Computer Architecture 2018 Fall
Project 1 Report

Team's Name

李胖一級胖

Members

B05902017 王盛立

- 工作比例: 35%
- 工作內容: 參與討論、接線、Debug，程式撰寫

B05902033 高晟瑋

- 工作比例: 35%
- 工作內容: 參與討論、接線、Debug，程式撰寫

B05902105 余友竹

- 工作比例: 30%
- 工作內容: 參與討論、接線、Debug，整理Report

Pipelined CPU Implementation

我們參考了上課投影片以及作業指示，照著Data Path & Module的圖，一步一步串連CPU的接線。

大體上的實作方向分成兩個部分：

1. 完成PipeLine式的處理模式
    - 將CPU區分成了五個Block。同一時間內，希望最多可以有5個不同的Instruction同時被執行(不考慮Flush、Stall的狀況下)。
    - 我們實作了四種Unit，包括IF_ID, ID_EX, EX_MEM, MEM_WB。每個Cycle都會讀進一個Instruction，並output出去，以達成上述的目標。
    - Pipeline之所以能夠提升效率的原理是，每個Instruction在從讀入到執行結束的過程中，同一時間內僅會占用其中兩個Unit的資源。這時，其他空閒的Unit可以再讀入其他的Instruction，以達成不浪費CPU運算資源的目的，提升運算效率。
2. 處理各種Hazard的問題
    - 此部分實作在Module的部分，將會在後面說明。

Modules

CPU

- 控制Modules輸入與輸出所對應的其他Modules。

MUX_PC

- 主要負責處理beq program counter的情況。

PC

- 根據前面MUX_PC選進來的值，傳送program counter
- 這裡根據Hazard_Detection傳進的訊號，可能會有Stall的需求
- Stall實作的原理其實就是將output的program counter改成上一個program counter。

ADD_PC

- 將每次的Program Counter傳進來，+4後回傳到MUX_PC做選擇。

Instruction Memory

- 讀出那個Program Counter應該對應的Instruction

IF_ID

- 重點在若有遇到Flush的情況，就必須將Inst_o設為0
- 而若有Hazard的情況，則Inst_o維持不變
- 經過一次的Clock Time後，將output傳送到對應的input上。

Hazard_Detection

- 若偵測到有Hazard的可能(根據助教投影片的提示、上課講義的介紹)，就必須傳送訊號告訴PC、IF_ID做Stall的動作。

Control

- 先在Module找出該Instruction對應的指令，用0~8來代表，並傳給ID_EX，可以快速地決定如是否要寫入Memory/Register，或著是判斷ALU的Operation。

Adder_Shift

- 將有beq判斷時的Immediate值補成正確的值，但此步不做判斷，判斷是交由MUX_PC負責。

Sign_Extend

- 將Immediate補成32-bit

Register

- 跟上次作業一樣，主要是從Register的addr.中讀出對應的Data，若有必須寫入的部分(寫入Register)，同樣也在這個Module處理。

muxhazard

- 判斷是否有Hazard的情況，在這裡若有遇到Hazard，我們會把Control設的指令改成新的參數`1111`並送到ID_EX。
- 若ID_EX遇到這個指令，就會將後續所有的Control Flag設成0，表示後面都不要再做事了。

Equal

- 從Control送來的指令若為beq，就要對兩個送過來的data進行比較，若一樣，則要Jump；若不用，則照常進行。
- 這個Module的實作上有重大的意義，在正常的實作下，會在EX階段確定是否相等(ALU)
- 但等到EX階段才判斷的話，若predict錯誤，就必須Flush掉兩個Instruction
- 而這個Module在ID階段就判斷完成，提前了一個階段，少讀了一個Instruction，這時就算predict錯誤，也僅要flush一個Instruction，提升整體效率。

ID_EX

- 同樣經過一次的Clock Time後，將output傳送到對應的input上。

MUX2/MUX3

- 這兩個Mux置於ALU前，用以判斷是否要用forwarding來取代當前的data

ALU

- 跟HW4的實作過程沒什麼不同，只是多了三種指令
  - 其中若是sw的指令，必須把傳送進來的兩個data相加。
  - 若是lw的指令，同樣也是把兩個data相加。
  - 而beq的指令被移動到Equal的地方先行判斷了。

MUX_ALUSRC

- 選擇使用data2(R-Type Instruction)或Immediate(addi, ld, sd...等)的值去做ALU運算

forwarding

- 根據投影片的提示完成，核心概念便是若前一個人的RD等於我所要取用的Register(RS/RT分別對應送至MUX2/MUX3)的值，則必須做forwarding的動作。

EX_MEM

- 同樣經過一次的Clock Time後，將output傳送到對應的input上。

Data Memory

- 若遇到有需要寫入到Memory的Instruction，會開啟`Mem_write_i`的FLAG，此時就會將對應的Data寫入對應的addr.
- 而若是有需要從Memory讀取東西的指令，則是從Input進來的addr.中找到對應的data，並傳送出去。

MEM_WB

- 同樣經過一次的Clock Time後，將output傳送到對應的input上。

Implementation Difficulty

- 中途遇到很多牽線時的錯誤，分工上命名的差異、Coding Style的不同....等小錯誤不計其數。
- 語法上的不精熟也是很嚴重的問題

    - 例如，`=`會直接進行賦值，但`<=`卻會在`begin end`結束後才賦值。這樣的小錯誤在我們初次撰寫verilog的時候根本不會特別注意，但直到遇到很多小bug後才了解其中的差異。
    - 在弄清楚各種不同的語法差異後，一步一步改正，才解決問題。
- 在進行Debug時相當不易。我們是先設計完所有的Module再一次測試所有的Instruction。這樣的壞處在於，一旦發生問題(輸出的值跟想像的不同)，很難找到錯誤。因為我們並不清楚究竟是哪個環節出錯。
    - 解決的辦法是，印出每個階段的參數，一個一個檢查是否不如預期。
    - 另外，我們在測試時，一次只測試一條新的Instrucion，直到這條Instruction完全符合預期。