PA3: Pipeline MIPS CPU

Part3:

Area: $18365\mu m^2$ slack: 1.69

學生：李勁磊

學號：B11107048

一、Screenshots and descriptions of each module:

ALU:

通過調整 ADD、SUB、func、shift 的數值以減小面積

```
1   `define ADD 2'b01
2   `define SUB 2'b10
3   `define SLL 2'b00
4   `define OR 2'b11
5
6   module ALU(
7   input [31:0] Src1,
8   input [31:0] Src2,
9   input [1:0] func,
10  input [4:0] shift,
11  output reg [31:0] Result);
12
13      wire [8:0] Carry_internal;
14      wire [31:0] Sum;
15      wire [31:0] ADD_SUB_Src2;
16      // Generate 8 CLA4 blocks
17      assign Carry_internal[0] = (func == `ADD) ? 0 : (func == `SUB) ? 1: 1'b0;
18      assign ADD_SUB_Src2 = (func == `ADD) ? Src2:
19                            (func == `SUB) ? ~Src2: 32'b0;
20      genvar i;
21      generate
22          for (i = 0; i < 8; i = i + 1) begin : CLA4_BLOCK
23              wire [3:0] A = Src1[i*4 +: 4];
24              wire [3:0] B = ADD_SUB_Src2[i*4 +: 4];
25              wire [3:0] G, P, C;
26
27              assign G = A & B;
28              assign P = A ^ B;
29
30              assign C[0] = Carry_internal[i];
31              assign C[1] = G[0] | (P[0] & C[0]);
32              assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
33              assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & C[0]);
34              assign Carry_internal[i+1] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) |
35                                           (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & C[0]);
36              assign Sum[i*4 +: 4] = P ^ C;
37          end
38      endgenerate
39
40      always@(*) begin
41          case(func)
42              `ADD: begin
43                  Result = Sum;
44              end
45              `SUB: begin
46                  Result = Sum;
47              end
48              `SLL: begin
49                  Result = Src1 << shift;
50              end
51              `OR: begin
52                  Result = Src1 | Src2;
53              end
54          endcase
55      end
56  endmodule
```

1~11	宣告接腳
12~42	CLA 電路合成
40~56	執行計算

Control:

將每個狀態的 control signal 分別寫出，讓每個 block 可以正常運作

```
`define ori_op 4'b1101
`define lw_op 4'b0011
`define sw_op 4'b1011
`define addi_op 4'b1001
`define R_TYPE_op 4'b0000

`define R_type 2'b00
`define ADD 2'b01
`define SUB 2'b10
`define OR 2'b11

module Control(
    input wire [3:0] OpCode,
    output reg [1:0] ALU_OP,
    output reg Reg_Dst,
    output reg Reg_w,
    output reg ALU_src,
    output reg Mem_to_reg,
    output reg Mem_w
);

    always@(*) begin
        if(OpCode == `R_TYPE_op) begin
            Reg_Dst = 1;
            Reg_w = 1;
            ALU_OP = `R_type;
            ALU_src = 0;
            Mem_to_reg = 0;
            Mem_w = 0;
        end
        else if(OpCode == `addi_op) begin
            Reg_Dst = 0;
            Reg_w = 1;
            ALU_OP = `ADD;
            ALU_src = 1;
            Mem_to_reg = 0;
            Mem_w = 0;
        end
        else if(OpCode == `sw_op) begin
            Reg_Dst = 0;
            Reg_w = 0;
```

```
42                  ALU_OP = `ADD;
43                  ALU_src = 1;
44                  Mem_to_reg = 0;
45                  Mem_w = 1;
46              end
47              else if(OpCode == `lw_op) begin
48                  Reg_Dst = 0;
49                  Reg_w = 1;
50                  ALU_OP = `ADD;
51                  ALU_src = 1;
52                  Mem_to_reg = 1;
53                  Mem_w = 0;
54              end
55              else if(OpCode == `ori_op) begin
56                  Reg_Dst = 0;
57                  Reg_w = 1;
58                  ALU_OP = `OR;
59                  ALU_src = 1;
60                  Mem_to_reg = 0;
61                  Mem_w = 0;
62              end
63              else begin
64                  Reg_Dst = 1'b?;
65                  Reg_w = 0;
66                  ALU_OP = 2'b?;
67                  ALU_src = 1'b?;
68                  Mem_to_reg = 1'b?;
69                  Mem_w = 0;
70              end
71          end
72      endmodule
```

1~20	宣告接腳
22~70	將每種可能寫出來

RF:

在 negedge 時把資料寫入

```
35  module RF(
36      // Outputs
37      output wire [31:0] RsData,
38      output wire [31:0] RtData,
39      // Inputs
40      input wire [4:0] RsAddr,
41      input wire [4:0] RtAddr,
42      input wire [4:0] RdAddr,
43      input wire [31:0] RdData,
44      input wire RegWrite,
45      input wire clk
46  );
47
48      /*
49       * Declaration of inner register.
50       * CAUTION: DONT MODIFY THE NAME AND SIZE.
51       */
52      reg [31:0]R[0:`REG_MEM_SIZE - 1];
53      assign RsData = R[RsAddr];
54      assign RtData = R[RtAddr];
55
56      always@(negedge clk) begin
57          if(RegWrite == 1) begin
58              R[RdAddr] = RdData;
59          end
60      end
61  endmodule
```

35~46	宣告接腳
48~61	控制讀出跟寫入訊號

DM

讓資料讀取用 combinational 寫入用 negedge

```
35  module DM (
36      // Outputs
37      output  reg     [31:0]  MemReadData,
38      // Inputs
39      input   wire    [31:0]  MemAddr,
40      input   wire    [31:0]  MemWriteData,
41      input   wire            MemWrite,
42      input   wire            clk );
43      /*
44       * Declaration of inner register.
45       * CAUTION: DONT MODIFY THE NAME AND SIZE.
46       */
47      reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
48
49      assign MemReadData = {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr +
50
51      always@(negedge clk)
52      begin
53          if(MemWrite) begin
54              {DataMem[MemAddr], DataMem[MemAddr + 1], DataMem[MemAddr + 2], DataMem[MemAddr + 3]} <= MemWr
55          end
56      end
57  endmodule
```

35~42	宣告接腳
47~57	控制讀出跟寫入訊號

ALU_control

根據 func_ctrl 跟 ALU_OP 看要控制 ALU 做甚麼運算，在觀察 Opcode 跟 func 後發現其實只要其中幾個幾個 bit 就可以但斷出結果，所以做了一些優化

```
1   `define R_type 2'b00
2   `define ADD 2'b01
3   `define SUB 2'b10
4   `define OR 2'b11
5   `define SLL 2'b00
6
7   `define R_type_ADD 3'b001
8   `define R_type_SUB 3'b011
9   `define R_type_SLL 3'b000
10  `define R_type_OR  3'b101
11
12  module ALU_control(
13      input [2:0] funct_ctrl,
14      input [1:0] ALU_OP,
15      output reg [1:0] ALU_function
16  );
17
18      always@(*) begin
19          if(ALU_OP == `R_type) begin
20              case(funct_ctrl)
21                  `R_type_ADD: begin
22                      ALU_function = `ADD;
23                  end
24                  `R_type_SUB: begin
25                      ALU_function = `SUB;
26                  end
27                  `R_type_SLL: begin
28                      ALU_function = `SLL;
29                  end
30                  `R_type_OR: begin
31                      ALU_function = `OR;
32                  end
33              endcase
34          end
35          else begin
36              ALU_function = ALU_OP;
37          end
38      end
39
40  endmodule
```

1~4	宣告接腳
20~38	只寫出 R_type 的 funct_ctrl 的部分，其他部分對不同的功能做排序，以減少面積。

Pipeline register:

將每個 state 的 register 分成一個大的 reg

```
module pipline_register_1(
    input [29:0] instruction,
    input clk,
    input stall,
    output reg [29:0] stage1
);
    always@(posedge clk) begin
        if(!stall) begin
            stage1 <= instruction;
        end
    end
endmodule
module pipline_register_2(
    input [109:0]input_bus,
    input clk,
    output reg [109:0]stage2 // shift, ALU_op, ALU_OP
);
    always@(posedge clk) begin
        stage2 <= input_bus;
    end
endmodule
module pipline_register_3(
    input [71:0]input_bus,
    input clk,
    output reg [71:0] stage3
);
    always@(posedge clk) begin
        stage3 <= input_bus;
    end
endmodule
module pipline_register_4(
    input [70:0]input_bus,
    input clk,
    output reg [70:0] stage4
);
    always@(posedge clk) begin
        stage4 <= input_bus;
    end
endmodule
```

IM:

使用 assign 把 IM 的資料傳入 CPU

```
module IM(
    // Outputs
    output wire [31:0] Instr,
    // Inputs
    input wire [31:0] InstrAddr
);

    /*
     * Declaration of instruction memory.
     * CAUTION: DONT MODIFY THE NAME AND SIZE.
     */
    reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
    assign Instr = {InstrMem[InstrAddr],InstrMem[InstrAddr + 1], InstrMem[InstrAddr + 2], InstrMem[InstrAddr + 3]};
endmodule
```

二、Descriptions test commands for each module:

R type:

```
addu $20, $1, $2
addu $21, $31, $29
addu $22, $23, $24
addu $23, $24, $25
subu $24, $31, $8
subu $25, $6, $5
subu $26, $26, $27
sll $27, $31, 4
sll $28, $31, 8
sll $1, $31, 31
or $2, $8, $9
or $3, $29, $29
or $4, $2, $0
```

分別測試:
正常的數值相加
最大最小值的相加
正常的數值相減
小減大
0 減 0
左移
OR 功能測試

```
// Instruction Memory in Hex
// addu $20, $1, $2
// 000000 00001 00010 10100 00000 100001
00 // Addr = 0x00
22 // Addr = 0x01
A0 // Addr = 0x02
21 // Addr = 0x03
// addu $21, $31, $29
// 000000 11111 11101 10101 00000 100001
03 // Addr = 0x04
FD // Addr = 0x05
A8 // Addr = 0x06
21 // Addr = 0x07
// addu $22, $23, $24
// 000000 10111 11000 10110 00000 100001
02 // Addr = 0x08
F8 // Addr = 0x09
B0 // Addr = 0x0A
21 // Addr = 0x0B
// addu $23, $24, $25
// 000000 11000 11001 10111 00000 100001
03 // Addr = 0x0C
19 // Addr = 0x0D
B8 // Addr = 0x0E
21 // Addr = 0x0F
// subu $24, $31, $8
// 000000 11111 01000 11000 00000 100011
03 // Addr = 0x10
E8 // Addr = 0x11
C0 // Addr = 0x12
23 // Addr = 0x13
// subu $25, $6, $5
// 000000 00110 00101 11001 00000 100011
00 // Addr = 0x14
C5 // Addr = 0x15
C8 // Addr = 0x16
23 // Addr = 0x17
```

```
// subu $26, $26, $27
// 000000 11010 11011 11010 00000 100011
03 // Addr = 0x18
5B // Addr = 0x19
D0 // Addr = 0x1A
23 // Addr = 0x1B
// sll $27, $31, 4
// 000000 11111 00000 11011 00100 000000
03 // Addr = 0x1C
E0 // Addr = 0x1D
D9 // Addr = 0x1E
00 // Addr = 0x1F
// sll $28, $31, 8
// 000000 11111 00000 11100 01000 000000
03 // Addr = 0x20
E0 // Addr = 0x21
E2 // Addr = 0x22
00 // Addr = 0x23
// sll $1, $31, 31
// 000000 11111 00000 00001 11111 000000
03 // Addr = 0x24
E0 // Addr = 0x25
0F // Addr = 0x26
C0 // Addr = 0x27
// or $2, $8, $9
// 000000 01000 01001 00010 00000 100101
01 // Addr = 0x28
09 // Addr = 0x29
10 // Addr = 0x2A
25 // Addr = 0x2B
// or $3, $29, $29
// 000000 11101 11101 00011 00000 100101
03 // Addr = 0x2C
BD // Addr = 0x2D
18 // Addr = 0x2E
25 // Addr = 0x2F
// or $4, $2, $0
// 000000 00010 00000 00100 00000 100101
00 // Addr = 0x30
40 // Addr = 0x31
20 // Addr = 0x32
25 // Addr = 0x33
```

I type:

```
addiu $20, $2, 0x0000_0007
addiu $21, $31, 0x0000_0001
addiu $22, $13, 0xFFFF_FFFE
sw $0, 5($29)
sw $19, 4($29)
sw $7, 16($29)
sw $9, 12($19)
lw $23, 9($29)
lw $24, 16($29)
lw $25, 16($12)
ori $26, $30, 0x1234_1234
ori $27, $3, 0x3333_3333
```

分別測試:

正常加法

最大值加 1

零加其他數值

存數值

讀數值

OR 功能測試

```
// Instruction Memory in Hex
// addiu $20, $2, 0x0000_0007
// 001001 00010 10100 0000000000000111
24 // Addr = 0x00
54 // Addr = 0x01
00 // Addr = 0x02
07 // Addr = 0x03
// addiu $21, $31, 0x0000_0001
// 001001 11111 10101 0000000000000001
27 // Addr = 0x04
F5 // Addr = 0x05
00 // Addr = 0x06
01 // Addr = 0x07
// addiu $22, $13, 0xFFFF_FFFE
// 001001 01101 10110 1111111111111110
25 // Addr = 0x08
B6 // Addr = 0x09
FF // Addr = 0x0A
FE // Addr = 0x0B
// sw $0, 5($29)
// 101011 11101 00000 0000000000000101
AF // Addr = 0x0C
A0 // Addr = 0x0D
00 // Addr = 0x0E
05 // Addr = 0x0F
// sw $19, 4($29)
// 101011 11101 10011 0000000000000100
AF // Addr = 0x10
B3 // Addr = 0x11
00 // Addr = 0x12
04 // Addr = 0x13
// sw $7, 16($29)
// 101011 11101 00111 0000000000010000
AF // Addr = 0x14
A7 // Addr = 0x15
00 // Addr = 0x16
10 // Addr = 0x17
// sw $9, 12($19)
// 101011 10011 01001 0000000000001100
AE // Addr = 0x18
69 // Addr = 0x19
00 // Addr = 0x1A
0C // Addr = 0x1B
```

```
// lw $23, 9($29)
// 100011 11101 10111 0000000000001001
8F // Addr = 0x1C
B7 // Addr = 0x1D
00 // Addr = 0x1E
09 // Addr = 0x1F
// lw $24, 16($29)
// 100011 11101 11000 0000000000010000
8F // Addr = 0x20
B8 // Addr = 0x21
00 // Addr = 0x22
10 // Addr = 0x23
// lw $25, 16($12)
// 100011 01100 11001 0000000000010000
8D // Addr = 0x24
99 // Addr = 0x25
00 // Addr = 0x26
10 // Addr = 0x27
// ori $26, $30, 0x1234_1234
// 001101 11110 11010 0001001000110100
37 // Addr = 0x28
DA // Addr = 0x29
12 // Addr = 0x2A
34 // Addr = 0x2B
// ori $27, $3, 0x3333_3333
// 001101 00011 11011 0011001100110011
```

Test bench with Hazard:

```
addu $1, $1, $2
addu $1, $1, $10
addu $1, $1, $11
addu $1, $1, $12
subu $28, $1, $2
subu $28, $28, $10
subu $28, $28, $11
subu $28, $28, $12
lw $20, 5($29)
or $21, $20, $6
lw $22, 5($29)
sll $23, $22, 4
sw $29, 8($29)
or $24, $8, $6
addu $2, $13, $14
addu $3, $2, $14
addu $4, $2, $3
subu $5, $13, $14
subu $6, $5, $14
subu $7, $6, $5
addiu $8, $13, 1
addiu $9, $8, 1
addiu $10, $8, 1
ori $11, $10, 0xFF00
```

測試 forwarding 跟 Data Hazard、lw 後面接 R type 測資

```
// Instruction Memory in Hex
// addu $1, $1, $2
// 000000 00001 00010 00001 00000 100001
00 // Addr = 0x00
22 // Addr = 0x01
08 // Addr = 0x02
21 // Addr = 0x03
// addu $1, $1, $10
// 000000 00001 01010 00001 00000 100001
00 // Addr = 0x04
2A // Addr = 0x05
08 // Addr = 0x06
21 // Addr = 0x07
// addu $1, $1, $11
// 000000 00001 01011 00001 00000 100001
00 // Addr = 0x08
2B // Addr = 0x09
08 // Addr = 0x0A
21 // Addr = 0x0B
// addu $1, $1, $12
// 000000 00001 01100 00001 00000 100001
00 // Addr = 0x0C
2C // Addr = 0x0D
08 // Addr = 0x0E
21 // Addr = 0x0F
// subu $28, $1, $2
// 000000 00001 00010 11100 00000 100011
00 // Addr = 0x10
22 // Addr = 0x11
E0 // Addr = 0x12
23 // Addr = 0x13
// subu $28, $28, $10
// 000000 11100 01010 11100 00000 100011
03 // Addr = 0x14
8A // Addr = 0x15
E0 // Addr = 0x16
23 // Addr = 0x17
// subu $28, $28, $11
// 000000 11100 01011 11100 00000 100011
03 // Addr = 0x18
8B // Addr = 0x19
E0 // Addr = 0x1A
23 // Addr = 0x1B
// subu $28, $28, $12
// 000000 11100 01100 11100 00000 100011
03 // Addr = 0x1C
8C // Addr = 0x1D
E0 // Addr = 0x1E
23 // Addr = 0x1F
// lw $20, 5($29)
// 100011 11101 10100 0000000000000101
8F // Addr = 0x20
B4 // Addr = 0x21
00 // Addr = 0x22
05 // Addr = 0x23
// or $21, $20, $6
// 000000 10100 00110 10101 00000 100101
02 // Addr = 0x24
86 // Addr = 0x25
A8 // Addr = 0x26
25 // Addr = 0x27
// lw $22, 5($29)
// 100011 11101 10110 0000000000000101
8F // Addr = 0x28
B6 // Addr = 0x29
00 // Addr = 0x2A
05 // Addr = 0x2B
// sll $23, $22, 4
// 000000 10110 00000 10111 00100 000000
02 // Addr = 0x2C
C0 // Addr = 0x2D
B9 // Addr = 0x2E
00 // Addr = 0x2F
// sw $29, 8($29)
// 101011 11101 11101 0000000000001000
AF // Addr = 0x30
BD // Addr = 0x31
00 // Addr = 0x32
08 // Addr = 0x33
// or $24, $8, $6
// 000000 01000 00110 11000 00000 100101
01 // Addr = 0x34
```

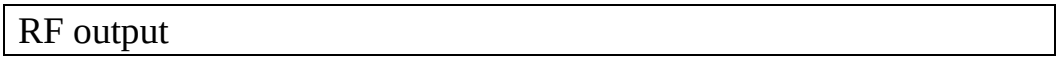
```
06 // Addr = 0x35
C0 // Addr = 0x36
25 // Addr = 0x37
// addu $2, $13, $14
// 000000 01101 01110 00010 00000 100001
01 // Addr = 0x38
AE // Addr = 0x39
10 // Addr = 0x3A
21 // Addr = 0x3B
// addu $3, $2, $14
// 000000 00010 01110 00011 00000 100001
00 // Addr = 0x3C
4E // Addr = 0x3D
18 // Addr = 0x3E
21 // Addr = 0x3F
// addu $4, $2, $3
// 000000 00010 00011 00100 00000 100001
00 // Addr = 0x40
43 // Addr = 0x41
20 // Addr = 0x42
21 // Addr = 0x43
// subu $5, $13, $14
// 000000 01101 01110 00101 00000 100011
01 // Addr = 0x44
AE // Addr = 0x45
28 // Addr = 0x46
23 // Addr = 0x47
// subu $6, $5, $14
// 000000 00101 01110 00110 00000 100011
00 // Addr = 0x48
AE // Addr = 0x49
30 // Addr = 0x4A
23 // Addr = 0x4B
// subu $7, $6, $5
// 000000 00110 00101 00111 00000 100011
00 // Addr = 0x4C
C5 // Addr = 0x4D
38 // Addr = 0x4E
23 // Addr = 0x4F
// addiu $8, $13, 1
// 001001 01101 01000 0000000000000001
```

```
25 // Addr = 0x50
A8 // Addr = 0x51
00 // Addr = 0x52
01 // Addr = 0x53
// addiu $9, $8, 1
// 001001 01000 01001 0000000000000001
25 // Addr = 0x54
09 // Addr = 0x55
00 // Addr = 0x56
01 // Addr = 0x57
// addiu $10, $8, 1
// 001001 01000 01010 0000000000000001
25 // Addr = 0x58
0A // Addr = 0x59
00 // Addr = 0x5A
01 // Addr = 0x5B
// ori $11, $10, 0xFF00
// 001101 01010 01011 1111111100000000
35 // Addr = 0x5C
4B // Addr = 0x5D
FF // Addr = 0x5E
00 // Addr = 0x5F
FF // Addr = 0x60
FF // Addr = 0x61
FF // Addr = 0x62
FF // Addr = 0x63
FF // Addr = 0x64
FF // Addr = 0x65
FF // Addr = 0x66
FF // Addr = 0x67
FF // Addr = 0x68
FF // Addr = 0x69
FF // Addr = 0x6A
FF // Addr = 0x6B
FF // Addr = 0x6C
FF // Addr = 0x6D
FF // Addr = 0x6E
FF // Addr = 0x6F
FF // Addr = 0x70
```

三、Stimulation Result:

R type:

輸出符合預期



1 00000001 2 80000000 3 ffffffff 4 00000000 5 ffffffff 6 f7f7f7f7 7 7fffffff 8 80000000 9 ffff0000 10 0000ffff 11 0000000a 12 000000a0 13 00000002 14 00000001 15 00000003 16 00000007	17 00000002 18 00000037 19 00000064 20 00000040 21 00000004 22 ffffffff 23 00000000 24 00000000 25 0000ffff 26 88080808 27 00000000 28 fffffff0 29 ffffff00 30 00000000 31 ffffffff 32 ffffffff

I type:

輸出結果符合預期

DM out	RF out	
1 ff 2 ff 3 ff 4 ff 5 00 6 00 7 00 8 40 9 01 10 ff 11 ff 12 ff 13 ff 14 ff 15 ff 16 ff 17 80 18 00 19 00 20 00	1 00000001 2 00000001 3 00000003 4 77777777 5 7f7f7f7f 6 f7f7f7f7 7 7fffffff 8 80000000 9 ffff0000 10 0000ffff 11 0000000a 12 000000a0 13 00000002 14 00000001 15 00000003 16 00000007	17 00000002 18 00000037 19 00000064 20 00000040 21 0000000a 22 00000000 23 0000ffff 24 ffffffff 25 80000000 26 0000ffff 27 ffffffff 28 77777777 29 00000000 30 00000000 31 ffffffff 32 ffffffff

Data Hazard and forwarding 測試:

輸出結果符合預期

DM out (只截圖一部分，因為檔案太長)	RF out	
1 ff 2 ff 3 ff 4 ff 5 ff 6 ff 7 ff 8 ff 9 00 10 00 11 00 12 00 13 ff 14 ff 15 ff 16 ff	1 00000001 2 000000b0 3 00000004 4 00000007 5 0000000b 6 fffffffe 7 fffffffb 8 fffffffd 9 00000002 10 00000003 11 00000003 12 0000ff03 13 00000002 14 00000001 15 00000003 16 00000007	17 00000002 18 00000037 19 00000064 20 00000040 21 ffffffff 22 ffffffff 23 ffffffff 24 fffffff0 25 ffffffff 26 00000000 27 00000000 28 00000000 29 00000001 30 00000000 31 ffffffff 32 ffffffff

Compare PA2(SimpleCPU) with PA3(pipline CPU):

The synthesis result of DM = 12:

	SimpleCPU	PiplineCPU
Area	$14186\mu m^2$	$14811\mu m^2$
Slack	4.26	1.8081+2.5=4.30
Power	2.92mW	2.81mW
總結:		

在面積方面，SimpleCPU 使用的面積約為 14186，而 PipelineCPU 略高，約為 14811 平方微米。這表示 PipelineCPU 由於其流水線結構，稍增增加了電路的面積需求。

時序延遲（Slack）方面，SimpleCPU 的 Slack 為 4.26，而 PipelineCPU 考慮正負源的關係約為 4.30，整體略優於 SimpleCPU，代表 PipelineCPU 在時序上具備更快執行速度。

功耗表現上，SimpleCPU 約為 2.92 mW，PipelineCPU 則稍低為 2.81 mW，顯示流水線設計在功耗控制方面也稍有優勢，因為 register 變多了。

四、Memory Rethinking

IM 跟 DM 皆可以增加 Cache。

將 DM 改成 Cache，判斷 Cache 是否有 miss，如果是那就暫停指令，直到 Cache 從 Data memory 取得資料，如果是 hit 則用一般的讀寫邏輯即可。

如果是將 IM 改成 Cache，判斷 Cache 是否有 miss，如果是那就暫停指令，直到 Cache 從 Instruction memory 取得資料，如果是 hit 則用一般的讀寫邏輯即可。

Cache 需要新增 Read_Miss, Write_Miss 接腳，讓資料沒有被正確讀取時可以 stall CPU 的運作。

五、Conclusion and insight on this homework.:

在寫這個作業時我發現有很多可以優化的部分，像是 Reg_to_Mem 可以把它移到 stage3，這樣 pipeline register 就可以縮小，雖然這樣會讓 slack 變差。另外+4 的那個 adder 我們其實只要 7 個 bit 就可以算到 127，因為 IM 的大小就只有到 128，所以不會有超過的可能。最後 sign extension 我們把它移到 stage 2，這樣可以縮小 pipeline register1 的位元數。通過以上的優化，我們可以把面積縮小大約 700 μm^2，然後 slack 只下降 0.1 左右