

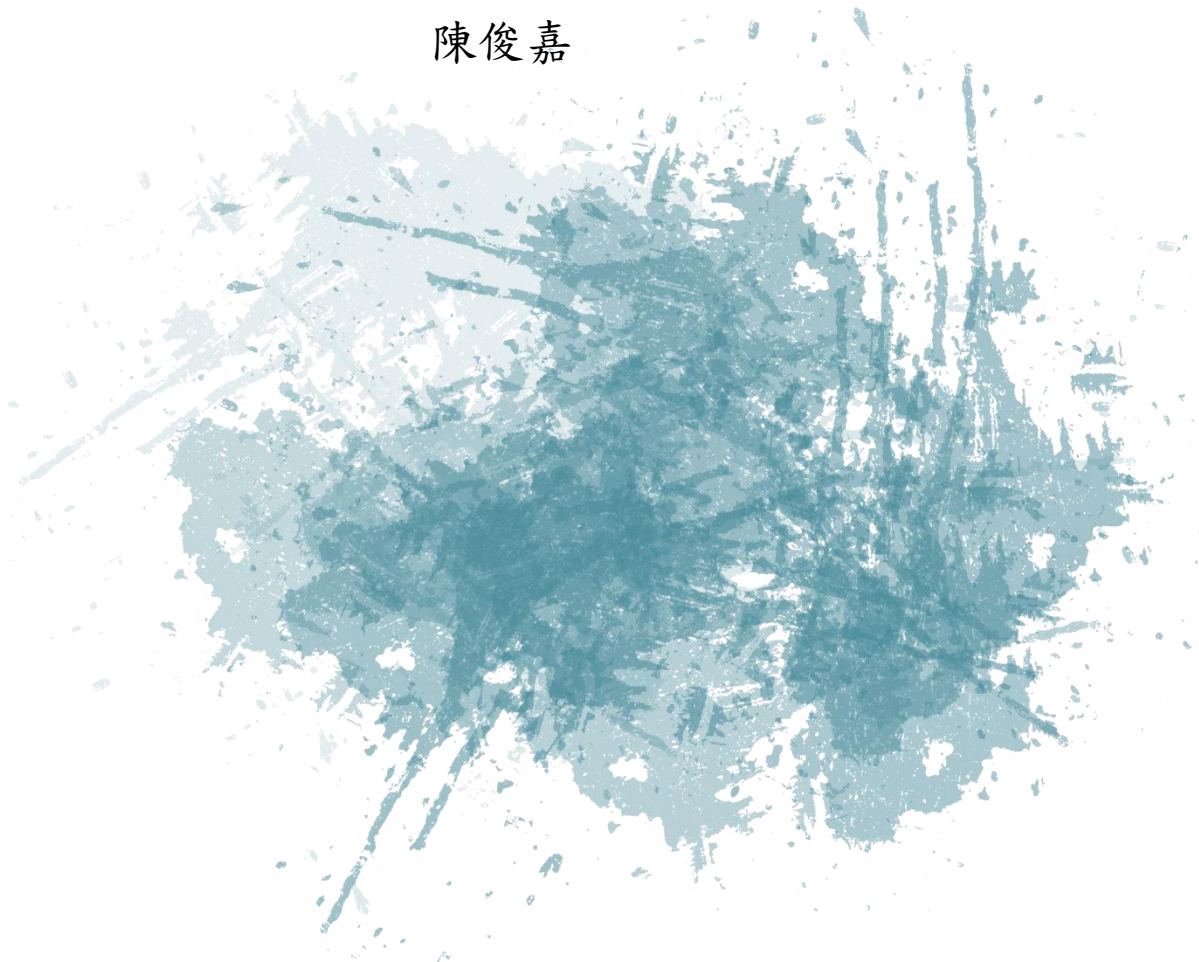
Computer Organization Project

2

四電機三甲

B10830009

陳俊嘉

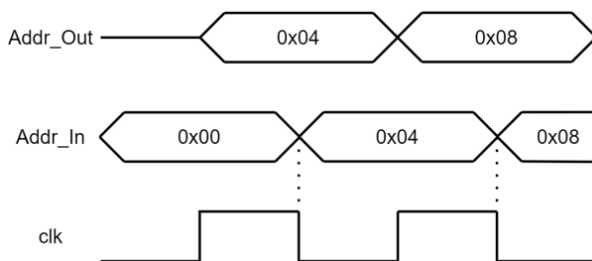


Part I: Single cycle processor with R-format instructions

- Adder Module

```
1 module Adder (addr_in, addr_out, clk) ;
2     output reg [31:0]addr_out;
3     input [31:0]addr_in;
4     input clk;
5
6     always@(posedge clk)begin
7         addr_out <= addr_in + 31'd4;
8     end
9 endmodule
```

Adder用來取得指向下一個Instruction的位址，在positive edge trigger是為了長得跟PA說明中的下圖一樣，直接用assign功能也正常。因為功能簡單且直接跑top level的test bench也能看到其輸出，所以就不另外做單獨的test bench。



- Instruction Memory Module

```
25 /*
26  * Macro of size declaration of instruction memory
27  * CAUTION: DONT MODIFY THE NAME AND VALUE.
28  */
29 `define INSTR_MEM_SIZE 128 // Bytes
30
31 /*
32  * Declaration of Instruction Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module IM(
36     // Outputs
37     output wire [31:0]instr,
38     // Inputs
39     input [31:0]instr_addr
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 //Big endian
48 assign instr = {InstrMem[instr_addr], InstrMem[instr_addr + 1], InstrMem[instr_addr + 2], InstrMem[instr_addr + 3]};
49
50 endmodule
```

Instruction Memory內儲存要執行的指令，以一個byte為單位，一次取四個byte (32bits)，為Big Endian，第一個指到的byte會放在MSB。

- Instruction Memory test bench

```

2 // Setting timescale
3 `timescale 10 ns / 1 ns
4
5 // Configuration
6 `define DELAY 1 // # * timescale
7 `define REGISTER_SIZE 8 // bit width
8 `define MAX_REGISTER 128 // index
9 `define DATA_FILE "testbench/IM.dat"
10
11 // Declaration
12 `define LOW 1'b0
13 `define HIGH 1'b1
14
15 module tb_IM;
16
17 // Inputs
18 reg [31:0] instr_addr;
19
20 // Outputs
21 wire [31:0] instr;
22
23 // Clock
24 reg clk = `LOW;
25
26 // Testbench variables
27 reg [`REGISTER_SIZE-1:0] register [0:`MAX_REGISTER-1];
28 //integer output_file;
29 integer i;
30
31 // Instantiate the Unit Under Test (UUT)
32 IM UUT(
33 // Inputs
34 .instr_addr(instr_addr),
35 // Outputs
36 .instr(instr)
37 );
38
39 initial
40 begin : Preprocess
41 // Initialize inputs
42 instr_addr = 0;
43 // Initialize testbench files
44 $readmemh(`DATA_FILE, register);
45
46 // Initialize internal register
47 for (i = 0; i < `MAX_REGISTER; i = i + 1)
48 begin
49 UUT.InstrMem[i] = register[i];
50 end
51
52 #`DELAY;
53 end
54
55 always
56 begin : ClockGenerator
57 #`DELAY;
58 clk <= ~clk;
59 end
60
61 always
62 begin : StimuliProcess
63 // Start testing
64 for (i = 0; i < `MAX_REGISTER; i = i + 4)
65 begin
66 instr_addr = i[`REGISTER_SIZE-1:0];
67
68 @(clk); // Wait clock
69 end
70 // Stop the simulation
71 $stop();
72 end
73 endmodule

```

- Instruction Memory Simulation

/tb_IM/instr_addr	0000007c	00000000	00000004	00000008	0000000c	00000010	00000014	00000018	0000001c	00000020	00000024	00000028	0000002c	00000030	00000034	00000038	0000003c	00000040
/tb_IM/instr	ffffff	014ba00b	01aca80d	0232b025	01c0ba82	0260c082	ffffff											
/tb_IM/clk	0																	

IM - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```

// Instruction Memory in Hex
01 // Addr = 0x00
4B // Addr = 0x01
A0 // Addr = 0x02
0B // Addr = 0x03
01 // Addr = 0x04
AC // Addr = 0x05
A8 // Addr = 0x06
0D // Addr = 0x07
02 // Addr = 0x08
32 // Addr = 0x09
B0 // Addr = 0x0A
25 // Addr = 0x0B
01 // Addr = 0x0C
C0 // Addr = 0x0D
BA // Addr = 0x0E
82 // Addr = 0x0F
02 // Addr = 0x10
60 // Addr = 0x11
C0 // Addr = 0x12
82 // Addr = 0x13

```

從testbench內資料夾中的IM.DAT檔中讀取資料寫進 Instruction Memory。每個cycle的地址加四，輸出正常。

• Register File Module

```

29 `define REG_MEM_SIZE    32      // Words
30
31 /*
32  * Declaration of Register File for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module RF(
36     // Outputs
37     output wire[31:0]src_data,
38     output wire[31:0]tar_data,
39     // Inputs
40     input  [4:0]src_addr,
41     input  [4:0]tar_addr,
42     input  [4:0]dst_addr,
43     input  [31:0]dst_data,
44     input  clk,
45     input  reg_write
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 value by register point by address
54 assign src_data = R[src_addr];
55 assign tar_data = R[tar_addr];
56
57 always@(negedge clk)begin
58     //write result to register addressed by dst_data
59     if(reg_write)begin
60         R[dst_addr] <= dst_data;
61     end
62 end
63
64 endmodule

```

Register File 用來存取暫存器的資料，在negative edge時判斷信號reg_write，若有訊號則將資料存進address指向的暫存器內。

• Register File test bench

```

1 // Setting timescale
2 `timescale 10 ns / 1 ns
3
4 // Configuration
5 `define DELAY            1      // # * timescale
6 `define REGISTER_SIZE    32    // bit width
7 `define MAX_REGISTER     32    // index
8 `define DATA_FILE       "testbench/RF.dat"
9 `define OUTPUT_FILE      "testbench/tb_RF.out"
10
11 // Declaration
12 `define LOW              1'b0
13 `define HIGH             1'b1
14
15 module tb_RF;
16
17     // Inputs
18     reg [4:0]src_addr, tar_addr, dst_addr;
19     reg [31:0]dst_data;
20     reg reg_write;
21
22     // Outputs
23     wire [31:0] src_data, tar_data;
24
25     // Clock
26     reg clk = `LOW;
27
28     // Testbench variables
29     reg [`REGISTER_SIZE-1:0] register [0:`MAX_REGISTER-1];
30     //integer output_file;
31     integer i;
32
33     // Instantiate the Unit Under Test (UUT)
34     RF UUT(
35         // Inputs
36         .src_addr(src_addr),
37         .tar_addr(tar_addr),
38         .dst_addr(dst_addr),
39         .dst_data(dst_data),
40         .clk(clk),
41         .reg_write(reg_write),
42         // Outputs
43         .src_data(src_data),
44         .tar_data(tar_data)
45     );
46
47     initial
48     begin : Preprocess
49         // Initialize inputs
50         src_addr = 5'b0;
51         tar_addr = 5'b0;
52         reg_write = `LOW;
53         dst_data = 32'h07070707;
54         dst_addr = 5'd3;
55         // Initialize testbench files
56         $readmemh("DATA_FILE", register);
57
58         // Initialize internal register
59         for (i = 0; i < `MAX_REGISTER; i = i + 1)
60         begin
61             UUT.R[i] = register[i];
62         end
63
64         #`DELAY; // Wait for global reset to finish
65     end
66
67     always
68     begin : ClockGenerator
69         #`DELAY;
70         clk <= ~clk;
71     end
72
73     always
74     begin : StimuliProcess
75         // Start testing
76         for (i = 0; i < `MAX_REGISTER; i = i + 1)
77         begin
78             if(i == 10)begin
79                 reg_write <= `HIGH;
80                 @(clk);
81             end else begin
82                 reg_write <= `LOW;
83             end
84             src_addr = i[`REGISTER_SIZE-1:0];
85             tar_addr = `REGISTER_SIZE'd`MAX_REGISTER-1 - i[`REGISTER_SIZE-1:0];
86             @(clk); // Wait clock
87         end
88     end
89
90     // Close output file for safety
91
92     // Stop the simulation
93     $stop();
94
95 end
96 endmodule

```

• Register File Simulation

Source address 和target address一個由0開始往後數，一個由31開始往前數，在，預設destination address為3，原本數值由source address先指到，為77777777，之後將reg_write打開一個clock，後來target address指到03時，成功讀取更改後的數值。

/tb_RF/src_addr	1f	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10
/tb_RF/tar_addr	00	1f	1e	1d	1c	1b	1a	19	18	17	16	15	14	13	12	11	10	0f
/tb_RF/dst_addr	03	03																
/tb_RF/src_data	ffffff	00000000	00000001	00000000	77777777	77777777	77777777	77777777	80000000	ffff0000	0000ffff	00000011	00000023	00000017	00000090	00000100	00000250	00000300
/tb_RF/dst_data	07070707	07070707												00000030	00000064	00000037	00000300	00000250
/tb_RF/reg_write	0																	
/tb_RF/clock	1																	

/tb_RF/src_addr	1f	0c	0d	0e	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f	
/tb_RF/tar_addr	00	13	12	11	10	0f	0e	0d	0c	0b	0a	09	08	07	06	05	04	03	02	01	00	
/tb_RF/dst_addr	03																					
/tb_RF/src_data	00000000	00000017	00000090	00000100	00000250	00000300	00000037	00000064	00000030	00000000										ffffff		
/tb_RF/tar_data	00000000	00000030	00000064	00000037	00000300	00000250	00000100	00000090	00000017	00000023	00000011	0000ffff	ffff0000	80000000	77777777	77777777	77777777	07070707	00	000002	00000001	00000000
/tb_RF/reg_write	0	07070707																				
/tb_RF/clock	1																					

• Control Module

```

1 module Control(OP, reg_write, ALU_OP);
2     //outputs
3     output reg [1:0]ALU_OP;
4     output reg reg_write;
5     //inputs
6     input [5:0]OP;
7
8     always@(*)begin
9         //if is R type -> set ALU_OP = 10
10        if(OP == 6'd0)begin
11            ALU_OP <= 2'b10;
12            reg_write <= 1;
13        end
14
15        else begin
16            ALU_OP <= 0;
17            reg_write <= 0;
18        end
19    end
20
21 endmodule

```

Control module用來處理控制信號，判斷OP code，若判斷為000000的話，代表這個指令為R-type的指令，該指令最後會將答案存回暫存器，所以要打開reg_write信號，並設ALU_OP為10給ALU controller進行ALU function的信號處理。

因為沒有時序問題，又只有OP code要判斷，不容易出錯，因此沒有做test bench

• ALU Control Module

```

1 module ALU_Control (
2     output reg [5:0]funct,
3     //inputs
4     input [5:0]funct_ctrl,
5     input [1:0]ALU_OP
6 );
7     always@(ALU_OP or funct_ctrl)begin
8         if(ALU_OP == 2'b10)begin
9             case(funct_ctrl)
10                6'b001011: funct <= 6'b001001; //ADDU
11                6'b001101: funct <= 6'b001010; //SUBU
12                6'b100101: funct <= 6'b010010; //OR
13                6'b000010: funct <= 6'b100010; //SRL
14            endcase
15        end else begin
16            funct <= 0;
17        end
18    end
19 endmodule

```

透過判斷ALU_OP訊號來給ALU function，若為10代表該指令為R-type，則判斷Instruction的最後6 bits來給ALU對應的function

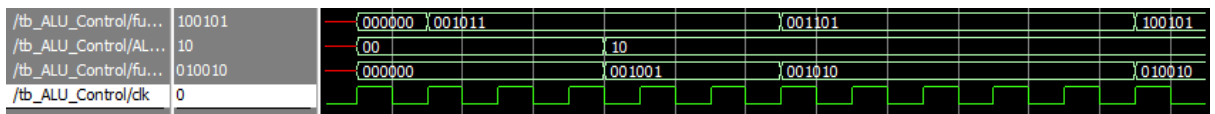
- ALU Control test bench

```

1  // Setting timescale
2  `timescale 10 ns / 1 ns
3  `define sub    6'b001101
4  `define add    6'b001011
5  `define orOP   6'b100101
6  `define SRL    6'b000010
7  module tb_ALU_Control;
8
9      // Inputs
10     reg [5:0]funcnt_ctrl;
11     reg [1:0]ALU_OP;
12
13     // Outputs
14     wire [5:0]funcnt;
15
16     // Clock
17     reg clk = 1'b0;
18
19     // Instantiate the Unit Under Test (UUT)
20     ALU_Control UUT(
21         .funcnt(funcnt),
22         .funcnt_ctrl(funcnt_ctrl),
23         .ALU_OP(ALU_OP)
24     );
25
26     // Generate Clock
27     always
28     begin : ClockGenerator
29         #1 clk <= ~clk;
30     end
31
32     initial begin
33         // Wait positive edge of clock signal
34         @(posedge clk);
35
36         // Reset UUT
37         funcnt_ctrl <= 0;
38         ALU_OP <= 0;
39
40         //assign value to input sources
41         @(posedge clk);
42         funcnt_ctrl <= `add;
43
44         #5;
45         ALU_OP <= 2'b10;
46
47         //Wait 5ns, change inputs
48         #5;
49         funcnt_ctrl <= `sub;
50         //Wait 5ns, change inputs
51         #10;
52         funcnt_ctrl <= `orOP;
53
54         // Wait some time
55         #2;
56
57         // Stop the simulation
58         $stop();
59     end
60
61 endmodule

```

- ALU Control Simulation



Function control判斷為10後，function輸出皆正常。

- ALU Module

```

1  module ALU (
2      //outputs
3      output reg [31:0]dst_data,
4      //inputs
5      input [31:0]src_data,
6      input [31:0]tar_data,
7      input [4:0]shamt,
8      input [5:0]funcnt
9  );
10
11     always@(*)begin
12         case(funcnt)
13             //ADDU
14             6'b001001: dst_data <= src_data + tar_data;
15             //SUBU
16             6'b001010: dst_data <= src_data - tar_data;
17             //OR
18             6'b010010: dst_data <= src_data | tar_data;
19             //SRL
20             6'b100010: dst_data <= src_data >> shamt;
21             default: dst_data <= 32'd0;
22         endcase
23     end
24 endmodule

```

ALU收到由ALU Controller傳來的function後，進行運算後將結果傳至destination data中，再交由Register File處存至相對應的位置。

在HW1與PA1中都有做過ALU，這次就不在做test bench

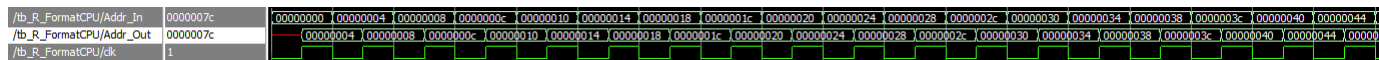
- R_Format CPU

```

66         ALU ALU_Unit(
67             //outputs
68             .dst_data(DST_DATA),
69             //inputs
70             .src_data(SRC_DATA),
71             .tar_data(TAR_DATA),
72             .shamt(INSTR[10:6]),
73             .funcnct(FUNCT)
74         );
75     ALU_Control ALU_Control_Unit(
76         //outputs
77         .funcnct(FUNCT),
78         //inputs
79         .funcnct_ctrl(INSTR[5:0]),
80         .ALU_OP(ALU_OP)
81     );
82     Adder ADDER(
83         //output
84         .addr_out(Addr_Out),
85         //input
86         .addr_in(Addr_In),
87         .clk(clk)
88     );
89 endmodule

```

- R_Format CPU Simulation



```

// Register File in Hex
0000_0000 // R[0]
0000_0001 // R[1]
0000_0002 // R[2]
7777_7777 // R[3]
7F7F_7F7F // R[4]
F7F7_F7F7 // R[5]
7FFF_FFFF // R[6]
8000_0000 // R[7]
FFFF_0000 // R[8]
0000_FFFF // R[9]
0000_0011 // R[10]
0000_0023 // R[11]
0000_0017 // R[12]
0000_0090 // R[13]
0000_0100 // R[14]
0000_0250 // R[15]
0000_0300 // R[16]
0000_0037 // R[17]
0000_0064 // R[18]
0000_0030 // R[19]
0000_0000 // R[20]
0000_0000 // R[21]
0000_0000 // R[22]
0000_0000 // R[23]
0000_0000 // R[24]
0000_0000 // R[25]
0000_0000 // R[26]
0000_0000 // R[27]
0000_0000 // R[28]
0000_0000 // R[29]
FFFF_FFFF // R[30]
FFFF_FFFF // R[31]

```

```

00000000
00000001
00000002
77777777
7f7f7f7f
f7f7f7f7
7fffffff
80000000
ffff0000
0000ffff
00000011
00000023
00000017
00000090
00000100
00000250
00000300
00000037
00000064
00000030
00000034 // [20]
00000079 // [21]
00000077 // [22]
00000000 // [23]
0000000c // [24]
00000000
00000000
00000000
00000000
00000000
ffffff
ffffff

```

Instructions

Hexadecimal	Binary	Meaning
014BA00B	000000_01010_01011_10100_00000_001011	\$R20 = \$R10 + \$R11
01ACA80D	000000_01101_01100_10101_00000_001101	\$R21 = \$R13 - \$R12
0232B025	000000_10001_10010_10110_00000_100101	\$R22 = \$R17 \$R18
01C0BA82	000000_01110_00000_10111_01010_000010	\$R23 = \$R14 >> 10
0260C082	000000_10011_00000_11000_00010_000010	\$R24 = \$R19 >> 2

Part I: Single cycle processor with R-format and I-format instructions

※ Instruction Memory、Adder、Register File、ALU 跟上個Part一樣

- Control Module

```
1 module Control(  
2     //outputs  
3     output reg [1:0]ALU_OP,  
4     output reg reg_write,  
5     output reg reg_dst,  
6     output reg ALU_src,  
7     output reg mem_write,  
8     output reg mem_read,  
9     output reg mem2reg,  
10    //inputs  
11    input [5:0]OP  
12);  
13  
14    always@(*)begin  
15        case(OP)  
16            //if is R type -> set ALU_OP = 10  
17            6'd0:begin  
18                ALU_OP <= 2'b10;  
19                reg_write <= 1;  
20                reg_dst <= 1;  
21                ALU_src <= 0;  
22                mem_write <= 0;  
23                mem_read <= 0;  
24                mem2reg <= 0;  
25            end  
26            //ADD immediate  
27            6'b001100:begin  
28                ALU_OP <= 2'b00;  
29                reg_write <= 1;  
30                reg_dst <= 0;  
31                ALU_src <= 1;  
32                mem_write <= 0;  
33                mem_read <= 0;  
34                mem2reg <= 0;  
35            end  
36            //SUB immediate  
37            6'b001101:begin  
38                ALU_OP <= 2'b01;  
39                reg_write <= 1;  
40                reg_dst <= 0;  
41                ALU_src <= 1;  
42                mem_write <= 0;  
43                mem_read <= 0;  
44                mem2reg <= 0;  
45            end  
46            //store  
47            6'b010000:begin  
48                ALU_OP <= 2'b00;  
49                reg_write <= 0;  
50                //reg_dst <= 0;    Don't care  
51                ALU_src <= 1;  
52                mem_write <= 1;  
53                mem_read <= 0;  
54                //mem2reg <= 0;    Don't care  
55            end  
56            //load  
57            6'b010001:begin  
58                ALU_OP <= 2'b00;  
59                reg_write <= 1;  
60                reg_dst <= 0;  
61                ALU_src <= 1;  
62                mem_write <= 0;  
63                mem_read <= 1;  
64                mem2reg <= 1;  
65            end  
66            default:begin  
67                ALU_OP <= 2'b00;  
68                reg_write <= 0;  
69                reg_dst <= 0;  
70                ALU_src <= 0;  
71                mem_write <= 0;  
72                mem_read <= 0;  
73                mem2reg <= 0;  
74            end  
75        endcase  
76    end  
77 endmodule
```

- Control test bench

```
1 // Setting timescale  
2 `timescale 10 ns / 1 ns  
3  
4 module tb_Control;  
5  
6     // Outputs  
7     wire [1:0]ALU_OP;  
8     wire reg_write;  
9     wire reg_dst;  
10    wire ALU_src;  
11    wire mem_write;  
12    wire mem_read;  
13    wire mem2reg;  
14    //inputs  
15    reg [5:0]OP;  
16  
17    // Clock  
18    reg clk = 1'b0;  
19  
20    // Instantiate the Unit Under Test (UUT)  
21    Control UUT(  
22        .ALU_OP(ALU_OP),  
23        .reg_write(reg_write),  
24        .reg_dst(reg_dst),  
25        .ALU_src(ALU_src),  
26        .mem_write(mem_write),  
27        .mem_read(mem_read),  
28        .mem2reg(mem2reg),  
29        .OP(OP)  
30    );  
31  
32    // Generate Clock  
33    always  
34    begin : ClockGenerator  
35        #1 clk <= ~clk;  
36    end  
37  
38    initial begin  
39        // Wait positive edge of clock signal  
40        @(posedge clk);  
41  
42        // Reset UUT  
43        OP <= 0;  
44  
45        //assign value to OP code  
46        @(posedge clk);  
47        OP <= 6'b001100;  
48  
49        //assign value to OP code  
50        @(posedge clk);  
51        OP <= 6'b001101;  
52  
53        //assign value to OP code  
54        @(posedge clk);  
55        OP <= 6'b010000;  
56  
57        //assign value to OP code  
58        @(posedge clk);  
59        OP <= 6'b010001;  
60  
61        //assign undefined value to OP code  
62        @(posedge clk);  
63        OP <= 6'b111111;  
64  
65        // Wait some time  
66        #2;  
67  
68        // Stop the simulation  
69        $stop();  
70    end  
71 endmodule
```


- ALU Control test bench

```

28      initial begin
29          // Wait positive edge of clock signal
30          @(posedge clk);
31
32          // Reset UUT
33          ALU_OP <= 0;
34          funct_ctrl <= 0;
35
36          //assign value to ALU_OP code
37          @(posedge clk);
38          ALU_OP <= 2'b00;
39
40          //assign value to function control
41          @(posedge clk);
42          funct_ctrl <= 6'b101010;
43
44          //assign value to ALU_OP code
45          @(posedge clk);
46          ALU_OP <= 2'b01;
47
48          //assign value to function control
49          @(posedge clk);
50          funct_ctrl <= 6'b001101;
51
52
53          // Wait some time
54          #2;
55
56          // Stop the simulation
57          $stop();
58      end
59
60 endmodule

```

- ALU Control Simulation

/tb_ALU_Control/funct	001010	001001		001010	
/tb_ALU_Control/funct_ctrl	001101	000000	101010		001101
/tb_ALU_Control/ALU_OP	01	00		01	
/tb_ALU_Control/dk	0				

I-type的ALU_OP對應的輸出正常，且不受funct_ctrl訊號影響

- Data Memory Module

```

29 `define DATA_MEM_SIZE 128 // Bytes
30
31 /*
32  * Declaration of Data Memory for this project.
33  * CAUTION: DONT MODIFY THE NAME.
34  */
35 module DM(
36     // Outputs
37     output reg [31:0] read_data,
38     // Inputs
39     input [31:0] write_data,
40     input [31:0] addr,
41     input mem_read,
42     input mem_write,
43     input clk
44 );
45
46 /*
47  * Declaration of data memory.
48  * CAUTION: DONT MODIFY THE NAME AND SIZE.
49  */
50 reg [7:0] DataMem[0:DATA_MEM_SIZE - 1];
51
52 always@(posedge clk)begin
53     if(mem_read)begin
54         read_data <= {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]};
55     end
56
57     else if(mem_write)begin
58         {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]} <= write_data;
59     end
60 end
61 endmodule

```

- Data Memory test bench

```

1 // Setting timescale
2 `timescale 10 ns / 1 ns
3
4 // Configuration
5 `define DELAY 1 // # * timescale
6 `define REGISTER_SIZE 8 // bit width
7 `define MAX_REGISTER 128 // index
8 `define DATA_FILE "testbench/DM.dat"
9 `define OUTPUT_FILE "testbench/tb_DM.out"
10
11 // Declaration
12 `define LOW 1'b0
13 `define HIGH 1'b1
14
15 module tb_DM;
16
17     // Inputs
18     reg [31:0] write_data, addr;
19     reg mem_write, mem_read;
20
21     // Outputs
22     wire [31:0] read_data;
23
24     // Clock
25     reg clk = `LOW;
26
27     // Testbench variables
28     reg [REGISTER_SIZE-1:0] register [0:MAX_REGISTER-1];
29     integer output_file;
30     integer i;
31
32     // Instantiate the Unit Under Test (UUT)
33     DM UUT(
34         .read_data(read_data),
35         .write_data(write_data),
36         .addr(addr),
37         .mem_read(mem_read),
38         .mem_write(mem_write),
39         .clk(clk)
40     );

```

```

42 initial
43 begin : Preprocess
44     // Initialize inputs
45     write_data = 32'h0000_001f;
46     addr = 0;
47     mem_write = `LOW;
48     mem_read = `LOW;
49
50     // Initialize testbench files
51     $readmemh("DATA_FILE", register);
52     output_file = $fopen("OUTPUT_FILE");
53
54     // Initialize internal register
55     for (i = 0; i < MAX_REGISTER; i = i + 1)
56     begin
57         UUT.DataMem[i] = register[i];
58     end
59
60     #`DELAY; // Wait for global reset to finish
61 end
62
63 always
64 begin : ClockGenerator
65     #`DELAY;
66     clk <= ~clk;
67 end
68
69 always
70 begin : StimuliProcess
71     // Start testing
72     for (i = 0; i < MAX_REGISTER; i = i + 4)
73     begin
74         if(i == 12)begin
75             mem_write <= `HIGH;
76             @(clk);
77         end else begin
78             mem_write <= `LOW;
79         end
80
81         if(i == 20)begin
82             mem_read <= `HIGH;
83             @(clk);
84         end else begin
85             mem_read <= `LOW;
86         end
87
88         addr = i[REGISTER_SIZE-1:0];
89         @(clk); // Wait clock
90
91         for(i = 0; i < MAX_REGISTER; i = i + 1)begin
92             register[i] = UUT.DataMem[i];
93             $fwrite(output_file, "%x\n", register[i]);
94         end
95
96         // Close output file for safety
97         $fclose(output_file);
98
99         // Stop the simulation
100         $stop();
101     end
102 end
103 endmodule

```

- Data Memory Simulation

/tb_DM/write_data	0000001f
/tb_DM/addr	0000007c
/tb_DM/mem_write	0
/tb_DM/mem_read	0
/tb_DM/read_data	ffffffff
/tb_DM/clk	0

Data memory 會由testbench中的DM.DAT初始資料，預設原本全都是FF，預設要寫進的data是0000_001f，而在address跑到12時打開mem_write訊號一個clock後關掉，由左圖得知write功能正常。而在address指到20時將mem_read訊號打開一個clock，輸出的read_data訊號正常。

```
*tb_DM - 記事4  
檔案(F)   編輯(E)  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
ff  
00      //[ 12]  
00  
00  
1f  
ff  
ff  
ff  
ff
```

- Multiplexer & Sing Extension Module

```

1 module MUX5(
2     //Outputs
3     output reg [4:0] data_out,
4     //Inputs
5     input [4:0] data0,
6     input [4:0] data1,
7     input select
8 );
9
10 always @(*) begin
11     if(select)begin
12         data_out <= data1;
13     end else begin
14         data_out <= data0;
15     end
16 end
17 endmodule

```

```

1 module MUX32(
2     //Outputs
3     output reg [31:0] data_out,
4     //Inputs
5     input [31:0] data0,
6     input [31:0] data1,
7     input select
8 );
9 always @(*) begin
10     if(select)begin
11         data_out <= data1;
12     end else begin
13         data_out <= data0;
14     end
15 end
16
17 endmodule

```

```

1 module Sign_Extend(
2     //output
3     output reg [31:0]extend_out,
4     //input
5     input [15:0]immediate
6 );
7
8 always @(*) begin
9     if(immediate[15] == 0)begin
10         extend_out <= {16'd0, immediate};
11     end
12     else if(immediate[15] == 1)begin
13         extend_out <= {16'hFFFF, immediate};
14     end
15 end
16
17 endmodule
18

```

因為R-type跟I-type的destination address放在instruction的不同地方，所以需要一個5-bit的multiplexer以RegDst訊號決定輸入訊號。而32-bit的multiplexer會用在選擇ALU的輸入及destination data的輸入訊號，分別由ALUSrc及MemToReg做控制。

Sign Extension是因為immediate只有16 bits，而ALU是用來運算32 bits的，所以處理I-type指令時會用到Sign Extension。

Part III: Single cycle processor with branch and jump instructions

※Instruction Memory、Register File、MUX、Sign extension、ALU Control、DM 跟上個Part一樣

- Adder Module

```
1 module Adder (data1, data2, data_out) ;  
2     output [31:0] data_out;  
3     input [31:0] data1, data2;  
4  
5     assign data_out = data1 + data2;  
6  
7 endmodule
```

跟前面兩個part不同的是前面的adder只有一個輸入，因為只會固定指向下一個指令，所以adder只要將輸入加四就好。但在這個part也會將adder用來加branch的位移，所以在這邊把原本的adder改成吃兩個輸入。

- Control Module

```

1 module Control (
2     //outputs
3     output reg [1:0]ALU_OP,
4     output reg reg_write,
5     output reg reg_dst,
6     output reg ALU_src,
7     output reg mem_write,
8     output reg mem_read,
9     output reg mem2reg,
10    output reg branch,
11    output reg jump,
12    //inputs
13    input [5:0]OP
14 );
15
16 always@(*)begin
17     case(OP)
18         //if is R type -> set ALU_OP = 10
19         6'd0:begin
20             ALU_OP <= 2'b10;
21             reg_write <= 1;
22             reg_dst <= 1;
23             ALU_src <= 0;
24             mem_write <= 0;
25             mem_read <= 0;
26             mem2reg <= 0;
27             branch <= 0;
28             jump <= 0;
29         end
30         //ADD immediate
31         6'b001100:begin
32             ALU_OP <= 2'b00;
33             reg_write <= 1;
34             reg_dst <= 1;
35             ALU_src <= 1;
36             mem_write <= 0;
37             mem_read <= 0;
38             mem2reg <= 0;
39             branch <= 0;
40             jump <= 0;
41         end
42         //SUB immediate
43         6'b001101:begin
44             ALU_OP <= 2'b01;
45             reg_write <= 1;
46             reg_dst <= 0;
47             ALU_src <= 1;
48             mem_write <= 0;
49             mem_read <= 0;
50             mem2reg <= 0;
51             branch <= 0;
52             jump <= 0;
53         end
54     endcase
55 end

```

```

53 //store
54 6'b010000:begin
55     ALU_OP <= 2'b00;
56     reg_write <= 0;
57     //reg_dst <= 0; Don't care
58     ALU_src <= 1;
59     mem_write <= 1;
60     mem_read <= 0;
61     //mem2reg <= 0; Don't care
62     branch <= 0;
63     jump <= 0;
64 end
65 //load
66 6'b010001:begin
67     ALU_OP <= 2'b00;
68     reg_write <= 1;
69     reg_dst <= 0;
70     ALU_src <= 1;
71     mem_write <= 0;
72     mem_read <= 1;
73     mem2reg <= 1;
74     branch <= 0;
75     jump <= 0;
76 end
77 //beq
78 6'b010011:begin
79     ALU_OP <= 2'b01;
80     reg_write <= 0;
81     //reg_dst <= 0; Don't care
82     ALU_src <= 0;
83     mem_write <= 0;
84     mem_read <= 0;
85     //mem2reg <= 1; Don't care
86     branch <= 1;
87     jump <= 0;
88 end
89 //jmp
90 6'b011100:begin
91     ALU_OP <= 2'b01;
92     reg_write <= 0;
93     //reg_dst <= 0; Don't care
94     ALU_src <= 0;
95     mem_write <= 0;
96     mem_read <= 0;
97     //mem2reg <= 1; Don't care
98     branch <= 0;
99     jump <= 1;
100 end
101 default:begin
102     ALU_OP <= 2'b00;
103     reg_write <= 0;
104     reg_dst <= 0;
105     ALU_src <= 0;
106     mem_write <= 0;
107     mem_read <= 0;
108     mem2reg <= 0;
109     branch <= 0;
110 end
111 endcase

```

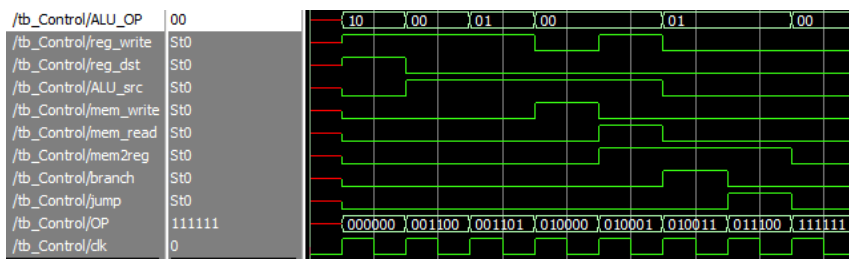
大致都與前面一樣，只是加了兩個輸出訊號：branch和jump，分別用來控制選擇有沒有branch和jump的兩個32-bit MUX。

• Control test bench & Simulation

```

1 // Setting timescale
2 `timescale 10 ns / 1 ns
3
4 module tb_Control;
5
6     // Outputs
7     wire [1:0]ALU_OP;
8     wire reg_write;
9     wire reg_dst;
10    wire ALU_src;
11    wire mem_write;
12    wire mem_read;
13    wire mem2reg;
14    wire branch;
15    wire jump;
16    //inputs
17    reg [5:0]OP;
18
19    // Clock
20    reg clk = 1'b0;
21
22    // Instantiate the Unit Under Test (UUT)
23    Control UUT(
24        .ALU_OP(ALU_OP),
25        .reg_write(reg_write),
26        .reg_dst(reg_dst),
27        .ALU_src(ALU_src),
28        .mem_write(mem_write),
29        .mem_read(mem_read),
30        .mem2reg(mem2reg),
31        .branch(branch),
32        .jump(jump),
33        .OP(OP)
34    );
35
36    // Generate Clock
37    always
38    begin : ClockGenerator
39        #1 clk <= ~clk;
40    end
41
42    initial begin
43        // Wait positive edge of clock signal
44        @(posedge clk);
45
46        // Reset UUT
47        OP <= 0;
48
49        //assign value to OP code
50        @(posedge clk);
51        OP <= 6'b001100;
52
53        //assign value to OP code
54        @(posedge clk);
55        OP <= 6'b001101;
56
57        //assign value to OP code
58        @(posedge clk);
59        OP <= 6'b010000;
60
61        //assign value to OP code
62        @(posedge clk);
63        OP <= 6'b010001;
64
65        //assign value to OP code
66        @(posedge clk);
67        OP <= 6'b010011; //beq
68
69        //assign value to OP code
70        @(posedge clk);
71        OP <= 6'b011100; //jump
72
73        //assign undefined value to OP code
74        @(posedge clk);
75        OP <= 6'b111111;
76
77        // Wait some time
78        #2;
79        // Stop the simulation
80        $stop();
81    end
82 endmodule

```



加入beq和jump的OP code 進行測試。

• ALU Module

```

1 module ALU (
2     //outputs
3     output reg [31:0]data_out,
4     output reg zero,
5     //inputs
6     input [31:0]src_data,
7     input [31:0]tar_data,
8     input [4:0]shamt,
9     input [5:0]funct
10 );
11
12 always@(*)begin
13     case(funct)
14         //ADDU
15         6'b001001: data_out <= src_data + tar_data;
16         //SUBU
17         6'b001010: data_out <= src_data - tar_data;
18         //OR
19         6'b010010: data_out <= src_data | tar_data;
20         //SRL
21         6'b100010: data_out <= src_data >> shamt;
22         default: data_out <= 32'd0;
23     endcase
24     zero <= (data_out == 0)? 1'b1 : 1'b0;
25 end
26 endmodule

```

大致上與前面的都一樣，加入Zero Flag，用在判斷beq的指令，在相減後是否等於零。

• ALU test bench & Simulation

```

1 // Setting timescale
2 `timescale 10 ns / 1 ns
3
4 module tb_ALU;
5
6     // Outputs
7     wire [31:0] data_out;
8     wire zero;
9     //inputs
10    reg [31:0] src_data;
11    reg [31:0] tar_data;
12    reg [4:0] shamt;
13    reg [5:0] funct;
14
15    // Clock
16    reg clk = 1'b0;
17
18    // Instantiate the Unit Under Test (UUT)
19    ALU UUT(
20        .data_out(data_out),
21        .zero(zero),
22        .src_data(src_data),
23        .tar_data(tar_data),
24        .shamt(shamt),
25        .funct(funct)
26    );
27
28    // Generate Clock
29    always
30    begin : ClockGenerator
31        #1 clk <= ~clk;
32    end
33
34    initial begin
35        // Wait positive edge of clock signal
36        @(posedge clk);
37
38        // Reset UUT
39        funct <= 0;
40        shamt <= 0;
41        src_data <= 0;
42        tar_data <= 0;
43
44        //assign value to OP code
45        @(posedge clk);
46        src_data <= 32'h0000_FFFF;
47        tar_data <= 32'h0000_8888;
48
49        //assign value to OP code
50        @(posedge clk);
51        funct <= 6'b001010;
52
53        //assign value to OP code
54        @(posedge clk);
55        tar_data <= 32'h0000_FFFF;
56
57        // Wait some time
58        #2;
59
60        // Stop the simulation
61        $stop();
62    end
63
64 endmodule

```

/tb_ALU/data_out	00000000	{00000000, 00007777, 00000000}
/tb_ALU/zero	St1	
/tb_ALU/src_data	0000ffff	{00000000, 0000ffff}
/tb_ALU/tar_data	0000ffff	{00000000, 00008888, 0000ffff}
/tb_ALU/shamt	00000	{00000}
/tb_ALU/funct	001010	{000000, 001010}
/tb_ALU/clk	0	

基本功能前面都測試過了，這邊主要在測試，Zero Flag有沒有正常反應。

• Simple CPU

```

29 module SimpleCPU(
30     // Outputs
31     output wire [31:0] Addr_Out,
32     // Inputs
33     input wire [31:0] Addr_In,
34     input wire clk
35 );
36
37 wire [31:0] INSTR, SRC_DATA, TAR_DATA, DST_DATA;
38 wire [31:0] ALU_IN, ALU_OUT, IMM, DM_OUT, NEXT_PC;
39 wire [31:0] BRANCH_ADDR, NON_JUMP_ADDR;
40 wire [5:0] FUNCT;
41 wire [4:0] DST_ADDR;
42 wire [1:0] ALU_OP;
43 wire REG_WRITE, REG_DST, ALU_SRC, MEM_READ;
44 wire MEM_WRITE, MEM2REG, BRANCH, JUMP, ZERO;
45
46 /* Declaration of Instruction Memory.
47 * CAUTION: DONT MODIFY THE NAME.
48 */
49 IM Instr_Memory(
50     // Outputs
51     .instr(INSTR),
52     // Inputs
53     .instr_addr(Addr_In)
54 );
55
56 Adder Address_Adder(
57     //output
58     .data_out(NEXT_PC),
59     //inputs
60     .data1(Addr_In),
61     .data2(32'd4)
62 );
63
64 MUX5 DST_Addr_MUX(
65     //outputs
66     .data_out(DST_ADDR),
67     //inputs
68     .data0(INSTR[20:16]),
69     .data1(INSTR[15:11]),
70     .select(REG_DST)
71 );
72
73 RF Register_File(
74     // Outputs
75     .src_data(SRC_DATA),
76     .tar_data(TAR_DATA),
77     // Inputs
78     .src_addr(INSTR[25:21]),
79     .tar_addr(INSTR[20:16]),
80     .dst_addr(DST_ADDR),
81     .dst_data(DST_DATA),
82     .reg_write(REG_WRITE),
83     .clk(clk)
84 );
85
86 Control_Controller(
87     //Outputs
88     .ALU_OP(ALU_OP),
89     .reg_write(REG_WRITE),
90     .reg_dst(REG_DST),
91     .ALU_src(ALU_SRC),
92     .mem_write(MEM_WRITE),
93     .mem_read(MEM_READ),
94     .mem2reg(MEM2REG),
95     .branch(BRANCH),
96     .jump(JUMP),
97     //Input
98     .OP(INSTR[31:26])
99 );
100
101 ALU_Control ALU_Controller(
102     //Output
103     .funct(FUNCT),
104     //Inputs
105     .funct_ctrl(INSTR[5:0]),
106     .ALU_OP(ALU_OP)
107 );
108
109 Sign_Extend Sign_Extend_Unit(
110     //output
111     .extend_out(IMM),
112     //input
113     .immediate(INSTR[15:0])
114 );

```



```

123 ALU ALU_Unit(
124     //Output
125     .data_out(ALU_OUT),
126     .zero(ZERO),
127     //Inputs
128     .src_data(SRC_DATA),
129     .tar_data(ALU_IN),
130     .shamt(INSTR[10:6]),
131     .funct(FUNCT)
132 );
133
134 DM Data_Memory(
135     // Outputs
136     .read_data(DM_OUT),
137     // Inputs
138     .write_data(TAR_DATA),
139     .addr(ALU_OUT),
140     .mem_read(MEM_READ),
141     .mem_write(MEM_WRITE),
142     .clk(clk)
143 );
144
145 MUX32 DST_MUX(
146     //Outputs
147     .data_out(DST_DATA),
148     //Inputs
149     .data0(ALU_OUT),
150     .data1(DM_OUT),
151     .select(MEM2REG)
152 );
153
154 Adder Branch_Adder(
155     //Output
156     .data_out(BRANCH_ADDR),
157     //Inputs
158     .data1(NEXT_PC),
159     .data2(IMM << 2)
160 );
161
162 MUX32 Branch_MUX(
163     //Output
164     .data_out(NON_JUMP_ADDR),
165     //Inputs
166     .data0(NEXT_PC),
167     .data1(BRANCH_ADDR),
168     .select(BRANCH && ZERO)
169 );
170
171 MUX32 ADDR_OUT_MUX(
172     //Output
173     .data_out(Addr_Out),
174     //Inputs
175     .data0(NON_JUMP_ADDR),
176     .data1((NEXT_PC[31:28], INSTR[25:0], 2'b00)),
177     .select(JUMP)
178 );
179
180 endmodule

```

- Simple CPU testbench

```

28 // Setting timescale
29 `timescale 10 ns / 1 ns
30
31 // Declarations
32 `define DELAY 1 // * timescale
33 `define INSTR_SIZE 8 // bit width
34 `define INSTR_MAX 128 // bytes
35 `define INSTR_FILE "testbench/IM.dat"
36 `define REG_SIZE 32 // bit width
37 `define REG_MAX 32 // words
38 `define REG_FILE "testbench/RF.dat"
39 `define DATA_SIZE 8 // bit width
40 `define DATA_MAX 128 // bytes
41 `define DATA_FILE "testbench/DM.dat"
42 `define OUTPUT_REG "testbench/RF.out"
43 `define OUTPUT_DATA "testbench/DM.out"
44
45 // Declaration
46 `define LOW 1'b0
47 `define HIGH 1'b1
48
49 module tb_SimpleCPU;
50
51     // Inputs
52     reg [31:0] Addr_In;
53
54     // Outputs
55     wire [31:0] Addr_Out;
56
57     // Clock
58     reg clk;
59
60     // Testbench variables
61     reg [`INSTR_SIZE-1:0] instrMem [0:`INSTR_MAX-1];
62     reg [`REG_SIZE-1:0] regMem [0:`REG_MAX-1];
63     reg [`DATA_SIZE-1:0] dataMem [0:`DATA_MAX-1];
64     integer output_reg;
65     integer output_data;
66     integer i;
67
68     // Instantiate the Unit Under Test (UUT)
69     SimpleCPU UUT(
70         // Outputs
71         .Addr_Out(Addr_Out),
72         // Inputs
73         .Addr_In(Addr_In),
74         .clk(clk)
75     );
76
77     initial
78     begin : Preprocess
79         // Initialize inputs
80         Addr_In = 32'd0;
81         clk = 'LOW;
82
83         // Initialize testbench files
84         $readmemh("INSTR_FILE", instrMem);
85         $readmemh("REG_FILE", regMem);
86         $readmemh("DATA_FILE", dataMem);
87         output_reg = $fopen("OUTPUT_REG");
88         output_data = $fopen("OUTPUT_DATA");
89
90         // Initialize instruction memory
91         for (i = 0; i < `INSTR_MAX; i = i + 1)
92         begin
93             UUT.Instr_Memory.InstrMem[i] = instrMem[i];
94         end
95
96         // Initialize register file
97         for (i = 0; i < `REG_MAX; i = i + 1)
98         begin
99             UUT.Register_File.R[i] = regMem[i];
100         end
101
102         // Initialize data memory
103         for (i = 0; i < `DATA_MAX; i = i + 1)
104         begin
105             UUT.Data_Memory.DataMem[i] = dataMem[i];
106         end
107
108         #`DELAY; // Wait for global reset to finish
109     end
110
111     always
112     begin : ClockGenerator
113         #`DELAY;
114         clk <= ~clk;
115     end
116 end

```


• Simple CPU Simulation

/tb_SimpleCPU/Addr_In	00000004	00000000	00000004	00000008	00000000	00000004	00000008	00000000	00000004	00000008	00000000	00000004	00000008	00000000
/tb_SimpleCPU/Addr_Out	00000008	00000004	00000008	00000000	00000004	00000008	00000000	00000004	00000008	00000000	00000004	00000008	00000000	00000004
/tb_SimpleCPU/clk	1													

/tb_SimpleCPU/Addr_In	0000007c	000...	00000000		00000004		00000008		00000000		0000...
/tb_SimpleCPU/Addr_Out	00000080	000...	00000004		00000008		00000000		0000007c		00000...
/tb_SimpleCPU/clk	1										

由模擬可以看到address不斷在0、4、8輪迴，也就是指到那三條指令，最後beq成立就跳到位移 $30 \times 4 + 4$ 個byte的位置，也就是7c。

Conclusion and insights

這次的PA我自己感覺沒有到很難，但是要做的Module比上次多蠻多的，在最後Top Module接線的部分就變得需要很細心，這次在三個Part裡都找Bug找很久，最後才發現是接線的部分接錯，不夠細心，下次要多注意一點。

這次除了Register Memory以外又多了Data Memory和Instruction Memory，所以在test bench要讀取外部的檔案的部分也比之前的作業還要多上許多，不過經過這次PA後我覺得我也越來越熟悉讀取和寫入檔案的部分了，但是寫test bench我個人真的認為好麻煩，雖然這次有許多小型而且功能間單的module，所以我就沒有特別去做test bench去做測試，但是每次都要想是不是每項功能都測試到了，而且有時候真的想不太到有甚麼邊界條件需要測試，就很擔心會不會有甚麼沒發現的問題存在，實在也是蠻累人的，但同時也覺得真的要好好想過才能更理解我們需要學到的東西，所以測試的部分我覺得也是我需要加強的部分。

之前都會想說把每個部份分成很多個module寫真的會有比較好除錯嗎，但經過這次稍微大型一點的PA就能感覺到差別，要是把所有的都寫在一起，一定會找錯找到懷疑人生，這應該也是所有程式都適用的技巧。

整體來說這次project也是讓我學到了不少東西，希望下一個PA我可以更細心，這樣一定可以節省更多時間。