**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](https://venus.cs61c.org/)。你可以通过查看Venus[参考](https://inst.eecs.berkeley.edu/~cs61c/sp21/resources/venus-reference)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？）
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？
3. n存储在内存中的哪个地址？提示：查看寄存器的内容。
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？

**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？
2. 表示总和变量sum的寄存器；
3. 分别指向源数组和目标数组的指针的寄存器；
4. 在汇编指令里，指针怎么表示？

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。**

**我的代码：**

**.globl map**

**.text**

**main:**

**jal ra, create\_default\_list#创建了一个链表，顶点是a0**

**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 -12**

**sw s0 0(sp)**

**sw s1 4(sp)**

**sw ra 8(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 ###**

**jalr s1**

**# 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 ###**

**add a1 x0 s1**

**# recurse**

**### YOUR CODE HERE ###**

**jal ra, map**

**done:**

**# Epilogue: Restore register values and free space from the stack**

**### YOUR CODE HERE ###**

**lw s0 0(sp)**

**lw s1 4(sp)**

**lw ra 8(sp)**

**addi sp sp 12**

**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**

**运行结果：**

****

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。**

## 我的代码：（map部分，其余部分未修改）

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 把add改成lw

lw t2, 4(s0) # load the size of the node's array into t2

addi t3,x0,4 #修改，每次调整地址应该考虑int的大小

mul t3 t0 t3

add t1, t1,t3 # offset the array address by the count

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

addi sp, sp, -4 #修改，t1值需要保存

sw t1 0(sp)

jalr s1 # call the function on that value.

lw t1 0(sp)

addi sp sp 4

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,修改，使用lw指令而不是la

add a1 x0 s1 # put the address of the function back into a1 to prepare for the recursion 修改，用add指令而不是ld

jal map # recurse

done:

lw s0, 8(sp)

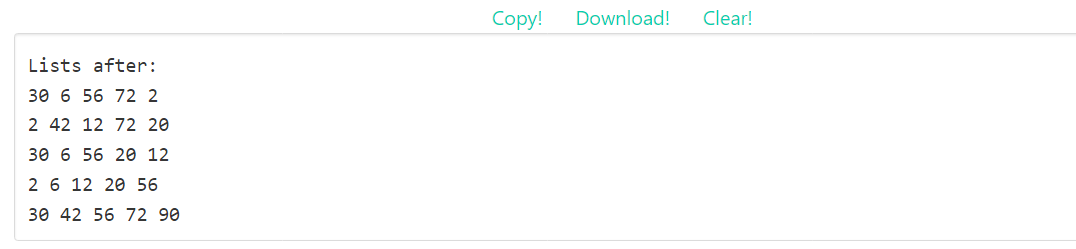
lw s1, 4(sp)

lw ra, 0(sp)

addi sp, sp, 12

jr ra

最终结果：



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。**

我的代码：

# 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 t1 a0 3

addi t2 x0 4

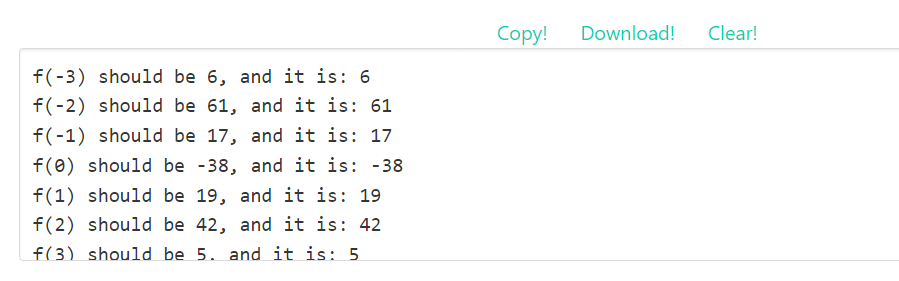
mul t1 t1 t2

add t3 a1 t1

lw a0 0(t3)

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

最终结果：



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/>