R10922171 陳嘉政 Lab1

- ## Development environment

  OS: Windows10
  IDE: visual code
  Compiler: iverilog

  使用 vs code 來編寫 verilog 語言，寫完之後利用 iverilog 來 compiling 產生出.vvp 檔，再從 command line 輸入 vvp xxx.vvp 來得到波型檔(.vcd)。之後開啟 gtkwave 程式，打開 CPU.vcd 觀察波型變化來逐一驗證每個 module。

- ## Modules explanation

  為避免編譯時會產生無法辨認是輸入還輸出的 warning，特別將每個 output port 及要接的 input port 額外宣告 wire 來接。

  - ### IF state

    1. **PC:**
       與 hw4 給的 PC 相似，只是 input 多了 PCWrite_reg。
       第一個 always block 作用為:訊號在 rst_i 的下降邊時會把 pc_o 32 bits 全設為 0。第二個作用為:只有在 clk_i 的上升邊才會做，如果 PCWrite_reg 為 1 表示 pc 可以更新，然後再看 start_i 是否為 1，若為 1 表示 pipeline 開始執行，將 pc_i 設給 pc_o，否則 pc_o 維持 pc_o，若 PCWrite_reg 為 0 則不做任何事。pc_o 的輸出為 instr_addr。
       (額外在 CPU 宣告 wire PCWrite 及 reg PCWrite_reg，並把 PCWrite_reg 初始為 1(表示不需要暫停)，用 always block，只要 PCWrite 值有變動，就將其設給 PCWrite_reg)
    2. **PC_Adder:**
       Input 分別為 pc 輸出的 instr_addr 及 32-bit 常數 4，將兩個值相加得到下一個 pc address next_pc，輸出給 PC_MUX 的 data1_i。
    3. **PC_MUX:**
       為 2 對 1 的多工器，輸入為 PC_Adder 輸出的 next_pc 及 branch_Adder 輸出的 branch target address，以及控制信號 flush_reg，藉由 flush_reg 的值來決定讓誰通過，0 讓 next_pc 過，1 讓 branch address 過，輸出傳到 PC 的 pc_i。
       (額外在 CPU 宣告 wire Flush 及 reg Flush_reg，並把 Flush_reg 初始為 0(因為一開始還沒遇到 branch)，用 always block，只要 Flush 值

有變動，就將其設給 Flush_reg)
(照 spec 給的訊號是 Stall 來說，作業要求的應該是 Stall 訊號暫停 IFID reg，但我的 IFIDWrite 訊號相反，0 才是暫停的意思，我的思路是這樣。做出來結果也是對的，請助教見諒)

4. **Instruction_Memory:**
   就是沿用 hw4 的 module。拿 PC 傳出的 instr_address 來讀取指令出來，並將讀出來的指令輸出為 instr 給 IFID register。

- ## IFID register:

是 pipelined CPU 才加入的暫存器，負責分隔開 IF state 及 ID state，輸入有 IFIDWrite_reg, PC 輸出的 instr_addr, instruction memory 讀出來的 instr, Flush_reg, clk_i 及 rst_i。第一個 always block 與 PC 相同，就是當訊號在 rst_i 的下降邊時，將兩個 output: IFID_instr_addr 及 IFID_instr 初始化為 0。第二個 always 作用為在 clk_i 的上升邊時，將 instr_addr 設給 IFID_instr_addr，instr 設給 IFID_instr。
(額外在 CPU 宣告 wire IFIDWrite 及 reg IFIDWrite_reg，並把 IFIDWrite_reg 初始為 1(表示不需要暫停)，用 always block，只要 IFIDWrite 值有變動，就將其設給 IFIDWrite_reg。Flush_reg 則與先前的相同)

- ## ID state:

1. **Branch_Adder:**
   也是 Adder，輸入分別為往左 shift 1 個 bit 後的 sign_extended constant 及 IFID_instr_addr，相加後即為 branch target address，輸出到 PC_MUX 的 data2_i。
2. **Sign_Extend:**
   將 32-bit 的 IFID_instr_addr 傳入，根據指令格式的不同，找出常數位元在哪些位置，拼成 12 bit 常數，然後再將 MSB copy 20 次，合成 32-bit 的 sign extended constant 輸出。
3. **Shift:**
   將輸入的 32-bit sign extended constant 往左移 1 個 bit，相當於乘以 2，然後輸出到 branch_Adder 的 data1_i。
4. **Register file:**
   與 hw4 不同的點在於，RD 暫存器號碼、要寫入 RD 的 data 及 RegWrite 信號都是從 MEMWB register 傳回來的，結果才會對。所以當傳回的 RegWrite = 1 則傳回的資料會在下一個 clock 來時寫入傳回的 RDaddr。
5. **Equal:**

這是為了將 branch 移到 ID state 決定是否跳才增加的 module，替代 ALU，將輸入的 RS1 與 RS2 的資料做比較，若相同，則輸出 1，否則輸出 0，輸出到 AndGate 的 data2_i。

6. **AndGate:**
   原本在 MEM，移到 ID。輸入分別為 control 輸出的 branch 信號及 Equal 的結果，兩個做 and，若為 1 表示 branch taken，輸出的 Flush 就設 1，否則就是 0。接著利用設定 Flush_reg 的方式將控制訊號傳給 IFID register 及 PC_MUX。
7. **Control:**
   與 hw4 相同，取 instr 的 0-6 bits，先判斷是哪類指令再設 control signals，若皆非本次的指令則將 control signals 全設 0。除了該 state 需要的 control signals 外，其他還沒用到的會沿著 pipeline register 傳下去。輸入還有 NoOp_reg，若為 1 表示要將所有 control signals 設為 0(nop)。
   (NoOp_reg 是由 Hazard_detection 輸出的 NoOp 來設定，初始為 0)

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

8. **Hazard_detection:**
   用來偵測指令間有沒有 load used data hazard。輸入分別為 RS1addr、RS2addr、IDEX 輸出的 IDEX_MemRead 及 IDEX_RDaddr，根據 spec 給的判斷方法即可判斷有沒有 load used data hazard，若有 hazard，表示要暫停 pipeline 一個 clock，讓 IF 及 ID state 的指令不變，將 control signal 全設為 0 傳下去即可將多 copy 一份傳到 EX state 的指令變無作用，所以 output: PCWrite = 0, IFIDWrite = 0, NoOp = 1;否則 output: PCWrite = 1, IFIDWrite = 1, NoOp = 0。之後設定 PCWrite_reg、IFIDWrite_reg 及 NoOp_reg 的值來控制其他 module。

- ## IDEX register:

  分隔 ID 及 EX state 的 register。輸入有除了 Branch 以外的 control signals 及 registers 傳出的 IFID_RS1data、IFID_RS2data 及 Sign_Extend 傳出的 sign extended constant、[9:0] funct (IFID_instr[31:25]與[14:12]的

合併)、RS1addr(IFID_instr[19:15])、RS2addr(IFID_instr[24:20])、RDaddr(IFID_instr[11:7])、clk_i 及 rst_i，每個 input 都會對應一個 output。

第一個 always block 用來初始化，如果遇到 rst_i 的下降邊，就將所有 input 對應的 output 設成 0。

第二個 always，如果遇到 clk_i 的上升邊，就直接把每個 input 設給對應 output。

- ## EX state

1. Forward:

用來偵測位於 EX state 的指令有沒有跟 MEM state 或 WB state 的指令有 data hazard。輸入有從 IDEX register 傳出的 IDEX_RS1addr 及 IDEX_RS2addr、從 EXMEM register 傳出的 EXMEM_RDaddr 及 EXMEM_RegWrite 還有從 MEMWB register 傳出的 MEMWB_RDaddr 及 MEMWB_RegWrite，output 為 2-bit 的 ForwardA 及 ForwardB。根據 spec 給的判斷方式來設定 ForwardA、B 的值，ForwardA 接到 RS1_MUX，而 ForwardB 接到 RS2_MUX。

(1)EX hazard

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs2)) ForwardB = 10
```

(2)MEM hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
     and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd == ID/EX.RegisterRs1)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
     and (EX/MEM.RegisterRd == ID/EX.RegisterRs2))

and (MEM/WB.RegisterRd == ID/EX.RegisterRs2)) ForwardB = 01
```

2. RS1_MUX:

輸入分別有從 IDEX 傳出的 IDEX_RS1data、MemtoReg_MUX 產生的結果 MEMWB_RDdata 及 EXMEM 輸出的 EXMEM_ALU_result，加

上控制線 ForwardA，若 ForwardA = 00，讓 IDEX_RS1data 通過；若為 01 讓 MEMWB_RDdata 通過，若 10 讓 EXMEM_ALU_result 通過，否則讓 IDEX_RS1data 通過。這樣一來就可以決定 RS1 正確的值並且輸出到 ALU_data1_i。

3. **RS2_MUX:**
   與 RS1_MUX 功能相同，可以決定 RS2 正確的值並輸出到 ALUSrc_MUX 的 data1_i。
4. **ALUSrc_MUX:**
   輸入分別為 RS2_MUX 的輸出 RS2_MUX_data、IDEX 傳出的 sign extended constant 及 IDEX 傳出的控制信號線 IDEX_ALUSrc。若 ALUSrc = 0，讓 RS2_MUX_data 通過，否則讓 constant 通過。
5. **ALU_Control:**
   輸入分別為從 IDEX 傳出的 2-bit 的 IDEX_ALUOp 及 10-bit 的 IDEX_funct(funct7+funct3)。用 slides 給的圖來做比對，先用 ALUOp 判斷指令格式，再由 funct 決定哪個指令並且賦予一個 4-bit 的 ALUCtrl 值，and 給 4'b0000、or 給 4'b0001、add 給 4'b0010、xor 給 4'b0011、sll 給 4'b0100、mul 給 4'0101、sub 給 4'b0110、srai 給 4'b0111，若指令用不到 ALU 或者皆非 spec 中指令，一律設為 4'dx，然後 ALUCtrl 傳給 ALU。

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

funct7	rs2	rs1	funct3	rd	opcode	function
0000000	rs2	rs1	111	rd	0110011	and
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000001	rs2	rs1	000	rd	0110011	mul
imm[11:0]		rs1	000	rd	0010011	addi
0100000	imm[4:0]	rs1	101	rd	0010011	srai
imm[11:0]		rs1	010	rd	0000011	lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011	beq

6. **ALU:**
   輸入分別為 RS1_MUX 傳出的 ALU_data1 及 ALUSrc_MUX 傳出的 ALU_data2，加上控制線 ALUCtrl。根據 ALUCtrl 的值做對應的計算，特別注意到 srai 的計算，constant 只取前 5 個 bits。將輸出 ALU_result 傳入 EXMEM register。若非 slides 給的指令，

ALU_result 直接設 32’dx。

- ## EXMEM register:

分隔 EX state 及 MEM state。輸入有 IDEX 輸出的 IDEX_MemRead、IDEX_MemtoReg、IDEX_RegWrite、IDEX_MemWrite 及 ALU 輸出的 ALU_result 及 RS2_MUX 輸出的 RS2_MUX_data 及 IDEX 輸出的 IDEX_RDaddr 加上 clk_i 跟 rst_i。
第一個 always，若訊號在 rst_i 的下降邊，將所有對應的 output 設成 0，與前面做法相同。
第二個 always，若訊號在 clk_i 的上升邊，將輸入設給對應的輸出，與前面做法相同。

- ## MEM state

1. **Data Memory:**
   輸入有 EXMEM 輸出的 EXMEM_ALU_result、DatatoMemory、EXMEM_MemRead、EXMEM_MemWrite 及 clk_i。若 MemRead 為 1，為 load 指令，把 ALU_result 當作 address，從 Data Memory 抓出 data 然後傳給 MEMWB register。若 MemWrite 為 1，為 store 指令，則等到 clk_i 為上升邊時將 DatatoMemory 寫入 ALU_result 表示的 address。

- ## MEMWB register:

輸入有從 EXMEM 輸出的 EXMEM_RegWrite、EXMEM_MemtoReg、EXMEMALU_result、EXMEM_RDaddr 及從 Data Memory 輸出的 Memory_data，加上 clk_i 及 rst_i。
第一個 always，若訊號在 rst_i 的下降邊，將所有對應 output 設成 0，與前面做法相同。
第二個 always，若訊號在 clk_i 的上升邊，將輸入設給對應的輸出，與前面做法相同。

- ## WB state

1. **MemtoReg_MUX:**
   輸入為從 MEMWB 輸出的 MEMWB_ALU_result、MEMWB_Memory_data 及控制信號 MEMWB_MemtoReg。透過 MemtoReg 的值來決定要讓誰通過，若為 1 讓 Memory_data 通過，否則讓 ALU_result 通過。輸出即為 MEMWB_RDdata(真正要寫入 RD 的 data)，傳回 Registers 的 RDdata_i。而 MEMWB 輸出的 MEMWB_RDaddr(真正要被寫得 RDaddr)傳回 registers 的

RDaddr_i。在下一次 clk_i 的上升邊時會寫入。

- **Difficulties encountered**

  在本次作業中最讓我受挫的地方是，一開始不管我怎麼仔細的檢查每個 module，能修改的一些小地方都改了，output 出來 registers 還是一樣每個 cycle 都全 0，debug 這件事真的花了我很多時間。後來有去實驗室找助教求助，他雖然也沒找出問題點在哪，可是他教我如何用 gtkwave 來 debug，我以前沒有太認真研究 gtkwave，不知道它能夠把各個模組的訊號叫出來看。用了 gtkwave 來 debug 之後，很快就找到了問題點，就是我用最直覺的方式把一個 module 的 output port 直接接到別的 module 的 input port，讓我想到我有問過助教 input is coerced to inout 這個 warning 的意思，助教解釋說這是 compiler 無法辨識這條 wire 是輸入還輸出。表面上編得過沒有 warning 但實際上資料都傳不出去，因此我使用 gtkwave 一個 module 一個 module，一個 state 一個 state 檢查，只要有 module 的 output 輸出是 unknown，我就會針對此 module 的 input port 及相接的 output port 做修正，我會直接在 CPU 中宣告一條 wire 給 input port 及 output port 做使用，全部檢查完之後，pipeline 也能夠順利運作了。