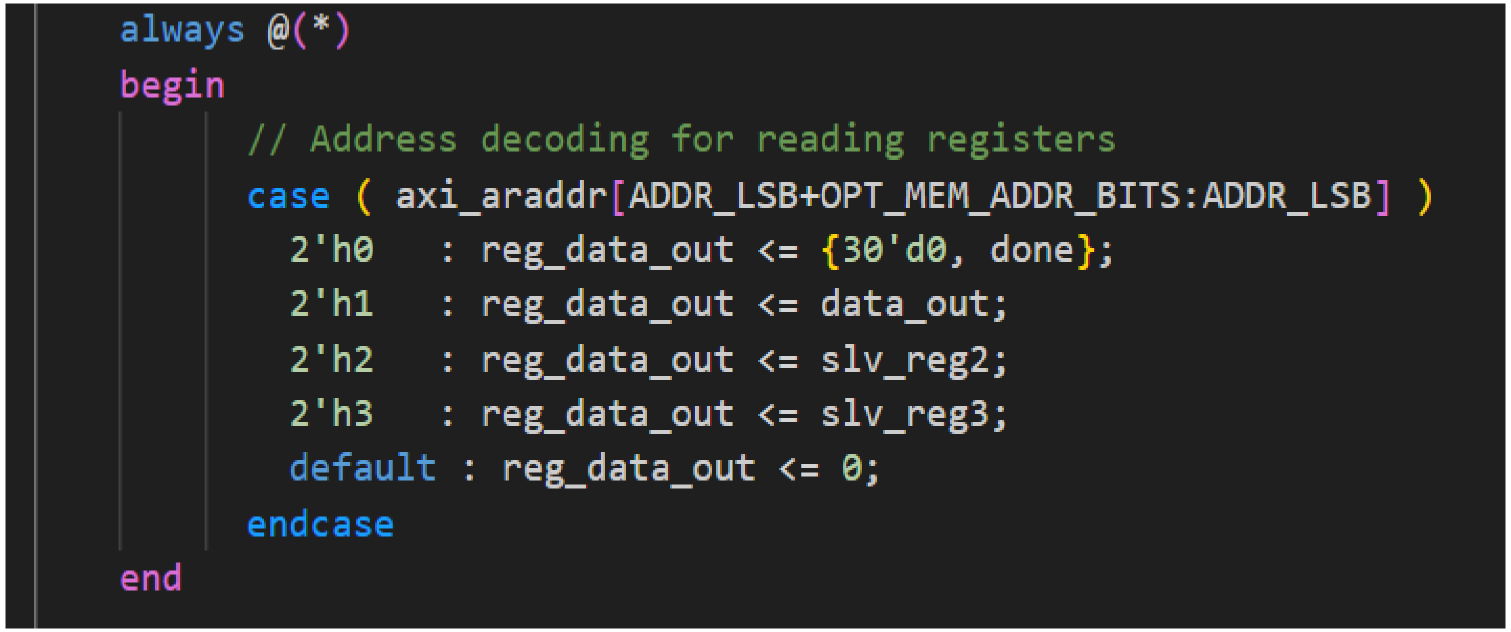


```
// Add user logic here
wire done;
wire [31:0] data_out;

Bitonic_sorter bitonic0(
    .clk(S_AXI_ACLK),
    .rst(S_AXI_ARESETN),
    .start(slv_reg0[0]),
    .direction(slv_reg2[0]),
    .done(done),
    .data_in(slv_reg1),
    .data_out(data_out)
);
// User logic ends
```

使用 slv_reg0[0] 作為計算start訊號
使用 slv_reg2[0] 決定了排序為遞增或遞減

slv_reg1 傳輸 8 個 4bits 資料(共32bits)
排序後的結果為 data_out

3-2 Sorting - 連接AXI IP

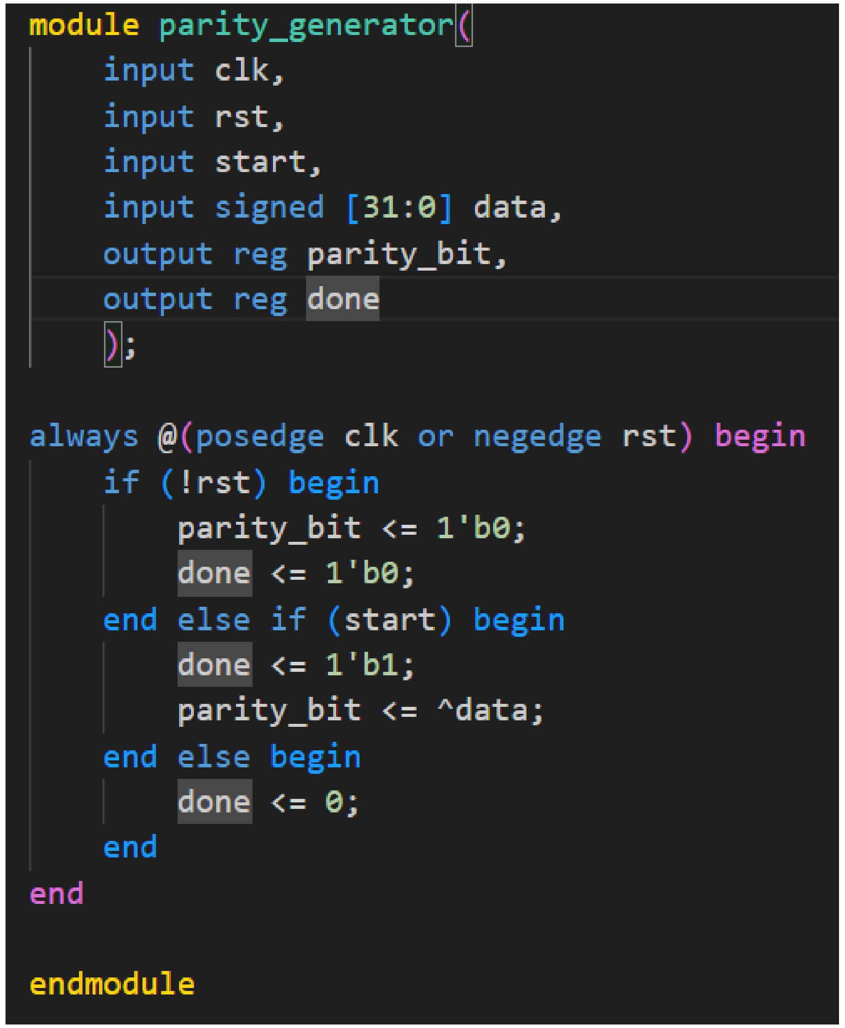


```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0   : reg_data_out <= {30'd0, done};
        2'h1   : reg_data_out <= data_out;
        2'h2   : reg_data_out <= slv_reg2;
        2'h3   : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
      endcase
end
```

輸出資料回到PS端

3-3 Parity Generator 設計

3-3 Parity Generator



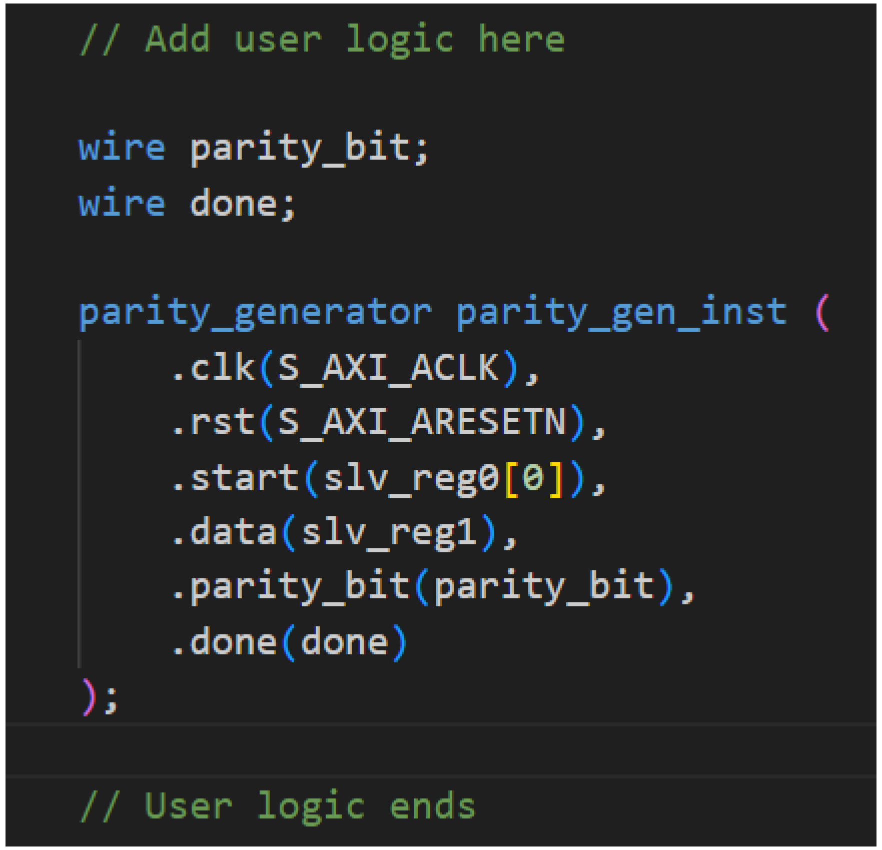
```
module parity_generator(
    input clk,
    input rst,
    input start,
    input signed [31:0] data,
    output reg parity_bit,
    output reg done
    );

always @(posedge clk or negedge rst) begin
    if (!rst) begin
        parity_bit <= 1'b0;
        done <= 1'b0;
    end else if (start) begin
        done <= 1'b1;
        parity_bit <= ^data;
    end else begin
        done <= 0;
    end
end

endmodule
```

同樣使用 start 以及 cal_done 訊號確保輸出結果正確

輸出 parity_bit <= ^data 表示

- 輸入data內含奇數個 1，回傳 1
- 輸入data內含偶數個 1，回傳 0

3-3 Parity Generator - 連接AXI IP

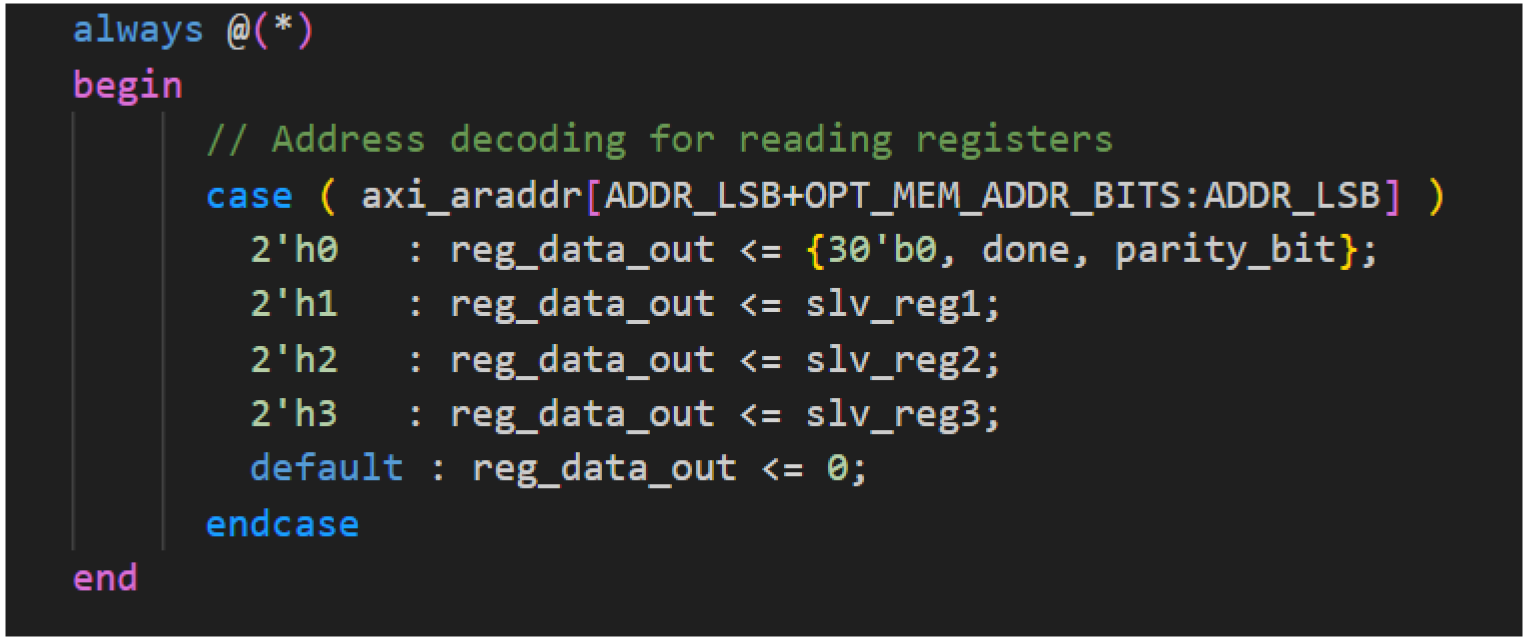


```
// Add user logic here

wire parity_bit;
wire done;

parity_generator parity_gen_inst (
    .clk(S_AXI_ACLK),
    .rst(S_AXI_ARESETN),
    .start(slv_reg0[0]),
    .data(slv_reg1),
    .parity_bit(parity_bit),
    .done(done)
);

// User logic ends
```

使用 slv_reg0[0] 作為計算start訊號

slv_reg1輸入 32-bit 資料

3-3 Parity Generator - 連接AXI IP

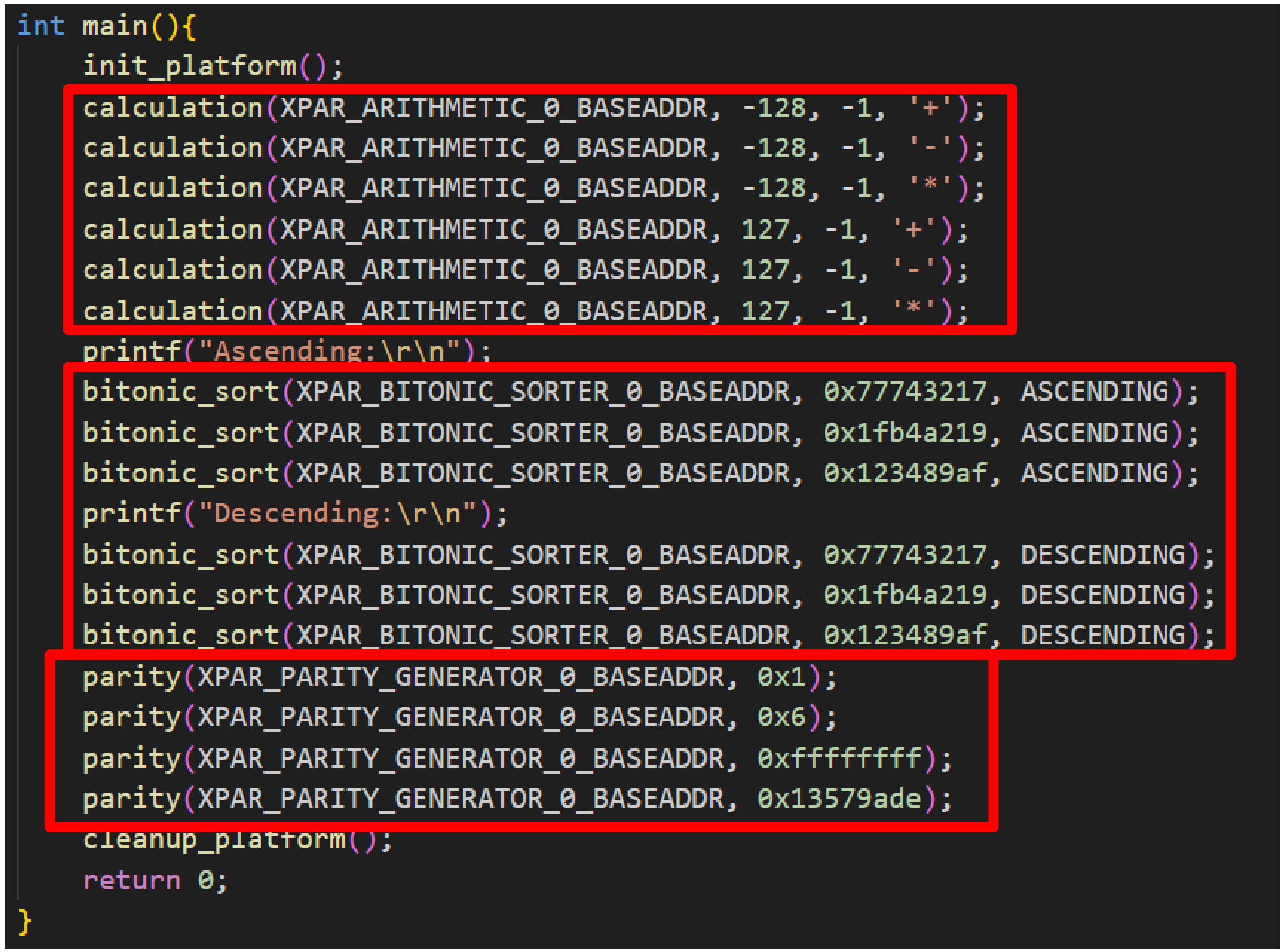


```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0   : reg_data_out <= {30'b0, done, parity_bit};
        2'h1   : reg_data_out <= slv_reg1;
        2'h2   : reg_data_out <= slv_reg2;
        2'h3   : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
      endcase
end
```

回傳 parity_bit

Software 設計

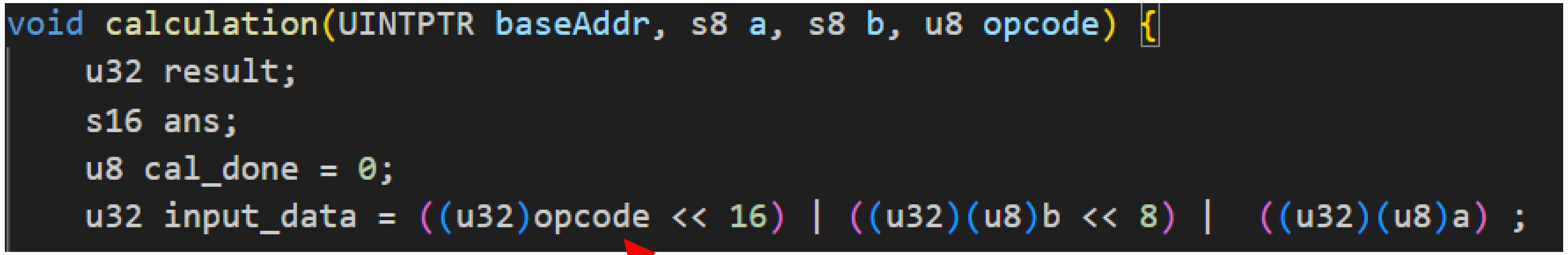
Main function



```c
int main(){
    init_platform();
    calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '+');
    calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '-');
    calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '*');
    calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '+');
    calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '-');
    calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '*');
    printf("Ascending:\r\n");
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x77743217, ASCENDING);
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x1fb4a219, ASCENDING);
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x123489af, ASCENDING);
    printf("Descending:\r\n");
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x77743217, DESCENDING);
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x1fb4a219, DESCENDING);
    bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x123489af, DESCENDING);
    parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x1);
    parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x6);
    parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0xffffffff);
    parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x13579ade);
    cleanup_platform();
    return 0;
}
```

把三種功能分別寫成function

Function - calculation



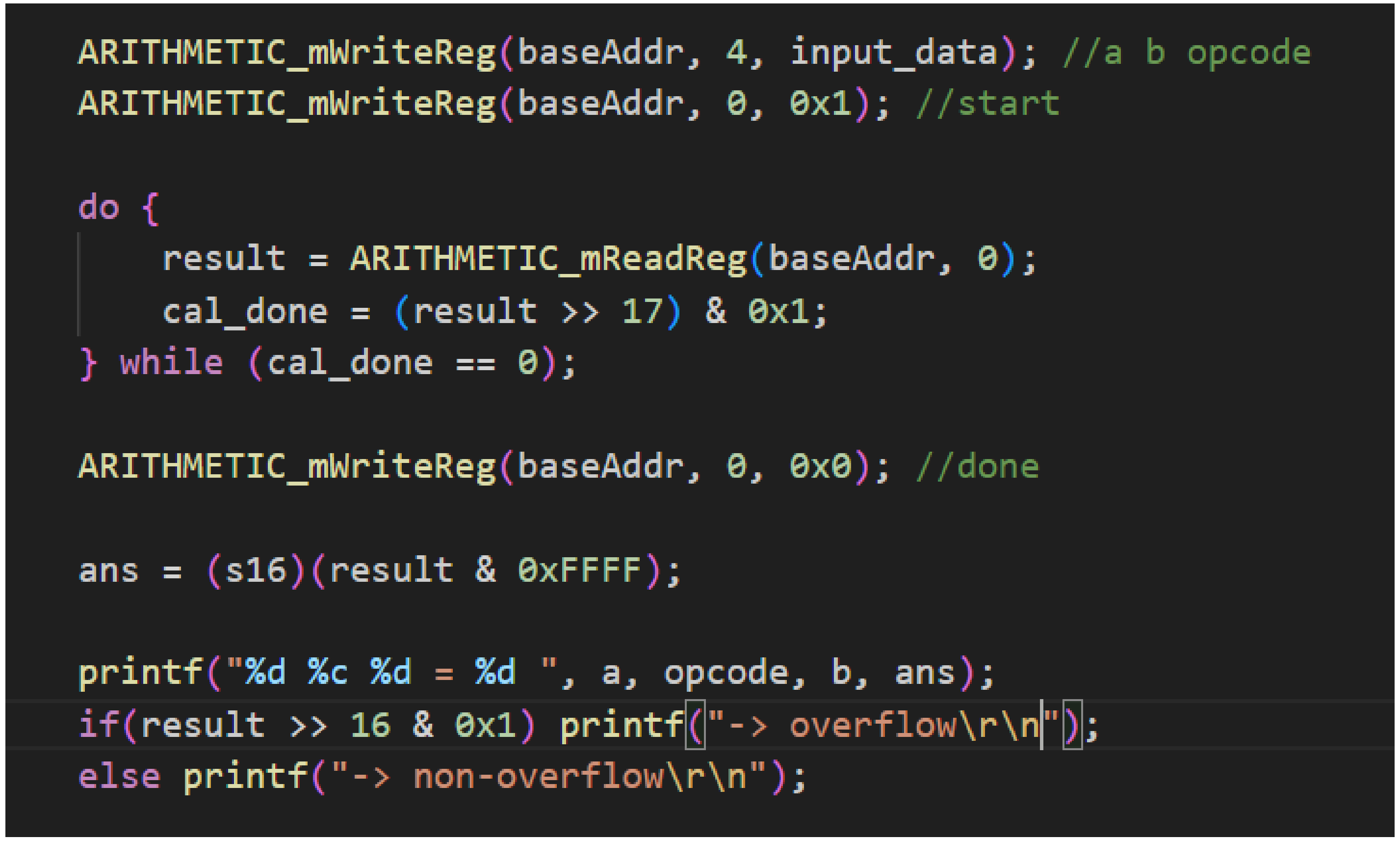
```
void calculation(UINTPTR baseAddr, s8 a, s8 b, u8 opcode) {
    u32 result;
    s16 ans;
    u8 cal_done = 0;
    u32 input_data = ((u32)opcode << 16) | ((u32)(u8)b << 8) |  ((u32)(u8)a) ;
```

整理要傳入PL端的input_data

Input:

- baseAddr
- 8 bits 有號數 a
- 8 bits 有號數 b
- 8 bits opcode

Function - calculation



```
ARITHMETIC_mWriteReg(baseAddr, 4, input_data); //a b opcode
ARITHMETIC_mWriteReg(baseAddr, 0, 0x1); //start

do {
    result = ARITHMETIC_mReadReg(baseAddr, 0);
    cal_done = (result >> 17) & 0x1;
} while (cal_done == 0);

ARITHMETIC_mWriteReg(baseAddr, 0, 0x0); //done

ans = (s16)(result & 0xFFFF);

printf("%d %c %d = %d ", a, opcode, b, ans);
if(result >> 16 & 0x1) printf("-> overflow\r\n");
else printf("-> non-overflow\r\n");
```

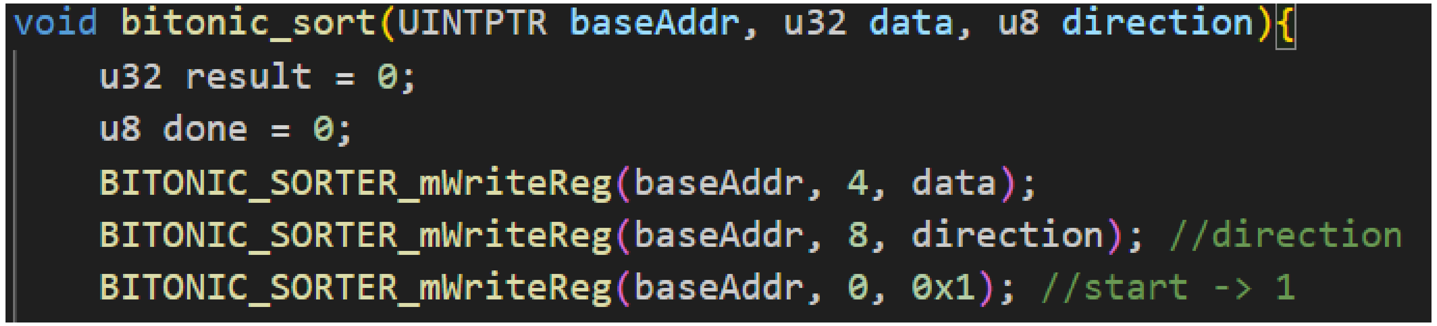
寫入input_data 以及 start 訊號

等待 cal_done 訊號表示計算完成

完成計算 輸出結果

Function - bitonic_sort

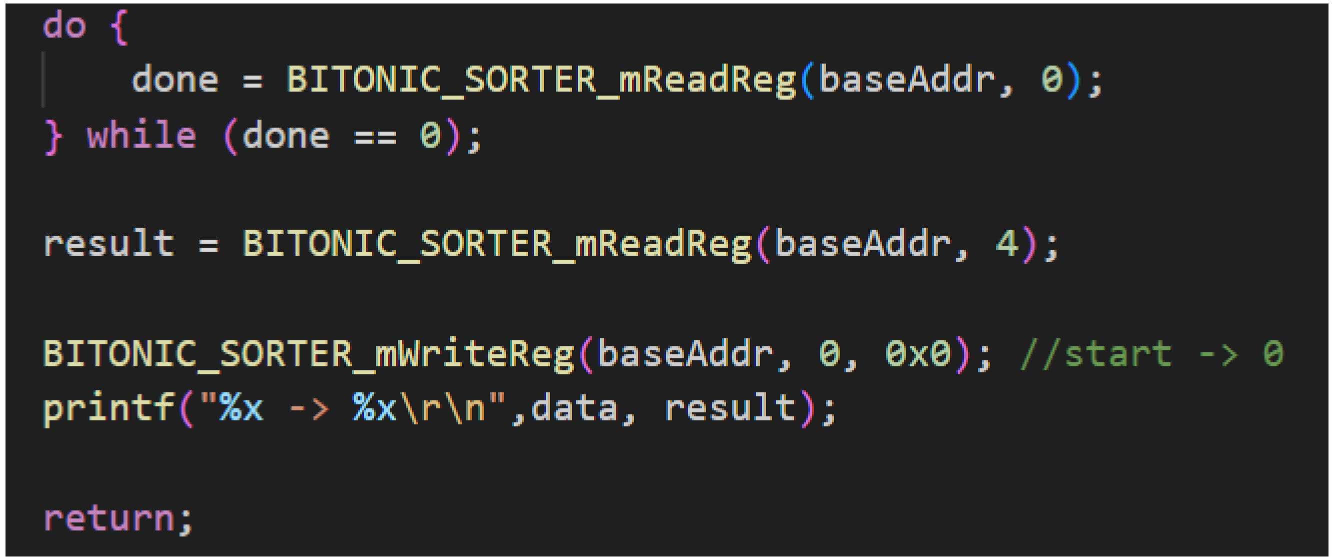
Input:
- baseAddr
- 32 bits unsigned data
- 8 bits   unsigned direction



```
void bitonic_sort(UINTPTR baseAddr, u32 data, u8 direction){
    u32 result = 0;
    u8 done = 0;
    BITONIC_SORTER_mWriteReg(baseAddr, 4, data);
    BITONIC_SORTER_mWriteReg(baseAddr, 8, direction); //direction
    BITONIC_SORTER_mWriteReg(baseAddr, 0, 0x1); //start -> 1
```

data 寫入 slv_reg1
direction 寫入 slv_reg2
slv_reg0 寫入 1 表示 start 訊號為1

Function - bitonic_sort



```c
do {
    done = BITONIC_SORTER_mReadReg(baseAddr, 0);
} while (done == 0);

result = BITONIC_SORTER_mReadReg(baseAddr, 4);

BITONIC_SORTER_mWriteReg(baseAddr, 0, 0x0); //start -> 0
printf("%x -> %x\r\n",data, result);

return;
```

等待 done 訊號為1

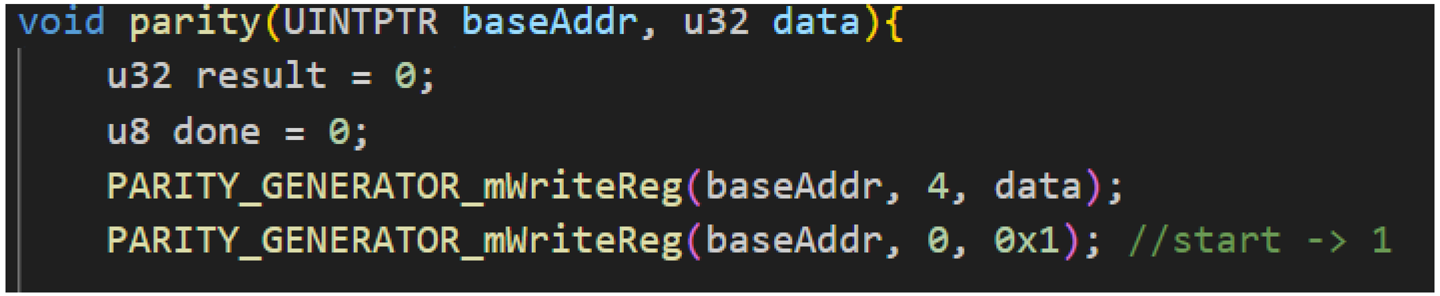
讀取slv_reg1的值
(slv_reg1內為排序好的序列)

start 訊號歸零

輸出經過排序後的序列

Function - parity

Input:
- baseAddr
- 32 bits unsigned data

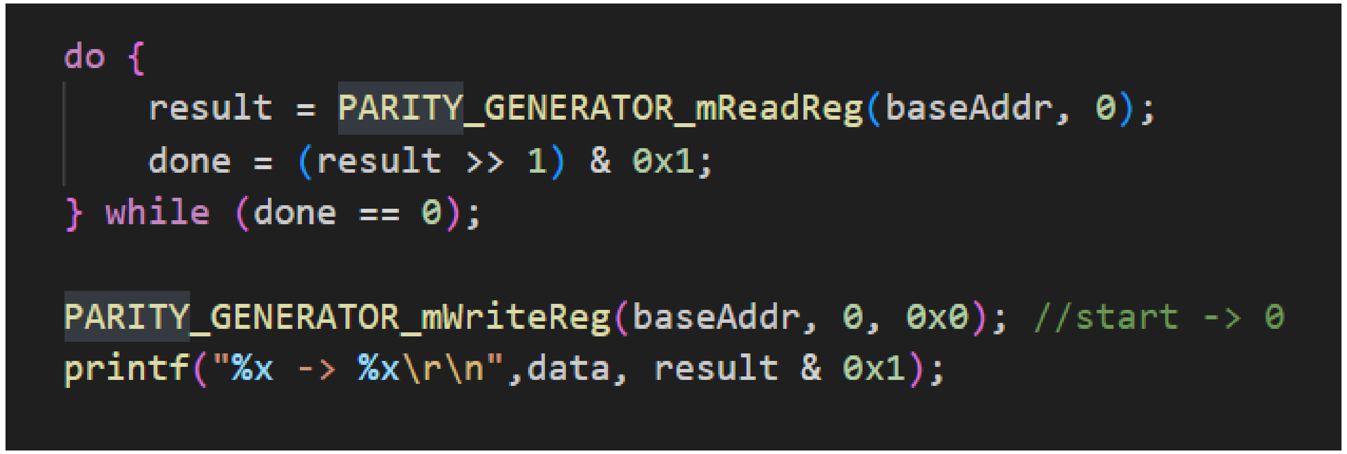


```
void parity(UINTPTR baseAddr, u32 data){
    u32 result = 0;
    u8 done = 0;
    PARITY_GENERATOR_mWriteReg(baseAddr, 4, data);
    PARITY_GENERATOR_mWriteReg(baseAddr, 0, 0x1); //start -> 1
```

data 寫入 slv_reg1

slv_reg0 寫入 1 表示 start 訊號為1

Function - parity



```
do {
    result = PARITY_GENERATOR_mReadReg(baseAddr, 0);
    done = (result >> 1) & 0x1;
} while (done == 0);

PARITY_GENERATOR_mWriteReg(baseAddr, 0, 0x0); //start -> 0
printf("%x -> %x\r\n",data, result & 0x1);
```

讀取slv_reg0的值

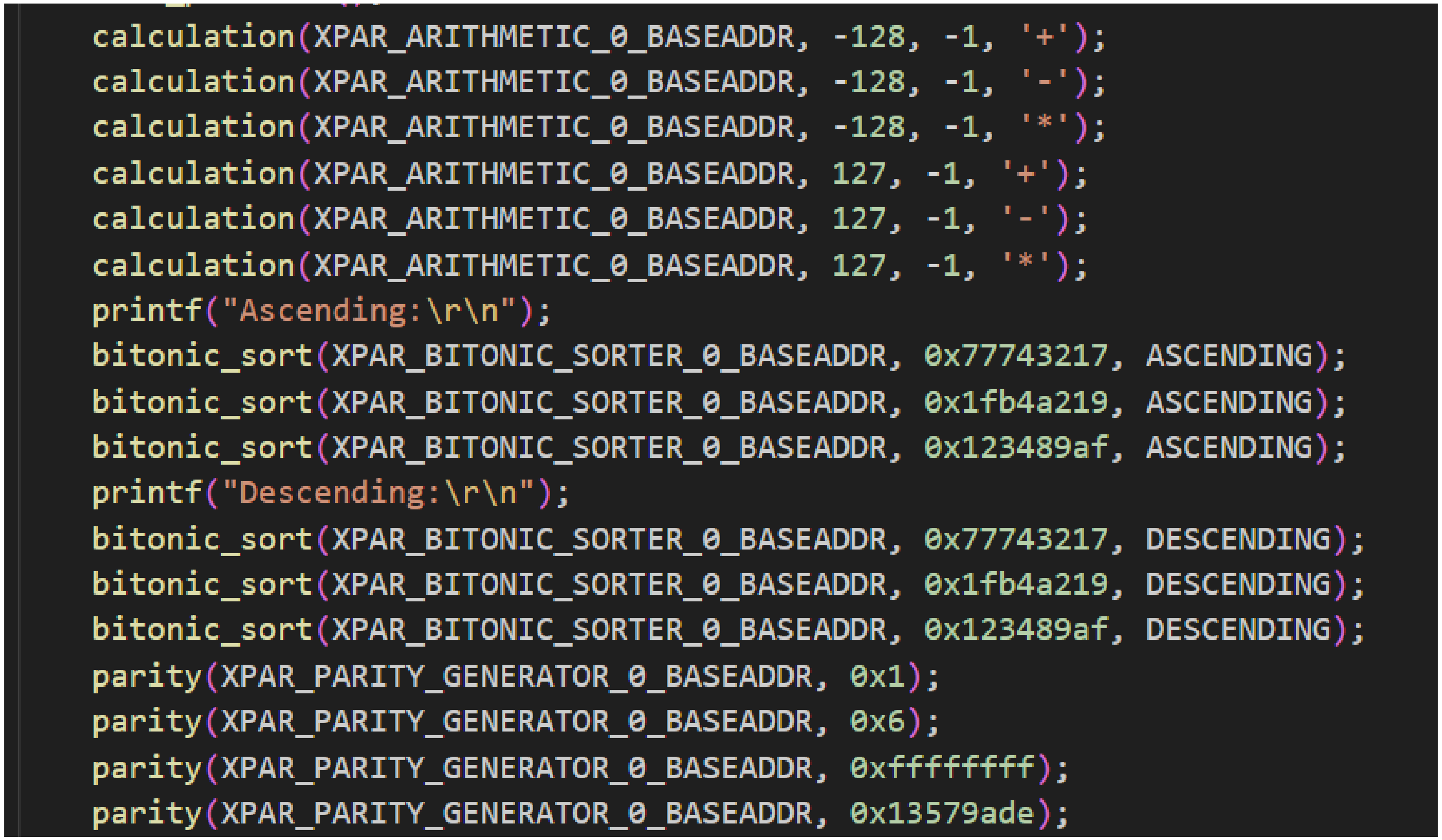
重複直到 done 訊號為1

start 訊號歸零

輸出parity bit

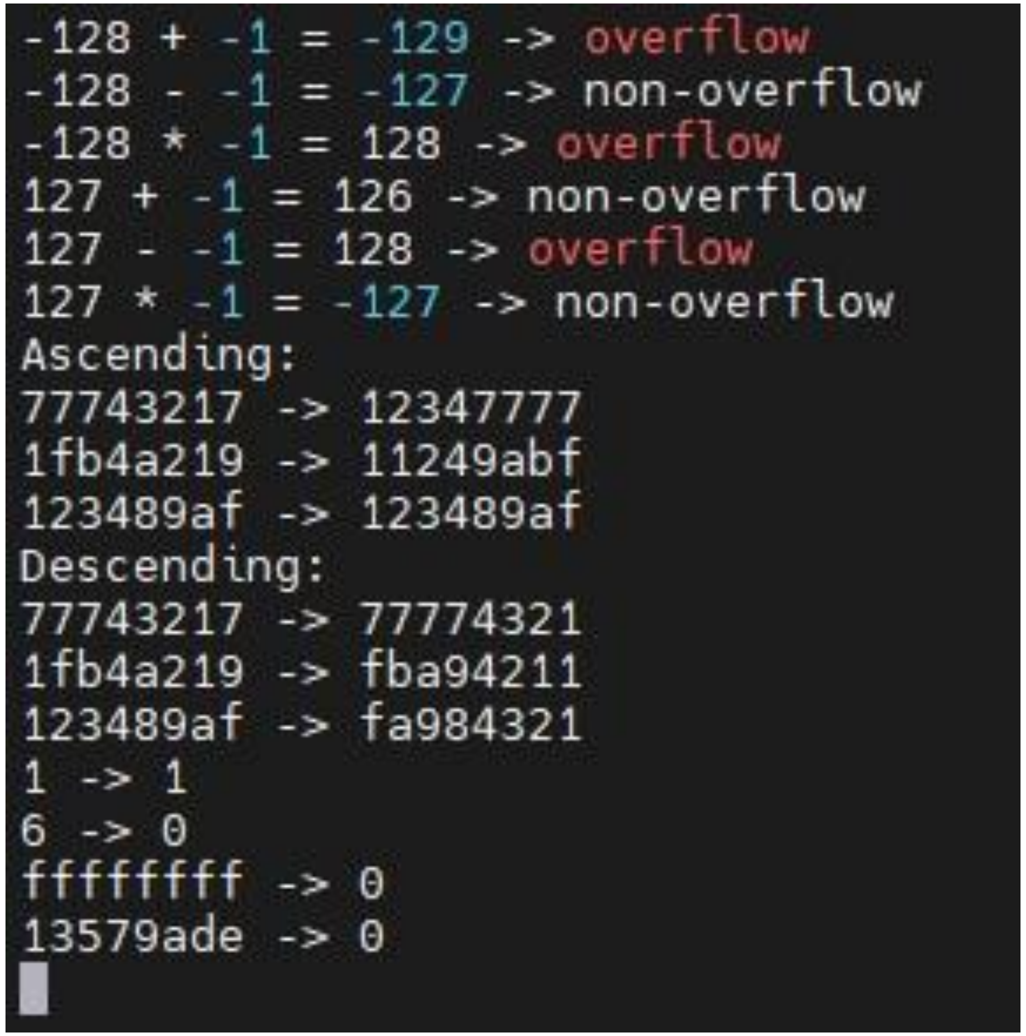
Result

輸入



```
calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '+');
calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '-');
calculation(XPAR_ARITHMETIC_0_BASEADDR, -128, -1, '*');
calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '+');
calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '-');
calculation(XPAR_ARITHMETIC_0_BASEADDR, 127, -1, '*');
printf("Ascending:\r\n");
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x77743217, ASCENDING);
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x1fb4a219, ASCENDING);
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x123489af, ASCENDING);
printf("Descending:\r\n");
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x77743217, DESCENDING);
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x1fb4a219, DESCENDING);
bitonic_sort(XPAR_BITONIC_SORTER_0_BASEADDR, 0x123489af, DESCENDING);
parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x1);
parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x6);
parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0xffffffff);
parity(XPAR_PARITY_GENERATOR_0_BASEADDR, 0x13579ade);
```

輸出



```
-128 + -1 = -129 -> overflow
-128 - -1 = -127 -> non-overflow
-128 * -1 = 128 -> overflow
127 + -1 = 126 -> non-overflow
127 - -1 = 128 -> overflow
127 * -1 = -127 -> non-overflow
Ascending:
77743217 -> 12347777
1fb4a219 -> 11249abf
123489af -> 123489af
Descending:
77743217 -> 77774321
1fb4a219 -> fba94211
123489af -> fa984321
1 -> 1
6 -> 0
ffffffff -> 0
13579ade -> 0
```