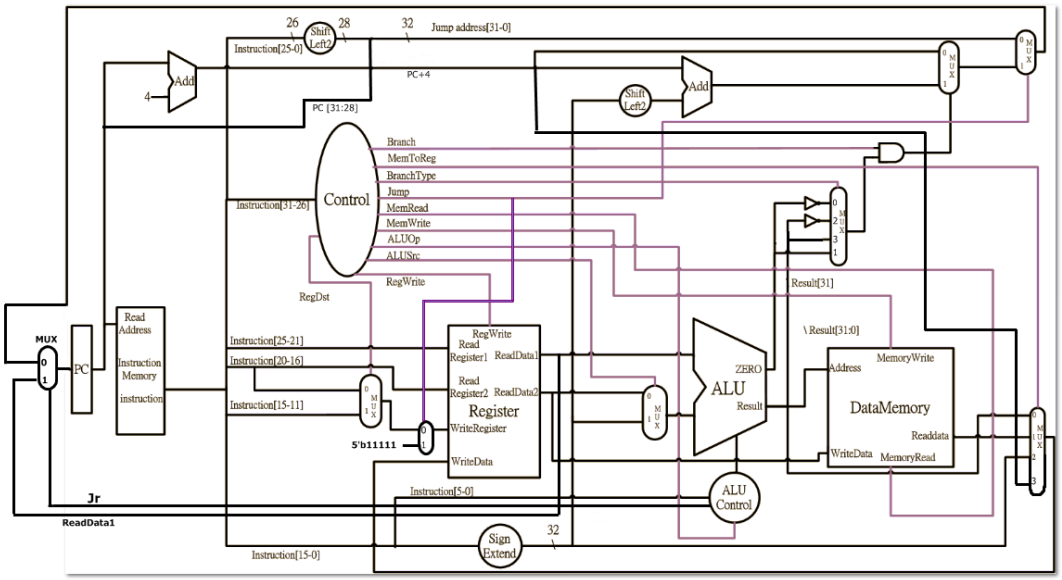
Computer Organization

Architecture diagram:



我有另外附上圖檔 single_cycle_picture.png 於 0111279.zip 當中。

Detailed description of the implementation:

Simple_single_CPU.v：

使用這個檔案將每一個小 module 用共通的變數連結起來，以下為在 ISE 當中的架構：

module name (file) ← 皆為這個形式

TestBench testbench (testbench.v)

----Simple_single_CPU CPU (Simple_single_CPU.v)

--------ProgramCounter PC (ProgramCounter.v)

--------Adder Adder1 (Adder.v)

--------Instr_Memory IM (Instr_Memory.v)

--------Shift_Left_Two_1 Shifter1 (Shift_Left_Two_1.v)

--------Decoder Decoder (Decoder.v)

--------MUX_2to1 Mux_Write_Reg1 (MUX_2to1.v)

--------MUX_2to1 Mux_Write_Reg2 (MUX_2to1.v)

--------Reg_File RF (Reg_File.v)

--------Sign_Extend SE (Sign_Extend.v)

--------ALU_Ctrl AC (ALU_Ctrl.v)

--------MUX_2to1 Mux_ALUSrc (MUX_2to1.v)

--------ALU ALU (ALU.v)

--------MUX_4to1 BRANCH (MUX_4to1.v)
--------Data_Memory Data_Memory (Data_Memory.v)
--------MUX_4to1 Mux_Write_Data_Source (MUX_4to1.v)
--------Shift_Left_Two_2 Shifter (Shift_Left_Two_2.v)
--------Adder Adder2 (Adder.v)
--------MUX_2to1 Mux_PC_Source (MUX_2to1.v)
--------MUX_2to1 Mux_J_Source (MUX_2to1.v)
--------MUX_2to1 JR (MUX_2to1.v)

以下只寫相較於 Lab2 有做修改或新加進入的 module：

Shift_Left_Two_1.v：

針對 Jump 類指令，目的是將 26 bit extend 成 28 bit。Code 如下：

```
always@( * )begin
    temp=data_i<<2;
    data_o=temp;
end
```

(另外，Shift_Left_Two_2.v 就是原本 Lab2 中的 Shift_Left_Two.v)

Decoder.v：

Decoder 相較於上次 Lab2，多了 5 個 control signal，分別是：BranchType_o(BEQ, BNE, BLT and BGEZ)、Jump_o、MemRead_o、MemWrite_o 及 MemtoReg_o(選擇要寫入 RegFile 的要是哪一種 Data)。然後也多出了幾個新的 Instruction，分別是：lw、sw、BNEZ、BLT、BGEZ、LI、MUL、J、Jal 及 Jr。然後針對不同 Instruction 的特性及要求，分別給予各個 control signal 不同的值。

ALU_Ctrl.v：

設計上與 Lab2 幾乎一樣，但針對一些新的 Instruction，所以新增了一些 ALUOp _i(從 000 ~111)和 ALUCtrl_o (分別有 0、1、2、4、6、7、11 (Decimal))。比較特別的地方在於，ALU_Ctrl.v 中我特別計算了一個變數 Jr_o，目的是讓這個變數去當作 **MUX_2to1 Mux_J_Source (MUX_2to1.v)** 的 select_i，寫法如下：

```
assign Jr_o = (ALUOp_i==3'b010 && funct_i==6'b001000);
//是 Jr 指令的時候，Jr_o 就等於 1
```

ALU.v：

設計上與 Lab2 幾乎一樣，但因為 ALU_Ctrl.v 新增了一些 ALUOp _i(從 000 ~111)和 ALUCtrl_o (分別有 0、1、2、4、6、7、11 (Decimal))，所以也比須有所對應。**其中 ALUCtrl_o 中的 11 比較特別一點，它是專門使用來處理 Instruction 是 BGEZ 的情況。**Code 如下：

```
always@(ctrl_i,src1_i,src2_i)begin
    case(ctrl_i)
        0:result_o <= src1_i & src2_i;
        1:result_o <= src1_i | src2_i;
```

```
            2:result_o <= src1_i + src2_i;
            4:result_o <= src1_i * src2_i;
            6:result_o <= src1_i - src2_i;
            7:result_o <= (src1_i < src2_i)? 1:0;
            //9:result_o <= src2_i >> src1_i ;
            //10:result_o <= src2_i << 16 ;
            11:result_o <= src1_i ; //直接將 src1_i (即為 rs)  存進 result_o
            default: result_o <= 0;
        endcase
end
```

Data_Memory.v：

針對 CO_P3_test_data1 及 CO_P3_test_data2，我特別先將所有 memory 的值歸 0，以免影響到最後的結果，如下：

```
initial begin
        for(i=0; i<128; i=i+1)
                Mem[i] = 8'b0;
        Mem[0] =    8'b0;
        Mem[4] =    8'b0;
        Mem[8] =    8'b0;
        Mem[12] = 8'b0;
        Mem[16] = 8'b0;
        Mem[20] = 8'b0;
        Mem[24] = 8'b0;
        Mem[28] = 8'b0;
        Mem[32] = 8'b0;
        Mem[36] = 8'b0;
end
```

原本是寫成如下，特別針對 CO_P3_test_data3 的 Bubble Sort 給值：

```
initial begin
        for(i=0; i<128; i=i+1)
                Mem[i] = 8'b0;
        Mem[0] = 8'b0100;    //4
        Mem[4] = 8'b0101;    //5
        Mem[8] = 8'b0110;    //6
        Mem[12] = 8'b0111;    //7
        Mem[16] = 8'b1000;    //8
        Mem[20] = 8'b1001;    //9
        Mem[24] = 8'b1010;    //10
```

```
        Mem[28] = 8'b0010;    //2
        Mem[32] = 8'b0001;    //1
        Mem[36] = 8'b0011;    //3
end
```

Problems encountered and solutions:

這次寫 Lab3 作業，一開始相當順利，只有遇到一些容易解決的小問題，像是：

我在寫 branch(E.G. BLT,BGEZ…)或是 Jump(J,Jal,Jr)這類的跳躍指令時，因為忘記定義，都把 Immediate(MIPS 指令最後 16 個 bit)寫成要跳去的那一行，而不是寫成跟目前的 PC+4 究竟是差了幾行。所以本來最後 16 bit 全部都是寫正整數，在修改之後，其中就有一些變成負數了(因為該 Jump 或 Branch 指令是跳躍到較前面的標籤)。

測完 CO_P3_test_data3 之後，Bubble Sort 的結果是 Ascending，因為基本上也排好了，所以就開始撰寫報告。但後來有人跟我說其實是要做 Descending 才對，仔細一看，發現指令為 BLT 的時候，我給錯值了。應該要給 MUX(用來選擇最後要跟 Branch 一起通過 AND 的值)ALUResult[31]卻給成了~ALUResult[31](**若 rs – rt < 0，則 ALUResult < 0，ALUResult[31]==1**)。

於是我就決定將 MUX 的寫法改成如下：

```
        MUX_4to1 #(.size(1)) BRANCH(
        .data0_i(~Zero),    //BNE
        .data1_i(Zero),    //BEQ
        .data2_i(ALUResult[31]),    //BLT
        .data3_i(~ALUResult[31]),    //BGEZ，若 rs >= 0，則 ALUResult[31]==0
        .select_i(BranchType),
        .data_o(Zero_)
        )
```

BLT 改寫的同時，我也將 BGEZ 的寫法從 **.data3_i(~(ZerolALUResult[31]))** 改寫成**.data3_i(~ALUResult[31])**。由於不夠小心，改寫完這兩個我就進行 simulate 了，結果卻發生了很大的問題，RegFile 連讀取值都有困難。後來經過仔細觀察後才終於發現，原來在 Decoder.v 當中誤將 ADDI 跟 BGEZ 的 ALU_op_o 都給成了 011，因而導致極為嚴重的問題。後來將 BGEZ 的 ALU_op_o 給成 111，然後再 ALU_Ctrl.v 當中操作如下：

```
else if(ALUOp_i==3'b111)begin
            ALUCtrl_o=4'b1011;    //11
end
```

這樣一來，指令為 BGEZ 的時候，ALU 就會直接將 src1_i (即為 rs) 存進 result_o。所以，如果 src1_i (即為 rs) >=0，那麼 result_o[31](也就是輸出的 ALUResult[31])

就會等於 0。