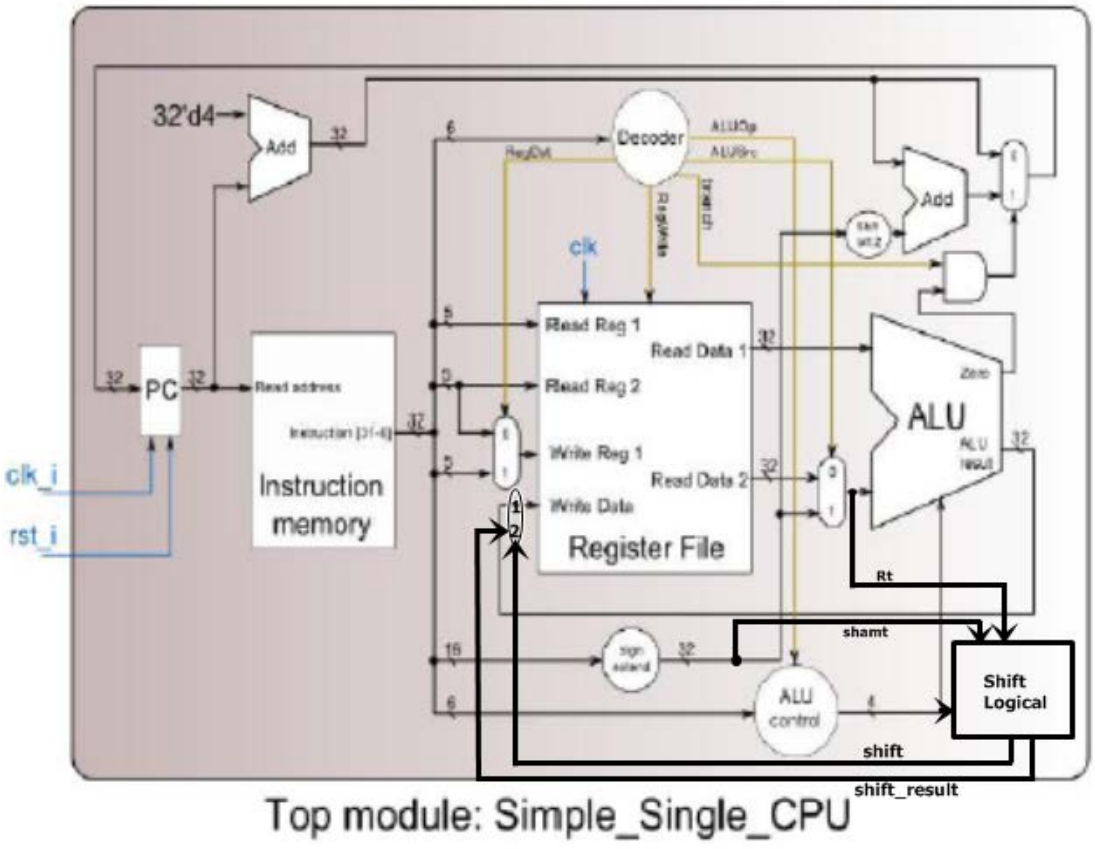
Computer Organization

Architecture diagram:



Top module: Simple_Single_CPU

Detailed description of the implementation:

Simple_Sigle_CPU.v：

基本上這個檔案就是將上圖的架構圖以 Verilog 實現，同樣的變數就像圖中的線路般將各個圖中的各個元件(即為 Verilog 當中 module)連結起來。

Adder.v：

Adder.v 的工作很簡單，就是將兩個 input 變數(src1_i 和 src2_i)相加之後給予 output 變數 sum_o。

```
assign sum_o=src1_i+src2_i;
```

MUX_2to1.v：

以 input 變數 select_i 來分辨要輸出 input 變數 data0_i 的值還是 data1_i 的值。

```
always@( * )begin
    if(select_i==1'b0)data_o=data0_i; //select_i 為 0， data_o 存成 data0_i 的值
```

```
    else if(select_i==1'b1)data_o=data1_i; //select_i 為 1， data_o 存成 data1_i 的值
end
```

Decoder.v：

在 always@當中以 if(instr_op_i==6'b??????)(或 else if)將不同的 instr_op_i(Op_code)做分類，並根據不同的 instr_op_i，分別給予 ALU_op_o、RegDst_o、RegWrite_o、ALUSrc_o、Branch_o 等 output 變數不同的輸出值。

ALUCtrl.v：

功能與 Decoder.v 相當類似，一樣是在 always@當中以 if(ALUOp_i==3'b???)來做分類。根據不同的 ALUOp_i，分別給予 ALUCtrl_o output 變數不同的輸出值。值得注意的是，當 ALUOp_i==3'b010 的時候，需再透過第二層的 if(funct_i==6'h??)(或 if(funct_i==6'b??????))來做分類。

Sign_Extend.v：

目的是將 16 位二進位數 extend 成 32 位二進位數。做法是透過判別 input 變數 data_i 的第 16 位(即 data_i[15])是 0 還是 1，來決定是要在 data_i 前面加上 16 個 0 還是 16 個 1。

```
always@( * )begin
    if(data_i[15]==1'b1)data_o={16'b1111111111111111,data_i};
    else data_o={16'b0000000000000000,data_i};
end
```

Shift_Logical.v：

使用這個檔案特別處理 SRL。input 變數分別有 src_i(R-type 中的 Rt)、sftamt_i(R-type 中的 shamt)、ctrl_i(輸入 ALUCtrl_o)，output 變數則分別為 shift_o(為 1 表示當前的 MIPS 指令是 SRL)、sft_result_o(若當前 MIPS 指令是 SRL，就是 SRL 後的 result；其他指令時都為 0)。

```
always@(ctrl_i,src_i,sftamt_i)begin
    if(ctrl_i==4'b1000)begin shift_o=1; sft_result_o <= src_i >> sftamt_i; end
    //我將 SRL 的 ALUCtrl_o 指定為 4'b1000
    else begin shift_o=0; sft_result_o <= 0; end
end
```

ALU.v：

有兩個輸出變數，分別是 zero_o 和 result_o，以下分別說明：(其中 ctrl_i 是輸入的 ALUCtrl_o)

1. zero_o：

```
assign zero_beq=(result_o==0); // result_o 為 0 時，將 zero_beq 設為 1
assign zero_bne=(result_o!=0); // result_o 不為 0 時，將 zero_bne 設為 1
assign zero_o=(ctrl_i==4'b1011)? zero_bne:zero_beq; // ctrl_i 為 4'b1011 時(當 MIPS 指令是 BNE 時，我將 ALUCtrl_o 設為 4'b1011)，zero_o 就是 zero_bne 的值，其他情況則都是 zero_beq 的值
```

2. result_o：
   在 always@當中以 case(ctrl_i)將不同的指令進行分類，並做出相應的計算，再將計算結果存進 result_o。

```
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;
        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 - src2_i;
        12:result_o <= ~(src1_i | src2_i);
        default: result_o <= 0;
  endcase
end
```

Shift_Left_Two_32.v：

將 input 變數 data_i 往左 shift 兩位並存進 output 變數 data_o。

```
data_o=data_i<<2;
```

Problems encountered and solutions:

在寫這一次的程式作業的過程中，我總共遇到了兩個問題，分別是：

1. 首先遇到的是 ISE 的 license 問題，在初次試圖模擬的時候就遇到這個問題，所以也無法對程式進行 debug。在自己的電腦無法模擬，到了系計中使用系上的電腦還是無法模擬，直到後來才明白是 license 的有效期限(30 天)早已過期。因為想不出別的辦法，只好又在 Xilinx 官網上申請一新的帳號，以便取得一個新的為期 30 天的 license。
2. 因為一開始不太了解 MUX #(.size(5)) Mux_Write_Reg 中 #(.size(5)) 的使用方法，所以特別把原本 MUX_2to1.v 分別改成 MUX_32bit_2to1.v 和 MUX_5bit_2to1.v 兩個檔案。但卻因為在 MUX_32bit_2to1.v 當中誤將原本的 size 改寫成 5(應該要改成 32)，因而導致模擬的時候，就連第一個測資都跑不過。這個錯誤修正之後，除了 advance 的測資以外，基本測資就全部都過了。(註：不過當我所有的測資都測試成功之後，我又將 MUX 的寫法改回原來的樣子。)