

订阅DeepL Pro以编辑此演示文稿。  
访问[www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document)，了解更多信息。

**CMPEN331 - 计算机组织与设计**

**实验室2**

其目的是学习使用MARS模拟器。你可以从<http://courses.missouristate.edu/KenVollmar/MARS/> 下载MARS程序。这是4.5版本。 它是一个Java程序，所以它应该在任何地方运行。我们将按照海勒教授编写的作业进行学习。

你可以查看Gary Shute教授的以下[教程：http://www.d.umn.edu/%7Egshute/mips/Mars/Mars.xhtml](http://www.d.umn.edu/%7Egshute/mips/Mars/Mars.xhtml)

要启动MARS，只需双击其图标。 你应该看到一个有四个部分的窗口--顶部的一些按钮，一个带有 "编辑 "和 "执行 "标签的部分，下面一个带有 "火星信息 "和 "运行I/O "标签的部分，还有一个显示处理器寄存器的部分。 在执行选项卡中，有文本段（内存中放置程序指令的部分）、数据段（内存中放置程序全局数据的部分）和标签（符号表，连接程序中的标签和内存位置）的面板。 如果你决定调整MARS窗口及其各个部分的大小，要确保Registers面板显示所有三列。如果你在 "执行 "选项卡中没有看到 "标签 "面板，进入 "设置 "菜单，选中 "显示标签窗口 "项目。

让我们从一个绝对最小的程序开始--它开始和结束，不做其他事情。 注释以#开头，一直到行尾。 汇编指令以点开始，并通常以制表符缩进。 指令通常用制表符缩进，而标签则在左页边。

* 启动MARS，然后点击新文件按钮（最左边的，或菜单文件/新）。 现在你可以在编辑标签下的编辑窗格中输入程序。 默认的文件名是mips1.asm。 如果你再次点击新建文件，你会得到第二个编辑窗格，文件名是mips2.asm。
* 不要用复制和粘贴的方式走捷径--你会错过MARS在你输入时显示的有用信息。 复制和粘贴以后就可以了（当然，前提是不要作弊）。

# 版本1，什么都不做，然后退出

#切换到文本段

        .文本

.globl main

主要的。

        # 主程序的其余部分将在这里进行

        # 结束程序，没有明确的返回状态

        li $v0, 10

        syscall

# 切换到数据段

        .数据

        # 全局数据将在此定义

* 将文件保存为version1.asm（第四个按钮，或菜单File / Save as ...）。 汇编语言程序的通常文件名应该是像version1.s或version1.asm。 在Windows上，Visual Studio通常与这些文件名扩展名有关，它不喜欢MIPS Assembler，所以要小心从文件夹显示中双击。
* QtSPIM只接受.s或.asm文件。 MARS将接受任何文件名；使用像version1.mips这样的文件将防止在双击图标时出现任何不良的互动，但它使使用其他模拟器更加困难。
* 组装程序（第13个按钮，第三组中的第一个，看起来像一把螺丝刀和扳手，或菜单Run / Assemble）。你应该在 "火星信息 "面板中看到编辑/执行部分的下方。 当然，你的目录名称会有所不同。
  + **请注意**，在文件被保存之前，你不能将其组装起来。
  + 在程序成功装配后，显示屏切换到执行标签，在文本段中显示机器代码。 寄存器信息显示程序将从地址0x00400000开始，这是寄存器pc的内容，即程序计数器；这个寄存器总是保存着下一条要执行的指令的地址。 文本段强调了该指令。
  + 全局变量显示在数据段中，只是还没有任何变量。 标签面板显示了全局名称，以后当你的程序使用函数并跨越几个文件时，将需要这些名称。 现在，它只显示main在位置0x00400000。
  + 运行程序（第十四个按钮，第三组中的第二个，看起来像一个向右的三角形，或菜单Run / Go）。你应该在 "*火星信息 "面板*和 "**运行I/O "面板中**看到。
  + *去：运行版本1.asm*
  + *去：执行成功。*
  + **--程序运行完毕 --**
* 你可以在程序中设置断点，方法是在 "文本段 "面板中的每条指令左侧勾选Bkpt框。 稍后会有更多的内容。
* 请注意，li指令（即时加载）变成了addiu指令（无符号即时添加）。 这是因为li实际上是一条伪指令。 寄存器$v0变成了寄存器$2，这是同一件事。 寄存器$0（或$zero）总是包含0位。
* syscall指令（系统调用）启动了一个操作系统动作；哪个动作取决于指令执行时寄存器$v0的值。 Syscall 10类似于C或C++中的exit()函数，而syscall 17，我们将在后面使用，实际上是exit()函数。
* 请注意，程序计数器寄存器（pc）以0x00400000开始，以0x00400008结束，也就是比程序中的最后一条指令多了一条。
* 如果你想重新运行程序，你可以用右边第二个按钮或菜单Run / Reset将内存和寄存器重置为默认值。 这将使$v0回到0x00000000，pc回到0x00400000。
* 如果你想一次运行一条指令（单步走），那么重置内存和寄存器，反复使用带下标1的运行按钮，或菜单Run / Step。 注意寄存器$v0和pc的变化；$v0在变化时被高亮显示，但pc没有。

接下来，我们添加一些函数定义，这样可以更容易地处理系统调用。 注意第二个.text指令，这样新的代码就不会被当作数据。 程序越来越长，所以我们用注释作为程序不同部分的分界线。

# 第二版，什么都不做，然后退出

#切换到文本段

        .文本

.globl main

主要的。

        # 主程序的其余部分将在这里进行

        #调用函数Exit0

        jal Exit0 # 结束程序，默认返回状态

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 切换到数据段

        .数据

        # 全局数据将在此定义

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 围绕一些系统调用的封装函数

# 完整清单见P&H COD，图A.9.1。

#切换到文本段

        .文本

.globl 打印\_整数

Print\_integer: # 打印寄存器$a0中的整数

        li $v0, 1

        syscall

        jr $ra

.globl Print\_string

Print\_string: # 打印字符串，其起始地址在寄存器$a0中

        li $v0, 4

        syscall

        jr $ra

.globl 退出

退出。 # 结束程序，没有明确的返回状态

        li $v0, 10

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit0

Exit0: # 结束程序，默认返回状态

        li $a0, 0 # 返回状态 0

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit2

Exit2: #结束程序，从寄存器$a0返回状态。

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

* 重复前面的步骤，并做适当的修改。 首先清除Mars Messages和Run I/O面板可能会有帮助。
* 注意，pc以0x00400034的值结束，这是函数Exit0中系统调用指令之后的指令地址。 通过文本段、标签和寄存器的工作来验证这一点。

还要注意的是，寄存器$ra的值是0x00400004，这是jal Exit0指令之后的指令地址。 稍后会有更多关于这方面的内容。 函数Print\_integer的第一条指令的地址是0x004000004，这本质上是一个巧合。

接下来是第三个版本，即无处不在的 "Hello World "程序。

# 第3版，打印一些东西，然后退出

#切换到文本段

        .文本

.globl main

主要的。

        # 主程序在此进行

        la $a0, hello\_string

        jal Print\_string

        jal Exit0 # 结束程序，默认返回状态

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 切换到数据段

        .数据

        # 全局数据在此定义

hello\_string:

.asciiz "Hello, world\n"

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 围绕一些系统调用的封装函数

# 完整清单见P&H COD，图A.9.1。

#切换到文本段

        .文本

.globl 打印\_整数

Print\_integer: # 打印寄存器$a0中的整数

        li $v0, 1

        syscall

        jr $ra

.globl Print\_string

Print\_string: # 打印字符串，其起始地址在寄存器$a0中

        li $v0, 4

        syscall

        jr $ra

.globl 退出

退出。 # 结束程序，没有明确的返回状态

        li $v0, 10

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit0

Exit0: # 结束程序，默认返回状态

        li $a0, 0 # 返回状态 0

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit2

Exit2: #结束程序，从寄存器$a0返回状态。

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

重复前面的步骤，并进行适当的修改。 你应该在 "*火星信息 "面板*和 "**运行I/O "面板中**看到。

*组装：操作成功完成。*

*去：运行版本3.asm*

*去：执行成功。*

**你好，世界**

**--程序运行完毕 --**

请注意，标签hello\_string在标签面板中没有被标记（全局）。 这是因为我们没有使用指令.globl hello\_string 。 这个标签在文件version3.asm中被定义并可以访问，但在文件之外无法看到。 试着添加该指令，看看标签面板会有什么变化。

最后，这里有一个源自P&H COD的程序，图A.1.4，为MARS模拟器进行了修正和调整。 MARS和QtSPIM在main()的起始地址和其他一些事情上有一些区别。 这个版本可以在两个模拟器上工作。 在这一点上，复制和粘贴似乎完全合适。

       .文本

# .align 2 # MARS不喜欢这样

.globl main

主要的。

        subu $sp, $sp, 40 # 堆栈推送，40字节

        sw $ra, 20($sp) # 保存返回地址寄存器（sw = 存储字）。

        sd $a0, 32($sp) # 保存寄存器$a0, $a1 (sd = 存储双字)

        sw $0, 24($sp) # sum = 0

        sw $0, 28($sp) # i = 0

循环。

        lw $t6, 28($sp) # i

        mul $t7, $t6, $t6 # i \* i

        lw $t8, 24($sp) # 总数

        addu $t9, $t8, $t7 # sum + i\*i

        sw $t9, 24($sp) # sum = sum + i\*i

        addu $t0, $t6, 1 # i + 1

        sw $t0, 28($sp) # i = i + 1

        ble $t0, 100, loop # if (i <= 100) goto loop

        la $a0, str1

        jal Print\_string # 打印起始地址为寄存器$a0的字符串。

        lw $a0, 24($sp) # 总数

        jal Print\_integer # 打印寄存器$a0中的整数。

        la $a0, str2

        jal Print\_string # 打印起始地址为寄存器$a0的字符串。

        # 这个退出序列可以用于QtSPIM，但不能用于MARS

# 移动 $v0, $0 # 返回状态 0

# lw $ra, 20($sp) # 恢复保存的返回地址

# addu $sp, $sp, 40 # 堆栈弹出（重要的是--和之前一样的40字节！）。

# jr $ra # 从main()返回到操作系统。

        # MARS喜欢这样，但我们需要演示Exit2

# jal Exit # 结束程序，没有明确的返回状态

        # MARS喜欢这个

        移动$a0, $0

        jal Exit2 # 结束程序，并从寄存器a0返回状态。

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        .数据

.对齐0

str1。

.asciiz "从0到100的总和是："

str2。

.asciiz ":\n"

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 围绕一些系统调用的封装函数

# 完整清单见P&H COD，图A.9.1。

        .文本

# .align 2 # MARS不喜欢这样

.globl 打印\_整数

Print\_integer: # 打印寄存器$a0中的整数

        li $v0, 1

        syscall

        jr $ra

.globl Print\_string

Print\_string: # 打印字符串，其起始地址在寄存器$a0中

        li $v0, 4

        syscall

        jr $ra

.globl 退出

退出。 # 结束程序，没有明确的返回状态

        li $v0, 10

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit0

Exit0: # 结束程序，默认返回状态

        li $a0, 0 # 返回状态 0

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

.globl Exit2

Exit2: #结束程序，从寄存器$a0返回状态。

        li $v0, 17

        syscall

        jr $ra # 这个指令永远不会被执行

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

* QtSPIM将main()视为由操作系统调用的函数，这在类Unix操作系统中确实如此。 MARS将main()视为一个不知从何处调用的函数（显然），因为$ra最初是0。

**现在我们需要验证你是否真的能成功使用MARS。**

* 当你运行前一个程序时，打印的是什么？
* 当程序结束时，寄存器$t7中的值是多少（十进制）？
* 在汇编源码的第13行设置一个指令的断点。 再次运行该程序；它应该停在断点处。 现在执行这一条指令。 执行这条指令后，哪些寄存器发生了变化？ 你可能需要在断点处继续运行几次，才能看到发生了什么。 请注意，P&H COD附录A.10有所有指令的描述，但你不能直接查找答案。 (你应该在App.A.10中的说明，但是答案需要你把几个不同的信息放在一起，而不是只有一个。）

你必须提到所有的寄存器，它们不仅在一次迭代中发生变化，而且在多次迭代中发生变化。$t7不是唯一变化的寄存器，继续运行，你会看到更多的变化。PC、lo、hi是每次运行都会变化的寄存器，所以它们不是很重要，但仍然值得在实验报告中提及。

**把这三个问题的答案放在你的实验室里。你还必须上传所用的源代码，全部放在一个压缩文件中。**