第 5 章 目标代码生成

【引言故事】

在我们的日常交流中，如果需要将一种方言转化为另外一种方言，通常可以将源方言先通过普通话表达给翻译员，然后再由翻译员将普通话转化为目标方言。在这种场景下，普通话就扮演了“中间方言”的角色。这个过程其实与我们编译器的工作流程类似，当我们获得作为中间表示的普通话时，最后一步就是将它翻译成为目标方言（即目标代码）。方言是我们日常交流中最有生机活力的一部分，但是将普通话转为方言却并不是一件容易的事情。

以前的学者和诗人是尝试过使用方言写诗，每个诗人在写作的时候会发出一些自己最习惯的声音，那么使用的声音是哪里的方言，是四川话、闽南话、广东话、还是吴语（苏州话）？方言对写作是非常重要的，如普通话说“谁”，四川话说的却是“哪个”，广东话却说“宾个”。一个人写诗也好写小说也好，如果在叙述一个人物时，不是通过广东话表达，就不会写“宾个”。一般情况下，在写作时一定会不自觉地把表达的意思翻译成普通话，那么实际上笔下的人物就丧失了一种可触摸的、在场的感觉，而作为写作主体的某些特质也可能丧失。由此可见，即使是在普通话与广东话这样同源的语言之间的翻译，也会导致一些语义信息上的变化。

另外，用方言表达的一些意思仅仅能够被特定人群理解。徐志摩诗歌中就曾大量运用过家乡话（海宁硤石方言）来进行创作。他的这类诗大致可以看懂，例如，《一条金色的光痕》开篇写道：“得罪那，问声点看”，其中，“得罪那”还听得懂，“问声点看”，就只能勉强知道是问一问的意思。贵州诗人蹇先艾也曾用贵州遵义方言写诗。如《回去》中写到，“哥哥：走，收拾铺盖赶紧回去。”“乱糟遭的年生做人太难。”其中，“年生”一词只有云贵川的人才懂，上海人也好，广东人也好，是很难理解“年生”的意思。《回去》中的第三句“想计设方跑起来搞些啥子”中的“搞些啥子”，普通话就会自动地翻译成“搞些什么”，而接下来的一句“这一扒拉整得来多惨道”，“这一扒拉”必然也会使其他方言区的人困惑[1]。

[1]柏桦,邓月娘.柏桦访谈[J].扬子江评论,2016

与诗歌创作过程中使用的方言必须要考虑目标读者的特点一样，编译器将源代码编译为目标代码的过程，也必须考虑目标机器的架构及指令集等特点。编译器将中间表示翻译为目标代码时，也可以理解为某种方言到普通话再到另外一种方言的转换。类似于普通话与方言无关一样，中间表示与程序运行的目标机器无关；而目标代码使用特定机器的指令，在A机器上执行的目标代码，在指令集不同的B机器上就无法执行。编译器将中间表示转为机器所能识别的语言，就是将普通话转为人们日常交流的语言。

【本章要点】

目标代码生成是编译器工作流程中的最后一个关键步骤。在该步骤中，编译器首先为中间表示选择对应目标机器平台上的指令集架构并进行初步翻译。在此基础上，完成寄存器分配并尝试进行目标代码优化等工作。本章内容主要涵盖了在编译过程中进行目标代码生成的过程以及其中的关键算法。本章首先重点讨论了基本块和流图的内容和构造算法，并给出了构造的流图示例；在指令选择过程和实现方法部分，本章介绍了线形 IR 和树形 IR 的指令选择算法；在寄存器分配部分，本章介绍了朴素寄存器分配、局部寄存器分配、活跃变量分析和图染色算法；在目标代码优化部分，本章给出了窥孔优化技术，并提供了实施的示例；最后，本章介绍了代码生成器的构建过程及目标代码示例。

此外，本章中也讨论了具体的实践技术内容，在所提供的技术指导下，将中间表示编写的中间代码翻译为 MIPS32 指令序列，并在 SPIM Simulator 上运行。选择 MIPS 作为目标体系结构的原因是，它属于 RISC 范畴，与 x86 等体系结构相比形式简单，便于我们处理。如果对于 MIPS 体系结构或汇编语言不熟悉也不用担心，我们会提供详细的参考资料。在相应的工具的帮助下，结合本章前面介绍的理论方法，实现一个目标代码生成器将不再是一件困难的事。

需要注意的是，由于本次实践内容的代码会与之前实践内容中已经写好的代码进行对接，因此保持良好的代码风格、系统地设计代码结构和各模块之间的接口对于整个实践内容来说是相当重要的。

思维导图：


第五章
目标代码生成
主要任务
输入：中间代码
输出：MIPS汇编代码
实现：代码生成器
主要流程
指令选择
寄存器分配
目标代码优化
指令选择
为中间代码选择对应机器平台上的指令集架构并初步翻译
指令集架构
由操作码和特定处理器执行的基本命令组成
复杂指令集计算机（CISC）
精简指令集计算机（RISC）
基本块与流图
基本块
描述中间代码指令控制流关系，用于目标地址跳转及后续优化
基本块生成算法
流图及其表示方式
数据结构
流图生成算法
循环和分枝处理
循环识别
分支优化
指令选择算法
线形IR
组成 — 指令和地址
选择算法 — 宏扩展
树形IR
表示方式 — 以树表示代码结构，语句或表达式可由子树构成
选择算法
Maximal Munch算法
动态规划算法
快速匹配算法
寄存器分配
为指令执行选择可用的寄存器来保存变量的值和指令的操作数
朴素寄存器分配算法 — 变量值存放在内存里，使用时加载到寄存器中
局部寄存器分配算法
使用启发式规则为基本块中出现的变量分配寄存器
结束时将所有修改过的变量写回内存
活跃变量分析
有效地从中间代码控制流中获取变量活跃信息
数据流方程构造
图染色算法
基于活跃变量分析
将寄存器分配问题转化为一个图染色问题
近似求解算法
目标代码优化
窥孔优化
冗余指令消除 — 消除目标代码中冗余的load/store指令
死代码消除 — 消除目标代码中的不可达代码
控制流优化 — 消除冗余的跳转指令

5.1 目标代码生成的理论方法

5.1.1 代码生成概述

我们通常使用高级程序设计语言（比如 C 、Python 等）编写程序。高级程序设计语言编写的程序由符合语法与语义规范的语句组成，并且更加贴近于自然语言，便于我们对程序进行理解与维护。然而，当前体系结构的计算机只能识别由0和1二进制码组成的代码序列，即机器语言编写的程序。在计算机发展的早期，程序员直接使用机器语言编写程序。这些程序可以直接被计算机识别，具有较快的运行速度。然而，不同架构的计算机支持的机器语言不同，不同机器语言定义了差异较大的指令与支持的数据格式。因此，使用机器语言编写的程序难以复用与维护，可移植性差。

为了更详细地阐述不同层次的程序设计语言的差异，我们给出一个简单的用于实现int类型变量自增的 C 语言程序，如下所示：

```
1  int inc(int a)
2  {
3    int b = a + 1;
4    return b;
5  }
6
7  int main()
8  {
9    int lcVar = 2 * 2;
10   int rtVar = inc(lcVar);
11   return 0;
12 }
```

我们可以将上述 C 语言程序保存到名为 Increment.c 的源代码文件中，然后在 Ubuntu 20.04 x86_64 平台上使用 GCC（7.5.0 版本）编译它。编译命令如下：

```
gcc Increment.c -o Increment
```

其中， GCC 编译 Increment.c 源代码文件生成了**可执行文件**Increment。该可执行文件存储着 C 语言程序所在的 Ubuntu 20.04 x86_64 平台上对应的机器指令以及相关数据，可以在计算机上直接运行。由于机器指令难以理解，为了便于理解，我们将其转换成汇编代码。表 5.1 展示了 inc 函数的机器指令、汇编代码以及 C 语言程序代码的三种不同表示形式：

表 5.1 inc 函数机器指令、汇编代码与 C 语言程序代码三种表示形式

机器指令（十六进制表示）	汇编代码	C 语言程序代码

0x55
0x48 0x89 0xe5
0x89 0x7d 0xec
	push %rbp
	mov %rsp, %rbp
	mov %edi, -0x14(%rbp)
		int inc(int a)
		{

0x8b 0x45 0xec
0x83 0xc0 0x01
0x89 0x45 0xfc
	mov -0x14(%rbp), %eax
	add $0x1, %eax
	mov %eax, -0x4(%rbp)
		int b = a + 1;

0x8b 0x45 0xfc	mov -0x4(%rbp), %eax	return b; }
0x5d 0xc3	pop %rbp retq	

表 5.1 的第一列是编译目标机器指令的十六进制表示形式，第二列是机器指令所对应的汇编代码，第三列是 inc 函数的 C 语言程序代码。每一行中的内容都具有相同的语义。

例如，在机器指令中，使用了 3 个字节来表示 inc 函数返回的指令：

```
0x8b 0x45 0xfc
```

这三条指令对应的汇编代码为：

```
mov -0x4(%rbp), %eax
```

而所对应的 C 语言程序语句位于源代码的第 4 行：

```
return b;
```

从表 5.1 中我们可以发现，除了函数声明外，C 语言程序中的 inc 函数体共有两条语句而 inc 函数的机器指令却包含了 21 个字节以及 9 条汇编代码指令。与 inc 函数类似，表 5.2 展示了 main 函数的 C 语言程序代码及经过编译后生成的机器指令与汇编代码的三种不同表示形式。

在表 5.2 中，我们可以看到 inc 函数调用的汇编代码为：

```
callq 114a <inc>
```

其中， callq 指令用于调用其他函数，该指令的操作数为被调用的目标函数的地址。在该指令中，目标函数地址为 114a <inc>，即 inc 函数的地址。114a 表示的是 inc 函数在可执行文件 Increment 中的偏移地址。在可执行文件中的指令装载到内存后，该地址会被转换为 inc 函数在内存中的实际地址，以实现函数调用功能。执行该指令后，程序会跳转到 inc 函数的地址，执行函数中的指令。注意，该指令会将下一条指令的地址保存到栈中，以便函数返回时能够返回到正确的地址。函数调用结束后，程序会继续执行 callq 指令下一条指令的地址，即：

```
mov  %eax, -0x4(%rbp)
```

表 5.2 main 函数机器指令、汇编代码与 C 语言程序代码三种表示形式

机器指令（十六进制表示）	汇编代码	C 语言程序代码

0x55
0x48 0x89 0xe5
0x48 0x83 0xec 0x10
	push %rbp
	mov %rsp, %rbp
	sub 0x10, %rsp
		int main()
		{

0xc7 0x45 0xf8 0x04 0x00 0x00 0x00	movl $0x4, -0x8(%rbp)	int lcVar = 2 * 2;

0x8b 0x45 0xf8
0x89 0xc7
0xe8 0xd2 0xff 0xff 0xff
0x89 0x45 0xfc
	mov -0x8(%rbp), %eax
	mov %eax, %edi
	callq 114a <inc>
	mov %eax, -0x4(%rbp)
		int rtVar = inc(lcVar);

0xb8 0x00 0x00 0x00 0x00	mov $0x0, %eax	return 0; }
0xc9 0xc3 0x66 0x2e 0x0f 0x1f 0x84 0x00 0x00 0x00 0x00 0x00 0x0f 0x1f 0x40 0x00	leaveq retq nopw %cs:0x0(%rax, %rax, 1) nopl 0x0(%rax)	

与 C 语言相比，使用汇编语言与机器指令编写程序均较为困难，这会大大降低软件开发与维护的效率。而现代高级程序设计语言相对更容易学习和使用，并且易于阅读理解，能够极大地提高软件开发与维护的效率。因此，现代几乎所有软件开发活动主要使用高级程序设计语言。然而，从计算机的角度来看，计算机只能运行由二进制代码组成的机器指令，无法直接理解高级程序设计语言编写的源代码。因此，我们需要使用编译器将高级程序设计语言编写的源代码自动转换为机器可以识别的机器指令。对我们的 C— 编译器来说，编译器实现的最后一步就是将中间代码翻译为目标机器平台上的指令，以实现在目标平台上运行 C— 程序的功能。

从高级语言到低级语言和机器指令的转换过程中的各个步骤都由相应的编译器组件来完成。高级程序设计语言编写的程序需要经过编译器编译转化为语义相同但较为低级的中间代码，而后编译器再将中间代码转化为目标机器指令序列，这个过程通常被称为**目标代码生成**。目标代码的生成过程通常指的是编译器在进行源程序的词法分析、语法分析和语义分析基础上，读取编译器前端生成的中间代码和相关的符号表信息，生成语义等价的汇编代码或机器指令，即目标代码。符号表信息可以用于确定符号对应的数据对象在运行时的地址。目标代码生成过程通常实现为编译器后端的代码生成器。在前面的实践内容中，我们已经学会了词法分析、语法分析、语义分析以及中间代码生成。在本章实践内容中，我们将介绍编译器实现的最后一步— 目标代码生成的理

论和实践技术，并实现自己的代码生成器。通常，我们对代码生成器的要求非常严格。一方面，代码生成器必须保证生成的目标代码与源代码的语义一致；另一方面，代码生成器需要有效地利用目标机器上的可用资源，以确保生成的目标代码具有高效的运行性能。然而，理论上已经证明，为源代码生成最优目标代码是不可判定问题，代码生成过程中的子过程（如指令选择和寄存器分配）处理的计算复杂度很高。因此，在代码生成器的实现中，我们通常使用成熟的启发式技术生成良好但不一定是最优的目标代码，以提高目标代码的生成效率。

目标代码生成主要包括两个步骤：

（ 1 ）**指令选择：**指令选择主要关注选择适用于特定平台的目标机器指令。不同平台的机器指令集不同，因此需要在生成目标代码时考虑这一点。

（ 2 ）**寄存器分配和指派：**寄存器分配和指派主要涉及将数据适当地分配到寄存器中，以便在执行目标代码时获得最佳性能。寄存器是 CPU 内用于暂存指令、数据和符号地址的存储器；与内存空间相比，寄存器数量较少，但访问速度较快。选择适当的寄存器存放操作数，能够有效地避免多次从内存加载数据到寄存器开销，以提升目标代码的运行效率。

代码生成器的设计和实现与IR的形式密切相关，例如线形 IR 和树形 IR 的处理方式差异较大。尽管 IR形式有很多种，包括三地址表示方式（如三元式、间接三元式、四元式等）、虚拟机表示形式（如字节码和堆栈机代码）、线性表示方式（如后缀表示）以及图形表示形式（如语法树和有向无环图）。针对不同的中间代码形式，代码生成器可能采用不同的翻译模式和算法。在实践中，我们主要考虑线形表示形式中的线形 IR （三地址码）和图形表示形式中的树形 IR 的翻译。我们假设代码生成器读入的IR已经经过严格的词法、语法和语义分析，不包含错误，按照步骤翻译后的目标代码是正确的。

目标代码

代码生成器的设计目标是将编译器前端生成的中间代码翻译为目标代码。目标代码通常被设计成与目标机器指令较为相似或相同，例如汇编代码或机器指令。这些指令以某种特定的格式存储在**目标文件（Object File）**中。例如，C语言程序代码通常会被GCC编译器编译为 .o 的目标文件（在 Windows 平台下通常被编译为 .obj 目标文件）。目标文件包含目标代码指令以及代码在

运行过程中使用的数据信息。目标代码指令的语义和格式与特定的目标机器密切相关，每种目标机器都有其支持的指令集。目标代码执行时使用的数据信息包含变量的初始值以及所占内存空间的大小等。除了考虑目标机器指令集，因为目标代码中的符号代表变量或函数等在内存中的位置，目标代码生成还需要妥善安排目标代码中各个符号的地址。早期的编译技术中，目标代码直接使用绝对地址将代码装载到内存中固定的位置上。虽然这样可以有效提升程序编译和执行速度，但随着程序规模的不断扩大，使用绝对地址的代码需要为每个符号设定唯一的地址，这使得代码难以维护和扩展。为解决绝对地址带来的问题，我们可以生成**可重定位的目标代码**。在这种目标代码中，符号在内存中的装载地址不固定，操作系统会在内存中寻找一块合适大小的空间来装载目标文件符号。这使得各个子程序可以被分别编译，独立链接、加载和运行，从而使得编译过程具有更高的灵活性。但这样做的代价是链接和加载过程较慢，需要重新计算目标代码中符号的内存位置。为了避免目标机器指令集的高复杂性，降低地址计算的难度，我们通常将中间代码转换为汇编指令，再借助汇编器生成目标机器指令。代码生成器的关键目标是生成正确的目标代码指令序列，以确保生成的目标代码的正确性且易于实现、测试和维护。为了实现这一目标，我们必须仔细设计指令选择、寄存器分配、目标代码优化等过程。

5.1.2 指令集架构

在编译器代码生成器的实现过程中，指令选择是目标代码生成的第一步，先于寄存器分配阶段。指令选择是编译器后端将中间代码翻译为特定目标平台上的更低级表示的目标代码的阶段，这一阶段的产出是指令选择器。在指令选择过程中，我们可以假设输出的目标代码中可以使用无限数量的寄存器，只需要关注目标代码生成即可。指令选择可以看作是一个模式匹配过程。对于给定的中间代码，我们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。指令选择可以是一个简单的匹配模式然后一一对应的过程，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丰富程度，以及目标机器采用的指令集的种类。指令选择生成的代码通常不是最优的，我们可以在后续过程中使用窥孔优化技术来提高目标代码的质量和执行效率。接下来，我们将首先介绍指令选择中的指令集架构。

指令集架构（Instruction Set Architecture），又称为**指令集体系**。指令集架构通常由操作码

以及由特定处理器执行的基本命令组成。具体来说，指令集架构包含了基本数据类型（如一个字节、两个字节、四个字节等不同长度的数据）、指令集、寄存器、寻址模式等。指令集架构可以细分为**复杂指令集计算机（Complex Instruction Set Computer ，CISC）**与**精简指令集计算机（Reduced Instruction Set Computer ，RISC）**两大类。CISC 的诞生与当时计算机硬件资源有限有关。20 世纪早期，计算机内存和磁盘存储成本非常高，高级语言代码编译后的目标代码占用空间较大，存储成本也很高。因此，很多计算机架构师尝试设计直接支持高级程序结构的指令集，例如过程调用、循环控制和复杂的寻址模式。这种指令集允许将数据结构和数组访问融合到单个指令中，从而提高代码密度和紧凑性。降低存储成本的代价则是主内存访问速度的降低。CISC 的每条指令可以执行多个操作，将数据读取、存储和计算操作集中在单个指令中。CISC 架构被广泛应用，包括复杂的大型计算机和微控制器。许多著名的微处理器和微控制器都采用了 CISC 架构，例如 System/360 、PDP-11 和摩托罗拉 68000 系列以及 Intel 8080 系列。 CISC 包含许多应用程序极少使用的特定指令，指令种类繁多，指令长度不固定。计算机执行 CISC 指令时需要花费时间解析指令，可能带来额外的性能负担。CISC 的设计初衷是直接支持高级编程结构，将数据结构和访问融合为单个指令，指令集更加紧凑，访问内存的次数更少，这在早期内存和磁盘资源受限的情况下可以大幅降低存储成本。尽管 CISC 使用更少的指令就能表达高级语言结构，但是某些情况下，使用一系列更简单的指令可以在复杂体系结构的低端版本中获得更好的性能。

随着现代计算机硬件的发展，计算机的存储资源和计算能力大大提升，指令格式相对简单的 RISC 受到了广泛的关注。RISC 架构历史悠久，最早可以追溯到 1964 年 Seymour Cray 设计的 CDC 6600 架构。 RISC 最早发展于 20 世纪 70 年代的 IBM 801 项目，该项目也被认为是第一个采用 RISC 架构的系统。然而在当时，RISC 并未立即在业界应用。直到 20 世纪 70 年代后期，801 项目在业界受到了很大的关注，RISC 也开始受到关注。RISC 通常只执行较为常用的指令，大部分指令长度统一，分离了数据加载和存储操作，简化了处理器结构。RISC 每条指令只执行一个功能，结合执行流水线执行方式在单个处理器上实现指令集并行技术，从而提升指令执行速度。与 CISC 计算机相比，RISC 计算机完成相同的任务需要更多的指令。然而，RISC 指令的执行速度平衡了代码中更多指令数目带来的性能开销。RISC 架构中相对于 CISC 架构的“简

单”本质上是每条指令所完成的工作量减少了。因此，RISC 较为“简单”。但是， RISC 的“简单”并不是等同于简单地删除了指令。实际上，自诞生以来， RISC 指令集规模一直扩展。RISC 家族中一些指令集与 CISC 指令集相等甚至更大，例如采用 RISC 处理器 PowerPC 指令集与 CISC IBM System/370 一样大。

指令集架构中不仅定义了指令的格式和功能，还定义了各种功能不同的寄存器。RISC 目标机器包含了数量众多的寄存器、三地址指令、多种寻址方式以及一个相对简单的指令集体系结构等特点。相反，CISC 目标机器只有数量较少但种类较多的寄存器、两地址指令、更多的寻址方式。此外， CISC 目标机器还支持可变长度指令和一些有副作用的指令。

除了 CISC 和 RISC，许多高级语言编写的源代码也可以在基于栈的虚拟机上直接执行。随着 Java 、Python 等语言的广泛应用，基于栈的体系结构受到了更多关注。基于栈的虚拟机提升了程序的可移植性。基于栈的体系结构通常是解释执行源代码，而不是将源代码编译为目标代码后再运行。其运算过程伴随着栈上运算分量的操作，相对来说会更加耗时。在本次实践内容中，我们将重点关注经典编译器的设计与实现，而不涉及虚拟机的设计与实现。我们将介绍精简指令集架构中的 MIPS，该指令集架构是 RISC 指令集架构家族成员之一，第一个版本于 1985 年随着 R2000 微处理器一起发布。MIPS 架构应用广泛，在业界受到广泛关注。接下来，我们将重点介绍 MIPS 架构中定义的 9 类指令。

（ 1 ）**逻辑操作指令**：此类指令共包含 8 条指令，and 、andi 、or 、ori 、xor 、xori 、nor 、lui，用于实现逻辑与、或、异或、或非等运算。

（ 2 ）**移位操作指令**：此类指令共包含 6 条指令，sll 、sllv 、sra 、srav 、srl 、srlv，用于实现逻辑移位、算术移位、循环移位等运算。

（ 3 ）**移动操作指令**：此类指令共包含 6 条指令，movn 、movz 、mfhi 、mthi 、mflo 、mtlo，用于通用寄存器之间的数据移动和通用寄存器与 HI 、LO 寄存器（整数乘法寄存器）之间的数据移动。

（ 4 ）**算术操作指令**：此类指令共包含 21 条指令，add 、addi 、addiu 、addu 、sub 、subu 、clo 、clz 、slt 、slti 、sltiu 、sltu 、mul 、mult 、multu 、madd 、maddu 、msub 、

msubu 、div 、divu，用于实现加法、减法、比较、乘法、乘累加、除法等算术运算。

（ 5 ）**转移指令**：此类指令共包含 14 条指令，jr 、jalr 、j 、jal 、b 、bal 、beq 、bgez 、bgezal 、bgtz 、blez 、bltz 、bltzal 、bne，用于实现无条件转移以及条件转移。

（ 6 ）**加载存储指令**：此类指令共包含 14 条指令，lb 、lbu 、lh 、lhu 、ll 、lw 、lwl 、lwr 、sb 、sc 、sh 、sw 、swl 、swr，其中以“ l ”开始的指令是加载指令、以“ s ”开始的指令是存储指令。这些指令用于从存储器中读取数据或者向存储器中写入数据。

（ 7 ）**协处理器访问指令**：此类指令共包含 2 条指令，mtc0 、mfc0，用于读取协处理器 CP0 中某个寄存器的值，或者将数据保存到协处理器 CP0 中的某个寄存器。

（ 8 ）**异常相关指令**：此类指令共包含 14 条指令，其中 12 条自陷指令，teq 、tge 、tgeu 、tlt 、tltu 、tne 、teqi 、tgei 、tgeiu 、tlti 、tltiu 、tnei，另外两条指令是系统调用指令 syscall 以及异常返回指令 eret。

（ 9 ）**特殊指令**：此类指令共包含 4 条指令，nop 、ssnop 、sync 、pref，其中 nop 是空指令，ssnop 是一种特殊类型的空指令，sync 用于保证加载、存储操作的顺序，pref 指令用于缓存预取。

除了上述 9 类 MIPS 机器指令，我们将在实践技术中详细介绍 MIPS32 汇编代码及相关内容。

5.1.3 基本块与流图

代码生成器将中间代码翻译成语义上等价的目标代码。目标代码包含逻辑操作指令、算术操作指令等，这些指令通常比较直观，翻译策略较为简单。然而，对于条件转移与函数调用指令，我们需要明确获知控制发生转移时的目的地址或者下一条指令的地址。例如，条件分支指令可能在条件成立时直接跳转到目的地址，为了确定目标代码中转移指令的目的地址，我们首先需要分析中间代码的**控制流（Control Flow）**，从而确定指令生成和执行的顺序。生成的目标代码中的控制流不考虑数据与算术运算，只关心指令执行的顺序。我们需要在控制流中考虑指令转移到的所有目的地址，构建完整的流图。因此，我们需要将中间代码中所有对条件转移无关的指令集中在一个**基本块（Basic Block，BB）**中，以形成**流图（Flow Graph）**中的节点。流图中的边指明

了基本块之间执行的顺序。构建好流图后，我们可以遍历流图，生成每个基本块对应的目标代码。

1. 基本块

在目前的编译器构建中，基本块是指满足以下条件的最大的连续中间代码指令序列：

（ 1 ）基本块只有一个入口点，控制流只能通过基本块中第一条指令进入基本块。

（ 2 ）除了基本块入口点，转移指令无法跳转到基本块中间的指令。

（ 3 ）基本块只有一个退出点，即基本块中的最后一条指令执行结束才会停机或者跳转到其他基本块。

基本块中包含一系列按顺序依次执行的指令，并且前部指令**支配（Dominate）**其后面的指令，不存在非相邻指令间的执行关系。这种高度内聚的指令序列使得基本块分析变得更加容易。编译器通常在中间代码分析阶段将中间代码分解为若干基本块，并构建流图。为了生成基本块，我们可以先遍历中间代码，找到基本块的首指令、最后一条指令或者转移指令，例如，转移指令及其目的地址就是基本块的边界位置。具体而言，基于中间代码生成基本块的算法如下：

（ 1 ）扫描中间代码指令序列，找到基本块的首指令。首指令表示一个基本块开始的第一条指令。两条首指令之间就定义了一个基本块。首指令通常包括指令序列的第一条指令，无条件或条件转移指令的目的指令，以及跟在一条条件或无条件转移指令之后的指令。

（ 2 ）从首指令开始查找当前基本块中的指令，直到找到下一条基本块首指令，两条首指令之间的中间代码都属于当前基本块。

（ 3 ）如果基本块最后一条指令不是条件转移指令或函数返回语句，就在这个基本块的末尾增加一条转移到下一个基本块的转移指令。

一旦入口基本块被确定，代码执行结果不会受到基本块的位置顺序的影响。因为每个基本块的末尾都可以转移到唯一正确位置，所以生成的基本块可以按任意顺序生成或放置。我们可以通过选择适当的基本块位置排列来减少跳转的次数，以更好地优化代码，并保证代码执行结果相同。我们可以将条件不成立时跳转的目标基本块直接跟在条件转移指令之后，这样便可以在条件为假时直接执行下一个基本块的指令，以减少分支跳转次数。另外，我们也可以在无条件转移指令之后直接跟随目的地址所在的基本块，从而删除这些冗余的无条件转移指令，提高编译生成的目标

代码执行速度。

2. 流图及其表示方式

一旦将中间代码划分为基本块，就可以使用流图来表示它们之间的执行顺序流。**控制流图（Control Flow Graph，CFG）**以图的形式展示程序执行过程中所有可能的路径，这是编译器优化以及程序分析的基础。每个 CFG 顶点都对应一个不包含分支指令的基本块。在 CFG 中，有向边表示分支，基本块 B_i 到基本块 B_j 之间存在一条边，当且仅当 B_j 的第一条指令可能直接跟在 B_i 最后一条指令之后执行。基本块之间的边主要存在以下两种情况：

（ 1 ）基本块 B_i 中最后一条指令是无条件或者条件转移语句，跳转的目标是基本块 B_j。

（ 2 ）在中间代码指令序列中，基本块 B_j 在基本块 B_i 之后，且基本块 B_i 的最后一条指令不是无条件转移指令。

在这种情况下，我们可以称基本块 B_i 是基本块 B_j 的**前驱（Predecessor）**，基本块 B_j 是基本块 B_i 的**后继(Successor)**。通常，我们会在控制流图中添加两个特殊的基本块，即**入口基本块（Entry Block）**和**出口基本块（Exit Block）**。它们不对应中间代码中任何一条指令，且满足以下特性：

（ 1 ）入口基本块是控制流图中第一个可执行节点（即中间代码中第一条指令所在的基本块）的前驱。

（ 2 ）如果中间代码最后一条指令不是无条件转移指令，那么最后一条指令所在的基本块是出口基本块的一个前驱。另外，任何包含可以跳转到中间代码之外的指令或者可能是最后执行指令所在的基本块也是出口的前驱。

例如，对于以下中间代码片段：

```
1  READ v1
2  v2 := #0
3  LABEL label1 :
4  IF v1 > #1 GOTO label2
5  GOTO label5
6  LABEL label2 :
7  t1 := v1 / #2
8  t2 := t1 * #2
9  v2 := v1 - t2
10  IF v2 == #0 GOTO label3
11  GOTO label4
12  LABEL label3 :
```

```
13  v1 := v1 / #2
14  GOTO label1
15  LABEL label4 :
16  t3 := #3 * v1
17  v1 := t3 + #1
18  GOTO label1
19  LABEL label5 :
20  RETURN #0
```

我们可以采用基本块划分算法将上述中间代码片段划分为基本块，并且添加入口基本块和出口基本块以构建出控制流图，如图5.1 所示。

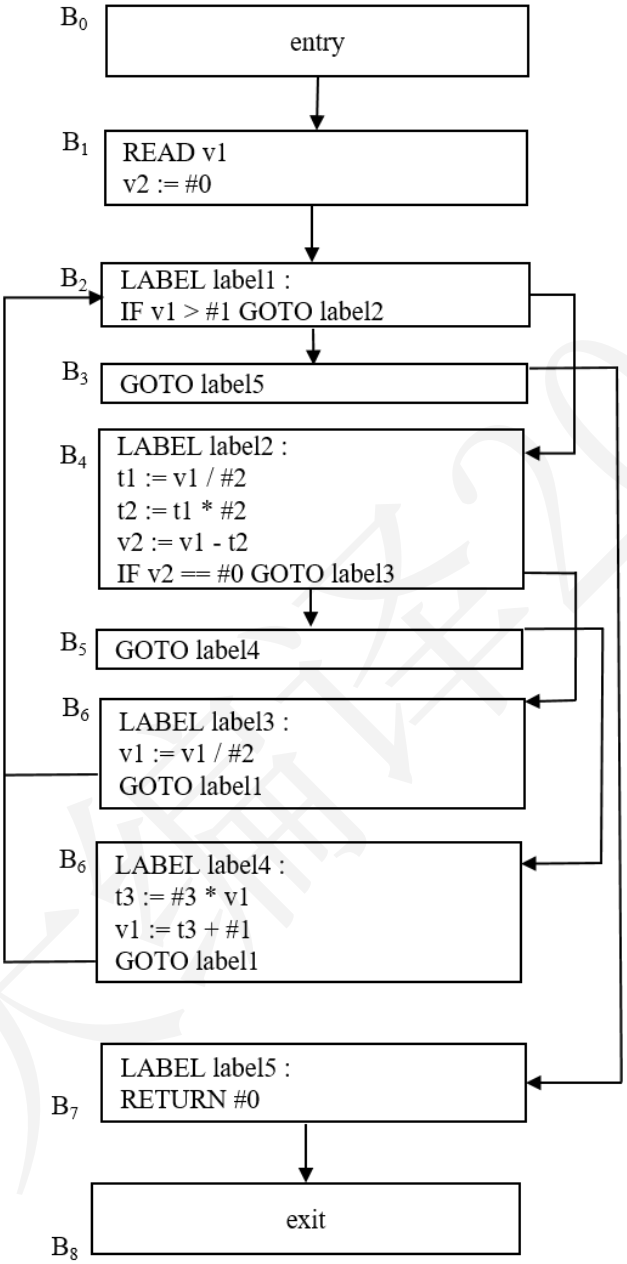

图 5.1 中间代码的流图表示形式示例

首先，根据基本块生成算法规则（1）可知第一条指令是一条首指令。为了找到其他的首指令，我们要找到跳转指令及其目的地址。在本例中有六条跳转指令（两条跳转跳转指令和四条无条件跳转指令），即指令 4 、5 、10 、11 、14 、18。根据算法规则（1），这些跳转指令的目

标是首地址，它们分别是指令 3 、6 、12 、15 、19。然后，根据算法规则（1），跟在一条跳转指令后面的每条指令都是首指令，即指令 5 、6 、11 、12 、15 、19。我们可以得出结论：指令 1 、3 、5 、6 、11 、12 、15 、19是首指令。每条首指令对应的基本块包括了从它开始直到下一条首指令之前的所有指令。因此，指令 1 的基本块是指令 1 和 2，指令 3 的基本块是指令 3 和 4，指令 5 的基本块是指令 5，指令 6 的基本块包含了从指令 6 到指令 10 的所有指令，指令 11 的基本块是指令 11，指令 12 的基本块是指令 12 、13 和 14，指令 15 的基本块包含了从指令 15 到 18 的所有指令。指令 19 的基本块是指令 19 和 20。

3. 循环与分支处理

在源代码中，循环结构通常会使用while语句、for语句等语句表示。相较于一般的线性中间代码指令，循环包含了转移和条件判断，因此执行开销更大。为了优化循环结构的代码执行效率，编译器通常会使用一些代码转换技术。这些技术通常需要先对流图中的“循环”进行识别。一个循环由流图中的一个节点集合构成，且满足以下特性：

（ 1 ）节点集合中存在一个**循环入口（Loop Entry）**，循环入口的前驱节点是唯一可能在循环外的节点。从整个流图的入口节点到循环中的任何节点都必须经过该循环入口。循环入口节点不是流图的入口节点，因为流图的入口节点是为了便于分析而添加的概念，并不存在于源代码中。

（ 2 ）循环中每一个节点都存在一条到循环入口的非空路径，并且该路径全部在该循环中。

图5.2 中{B_2 、B_3 、B_4 、B_5}这四个基本块就构成了一个循环。除了循环结构，分支结构也可以进行优化处理。例如，我们可以重新排序基本块，删除多余的无条件转移指令。对于基本块条件转移指令的条件不成立时跳转的目标基本块，我们可以将其调整到当前基本块之后。调整基本块顺序后，我们就可以删除条件不成立时的需要执行的无条件转移指令 5 和 11，以此完成优化。

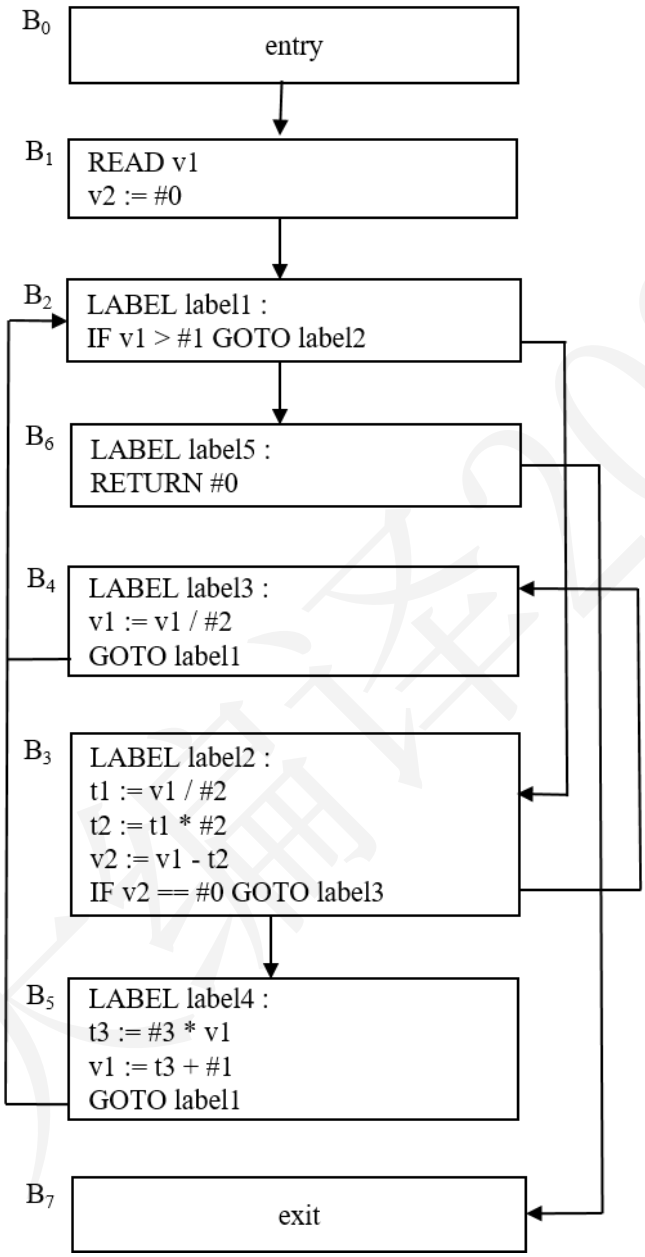

图 5.2 根据图 5.1 示例的基本块重排

5.1.4 指令选择算法

1. 线形 IR 指令选择算法

我们已经介绍了主流的指令集架构以及基本块和流图相关内容，构建好 CFG 后，我们便可

以确定目标代码中指令转移指令的目的地址。然后，我们需要按照翻译规则将每一个基本块中的中间代码翻译为目标代码，这一翻译过程就是指令选择阶段的任务。指令选择就是为给定的中间表示生成恰当的目标代码。这一过程与中间表示形式有关，比如线形 IR 的翻译过程通常与树形 IR 不同。因此，接下来我们将介绍线形 IR 和树形 IR 的指令选择算法。线形 IR 通常由指令和地址构成，例如三地址码、四地址码、SSA。如果我们的程序使用了线形 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。

宏扩展是线形 IR 指令选择算法中最简单的一种方法，指令选择器可以在 IR 上匹配模板来运行，这些模板可以实现为宏扩展器。每种线形 IR 都有自己的特定的语言结构，因此需要实现自己的宏定义。这些宏定义用于在目标机器上生成实现相应语义的汇编代码。在宏匹配成功时，宏扩展器使用匹配的字符串作为参数来执行相应的宏或者函数。宏扩展流程如下：

（ 1 ）对于给定的 IR 代码，指令选择器遍历 IR，宏扩展器根据输入的 IR 种类，选择对应的生成模板（模板可以实现为宏定义或者函数）。当找到一个匹配时，一个宏或者函数被执行，匹配的 IR 作为输入。然后，宏或者函数将输出与输入 IR 一致且正确的目标代码。

（ 2 ）如果指令选择器无法匹配某些 IR，指令选择器匹配失败并报告错误，目标代码生成失败。

通常，简单的宏扩展生成的目标代码执行效率较低。我们可以结合宏扩展与窥孔优化技术对生成的目标代码进行优化，使用更加高效的等效指令替换简单的指令组合。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）**滑过中间代码序列并查找可能的优化方案的过程。我们将在后续章节中详细介绍**窥孔优化（Peephole Optimization）**这种局部代码优化技术。此外，我们将在 5.2.3 节详细介绍线形 IR 指令选择算法的实现。

2. 树形 IR 指令选择算法

树形 IR 以树的形式表示代码的结构，例如抽象语法树是一种常见的树形 IR，源代码中一条语句或表达式可以表示成一棵子树，树中节点之间通过父子关系相连。这种表示形式可以更直观地反映出程序的结构，易于进行代码优化，例如子树代换、基本块识别、循环优化等。同时，

树形 IR 也可以很方便地进行语法分析，因为树形结构与语法规则有着自然的对应关系。树形 IR 指令选择算法的目标就是为给定的程序树找出一个恰当的目标代码指令序列。树形 IR 的指令选择可以看成一个**“树重写”**问题。其中，每一条目标代码表示成树形 IR 的一段**树枝（Fragment），**也称为**树型（Tree Pattern）**。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线形 IR 的种类，而是某种结构的子树。我们可以定义一组树重写规则把输入的树形 IR 规约为单个节点。这个应用树重写规则替换子树的过程称为对该子树的一次**覆盖（Tiling）**。某些情况下，可能会出现有多个树重写规则和某个子树匹配，我们需要设计算法找出可以覆盖树形 IR 的对应树型的最小集合。该集合对应于更少数目的目标代码，从而提升代码运行效率。

线形 IR 通常比较简单，也方便阅读，比如我们将 C— 表达式 `(x-y)/4` 翻译为如下的线形 IR：

```
1  t1 = x - y
2  t2 = t1 / 4
```

图5.3 所示的树形 IR 就是上述线形 IR 的等价表示，节点中存储了操作符以及变量，比如 `x-y` 就可以使用三个节点表示并将结果存储在“-”节点中，即线形 IR 中临时变量 t1 中。

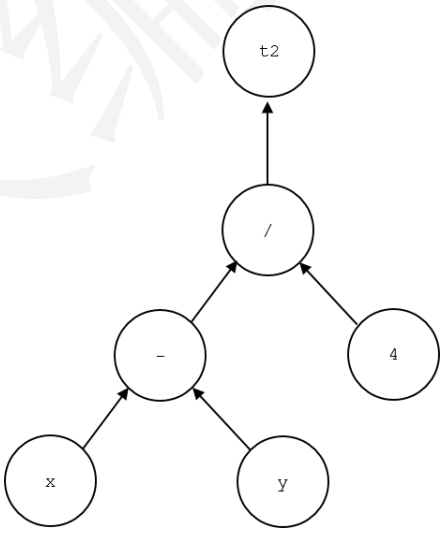

图 5.3 树形 IR 示例

同时，为了更好表征代码运行开销，我们假设每一种指令运行都有其对应的代价，比如运行

时间或者指令序列的长度。树形 IR 指令选择算法需要找出树的**最优（Optimum）覆盖**，即生成的目标代码运行代价最小。除了最优覆盖，树形 IR 指令选择算法还可以生成**最佳（Optimal）覆盖**，即不存在某个树型可以被拆分为更小组合代价的树型。目前学者们已经提出来了一些最佳覆盖和最优覆盖的树形 IR 执行选择算法，比如 Maximal Munch 算法、动态规划算法、快速匹配算法等。下面我们对这些算法进行概要性的介绍。

Maximal Munch 算法是一种最佳覆盖算法，算法的核心流程如下：

（1）从树形 IR 根节点开始，寻找覆盖树的最大的树型。如果能够找到对应的树型，那么原来的树形 IR 被分割为若干子树。

（2）重复这一覆盖过程直到所有的节点都被覆盖，算法终止。

（3）如果存在无法覆盖的子树，算法终止，报告错误。

Maximal Munch 算法指令生成过程与覆盖过程正好相反，覆盖过程从根节点开始，指令生成阶段需要从子节点开始逆序生成目标代码序列。根节点对应的指令需要等其子节点目标代码生成完毕。树形 IR 目标代码的逆序生成也符合中间代码的执行顺序。在图 5.3 中，虽然我们先生成了节点“-”的指令，但是只有当 x 、y 操作数加载到寄存器才能执行减法运算。树形 IR 目标代码的逆序生成是和中间代码执行顺序是一致的。

与最佳覆盖算法 Maximal Munch 不同，**动态规划（Dynamic Programming）**可以生成树形 IR 的最优覆盖。动态规划本身是一种根据每个子问题的最优解找到大问题的最优解的一种技术。因此，我们也可以将动态规划算法应用到树形 IR 覆盖。动态规划算法需要为树中每一个节点指定一个对应的代价，例如代价可以表示为以该节点为根的子树的最优目标代码序列代价之和。因此，动态规划算法的自底向上的，算法流程如下：

（1）对于给定的节点 *node*，动态规划算法递归求出 *node* 和子节点的代价 *cost*。然后，将每一种树型与 *node* 进行匹配。

（2）对每一个以代价 *cost* 与节点 *node* 匹配的树型 tp_j，tp_j 可能会有 0 个或者多个叶子节点，这些叶子节点不包含在树型中，可以用于连接子树。

（3）假设 tp_j 每个叶子节点 *leaf* 的代价为 $cost_i$，我们可以计算出 tp_j 的代价是 $cost+\Sigma$

$cost_i$。动态规划算法的目标就是找到代价最小的 tp_j。

当我们求出了根节点的代价之后，便开始生成与节点对应的目标代码，过程如下：

针对在树形 IR 中的节点 *node* 选择的树型，我们首先生成该树型的每一个叶节点 $leaf_i$ 所对应的目标代码，然后再生成与 *node* 结点匹配的目标代码，整个生成过程一直持续到根节点的目标代码生成完毕为止。此过程不是重复地作用于节点 *node* 的子节点，而是只作用于与节点 *node* 相匹配的树型的叶子结点。Maximal Munch 和动态规划算法都要检查结点 *node* 可能匹配的所有树型。匹配成功的条件是树型的每一个非叶子结点上标记的操作符与树型 IR 中对应结点的操作符相同，此过程相对耗时。为了降低匹配过程的开销，我们可以选择一种简单的快速匹配算法。**快速匹配算法**考虑结点 *node* 种类，例如我们将树形 IR 中 *BINOP* 类型的节点标记为 *BINOP* 标号，后续的匹配过程只考虑以 *BINOP* 标号为根的树型，缩小搜索空间。我们可以将所有的标号实现为 switch-case 结构，标号作为 case 语句的标号，就可以快速查找匹配的树型。我们将在本章后续的实践技术部分介绍树形 IR 指令选择算法的实现。

5.1.5 寄存器分配算法

RISC 架构的一个显著的特点是，除了 load/store 型指令之外，其余所有指令的操作数都必须来自寄存器而非内存。除了数组和结构体必须放到内存之外，中间代码里的非空变量或临时变量，只要参与运算，其值就必须被载入寄存器中。在某个特定的程序点上如何选择合适并可用的寄存器来保存变量或者运算指令的操作数，这就是寄存器分配算法所要解决的问题。

现代计算机中，寄存器的访问性能远高于内存，因此寄存器分配算法的优劣对目标代码的运行效率影响非常大。对于单个基本块中包含的只有一种数据类型且访存代价固定的中间代码，可以在多项式时间内计算出生成代价最小的寄存器分配方案。但是，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 NP 难[1]问题 。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。

本节将介绍三种不同的寄存器分配算法，包括朴素寄存器分配算法、局部寄存器分配算法以

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

及图染色算法。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依次递减。

1. 朴素寄存器分配算法

朴素寄存器分配算法是一种思想简单但效率最低的算法：将所有的变量或临时变量都放在内存里。当需要用到寄存器且可用寄存器数量不足时，将寄存器中的已有的数据先压入到栈中，再将释放出来的寄存器另做他用。这种方法会导致每翻译一条中间代码都需要将所需的数据先加载到寄存器中，计算出结果后又需要立刻将结果写回内存。虽然这种方法可以将中间代码翻译成可以正常运行的目标代码，并且实现和调试都比较容易，但其最大的问题在于寄存器的利用率实在太低。在后续的实践技术部分中，我们将介绍朴素寄存器分配算法的具体实现方式。

2. 局部寄存器分配算法

寄存器分配是一项困难的任务，因为在实际情况下，可用寄存器数量是有限的，例如，MIPS 指令集架构只提供了 32 个寄存器，其中有些寄存器被保留，无法使用。同时，在目标代码中，可能有许多变量，这些变量因无法独占空闲的寄存器而被迫和其他变量共用同一个寄存器，导致在使用这些变量时需要在寄存器中频繁换入和换出，从而增加了访存开销。为了尽量减少寄存器内容换入和换出的代价，我们可以通过合理地安排变量对寄存器的共用关系来达到这一目的。**局部寄存器分配算法**是一种较好的解决方案，它在每个基本块内部为出现的变量分配寄存器。在基本块结束时，该算法需要将该基本块中所有修改过的变量都写回内存。

该算法的核心思想很简单：逐条扫描基本块内部的中间代码，如果当前代码中有变量需要使用寄存器并且当前有空闲的寄存器，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，则选择那个在本基本块内不使用或者最晚使用的变量的寄存器进行**溢出**（**Spilling**）操作，从而减小溢出代价。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。我们将在后续的实践技术部分介绍局部寄存器分配算法的实现。

局部寄存器分配算法在实际应用中对于单个基本块内的中间代码非常有效。但是，当我们尝试将其推广到多个基本块时，会遇到一个非常难以克服的困难：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转移。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？在这种情况下，我们需要借助其他的技术获知变量在其后基本块中是否被使用，比如我们接下来将要介绍的活跃变量分析。

局部寄存器分配算法的弱点使得我们需要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。现在我们来讨论如何得到变量在中间代码中的活跃信息。首先我们需要定义活跃变量，我们认为变量 *x* 在某一特定的程序点是活跃变量当且仅当其满足如下条件：

（ 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* 没有在该赋值表达式右部出现，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。

如果第 *i* 条指令通过转移指令跳转到第 *j* 条指令，那么第 *j* 条指令是第 *i* 条指令的后继指令。如果第 *i* 条指令不是转移指令或 RETURN 语句且与第 *i*+1 条指令相邻，第 *i*+1 条指令也是第 *i* 条指令的后继指令。因此，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转移语句 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 条中间代码为条件转移语句 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 $succ[i]=\phi$。

（ 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 $out[i] = \bigcup_{j \in succ[i]} in[j]$。

我们可以遍历每一条中间代码指令，求解上述的数据流方程，获取中间代码中每一个程序点的活跃变量。我们将在 5.2.6 节介绍活跃变量分析算法实现所需的数据结构及数据流方程求解过程。

3. 图染色算法

基于活跃变量分析，我们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是避免为同时活跃的两个变量分配相同的寄存器。这是因为这些变量可能在后续运行过程仍然需要被使用，如果把它们分配到同一个寄存器，那么就需要进行频繁的寄存器内换入 / 换出操作，增加访存代价。但是在某些情况下，我们可以允许两个变量共用一个寄存器：

（ 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 $x \in out[i]$ 且 $y \in out[i]$。

（ 2 ）存在一条中间代码 *i*，这条代码不是赋值操作 *x:=y* 或 *y:=x*，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。在这种情况下，我们应该尽可能地为 *x* 和 *y* 两个变量分配不同的寄存器，以避免它们之间相互干扰。

如果我们将中间代码中出现的所有变量和临时变量都看作顶点，当两个变量相互干扰时，在它们所对应的顶点之间连一条边，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，那么我们希望在干涉图中相邻两个顶点不能染同一种颜色。这样，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 完全问题。然而，图染色问题存在一种能给出较好结果的线性时间近似算法，包含构造、简化、溢出和选择四个步骤。该算法的四个步骤具体如下：

构造：构造干涉图。对中间代码中每一个程序点，我们可以使用活跃变量分析方法计算在每个程序点处同时活跃的临时变量集合。该集合中每一对临时变量都相互干扰，形成一条边，我们将这些边加入干涉图中。

简化：为了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对干涉图进行着色。我们可以使用一个比较简单的启发式染色算法（称为 Kempe 算法）。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。删掉这类顶点可以简化原问题。重复执行上述操作，如果最后干涉图中只剩下少于 *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。

溢出：当删除顶点的过程中，如果干涉图中所有的顶点都至少包含了 *K* 个邻居，能否断定原图不能被 *K* 着色呢？如果我们能证明这一点，就相当于构造性地证明 P=NP。如果出现了干涉图中所有的顶点至少为 *K* 度的情况，我们仍会选择一个顶点进行删除，并将其压入栈中。同时，将这样的顶点标记为待溢出的顶点，并在之后继续执行删除顶点操作。

选择：对于标记为待溢出的顶点，在其最后被弹出栈时，可能其邻居节点总共只被染了少于 K 种颜色。在这种情况下，我们可以为该顶点成功染色并清除其溢出标记。否则，我们将无法为该顶点分配颜色，其所代表的变量就必须被溢出到内存中。

现在，所有在中间代码中出现的变量要么被分配了寄存器，要么被标记为溢出。然而，被标记为溢出的变量在参与运算时仍然需要被临时载入到某个寄存器中，并在运算结束后也需要某个寄存器临时保存其要溢出到内存中的值。那么，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。然而，这种方法浪费了寄存器资源。如果想要追求更高效的分配方案，可以通过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，以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。此外，对于干涉图中不相邻的顶点，还可以通过合并顶点的操作将它们显式地共用同一个寄存器，以减少寄存器的使用量。引入合并和溢出变量这两种机制会使全局寄存器分配算法更加复杂。如果想要了解具体细节，请自行查阅更多相关资料。我们将在本章实践技术部分介绍图染色算法的数据结构及其具体的算法实现。

5.1.6 窥孔优化

我们可以通过选择合适的指令集和寄存器分配算法来生成高效的目标代码。然而，随着指令集和寄存器分配算法的不断增加，我们很难选择最合适的目标代码生成策略。因此，实践中，一些编译器通常先生成最基本的目标代码，然后对其进行优化转换，最后输出高效的目标代码。然而，编译优化技术很难保证生成的目标代码是最优的，因为目前从理论上很难衡量目标代码的最优值。虽然编译器无法保证生成最优的目标代码，但是一些常用的优化技术可以降低目标代码的运行时间和空间开销。其中，窥孔优化是这些优化技术中最具代表性且相对简单高效的局部优化技术之一。窥孔指的是目标代码中的一个小的滑动窗口。窥孔优化会检查目标代码的一个滑动窗口，尝试用执行速度更快或者更简短的指令替换当前滑动窗口中的指令。尽管有些窥孔优化实现要求代码连续，但窥孔优化技术并不要求在窥孔中的目标代码一定是连续的。窥孔优化后，目标代码指令发生了改变，可能会产生新的优化机会。因此，我们可以对目标代码多次实施窥孔优化，

以最大程度地提高目标代码的运行效率。除了优化目标代码指令，窥孔优化也可以优化中间代码指令。本节将简要介绍窥孔优化中的冗余指令消除、死代码消除和控制流优化三种主要优化技术。

1. 冗余指令消除

在目标代码中，可能存在一些指令是冗余的，这些冗余指令包含了重复的信息，可以通过优化来消除，例如重复的 load 和 store 指令。冗余指令消除可以删除冗余的目标代码指令，以减小目标代码规模和提升运行效率。下面给出一个中间代码的例子：

```
1  sub1 := x - y
2  a := sub1
3  sub2 := a - z
4  b := sub2
```

假设目标代码中只有 load 和 store 指令可以直接操作内存，其他指令的操作数需要借助寄存器传递。根据上述的中间代码，生成的目标代码可能如下所示（注释以#开头）：

```
1  load reg0, x          # 把变量 x 的值加载到 0 号寄存器中
2  load reg1, y          # 把变量 y 的值加载到 1 号寄存器中
3  sub reg2, reg0, reg1# 把 x-y 的结果保存到 2 号寄存器
4  store reg2, a         # 把结果保存变量 a 中
5  load reg0, a          # 把 a 的结果加载到 0 号寄存器中
6  load reg1, z          # 把变量 z 的值加载到 1 号寄存器中
7  sub reg2, reg0, reg1# 把 a-z 结果保存到 2 号寄存器中
8  store reg2, b         # 把 2 号寄存器的值保存到 b 中
```

我们可以消除冗余的 load 和 store 指令。例如，对于上述目标代码，我们可以通过使用寄存器 reg2 来保存中间结果，从而消除第 4 条和第 5 条指令。消除冗余 load 和 store 指令后，目标代码将变为：

```
1  load reg0, x          # 把变量 x 的值加载到 0 号寄存器中
2  load reg1, y          # 把变量 y 的值加载到 1 号寄存器中
3  sub reg2, reg0, reg1# 把 x-y 的结果保存到 2 号寄存器
4  load reg0, z          # 把变量 z 的值加载到 0 号寄存器中
5  sub reg1, reg2, reg0# 把 a-z 结果保存到 1 号寄存器中
6  store reg1, b         # 把 1 号寄存器的值保存到 b 中
```

2.死代码消除

窥孔优化还可以用于消除目标代码中的不可达代码，即死代码消除。死代码消除可以用于删除无条件转移之后的不带标号的指令。重复执行这个删除过程，就可以保证将不可达的指令序列全部删除。例如，对于如下的中间代码[1]：

```
1  cond := #1
2  IF cond == #1 GOTO label1
```

[1] 该例子来源于：《编译原理》，Alfred V. Aho等著，赵建华、郑滔和戴新宇译，机械工业出版社，第354页，2009年。

```
3  GOTO label2
4  LABEL label1 :
5  x := 1
6  LABEL label2 :
7  y := 0
```

我们可以生成上述中间代码相应的目标代码：

```
 1    load reg0, 1           # 把常量 1 加载到 0 号寄存器中
 2    load reg1, 1           # 把常量 1 加载到 1 号寄存器中
 3    beq reg1, reg0, label1# 判断 cond==#1 是否成立
 4    j label2               # cond==#1 不成立，无条件转移到 lable2
 5  label1:
 6    load reg0, 1           # 把常量 1 加载到 0 号寄存器
 7    store reg0, x          # 把 0 号寄存器的值保存到变量 x 中
 8  label2:
 9    load reg0, 0           # 把常量 0 加载到 0 号寄存器
10    store reg0, y          # 把 0 号寄存器的值保存到变量 y 中
```

我们发现上述目标代码中第 3 条语句条件值恒为真，可以将其替换为无条件跳转语句。替换之后，第 4 行指令不可达，因此我们可以将其删除。实施死代码消除后的目标代码如下所示：

```
1    load reg0, 1           # 把常量 1 加载到 0 号寄存器中
2    load reg1, 1           # 把常量 1 加载到 1 号寄存器中
3    j label1               # cond==#1 恒成立
4  label1:
5    load reg0, 1           # 把常量 1 加载到 0 号寄存器
6    store reg0, x          # 把 0 号寄存器的值保存到变量 x 中
7  label2:
8    load reg0, 0           # 把常量 0 加载到 0 号寄存器
9    store reg0, y          # 把 0 号寄存器的值保存到变量 y 中
```

控制流优化

如果我们采用一些简单直观的目标代码生成策略，例如宏扩展，那么目标代码中经常存在一些转移指令，而这些指令转移的目的地址也是转移指令，比较常见的情况是无条件转移指令跳转到无条件转移指令、无条件转移指令跳转到有条件转移指令，以及有条件转移指令跳转到无条件转移指令。但是，并非所有指令间的多次转移都是必要的。因此，我们可以采用控制流优化技术来消除这些冗余的转移指令。例如，对于如下的中间代码：

```
1  IF cond == result GOTO label1
2  GOTO lable2
3  LABEL label1 :
4  GOTO label2
5  LABEL label2 :
6  x := 1
```

上述中间代码对应的目标代码是：

```
1    load reg0, cond        # 把变量 cond 的值加载到 0 号寄存器中
2    load reg1, result      # 把变量 result 的值加载到 1 号寄存器中
3    beq reg0, reg1, label1 # 判断 cond==result 是否成立
4    j label2               # 跳转到 label2
5  label1:
6    j label2               # 无条件转移到 label2
```

```
7  label2:
8    load reg0, 1            # 把常量 1 加载到 0 号寄存器
9    store reg0, x           # 把 0 号寄存器的值保存到变量 x 中
```

经过观察目标代码，我们可以发现第 4 行无条件转移指令的目的地址指向了另一条无条件转移指令，造成了冗余。为了减少不必要的跳转，我们可以直接将第 3 行条件转移指令的目的地址设置为第 7 行的 *label2*。这样我们就可以使用控制流优化技术，消除这些多余的转移指令。优化后的目标代码如下所示：

```
1    load reg0, cond         # 把变量 cond 的值加载到 0 号寄存器中
2    load reg1, result       # 把变量 result 的值加载到 1 号寄存器中
3    beq reg0, reg1, label2  # 判断 cond==result 是否成立
4  label2:
5    load reg0, 1            # 把常量 1 加载到 0 号寄存器
6    store reg0, x           # 把 0 号寄存器的值保存到变量 x 中
```

5.1.7 代码生成器构建

在目标代码生成过程中，我们已经介绍了指令选择、寄存器分配和窥孔优化等内容。接下来，我们将着重介绍中间代码到目标代码的转换细节。这些细节包括目标代码中的地址表示、寄存器使用等方面。在前面的章节中，我们已经了解了如何构建基本块和控制流图，并且结合控制流优化技术，删除了不必要的转移语句，提高了代码的性能。然而，在从中间代码到目标代码转换中，我们还需要考虑更多的细节，例如变量的底层表示、函数入口地址、指令集中寄存器的表示和其使用场景等。因此，在接下来的小节中，我们将详细介绍目标代码中的地址、寄存器和地址描述符等内容，以构建完整的代码生成器。

1. 目标代码中的地址

中间代码表示了程序的执行流程和数据操作，但它还没有直接对硬件的操作，因此需要将其转换为目标代码。在中间代码中，包含了函数、局部变量和临时变量等符号信息，而在目标代码中这些符号信息需要转化为具体的地址。由于 C-- 语言是 C 语言的一个子集，我们可以直接将一些 C 语言程序保存为 C-- 语言程序，并将其翻译为目标代码。以 5.1.1 节代码生成概述中 C 语言程序代码 Increment.c 为例，我们可以直接将其保存为 Increment.cmm，它对应的中间代码如下所示：

```
1  FUNCTION inc :
2  PARAM a
3  addtemp1 := a + #1
4  b := addtemp1
5  RETURN b
6  FUNCTION main :
```

```
 7  multemp1 := #2 * #2
 8  lcVar := multemp1
 9  ARG lcVar
10  calltemp1 := CALL inc
11  rtVar := calltemp1
12  RETURN #0
```

在中间代码中，我们使用临时变量存储运算的中间结果，而在将中间代码翻译为目标代码时，这些临时变量需要被存储到寄存器或内存中。因此，接下来我们将要介绍一些寄存器相关的内容以及目标代码生成方法。

2. 寄存器和地址描述符

在 RISC 指令集架构中，单个基本块内的指令执行过程通常需要依赖寄存器。这些寄存器可以存储运算指令的操作数、表达式的中间结果以及多个基本块使用的全局变量值等。在函数调用时，寄存器还可以用于传递参数、保存函数返回结果和维护运行时的堆栈。然而，由于寄存器数量有限，因此必须使用寄存器分配算法来妥善分配寄存器。为了实现寄存器分配算法，我们需要在代码生成器中使用相应的数据结构来表示寄存器和变量。

对于每个可用的寄存器，我们可以使用一个**寄存器描述符（Register Descriptor）**来表示当前寄存器中存放了哪些变量值。在一个基本块内进行寄存器分配时，我们可以简单地考虑寄存器描述符的初始状态都为空，所有可用的寄存器都未被分配。随着指令的执行，寄存器会被分配 0 个或多个变量的值。为了记录已经分配的变量名称，我们的寄存器分配算法需要在指令执行过程中提供一个或者多个可用的寄存器，并在寄存器描述符中更新已经分配的变量的名称。如果没有可用的寄存器，我们的算法还需借助堆栈中的空间来保存已使用的寄存器描述符的内容，以便为后续的指令提供足够数目的寄存器。

除了寄存器，目标代码中还需要使用程序中定义的变量，例如局部变量、临时变量、函数返回值、数组以及结构体等复合变量。因此，我们需要为每个程序变量定义一个**地址描述符（Address Descriptor）**。地址描述符记录了变量的位置，我们可以通过地址描述符查询变量存储在寄存器还是堆栈的内存空间中。地址描述符通常存放在目标代码的符号表中，作为变量符号名所在的条目值。当我们完成对寄存器还有变量的抽象表示后，就可以开始设计代码生成器了。

3. 代码生成器框架

根据我们提及的代码生成相关理论，我们可以设计一个代码生成器的框架，以实现目标代码

生成。针对中间代码中的一个基本块，代码生成器需要为每一条中间代码指令生成相应的目标代码指令，如分支、函数调用和表达式等。在生成目标代码指令时，需要使用寄存器来存储变量值和运算指令的操作数。由于寄存器数量有限，如果只为了生成可以运行的目标代码，那么我们可以频繁生成 load 和 store 指令，将寄存器描述符内容与内存交换。然而，这样生成的目标代码通常效率不高。

一旦选择目标代码指令集后（例如 MIPS 指令集），就可以遍历中间代码指令，选择相应中间代码指令种类的目标代码生成函数（例如函数名为“ codegen ”）。目标代码生成函数主要实现寄存器分配和指令生成。

（ 1 ）根据寄存器分配算法，实现一个函数 getReg(IR)，用于为中间代码 IR 中使用的变量选择寄存器；

（ 2 ）对于当前中间代码指令 IR，调用对应的目标代码生成函数，例如对于双目运算符，我们可以调用 binoop_codegen 函数生成目标代码。我们可以为每种中间代码类型生成对应的目标代码生成函数，并一次性翻译一行中间代码。除了单行翻译，我们还可以调用对应的目标代码生成函数来缓存已经生成的目标代码，等所有代码翻译完毕优化后再生成目标代码文件。

4. 目标代码示例

为了帮助大家更好的理解目标代码生成过程，我们可以尝试生成与前面的 C— 程序 Increment.cmm 对应的中间代码等价的目标代码。假设函数传递参数保存在寄存器 *rega0*，返回值保存在寄存器 *regv0*，函数调用指令为 jal，函数返回指令为 jr，函数返回地址保存在寄存器 regra 中，栈顶地址保存在寄存器 regsp 中。下面是 Increment.cmm 对应的中间代码可能翻译成的目标代码：

```
 1  inc:                    # inc 函数定义，参数保存到 rega0 中
 2    load reg0, 1          # 把常量 1 加载到 0 号寄存器中
 3    add reg1, rega0, reg0# 把 a + 1 的结果保存到 1 号寄存器中
 4    store reg1, regv0     # 假设把返回结果保存到 regv0 寄存器中
 5    jr regra              # 跳转到函数调用后的下一条指令
 6  main:
 7    load reg0, 2          # 把常量 2 加载到 0 号寄存器
 8    load reg1, 2          # 把常量 2 加载到 1 号寄存器
 9    mul reg2, reg0, reg1 # 把 2*2 结果保存到 2 号寄存器
10    store reg2, lcVar     # 把结果保存到 lcVar 变量
11    store lcVar, rega0    # 传递实参
12    store regra, regsp    # 把当前指令地址保存到 regra 中
```

```
13   jar inc              # inc 函数调用
14   move regv0, reg0     # 把函数调用结果的值保存到 0 号寄存器中
15   store reg0, rtVar    # 把函数调用结果保存到 rtVar 中
16   load regv0, 0        # 把常量 0 加载到 regv0 寄存器
17   jr regra             # 函数返回
```

为了更加的具体阐述目标代码生成方法，我们选择一些典型的中间代码指令，并描述其生成过程。以中间代码中的第 7 行为例：

```
7  multemp1 := #2 * #2
```

为了生成目标代码，我们首先调用 getReg(*MUL*) 为 *MUL* 操作符分配三个寄存器，分别为 *reg0* 、*reg1* 和 *reg2*。*reg0* 和 *reg1* 用于加载 *MUL* 运算符的操作数，而 *reg2* 用于存储 *MUL* 运算结果。接下来，我们调用 binop_codegen 函数，使用 *MUL* 操作符生成目标代码。该函数将生成 MIPS 指令 mul 指令，并将其存储在目标代码中。mul 指令将 *reg0* 和 *reg1* 中的值相乘，然后将结果存储在 *reg2* 中。最后，生成的目标代码将被添加到目标代码序列中，以便在程序执行时使用。：

```
7  load reg0, 2            # 把常量 2 加载到 0 号寄存器
8  load reg1, 2            # 把常量 2 加载到 1 号寄存器
9  mul reg2, reg0, reg1    # 把 2*2 结果保存到 2 号寄存器
```

对于函数调用，例如中间代码第 8～10 三行：

```
 8  lcVar := multemp1
 9  ARG lcVar
10  calltemp1 := CALL inc
```

首先，我们将寄存器的值保存到一个名为 *lcVar* 变量中，然后使用 store 指令将参数传递到 *rega0* 中。通常情况下，指令集中有特定的寄存器用于参数传递和返回值，这些寄存器通常不会被分配给多个变量。为了处理函数递归，我们还需要提前保存 *rega0* 的值，并将其保存到堆栈中，在函数调用结束后再恢复 *rega0* 的值。接着，我们将当前栈帧指令保存到 *regra* 中，最后使用 jal 指令完成函数调用。

```
10  store reg2, lcVar      # 把结果保存到 lcVar 变量
11  store lcVar, rega0     # 传递实参
12  store regra, regsp     # 把当前指令地址保存到 regra 中
13  jar inc                # inc 函数调用
```

类似于函数调用，当函数有返回值时，我们需要将返回值保存到对应的寄存器或者内存中。通常情况下，指令集中会为返回值预留特定的寄存器，如 *regv0*。

```
14   move regv0, reg0     # 把函数调用结果的值保存到 regv0 号寄存器中
15   store reg0, rtVar    # 把函数调用结果保存到 rtVar 中
```

在目标代码生成的理论部分中，我们已经详细讨论了指令选择、寄存器分配、窥孔优化等过

程。接下来，我们将主要关注目标代码生成的实践技术，包括 QtSPIM 模拟器、MIPS 汇编代码编写、MIPS 指令集理解、寄存器分配算法实现等。当我们了解并掌握这些理论和实践技术后，结合本节的理论部分就可以实现编译器后端了。

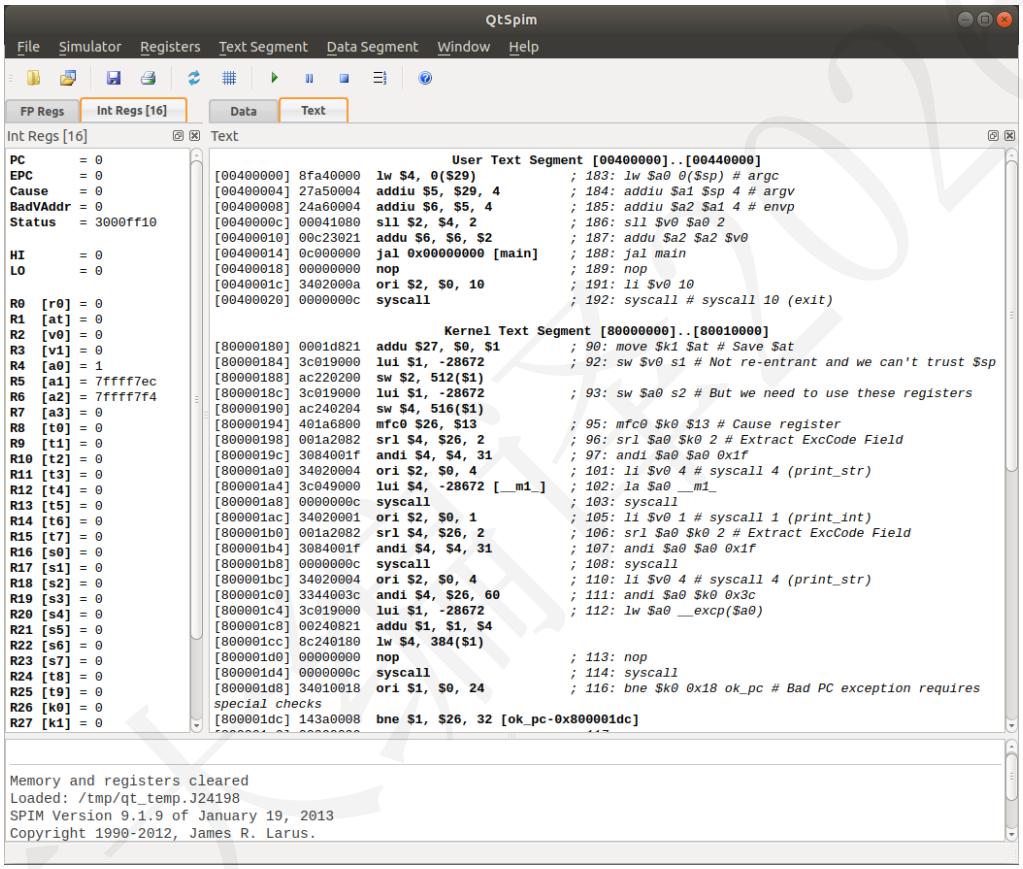

图 5. 4 QtSPIM 运行界面

5.2 目标代码生成的实践技术

在实践内容三中，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 ）我们并没有在中间代码中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问题，第二个问题被称为寄存器分配问题，第三个问题则需要考虑如何对栈进行管理。在本节中，我们的主要任务就是编写程序来处理这三个问题。

5.2.1. QtSPIM 简介

“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，让我们先来考察实践内容四所要用到的工具 SPIM Simulator。这是由原 Wisconsin-Madison 的 Jame Larus 教授领导编写的一个功能强大的 MIPS32 汇编语言的汇编器和模拟器，其最新的图形界面版本 QtSPIM 由于使用了 Qt 组件因而可以在各大操作系统平台（如 Windows 、Linux 、Mac等）上运行，推荐安装。我们会在后面介绍有关 SPIM Simulator 的使用方法。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，我们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5. 4 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。

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（例如实践内容小节样例 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。我们也可以使用 QtSPIM 工具栏上的按钮或按 F10 快捷键来单步执行该代码。

图 5.5 内存中的数据信息

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）的存储方式取决于目标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，不出意外的话我们会发现自己的 SPIM Simulator 是小端机）。

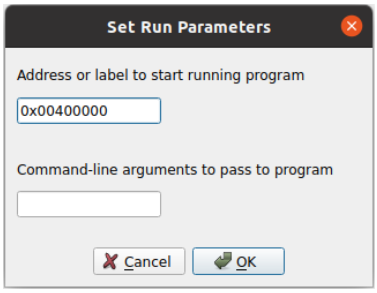

图 5.6 运行参数设置对话框

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5.5 所示。单击菜单栏上的 Simulator Run parameters，在弹出的对话框（如图5.6 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 汇编代码简介

为实现存储程序的思想，冯·诺依曼将计算机分解为五大部件：**存储器（memory）**、**运算器（Arithmetic Logic Unit，ALU）**、**控制器（control unit）**、**输入设备（input）**和**输出设备（output）**。五个部件各司其职，并有效连接以实现整体功能。运算器是负责执行逻辑运算（如与、或、非等）和算术运算（如加、减、乘、除等）的部件。控制器是能读取指令、分析指令并执行指令，以调度运算器进行计算、调度存储器进行读写的部件，它能够控制程序（一组有序的操作命令）对数据进行输入、计算或变换、以及输出，它依据事先编制好的程序，来控制计算机各个部件有条不紊地工作，完成所期望的功能。存储器是负责数据和指令保存与自动读写的部件。输入设备负责将程序和指令输入到计算机中。输出设备负责将计算机处理的结果显示或打印出来。

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，我们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。

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5.3 所列的几类。

表 5.3 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（8 位）的值于内存中。
.half h1, h2, …, hn	连续存储 n 个半字（16 位）的值于内存中。
.word w1, w2, …, wn	连续存储 n 个字（32 位）的值于内存中。
.space n	在当前段分配 n 个字节的空间。

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。后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5.4 所示。

表 5.4 QTSPIM 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）	lui Rdest, upper(addr)

[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。

	加载到寄存器 Rdest 中。	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

MIPS 体系结构共有 32 个寄存器，在汇编代码中我们可以使用 $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 5.5 所示。

表 5.5 MIPS 体系结构中的寄存器及主要功能

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。

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，我们首先需要向寄存器 $v0 中存入一个代码以指定具体要进行哪种系统调用。如有必要还需向其他寄存器中存入相关的参数，

最后再写一句 syscall 即可。例如：

表 5.6 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	

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

进行了系统调用 `print_string(_prompt)`。与实践内容四相关的系统调用类型如表 5.6 所示。

到此，如果对照后面的样例输出并仔细阅读本节的内容，我们便能基本了解在实践内容四中的程序需要输出什么。

5.2.3. 指令选择算法实现

1. 翻译模式（线形 IR）

指令选择可以看成是一种模式匹配问题。无论中间代码是线形还是树形的，我们都需要在其中找到特定的模式，然后将这些模式对应到目标代码上。指令选择过程的复杂度与中间代码本身所蕴含的信息量以及目标机器采用的指令集类型有关。我们采用的 MIPS32 指令集是一种相对简单的 RISC 指令集，因此在实践内容四中，指令选择属于比较简单的任务。

如果我们的程序使用了线形 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，即使用我们的宏扩展方法。表 5.7 提供了一个将实践内容三的中间代码对应到 MIPS32 指令的示例，当然这个翻译方案并不唯一。

表 5.7 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

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。如果按照表 5.7 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
```

[1] reg(x) 表示变量 x 所分配的寄存器。

[2] 乘法、除法以及条件转移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。

```
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或一个窥孔滑过中间代码并查找可能的翻译方案的过程。

2. 翻译模式（树形 IR）

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。但是，我们需要寻找的模式不再是一条条线形代码，而是某种结构的子树。我们可以使用树形 IR 匹配和翻译算法—树重写—来简化匹配和翻译过程。我们仍用一个例子来说明这种方法。假设现有翻译模式，如图 5.7 所示。

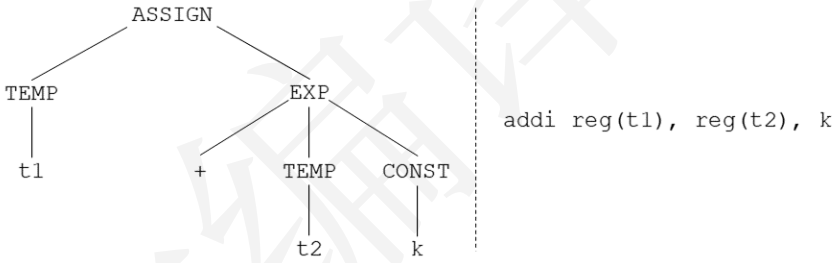

图 5.7 树形 IR 翻译示例

如何在中间代码中找到该图所对应的模式呢？答案是遍历。具体来说，我们可以按照深度优先的顺序遍历树形 IR 中的每个结点，并检查其他的类型是否符合我们所寻找的模式。例如，我们可以使用以下代码实现图 5.7 中的模式匹配。

```
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 }
```

我们可以根据自己的树形IR编写翻译模式，并使用类似上述方法进行翻译。为了提高子树覆盖效率，我们还介绍了 Maximal Munch 算法、动态规划算法和快速匹配算法等，这些算法的实现可能会相对复杂。我们可以根据自己的需求选择相应的算法实现。使用简单的树重写规则就可以完成指令选择，结合窥孔优化技术可以提升目标代码性能。

5.2.4. 朴素寄存器分配算法实现

朴素寄存器分配算法的思想简单，但是寄存器利用率较低。该算法的基本思想是将所有的变量或临时变量都放在内存里，而寄存器则只用来存放运算的操作数、函数调用参数以及返回值。由于 MIPS 寄存器数量已经能够满足目标代码指令的使用，所以我们可以使用这种简单的寄存器分配方法。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 *op* 代表一个任意的二元运算符（一元运算符或多元运算符的处理与二元运算符类似），*x* 和 *y* 是 *op* 运算符的两个操作数，*z* 保存运算结果。在基本块开始时，所有的寄存器都是闲置的。算法框架如图 5.8 所示。

```
for (对于每一个操作 z = x op y)
{
  rx = Allocate (x);
  输出 MIPS32 代码 [lw rx, x];
  ry = Allocate (y);
  输出 MIPS32 代码 [lw ry, y];
  rz = Allocate (z);
  输出操作 rz = rx op ry 的 MIPS32 代码;
  输出 MIPS32 代码 [sw rz, z];
  Free (rx);
  Free (ry);
  Free (rz);
}
```

图 5.8 朴素寄存器分配算法伪代码

其中，Free(*r*) 表示将寄存器 *r* 标记为空闲状态。该算法还用到另外一个辅助函数 Allocate，其实现如图 5.9 所示。

```
Allocate (x)
{
  if  (存在一个空闲的寄存器 r)
    result = r;
  else
  {
    把寄存器中值最晚使用的寄存器给 result;
    溢出 result;
  }
  return result;
}
```

图 5.9 Allocate 算法伪代码

Allocate 函数可以通过一次从基本块末尾向前的扫描，获取每个变量在不同程序点的使用情况。

5.2.5. 局部寄存器分配算法实现

局部寄存器分配算法会将整个代码分解成多个基本块。在每个基本块内部，我们使用各种启发式原则为出现在基本块内的变量分配寄存器。当基本块结束时，我们需要将本块中所有被修改过的变量都写回内存。我们仍然假设所有的中间代码都形如 “`z := x op y`”。在基本块开始时，所有的寄存器都是空闲的。算法框架如图 5.10 所示：

```
for (对于每一个操作 z = x op y)
{
  rx = Ensure (x);
  ry = Ensure (y);
  if (x 后续不被使用)
    Free (rx);
  if (y 后续不被使用)
    Free (ry);
  rz = Allocate (z);
  输出操作 rz = rx op ry MIPS32 代码;
}
```

图 5.10 局部寄存器分配算法伪代码

除了前面提到的 Allocate 函数之外，该算法还使用了另一个辅助函数 Ensure，其的实现如图 5.11 所示：

```
Ensure (x)
{
  if (x 已经在寄存器 r 中)
    result = r;
  else
  {
    result = Allocate (x);
    输出 MIPS32 代码 [lw result, x];
  }
  return result;
}
```

图 5.11 Ensure 算法伪代码

本书还介绍了一种功能类似于之前算法的局部寄存器分配函数 get_reg。这种方法引入了寄存器描述符和变量描述符这两种数据结构，完全消除了寄存器之间的数据移动，并且期望通过最小化溢出操作所产生的 store 指令数量来优化目标代码。

5.2.6. 活跃变量分析算法实现

1. 数据结构设计

活跃变量分析问题可以被转化为求解数据流方程的问题。实现活跃变量分析算法时，遍历基本块中每条中间代码，计算它们的入口活跃集合*in*和出口活跃集合*out*。为了更高效地计算这两个集合，通常采用**位向量**（**Bit Vector**）来表示。如果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 10 个比特位组成，其中第 *j* 个比特位为 1 代表第 *j* 个变量属于 *in*[*i*]（或 *out*[*i*]）。集合的并集对应于位向量中的或运算，集合的交集对应于位向量中的与运算，集合的补集对应于位向量中的非运算。位向量的紧凑表示和快速运算使它成为几乎所有数据流方程算法中用于表示集合的理想数据结构。

2. 活跃性计算

```
LivenessAnalysis (ir)
{
  for (对于 ir 中每一条中间码 i)
  {
    in[i] = ϕ;
    out[i] = ϕ;
  }
  loop:
  for (对于 ir 中每一条中间码 i)
  {
    in'[i] = in[i];
    out'[i] = out[i];
    in[i] = use[i] ∪(out[i]-def[i]);
    out[i] = U_{j∈succ[j]} in[j];
  }
  for (对于 ir 中每一条中间码 i)
    if (不满足(in'[i] == in[i] and out'[i] == out[i]))
      goto loop;
}
```

图 5. 12 活跃性计算的迭代算法伪代码

我们可以通过迭代的方式来求解每条中间代码的活跃变量集合。在算法开始时，我们将所有的 $in[i]$ 初始化为空集 ϕ。接下来，我们对于每条中间代码对应的 *in* 和 *out* 集合进行运算，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* 从大到小的顺序进行计算 *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。

活跃性计算的迭代方法如图5. 12 所示。LivenessAnalysis 函数用于迭代计算中间代码中的活跃变量信息。当算法终止时，我们就已经求解了每条中间代码的入口活跃集合和出口活跃集合。

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，还可以用于**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。我们上面介绍的方法是以语句为单位进行分析的，类似的方法也适用于以基本块为单位的情况，而使用基本块的话分析效率还会更高一些。

5.2.7. 图染色算法实现

1. 数据结构设计

为了实现图染色算法，我们需要先构造冲突图。冲突图可以用邻接表或二维矩阵表示。在算法执行过程中，我们需要频繁地查询冲突图，通常包含两种操作：

（ 1 ）查询冲突图中与顶点 x 相邻的所有节点。

（ 2 ）判断图中任意两个节点 x 和 y 之间是否存在一条边。

如果我们使用邻接表表示冲突图，可以快速完成第一种查询。对于第二种查询，需要遍历 x 对应的相邻节点链表，这个过程通常比较慢。如果我们使用矩阵实现，那么正好与邻接表实现的效果相反。我们可以快速完成第二种查询，但第一种查询效率会降低。在实现图染色算法时，我们可以根据需要选择合适的数据结构来表示冲突图。

2. 算法实现[1]

```
ColoringGraphAlgorithm (ir, K, stack)
{
   graph = BuildGraph (ir);
   allocation = ColorGraph (graph, stack, K);
   if (allocation == false)
   {
      cost = EstimateSpillCosts (ir);
      spilled = DecideSpills (graph, ir, K, cost);
      InsertSpillCode (ir, spilled);
      graph = ColoringGraphAlgorithm (ir, K, stack);
   }
   return graph;
}
```

图 5. 13 图染色算法伪代码

[1] 该算法来源于：https://github.com/johnflanigan/graph-coloring-via-register-allocation

针对固定的颜色数 *K*，我们可以采用上述算法对干涉图进行构造和着色，算法如图 5.13 所示。具体而言，我们先使用 BuildGraph 函数构造干涉图，该函数接受中间代码 *ir* 作为参数。接下来，我们使用 ColorGraph 函数对干涉图进行着色。该函数接受中间代码 *ir*、保存顶点信息的栈 *stack* 以及可用的寄存器数目 *K* 作为参数，并判断当前干涉图是否可以被 *K* 着色。这两个函数的实现细节可以参考图 5.14 和 图 5.15。

```
BuildGraph (ir)
{
   graph = Graph();
   LivenessAnalysis (ir);
   for (对于 ir 中每一条中间码 i )
      for (对于 def[i] 集合中每一个变量 d)
         for (对于 out[i]集合中每一个变量 o)
         {
           AddEdge (graph, d, o);
           out 集合中每一对变量加入 graph 中;
         }
   return graph;
}
```

图 5.14 干涉图构造算法伪代码

```
ColorGraph (graph, stack, K)
{
   if (节点 n 的度数小于 K)
   {
      RemoveNode (graph, n);
      Push (stack, n);
      allocation = ColorGraph (graph, stack, K);
      if (allocation == false)
         return false;
      for (对于 graph 中每一个节点 n)
         从可用的寄存器集合选择一个合适的寄存器保存到 n.reg 中;
   }
   return false;
}
```

图 5.15 图染色算法伪代码

如果我们能够对干涉图进行染色，那么就无须将干涉图中的节点的溢出到内存中。我们可以直接返回干涉图，并根据栈中保存的节点信息为程序中的变量分配寄存器。

但是当可用的寄存器数目小于变量数目时，我们需要将干涉图中的那些邻居节点数量大于可用寄存器数目的节点溢出到内存中。为此，我们定义了一个 DecideSpills 函数，用于确定待溢出的节点，具体实现如图 5.17 所示。其中，EstimateSpillCosts 函数用于评估溢出节点的代价，如图 5.16 所示。我们遍历每一个 *label* 内的程序点的 *def* 集合以及 *use* 集合中使用的变量，计算这些变量的邻居节点的数量。当 DecideSpills 函数确定需要溢出到内存的变量时，我们就从 *cost* 集合中选择溢出代价最小的顶点，并将其标记为溢出，且从干涉图中删除该节点。

```
EstimateSpillCosts (ir)
{
  初始化 spilled 为一个空集合;
  frequency = 1;
  for (对于每一条 ir i)
  {
    if (i 是一个 label)
      frequency = 1;
    else
    {
      初始化 neighbors 为一个空集合;
      for (对于每一个 def[i] 集合中的变量 d)
        neighbors.add(d);
      for (对于每一个 use[i] 集合中的变量 u)
        neighbors.add(u);
      for (对于 neighbors 中每一个节点 n)
        if (n 在 cost 集合中)
          cost[n] = cost[n] + frequency;
        else
          cost[n] = frequency;
    }
  }
  return cost;
```

图 5.16 估溢出代价算法伪代码

在 DecideSpills 函数调用之后，我们就确定了需要溢出到内存的节点。接下来，我们可以使用 InsertSpillCode 函数来插入变量溢出到内存的操作，并更新中间代码，具体实现详见图 5.18 所示：

```
DecideSpills (graph, ir, K, cost)
{
   初始化 spilled 为一个空集合;
   复制 graph 到 g 中;
   for (对于每一个 g 中的节点 n)
   {
      选择节点边数小于 K 的节点 n;
      if (n 不存在)
      {
         从 cost 集合中选择代价最小的节点 node;
         spilled.add(node);
      }
      RemoveNode (g, node);
   }
   return spilled;
}
```

图 5.17 *DecideSpills* 算法伪代码

由于中间代码中主要包含函数或转移指令的目的地址所在的 *label*，因此在插入溢出节点的操作中，我们需要遍历每一条中间代码，并修改每个 *label*。然后，我们遍历 *label* 中的每一条中间代码指令中的 *use* 和 *def* 集合。如果 *use* 集合中的节点被标记为溢出，那么我们需要插入一条 *load* 指令来从内存中加载 *use* 中使用的变量。如果 *def* 集合中的节点被标记为溢出，那么我们需要插入一条 *store* 指令，用于将节点溢出到内存中。最后，我们更新当前的中间代码即可完成插入溢出代码的操作。

```
InsertSpillCode (ir, spilled)
{
  创建一个新的 ir 数组 new_ir;
  for (对于每一条 ir i)
  {
    if (i 是一个 label)
      添加 I 到 new_ir 中;
    else
    {
      before = [];
      after = [];
      newdef = [];
      newuse = [];
      for (对于每一个 use[i] 集合中的变量 u)
        if (u 在 spilled 集合中)
        {
          newuse.append(u);
          before.append(load ir);
        }
        else
          newuse.append(u);
      for (对于每一个在 def[i] 集合中的变量 d)
      {
        if (d 在 spilled 集合中)
        {
          newdef.append(d);
          after.append(store ir);
        }
        else
          newdef.append(d);
      }
      new_ir.extend(before + ir(opcode, newdef, newuse) + after);
  }
  使用 new_ir 覆盖 ir;
}
```

图 5. 18 *InsertSpillCode* 算法伪代码

5.2.8. 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提出了一些约定。“约定”这个词有两层含义：其一，因为它是约定，所以我们不必强制遵守它，大可违反它却仍然使我们写出的汇编代码能够正常运行起来；其二，因为它是约定，所以除了我们之外的绝大部分人（还有包括 SPIM Simulator 在内的绝大多数汇编器）都在遵守它。如果我们不遵守它，那么我们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 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，如果我们尝试在汇编代码中访问或修改它们，则 SPIM Simulator 会报错。$v0 和 $v1 这两个寄存器专门用来存放函数的返回值。在函数内部也可以使用，不过要注意在当前函数返回或调用其他函数时应妥善处理这两个寄存器中原有的数据。$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 ，则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 $fp 使用，我们可以在这两个方案里任选其一。$ra 专门用来保存函数的返回地址，

MIPS32 中与函数转移有关的 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. MIPS 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，我们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 可以挪作他用。

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掉。为了使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，实践内容四中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。

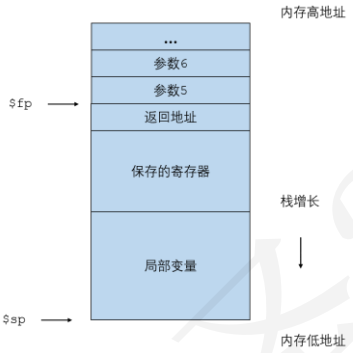

图 5.19 函数的活动记录结构

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 5.19 所示。

在图 5.19 中，图上方是高地址区，下方是低地址区，栈是从高地址区往低地址区增长。栈指针 $sp 总指向最后一个压入栈的数据所在位置（栈顶），帧指针 $fp 则指向当前活动记录的底部；在图中，$fp 之上是传给该函数的参数（只有多于 4 个参数时这里才会有内容），而 $fp 之下则是返回地址、被调用者保存的寄存器内容以及局部数组、变量或临时变量等信息。

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。假如我们学过有关 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，那么为了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应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 这三个寄存器由调用者保存，而 %ebx 、%esi 和 %edi 这三个寄存器则由被调用者保存。不过由于存在使用方便的 pushad 和 popad 指令，在人工书写汇编代码时人们常常在 6 个通用寄存器的基础上添加 %ebp 和 %esp （共 8 个寄存器），并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, $sp, max{0, 4 * (n - 4)}
13 lw live1, offset_live1($sp)
14 …
15 lw livek, offset_livek($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，则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，则应使用帧指针 $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他函数，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, $sp, framesizeg
5  sw reg1, offset_reg1($sp)
6  …
```

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

```
 7  sw reg_k, offset_regk($sp)
 8  lw p_5, (framesize_g)($sp)
 9  …
10  lw p_n, (framesize_g + 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 reg_1, offset_reg1($sp)
2  …
3  lw reg_k, offset_regk($sp)
4  lw $ra, (framesize_g - 4)($sp)
5  lw $fp, (framesize_g -8)($sp)
6  addi $sp, $sp, framesize_g
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 $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，因此我们不需要特殊考虑它们。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，我们就需要在调用序列中将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。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，我们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。如果对 x86 足够了解，就会发现这个 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：凡是我们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，对于我们的实践内容而言，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。我们不由得反思，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. 目标代码生成实践的额外提示

实践内容四需要我们在实践内容三的基础上完成。在开始写代码之前，我们需要先熟悉 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。

完成实践内容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我们先定义一系列数据结构。如果采用了局部寄存器分配算法，我们可能需要考虑如何实现寄存器描述符和变量描述符。如果使用前面介绍的局部寄存器分配算法，我们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；如果使用前面部分介绍的局部寄存器分配算法，我们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，我们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，专心处理其他类型的中间代码。我们可以先假设寄存器有无限多个（编号为：$t0 ，$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，这时我们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，我们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。我们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 时要注意不要搞错实参和形参的顺序，另外计算实参时如果我们没有使用 $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 的单步执行功能对编译器输出的代码进行调试。

5.3 目标代码生成的实践内容

5.3.1. 实践要求

为了完成实践内容四，我们需要下载并安装 SPIM Simulator 用于对生成的目标代码进行检查和调试。我们需要做的就是将实践内容三中得到的中间代码经过与具体体系结构相关的指令选择、寄存器选择以及栈管理之后，转换为 MIPS32 汇编代码。我们的程序需要输出正确的汇编代码。“正确”是指该汇编代码在 SPIM Simulator（命令行或 Qt 版本均可）上可正确运行。因此，以下几个方面不属于检查范围：

（ 1 ）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，我们可以随意分配 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。

（ 2 ）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。我们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，前提是输出的目标代码（即 MIPS32 汇编代码）能运行正确。

当然，不检查并不代表不重要。我们可以试着去遵守 MIPS32 中的各种约定，否则代码生成器生成的目标代码在 SPIM Simulator 中运行时可能会出现一些意想不到的错误。

另外，实践内容四对作为输入的 C— 源代码有如下的假设：

（ 1 ）**假设1**：输入文件中不包含任何词法、语法或语义错误（函数必有 return 语句）。

（ 2 ）**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。

（ 3 ）**假设3**：整型常数都在 16 位的整数范围内，也就是说我们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。

（ 4 ）**假设4**：不会出现类型为结构体或高维数组（高于一维的数组）的变量。

（ 5 ）**假设5**：不使用全局变量，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。

（ 6 ）**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。

（ 7 ）**假设7**：函数不会返回结构体或数组类型的值。

（ 8 ）**假设8**：函数只会进行一次定义（没有函数声明）。

在进行实践内容四时，请仔细阅读前面的理论部分和实践内容指导部分，确保我们已经了解 MIPS32 汇编语言以及 SPIM Simulator 的使用方法，这些内容是我们能够顺利完成实践内容四的前提。

5.3.2. 输入格式

程序的输入是一个包含 C— 源代码的文本文件，我们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。例如，假设我们的程序名为 cc 、输入文件名为 test1.cmm 、输出文件名为 out1.s，程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行 `./cc test1.cmm out1.s` 即可将输出结果写入当前目录下名为 out1.s 的文件中。

5.3.3. 输出格式

实践内容四要求我们的程序将运行结果输出到文件。对于每个输入文件，我们的程序应当输出相应的 MIPS32 汇编代码。我们可以使用 SPIM Simulator 对代码生成器输出的汇编代码的正确性进行测试，任何能被 SPIM Simulator 执行并且结果正确的输出都将被接受。

5.3.4. 验证环境

我们的程序将在如下环境中被编译并运行：

- GNU Linux Release: Ubuntu 20.04, kernel version 5.13.0-44-generic
- GCC version 7.5.0
- GNU Flex version 2.6.4
- GNU Bison version 3.5.1
- QtSPIM version 9.1.9

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实践内容四的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在我们的程序中使用它们。

5.3.5. 提交要求

实践内容四要求提交如下内容（同实践内容一）：

（ 1 ）Flex、Bison 以及 C 语言的可被正确编译运行的源程序。

（ 2 ）一份PDF格式的实验报告，内容包括：

1 ）**你的程序实现了哪些功能**？简要说明如何实现这些功能。清晰的说明有助于助教对你的程序所实现的功能进行合理的测试。

2 ）**你的程序应该如何被编译**？可以使用脚本、makefile 或逐条输入命令进行编译，请详细说明应该如何编译我们的程序。无法顺利编译将导致助教无法对你的程序所实现的功能进行任何测试，从而丢失相应的分数。

3 ）**实验报告的长度不得超过三页**！所以报告中需要重点描述的是程序中的亮点，是开发人员认为最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。实验报告中所出现的最小字号不得小于 5 号字（或英文 11 号字）。

5.3.6. 样例（必做部分）

实践内容四无选做要求，因此下面只列举必做内容样例。请仔细阅读样例，以加深对实践内容要求以及输出格式要求的理解。

【样例 1】

输入：

```
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;
}
```

输出：

该样例程序读入一个整数 *n*，然后计算并输出前 *n* 个 Fibonacci 数的值。将其翻译为一段能在 SPIM Simulator 中执行的正确的目标代码，如下所示：

```
 1  .data
 2  _prompt: .asciiz "Enter an integer:"
 3  _ret: .asciiz "\n"
 4  .globl main
 5  .text
 6  read:
 7    li $v0, 4
 8    la $a0, _prompt
 9    syscall
10    li $v0, 5
11    syscall
12    jr $ra
13
14  write:
15    li $v0, 1
16    syscall
17    li $v0, 4
18    la $a0, _ret
19    syscall
20    move $v0, $0
21    jr $ra
22
23  main:
24    li $t5, 0
25    li $t4, 1
26    li $t3, 0
27    addi $sp, $sp, -4
28    sw $ra, 0($sp)
29    jal read
30    lw $ra, 0($sp)
31    addi $sp, $sp, 4
32    move $t1, $v0
33    move $t2, $t1
34  label1:
35    blt $t3, $t2, label2
36    j label3
37  label2:
38    add $t1, $t5, $t4
39    move $a0, $t4
40    addi $sp, $sp, -4
41    sw $ra, 0($sp)
42    jal write
43    lw $ra, 0($sp)
44    addi $sp, $sp, 4
45    move $t5, $t4
46    move $t4, $t1
47    addi $t1, $t3, 1
48    move $t3, $t1
49    j label1
50  label3:
51    move $v0, $0
52    jr $ra
```

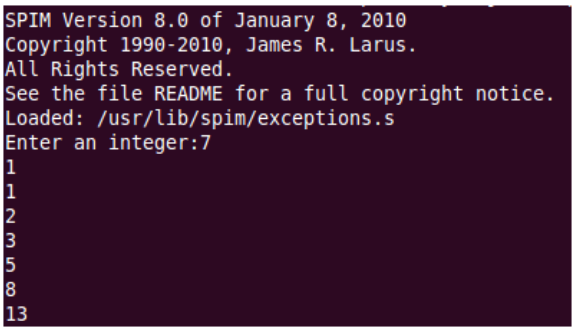

图 5.20 样例 1 汇编代码的运行结果

该汇编代码在命令行 SPIM Simulator 中的运行结果如图 5.20 所示（输入 7，则输出前 7 个 Fibonacci 数）。

【样例2】

输入：

```
1  int fact(int n)
2  {
3    if (n == 1)
4      return n;
5    else
6      return (n * fact(n - 1));
7    }
8
9  int main()
10 {
11   int m, result;
12   m = read();
13   if (m > 1)
14     result = fact(m);
15   else
16     result = 1;
17   write(result);
18   return 0;
19 }
```

输出：

该样例程序读入一个整数 n ，然后计算并输出 $n!$ 的值。将其翻译为一段能在 SPIM Simulator 中执行的正确的目标代码，如下所示：

```
1  .data
2  _prompt: .asciiz "Enter an integer:"
3  _ret: .asciiz "\n"
4  .globl main
5  .text
6  read:
7    li $v0, 4
8    la $a0, _prompt
```

```
 9   syscall
10   li $v0, 5
11   syscall
12   jr $ra
13
14 write:
15   li $v0, 1
16   syscall
17   li $v0, 4
18   la $a0, _ret
19   syscall
20   move $v0, $0
21   jr $ra
22
23 main:
24   addi $sp, $sp, -4
25   sw $ra, 0($sp)
26   jal read
27   lw $ra, 0($sp)
28   addi $sp, $sp, 4
29   move $t1, $v0
30   li $t3, 1
31   bgt $t1, $t3, label6
32   j label7
33 label6:
34   move $a0, $t1
35   addi $sp, $sp, -4
36   sw $ra, 0($sp)
37   jal fact
38   lw $ra, 0($sp)
39   addi $sp, $sp, 4
40   move $t2, $v0
41   j label8
42 label7:
43   li $t2, 1
44 label8:
45   move $a0, $t2
46   addi $sp, $sp, -4
47   sw $ra, 0($sp)
48   jal write
49   lw $ra, 0($sp)
50   addi $sp, $sp, 4
51   move $v0, $0
52   jr $ra
53
54 fact:
55   li $t4, 1
56   beq $a0, $t4, label1
57   j label2
58 label1:
59   move $v0, $a0
60   jr $ra
61 label2:
62   addi $sp, $sp, -8
63   sw $a0, ($sp)
64   sw $ra, 4($sp)
65   sub $a0, $a0, 1
66   jal fact
67   lw $a0, ($sp)
```

```
68    lw $ra, 4($sp)
69    addi $sp, $sp, 8
70    mul $v0, $v0, $a0
71    jr $ra
```

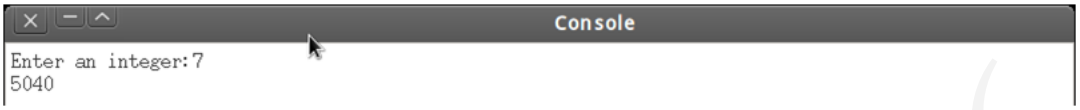

图 5.21 样例 2 汇编代码的运行结果

该汇编程序在 QtSPIM 中的运行结果如图 5.21 所示（输入 7，输出 5040）。

除了上面给的两个样例以外，我们的程序要能够将其他符合假设的 C— 源代码翻译为目标代码，我们将通过检查目标代码是否能在 SPIM Simulator 上运行并得到正确结果来判断程序的正确性。

5.4 本章小结

在本章中，我们已经详细讨论了目标代码生成的理论以及实践技术。目标代码生成的任务是在词法分析、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。其中，我们着重分析了目标代码生成的线形 IR 和树形 IR 指令选择算法、寄存器分配算法和窥孔优化内容及关键算法设计与实现。在此基础上，我们实现了编译器后端——代码生成器，从而完整地实现一个 C— 的编译器。此外，我们还可以扩展编译器的功能，优化编译器各个阶段输出的结果，以提升目标代码的运行效率。

习题

5.1 有如下代码：

```
 1   L1: temp = x + y
 2   z = z - temp
 3   u = z + v
 4   if u > 0 goto L2
 5   goto L3:
 6   L2: v = u + 1
 7   goto L4
 8   L3: x = z + u
 9   u = u - 1
10   L4: x = v + y
11   goto L1
```

试划分基本块，并构造流图。

5.2 根据第 1 题的流图，构造干涉图。

5.3 考虑如下基本块，其中 u 是出口活跃变量，计算各个变量待用信息和活跃信息。

```
100   T1 := a + b
101   T2 := a - b
102   T3 := a * b
103   T4 := a / b
104   T5 := T1 + T4
105   T6 := T2 + T3
106   T7 := T1 * T6
107   u := T5 + T7
```

5.4 将以下中间代码生成为目标代码，假设寄存器 R0～R3 可用。

```
100   if x<y goto 102
101   goto 107
102   if a<b goto 104
103   goto 107
104   T1 := y + z
105   x := T1
106   goto 109
107   T2 := y - z
108   x := T2
109   ……
```

5.5 将以下中间代码生成为目标代码，假设寄存器 R0～R3 可用。

```
100    if x<y goto 102
101    goto 107
102    if a<b goto 104
103    goto 107
104    T1 := y + z
105    x := T1
106    goto 100
107    ……
```

5.6 在以下流图中，非局部变量 b, d 在循环出口处活跃，求各基本块入口和出口处的活跃变量。

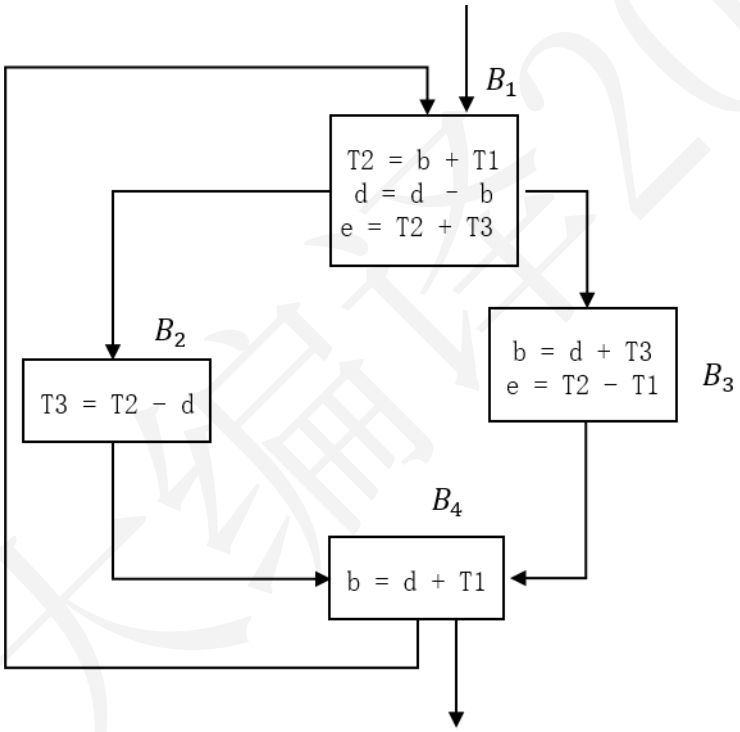

图 5.22 习题 5.5 的流图

5.7 由以下代码，写出窥孔优化消除冗余存取后的代码。

```
1    load R0, x
2    load R1, y
3    addi R2, R0, R1
4    store R2, x
5    store R1, y
6    load R2, x
7    load R1, y
8    addi R0, R1, R2
```

```
9    store R0, z
```

5.8 由以下代码，对控制流进行窥孔优化，写出优化后的代码。

```
1    if a < b goto L1
2    load R0, x
3    load R1, y
4    add R2, R0, R1
5    store R2, z
6    L1: goto L2
7    ……
```