2.6 其他提示

1. 实验三需要你在实验二的基础上完成。你可以在实验二的语义分析部分添加中间代码生成的内容，使编译器可以一边进行语义检查一边生成中间代码；也可以将关于中间代码生成的所有内容写到一个单独的文件中，等到语义检查全部完成并通过之后再生成中间代码。
2. 确定了在哪里进行中间代码生成之后，下一步就要实现中间代码的数据结构（最好能写一系列可以直接生成一条中间代码的构造函数以简化后面的实现），然后按照输出格式的要求自己编写函数将你的中间代码打印出来。完成之后建议先自行写一个测试程序，在这个测试程序中使用构造函数人工构造一段代码并将其打印出来，然后使用我们提供的虚拟机小程序简单地测试一下，以确保自己的数据结构和打印函数都能正常工作。准备工作完成之后，再继续做下面的内容。
3. 接下来的任务是根据前面介绍的翻译模式完成一系列的translate函数。我们已经给出了Exp和Stmt的翻译模式，你还需要考虑包括数组、结构体、数组与结构体定义、变量初始化、语法单元CompSt、语法单元StmtList在内的翻译模式。
4. 最后，虚拟机小程序将以总共执行过的中间代码条数为标准来衡量你的编译器所输出的中间代码的运行效率。因此如果要进行代码优化，重点应该放在精简代码逻辑以及消除代码冗余上。

实验指导5 目标代码生成

实验五 目标代码生成

> 这是最后一个实验啦！同志们，坚持就是胜利！

5.1 实验要求

在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将C−−源代码翻译为MIPS32指令序列（可以包含伪指令），并在 SPIM Simulator上运行。

当你完成之后，你就拥有了一个自己独立编写、可以实际运行的编译器。

基本要求：

1. 将中间代码**正确**转化为MIPS32汇编代码。

“正确”是指该汇编代码在SPIM Simulator（命令行或Qt版本均可）上运行结果正确。

附加要求：（不额外加分，供有兴趣的同学探索）

1. 对寄存器分配进行优化，如使用局部寄存器分配办法等。
2. 对指令进行优化

化简要求：

- **寄存器的使用与指派可以不必遵循MIPS32的约定。**只要不影响在SPIM Simulator中的正常运行，你可以随意分配MIPS体系结构中的32个通用寄存器，而不必在意哪些寄存器应该存放参数、哪些存放返回值、哪些由调用者负责保存、哪些由被调用者负责保存，等等。
- **栈的管理（包括栈帧中的内容及存放顺序）也不必遵循MIPS32的约定。**你甚至可以使 用栈以外的方式对过程调用间各种数据的传递进行管理，前提是你输出的目标代码“正确”。

5.2 测试样例特征假设

本实验所有测试样例将符合以下假设：

- **假设1**：输入文件中不包含任何词法、语法或语义错误（函数也必有return语句）。
- **假设2**：不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量。
- **假设3**：整型常数都在16bits位的整数范围内，也就是说你不必考虑如果某个整型常数无法在addi等包含立即数的指令中表示时该怎么办。
- **假设4**：不会出现类型为结构体或高维数组（高于1维的数组）的变量。
- **假设5**：没有全局变量的使用，并且所有变量均不重名，变量的存储空间都放到该变量所在的函数的活动记录中。
- **假设6**：任何函数参数都只能是简单变量，也就是说数组和结构体不会作为参数传入某个函数中。
- **假设7**：函数不会返回结构体或数组类型的值。
- **假设8**：函数只会进行一次定义（没有函数声明）。

5.3 输入输出要求

5.3.1 输入要求

你的程序需要能够接收一个输入文件名和一个输出文件名作为参数。 假设你的程序明为compile，输入文件名为test1.cmm，输出文件名为out.ir，你应该这样运行程序。

```
./compile test1.cmm out.s
```

5.3.2 输出格式

程序将运行结果输出到文件。 运行上面的代码，你将在同一文件夹下新生成一个名为out.s的文件。

5.4 测试用例

样例1：

```
int main()
{
  int a = 0, b = 1, i = 0, n;
  n = read();
  while (i < n)
  {
```

```
    int c = a + b;
    write(b);
    a = b;
    b = c;
    i = i + 1;
  }
  return 0;
}
```

期望输出：

```
.data
_prompt: .asciiz ”Enter an integer:”
_ret: .asciiz ”\n”
.globl main
.text
read:
  li $v0, 4
  la $a0, _prompt
  syscall
  li $v0, 5
  syscall
  jr $ra

write:
  li $v0, 1
  syscall
  li $v0, 4
  la $a0, _ret
  syscall
  move $v0, $0
  jr $ra

main:
  li $t5, 0
  li $t4, 1
  li $t3, 0
  addi $sp, $sp, –4
  sw $ra, 0($sp)
  jal read
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t1, $v0
  move $t2, $t1
label1:
  blt $t3, $t2, label2
  j label3
label2:
  add $t1, $t5, $t4
  move $a0, $t4
  addi $sp, $sp, –4
  sw $ra, 0($sp)
  jal write
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t5, $t4
  move $t4, $t1
  addi $t1, $t3, 1
```

```
  move $t3, $t1
  j label1
label3:
        move $v0, $0
  jr $ra
```

样例2：

```
int fact(int n)
{
  if (n == 1){
          return n;
  }
  else
          return (n * fact(n – 1));
}

int main()
{
  int m, result;
  m = read();
  if (m > 1)
          result = fact(m);
  else
    result = 1;
  write(result);
  return 0;
}
```

期望输出：

```
.data
_prompt: .asciiz ”Enter an integer:”
_ret: .asciiz ”\n”
.globl main
.text
read:
  li $v0, 4
  la $a0, _prompt
  syscall
  li $v0, 5
  syscall
  jr $ra

write:
  li $v0, 1
  syscall
  li $v0, 4
  la $a0, _ret
  syscall
  move $v0, $0
  jr $ra

main:
  addi $sp, $sp, –4
  sw $ra, 0($sp)
  jal read
  lw $ra, 0($sp)
```

```
  addi $sp, $sp, 4
  move $t1, $v0
  li $t3, 1
  bgt $t1, $t3, label6
  j label7
label6:
  move $a0, $t1
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal fact
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t2, $v0
  j label8
label7:
        li $t2, 1
label8:
  move $a0, $t2
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal write
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $v0, $0
  jr $ra

fact:
  li $t4, 1
  beq $a0, $t4, label1
  j label2
label1:
  move $v0, $a0
  jr $ra
label2:
  addi $sp, $sp, -8
  sw $a0, ($sp)
  sw $ra, 4($sp)
  sub $a0, $a0, 1
  jal fact
  lw $a0, ($sp)
  lw $ra, 4($sp)
  addi $sp, $sp, 8
  mul $v0, $v0, $a0
  jr $ra
```

5.5 实验指导

在之前的实验中，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：

- 指令选择问题
    - 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。

- 寄存器分配问题
    - 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。
- 函数调用问题（栈管理）
    - 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。

5.5.1 QtSPIM的使用

1. 命令行版（最终检查会使用这个版本）

   安装：`sudo apt-get install spim`

   测试：`spim -file [汇编代码文件名]`

2. GUI版（可以单步调试）

   下载地址：http://pages.cs.wisc.edu/~larus/spim.html

   使用方法：载入程序生成的目标代码文件

5.5.2 MIPS32汇编代码简介

文件特征：以.s或者.asm作为后缀名

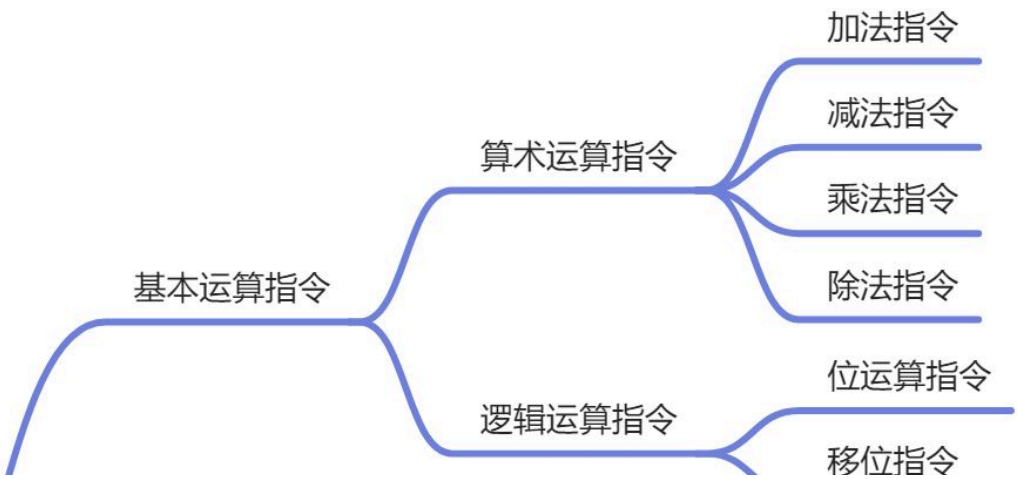
整体结构：

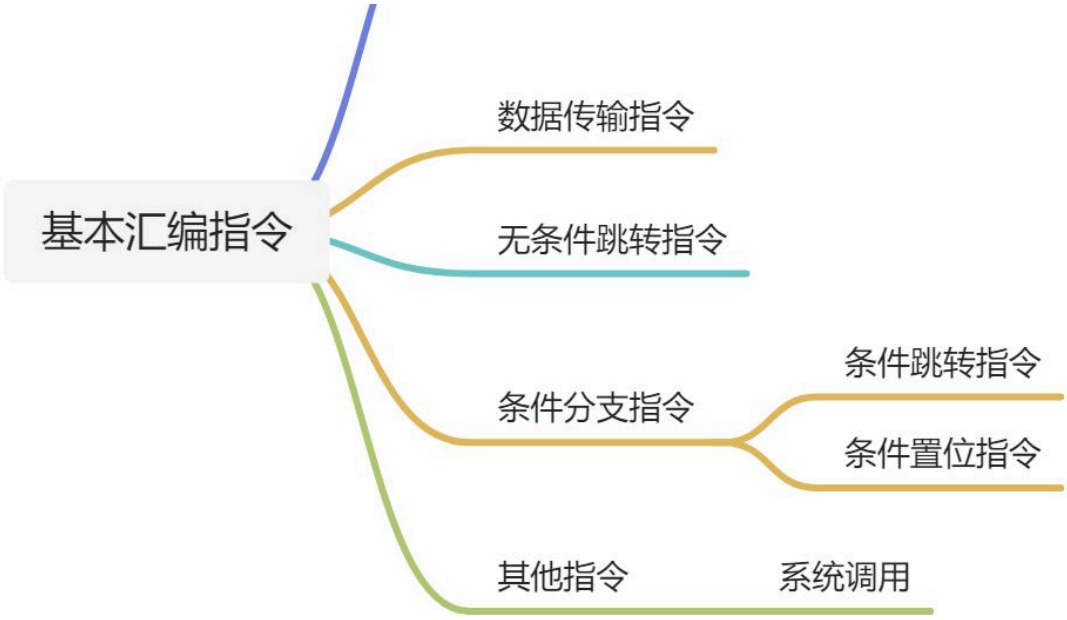
代码段：以.text开头

数据段：以.开头

注释：以#开头

指令：





说明：

1. 算术运算指令:执行加减乘除逻辑运算的操作，可以操作的对象是寄存器或者立即数
2. 数据传输指令:在寄存器、内存之间搬运数据，往寄存器、内存中写入数据
3. 无条件跳转指令：可以跳转到绝对、相对偏移量的位置，也可跳转到寄存器保存的位置
4. 条件分支指令：跳转到label
5. 其他指令：系统调用

参考：

> 附：MIPS32参考资料（包含所有详细资料介绍）
>
> MIPS32_and_SPIM.pdf
>
> 简明参考网址：https://freeflyingsheep.github.io/posts/mips/assembly/

常用伪指令

伪指令：伪指令(Pseudo Instruction)是用于对汇编过程进行控制的指令，该类指令并不是可执行指令，没有机器代码，只用于汇编过程中为汇编程序提供汇编信息。（人话：一些常用的指令组合弄成1个指令表示，便于理解和使用）

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

Hardware Software Interface

Some assemblers also implement *pseudoinstructions*, which are instructions provided by an assembler but not implemented in hardware. Chapter 2 contains many examples of how the MIPS assembler synthesizes pseudoinstructions and addressing modes from the spartan MIPS hardware instruction set. For example, Section 2.6 in Chapter 2 describes how the assembler synthesizes the `blt` instruction from two other instructions: `slt` and `bne`. By extending the instruction set, the MIPS assembler makes assembly language programming easier without complicating the hardware. Many pseudoinstructions could also be simulated with macros, but the MIPS assembler can generate better code for these instructions because it can use a dedicated register (`$at`) and is able to optimize the generated code.

注意：书上给的这段伪指令是错的！应该更正为：

bgt Rsrc1, Rsrc2, label	各种条件分支指令。	sgt $1, Rsrc1, Rsrc2 bne $1, $0, label
bge Rsrc1, Rsrc2, label		sge $1, Rsrc1, Rsrc2 bne $1, $0, label
blt Rsrc1, Rsrc2, label		slt $1, Rsrc1, Rsrc2 bne $1, $0, label
ble Rsrc1, Rsrc2, label		sle $1, Rsrc1, Rsrc2 bne $1, $0, label

为汇编代码中所要用到的常量和全局变量申请空间：

name: storage_type value(s)

`name` 代表内存地址（标签）名， `storage_type` 代表数据类型， `value` 代表初始值。

常见的类型包括：

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。

寄存器简介

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。

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。

系统调用

方便进行控制台交互的机制。

方法：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。

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	

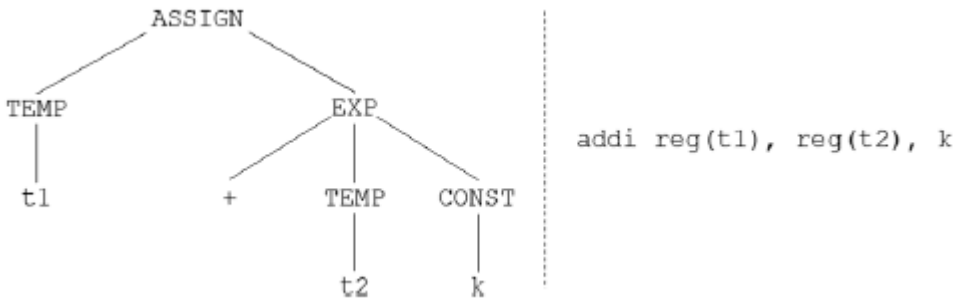
例子：

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

进行了系统调用print_string(_prompt)。

5.5.3 指令选择问题的解决方案

指令选择可以看成是一个模式匹配问题。也就是说，找到中间代码中的“特定结构”，对应地转化到目标代码地“特定结构”上。



对于线性IR：

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。

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

注：

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。

局限性：这种逐条翻译的方式往往得不到高效的目标代码。

对于树形IR：

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：

```
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));
```

```
  }
}
```

5.5.4 寄存器分配的解决方案

（1）朴素寄存器分配算法

思想：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。

优点：实现和调试都特别容易

缺点：对寄存器的利用率实在太低

（2）局部寄存器分配算法

思想：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。

代码框架：

```
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
```

关键函数：

```
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
```

（3）图染色算法

问题：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?

思想：活跃变量分析（Liveliness Analysis）

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，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）。

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。

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。

合并和溢出变量机制：

问：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。

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。

活跃变量分析

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) 寄存器的使用

- $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。
- $gp固定指向64K静态数据区的中央，$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。

总结：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。

5.5.5 栈管理的解决方案

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。

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。

控制流：

> 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。

数据流：

> 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。

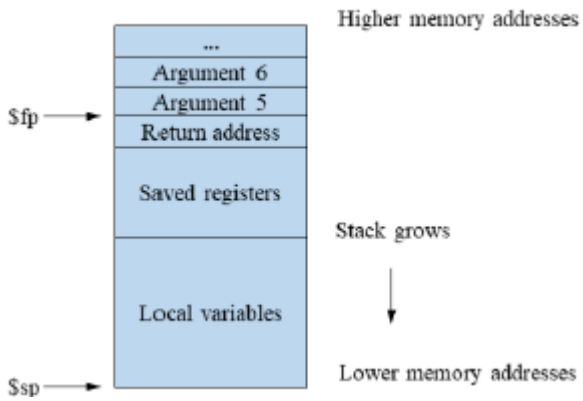
栈

不是要去实现栈，而是利用好栈！

作用：

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，本实验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。



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。

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live1、live2、…、livek写到栈中，之后将参数arg1、arg2、…、argn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。

```
sw live1, offsetlive1($sp)   # 将活跃变量入栈
…
sw livek, offsetlivek($sp)
subu $sp, $sp, max{0, 4 * (n – 4)}  # 为多余参数腾出空间
move $a0, arg1               # 存入参数（到寄存器）
…
move $a3, arg4
sw arg5, 0($sp)              # 存入参数（到栈）
…
sw argn, (4 * (n – 5))($sp)
jal g                                                    # 跳到被调用者
addi $sp, $sp, max{0, 4 * (n – 4)}     # 弹出参数
lw live1, offsetlive1($sp)             # 恢复活跃变量
…
lw livek, offsetlivek($sp)
```

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reg1、reg2、…、regk存入栈，最后将调用者由栈中传入的实参作为形参p5、p6、…、pn取出。整个过程如下所示：

```
subu $sp, $sp, framesizeg                          # 在栈中分配空间
sw $ra, (framesizeg – 4)($sp)                  # 压入ra
sw $fp, (framesizeg – 8)($sp)                  # 压入fp
addi $fp, $sp, framesizeg                          # fp指针指向栈顶
sw reg1, offsetreg1($sp)                           # 被调用者寄存器入栈
…
sw regk, offsetregk($sp)
lw p5, (framesizeg)($sp)                           # 参数取出
…
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：

```
lw reg1, offsetreg1($sp)                    # 从栈恢复调用者的寄存器
…
lw regk, offsetregk($sp)
lw $ra, (framesizeg – 4)($sp)               # 恢复ra
lw $fp, (framesizeg –8)($sp)                # 恢复fp
addi $sp, $sp, framesizeg                   # 恢复栈顶
jr $ra                                      # 跳回去
```

5.6 提示

1. 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。
2. 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。
3. 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一一测试程序。
4. 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。
5. 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。