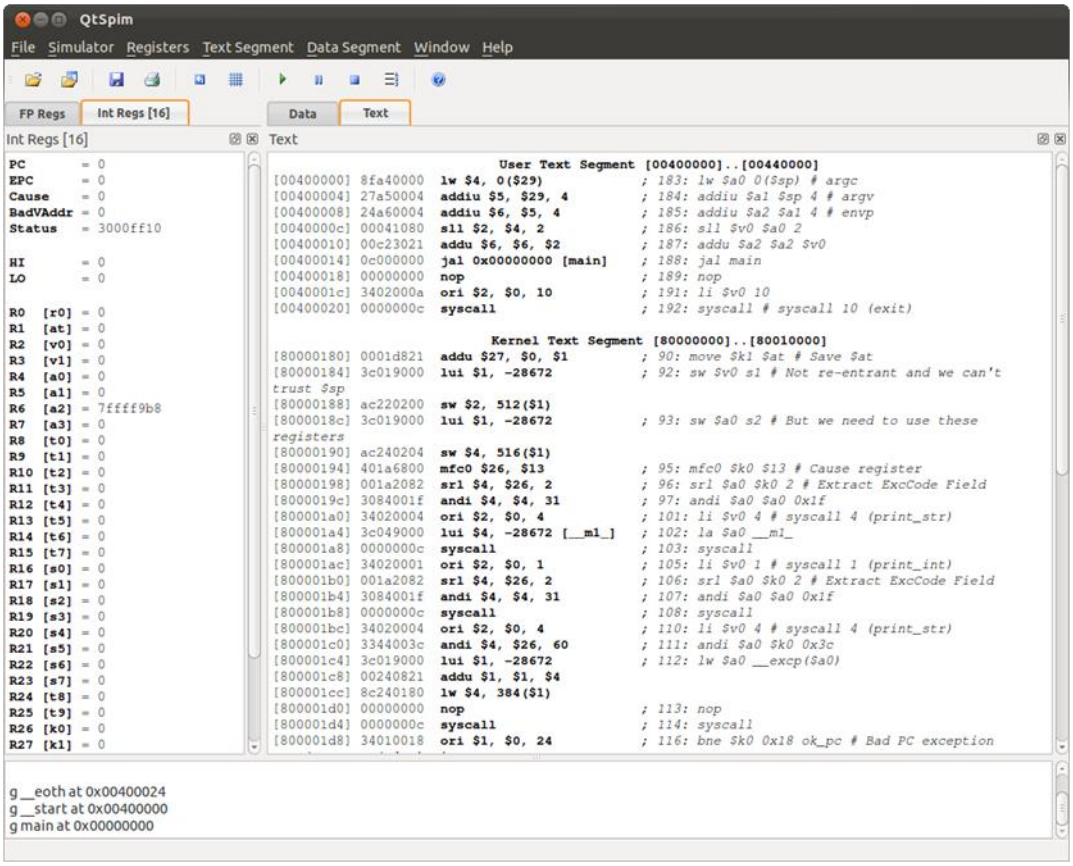

图17. QtSPIM运行界面。

5.2 实验指导

在实验三中，我们已经将输入程序翻译为涉及相当多底层细节的中间代码。这些中间代码在很大程度上已经可以很容易地翻译成许多RISC的机器代码，不过仍然存在以下问题：

1) 中间代码与目标代码之间并不是严格一一对应的。有可能某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。

2) 中间代码中我们使用了数目不受限的变量和临时变量，但处理器所拥有的寄存器数量是有限的。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获得。

3) 中间代码中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中被抽象为若干条ARG语句和一条CALL语句，但在目标机器上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栈来完成这一点。

其中，第一个问题被称为**指令选择（Instruction Selection）**问题，第二个问题被称为**寄存器分配（Register Allocation）**问题，第三个问题则需要考虑如何对栈进行管理。在实验四中我们的主要任务就是编写程序来处理这三个问题。

图18. 内存中的数据信息。

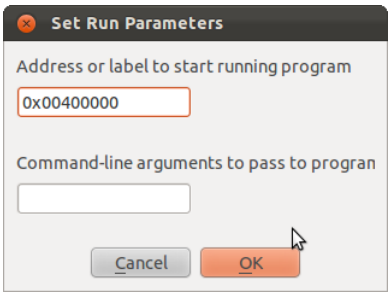

图19. 运行参数设置对话框。

5.2.1 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，让我们先来考察实验四所要用到的工具SPIM Simulator。

SPIM Simulator有两种版本：命令行版和GUI版，这两个版本功能相似。命令行版使用更简洁，GUI版使用更直观，你可以根据自己的喜好进行选择。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（注意需要机器已经连接外网），如果选择GUI版，则需要访问SPIM Simulator的官方地址http://pages.cs.wisc.edu/~larus/spim.html来下载并安装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。

成功安装并运行QtSPIM之后，可以看到如图17所示的界面。其中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，而左侧列出了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。

表7. 数据段中常见的storage_type。

storage_type	描述
.ascii str	存储str于内存中，但不以null结尾。
.asciiz str	存储str于内存中，并以null结尾。
.byte b1, b2, …, bn	连续存储n个字节（8bits位）的值于内存中。
.half h1, h2, …, hn	连续存储n个半字（16bits位）的值于内存中。
.word w1, w2, …, wn	连续存储n个字（32bits位）的值于内存中。
.space n	在当前段分配n个字节的空间。

从图中我们可以看到用户代码区已经存在一部分代码了，该代码的主要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，如果我们运行这段代码，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MIPS32代码（例如前面样例1的输出）保存为后缀名为.s或者.asm的文件。单击QtSPIM工具栏上的按钮来选择我们保存好的文件，此时就可以看到文件中的代码已经被载入到QtSPIM的代码区，再运行这段代码就能在Console窗口观察到运行结果了。你也可以使用QtSPIM工具栏上的按钮或按F10快捷键来单步执行该代码。

使用SPIM Simulator的一个好处就是我们不需要干预内存的分配，它会帮我们自动划分内存中的代码区、数据区和栈区。SPIM Simulator具体采用大端（Big Endian，即数据从高位字节到低位字节在内存中按照从低地址到高地址的顺序依次存储）还是小端（Little Endian，即数据从低位字节到高位字节在内存中按照从低地址到高地址的顺序依次存储）的存储方式取决于你机器的处理器的存储方式（由于大多数台式机或笔记本都使用了Intel x86体系结构的处理器，不出意外的话你会发现自己的SPIM Simulator是小端机）。

在代码区上方的选项卡处切换到“Data”选项卡，就可以看到当前内存中的数据信息，如图18所示。单击菜单栏上的Simulator → Run parameters，在弹出的对话框（如图19所示）中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。

5.2.2 MIPS32汇编代码书写

SPIM Simulator不仅是一个MIPS32的模拟器，也是一个MIPS32的汇编器。想要让SPIM Simulator正常模拟，你首先需要为它准备符合格式的MIPS32汇编代码文本文件。非操作系统内核的汇编代码文件必须以.s或者.asm作为文件的后缀名。汇编代码由若干代码段和若干数据段组成，其中代码段以.text开头，数据段以.data开头。汇编代码中的注释以#开头。

表8. 常用的伪指令。

伪指令	描述	对应的MIPS32指令
li Rdest, imm	把立即数imm（小于等于oxffff）加载到寄存器Rdest中。	ori Rdest, $0, imm
	把立即数imm（大于oxffff）加载到寄存器Rdest中。	lui Rdest, upper(imm)[1] ori Rdest, Rdest, lower(imm)
la Rdest, addr	把地址（而非其中的内容）加载到寄存器Rdest中。	lui Rdest, upper(addr) ori Rdest, Rdest, lower(addr)
move Rdest, Rsrc	把寄存器Rsrc中的内容移至寄存器Rdest中。	addu Rdest, Rsrc, $0
bgt Rsrc1, Rsrc2, label	各种条件分支指令。	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

数据段可以为汇编代码中所要用到的常量和全局变量申请空间，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有表7所列的几类。

下面是三个例子：

```
1  var1: .word 3            # create a single integer variable with
2                           # an initial value of 3
3  array1: .byte 'a','b'    # create a 2-element character array with
4                           # its elements initialized to a and b
5  array2: .space 40        # allocate 40 consecutive bytes, with storage
6                           # uninitialized; could be used as a 40-element
7                           # character array, or a 10-element integer array
```

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。前面的样例输出中有很多像la、li这样的指令。这些指令不属于MIPS32指令集，它们叫**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表8所示。

MIPS体系结构共有32个寄存器，在汇编代码中你可以使用$0至$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，如表9所示。

[1] 表中包含的upper和lower指令并非真实的MIPS32指令，upper(num)表示取一个32位整数num的第16–31位，lower(num)表示取一个32位整数num的第0–15位。

表9. MIPS体系结构中的寄存器。

寄存器编号	别名	描述
$0	$zero	常数0。
$1	$at	(Assembler Temporary) 汇编器保留。
$2 – $3	$v0 – $v1	(Values) 表达式求值或函数结果。
$4 – $7	$a0 – $a3	(Arguments) 函数的首四个参数（跨函数不保留）。
$8 – $15	$t0 – $t7	(Temporaries) 函数调用者负责保存（跨函数不保留）。
$16 – $23	$s0 – $s7	(Saved Values) 函数负责保存和恢复（跨函数不保留）。
$24 – $25	$t8 – $t9	(Temporaries) 函数调用者负责保存（跨函数不保留）。
$26 – $27	$k0 – $k1	中断处理保留。
$28	$gp	(Global Pointer) 指向静态数据段64K内存空间的中部。
$29	$sp	(Stack Pointer) 栈顶指针。
$30	$s8或$fp	MIPS32作为$s8，GCC作为帧指针。
$31	$ra	(Return Address) 返回地址。

表10. 系统调用。

服务	Syscall代码	参数	结果
print_int	1	$a0 = integer	
print_string	4	$a0 = string	
read_int	5		integer (在$v0中)
read_string	8	$a0 = buffer, $a1 = length	
print_char	11	$a0 = char	
read_char	12		char (在$a0中)
exit	10		
exit2	17	$a0 = result	

最后，SPIM Simulator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中存入一个代码以指定具体要进行哪种系统调用。如有必要还需向其它寄存器中存入相关的参数，最后再写一句syscall即可。例如：

```
1  li $v0, 4
2  la $a0, _prompt
3  syscall
```

进行了系统调用print_string(_prompt)。与实验四相关的系统调用类型如表10所示。

到此，如果对照前面的样例输出并仔细阅读这节的内容，你可以基本了解在实验四中你的程序需要输出什么。

5.2.3 指令选择

指令选择可以看成是一个模式匹配问题。无论中间代码是线形还是树形的，我们都需要在

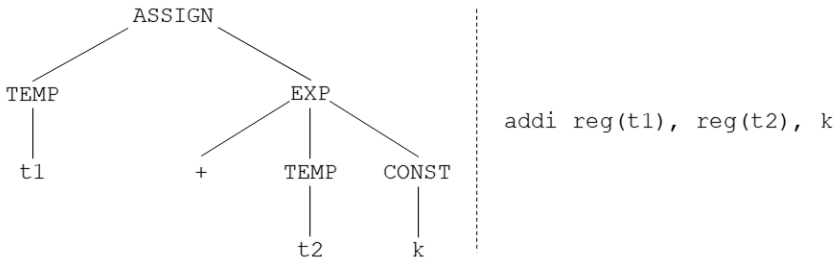

图20. 树形IR翻译示例。

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）。指令选择可以是简单寻找以一一对应，也可以是涉及到许多细节处理和计算的复杂过程。这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。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，因此指令选择在实验四中也属于比较简单的任务。

如果你的程序使用了线形IR，那么最简单的指令选择方式是逐条将中间代码对应到目标代码上。表11是将实验三的中间代码对应到MIPS32指令的一个例子，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目标代码。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$t1中，我们希望将保存在内存中的a[3]的值放到$t2里。如果按照上表使用的逐条翻译的方式，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MIPS32代码也需要两条指令：

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

但这两条指令可以利用MIPS32中的基址寻址机制合并成一条指令：

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

这个例子启示我们，有的时候为了得到更高效的目标代码，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MIPS32代码。这个过程可以看作是一个多行的模式匹配，也可以看成用一个**滑动窗口（Sliding Window）**或**一个窥孔（Peephole）**滑过中间代码并查找可能的翻译方案的过程。这非常类似于我们课本上介绍的**“窥孔优化”（Peephole Optimization）**的局部代码优化技术。

树形IR的翻译方式类似于线形IR，也是一个模式匹配的过程。不过我们需要寻找的模式不再是一句句线形代码，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，这在课本上也有介绍。我们仍用一个例子来说明这种方法，假设我

表11. 中间代码与MIPS32指令对应的一个示例。

中间代码	MIPS32指令
LABEL x:	x:
x := #k	li reg(x)[1], k
x := y	move reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	addi reg(x), reg(y), -k
x := y - z	sub reg(x), reg(y), reg(z)
x := y * z[2]	mul reg(x), reg(y), reg(z)
x := y / z	div reg(y), reg(z) mflo reg(x)
x := *y	lw reg(x), 0(reg(y))
*x = y	sw reg(y), 0(reg(x))
GOTO x	j x
x := CALL f	jal f move reg(x), $v0
RETURN x	move $v0, reg(x) jr $ra
IF x == y GOTO z	beq reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z
IF x > y GOTO z	bgt reg(x), reg(y), z
IF x < y GOTO z	blt reg(x), reg(y), z
IF x >= y GOTO z	bge reg(x), reg(y), z
IF x <= y GOTO z	ble reg(x), reg(y), z

们有一条翻译模式如图20所示。

如何在中间代码中找到该图所对应的模式呢？答案是遍历。我们可以按照深度优先的顺序考察树形IR中的每一个结点及其结节点的类型是否满足相应的模式。例如，图中的模式匹配写成代码可以是：

```
1  if (current_node -> kind == ASSIGN)
2  {
3    left = current_node -> left;
4    right = current_node -> right;
5    if (left->kind == TEMP && right->kind == EXP)
6    {
7      op1 = right -> op1;
8      op2 = right -> op2;
9      if (right->op == '+' && op1->kind == TEMP && op2->kind == CONST)
10       emit_code("addi " + get_reg(left) + ", " + get_reg(op1) + ", "
11         + get_value(op2));
12   }
13 }
```

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。

5.2.4 寄存器分配（朴素寄存器分配算法）

[1] reg(x)表示变量x所分配的寄存器。

[2] 乘法、除法以及条件跳转指令均不支持非零常数，所以如果中间代码包括类似于“x := y * #7”的语句，其中的立即数7必须先加载到一个寄存器中。

RISC机器的一个很显著的特点是，除了load/store型指令之外，其余指令的所有操作数都必须来自寄存器而不是内存。除了数组和结构体必须放到内存中之外，中间代码里的任何一个非零变量或临时变量，只要它参与运算，其值必须被载入到某个寄存器中。在某个特定的程序点上选择哪个寄存器来保存哪个变量的值，这就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。由于寄存器的访问性能远高于内存，这使得寄存器分配算法对于我们编译出来的目标代码的效率的影响尤其明显。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（如多于一个基本块、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[1]。因此，目前编译器使用的寄存器分配算法大都是近似最优分配方案。下面我们将介绍三种不同的寄存器分配算法，它们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。这节我们介绍朴素寄存器分配算法，其它算法以及相关的讨论我们放到后面介绍。

朴素寄存器分配算法的思想最简单，也最低效：将所有的变量或临时变量都放在内存里。如此一来，每翻译一条中间代码之前我们都需要把要用到的变量先加载到寄存器中，得到该代码的计算结果之后又需要将结果写回内存。这种方法的确能将中间代码翻译成可以正常运行的目标代码，而且实现和调试都特别容易，不过它最大的问题是对寄存器的利用率实在太低。它不仅闲置了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。

5.2.5 寄存器分配（局部寄存器分配算法）

寄存器分配之所以难是因为寄存器的数量有限，被迫共用同一寄存器的变量太多，导致这些变量在使用时不得不在寄存器里换入换出，从而产生较大的访存开销。我们考虑如何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。有一种较好的方法叫**局部寄存器分配算法**，该方法会事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课本上有介绍，我们这里不再赘述），在每个基本块内部我们根据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朴素算法一样，需要将本块中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 op y，其中x和y是两个要用到的操作数，z是运算结果，op代表一个任意的二元运

[1] 《Engineering a Compiler》，第2版，Keith D. Cooper和Linda Torczon著，Morgan Kaufmann出版社，第689页，2011年。

算符（一元或多元运算符的处理与二元类似）。在基本块开始时，所有的寄存器都是闲置的。算法的大概框架如下：

```
for each operation z = x op y
  rx = Ensure(x)
  ry = Ensure(y)
  if (x is not needed after the current operation)
    Free(rx)
  if (y is not needed after the current operation)
    Free(ry)
  rz = Allocate(z)
emit MIPS32 code for rz = rx op ry
```

其中，Free(r)表示将寄存器r标记为闲置。算法还用到另外两个辅助函数Ensure和Allocate，它们的实现为：

```
Ensure(x):
  if (x is already in register r)
    result = r
  else
    result = Allocate(x)
    emit MIPS32 code [lw result, x]
  return result

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Allocate函数会用到每个变量在各个程序点的使用信息，这些信息可以由一次对基本块中代码自后向前的扫描而得到。

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使用寄存器，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**溢出或spilling**）时，则选择那个包含本基本块内将来用不到或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。

课本上也介绍了一种和上述算法功能相似的局部寄存器分配函数get_reg。课本上的方法是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。这里介绍的方法与课本上的方法各有优劣，你可以酌情选择，也可以在它们的基础上设计自己的局部寄存器分配算法。

5.2.6 寄存器分配（图染色算法）

局部寄存器分配算法虽然是启发式算法，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，而当前基本块的最后一条中间代码是条件跳转。我们知道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，还是应该继续保留在寄存器里呢?

局部寄存器分配算法的这一弱点启发我们去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，而**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在后面介绍，现在假设我们已经进行过这种分析并了解到了在每个程序点上哪些变量在将来的控制流中可能还会被使用到。一个显而易见的寄存器分配原则就是，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，从而增加访存代价。不过这里存在两个特例:

1) 在赋值操作x := y中，即使x和y在这条代码之后都活跃，因为二者值是相等的，它们仍然可以共用寄存器。

2) 在类似于x := y + z这样的中间代码中，如果变量x在这条代码之后不再活跃，但变量y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以防止之后对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相互干扰的条件为:

1) 存在一条中间代码i，满足x ∈ out[i]且y ∈ out[i]。

2) 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 ∈ def[i]且y ∈ out[i] 。

其中out[i]与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，建议阅读完后面的活跃变量分析一节之后再返回来这里考察这个定义。这里你只需要明白，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。

如果将中间代码中出现的所有变量和临时变量都看作顶点，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）**。如果此时我们为每个变量都分配一个固定的寄存器，而将处理器中的k个寄存器看成k种颜色，我们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，我们只能使用启发式算法来对干涉图进行着色。一个比较简单的启发式染色算法（称作Kempe算法）为：

1) 如果干涉图中包含度小于或等于k-1的顶点，就将该顶点压入一个栈中并从干涉图中删除。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。删掉这类顶点可以对原问题进行简化。

2) 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。

3) 当我们删除顶点到某一步时，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，就相当于构造性地证明P = NP。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所有的顶点都至少为k度，我们仍然选择一个顶点删除并且将其压栈，并且标记这样的顶点为待溢出的顶点，之后继续删点操作。

4) 被标记为待溢出的顶点在最后被弹出栈时，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。此时我们就可以成功地为该顶点染色并清除它的溢出标记。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，中间代码中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中，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？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顶点的操作来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，想要了解具体细节可以自行查阅更多的资料。

5.2.7 寄存器分配（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。现在我们来讨论如何得到变量在中间代码中的活跃信息。首先我们严格定义什么叫活跃变量：称变量x在某一特定的程序点是活跃变量当且仅当：

1) 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。

2) 如果变量x在某条中间代码中被赋值，并且x没有被该代码使用到，则x在这条代码运行之前是不活跃的。

3) 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也是活跃的。

4) 如果变量x在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。

在上述的四条规则中，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。

我们定义第i条中间代码的后继集合succ[i]为：

1) 如果第i条中间代码为无条件跳转语句GOTO，并且跳转的目标是第j条中间代码，则succ[i] = { j }。

2) 如果第i条中间代码为条件跳转语句IF，并且跳转的目标是第j条中间代码，则succ[i] = { j, i + 1 }。

3) 如果第i条中间代码为返回语句RETURN，则succ[i] = ϕ。

4) 如果第i条中间代码为其他类型的语句，则succ[i] = { i + 1 }。

我们再定义def[i]为被第i条中间代码赋值了的变量的集合，use[i]为被第i条中间代码使用到的变量的集合，in[i]为在第i条中间代码运行之前活跃的变量的集合，out[i]为在第i条中间代码运行之后活跃的变量的集合。活跃变量分析问题可以转化为解下述数据流方程的问题：

$in[i] = use[i] \cup (out[i] - def[i])$和$out[i] = \cup_{j \in succ[i]} in[j]$。

我们可以通过迭代的方法对这个数据流方程进行求解。算法开始时我们令所有的in[i]为ϕ，之后每条中间代码对应的in和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，in和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in和out往往要比按照i从小到大的顺序进行计算要快得多。

为了能更高效地对集合in和out进行计算，在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10个变量或临时变量，那么in[i]（或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in[i]（或out[i]）。两个集合之间的并集对应于位向量中的或运算，两个集合之间的交集对应于位向量中的与运算，一个集合的补集对应于位向量中的非运算。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的，而类似的方法也适用于以基本块为单位的情况，并且使用基本块的话分析效率还会更高一些。有关数据流方程的其它应用，课本上有很详细的介绍，我们在这里就不再赘述了。

5.2.8 寄存器分配（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MIPS32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便用、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MIPS32标准对于汇编代码的书写的确是提出了一些约定。“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 Simulator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 Simulator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修改它们的话SPIM Simulator会报错。$v0和$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。$a0至$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$t0至$t9等同。

$t0至$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中。$s0至$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$s0至$s7的话，需要在函数的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，我们会在后面详细介绍。

$gp固定指向64K静态数据区的中央，$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，不能随便将数据往里送。$30这个寄存器比较特殊，有些汇编器将其作为$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，MIPS32中与函数跳转有关的jal指令和jr指令都会对该寄存器进行操作，因此我们也不要随便去修改$ra的值。

总而言之，MIPS的32个通用寄存器中能让我们随意使用的有$t0至$t9以及$s0至$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$v0至$v1以及$a0至$a3，最后$0可用但其值无法修改。

5.2.9 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$ra中然后跳转到目标函数的第一句处，这件事情已经由硬件帮我们做好，我们可以直接使用jal指令实现。因此，编译器编写者在目标代码生成时所需要考虑的问题是如何在函数的调用者与被调用者之间进行数据流的转移。当一个函数被调用时，调用者需要为这个函数传递参数，然后将控制流转移到被调用函数的第一行代码处；当被调用函数返回时，被调用者需要将返回值保存到某个位置，然后将控制流转移回调用者处。在MIPS32中，函数调用使用jal指令，函数返回使用jr指令。参数传递采用寄存器与栈相结合的方式：如果参数少于4个，则使用$a0至$a3这四个寄存器传递参数；如果参数多于4个，则前4个参数保存在$a0至$a3中，剩下的参数依次压到栈里。返回值的处理方式则比较简单，由于我们约定C--中所有函数只能返回一个整数，因此直接将返回值放到$v0中即可，$v1可以挪作它用。

下面我们着重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：

1) 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

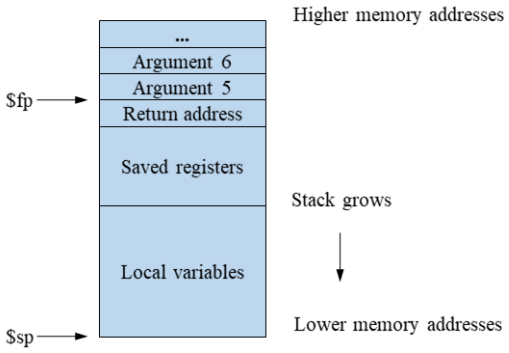

图21. 函数的活动记录结构。

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。

2) 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，实验四中你的程序可以将所有需要被溢出的变量都安排到栈上。

3) 不管占用多大的空间，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。

每个函数在栈上都会占用一块单独的内存空间，这块空间被称为**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所不同，但基本结构都差不多。一个比较典型的活动记录结构如图21所示。

在图中，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、变量或临时变量。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，同一体系结构的在不同的介绍中也可能不一样，这里并没有一个统一的标准。理论上来讲只要能将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。你的程序不必完全遵循上图中的布局方式。

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$fp挪用作$s8。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$fp访问局部变量时并不需要考虑偏移量的变化问题。假如你学过有关x86汇编的知识就会发现，MIPS32中的$sp实际上相当于x86中的%esp，而MIPS32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。

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）**。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中，等到被调用者返回之前再从栈中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是由调用者完成还是由被调用者完成？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和弹栈操作；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，它同样也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和弹栈操作。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：MIPS32约定$t0至$t9由调用者负责保存，而$s0~$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函数调用前后有可能会发生改变，被调用者负责保存的寄存器中的值在函数调用的前后则一定不会发生改变。这也就启示我们，$t0至$t9应该尽量分配给那些短期使用的变量或临时变量，而$s0至$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或临时变量。

类似的，在C风格的x86汇编中，GCC规定%eax、%ecx和%edx这三个寄存器为调用者保存，而%ebx、%esi和%edi这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。

我们先考虑调用者的**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$live_1$、$live_2$、…、$live_k$写到栈中，之后将参数arg_1、arg_2、…、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：

```
1  sw live1, offset_live1($sp)
2  …
3  sw livek, offset_livek($sp)
4  subu $sp, $sp, max{0, 4 * (n - 4)}
```

```
 5  move $a0, arg1
 6  …
 7  move $a3, arg4
 8  sw arg5, 0($sp)
 9  …
10  sw argn, (4 * (n - 5))($sp)
11  jal g
12  addi $sp, $sp, max{0, 4 * (n - 4)}
13  lw live1, offsetlive1($sp)
14  …
15  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寄存器中。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，前面我们也提到过解决方法，那就是使用帧指针$fp而不是栈指针$sp来对当前活动记录中的内容进行访问。

我们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，分别在函数的开头和结尾。我们将函数开头的那部分调用序列称为Prologue，在函数结尾的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，将本函数内所要用到的所有被调用者保存的寄存器reg_1、reg_2、…、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、…、p_n取出。整个过程如下所示[1]：

```
 1  subu $sp, $sp, framesizeg
 2  sw $ra, (framesizeg - 4)($sp)
 3  sw $fp, (framesizeg - 8)($sp)
 4  addi $fp, $sp, framesizeg
 5  sw reg1, offsetreg1($sp)
 6  …
 7  sw regk, offsetregk($sp)
 8  lw p5, (framesizeg)($sp)
 9  …
10  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：

```
1  lw reg1, offsetreg1($sp)
2  …
3  lw regk, offsetregk($sp)
4  lw $ra, (framesizeg - 4)($sp)
5  lw $fp, (framesizeg -8)($sp)
6  addi $sp, $sp, framesizeg
7  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$fp的偏移量来进行，而不必去使用$sp。

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

[1] 第2行代码只有在函数内部调用了其它函数才会用到，第3–4行代码只有在使用了$fp时才会用到。

$s0至$s8在函数调用前后由被调用者保证其内容不会发生变化，因此我们不需要对它们特殊考虑。而调用者保存的寄存器$t0至$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$t0至$t9中保存有任何变量的值，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的值读取回来。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$t0至$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$s0至$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$t0至$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。

最后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MIPS32。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问内存的，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作为栈指针和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。再次，它的很多指令本身并不独立于通用寄存器，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这两个寄存器的值，这迫使我们在编写编译器时必须对像mul这样的指令单独进行处理。最后，x86对于浮点数的支持太差，其x87浮点数扩展指令更是糟糕，这一情况直到SSE2指令集出来以后才有所缓解。因此，对于我们的实验而言，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”。在其它现代ISA都已经采用分页机制时，x86还在支持分段；在其它现代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；在其它现代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着统治地位呢？我们只

能说，在桌面甚至是服务器领域中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之所以在今天能在嵌入式领域做得风生水起，一定程度上也归功于其ISA出现得更晚、而设计理念更先进的缘故。

5.2.10 目标代码生成提示

实验四需要你在实验三的基础上完成。在你动手写代码之前，我们强烈建议你先熟悉SPIM Simulator的使用方法，然后自己写几个简单的MIPS32汇编程序送到SPIM Simulator中运行一下，以确定自己是否已经清楚MIPS32代码应该如何书写。

完成实验四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而得到。寄存器分配算法则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，你可能需要考虑如何实现寄存器描述符和变量描述符。如果使用我们前面介绍的局部寄存器分配算法，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；如果使用课本上介绍的局部寄存器分配算法，你需要记录每个寄存器中保存了哪些变量，以及每个变量的有效值位于哪个寄存器中，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。无符号的整型数组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。

确定了算法之后就可以开始动手实现。开始的时候我们可以无视与函数调用有关的ARG、PARAM、RETURN和CALL语句，专心处理其它类型的中间代码。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。当寄存器分配也完成之后，你可以试着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 Simulator中运行以查看结果是否正确。

如果测试没有问题，请继续下面的内容。你首先需要设计一个活动记录的布局方式，然后完成对ARG、PARAM、RETURN和CALL语句的翻译。对这些中间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 Simulator的单步执行功能对你的编译器输出的代码进行调试。

5.2.11 SPIM安装指导

从SPIM Simulator的官方网站上下载QtSPIM软件（用于目标代码执行），其下载地址为：http://pages.cs.wisc.edu/~larus/spim.html，文件名为：qtspim_9.1.9_linux32.deb。双击该文件以进行安装，在安装过程中会弹出“The package is of bad quality”的提示框，这时选择“Ignore and install”可继续安装。