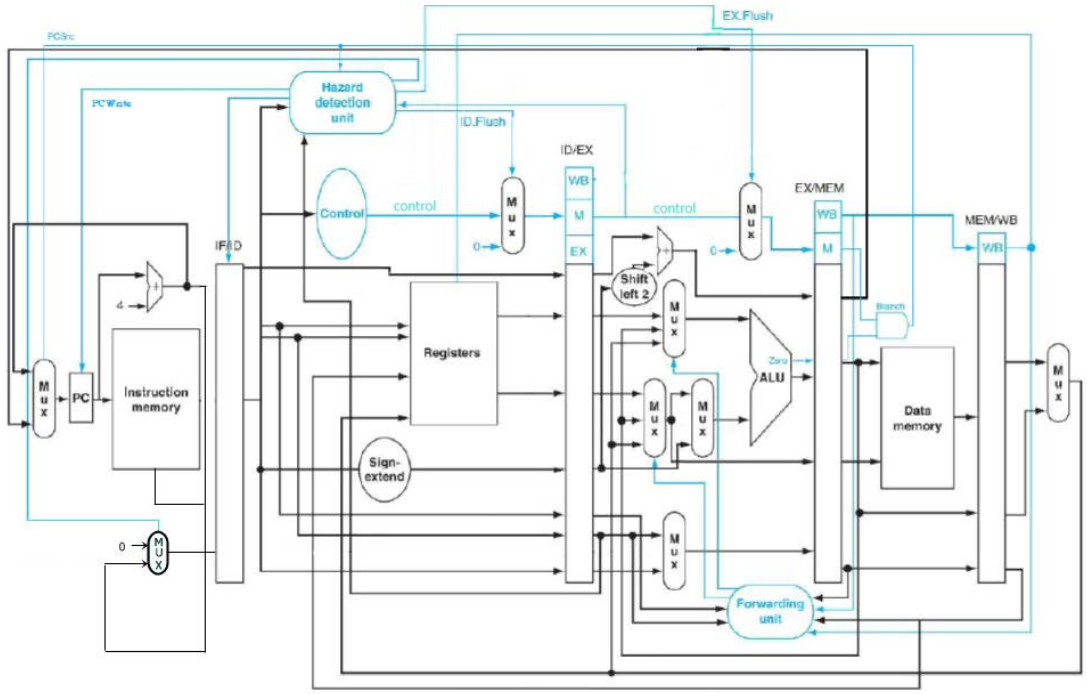
Computer Organization

Architecture diagram(以下結構圖不包含SRL指令的處理):



Detailed description of the implementation:

程式結構如下：

TestBench (TestBench.v)

cpu - Pipe_CPU_1 (Pipe_CPU_1.v)

PC - ProgramCounter (ProgramCounter.v)

Add_pc - Adder (Adder.v)

IM - Instr_Memory (Instr_Memory.v)

Mux6 - MUX_2to1 (MUX_2to1.v) // IFFlush

IF_ID - Pipe_Reg_spec (Pipe_Reg_spec.v)

RF - Reg_File (Reg_File.v)

Control - Decoder (Decoder.v)

Mux7 - MUX_2to1 (MUX_2to1.v) // IDFlush

Sign_Extend - Sign_Extend (Sign_Extend.v)

ID_EX - Pipe_Reg (Pipe_Reg.v)

Mux8 - MUX_2to1 (MUX_2to1.v) // EXFlush

Mux4 - MUX_3to1 (MUX_3to1.v) // Forward1

Mux5 - MUX_3to1 (MUX_3to1.v) // Forward2

Mux1 - MUX_2to1 (MUX_2to1.v) // choose from ***RF 的 data2*** or ***immediate(extended)***

ALU_Control - ALU_Control (ALU_Control.v)
SL - Shift_Logical (Shift_Logical.v)　//用來特別處理SRL 的情況
ALU - ALU (ALU.v)
Mux2 - MUX_2to1 (MUX_2to1.v)　// choose from　***rt***　or　***rd***
Shifter - Shift_Left_Two_32 (Shift_Left_Two_32.v)
Add_branch - Adder (Adder.v)
EX_MEM - Pipe_Reg (Pipe_Reg.v)
DM - Data_Memory (Data_Memory.v)
Mux3 - MUX_2to1 (MUX_2to1.v)　// choose from　***pc_not_branch***　or　***pc_branch***
MEM_WB - Pipe_Reg (Pipe_Reg.v)
Mux - MUX_3to1_spec (MUX_3to1_spec.v)　// choose from　***ALUResult***　or　***DMResult***　or　***SRLResult***
FU - Forwarding_Unit (Forwarding_Unit.v)
HDU - Hazard_Detection_Unit (Hazard_Detection_Unit.v)

以上前幾次 Lab 有說過的元件就不再次說明，以下僅說明這次 Lab4 新增或是有修改的的 module：

Mux6 & Mux7 & Mux8：

分別對應 **IFFlush & IDFlush & EXFlush**。當 branch 發生時，這三個變數就會拉高為 1，所以這三個 Mux 就會選擇 data1_i(也就是 0)，而非選擇本來的 control signals 或是 instruction，因為達成 flush 的效果。

Mux4 & Mux5：

分別對應 **Forward1(rs) & Forward2(rd)**。

Forward1 可能會是 00/ 01/ 10，當其為 00 時，Mux4 會選擇那個 clock 時剛從 RegFile 讀出的資料；而當其為 01 的時候，Mux4 會選擇那個 clock 時剛從 DataMemory 當中取出的值(即為那個 clock 時要寫回 RegFile 的資料)；最後一種情況是當其為 10 時，Mux4 會選擇那個 clock 時剛被 ALU 計算後所輸出的結果。Mux4 選完後將其結果輸入 ALU，當作 ALU 的 src1_i。

Mux5 做的事情跟 Mux4 一模一樣，不同點在於 Mux5 是依照 Forward2 是 00/ 01/ 10 來做選擇，選完後將其結果輸入 ALU，當作 ALU 的 src2_i。

Mux (MUX_3to1_spec.v)：

這是一個我有特別改寫的 File，目的是讓他能夠從 ALUResult、DMResult 及 SRLResult 三個資料當中進行選擇。之前 Lab3 有一個 2-bit 的 control signal 叫做 MemtoReg，是用來控制最終寫回 RegFile 的資料要是 ALUResult、DMResult、extend_out 及 pc_not_branch 當中的哪一個。

而這一次 Lab4 我繼續沿用 2-bit 的 MemtoReg，但卻只有用到當中的 00 和 01，用以分辨 ALUResult 和 DMResult。然後在 Pipe_CPU 當中多輸入一個變數 SRL(該 signal 表示 SRL 指令發生)，當 SRL 為 1 時，就直接讓輸出資料 data_o 等於輸入資料 data2_i(也就是 SRLResult)。Code 如下：

```
always@( * )begin
    if(~srl_i) begin
        if(select_i==2'b00)data_o=data0_i;
        else if(select_i==2'b01)data_o=data1_i;
```

```
        end
        else begin data_o=data2_i; end
    end
```

IF_ID (Pipe_Reg_spec.v)：

這一個 Pipe_Reg 我有特別改寫，讓他多了一個 IF_ID_Write_i 的變數(在 Pipe_CPU 當中輸入 IFIDWrite 這個變數)，其用途是當程式發生需要 stall 的情況時，IFIDWrite 就會拉高為 1，因而使的 IF_ID 在下一次 clock 的時候會 fetch 跟這一次 clock 同一條 instruction。Pipe_Reg_spec.v 內部寫法如下：

```
    always @(posedge clk_i) begin
        if(~rst_i)
            data_o <= 0;
         else if(rst_i && IF_ID_Write_i)
            data_o <= data_o;
        else
            data_o <= data_i;
    end
```

ID_EX & EX_MEM & MEM_WB (Pipe_Reg.v)：

這三個 Pipe_Reg 就維持原來老師給的寫法，如下：

```
        always @(posedge clk_i) begin
            if(~rst_i)
                data_o <= 0;
            else
                data_o <= data_i;
        end
```

在 clock 拉高為 1 的時候，將輸入資料 data_i 存進輸出資料 data_o。

Control (Decoder.v)：

這次 Lab4 只有一種類型的 branch(BEQ)，然後也沒有 jump 類的指令，所以我將上次 Lab3 中 Decoder 本來存在的 BranchType 和 Jump 從中去除。

FU (Forwarding_Unit.v)：

Forwading_Unit 會偵測目前是不是發生了 data dependency 的情況，並判別是哪一類(EX or DM 的 result)的 data 需要被 forward，然後決定是要 forward 到哪一個位置(rt or rd)。

HDU (Hazard_Detection_Unit.v)：

Hazard_Detection_Unit 要負責偵測需要 stall 以及需要 flush 的情況。

當需要 stall 的情況(lw 指令與其下一行指令之間有 data dependency 的情況)發生，就要讓當下被 fetch 的指令再被 fetch 一次(PCWrite ← 1)，以及讓當下被 decode 的指令再被 decode 一次(IFIDWrite ← 1 & IDFlush ← 1)。

而當需要 flush 的情況(Branch 指令的計算結果是要 branch)發生，就要 flush 掉當下在 IF、ID 及 EX 三個階段運行的指令(也就是緊接 Branch 後的三條指令)。IFIDFlush ← 1，直接將當下在 IF 階段 fetch 到的指令和計算出的下一個 PC 位置都改成 0；IDFlush ← 1 & EXFlush ← 1，則是將當下在 ID 及 EX 階段的任何 control signal 都改成 0，使得在那兩個階段的指令不管

是要執行哪一種計算，或是動作，都沒有辦法真正執行。

Problems encountered and solutions:

這次的 Lab 作業遇到了不少問題，整整花了四天才完成。由於這次 Lab 是寫 Pipe-Line，在 debug 的時候變得困難許多，因為發生問題的那一個 clock，通常並非是那一個 clock 所產生的 instruction 引起，反而很可能是前一個(或是前兩個、前三個)clock 所產生的 instruction 造成的，所以每次發現波型圖上有奇怪的地方時，都要往前面幾個 clock 回推，才能真正找到引起問題的 instruction 或是位置(IF or ID or EX or MEM or WB)。時序的問題造成我很大的困擾，所以才花了非常久的時間在 debug。

Debug 過程中，最常遇到的問題是接錯線，這種類型的 bug 大概出現了三四次。接錯線這種問題只能說自己不夠小心，沒有看清楚 Pipe-Line 的線路圖，或是在新增一個元件的時候，忘記重新將線路連接完善。比方說，當我在兩個本來相連的元件(其實是 ID_EX 和 EX_MEM)之間放入一個 MUX，我卻忘記把 MUX 選出來的結果接到 EX_MEM，而是繼續讓 ID_EX 的 output 送到 EX_MEM，使的新增的 MUX 根本發揮不了任何作用。

另外，最後一組測資(CO_P4_test3.txt)發生了一個很大的問題，就是前幾次 Lab 可以安全運作的 code，用到了第三組測資時竟然就完全失效，以下將進一步說明。

ALU 內部的 code 我是按照 Lab2 PDF 中所提供的寫法撰寫，而非直接調用 Lab1 寫好的 ALU。其寫法如下：

```
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 ;
        default: result_o <= 0;
    endcase
end
```

因為本來都是假設這段 code 是正確無誤的，所以那時候發現結果不如預期時，都一直在檢查是否是因為 branch 的失敗，是不是沒有成功的 branch 或是 branch 時沒有把緊接著的三個指令 flush 掉，檢查了很久卻發現這些好像並沒有出錯。所以又再仔細對照了每一個 register 的變化，結果發現$8 的變化其實不合預期，照理來說每次迴圈$8 都會減少 4(addi $13, $0, 4 & sub $8, $8, $13)，然後再跟 0 比較，若$8 小於 0 就會跳出迴圈(slt $2, $0, $8 & beq $2, $1, -11)。但在波型圖當中看到的情況是$8 從來不會小於 0，最小也只會到 4。所以就開始檢查這四行 code，結果發現 slt $2, $0, $8 十分詭異，竟然會在$8 比 0 大的時候輸出 0！所以開始改寫 ALU 當中的 7(也就是 SLT)，經過反覆嘗試後，終於使的$8 比 0 大的時候 SLT 可以輸出 1，然後才終於完成了這次的 Lab4 作業。改寫後的 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:begin slt = src1_i - src2_i; result_o = slt[31]; end
        9:result_o <= src2_i >> src1_i ;
        default: result_o <= 0;
    endcase
end
```

Finished part and summary:

Finished part:

Test1、test2及test3全部完成。

Summary:

從Project1到這一次的Project4，從最一開始的ALU一直到這次寫簡易版的Pipeline，很扎實地把上課內容完整的實做出來，覺得學習成果相當良好，真的是學到很多。這一連串的project內容，每一次作業發下來，從codeing、debug到Report都是自己完成，一階段接著一階段的把這一個複雜的電路做出來，不但讓我對於計算機組織課程的學習，了解得更為透徹，更讓我感到很有成就感，覺得自己對於Verilog及模擬產生的電路圖也不是完全地不在行。