國 立 台 灣 科 技 大 學

計 算 機 組 織

指 導 教 授：陳雅淑 教 授

計算機組織作業PA2報告

班 級　　：　四電機三甲
學 生　　：　呂佳祐
學 號　　：　B11107055
SimpleCPU ：　Area: 33892.922、Slack: 2.0369

1. Instruction Memory

指令記憶體用於儲存程式的指令，由總共 128 格、大小為 1byte 的空間組成。

```
1  `define INSTR_MEM_SIZE  128 // Bytes
2  module IM(
3      // Outputs
4      output reg [31:0] Instr,
5      // Inputs
6      input wire [31:0] InstrAddr
7  );
8      reg [7:0]InstrMem[0:`INSTR_MEM_SIZE - 1];
9      always @(*) begin
10         Instr = {
11             InstrMem[InstrAddr],
12             InstrMem[InstrAddr+1],
13             InstrMem[InstrAddr+2],
14             InstrMem[InstrAddr+3]
15         };
16     end
17 endmodule
```

IM.v 程式碼

記憶體透過輸入指定的記憶體位置 InstrAddr，輸出從該位址算起的後四個空間組成的指令，總共 4 個 bytes。本作業使用的是 MIPS 架構，資料排序為 Big endian，所以在輸出的值中，最高位元會是最上層位址空間的資料，而最低位元則是最下層位址空間的資料。

2. Register File

```
1  `define REG_MEM_SIZE    32  // Words
2  module RF(
3      // Outputs
4      output reg [31:0] RsData,
5      output reg [31:0] RtData,
6      // Inputs
7      input wire [4:0] RsAddr,
8      input wire [4:0] RtAddr,
9      input wire [4:0] RdAddr,
10     input wire [31:0] RdData,
11     input wire RegWrite,
12     input wire clk
13 );
14     reg [31:0]R[0:`REG_MEM_SIZE - 1];
15     always@(negedge clk) begin
16         if(RegWrite && RdAddr != 5'h0) begin
17             R[RdAddr] <= RdData;
18         end
19     end
20     always @(RsAddr, RtAddr, RdAddr) begin
21         RsData = R[RsAddr];
22         RtData = R[RtAddr];
23     end
24 endmodule
```

RF.v 程式碼

暫存器堆由 32 個大小為 4 個 bytes 的暫存器組成，為運算資料暫時保存的地方。若要寫入資料，則資料會在時鐘上緣時被存入由 RdAddr 指定位址的暫存器儲存。若要讀取資料，則透過輸入的 RsAddr 及 RtAddr，來讀取該位分別對應的 Source Register 及 Target Register，讀取資料與時鐘無關。

3. Data Memory

```
`define DATA_MEM_SIZE   128 // Bytes
module DM(
    // Outputs
    output reg [31:0] MemReadData,
    // Inputs
    input wire [31:0] MemAddr,
    input wire [31:0] MemWriteData,
    input wire MemWrite,
    input wire MemRead,
    input wire clk
);
    reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];
    always @(negedge clk) begin
        if(MemWrite) begin
            DataMem[MemAddr] <= MemWriteData[31:24];
            DataMem[MemAddr+1] <= MemWriteData[23:16];
            DataMem[MemAddr+2] <= MemWriteData[15:8];
            DataMem[MemAddr+3] <= MemWriteData[7:0];
        end
    end
    always @(posedge clk) begin
        if(MemRead) MemReadData = {DataMem[MemAddr], DataMem[MemAddr+1], DataMem[MemAddr+2], DataMem[MemAddr+3]};
    end
endmodule
```

DM.v 程式碼

資料記憶體用於儲存相比於暫存器，更不須被立馬運算，且資料保存時間較長。其由 128 格大小為 1 個 Byte 的空間組成。

資料記憶體在時鐘上緣時輸出資料，且在時鐘下緣時讀取輸入資料。不管是輸出還是輸入資料，都是 Big endian。

4. Adder

```
module Adder(
    output reg [31:0] Adder_result,
    input wire [31:0] Input1,
    input wire [31:0] Input2
);
    always @(*) begin
        Adder_result = Input1 + Input2;
    end
endmodule
```

Adder.v 程式碼

加法器用於計算下一個指令或是 beq 成立跳躍的位址。

5. ALU

```
module ALU(
    output reg [31:0] ALU_result,
    output reg Zero,
    input wire [31:0] Rs_data,
    input wire [31:0] Second_data,
    input wire [4:0] Shamt,
    input wire [5:0] Funct
);
    assign ALU_result = 32'b0;
    always @(*) begin
        case(Funct)
            6'b001001: ALU_result = Rs_data + Second_data;
            6'b001010: ALU_result = Rs_data - Second_data;
            6'b100001: ALU_result = Rs_data << Shamt;
            6'b100101: ALU_result = Rs_data | Second_data;
            default: ALU_result = 32'b0;
        endcase
        Zero = (ALU_result == 32'b0);
    end
endmodule
```

ALU.v 程式碼

ALU 負責處理兩輸入資料的運算，且運算功能會由輸入的 Funct 指定。若當運算結果為 0 時，Zero 會被設為 1，用於判斷 beq 指令是否成立。各 Funct 對應功能如下：

Funct Code	Corresponding Function
001001	Addition
001010	Substraction
100001	Shift Left Logical
100101	Or

6. ALU Controller

```
module ALU_Control(
    output reg [5:0] Funct,
    input wire [5:0] Funct_ctrl,
    input wire [1:0] ALU_op
);
    always @(*) begin
        // R-type
        if(ALU_op == 2'b10) begin
            case(Funct_ctrl)
                6'b100001: Funct = 6'b001001; // Add
                6'b100011: Funct = 6'b001010; // Sub
                6'b000000: Funct = 6'b100001; // SLL
                6'b100101: Funct = 6'b100101; // Or
                default: Funct = 6'b000000;
            endcase
        end
        // I-type, J-type (Don't care about Funct_ctrl)
        else if(ALU_op == 2'b00) Funct = 6'b001001; // Add
        else if(ALU_op == 2'b01) Funct = 6'b001010; // Sub
        else if(ALU_op == 2'b11) Funct = 6'b100101; // Or
        else Funct = 6'b000000;
    end
endmodule
```

ALU_Control.v 程式碼

MIPS 架構中，程式指令分為三個類型，分別是 R-type、I-type 及 J-type，為使 ALU 功能簡單，所以處理分別指令類型的邏輯判斷，由 ALU_Control 及 Control 負責。

在 ALU_Control 中，決定要使用什麼運算功能由兩個輸入資料決定，分別是在 R-type 的 Instruction 裡的 Funct_ctrl，以及由 Control 輸出的 ALU_op。對應功能如下：

Instruction Type	ALU_op	Funct_ctrl	Correspoding function	MIPS Instruction
R-type	10	100001	Add unsigned	addu
		100011	Sub unsigned	subu
		000000	Shift Left Logic	sll
		100101	Or	or
I-type、J-type	00	Don't care	Add imm unsigned	addiu, sw, lw
	01		Sub imm unsigned	beq, j
	11		Or imm	ori

7. Controller

```
module Control(
    output reg Reg_dst, Branch, Reg_w, ALU_src, Mem_w, Mem_r, Mem_to_reg, Jump,
    output reg [1:0] ALU_op,
    input reg [5:0] OpCode
);
    always @(*) begin
        // All reset to 0
        Reg_dst    = 0;
        ALU_src    = 0;
        ALU_op     = 2'b00;
        Mem_r      = 0;
        Mem_w      = 0;
        Mem_to_reg = 0;
        Reg_w      = 0;
        Branch     = 0;
        Jump       = 0;
        case (OpCode)
            6'b000000: begin  // R-type
                Reg_dst    = 1;
                ALU_src    = 0;
                ALU_op     = 2'b10;
                Reg_w      = 1;
            end
            6'b001001: begin  // addiu
                ALU_src    = 1;
                ALU_op     = 2'b00;
                Reg_w      = 1;
            end
            6'b100011: begin  // lw
                ALU_src    = 1;
                ALU_op     = 2'b00;
                Mem_r      = 1;
                Mem_to_reg = 1;
                Reg_w      = 1;
            end
            6'b101011: begin  // sw
                ALU_src    = 1;
                ALU_op     = 2'b00;
                Mem_w      = 1;
            end
            6'b001101: begin  // ori
                ALU_src    = 1;
                ALU_op     = 2'b11;
                Reg_w      = 1;
            end
            6'b000100: begin  // beq
                ALU_src    = 0;
                ALU_op     = 2'b01;
                Branch     = 1;
            end
            6'b000010: begin  // j
                Jump       = 1;
            end
            default: begin
                // all output reset to zero if opcode is not support
            end
        endcase
    end
endmodule

```

Control.v 程式碼

控制器猶如整顆 CPU 的指揮官，它會根據 Instruction 的 Opcode，判斷指令是什麼類型，控制各多工器的輸出、記憶體的讀寫狀態以及向 ALU Control 輸出對應的 ALU_op。

控制器會先將所有輸出設為 0，再根據 Instruction，將對應的輸出設為 1；如果 Opcode 不是規定的值，控制器會將所有輸出預設為 0。各 Opcode 對應的輸出可以參考圖上的程式碼。

8. Simple CPU

```v
module SimpleCPU(
    // Outputs
    output  reg [31:0]  Output_Addr,
    // Inputs
    input   wire    [31:0]  Input_Addr,
    input   wire            clk
);
    wire [31:0] Instruction, Rs_Data, Rt_Data;
    reg [31:0] Mem_to_Reg_Data;
    reg [4:0] Rd_Addr;
    wire Reg_w;
    IM Instr_Memory(
        // Outputs
        .Instr(Instruction),
        // Inputs
        .InstrAddr(Input_Addr)
    );
    RF Register_File(
        // Outputs
        .RsData(Rs_Data),
        .RtData(Rt_Data),
        // Inputs
        .RsAddr(Instruction[25:21]),
        .RtAddr(Instruction[20:16]),
        .RdAddr(Rd_Addr),
        .RdData(Mem_to_Reg_Data),
        .RegWrite(Reg_w),
        .clk(clk)
    );
    wire Mem_Write, Mem_Read;
    wire [31:0] ALU_src_Data, Mem_r_data;
    DM Data_Memory(
        // Outputs
        .MemReadData(Mem_r_data),
        // Inputs
        .MemAddr(ALU_src_Data),
        .MemWriteData(Rt_Data),
        .MemWrite(Mem_Write),
        .MemRead(Mem_Read),
        .clk(clk)
    );
    wire [31:0] NextPC;
    Adder Plus4Adder(
        // Outputs
        .Adder_result(NextPC),
        // Inputs
        .Input1(4),
        .Input2(Input_Addr)
    );
    wire [31:0] BranchPC;
    reg [31:0] SignExtendedAddr, SignExtendedShiftedAddr;
    Adder PlusAddrAdder(
        // Outputs
        .Adder_result(BranchPC),
        // Inputs
        .Input1(NextPC),
        .Input2(SignExtendedShiftedAddr)
    );
    reg [31:0] ALU_src_Mux_Data;
    wire Zero;
    wire [5:0] Funct;
    ALU ALUBlock(
        // Outputs
        .ALU_result(ALU_src_Data),
        .Zero(Zero),
        // Inputs
        .Rs_data(Rs_Data),
        .Second_data(ALU_src_Mux_Data),
        .Shamt(Instruction[10:6]),
        .Funct(Funct)
    );
    wire [1:0] ALU_op;
    ALU_Control ALUController(
        // Outputs
        .Funct(Funct),
        // Inputs
        .Funct_ctrl(Instruction[5:0]),
        .ALU_op(ALU_op)
    );
    wire Reg_dst, Branch, ALU_src, Mem_to_reg, Jump;
    Control Controller(
        // Outputs
        .Reg_dst(Reg_dst),
        .Branch(Branch),
        .Reg_w(Reg_w),
        .ALU_src(ALU_src),
        .Mem_w(Mem_Write),
        .Mem_r(Mem_Read),
        .Mem_to_reg(Mem_to_reg),
        .Jump(Jump),
        .ALU_op(ALU_op),
        // Inputs
        .OpCode(Instruction[31:26])
    );
    always @(*) begin
        Rd_Addr = (Reg_dst) ? Instruction[15:11] : Instruction[20:16];
        SignExtendedAddr = {{16{Instruction[15]}}, Instruction[15:0]};
        SignExtendedShiftedAddr = SignExtendedAddr << 2;
        ALU_src_Mux_Data = (ALU_src) ? SignExtendedAddr : Rt_Data;
        Mem_to_Reg_Data = (Mem_to_reg) ? Mem_r_data : ALU_src_Data;
        Output_Addr = (Jump) ? {NextPC[31:28], Instruction[25:0], 2'b00} : (Zero & Branch) ? BranchPC : NextPC;
    end
endmodule
```

SimpleCPU.v 程式碼

SimpleCPU 將所有模組整合，連接個模組對應腳位，並且將所有多工器、Sign Extend、位址左移兩位元在此實作。若是單純連接兩模組的導線就以 wire 宣告；若是在 SimpleCPU 裡會被賦值的導線，就會以 reg 宣告，例如多工器的輸出。

9. Testbench and Simulation

I. Program of Testbench of SimpleCPU

```
1  // Setting timescale
2  `timescale 10 ns / 1 ns
3  // Declarations
4  `define DELAY          1   // # * timescale
5  `define INSTR_SIZE     8   // bit width
6  `define INSTR_MAX      128 // bytes
7  `define INSTR_FILE     "testbench/IM.dat"
8  `define REG_SIZE       32  // bit width
9  `define REG_MAX        32  // words
10 `define REG_FILE       "testbench/RF.dat"
11 `define DATA_SIZE      8   // bit width
12 `define DATA_MAX       128 // bytes
13 `define DATA_FILE      "testbench/DM.dat"
14 `define OUTPUT_REG     "testbench/RF.out"
15 `define OUTPUT_DATA    "testbench/DM.out"
16 // Declaration
17 `define LOW     1'b0
18 `define HIGH    1'b1
```

```
19 module tb_SimpleCPU;
20     // Inputs
21     reg [31:0] Input_Addr;
22     // Outputs
23     wire [31:0] Output_Addr;
24     // Clock
25     reg clk;
26     // Testbench variables
27     reg [`INSTR_SIZE-1  :0] instrMem    [0:`INSTR_MAX-1];
28     reg [`REG_SIZE-1    :0] regMem      [0:`REG_MAX-1];
29     reg [`DATA_SIZE-1   :0] dataMem     [0:`DATA_MAX-1];
30     integer output_reg;
31     integer output_data;
32     integer i;
33     // Instantiate the Unit Under Test (UUT)
34     SimpleCPU UUT(
35         // Outputs
36         .Output_Addr(Output_Addr),
37         // Inputs
38         .Input_Addr(Input_Addr),
39         .clk(clk)
40     );
```

tb_SimpleCPU.v 程式碼－宣告變數與連接模組等前置動作

設定時間單位為 10 ns、時間精度 1 ns。定義所有常數、宣告變數並連接對應 SimpleCPU 模組的腳位。

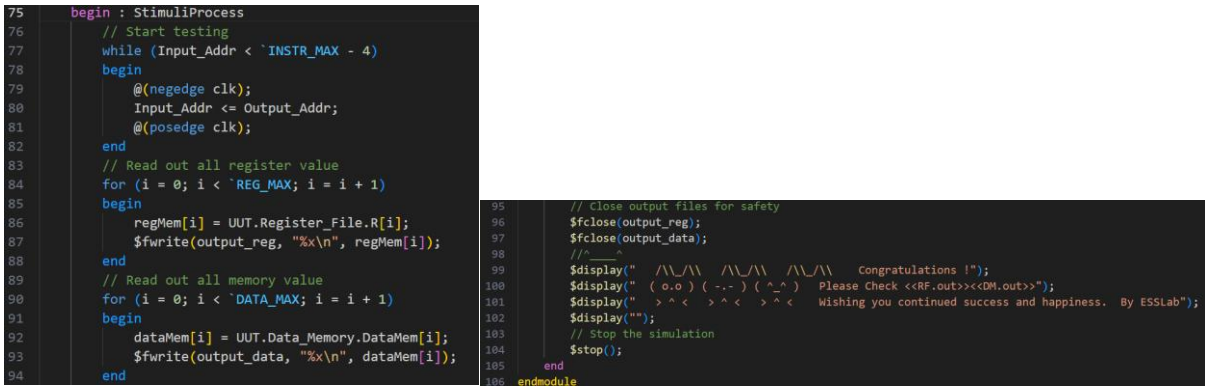
```
41     initial
42     begin : Preprocess
43         // Initialize inputs
44         Input_Addr  = 32'd0;
45         clk = `LOW;
46         // Initialize testbench files
47         $readmemh(`INSTR_FILE,  instrMem);
48         $readmemh(`REG_FILE,    regMem);
49         $readmemh(`DATA_FILE,   dataMem);
50         output_reg  = $fopen(`OUTPUT_REG);
51         output_data = $fopen(`OUTPUT_DATA);
52         // Initialize intruction memory
53         for (i = 0; i < `INSTR_MAX; i = i + 1)
54         begin
55             UUT.Instr_Memory.InstrMem[i] = instrMem[i];
56         end
57         // Initialize register file
58         for (i = 0; i < `REG_MAX; i = i + 1)
59         begin
60             UUT.Register_File.R[i] = regMem[i];
61         end
62         // Initialize data memory
63         for (i = 0; i < `DATA_MAX; i = i + 1)
64         begin
65             UUT.Data_Memory.DataMem[i] = dataMem[i];
66         end
67         #`DELAY;    // Wait for global reset to finish
68     end
```

```
70     begin : ClockGenerator
71         #`DELAY;
72         clk <= ~clk;
73     end
```

tb_SimpleCPU.v 程式碼－預處理與時鐘設定

Preprocess 區塊負責開啟 IM.dat、RF.dat、DM.dat 這三個輸入檔，並讀取其中資料，將資料分別存入 Instruction Memory、Register File、Data Memory 中。

ClockGenerator 區塊負責產生週期為 2 個時間單位的時鐘訊號。且為使等等從 IM 的第一個指令開始運作，所以在 Preprocess 中有定義時鐘訊號會先從低位元開始(Line 45: clk = `LOW;)。



```
75      begin : StimuliProcess
76          // Start testing
77          while (Input_Addr < `INSTR_MAX - 4)
78          begin
79              @(negedge clk);
80              Input_Addr <= Output_Addr;
81              @(posedge clk);
82          end
83          // Read out all register value
84          for (i = 0; i < `REG_MAX; i = i + 1)
85          begin
86              regMem[i] = UUT.Register_File.R[i];
87              $fwrite(output_reg, "%x\n", regMem[i]);
88          end
89          // Read out all memory value
90          for (i = 0; i < `DATA_MAX; i = i + 1)
91          begin
92              dataMem[i] = UUT.Data_Memory.DataMem[i];
93              $fwrite(output_data, "%x\n", dataMem[i]);
94          end
95          // Close output files for safety
96          $fclose(output_reg);
97          $fclose(output_data);
98          //^___^
99          $display("   /\\_/\\   /\\_/\\   /\\_/\\    Congratulations !");
100         $display("  ( o.o ) ( -.- ) ( ^_^ )   Please Check <<RF.out>><<DM.out>>");
101         $display("   > ^ <   > ^ <   > ^ <    Wishing you continued success and happiness.  By ESSLab");
102         $display("");
103         // Stop the simulation
104         $stop();
105     end
106 endmodule
```

tb_SimpleCPU.v 程式碼－執行 IM 指令與輸出結果

StimuliProcess 區塊開始執行 IM 裡的程式，並將本次的輸出位址設成下次執行指令的位址。程式結束後將 Register File 及 Data Memory 資料，存到 RF.out 及 DM.out 檔案中，最後在終端輸出可愛的顏文字。

```
 /\\__/\\      /\\__/\\      /\\__/\\
(O . O)      ( - . - )    ( ^ _ ^ )
 > ^ <        > ^ <        > ^ <
```

II. Test Pattern

要測試的 Instruction，以組合語言表達，且其轉換成十六進制的指令，分別註解在每行程式後：

```
1   sw $17, MEM[0x04]   // AC 11 00 04
2   sw $2, MEM[0x0C]    // AC 02 00 0C
3   lw $24, MEM[0x0C]   // 8C 18 00 8C
4   lw $25, MEM[0x04]   // 8C 19 00 04
5   addi $24, $24, 0x0C // 27 18 00 0C
6   addi $25, $24, 0xB0 // 27 19 00 B0
7   ori $24, $24, 0x0A  // 37 18 00 0A
8   sll $25, $25, 5     // 03 20 C9 40
9   J 0x0B              // 08 00 00 0B
10  sw $26, MEM[0x10]   // AC 1A 00 10
11  sw $25, MEM[0x14]   // AC 19 00 14
12  sw $24, MEM[0x18]   // AC 18 00 18
```

每行程式碼執行後，某記憶體空間或暫存器的值如下圖每行程式碼後的註解所示，最後結果也統整在 Result 區塊裡：

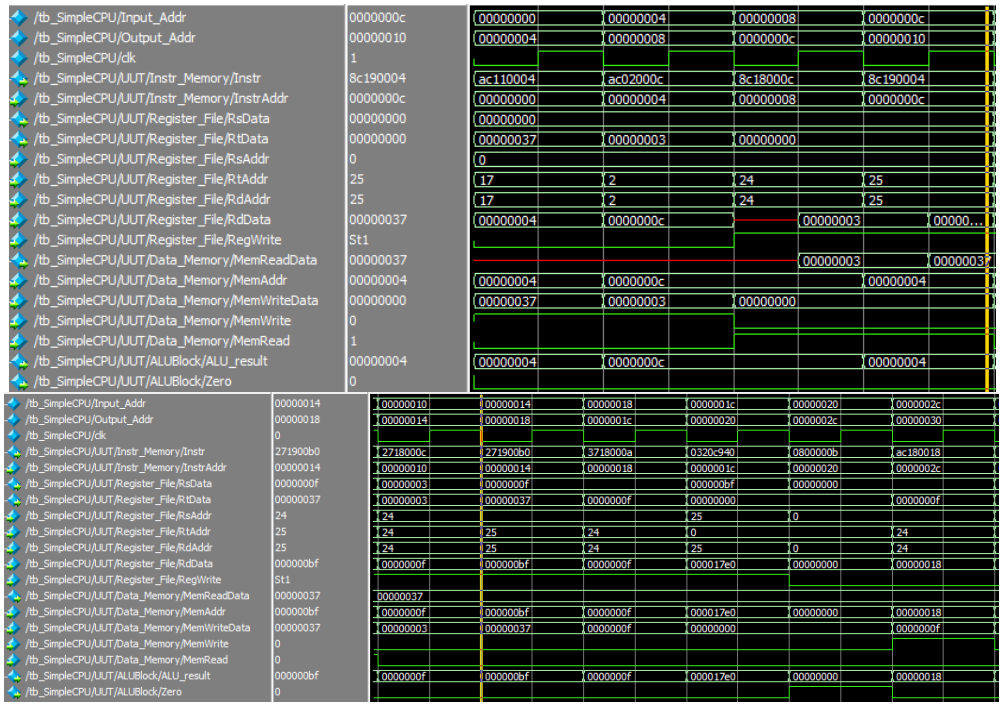
```
sw $17, MEM[0x04]   // MEM[0x04] = 0000_0037
sw $2, MEM[0x0C]    // MEM[0x0C] = 0000_0003
lw $24, MEM[0x0C]   // $24 = 0000_0003
lw $25, MEM[0x04]   // $25 = 0000_0037
addi $24, $24, 0x0C // $24 = 0000_000F
addi $25, $24, 0xB0 // $25 = 0000_00BF
ori $24, $24, 0x0A  // $24 = 0000_000F
sll $25, $25, 5     // $25 = 0000_17E0
J 0x0B              // Jump to "sw $24, MEM[0x18]"
sw $26, MEM[0x10]   // Not execute
sw $25, MEM[0x14]   // Not execute
sw $24, MEM[0x18]   // MEM[0x18] = 0000_000F

Result:
    In Register File:
        $24 = 0000_000F
        $25 = 0000_17E0
    In Data Memory:
        MEM[0x04] = 0000_0037
        MEM[0x0C] = 0000_0003
        MEM[0x18] = 0000_000F
```

把事先模擬的指令存入 IM.dat：

```
// Instruction Memory in Hex
AC  // Addr = 0x00
11  // Addr = 0x01
00  // Addr = 0x02
04  // Addr = 0x03
AC  // Addr = 0x04
02  // Addr = 0x05
00  // Addr = 0x06
0C  // Addr = 0x07
8C  // Addr = 0x08
18  // Addr = 0x09
00  // Addr = 0x0A
0C  // Addr = 0x0B
8C  // Addr = 0x0C
19  // Addr = 0x0D
00  // Addr = 0x0E
04  // Addr = 0x0F
27  // Addr = 0x10
18  // Addr = 0x11
00  // Addr = 0x12
0C  // Addr = 0x13
27  // Addr = 0x14
19  // Addr = 0x15
00  // Addr = 0x16
B0  // Addr = 0x17
37  // Addr = 0x18
18  // Addr = 0x19
00  // Addr = 0x1A
0A  // Addr = 0x1B
03  // Addr = 0x1C
20  // Addr = 0x1D
C9  // Addr = 0x1E
40  // Addr = 0x1F
08  // Addr = 0x20
00  // Addr = 0x21
00  // Addr = 0x22
0B  // Addr = 0x23
AC  // Addr = 0x24
1A  // Addr = 0x25
00  // Addr = 0x26
10  // Addr = 0x27
AC  // Addr = 0x28
19  // Addr = 0x29
00  // Addr = 0x2A
14  // Addr = 0x2B
AC  // Addr = 0x2C
18  // Addr = 0x2D
00  // Addr = 0x2E
18  // Addr = 0x2F
```

從 Addr = 0x30 後都是 FF，所以就沒列出。

III. Simulation Result Waveform



Simulation Waveform

可以看到 CPU 的 Input_Addr 及 Output_Addr 是跟預期的一樣，Jump 指令也有正確跳到指定位址，然後控制器的輸出也有在正確的時機輸出正確的值，整體看起來是沒問題的。

IV. RF.out and DM.out after simulation

```
20  00000040
21  00000000
22  00000000
23  00000000
24  00000000
25  0000000f // R[24]
26  000017e0 // R[25]
27  00000000
28  00000000
29  00000000
30  00000000
31  ffffffff
32  ffffffff
```

RF.out

可以看到在 Register File 裡，$24 跟$25 如預期存入了 0000_000F 和 0000_17E0。

```
ff // Addr = 0x00
ff
ff
ff
00 // Addr = 0x04
00
00
37
ff // Addr = 0x08
ff
ff
ff
00 // Addr = 0x0C
00
00
03
ff // Addr = 0x10
ff
ff
ff
ff // Addr = 0x14
ff
ff
ff
00 // Addr = 0x18
00
00
0f
ff
ff
ff
ff
```

DM.out

可以看到 Data Memory 的 MEM[0x04]是 0000_0037、MEM[0x0C]是 0000_0003、MEM[0x18]是 0000_000F，跟上面預期的一樣。

根據上面所有的測試，可以得知 SimpCPU 及其他模組的功能是正常的。

10. Datapath Rethinking

建立一個乘數與除數共用的暫存器 A，其輸入來自 ALU_src 多工器的輸出，並將 ALU_src 的輸出與 A 的輸出，各自連接到一顆新多工器 X 的輸入，並將 X 的輸出連接到 ALU 的其中一個輸入。如果要使用乘除法運算就將 X 的輸出導向到 A 的輸出；如果不使用乘除法，就將 X 的輸出導向到 ALU_src 的輸出。

建立一個 64-bits 的暫存器 B，商數/餘數放置在左 32-bits，被乘數/被除數放置在右 32-bits。將 RF 的 Rs_data 輸出到 B 的右半部，且將 Rs_data 與 B 的左半部輸出到一顆新多工器 Y，並將 Y 的輸出連接到 ALU 另一個輸入。若要使用乘除法運算，就將 Y 的輸出導向到 B 的左半部；若不使用乘除法運算，就將 Y 的輸出導向到 Rs_data。

Controller 控制 ALU Controller 要不要執行乘除法運算，控制多工器 X、Y。

ALU Controller 控制暫存器 A、B 要不要從 ALU_src 與 Rs_data 接收資料，控制 B 的位元左右移，控制 B 的左半部要不要寫入 ALU，控制 ALU 的運算功能，檢查是否乘除法運算結束。

11. Conclusion and insights

我覺得 Single Cycle 的 CPU 算是蠻好做的，也是個很適合新手入門的專案。未來後，在這基礎上加入 pipeline，會更好實作。

在測試的時候我覺得比較麻煩的是指令的轉換，從組合語言轉成 4-bytes 十六進制的形式。如果像我一樣，一個一個位元寫出來，用計算機轉十六進制，再一個一個貼到 IM.dat 裡，那會有點花時間。但是當程式跑出來的結果，跟預期想的一樣，那就很有成就感。

這次的 CPU 雖來看起來 datapath 很複雜，但只要釐清之間的關係，加上每個模組的分工清晰，其實做起來很快，要找 bug 沒那麼難。總之，這次作業讓我更熟悉上課內容，從講義上抽象的概念，變成實際看的到的結果，我想這能讓我更了解課堂內容的運作邏輯，也可以從實作中學到課堂上沒教到的、更細節的內容。