

RISC-V 汇编语言 作业与实验

目标

- ◆ 熟悉使用 Venus 模拟器
- ◆ 练习运行和调试 RISC-V 汇编代码。
- ◆ 编写 RISC-V 函数。
- ◆ 理解嵌套函数调用、指针、数组、链表在汇编语言中的实现
- ◆ 理解变址寻址方式的多种用途

RISC-V 模拟器简介

汇编语言是接近机器代码的低级语言。因为您的计算机只能运行（x86 或 ARM）的机器代码，无法直接在您的机器上直接执行 RISC-V 代码。所以使用 RISC-V 模拟器 [Venus](#)。你可以通过查看 Venus [参考手册](#) 了解它的使用方法。

热身练习 1：熟悉 Venus

用任何一个编辑器打开 `ex1.s` 文件，观察文件内容我们会发现：标号后面有一个冒号(:)，注释以井号(#)开始，每行只能写一条指令。程序最开始部分是 `main` 函数的语句；程序结束(`exit`)时，将参数值设置为 10，并调用 `ecall` 指令。通过设置不同的参数并调用 `ecall`，可以实现各种系统调用。例如：输出一些内容到显示器终端、读文件、写文件、创建子进程等，都是系统调用。

- ◆ 直接将 `ex1.s` 从本地计算机复制/粘贴到 Venus 编辑器中。
- ◆ 单击“Simulator”选项卡，然后单击“Assemble & Simulate from Editor”按钮。模拟器就准备好要执行的代码了。如果单击“Editor”选项卡，模拟将重置。
- ◆ 在模拟器中，要执行下一条指令，请单击“step”按钮。
- ◆ 要撤消指令，请单击“prev”按钮。
- ◆ 要运行程序直至完成，请单击“run”按钮。
- ◆ 要重置程序，请单击“reset”按钮。
- ◆ 所有 32 个寄存器的内容都在右侧，控制台输出在底部。
- ◆ 要查看内存内容，请单击右侧的“Memory”选项卡。您可以使用底部的下拉菜单导航到内存的不同部分。

思考以下问题（注意：不需要在作业中回答）：

- 1) `.data`, `.word`, `.text` 指令的含义是什么？（即：它们的用途是什么？）
`.data` 表示后续内容从 `data` 段的下一个可用地址开始存储。
`.text` 表示后续内容从 `text` 段（指令段）的下一个可用地址开始存储。
`.word` 表示后续内容占用一个字（32bit）的空间。
- 2) 运行程序直到完成。程序输出了什么数字？如果 0 是第 0 个斐波那契数，那么这是第几个斐波那契数？9
- 3) `n` 存储在内存中的哪个地址？提示：查看寄存器的内容。0x10000010
- 4) 在不修改代码（不改变“Edit”栏下的代码）的情况下，手动修改寄存器的值来计算第 13 个斐波那契数（索引从 0 开始）。应该修改的寄存器是哪个？
将地址 0x10000010 中的值从 9 修改为 13

热身练习 2: 从 C 翻译到 RISC-V

打开文件 `ex2.c` 和 `ex2.s`。汇编代码 (`ex2.s`) 是给定 C 程序 (`ex2.c`) 到 RISC-V 的汇编翻译。

阅读汇编代码，思考以下问题（注意：不需要在作业中回答）：

- 1) 表示变量 `k` 的寄存器？ `s3`
- 2) 表示总和变量 `sum` 的寄存器？ `s0`
- 3) 分别指向源数组和目标数组的指针的寄存器？ `s1`、`s2`
- 4) 在汇编指令里，指针怎么表示？ 寄存器里存地址

正式作业：

练习 3: 用 `map` 调用 RISC-V 函数

本练习将使用文件 `list_map.s` 中的汇编语言程序。

在 C 语言中，链表中的一个结点的数据类型被定义为：

```
struct node {
    int value;
    struct node *next;
};
```

为了实现一个 `map` 函数，其功能是：递归地遍历链表，将指定函数应用于链表的每一个结点的 `value`，并将返回的值存储在相应的节点中。在 C 语言中，`map` 函数的定义是这样的：

```
void map(struct node *head, int (*f)(int))
{
    //f 是函数指针，指向某个函数的起始地址， 这个函数只有一个参数，该参数类型为 int
    if (!head) { return; }
    head->value = f(head->value);
    // 提示，转换为 RISC-V 汇编调用函数 f 时，应该使用 JALR，而不是 JAL
    map(head->next, f);
}
```

如果你不了解“函数指针”，建议先补充这方面的知识。函数指针是一个指向函数的指针变量，其本质是一个指针，代表函数的内存地址。

例如：有一个函数 `square`：

```
int square(int i) { return i*i; }
```

`map` 的第一个参数是一个值为 32 位整数的单链表的头节点的地址。

`square` 可以作为第二个参数传递给 `map`。

在本练习中，我们将在 RISC-V 汇编程序中完成 `list map` 的实现。函数的实现过程中，是对链表原位置中的值进行改变，而不是创建并返回带有修改值的新链表。

补充 `list_map.s` 中的汇编语言程序，使得其输出为：

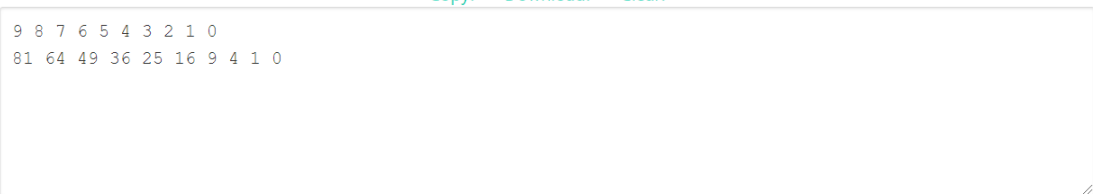
```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

第一行是原始链表中每一个结点的值，第二行是应用 `map` 的第二个参数指定的函数（本例为：`square`）对每一个结点修改后、链表中各个结点的值。

添加您的代码，确保运行的结果为上述结果。

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你添加的部分高亮显示）贴上来。

运行结果：



```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

源代码：

```
.globl map
```

```
.text
```

```
main:
```

```
    jal ra, create_default_list
```

```
    add s0, a0, x0 # a0 = s0 is head of node list
```

```
    #print the list
```

```
    add a0, s0, x0
```

```
    jal ra, print_list
```

```
    # print a newline
```

```
    jal ra, print_newline
```

```
    # load your args
```

```
    add a0, s0, x0 # load the address of the first node into a0
```

```
    # load the address of the function in question into a1 (check out la on
the green sheet)
```

```
    ### YOUR CODE HERE ###
```

```
    la a1, square
```

```
    # issue the call to map
```

```
    jal ra, map
```

```
    # print the list
```

```
    add a0, s0, x0
```

```
    jal ra, print_list
```

```

    # print another newline
    jal ra, print_newline

    addi a0, x0, 10
    ecall #Terminate the program

map:
    # Prologue: Make space on the stack and back-up registers
    ### YOUR CODE HERE ###
    addi sp, sp, -8
    sw s0, 4(sp)
    sw s1, 0(sp)

    beq a0, x0, done    # If we were given a null pointer (address 0),
we're done.

    add s0, a0, x0 # Save address of this node in s0
    add s1, a1, x0 # Save address of function in s1

    # Remember that each node is 8 bytes long: 4 for the value followed by
4 for the pointer to next.
    # What does this tell you about how you access the value and how you
access the pointer to next?

    # load the value of the current node into a0
    # THINK: why a0?
    ### YOUR CODE HERE ###
    lw a0, 0(s0)

    # Call the function in question on that value. DO NOT use a label (be
prepared to answer why).
    # What function? Recall the parameters of "map"
    ### YOUR CODE HERE ###
    addi sp, sp, -4
    sw ra, 0(sp)
    jalr s1
    lw ra, 0(sp)
    addi sp, sp, 4

    # store the returned value back into the node
    # Where can you assume the returned value is?
    ### YOUR CODE HERE ###
    sw a0, 0(s0)

```

```

# Load the address of the next node into a0
# The Address of the next node is an attribute of the current node.
# Think about how structs are organized in memory.
### YOUR CODE HERE ###
lw a0, 4(s0)

# Put the address of the function back into a1 to prepare for the
recursion
# THINK: why a1? What about a0?
### YOUR CODE HERE ###
mv a1, s1

# recurse
### YOUR CODE HERE ###
addi sp, sp, -4
sw ra, 0(sp)
jal ra, map
lw ra, 0(sp)
addi sp, sp, 4

done:
# Epilogue: Restore register values and free space from the stack
### YOUR CODE HERE ###
lw s1, 0(sp)
lw s0, 4(sp)
addi sp, sp, 8

jr ra # Return to caller

square:
mul a0, a0, a0
jr ra

create_default_list:
addi sp, sp, -12
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
li s0, 0      # pointer to the last node we handled
li s1, 0      # number of nodes handled
loop: #do...
li a0, 8

```

```

        jal ra, malloc      # get memory for the next node
        sw  s1, 0(a0)      # node->value = i
        sw  s0, 4(a0)      # node->next = last
        add s0, a0, x0     # last = node
        addi s1, s1, 1     # i++
        addi t0, x0, 10
        bne s1, t0, loop    # ... while i!= 10
        lw  ra, 0(sp)
        lw  s0, 4(sp)
        lw  s1, 8(sp)
        addi sp, sp, 12
        jr  ra

print_list:
        bne a0, x0, printMeAndRecurse
        jr  ra             # nothing to print
printMeAndRecurse:
        add t0, a0, x0     # t0 gets current node address
        lw  a1, 0(t0)      # a1 gets value in current node
        addi a0, x0, 1     # prepare for print integer ecall
        ecall
        addi a1, x0, ' '   # a0 gets address of string containing space
        addi a0, x0, 11    # prepare for print string syscall
        ecall
        lw  a0, 4(t0)      # a0 gets address of next node
        jal x0, print_list # recurse. We don't have to use jal because we
already have where we want to return to in ra

print_newline:
        addi a1, x0, '\n' # Load in ascii code for newline
        addi a0, x0, 11
        ecall
        jr  ra

malloc:
        addi a1, a0, 0
        addi a0, x0 9
        ecall
        jr  ra

```

练习 4:

本练习和练习 3 稍微有些不同， 在本练习中，链表中的一个结点的数据类型被定义为：

```
struct node {
    int *arr;
    int size;
    struct node *next;
};
```

链表中每一个节点是数组，arr 是数组的地址，size 是数组的大小。

新的 `map` 函数的作用是：遍历链表中的每一个节点中的数组，将函数 `f` 运用于数组中的每一个元素，结果写回数组中对应位置。

```
void map(struct node *head, int (*f)(int)) {
    if (!head) { return; }
    for (int i = 0; i < head->size; i++) {
        head->arr[i] = f(head->arr[i]);
    }
    map(head->next, f);
}
```

给定的文件 `megalistmanips.s`，其中函数 `f(int x)` 的功能是计算并返回 $x*(x+1)$ ，本次 `map` 的正确运行结果应该如下：

Lists before:

```
5 2 7 8 1
1 6 3 8 4
5 2 7 4 3
1 2 3 4 7
5 6 7 8 9
```

Lists after:

```
30 6 56 72 2
2 42 12 72 20
30 6 56 20 12
2 6 12 20 56
30 42 56 72 90
```

但 `megalistmanips.s` 这个文件存在错误，没有正确运行，请找出它的错误，并修正。
一些提示：

- `jal` 进行函数调用之前，我们需要将哪些内容入栈？
- `add t0, s0, x0` 和 `lw t0, 0(s0)` 这两条指令区别是什么？
- 注意正确表示结构体 `node` 中的成员所属的数据类型；
- 重点修改 `map`，`mapLoop` 部分，其余函数，例如 `done` 不用修改，但它可以帮助理解整个程序；
- 除了 `s0` 和 `s1`，不允许使用另外的 `s` 开头的寄存器（save register: `s2-s11` 不能再使用），你可以使用 temporary register（例如 `t1`，`t2` 等），并遵循约定的寄存器使用规范。

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你修改的部分高亮显示）贴上来。

运行结果：

Lists before:

```
5 2 7 8 1
1 6 3 8 4
5 2 7 4 3
1 2 3 4 7
5 6 7 8 9
```

Lists after:

```
30 6 56 72 2
2 42 12 72 20
30 6 56 20 12
2 6 12 20 56
30 42 56 72 90
```

源代码：

```
.globl map
```

```
.data
```

```
arrays: .word 5, 6, 7, 8, 9
        .word 1, 2, 3, 4, 7
        .word 5, 2, 7, 4, 3
        .word 1, 6, 3, 8, 4
        .word 5, 2, 7, 8, 1
```

```
start_msg: .ascii "Lists before: \n"
```

```
end_msg:   .ascii "Lists after: \n"
```

```
.text
```

```
main:
```

```
    jal create_default_list
    mv s0, a0    # v0 = s0 is head of node list
```

```
    #print "lists before: "
```

```
    la a1, start_msg
```

```
    li a0, 4
```

```
    ecall
```

```
    #print the list
```

```
    add a0, s0, x0
```

```
    jal print_list
```

```
    # print a newline
```

```
    jal print_newline
```



```

# issue the map call
add a0, s0, x0      # load the address of the first node into a0
la a1, mystery      # load the address of the function into a1

jal map

# print "lists after: "
la a1, end_msg
li a0, 4
ecall

# print the list
add a0, s0, x0
jal print_list

li a0, 10
ecall

map:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s1, 4(sp)
    sw s0, 8(sp)

    beq a0, x0, done    # if we were given a null pointer, we're done.

    add s0, a0, x0      # save address of this node in s0
    add s1, a1, x0      # save address of function in s1
    add t0, x0, x0      # t0 is a counter

    # remember that each node is 12 bytes long:
    # - 4 for the array pointer
    # - 4 for the size of the array
    # - 4 more for the pointer to the next node

    # also keep in mind that we should not make ANY assumption on which
registers
    # are modified by the callees, even when we know the content inside the
functions
    # we call. this is to enforce the abstraction barrier of calling
convention.
mapLoop:

```

```

    lw t1, 0(s0)          # load the address of the array of current node into t1
#edited
    lw t2, 4(s0)          # load the size of the node's array into t2

    li t4, 4              #edited
    mul t3, t0, t4        #edited
    add t1, t1, t3        # offset the array address by the count #edited

    lw a0, 0(t1)          # load the value at that address into a0

#edited
    addi sp, sp, -12
    sw t0, 0(sp)
    sw t1, 4(sp)
    sw t2, 8(sp)

    jalr s1                # call the function on that value.

#edited
    lw t0, 0(sp)
    lw t1, 4(sp)
    lw t2, 8(sp)
    addi sp, sp, 12

    sw a0, 0(t1)          # store the returned value back into the array
    addi t0, t0, 1        # increment the count
    bne t0, t2, mapLoop   # repeat if we haven't reached the array size yet

    lw a0, 8(s0)          # load the address of the next node into a0 #edited

    add a1, s1, x0        # put the address of the function back into a1 to
prepare for the recursion #edited

    jal map                # recurse
done:
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 12
    jr ra

mystery:
    mul t1, a0, a0
    add a0, t1, a0

```

```

        jr ra

create_default_list:
    addi sp, sp, -4
    sw ra, 0(sp)
    li s0, 0 # pointer to the last node we handled
    li s1, 0 # number of nodes handled
    li s2, 5 # size
    la s3, arrays
loop: #do...
    li a0, 12
    jal malloc # get memory for the next node
    mv s4, a0
    li a0, 20
    jal malloc # get memory for this array

    sw a0, 0(s4) # node->arr = malloc
    lw a0, 0(s4)
    mv a1, s3
    jal fillArray # copy ints over to node->arr

    sw s2, 4(s4) # node->size = size (4)
    sw s0, 8(s4) # node-> next = previously created node

    add s0, x0, s4 # last = node
    addi s1, s1, 1 # i++
    addi s3, s3, 20 # s3 points at next set of ints
    li t6, 5
    bne s1, t6, loop # ... while i!= 5
    mv a0, s4
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra

fillArray: lw t0, 0(a1) #t0 gets array element
    sw t0, 0(a0) #node->arr gets array element
    lw t0, 4(a1)
    sw t0, 4(a0)
    lw t0, 8(a1)
    sw t0, 8(a0)
    lw t0, 12(a1)
    sw t0, 12(a0)
    lw t0, 16(a1)
    sw t0, 16(a0)

```

```

        jr ra

print_list:
    bne a0, x0, printMeAndRecurse
    jr ra    # nothing to print
printMeAndRecurse:
    mv t0, a0 # t0 gets address of current node
    lw t3, 0(a0) # t3 gets array of current node
    li t1, 0    # t1 is index into array
printLoop:
    slli t2, t1, 2
    add t4, t3, t2
    lw a1, 0(t4) # a0 gets value in current node's array at index t1
    li a0, 1    # prepare for print integer ecall
    ecall
    li a1, ' '  # a0 gets address of string containing space
    li a0, 11   # prepare for print string ecall
    ecall
    addi t1, t1, 1
li t6 5
    bne t1, t6, printLoop # ... while i!= 5
    li a1, '\n'
    li a0, 11
    ecall
    lw a0, 8(t0) # a0 gets address of next node
    j print_list # recurse. We don't have to use jal because we already have
where we want to return to in ra

print_newline:
    li a1, '\n'
    li a0, 11
    ecall
    jr ra

malloc:
    mv a1, a0 # Move a0 into a1 so that we can do the syscall correctly
    li a0, 9
    ecall
    jr ra

```

练习 5：写一个没有条件分支（branch）的函数

有一个 (discrete valued function) 离散值函数 f ，作用于一个整数集合 $\{-3, -2, -1, 0, 1, 2, 3\}$ 。函数的定义如下：

```
f(-3) = 6
f(-2) = 61
f(-1) = 17
f(0) = -38
f(1) = 19
f(2) = 42
f(3) = 5
```

在 `discrete_fn.s` 文件中实现该函数，注意：不能使用任何 `branch` 或者 `jump` 指令

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你添加的部分高亮显示）贴上来。

运行结果：

```
f(-3) should be 6, and it is: 6
f(-2) should be 61, and it is: 61
f(-1) should be 17, and it is: 17
f(0) should be -38, and it is: -38
f(1) should be 19, and it is: 19
f(2) should be 42, and it is: 42
f(3) should be 5, and it is: 5
```

源代码：

```
.globl f

.data
neg3: .ascii "f(-3) should be 6, and it is: "
neg2: .ascii "f(-2) should be 61, and it is: "
neg1: .ascii "f(-1) should be 17, and it is: "
zero: .ascii "f(0) should be -38, and it is: "
pos1: .ascii "f(1) should be 19, and it is: "
pos2: .ascii "f(2) should be 42, and it is: "
pos3: .ascii "f(3) should be 5, and it is: "

output: .word 6, 61, 17, -38, 19, 42, 5
.text
main:
    la a0, neg3
    jal print_str
    li a0, -3
    la a1, output
    jal f          # evaluate f(-3); should be 6
    jal print_int
```

```

jal print_newline

la a0, neg2
jal print_str
li a0, -2
la a1, output
jal f          # evaluate f(-2); should be 61
jal print_int
jal print_newline

la a0, neg1
jal print_str
li a0, -1
la a1, output
jal f          # evaluate f(-1); should be 17
jal print_int
jal print_newline

la a0, zero
jal print_str
li a0, 0
la a1, output
jal f          # evaluate f(0); should be -38
jal print_int
jal print_newline

la a0, pos1
jal print_str
li a0, 1
la a1, output
jal f          # evaluate f(1); should be 19
jal print_int
jal print_newline

la a0, pos2
jal print_str
li a0, 2
la a1, output
jal f          # evaluate f(2); should be 42
jal print_int
jal print_newline

la a0, pos3
jal print_str

```

```

li a0, 3
la a1, output
jal f          # evaluate f(3); should be 5
jal print_int
jal print_newline

```

```

li a0, 10
ecall

```

f takes in two arguments:
 # a0 is the value we want to evaluate f at
 # a1 is the address of the "output" array (defined above).
 # Think: why might having a1 be useful?
 f:

```

    # YOUR CODE GOES HERE!

```

```

    addi a0, a0, 3
    slli a0, a0, 2
    add a0, a0, a1
    mv t0, x0

```

```

    lb t0, 0(a0)
    mv a0, t0

```

```

    jr ra          # Always remember to jr ra after your function!

```

```

print_int:
    mv a1, a0
    li a0, 1
    ecall
    jr    ra

```

```

print_str:
    mv a1, a0
    li a0, 4
    ecall
    jr    ra

```

```

print_newline:
    li a1, '\n'
    li a0, 11
    ecall
    jr    ra

```

注：本作业选自 UC Berkeley 大学 CS61C 课程 Lab3 和 lab4，相关链接：
<https://inst.eecs.berkeley.edu/~cs61c/su20/labs/lab03/>
<https://inst.eecs.berkeley.edu/~cs61c/su20/labs/lab04/>