Report for Architecture project 2

archi01, member:101021120 鄭宇翔, x1022108 曹士杰

Simulator Design

1. Data structure

我使用 C 語言實作本次 project，除了第一次 project 所使用的 data structure 外，我還使用了以下 struct:

```
typedef struct INSTRUCTION{
    int IR;
    int type; // 1: R, 2: I, 3: J
    char *name;
    int opcode;
    int rd;
    int rs;
    int rt;
    int shamt;
    int funct;
    int imm;
    int address;
}INSTR;

static struct BUFFER_BETWEEN_PIPE{
    INSTR instr;
    int ALUout;
} IFIDin, IFIDout, IDEXin, IDEXout, EXDMin, EXDMout, DMWBin, DMWBout, WB;
```

作為各管道間的 buffer，buffer 裡有 INSTR instr 來作為指令有關的所有資訊，包括 32 bit IR 原碼、指令類型、指令名稱、opcode、rd、rs、rt、shamt、funct、imm、address。注意不論何種 type 不論何種管道我的 struct 都一樣，只是會因 parsing 不同存到不同部分，但不會衝突。

buffer 裡除了 INSTR instr 外，我還有 int ALUout 來儲存 EXE 算出的值以及 DM 取出的 memory 值。注意我沒有多使用 MEMout 而是使用同樣的 ALUout，只是在 DM/WB 階段後 ALUout 當 MEMout 使用，因為有了 MEMout

可以不再用到 ALUout，因此存在同一個 ALUout 裡沒差。

注意我每個 buffer 都有分兩層(in 與 out)，每個 stage 執行結果些存在 in，在最後 synchronize 時把所有 in 送到 out，因此不會因為做了一個 stage 就洗掉了 buffer 內容(好維護)。也就是說，我用了 in 與 out 兩端來實現軟體較難實現的 synchronization 的問題，因為硬體來說 5 個 stage 是同時進行的，但軟體必會先執行完一個函式再執行下一個，這樣的方式可以說是一種 pseudo synchronization，是一個很棒的架構！

接著，我使用以下 struct(舉 I-type 為例):

```
static char *opcode_I[I_num] = {
    "ADDI", "LW", "LH", "LHU", "LB", "LBU", "SW", "SH",
    "SB", "LUI", "ANDI", "ORI", "NORI", "SLTI", "BEQ", "BNE"
};
static int mappingTable_I[I_num] = {
        0x08, 0x23, 0x21, 0x25, 0x20, 0x24, 0x2B, 0x29,
        0x28, 0x0F, 0x0C, 0x0D, 0x0E, 0x0A, 0x04, 0x05
};
```

作為名字與 opcode 之對應，使 ID stage 時可以有了 opcode 直接對應到名稱。

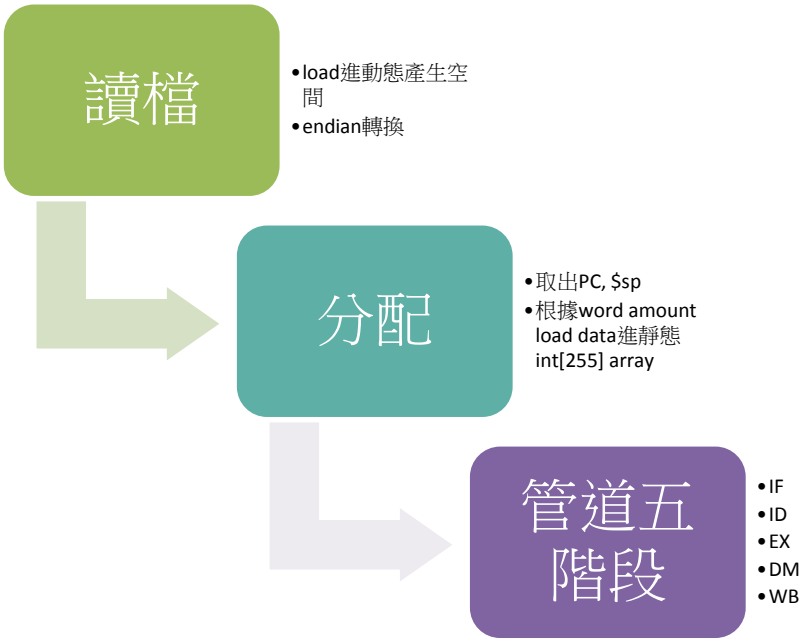
最後，我使用以下 struct:

```
static struct{
    char *where;
    int flag;
}fwd_EX_rs, fwd_EX_rt;
```

來作為指示 forwarding to EX 的 flag 與從哪裡 forwarding 來的。

2. Program structure

以下為程式整體流程圖：



以上是第二次 project 之流程圖。

與第一次 project 不一樣的是原本在上圖中的「解碼」位置改成了「管道五階段」，並分成 5 個 stages – IF | ID | EX | DM | WB。

IF 是 instruction fetch，也就是從當下 PC 指的位置拿指令，因此在此 stage 可能發生 I-memory access error，也就是 Miss align or Address overflow，但本次 project 不允許發生 I-memory access error。

ID 是 instruction decode，也就是將上次 IF 拿到的指令做解碼，分成各種 type 並 parse 出細項。本次 project 與 jump 有關的指令(jr , beq, bne, j, jal)所要跳的 address 都必須在此 stage，因此可能會有 stall(IF 與 ID)與 forwarding(EX/DM to ID)的問題，以及當指令成立(taken)時，IF 必須 flush 掉。

EXE 是 execute，所有指令需要做計算的都必須在此 stage 算出來(還沒有要存進 register)，因此可能會有 Number overflow 的問題。也可能會有 forwarding(EX/DM or DM/WB to EX)的問題。
DM 是 Data memory access，與 load or store 有關的指令在此執行，因此可能會發生 D-memory access error，也就是 Miss align or Address

overflow。

WB 是 Write back，所有需要寫回 register 的指令會在此執行，因此 stage 可能發生 Write to $0 之問題。

結論：

occur / stage	stall, forwarding, flush	error
IF	stall xor flush	I-memory Misalign or Address overflow
ID	stall xor fwd (from EX/DM)	
EX	fwd (from EX/DM or DM/WB)	Number overflow
DM		D-memory Misalign or Address overflow
WB		Write to $0

main():

在 while(1)之前的結構與 project1 相同。

在 while(1)內一開始是 initialize()函式，功用是初始化變數，包括 stall, forwarding, flush 之 flag，以及 project1 裡的變數。

接著判斷 PC 是否有問題，有則印 error(Next cycle)，然後 halt。

接著我們的管道實作方式是逆著正常的 stage order，從 WB, DM, EX, ID, 寫到 IF，這樣子在 WB 階段寫入 register 則 ID 階段可順利取出正確值，也就是達成了要求「在 cycle 前半部寫入，後半部取出。」的實現。注意我們的管道五階段的五個函式皆會執行 buffer 中 out 端的指令(recall 在 data structure 部分我使用的管道間 buffer 分 in 端與 out 端)，因此會判斷 stall, forwarding, flush 以及可能發生的 error 並且執行完指令，並接執行完結果存於管道間 buffer 中的 in 端。

接著是 PrintCycle()函式，將 cycle 狀態印出來，注意 format 中的 R0~R31 是指上一個 cycle 管道執行完的結果，並且 IF ~ WB 管道顯示的是下一次將執行的指令。

接著是 Synchronize()函式，理論上將所有 buffer 的 in 端傳入 out 端，

如果需要 stall(flag == 1)，則 IF_ID 與 ID_EX 不動；如果需要 flush(flag == 1)，則 IF_IDin 清空當作未抓到指令。

接著移動 PC，到 jump or branch 指令所指定的位置，若沒有則 PC += 4。接著 CycleCount++迴圈結束。

注意因為 snapshot 的管道所指的是下一次將執行的指令，但實際上為了顯示是否有 stall, forwarding, flush，因此在這個 cycle 我已經順便將 error 測試完並執行完了。所以印 error 時的 cycle 必須加 1。且印的 R0~R31 是指上一個 cycle 管道執行完的結果，因此我是印 R_buf[32]等到印完再在 Synchronize()函式裡 R_buf[32]通通 get R[32]。

InstrFetch():

在這個函式裡，就是一條式子，從 PC 指到的位置拿指令。

InstrDecode():

在這個函式裡，我會先判斷 buffer 中的指令是哪個 type，並且依該 type 分析出各部位，注意有些指令內含 don't care 的部位必須指定成 0，否則將影響 forwarding(以下通稱 fwd)判斷。

接著在 R-type 與 I-type 內判斷 stall/fwd/flush。

當 R-type 中的 rs 或 rt 跟 load 指令的 rt 一樣並且不是 0 則需要 stall(以下只說「可能需」，代表只要符合這句類似情形則會發生後面所指事項)。

R-type 中的 JR 指令需多判斷若前一個是 R-type 或 I-type 非 store 或 branch 則可能需 stall，若前兩個是 load 指令也有可能需 stall，否則若是 R-type 或 I-type 非 store 或 branch 或 J-type 之 JAL 並且用到$31 則可能需 fwd，並且若無 stall, IF 必須 flush 掉。

I-type 較複雜，其中的 BEQ 與 BNE 更有兩個 register 需判斷，但大抵上方法跟 R-type 中的 JR 指令類似，不多詳述。

J-type 無需判斷 fwd 或 stall，原因是它不看 register 值而是直接跳躍。但 JAL 之$31 要等到 WB 再存入，並且這個$31 可能被別人 fwd 要注意。

注意在這個階段的任何指令前一個一定不會是 JR, J 或 JAL 指令，因為跳躍後管道中會插入 NOP(因為 IF 之 flush)，因此若現在 EXE 將執行 JR, J 或 JAL 指令，則這個 ID 必將執行 NOP。

Execute():

在這個階段會先判斷 buffer 內若是 branch 或 junp 或 NOP 則不執行。

接著需依各種 type 判斷 fwd，很複雜，但重要的是，必須先判斷從 DM-WB 來的 fwd 再判斷 EX-DM 來的 fwd，不然判斷 EX-DM 的 fwd 可能會被 DM-WB 的 fwd 蓋過去，也就是說當可能需同時從前兩個與前一個指令 fwd，會拿到前兩個指令 fwd 來的舊值，這是錯的，因此順序必須像紅字一樣。

I-type 判斷 fwd 時最容易被忘記的情形是：

```
DM: add $1, $2, $3
EX: sw $1, 0($0)
```

此時的$1 需要 fwd to EX。

判斷完 fwd 接著執行的指令就跟 project1 一樣，只是結果不是存到 register 而是 EXDMin.ALUout 裡，並且要判斷是否有 Number overflow。而 store 與 load 指令所要跳的地址需在此 stage 算出來，我會先算 base + offset 之結果，到 DM 階段再處理 misalign 與 address overflow 之判斷。

DataMemoryAccess():

在這個階段會先判斷 buffer 內若不是 store 或 load 則不執行。

接著執行的指令就跟 project1 一樣，只是結果不是存到 register 而是 DMWBin.ALUout 裡，並且要判斷是否有 misalign 與 address overflow。

WriteBack():

在這個階段 JR, branch, store, NOP 指令會被排除(不執行)，接著判斷是否有 Write to $0，沒有的話則目標暫存器拿到 buffer 中 ALUout 之值。也就是 R[WB.instr.rt] = WB.ALUout。

PrintCycle():

先印 R_buf[0]~ R_buf[31]值，再印 PC，再印各管道狀態，可能有的 stall/fwd/flush 可參照第 4 頁 table，要注意的是同時需 fwd rs 與 fwd rt 的狀況，必須先印 rs 再印 rt，因此判斷 flag 時需考慮此狀況。

ErrorMsg():

與 project1 相同，唯一不同的是印的 Cycle 數需+1，原因請參照第 5 頁第 3 段。

==========

※注意事項：遇到需判斷 fwd 的情況只能用 if 不能用 else if，否則當 if 成立，只執行並 fwd 其中一個 register，而另一個 else if 就不判斷了。

Conclusion:

整個 code 扣掉註解約 900 行，架構挺鮮明、易懂，雖然 fwd 的部分稍複雜些，必須細心 debug，但整體而言，程式仍挺好維護的。

Data structure 部分用的 struct 頗簡潔的，每個管道間 buffer 皆相同，並且用了 in 與 out 兩端來實現軟體較難實現的 synchronization 的問題，雖然可能會覺得 struct 很多很複雜，但這種方式較易懂並且較易維護。

Program structure 部分也是架構相當鮮明，很容易了解程式碼的作用。

Testcase 也是比 project1 更加 tricky(下一部分將會詳細介紹)，包括此份 report 也是很完善，整體來看算是相當工整且完整的作品，甚至可以堪稱一個不錯的軟體。

Testcase design

Preface

我們的 testcase 是在 project1 的基礎之上添加的。當然我們去除了 project1–testcase 中會導致 halt 的指令以使 pipeline 能夠正常執行我們為 project2 所特別設計的 testcase。其目的在於觀察在 single cycle 中和 pipeline 中執行結果的差異，能夠更深層次的瞭解 pipeline。同時測試其他組是否對在 single cycle 中出現的錯誤進行修改。

Testcase for project2

1、lw 後 bne 的 stall 檢測（無 flush），指令如下：

```
Back3:
    lw $4, 0($0)
    lw $5, 0($0)
    bne $4, $5, Back3
```

bne 指令會是 snapshot stall 兩次。

2、lw 後 add 的 stall 檢測（仍然需要 forwarding），指令如下：

```
lw $1, 0($0)      # $1 = 0x80000000      data at 0
add $2, $0, $1    # stall
sub $3, $2, $1
```

add 指令的$1 導致 stall 一次。同時，即使 stall 過後依然需要 forwarding。當然為了檢測錯誤，我們使用 project1 中 number overflow 錯誤已檢測錯誤在 error-dump 中的 cycle 數(同時$2 需要 forwarding)。

3、forwarding 檢測，指令如下：

```
lw $1, 0($0)      # $1 = 0x80000000      data at 0
add $2, $0, $0    # no stall
sub $3, $2, $1
```

在不會 stall 的狀況之下，我們的 testcase 使 rs、rt 均需要

forwarding，其目的在於測試同學們 forwarding 是否足夠完整安全。

4、beq 指令的 flush 測試，指令如下：

```
        beq $0, $0, Target1 # Target1 = 1
        sub $0, $0, $1
    Target1:
        sub $0, $1, $1
        beq $0, $0, Target2 # Target2 = 0
    Target2:
        sub $0, $1,$1
```

我們分別測試了 beq 的指令 jump 到的位置不是下一個 PC 所要 fetch 的指令和 jump 到的位置就是下一個 PC 就要 fetch 的指令，其目的在於檢測其他組對於 beq 指令尤其是 flush 部分處理是否完整準確。

5、j/jal 的 flush 檢測，指令如下：

```
        lw $0, 1($sp)
        lw $1, 0($0)      # $1 = 0x80000000     data at 0
        sub $0, $1, $1
        j next1           # jump to next line
    next1:
        sll $0, $0, 0     # nop
        j next2
    next2:
        jal next3         # $31=pc+4
    next3:
        addi $3, $31, 12
```

此部分應該不是非常困難，主要是檢測其他組是否對於 j 指令和 jal 指令的 flush 處理正確。同時我們也測試了 jump 到的位置本身就是下一個 PC 要 fetch 的指令。

6、jr 指令的 flush/stall/forwarding 測試。指令如下：

```
    addi $3, $31, 12      # prepare for jr
     lw $2, 0($0)
```

```
    jr $3                # jump to next line, rt=$2, rd=$2
    add $0, $2, $2       # pc164

    addi $3, $31, 32     # prepare for jr
    jr $3                # jump to addi
    sub $0, $1, $1       # write to $0 & number overflow
    sub $0, $1, $1       # write to $0 & number overflow
```

這部分應該是此份 textcase 中最難的部分，首先 jr 是一種特殊的 jump 指令，但是他不是 j-type 而是 r-type，這一點需要大家首先注意。同時他會造成後邊指令的 flush(與 j-type 一致)。同時 jr 指令的 rd 會有 forwarding 的情況。我們的 testcase 中均有涉及到。但是大家很有可能忘記 jr 指令也有可能導致 pipeline stall。我們的 testcase 涉及中會是 pipeline stall 一次或者兩次，同時 stall 之後也需要 forwarding。

7、結束指令：

```
    lw $3, 8($0)
    jr $3
    add $0, $0, $0
    lh $0, 1022($sp)
    sub $0, $1, $1
    add $0, $0, $0
    halt
    halt
    halt
    halt
    Halt
```

結束指令較為簡單，與助教之前提供的 testcase 有重複，並且最後一個 cycle 出現 4 個 error(Write $0 error, Address overflow, Misalignment error, Number overflow)。

最後，我們的 testcase 是用我寫的 assembler 做的，這個 assembler 還附加了 RDC(random don't care)功能，也就是將 don't care 欄位填入隨機值，使得某些同學的 simulator ID 時若未讓 don't care 欄位 fit 0，可能會發生不必要的 stall/fwd/flush，相當 tricky！