Computer Architecture Lab2 Report

Name: 何明洋
Student ID: r11922208

訂閱

Modules Explanation

- Note: 以下針對增加或更動(from Lab 1)的modules進行解釋

New Modules

`Forwarder`

邏輯參照spec，簡而言之是以 `EX_Rs1`, `EX_Rs2`, `MEM_RegWrite`, `MEM_Rd`, `WB_RegWrite`, `WB_Rd` 來決定 `Forward_A`, `Forward_B` 的值。在檢查當下階段的 `RegWrite` 與 `Rd` 後，對於 `Forward_A`, `Forward_B` 數值定義上都是若 `MEM_Rd` 與 `EX_RsX` 相等就是forward from MEM to EX；
若 `WB_Rd` 與 `EX_RsX` 相等就是forward from WB to EX。

`Hazard_Detection`

照著spec邏輯，若偵測到lw (`EX_MemRead` =1及 `ID_RsX = EX_Rd`)，那就要stall，故設定 `NoOp=1` 與 `Stall=1` 。

`Imm_Gen`

這個module就是根據instruction[6:0]再來拿instruction不同的區域拼湊出immediate，照著spec刻即可完成。而為了方便檢查，定義了 `I_TYPE_ALU=7'b0010011`, `I_TYPE_LW=7'b0000011`, `S_TYPE=7'b0100011`, `SB_TYPE=7'b1100011` ，後續也會沿用這些定義的常數。

`Branch`

藉由pc與immediate，對於 `branch_pc` 的計算在這個module中處理，以及output判斷是否為branch的control signal `branch_ctr` 。為了實作方便， `branch_ctr` 可以由簡單判斷是否為branch instruction以及register的值是否相同來設定。

`MUX32_Double`

是Lab1中MUX32的延伸版，input 4個訊號，用2 bits來決定要output哪個。實作上就是直接一對一即可，使用case來處理。

`Pipeline_IF_ID`

定義了IF_ID stage，直接以spec來拉線。參考 `PC` module作法，當 `if(~rst_i)` 時就進行signal初始化。接著考慮若是 `Stall` 就不做事維持原樣， `Flush` 就清掉pc與instruction；其餘則直接把pc與instruction傳下去。

`Pipeline_ID_EX`

定義了ID_EX stage，直接以spec來拉線。參考 `PC` module作法，當 `if(~rst_i)` 時就進行signal初始化，其餘狀況就把signal傳下去。

`Pipeline_EX_MEM`

定義了EX_MEM stage，直接以spec來拉線。參考 `PC` module作法，當 `if(~rst_i)` 時就進行signal初始化，其餘狀況就把signal傳下去。

`Pipeline_MEM_WB`

定義了MEM_WB stage，直接以spec來拉線。參考 `PC` module作法，當 `if(~rst_i)` 時就進行signal初始化，其餘狀況就把signal傳下去。

Modified Modules

`Adder`

與Lab1相同，直接使用。

- 這個 `Adder` 主要是拿來執行 `PC=PC+4` ，故簡單實作了一個接受兩個32 bits source並回傳相加值的module，而加法本身即有 + operation支援。

`Control`

比起Lab1，增加了對於 `lw` , `sw` , `beq` 的處理。根據題目要求的data path， `Control` module接受 `instrucion[6:0]` , `NoOp` ，回傳 `ALUOp` , `ALUSrc` , `RegWrite` , `MemtoReg` , `MemRead` , `MemWrite` , `Branch` 。

首先只要是 `NoOp` 為1的話，那就是stall，故所有回傳值直接設為0。

若 `NoOp` 為0，根據題目給的data path可以整理成下表：

TYPE	opcode	function	ALUOp	ALUSrc	RegWrite	MemtoReg	MemRead	MemWrite	Branch
R_TYPE	0110011	and, xor, sll, add, sub, mul	10	0	1	0	0	0	0
I_TYPE_ALU	0010011	addi, srai	11	1	1	0	0	0	0
I_TYPE_LW	0000011	add	01	1	1	1	1	0	0
S_TYPE	0100011	add	01	1	0	0	0	1	0
SB_TYPE	1100011	sub	00	0	0	0	0	0	1

直接使用case來判斷opcode，把所有其他值assign完即可。

`MUX32`

與Lab1相同，直接使用。

- 這個module input `src0` 與 `src1` ，用 `select` signal選擇其一output，故實作上相當容易，當 `select` 為 `0` 就output `src0` ，否則為 `src1` 。

`Sign_Extend`

與Lab1相同，直接使用。

- 這個module要input一個12bits的 `immediate` ，回傳一個sign extended 32 bits output，所以實作上就是把最高位的bits複製20次。對於這樣的操作，參考了課堂上建議的HDLBits教程，用 `{20{data_i[11]}}` 這樣的寫法達成複製的操作。

`ALU_Control`

比起Lab1，在Lab2要處理的狀況變更多了。但回傳的 `ALUCtr` 仍為八種，而整個邏輯根據Control中的表格可以再整理成下表：

ALUOp	func7	func3	function	Self-defined ALUCtr
10	0000000	111	and	000
10	0000000	100	xor	001
10	0000000	001	sll	010
10	0000000	000	add	011
10	0100000	000	sub	100
10	0000001	000	mul	101
11	xxxxxxx	000	addi	110
11	xxxxxxx	101	srai	111
01	xxxxxxx	xxx	add	011
00	xxxxxxx	xxx	sub	100

實作上先從 `ALUOp` 判斷，假設是 `00` 那直接回傳 `100` ；假設是 `01` 那直接回傳 `011` ；假設是 `10` 那要判斷func7+func3共六種形式來回傳 `ALUCtr` ，實作與Lab1相同；針對 `11` 的case，則只要判斷func3是哪個再回傳對應的 `ALUCtr` 。

ALU

與Lab1相同，僅把原本全部使用if else改成case來實作，並移除原本用不到的 `zero` 。

- 根據data path， `ALU` module要input `src1` , `src2` , and `ALUCtr` ，回傳計算結果 `res` 。

- 根據前一個 `ALU_Control` module，每一種function都已經有對應的 `ALUCtr` signal，所以只要簡單判斷即可：

function	Self-defined ALUCtr	operation
and	000	&
xor	001	^
sll	010	<< (unsigned)
add	011	+
sub	100	-
mul	101	*
addi	110	+
srai	111	>>> (signed)

- 因verilog已經提供了這些operation，所以根據對應結果對 `src1` 與 `src2` 操作就完成了。唯一要注意的是 `srai` 只拿 `src2` 5 bits做操作。

CPU

與Lab 1類似，最後就是根據題目提供的data path圖把所有module串起來。串接上需要額外加入wire，但Lab 2因為使用pipeline，許多wire要額外寫不同stage的版本。而Lab2因引入了lw而需要有flush, stall的機制， `Hazard_Detection` 在偵測到後便會送 `NoOp` 與 `Stall_o` 給 `Control` module。而透過 `ID_branch_ctr` 在發現有branch taken時也會設定 `ID_FlushIF` 來讓IF_ID stage flush instruction。

Difficulties Encountered and Solutions in This Lab

- 為了避免環境問題，本此嘗試同步在Container裡寫，但docker-compose似乎有各種版本的問題導致config會有差異，加上 `-build: context:` 才得以執行。
- 本次Lab 2比起Lab 1有更多的判斷條件要處理，在Lab 1我都用? :來處理，但發現Lab 2這樣寫起來會過度複雜，故學習了case的寫法。
- 但用case寫法後發現一直出錯，才了解到若用case或if，那要用output reg來命名變數。
- 而上述的判斷許多都是要確定signal是不是某個值，導致整份code非常混亂，故改用define variable的方式避免太多magic number存在在code中。
- 引入pipeline後要串起CPU變得相當複雜，常常漏寫了幾個wire，最後透過詳盡的畫圖並把變數名稱命好才得以完成。

Development Environment

這個作業主要是在 `MacOS 12.3` 搭載 `Apple M1` 晶片上開發的，並也在 `Ubuntu 20.04` 環境中起 `Ubuntu 22.04` Docker container(本題提供的 dockerfile)測試過