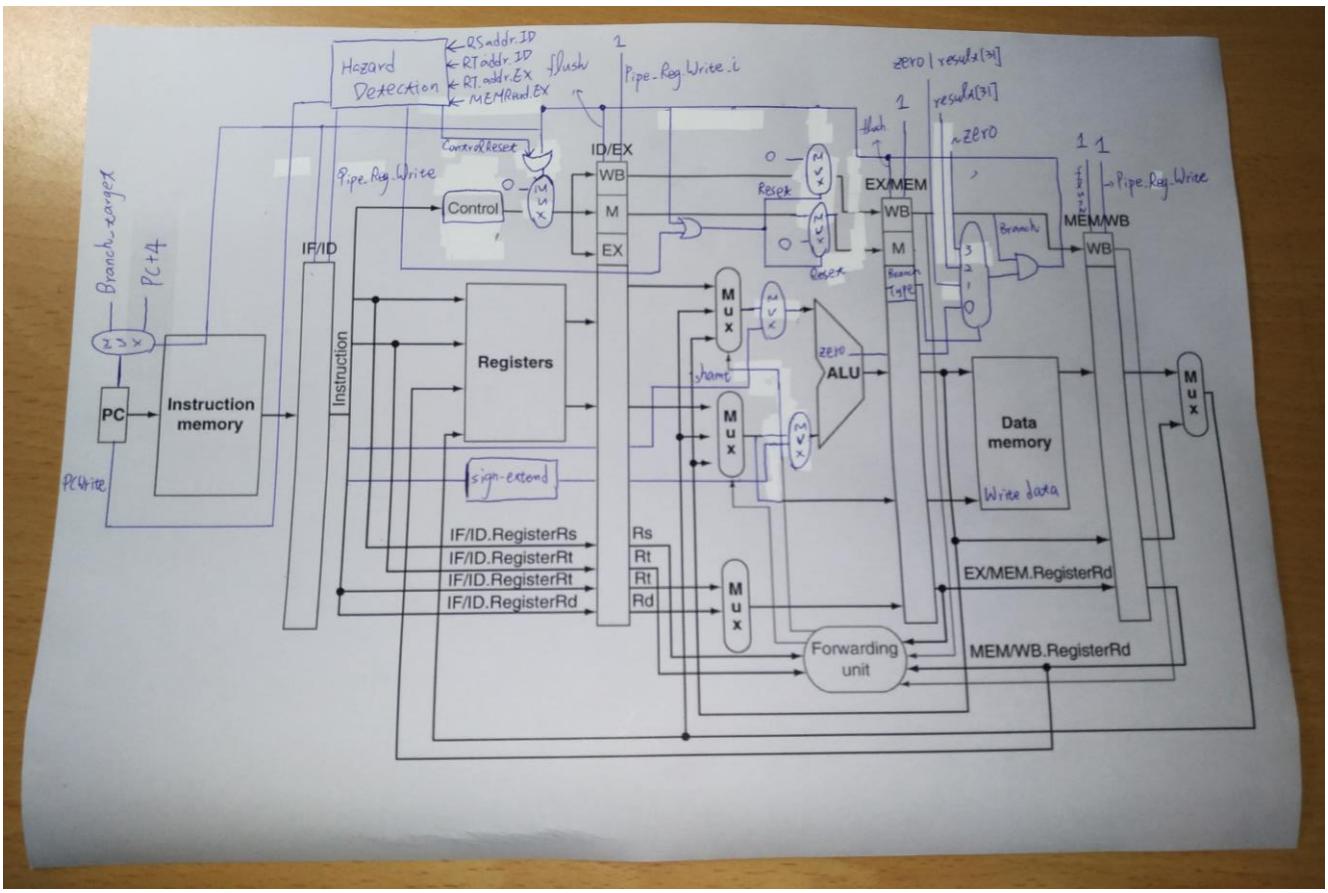
0416315 王定偉、0416005 張彧豪

Computer Organization Lab4

Architecture diagram:



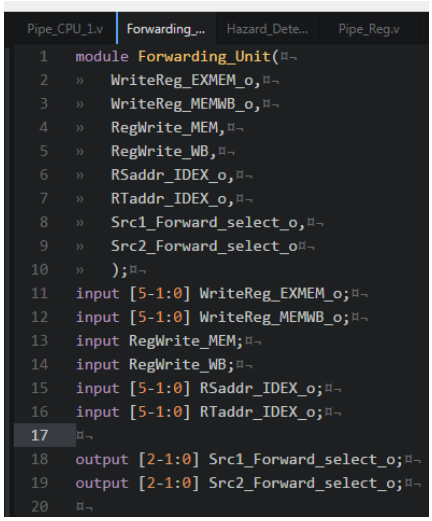
Detailed description of the implementation:

一、以 Pipe_CPU_1 為主要的模組，此模組會去呼叫它所需要的所有模組，包含 PC、IM、RG、Decoder、ALU_Ctrl、Sign_Extend、MUX、ALU、Data_Memory、PipeReg、Forwarding_Unit、Hazard_Detection_Unit 等模組。此 CPU 為 pipeline 版本的 CPU。

二、這次大部分的模組都跟上一次 lab 一樣，唯獨新增了 pipeline 所需要的 Pipe_Reg、Forwarding_Unit、Hazard_Detection_Unit 這三個模組。而主要模組也與上次大同小異，只差在這次的主模組須呼叫 Pipe_Reg 等模組。

三、Forwarding_Unit 模組說明：

1. 輸入輸出：



2. 此模組的功能就是判斷需不需要 forwarding 需要的值給前面的 stage，判斷方法是講義上那種判斷方法。

3. 輸出對應說明：

Src1_Forward_select_o, Src2_Forward_select_o	運算值來源
00	原本 register 讀取的值
01	EXMEM pipeline register 傳回來的值
10	MEMWB pipeline register 傳回來的值

四、Decoder 模組說明：

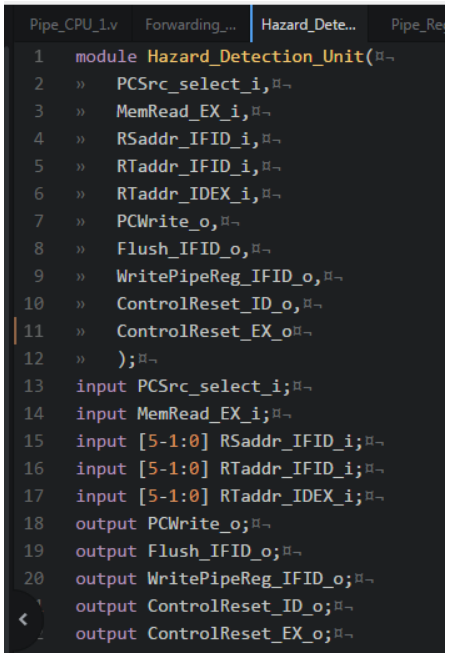
1. 輸入輸出：

```
module Decoder(
    instr_op_i,
    Branch_o,
    MemToReg_o,
    BranchType_o,
    //Jump_o,
    MemRead_o,
    MemWrite_o,
    ALU_op_o,
    ALUSrc_o,
    RegWrite_o,
    RegDst_o
    );

//I/O ports
input  [6-1:0] instr_op_i;

output  Branch_o;
output [2-1:0] MemToReg_o;
output [2-1:0] BranchType_o;
//output  Jump_o;      //Need
output  MemRead_o;
output  MemWrite_o;
output [3-1:0] ALU_op_o;
output  ALUSrc_o;
output  RegWrite_o;
output [2-1:0] RegDst_o;
```

輸出的控制訊號一個 14 bit。

五、Hazard_Detection_Unit 模組說明：

1. 輸入輸出：



2. 此模組的功能是判斷有無 load-use data hazard 發生。
3. 當有 Load-use data hazard 發生時，PCwrite_o 要為 0，因為要 stall 一個 cycle；WritePipeReg_IFID_o 也要為 0，重新 decode 一次指令；Flush_IFID_o 要為 1，因為要將原本指令變為 bubble 指令。
4.

六、Pipe_Reg 模組說明：

1. 輸入：clk_i, rst_i, data_i, Pipe_Reg_Write_i, Flush_i，輸出：data_o
2. Flush_i 的用意在於判斷是否要將 pipeline register 裡面的重設為 0，在 control hazard 時會需要用到；Pipe_Reg_Write_i 的用意在於當 Flush_i 為 0 時是否要寫入新值進入 pipeline register，這在需要 stall cycle 的時候會用到。

七、因為有 control hazard，所以有可能要將某些 stage 的控制訊號 reset 成 0。在這次 lab 中，若 branch 發生了，則需將 ID、EX 這兩個 stage 的控制訊號 reset 成 0，這部分的工作是交給兩個 MUX 來做，分別是 Mux_ControlReset_ID 及 Mux_ControlReset_EX，經過處理的控制訊號分別為 Real_Control_IDEX_i 及 Real_Control_EXMEM_i。

八、各階段 Pipeline register 說明：

1. IF/ID pipeline register
   i. 資料量：64 bit。
   ii. 資料：PC+4(32bit)、instruction(32bit)。
   iii. Pipe_Reg_Write_i 是由 Hazard_Detection_Unit 的 WritePipeReg_IFID_o 傳來的。
2. ID/EX pipeline register
   i. 資料量：189 bit
   ii. 資料：
      ① Real_Control_IDEX_i (14bit)
      ② PC+4 (32bit)
      ③ shift amout (32bit)
      ④ RSdata (32bit)
      ⑤ RTdata (32bit)
      ⑥ SE_data_o (32bit, sign-extend data)
      ⑦ RSaddr (5bit)
      ⑧ RTaddr (5bit)
      ⑨ RDaddr (5bit)
   iii. Pipe_Reg_Write_i 永遠是 1，因為不會在這個階段有 stall 發生。
3. EX/MEM pipeline register
   i. 資料量：142 bit
   ii. 資料：
      ① Real_Control_EXMEM_i
      (8bit, {Branch_o, MemtoReg_o, BranchType_o, MemRead_o, MemWrite_o, RegWrite_o})
      ② Branch_target_o (32bit)
      ③ zero_o (1bit)
      ④ result_o (32bit, ALU result)
      ⑤ RTdata_IDEX_o (32bit, RTdata come from IDEX pipeline register)
      ⑥ WriteReg (5bit, Write Register address)
      ⑦ SE_data_IDEX_o(32bit, sign-extend data from IDEX pipeline register)
   iii. Pipe_Reg_Write_i 永遠是 1，因為不會在這個階段有 stall 發生。
4. EX/MEM pipeline register
   i. 資料量：104 bit
   ii. 資料：
      ① Ori_Control_MEM (3bit, {MemtoReg_o, RegWrite})

② MEM_Read_data_o (32bit, Date read from Memory)
③ ALU_result_EXMEM_o (32bit, ALU result come from EXMEM pipeline register)
④ Write_Reg_EXMEM_o (5bit, Write Register address come from EXMEM pipeline register)
⑤ SE_data_EXMEM_o (32bit, sign-extend data from EXMEM pipeline register)

iii. Pipe_Reg_Write_i 永遠是 1，因為不會在這個階段有 stall 發生。
iv. Flush_i 皆為 0，因為 branch 發生時不會影像到在此 stage 的指令。

5. 當 Branch 指令在 MEM stage 發現要 branch 時，PCSrc_select_o 會變成 1，代表要選 branch target 當作下一個 pc number，而這個 PCSrc_select_o 也是 IFID、IDEX、EXMEM 三個 pipeline register 的 Flush_i 訊號來源。

九、Stall 方法說明，在需要的 stall 的 register 上面我們是利用控制 Pipe_Reg_Write_i 來控制要不要將 pipeline register 的值保留住，當其值為 1 時，代表將新的值寫入 register；反之則不寫入新值，利用這種方式達到 stall 的效果。

Problems encountered and solutions:

1. Register 不能同時讀寫的問題，照理說如果在 ID stage 要對同一個 register 做讀跟寫的動作應該是不會有 hazard 的問題，但是在測第一個測資的時候發現當現在的 cycle 要同時對同一個 register 做讀取時無法讀到正確的值出來。看了一下 code 發現原來是寫法造成的，因為當這個 cycle 要寫入 register 時，要下一個 cycle 那個新的值才會顯現在 register 上，這樣的現象造成讀取該 register 永遠讀到舊值，解決方法就是在要寫也要讀的時候，直接把要寫入的新值指派到輸出，即可解決此問題。如下圖所示：

```
//assign RSdata_o = Reg_File[RSaddr_i] ;
assign RSdata_o = ((RSaddr_i==RDaddr_i) && RegWrite_i) ? RDdata_i : Reg_File[RSaddr_i];
//assign RTdata_o = Reg_File[RTaddr_i] ;
assign RTdata_o = ((RTaddr_i==RDaddr_i) && RegWrite_i) ? RDdata_i : Reg_File[RTaddr_i];/*-*/
/*-----------------------------------------------------------------*/
```

2. Forwarding 判斷的方法的流程其實一開始很困惱我們，因為一開始我們是分成四個 case 分開處理，可是這樣處理的結果就是 select 訊號會被改兩次，造成訊號錯誤。後來我們直接分成兩種 case，一種是要 forwarding 給 operand 1 跟要 forwarding 給 operand 2，至於是要從哪裡 forwarding 則是留作每種 case 內部判斷，這樣確保 select 訊號不會被改到兩次。

Lesson learnt (if any):

1. 簡易版 Pipeline CPU 的實作方法。
2. Control hazard 的處理方法。