

中山大学数据科学与计算机学院本科生实验报告

(2019 学年秋季学期)

课程名称：计算机组成原理实验

任课教师：

助教：

年级&班级	19 计科超算	专业(方向)	
学号	18324034	姓名	林天皓
电话		Email	linth5@mail2.sysu.edu.cn
开始日期	2020.10.9	完成日期	2020.10.16

一、实验题目

实验 5 MIPS 实验 2

二、实验目的

本实验的目的是进一步编写 mips 汇编语言，熟悉 mips 的指令系统，同时本次实验进行了子函数的调用，需要进一步学习循环语句，子程序调用指令的汇编使用方法

三、实验内容

1 实验原理

继续通过 MARS4.5 程序解析与执行 mips 汇编程序，在练习 1 中，需要学会间接寻址的方法，修改内存中储存的值，在练习 2 中，学会通过系统调用来输入值与输出值。在练习 3 中了解如何编写具有循环结构的 mips 汇编程序。在练习 4 中，了解调用子程序的方法，学会如何传递给子程序参数，子程序结束时如何返回主程序的方法。

2. 实验步骤

练习 1：不采用循环实现。

实现代码如下：

```
.data
foo:
```

```

        .word 1,2,3,4,5
.text
main:
    la $t0,foo

    lw $t1,0($t0)
    addiu $t1,$t1,2
    sw $t1,0($t0)

    lw $t1,4($t0)
    addiu $t1,$t1,2
    sw $t1,4($t0)

    lw $t1,8($t0)
    addiu $t1,$t1,2
    sw $t1,8($t0)

    lw $t1,12($t0)
    addiu $t1,$t1,2
    sw $t1,12($t0)

    lw $t1,16($t0)
    addiu $t1,$t1,2
    sw $t1,16($t0)

exit: ori $v0, $0, 10          #System call code 10 for exit
    syscall

```

如果采用循环实现代码如下：

```

.data
foo:
    .word 1,2,3,4,5

.text
main:
    la $t0,foo
    addi $t2,$0,0
loop:
    lw $t1,0($t0)
    addiu $t1,$t1,2
    sw $t1,0($t0)

    addi $t0,$t0,4
    addi $t2,$t2,1
    bne $t2,5,loop

```

```
exit: ori $v0, $0, 10          #System call code 10 for exit
      syscall
```

其中第一条 la 指令是使得链接器链接的得到的 foo 的地址，存入 t0 寄存器中。

通过依次读取与写入内存地址中的值，来完成把 foo 中每个数字加 2 的目的。

回答下列关于 MARS 的问题.

- a.data, .word, .text 指示器 (directives) 的含义是什么(即, 在每段中放入什么内容)?
- b.在 MARS 中如何设置断点 breakpoint?
- c.在程序运行到断点处停止时, 如何继续执行? 如何单步调试代码?
- d.如何知道某个寄存器 register 的值是多少? 如何修改寄存器的值.

回答:

- a. 1.data subsequent items stored in data segment at next available address

在下一个可用地址存储在数据段中的后续项'

- 2..word Store the listed value(s) as 32 bit words on word boundary

将列出的值存储为 32 位的单词边界

- 3..text subsequent items(instructions) stored in Text segment at next available address

后续项目(指令)存储在下一个可用地址的文本段中

- b.在 Excute 选项卡中勾选指令前方的 Bkpt 选框, 即可设置断点。

- c.运行到断点后, 可以继续选择运行按钮继续运行, 直到遇到下一个断点或者程序结束才停止。点击单步执行按钮即可单步执行, 仅仅会执行一条指令。

- d.可以在右侧 Register 选项卡中查看某个寄存器的值。双击寄存器对应的值可以编辑对应寄存器的值。

练习2:

通过向 v0 寄存器中存入 5, 然后进行系统调用, 来完成从键盘输入整数的作用。

在一下的实验中, t0 和 t1 分别储存两次从键盘输入的值, 然后调用 add 指令将 a0 存储两个

输入的值之和，然后调用系统调用输出

实现代码如下：

```
.data
str1:  .ascii "Enter 2 numbers:"
str2:  .ascii "The sum is "
.text
main:
    ori $v0, $0, 4          #System call code 4 for printing a string
    la $a0, str1             #address of Str1 is in $a0
    syscall                  #print the string
    ori $v0, $0, 5          #System call code 5 for read integer? $v0 contain
s integer read
    syscall
    add $t0, $v0, $zero      #

    ori $v0, $0, 5          #System call code 5 for read integer? $v0 contain
s integer read
    syscall
    add $t1, $v0, $zero      #

    ori $v0, $0, 4          #System call code 4 for printing a string
    la $a0, str2             #address of Str2 is in $a0
    syscall                  #print the string
    ori $v0, $0, 1          #System call code 4 for print integer? $a0 = inte
ger to print
    add $a0, $t0, $t1        #calculate the sum
    syscall                  #print the sum

exit: ori $v0, $0, 10        #System call code 10 for exit
    syscall                  #print the sum
```

练习3:

本体需要通过循环的结构来实现求这个平方和。

在我的实现中，使用 t0 储存下一个要添加的数字，t2 寄存器储存平方后的值，用 t1 储存之前所有加过的平方的和，通过条件判断 t0 是否等于 101 来决定是否退出循环并输出 t1 中储存的数字计算结果

实现代码如下：

```

.data
.align 2
Str: .ascii "The sum of square from 1 to 100 is "
.text
main:
    addi $t0,$0,1
    addi $t1,$0,0
loop:
    mult $t0,$t0
    mflo $t2
    add $t1,$t1,$t2
    addi $t0,$t0,1
    bne $t0,101,loop

    ori $v0, $0, 4
    la $a0, Str
    syscall

    ori $v0, $0, 1
    add $a0,$0,$t1
    syscall

exit:
    ori $v0, $0, 10          #System call code 10 for exit
    syscall                  #print the sum

```

练习4:

方法1: 通过设定 t0 的值为 0x80000000, 即第一位为 1, 剩余的 31 位均为 0。首先判断 a0 的值是否为零, 如果为零直接输出 -1 并跳出。用 v0 表示当前位移了几位, 接下来通过将 a0 与 t0 做与运算, 如果运算得到的值为 0 则继续将 a0 左移, 同时使用用于统计位移次数的 v0+1, 继续做以上的循环, 直到做与运算得到的值大于零, 即可输出 v0 的值。

实现代码如下:

```

.text
main:
    lui $a0,0x8000
    jal first1pos
    jal printv0
    lui $a0,0x0001
    jal first1pos

```

```

jal printv0
li $a0,1
jal firstlpos
jal printv0
add $a0,$0,$0
jal firstlpos
jal printv0
li $v0,10
syscall

```

firstlpos: # your code goes here

```

addi $sp,$sp,-4
sw $ra ,0($sp)

lui $t0,0x8000
beq $a0,$0,zeroend #test if a0 equal 0

addi $v0,$0,0

loop:
    and $t3,$t0,$a0
    bnez $t3, end    # if $t3 >=0  endtarget
    addi $v0,$v0,1
    sll $a0,$a0,1
    j loop

zeroend:
    addi $v0,$0,-1
end:
    jr $ra

```

printv0:

```

addi $sp,$sp,-4
sw $ra,0($sp)
add $a0,$v0,$0
li $v0,1
syscall
li $v0,11
li $a0,'\n'
syscall
lw $ra,0($sp)
addi $sp,$sp,4
jr $ra

```

方法 2:

基本与方法 1 一致，只是在位移的时候将 a0 左移改为 t0 的右移。

实现代码如下:

```
.text
main:
    lui $a0,0x8000
    jal first1pos
    jal printv0
    lui $a0,0x0001
    jal first1pos
    jal printv0
    li $a0,1
    jal first1pos
    jal printv0
    add $a0,$0,$0
    jal first1pos
    jal printv0
    li $v0,10
    syscall

first1pos:
    addi $sp,$sp,-4
    sw $ra ,0($sp)

    lui $t0,0x8000
    beq $a0,$0,zeroend #test if equal 0

    addi $v0,$0,0
loop:
    and $t3,$t0,$a0
    bnez $t3, end    # if $t3 >=0  endtarget
    addi $v0,$v0,1
    srl $t0,$t0,1 #掩码
    j loop

zeroend:
    addi $v0,$0,-1
end:
    jr $ra

printv0:
    addi $sp,$sp,-4
```

```

sw $ra,0($sp)
add $a0,$v0,$0
li $v0,1
syscall
li $v0,11
li $a0,'\n'
syscall
lw $ra,0($sp)
addi $sp,$sp,4
jr $ra

```

其中用到了 jal 这个指令来调用子程序，将目前的 ra 指针存入内存中，退出子程序时候 sp 指针退回，但是这里的子程序并没有调用其他的子程序，ra 寄存器的值不会被破坏，所以这里如果不通过堆栈来储存返回地址也是可行的。

四、实验结果

练习1:

运行前

Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	

图 1-练习 1 运行前的数组数字

运行后:

Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	
0x10010000	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007	
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	

图 2-练习 1 运行后的数组数字

分析：5 个字的数组，运行后每个数字都加 2，达到了目的。

采用循环方式的输出结果相同，此处不再赘述。

练习2:

运行程序，输入 100 和 200 两个数字，输出为 300.

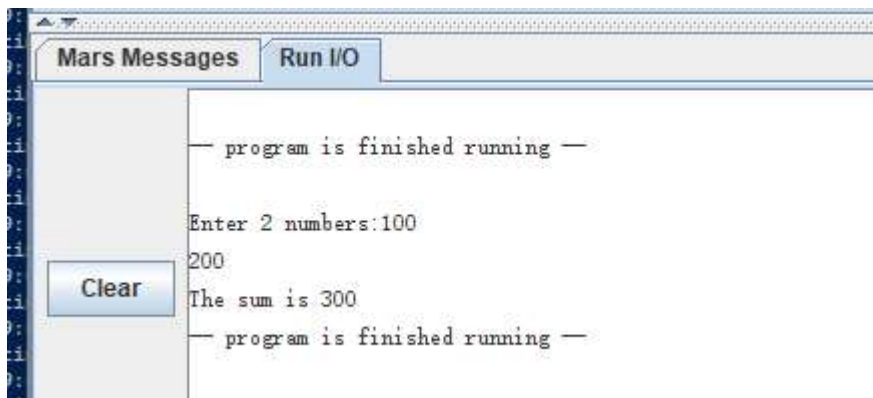


图 3-练习 2 的输出

分析：实现了输入两个数字并求和输出的任务。

练习3:

输出如下，平方和为 338350

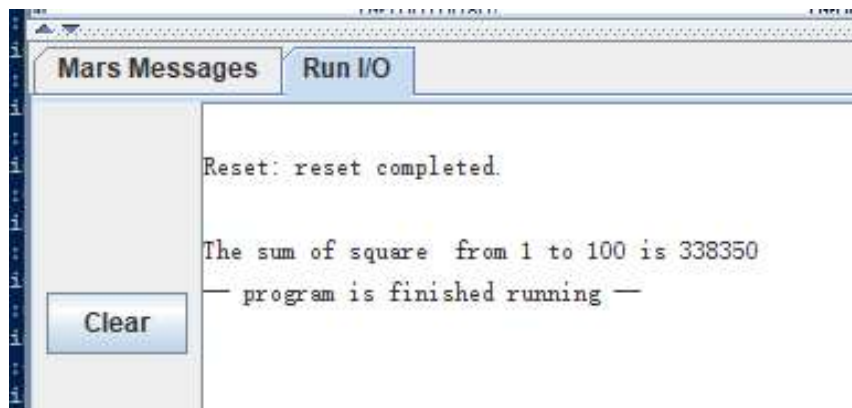


图 4-练习 3 的输出

分析：经过检验该程序计算 1-100 平方和的结果正确，完成了任务。

练习4:

方法 1:

输出结果



图 5-练习 4 的输出

方法 2 输出结果与方法 1 相同，此处不再赘述

分析：输出结果为 a0 最左边的 1 的位置，完成了练习 4 任务。

综上所述，练习 1-4 均完成了实验任务

五、实验感想

本次实验是第二次使用 mips 汇编语言编写程序。本次实验进行了子函数的调用，需要进一步学习了循环语句，子程序调用指令的汇编使用方法。例如在练习 3 中调用 jal firstpos 时，在子程序中把返回值 ra 记录到堆栈中。同时还尝试了一些编译器从 C++ 语言生成的汇编代

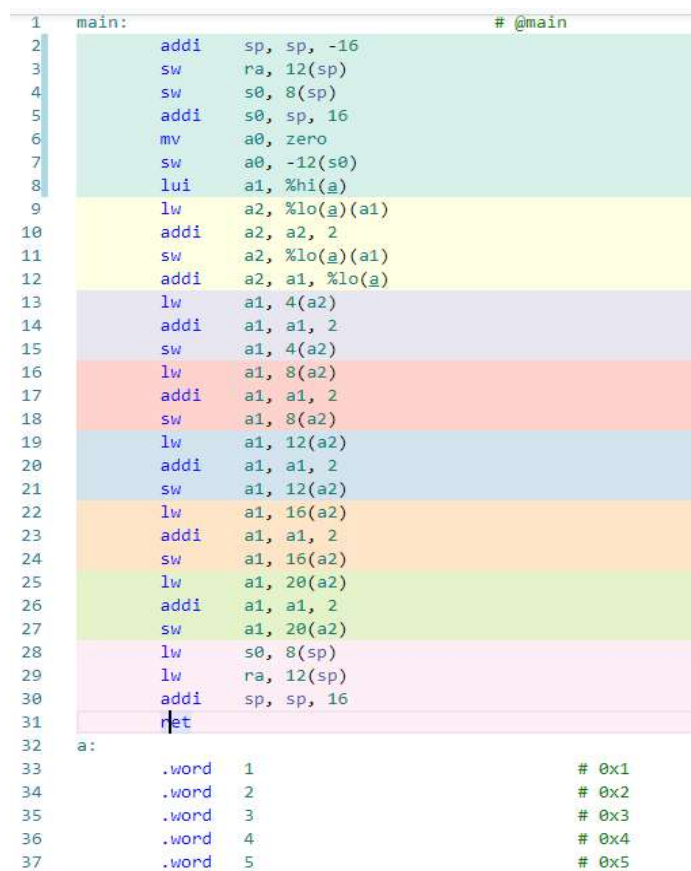
码。例如对于练习 1:



```
1 // Type your code here, or load an example.
2 int a[5]={1,2,3,4,5};
3 int main(){
4     a[0]+=2;
5     a[1]+=2;
6     a[2]+=2;
7     a[3]+=2;
8     a[4]+=2;
9     a[5]+=2;
10 }
```

图 6-c++语言版本的练习 1

由于该网站: gcc.godbolt.org上没有选择编译生成mips汇编代码的交叉编译器, 所以使用与mips非常相似的rsic-v指令集生成的汇编代码如下:



```
1 main:                                     # @main
2     addi    sp, sp, -16
3     sw      ra, 12(sp)
4     sw      s0, 8(sp)
5     addi    s0, sp, 16
6     mv      a0, zero
7     sw      a0, -12(s0)
8     lui     a1, %hi(a)
9     lw      a2, %lo(a)(a1)
10    addi    a2, a2, 2
11    sw      a2, %lo(a)(a1)
12    addi    a2, a1, %lo(a)
13    lw      a1, 4(a2)
14    addi    a1, a1, 2
15    sw      a1, 4(a2)
16    lw      a1, 8(a2)
17    addi    a1, a1, 2
18    sw      a1, 8(a2)
19    lw      a1, 12(a2)
20    addi    a1, a1, 2
21    sw      a1, 12(a2)
22    lw      a1, 16(a2)
23    addi    a1, a1, 2
24    sw      a1, 16(a2)
25    lw      a1, 20(a2)
26    addi    a1, a1, 2
27    sw      a1, 20(a2)
28    lw      s0, 8(sp)
29    lw      ra, 12(sp)
30    addi    sp, sp, 16
31    ret
32
33 a:
34     .word   1                # 0x1
35     .word   2                # 0x2
36     .word   3                # 0x3
37     .word   4                # 0x4
38     .word   5                # 0x5
```

图 7-由 rsic-v 交叉编译器编译生成的 rsic-v 汇编代码

先通过编写我们所熟悉的C++代码，通过编译器程序自动的生成汇编代码，有助于我们开始编写复杂的汇编代码，也有助于我们理解其他指令集的代码，更全面的理解计算机体系结构。

附录（流程图，注释过的代码）：

练习1流程图



图 8-练习 1 流程图

练习2流程图



图 1-练习 2 流程图

练习3流程图

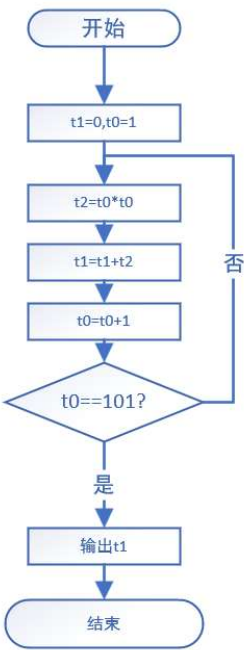


图 1-练习 3 流程图

练习4流程图

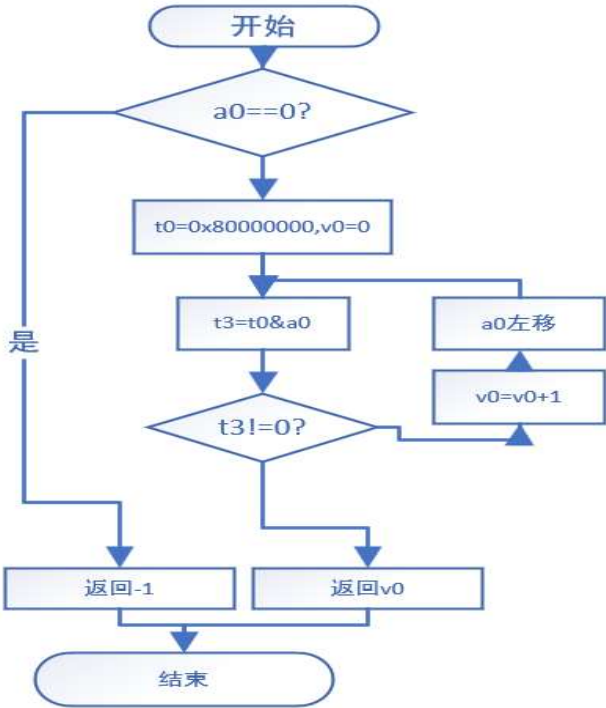


图 1-练习 4 流程图

本次实验涉及到自己修改过的代码均在上文出现，此处不再重复。