



計算機組織 PA2

電機三甲 B11107056 黃敏聖

Area : 14372 (DM 縮到 12) Worst slack : 3.59

```
Chip area for top module '\SimpleCPU': 14372.512000
of which used for sequential elements: 0.000000 (0.00%)
finish report_worst_slack
-----
worst slack 3.59
```

壹、Part1

一、程式碼

R-FormatCPU:

```
29 module R_FormatCPU(
30   // Outputs
31   output wire [31:0] Output_Addr,
32   // Inputs
33   input  wire [31:0] Input_Addr,
34   input  wire          clk
35 );
36
37   wire [31:0] Instruction;
38   wire Reg_w, ALU_op;
39   wire [1:0] Funct;
40   wire [31:0] Rs_data, Rt_data, Rd_data;
41
42   /*
43    * Declaration of Instruction Memory.
44    * CAUTION: DONT MODIFY THE NAME.
45    */
46   IM Instr_Memory(
47     // Outputs
48     .Instr(Instruction),
49     // Inputs
50     .InstrAddr(Input_Addr)
51 );
52
53   /*
54    * Declaration of Register File.
55    * CAUTION: DONT MODIFY THE NAME.
56    */
57   RF Register_File(
58     // Outputs
59     .RsData(Rs_data),
60     .RtData(Rt_data),
61     // Inputs
62     .RdData(Rd_data),
63     .RsAddr(Instruction[25:21]),
64     .RtAddr(Instruction[20:16]),
65     .RdAddr(Instruction[15:11]),
66     .RegWrite(Reg_w),
67     .clk(clk)
68 );
69
70   Adder ad0(
71     // Inputs
72     .Input_addr(Input_Addr),
73     // Outputs
74     .Output_addr(Output_Addr)
75 );
76
77   control c0(
78     // Inputs
79     .Opcode(Instruction[31:26]),
80     // Outputs
81     .Reg_w(Reg_w),
82     .ALU_op(ALU_op)
83 );
84
85   ALU_control alu_c0(
86     // Inputs
87     .Funct_ctrl(Instruction[5:0]),
88     .ALU_op(ALU_op),
89     // Outputs
90     .Funct(Funct) //0:+ , 1:- , 2:<< , 3:|
91 );
92
93   ALU alu0(
94     // Inputs
95     .Rs_data(Rs_data),
96     .Rt_data(Rt_data),
97     .shamt(Instruction[10:6]),
98     .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
99     // Outputs
100    .Rd_data(Rd_data)
101 );
102
103 endmodule
```

這段程式主要是將各功能的 module 連接在一起。

Adder:

```
1 module Adder (
2   input [31:0] Input_addr,
3   output [31:0] Output_addr
4 );
5   assign Output_addr = Input_addr + 32'd4;
6 endmodule
```

這個 module 主要負責處理加法的運算，是用於在執行一串指令後，將指令執行完畢後，PC 會自動加 4 來指向下一條指令的位置。

ALU_control:

```
ALU_control.v
1 module ALU_control (
2     input [5:0] Funct_ctrl,
3     input ALU_op,
4     output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5 );
6     always @(*) begin
7         if (ALU_op) begin
8             case (Funct_ctrl)
9                 6'b100001:Funct <= 2'd0;//+
10                6'b100011:Funct <= 2'd1;//-
11                6'b000000:Funct <= 2'd2;//<<
12                6'b100101:Funct <= 2'd3;//|
13                default:Funct <= 2'bz;
14            endcase
15        end
16    end
17 endmodule
```

這個 module 的功能是在處理 R-type 指令時，接收指令的後 6 位 (Funct_ctrl)，這 6 位用來表示 ALU 功能的控制。當該指令被解析後，會將 Funct 傳遞給 ALU，告訴 ALU 具體應該執行哪種運算操作，能根據不同的 R-type 指令執行相應的運算，從而完成加法、減法、邏輯運算等各種基本操作。

ALU:

```
ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] shamt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data
6 );
7     always @(*) begin
8         case (Funct)
9             2'd0: Rd_data <= Rs_data + Rt_data;
10            2'd1: Rd_data <= Rs_data - Rt_data;
11            2'd2: Rd_data <= Rs_data << shamt;
12            2'd3: Rd_data <= Rs_data | Rt_data;
13            default: Rd_data <= 32'bz;
14        endcase
15    end
16 endmodule
```

這個 module 的功能是用來執行暫存器之間的運算。它根據來自 ALU_control 的 Funct 信號來決定該執行哪種運算動作。Funct 提供了指令的具體類型（加法、減法、left shift、或），從而指導該單元進行相應的計算。

Control:

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w,
4     output reg ALU_op
5 );
6     always @(*) begin
7         if (Opcode == 6'd0) begin
8             ALU_op <= 1;
9             Reg_w <= 1;
10        end
11        else begin
12            ALU_op <= 0;
13            Reg_w <= 0;
14        end
15    end
16 endmodule
```

這個 module 的功能是根據指令的前 6 位 (Opcode) 來決定啟動的功能。根據 Opcode，該 module 能夠識別不同類型的指令並啟動對應的處理邏輯。然而，目前只處理 R-type 指令，因此整體結構相對簡單。

IM:

```
35 module IM(
36     // Outputs
37     output reg [31:0] Instr,
38     // Inputs
39     input [31:0] InstrAddr
40 );
41
42 /*
43  * Declaration of instruction memory.
44  * CAUTION: DONT MODIFY THE NAME AND SIZE.
45  */
46 reg [7:0] InstrMem[0:`INSTR_MEM_SIZE - 1];
47
48 always @(*) begin
49     Instr[31:24] <= InstrMem[InstrAddr];
50     Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
51     Instr[15:8] <= InstrMem[InstrAddr + 32'd2];
52     Instr[7:0] <= InstrMem[InstrAddr + 32'd3];
53 end
54 endmodule
```

這個 module 負責將指令從記憶體中讀取出來。在此過程中，使用的是 MIPS 的 Big Endian 排法，也就是將記憶體的最低位元存儲資料的最高位元。

RF:

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

這個 module 的功能是負責將 Rs 和 Rt 的值從暫存器中讀取出來，並將計算結果寫入 Rd 暫存器。這個 module 會根據暫存器地址進行查找。另一方面，當計算完成後，這個 module 還會將結果寫入 Rd 暫存器中，但這個寫入操作需要等待時鐘 (clk) 信號的觸發，並且必須滿足 RegWrite 信號為 1 的條件，這樣才會觸發對 Rd 暫存器的寫入。

二、波形圖

| | | | | | | |
|-------------------------|--------|----------|----------|----------|----------|------------|
| /Register_File/RsData | 3 | [10] | [2] | [3] | [55] | 4294967295 |
| /Register_File/RtData | 7 | [160] | [1] | [7] | [100] | 4294967295 |
| /Register_File/RdData | 24 | [170] | [1] | [24] | [119] | |
| /Register_File/RsAddr | 14 | [10] | [12] | [14] | [17] | 31 |
| /Register_File/RtAddr | 15 | [11] | [13] | [15] | [18] | 31 |
| /Register_File/RdAddr | 24 | [20] | [21] | [24] | [27] | 31 |
| /Register_File/RegWrite | S1 | | | | | |
| /alu_c0/Funct_ctrl | 000000 | [100001] | [100011] | [000000] | [100101] | |
| /alu_c0/ALU_op | S1 | | | | | |
| /alu_c0/Funct | 10 | [00] | [01] | [10] | [11] | |
| /alu0/Rs_data | 3 | [10] | [2] | [3] | [55] | |
| /alu0/Rt_data | 7 | [160] | [1] | [7] | [100] | |
| /alu0/shamt | 00011 | [000000] | | [00011] | [00100] | |
| /alu0/Funct | 10 | [00] | [01] | [10] | [11] | |
| /alu0/Rd_data | 24 | [170] | [1] | [24] | [119] | |

ALU 透過 ALU_control 提供的 Funct 信號來執行四種基本運算，分別是加法、減法、向左位移和邏輯或操作。加法時會將 Rs + Rt 的值送到 Rd，減法時會將 Rs - Rt 的值送到 Rd，位移時會將 Rs 位移 shamt 次，這些運算結果會先被送到 Rd，但 Rd 的值不會立即被寫入，需要等待來自 RegWrite 信號的控制。只有當 RegWrite 信號為高時，Rd 的結果才會被實際寫入到對應的暫存器內。

三、結果

| | |
|--|------------------|
| opcode rs rt rd shamt funct 000000 01010 01011 10100 00000 100001 add \$20, \$10, \$11 | 21 000000aa //+ |
| opcode rs rt rd shamt funct 000000 01100 01101 10101 00000 100011 sub \$21, \$12, \$13 | 22 00000001 //- |
| opcode rs rt rd shamt funct 000000 01110 01111 11000 00011 000000 sll \$24, \$14, \$15 | 23 00000000 |
| opcode rs rt rd shamt funct 000000 10001 10010 11011 00100 100101 or \$27, \$17, \$18 | 24 00000000 |
| | 25 00000018 //<< |
| | 26 00000000 |
| | 27 00000000 |
| | 28 00000077 // |

| | Rs_addr | Rt_addr | Rd_addr | Rs_data | Rt_data | Rd_data |
|---|------------|------------|------------|---------|----------------|---------|
| 1 | 01010 (10) | 01011 (11) | 10100 (20) | 10 | 160 | 170 |
| 2 | 01100 (12) | 01101 (13) | 10101 (21) | 2 | 1 | 1 |
| 3 | 01110 (14) | 01111 (15) | 11000 (24) | 3 | 7 (don't care) | 24 |
| 4 | 10001 (17) | 10010 (18) | 11011 (27) | 55 | 100 | 119 |

這個結果基本上符合我的預期。上面最左邊那張圖是我用 Python 寫出來的指令翻譯器，這個程式的功能是將 32-bit 二進制指令轉換成對應的組合語言指令，之後的測試也會用到這個程式，從功能的角度來看，這個指令翻譯的結果和我在波形分析部分的解釋完全一致。

貳、Part2

一、程式碼

I_FormatCPU:

```
29 module I_FormatCPU(
30     // Outputs
31     output wire [31:0] Output_Addr,
32     // Inputs
33     input wire [31:0] Input_Addr,
34     input wire clk
35 );
36     wire [31:0] Instruction;
37
38     wire Reg_w, Reg_dst, ALU_src, Mem_w, Mem_to_reg;
39     wire [1:0] Funct, ALU_op;
40
41     wire [31:0] Rs_data, Rt_data, Rd_data;
42     wire [4:0] Rd_addr;
43
44     wire [31:0] ALU_in_data, ALU_out_data;
45
46     wire [31:0] Mem_r_data;
47
48     assign Rd_addr = Reg_dst ? Instruction[15:11] : Instruction[20:16];
49     assign ALU_in_data = ALU_src ? {16'b00000000_00000000, Instruction[15:0]} : Rt_data;
50     assign Rd_data = Mem_to_reg ? Mem_r_data : ALU_out_data;
51
52     /*
53      * Declaration of Instruction Memory.
54      * CAUTION: DONT MODIFY THE NAME.
55      */
56     IM Instr_Memory(
57         // Outputs
58         .Instr(Instruction),
59         // Inputs
60         .InstrAddr(Input_Addr)
61     );
62
63     Adder ad0(
64         // Inputs
65         .Input_addr(Input_Addr),
66         // Outputs
67         .Output_addr(Output_Addr)
68     );
69
70     ALU_control alu_c0(
71         // Inputs
72         .Funct_ctrl(Instruction[5:0]),
73         // Outputs
74         .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
75         .ALU_op(ALU_op) //10:R , 01:+ , 11:|
76     );
77
78     /*
79      * Declaration of Register File.
80      * CAUTION: DONT MODIFY THE NAME.
81      */
82     RF Register_File(
83         // Outputs
84         .RsData(Rs_data),
85         .RtData(Rt_data),
86         // Inputs
87         .RdData(Rd_data),
88         .RsAddr(Instruction[25:21]),
89         .RtAddr(Instruction[20:16]),
90         .RdAddr(Rd_addr),
91         .RegWrite(Reg_w),
92         .clk(clk)
93     );
94
95     /*
96      * Declaration of Data Memory.
97      * CAUTION: DONT MODIFY THE NAME.
98      */
99     DM Data_Memory(
100        // Outputs
101        .MemReadData(Mem_r_data),
102        // Inputs
103        .MemAddr(ALU_out_data),
104        .MemWriteData(Rt_data),
105        .MemWrite(Mem_w),
106        .MemRead(Mem_r),
107        .clk(clk)
108    );
109
110     ALU alu0(
111         // Inputs
112         .Rs_data(Rs_data),
113         .Rt_data(ALU_in_data),
114         .shamt(Instruction[10:6]),
115         .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
116         // Outputs
117         .Rd_data(ALU_out_data)
118     );
119
120     control c0(
121         // Inputs
122         .Opcode(Instruction[31:26]),
123         // Outputs
124         .Reg_dst(Reg_dst),
125         .Reg_w(Reg_w),
126         .ALU_src(ALU_src),
127         .Mem_w(Mem_w),
128         .Mem_r(Mem_r),
129         .Mem_to_reg(Mem_to_reg),
130         .ALU_op(ALU_op)
131     );
132
133
134 endmodule
```

這個 module 將所有的 module 並加上一些 Mux，這些 Mux 的功能會在 control 的解說詳細講。

Adder:

```
≡ Adder.v
1 module Adder (
2     input [31:0] Input_addr,
3     output [31:0] Output_addr
4 );
5     assign Output_addr = Input_addr + 32'd4;
6 endmodule
```

功能同 Part1。

ALU_control:

```
≡ ALU_control.v
1 module ALU_control (
2     input [5:0] Funct_ctrl,
3     input [1:0] ALU_op,
4     output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5 );
6     always @(*) begin
7         case (ALU_op)
8             2'b10: begin//R type
9                 case (Funct_ctrl)
10                     6'b100001: Funct <= 2'd0;//+
11                     6'b100011: Funct <= 2'd1;//-
12                     6'b000000: Funct <= 2'd2;//<<
13                     6'b100101: Funct <= 2'd3;//|
14                     default: Funct <= 2'bz;
15                 endcase
16             end
17             2'b01:begin//I type +
18                 Funct <= 2'd0;
19             end
20             2'b11: begin//I type |
21                 Funct <= 2'd3;
22             end
23             default: Funct <= 2'bz;
24         endcase
25     end
26 endmodule
27
```

除了 R-type 指令以外，其他類型的指令（例如 I-type、J-type 等）的後 6 位（即 Funct_ctrl 字段）並不是用來直接分辨運算功能的。這是因為在這些指令中，ALU 的操作功能並不完全依賴於 Funct_ctrl 字段來確定，而是依賴於控制信號中的 ALU_op 來進行判斷。當 ALU_op 被設置時，控制信號會決定 ALU 應該執行哪種運算。

ALU:

```
≡ ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] sharmt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data
6 );
7     always @(*) begin
8         case (Funct)
9             2'd0: Rd_data <= Rs_data + Rt_data;
10            2'd1: Rd_data <= Rs_data - Rt_data;
11            2'd2: Rd_data <= Rs_data << sharmt;
12            2'd3: Rd_data <= Rs_data | Rt_data;
13            default: Rd_data <= 32'bz;
14        endcase
15    end
16 endmodule
17
```

功能同 Part1

Control:

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w, Reg_dst,
4     output reg ALU_src, Mem_w, Mem_r, Mem_to_reg,
5     output reg [1:0] ALU_op
6 );
7     always @(*) begin
8         Mem_r <= 1;//反正是Mem_to_reg在控制
9
10        case (Opcode)
11            6'b000000: begin//I type
12                Reg_w <= 1;
13                Reg_dst <= 1;
14                ALU_src <= 0;
15                Mem_w <= 0;
16                Mem_to_reg <= 0;
17                ALU_op <= 2'b10;
18            end
19            6'b001001:begin//I type addiu
20                Reg_w <= 1;
21                Reg_dst <= 0;
22                ALU_src <= 1;
23                Mem_w <= 0;
24                Mem_to_reg <= 0;
25                ALU_op <= 2'b01;
26        end
27        6'b101011:begin//I type sw
28            Reg_w <= 0;
29            Reg_dst <= 0;//可以不管
30            ALU_src <= 1;
31            Mem_w <= 1;
32            Mem_to_reg <= 0;//可以不管
33            ALU_op <= 2'b01;
34        end
35
36    6'b100011:begin//I type lw
37        Reg_w <= 1;
38        Reg_dst <= 0;
39        ALU_src <= 1;
40        Mem_w <= 0;
41        Mem_to_reg <= 1;
42        ALU_op <= 2'b01;
43
44    6'b001101:begin//I type ori
45        Reg_w <= 1;
46        Reg_dst <= 0;
47        ALU_src <= 1;
48        Mem_w <= 0;
49        Mem_to_reg <= 0;
50        ALU_op <= 2'b11;
51
52    default: begin//反正不要write就好
53        Mem_w <= 0;
54        Reg_w <= 0;
55
56    endcase
57 endmodule
```

這個 module 是用來控制整個 CPU 的控制訊號，並且依據指令的前 6 位 (Opcode) 來決定應該啟動哪些功能。首先，Reg_dst 信號是用來將 Rt 的地址導入到 Rd 的地址。這個控制信號的設計是為了處理某些指令的特殊需求，例如 addiu 指令。這類指令會將 Rs 與立即數相加，並將結果存入 Rt 暫存器，而不是傳統的 R-type 指令那樣將結果存入 Rd。因此，這個 Reg_dst 控制信號允許 Rt 暫存器的地址被指定為 Rd 暫存器的地址，確保正確的結果能夠存儲。

接著，ALU_src 信號則是用來控制 ALU 的輸入。這個信號的作用是將立即數作為 ALU 的輸入之一。由於有些指令（例如 addiu）需要將立即數與 Rs 的值相加，而不是使用 Rt 暫存器的值，這時候 ALU_src 信號就會將立即數傳入 ALU 的一個輸入端，以完成加法運算或其他邏輯運算。

Mem_to_reg 信號則用來控制從記憶體讀取的數據是否取代 ALU 的計算結果。當執行 lw (load word) 指令時，數據會從記憶體中讀取並存入暫存器，而這時候 Mem_to_reg 信號會被設為高，讓 CPU 將 Mem 中的讀取值送入暫存器，而不是 ALU 的計算結果。

IM:

```
35 module IM(
36     // Outputs
37     output reg [31:0] Instr,
38     // Inputs
39     input [31:0] InstrAddr
40 );
41
42 /*
43  * Declaration of instruction memory.
44  * CAUTION: DONT MODIFY THE NAME AND SIZE.
45  */
46 reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
47
48 always @(*) begin
49     Instr[31:24] <= InstrMem[InstrAddr];
50     Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
51     Instr[15:8]  <= InstrMem[InstrAddr + 32'd2];
52     Instr[7:0]   <= InstrMem[InstrAddr + 32'd3];
53 end
54 endmodule
```

功能同 Part1

RF:

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

功能同 Part1

DM:

```
36 module DM(
37     // Outputs
38     output [31:0] MemReadData,
39     // Inputs
40     input [31:0] MemAddr,
41     input [31:0] MemWriteData,
42     input MemWrite, MemRead, clk
43 );
44
45 /*
46  * Declaration of data memory.
47  * CAUTION: DONT MODIFY THE NAME AND SIZE.
48  */
49 reg [7:0]DataMem[0:DATA_MEM_SIZE - 1];
50
51 assign MemReadData = MemRead ? {DataMem[MemAddr], DataMem[MemAddr+32'd1], DataMem[MemAddr+32'd2], DataMem[MemAddr+32'd3]} : 32'bz;
52
53 always @(posedge clk) begin
54     if (MemWrite) begin
55         DataMem[MemAddr] <= MemWriteData[31:24];
56         DataMem[MemAddr+32'd1] <= MemWriteData[23:16];
57         DataMem[MemAddr+32'd2] <= MemWriteData[15:8];
58         DataMem[MemAddr+32'd3] <= MemWriteData[7:0];
59     end
60     else;
61 end
62
63 endmodule
```

這個 module 的功能是用來將資料寫入或讀出記憶體。資料的寫入和讀出都遵循 Big Endian 排法，這意味著資料的最高位元組會存放在記憶體的最低位元地址。此外，這個 module 與 RF（寄存器檔案）相似，也需要等到時鐘信號（clk）和 MemWrite 控制信號的觸發，才能進行寫入操作。

二、波形圖

| | | | | | | | | |
|-------------------------|--------|----------|--------|--------|--------|------------|-----|------------|
| /ALU_in_data | 20 | (9) | 3 | | 20 | 65535 | | |
| /Register_File/RsData | 10 | (10) | 2 | | 10 | 4294967295 | | |
| /Register_File/RdData | 30 | (0) | 19 | 65535 | 19 | 0 | 130 | 4294967295 |
| /Register_File/RdData | 30 | (19) | 5 | 19 | 30 | | | 4294967295 |
| /Register_File/RsAddr | 10 | (10) | 12 | | 10 | | | 31 |
| /Register_File/RAddr | 28 | (25) | | 27 | 28 | | | 31 |
| /Register_File/RdAddr | 28 | (25) | | 27 | 28 | | | 31 |
| /Register_File/RegWrite | St1 | | | | | | | |
| /Register_File/dlk | St1 | | | | | | | |
| Data_Memory/dlk | St1 | | | | | | | |
| c0/Opcode | 001101 | (001001) | 101011 | 100011 | 001101 | 111111 | | |
| c0/Reg_w | 1 | | | | | | | |
| c0/Reg_dst | 0 | | | | | | | |
| c0/ALU_src | 1 | | | | | | | |
| c0/Mem_w | 0 | | | | | | | |
| c0/Mem_r | 1 | | | | | | | |
| c0/Mem_to_reg | 0 | | | | | | | |
| c0/ALU_op | 11 | (01) | | | 11 | | | |

在這些指令中，ALU_in_data 的輸入全部來自立即值，因此 ALU_src 控制信號會一直處於高電位。這是因為在這些指令中，ALU 的運算必須用到立即數，無論是進行加法還是記憶體位址運算，都需要將立即數作為 ALU 的輸入之一。

Reg_dst 信號除了在 sw (store word) 指令中是 don't care，在其他三個指令中，Reg_dst 都會保持為低電位。這是因為在 addiu、lw、ori 這些指令中，都需要將 Rt 的值導入到 Rd，確保 ALU 的結果能夠正確存儲到 Rt。

Mem_to_reg 控制信號只有在 lw (load word) 指令中才會為高電位，因為 lw 指令需要從記憶體讀取數據並將其存儲到寄存器中。對於 addiu、sw、ori 三個指令，Mem_to_reg 會保持低電位，因為這些指令的結果是來自 ALU 運算，不需要讀取記憶體的數據。

在這四條指令中，ALU 的操作也有所區別。對於前三條指令 addiu、sw、lw，ALU 執行的是 $Rs +$ 立即值，進行加法運算或邏輯運算。而在 ori 指令中，ALU 會執行 $Rs |$ 立即值。

三、結果

```
opcode rs rd immidiate  
001001 01010 11001 000000000001001  
addi $25, $10, 9  
  
opcode rs rd immidiate  
101011 01100 11001 0000000000000011  
sw $25, $12, 3  
  
opcode rs rd immidiate  
100011 01100 11011 0000000000000011  
lw $27, $12, 3  
  
opcode rs rd immidiate  
001101 01010 11100 0000000000010100  
ori $28, $10, 20  
  
testbench > DM.out  
1 ff  
2 ff  
3 ff  
4 ff  
5 ff  
6 00  
7 00  
8 00  
9 13 //sw  
10 ff  
25 0000000  
26 00000013 //addiu  
27 0000000  
28 00000013 //lw  
29 0000001e //ori  
30 0000000  
31 ffffffff  
32 ffffffff
```

| | Rs_addr | Rt_addr | immidiate | Rs_data | ALU_out_data | Rt_data |
|---|------------|------------|-----------|---------|--------------|-------------|
| 1 | 01010 (10) | 11001 (25) | 9 | 10 | 19 | 0 -> 19 |
| 2 | 01100 (12) | 11001 (25) | 3 | 2 | 5 | 19 |
| 3 | 01100 (12) | 11011 (27) | 3 | 2 | 5 | 65535 -> 19 |
| 4 | 01010 (10) | 11100 (28) | 20 | 10 | 30 | 0 -> 30 |

這個結果基本符合我的預期。在 sw (store word) 指令中，資料會被存入從指定地址開始的四個 8-bit 記憶體單位中。在這條指令中，記憶體位置的計算結果是 5。因此資料會從第 5 個記憶體單位開始，根據 Big Endian 排法依次存放四個 8-bit 的資料。

除了 sw 指令之外，其他三條指令 addiu、lw、ori 的處存位置跟結果也與我預期的一樣。

參、Part3

一、程式碼

SimpleCPU:

```
29 module SimpleCPU(
30     // Outputs
31     output wire [31:0] Output_Addr,
32     // Inputs
33     input  wire [31:0] Input_Addr,
34     input  wire          clk
35 );
36     wire [31:0] Instruction;
37
38     wire Reg_w, Reg_dst, ALU_src, Mem_w, Mem_r, Mem_to_reg, Zero, Branch, Jump;
39     wire [1:0] Funct, ALU_op;
40
41     wire [31:0] Rs_data, Rt_data, Rd_data;
42     wire [4:0] Rd_addr;
43
44     wire [31:0] ALU_in_data, ALU_out_data, Signed_Extend;
45
46     wire [31:0] Mem_r_data;
47
48     wire [31:0] Addr_p4, Addr_p4_pi, Addr_p4_or_p4_pi;
49
50     assign Rd_addr = Reg_dst ? Instruction[15:11] : Instruction[20:16];
51     assign ALU_in_data = ALU_src ? {16'b00000000_00000000, Instruction[15:0]} : Rt_data;
52     assign Rd_data = Mem_to_reg ? Mem_r_data : ALU_out_data;
53     assign Addr_p4_or_p4_pi = (Branch & Zero) ? Addr_p4_pi : Addr_p4;
54     assign Output_Addr = Jump ? {Addr_p4[31:28], Instruction[25:0], 2'b00} : Addr_p4_or_p4_pi;
55
56 /*
57  * Declaration of Instruction Memory.
58  * CAUTION: DONT MODIFY THE NAME.
59  */
60 IM Instr_Memory(
61     // Outputs
62     .Instr(Instruction),
63     // Inputs
64     .InstrAddr(Input_Addr)
65 );
```

接下一页

```

72  RF Register_File(
73    // Outputs
74    .RsData(Rs_data),
75    .RtData(Rt_data),
76    // Inputs
77    .RdData(Rd_data),
78    .RsAddr(Instruction[25:21]),
79    .RtAddr(Instruction[20:16]),
80    .RdAddr(Rd_addr),
81    .RegWrite(Reg_w),
82    .clk(clk)
83  );
84
85  /*
86   * Declaration of Data Memory.
87   * CAUTION: DONT MODIFY THE NAME.
88   */
89  DM Data_Memory(
90    // Outputs
91    .MemReadData(Mem_r_data),
92    // Inputs
93    .MemAddr(ALU_out_data),
94    .MemWriteData(Rt_data),
95    .MemWrite(Mem_w),
96    .MemRead(Mem_r),
97    .clk(clk)
98  );
99
100 Adder ad0(
101   // Inputs
102   .Input_1_adder(Input_Addr),
103   .Input_2_adder(32'd4),
104   // Outputs
105   .Output_adder(Addr_p4)
106 );
107
108 Adder ad1(
109   // Inputs
110   .Input_1_adder(Addr_p4),
111   .Input_2_adder({14'b00000000_00000 , Instruction[15:0], 2'b00}),
112   // Outputs
113   .Output_adder(Addr_p4_pi)
114 );

```

```

116  ALU_control alu_c0(
117    // Inputs
118    .Funct_ctrl(Instruction[5:0]),
119    // Outputs
120    .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
121    .ALU_op(ALU_op) //10:R , 01:+ , 11:|
122  );
123
124  ALU alu0(
125    // Inputs
126    .Rs_data(Rs_data),
127    .Rt_data(ALU_in_data),
128    .shamt(Instruction[10:6]),
129    .Funct(Funct), //0:+ , 1:- , 2:<< , 3:|
130    // Outputs
131    .Rd_data(ALU_out_data),
132    .Zero(Zero)
133  );
134
135  control c0(
136    // Inputs
137    .Opcode(Instruction[31:26]),
138    // Outputs
139    .Reg_dst(Reg_dst),
140    .Reg_w(Reg_w),
141    .ALU_src(ALU_src),
142    .Mem_w(Mem_w),
143    .Mem_r(Mem_r),
144    .Mem_to_reg(Mem_to_reg),
145    .Branch/Branch,
146    .Jump(Jump),
147    .ALU_op(ALU_op)
148  );
149
150 endmodule

```

這個 module 是將全部的 module 連接起來，然後加上變更路徑的 Mux。Jump 為高電位時，將下一指令的位

址變更為 PC 得最高 4 bit 加上乘 4 後的立即值。Branch 和 zero 的訊號若同時高電位意味著需要將下一筆指令的位址加上乘 4 後的立即值。

Adder:

```

Adder.v
1  module Adder (
2    input [31:0] Input_1_adder, Input_2_adder,
3    output [31:0] Output_adder
4  );
5    assign Output_adder = Input_1_adder + Input_2_adder;
6  endmodule

```

將兩個 32 bit 作半加法，因為 jump 需要加立即值和下一個指令的地址所以將 +4 改成輸入端子。

ALU_control:

```
ALU_control.v
1 module ALU_control (
2     input [5:0] Funct_ctrl,
3     input [1:0] ALU_op,
4     output reg [1:0] Funct //0:+ , 1:- , 2:<< , 3:|
5 );
6     always @(*) begin
7         if (ALU_op[1]) begin
8             if (ALU_op[0]) Funct <= 2'd3;//ALU_op:11, |
9             else begin//ALU_op:10, R type
10                 if (Funct_ctrl[0]) begin
11                     if (Funct_ctrl[1]) Funct <= 2'd1; //-
12                     else begin
13                         if (Funct_ctrl[2]) Funct <= 2'd3;//|
14                         else Funct <= 2'd0;//+
15                     end
16                 end
17                 else Funct <= 2'd2;//<<
18             end
19         end
20         else begin
21             if (ALU_op[0]) Funct <= 2'd0;//ALU_op:01, +
22             else Funct <= 2'd1;//ALU_op:00, -
23         end
24     end
25 endmodule
```

功能同 Part2，只是將判斷式更改成判斷關鍵的 bit 以減少面積。

ALU:

```
ALU.v
1 module ALU (
2     input [31:0] Rs_data, Rt_data,
3     input [4:0] shamt,
4     input [1:0] Funct, //0:+ , 1:- , 2:<< , 3:|
5     output reg [31:0] Rd_data,
6     output Zero
7 );
8     assign Zero = (Rd_data == 32'd0) ? 1 : 0;
9
10    always @(*) begin
11        if (Funct[1]) begin
12            if (Funct[0]) Rd_data = Rs_data | Rt_data;//Funct == 3
13            else Rd_data = Rs_data << shamt;//Funct == 2
14        end
15        else begin
16            if (Funct[0]) Rd_data = Rs_data - Rt_data;//Funct == 1
17            else Rd_data = Rs_data + Rt_data;//Funct == 0
18        end
19    end
20 endmodule
```

功能同 Part1，只是將判斷式更改成判斷關鍵的 bit 以減少面積。

Control:

```
control.v
1 module control (
2     input [5:0] Opcode,
3     output reg Reg_w, Reg_dst,
4     output reg ALU_src, Mem_w, Mem_r, Mem_to_reg, Branch, Jump,
5     output reg [1:0] ALU_op
6 );
7     always @(*) begin
8         Mem_r <= 1'b1;//反正是Mem_to_reg在控制
9
10        if (Opcode[4]) begin //illgel
11            Mem_w <= 1'b0;
12            Reg_w <= 1'b0;
13            Jump <= 1'b0;
14            Branch <= 1'b0;
15        end
16        else if (Opcode[0]) begin //I type
17            Reg_dst <= 1'b0;
18            ALU_src <= 1'b1;
19            Branch <= 1'b0;
20            Jump <= 1'b0;
21            if (Opcode[3]) begin
22                Mem_to_reg <= 1'b0;
23                if (Opcode[2]) begin //ori
24                    Reg_w <= 1'b1;
25                    Mem_w <= 1'b0;
26                    ALU_op <= 2'b11;
27                end
28                else begin
29                    ALU_op <= 2'b01;
30                    if (Opcode[1]) begin //sw
31                        Reg_w <= 1'b0;
32                        Mem_w <= 1'b1;
33                    end
34                    else begin //addi
35                        Reg_w <= 1'b1;
36                    end
37                end
38            end
39            else begin //lw
40                Reg_w <= 1'b1;
41                Mem_w <= 1'b0;
42                Mem_to_reg <= 1'b1;
43                ALU_op <= 2'b01;
44            end
45        end
46    end
47
48    else begin //R type, beq, jump
49        Reg_dst <= 1'b1;
50        ALU_src <= 1'b0;
51        Mem_w <= 1'b0;
52        Mem_to_reg <= 1'b0;
53        if (Opcode[2]) begin //beq
54            Reg_w <= 1'b0;
55            ALU_op <= 2'b00;
56            Branch <= 1'b1;
57            Jump <= 1'b0;
58        end
59        else begin
60            ALU_op <= 2'b10;
61            Branch <= 1'b0;
62            if (Opcode[1]) begin //jump
63                Reg_w <= 1'b0;
64                Jump <= 1'b1;
65            end
66            else begin //R type
67                Reg_w <= 1'b1;
68                Mem_to_reg <= 1'b0;
69                ALU_op <= 2'b10;
70                Jump <= 1'b0;
71            end
72        end
73    end
74 endmodule
```

功能基本和 Part2 一樣，只是加上了在 j 和 beq 時將 Jump 和 Branch 訊號拉成高電位而已，另外判斷式跟前面一樣式更改成指判斷關鍵的 bit 以減少面積。

IM:

```
IM.v
35 module IM(
36     // Outputs
37     output reg [31:0] Instr,
38     // Inputs
39     input [31:0] InstrAddr
40 );
41
42 /*
43  * Declaration of instruction memory.
44  * CAUTION: DONT MODIFY THE NAME AND SIZE.
45  */
46 reg [7:0] InstrMem[0:`INSTR_MEM_SIZE - 1];
47
48 always @(*) begin
49     Instr[31:24] <= InstrMem[InstrAddr];
50     Instr[23:16] <= InstrMem[InstrAddr + 32'd1];
51     Instr[15:8]  <= InstrMem[InstrAddr + 32'd2];
52     Instr[7:0]   <= InstrMem[InstrAddr + 32'd3];
53 end
54 endmodule
```

功能同 Part1

RF:

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

功能同 Part1

DM:

```
36 module DM(
37     // Outputs
38     output [31:0] MemReadData,
39     // Inputs
40     input [31:0] MemAddr,
41     input [31:0] MemWriteData,
42     input MemWrite, MemRead, clk
43 );
44
45 /*
46  * Declaration of data memory.
47  * CAUTION: DONT MODIFY THE NAME AND SIZE.
48  */
49 reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
50
51 assign MemReadData = MemRead ? {DataMem[MemAddr], DataMem[MemAddr+32'd1], DataMem[MemAddr+32'd2], DataMem[MemAddr+32'd3]} : 32'bz;
52
53 always @ (posedge clk) begin
54     if (MemWrite) begin
55         DataMem[MemAddr] <= MemWriteData[31:24];
56         DataMem[MemAddr+32'd1] <= MemWriteData[23:16];
57         DataMem[MemAddr+32'd2] <= MemWriteData[15:8];
58         DataMem[MemAddr+32'd3] <= MemWriteData[7:0];
59     end
60     else;
61 end
62
63 endmodule
```

功能同 Part2

二、波形圖

| | | | | | | | | | | | | | | |
|---------------|-----|------------|----|----|---|------------|----|---|----|------------|----|---------|---------|---------|
| RsData | 1 | 1 | 10 | 7 | 1 | | 7 | 4 | 1 | | 4 | 1 | | 4294... |
| RtData | 1 | 10 | 3 | 1 | | 7 | 3 | 1 | 4 | 3 | 1 | | 4294... | |
| RdData | 0 | 4294967287 | 7 | 4 | 1 | 4294967290 | 4 | 1 | | 4294967293 | 1 | 4294... | 1 | 0 |
| RsAddr | 0 | 0 | 19 | 0 | | 19 | 0 | | | 19 | 0 | | 31 | |
| RtAddr | 19 | 19 | 2 | 0 | | 19 | 2 | 0 | 19 | 2 | 0 | | 31 | |
| RdAddr | 0 | 0 | 19 | 0 | | 19 | 0 | | | 19 | 0 | | 31 | |
| RegWrite | St0 | | | | | | | | | | | | | |
| /clk | St1 | | | | | | | | | | | | | |
| /alu0/Rs_data | 1 | 1 | 10 | 7 | 1 | | 7 | 4 | 1 | | 4 | 1 | | 4294... |
| /alu0/Rt_data | 1 | 10 | 3 | 1 | | 7 | 3 | 1 | 4 | 3 | 1 | | 4294... | |
| /alu0/shamt | 0 | 0 | | | | | | | | | | | 31 | |
| /alu0/Funct | 01 | 01 | 10 | 01 | | 10 | 01 | | | 10 | 01 | | 10 | |
| /alu0/Rd_data | 0 | 4294967287 | 7 | 4 | 1 | 4294967290 | 4 | 1 | | 4294967293 | 1 | 4294... | 1 | 0 |
| /alu0/Zero | St1 | | | | | | | | | | | | | |

這個部分處理的是 beq、sub、j 這三個指令的程式循環。這些指令會一直執行，直到 64 被減到 1，此時會進行 branch 操作，跳轉到最後一條指令。當暫存器 19 的值被減至 1 時，Zero 信號會被拉高，這樣就會觸發 branch 指令。如果 Zero 信號沒有被拉高，則繼續執行下一條指令，不會進行跳轉。

三、結果

```
opcode rs rd immediate
000100 00000 10011 0000000000011110
beq $19, $0, 30

opcode rs rt rd shamt funct
000000 10011 00010 10011 00000 100011
sub $19, $19, $2

opcode immediate
000010 00000000000000000000000000
j 0
```

| | |
|----|------------------------|
| 16 | 00000007 |
| 17 | 00000002 |
| 18 | 00000037 |
| 19 | 00000064 |
| 20 | 00000001 //beq, sub, j |
| 21 | 00000000 |
| 22 | 00000000 |
| 23 | 00000000 |
| 24 | 00000000 |

這個結果基本符合我的預期。開始時，暫存器 19 的初始值為 64，暫存器 2 存儲的是 3，而暫存器 0 則存儲 1。根據設計，第一條指令會檢查暫存器 19 的值是否等於 1。如果值不等於 1，則會執行暫存器 19 減去 3 的操作，然後跳轉回到指令的開始處，重新執行這個過程。這樣的循環會持續進行，直到暫存器 19 的值減到 1 為止。當暫存器 19 的值變為 1 時，beq 指令將會執行，並且進行跳轉到後面的指令，結束這個循環。

肆、自己的 testbench

```
2 10 // Addr = 0x00 //beq
3 13 // Addr = 0x01
4 00 // Addr = 0x02
5 02 // Addr = 0x03
6 02 // Addr = 0x04
7 62 // Addr = 0x05
8 98 // Addr = 0x06
9 23 // Addr = 0x07
10 08 // Addr = 0x08
11 00 // Addr = 0x09
12 00 // Addr = 0x0A
13 00 // Addr = 0x0B
14 01 // Addr = 0x0C //r type
15 4B // Addr = 0x0D
16 A0 // Addr = 0x0E
17 21 // Addr = 0x0F
18 01 // Addr = 0x10
19 8D // Addr = 0x11
20 A8 // Addr = 0x12
21 23 // Addr = 0x13
22 01 // Addr = 0x14
23 CF // Addr = 0x15
24 C0 // Addr = 0x16
25 C0 // Addr = 0x17
26 02 // Addr = 0x18
27 32 // Addr = 0x19
28 D1 // Addr = 0x1A
29 25 // Addr = 0x1B
30 25 // Addr = 0x1C //i type
31 59 // Addr = 0x1D
32 00 // Addr = 0x1E
33 09 // Addr = 0x1F
34 AD // Addr = 0x20
35 99 // Addr = 0x21
36 00 // Addr = 0x22
37 03 // Addr = 0x23
38 8D // Addr = 0x24
39 9B // Addr = 0x25
40 00 // Addr = 0x26
41 03 // Addr = 0x27
42 35 // Addr = 0x28
43 5C // Addr = 0x29
44 00 // Addr = 0x2A
45 14 // Addr = 0x2B
46 FF // Addr = 0x2C //over
47 FF // Addr = 0x2D
48 FF // Addr = 0x2E
49 FF // Addr = 0x2F
```

opcode rs rd immidiate
000100 00000 10011 0000000000000000
beq \$19, \$0, 2

opcode rs rt rd shamrt funct
000000 10011 00010 10011 00000 100011
sub \$19, \$19, \$2

opcode immidiate
000010 000000000000000000000000000000
j 0

opcode rs rt rd shamrt funct
000000 01010 01011 10100 00000 100001
add \$20, \$10, \$11

opcode rs rt rd shamrt funct
000000 01100 01101 10101 00000 100011
sub \$21, \$12, \$13

opcode rs rt rd shamrt funct
000000 01110 01111 11000 00011 000000
sll \$24, \$14, \$15

opcode rs rt rd shamrt funct
000000 10001 10010 11010 00100 100101
or \$26, \$17, \$18

opcode rs rd immidiate
001001 01010 11001 00000000001001
addi \$25, \$10, 9

opcode rs rd immidiate
101011 01100 11001 00000000000001
sw \$25, \$12, 3

opcode rs rd immidiate
100011 01100 11011 00000000000001
lw \$27, \$12, 3

opcode rs rd immidiate
001101 01010 11100 000000000010100
ori \$28, \$10, 20

我把三個部分的城市都跑過一遍，確認是否有出現錯誤。裡面有一些指令位置我有稍微改一下，讓他結果輸出都在不同的暫存器，以方便更好的觀察是否有錯誤。

| testbench > ≡ RF.out | | testbench > ≡ DM.out | |
|----------------------|--------------------|----------------------|----------------------------|
| 2 | 0000_0001 // R[0] | 1 | 00000001 |
| 3 | 0000_0001 // R[1] | 2 | 00000001 |
| 4 | 0000_0003 // R[2] | 3 | 00000003 |
| 5 | 7777_7777 // R[3] | 4 | 77777777 |
| 6 | 7F7F_7F7F // R[4] | 5 | 7f7f7f7f |
| 7 | F7F7_F7F7 // R[5] | 6 | f7f7f7f7 |
| 8 | 7FFF_FFFF // R[6] | 7 | 7fffffff |
| 9 | 8000_0000 // R[7] | 8 | 80000000 |
| 10 | FFFF_0000 // R[8] | 9 | ffff0000 |
| 11 | 0000_FFFF // R[9] | 10 | 0000ffff |
| 12 | 0000_000A // R[10] | 11 | 0000000a |
| 13 | 0000_00A0 // R[11] | 12 | 000000a0 |
| 14 | 0000_0002 // R[12] | 13 | 00000002 |
| 15 | 0000_0001 // R[13] | 14 | 00000001 |
| 16 | 0000_0003 // R[14] | 15 | 00000003 |
| 17 | 0000_0007 // R[15] | 16 | 00000007 |
| 18 | 0000_0002 // R[16] | 17 | 00000002 |
| 19 | 0000_0037 // R[17] | 18 | 00000037 |
| 20 | 0000_0064 // R[18] | 19 | 00000064 |
| 21 | 0000_0040 // R[19] | 20 | 00000001 //beq and jump re |
| 22 | 0000_0000 // R[20] | 21 | 000000aa //r type + re |
| 23 | 0000_0000 // R[21] | 22 | 00000001 //r type - re |
| 24 | 0000_0000 // R[22] | 23 | 00000000 |
| 25 | 0000_0000 // R[23] | 24 | 00000000 |
| 26 | 0000_0000 // R[24] | 25 | 00000018 //r type << re |
| 27 | 0000_0000 // R[25] | 26 | 00000013 //i type +i re |
| 28 | 0000_0000 // R[26] | 27 | 00000077 //r type re |
| 29 | 0000_0000 // R[27] | 28 | 00000013 //i type lw re |
| 30 | 0000_0000 // R[28] | 29 | 0000001e //i type i re |
| 31 | 0000_0000 // R[29] | 30 | 00000000 |
| 32 | FFFF_FFFF // R[30] | 31 | ffffffff |
| 33 | FFFF_FFFF // R[31] | 32 | ffffffff |

這些結果都跟我預想的一樣。

伍、如果想要將 PA1 加進來

我認為是沒辦法直接加進來的，因為 PA1 是將乘法跟除法用 shift 的方式分 32 個 cycle 來計算，而 PA2 是 single cycle 的計算，所以要直接用是沒有辦法的。但是如果要加我會直接加在 ALU 裏面，時序的部分可能還需要再設計，或是把它設計成組合電路，但是這樣就要看到底是一般的乘法器器還是 PA1 的功能比較好了。

陸、結論

這次的作業我覺得跟上一個作業的難度是差不多的，上一個作業的重點是需要設計好時序的關係，而這次則是需要設計好 module 之間的資料流動。而且，由於上個作業同一個 clock 做的事情較少，所以在面積和速度上都表現得比較好；而這次則是相反，因為資料流動比較複雜，涉及更多的運算與控制，因此面積可能會比較大，速度則會有不同的挑戰。我覺得這次的作業我有稍微遇到的問題是在合成過程中，尤其是 DM 在合成時面積過大，甚至達到一個誇張的程度，最終導致出現了 DRC 錯誤。後來，助教告訴我其實如果縮小後還能通過，則這樣的設計是可行的，並且實際上，在之後打分數時的合成中大家都會使用相同的 DM 來進行測試。