Computer Architecture Project 1

Member（按學號）: B04703001 蔡明宏, B04705001陳約廷, B04901003 許傑盛

How to implement this pipelined CPU ?

把每個指令分成五個階段，分別是 `IF / ID / EX / MEM / WB` 。為了讓指令在各個階段正確執行，拿到的資料（ID / MEM），決定好的 `control signal` (ID)， 算出來的值（EX)，必須傳下去。因此在每個階段中間都放一個 `Register` 作為訊號的儲存與傳遞。

Signal	Meaning	產生的 stage	EX 使用	MEM 使用	WB 使用
RegWrite	是否要寫 register 的值	ID			x
MemToReg	是否要從 mem 讀值寫入 register 的值	ID			x
Reg_W	寫入的 register Id	ID			x
Ddata_rdata	從 data memory 讀到的資料	MEM			x
ALUResult	EX 階段 ALU Unit 算出來的值	EX		x	x
MemWrite	是否要寫入 memory	ID		x	
MemRead	是否要讀取 memory	ID		x	
ALUCtrl	選擇 ALU 的運算子	ID	x		
ALUSrc	選擇 ALU 的運算元	ID	x		
Readdata1	從 rs 拿到的資料	ID	x		
Readdata2	從 rt 拿到的資料	ID	x		
Imm_I	I format 的 immediate	ID	x		
Imm_S	S format 的 immediate	ID	x		
Reg_R1	rs	ID	x		
Reg_R2	rt	ID	x		

然而， `Pipeline` 會因為 dependency 產生 `hazard` ，以下針對什麼時候 `stall / flush / forward` 等與 `Hazard` 相關的問題進行回答
註： 我們有實作 ID stage 的 `forwarding` ，因此也要特別處理 branch 的狀況

Hazard unit and Forwarding unit

Forwarding unit (module `FWD_UNIT_ID` and `FWD_UNIT_EX`)

1. `FWD_UNIT_ID`

Output two signals:

- `isFWD_R1_ID` : forwarding for `$rs`
- `isFWD_R2_ID` : forwarding for `$rt`

Assign value `0` , `1` , `2` to the signal.
- `0` : not forwarding, use register value `ReadData_tmp`
- `1` : forward from `ALUresult_MEM` , from MEM stage
- `2` : forward from `WriteData` , from WB stage

The signal if connected to ID stage logic, where the ID stage: `ReadData1` , `ReadData2` , or the EX stage: `ALUin1` , `ALUin2_reg` may be modified by the signal.

```
// isFWD_R2_ID : 0 -> ReadData2_tmp,    (IF)
//               1 -> ALUresult_MEM,    (EX)
//               2 -> WriteData         (WB)
if (RegWrite_WB && (Reg_W_WB == Reg_R2)) isFWD_R2_ID = 2'd2;
else if (Branch && RegWrite_MEM && (Reg_W_MEM == Reg_R2)) begin
    isFWD_R2_ID = 2'd1;
end
else isFWD_R2_ID = 2'd0;
```

2. `FWD_UNIT_EX`

Output two signals:

- `isFWD_R1_EX` : forwarding for `$rs`
- `isFWD_R2_EX` : forwarding for `$rt`

Assign value `0` , `1` , `2` to the signal.

- `0` : not forwarding, use register value `ReadData_EX`
- `1` : forward from `WriteData` , from WB stage
- `2` : forward from `ALUresult_MEM` , from MEM stage

The signal is connected to the MUX before ALU unit, MUX will select which one to be inserted into the ALU unit.

```
if (RegWrite_MEM && (Reg_W_MEM == Reg_R1_EX)) begin
    isFWD_R1_EX = 2'b10;
end
else if (RegWrite_WB && (Reg_W_WB == Reg_R1_EX)) begin
    isFWD_R1_EX = 2'b01;
end
else begin
    isFWD_R1_EX = 2'b00;
end
```

Stalling Unit (module `STALL_UNIT`)

We need to stall when facing:

- load-use hazard
- branch evaluation requires registers not written back yet

We need to differentiate two cases because `Branch` use value in `ID` stage, all the others use value in `EX` stage.

If it is the `load-use case`, though the data has been forwarded, it still need stalling. After stall one cycle the load-use data is now in WB stage, then the forward is applied first forwarding rule correctly.

Output signal: `isSTALL`

`isSTALL` is determined in module `STALL_UNIT`, here is the if-else statement in our verilog. If signal is true, the PC does not change and insert `nop` into the `IF/ID` register.

```
// load-use hazard
if (MemRead_EX && ((Reg_W_EX == Reg_R1) || (Reg_W_EX == Reg_R2))) begin
    isSTALL = 1'b1;
end

// Branch evaluation requiring writeback results
else if (Branch) begin
    if (MemRead_MEM && ((Reg_W_MEM == Reg_R1) || (Reg_W_MEM == Reg_R2))) begin
        isSTALL = 1'b1;
    end
    else if (RegWrite_EX && ((Reg_W_EX == Reg_R1) || (Reg_W_EX == Reg_R2))) begin
        isSTALL = 1'b1;
    end
    else isSTALL = 1'b0;
end
else begin
    isSTALL = 1'b0;
end
```

Global logic:

- `isFlush` = `Branch_true` (evaluate at ID stage)

Other Modules

Module between stages

Module between stages mainly passes the bytes forward to the next stage.

- module `REG_IF_ID` : keep current PC and pad `0` into next stage if `isFlush=1`
- module `REG_ID_EX` : insert `nop` if `isSTALL=1`
- module `REG_EX_MEM` : nothing special
- module `REG_MEM_WB` : nothing special

Basic modules

- module `ALU` : at EX stage, for arithmetic logic
- module `CONTROl` : at ID stage, for instruction parsing
- module `Register_file` : 32 x 32 register

Memory (at `memory.v`)

- module `Instruction_Memory` : instruction memory from input (**debuged: original PC shall add 4, but testbench +1 adds one per instruction**)
- module `Data_Memorys` : data memory

Problems and solution

如何實作整個 pipeline 系統

因為整個系統非常大，我們最後決定分成幾個階段來測試，降低每個階段的複雜度，讓實作更容易一些。各個階段分別為
1. 沒有 hazard 狀況的 pipeline
2. 處理 EX stage 的 forwarding
3. 處理 ID stage 的 forwarding

存取記憶體

不同於 MIPS ， RISC 的記憶體只省一個 bit，因此只要 left shift 一格就好了

不處理 start_i 會無法運作

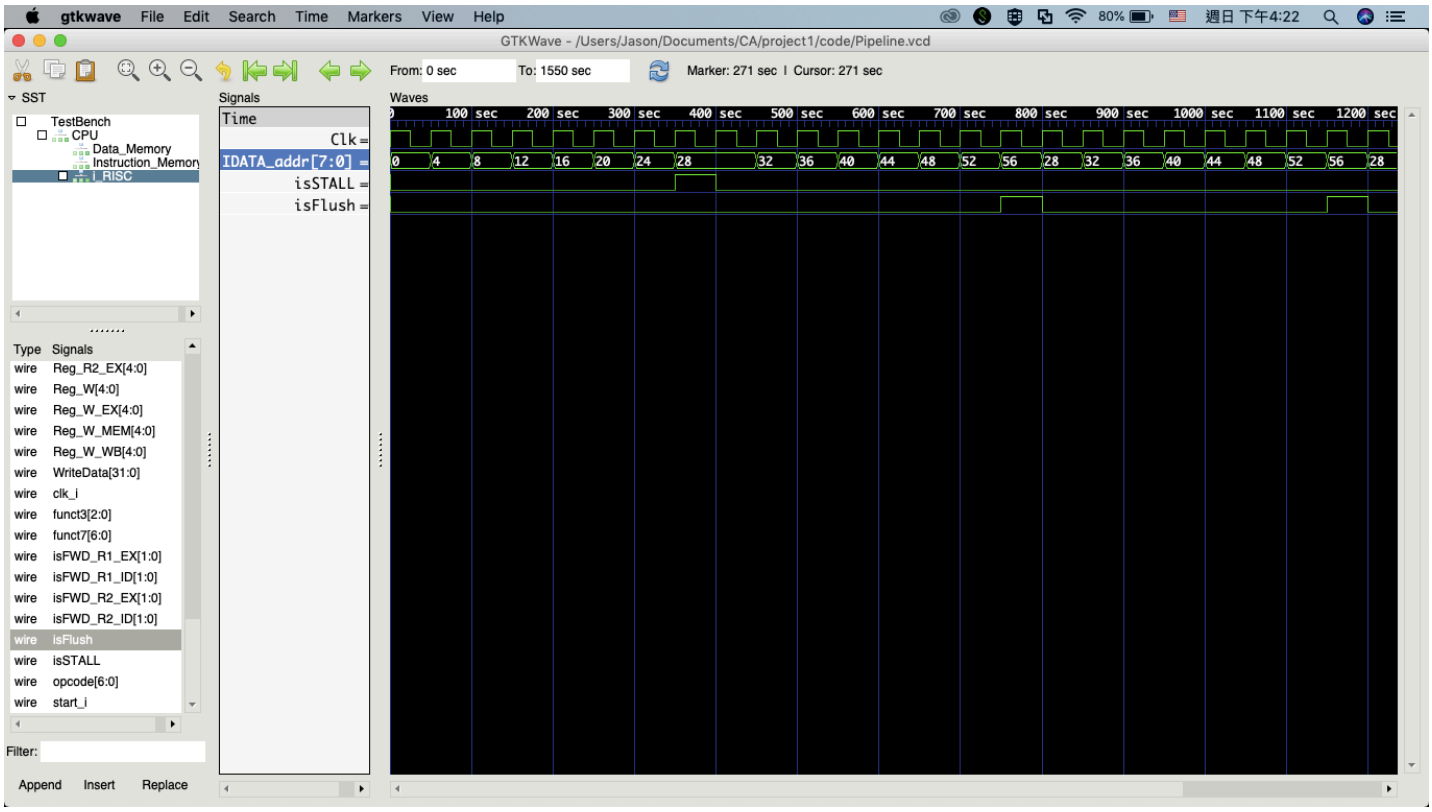
傳入 starti 訊號，在各個 always block 處理 starti 訊號

Sample Output

In our CPU designed, our count stall and flush is slightly different to the sample output. Our CPU is 1 cycle faster than the sample one, but the resulting registers are the same.
#Note that our code was executing on MacOS with `iverilog`

Fibonacci_instructions

```
PC = 20:    lw $s0, 0($a0)
PC = 24:    addi $s4, $s0, 0     # load-use hazard
    ...
PC = 52:    beq $0, $0, loop
PC = 56:    sw $s1, 4($a0)       # may need flush
```



Teamwork

- Pipelining: 蔡明宏、陳約廷
- Hazard Control: 許傑盛
- Report: 大家