

实验1 MIPS程序设计

实验时间：第一周至第四周

实验人员：杨乙 21307130076

指导老师：王雪平

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 实验过程

2.1 调试

(注：程序运行结果仅记录有变化的寄存器，且重点对教材上没有出现的内容进行了分析)

p1.asm

程序无输出，程序运行结束后寄存器 `v0` 值为 `0xa`，`t2` 值为 `0x28`，`t3` 值为 `0x39`

分析

1. 程序中出现了 `syscall` 指令，这一指令用于系统功能调用。它接收 `v0` 寄存器的值并执行对应的功能。在本程序中出现的功能号为 `10`，控制程序退出
2. 程序中出现 `.global` 伪指令，声明了全局符号 `main` 过程；`.text` 伪指令定义了程序的代码段

p2.asm

设置 QtSpim 参数为 Simple Machine 并调试程序（否则会无法识别 `li` 指令并报错）

程序无输出，运行结束后寄存器 `v0` 值为 `0xa`，`t2` 和 `t3` 值为 `0x12340028`，`at` 值为 `0x12340000`

分析

除了 MIPS 常用汇编指令，程序引入了宏指令 `li Rd, value`，相当于直接对寄存器 `Rd` 赋值 `value`；观察 QtSpim 中的反汇编后的指令可以得到如下结论：

当 `value` 高低半字都不为0时，`li` 指令展开为如下两条常用指令：

```
1 | lui      $at, Upper 16-bits of value
2 | ori      Rd, $at, Lower 16-bits of value
```

当 `value` 仅低半字不为0时，`li` 指令替换为后一条常用指令；这一结论与上述的程序运行结果相符合在原来代码的基础上更改数值（将 `value` 分别替换为 `0x12340000` 和 `0x0`）并调试，得到如下结论：

当 `value` 仅高半字不为0时，`li` 指令替换为前一条常用指令；当 `value` 为0时，`li` 指令替换为后一条常用指令；这说明 QtSpim 在反汇编的过程中进行了一定程度的优化

p3.asm

设置参数为 Simple Machine 并调试程序（否则会无法识别 `la` 指令并报错）

程序无输出，运行结束后寄存器 `v0` 值为 `0xa`，`t2` 值为 `0x28`，`t3` 值为 `0x3b`

分析

1. 除了 MIPS 常用汇编指令，程序引入了宏指令 `la Rd, Label`，相当于直接取地址 `Label` 到寄存器 `Rd`；观察反汇编后的指令可以得到如下结论：

当 `Label` 高低半字都不为0时，`la` 指令展开为如下两条常用指令：

```
1 | lui      $at, Upper 16-bits of Label
2 | ori      Rd, $at, Lower 16-bits of Label
```

其余规则可以类比 `li` 指令

2. 观察 QtSpim 的 Data 窗口，得到 `A` 的起始地址 `0x10010000`，`h` 的地址 `0x10010040`，每个数字占四个字节

3. `.data` 伪指令定义了程序的数据段，程序变量 `A` 和 `h` 在该伪指令下定义，汇编程序会分配和初始化变量的存储空间

2.2 改写程序

主要改动如下：

系统功能调用：读入整数

为了读入用户输入的两个整数和跳转值（0或1），需要进行读入整数的系统调用。需要先用 `li` 宏指令将寄存器 `v0` 值置为5。示例如下：

```
1 |      li      $v0, 5
2 |      syscall
3 |      move     $t0, $v0      # 其余省略
```

读入的整数值存入 `v0` 寄存器，其中 `move Rd, Rs` 宏指令相当于将寄存器 `Rs` 的值复制到 `Rd`

系统功能调用：输出字符串

为了按照样例输出字符串，需要进行输出字符串的系统调用。首先需要在 `.data` 字段定义所有需要输出的字符串，示例如下：

```
1 | .data
2 |     msg1:    .asciiz "Please enter 1st number: "    # 其他定义在此省略
```

还需要在准备输出字符串时进行以下两步：用 `la` 宏指令将字符串的起始地址存入 `a0` 寄存器；将 `v0` 值置为4。示例如下：

```
1 |      la      $a0, msg1
2 |      li      $v0, 4
3 |      syscall      # 其余省略
```

系统功能调用：输出整数

在输出两数之和时还输出了用户输入的两个整数，需要进行整数输出的系统调用。需要将 `v0` 值置为1，并且用 `move` 宏指令将存有用户输入的寄存器中的值复制到 `a0` 中。具体示例如下：

```
1 |      li      $v0, 1
2 |      move     $a0, $t0
3 |      syscall      # 其余省略
```

条件分支

实现读入0或1并完成跳转的代码如下：

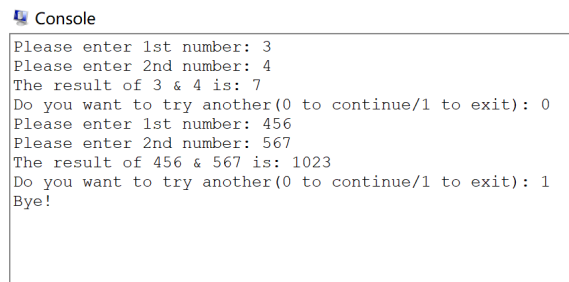
```

1 Continue:
2     # 省略中间代码
3
4     li      $v0, 5
5     syscall
6     move    $t0, $v0
7
8     beq     $t0, $0, Continue

```

`Continue` 位于 `.text` 字段起始位置。复用 `t0` 寄存器，保存用户输入的跳转值。

改写后程序的运行截图如下：



```

Console
Please enter 1st number: 3
Please enter 2nd number: 4
The result of 3 & 4 is: 7
Do you want to try another(0 to continue/1 to exit): 0
Please enter 1st number: 456
Please enter 2nd number: 567
The result of 456 & 567 is: 1023
Do you want to try another(0 to continue/1 to exit): 1
Bye!

```

2.3 把C代码翻译成MIPS代码

主要内容如下：

过程调用

过程 `sumn` 是叶子过程，因此不需要保存返回地址和帧指针，即过程对应的栈帧为空，结尾段直接用 `jr` 指令返回即可

在 `main` 函数中使用 `jal` 指令保存返回地址到寄存器 `ra` 并跳转到 `sumn` 处执行

调用 `sumn` 过程前，需要事先将 `arrs` 首地址存入 `a0` 寄存器，将寄存器 `a1` 值置为8

用 `t0`、`t1`、`t2` 作为过程中的临时变量

因为 `v0` 寄存器被系统功能调用占用，所以将返回值存入 `v1`

循环结构

循环结构实现如下：

```
1      # arrs  in $a0
2      # n      in $a1
3      # idx   in $t0
4      # sum   in $v1
5
6 Loop: beq     $a1, $t0, Exit
7       add     $t1, $t0, $t0    # idx * 2
8       add     $t1, $t1, $t1    # idx * 4
9       add     $t1, $t1, $a0
10
11      lw      $t2, 0($t1)      # $t2 = arrs[idx]
12      add     $v1, $v1, $t2    # sum += arrs[idx]
13      addi    $t0, $t0, 1      # idx++
14      j       Loop
```

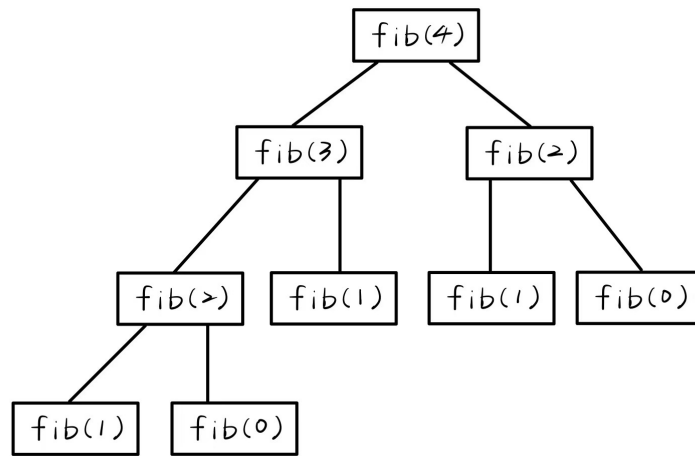
MIPS 代码运行截图如下：



2.4 优化代码

递归算法转化为循环算法

程序 `fib-op` 采用递归算法，存在重复计算的问题。以 `n = 4` 时的过程举例，画出递归调用树：



可以发现其中 `fib(3)`、`fib(2)`、`fib(1)`、`fib(0)` 都被计算了多次。随着 `n` 值增大，重复调用的次数会越来越多，严重降低程序性能。因此可以根据 Fibonacci 公式，将递归算法转化为循环算法。程序改动部分对应的代码如下：

```

1  fib:
2      bgt      $a0, 1, fib_recurse    # if (n < 2) return 1;
3      li      $v1, 1
4      jr      $ra
5
6  fib_recurse:
7      li      $t0, 1                  # int f1 = 1
8      li      $t1, 1                  # int f2 = 1
9      li      $t3, 1                  # int i = 1
10 Loop: beq    $a0, $t3, Exit          # for (i = 1; i = n; ++i){
11      add     $v1, $t0, $t1          #     sum = f1 + f2;
12      move    $t1, $t0              #     f2 = f1;
13      move    $t0, $v1              #     f1 = sum;
14      addi    $t3, $t3, 1            # }
15      j       Loop
16 Exit: jr     $ra

```

程序的时间复杂度降低到 `O(n)`；

栈帧不保存信息

优化后的 `fib` 过程是叶子过程，且过程中不需要用到任何保存寄存器。所以栈帧中不需要保存任何信息。事实上，甚至可以直接将 `fib` 过程并入 `main` 过程，去除调用过程时的 `jr` 和 `jal` 指令，以进一步减少程序执行时间

在 Linux 虚拟机上运行 `fib-o.asm` 和 `fib-op.asm`，比较程序运行时间：

优化前：

```
yangyi@LAPTOP-E0AG1RH8:/mnt/d/desktop$ echo | /bin/time spim -file fib-o.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
109460.03user 0.04system 0:00.06elapsed 111%CPU (0avgtext+0avgdata 1704maxresident)k
0inputs+0outputs (0major+454minor)pagefaults 0swaps
```

用户 CPU 时间: **0.03**

系统 CPU 时间: **0.04**

程序执行时间: **0:00.06**

优化后:

```
yangyi@LAPTOP-E0AG1RH8:/mnt/d/desktop$ echo | /bin/time spim -file fib-op.asm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
109460.00user 0.00system 0:00.00elapsed 0%CPU (0avgtext+0avgdata 1704maxresident)k
0inputs+0outputs (0major+452minor)pagefaults 0swaps
```

用户 CPU 时间: **0.00**

系统 CPU 时间: **0.00**

程序执行时间: **0:00.00**

说明程序性能显著提高

3 实验结论

MIPS 体系结构

在这一实验中, 我重点练习了使用 QtSPIM 模拟器对程序进行运行和调试; 熟悉了 MIPS 体系结构相关的内容

优化程序运行速度

对于程序运行速度的优化既可以在算法和数据结构的层面进行, 也可以在汇编语言的层面进行。前者所带来的优化程度更大, 有时会相差几个数量级; 而后者进行的优化更加细致、精准。因此程序运行速度的优化也可以从这两方面来进行: 先从程序的功能出发, 通过改善算法和数据结构来进行程序的优化; 对于时间敏感的程序, 可以在汇编语言的基础上进一步优化

4 实验感想

通过学习 MIPS 指令，我理解了程序在计算机内部的执行方式；通过对示例程序的优化，我掌握了优化程序运行速度的方法，同时也理解了多种程序语言各自的优缺点：越接近底层的语言往往性能更高，但编写困难，可读性差；封装程度越高的语言对编写者来说更加友好，可读性强，但性能一般较低。我们在开发不同类别的程序时，要根据程序的特点（如是否时间敏感）和实际应用情况选择合适的程序语言，在优化程序运行速度时要按从整体到细节的顺序进行优化