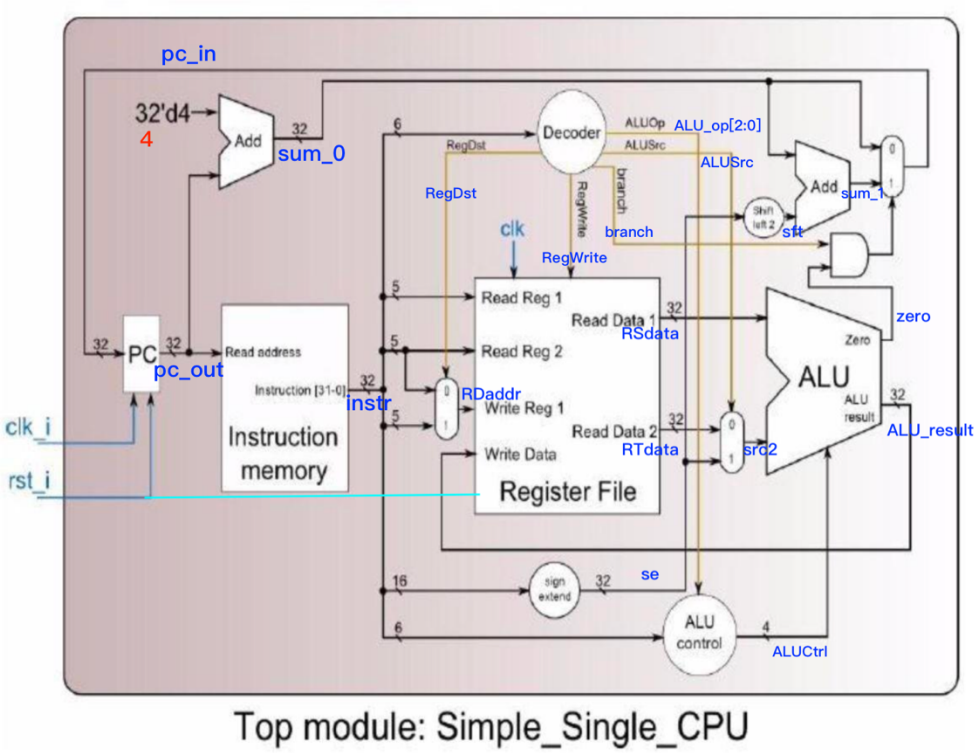
Computer Organization HW2

- Architecture diagrams:



Top module: Simple_Single_CPU

- Hardware module analysis:
  - Adder.v

    此 Adder 將每 4 個分為一組，共計 3 層。

    第一層$g = src1 \cdot src2$、 $p = src1 + src2$，

    第二層$g_{4i\sim4(i+1)} = g_{4i+3} + p_{4i+3} \cdot g_{4i+2} + p_{4i+3} \cdot p_{4i+2} \cdot g_{4i+1} + p_{4i+3} \cdot p_{4i+2} \cdot p_{4i+1} \cdot g_{4i}$、

    $p_{4i\sim4(i+1)} = p_{4i+3} \cdot p_{4i+2} \cdot p_{4i+1} \cdot p_{4i}$，

    第三層$g_{16i\sim16(i+1)} = g_{16i+12\sim16i+16} + p_{16i+12\sim16i+16} \cdot g_{16i+8\sim16i+12} + p_{16i+12\sim16i+16} \cdot p_{16i+8\sim16i+12} \cdot g_{16i+4\sim16i+8} + p_{16i+12\sim16i+16} \cdot p_{16i+8\sim16i+12} \cdot p_{16i+4\sim16i+8} \cdot g_{16i\sim16i+4}$、

    $p_{16i\sim16(i+1)} = p_{16i+12\sim16i+16} \cdot p_{16i+8\sim16i+12} \cdot p_{16i+4\sim16i+8} \cdot p_{16i\sim16i+4}$，

    計算完所有 p、g 後回推 c，令$c_0 = 0$，$c_{16} = g_{0\sim16} + c_0 \cdot p_{0\sim16}$，

    $c_{4i} = g_{4(i-1)\sim4i} + c_{4(i-1)} \cdot p_{4(i-1)\sim4i}$，

    $c_i = g_{i-1} + c_{i-1} \cdot p_{i-1}$。

    得出 p、g、c 後，將三者做 XOR，即為兩數相加結果。

```
//Internal Signals
reg [32-1:0] sum_o;
reg [32-1:0] p, g, c;
reg [8-1:0] pp, gg;
reg [1:0] ppp, ggg;
integer i;

//Parameter

//Main function
always @(src1_i, src2_i) begin

        g=src1_i&src2_i;
        p=src1_i|src2_i;
        for(i=0; i<8; i=i+1) begin
                gg[i]=g[i*4+3]|(p[i*4+3]&g[i*4+2])|(p[i*4+3]&p[i*4+2]&g[i*4+1])|
(p[i*4+3]&p[i*4+2]&p[i*4+1]&g[i*4]);
                pp[i]=p[i*4+3]&p[i*4+2]&p[i*4+1]&p[i*4];
        end
        for(i=0; i<2; i=i+1) begin
                ggg[i]=gg[i*4+3]|(pp[i*4+3]&gg[i*4+2])|(pp[i*4+3]&pp[i*4+2]&gg[i*4+1])|
(pp[i*4+3]&pp[i*4+2]&pp[i*4+1]&gg[i*4]);
                ppp[i]=pp[i*4+3]&pp[i*4+2]&pp[i*4+1]&pp[i*4];
        end
        c[0]=0; c[16]=ggg[0]|(ppp[0]&c[0]);
        for(i=0; i<2; i=i+1) begin
                c[i*16+4]=gg[i*4]|(pp[i*4]&c[i*16]);
                c[i*16+8]=gg[i*4+1]|(pp[i*4+1]&c[i*16+4]);
                c[i*16+12]=gg[i*4+2]|(pp[i*4+2]&c[i*16+8]);
        end
        for(i=0; i<8; i=i+1) begin
                c[i*4+1]=g[i*4]|(p[i*4]&c[i*4]);
                c[i*4+2]=g[i*4+1]|(p[i*4+1]&c[i*4+1]);
                c[i*4+3]=g[i*4+2]|(p[i*4+2]&c[i*4+2]);
        end
        sum_o=p^g^c;
end
```

- MUX_2to1.v

  當 select 或輸入的 data 有任何更動時，依照 select 的值將對應 data 放入 output 中輸出。

```
//Internal Signals
reg     [size-1:0] data_o;

//Main function
always @(data0_i, data1_i, select_i) begin
        if(select_i) data_o<=data1_i;
        else data_o<=data0_i;
end
```

- Shift_Left_Two_32.v

  由於輸入與輸出需有相同正負號，因此將輸出的 sign bit 設定為輸入的 sign bit，並將輸入的後 29 bits 放至輸出 sign bit 後的 29 bits，最後將輸出的末二位補 0。

```
//I/O ports
input [32-1:0] data_i;
output [32-1:0] data_o;

reg [32-1:0] data_o;

//shift left 2
always @(data_i) begin
        data_o[31]=data_i[31];
        data_o[30:2]<=data_i[28:0];
        data_o[1:0]<=2'b00;
end
```

- Sign_Extend.v

  將輸入放入輸出的右方 16 bits，並以輸入的 sign bit 填滿輸出左方的 16

bits。

```
//Internal Signals
reg     [32-1:0] data_o;

//Sign extended
always @(data_i) begin
        data_o[15:0]<=data_i;
        if(data_i[15]==1) data_o[31:16]<=16'b1111111111111111;
        else data_o[31:16]<=15'b0;
end
```

- Decoder.v

  由於 Decoder 輸出的 ALU operation 僅有 3 bit，而實際的 ALU operation 有 4 bit，又大多數指令的 ALU operation 首位為 0（除 NOR，且不在此 CPU 的 Instruction Set 內），因此取末 3 bits 輸出。

  此 CPU 的 Opcode 有 0、4、8、10 共 4 種可能，

  - 若 Opcode 為 0，表示此指令為 R-format，ALU_op_o 輸出 011（011 或 101 不會與其他指令衝突），待 ALU Control 根據 Funtion Code 進一步判斷實際 ALU operation；需要以第二個暫存器內的資料做運算，ALUSrc_o 輸出 0；需要將資料寫入第三個暫存器內，RegDst_o 輸出 1，RegWrite_o 輸出 1；不需進行 branch，Branch_o 輸出 0。
  - 若 Opcode 為 4，表示此指令為 beq，需將輸入的兩數相減並判斷結果是否為 0，因此 ALU_op_o 輸出 110 做減法；需要以第二個暫存器內的數做比較，ALUSrc_o 輸出 0；不需將資料寫入第三個暫存器內，RegDst_o 輸出 0 或 1 皆可，RegWrite_o 輸出 0；輸入兩數相等則需進行 branch，Branch_o 輸出 1。
  - 若 Opcode 為 8，表示此指令為 addi，ALU_op_o 輸出 010 做加法；以第一個暫存器內的資料與常數做運算，並將結果寫入第二個暫存器內，ALUSrc_o 輸出 1，RegDst_o 輸出 0，RegWrite_o 輸出 1；不需進行 branch，Branch_o 輸出 0。
  - 若 Opcode 為 10，表示此指令為 slti，ALU_op_o 輸出 111 做 slt；以第一個暫存器內的資料與常數做運算，並將結果寫入第二個暫存器內，ALUSrc_o 輸出 1，RegDst_o 輸出 0，RegWrite_o 輸出 1；不需進行 branch，Branch_o 輸出 0。

```
//Internal Signals
reg    [3-1:0] ALU_op_o;
reg            ALUSrc_o;
reg            RegWrite_o;
reg            RegDst_o;
reg            Branch_o;

//Parameter
parameter R=3'b011;

//Main function
always @(instr_op_i) begin
        case(instr_op_i)
                6'b000000: begin //R-format
                        ALU_op_o<=R; ALUSrc_o<=0; RegDst_o<=1; RegWrite_o<=1; Branch_o<=0;
                end
                6'b000100: begin //beq
                        ALU_op_o<=3'b110; ALUSrc_o<=0; RegDst_o<=0; RegWrite_o<=0; Branch_o<=1;
                end
                6'b001000: begin //addi
                        ALU_op_o<=3'b010; ALUSrc_o<=1; RegDst_o<=0; RegWrite_o<=1; Branch_o<=0;
                end
                6'b001010: begin //slti
                        ALU_op_o<=3'b111; ALUSrc_o<=1; RegDst_o<=0; RegWrite_o<=1; Branch_o<=0;
                end
                default: begin
                        ALU_op_o<=R; ALUSrc_o<=0; RegDst_o<=1; RegWrite_o<=1; Branch_o<=0;
                end
        endcase
end
```

- ALU.v

  此部分參照課本提供之 behavioral 32-bit ALU。

- ALU_Ctrl.v

  若從 Decoder 傳入的 ALUOp_i 為 011，表示此指令為 R-format，需讀取傳入的 funct_i 判斷實際為何種指令，並將對應的 4-bit ALU operation 輸出；若從 Decoder 傳入的 ALUOp_i 非 011，表示此指令為 I-format，輸入的 funct_i 實際為常數的一部份，且 Decoder 已找出正確的 ALU operation，將 ALUOp_i 前方補 0 輸出即可。

```
//Internal Signals
reg        [4-1:0] ALUCtrl_o;

//Parameter
parameter R=3'b011;

//Select exact operation
always @(funct_i, ALUOp_i) begin
        if(ALUOp_i==R) begin
                case(funct_i)
                        32: ALUCtrl_o<=4'b0010; //add
                        34: ALUCtrl_o<=4'b0110; //sub
                        36: ALUCtrl_o<=4'b0000; //AND
                        37: ALUCtrl_o<=4'b0001; //OR
                        42: ALUCtrl_o<=4'b0111; //slt
                endcase
        end
        else begin
                ALUCtrl_o[2:0]<=ALUOp_i;
                ALUCtrl_o[3]=1'b0;
        end
end
```

- Simple_Single_CPU.v

  根據 Architecture Diagram 連接各 Module，若 rst_i 為 0，表示尚不需讀取指令，設定第一個 Adder 相加的數為 0；若 rst_i 為 1，則設定第一個 Adder 相加的數為 4，每次讀取指令移動 4 bytes。

```
//Internal Signles
wire [32-1:0] pc_in, pc_out, sum_0, sum_1, instr, ALU_result, RSdata, RTdata, se, src2, sft;
wire [5-1:0] RDaddr;
wire [4-1:0] ALUCtrl;
wire [3-1:0] ALU_op;
wire zero, branch, RegWrite, RegDst, ALUSrc;
reg [32-1:0] constant;

always @(posedge clk_i) begin
        if(~rst_i) constant=0;
        else constant=4;
end

//Greate componentes
ProgramCounter PC(
        .clk_i(clk_i),
            .rst_i (rst_i),
            .pc_in_i(pc_in) ,
            .pc_out_o(pc_out)
            );

Adder Adder1(
        .src1_i(constant),
            .src2_i(pc_out),
            .sum_o(sum_0)
            );

Instr_Memory IM(
        .pc_addr_i(pc_out),
            .instr_o(instr)
            );

MUX_2to1 #(.size(5)) Mux_Write_Reg(
        .data0_i(instr[20:16]),
        .data1_i(instr[15:11]),
        .select_i(RegDst),
        .data_o(RDaddr)
        );

Reg_File RF(
        .clk_i(clk_i),
            .rst_i(rst_i) ,
        .RSaddr_i(instr[25:21]) ,
        .RTaddr_i(instr[20:16]) ,
        .RDaddr_i(RDaddr) ,
        .RDdata_i(ALU_result)  ,
        .RegWrite_i(RegWrite),
        .RSdata_o(RSdata) ,
        .RTdata_o(RTdata)
        );

Decoder Decoder(
        .instr_op_i(instr[31:26]),
            .RegWrite_o(RegWrite),
            .ALU_op_o(ALU_op),
            .ALUSrc_o(ALUSrc),
            .RegDst_o(RegDst),
                .Branch_o(branch)
            );
```

```
ALU_Ctrl AC(
        .funct_i(instr[5:0]),
        .ALUOp_i(ALU_op),
        .ALUCtrl_o(ALUCtrl)
        );

Sign_Extend SE(
        .data_i(instr[16-1:0]),
        .data_o(se)
        );

MUX_2to1 #(.size(32)) Mux_ALUSrc(
        .data0_i(RTdata),
        .data1_i(se),
        .select_i(ALUSrc),
        .data_o(src2)
        );

ALU ALU(
        .src1_i(RSdata),
            .src2_i(src2),
            .ctrl_i(ALUCtrl),
            .result_o(ALU_result),
                .zero_o(zero)
            );

Adder Adder2(
        .src1_i(sum_0),
            .src2_i(sft),
            .sum_o(sum_1)
            );

Shift_Left_Two_32 Shifter(
        .data_i(se),
        .data_o(sft)
        );

MUX_2to1 #(.size(32)) Mux_PC_Source(
        .data0_i(sum_0),
        .data1_i(sum_1),
        .select_i(branch&zero),
        .data_o(pc_in)
        );
```

- Finished part:
  - Testcase 1

    addi r1,r0,10 -> r1=10

    addi r2,r0,4 -> r2=4

    slt r3,r1,r2 -> r3=0

    beq r3,r0,1 -> 跳至下 1+1=2 個指令

    sub r5,r1,r2 -> r5=6

    => (r1,r2,r3,r4,r5)=(10,4,0,0,6)，結果正確

```
r0=          0
r1=         10
r2=          4
r3=          0
r4=          0
r5=          6
r6=          0
r7=          0
r8=          0
r9=          0
r10=          0
r11=          0
r12=          0
```

  - Testcase 2

    addi r6,r0,2 -> r6=2

addi r7,r0,14 -> r7=14
and r8,r6,r7 -> r8=2
or r9,r6,r7 -> r9=14
addi r6,r6,-1 -> r6=1
slti r1,r6,1 -> r1=0
beq r1,r0,-5 -> 跳至前 5-1=4 個指令
and r8,r6,r7 -> r8=0
or r9,r6,r7 -> r9=15
addi r6,r6,-1 -> r6=0
slti r1,r6,1 -> r1=1
beq r1,r0,-5 -> r1≠r0，結束
=> (r6,r7,r8,r9,r1)=(0,14,0,15,1)，結果正確

```
r0=          0
r1=          1
r2=          0
r3=          0
r4=          0
r5=          0
r6=          0
r7=         14
r8=          0
r9=         15
r10=         0
r11=         0
r12=         0
```

- Problems you met and solutions:

此次的 CPU 實作使用到了許多 submodules，且 submodules 間的連接也較為複雜，因此 debug 需要花比較久的時間慢慢對照波形圖，找出錯誤的模組。而 Decoder 輸出的 ALU operation 僅有 3 bits，在對照各項指令實際的 4-bit ALU operation 後，採用了取末三位的方法，並找出未使用的編碼表示 R-format。

一開始執行時發現 CPU 僅能進行第一個指令，最後發現是由於 Multiplexer 中 always 偵測的參數未設定所有的輸入所致；Adder 的部分使用了較為複雜的 3 層計算，因此初步先以直接相加 src1 與 src2 實作，最後再修正此模組的 bug。

- Summary:

在上這門課前，曾經我以為 CPU 是非常複雜的黑盒子，而透過這次的 Lab 實作，才了解原來 CPU 是由已經學習過的各種 submodules 連接而成，也體會到了所學的小元件慢慢堆疊成完整工具的成就感。