



成绩

# 中国农业大学

## 课程设计

(2023 -2024 学年夏季学期)

题    目： 计算机系统综合训练-任务 3

课程名称： 计算机系统工程综合实践

任课教师： 王耀君, 黄岚, 段青玲, 史银雪

班    级： 计算机 211

学    号： 2020301010225

姓    名： 夏婉可

开始时间	<u>2024/09/06</u>	完成时间	<u>2024/09/09</u>	报告成绩	
所在组序号	<u>10 组</u>	个人完成			
本组成员情况					
姓 名	学 号	承担的任务			
夏婉可	<u>2020301010225</u>	<u>Pcode 和 MIPS 的映射、PL/O 程序映射后的 MIPS 仿真</u>			

**（课程设计报告包含以下内容：）**

一、P-code 语言与 MIPS 汇编的映射程序实现（30 分）

1. 制作类 P-code 语言和 32 位 MIPS 汇编语言各指令的映射关系表；
2. 实现 P-code 语言与 32 位 MIPS 汇编语言的映射程序；

二、PL/O 程序在 MIPS 架构上运行仿真（50 分）

1. 编写不少于三个 PL/O 程序
2. 对于每个 PL/O 程序编译成 MIPS 程序在 Spim 中的运行结果

**（三个测试用例的运行结果截图和结果说明，并附：测试用例 PL/O 源码、MIPS 程序）**

三、实验心得与体会（10 分）**（独立完成）**

1. 遇到的困难
2. 如何解决困难

四、参考文献（10 分）**（独立完成）**

**（参考的博客、网站、图书、技术手册和论文等）**

# 计算机系统工程综合实践 —— 任务 2（编译原理）

## 一：P-code 语言与 MIPS 汇编的映射程序实现

### （1）制作类 P-code 语言和 32 位 MIPS 汇编语言各指令的映射关系表

P-code 指令	逻辑关系	核心 MIPS 指令
Opr 0	调用返回	add \$sp, \$zero, \$fp lw \$ra, -8(\$sp) lw \$fp, -4(\$sp) jr \$ra
Opr 1	取反	neg \$t0, \$t0
Opr 2	加法	add \$t0, \$t0, \$t1
Opr 3	减法	sub \$t0, \$t0, \$t1
Opr 4	乘法	mul \$t0, \$t0, \$t1
Opr 5	除法	div \$t0, \$t0, \$t1
Opr 6	奇数偶数判断	andi \$t0, \$t0, 1
Opr 8	等于判断	sub \$t0, \$t1, \$t0 sltu \$t0, \$zero, \$t0 xori \$t0, \$t0, 1
Opr 9	不等于判断	sub \$t0, \$t1, \$t0 sltu \$t0, \$zero, \$t0
Opr 10	小于判断	slt \$t0, \$t1, \$t0
Opr 11	不小于判断	slt \$t0, \$t1, \$t0 xori \$t0, \$t0, 1
Opr 12	大于判断	slt \$t0, \$t0, \$t1 xori \$t0, \$t0, 1
Opr 13	不大于判断	slt \$t0, \$t0, \$t1 xori \$t0, \$t0, 1
Opr 14	输出栈顶值	li \$v0, 1 lw \$a0, 0(\$sp)

Opr 15	输出换行	li \$v0, 11 li \$a0, 10
Opr 16	输入值到栈顶	li \$v0, 11 la \$a0, '?'
Lit	取值到栈顶	li \$t0, {y}
Lod	加载内存值到栈顶	addi \$t0, \$v0, -{offset}
Sto	保存栈顶值到内存	addi \$t0, \$v0, -{offset}
Cal	调用子程序	/
Int	分配内存	addi \$sp, \$sp, -{offset}
Jump	直接跳转	j _{y}
Jpc	条件跳转	lw \$t0, 0(\$sp) beq \$t0, \$zero, _{y}

P-code 语言和 32 位 MIPS 汇编语言的映射关系表，如上表所示。

其中，

## (2) 实现 P-code 语言与 32 位 MIPS 汇编语言的映射程序

本实验通过 Python 语言撰写 P-code 语言和 32 位 MIPS 汇编语言的映射程序，如下表所示。

```
# embedding func for pcode to mips
def pcode_to_mips():
    # head codes
    mips = [
        ".data",
        "stack: .space 16384",    # define the max space for stack
        ".text",
        "main:",                # main func

        "la $fp, stack",
        "addi $fp, $fp, 16384",
        "add $sp, $zero, $fp",

        "la $t0, end",
        "sw $sp, 0($sp)",
        "sw $sp, -4($sp)",
        "sw $t0, -8($sp)",
        "addi $sp, $sp, -12"
```

```
]
```

```
# open the pcode file, which is fa.tmp
with open(r'C:\Users\86158\Desktop\PL0c\fa.tmp') as program:
    # go thru every line in the tmp file
    for instruction in program.readlines():
        parts = instruction.strip().split()
        # instruction = num + command + x + y
        # eg: 1 int 0 7
        # split each instruction into different variables
        num = parts[0]
        command = parts[1]
        x = parts[2]
        y = parts[3]

        # set jump point as num
        mips.append(f"_{num}:")

        # set stack movement
        # ATTENTION: _read2 && base is perplex commands

        # 1: basic operations for stack-pointer
        stack_up = "addi $sp, $sp, 4"
        stack_down = "addi $sp, $sp, -4"

        # 2: read 1 figure, hold the stack-pointer
        _read1 = "lw $t0, 0($sp)"      # load to t0
        read1_ = "sw $t0, 0($sp)"     # load back

        # 3: read 2 figures, move the stack-pointer
        _read2 = [
            stack_up,
            "lw $t0, 0($sp)",
            "lw $t1, 4($sp)"
        ]
        read2_ = "sw $t0, 4($sp)"

        # 4: get base level
        base = [
            f"addi $a0, $zero, {x}",
            "add $a1, $zero, $fp",
            "jal base"
        ]

        # judge command
        if command == 'opr':
```

```

# return
if y == '0':
    mips.append("add $sp, $zero, $fp")
    mips.append("lw $ra, -8($sp)")
    mips.append("lw $fp, -4($sp)")
    mips.append("jr $ra")

# neg
elif y == '1':
    mips.append(_read1)
    mips.append("neg $t0, $t0")      # neg
    mips.append(read1_)

# add
elif y == '2':
    for code in _read2:
        mips.append(code)
    mips.append("add $t0, $t0, $t1")# add, store at t0
    mips.append(read2_)

# sub
elif y == '3':
    for code in _read2:
        mips.append(code)
    mips.append("sub $t0, $t0, $t1")# sub, store at t0
    mips.append(read2_)

# mul
elif y == '4':
    for code in _read2:
        mips.append(code)
    mips.append("mul $t0, $t0, $t1")# mul, store at t0
    mips.append(read2_)

# div
elif y == '5':
    for code in _read2:
        mips.append(code)
    mips.append("div $t0, $t0, $t1")# div, store at t0
    mips.append("mflo $t0")        # remove the remaining
    mips.append(read2_)

# if singular?
elif y == '6':
    mips.append(read1_)
    mips.append("andi $t0, $t0, 1") # and, store at t0

```

```

        mips.append(_read1)

# y == '7' is not defined

# equal
elif y == '8':
    for code in _read2:
        mips.append(code)

        mips.append("sub $t0, $t1, $t0")
        mips.append("sltu $t0, $zero, $t0")
        mips.append("xori $t0, $t0, 1")

    mips.append(read2_)

# not equal
elif y == '9':
    for code in _read2:
        mips.append(code)

        mips.append("sub $t0, $t1, $t0")
        mips.append("sltu $t0, $zero, $t0")

    mips.append(read2_)

# less than
elif y == '10':
    for code in _read2:
        mips.append(code)

        mips.append("slt $t0, $t1, $t0")

    mips.append(read2_)

# less equal than
elif y == '11':
    for code in _read2:
        mips.append(code)

        mips.append("slt $t0, $t1, $t0")
        mips.append("xori $t0, $t0, 1")

    mips.append(read2_)

# greater than
elif y == '12':

```

```

        for code in _read2:
            mips.append(code)

        mips.append("slt $t0, $t0, $t1")
        mips.append("xori $t0, $t0, 1")

        mips.append(read2_)

# greater equal than
elif y == '13':
    for code in _read2:
        mips.append(code)

    mips.append("slt $t0, $t0, $t1")
    mips.append("xori $t0, $t0, 1")

    mips.append(read2_)

# output stack_up
elif y == '14':
    mips.append("li $v0, 1")           # output integer, using func 1
    mips.append(stack_up)
    mips.append("lw $a0, 0($sp)")     # load sp to a0
    mips.append("syscall")           # call system

# change line
elif y == '15':
    mips.append("li $v0, 11")         # output character, using func 11
    mips.append("li $a0, 10")         # load change-line ASCII to a0
    mips.append("syscall")           # call system

# input
elif y == '16':
    # show ?
    mips.append("li $v0, 11")
    mips.append("la $a0, '?'")
    mips.append("syscall")           # call system

    mips.append("li $v0, 5")          # read integer, using func 5
    mips.append("syscall")           # call system
    mips.append("sw $v0, 0($sp)")     # load to stack_up
    mips.append(stack_down)          # stack_down

# load to stack-top
elif command == 'lit':
    mips.append(f"li $t0, {y}")      # put y into t0

```



```

        mips.append("sw $t0, 0($sp)")      # load to stack_up
        mips.append(stack_down)           # stack_down

# direct jump
elif command == 'jmp':
    mips.append(f"j {_y}")                # jump to [_y]

# conditional jump
elif command == 'jpc':
    mips.append(stack_up)
    mips.append("lw $t0, 0($sp)")         # load to t0
    mips.append(f"beq $t0, $zero, {_y}")  # if t0 = 0, then jump to [_y]

# allocate memory
elif command == 'int':
    offset = int(y) * 4
    mips.append(f"addi $sp, $sp, -{offset}") # stack down y-step

# call sub-process
elif command == 'cal':
    for baseline in base:
        mips.append(baseline)
    number = int(num) + 1
    mips.append(f"la $t0, {_number}")
    mips.append("sw $v0, 0($sp)")
    mips.append("sw $fp, -4($sp)")
    mips.append("sw $t0, -8($sp)")
    mips.append("add $fp, $sp, $zero")
    mips.append(f"j {_y}")

# store stack top to variable
elif command == 'sto':
    for baseline in base:
        mips.append(baseline)
    mips.append(stack_up)
    offset = int(y) * 4
    mips.append(f"addi $t0, $v0, -{offset}")
    mips.append("lw $t1, 0($sp)")
    mips.append("sw $t1, 0($t0)")

# load variable to stack top
elif command == 'lod':
    for baseline in base:
        mips.append(baseline)
    offset = int(y) * 4
    mips.append(f"addi $t0, $v0, -{offset}")

```

```

        mips.append("lw $t0, 0($t0)")
        mips.append("sw $t0, 0($sp)")
        mips.append(stack_down)

    # error alert
    else:
        print(f"unknown command: {instruction}")

    mips.append("")

    return mips

# apply the embedding func
mips = pcode_to_mips()

# end codes
end = [
    "end:",
    "li $v0, 10",
    "syscall",
    ""
]

# base codes
base_end = [
    "base:",
    "beqz $a0, baseend",
    "baseloop:",
    "lw $a1, 0($a1)",
    "addi $a0, $a0, -1",
    "bgtz $a0, baseloop",
    "baseend:",
    "add $v0, $a1, $zero",
    "jr $ra",
    ""
]

for endline in end:
    mips.append(endline)

for baseendline in base_end:
    mips.append(baseendline)

# output mips in the terminal
for line in mips:
    print(line)

```

```
# if you'd like to save the converted codes as an asm file,  
# you need to follow the command below:  
# -----  
# py pcode2mips.py | out-file "output.asm" -encoding ascii  
# -----  
# then there should be an asm file on the same path of this python file
```

上述程序主要包括三个部分：pcode\_to\_mips 映射函数的构建、调用映射函数进行转换、mips 结尾汇编代码的添加。

## 二：PL/0 程序在 MIPS 架构上运行仿真

### （1）斐波那契数列

斐波那契数列的 p10 代码，如下表所示。

```
const n=20;  
var f1,f2,f,i;  
begin  
    f1:=1;  
    f2:=1;  
    i:=1;  
    while i<=n do  
        begin  
            write(f1);  
            f:=f1+f2;  
            f1:=f2;  
            f2:=f;  
            i:=i+1  
        end  
    end.  
end.
```

斐波那契数列的 tmp 代码，如下表所示。

```
1 int 0 7  
2 lit 0 1
```

```
3 sto 0 3
4 lit 0 1
5 sto 0 4
6 lit 0 1
7 sto 0 6
8 lod 0 6
9 lit 0 20
10 opr 0 13
11 jpc 0 28
12 lod 0 3
13 opr 0 14
14 opr 0 15
15 lod 0 3
16 lod 0 4
17 opr 0 2
18 sto 0 5
19 lod 0 4
20 sto 0 3
21 lod 0 5
22 sto 0 4
23 lod 0 6
24 lit 0 1
25 opr 0 2
26 sto 0 6
27 jmp 0 8
28 opr 0 0
```

C 语言转换程序的执行结果，如下图所示。

```

start pl0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
-----
Process exited after 16.17 seconds with return value 0
请按任意键继续. . .

```

Python 转换程序输出的 mips 汇编语言代码结果，如下表所示。

```

.data
stack: .space 16384
.text
main:
la $fp, stack
addi $fp, $fp, 16384
add $sp, $zero, $fp
la $t0, end
sw $sp, 0($sp)
sw $sp, -4($sp)
sw $t0, -8($sp)
addi $sp, $sp, -12
_1:
addi $sp, $sp, -28
_2:

```

```

li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4

_3:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -12
lw $t1, 0($sp)
sw $t1, 0($t0)

_4:
li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4

_5:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -16
lw $t1, 0($sp)
sw $t1, 0($t0)

_6:
li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4

```

```
_7:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -24
lw $t1, 0($sp)
sw $t1, 0($t0)
```

```
_8:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -24
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_9:
li $t0, 20
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_10:
addi $sp, $sp, 4
lw $t0, 0($sp)
lw $t1, 4($sp)
slt $t0, $t0, $t1
xori $t0, $t0, 1
sw $t0, 4($sp)
```

```
_11:
addi $sp, $sp, 4
lw $t0, 0($sp)
beq $t0, $zero, _28
```

```
_12:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -12
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_13:
li $v0, 1
addi $sp, $sp, 4
lw $a0, 0($sp)
syscall
```

```
_14:
li $v0, 11
li $a0, 10
syscall
```

```
_15:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -12
```



```
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_16:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -16
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_17:
addi $sp, $sp, 4
lw $t0, 0($sp)
lw $t1, 4($sp)
add $t0, $t0, $t1
sw $t0, 4($sp)

_18:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -20
lw $t1, 0($sp)
sw $t1, 0($t0)

_19:
addi $a0, $zero, 0
```

```
add $a1, $zero, $fp
jal base
addi $t0, $v0, -16
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_20:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -12
lw $t1, 0($sp)
sw $t1, 0($t0)

_21:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -20
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_22:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -16
```

```
lw $t1, 0($sp)
sw $t1, 0($t0)

_23:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -24
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_24:
li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4

_25:
addi $sp, $sp, 4
lw $t0, 0($sp)
lw $t1, 4($sp)
add $t0, $t0, $t1
sw $t0, 4($sp)

_26:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -24
lw $t1, 0($sp)
```

```
sw $t1, 0($t0)

_27:
j _8

_28:
add $sp, $zero, $fp
lw $ra, -8($sp)
lw $fp, -4($sp)
jr $ra

end:
li $v0, 10
syscall

base:
beqz $a0, baseend
baseloop:
lw $a1, 0($a1)
addi $a0, $a0, -1
bgtz $a0, baseloop
baseend:
add $v0, $a1, $zero
jr $ra
```

上述转换后的 asm 文件在 QtSpim 上的运行结果，如下图所示。

可以发现，spim 控制台输出了斐波那契数列中的前 20 位数字，结果符合代码预期。



```
Console
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

## (2) 带用户输入的两数之和

带用户输入的两数之和的 p10 代码，如下表所示。

```
var a, b, sum;
begin
  read(a, b);
  sum := a + b;
  write(sum)
end.
```

带用户输入的两数之和的 tmp 代码，如下表所示。

```
1 int 0 6
2 opr 0 16
3 sto 0 3
4 opr 0 16
5 sto 0 4
6 lod 0 3
7 lod 0 4
8 opr 0 2
9 sto 0 5
10 lod 0 5
```

```
11 opr 0 14
12 opr 0 15
13 opr 0 0
```

C 语言转换程序的执行结果，如下图所示。

```
start pl0
?10
?110
120
-----
Process exited after 11.17 seconds with return value 0
请按任意键继续. . .
```

Python 转换程序输出的 mips 汇编语言代码结果，如下表所示。

```
.data
stack: .space 16384
.text
main:
la $fp, stack
addi $fp, $fp, 16384
add $sp, $zero, $fp
la $t0, end
sw $sp, 0($sp)
sw $sp, -4($sp)
sw $t0, -8($sp)
addi $sp, $sp, -12
_1:
addi $sp, $sp, -24

_2:
li $v0, 11
la $a0, '?'
syscall
li $v0, 5
```

```

syscall
sw $v0, 0($sp)
addi $sp, $sp, -4

_3:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -12
lw $t1, 0($sp)
sw $t1, 0($t0)

_4:
li $v0, 11
la $a0, '?'
syscall
li $v0, 5
syscall
sw $v0, 0($sp)
addi $sp, $sp, -4

_5:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -16
lw $t1, 0($sp)
sw $t1, 0($t0)

```

```
_6:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -12
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_7:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -16
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_8:
addi $sp, $sp, 4
lw $t0, 0($sp)
lw $t1, 4($sp)
add $t0, $t0, $t1
sw $t0, 4($sp)
```

```
_9:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -20
```



```
lw $t1, 0($sp)
sw $t1, 0($t0)

_10:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -20
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_11:
li $v0, 1
addi $sp, $sp, 4
lw $a0, 0($sp)
syscall

_12:
li $v0, 11
li $a0, 10
syscall

_13:
add $sp, $zero, $fp
lw $ra, -8($sp)
lw $fp, -4($sp)
jr $ra

end:
li $v0, 10
```

```

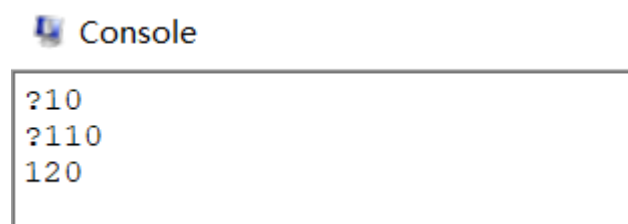
syscall

base:
beqz $a0, baseend
baseloop:
lw $a1, 0($a1)
addi $a0, $a0, -1
bgtz $a0, baseloop
baseend:
add $v0, $a1, $zero
jr $ra

```

上述转换后的 asm 文件在 QtSpim 上的运行结果，如下图所示。

可以发现，spim 控制台的前两行首先输出了问号？，提示用户输入变量 A 和 B。用户输入完毕后，自动计算出两数之和并输出，结果符合代码预期。



### (3) 求数列中的最小值

求数列中的最小值的 p10 代码，如下表所示。

```

const n = 5;
var min, i, current;

begin
    read(current);
    min := current;
    i := 1;
    while i < n do
        begin

```

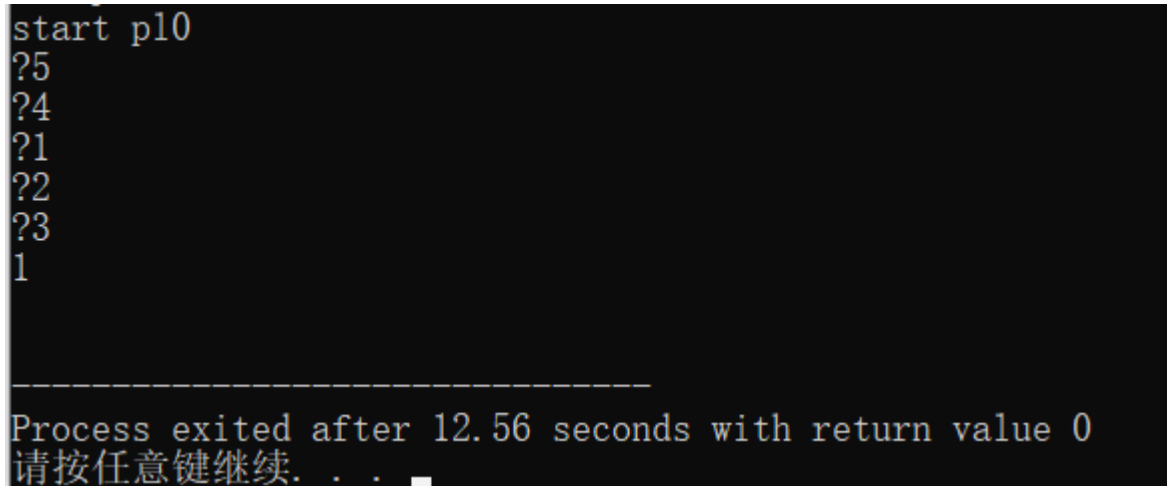
```
    read(current);  
    if current < min then  
        min := current;  
        i := i + 1  
    end;  
    write(min)  
end.
```

求数列中的最小值的 tmp 代码，如下表所示。

```
1 int 0 6  
2 opr 0 16  
3 sto 0 5  
4 lod 0 5  
5 sto 0 3  
6 lit 0 1  
7 sto 0 4  
8 lod 0 4  
9 lit 0 5  
10 opr 0 10  
11 jpc 0 25  
12 opr 0 16  
13 sto 0 5  
14 lod 0 5  
15 lod 0 3  
16 opr 0 10  
17 jpc 0 20  
18 lod 0 5  
19 sto 0 3  
20 lod 0 4  
21 lit 0 1  
22 opr 0 2  
23 sto 0 4
```

```
24 jmp 0 8
25 lod 0 3
26 opr 0 14
27 opr 0 15
28 opr 0 0
```

C 语言转换程序的执行结果，如下图所示。



```
start pl0
?5
?4
?1
?2
?3
1
-----
Process exited after 12.56 seconds with return value 0
请按任意键继续. . .
```

Python 转换程序输出的 mips 汇编语言代码结果，如下表所示。

```
.data
stack: .space 16384
.text
main:
la $fp, stack
addi $fp, $fp, 16384
add $sp, $zero, $fp
la $t0, end
sw $sp, 0($sp)
sw $sp, -4($sp)
sw $t0, -8($sp)
addi $sp, $sp, -12
_l:
addi $sp, $sp, -24
```

```

_2:
li $v0, 11
la $a0, '?'
syscall
li $v0, 5
syscall
sw $v0, 0($sp)
addi $sp, $sp, -4

_3:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -20
lw $t1, 0($sp)
sw $t1, 0($t0)

_4:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -20
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4

_5:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base

```

```
addi $sp, $sp, 4
addi $t0, $v0, -12
lw $t1, 0($sp)
sw $t1, 0($t0)
```

```
_6:
li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_7:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -16
lw $t1, 0($sp)
sw $t1, 0($t0)
```

```
_8:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -16
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_9:
li $t0, 5
sw $t0, 0($sp)
```

```
addi $sp, $sp, -4
```

```
_10:
```

```
addi $sp, $sp, 4
```

```
lw $t0, 0($sp)
```

```
lw $t1, 4($sp)
```

```
slt $t0, $t1, $t0
```

```
sw $t0, 4($sp)
```

```
_11:
```

```
addi $sp, $sp, 4
```

```
lw $t0, 0($sp)
```

```
beq $t0, $zero, _25
```

```
_12:
```

```
li $v0, 11
```

```
la $a0, '?'
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
sw $v0, 0($sp)
```

```
addi $sp, $sp, -4
```

```
_13:
```

```
addi $a0, $zero, 0
```

```
add $a1, $zero, $fp
```

```
jal base
```

```
addi $sp, $sp, 4
```

```
addi $t0, $v0, -20
```

```
lw $t1, 0($sp)
```

```
sw $t1, 0($t0)
```

```
_14:  
addi $a0, $zero, 0  
add $a1, $zero, $fp  
jal base  
addi $t0, $v0, -20  
lw $t0, 0($t0)  
sw $t0, 0($sp)  
addi $sp, $sp, -4
```

```
_15:  
addi $a0, $zero, 0  
add $a1, $zero, $fp  
jal base  
addi $t0, $v0, -12  
lw $t0, 0($t0)  
sw $t0, 0($sp)  
addi $sp, $sp, -4
```

```
_16:  
addi $sp, $sp, 4  
lw $t0, 0($sp)  
lw $t1, 4($sp)  
slt $t0, $t1, $t0  
sw $t0, 4($sp)
```

```
_17:  
addi $sp, $sp, 4  
lw $t0, 0($sp)  
beq $t0, $zero, _20
```



```
_18:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -20
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_19:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -12
lw $t1, 0($sp)
sw $t1, 0($t0)
```

```
_20:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -16
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_21:
li $t0, 1
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_22:
addi $sp, $sp, 4
lw $t0, 0($sp)
lw $t1, 4($sp)
add $t0, $t0, $t1
sw $t0, 4($sp)
```

```
_23:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $sp, $sp, 4
addi $t0, $v0, -16
lw $t1, 0($sp)
sw $t1, 0($t0)
```

```
_24:
j _8
```

```
_25:
addi $a0, $zero, 0
add $a1, $zero, $fp
jal base
addi $t0, $v0, -12
lw $t0, 0($t0)
sw $t0, 0($sp)
addi $sp, $sp, -4
```

```
_26:
li $v0, 1
```

```

addi $sp, $sp, 4
lw $a0, 0($sp)
syscall

_27:
li $v0, 11
li $a0, 10
syscall

_28:
add $sp, $zero, $fp
lw $ra, -8($sp)
lw $fp, -4($sp)
jr $ra

end:
li $v0, 10
syscall

base:
beqz $a0, baseend
baseloop:
lw $a1, 0($a1)
addi $a0, $a0, -1
bgtz $a0, baseloop
baseend:
add $v0, $a1, $zero
jr $ra

```

上述转换后的 asm 文件在 QtSpim 上的运行结果，如下图所示。

可以发现，spim 控制台的前五行首先输出了问号？，提示用户输入数列中的变量。用户输入完毕后，自动计算出数列中的最小值并输出，结果符合代码预期。

Console

```
?5
?4
?1
?2
?3
1
```

### 三：实验心得与体会

1: P-code 指令的实习流程，如下表所示。

P-code 指令	逻辑关系	实现流程
Opr 0	调用返回	取 3 个之前存的指针到对应的值
Opr 1	取反	栈顶值 存入 寄存器 t0 寄存器 t0 值 取反 寄存器 t0 值 存入 栈顶
Opr 2	加法	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算加法结果后 存入 寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 3	减法	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算减法结果后 存入 寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 4	乘法	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算乘法结果后 存入 寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 5	除法	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算除法且舍弃余数的结果后 存入 寄存器 t0

		寄存器 t0 值 存入 栈顶
Opr 6	奇数偶数判断	栈顶值 存入 寄存器 t0 寄存器 t0 值 和 1 作与运算 寄存器 t0 值 存入 栈顶
Opr 8	等于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算 t0-t1 结果后 存入 寄存器 t0 判断寄存器 t0 值是否小于 0 结果存入寄存器 t0 寄存器 t0 值 和 1 作异或运算 寄存器 t0 值 存入 栈顶
Opr 9	不等于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 计算 t0-t1 结果后 存入 寄存器 t0 判断寄存器 t0 值是否小于 0 结果存入寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 10	小于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 判断寄存器 t1 值是否小于寄存器 t0 值 结果存入寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 11	不小于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 判断寄存器 t1 值是否小于寄存器 t0 值 结果存入寄存器 t0 寄存器 t0 值 和 1 作异或运算 寄存器 t0 值 存入 栈顶
Opr 12	大于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移 判断寄存器 t0 值是否小于寄存器 t1 值 结果存入寄存器 t0 寄存器 t0 值 存入 栈顶
Opr 13	不大于判断	栈顶值和次栈顶值 存入 寄存器 t0 和 t1 栈顶指针偏移

		判断寄存器 t0 值是否小于寄存器 t1 值 结果存入寄存器 t0 寄存器 t0 值 和 1 作异或运算 寄存器 t0 值 存入 栈顶
Opr 14	输出栈顶值	栈顶指针偏移 栈顶值 存入 寄存器 a0 系统调用功能 1
Opr 15	输出换行	换行的 ASCII 存入 寄存器 a0 系统调用功能 11
Opr 16	输入值到栈顶	系统调用功能 5 寄存器 v0 值 存入 栈顶 栈顶指针偏移
Lit	取值到栈顶	需要的值 存入 寄存器 t0 寄存器 t0 值 存入 栈顶 栈顶指针偏移
Lod	加载内存值到 栈顶	基地址 存入 寄存器 v0 寄存器 t0 值 = $v0 - step * 4$ 寄存器 t0 值所对应的地址的值 存入 栈顶 栈顶指针偏移
Sto	保存栈顶值到 内存	基地址 存入 寄存器 v0 寄存器 t0 值 = $v0 - step * 4$ 栈顶指针偏移 栈顶值 存入 寄存器 t0 值所对应的地址
Cal	调用子程序	/
Int	分配内存	栈顶指针偏移
Jmp	直接跳转	直接跳转到相应地址
Jpc	条件跳转	栈顶指针偏移 栈顶值 存入 寄存器 t0 如果寄存器 t0 值=0 跳转相应地址

2: 问题: 转换程序不方便调试, 跳转设置存在问题。

解决方法: 采用一开始的 tmp 程序中的行号, 前面加上下划线组成跳转点。

3: 问题: P-code 的实现流程有错。

解决方法：对比源压缩包中的 cpp 程序代码进行调试。

## 四：参考文献

1: [https://github.com/gdut-yy/PL0\\_Cpp/tree/master/PL0\\_Cpp/Samples](https://github.com/gdut-yy/PL0_Cpp/tree/master/PL0_Cpp/Samples)

2: <https://github.com/warkul23/PL0-to-MIPS>

3: 《编译原理》第 1 版. pdf。

4: 计算机组成课程设计手册 2.0 版. pdf。

5: MIPS32-31 条核心指令用法. pdf。

6

:

[https://blog.csdn.net/qq\\_44639125/article/details/120936376?spm=1001.2014.3001.5506](https://blog.csdn.net/qq_44639125/article/details/120936376?spm=1001.2014.3001.5506)