Computer Architecture

Project 2 : Pipeline Simulator

Peter-Huang

A. Simulator Design

1. Data Structure

大部份保留並延伸 single cycle simulator 的設計方式。並新增了，相關的 structure for pipeline simulator. 例如：control unit for getting PC value, forwarding Unit for recording forwarding information. PipelineBuffer for recording result for each stage.

```
typedef struct PipelineBufferStructure{
    Instruction instruction;   /// 相關的 I-instruction decoded value.
    int result;                ///每 stage 運算完的結果
} PipelineBuffer;

typedef struct ControlUnitStructure{
    bool isFlush;              /// 是否要 flush 下一執行指令
    bool isStall;              /// 是否要 stall 目前的指令
    int updatePC;      /// 是否要 jump updated Program counter
}   ControlUnit;

typedef struct ForwardingUnitStructure{
    bool isForwardRsToID;
                               ///是否在 ID stage, 需要 forwarding rs data
    bool isForwardRtToID;
                               ///是否在 ID stage, 需要 forwarding rt data
    char *fromStage_ForwardRsToID;
                               /// rs 從哪個 stage forwards to ID stage
    char *fromStage_ForwardRtToID;
                               // rt 從哪個 stage forwards to ID stage
    bool isForwardRsToEXE;
    bool isForwardRtToEXE;
                           //是否在 EXE stage, 需要 forwarding rs, rt data
    char *fromStage_ForwardRsToEXE;
```

```
    char *fromStage_ForwardRtToEXE;
                        // rt,rs 從哪個 stage forwards to EXE stage
} ForwardingUnit;
```

PipelineBufferStructure 是主要存 mips instruction 的 buffer. 這功能主要是實作 pipeline 5 stage. 讓 5 個指令能同時不同的 pipeline stage 裡裡執行。 Pipeline buffer 總共有 IF_ID_Buffer, ID_EXE_Buffer , EXE_DM_Buffer , DM_WB_Buffer 前四個，實作是參考課本的設計，但是，需要 print out information of WB Stage，所以所以額外增加了 WB_Buffer, 所以共 5 個 buffer. 裡面留相關的 instruction decoded 的內容，與每一 stage 運算完的結果。(當初設定，在每一 stage 裡，會依造不同的相關運算結果，存到不一樣的變數，例如：算 memory 存在 address_result 裡；算 calculation 存在 ALU result 裡；算 pc 的存在，pc result 裡；但後來，通通移除，只存在 result 裡。是因爲，在 debugging 時，我有 5stage. 而每個 stage 又是去上一 stage 抓值，所以會變的 stage 間相依性很高，用多變數，反而會造成 coding 時 及 debugging 時的複雜度大幅提高 且不好 maintain ，這是因爲當 instruction 不一樣時，我要 reference 不一樣的變數。)

ControlUnitStructure 主要是來控制是否 flush , stall 及抓到相關的更改的 pc value.因爲當這 2 個動作發生時，我就不會依造正常的 pc 值去抓相關的指令，所以透過這，去記錄是否每一 cycle 有發生 flush or stall, 再去抓相關之前記錄的 pc value.

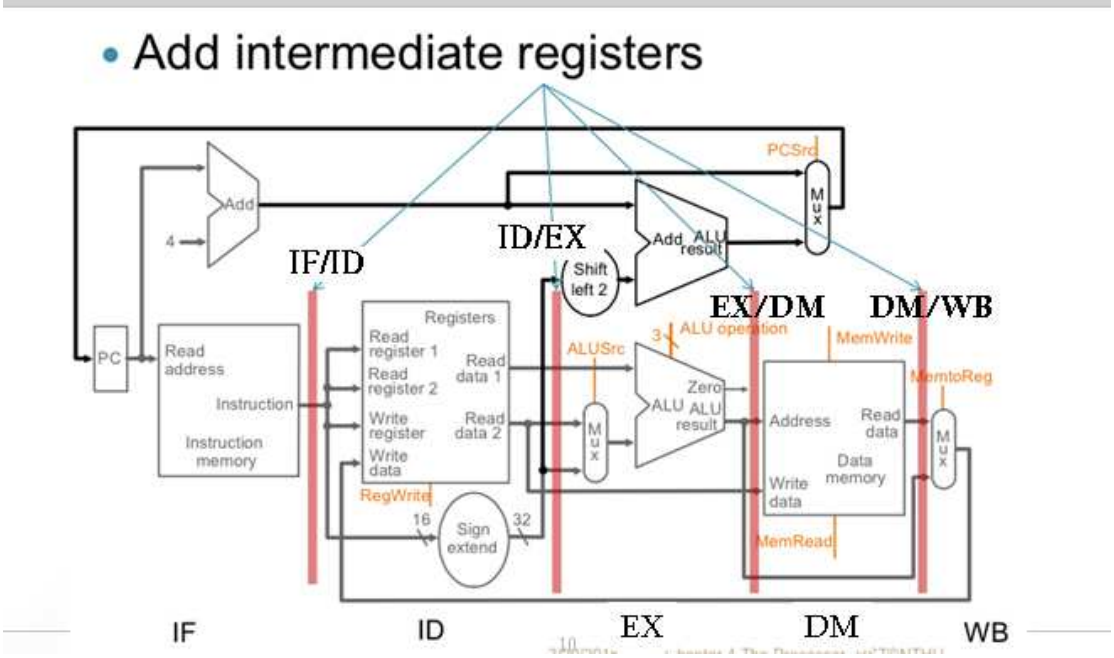
ForwardingUnitStructure 主要是記錄當有 data hazard 發生時，是發生在哪個 stage :ID or EXE，而爲何是是有這 2 個 stage 呢？因爲這 2 stage 做的事情，ID:decoding 及 EXE:calculation 他的來源: rs or rt 會是後 2 個 stage 的運算結果。所以只有在這 2 stage，我們需要 detect data hazard. (因 data dependency 是發生在之前 2 筆指令內，才會有 data hazard!!) 而當有 data hazard 時，我們要記錄是從哪個 stage 他的目的 register 與來源 register 相同。爲了在 output 上顯示。

2、Execution Flow (如助教投片中)

Main flow (1) WB (2)DM (3) EXE (4) ID (5)IF

上面程式的執行順序，就確保了，實際 pipeline 的運算結果（如下圖），因爲 WB 先作，就等相是模擬 pipeline 實際在 rising、falling edge 的，先 write 後讀的動作. 所以在程式裡，倒著執行，就會得到正確的結果。

Pipeline Stage



Detail Flow:

(1) Load binary data form D & I Image.bin

(2) fetch instruction from binary data and save in to dMemory or IMemory int[] array

(3) **WB(); DM(); EXE(); ID(); IF();**

(4) output Snapshot

(5) judge if 5 stage are filled with NOP instruction

(6) judge if flush or stall then update PC value.

5 Stage Detail flow:

在 **WB()** 先抓上一 STAGE 的 BUFFER (DM_WB_Buffer)，之後 WB()做完再存在 WB_Buffer

在 **DM()** 先抓上一 STAGE 的 BUFFER (EXE_DM_Buffer)，之後 DM()做完再存在 DM_WB_Buffer

在 **EXE()** 先抓上一 STAGE 的 BUFFER (ID_EXE_Buffer)，之後 EXE()做完再存在 EXE_DM_Buffer

在 **ID()** 先抓上一 STAGE 的 BUFFER (IF_ID_Buffer)，之後 ID()做完再存在 ID_EXE_Buffer

在 **IF()**　用 PC 值，抓 instruction 之後，IF()做完再存在 IF_ID_Buffer

實作的準備：　劃分每一 stage ，每一 instruction 在這 stage 所做的 task.

WB stage：寫回 register
R– 只有 JR 不會寫回 register
I – 只有 SW, SH, SB, BEQ, BNE 不會寫回 register
J – 只有 JAL 會寫回 register[31]

DM Stage：讀寫　memory data
I – 只有 LW, LH, LHU, LB, LBU, SW, SH, SB 會對 memory access

EXE Stage: 做 ALU 運算
R –只有 JR 不會在 EXE stage 運算
I –只有 BEQ 、BNE　不會在 EXE stage 運算
J– 都不在 EXE stage 運算

ID Stage：做 decoding : 每一指令都要做 decoding
處理 jump 類 及 beq 判斷類指令

IF Stage: 做 instruction fetching
每一指令都要做 instruction fetching action.

例外處理：由上面的 STAGE 的處理，就可以清楚知道，各 exception 會發生在何處。

WB stage：write to zero error

DM Stage：D memory address overflow. 造成 halt
D memory misalignment 造成 halt

EXE Stage: number overflow when doing calculation action.

ID Stage：check instruction decoding not match (造成程式 shut down)
IF Stage:　check I memory address 有無 misalignment (造成程式 shut down)

Data Hazard detection ：這部份應是整個 **project** 核心。作法：先與同學討論，明確列出來各自每一指令可能會發生的 **data hazard**。之後，再依此進行實作。

In EXE Stage:

1. 當指令是 **R type:**

   a. 當後２**Stage** 指令是 **Rtype:**

   檢查, **EXE** 後２**stage** 的ｒｄ相同　與 **exe stage rs**、**rt** 相同！

```
EX: sub $6, $1, $3        IF   ID   EX\   ME\   WB                a.
    add $2, $2, $5             IF   ID \  EX \  ME   WB           b.
    sub $3, $6, $2                  IF    ID   \   EX   ME   WB    c.
```

   b. 當後２**Stage** 指令是 **I type:**

   檢查，**EXE** 後２**stage** 的 **rt** 與 **exe stage rs**、**rt** 相同！

   Ex:

   Addi $rt. $rs. C

   Add rd, rs, rt

且當指令在 **EXE** 後２**stage** 是 **SW, SH, SB,**時，不用判斷 **data hazard,**因爲不會對 **rt** 做寫入的動作。而 **BEQ, BNE** 也不用判斷，是因爲 **BEQ, BNE** 在 **ID stage** 就處理完了。　（而這裡面的特例，是當有 **Load-use data hazard** 時，我們會在 **ID stage stall 1cycle**，所以，這會變成，多了１**NOP** 指令，所以其實只有檢查後２**stage** 就好，不用檢查後１**stage** ）

ｃ．當後２**stage** 有 **JAL** 指令時，因爲他會寫入 **register[31]** ,要判斷目前 **exe stage** 的來源是否有 **$31**，如有要 **forwarding**

2. 當指令是 **I type:**

   a. 當後２**Stage** 指令是 **Rtype:**

   當目前指令的 **rs** 與後 **2 stage rd** 相同，做 **forwarding.**

   當目前指令是 **sw, sh , sb** 且他的 **rt** 與後 **2 stage rd** 相同，做 **forwarding.**

特例：

I type 裡的 rt，如果有 data hazard 要偵測，並記錄 forwarding.
然而在 sw, sb, sh 在 exe stage, 我並不會真的計算他。 而是等到 DM stage 去 memory load rt 的值。且 I type 裡，RT 的值，會被正確讀到，但是必需 forwarding 相關 information, 即使不會運算 。

ex: add $s1, $s2, $s3 IF ID EXE MEM [WB]前半部寫入 register
sw $s1, 0($t0) IF ID EXE [MEM] WB 後半部才讀入 register

b. 當後 2 Stage 指令是 I type:
當後 2 Stage 指令不是 SW, SB, SH, BEQ, BNE
而且當目前指令的 rs 與後 2 stage rd 相同，做 forwarding.
當目前指令是 sw, sh , sb 且他的 rt 與後 2 stage rd 相同，做 forwarding.

c. 當後 2 stage 有 JAL 指令時，因爲他會寫入 register[31] ,要判斷目前 exe stage 的來源 rt 是否是 $31，且不能是 sw、sh、sb，因爲沒有做寫入動作。如有要 forwarding

ID Stage 的判斷方式，如同，EXE stage .也就是 summarize 出每對應指令的來源與後 2 STAGE 的目的是否相同，如有就 forwarding，其細節可以參考 code 就不列出了。

處理 double hazard：

in EXE & ID Stage： 下面其實 c.指令的來源 RS ，其實是要吃 b 指令的 RD 而不是 a 指令 Rd. 在程式的實作方式就是，我先 detect 後 2 stage 的 data hazard (a 指令), 之後，再 detect 後 1 stage 的 data hazard(b 指令). 這樣的 detect 方式，如果當有 double hazard 發生時，我就可以抓到最近的指令(b 指令會蓋過 a 指令的 forwarding information)的 value.

double data hazard
add $1, $1, $2 a.
add $1, $1, $3 b.
add $1, $1, $4 c.

Load-Use data hazard & stall happen

1. 發生情形，當前面 1 個指令有 lw, lh, lb 指令時，如果來源　與　Lw 的目的相同，必需透過 stall ，讓目前的指令在 ID stage 可以正確抓到 register value.
2. 發生情形，當後 2 stage 指令是 JR, 因爲他會 PC 跳到 register's value. 所以當他後 2 stage 的目的有與 JR rs register number 相同時，要做 stall, 等 JR 在 ID stage 可以正確抓到 register value。
3. 如果是 beq 及 bne 時，他會在 ID stage 比對，所以當後 2 指令的目的 register number == beq & bne rs, rt number 時，也要 stall 讓 beq& bne 的指令在 ID stage 可以正確抓到 register value.

Flush action happen

1. **Conditional instruction: beq & bne 因爲在 ID stage 才知道要不要 jump. 所以如果要 JUMP, 必需把下一指令 flush 掉，因爲他已經在 IF stage 了。**
2. **Unconditional Instruction: JR & Jtype instruction, 這類的 instruction 都會在 ID Stage 執行 jump，但，他下一指令，已經在 IF stage 了，所以要把他 flush 掉。**

B. Simulator Elaboration

原則就是，**1、make common simple　2、抽象化 each function，讓功能能簡單，專注在自己的功能要用的事上**。 Follow 這原則，設計 pipeline 的相關功能：

例如：

1、透過統一的 Data hazard detection for specific stage, 決定，有否 forwarding action. 而如果有 forwarding action 沒 detect 到，我也只需改那功能就可以。

2、統一的　stall 的偵測，如果任何的 stall condition , summarize 到同一 function ，有錯的話，我也只要改那一部份，好維護。

3、透過統一的　structure，知道記錄相關的 information, 如：如果要知道 forwarding information, 就去 forwarding unit 裡查，就可以了，如果，要知道是否程式會改變 program counter, 只要去，control unit 查即可。

C. Test Case Design

1、 How to implement your test case in C code or assembly codes.

我是直接寫組語，然後，透過，同學享的 assembler 去轉成 binary code.

2、 What corner case are you targeting?

以下的組語只列出片段，其它的因爲太多了，只好沒列上來，只把同學可能會忽略的列出。 (且其中組語裡也有包含，project 1 的 error 的情形，因爲，找出粗心的同學，沒去修正 project1 錯誤的 case，但沒放上來.)

Just focus on extremely case.

A. Check unused register number has been set to be 0

srll ,srl,sra -- rs；jr -- rt, rd ；lui -- rs，我在 binary code 裡設定這些 un-used to be 1

nori $1 , $0 , 33

lui $4 , -4096 (實際不會 forwarding $1，但如果，沒 set to be zero, 會 forwarding)

B. check JAL

因爲 jal 有寫回 registers[31] ，所以找出粗心的同學忘了 check 是否有 data hazard 與 flush 與 stall

```
sw     $31, 0($20)
jal    0x20          ## pc=0x80
addi   $21, $31, 0
```

C. check JR

因爲 jr 是 r type，但是又執行 JUMP 的指令，所以，測試同學有無偵測 data hazard 與 flush 與 stall

```
addi $5 , $0 ,0x20     ##pc= 0x1c
jr     $5
sra    $9, $5, 1       (如果 unused 沒設 0，也會錯誤的做 forward 如 A)
```

D. check beq & bne

這部份只是測試，是否相關的 beq & bne 指令會測 data hazard 與 flush 與 stall，或偵測錯誤的 forwarding register number

```
        lw      $1 , 0($10)
        beq     $0 , $1, beq_3
beq_3:  beq     $1 , $7, beq_4
beq_4:  beq     $0 , $1, beq_5
beq_5:  sw      $1 , 0($10)
        beq     $0 , $1, beq_6
beq_6:  sll     $7 , $3, 3        # rs=1
        beq     $0 , $1, beq_7
```

E.　最後用枚舉法，盡可能混合個種的 R & I type

透過各種不同的組合，去偵測相關的 data hazard / stall/ flush，也爲自己做最後的 debugging 且找出粗心的同學。且透過 來源 與 目的，限制在 1~3 register number 提高 data hazard happen.

```
addi    $2 , $1 , 9
lh      $2 , 0($1)
addi    $2 , $1 , 9
lhu     $2 , 0($1)
sw      $2 , 0($1)
sh      $2 , 0($1)
slti    $3 , $2 , 2
lb      $2 , 16($1)
lw      $2 , 0($0)
lui     $1 , -4096
addi    $2 , $1, 0
bne     $2 , $1 , Loc_2
slti    $11, $8 , 3
addi    $2, $2, 1
lbu     $2, 16($0)
sb      $2, 4($2)
ori     $2, $1, 1
beq     $2, $1 , Loc_2
add     $3 , $0 , $0
nori    $2, $1, 1
addi    $3, $2 , 2
beq     $3, $2 , Loc_2
addi    $2, $0 ,0
sw      $2, 4($2)
```

……(其它相關的組語，是我覺的，同學可以想到的，就沒列出)