Computer Organization

**Architecture diagram:**

**Detailed description of the implementation:**

PipeLine CPU - Overview

這次的lab重點是將的Lab3的CPU轉為pipeLine版本。除了要產生pipeLine Register以外，還需要處理因為pipe Line而產生的data hazard & control hazard,這兩項問題分別是透過新增forwarding unit 與 hazard detection unit來解決。另外，因為PipeLine的產生，所以將硬體平均分配在各stage也是十分重要的一件事情。

**Pipe\_Reg**

Pipe\_Reg 主要是拿來儲存各個階段訊號的的結果，並將需要在往後 stage 使用到的訊號傳下去。整個模組使用參數式寫法，總暫存器長度是由使用到的模組決定，而會外部用到pipe\_Reg的模組也必須自行確保接進來的資料是輸入輸出有兩兩對應到的。

**Forwarding Unit**

(往後都將 Forwarding Unit 簡稱為FU)

這個東西專門處理Data Hazard, Data Hazard即指正確的答案已經在CPU其中一個stage中出現，但是因為該stage還沒將答案寫回正常可取用的地方(暫存器)的情況。FU 就是負責判斷哪些stage的資料擁有依存性，並在需要的時候將正確資料傳送給需要的stage,以確保運算“最後結果”的正確性。

實作細節：

因為alu的兩個輸入可能都來自於register, 且EX stage 後方還有MEM,WB兩個stage。所以我們必須針對alu的兩個輸入都進行forwarding判斷、以及將forwarding的來源設置為MEM,WB兩個階段。FU會判斷alu的每個輸入來源的register代號,同時也會知道在MEM,WB階段的指令如果要寫入的話會寫到哪個reg之中，只要MEM,WB的指令打算要寫入暫存器，且FU比對EX階段使用的暫存器與後方兩個階段具有相依性，forwarding機制就會啟動並根據來源遞送正確資料。

在這裡會出現MEM,WB兩個階段接寫入到同一個reg的問題，所以我們必須要定義遞送的資料來源優先性。我們定義MEM的優先性比WB高，因為在時序上MEM的資料會是較新的。

另外，FU不需要知道rs,rt欄位的值是否真的有被使用到(有意義)，因為對應到不同的指令類別，decoder都會建造出正確的datapath，即便沒有用到rs但FU進行forwarding的話reg的資料也是不會流進alu中並被使用到的。

因為forwarding unit會用到MEM,WB的rd reg值，所以必須要把經過mux過濾後的rd(真正會寫回的reg代號)傳回forwarding unit中。

**Hazard Detection Unit**

(往後都將 Hazard Detection Unit 簡稱為HDU)

因為pipeLine的存在，可能會存在有些資料並沒辦法透過forwarding來即時取得，也因為branch指令的存在，可能沒辦法即時確定接下來要執行哪一段指令而使CPU在執行branch後必須先行預測結果。 HDU會檢查以上兩種情況的發生，並確保執行的先後順序與安全性。

實作細節：

把需要判斷的情況分成兩種

1.因為Load use hazard 而需要 stall pipeLine

Load use hazard的偵測是透過觀察 EX階段的指令是否需要讀取記憶體，以及剛剛decode完的ID階段指令所需要的reg資料是否來自EX階段讀取的資料。如果都為真，這會造成沒辦法敨過現有架構遞送正確資料的情況。需要ID stage的指令清空，並將其後的指令全部往後延遲。

2.branch mispredict

這個CPU的branch預測機制就是猜測所有的branch都是Not taken, 也就是說， CPU再遇到branch指令後會繼續讀取接下來的指令。這種static prediction的方法非常好寫，並透過branch於EX stage後執行的結果來確認預測結果。如果最後發現是要跳轉，那就把這段期間多讀進pipeline的後續指令全部清空。並更改PCSrc來讓PC讀取跳轉後該讀取的指令位置。

**Mux\_3to1**

在上一次Lab中，提到沒有實踐3to1的Mux而是使用4to1的Mux來替代，因為兩者的I/O界面完全相同，且使用的4to1Mux數量較多。但這次lab並沒有使用到4to1的Mux, 所以實踐3to1來簡化合成後的電路。(3to1的Mux doncare情況較4to1多，合成上的簡化應該也會較好)

*更動的硬體硬體單元(與lab3比較)*

**Reg\_File**

修改成為能夠在同一個clock同時進行寫入與讀取而不會出問題，且在同一個clock cycle時永遠先寫入再進行讀取。會這樣設計的原因是假設一個指令在CPU執行時，總是會在前面的stage讀取資料，再在後面的stage才進行寫入。

轉而使用存在多個stage CPU的角度來想，會進行寫入的指令在程式碼中的執行順序一定是發生在進行讀取的指令之前。所以Register File 如果要在同一個cycle進行讀取跟寫入的話，就一定要設計成先寫入後讀取，才能在符合程式序列執行的假定。

**PC**

針對hazard detection Unit的stall功能，PC增加了一個新port: pcWrite,這個介面可以讓PC決定下一個 clock edge 要不要更新PC的值，搭配上IF\_ID PipeLine Reg的 Write 介面跟HDU後可以做出 stall 的行為。

**Decoder, ALU\_control, ALU**

這些架構在基本上都沒有多大變遷，只是這次實踐的總指令類別較Lab3少，所以對應的拔掉了一些對這次Lab不需要的東西。

Lab4比Lab3還少支援的指令有:

+ Jump類指令

+ JR指令

+ 較細分的branch類別指令(ble, blt)。

只是針對指令的內部編碼並不是連續的，因為預期之後可能會再將指令期擴充成跟lab3相同，保持彈性。

**整體架構寫法**

把內部模組依據stage的位置編排放置，這樣比較好維護，且在每個stage的變數上分別註記是哪個stage所使用到的資料，這樣才不會有所混淆。

**Problems encountered and solutions:**

1.發現 PipeLine 沒有控管好，不小心把不同stage的線直接接再一起，導致指令亂掉

2.pipeLine reg的 rst訊號很重要，是用來清除開機時的不穩定狀況，一開始沒接結果讓訊號不太乾淨，後來修正了。

3.更改RegFile 模組，讓同一個clock可以同時讀取跟寫入而不會有資料問題

4.IF\_ID的 rst訊號比較特別，是active low reset 對應到的邏輯式 為rst\_i & ~IF\_Flush ,其中有一個訊號掉下來整個輸入就要掉下來。

5.把 hazard dection 的 branch 輸入接錯了，接成 branch\_MEM 而不是 PCSrc, 導致只要有branch就會flush

6.發現沒有把branch 訊號設為會被flush的訊號,所以把它加進去

**Lesson learnt (if any):**

這次的lab有很多小細節在還沒實做之前完全不會想到，但是這次在還沒做之前太執著在空想。

結果一次思考太多問題卻因為問題太多而完全沒有頭緒來解決。最後還是老老實實一次只想一些比較實際的小問題，每次都規劃一些小進度才慢慢把整個cpu給組起來。這種方法明明平常生活都在用，但我是在這次lab後才真正在寫code中使用這種方法。以後可能都會想辦法傾向這種模式吧!