Logic Design Experiment Final Project
Single cycle ARM32 CPU

電機一丙 E24086129 林哲緯

一、 程式碼

1. ARM.v

```
`timescale 1ns / 1ps
module ARM(clk,rst);
    input clk,rst;

    //register
    reg [31:0] pc;

    //wire
    wire [31:0]pc_add_4, mem_read_data, instruction;
    wire [1:0] alu_src;
    wire [31:0] alu_out;
    wire [31:0] read_data_1, read_data_2,  read_data_3;
    wire [31:0] sign_extend_out, rotate_out, shift_out, unsign_extend_out;
    wire [3:0] alu_op;
    wire [3:0] nzcv;
    reg [3:0] nzcv_n;

    //adder
    assign pc_add_4 = pc + 32'd4;
    wire [31:0] pc_branch = sign_extend_out + pc_add_4;
    wire [31:0] reg_write_data = mem_to_reg? (mem_read_data):(alu_out);
    wire [31:0] pc_next = pc_write? ( pc_src? alu_out:alu_out ):( pc_src? pc_branch:pc_add_4 );
    wire [31:0] alu_operation_2 = alu_src[1]? ( alu_src[0]?
unsign_extend_out:shift_out ):( alu_src[0]? rotate_out:shift_out );
    ins_mem _ins_mem( .pc(pc), .ins(instruction) );
    data_mem
_data_mem( .clk(clk), .rst(rst), .mem_write(mem_write), .addr(alu_out), .write_data(read_data_3), .r
ead_data(mem_read_data));
    register_file _register_file(
        .clk(clk), .rst(rst), .reg_write(reg_write), .link(link),
        .read_addr_1(instruction[19:16]), .read_addr_2(instruction[3:0]), .read_addr_3(instruction[15
:12]),
        .write_addr(instruction[15:12]), .write_data(reg_write_data), .pc_content(pc_add_4),
        .pc_write(pc_write),
        .read_data_1(read_data_1), .read_data_2(read_data_2), .read_data_3(read_data_3));
    multi_4
_multi_4(.sign_immediate_in(instruction[23:0]), .sign_extend_immediate_out(sign_extend_out));
    rotate _rotate(.immediate_in(instruction[11:0]), .rotate_immediate_out(rotate_out));
    shift
_shift(.shift_type(instruction[6:5]), .shift_number(instruction[11:7]), .reg_data(read_data_2), .shi
ft_out(shift_out));

    unsigned_extend
_unsigned_extend(.unsign_immediate_in(instruction[11:0]), .unsign_extend_immediate_out(unsign_extend
_out));
    alu
_alu(.source_1(read_data_1), .source_2(alu_operation_2), .alu_op(alu_op), .c_in(nzcv_n[1]), .nzcv(nz
cv), .alu_out(alu_out));
```

```
   controller
_controller(.nzcv(nzcv_n), .opfunc(instruction[31:20]), .reg_write(reg_write), .alu_src(alu_src), .a
lu_op(alu_op),
      .mem_to_reg(mem_to_reg), .mem_write(mem_write), .pc_src(pc_src), .update_nzcv(update_nzcv), .
link(link));
   always@(posedge clk or posedge rst)
      if (rst)
         nzcv_n <= 4'b0;
      else
         nzcv_n <= (update_nzcv)? nzcv:nzcv_n;
   always@(posedge clk or posedge rst)
   begin
      if( rst == 1'b1 )
         pc <= 32'd0;
      else
         pc <= pc_next;
   end
endmodule
```

2. alu.v

```
module alu(source_1, source_2, alu_op, c_in, nzcv, alu_out);
   input [31:0] source_1, source_2;
   input [3:0] alu_op;
   input c_in;
   output [3:0] nzcv;
   output reg [31:0] alu_out;
   reg c, v;

   // A carry occurs:
   //  if the result of an addition is greater than or equal to 2^32
   //  if the result of a subtraction is positive or zero (*)
   //  as the result of an inline barrel shifter operation in a move or logical instruction.

   always@(*)
      case (alu_op)
         4'b0000: {c, alu_out} = {1'b0, source_1 & source_2};                                  // AND
         4'b0001: {c, alu_out} = {1'b0, source_1 ^ source_2};                                  // EOR
         4'b0010: {c, alu_out} = {1'b1, source_1} - {1'b0, source_2};                          // SUB
         4'b0011: {c, alu_out} = {1'b1, source_2} - {1'b0, source_1};                          // RSB
         4'b0100: {c, alu_out} = {1'b0, source_1} + {1'b0, source_2};                          // ADD
         4'b0101: {c, alu_out} = {1'b0, source_1} + {1'b0, source_2} + {32'b0, c_in};          // ADC
         4'b0110: {c, alu_out} = {1'b1, source_1} - {1'b0, source_2} + {32'b0, c_in} - 33'b1;// SBC
         4'b0111: {c, alu_out} = {1'b1, source_2} - {1'b0, source_1} + {32'b0, c_in} - 33'b1;// RSC
         4'b1000: {c, alu_out} = {1'b0, source_1 & source_2};                                  // TST
         4'b1001: {c, alu_out} = {1'b0, source_1 ^ source_2};                                  // TEQ
         4'b1010: {c, alu_out} = {1'b1, source_1} - {1'b0, source_2};                          // CMP
         4'b1011: {c, alu_out} = {1'b0, source_1} + {1'b0, source_2};                          // CMN
         4'b1100: {c, alu_out} = {1'b0, source_1 | source_2};                                  // OR
         4'b1101: {c, alu_out} = {1'b0, source_2};                                             // MOV
         4'b1110: {c, alu_out} = {1'b0, source_1 & ~source_2};                                 // BIC
         4'b1111: {c, alu_out} = {1'b0, ~source_2};                                            // MVN
      endcase

   always@(*)
      casex(alu_op)
         // Operand1 + Operand2: 0100 0101 1011           => 010x 1011
         // Operand1 - Operand2: 0010 0110 1010           => 0x10 1010
         // Operand2 - Operand1: 0011 0111                => 0x11
```

```
          // Logic: 0000 0001 1000 1001 1100 1101 1110 1111   => x00x 11xx (default)
          4'b010x, 4'b1011: v = (source_1[31] ^ alu_out[31]) & (source_1[31] ^~ source_2[31]);
          4'b0x10, 4'b1010: v = (source_1[31] ^ alu_out[31]) & (source_1[31] ^  source_2[31]);
          4'b0x11:         v = (source_2[31] ^ alu_out[31]) & (source_2[31] ^  source_1[31]);
          default: v = 1'b0;
       endcase

   assign nzcv = {alu_out[31], ~(|alu_out), c, v};
endmodule
```

3. controller.v

```
module controller(nzcv, opfunc, reg_write, alu_src, alu_op, mem_to_reg, mem_write, pc_src,
update_nzcv, link);
   input [3:0]nzcv;
   input [11:0]opfunc;
   output reg reg_write, mem_to_reg, mem_write, pc_src, update_nzcv, link;
   output reg [1:0]alu_src;
   output reg [3:0]alu_op;

   assign {n, z, c, v} = nzcv[3:0];
   wire condition =
     ((opfunc[11:8] == 4'b0000) &  z) |       // EQ, Z=1
     ((opfunc[11:8] == 4'b0001) & ~z) |       // NE, Z=0
     ((opfunc[11:8] == 4'b0010) &  c) |       // CS, C=1
     ((opfunc[11:8] == 4'b0011) & ~c) |       // CC, C=0
     ((opfunc[11:8] == 4'b0100) &  n) |       // MI, N=1
     ((opfunc[11:8] == 4'b0101) & ~n) |       // PL, N=0
     ((opfunc[11:8] == 4'b0110) &  v) |       // VS, V=1
     ((opfunc[11:8] == 4'b0111) & ~v) |       // VC, V=0
     ((opfunc[11:8] == 4'b1000) & (c & ~z)) |      // HI, C=1 & Z=0
     ((opfunc[11:8] == 4'b1001) & (~c | z)) |      // LS, C=0 | Z=1
     ((opfunc[11:8] == 4'b1010) & (n ~^ v)) |      // GE, N = V
     ((opfunc[11:8] == 4'b1011) & (n ^  v)) |      // LT, N ≠V
     ((opfunc[11:8] == 4'b1100) & (~z & (n ~^ v))) |   // GT, Z=0 & N = V
     ((opfunc[11:8] == 4'b1101) & ( z | (n ^  v))) |   // LE, Z=1 | N ≠V
      (opfunc[11:8] == 4'b1110);          // AL, always (nzcv ignored)

   always@(*) begin
      casex({condition, opfunc[7:5]})
         4'b1101: begin // branch
            reg_write = 1'b0;
            alu_src = 2'b00;
            alu_op = 4'b0000;
            mem_to_reg = 1'b0;
            mem_write = 1'b0;
            pc_src = 1'b1;
            update_nzcv = 1'b0;
            link = opfunc[4];
         end
         4'b100x: begin // data processing
            reg_write = (opfunc[4:3] == 2'b10)? 1'b0 : 1'b1;
            alu_src = (opfunc[5])? 2'b01 : 2'b00;
            alu_op = opfunc[4:1];
            mem_to_reg = 1'b0;
            mem_write = 1'b0;
            pc_src = 1'b0;
            update_nzcv = opfunc[0];
```

```
                link = 1'b0;
            end
            4'b101x: begin // data transfer
                reg_write = opfunc[0];
                alu_src = (opfunc[5])? 2'b10 : 2'b11;
                alu_op = (opfunc[3])? 4'b0100 : 4'b0010;
                mem_to_reg = 1'b1;
                mem_write = ~opfunc[0];
                pc_src = 1'b0;
                update_nzcv = 1'b0;
                link = 1'b0;
            end
            default: begin // fail
                reg_write = 1'b0;
                alu_src = 2'b00;
                alu_op = 4'b0000;
                mem_to_reg = 1'b0;
                mem_write = 1'b0;
                pc_src = 1'b0;
                update_nzcv = 1'b0;
                link = 1'b0;
            end
        endcase
    end
endmodule
```

4. data_mem.v

```
module data_mem(clk, rst, addr, write_data, mem_write, read_data);
    input clk, rst, mem_write;
    input [31:0]addr, write_data;
    output [31:0]read_data;

    parameter DATA_MEM_SIZE = 64;

    reg [31:0]mem[DATA_MEM_SIZE-1:0];
    integer i;
    assign read_data = mem[addr[31:2]];

    always@(posedge clk or posedge rst) begin
        if(rst == 1'b1)
            for(i = 0; i < DATA_MEM_SIZE; i = i + 1)
                mem[i] <= 0;
        else if (mem_write == 1'b1)
            mem[addr[31:2]] <= write_data;
    end
endmodule
```

5. ins_mem.v

```
module ins_mem(pc, ins);
    input [31:0]pc;
    output [31:0]ins;
    parameter INS_MEM_SIZE = 32;
    reg [31:0]mem[INS_MEM_SIZE-1:0];
    assign ins = mem[pc[31:2]];
endmodule
```

6. multi_4.v

```
module multi_4(sign_immediate_in, sign_extend_immediate_out);
   input [23:0]sign_immediate_in;
   output [31:0]sign_extend_immediate_out;

   assign sign_extend_immediate_out = {{6{sign_immediate_in[23]}}, sign_immediate_in, 2'b00};
endmodule
```

7. register_file.v

```
module register_file(clk, rst, reg_write, link,
                 read_addr_1, read_addr_2, read_addr_3, write_addr,
                 write_data, pc_content,
                 pc_write, read_data_1, read_data_2, read_data_3);

   input clk, rst, reg_write, link;
   input [3:0]read_addr_1, read_addr_2, read_addr_3, write_addr;
   input [31:0]write_data, pc_content;
   output pc_write;
   output [31:0]read_data_1, read_data_2, read_data_3;
   reg [31:0]memory[14:0];
   integer i;

   always@(posedge clk or posedge rst) begin
      if (rst) begin
         for (i = 0; i < 15; i = i + 1)
            memory[i] <= 0;
      end
      else begin
         if (reg_write & (write_addr < 4'b1110)) memory[write_addr] <= write_data;
         else if (link) memory[14] <= pc_content;
      end
   end

   assign pc_write = (&write_addr) & reg_write;
   assign read_data_1 = (&read_addr_1)? pc_content : memory[read_addr_1];
   assign read_data_2 = (&read_addr_2)? pc_content : memory[read_addr_2];
   assign read_data_3 = (&read_addr_3)? pc_content : memory[read_addr_3];
endmodule
```

8. rotate.v

```
module rotate(immediate_in, rotate_immediate_out);
   input [11:0]immediate_in;
   output [31:0]rotate_immediate_out;
   wire [31:0] tmp;
   assign {tmp, rotate_immediate_out} = {{24'b0, immediate_in[7:0]}, {24'b0, immediate_in[7:0]}} >>
{immediate_in[11:8], 1'b0};
endmodule
```

9. shift.v

```
module shift(reg_data, shift_type, shift_number, shift_out);
   input [1:0]shift_type;
   input [4:0]shift_number;
   input [31:0]reg_data;
```

```
    output  [31:0]shift_out;

    reg signed [31:0]shift_out;
    reg [31:0] tmp;

    always@(*) begin
        case(shift_type)
            2'b00: shift_out = reg_data << shift_number;
            2'b01: shift_out = reg_data >> shift_number;
            2'b10: shift_out = reg_data >>> shift_number;
            2'b11: {tmp, shift_out} = {reg_data, reg_data} >> shift_number; // right rotate
        endcase
    end
endmodule
```

10. unsigned_extend.v

```
module unsigned_extend(unsign_immediate_in, unsign_extend_immediate_out);
    input [11:0]unsign_immediate_in;
    output [31:0]unsign_extend_immediate_out;
    assign unsign_extend_immediate_out = {20'b0, unsign_immediate_in};
endmodule
```

二、 輸出結果

執行 run.bat (若有 Icarus Verilog)，即產生 log1.txt、log2.txt、log3.txt、log4.txt，分別對應 4 個 testbench 的輸出。

✧ run.bat

```
iverilog -o tb1 alu.v controller.v data_mem.v ins_mem.v multi_4.v
register_file.v rotate.v shift.v unsigned_extend.v ARM.v tb_ARM_1.v
iverilog -o tb2 alu.v controller.v data_mem.v ins_mem.v multi_4.v
register_file.v rotate.v shift.v unsigned_extend.v ARM.v tb_ARM_2.v
iverilog -o tb3 alu.v controller.v data_mem.v ins_mem.v multi_4.v
register_file.v rotate.v shift.v unsigned_extend.v ARM.v tb_ARM_3.v
iverilog -o tb4 alu.v controller.v data_mem.v ins_mem.v multi_4.v
register_file.v rotate.v shift.v unsigned_extend.v ARM.v tb_ARM_4.v
vvp tb1 > log1.txt
vvp tb2 > log2.txt
vvp tb3 > log3.txt
vvp tb4 > log4.txt
```

✧ 一到四測試結果皆正確 (詳見 log.txt)

三、 心得

經過幾個禮拜慢慢琢磨，終於打出來了，實在很有成就感，感謝助教們這學期的教導，我受益良多。