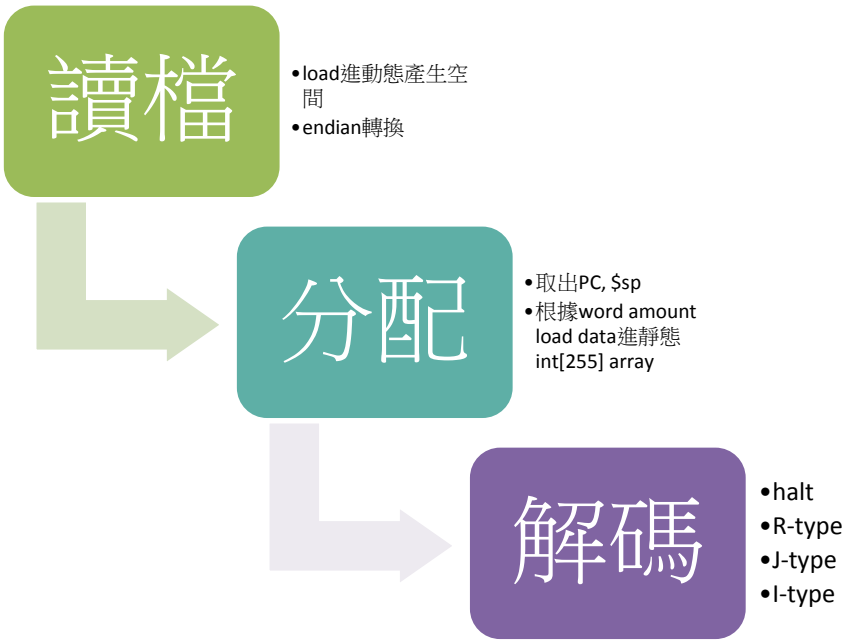
Report for Architecture project1

archi01, member:101021120 鄭宇翔, x1022108 曹士杰

Simulator Design

以下為程式整體流程圖：



我使用 C 語言來實現此 Simulator。合乎常理地，我使用了 array 結構來存 instruction 與 data。Array 規定大小為 1Kbyte，因此我使用 int[256] Array 來實作。

main():

在main()裡，一開始我先進入ReadFile()來讀檔(有測試是否讀到檔案)，得知檔案大小後，動態取得該大小的空間(有測試是否能取得這些空間)，將 iimage 與 dimage 各 4bytes 為一單位，存入動態取得大小的 int pointer(array。接著，我利用在 local 上創造 union 如下：

```
============================================================
union {
    int i;
    char buf[4];
} u;
u.i = 0x12345678;
```

```
if (u.buf[0] == 0x12) {
    /* transform data from little endian into big endian */
}
============================================================
```

來測試環境是否為 big endian；若是，則作 endian 轉換；若否，略過。然後回到 main()。(目前認為這方法蠻好的、蠻簡潔有力的。)

接著，我呼叫 CleanArrays()，基本上就是用 for 迴圈將 global array 的內容都設成 0。

然後，呼叫 distribute()，將動態的 array 前兩個 words 取出並放入 PC 與$sp，再照著第二個 word(word amount)去將動態的 array 剩下的內容搬入 global int[256] array，imem[256]起始位置為 PC，dmem[256]起始位置為 0，並判斷是否存放範圍超過 1Kbyte 範圍，若超過，則直接 halt。最後 free 掉動態 array。

接著,我用 while(1)迴圈,先印 cycle(此為 Cycle 0),再呼叫 decode()，接著看是否 opcode 為 0x3f，若是，代表是 halt，break 迴圈並正常結束。

在迴圈裡，有 initialize()的動作來確保每次所使用的 global 變數是乾淨的，並且，保險起見，有測試是否 CycleCount 大於 500K 圈。

在 decode()前先測試是否 PC 超過範圍，再測試 PC 與$sp 是否為 4bytes 的倍數，這是必須，待兩者測完後，若違反其一則 halt。

decode()--R_type()/I_type()/J_type()

在 decode()裡，parse 出 opcode([31:26]bit)，判斷若為 0x3f(halt)則 return 回 main()。再作 PC += 4(我採用先加，為 project2 方便)，再分類，若 opcode 為 0，跳去 R_type()；若為 0x02 或 0x03，跳去 J_type()；其他丟入 I_type()。

在各 type 裡，先 parse 出該有的部分，再判斷是否為未知的 instruction(Appendix A 上未寫到的)，在 R_type()與 I_type()需判斷是否有寫到$0,接著 switch case 判斷是何者 instruction 並做各 instruction 該做的事，包含執行指令與錯誤測試(會跳到 ErrorMsg()去顯示錯誤再跳回來)。

Write to $0 error 是在進 case 前先測，其餘的 error 是在 case(instruction)裡測。

在 add, sub, addi 我使用 MSB 判斷是否 Number overflow，並且，sub 指令需多判斷 a - b 之 b 是否為 0x80000000,若是,也為 Number overflow(因為 sub 是以 a + (-b)實作，而 -0x80000000 不存在)，且運算變為作 a + 0x80000000；srl 用 mask 的方式實現。

在 MSB 判斷是否 Number overflow 裡，需先將 R[rs]與 R[rt](R-type)

之 MSB 在執行指令前存起來，因為不知道 R[rd] = R[rs] op R[rt]之 rd 是否與 rs 或 rt 一樣，若不預先存起來，在執行指令後判斷 MSB 會出錯。(因為原本的 R[rs]與 R[rt]可能會因為與 R[rd]是相同的而被改變值，MSB 可能會被改變。)
MSB 判斷法請參考以下範例：

```
============================================================
(for case add)
/* P+P=N or N+N=P */
if ( MSBrs == MSBrt && (R[rd] >> 31 & 1) != MSBrs ){
    /* Number overflow */
}
============================================================
```

在某些地方，我們需要 sign extension，我的方法是先左移使 MSB 對到 highest bit，再右移回來(因為 C 語言裡的右移是算術邏輯右移 sra)。

而要 unsigned extension 的話，就直接右移並將右移產生的 n 個最高位元用 n 個 0 bitwise and mask 掉即可。

在與 access memory 有關的 instruction 裡，先測試 Address overflow，再測試 Misalignment error，並用一些乾淨、簡潔、漂亮的算法(組合了一些 modulo, or & and mask…)作存取(在此不詳述，請參照以下範例 code)。關於 sb, sh 較特殊，我先將要存的位置 &0 (清空)，再以 |= 的方式放入。

最後，handle error 的方式是反應在該 cycle 遇到的所有 error 後，該 halt 則 halt(Address overflow & Misalignment error)，該 keep executing 則 keep executing(Write to $0 & Number overflow)。
關於如何實現請參考以下範例：

```
============================================================
(for case srl)
if (shamt == 0) R[rd] = R[rt];
else  R[rd] = (R[rt] >> shamt) & ~(0xffffffff << (32 - shamt));

(for case lh)
R[rt] = (mem_data_d[(R[rs] + imm)/4] << ((2 - finite_groupZ4(R[rs] + imm)
* 8)) >> 16;

(for case sh)
mem_data_d[(R[rs] + imm)/4] &= ~(0xffff << (finite_groupZ4(R[rs] + imm)
* 8);
```

```
mem_data_d[(R[rs] + imm)/4] |= (R[rt] & 0xffff) << (finite_groupZ4(R[rs]
+ imm) * 8);
============================================================
```

在 srl 部分，因為 C 語言不允許一個變數<< 32 位元，因此特別分開 shamt == 0 之 case。

coding 途中曾遇到我想算 modulo 的問題，我想要 1 mod 4 = 1, 5 mod 4 = 1, (-1) mod 4 = 3, (-5) mod 4 = 3 …。在 C 語言裡，modulo 可用 a % b 實現(b > 0)，if a > 0 結果沒問題；然而，若 a 是負數，C 語言是以這種方式實現負數之 modulo: -(abs(a) % b)，因此，(-1) % 4 = -1, (-5) % 4 = -1。

為了 handle 以上狀況，我另額外寫了小函式解決，名叫 finite_groupZ4()，內容如下：

```
=====================================================================
int finite_groupZ4(int x){
    if (x < 0)
        return (x % 4) + 4;
    return x % 4;
}
=====================================================================
```

因此 finite_groupZ4(-1) = ((-1) % 4) + 4 = (-1) + 4 = 3, and finite_groupZ4(-5) = ((-5) % 4) + 4 = (-1) + 4 = 3，達到我想要的結果。

(p.s 為何叫做 finite_groupZ4 請參考抽象代數課本對有限群之討論，我本身為數學系的學生因此這樣命名 XDD。

基本上是定義 $Z_n := \{0, 1, 2, \ldots, n-1\}$，大於 n-1 的正整數在 Z_n 裡代表該數 mod n 之結果；而負整數則是一再加 n 加成最小正整數之結果。
想迅速了解 Z_n 可參考下方維基百科網址：
http://en.wikipedia.org/wiki/Cyclic_group)

ErrorMsg():

在 error handler 方面，原本以為遇到 Write to $0 與 Number overflow 是顯示 error 後直接跳過此指令執行下一圈，也以為遇到 Address overflow & Misalignment error 是直接馬上 halt；但助教臨時改成顯示全部錯誤後再執行下一圈 or halt。不過修正也是挺簡單的，原本馬上 halt 的 error 將 exit() 刪去而改成 set HaltFlag = 1，除了 load, store 相關之指令需立即 halt

之外(否則 Address overflow 可能會造成 Segmentation fault 之問題)，其餘可在執行完指令之後判斷 HaltFlag == 1 則 halt；而 Write to $0 方面則是將原本的 return 刪去改成不 return，但需注意，這樣會使指令正常運作並將值寫入$0。解決方式依然簡單，在 main()裡要 print cycle 之前的 initialize()函式裡加上 R[0] = 0 即可。

Conclusion:

整個 code 扣掉註解約 400 行，我覺得是架構挺鮮明、易懂與 compact，並且挺好維護的，中間各 instruction 的 decode 部分，算法也挺漂亮的，幾乎一至二行可解決，算是頗精闢的算法。除了 code 部分相當完整以外，註解也相當豐富易讀。除了程式部分以外，Makefile 的撰寫方式可抵除工作站上執行時差；README 的內容也是相當完整，testcase 也是相當 tricky(下一部分將會介紹)；甚至包括此份 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

我們的 testcase 中同時設計了錯誤指令和正確指令測試。大約各佔 50%，錯誤測試有很細的層面，畢竟測試的目的是「證明軟體中有 bug」(參照軟體工程課程內容)，請參照以下。

錯誤指令測試：

錯誤指令測試的意圖在於講所有的錯誤可能都展示出來，以此來檢查其他組的 simulator 是否可以檢查出所有的錯誤種類 我們首先單獨進行簡單的 write to zero 和 number overflow 錯誤測試，再進行一條指令同時出現 write to zero 和 number overflow 的錯誤測試，最終進行一行指令同時出現 write to zero、address overflow 和 misalignment 三條錯誤的錯誤測試并因為 misalignment 而終止程式。

E.g1 錯誤 write to zero 測試：

```
sll $0, $0, 0
srl $0, $1, 16     # write to $0
lui $0, -2^15      # write to $0
```

Write to zero 大家應該都會考慮，我們使用了助教測資裡未用到的指令來測試 write to zero。(用 grep 指令發現助教沒用過)

E.g2 錯誤 number overflow 測試：

```
lw $1, 0($sp)      # $1 = 0x80000000      data at 0
sub $1, $0, $1     # -$1 doesn't exist, number overflow
sub $25, $8, $1    # -$1 doesn't exist, number overflow,
                   # $8 = 0xABCD0000  =>
                   # $8 = 0x2BCD0000
addi $12, $1, -1   # $12 = 0x7FFFFFFF, number overflow
sub $12, $12, $1   # -$1 doesn't exist, number overflow,
                   # do $12 = $12 + $1, $12 => 0xFFFFFFFF
addi $1, $1, -1    # $1 = 0x7FFFFFFF, number overflow
addi $1, $1, 1     # $1 = 0x80000000, number overflow
```

首先我們使用多次改變最高位的方式連續測試 overflow，並且，若是 b = 0x80000000，-b 不存在之 number overflow，原本的 c = a – b =

a + (-b) 要改成 c = a + b。

```
lui $1, -2^15      # $1 = 0x80000000
sub $1, $1, $1     # -$1 doesn't exist, number overflow
```

進而我們測試將$1 賦值為 0x80000000 后進行自身減法，減法操作中 $1 要先變為（- $1），这時 0x80000000 取負數時出現 number overflow。

E.g3 錯誤 write to zero & number overflow 同時出現測試：

```
lw $1, 0($sp)      # $1 = 0x80000000      data at 0
sub $0, $1, $1     # -$1 doesn't exist, number overflow
```

將 rd 改為$0 將造成 write to zero & number overflow 同時出現

E.g4 錯誤 write to zero & address overflow & misalignment 同時出現測試：

```
addi $sp, $sp, 3   # $sp 此時為 3
lh   $0,  1022($sp)
```

此時將同時出現 write to zero & address overflow & misalignment 三條錯誤。

正確指令測試：

E.g1 與 access 記憶體有關的 instruction 測試：

```
addi $1, $0, 0x400
addi $2, $0, 0xABCD
sb $2, -1($1)      # store to 1023
lw $4, 1020($0)    # $4 = 0xCD000000
sh $2, -4($1)      # store to 1020
lw $5, 1020($0)    # $5 = 0xCD00ABCD
sb $2, -2($1)      # store to 1022
lw $3, 1020($0)    # $3 = 0xCDCDABCD
```

```
        sh $2, -8($1)      # store to 1016
        lw $7, 1016($0)    # $7 = 0x0000ABCD
        sw $2, -8($1)      # store to 1016
        lw $6, 1016($0)    # $06 = 0xFFFFABCD
        sb $2, -10($1)     # store to 1014
        lw $9, -12($1)     # $9 = 0x00CD0000
        sh $2, 1014($0)    # store to 1014
        lw $8, -12($1)     # $8 = 0xABCD0000

        # lb & lbu
        lbu $14, 1023($0) # $14 = 0x000000CD
        lb $15, 1023($0)  # $15 = 0xFFFFFFCD
```

正確指令測試我們決定採用連續執行 lb、lh、lw、sb、sh、sw 指令，最後比較 lbu 與 lb 結果之不同。以測驗其他組的 simulator 是否能正確執行這幾條指令并正確處理 offset 為負的計算問題。并通過連續的內存修改操作和暫存器的多次賦值来检验其他组 snapshot 能否正確追蹤寄存器的值。

E.g2 shift、遮罩、與$0 作 bne 之測試：

```
        # addi & bne & sll
        addi $10, $10, 0x8000 # $10 = 0xFFFF8000
        addi $11, $0, 0xFFFe  #
Back:   and $10, $10, $11     #
        sll $11, $11, 1       # $11 <<= 1
        bne $10, $0, Back     # Back = -3

        # nor & bne & sra
        addi $10, $10, 7      # $10 = 7
        lui $11, 0x8000       # $11 = 0x80000000
Back2:
        nor $12, $10, $11     #
        sra $11, $11, 1       # $11 >>= 1
        bne $12, $0, Back2    # Back2 = -3
```

此段指令考慮同學執行之正確度，並且可藉由這種方式增加 cycle 數目。

E.g2 是否寫進$0 之測試：

```
srl $0, $1, 16        # $1 = 0x8000000, but write to $0
bne $0, $31, 3        # see whether ($0 == 0)?

lui $0, -2^15         # write to $0
bne $0, $31, 2        # see whether ($0 == 0)?

sub $0, $1, $1        # $1 = 0x8000000
                      # number overflow & write to $0
bne $0, $31, 1        # see whether ($0 == 0)?
```

撰寫此段指令的原因是，因為在 simulator 裡，助教希望將所有 error 都 point out 再繼續下一圈 or 直接 halt，言下之意，有 Write to $0 error 的指令為了顯示該指令的其餘錯誤，必須繼續執行該指令到結束，因此$0 的值可能被改變，但同學可投機取巧在 print cycle 時$0 不套 R[0]變數而直接印常數 0x0000000，因此，為了檢查同學是否不小心改到$0 但忘了改回來而寫此測試。

在最後，最 op 的測試是這個：

```
# three errors
addi $sp, $sp, 3   # $sp => 0x3
lh  $0, 1022($sp)  # access address 1025
```

在此圈會同時產生 3 個錯誤(Write $0 error, Address overflow, Misalignment error)並 halt，在此考驗同學的錯誤測試順序以及時機，在 iimage.bin 與 dimage.bin 之部分，指定數目的 word 之後放了些許垃圾值，使資料不為 4byte 的倍數，正確來說應該要當作沒看見垃圾值 simulate，但有些同學 simulator 沒設計好可能會跑到垃圾值，這也是一項隱藏的小測試，very tricky。

在 testcase 資料夾附有名為 archi01_testcase.S 的 pseudo code，以及本 simulator 跑出的 snapshot.rpt 與 error_dump.rpt。

Conclusion:

整體而言，testcase 算是相當精闢，雖然轉回高階語言內容可能沒什麼實質意義，但就像我開頭所說的：「測試的目的是『證明軟體中有 bug』」，正確的要跑對，錯誤也要處理對，才算是好的軟體。