实验一：MIPS程序设计

姓名： 雍崔扬
学号： 21307140051
日期： 2024 年 2 月 28 日

1. 实验目的

- 熟悉 QtSPIM 模拟器；
- 熟悉编译器、汇编程序和链接器；
- 熟悉 MIPS 体系结构的计算，包括：
  - MIPS 的数据表示；
  - 熟悉 MIPS 指令格式和寻址方式；
  - 熟悉 MIPS 汇编语言；
  - 熟悉 MIPS 的各种机器代码表示，包括：
    - 选择结构；
    - 循环结构；
    - **过程调用：调用与返回、栈、调用约定**等；
    - 系统调用；

2. 实验任务

2.1 调试

调试给定的程序 `p1.asm`、`p2.asm`、`p3.asm`，记录程序运行的结果.

(1) p1.asm

```
.globl main

.text

main:
    ori $t2, $0, 40         # Register $t2 gets 40
    ori $t3, $0, 17         # Register $t3 gets 17
    add $t3, $t2, $t3       # Register $t3 gets 40 + 17

    ori $0, $0, 40          # Register $0 appears to get 40 ...
    ori $t4, $0, 0          # ... but it really doesn't

    ori $v0, $0, 10         # Prepare to exit.
    syscall                 # Exit.
```

运行结果：(这个程序没有输入和输出)

1. 寄存器 `$t2` 的值从 `0` 变为 `40 (0x00000028)`；
2. 寄存器 `$t3` 的值从 `0` 变为 `17`，最后变为 `57 (0x00000039)`；
3. 寄存器 `$0` 的值无法被改动，仍为 `0 (0x00000000)`；

4. 寄存器 `$v0` 的值从 `0` 变为 `10 (0x0000000a)`，用于执行 `syscall(10)` 以退出；

(2) p2.asm

```
.globl main

.text

main:
    ori $t2, $0, 40         # Register $t2 gets 40
    lui $t2, 0x1234         # Upper half of register $t2 gets 0x1234
    ori $t2, $t2, 40        # Lower half of register $t2 gets 40

    li  $t3, 0x12340028     # Register $t3 gets 0x12340028

    li  $v0, 10             # Prepare to exit.
    syscall                 # Exit.
```

运行结果：(这个程序没有输入和输出)

1. 寄存器 `$t2` 的值从 `0` 变为 `40 (0x00000028)`，然后高4位被设为 `0x1234`，
   低4位又被设为 `40 (0x0028)`，最终结果为 `0x12340028`；
2. 寄存器 `$t3` 的值从 `0` 变为 `0x12340028`；
3. 寄存器 `$v0` 的值从 `0` 变为 `10 (0x0000000a)`，用于执行 `syscall(10)` 以退出；

(3) p3.asm

```
.globl main             # Make main, A, and h globl so we can
.globl A                # refer to them by name in SPIM.
.globl h

.data                   # Data section of the program

    A:      .word   1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
50
    h:      .word   40

.text                   # Text section of the program

main:                   # Program starts at main.
    la      $t0, h          # Register $t0 gets address of h

    la      $t1, A          # Register $t1 gets address of A

    lw      $t2, 0($t0)     # Register $t2 gets h

    lw      $t3, 32($t1)    # Register $t3 gets A[8]

    add     $t3, $t2, $t3   # Register $t3 gets h + A[8]

    sw      $t3, 48($t1)    # A[12] gets h + A[8]

    li      $v0, 10         # Prepare to exit.
    syscall                 # Exit.
```

运行结果：(这个程序没有输入和输出)
它实现的是(用高级语言描述) `A[12] = A[8] + h`，实现步骤如下：

1. 用 `la` 指令将整数 `h` 的地址和数组 `A` 的首地址分别装入寄存器 `$t0` 和 `$t1`；
2. 用 `lw` 指令将整数 `h` 的值 `40 (0x28)` 和 `A[8]` 的值 `19 (0x13)` 分别装入寄存器 `$t2` 和 `$t3`；
3. 用 `add` 指令计算寄存器 `$t2` 和 `$t3` 的和 `40 + 19 = 59 (0x3b)` 并将结果存入寄存器 `$t3`；
4. 用 `sw` 指令将寄存器 `$t3` 的值 `59 (0x3b)` 存入 `A[12]`；
5. 最后设置寄存器 `$v0` 的值为 `10 (0x0a)`，用于执行 `syscall(10)` 以退出；

2.2 改写程序

我们的任务是改写 `p1.asm`，以接收两个整数，计算结果后输出，并询问用户是否继续下一轮.

设计过程：

为实现部分功能的模块化，我创建了四个函数：

(1) `change_line` 函数通过打印 `msg6 ("\n")` 来实现换行：

```
change_line:          # Change line by outputing "\n"
    li $v0 4
    la $a0 msg6
    syscall
    jr $ra
```

(2) `print_get` 函数打印参数寄存器 `$a0` 中的字符串，并返回用户的输入(一个整数)：

```
print_get:            # Print a string and get an integer
    li $v0 4
    syscall
    li $v0 5
    syscall
    jr $ra
```

(3) `print_print` 函数打印参数寄存器 `$a0` 中的字符串和 `$a1` 中的数据值：

```
print_print:          # Print a string and print an integer
    li $v0 4
    syscall
    move $a0 $a1
    li $v0 1
    syscall
    jr $ra
```

(4) `continue_or_exit` 函数实现是否进行下一轮计算的判断.
如果用户的输入是 `0`，则换行后继续下一轮；
如果用户的输入是 `1`，则高喊 `"EXIT!"` 后终止程序；
如果用户的输入既不是 `0` 也不是 `1`，则继续向用户索要指示值；
这个函数调用了 `print_get` 函数和 `change_line` 函数.

```
continue_or_exit:           # Judge whether to continue or exit
```

```
    keep_on_asking:
        la $a0 msg7         # Print "Do you want to try another(0_continue/1_exit):"
        jal print_get
        move $t4 $v0        # Get user's instruction
    case0:
        bne $t4 $0 case1    # if input==0 change line and loop
        jal change_line
        j loop
    case1:                  # else if input==1, print "EXIT!" and exit
        addi $t5 $0 1
        bne $t4 $t5 default
        li $v0 4
        la $a0 msg8
        syscall
        j EXIT
    default:                # else keep on asking
        j keep_on_asking
```

(5) 核心代码： 主函数 `main`、数据 `data`、文本 `text`

```
.data
    msg1: .asciiz "Please enter 1st number:"
    msg2: .asciiz "Please enter 2nd number:"
    msg3: .asciiz "The result of "
    msg4: .asciiz " & "
    msg5: .asciiz " is:"
    msg6: .asciiz "\n"
    msg7: .asciiz "Do you want to try another(0_continue/1_exit):"
    msg8: .asciiz "EXIT!"

.text

main:
    loop:
        la $a0 msg1             # Print "Please enter 1st number:"
        jal print_get
        move $t1 $v0            # Get 1st number into $t1

        la $a0 msg2             # Print "Please enter 2nd number:"
        jal print_get
        move $t2 $v0            # Get 2nd number into $t2

        add $t3 $t1 $t2         # Calculate $t3=$t1+$t2

        la $a0 msg3             # Print "The result of " and 1st number
        move $a1 $t1
        jal print_print

        la $a0 msg4             # Print " & " and 2nd number
        move $a1 $t2
        jal print_print

        la $a0 msg5             # Print " is:" and sum
        move $a1 $t3
```

```
        jal print_print
        jal change_line         # Change_line
        jal continue_or_exit    # Judge whether to continue or exit

    EXIT:
        li $v0 10
        syscall
```

运行结果：

Console

```
Please enter 1st number:114514
Please enter 2nd number:985211
The result of 114514 & 985211 is:1099725
Do you want to try another(0_continue/1_exit):0

Please enter 1st number:20030320
Please enter 2nd number:20160111
The result of 20030320 & 20160111 is:40190431
Do you want to try another(0_continue/1_exit):2333
Do you want to try another(0_continue/1_exit):918
Do you want to try another(0_continue/1_exit):1223
Do you want to try another(0_continue/1_exit):0

Please enter 1st number:010205
Please enter 2nd number:019602
The result of 10205 & 19602 is:29807
Do you want to try another(0_continue/1_exit):1
EXIT!
```

每一轮打印的解释：

- ① 前两行都通过调用 `print_get` 函数实现，输出字符串并接收用户的输入；
- ② 第三行打印的 C 语言表示是 `printf("%s%d%s%d%s%d\n");`
  可由三个 `print_print` 函数和一个 `change_line` 函数完成.
- ③ 最后一行打印通过调用 `continue_or_exit` 函数完成，
  根据用户的输入判断继续还是退出.

2.3 把 C 代码转换为 MIPS 汇编代码

考虑如下的 C 代码：

```c
#include <stdio.h>
int sumn(int *arr, int n)
{
    int sum = 0;
    for (int idx = 0; idx < n; idx++)
        sum += arr[idx];
    return sum;
```

```
}
int main()
{
    int arrs[] = {9, 7, 15, 19, 20, 30, 11, 18};
    int N = 8;
    int result = sumn(arrs, N);
    printf("The result is: %d", result);

    return 0;
}
```

设计过程：

(1) 首先：
C 代码将 `sumn` 放在了 `main` 函数前，这样省去了声明的麻烦.
转变为 MIPS 汇编代码时，也可以将 `sumn` 函数放在 `main` 函数前，
但要在 `sumn` 函数前加上 `j main` 语句.
习惯上应将 `sumn` 函数放在 `main` 函数之后.

(2) 其次：
C 代码将初始化数组的语句 `int arrs[]={9,7,15,19,20,30,11,18};` 放在了 `main` 函数中，
这样 `sumn` 函数便无法直接使用数组 `arrs`，
必须由 `main` 函数将 `arrs` 的首地址作为参数传给 `sumn` 函数，
但在 MIPS 汇编代码中，初始化数组的语句放在 `.data` 中更方便：

```
.data
    arrs:    .word    9,7,15,19,20,30,11,18
```

这样定义的数组 `arrs` 是一个全局数组， `sumn` 函数可以直接使用 `arrs` 的首地址.

(3) 最后也是最关键的：
`sumn` 函数中的 `for` 循环如何转换为 MIPS 汇编代码：

- ① 循环条件及其判断可以用 `slt` 和 `beq` 语句实现：

  ```
  for:
      slt $t1 $t0 $a1 #$t1=idx<N, idx=$t0, N=$a1
      beq $t1 $0 done
  ```

  `slt` 判断条件 `idx < N` 是否成立，成立则 `$t1` 置为 `1`，否则 `$t1` 置为 `0`；
  `beq` 判断 `$t1` 是否为 `0`，为 `0` 则跳出循环.

- ② 循环语句的翻译：

  ```
  sll $t2 $t0 2       # $t2 = 4 * idx + (int*)arrs = (int*)arrs[idx]
  add $t2 $t2 $a0
  lw $t3 0($t2)       # $t3 = arrs[idx]
  add $t4 $t4 $t3     # $t4 = $t4 + $t3, i.e. sum =s um + arrs[idx]
  ```

`$t2` 为 `4*idx`（通过 `idx` 左移 2 位实现）+ `arrs` 的首地址，构成 `arrs[idx]` 的地址;
`$t3` 为 `arrs[idx]` 的值;
`$t4` 为 `sum`，加上 `arrs[idx]` 以更新 `sum` 的值.

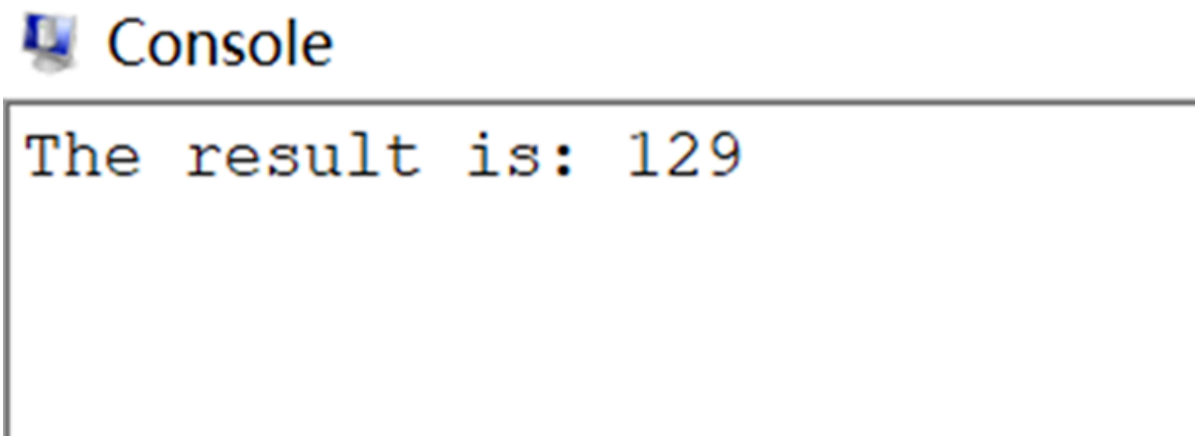
- ③ 更新循环变量并开始下一轮的循环条件判断：

```
addi $t0 $t0 1      # $t0=$t0+1, i.e. idx++
j for
```

MIPS 汇编代码：

```
.data
    arrs:   .word   9, 7, 15, 19, 20, 30, 11, 18
    # int arrs[]={9, 7, 15, 19, 20, 30, 11, 18}

    msg: .asciiz "The result is: "

.text

j main
sumn:
    addi $t4 $0 0           # int sum = 0
    addi $t0 $0 0           # int idx = 0
    la $a0 arrs             # Get initial address of arrs
    for:
        slt $t1 $t0 $a1     # $t1 = idx < N
        beq $t1 $0 done
        sll $t2 $t0 2       # $t2 = idx * 4 + (int*)arrs = (int*)arrs[idx]
        add $t2 $t2 $a0
        lw $t3 0($t2)       # $t3 = arrs[idx]
        add $t4 $t4 $t3     # $t4 = $t4 + $t3, i.e. sum = sum + arrs[idx]
        addi $t0 $t0 1      # $t0 = $t0 + 1, i.e. idx++
        j for
    done:
        move $v0 $t4        # return $t4 = sum
        jr $ra

main:
    addi $s1 $0 8           # int N = 8
    move $a1 $s1            # int result = sumn(arrs,n)
    jal sumn
    move $s2 $v0
    li $v0 4                # Print "The result is: "
    la $a0 msg
    syscall
    li $v0 1                # Print integer result
    move $a0 $s2
    syscall
    li $v0 10               # Terminate main
    syscal
```

运行结果：



2.4 优化汇编程序 `fib-o.asm`

参考资料 [1994]MIPS Assembly Language Programming CS50 Discussion and Project Book 3.1.1.3 处的 `fib-o.asm` 汇编程序的主要思想 (用 C 代码描述) 为：

```c
int fib(int n)
{
    if(n < 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

其子问题个数，即递归树中节点的总数，显然为指数级别 $O(2^n)$；
解决一个子问题需要 `fib(n) = fib(n-1) + fib(n-2)` 一个加法操作，时间复杂度为 $O(1)$.
因此整个算法的时间复杂度为 $O(2^n)$.
从递归树我们可以看到，在这个算法里存在着大量的重复计算.

实际上，我们可以从 Fibonacci 数列的第 0 个第和 1 个数出发，
每次将已计算出的末尾两个数相加得到下一个数，迭代 n 次，
就可以得到第 n 个 Fibonacci 数.
用 C 代码描述如下：

```c
int fib(int n, int first, int second)
{
    if(n == 0)
        return first;
    else
        return fib(n - 1, second, first + second);
}
```

这里 n 是还需要迭代的次数.
若 n 等于 0，则说明不需要再迭代了，返回第一个尾数;
否则继续迭代.
这个算法一共有 n 个子问题，
每个子问题的解决只需要加减法 (用于迭代参数) 以及调用等常数个操作，
因此时间复杂度是 $O(n)$.

尾数递归算法的 MIPS 代码中，`n==0` 的情况直接返回 `first`；
其他情况，将参数调整为 `n - 1`、`second` 和 `first + second`，然后调用 `fib` 函数.
由于递归只关心当前情况下末尾的两个数和当前剩余迭代次数，
因此 `$a0`、`$a1` 和 `$a2` 调用前的值不用保存在栈框架中，直接更新即可.
只需将返回地址寄存器 `$ra` 的值保存在栈框架中，故栈指针只需开辟 4 字节大小的空间即可.

运行时间测试：

由于 QtSPIM 不支持 `syscall` 第 30 号指令，故无法直接测量程序运行时间，
但是我们可以将调用 `fib` 函数的操作循环多次，用秒表测量总运行时间，
来间接反映 `fib` 函数的性能.

核心代码：

```
# $a0 = n, $a1 = first, $a2 = second

fib:
    bne $a0 $0 fib_recurse  # If(n==0), return(first)
    move $v0 $a1
    jr $ra                  # Otherwise, return(fib(n-1, second, first +
second))

fib_recurse:
    addi $sp $sp -4         # Save $ra by creating a 4-byte stack
    sw $ra 0($sp)
    addi $a0 $a0 -1         # n--
    add $a2 $a2 $a1         # second2 = first1 + second1
    sub $a1 $a2 $a1         # first2 = second2 - first1 = second1
    jal fib                 # Call fib(n-1, second, first + second)
    lw $ra 0($sp)           # Restore $ra
    addi $sp $sp 4
    jr $ra                  # return(fib(n-1, second, first + second))
```

下表列出了 `fib-o.asm` (循环调用 100 次) 和 `fib-op.asm`(循环调用 200000 次)
计算第 21、23、25 个 Fibonacci 数所耗费的时间.

	21st	23rd	25th
fib-o.asm 暴力递归(100次循环)耗时(\s)	33.7	78.0	215
fib-op.asm 尾数递归(200000次循环)耗时(\s)	22.8	34.1	37.6
二者单次调用耗时比值	2.34×10^3	4.57×10^3	1.14×10^4

该测试验证了前者的时间复杂度是 $O(2^n)$，而后者的时间复杂度是 $O(n)$.
在计算第 21 以及更高位的 Fibonacci 数时，两种算法的效率相差了至少 3 个数量级.

3. 实验感想

- (1) 原则上，被调用函数应该计算返回值，但不应产生其他负面影响，
  即除了包含返回值的寄存器 `$v0` (和 `$v1`, 如果结果为 64 位数)，
  其他寄存器都不应该被修改， 否则这些寄存器的内容会被破坏.
  因此我们应该使用栈保存和恢复被调用函数使用的寄存器.
  但如果调用函数并不使用被调用函数使用的寄存器，

则对它们的保存和恢复就是无用的操作.
因此 MIPS 将寄存器分为**受保护类型**(`$ra`, `$s0`~`$s7`)和**不受保护类型**(`$a0~$a3`, `$t0~$t9`)
函数必须保存和恢复任何需要使用的受保护寄存器.
`lw` 和 `sw` 操作都比较费时，所以我们应该弄清楚哪些寄存器的值是必须保护的，以使代价最小化.

- (2) 我在阅读不同代码时，发现有些时候使用 `addi` 指令对寄存器赋值，
有些时候使用 `move` 指令对寄存器赋值，有些时候用 `li` 指令对寄存器赋值.
例如将 0 赋给 `$t0`，
可以写作 `addi $t0 $0 0`, 也可以写作 `move $t0 $0`，还可以写作 `li $t0 0`.
我好奇这三种赋值方式是否有性能上的差异，
于是编写了 `move-time.asm`、`addi-time.asm` 和 `li-time.asm`，
分别执行 move、addi 和 li 指令 1.0×10^8 次，
其中一半用于赋值 `0xFFFFFFFF`，另一半用于赋值 `0x00000000`.
用秒表测得总运行时间分别为 1 分 42 秒、1 分 41 秒和 1 分 59 秒，
`move` 指令和 `addi` 指令速度相近，而 `li` 指令要慢一些.

  assembly事实上，`move` 指令和 `li` 指令都是伪指令，前者执行时化作 `addu` 指令，
例如 `move $s1 $s0` 执行时会被转变为 `addu $17, $0, $16`;
后者执行时化为 `lui` 指令和 `ori` 指令，当要赋的值的前 16 位全为 0 时，只化为 `ori` 指令.
例如 `li $s1 0xFFFFFFFF` 将被转化为:

```
lui $1, -1
ori $17, $1, -1     # $1 = $at
```

  再例如 `li $s1 0x0000FFFF` 将被转化为 `ori $17, $0, -1`
如果将 `0xFFFFFFFF` 改成 `0x0000FFFF`，再次测试，
则总运行时间分别为 1 分 42 秒、1 分 41 秒和 1 分 42 秒，这验证了上述转换.

  综合两次实验结果，我发现 `move` 指令和 `addi` 指令的性能不随要赋的值的变化而变化，
但 `li` 指令在要赋的值的高 16 位全为 0 和不全为 0 的两种情况下表现出不同性能.
在不考虑溢出时，`addu` 指令与 `add` 指令等价，故其性能与 `addi` 相近.
因此 `move` 指令性能与 `addi` 指令相近，运行时间测试也验证了这一点.

  总体而言，`move` 指令和 `li` 指令表示赋值更加简洁，可读性较强，但灵活性不及 `addi` 指令;
由于 16 位立即数有表示范围的限制，故赋非常大的值时使用 `addi` 会比较困难，
这时可以用 `li` 指令替代.
编码时，应具体情况具体分析，采用合适的赋值方式.

- (3) `syscall` 函数的第 30 及以上的服务指令独属于 MARS，QtSPIM 不支持.
例如 `syscall(30)` 指令可以测量程序的运行时间，
`syscall(36)` 可以将寄存器中的值按无符号数形式输出，
但这两种指令在 QtSPIM 中都不能被识别.
(参考 SYSCALL functions available in MARS)

  我在计算第 46 个 Fibonacci 数时，出现了 `Arithmetic overflow` 的错误提示，
说明 `add` 指令在让第 44 和 45 个 Fibonacci 数相加时，出现了算数溢出.

```
add $a2 $a2 $a1     # second2 = first1 + second1
sub $a1 $a2 $a1     # first2 = second2 - first1 = second1
```

  我将这段代码的 `add` 改成 `addu`，将 `sub` 改成 `subu`:

```
addu $a2 $a2 $a1    # second2 = first1 + second1
subu $a1 $a2 $a1    # first2 = second2 - first1 = second1
```

则输出为 -1323752223

第 44 个 Fibonacci 数是 1134903170，第 45 个 fibonacci 数是 1836311903
由它们相加得到第 46 个 Fibonacci 数是 2971215073.
错误的结果 -1323752223 恰好等于 $2971215073–2^{32}$，这验证了算术溢出的存在.

我希望能找到让 QtSPIM 将寄存器中的数据按无符号数输出，
这样就可以正确地打印第 46 个 Fibonacci 数的值.
进一步，我希望能找到让 QtSPIM 将寄存器中的数据按十六进制或二进制形式输出，
这样我可以通过两个寄存器分别保存一个 64 位数的高位和低位，
并且定义 64 位数的加法和减法，以此来计算更高位的 Fibonacci 数，
最终的结果可以通过先后打印高位和低位的十六进制值来表示.
但不幸的是，无论是以无符号数形式打印整数的 `syscall(36)` 指令，
还是以十六进制打印整数的 `syscall(34)` 指令，QtSPIM 都不支持.

THE END