Computer Architecture

Project 1 : Single Cycle Simulator

Peter Huang

A. Simulator Design

1. Data Structure

```
typedef enum { R,I,J,S } InstructionType;

typedef struct InstructionStructure {
        InstructionType instrType;
        int binaryCode;   // 存原來的原始的 binary code
        int op;           // 代表 4 種指令都會用
        int r_i_rs;       //代表這 field 是給 R, I type 指令用
        int r_i_rt;
        int r_rd;         //代表 field 是給 R 指令用
        int r_shamt;
        int r_funct;
        int i_immediate;     //代表 field 是給 I 指令用
        int j_s_addr;        //代表 field 是給 J, S 指令用

} Instruction;
```

因爲 mpis 指令，都是 32bit，且分成 4 種 type，所以我設定了一資料結構 InstructionStructure 去存他，這樣的好處是把指令的 type 都抽象化，且對各個指令的 field: 如 R 的 rd ，去對應欄位抓就好，容易理解。此外，這樣處是，對不同種類的的行爲及動作，實作時，可以單單看自己需要的欄位來處理，而乎略其它的。如：J type 時，就只要讀取 op 及 j_s_addr, 可以不管其它 fields. 而保留 binary code 的原因，是因爲，如果，有人需要重新改指令抓取指令不同位置的長度，有額外特別需求，我還是可以透過 binary code 去抓出他要的值。當然目前程式沒有，只是考慮，讓他設計的彈性大點。

```
typedef struct InstructionSpec{
        InstructionType instrType;
        int op;
        char *instrName;
        int r_funct;
} InstrSpec;
```

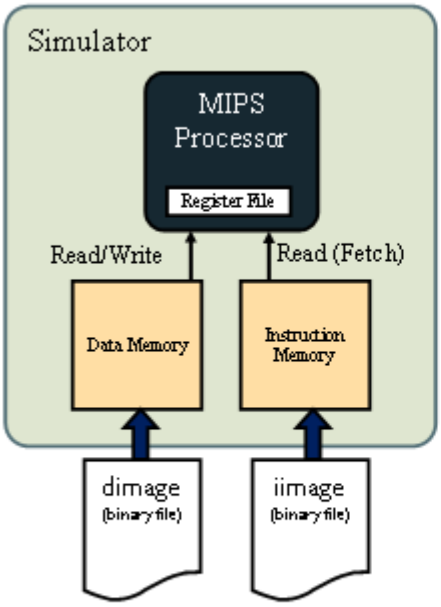
而對不同的指令，在 appendix 上有列出 specification ，後來決定，把這些指令的 spec 都定義在 macro 裡，如：

#define R_OPCODE (0x00) , #define R_ADD_FUNCT (0x20)

這樣的好處是我不用 **hard code 散落在程式裡**，且如果 spec 更動，我只要更動對應的 SPEC code ，不用動到或檢查任何程式，擔心被 hard code 在程式中。然後，再把這些不同指令的 spec，用 InstructionSpec 去區分他。上述的好處，就是**抽象化**，讓以後用的人或維護的人可以切割與底層細節，且這好處是在開發時，如果規模很大時，不同指令，我可以交給不同人，分別去實作不同的指令行爲，因爲我的 spec 都定義出來了。

2、Execution Flow (如助教投片中)

Main flow (1) Fetch Instruction (2) Decoding (3) Execution



- **Simulation steps:**
  (1) Fetch Instruction →
  (2) Decoding →
  (3) Execution(translation/emulation)
- **Fetch Instruction:**
- **Decoding**
  - R-type
  - I-type
  - J-type

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) output Snapshot before instruction execution

(4) Decoding binary code to InstructionStructure

(5) ProgramCounter = ProgramCounter +4

(6) execution

(7) If any error detected, error handler will take care of the error. In our project is to output message or halt instructions

實作想法，因爲 mips 指令都是 32 bit 長度，剛好 match int 長度，這樣，用 int 來存，剛好就可以當成一筆指令或 data 來存取，我就可以抓到一完整的指令或資料。實作困難，發覺 cpu 存取 int 的方式剛好是 little endian 的方式，所以必需轉成 big endian 的方式，讓資料在 int 裡可以呈現 正確的 16 進位的排列順序。實作的便利性，在 decoding 時，發現，幾乎很多不同 type 的指令，確共用相的 bit 數，如: R & I 的 Rt code 佔 5bit 且都是 16-20，所以決定，用一個 marco funciton 去定義抓到的指令 code，這樣的好處，也是方便統一維護。如：

#define GET_R_I_RtCode(x) ((x >> 16) & 0x1f)

實作的 **trick:**

1、 在**(Srl) Shit right logic 0xFF020304 >> 8 => 0x00FF0203** 因爲右移 **c** 會，自動又 **singed bit** 去判斷，最後，解決方式，就是把目前的 **bit** 轉成 **unsigned int first,** 然後去轉，就可以的到想要的結果。

2、 **overflowed's detection:** 如果是加法，可以透過 正+正 = 負 **or** 負+負 = 正；如果是減法：正 – 負= 負 **or** 負- 正=正 去 **detect overflowed error.**

特例情形是，當被減數是 **0x80000000** 時， **a - 0x8000000** 會變成 **a + (-b) => a + 0x80000000**，所以此時減法的 **overflow detection** 方式，會變成 加法的 **overflow detection** 方式。

3、 **Mips** 的指令，大部份都可以透過 **shift + mask** 來得到，**mips** 指令的結果，在實作上，只要注意，是否 **bit shit** 前，他是 **signed or unsigned bit,** 如果是 **signed,** 我 **shift** 之前，一定會把相關的位元，移至最高位元，之後的 **shift** 就會出現相關的

signed. 如果是 **unsigned,** 可以先轉成 **unsigned** 再位移到你要的位置，就不會出錯。

4、 如何確認單一指令實作的的正確性。我會先在 **main** 裡面用 **c** 寫一個，對應指令的操作，再簡單給與，正 與負的 **signed bit,** 然後給出極大與極小值，就可以得到，或預期出，是否得到指令對應的結果。 這樣你就可以確認你指令實作的正確性。

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**。

如在讀檔時，我會 load D & I image file 各 1 次，所以我就實作一 load file 的 function，只要給我檔名，我就會回傳 int[] data array. 這樣我的方法寫好了之後，我 2 次的 load file action 就不需要再重覆爲不同的讀檔，而再寫一段同檔的 code。之後，不管有幾個 file 要讀，我只要 call 那 function 就好。

例如：

1、透過統一的 error hander 處理所有 error 的問題，1、秀出 message 2、決定 continue or halt 都透過，這 handler 決定，要顯示什麼，以及接下來的動作是什麼，或都還是直接離開程式。

2、統一的 memory acesss 存取的 error detection。當有對 memory access，定義一個 function ，完全的處理如何去 detect 相關 memory access 可能的錯誤，這樣的好到，也是方便維護與 code 如果，有 memory access 的其他錯誤要加，我只要在那 function 上調整就好。

3、專注在自己的功能上，如：我們的指令有 4 種 type, 我就設計各別的 4 種功能，各自去處理他。原因是因爲，每種指令，要 detect 的 error 可能會不同，且每種指令的功能也不同，各自獨立，就可以 focus 在目前那種指令的行爲上，以後要維護也容易，因爲不用全部改，只要改特定的 type 的指令。

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?

Just focus on extremely case.

A.

在 dImage 裡放 a、max=0x7fffffff (2,147,483,647)
b、min=0x80000000 (-2,147,483,648)
c、1=0x01
d、-1=0xffffffff
f、0x81828384

因爲，在助教的 open test case & hidden test case 裡，如果都測過。那麼基本上，2/3 的功能的 instruction 都被正確實作，且程式有一定的完整度。

所以，設計的自己的測資，就不能考慮，正常的情形，需要考慮 boundary 附近的 case ，所以我的運算的測資，就是利用 a ~ d 這 4 個數去做運算，而這幾個數，都是在 boundary case 上也就是極大 與極小值，再配上 2 個 1or -1，我就可以透過這 4 種數字不同，可以想到的排列組合去，枚舉出可能會發生 number overflow 的情形。

以下列出幾種相關的方式:

max - (-1)，1+ max ，-1 - max ，-1 – min， min - min ，(1) - min ，0 - min 詳細的情形，請看最後一部份列出的組語, 透過不同的枚舉，再配合上，運算完後，存會相同的 register 位置，提高出錯的可能，因爲在 detect sign bit 有些人可能會沒想好先後順序，等運算才去 detect sign bit, 但，透過存 與 取 相同的位置，找出 粗心的 case .

B.

執行檢查每一種 type instruction R & I instruction，確保每一種結果，要是正確的。 (目的是找出粗心的部份，也爲自己做最後的 debug 工作，確保我程式的結果。)

C.

著重在 lw, lh, lhu, lb, lbu, sb, sh, sw 的功能上，因爲這部份，我自己在實作時，一發始也搞錯，所以，我特別故意針對這功能去做不同的 case 的組合。透過給出不同位址與 offset 的組合，看看是否，值先存在記憶體，之後再取不同的值存回暫存器，及再存回記憶體來回搭配記憶體與暫存器，看看最後的結果，是否能 match 實際的結果值。

```
如：    lb $16, 16($0)          # load byte from dmemory[16] to $16
        sb $16, 1016($0)         # store byte from $16 to    dmemory[1016]
        lb $16, 17($0)          #    load byte from dmemory[17] to $16
        sb $16, 1017($0)         # store byte from $16 to    dmemory[1017]
        lb $16, 18($0)          # load byte from dmemory[18] to $16
        sb $16, 1018($0)         # store byte from $16 to    dmemory[1018]
        lb $16, 19($0)          # load byte from dmemory[19] to $16
        sb $16, 1019($0)         # store byte from $16 to    dmemory[1019]
        lw $16, 1016($0)        (看實際暫存器與記憶體的值有無取錯)
```

D.

Write to $zero Error

把所有會發生 write to $zero 的情形都列出來。找出過濾出，粗心的 case，例如，有人某些指令可能會沒加判斷，造成 Write to $zero Error 發生。

E.

你會發現，目前我的測資都很少有 error 的 detect，是因爲，不想浪費在錯誤上，因爲一發生，memory overflowed 及 data misaligned 就會 halt 住。所以儘可能不做就不做。最後，測試是，access address 是合法的，但加上存取的 offset 時，會存取超過 memory available address. 且配合上，奇數的的存取位置，讓 lh 會發生 3 個錯誤。 Write to zero , Memory address overflow 及 Data misaligned error.

D. Test Case Elaboration

```
##
## dimage=5
## dimage[0], max=0x7fffffff (2,147,483,647)
## dimage[1], min=0x80000000 (-2,147,483,648)
```

```
## dimage[2], 1=0x01
## dimage[3], -1=0xffffffff
##    dimage[4],    0x81828384
##      pc = 0x08 , sp =0x2
##

lw $9, 2($sp)              #    $t1,$9 = min
lw $8, 0($0)               #    $t0,$8 = max
lw $10, 6($sp)             #    $t2,$10 = 1
lw $11, 12($0)             #    $t3,$11 = -1
## 枚舉出可能會發生 number overflow 的情形
addi $13, $8 , 0           #    [check Overflowed]    $t5,$13 = max
add $13, $13, $13          #    $t5,$13=max + max
addi $13, $8 , 0           #    $t5,$13 = max
sub $13,    $13, $13       #    $t5,$13 = max - max
addi $13, $8 ,0            #    $t5,$13 = max
add $13,    $13, $10       #    $t5,$13 =    max +1
addi $13, $8 , 0           #    $t5,$13 = max
sub    $13, $13, $11       # $t5,$13 =    max - (-1)
addi $13,$8 ,0
add    $13, $10, $13       #    $t5, $13 = 1+ max
addi $13,$8 , 0
sub    $13, $13,    $9     #    $t5,$13 =     max - min
addi $13,$8 , 0
sub    $13, $11, $13       # $t5,$13 =    -1 - max
addi    $13,$9 , 0         #     $t5,$13 = min
add    $13, $13, $13       #     $t5,$13 =    min + min
addi $13,$9 , 0            #     $t5,$13 = min
sub    $13, $13, $13       #     $t5,$13 =    min - min
addi $13,$9 , 0            #     $t5,$13 = min
add $13, $13, $11          # $t5,$13 =    min + -1
addi $13,$9 , 0            #     $t5,$13 = min
sub $13, $13, $11          # $t5,$13 =    min - (-1)
addi $13,$9 , 0            #     $t5,$13 = min
sub $13, $13, $10          # $t5,$13 =    min - (1)
addi $13,$9 , 0            #     $t5,$13 = min
add $13, $11, $13          # $t5,$13 =    -1 + min
addi $13,$9 , 0
```

```
sub $13, $0, $13    # $t5, $13 = 0 - min
addi $13,$9 , 0
sub $13, $13 , $0    #$t5 , $13 = min - 0
addi $13,$9 , 0
add $13, $0,    $13          # $t5,$13 =      0 + min
addi $13,$9 , 0
sub $13, $10, $13             # $t5,$13 =    (1) - min
srl $13, $11, 3         # [ check Rtype]    (logic) -1 >> 3
sra $13, $11, 3          # (arithmetic) -1 >> 3
and $13, $8, $10        #    and max & 1
or    $13, $8, $10       #    or max &    1
xor $13, $8, $10        #    xor max |    1
nor $13, $8, $10        #     xor max ^    1
nand $13, $8, $10      #     ~ (max &    1)
sll    $13, $11, 3       #     -1 << 3
sll $0, $0, 0            #     nop
sll          $0,          $11,          3          #                    -1          <<          3
sb $10, 1023($0)           # [check I type]    save 1 -> dmemory[1023]
addi $14, $0, 0x400       # $14= 1024 value
sb $10, -2($14)            # save 1 -> dmemory[1022]
sh $10, 1020($0)           # save 0,1 -> dmemory[1020],dmemory[1021]
lw  $15,  -4($14)            # [check sb, lw] load word from dmemeory[1020]
to$15
lb $16, 16($0)             # load byte from dmemory[16] to $16
sb $16, 1016($0)          # store byte from $16 to    dmemory[1016]
lb $16, 17($0)             #    load byte from dmemory[17] to $16
sb $16, 1017($0)          # store byte from $16 to    dmemory[1017]
lb $16, 18($0)             # load byte from dmemory[18] to $16
sb $16, 1018($0)          # store byte from $16 to    dmemory[1018]
lb $16, 19($0)             # load byte from dmemory[19] to $16
sb $16, 1019($0)          # store byte from $16 to    dmemory[1019]
lw $16, 1016($0)          # load word form dmemory[1016]
lbu $16, 16($0)           # [check lbu]    load byte from dmemory[16] to $16
sb $16, 1016($0)          # store byte from $16 to    dmemory[1016]
lbu $16, 17($0)           # load byte from dmemory[17] to $16
sb $16, 1017($0)          # store byte from $16 to    dmemory[1017]
lbu $16, 18($0)           # load byte from dmemory[18] to $16
sb $16, 1018($0)          # store byte from $16 to    dmemory[1018]
```

```
lbu $16, 19($0)          # load byte from dmemory[19] to $16
sb $16, 1019($0)          # store byte from $16 to    dmemory[1019]
lw $16, 1016($0)          # load word form dmemory[1016]
lh $16, 16($0)           # [check lh sh] load byte from dmemory[16] to $16
sh $16, 1016($0)          # store byte from $16 to    dmemory[1016]
lh $16, 18($0)           # load byte from dmemory[17] to $16
sh $16, 1018($0)          # store byte from $16 to    dmemory[1017]
lw $16, 1016($0)          # load word form dmemory[1016]
lhu $16, 16($0)          # [check lhu] load byte from dmemory[16] to $16
sh $16, 1016($0)          # store byte from $16 to    dmemory[1016]
lhu $16, 18($0)          # load byte from dmemory[17] to $16
sh $16, 1018($0)          # store byte from $16 to    dmemory[1017]
lw $16, 1016($0)          # load word form dmemory[1016]
lui $16, -3
## check write to $0 Error
add $0, $10, $10     # check write to $0 for I & R type [write to $0 Error]
sub $0, $10, $10
and $0, $10, $10
or $0, $10, $10
xor $0, $10, $10
nor $0, $10, $10
nand $0, $10, $10
slt $0, $10, $10
srl $0, $10, 3
sra $0, $10, 3
addi $0, $10, -2
lw $0, 4($0)
lh $0, 2($0)
lhu $0, 4($0)
lb   $0, 3($0)
lbu $0, 1($0)
lui $0, -3
andi $0, $10, -3
ori $0,$10,-3
nori $0,$10,-3
slti   $0, $10 , 3
lh $0, 1021($sp)     # access illegal address for halt simulator   and generate many errors at the same time.
```