

# 目标代码生成 实习指导

许畅 陈嘉 朱晓瑞

Language is a process of free creation: its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.

—— Stanley Milgram

上一次实习中，我们已经将输入程序翻译为一个涉及相当多底层细节的中间代码。这套中间代码在很大程度上已经可以很容易地翻译成很多 RISC 的机器代码，不过仍然存在以下问题：

- ❖ 中间代码与目标代码并不一定是严格的一一对应关系。有可能某一条中间代码对应多条目标代码，也有可能多条中间代码对应一条目标代码。
- ❖ 中间代码里我们使用了数目无限的变量以及临时变量，但处理器所拥有的寄存器数量总是有限的。RISC 机器的一大特点就是运算指令的操作数总是从寄存器中取得。
- ❖ 中间代码里我们并没有处理有关函数调用的细节。函数调用在中间代码里被我们抽象为若干条 ARG 语句和一条 CALL 语句，但在目标机器里一般不会有专门的器件为我们进行参数传递，我们必须借助于寄存器或者栈完成这一点。

其中，第一个问题被称为指令选择（Instruction Selection），第二个问题被称为寄存器分配（Register Allocation），第三个问题需要考虑如何对栈进行管理。

本次实习我们的主要任务就是编写程序来处理上述三个问题。

## QtSPIM 简易教程





“工欲善其事，必先利其器”，在着手解决前面所说的三个问题之前，让我们先来考察本次实习中所要用到的工具——SPIM。

SPIM 有两种版本：命令行版与 GUI 版。两个版本功能相似，命令行版使用起来更简洁，GUI 版使用起来更直观，你可以根据你的喜好进行选择。如果选择命令行版则可以直接在终端键入 `sudo apt-get install spim` 命令进行安装（注意需要在机器已连接外网的前提下使用此命令进行安装），如果是 GUI 版则需要访问 SPIM 的官方网站 <http://pages.cs.wisc.edu/~larus/spim.html> 下载并安装 QtSPIM 的 Linux 版本。命令行版的使用很简单，键入 `spim -file [汇编代码文件名]`

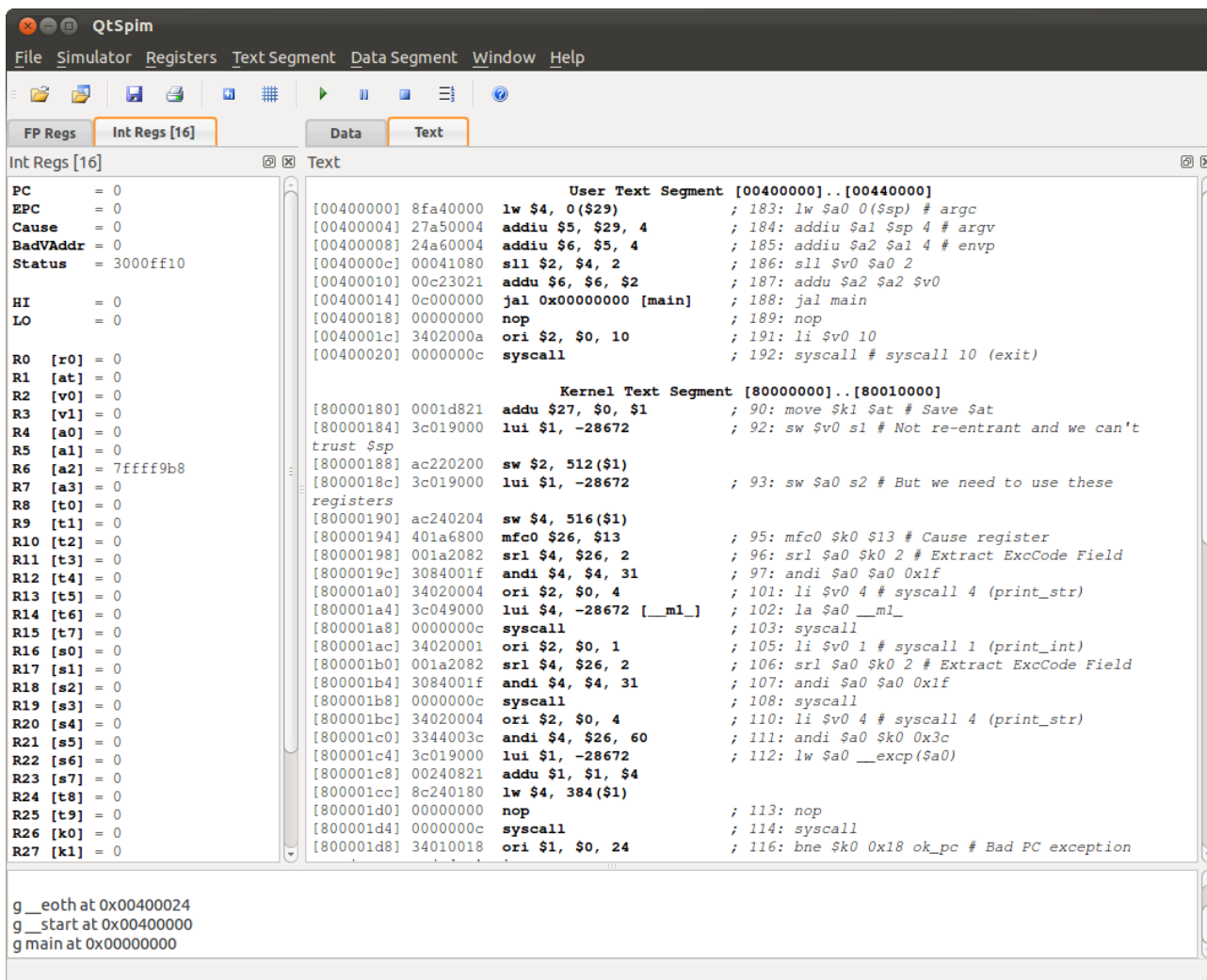
即可汇编并运行。其更详细的使用方法可以通过阅读手册 `man spim` 进行学习，下面的介绍主要针对 GUI 版本。

安装成功并运行后，可以看到如下页最上方所示的界面：

中间面积最大的一片是代码区，里面显示了许多 MIPS 用户代码和内核代码；左侧列出了 MIPS 中的各个寄存器以及这些寄存器中保存的内容。无论是代码还是寄存器内容，都可以通过上面的菜单选项切换二进制/十进制/十六进制的显示方式。

可以看到用户代码区已经存在了一部分代码，这些代码主要的作用是布置初始运行环境并调用名为 `main` 的函数。此时由于我们没有载入任何包含 `main` 标签的代码，因此如果我们单击  运行这段代码，会发现运行到 `jal main` 那一行就会出错。现在我们将一段包含 `main` 标签以及声明 `main` 标签为全局标签的 `“.globl main”` 语句的 MIPS 代码（例如，实习要求中的样例输出 1）保存成后缀名为 `.s` 或者 `.asm` 文件。单击 QtSPIM 工具栏上的  按钮，选择我们刚刚保存好的文件，此时就可以看到文件中的代码已经被载入到 QtSPIM 的代码区，再单击  运行就能在 Console 窗口观察到运行结果了。如果你想要单步执行，单击  或者按 `F10` 就行了。很简单吧？

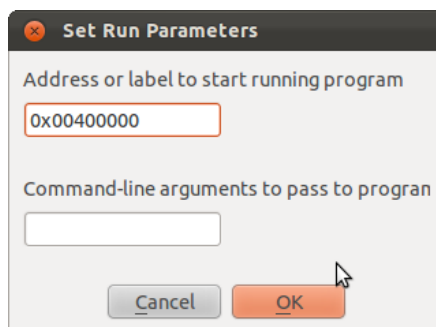
使用 SPIM 的好处之一就在于我们不需要干预内存的分配，SPIM 会自动帮助我们划分内存中的代码区、数据区和栈区。SPIM 具体采用大端还是小端的存储方式和你机器的处理器的存储方式相同（由于绝大多数消费类台式机和笔记本都使用了 Intel x86 体系结构的处理器，不出意外的话你会发现自己的 SPIM 是小端机）。



在代码区上方的选项卡 **Data** **Text** 处切换到 **Data** 选项卡，就可以看到当前内存中的数据信息，如下图所示：



单击菜单栏上的 Simulator → Run parameters，在弹出的对话框里可以设置程序运行的起始地址以及传给 main 函数的命令行参数。



# MIPS32 汇编代码书写

SPIM 不仅是一个 MIPS 的模拟器，也是一个 MIPS 的汇编器。想要让 SPIM 正常模拟，你首先需要为它准备符合格式的 MIPS 汇编代码文本文件。非操作系统内核的汇编代码文件由若干代码段和若干数据段组成，必须以.s 或者.asm 作为文件的后缀名，其中代码段以.text 开头，数据段以.data 开头。汇编代码文件中的注释以#开头。

数据段可以为程序中所要用到的常量、字符串字面量以及全局变量申请空间。其格式为：

name: storage\_type value(s)

其中 name 代表内存地址（标签）名，storage\_type 代表数据类型，value 代表初始值。常见的 storage\_type 和 value 有如下几类：

Storage Type	Description
.ascii str	Store the string <i>str</i> in memory, but do not null-terminate it
.asciiz str	Store the string <i>str</i> in memory and null-terminate it
.byte b1, b2, ..., bn	Store the n values in successive bytes of memory
.half h1, h2, ..., hn	Store the n 16-bit quantities in successive bytes of memory
.word w1, w2, ..., wn	Store the n 32-bit quantities in successive bytes of memory
.space n	Allocate n bytes of spaces in the current segment

下面是几个例子：







```
var1: .word 3           # create a single integer variable with initial
                        # value 3
array1: .byte 'a','b'   # create a 2-element character array with
                        # elements initialized to a and b
array2: .space 40        # allocate 40 consecutive bytes, with storage
                        # uninitialized; could be used as a 40-element
                        # character array, or a 10-element integer
                        # array; a comment should indicate which!
```

代码段由一条条 MIPS32 指令或者标签组成，标签后面要跟冒号，而指令与指令之间要以换行符分开。如果你还对体系结构课上学过的 MIPS 指令有些印象的话，读到样例输出中的汇编程序时你就会发现有很多像 la、li 这样的指令自己之前从来没有见过。实际上，这些指令的确不属于 MIPS32 指令集，它们的学名叫“伪指令”（pseudo-instruction），每条伪指令对应一条或者多条 MIPS32 指令，便于汇编指令的书写和记忆。几条比较常用的伪指令如下表所示：

Pseudo-instruction	Description	Corresponding MIPS32 Instruction
li Rdest, imm	(Load Immediate) Move the immediate <b>imm</b> into register <b>Rdest</b>	ori Rdest, \$0, imm
la Rdest, address	(Load Address) Load computed <b>address</b> , not the contents of the location, into the register <b>Rdest</b>	lui Rdest, address
move Rdest, Rsrc	Move the contents of <b>Rsrc</b> to <b>Rdest</b>	addu Rdest, Rsrc, \$0
bgt Rsrc1, Rsrc2, label	Conditional branch instructions	slt \$1, Rsrc1, Rsrc2 bne \$1, \$0, label
bge Rsrc1, Rsrc2, label		sle \$1, Rsrc1, Rsrc2 bne \$1, \$0, label
blt Rsrc1, Rsrc2, label		sgt \$1, Rsrc1, Rsrc2 bne \$1, \$0, label
ble Rsrc1, Rsrc2, label		sge \$1, Rsrc1, Rsrc2 bne \$1, \$0, label

如果你想知道 SPIM 还支持哪些指令和伪指令，请参阅我们提供的各种文档。

MIPS 体系结构总共有 32 个寄存器，在汇编代码中你可以使用 \$0 ~ \$31 来表示它们。为了便于表示和记忆，这 32 个寄存器也拥有各自的别名，如下表所示：

Register Number	Alternative Name	Description
 \$0	\$zero	The constant 0
\$1	\$at	(Assembler Temporary) reserved by the assembler
 \$2 - \$3	\$v0 - \$v1	(Values) from expression evaluation and function results
 \$4 - \$7	\$a0 - \$a3	(Arguments) First four parameters for subroutine. Not preserved across procedure calls.
 \$8 - \$15	\$t0 - \$t7	(Temporaries) Caller saved if needed. Subroutines can use without saving. Not preserved across procedure calls.
\$16 - \$23	\$s0 - \$s7	(Saved values) Callee saved if needed. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
\$24 - \$25	\$t8 - \$t9	(Temporaries) Caller saved if needed. Subroutines can use without saving. Not preserved across procedure calls.
 \$26 - \$27	\$k0 - \$k1	reserved for use by the interrupt/trap handler
\$28	\$gp	(Global Pointer) Points to the middle of the 64K block of memory in the static data segment.
\$29	\$sp	(Stack Pointer) Points to last location on the stack.
\$30	\$s8 / \$fp	Native MIPS compiler treats \$30 as \$s8, while GCC treats it as the frame pointer.
 \$31	\$ra	(Return Address)

最后，SPIM 也为我们提供了方便进行控制台交互的机制，这些机制通过系统调用（syscall）的形式来体现。为了进行系统调用，你首先需要向寄存器 \$v0 中存入一个数字指定具体要进行哪种系统调用，如有必要还需向其他指定寄存器中存入参数，最后再写一句 syscall 即可，例如：

```
li $v0, 4
la $a0, _prompt
syscall
```

调用了系统调用 print\_string(\_prompt)。

与本次实习相关的系统调用类型如下表所示：

Service	Syscall code	Argument	Result
print_int	1	\$a0 = integer	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_string	8	\$a0 = buffer, \$a1 = length	

<b>print_char</b>	11	\$a0 = char	
<b>read_char</b>	12		char (in \$a0)
<b>exit</b>	10		
<b>exit2</b>	17	\$a0 = result	

到此为止，如果对照样例输出并仔细阅读过上文，我们相信你已经基本了解了在本次实习中你的程序要输出什么。如果还有疑问，可以再去阅读我们为你提供的其他资料。

## 指令选择

指令选择可以看成是一类模式匹配问题。无论中间代码是树形的还是线形的，我们都需要在其中找到特定的模式，然后将这些模式对应到目标代码上（这有点类似于将语法树翻译为中间代码的过程）。取决于中间代码本身蕴含信息的多少，以及目标机器采用的是 RISC 还是 CISC 类型的指令集，指令选择可以是简单的寻找一一对应，也可以是涉及到许多细节处理和计算的复杂过程。相对而言，我们所采用的 MIPS32 指令集属于处理起来比较简单的 RISC 指令集，因此指令选择在本次实习里也属于比较简单的任务。

如果你使用了线形 IR，那么最简单的指令选择方式是逐条将中间代码进行对应到目标代码上。下表是由上次实习的中间代码对应到 MIPS 指令的一个例子：

IR	MIPS code
<b>LABEL x:</b>	x:
<b>x := #k</b>	li reg(x) <sup>1</sup> , k
<b>x := y</b>	move reg(x), reg(y) <sup>2</sup>
<b>x := y + #k</b>	addi reg(x), reg(y), k
<b>x := y + z</b>	add reg(x), reg(y), reg(z)
<b>x := y - #k</b>	addi reg(x), reg(y), -k
<b>x := y - z</b>	sub reg(x), reg(y), reg(z)
<b>x := y * z<sup>3</sup></b>	mul reg(x), reg(y), reg(z)
<b>x := y / z</b>	div reg(y), reg(z) mflo reg(x)
<b>x := *y</b>	lw reg(x), 0(reg(y))
<b>*x = y</b>	sw reg(y), 0(reg(x))
<b>GOTO x</b>	j x
<b>x := CALL f</b>	jal f move reg(x), \$v0
<b>RETURN x</b>	move \$v0, reg(x) jr \$ra
<b>if x==y GOTO z</b>	beq reg(x), reg(y), z
<b>if x!=y GOTO z</b>	bne reg(x), reg(y), z
<b>if x&gt;y GOTO z</b>	bgt reg(x), reg(y), z
<b>if x&lt;y GOTO z</b>	blt reg(x), reg(y), z
<b>if x&gt;=y GOTO z</b>	bge reg(x), reg(y), z
<b>if x&lt;=y GOTO z</b>	ble reg(x), reg(y), z

当然，这个翻译方案并不唯一。

很多时候，这种逐条翻译的方式往往得不到高效的代码。举个简单的例子：假设要访问某个数组元素 **a[3]**。变量 **a** 的首地址已经被保存到了寄存器 **\$t1** 中，我们希望将保存于内存中的 **a[3]** 的值放到 **\$t2** 里。如果按照上表使用逐

<sup>1</sup> “reg(x)” means the register assigned to variable x

<sup>2</sup>With a proper register allocator, we rarely have to generate a move instruction for assignment

<sup>3</sup>The multiplication, division and conditional jump instructions do not support constant other than 0, so if the IR contains something like “x := y \* #7”, the immediate “#7” must be loaded into a register first



条翻译的方式，由于这段功能对应到我们的中间代码里至少需要两条，故翻译出来的 MIPS 代码也需要两条指令：

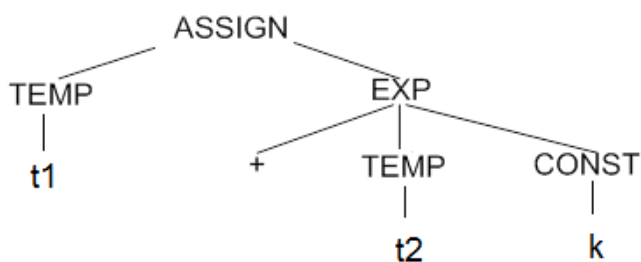
```
addi $t3, $t1, 12
lw $t2, 0($t3)
```

显然这两条指令可以利用 MIPS 中的基址寻址机制合并成一条指令：

```
lw $t2, 12($t1)
```

这个例子启示我们，有的时候为了得到更加高效的代码，我们需要一次考察多条中间代码，以期可以将多条中间代码翻译为一条 MIPS 代码。这个过程可以看作是一个多行的模式匹配，也可以看作用一个滑动窗口（sliding window）或是一个窥孔（peephole）滑过中间代码并查找可能的翻译方案的过程。同学看到这里也许会联想到课上学到的被称为“窥孔优化”（Peephole Optimization）的局部代码优化技术。没错！一次考察多条中间代码的做法在本质上非常类似于窥孔优化。实际上，你还可以将教材上介绍过的其他形式的窥孔优化添加到指令选择部分的代码里，使你的指令选择器在将中间代码翻译成目标代码的同时，还可以进行一定程度的局部代码优化。

树形 IR 的翻译方式类似于线形 IR，也是一个模式匹配过程，不过我们需要寻找的模式不是一句句线形代码，而是某种结构的子树。树形 IR 的匹配与翻译算法被称为“树重写”（Tree-rewriting）算法，这个算法在教材上也有介绍。仍然用一个例子说明这种方法，假设我们有一条翻译模式如下：



addi reg(t1), reg(t2), k

如何在中间代码中找到左图所对应的模式呢？答案还是那句话——遍历。我们可以按照深度优先的顺序考察树形 IR 中的每一个节点及其子节点的类型是否满足相应的模式。例如，上左图中的模式匹配写成代码可以是：

```
if (current_node -> kind == ASSIGN)
{
    left = current_node -> left;
    right = current_node -> right;
    if (left->kind == TEMP && right->kind == EXP)
    {
        op1 = right -> op1;
        op2 = right -> op2;
        if (right->op == '+' && op1->kind == TEMP &&
            op2->kind == CONST)
            emit_code("addi " + get_reg(left) + ", "
                      + get_reg(op1) + ", " + get_value(op2));
    }
}
```

你可以根据自己的树形 IR 写出翻译模式，然后使用类似于上面的方法进行翻译。

## 寄存器分配

RISC 机器的一个很显著的特点在于，除了 load/store 型指令之外，其余指令的所有操作数都来自寄存器而不能访问内存。除了数组、结构体必须强制放到内存中之外，中间代码里的任何一个非零的变量或者临时变量，只要它参与运算，则其值必将被载入到某一个寄存器中。在某个特定的程序点上选择哪一个寄存器保存哪一个变量的值，便是寄存器分配所要研究的问题。

寄存器可以说是现代计算机中最宝贵的资源之一。由于寄存器访问的性能远高于内存访问，使得寄存器分配算

法对于我们编译出的目标代码的效率影响尤其明显。为一段包含单个基本块、只有一种数据类型、访存代价固定的中间代码生成代价最少的寄存器分配方案是可以在多项式时间内被计算出来的。在此基础上，几乎添加任何假设（多于一个基本块、多于一种数据类型、使用多级存储模型等）都会使得寻找最优寄存器分配方案变成一个 NP-hard 问题。因此，目前的编译器使用的寄存器分配算法大都只是近似最优分配方案。在这一篇指导攻略中，我们将见到三种不同的寄存器分配算法，这三种算法的实现难度依次递增，但产生的目标代码的访存代价却是依次递减。

先来看最简单，也最低效的算法：将所有的变量/临时变量全都放在内存里。如此一来，每翻译一条中间代码前我们都将要用到的变量读到寄存器里，得到该条代码的计算结果之后又立即将结果写回内存。这种方法的确能将中间代码翻译成能正常运行的目标代码，而且实现和调试都特别容易，不过它的最大问题是对寄存器的利用率实在太低——不仅闲置了 MIPS 为我们提供的大部分通用寄存器，那些未被闲置的寄存器也没有对减少目标代码的访存次数做出任何贡献。如果你时间比较紧张，只是想快点完成实验，那么这种方法显然就是你的最佳选择。

寄存器分配之所以难是因为寄存器数量有限，被迫共用同一寄存器的变量数量太多，导致这些变量在使用时不得不在寄存器里换入换出，产生比较大的访存开销。下面要介绍的这两种方法都在考虑通过合理安排变量对寄存器的共用关系来最大限度地减少寄存器内容换入换出的代价。第一种方法叫做局部寄存器分配算法。之所以叫局部分配算法是因为这种方法会先将整段代码分拆成一个个基本块（将一段代码划分成基本块的过程教材上有提到，这里不再赘述），在基本块内部我们可以以各种启发式原则为块里出现的变量分配寄存器，但在基本块结束时这种算法会与前面提到的低效算法一样，需要将本块所有修改过的变量都写回内存。为了方便说明，我们假设所有的中间代码都形如  $z := x \text{ op } y$ ，其中  $x$  和  $y$  是两个要用到的操作数， $z$  是运算结果， $\text{op}$  代表一个任意的二元运算符（一元或者多元操作的处理与二元类似，请自行推广）。基本块开始时，所有寄存器都是闲置的。算法的大概框架如下：

```
for each operation  $z_i = x_i \text{ op } y_i$ 
   $r_x = \text{Ensure}(x_i)$ 
   $r_y = \text{Ensure}(y_i)$ 
  if ( $x_i$  is not needed after current operation)
     $\text{Free}(r_x)$ 
  if ( $y_i$  is not needed after current operation)
     $\text{Free}(r_y)$ 
   $r_z = \text{Allocate}(z_i)$ 
  emit MIPS code for  $r_z = r_x \text{ op } r_y$ 
```

其中  $\text{Free}(r)$  代表将寄存器  $r$  标记为闲置。该算法中还会用到另外两个辅助函数  $\text{Ensure}()$  和  $\text{Allocate}()$ ，它们的实现为：

```
 $\text{Ensure}(x)$ :
  if ( $x$  is already in register  $r$ )
    result =  $r$ 
  else
    result =  $\text{Allocate}(x)$ 
    emit MIPS code [lw result, x]
  return result

 $\text{Allocate}(x)$ :
  if (there exists a register  $r$  that currently has not been assigned
  to any variable)
    result =  $r$ 
  else
    result = the register that contains a value whose next use is
              farthest in the future
    spill result
  return result
```

其中  $\text{Allocate}()$  函数会用到每个变量在各程序点的使用信息，这些信息可以由一次对基本块中代码自后向前的扫描得到。

上述算法的核心思想其实很简单：对基本块内部的中间代码逐条扫描，如果当前代码中有变量需要寄存器，就从当前空闲的寄存器中选一个分配出去；如果没有空闲的寄存器，不得不将某个寄存器中的内容写回内存（该操作称为**溢出**，**spilling**）时，则选择那个包含本基本块内将来**用不到**或者**最久以后才用到**的变量的寄存器。通过这种启发式规则，该算法期望可以最大化每次溢出操作的收益，从而减少访存所需要的次数。

我们的教材上也介绍了一种和上述算法功能相似的局部寄存器的分配函数 `get_reg()`。教材上的方法通过引入寄存器描述符与变量描述符这两种数据结构，完全消除寄存器之间的数据移动，并且期望使溢出操作所产生的 `store` 指令的数量最小化。这里介绍的方法和教材上的方法各有优劣，你可以酌情选择，也可以在这些方法的基础上设计属于自己的局部分配算法。

## 图染色算法

局部寄存器分配算法虽然是启发式的算法，但在实际应用中它对于只包含一个基本块的中间代码段来说非常有效。不过，当我们尝试将其推广到多个基本块时，会遇到一个非常不易克服的困难：我们无法单看中间代码就确定程序的控制流走向。例如，假设当前的基本块运行结束时寄存器中有一个变量  $x$ ，当前基本块的最后一条中间代码又是条件跳转——控制流既有可能跳转到一个不使用  $x$  的基本块中，又有可能跳转到一个使用  $x$  的基本块中，那么此时变量  $x$  的值究竟是应该溢出到内存里呢，还是应该保留在寄存器里呢？

局部寄存器分配算法的这一弱点启发我们寻找一个适用于全局的寄存器分配算法，这种全局分配算法必须要能有效地从中间代码的控制流中获取变量的活跃信息——活跃变量分析（**liveliness analysis**）恰好可以为我们提供这些信息。如何进行活跃变量分析我们稍后进行介绍，现在假设在进行过这种分析之后我们已经得到了在每一个程序点上哪些变量在将来的控制流中可能会被使用到。一个显而易见的寄存器分配原则是，同时活跃的两个变量尽量不要分配相同的寄存器。这是因为同时活跃的变量都可能在之后的运行过程中被用到，如果把它们分到一起那么很可能会产生寄存器内变量的换入换出操作，增加访存代价。不过这里存在两个特例：

- ❖ 在赋值操作  $x := y$  中，即使  $x$  和  $y$  在该条代码之后都活跃，因为二者值是相等的，因此它们仍然可以共用寄存器
- ❖ 在类似于  $x := y + z$  这样的中间代码里，如果变量  $x$  在该条代码之后不再活跃，但变量  $y$  在之后仍然活跃，那么此时虽然  $x$  和  $y$  不同时活跃，二者仍然要避免共用寄存器从而防止对  $x$  的赋值会将活跃变量  $y$  在寄存器中的值覆盖掉。

据此我们定义，两个不同变量  $x$  和  $y$  相互**干扰**（**interference**）的条件为：

- ❖ 存在一条中间代码  $i$ ，满足  $x \in \text{out}[i]$  且  $y \in \text{out}[i]$
- ❖ 或者存在一条中间代码  $i$ ，这条代码不是赋值操作  $x := y$  或  $y := x$ ，且满足  $x \in \text{def}[i]$  且  $y \in \text{out}[i]$

其中  $\text{out}[i]$  与  $\text{def}[i]$  都是活跃变量分析所返回给我们的信息，它们的具体含义后面会有介绍，建议阅读完后文活跃变量分析一节之后再返回来仔细考察这个定义。这里你只需要弄明白， $x$  和  $y$  相互干扰就意味着我们应当尽可能地为二者分配不同的寄存器即可。

如果将中间代码中出现的所有变量/临时变量看作顶点，两个变量之间若相互干扰则在二者所对应的顶点之间连一条边，那么我们就可以得到一张干涉图（**interference graph**）。而如果此时我们为每一个变量都分配一个固定的寄存器，将处理器中的  $k$  个寄存器看成  $k$  种颜色，又要求干涉图中相邻两顶点不能染同一种颜色，那么寄存器分配问题就变成了一个图染色（**graph-coloring**）问题。对于一个固定的颜色数  $k$ ，判断一张干涉图是否能被  $k$  着色是一个 **NP-Complete** 问题，因此为了能够在多项式时间内得到寄存器分配结果我们只能使用一些启发式算法对干涉图进行着色。一个比较简单的启发式染色算法（称作 **Kempe** 算法）为：

- ❖ 如果干涉图中包含度小于等于  $k-1$  的顶点，就将这个顶点压入一个栈里并从干涉图中删除。这样做的意义在于，如果我们能够为删除该顶点之后的那张图找到一个  $k$  着色的方案，那么原图也一定是  $k$  可着色的（为什么？）。删掉该顶点可以对原问题进行简化。
- ❖ 重复执行上述操作，如果最后干涉图中只剩下了少于  $k$  个顶点，那么此时就可以为剩下的每个顶点分配一个颜色，然后依次弹出栈中的顶点添加回干涉图中，并选择它的邻居都没有使用过的颜色对弹出的顶点进行染色。
- ❖ 当我们删顶点删到某一步时，如果干涉图中所有的顶点都至少包含了  $k$  个邻居，此时能否断定原图不能被  $k$  着色呢？如果你能证明这一点的话，你就相当于构造性地证明  $P=NP$  了。事实上，这样的图在某些情况下仍然是  $k$  可着色的。如果出现了干涉图中的所有顶点都至少为  $k$  度，我们仍然选择一个顶点删除并且压栈，并且标记这样的顶点为待溢出的顶点，之后继续删点操作。



❖ 被标记为待溢出的顶点在最后被弹栈时，如果我们足够幸运，有可能它的邻居总共被染了少于  $k$  种颜色（为什么？）。此时我们就可以成功地为该顶点染色并清除这个它的溢出标记。否则，我们无法为这个顶点分配一个颜色，它所代表的变量也就必须要被溢出到内存中了。

到这里，中间代码里出现的所有变量要么被分配了一个寄存器，要么被标记为溢出。现在的问题是，那些被标记为溢出的变量的值在参与运算时仍然需要临时被载入到某个寄存器里，在运算结束后也仍然需要某个寄存器临时保存要溢出到内存里的值，这些临时使用的寄存器从哪里来呢？当然，最简单的解决方法是在进行前面图染色算法之前预留出专门用来临时存放溢出变量的值的寄存器。如果你觉得这样做比较浪费寄存器资源，想要追求更有效率的分配方案，你可以通过不断地引入更多的临时变量重写中间代码、重新进行活跃变量分析和图染色来不断减少需要溢出的变量的个数，直到所有的溢出变量全部被消除掉。另外，对于干涉图中那些不相邻的顶点，我们还可以通过合并（coalesce）顶点操作显式地令这些不互相干扰的变量共用同一个寄存器。引入合并以及溢出变量这两种机制会使得全局寄存器分配算法变得更加复杂，想要了解具体细节的话请自行查阅资料。限于篇幅，本文对于全局寄存器分配算法的讨论只能说是点到为止，以期抛砖引玉。

## 活跃变量分析

之前在图染色算法中，为了构造干涉图我们需要明确变量与变量之间的干扰关系，而为了得到干扰关系又需要知道哪些变量是同时活跃的。这一节我们就来讨论如何得到变量在中间代码里的活跃信息。首先来严格定义一下什么叫活跃变量：称变量  $x$  在某一特定的程序点是活跃变量当且仅当：

- ❖ 如果某条中间代码使用到了变量  $x$  的值，则  $x$  在这条代码运行之前是活跃的。
- ❖ 如果变量  $x$  在某条中间代码中被赋值，并且  $x$  没有被该条代码使用到，则  $x$  在这条代码运行之前是不活跃的。
- ❖ 如果变量  $x$  在某条中间代码运行之后是活跃的，而这条中间代码并没有给  $x$  赋值，则  $x$  在这条代码运行之前是活跃的。
- ❖ 如果变量  $x$  某条中间代码运行之后是活跃的，则  $x$  在这条中间代码运行之后可能跳转到的所有的中间代码运行之前都是活跃的。

上述四条规则中，第一条规则指出了活跃变量是如何产生的，第二条规则指出了活跃变量是如何消亡的，第三和第四条规则指出了活跃变量是如何传递的。

定义第  $i$  条中间代码的后继集合  $\text{succ}[i]$  为：

- ❖ 若第  $i$  条中间代码为无条件跳转语句 `GOTO`，且跳转目标为第  $j$  条中间代码，则  $\text{succ}[i] = \{ j \}$
- ❖ 若第  $i$  条中间代码为条件跳转语句 `IF`，且跳转目标为第  $j$  条中间代码，则  $\text{succ}[i] = \{ j, i+1 \}$
- ❖ 若第  $i$  条中间代码为返回语句 `RETURN`，则  $\text{succ}[i] = \emptyset$
- ❖ 若第  $i$  条中间代码为其他类型的语句，则  $\text{succ}[i] = \{ i+1 \}$

再定义  $\text{def}[i]$  为被第  $i$  条中间代码赋值了的变量的集合， $\text{use}[i]$  为被第  $i$  条中间代码使用到的变量的集合， $\text{in}[i]$  为在第  $i$  条中间代码运行之前活跃着的变量的集合， $\text{out}[i]$  为在第  $i$  条中间代码运行之后活跃着的变量的集合。正如理论课上学到的那样，活跃变量分析问题可以转化为解下述数据流方程的问题：

$$\begin{aligned}\text{in}[i] &= \text{use}[i] \cup (\text{out}[i] - \text{def}[i]) \\ \text{out}[i] &= \bigcup_{j \in \text{succ}[i]} \text{in}[j]\end{aligned}$$

我们可以通过迭代的方法对这个数据流方程进行求解。算法开始时我们令  $\forall i, \text{in}[i] = \emptyset$ ，之后每一条中间代码对应的  $\text{in}$  和  $\text{out}$  集合按照上式进行运算，直到这两个集合的运算结果收敛为止。格理论告诉我们， $\text{in}$  和  $\text{out}$  集合的运算顺序不影响数据流方程解的收敛性，但会影响解的收敛速度。对于上述数据流方程而言，按照  $i$  从大到小的顺序来计算  $\text{in}$  和  $\text{out}$  往往要比按照  $i$  从小到大的顺序进行计算在收敛速度上要快很多。

为了能够更加高效地对集合  $\text{in}$  和  $\text{out}$  进行计算，在实际实现时我们往往采用位向量（bit vector）来表示这两个集合。假设待处理的中间代码包含了 10 个变量/临时变量，那么  $\text{in}[i]$ （或者  $\text{out}[i]$ ）可以分别由 10bits 组成，其中第  $j$  个比特位为 1 就代表第  $j$  个变量属于  $\text{in}[i]$ （或者  $\text{out}[i]$ ）。两个集合之间的并集对应位向量中的或运算，两个集合之间的交集对应位向量中的与运算，一个集合的补集对应位向量中的非运算。位向量表示法凭借其表示紧凑、运算速度快的特点，几乎成为了解数据流方程所采用数据结构的不二之选。

实际上，数据流方程这一强大的工具不仅可以用于活跃变量分析，也可以用在诸如到达定值（reaching definition）、可用表达式（available expression）等各种与代码优化有关的分析中。另外我们上面介绍的方法是以语句为单位来进

行分析，而类似的方法也适用于以基本块为单位的情况，并且使用基本块的效率还会更高一些。有关数据流方程的其他应用，教材上有非常详细的讨论，理论课上也会讲到，这里就不再赘述了。

## MIPS 寄存器的使用

在结束对寄存器分配问题的讨论之前，我们还需要了解 MIPS 指令集对于寄存器的使用有哪些规范，从而明确哪些寄存器可以随使用、哪些不能用、哪些可以用但要小心。严格地讲，采用 MIPS 体系结构的处理器本身并没有强制规定其 32 个通用寄存器应该如何使用（除了 \$0 之外，其余 31 个寄存器在硬件上都是等价的），但 MIPS 标准对于汇编代码的书写的确是提出了一些约定（convention）。“约定”这个词有两层意思：其一，因为它是约定，所以没有人逼你去遵守它，你大可以悍然违反它却仍然使你写出的汇编代码能够正常运行起来；其二，因为它是约定，所以除了你之外的绝大部分人（还有包括 SPIM 在内的绝大多数汇编器）都在遵守它，如果你不遵守它那么你的程序不仅在运行效率以及可移植性等方面会遇到各种问题，同时也可能无法被 SPIM 所正常执行。

\$0 这个寄存器非常特殊，它在硬件上本身就是接地的，因此其中的值永远是 0，我们无法改变。\$at、\$k0、\$k1 这三个寄存器是专门预留给汇编器使用的，如果你尝试在汇编代码中访问或者修改他们的话 SPIM 是会报错的。\$v0 和 \$v1 这两个寄存器专门用来存放函数的返回值，在函数内部也可以使用不过要注意在当前函数返回或者调用其它函数时应妥善处理这两个寄存器中原有的数据。\$a0~\$a3 四个寄存器专门用于存放函数参数，在函数内部它们可以视作与 \$t0~\$t9 等同。

\$t0~\$t9 这 10 个寄存器可以由我们任意使用，但要注意它们属于调用者保存的寄存器，在函数调用之前如果其中保存有任何有用的数据都要先溢出到内存之中。\$s0~\$s7 也可以任意使用，不过它们是被调用者保存的寄存器，如果一个函数内要修改 \$s0~\$s7 的话，需要在函数开头先将其中原有的数据压入栈，并在函数末尾恢复这些数据。关于调用者保存和被调用者保存这两种机制，后文会详细介绍。

\$gp 固定指向 64K 静态数据区的中央，\$sp 固定指向栈的顶部。这两个寄存器都是具有特定功能的，对它们的使用和修改必须伴随明确的语义，不能随随便便地将任何数据都往里送。\$30 这个寄存器比较特殊，有些汇编器将其作为 \$s8 使用，也有一些汇编器将其作为栈帧指针 \$fp 使用，你可以在这两个方案里任选其一。\$ra 专门用来保存函数的返回地址，MIPS 中与函数跳转有关的 jal 指令和 jr 指令都会对这个寄存器进行操作，因此我们也一定不要随便去修改 \$ra 的值。

总而言之，MIPS 的 32 个通用寄存器里能让我们随意使用的有 \$t0~\$t9 以及 \$s0~\$s8，不能随意使用的有 \$at、\$k0、\$k1、\$gp、\$sp 和 \$ra，可以使用但在某些情况下需要特殊处理的有 \$v0~\$v1 以及 \$a0~\$a3。

## 栈管理

在过程式程序设计语言中，函数调用包括控制流转移和数据流转移两个部分。控制流转移指的是将程序计数器 PC 当前的值保存到 \$ra 中然后跳转到目标函数的第一句处，这件事情已经由硬件帮我们做好，我们可以直接使用 jal 指令实现。因此，编译器编写者在目标代码生成所需要考虑的问题是如何在函数的调用者与被调用者之间进行数据流的转移。当一个函数被调用时，调用者需要为这个函数传递参数，然后将控制流跳转到被调用函数的第一行代码处；当被调用函数返回时，被调用者需要将返回值保存到某个位置，然后将控制流跳转回调用者处。MIPS 中，函数调用使用 jal 指令，函数返回使用 jr 指令。参数传递采用寄存器与栈相结合的方式：如果参数少于 4 个，则使用寄存器 \$a0~\$a3 四个寄存器传递参数；如果参数多于 4 个，则前 4 个参数保存在 \$a0~\$a3 中，剩下的参数依次压到栈里。返回值的处理方式则比较简单，由于我们假定 C++ 中所有函数只能返回一个整数，因此直接将返回值放到 \$v0 里即可，\$v1 可以挪作他用。

思考：如果允许返回值是一个结构体变量，这个结构体变量的规模又比较大，那么将返回值放到哪里比较好？

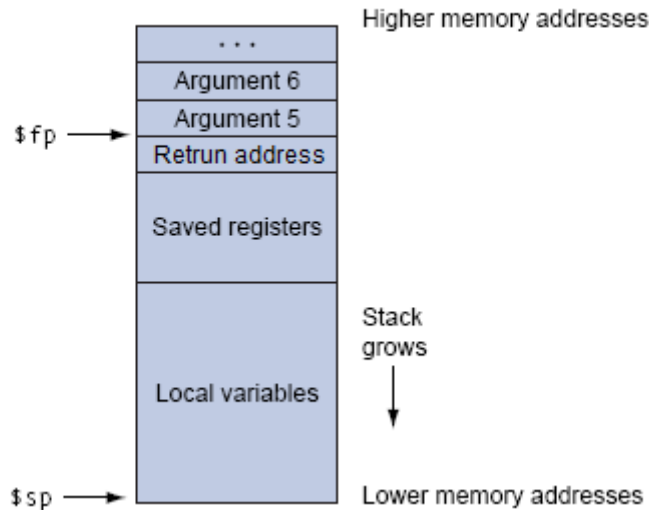
下面着重来讨论在函数调用过程中至关重要的结构——栈。栈在本质上就是按照后进先出原则维护的一块内存区域。除了上面提到的参数传递之外，栈在程序运行过程中还具有如下功能：

- ❖ 如果我们在一个函数中使用 jal 指令调用了另一个函数，寄存器 \$ra 中的内容就会被覆盖掉。为了能够使得另一个函数返回之后能将 \$ra 中原来的内容恢复出来，调用者在进行函数调用之前需要负责把 \$ra 暂存起来，而这暂存的位置自然是在栈中。
- ❖ 对于那些在寄存器分配过程中需要溢出到内存中的变量来说，它们究竟要溢出到内存中的什么地方呢？如果是

全局变量，则需要被溢出到静态数据区；如果是局部变量，则一般会被溢出到栈中。为了简化处理，本次实习假设输入程序中不存在全局变量，因此你的编译器应该将所有需要被溢出的变量都安排到栈上。

❖ 不管占用多大的内存空间，数组和结构体一定会被分配到内存中去。同溢出变量一样，这些内存空间实际上都在栈上。

每一个函数在栈上都会占用一块单独的内存空间，这块内存一般被称为活动记录（activation record）或者栈帧（stack frame）。不同函数的活动记录虽然在占用内存大小上可能会有不小的差异，但基本结构都差不多。一个比较典型的活动记录结构如下：



其中栈指针\$sp 指向栈的顶部，帧指针\$fp 指向当前活动记录的底部。假设图中栈顶为上方而栈底为下方，则\$fp 之下是传给本函数的参数（只有多于 4 个参数时这里才会有内容），而\$fp 之上是返回地址、被调用者保存的寄存器内容以及局部数组、变量/临时变量。这个活动记录的布局并不是唯一可行的，不同的体系结构之间布局不尽相同（例如 x86 体系结构的 call 语句会自动将%eip 压到栈顶，因此与上图相比 x86 中帧指针应该指在图中返回地址以下），同一体系结构的不同教材之间介绍的布局甚至都不一样，可见这里并没有一个统一的标准。理论上讲只要能将该应该保存的内容都存下来，并且能够正确地将它们都取出来就行。你大可以不必完全遵循上图中的布局方式。

关于帧指针\$fp 这里再多说两句。在栈的管理中，有一个栈指针\$sp 其实已经足够了，\$fp 并不是必需的，前面也提到过某些编译器甚至将\$fp 挪用作\$ss。之所以要引入\$fp 主要是为了方便访问活动记录中的内容：在函数的运行过程中，\$sp 是会经常发生变化的（例如，当你想要压入新的临时变量、压入将要调用的另一个函数的参数、或者想在栈上保存动态大小的数组时），根据\$sp 来访问栈帧里保存的局部变量比较麻烦，因为这些局部变量相对于\$sp 的偏移量会经常改变。而在函数内部\$fp 一旦确定就不再变化，所以根据\$fp 访问局部变量时并不需要考虑偏移量的变化问题。假如你学过有关 x86 汇编的知识就会发现，MIPS 中的\$sp 实际上相当于 x86 中的%esp，而 MIPS 中的\$fp 则相当于 x86 中的%ebp。如果决定使用\$fp 的话，为了能够使本函数返回之后能够恢复上层函数的\$fp 需要在活动记录中找地方把\$fp 中的旧值也存起来——正如 x86 汇编里每进入一个函数体的头两句话一般是 push %ebp 和 movl %esp, %ebp 一样<sup>4</sup>。

如果一个函数 f 调用了另一个函数 g，我们称函数 f 为调用者（caller），函数 g 为被调用者（callee）<sup>5</sup>。控制流从调用者转移到被调用者之后，由于被调用者使用到一些寄存器，而这些寄存器中有可能原先保存着有用的内容，故被调用者在使用这些寄存器之前需要先将其中的内容保存到栈里，等到被调用者返回之前再从栈里将这些内容恢复出来。现在的问题是：保存寄存器中原有数据这件事情究竟由调用者完成还是由被调用者完成呢？如果由调用者保存，由于调用者事先不知道被调用者会使用到哪些寄存器，它只能将所有的寄存器内容全部保存，于是会产生一些无用的压栈/弹栈操作；如果由被调用者保存，由于被调用者事先不知道调用者在调用自己之后有哪些寄存器不需要了，它同样只能将所有的寄存器内容全部保存，于是同样会产生一些无用的压栈/弹栈操作。为了减少这些无用的访存操作，可以采用一种调用者和被调用者共同保存的策略：MIPS 约定\$t0~\$t9 由调用者负责保存，而\$ss~\$s8 由被调用者负责保存。从调用关系的角度看，调用者负责保存的寄存器中的值在过程调用前后有可能会发生改变，

<sup>4</sup>Gcc 在默认情况下会为每一个函数都生成这样两句代码。但当我们在编译时打开了任何一个级别的-O 选项之后，为了提高输出代码的效率，Gcc 也会放弃将%ebp 作为帧指针使用。另外，在 64 位扩展后的 x86-64 体系结构中，%rbp 也不再被用作帧指针。

<sup>5</sup>调用者和被调用者是一个相对的概念。如果函数 f 被其它函数所调用，那么相对于那个函数 f 又会成为被调用者；如果函数 g 调用了其它函数，那么相对于那个函数 g 又会成为调用者。



被调用者负责保存的寄存器中的值在过程调用的前后则一定不会发生改变。这也就启示我们， $\$t0 \sim \$t9$  应该尽量分配给那些短期使用的变量/临时变量， $\$s0 \sim \$s9$  应当尽量分配给那些生存期比较长，尤其是生存期跨越了函数调用的变量/临时变量。

在 C 风格的 x86 汇编中，Gcc 规定  $\%eax$ 、 $\%ecx$ 、 $\%edx$  三个寄存器为调用者保存，而  $\%ebx$ 、 $\%esi$ 、 $\%edi$  三个寄存器则为被调用者保存。不过由于有方便的 `pushal` 和 `popal` 指令的存在，在人工书写汇编代码时人们常常将这 6 个通用寄存器全部作为被调用者保存。

先考虑调用者的过程调用序列（procedure call sequence）。首先，调用者  $f$  在调用函数  $g$  之前需要将保存着活跃变量的所有调用者保存寄存器  $live_1, live_2, \dots, live_k$  写到栈中，之后将参数  $arg_1, arg_2, \dots, arg_n$  传入寄存器或者栈。在函数调用结束后，依次将之前保存的内容从栈中恢复出来。上述整个过程如下所示：

```
sw live1, offsetlive1($sp)
...
sw livek, offsetlivek($sp)
subu $sp, $sp, max{0, 4 × (n-5)}
move $a0, arg1
...
move $a3, arg4
sw arg5, 0($sp)
...
sw argn, (4 × (n-5))($sp)
jal g
addi $sp, $sp, max{0, 4 × (n-5)}
lw live1, offsetlive1($sp)
...
lw livek, offsetlivek($sp)
```

上面这份代码假设所有参数在函数调用之前都已经保存在了寄存器里。但在实际编译的过程中，如果函数  $g$  的参数很多的话，可以逐个进行参数计算以及压栈。不过如果多个参数是被逐个压栈的，那么在一个参数压栈后再计算下一个参数时，由于  $\$sp$  已经发生了变化，当前活动记录内所有变量相对于  $\$sp$  的偏移量都会发生变化！如果想要避免这个问题，前文也提到过解决方法，那就是使用帧指针  $\$fp$  而不是栈指针  $\$sp$  对当前活动记录中的内容进行访问。

再来看被调用者的过程调用序列。被调用者的调用序列分为两个部分，分别在函数开头和函数结尾。我们将函数开头的那部分调用序列称为 Prologue，在函数结尾那部分调用序列称为 Epilogue。在 Prologue 中，我们首先要负责布置好本函数的活动记录。如果本函数内部还要调用其它函数，则需要将  $\$ra$  压栈；如果用到了  $\$fp$ ，还要将  $\$fp$  压栈并设置好新的  $\$fp$ 。随后，将本函数内所要用的所有被调用者保存的寄存器  $reg_1, reg_2, \dots, reg_k$  存入栈，最后将调用者由栈中传入的实参作为形参  $p_5, p_6, \dots, p_n$  取出。整个过程如下所示<sup>6</sup>：

```
subu $sp, $sp, framesizeg
sw $ra, (framesizeg - 4)($sp)
sw $fp, (framesizeg - 8)($sp)
addi $fp, $sp, framesizeg
sw reg1, offsetreg1($sp)
...
sw regk, offsetregk($sp)
lw p5, (framesizeg)($sp)
...
lw pn, (framesizeg + 4 × (n-5))($sp)
```

在 Epilogue 中，我们需要将函数开头保存过的寄存器恢复出来，然后将栈恢复原样：

<sup>6</sup>蓝字部分只有在函数内部调用了其它函数才会用到，红字部分只有在使用了  $\$fp$  时才会用到



```
lw reg1, offsetreg1($sp)
...
lw regk, offsetregk($sp)
lw $ra, (framesizeg - 4)($sp)
lw $fp, (framesizeg - 8)($sp)
addi $sp, $sp, framesizeg
jr $ra
```

与前面一样，在设置好\$fp 之后，对活动记录内部数据的访问也可以根据\$fp 以及这些数据相对于\$fp 的偏移量来进行，不必去使用\$sp。

思考：如果足够细心，你会发现每当我们向栈中压入数据时，总是最先修改\$sp；每当我们从栈中弹出数据时，总是最后修改\$sp。例如，向栈中压入\$ra 时，我们会先写 `subu $sp, $sp, 4` 再写 `sw $ra, 4($sp)`。问题是，这个顺序可否调换一下？即我们能不能先写 `sw $ra, -4($sp)` 再写 `subu $sp, $sp, 4` 呢？为什么？

最后我们来简单讨论一下函数调用对寄存器分配算法有什么影响。由于被调用者保存的寄存器\$*s0*~\$*s8* 在函数调用前后由被调用者保证其内容不发生变化，因此我们不需要对它们特殊考虑。而调用者保存的寄存器\$*t0*~\$*t9* 在函数调用之后其中的内容会全部丢失，所以这些寄存器才是函数调用对于寄存器分配过程影响最大的地方。如果采用了局部寄存器分配算法，那么在处理到中间代码 CALL 时，如果\$*t0*~\$*t9* 中保存有任何变量，那么你就需要在调用序列中将所有的这些变量全部溢出到内存中，等到调用结束再重新将溢出的变量读取回来。当然，如果你觉得这样做比较麻烦，更简单的做法是将中间代码 CALL 单独作为一个基本块进行处理。由于将所有变量溢出到内存这件事情在上一个基本块结束时已经做过了，故到了 CALL 语句这里我们几乎可以不做任何事。如果采用了全局寄存器分配算法，你需要在图染色阶段避免为那些在 CALL 语句处活跃的变量染上代表\$*t0*~\$*t9* 之中任何寄存器的颜色。这样一来，我们的算法会自动地为那些生存期跨越函数调用的变量去分配\$*s0*~\$*s8*。如果这样的变量多于被调用者保存的寄存器个数，则算法会自动将多出来的变量溢出到内存。这样一来在调用者的调用序列中我们甚至都不需要专门将\$*t0*~\$*t9* 压栈，因为里面保存的内容在函数调用之后一定是不活跃的！

在结束本文的主要内容之前我们简单解释一下为什么我们的目标代码不采用 Intel x86 ISA 而采用了 MIPS。如果你对 x86 足够了解的话，你会发现这个 ISA 对于汇编程序员可能是友好的，但对编译器的书写者来说则是极不友好的——凡是你能够想得到的牵扯到目标代码生成与优化的问题，x86 基本上都会把本来就已经不容易的事情变得更糟：首先，它是一个 CISC 指令集，并且大部分指令中的操作数都是可以访存的，因此在指令选择这个问题上要比 RISC 指令集困难很多；其次，它只有 8 个通用寄存器（其中还有 1 个%esp 作为栈指针、1 个%ebp 作为帧指针不能随便用），而实践表明采用图染色的全局寄存器分配算法只有在可用的通用寄存器数目达到或超过 16 个时才能产生出令人满意的寄存器分配方案；第三，它的很多指令本身和通用寄存器并不是独立的，例如乘法指令 mul 的一个操作数必须是%eax，而且乘积会同时覆盖掉%eax 和%edx 两个寄存器的值，这迫使我们编写编译器时必须对像 mul 这样的指令单独进行处理；第四，x86 对于浮点数的支持太差，其 x87 浮点数扩展指令简直恶心到令人发指，这一情况直到 SSE2 指令集出来以后才有所缓解。因此，对于以教学为目的的我们而言，x86 的复杂性有些过大了。

事实上，x86 是一个相当具有历史沧桑感的 ISA，Hennesy 教授称 “This instruction set architecture is one only its creators could love”：在其他现代 ISA 都已经采用分页机制时，x86 还在支持分段；在其他现代 ISA 都全面转向通用寄存器时，x86 还残留着累加器的一些特性；在其他现代 ISA 都放弃栈式体系结构时，x87 浮点数操作还是在栈上完成的。你可能会问，为什么沧桑到可以说有些落伍的 x86 还能在现在的桌面市场上占据统治地位呢？我只能说，在桌面甚至是服务器领域一款处理器的性能高低并不完全取决于 ISA 的好坏，而这款处理器在市场上是否成功与 ISA 的关系则更少。不过在嵌入式领域中，x86 的某些糟糕设计所带来的影响已经开始凸现出来，ARM 之所以今天能在嵌入式领域做得这么风生水起，一定程度上也归功于其 ISA 出现得更晚、设计理念更先进的缘故。

## 如何完成本次实习

本次实习需要你在上一次实习的基础上完成。在动手写代码之前，强烈建议你先熟悉一下 SPIM 的使用方法，然后自己写几个简单的 MIPS 汇编程序送到 SPIM 中运行一下，确定自己已经很清楚 MIPS 代码应该如何书写。毕竟如果你自己都不知道 MIPS 汇编该怎么写，你又如何编写程序帮助自己输出 MIPS 代码呢？

完成本次实习的第一步是确定指令选择机制以及寄存器分配算法。指令选择算法比较简单，其功能甚至可以由

中间代码的打印函数稍加修改得到。寄存器分配算法则可能需要你先定义一系列数据结构。如果采用了局部寄存器分配算法，你需要考虑如何实现寄存器描述符和变量描述符。如果使用本文介绍的局部分配算法，你只需要保存每个寄存器是否空闲、每个变量下次被使用到的位置是哪里即可；而如果使用教材上的局部分配算法，你可能需要记录每一个寄存器中保存了哪些变量，以及每一个变量的有效值位于哪一个寄存器中，在这种情况下我们建议使用位向量作为寄存器描述符和变量描述符的数据结构。如果采用了全局寄存器分配算法，你需要考虑如何实现位向量与干涉图。无符号的整型（动态）数组一般可以用来表示位向量，而邻接表则非常适合作为像干涉图这种需要经常访问某个顶点的所有邻居的图结构。

确定算法之后就可以开始动手实现了。一开始我们可以无视与函数调用有关的 `CALL`、`ARG`、`PARAM`、`RETURN` 等语句，专心处理其他类型的中间代码。你可以先假设寄存器有无限多个（编号 `$t0`, `$t1`, ..., `$t99`, `$t100`, ...），试着完成指令选择，然后将经过指令选择之后的代码打印出来看一下是否正确。随后，完成寄存器分配算法，这时你就会开始考虑如何向栈里溢出变量的问题了。当寄存器分配也完成之后，你可以试着自己写几个不带函数调用的 C--测试程序，将编译器输出的目标代码送入 `SPIM` 中运行查看结果是否正确。

如果测试没有问题，请继续下面的内容。你需要首先设计一个活动记录的布局方式，然后完成对 `ARG`、`PARAM`、`RETURN` 和 `CALL` 的翻译。对这些代码的翻译实际上就是一个输出过程调用序列的过程，调用者和被调用者的调用序列要互相配合着来做，这样不容易出现问题。处理 `ARG` 和 `PARAM` 时要注意实参和形参的顺序不要搞错，另外计算实参时如果你没有使用 `$fp` 那么也要注意各临时变量相对于 `$sp` 偏移量的修改。如果调用序列出现问题，请善于利用 `SPIM` 的单步执行功能对你的编译器的输出代码进行调试。

作为本学期最后一次编译实习，本次实习的内容可能会相对地的偏多一些，这主要是由于实习部分需要配合理论课的课程进度而导致的。祝愿大家早日写出自己的编译器，也希望大家在实习结束之后好好复习，争取期末考试能得到一个理想的成绩。

祝你好运！

Our destiny offers not the cup of despair, but the chalice of opportunity. So let us seize it, not in fear, but in gladness.

—— Richard M. Nixon