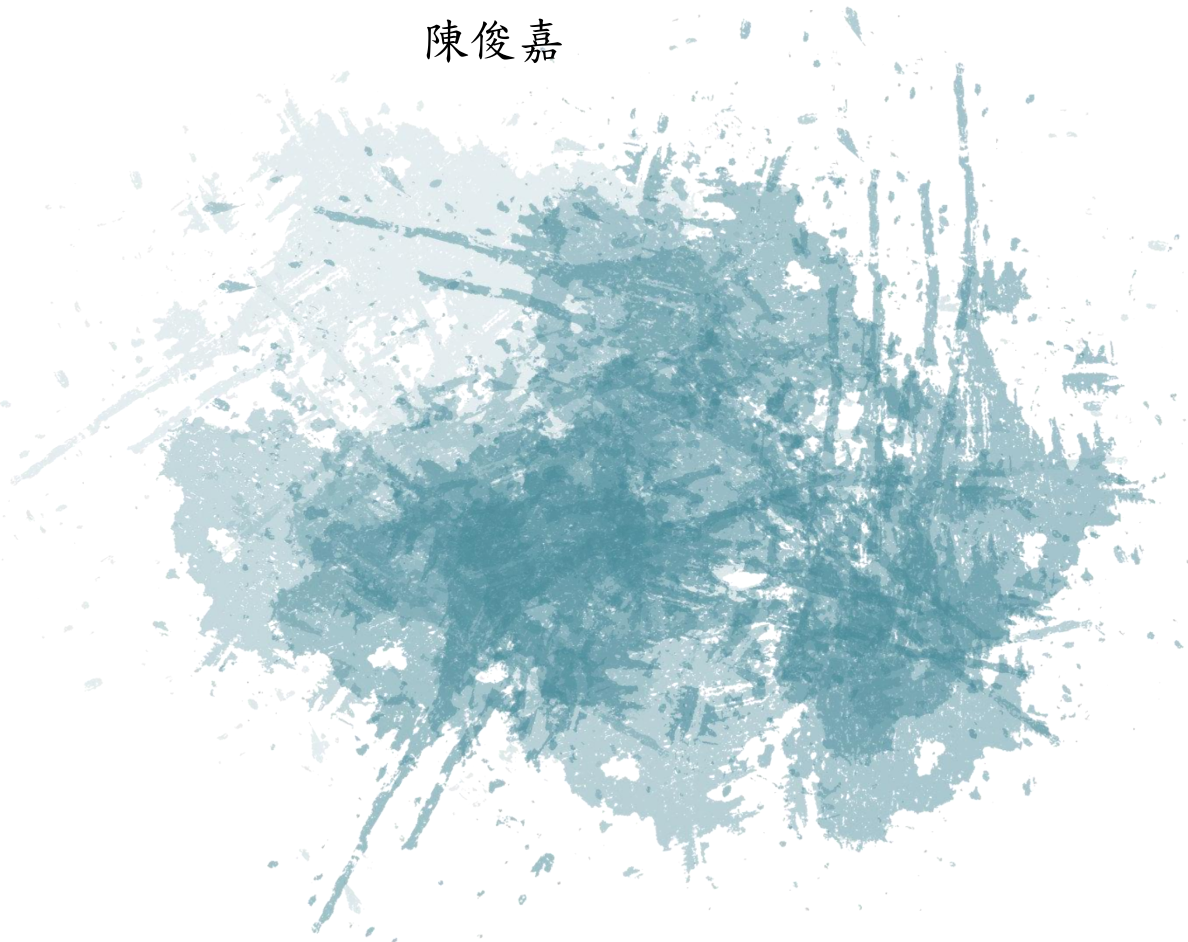
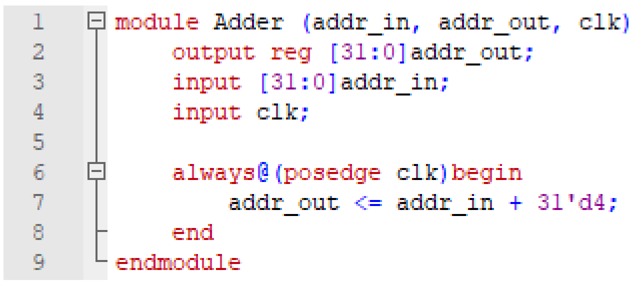
Computer Organization Project 2

四電機三甲
B10830009
陳俊嘉



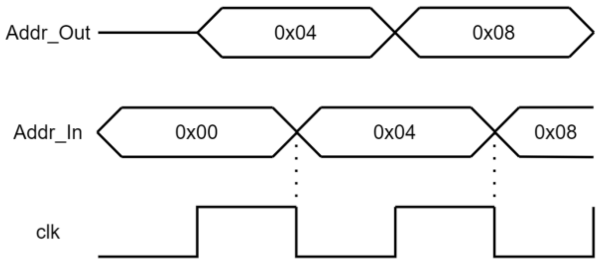
Part I: Single cycle processor with R-format instructions

- Adder Module

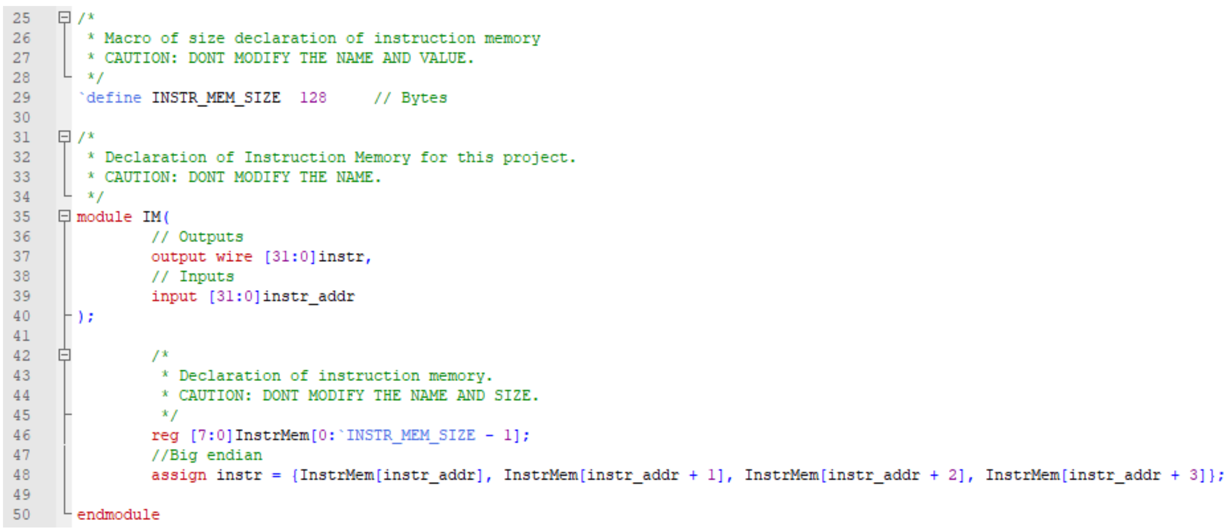


```verilog
module Adder (addr_in, addr_out, clk) ;
    output reg [31:0]addr_out;
    input [31:0]addr_in;
    input clk;

    always@(posedge clk)begin
        addr_out <= addr_in + 31'd4;
    end
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。



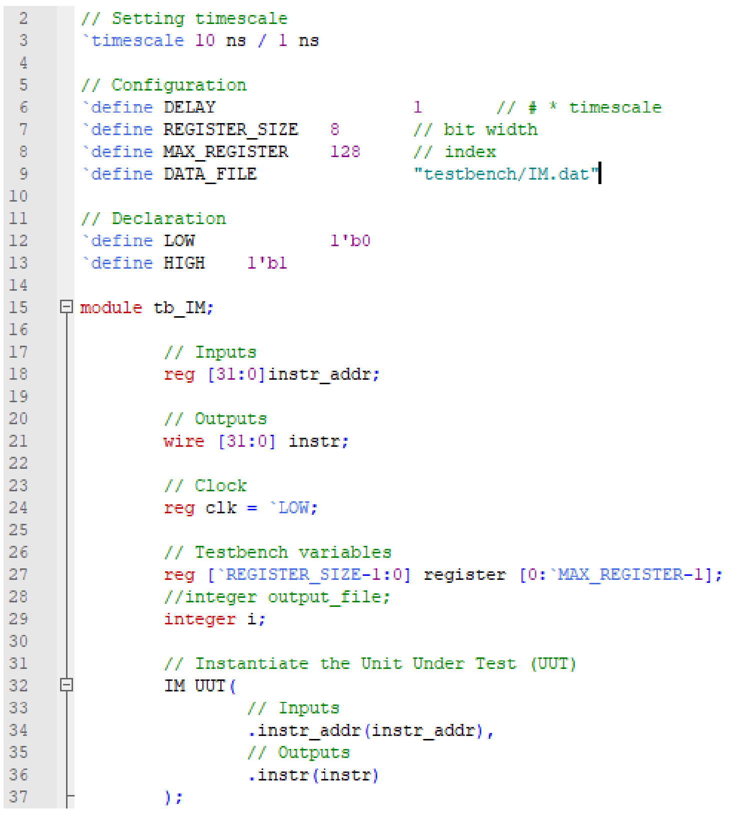
- Instruction Memory Module



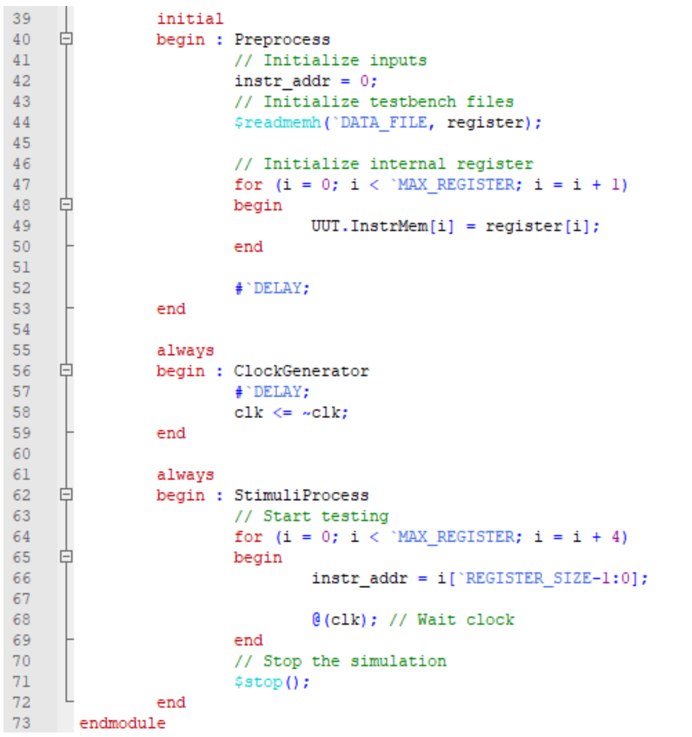
```verilog
/*
 * Macro of size declaration of instruction memory
 * CAUTION: DONT MODIFY THE NAME AND VALUE.
 */
`define INSTR_MEM_SIZE  128     // Bytes

/*
 * Declaration of Instruction Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module IM(
        // Outputs
        output wire [31:0]instr,
        // Inputs
        input [31:0]instr_addr
);

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

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

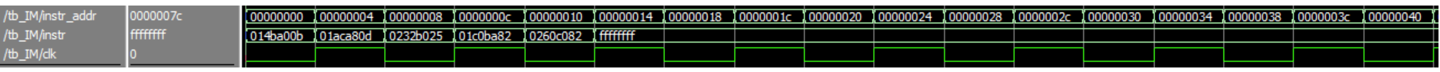


```verilog
// Setting timescale
`timescale 10 ns / 1 ns

// Configuration
`define DELAY                   1       // # * timescale
`define REGISTER_SIZE   8       // bit width
`define MAX_REGISTER    128     // index
`define DATA_FILE               "testbench/IM.dat"

// Declaration
`define LOW             1'b0
`define HIGH    1'b1

module tb_IM;

        // Inputs
        reg [31:0]instr_addr;

        // Outputs
        wire [31:0] instr;

        // Clock
        reg clk = `LOW;

        // Testbench variables
        reg [`REGISTER_SIZE-1:0] register [0:`MAX_REGISTER-1];
        //integer output_file;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        IM UUT(
                // Inputs
                .instr_addr(instr_addr),
                // Outputs
                .instr(instr)
        );
```

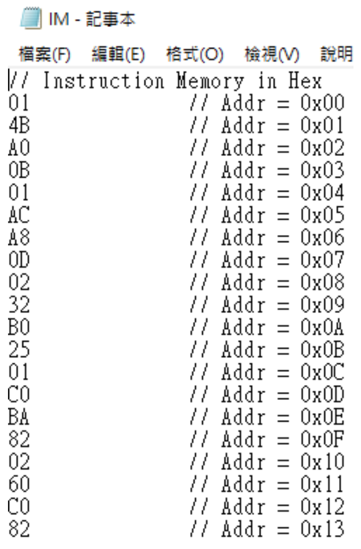


```verilog
        initial
        begin : Preprocess
                // Initialize inputs
                instr_addr = 0;
                // Initialize testbench files
                $readmemh(`DATA_FILE, register);

                // Initialize internal register
                for (i = 0; i < `MAX_REGISTER; i = i + 1)
                begin
                        UUT.InstrMem[i] = register[i];
                end

                #`DELAY;
        end

        always
        begin : ClockGenerator
                #`DELAY;
                clk <= ~clk;
        end

        always
        begin : StimuliProcess
                // Start testing
                for (i = 0; i < `MAX_REGISTER; i = i + 4)
                begin
                        instr_addr = i[`REGISTER_SIZE-1:0];

                        @(clk); // Wait clock
                end
                // Stop the simulation
                $stop();
        end
endmodule
```

- Instruction Memory Simulation



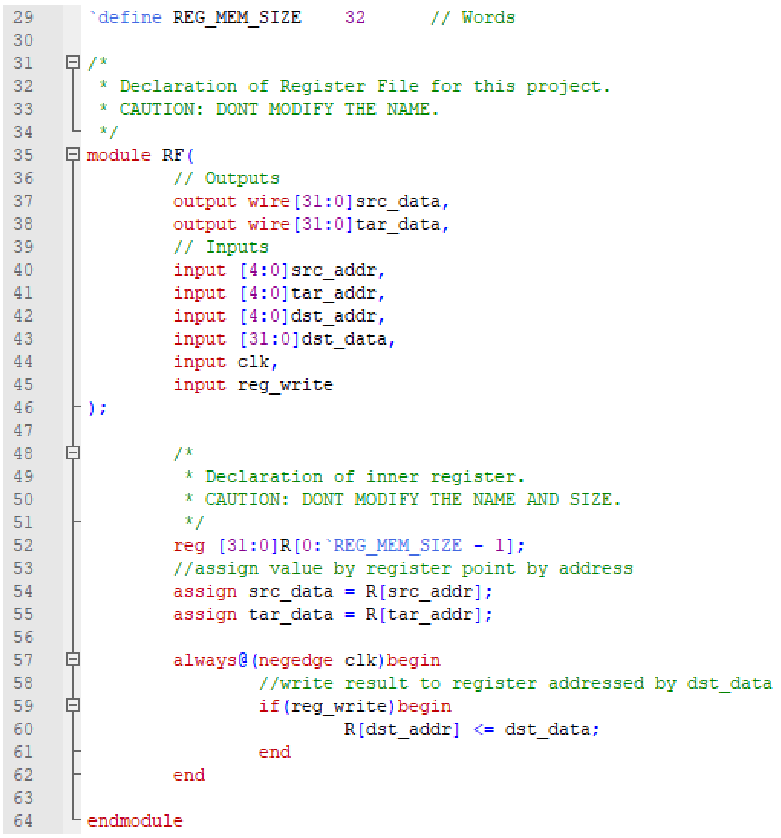
```
/tb_IM/instr_addr  0000007c  00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
/tb_IM/instr       ffffffff  014ba00b 01aca80d 0232b025 01c0ba82 0260c082 ffffffff
/tb_IM/clk         0
```



```
// 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address加四，輸出正常。

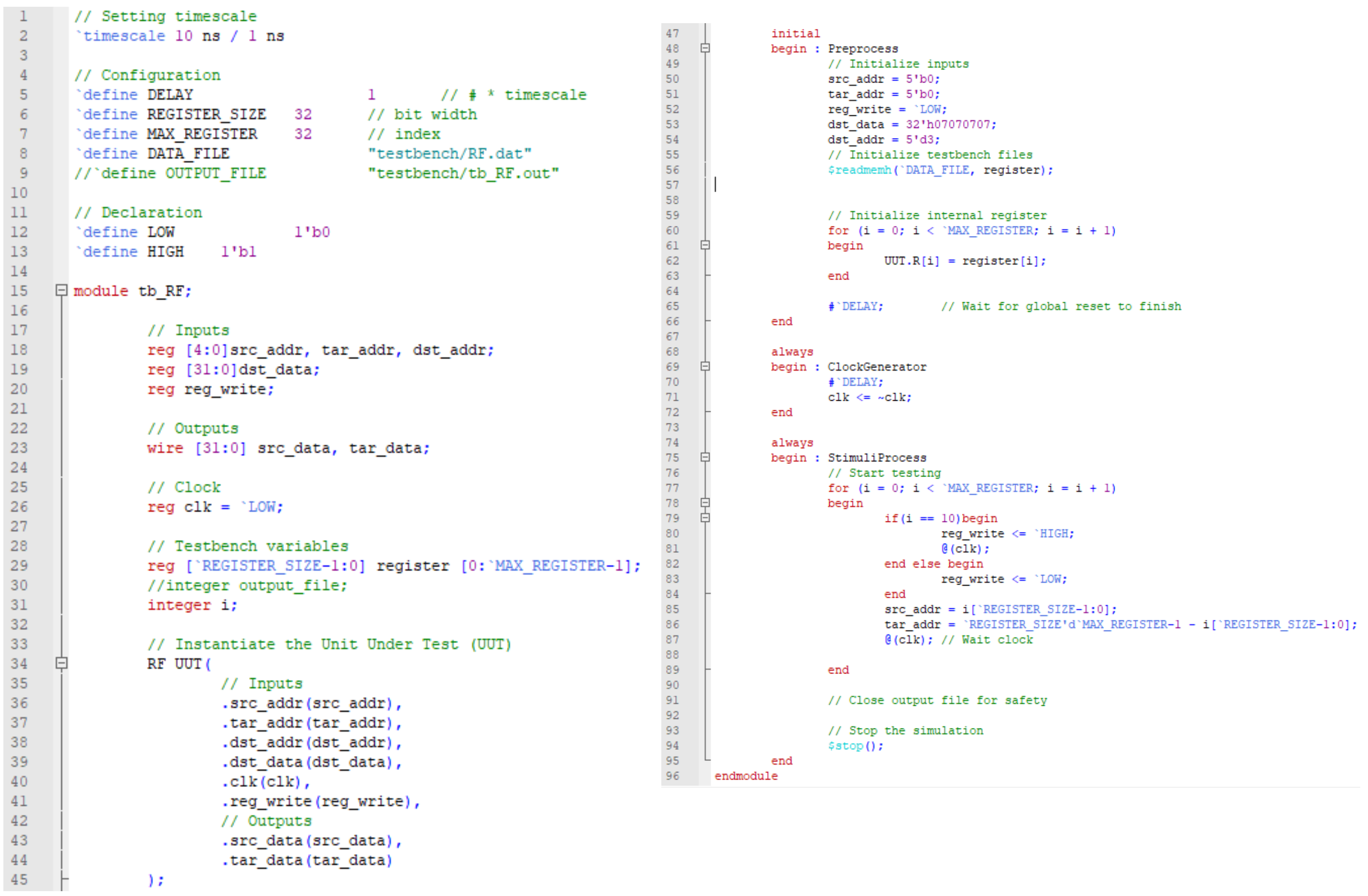
- Register File Module



```verilog
`define REG_MEM_SIZE    32      // Words

/*
 * Declaration of Register File for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module RF(
        // Outputs
        output wire[31:0]src_data,
        output wire[31:0]tar_data,
        // Inputs
        input [4:0]src_addr,
        input [4:0]tar_addr,
        input [4:0]dst_addr,
        input [31:0]dst_data,
        input clk,
        input reg_write
);

        /*
         * Declaration of inner register.
         * CAUTION: DONT MODIFY THE NAME AND SIZE.
         */
        reg [31:0]R[0:`REG_MEM_SIZE - 1];
        //assign value by register point by address
        assign src_data = R[src_addr];
        assign tar_data = R[tar_addr];

        always@(negedge clk)begin
                //write result to register addressed by dst_data
                if(reg_write)begin
                        R[dst_addr] <= dst_data;
                end
        end

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。

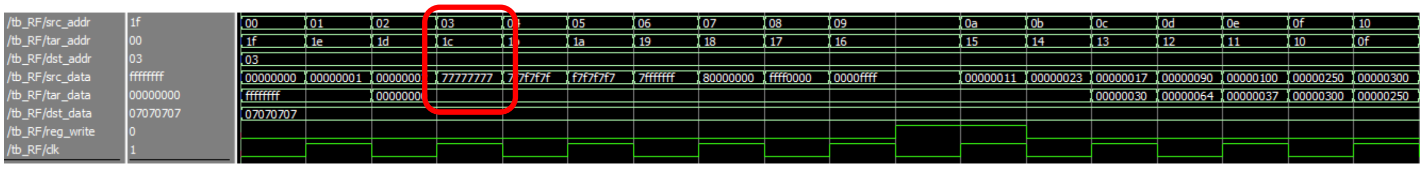
- Register File test bench

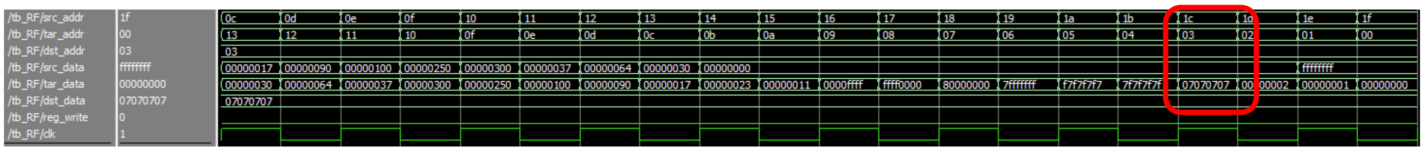


```verilog
// Setting timescale
`timescale 10 ns / 1 ns

// Configuration
`define DELAY                   1       // # * timescale
`define REGISTER_SIZE   32      // bit width
`define MAX_REGISTER    32      // index
`define DATA_FILE               "testbench/RF.dat"
//`define OUTPUT_FILE           "testbench/tb_RF.out"

// Declaration
`define LOW             1'b0
`define HIGH    1'b1

module tb_RF;

        // Inputs
        reg [4:0]src_addr, tar_addr, dst_addr;
        reg [31:0]dst_data;
        reg reg_write;

        // Outputs
        wire [31:0] src_data, tar_data;

        // Clock
        reg clk = `LOW;

        // Testbench variables
        reg [`REGISTER_SIZE-1:0] register [0:`MAX_REGISTER-1];
        //integer output_file;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        RF UUT(
                // Inputs
                .src_addr(src_addr),
                .tar_addr(tar_addr),
                .dst_addr(dst_addr),
                .dst_data(dst_data),
                .clk(clk),
                .reg_write(reg_write),
                // Outputs
                .src_data(src_data),
                .tar_data(tar_data)
        );

        initial
        begin : Preprocess
                // Initialize inputs
                src_addr = 5'b0;
                tar_addr = 5'b0;
                reg_write = `LOW;
                dst_data = 32'h07070707;
                dst_addr = 5'd3;
                // Initialize testbench files
                $readmemh(`DATA_FILE, register);


                // Initialize internal register
                for (i = 0; i < `MAX_REGISTER; i = i + 1)
                begin
                        UUT.R[i] = register[i];
                end

                #`DELAY;        // Wait for global reset to finish
        end

        always
        begin : ClockGenerator
                #`DELAY;
                clk <= ~clk;
        end

        always
        begin : StimuliProcess
                // Start testing
                for (i = 0; i < `MAX_REGISTER; i = i + 1)
                begin
                        if(i == 10)begin
                                reg_write <= `HIGH;
                                @(clk);
                        end else begin
                                reg_write <= `LOW;
                        end
                        src_addr = i[`REGISTER_SIZE-1:0];
                        tar_addr = `REGISTER_SIZE'd`MAX_REGISTER-1 - i[`REGISTER_SIZE-1:0];
                        @(clk); // Wait clock

                end

                // Close output file for safety

                // Stop the simulation
                $stop();
        end
endmodule
```

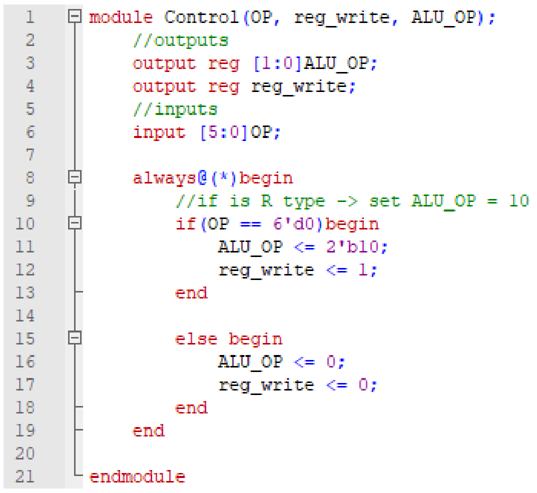
- ## Register File Simulation

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。





- ## Control Module

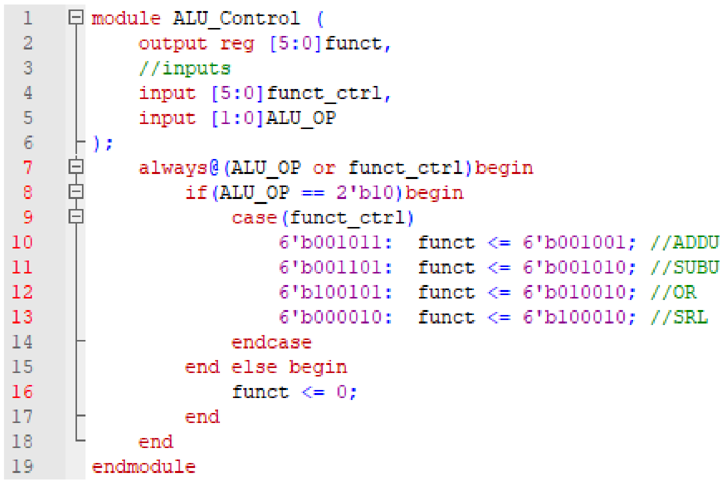


```
module Control(OP, reg_write, ALU_OP);
    //outputs
    output reg [1:0]ALU_OP;
    output reg reg_write;
    //inputs
    input [5:0]OP;

    always@(*)begin
        //if is R type -> set ALU_OP = 10
        if(OP == 6'd0)begin
            ALU_OP <= 2'b10;
            reg_write <= 1;
        end

        else begin
            ALU_OP <= 0;
            reg_write <= 0;
        end
    end

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

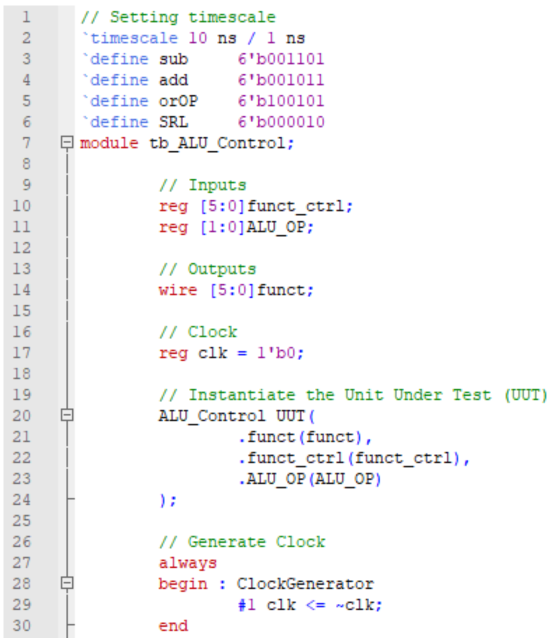
- ## ALU Control Module



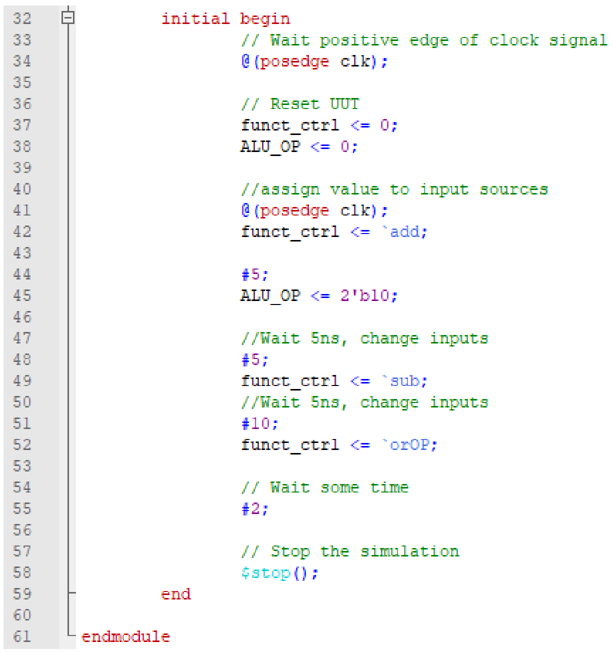
```
module ALU_Control (
    output reg [5:0]funct,
    //inputs
    input [5:0]funct_ctrl,
    input [1:0]ALU_OP
);
    always@(ALU_OP or funct_ctrl)begin
        if(ALU_OP == 2'b10)begin
            case(funct_ctrl)
                6'b001011:  funct <= 6'b001001; //ADDU
                6'b001101:  funct <= 6'b001010; //SUBU
                6'b100101:  funct <= 6'b010010; //OR
                6'b000010:  funct <= 6'b100010; //SRL
            endcase
        end else begin
            funct <= 0;
        end
    end
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

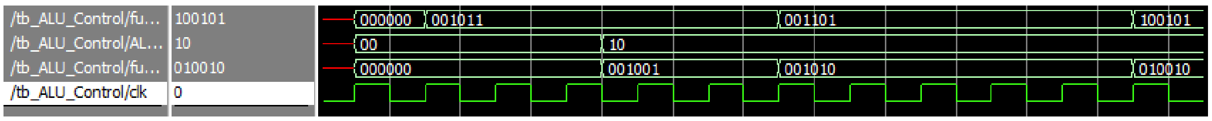


```verilog
// Setting timescale
`timescale 10 ns / 1 ns
`define sub     6'b001101
`define add     6'b001011
`define orOP    6'b100101
`define SRL     6'b000010
module tb_ALU_Control;

        // Inputs
        reg [5:0]funct_ctrl;
        reg [1:0]ALU_OP;

        // Outputs
        wire [5:0]funct;

        // Clock
        reg clk = 1'b0;

        // Instantiate the Unit Under Test (UUT)
        ALU_Control UUT(
                .funct(funct),
                .funct_ctrl(funct_ctrl),
                .ALU_OP(ALU_OP)
        );

        // Generate Clock
        always
        begin : ClockGenerator
                #1 clk <= ~clk;
        end
```



```verilog
        initial begin
                // Wait positive edge of clock signal
                @(posedge clk);

                // Reset UUT
                funct_ctrl <= 0;
                ALU_OP <= 0;

                //assign value to input sources
                @(posedge clk);
                funct_ctrl <= `add;

                #5;
                ALU_OP <= 2'b10;

                //Wait 5ns, change inputs
                #5;
                funct_ctrl <= `sub;
                //Wait 5ns, change inputs
                #10;
                funct_ctrl <= `orOP;

                // Wait some time
                #2;

                // Stop the simulation
                $stop();
        end
endmodule
```

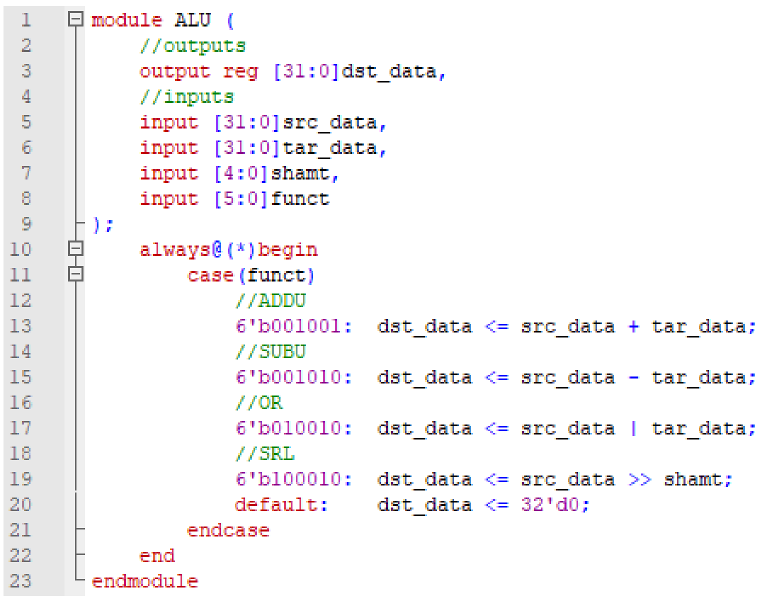
- ALU Control Simulation



```
/tb_ALU_Control/fu...   100101   000000  001011  001101  100101
/tb_ALU_Control/AL...   10       00  10
/tb_ALU_Control/fu...   010010   000000  001001  001010  010010
/tb_ALU_Control/clk     0
```

Function control判斷為10後，function輸出皆正常。

- ALU Module

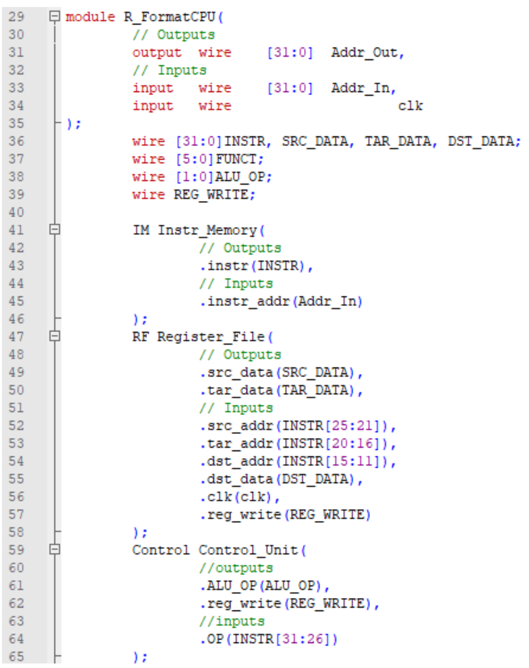


```verilog
module ALU (
    //outputs
    output reg [31:0]dst_data,
    //inputs
    input [31:0]src_data,
    input [31:0]tar_data,
    input [4:0]shamt,
    input [5:0]funct
);
    always@(*)begin
        case(funct)
            //ADDU
            6'b001001:  dst_data <= src_data + tar_data;
            //SUBU
            6'b001010:  dst_data <= src_data - tar_data;
            //OR
            6'b010010:  dst_data <= src_data | tar_data;
            //SRL
            6'b100010:  dst_data <= src_data >> shamt;
            default:    dst_data <= 32'd0;
        endcase
    end
endmodule
```

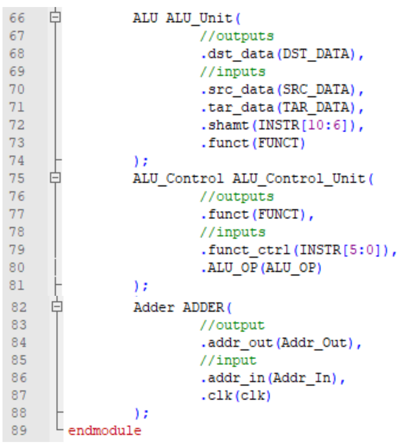
ALU收到由ALU Controller傳來的function後，進行運算後將結果傳至destination data中，再交由Register File處存至相對應的位置。

在HW1與PA1中都有做過ALU，這次就不在做test bench

- ## R_Format CPU

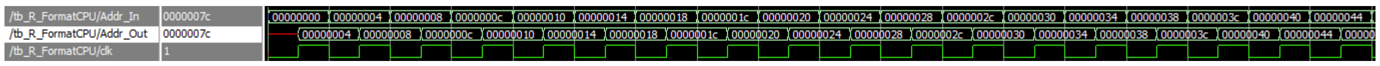


```verilog
module R_FormatCPU(
        // Outputs
        output  wire    [31:0]  Addr_Out,
        // Inputs
        input   wire    [31:0]  Addr_In,
        input   wire                    clk
);
        wire [31:0]INSTR, SRC_DATA, TAR_DATA, DST_DATA;
        wire [5:0]FUNCT;
        wire [1:0]ALU_OP;
        wire REG_WRITE;

        IM Instr_Memory(
                // Outputs
                .instr(INSTR),
                // Inputs
                .instr_addr(Addr_In)
        );
        RF Register_File(
                // Outputs
                .src_data(SRC_DATA),
                .tar_data(TAR_DATA),
                // Inputs
                .src_addr(INSTR[25:21]),
                .tar_addr(INSTR[20:16]),
                .dst_addr(INSTR[15:11]),
                .dst_data(DST_DATA),
                .clk(clk),
                .reg_write(REG_WRITE)
        );
        Control Control_Unit(
                //outputs
                .ALU_OP(ALU_OP),
                .reg_write(REG_WRITE),
                //inputs
                .OP(INSTR[31:26])
        );
```

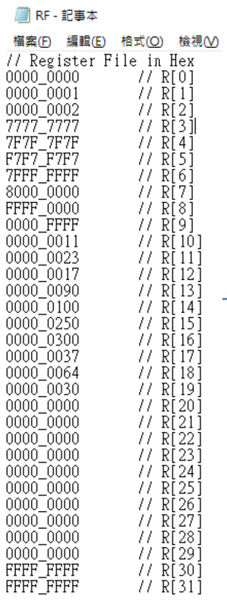


```verilog
        ALU ALU_Unit(
                //outputs
                .dst_data(DST_DATA),
                //inputs
                .src_data(SRC_DATA),
                .tar_data(TAR_DATA),
                .shamt(INSTR[10:6]),
                .funct(FUNCT)
        );
        ALU_Control ALU_Control_Unit(
                //outputs
                .funct(FUNCT),
                //inputs
                .funct_ctrl(INSTR[5:0]),
                .ALU_OP(ALU_OP)
        );
        Adder ADDER(
                //output
                .addr_out(Addr_Out),
                //input
                .addr_in(Addr_In),
                .clk(clk)
        );
endmodule
```

- ## R_Format CPU Simulation



```
/tb_R_FormatCPU/Addr_In   0000007c  00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040 00000044
/tb_R_FormatCPU/Addr_Out  0000007c  00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040 00000044 00000
/tb_R_FormatCPU/clk       1
```



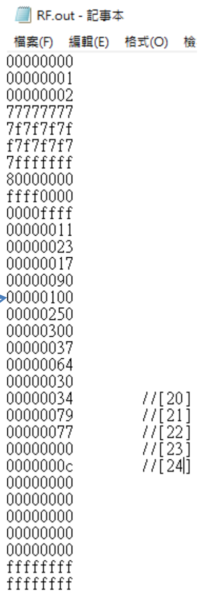
RF - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V)

```
// 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]
```

RF.DAT

RF.OUT



RF.out - 記事本

檔案(F) 編輯(E) 格式(O) 檢

```
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
ffffffff
ffffffff
```

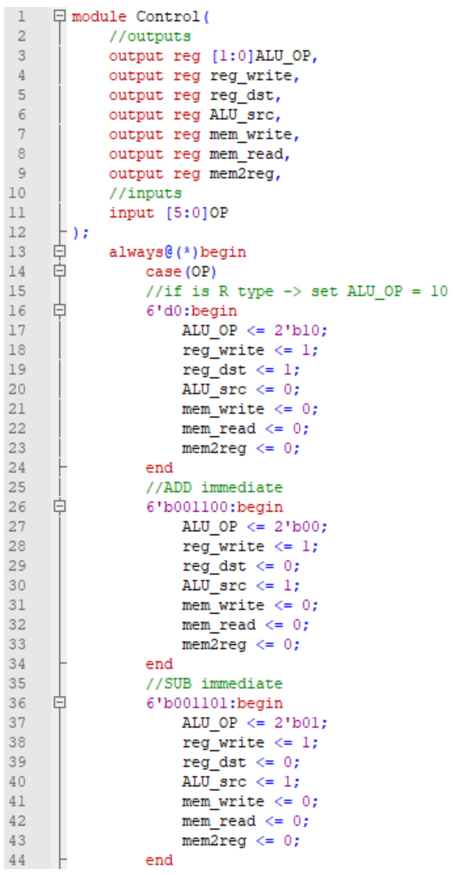
Instructions

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

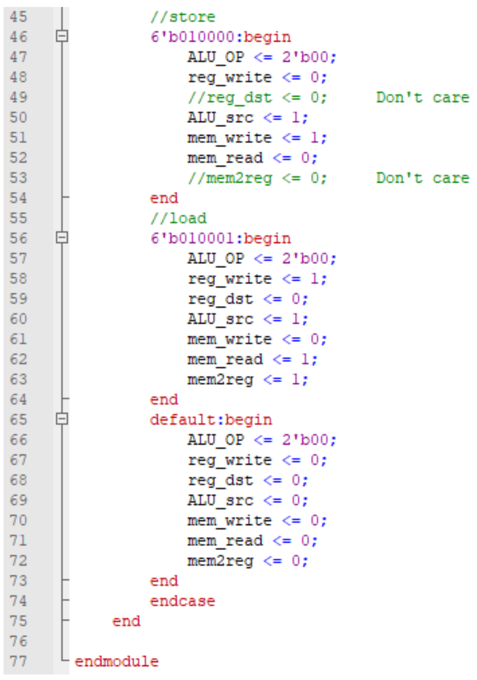
Part I: Single cycle processor with R-format and I-format instructions

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

- Control Module

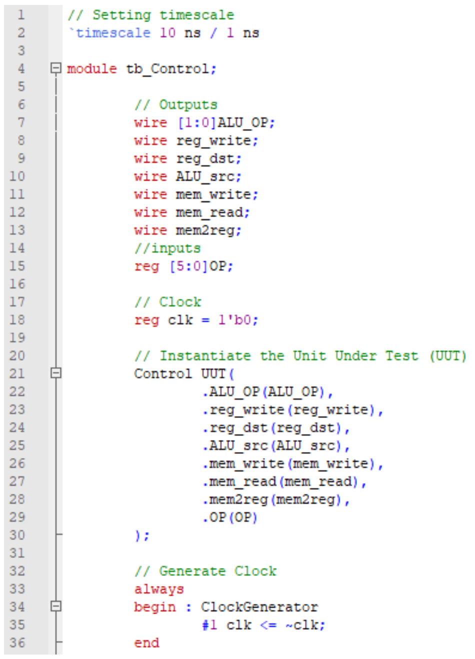


```verilog
module Control(
    //outputs
    output reg [1:0]ALU_OP,
    output reg reg_write,
    output reg reg_dst,
    output reg ALU_src,
    output reg mem_write,
    output reg mem_read,
    output reg mem2reg,
    //inputs
    input [5:0]OP
);
    always@(*)begin
        case(OP)
        //if is R type -> set ALU_OP = 10
        6'd0:begin
            ALU_OP <= 2'b10;
            reg_write <= 1;
            reg_dst <= 1;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
        end
        //ADD immediate
        6'b001100:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
        end
        //SUB immediate
        6'b001101:begin
            ALU_OP <= 2'b01;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
        end
```

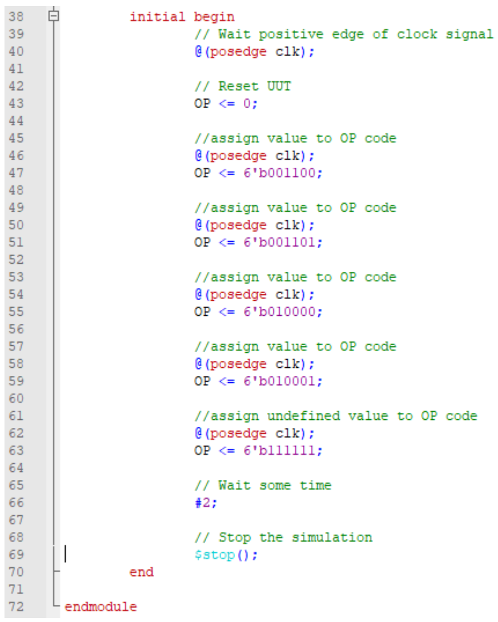


```verilog
        //store
        6'b010000:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            //reg_dst <= 0;     Don't care
            ALU_src <= 1;
            mem_write <= 1;
            mem_read <= 0;
            //mem2reg <= 0;     Don't care
        end
        //load
        6'b010001:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 1;
            mem2reg <= 1;
        end
        default:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            reg_dst <= 0;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
        end
        endcase
    end

endmodule
```

- Control test bench

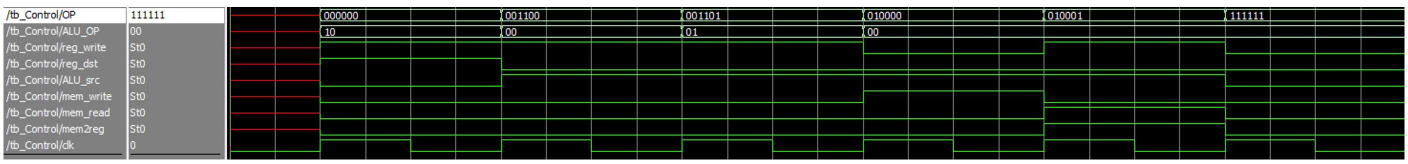


```verilog
// Setting timescale
`timescale 10 ns / 1 ns

module tb_Control;

        // Outputs
        wire [1:0]ALU_OP;
        wire reg_write;
        wire reg_dst;
        wire ALU_src;
        wire mem_write;
        wire mem_read;
        wire mem2reg;
        //inputs
        reg [5:0]OP;

        // Clock
        reg clk = 1'b0;

        // Instantiate the Unit Under Test (UUT)
        Control UUT(
                .ALU_OP(ALU_OP),
                .reg_write(reg_write),
                .reg_dst(reg_dst),
                .ALU_src(ALU_src),
                .mem_write(mem_write),
                .mem_read(mem_read),
                .mem2reg(mem2reg),
                .OP(OP)
        );

        // Generate Clock
        always
        begin : ClockGenerator
                #1 clk <= ~clk;
        end
```



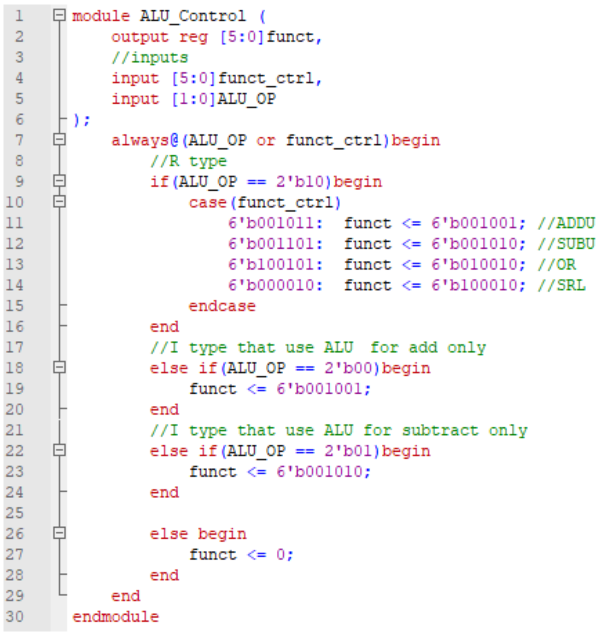
```verilog
        initial begin
                // Wait positive edge of clock signal
                @(posedge clk);

                // Reset UUT
                OP <= 0;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b001100;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b001101;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b010000;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b010001;

                //assign undefined value to OP code
                @(posedge clk);
                OP <= 6'b111111;

                // Wait some time
                #2;

                // Stop the simulation
                $stop();
        end

endmodule
```

- Control Simulation



各訊號都依OP code所對應的條件正常設定，沒有對應的OP code也依default condition將全部訊號設為零。

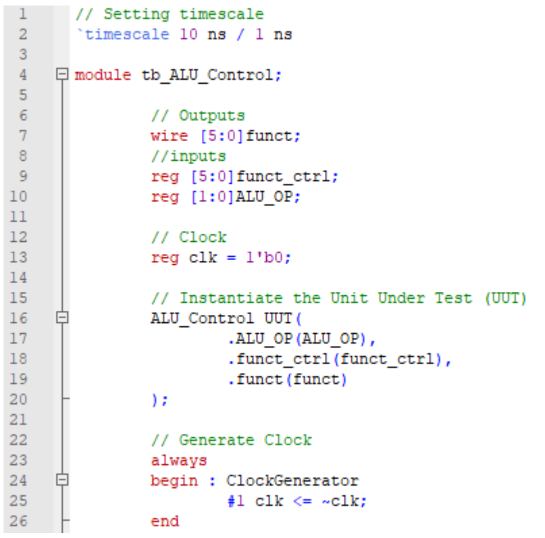
- ALU Control Module



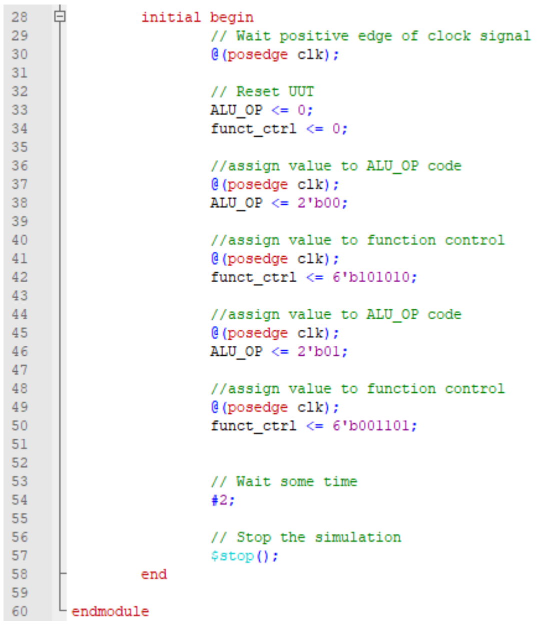
```verilog
module ALU_Control (
    output reg [5:0]funct,
    //inputs
    input [5:0]funct_ctrl,
    input [1:0]ALU_OP
);
    always@(ALU_OP or funct_ctrl)begin
        //R type
        if(ALU_OP == 2'b10)begin
            case(funct_ctrl)
                6'b001011:  funct <= 6'b001001; //ADDU
                6'b001101:  funct <= 6'b001010; //SUBU
                6'b100101:  funct <= 6'b010010; //OR
                6'b000010:  funct <= 6'b100010; //SRL
            endcase
        end
        //I type that use ALU  for add only
        else if(ALU_OP == 2'b00)begin
            funct <= 6'b001001;
        end
        //I type that use ALU for subtract only
        else if(ALU_OP == 2'b01)begin
            funct <= 6'b001010;
        end

        else begin
            funct <= 0;
        end
    end
endmodule
```

I-type instruction沒有function control的訊號，因為instruction後面16 bits都是immediate，所以由controller判斷OP code後就由相對應的ALU_OP給ALU其function。

- ALU Control test bench

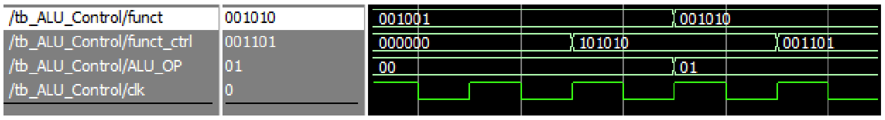


```verilog
// Setting timescale
`timescale 10 ns / 1 ns

module tb_ALU_Control;

        // Outputs
        wire [5:0]funct;
        //inputs
        reg [5:0]funct_ctrl;
        reg [1:0]ALU_OP;

        // Clock
        reg clk = 1'b0;

        // Instantiate the Unit Under Test (UUT)
        ALU_Control UUT(
                .ALU_OP(ALU_OP),
                .funct_ctrl(funct_ctrl),
                .funct(funct)
        );

        // Generate Clock
        always
        begin : ClockGenerator
                #1 clk <= ~clk;
        end
```



```verilog
        initial begin
                // Wait positive edge of clock signal
                @(posedge clk);

                // Reset UUT
                ALU_OP <= 0;
                funct_ctrl <= 0;

                //assign value to ALU_OP code
                @(posedge clk);
                ALU_OP <= 2'b00;

                //assign value to function control
                @(posedge clk);
                funct_ctrl <= 6'b101010;

                //assign value to ALU_OP code
                @(posedge clk);
                ALU_OP <= 2'b01;

                //assign value to function control
                @(posedge clk);
                funct_ctrl <= 6'b001101;


                // Wait some time
                #2;

                // Stop the simulation
                $stop();
        end

endmodule
```

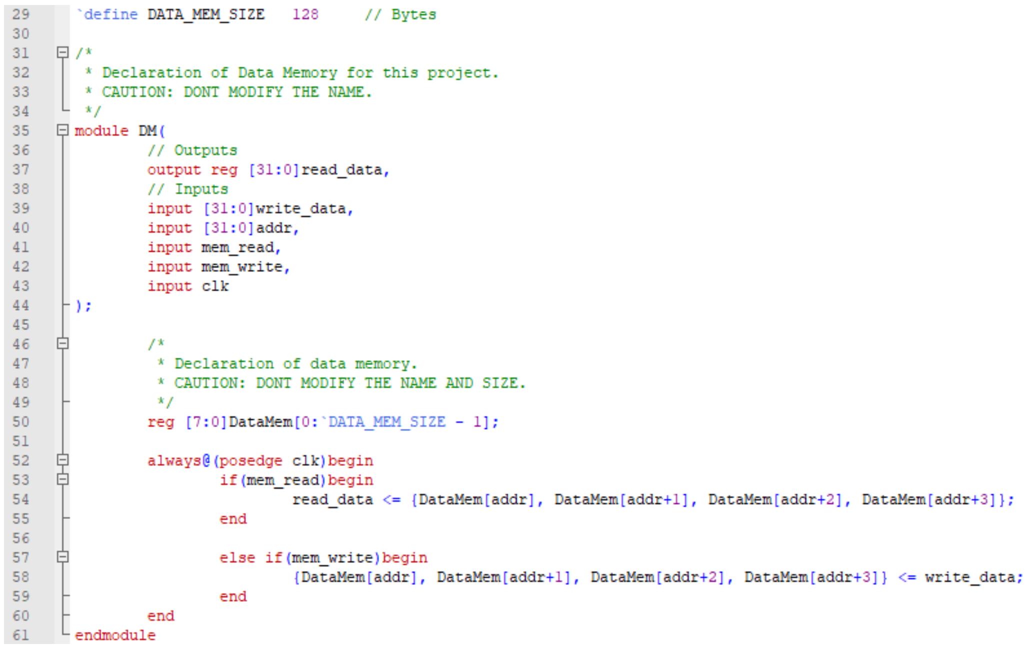
- ALU Control Simulation



```
/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/clk          0
```

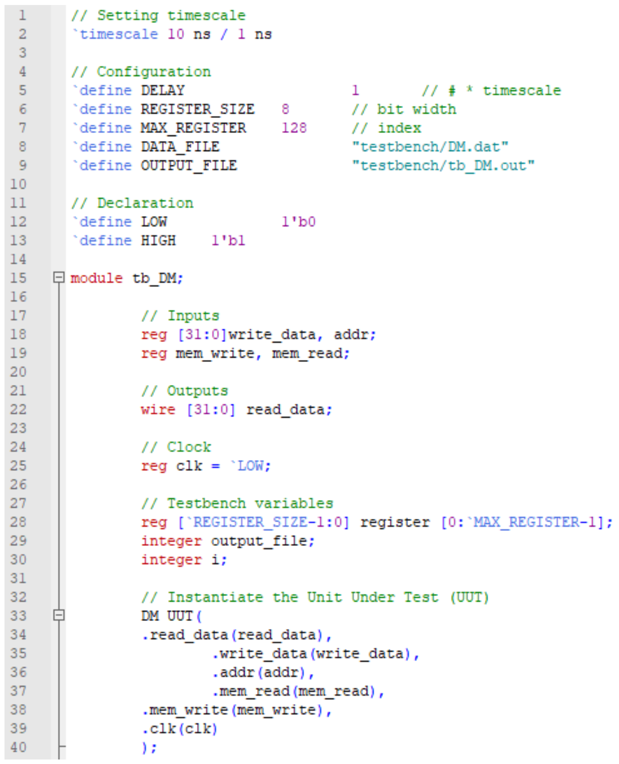
I-type的ALU_OP對應的輸出正常，且不受funct_ctrl訊號影響

- Data Memory Module

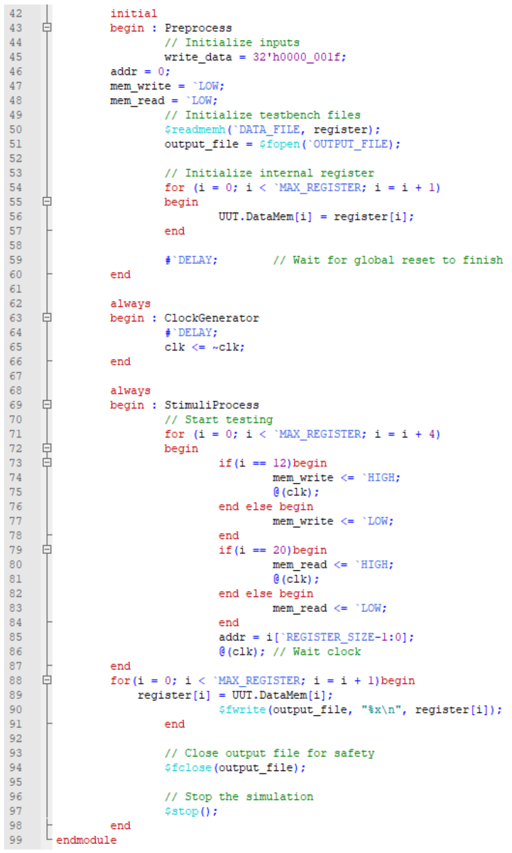


```verilog
`define DATA_MEM_SIZE   128     // Bytes

/*
 * Declaration of Data Memory for this project.
 * CAUTION: DONT MODIFY THE NAME.
 */
module DM(
        // Outputs
        output reg [31:0]read_data,
        // Inputs
        input  [31:0]write_data,
        input  [31:0]addr,
        input mem_read,
        input mem_write,
        input clk
);

        /*
         * Declaration of data memory.
         * CAUTION: DONT MODIFY THE NAME AND SIZE.
         */
        reg [7:0]DataMem[0:`DATA_MEM_SIZE - 1];

        always@(posedge clk)begin
                if(mem_read)begin
                        read_data <= {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]};
                end

                else if(mem_write)begin
                        {DataMem[addr], DataMem[addr+1], DataMem[addr+2], DataMem[addr+3]} <= write_data;
                end
        end
endmodule
```

- Data Memory test bench

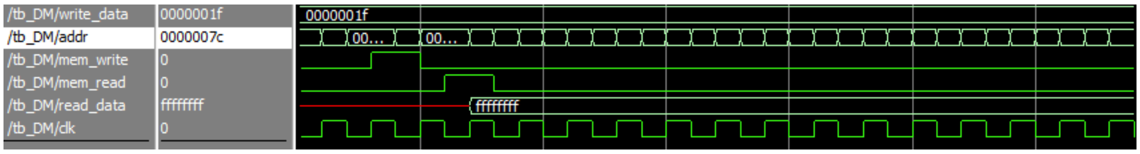


```verilog
// Setting timescale
`timescale 10 ns / 1 ns

// Configuration
`define DELAY                   1       // # * timescale
`define REGISTER_SIZE   8       // bit width
`define MAX_REGISTER    128     // index
`define DATA_FILE               "testbench/DM.dat"
`define OUTPUT_FILE             "testbench/tb_DM.out"

// Declaration
`define LOW             1'b0
`define HIGH    1'b1

module tb_DM;

        // Inputs
        reg [31:0]write_data, addr;
        reg mem_write, mem_read;

        // Outputs
        wire [31:0] read_data;

        // Clock
        reg clk = `LOW;

        // Testbench variables
        reg [`REGISTER_SIZE-1:0] register [0:`MAX_REGISTER-1];
        integer output_file;
        integer i;

        // Instantiate the Unit Under Test (UUT)
        DM UUT(
        .read_data(read_data),
                .write_data(write_data),
                .addr(addr),
                .mem_read(mem_read),
        .mem_write(mem_write),
        .clk(clk)
        );
```

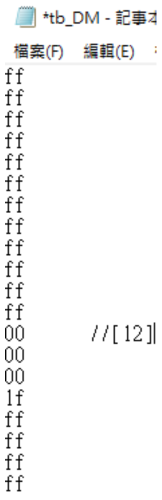


```verilog
        initial
        begin : Preprocess
                // Initialize inputs
                write_data = 32'h0000_001f;
        addr = 0;
        mem_write = `LOW;
        mem_read = `LOW;
                // Initialize testbench files
                $readmemh(`DATA_FILE, register);
                output_file = $fopen(`OUTPUT_FILE);

                // Initialize internal register
                for (i = 0; i < `MAX_REGISTER; i = i + 1)
                begin
                        UUT.DataMem[i] = register[i];
                end

                #`DELAY;        // Wait for global reset to finish
        end

        always
        begin : ClockGenerator
                #`DELAY;
                clk <= ~clk;
        end

        always
        begin : StimuliProcess
                // Start testing
                for (i = 0; i < `MAX_REGISTER; i = i + 4)
                begin
                        if(i == 12)begin
                                mem_write <= `HIGH;
                                @(clk);
                        end else begin
                                mem_write <= `LOW;
                        end
                        if(i == 20)begin
                                mem_read <= `HIGH;
                                @(clk);
                        end else begin
                                mem_read <= `LOW;
                        end
                        addr = i[`REGISTER_SIZE-1:0];
                        @(clk); // Wait clock
        end
        for(i = 0; i < `MAX_REGISTER; i = i + 1)begin
            register[i] = UUT.DataMem[i];
                        $fwrite(output_file, "%x\n", register[i]);
                end

                // Close output file for safety
                $fclose(output_file);

                // Stop the simulation
                $stop();
        end
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
```

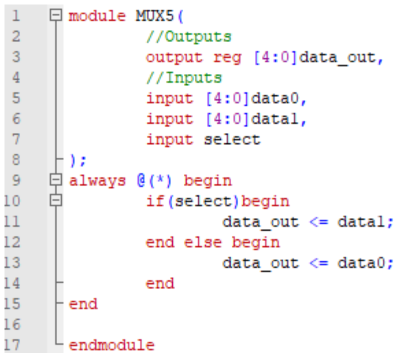


*tb_DM - 記事本

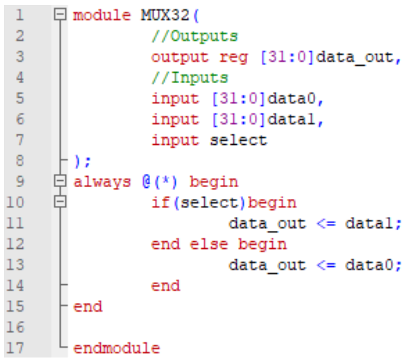
```
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
00       //[12]
00
00
1f
ff
ff
ff
ff
```

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。

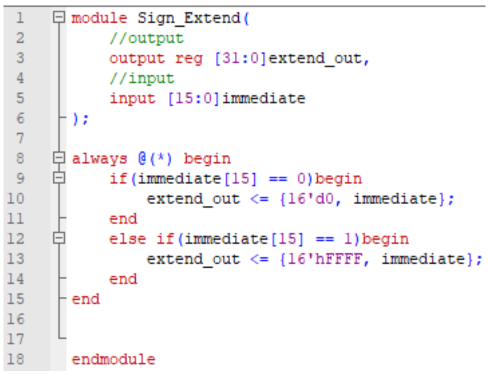
- Multiplexer & Sing Extension Module



```verilog
module MUX5(
        //Outputs
        output reg [4:0]data_out,
        //Inputs
        input [4:0]data0,
        input [4:0]data1,
        input select
);
always @(*) begin
        if(select)begin
                data_out <= data1;
        end else begin
                data_out <= data0;
        end
end

endmodule
```



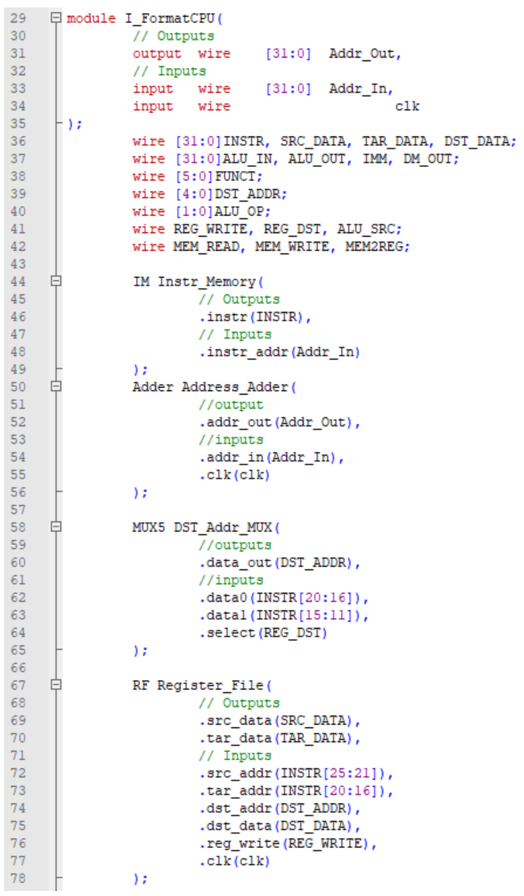
```verilog
module MUX32(
        //Outputs
        output reg [31:0]data_out,
        //Inputs
        input [31:0]data0,
        input [31:0]data1,
        input select
);
always @(*) begin
        if(select)begin
                data_out <= data1;
        end else begin
                data_out <= data0;
        end
end

endmodule
```



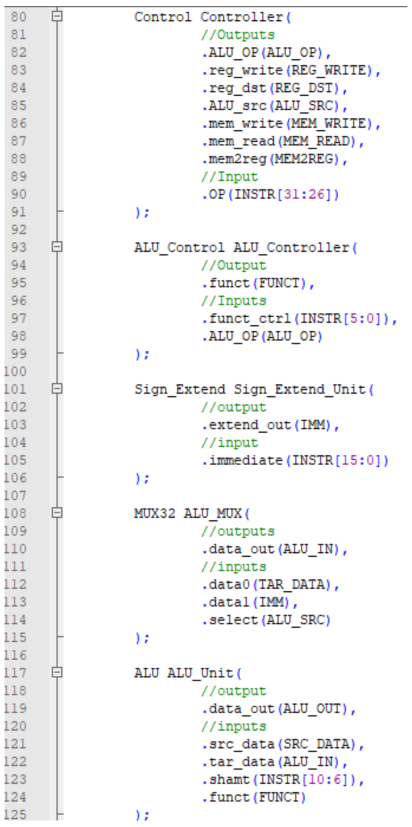
```verilog
module Sign_Extend(
        //output
        output reg [31:0]extend_out,
        //input
        input [15:0]immediate
);

always @(*) begin
        if(immediate[15] == 0)begin
                extend_out <= {16'd0, immediate};
        end
        else if(immediate[15] == 1)begin
                extend_out <= {16'hFFFF, immediate};
        end
end


endmodule
```

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。

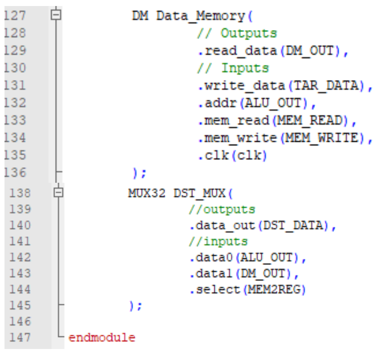
• I_Format CPU



```
module I_FormatCPU(
        // Outputs
        output  wire    [31:0]  Addr_Out,
        // Inputs
        input   wire    [31:0]  Addr_In,
        input   wire                    clk
);

        wire [31:0]INSTR, SRC_DATA, TAR_DATA, DST_DATA;
        wire [31:0]ALU_IN, ALU_OUT, IMM, DM_OUT;
        wire [5:0]FUNCT;
        wire [4:0]DST_ADDR;
        wire [1:0]ALU_OP;
        wire REG_WRITE, REG_DST, ALU_SRC;
        wire MEM_READ, MEM_WRITE, MEM2REG;

        IM Instr_Memory(
                // Outputs
                .instr(INSTR),
                // Inputs
                .instr_addr(Addr_In)
        );
        Adder Address_Adder(
                //output
                .addr_out(Addr_Out),
                //inputs
                .addr_in(Addr_In),
                .clk(clk)
        );

        MUX5 DST_Addr_MUX(
                //outputs
                .data_out(DST_ADDR),
                //inputs
                .data0(INSTR[20:16]),
                .data1(INSTR[15:11]),
                .select(REG_DST)
        );

        RF Register_File(
                // Outputs
                .src_data(SRC_DATA),
                .tar_data(TAR_DATA),
                // Inputs
                .src_addr(INSTR[25:21]),
                .tar_addr(INSTR[20:16]),
                .dst_addr(DST_ADDR),
                .dst_data(DST_DATA),
                .reg_write(REG_WRITE),
                .clk(clk)
        );
```

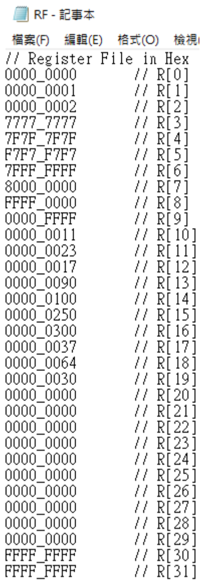


```
        Control Controller(
                //Outputs
                .ALU_OP(ALU_OP),
                .reg_write(REG_WRITE),
                .reg_dst(REG_DST),
                .ALU_src(ALU_SRC),
                .mem_write(MEM_WRITE),
                .mem_read(MEM_READ),
                .mem2reg(MEM2REG),
                //Input
                .OP(INSTR[31:26])
        );

        ALU_Control ALU_Controller(
                //Output
                .funct(FUNCT),
                //Inputs
                .funct_ctrl(INSTR[5:0]),
                .ALU_OP(ALU_OP)
        );

        Sign_Extend Sign_Extend_Unit(
                //output
                .extend_out(IMM),
                //input
                .immediate(INSTR[15:0])
        );

        MUX32 ALU_MUX(
                //outputs
                .data_out(ALU_IN),
                //inputs
                .data0(TAR_DATA),
                .data1(IMM),
                .select(ALU_SRC)
        );

        ALU ALU_Unit(
                //output
                .data_out(ALU_OUT),
                //inputs
                .src_data(SRC_DATA),
                .tar_data(ALU_IN),
                .shamt(INSTR[10:6]),
                .funct(FUNCT)
        );
```



```
        DM Data_Memory(
                // Outputs
                .read_data(DM_OUT),
                // Inputs
                .write_data(TAR_DATA),
                .addr(ALU_OUT),
                .mem_read(MEM_READ),
                .mem_write(MEM_WRITE),
                .clk(clk)
        );
        MUX32 DST_MUX(
                //outputs
                .data_out(DST_DATA),
                //inputs
                .data0(ALU_OUT),
                .data1(DM_OUT),
                .select(MEM2REG)
        );

endmodule
```

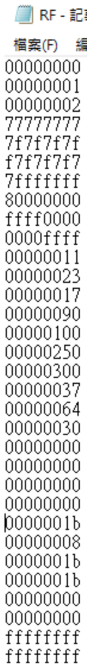
• I_Format CPU Simulation



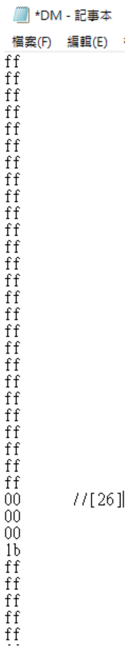
```
// Register File in Hex
0000_0000       // R[0]
0000_0001       // R[1]
0000_0002       // R[2]
7777_7777       // R[3]
7F7F_7F7F       // R[4]
F7F7_F7F7       // R[5]
7FFF_FFFF       // R[6]
8000_0000       // R[7]
FFFF_0000       // R[8]
0000_FFFF       // R[9]
0000_0011       // R[10]
0000_0023       // 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]
```

RF.DAT

RF.OUT
DM.OUT



```
00000000
00000001
00000002
77777777
7f7f7f7f
f7f7f7f7
7fffffff
80000000
ffff0000
0000ffff
00000011
00000023
00000017
00000090
00000100
00000250
00000300
00000037
00000064
00000030
00000000
00000000
00000000
00000000
0000001b
00000008
0000001b
0000001b
00000000
00000000
ffffffff
ffffffff
```



```
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
ff
00          //[26]
00
00
1b
ff
ff
ff
ff
ff
```

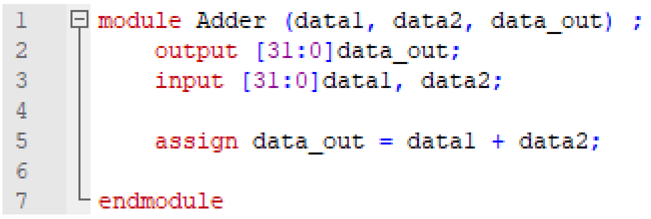
Instructions

Hexadecimal	Binary	Meaning
3158000A	001100_01010_11000_0000000000001010	\$R24 = \$R10 + 10
35590009	001101_01010_11001_0000000000001001	\$R25 = \$R10 – 9
41980003	010000_01100_11000_0000000000000011	Mem[\$R12 + 3] = \$R24
459A0003	010001_01100_11010_0000000000000011	\$R26 = Mem[\$R12 + 3]
315B000A	001100_01010_11011_0000000000001010	\$R27 = \$R10 + 10

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

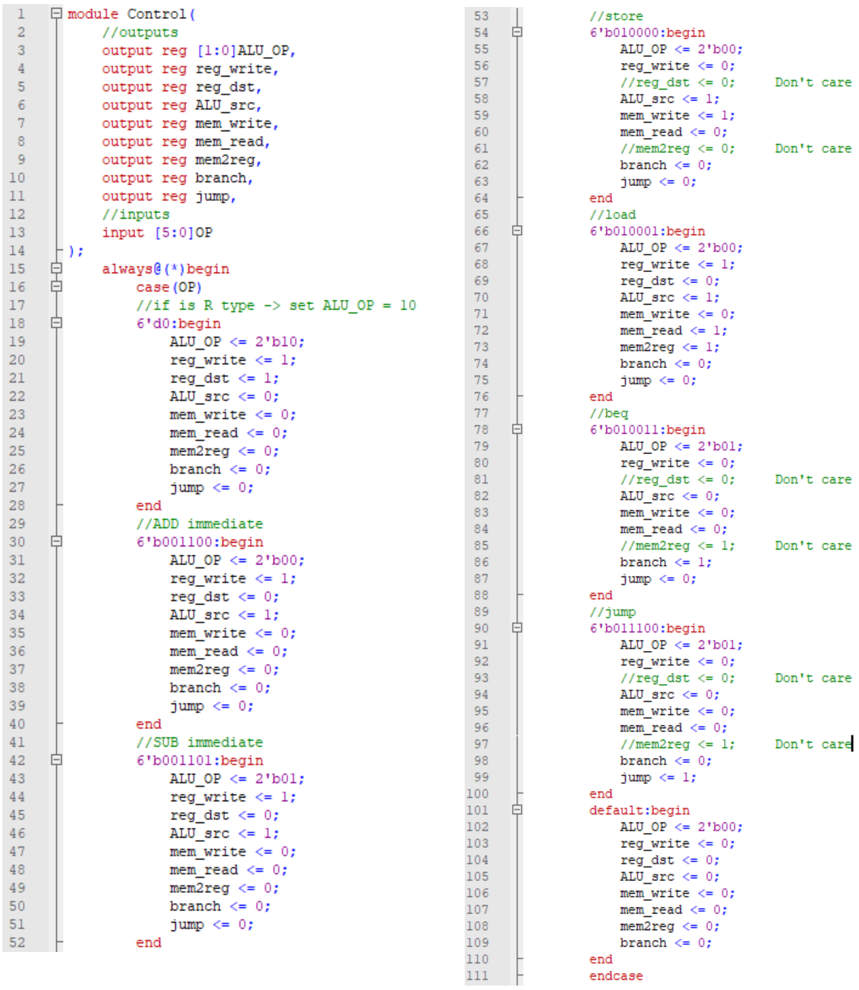
- Adder Module



```
module Adder (data1, data2, data_out) ;
    output [31:0]data_out;
    input [31:0]data1, data2;

    assign data_out = data1 + data2;

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。

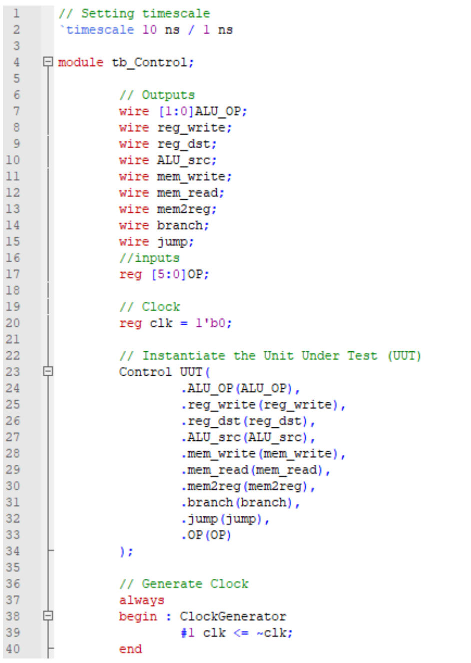
- Control Module



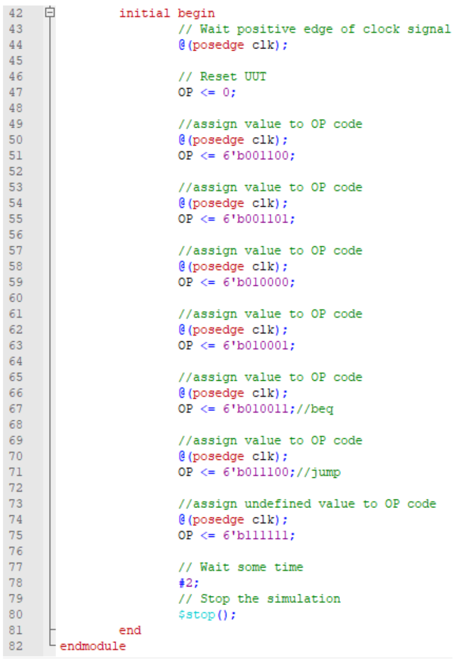
```
module Control(
    //outputs
    output reg [1:0]ALU_OP,
    output reg reg_write,
    output reg reg_dst,
    output reg ALU_src,
    output reg mem_write,
    output reg mem_read,
    output reg mem2reg,
    output reg branch,
    output reg jump,
    //inputs
    input [5:0]OP
);
    always@(*)begin
        case(OP)
        //if is R type -> set ALU_OP = 10
        6'd0:begin
            ALU_OP <= 2'b10;
            reg_write <= 1;
            reg_dst <= 1;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //ADD immediate
        6'b001100:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //SUB immediate
        6'b001101:begin
            ALU_OP <= 2'b01;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
            jump <= 0;
        end
        //store
        6'b010000:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            //reg_dst <= 0;     Don't care
            ALU_src <= 1;
            mem_write <= 1;
            mem_read <= 0;
            //mem2reg <= 0;     Don't care
            branch <= 0;
            jump <= 0;
        end
        //load
        6'b010001:begin
            ALU_OP <= 2'b00;
            reg_write <= 1;
            reg_dst <= 0;
            ALU_src <= 1;
            mem_write <= 0;
            mem_read <= 1;
            mem2reg <= 1;
            branch <= 0;
            jump <= 0;
        end
        //beq
        6'b010011:begin
            ALU_OP <= 2'b01;
            reg_write <= 0;
            //reg_dst <= 0;     Don't care
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            //mem2reg <= 1;     Don't care
            branch <= 1;
            jump <= 0;
        end
        //jump
        6'b011100:begin
            ALU_OP <= 2'b01;
            reg_write <= 0;
            //reg_dst <= 0;     Don't care
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            //mem2reg <= 1;     Don't care
            branch <= 0;
            jump <= 1;
        end
        default:begin
            ALU_OP <= 2'b00;
            reg_write <= 0;
            reg_dst <= 0;
            ALU_src <= 0;
            mem_write <= 0;
            mem_read <= 0;
            mem2reg <= 0;
            branch <= 0;
        end
        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。

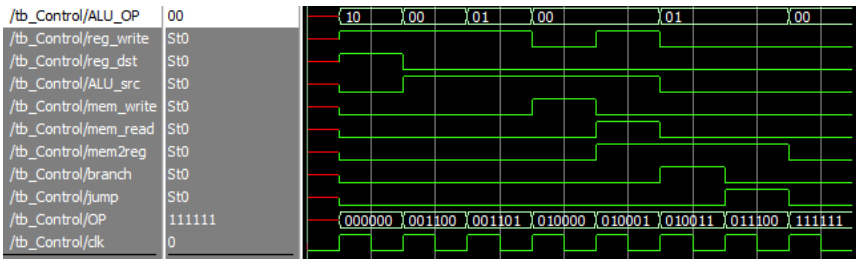
- ## Control test bench & Simulation



```verilog
// Setting timescale
`timescale 10 ns / 1 ns

module tb_Control;

        // Outputs
        wire [1:0]ALU_OP;
        wire reg_write;
        wire reg_dst;
        wire ALU_src;
        wire mem_write;
        wire mem_read;
        wire mem2reg;
        wire branch;
        wire jump;
        //inputs
        reg [5:0]OP;

        // Clock
        reg clk = 1'b0;

        // Instantiate the Unit Under Test (UUT)
        Control UUT(
                .ALU_OP(ALU_OP),
                .reg_write(reg_write),
                .reg_dst(reg_dst),
                .ALU_src(ALU_src),
                .mem_write(mem_write),
                .mem_read(mem_read),
                .mem2reg(mem2reg),
                .branch(branch),
                .jump(jump),
                .OP(OP)
        );

        // Generate Clock
        always
        begin : ClockGenerator
                #1 clk <= ~clk;
        end
```

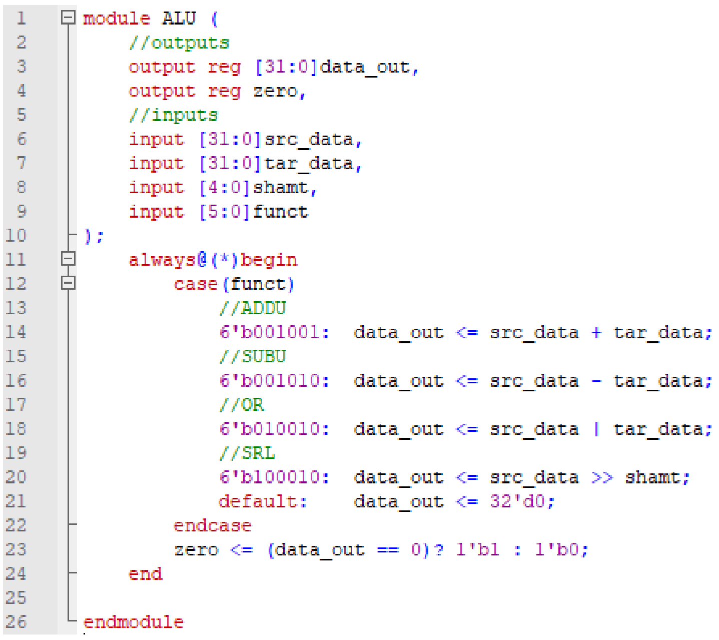


```verilog
        initial begin
                // Wait positive edge of clock signal
                @(posedge clk);

                // Reset UUT
                OP <= 0;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b001100;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b001101;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b010000;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b010001;

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b010011;//beq

                //assign value to OP code
                @(posedge clk);
                OP <= 6'b011100;//jump

                //assign undefined value to OP code
                @(posedge clk);
                OP <= 6'b111111;

                // Wait some time
                #2;
                // Stop the simulation
                $stop();
        end
endmodule
```



加入beq和jump的OP code 進行測試。

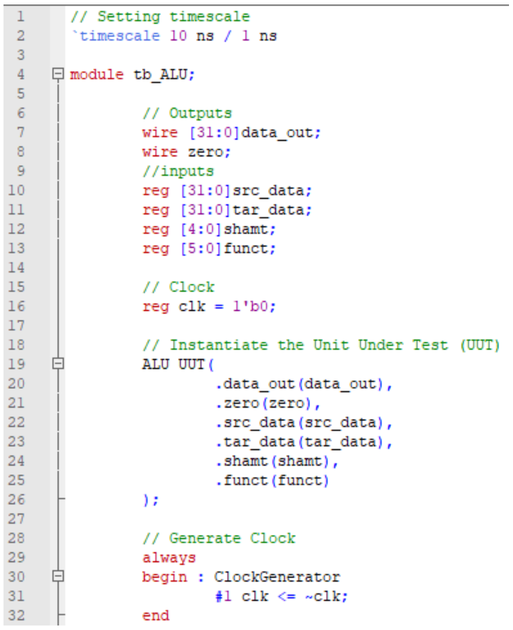
- ## ALU Module



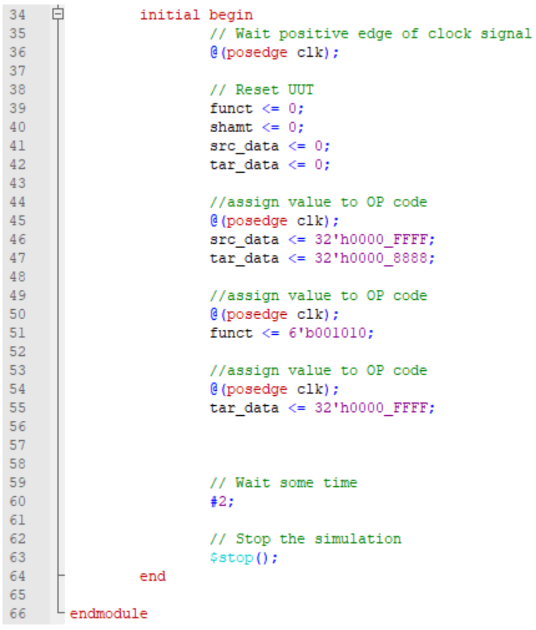
```verilog
module ALU (
    //outputs
    output reg [31:0]data_out,
    output reg zero,
    //inputs
    input [31:0]src_data,
    input [31:0]tar_data,
    input [4:0]shamt,
    input [5:0]funct
);
    always@(*)begin
        case(funct)
            //ADDU
            6'b001001:  data_out <= src_data + tar_data;
            //SUBU
            6'b001010:  data_out <= src_data - tar_data;
            //OR
            6'b010010:  data_out <= src_data | tar_data;
            //SRL
            6'b100010:  data_out <= src_data >> shamt;
            default:    data_out <= 32'd0;
        endcase
        zero <= (data_out == 0)? 1'b1 : 1'b0;
    end

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。

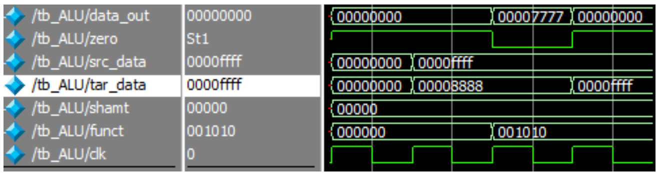
- ALU test bench & Simulation



```verilog
// Setting timescale
`timescale 10 ns / 1 ns

module tb_ALU;

        // Outputs
        wire [31:0]data_out;
        wire zero;
        //inputs
        reg [31:0]src_data;
        reg [31:0]tar_data;
        reg [4:0]shamt;
        reg [5:0]funct;

        // Clock
        reg clk = 1'b0;

        // Instantiate the Unit Under Test (UUT)
        ALU UUT(
                .data_out(data_out),
                .zero(zero),
                .src_data(src_data),
                .tar_data(tar_data),
                .shamt(shamt),
                .funct(funct)
        );

        // Generate Clock
        always
        begin : ClockGenerator
                #1 clk <= ~clk;
        end
```

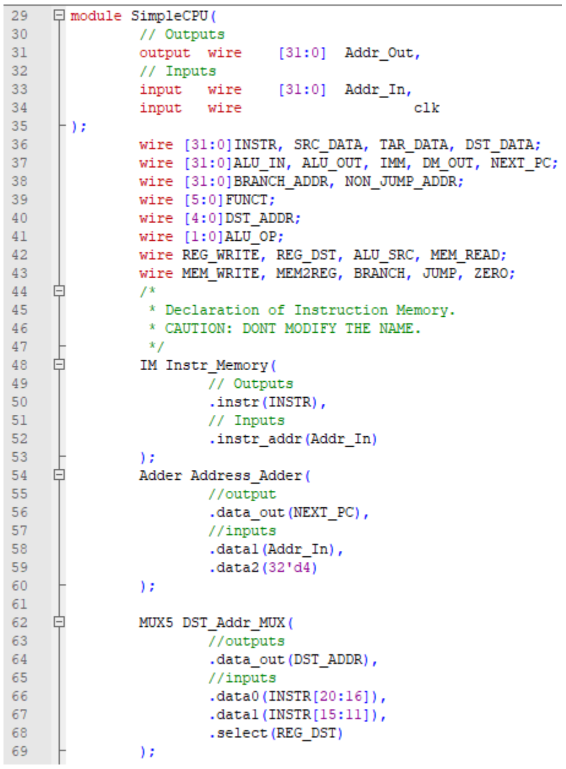


```verilog
        initial begin
                // Wait positive edge of clock signal
                @(posedge clk);

                // Reset UUT
                funct <= 0;
                shamt <= 0;
                src_data <= 0;
                tar_data <= 0;

                //assign value to OP code
                @(posedge clk);
                src_data <= 32'h0000_FFFF;
                tar_data <= 32'h0000_8888;

                //assign value to OP code
                @(posedge clk);
                funct <= 6'b001010;

                //assign value to OP code
                @(posedge clk);
                tar_data <= 32'h0000_FFFF;


                // Wait some time
                #2;

                // Stop the simulation
                $stop();

        end

endmodule
```

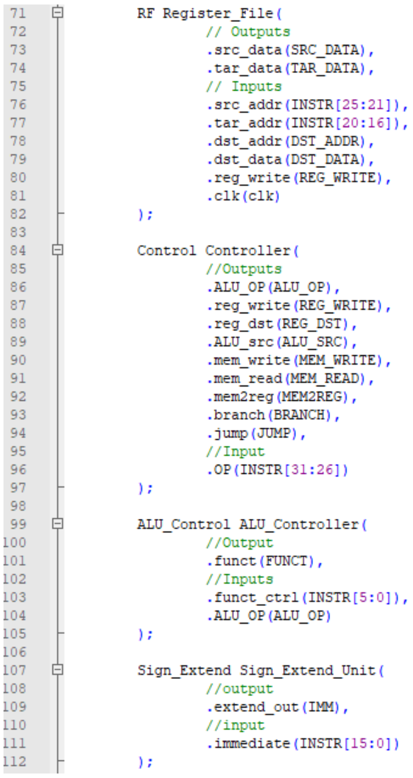


基本功能前面都測試過了，這邊主要在測試，Zero Flag有沒有正常反應。

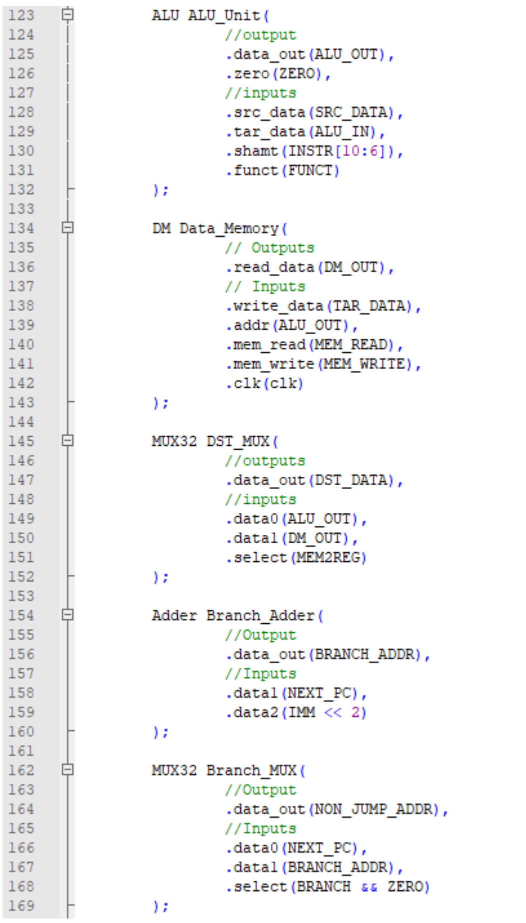
- Simple CPU



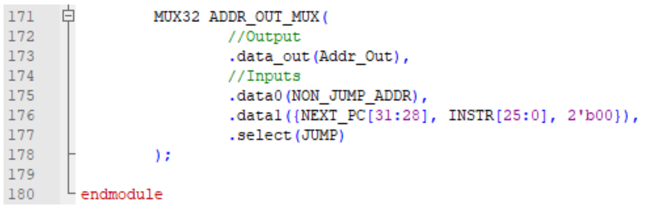
```verilog
module SimpleCPU(
        // Outputs
        output  wire    [31:0]  Addr_Out,
        // Inputs
        input   wire    [31:0]  Addr_In,
        input   wire                    clk
);
        wire [31:0]INSTR, SRC_DATA, TAR_DATA, DST_DATA;
        wire [31:0]ALU_IN, ALU_OUT, IMM, DM_OUT, NEXT_PC;
        wire [31:0]BRANCH_ADDR, NON_JUMP_ADDR;
        wire [5:0]FUNCT;
        wire [4:0]DST_ADDR;
        wire [1:0]ALU_OP;
        wire REG_WRITE, REG_DST, ALU_SRC, MEM_READ;
        wire MEM_WRITE, MEM2REG, BRANCH, JUMP, ZERO;
        /*
         * Declaration of Instruction Memory.
         * CAUTION: DONT MODIFY THE NAME.
         */
        IM Instr_Memory(
                // Outputs
                .instr(INSTR),
                // Inputs
                .instr_addr(Addr_In)
        );
        Adder Address_Adder(
                //output
                .data_out(NEXT_PC),
                //inputs
                .data1(Addr_In),
                .data2(32'd4)
        );

        MUX5 DST_Addr_MUX(
                //outputs
                .data_out(DST_ADDR),
                //inputs
                .data0(INSTR[20:16]),
                .data1(INSTR[15:11]),
                .select(REG_DST)
        );
```



```verilog
        RF Register_File(
                // Outputs
                .src_data(SRC_DATA),
                .tar_data(TAR_DATA),
                // Inputs
                .src_addr(INSTR[25:21]),
                .tar_addr(INSTR[20:16]),
                .dst_addr(DST_ADDR),
                .dst_data(DST_DATA),
                .reg_write(REG_WRITE),
                .clk(clk)
        );

        Control Controller(
                //Outputs
                .ALU_OP(ALU_OP),
                .reg_write(REG_WRITE),
                .reg_dst(REG_DST),
                .ALU_src(ALU_SRC),
                .mem_write(MEM_WRITE),
                .mem_read(MEM_READ),
                .mem2reg(MEM2REG),
                .branch(BRANCH),
                .jump(JUMP),
                //Input
                .OP(INSTR[31:26])
        );

        ALU_Control ALU_Controller(
                //Output
                .funct(FUNCT),
                //Inputs
                .funct_ctrl(INSTR[5:0]),
                .ALU_OP(ALU_OP)
        );

        Sign_Extend Sign_Extend_Unit(
                //output
                .extend_out(IMM),
                //input
                .immediate(INSTR[15:0])
        );
```

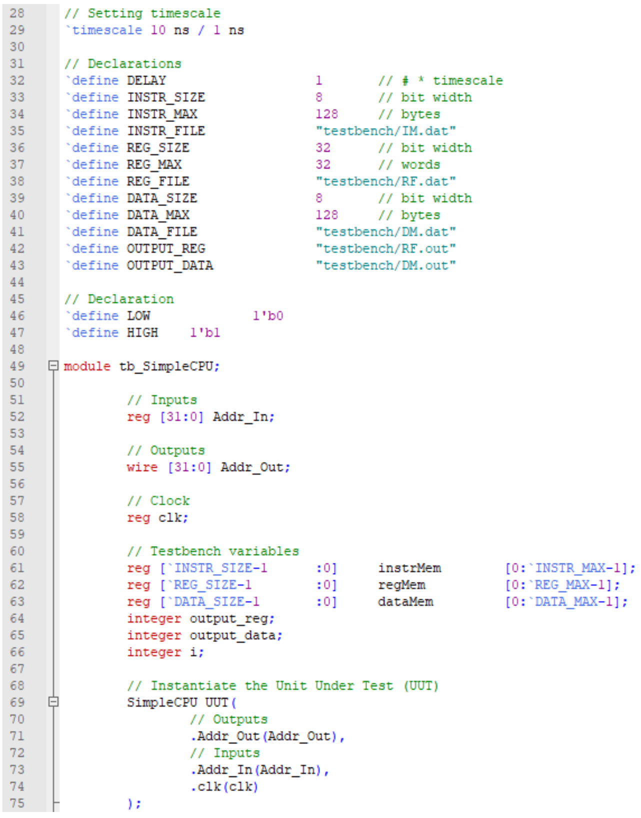


```verilog
	ALU ALU_Unit(
		//output
		.data_out(ALU_OUT),
		.zero(ZERO),
		//inputs
		.src_data(SRC_DATA),
		.tar_data(ALU_IN),
		.shamt(INSTR[10:6]),
		.funct(FUNCT)
	);

	DM Data_Memory(
		// Outputs
		.read_data(DM_OUT),
		// Inputs
		.write_data(TAR_DATA),
		.addr(ALU_OUT),
		.mem_read(MEM_READ),
		.mem_write(MEM_WRITE),
		.clk(clk)
	);

	MUX32 DST_MUX(
		//outputs
		.data_out(DST_DATA),
		//inputs
		.data0(ALU_OUT),
		.data1(DM_OUT),
		.select(MEM2REG)
	);

	Adder Branch_Adder(
		//Output
		.data_out(BRANCH_ADDR),
		//Inputs
		.data1(NEXT_PC),
		.data2(IMM << 2)
	);

	MUX32 Branch_MUX(
		//Output
		.data_out(NON_JUMP_ADDR),
		//Inputs
		.data0(NEXT_PC),
		.data1(BRANCH_ADDR),
		.select(BRANCH && ZERO)
	);
```

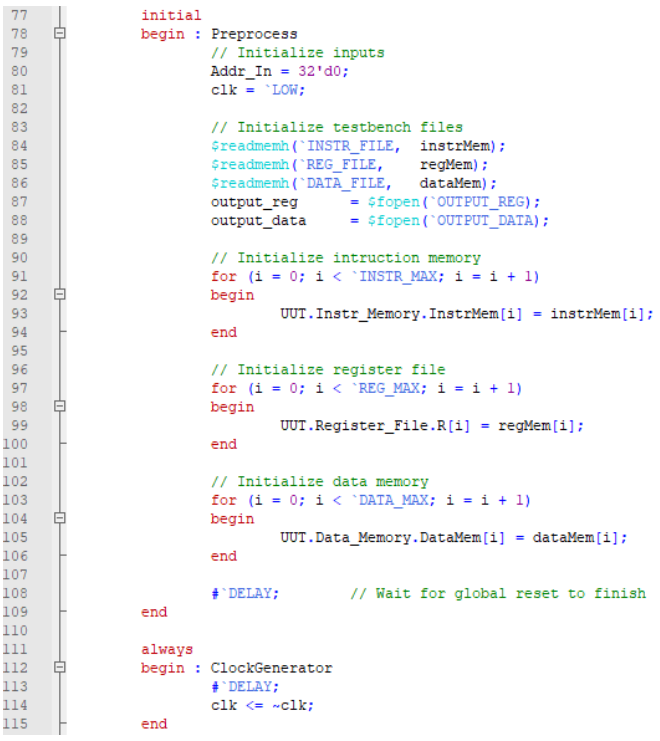


```verilog
	MUX32 ADDR_OUT_MUX(
		//Output
		.data_out(Addr_Out),
		//Inputs
		.data0(NON_JUMP_ADDR),
		.data1({NEXT_PC[31:28], INSTR[25:0], 2'b00}),
		.select(JUMP)
	);

endmodule
```

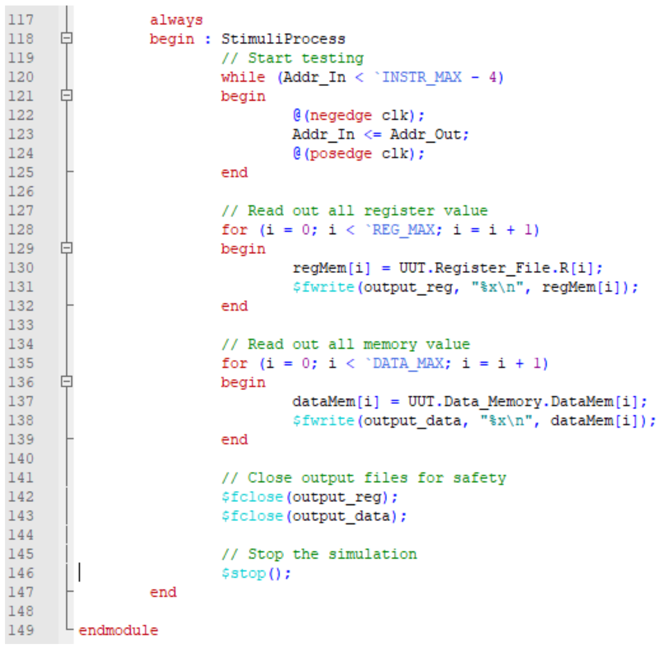
- Simple CPU testbench



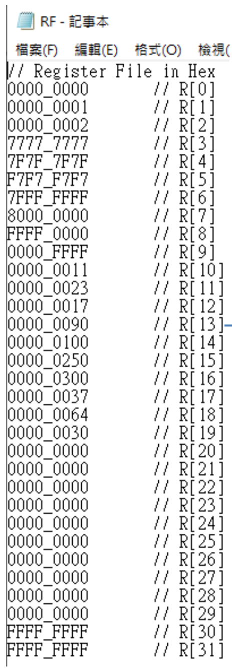
```verilog
// Setting timescale
`timescale 10 ns / 1 ns

// Declarations
`define DELAY			1	// # * timescale
`define INSTR_SIZE		8	// bit width
`define INSTR_MAX		128	// bytes
`define INSTR_FILE		"testbench/IM.dat"
`define REG_SIZE		32	// bit width
`define REG_MAX			32	// words
`define REG_FILE		"testbench/RF.dat"
`define DATA_SIZE		8	// bit width
`define DATA_MAX		128	// bytes
`define DATA_FILE		"testbench/DM.dat"
`define OUTPUT_REG		"testbench/RF.out"
`define OUTPUT_DATA		"testbench/DM.out"

// Declaration
`define LOW		1'b0
`define HIGH	1'b1

module tb_SimpleCPU;

	// Inputs
	reg [31:0] Addr_In;

	// Outputs
	wire [31:0] Addr_Out;

	// Clock
	reg clk;

	// Testbench variables
	reg [`INSTR_SIZE-1	:0]	instrMem	[0:`INSTR_MAX-1];
	reg [`REG_SIZE-1	:0]	regMem		[0:`REG_MAX-1];
	reg [`DATA_SIZE-1	:0]	dataMem		[0:`DATA_MAX-1];
	integer output_reg;
	integer output_data;
	integer i;

	// Instantiate the Unit Under Test (UUT)
	SimpleCPU UUT(
		// Outputs
		.Addr_Out(Addr_Out),
		// Inputs
		.Addr_In(Addr_In),
		.clk(clk)
	);
```



```verilog
	initial
	begin : Preprocess
		// Initialize inputs
		Addr_In = 32'd0;
		clk = `LOW;

		// Initialize testbench files
		$readmemh(`INSTR_FILE,	instrMem);
		$readmemh(`REG_FILE,	regMem);
		$readmemh(`DATA_FILE,	dataMem);
		output_reg	= $fopen(`OUTPUT_REG);
		output_data	= $fopen(`OUTPUT_DATA);

		// Initialize intruction memory
		for (i = 0; i < `INSTR_MAX; i = i + 1)
		begin
			UUT.Instr_Memory.InstrMem[i] = instrMem[i];
		end

		// Initialize register file
		for (i = 0; i < `REG_MAX; i = i + 1)
		begin
			UUT.Register_File.R[i] = regMem[i];
		end

		// Initialize data memory
		for (i = 0; i < `DATA_MAX; i = i + 1)
		begin
			UUT.Data_Memory.DataMem[i] = dataMem[i];
		end

		#`DELAY;		// Wait for global reset to finish
	end

	always
	begin : ClockGenerator
		#`DELAY;
		clk <= ~clk;
	end
```



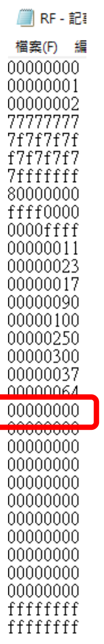
```
        always
        begin : StimuliProcess
                // Start testing
                while (Addr_In < `INSTR_MAX - 4)
                begin
                        @(negedge clk);
                        Addr_In <= Addr_Out;
                        @(posedge clk);
                end

                // Read out all register value
                for (i = 0; i < `REG_MAX; i = i + 1)
                begin
                        regMem[i] = UUT.Register_File.R[i];
                        $fwrite(output_reg, "%x\n", regMem[i]);
                end

                // Read out all memory value
                for (i = 0; i < `DATA_MAX; i = i + 1)
                begin
                        dataMem[i] = UUT.Data_Memory.DataMem[i];
                        $fwrite(output_data, "%x\n", dataMem[i]);
                end

                // Close output files for safety
                $fclose(output_reg);
                $fclose(output_data);

                // Stop the simulation
                $stop();
        end

endmodule
```



```
// 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]
```

RF.DAT

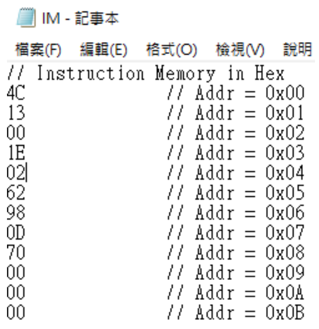
RF.OUT



```
00000000
00000001
00000002
77777777
7f7f7f7f
f7f7f7f7
7fffffff
80000000
ffff0000
0000ffff
00000011
00000023
00000017
00000090
00000100
00000250
00000300
00000037
00000064
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
ffffffff
ffffffff
```

Instructions

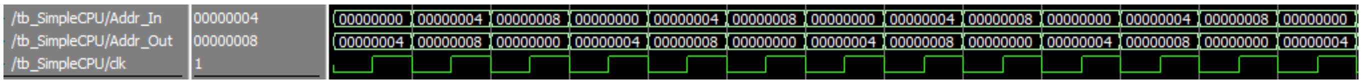
Hexadecimal	Binary	Meaning
4C13001E	010011_00000_10011_0000000000011110	Beq $R19, $R0, 30
0262980D	000000_10011_00010_10011_00000_001101	$R19 = $R19 – $R2
70000000	011100_00000000000000000000000000	Jump 0

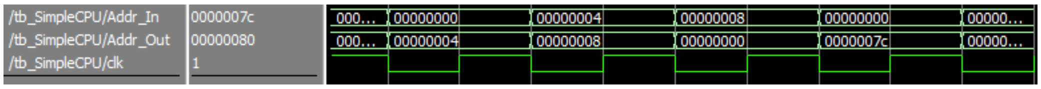


```
// Instruction Memory in Hex
4C              // Addr = 0x00
13              // Addr = 0x01
00              // Addr = 0x02
1E              // Addr = 0x03
02              // Addr = 0x04
62              // Addr = 0x05
98              // Addr = 0x06
0D              // Addr = 0x07
70              // Addr = 0x08
00              // Addr = 0x09
00              // Addr = 0x0A
00              // Addr = 0x0B
```

這三行指令算是一個迴圈，每個迴圈都會判斷位置是19的暫存器(30)有沒有等於位置是0的暫存器(0)，若不等於就會執行下一個指令，也就是把R19減R2(2)，然後再跳回第一個指令，一直到R19 = R0。

- Simple CPU Simulation





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*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，要是把全部的都寫在一起，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。