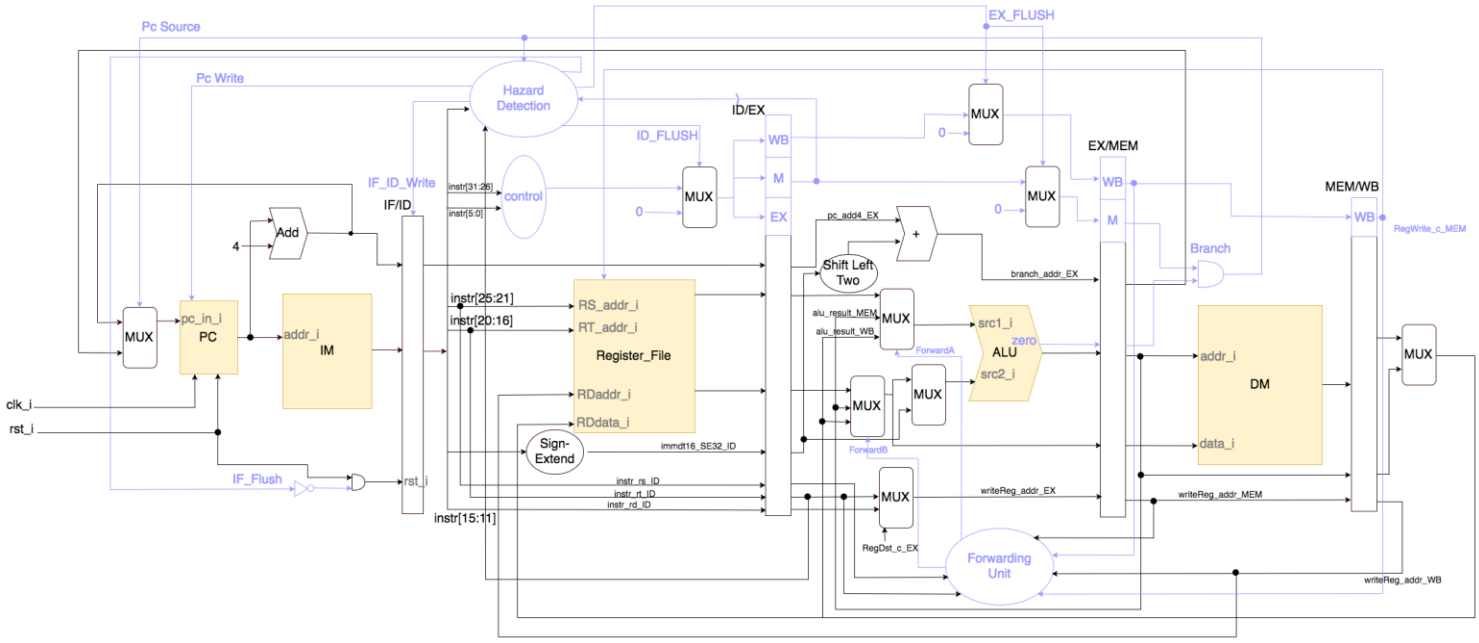
0411276 陳奕安 0413335 郭逸琳

Computer Organization

Lab4 - 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。

0411276 陳奕安 0413335 郭逸琳

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!