UNICORE32-MINIC 实习报告

作者:	ID
彭焯	00848303
华连盛	00848143
王衍	00848324
李春奇	00848083

2010 年 12 月 30 日

目录

第一章 词法、语法分析 **6**

1.1 词法分析, flex . . . 6

1.2 语法分析, bison . . . 6

1.2.1 MiniC 文法 . . . 6

1.2.2 利用 bison 进行文法分析 . . . 7

1.2.3 语法错误检查 . . . 8

1.3 谬误与陷阱 . . . 9

第二章 中间表示 **10**

2.1 抽象语法树 AST . . . 10

2.1.1 生成 . . . 10

2.1.2 生成的语法树示例 . . . 11

2.1.3 AST 的通用周游模板 . . . 11

2.1.4 符号表 . . . 12

2.1.5 符号表示例 . . . 13

2.1.6 类型检查 . . . 14

2.2 三元式 . . . 15

2.2.1 三元式的结构 . . . 16

2.2.2 三元式指令列表 . . . 16

2.2.3 将 AST 转换为三元式 . . . 17

2.2.4 生成基本块 . . . 18

2.3 机器无关优化 . . . 19

2.3.1 窥孔优化 . . . 19

2.3.2 数据流分析及相关优化 . . . 20

2.4 谬误与陷阱 . . . 23

第三章 目标代码 **25**

3.1 编译器 -目标机器二进制接口 . . . 25

3.1.1 寄存器使用情况 . . . 25

3.1.2 内存的维护 . . . 26

3.2 寄存器分配 . . . 27

3.3 将三元式转换为目标代码 . . . 28

3.3.1 控制流语句的翻译 . . . 28

3.3.2 寄存器保护与同步 . . . 28

3.3.3 立即数的处理 . . . 28

3.4 机器相关优化 . . . 29

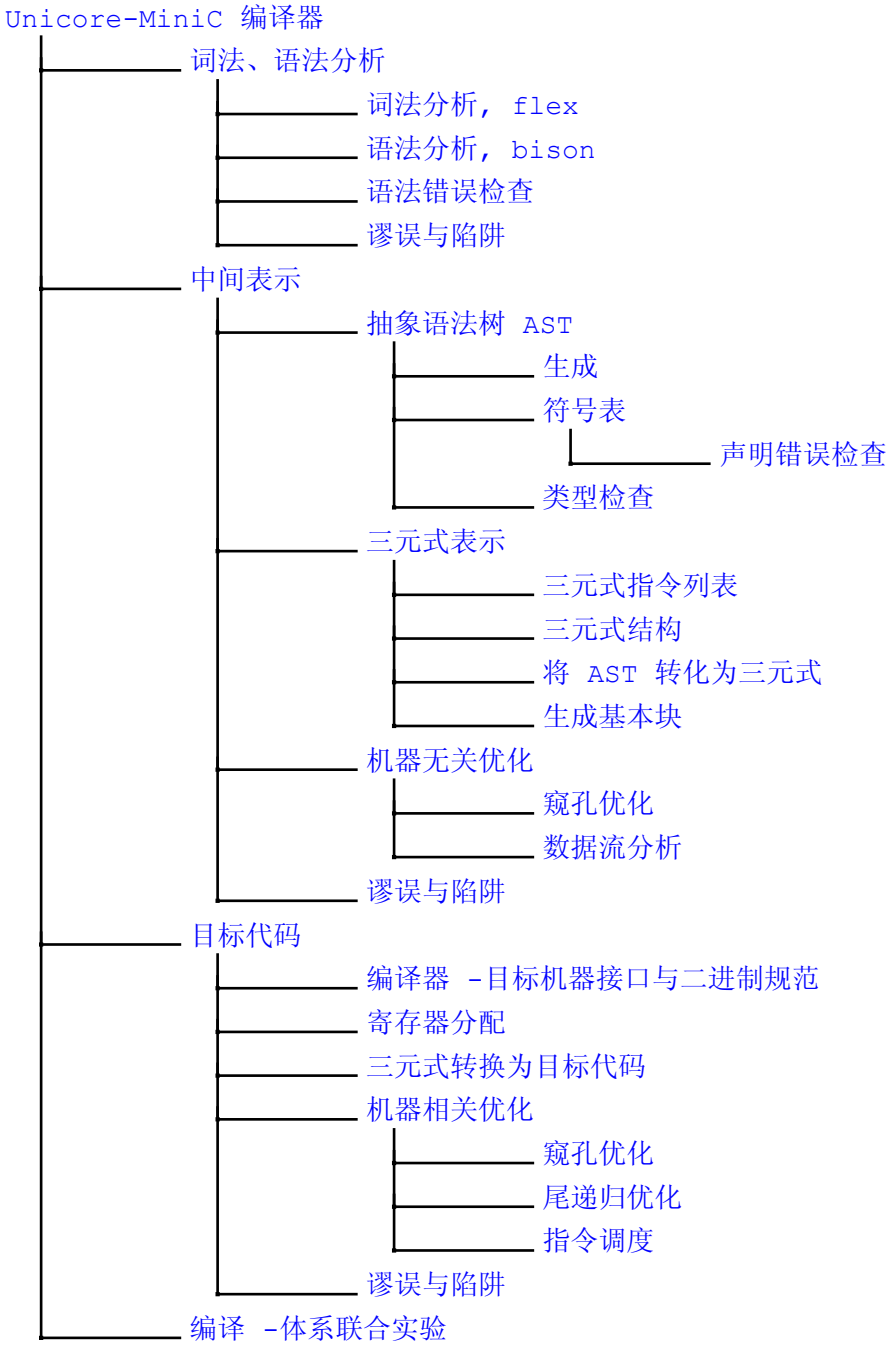
3.4.1 窥孔优化 29
3.4.2 尾递归优化 30
3.4.3 指令调度 31
3.5 谬误与陷阱 32

第四章 MiniC-Unicore32 模拟器性能实验 34

简介

如何阅读本文档

下面给出了本文档的结构，单击目录树上的节点便可以查看文档的相关部分。



本文档包括的内容

本文档的目标并不是用于展示 MiniC 冗杂的代码，而是提供一个概括的全局视角来了解这个实习项目。文档涵盖了如下的几个方面：

- Unicore-MiniC 编译器（以下在没有歧义的情况下简称之为 MiniC）的总体设计方案
- MiniC 的各个单元的设计方案，包括主要的数据结构和重要的算法
- MiniC 各个单元之间的协同关系
- 为相关代码的阅读提供的指引
- 在开发过程中遇到的困难，遭遇的陷阱和付出的代价[1]

MiniC In a Nutshell

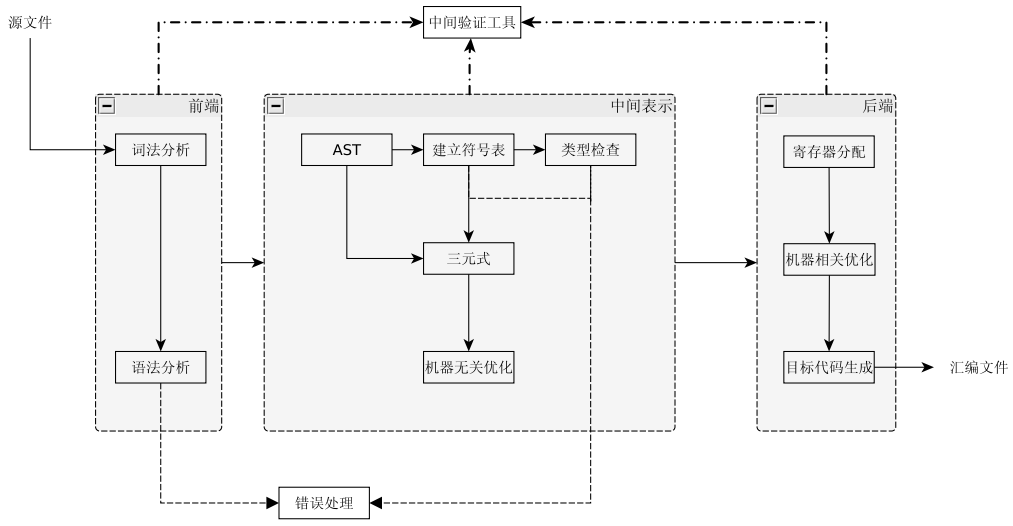
项目目标

- 输入 MiniC 源代码，输出 Unicore32 汇编代码
- 利用 Unicore32 二进制工具链中的链结器链接后的程序能在 Unicore32 实体机器上运行
- 链接后的程序能够在同时设计的 Unicore32 模拟器上运行
- 通过对模拟器相关参数、指令系统的修改，考察编译 -ISA 对性能的综合影响

整个项目将全部以 C 语言实现，用`gcc` 进行编译；以`GNU/Linux, i386` 为运行平台。使用`svn` 进行版本控制。项目主页：code.google.com/p/minic

项目结构

下面是一个 MiniC 编译器的总体结构图：



[1] 参见某些章中的“谬误与陷阱”一节

总体上看，MiniC 由三大模块构成：

1. 前端：接收输入文件，结合文法，负责语法、词法分析；将源文件的结构和内容信息提取到抽象语法树 (AST) 上
2. 中间表示：共有两层：AST 和三元式；语义分析在 AST 上完成后，将 AST 变换为三元式；机器无关优化在三元式表示上进行。
3. 后端：目标机器代码（汇编代码）生成；结合目标机器二进制规范和 ISA，生成目标代码；机器相关优化在目标代码上进行。

与此同时，还有错误处理模块，用于在源文件有误时产生出错提示；中间验证工具集，用于检查工程进展的每一个步骤的正确性。
在下面的章节中，我们将一一介绍以上模块。并且还将在最后一章展示 MiniC 编译器是如何与体系实习的模拟器项目配合，进行一系列编译 -体系相关的实验。

致谢

- 感谢刘先华老师对我们小组的指导。他总能及时、耐心地解答我们的问题，与此同时又鼓励我们积极思考和大胆尝试，使我们从这个实习项目中收获了知识和快乐。
- 感谢刘锋老师在工程进展和规划、模拟器 -编译器结合方面提供的帮助。
- 感谢与我们一同选择 MiniC 项目的另外两个小组的成员，你们的帮助和启发使我们受益匪浅。
- 当然，最需要感谢的是这个项目本身：我们通过它同时体验了 CPU 设计人员、模拟器设计人员、甚至是（一部分）操作系统设计人员在设计一个大型的、可靠的系统时所面临的挑战，以及克服这些挑战后所获得的成就感。

下面，我们将从头开始，介绍 UNICORE-MINIC编译器。

第一章　词法、语法分析

1.1　词法分析, flex

词法分析的目的是将输入的源文件中的符号识别为语法分析器能够接受的 tokens。其基本原理是利用正则表达式 pattern 来匹配、分割源文件中的符号。同时，词法分析还需要识别并保存一些名字和常量，比如函数名、变量名和字符串常量。
MiniC 的词法规则同标准 C 的对应规则基本相同：

1. 标识符必须以下划线或大小写字母开头，由下划线、大小写字母和数字组成
2. 字符常量需要放在单引号'' 中，字符串常量需要放在双引号"" 中
3. 有如下保留字不能当作标识符：`extern`, `register`, `void`, `int`, `char`, `if`, `else`, `for`, `while`, `return`
4. 特殊符号包括：`{}`, `()`, `[]`, `+`, `-`, `*`, `!`, `&`, `=`, `|`, `>`, `<`

由于语法分析器 bison 实际上不能获得AST 叶节点上的信息，因此生成 AST 叶节点的工作交由 flex 完成。

⚓*flex 源文件请参阅*：`minic.l`

1.2　语法分析, bison

语法分析的目的是根据语言的 BNF 范式，将词法分析器提交的 token 流进行规约，在规约的同时应用一些语法规则。语法分析的结果有两个：

1. 将语法正确的源文件转换为语法规则所要求的中间形式，这一中间形式不在具有语言的语法特性，或
2. 发现语法错误，如果错误能够暂时恢复，就继续语法分析，在完成后提示错误信息；否则，直接停止语法分析，报告错误。

MiniC 项目利用 bison 辅助进行语法分析，分析完成后，将建立一棵 AST。
在介绍对 bison 的利用前，由于我们的项目修改了给定的 MiniC 文法，所以首先要对修改后的文法进行说明。

1.2.1　MiniC 文法

我们实现的 MiniC 文法和实习项目开始时给定的有所不同，最大的不同之处在表达式文法部分。以下是原表达式文法：

```
unary_expr := unary_op unary_expr | postfix_expr
unary_op := ! | + | - | ++ | -- | & | *
postfix_expr := postfix_expr ( expression )
| ident ( argumentlist )
| ident ( )
| postfix_expr ++
| postfix_expr --
| primary_expr
```

注意到这种文法允许诸如下述的语法：

- `++ ++ a;`（等价于`a=a+1=a+1`）
- `!a = 1;`
- `f()=1;`

而这些语法实际上是语言所不能处理的，因为“计算结果”和“代表一个存储位置的符号”是完全不同的概念，前者没有地方存储也因此不能被赋值。 我们修改后的表达式文法基于“左值”——代表一个存储区域的符号和“右值”——计算结果，并且是二义性文法：

```
expression := rvalue | assignment_expression
assignment_expression:= lvalue = assignment_expression
| lvalue = rvalue
lvalue := * rvalue | IDENT | IDENT [ expression ]
rvalue := lvalue
| rvalue + rvalue
| rvalue - rvalue
| rvalue * rvalue
| rvalue op rvalue
| ( rvalue )
| + rvalue
| - rvalue
| ! rvalue
| & lvalue
| IDENT ( argument_list )
| IDENT ( )
...
```

这样定义的文法在表达的意义上更清晰，同时也避免了上述的错误。

⚓ 有关原文法漏洞的详细说明，请参阅：`BNF_leaks.pdf`

⚓ 本项目的全部 *BNF* 范式请参阅：`BNF.pdf`

我们对原 MiniC 文法的改动主要在其表达式文法部分。

1.2.2 利用 bison 进行文法分析

bison 是一个能够自动构建 $LALR(1)$（或 $LR(K)$）语法分析器的软件。它的输入是一个文法描述，输出是一个能够直接编译的 C 源文件，只需调用`yyparse()`函数，就可以开始按照文法进行规约，同时执行相应的文法规则。

使用 bison 需要注意几个问题：

1. LR 分析允许文法中有左递归
2. bison 允许使用二义性文法，但是需要指名一个规约规则，否则会提示规约 -规约冲突。例如，为了使文法设计简单，MiniC 使用的表达式文法就是二义性文法，但是利用 bison 提供的“定义规约优先级”和“定义结合律”功能，就可以实现按照指定的算符优先级来进行规约。
3. 每个终结符和非终结符都有同样的“属性”结构，因此，在声明这个结构时需要考虑到属性文法中所要用到的所有属性。下面是 MiniC 语法分析器所用的属性结构：

```
1 %union{
2     iattr int_attr; //可能具有的整数属性
3     cattr char_attr; //可能具有的字符属性
4     sattr string_attr; //可能具有的字符串属性
5     AST_NODE* ptr; //指向子节点的指针
6 }
```

只有叶子节点才会使用上面三种属性，因为它们可能包含标识符名称或常量。

⚓ 本项目的 *bison* 源文件请参阅：`minic.y`

1.2.3　语法错误检查

bison 提供了完善的错误检查功能，当发现一个语法错误时，它会暂停当前的规约过程，并在当前规约到的表达式后添加一个”error” 终结符，然后开始在规约栈中从error 以下弹出符号，直到找到一个带有”error” 终结符的产生式，便用它规约，并应用相应的文法规则（一般来说是错误处理代码）；然后文法分析恢复正常运行。如果找不到用于处理错误的文法，那么分析器报错退出。
与此同时，bison 还提供了追踪终结符/非终结符在源文件中的位置的机制，只需要词法分析器在一个名为yylloc 的结构中填入信息，这个结构中记录了终结符的开始行列编号和结束行列编号。bison 将在规约的过程中自动维护每个文法符号的yylloc。
MiniC 添加了简单的错误处理/错误代码追踪功能。我们在lvalue 和rvalue 产生式集中各添加了一个error 单一产生式，并且加入了报错的代码，因此，一旦 MiniC 源文件中表达式出现错误，语法分析器就会指示其位置。
例如，下面代码第 4 行少了一个分号：

```
1 int main()
2 {
3     int i;
4     i = i + 1
5 }
```

编译器会提示：

```
Syntax error
Parsing error: line 4.0-5.1: bad lvalue
 Parsing error: line 4.0-5.1: bad lvalue
 Parser: terminated, 2 error(s).
```

1.3　谬误与陷阱

谬误 1： 文法的设计可以草率为之，将问题留给语义检查会比考虑一个正确的文法更方便。
文法具有其数学上的严谨性，一条有漏洞的文法可能会导致允许的语言集合过大或过小。最为危险的是如果这个集合过大，那么一类或几类错误的语言集合就会顺利通过语法分析：这类集合在文法上应当具有相似的结构，但是在语义上就可能很不相同，如果在语义分析时再检查这些错误，将造成复杂的代码逻辑和冗长的分支。

谬误 2： 语法所支持的就是语言应支持的。
通过了语法分析仅仅以为着符合语法的规范，确不意味着语义的正确。例如：`void` 关键字可以修饰一个变量，这是符合 MiniC 语法的，然而，一个`void` 形的变量既不能赋值又不能引用，如果允许它出现，那么在生成三元式，生成目标代码时将会不知道如何去处理它。所以我们发现了这个问题后，就 MiniC 的语义分析中，禁用了这种表示。

第二章　中间表示

编译器的目的是将源文件“翻译”成目标机器的汇编文件。为了减小在翻译过程中所面对的源语言和目标汇编语言之间的复杂程度差距，编译器一般会将源文件变换成若干种中间表示，这些中间表示同目标汇编语言的相似程度逐渐增大。同时，在一些中间表示上，编译器能够相对容易地进行语义检查；在另一些些中间表示上，编译器还能利用相对简单、与源语言和目标汇编语言分别隔离的中间表示来进行机器无关优化。

MiniC 采用了两重中间表示：抽象语法树 (Abstract Syntax Tree) 和三元式。语义检查在前者上完成，机器无关优化在后者上完成。下面将介绍这两种中间表示的生成以及在其上所做的相关工作。

2.1　抽象语法树 AST

AST 是 MiniC 采用的第一重中间表示，它由在词法/语法分析阶段生成。由于我们在 AST 上完成了符号表的生成和类型检查，因此这两部分也在本章介绍。

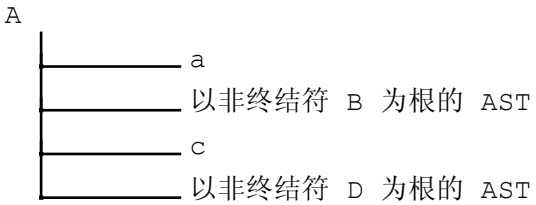
2.1.1　生成

AST 的结构是根据上下文无关文法的产生式定义的：一个终结符节点是一棵 AST；一棵以产生式左部非终结符为根的 AST 的子节点按从左到右顺序分别是其右部分法符号所产生的 AST。

比如有如下产生式：

$$A := aBcD$$

其中大写字母代表非终结符，小写字母代表终结符，那么以非终结符 A 为根的 AST 就是如下形式：



注意到 AST 的定义实际上也给出了其产生方法：只要在做 LR 分析时，随着规约的进行，将子树与根进行连接即可。

在实现时，AST 的叶节点在词法分析时生成，内部节点在语法分析时在不同的产生式的语法规则指导下生成，并在规约时连接到其父节点上。

下面给出一个 MiniC 的 AST 节点所包含的内容：

```
typedef struct AST_NODE{
    int nodeType;//节点类型，即节点代表了哪个文法符号
    int nodeLevel;//节点深度
    union node_content content; //节点包含的内容，只有叶子节点才不为空，在词法分析时添加
    AST_NODE * father;//父节点
    AST_NODE * leftChild;//最左子节点
    AST_NODE * rightSibling;//右兄弟节点
```

```
8
9       struct symtbl_hdr* symtbl;//文法符号所在范围的符号表
10      AST_NODE* double_list;
11  }AST_NODE;
```

⚓ 有关节点类型的详细定义，请参阅：AST.h

⚓ 有关建立 *AST* 树的相应过程，请参阅：AST_operation.c, minic.y

2.1.2　生成的语法树示例

我们利用dot 生成了下面代码的 AST：

```
1  int main()
2  {
3      int i;
4      i = i + 1;
5  }
```

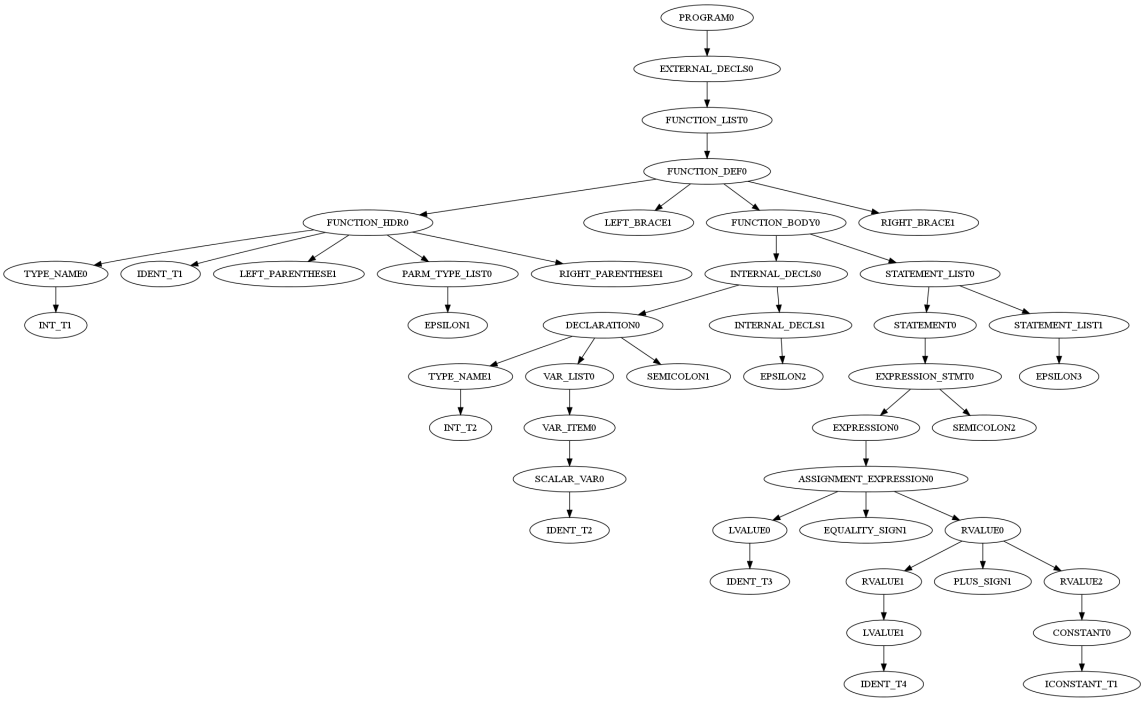


图 2.1: AST 示例

↜*MiniC* 的验证工具中提供了两种方式查看输入源文件的语法树：dot 和 *ASCII ART*，请参阅 *MiniC* 使用手册

2.1.3　AST 的通用周游模板

由于建立符号表、类型检查以及将 AST 变换成三元式都涉及到在 AST 上做深度游先周游，所以我们设计了一个通用的周游函数tree_traversal，该函数接受两个参数：AST 的根节点指针以及返回值为int，参数为 AST 节点指针的函数指针的数组。由于我们将所有的节点类型都定义成了不同的整数，因此只需要为不同的周游阶段构造不同的函数指针数组，在tree_traversal 中周游到某个节点时以节

点类型为数组下标定位到对这个节点进行操作的函数并调用，就能够用同一个深度游先周游函数完成不同的周游功能。

⚓ 有关深度优先周游模板的代码，请参阅：symtbl_operation.h

2.1.4　符号表

符号表对于一个编译程序而言是最为重要的一个部分，从它创建以后开始的每一个步骤中，“标识符”（函数名、变量名）的出现，就意味着要在符号表中找到对应的表项，提取要使用的信息。

符号表的结构　符号表的层次结构需要根据语言的特性来决定。例如在 MiniC 中，由于存在着全局变量、函数声明、语句块等特点，因此符号表需要组织成树状结构。

符号表的具体底层数据结构可以有很多种选择，常见的有哈希表、链表和线性表，在 MiniC 中，由于考虑到输入源文件的规模都较小，所以采用可扩张的线性表来实现[1]。在这种实现下，插入一个表项的均摊开销是 $O(1)$，检索一个表项的开销是 $O(n)$， n 为表项总数目。

MiniC 的符号表结构如下：

1. 每个作用域（函数、复合语句）一张符号表：

```
struct symtbl_hdr
{
    symtbl_hdr* parent_tbl; //父表
    symtbl_hdr* leftChild_tbl; //最左子表
    symtbl_hdr* rightSibling_tbl;    //右兄弟子表

    //ret_type, ret_star, para_num, func are useful only for function's symtbl
    char* func_name; //函数名，若该表属于复合语句则该项为空
    int ret_type; //函数返回值基类型
    int ret_star; //函数返回值是否为指针类型
    int para_num; //该函数的参数个数
    int func_def; //该函数是否有定义
    int item_num; //符号表表项数目
    int maxSize; //全部表项占用的内存大小，字节
    symtbl_item* item; //表项列表
}
```

2. 每张符号表都有一个表项列表：

```
struct symtbl_item
{
    int isGlobal; //是否为全局变量
    int type; //表项的基类型
    int star_num; //表项是否为指针
    int writable; //表项是否为只读类型
    char* name; //符号名
    int size; //数组大小，非数组为0
    int func_off; //
    int offset; //在内存分配中的偏移
};
```

[1] 这种线性表在空间不足时会另外申请一块两倍的空间并拷贝自己

注意到在一个函数的符号表中，函数的参数和它的局部变量是存放在一起的，这种将参数和局部变量视作同类的安排能够为以后的内存分配、寄存器分配等提供一些方便。

在 AST 上生成符号表　如前所述，AST 上包含了源文件中所有的信息，因此只需要扫描 AST 的某些子树，提取符号名称、类型、作用域等信息，就可以生成符号表。
具体的做法是：先序深度优先周游 AST，找到作用域入口（FUNCTION_DEF 和COMPOUND_STMT）并将当前作用于压入作用域栈，然后在该作用域节点下寻找类型为EXTERNAL_DECLS 和INTERNAL_DECLS 的节点，分情况处理该节点的函数和变量的声明，将名称和类型（包括函数的返回类型，参数、参数类型）加入符号表，作用域从作用域栈顶取得。在作用域出口（周游函数返回时）弹出作用域栈顶。
⚓ 有关符号表生成的相关代码，请参阅：gen_symtbl.h

在符号表上查询符号　由于符号表是树状组织，因此查询时要从给定的一张符号表开始，在表中查找，如果未找到符号，就在当前表的父表中查找，直到找到该符号或者在最高级的Global Scope 也没有找到返回空。
⚓ 有关符号表查询的代码，请参阅：symtbl_operation.h

符号表生成与语义检查　在符号表生成的过程中，实际上还要进行一项语义检查：同一作用域的多重定义问题。即在向当前符号表中添加符号时，如果该符号已经出现在了当前表中，就要报告错误，停止建立符号表。

2.1.5　符号表示例

下图展示了以下代码的符号表:

```
1  int i,j;
2  void f(int k)
3  {
4      int i,j;
5  }
6  int main()
7  {
8      int i,j;
9      {int k;}
10 }
```

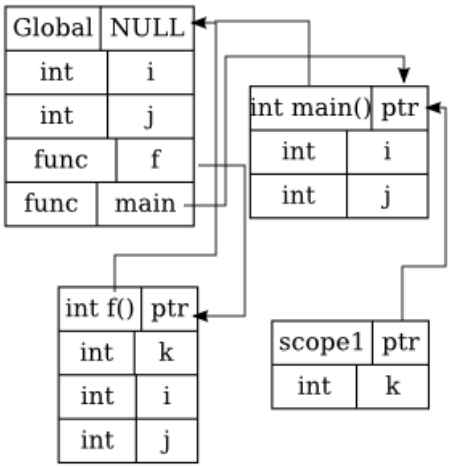


图 2.2: 符号表示例

↝*MiniC* 的验证工具中提供了查看源文件生成的符号表的方法：请参阅 *MiniC* 使用手册

2.1.6　类型检查

类型检查是语义检查的一部分，这一步骤的主要目的是排除所有符合语法但不符合运算规则、函数调用规则的语句。类型检查的对象是表达式语句，这包括：

- 检查一元、二元运算符作用的对象；报告类型无效的运算对象
- 检查传入函数的实参类型和函数的参数表对应形参的类型是否兼容；报告兼容性警告或者错误
- 检查赋值语句两边类型是否是赋值兼容的（包括检查函数返回值）；报告兼容性警告或者错误

类型检查一旦完成，表明这份源代码已经能够且必须通过后续的变换，直到生成目标代码。

类型检查的规则　下面给出了 MiniC 类型检查的所有规则以及应对方法。这些规则的错误/警告处理基本同gcc 的对应处理相同。

- 对于每个表达式，若其中有一部分类型检查返回的类型为错误类型，则直接返回错误类型。
- 对于赋值语句，若赋值两边的表达式类型不相同，则要进行必要的转换或者报错：
  - 若赋值符号左边为指针类型，但同时又是数组类型：报错;
  - 若两边一边为指针，另一边为int或者char：警告
  - 若两边均为指针或者均非指针，则三元式临时变量的类型同操作数的类型；除非除非某操作数的类型为char且另一操作数的类型为int，此时警告，并且三元式临时变量的类型同第一个操作数类型。
- 对于左值，分以下情况：
  - 若左值为一个变量，则在符号表中查找它，没找到则报错，否则正常返回一个代表该变量类型。另外，还要判断一下该变量的类型，若为void类型或者声明为函数类型同样报错。

  - 若左值为一个指针的解引用，则判断对应的被解引用的右值是否是一个指针类型，若是则将返回的变量类型置为普通类型。
  - 若左值为一个数组元素，则首先检查数组头是否为指针类型，若不是则报错。其次检查数组下标对应的元素是否是一个非指针的int型表达式或者一个char型表达式，若不是则报错。返回对应数组元素类型即可。

- 对于右值，分以下情况：
  - 对应的表达式为一个单目运算符和一个操作数，则调用函数对单目运算符表达式进行类型检查。
  - 单目运算符：
    * 正号和负号：判断其操作数是否是 int 型变量或 char 型变量，若不是则报错。
    * 对于自增运算符 ++ 和自减运算符 –，若其操作数为数组变量，则报错。
  - 若该右值是一个无关的函数调用，则在符号表中查询该函数，若该函数没有声明为一个无参函数则报错。
  - 若是一个双目运算符的表达式，则调用函数对双目运算符表达式进行类型检查：
    * 对于关系运算符（>，<，==，<=，>=，!=），检查两个操作数的类型，若不完全相同则警告。
    * 对于布尔操作符（||，&&），检查两个操作数类型，若不完全相同则警告
    * 对于减法运算，需要报错的情况有：第一个操作数非指针，第二个操作数是指针；两个操作数均为指针，但类型不同
    * 对于加法运算，只有当两个操作数均为指针时需要报错，其它情况进行必要的类型转换即可。
    * 对于乘法运算，当两个操作数任何一个为指针时报错，否则进行必要的类型转换并返回即可。
  - 若该右值是含参数的函数调用，那么先求出所有参数类型，然后检查参数个数，若调用的参数个数与函数在符号表中的函数个数不符，则报错。否则若参数类型和函数在符号表中的参数的类型不同，则警告。

在 AST 上做类型检查　类似于生成符号表，类型检查同样要对 AST 进行深度优先周游，在发现`STATMENT` 类型的节点时开始按照不同的语句类型以及语句中不同的元素类型开始进行类型检查。

2.2　三元式

三元式是 MiniC 的第二重中间表示。同 MiniC 源文件中复杂的语句和操作相比，三元式相对简单、只有少数指令并且更接近目标代码表示。
MiniC 中，一条三元式可以表示为如下形式：

- A: 一般的运算指令：`(i): op arg1, arg2, arg` 可以是`(j)` 也可以是标识符或立即数；`arg2` 若无效则为单元操作
- B: 条件跳转指令：`(i): if arg goto (j)`，`arg` 可以是`(j)` 也可以是标识符或立即数
- C: 无条件跳转指令：`(i): goto (j)`

- D: 传参指令：(i)： param arg，arg 可以是(j) 也可以是标识符或立即数；出现在调用指令前
- E: 调用指令：(i)： call (j)
- F: 作用域指令：(i)： enter, (i)： leave 用于指示作用域的变化
- G: 布尔专用指令：(i)： setrb, (i)： getrb 用于组合布尔表达式的翻译，后面会有详细叙述

2.2.1　三元式的结构

考虑到后续在三元式上的操作，三元式并不以上述的字符串的形式储存，以结构的形式存在一个可扩张的线性表中（类似于符号表的实现）。下面给出表项的结构：

```
typedef struct triple
{
    enum operator op;
    union arg arg1;
    union arg arg2;
    int result_type; //运算结果的类型
    int arg1_type; //参量的类型
    int arg2_type;
    symtbl_hdr* symtbl; //本条三元式所属范围的符号表
    struct basic_block *block; //本条三元式所属的基本块
    int arg1_uni; //“联合查询表”中第一个参数的编号
    int arg2_uni; //“联合查询表”中第二个参数的编号
    int tmp_uni; //“联合查询表”中三元式标号（临时变量）的编号
    int label; //本条三元式的标号（仅当本条三元式是跳转语句的目标时有效）
}triple;
```

有关“联合查询表”的作用，参见“机器无关优化”一节的“基础数据结构”。

2.2.2　三元式指令列表

下面的表格给出了 MiniC 的三元式中所有的op 以及对应的功能和三元式类型（每种类型的具体解释见本节起始处的说明）：

op	类型	功能
if	B	条件成立就跳转
if_not	B	条件不成立就跳转
goto	C	无条件跳转
negative	A	取相反数
not	A	取布尔非
address	A	取地址
star	A	取地址处存放的数据
positive	A	返回操作数本身
assign	A	值赋给arg1
star_assign	A	赋值给地址
add	A	加法
minus	A	减法
multiply	A	乘法
char_to_int	A	类型转换
equal	A	判断相等
less	A	判断是否小于
larger	A	判断是否大于
eqlarger	A	判断是否大于等于

op	类型	功能
eqless	A	判读是否小于等于
noteq	A	判读是否不等于
or	A	布尔或
and	A	布尔与
setrb	G	将rb 置为arg 的值
getrb	G	获取rb 的值
call	E	调用函数
param	D	传参
enterF	F	进入函数
enterS	F	进入复合语句
leaveF	F	离开函数
leaveS	F	离开复合语句
return	A	返回
adds	A	左移两位后加
minuss	A	左移两位后减
r_shift	A	右移两位
c_str	A	表示一个字符串
Imm	A	表示一个大于 511 的立即数

表 2.1: MiniC 三元式op定义

⚓ 三元式的详细结构，请参阅：gen_intermediate_code.h

2.2.3　将 AST 转换为三元式

这一工作仍然使用AST 的通用周游模板进行，只是操作的对象换成了不同类型的语句（STATEMENT）。有关一些特殊语句，例如for，while，if 的翻译，由于使用的方法同 [1] 的第 8 章对应部分相似，本文档不再赘述。下面仅介绍转化工作的最困难的部分。

布尔表达式的转换　布尔表达式的翻译涉及到短路问题。另外由于语法要求，在实现短路的同时也要实现布尔表达式的求值。由于我们使用了三元式，故需要用一个专用的变量rb 来记录布尔表达式的值。我们利用了两个中间代码上的运算：set_rb 和get_rb，用来记录布尔表达式的值。
以|| 连接的两项的布尔表达式为例，在布尔表达式开始求值之前，先将rb 置为 1，然后开始计算布尔表达式左边一项的值。之后根据左边这项的值进行判断，若值为非 0，则此项为真，短路条件成立故直接跳转到这个布尔表达式的尾部；否则，短路条件不成立，继续对布尔表达式右边一项求值。若第二项的值使得整个布尔表达式值为 1，则跳转到布尔表达式尾部；否则，将布尔表达式置为 0。在布尔表达式的结尾处，利用get_rb 将布尔表达式的值赋给一个临时变量。对于&& 连接的布尔表达式，反之即可。对于多项的布尔表达式，由于我们语法树的结构，只需要递归的对布尔表达式逐项翻译即可。

赋值表达式的翻译　赋值表达式的翻译涉及到变量赋值和地址赋值两种不同的情况。由于我们建立的语法树上区分了左值和右值，故我们需要对左值的情况进行讨论。若左值为一个变量，那么一直以变量名来代表这个左值。若左值为一个地址对应的值，如指针的解引用或者数组元素，那么应该用一个地址值来代表这个左值，而当这个左值转化为 rvalue 时，应该把其值转化为内存地址上对应的数据。

在规定了左值的类型和翻译方法后，即可对赋值语句进行翻译。对于变量赋值的情况，只需要对表达式逐项翻译即可。如果左值是对地址类型的解引用，例如数组元素或者对指针类型的解引用，我们用一个中间代码上的操作star_assign(*=) 来翻译这种情况。

⚓ 有关从 *AST* 生成三元式的代码，请参阅：gen_intermediate_code.c

2.2.4　生成基本块

基本块，是一些机器无关的全局优化的操作对象，其定义见 [1] 的第 9 章。实际实现时，基本块的基本成员仅有几个指针，分别指向块首和块尾的三元式以及该块的前驱和后继。同时，为了方便遍历某个函数的所有基本块，我们还将所有基本块连成了一条链。注意，一个基本块的前驱可以有很多个，但后继最多只有两个。

```
typedef struct basic_block {
    int begin; //起始三元式编号
    int end; //结束三元式编号
    int m; //本块在链表中的位置
    struct basic_block* prev; //链表中的上一块
    struct basic_block* next; //链表中的下一块
    struct basic_block* follow; //直接后继
    struct basic_block* jump; //跳转后继
    PreList* predecessor; //前驱链表
    struct func_block* fb; //所属函数
}basic_block;
```

⚓ 有关基本块的完整结构的定义以及基本块的生成函数，请参阅：basic_block.h, gen_basic_block.c

基本块示例　对于以下的一个函数，它的基本块如下图所示：

```
int main()
{
    int i;
    char p[10];
    for(i = 0 ; i < 10 ; i++){
        p[i] = 'z' - i;
    }
}
```

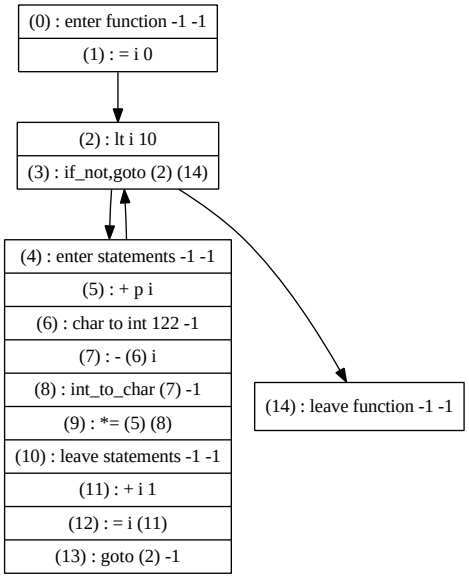


图 2.3: 基本块示例

↝*MiniC* 的验证工具中提供了生成基本块图形表示的 *dot* 文件的方法，请参阅 *MiniC* 使用手册

2.3　机器无关优化

MiniC 在三元式表示上进行了如下优化：

- 窥孔优化：常量表达式计算和面向机器的强度削减
- 基于数据流分析的可用表达式传播和指针分析

另外，活跃变量分析虽然不能算是优化的一部分，但由于也是基于数据流分析的迭代模型的算法，因此也在本节介绍。

2.3.1　窥孔优化

我们首先对三元式进行了简单的窥孔优化：

2.3.1.1　常量表达式计算

对于操作数都是常量的三元式，我们计算出该三元式的值，如果用到该三元式值的三元式又能够直接计算，那么要迭代地进行这个过程，直到结果是编译时不可计算的为止。由于目标代码生成方面的需要，这个优化不考虑对于带有常数的布尔运算。

示例　下面左图是优化前的一段三元式，右图是优化后的结果：

图 2.4: 常量表达式优化示例

简单的强度削减　虽然这部分是机器相关的（在 Unicore32 中，乘法指令需要 2 CPU cycles，而移位指令仅需要 1 CPU cycle），但在三元式部分做较为简单，即把乘以 2 的幂次的三元式都替换成左移幂次位数的三元式。在目标代码生成阶段，左移的三元式会被翻译成移位mov，然后通过目标代码的窥孔优化将其和其它代码合并。

2.3.2　数据流分析及相关优化

数据流分析技术都基于相似的算法框架和数据结构，只是在初值和迭代方程处不同。下面将介绍我们用于数据流分析的数据结构，并将重点介绍我们设计或改造的数据流分析算法。
数据流分析算法的范围是一个函数，因此下面的讨论的都是怎样在单个函数中进行数据流分析。

2.3.2.1　基础数据结构

数据流分析算法具有以下特点：

- 每一个“程序点”，即每个三元式之前和之后，都有不同的状态集，并且该状态集的成员可能是所有三元式（标号，也即临时变量）或所有变量
- 状态集成员的状态只有 2 种
- 需要多次迭代，每个基本块的输入或输出是根据其前驱的输入和输出进行交汇运算而得，交汇运算一般为集合并或集合交，因此需要能够快速进行这两种操作的集合表示。

这些特点决定了数据流分析算法需要位向量来表示状态集，因此就会引出如何将变量或或三元式标号映射到位向量的某一位的方法；与此同时，一些算法还需要快速判断两个变量名是否代表同一个变量（因为有不同的作用域），所以我们为每一个有值的三元式建立了符号表项，并使用了一个“联合查询表”统一地保存函数中出现的所有变量和标号的符号表项指针。我们还将三元式中操作数及结果所代表的变量或标号在“联合查询表”中的位置保存了在三元式的结构中。现在，某个变量或标号在位向量中的位置就是它在“联合查询表”中的索引；比较变量相同只需判断它们在“联合查询表”中的索引是否相等。
由于是在函数范围内做数据流分析，因此我们还建立了表示一个函数的数据结构，它包含了指向该函数首、尾基本块的指针以及该函数的所有数据流分析状态信息，还包括了以后的寄存器分配信息。

```
1  typedef struct func_block {
2      basic_block* start; //函数入口基本块
3      basic_block* over; //结束基本块
4      struct func_block* prev; //函数链中上一个函数
5      struct func_block* next; //函数链中下一个函数
6
7      int code_num; //函数中三元式个数
8      int bb_num; //函数中基本块个数
9      symtbl_item** uni_table; //“联合查询表”
10     int uni_item_num;
11     int uni_table_size;
12
13     map_table* mapping; //记录“联合查询表”中标号的映射信息
14     int map_table_size;
15     /*活跃变量分析信息*/
16     ...
17     /*————————————————*/
18
19     /*可用表达式分析信息*/
```

```
20      ...
21      /*————————————————*/
22
23      /*寄存器分配信息*/
24      ...
25      /*————————————————*/
26
27  }func_block;
```

⚓ 有关func_block 的完整定义以及“联合查询表”的建立方法，请参阅：basic_block.h, prepare_dataflow.c

活跃变量分析和可用表达式分析算法参见 [1] 中第 9 章。下面介绍两个我们改进和设计的关于数据流分析的优化算法。

2.3.2.2　可用表达式传播

这种算法只利用可用表达式信息，产生的效果类似于但劣于复写传播。

目的　某些表达式在同一函数中计算了多次，需要在保证正确的情况下用第一次计算的结果代替其余计算的结果。举例如下：
在下面左图的流图中，(1)(4)(7)(9) 均是相同的表达式，并且从 (1) 开始，a 和 b 都没有重新定值，因此可以用 (1) 去替换下面出现的三元式操作数 (4)(7)(9)，并且删去重复计算的后三句，得到下面右图。

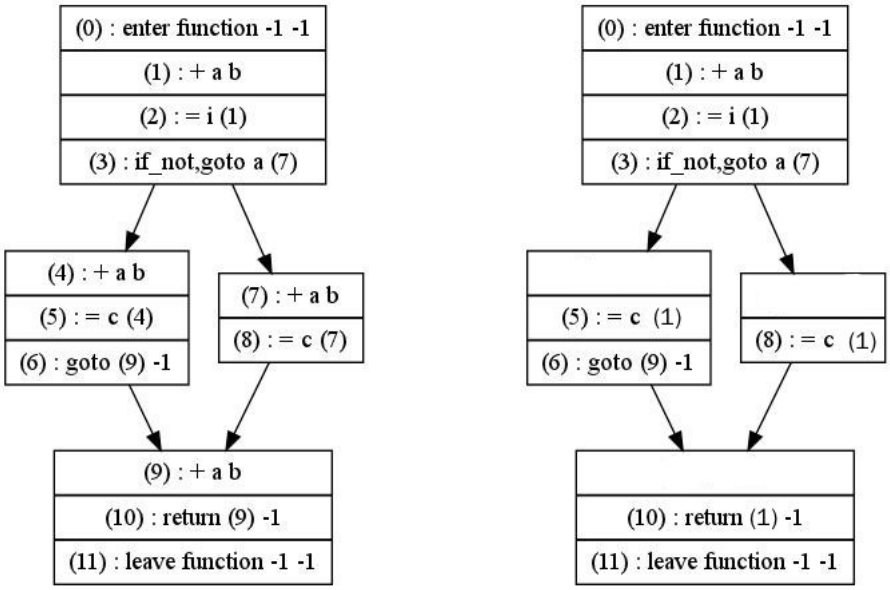


图 2.5: 可用表达式传播示例

算法

1. 进行可用表达式分析
2. 遍历三元式列表，对于一个未被修改或删除的三元式，如果它之前的程序点上有和它进行相同计算的可用表达式，那么沿着该三元式的所有非环前驱路径向上查找它所有最远的与它进行相同计算的可用表达式，如果仅找到 1 个，假设其标号为 i，那么进行 3，否则继续遍历三元式列表，遍历完毕则进行 4。

3. 从当前三元式向下查找所有用当前三元式标号当操作数的三元式，将该操作数替换成 i，删除当前三元式，记录被修改的三元式，继续遍历。

4. 如果没有三元式被删除，那么算法结束，否则进行 1

之所以进行迭代，是因为在修改三元式后，会出现新的替换机会。

缺陷　若某个三元式所计算的表达式在该点可用，但这些可用表达式没有公共的可用表达式前驱，则这种算法不能省去该三元式。例如，如果去掉上图中的 (1) 句，虽然插入赋值语句后再运用复写传播仍然能够将 (9) 省掉，但由于本算法的只进行简单的替换而不进行语句的插入，所以不能确定应当将 (9) 替换成 (4) 还是 (7)。这个缺陷的更深层次的原因请见本章的“谬误与陷阱”。

⚓ 有关可用表达式传播的代码，请参阅：`available_expr.c`

2.3.2.3　指针分析

在面向 RISC 结构的处理器的 C 编译器实现中，为了提高运算性能，大部分的变量留在寄存器中以减少访存次数。但是指针能够直接访问变量所属的内存位置。因此，需要保证在指针操作后，某个变量在寄存器中的值和其在栈帧中的值同步，为了掌握需要将哪些变量在内存中的值更新这一信息，我们需要对三元式代码进行指针分析。为此，我们设计了一个基于数据流分析的算法，来相对精确地得到在某一点每个指针可能指向哪些变量。

算法　基于数据流分析迭代框架：

- 操作集合是指针 -变量序对 `(p,v)` 的集合
- 对于每个基本块 B，$GEN(B)$ 是在该块中产生的指针 -变量序对（即有指针赋值或取地址操作），$KILL(B)$ 是在该块中注销的指针 -变量序对（如果某指针指向了别的变量，就注销掉该指针以前指向的变量）
- 迭代顺序：正向；迭代方程：$OUT(N) = GEN(N) + (IN(N) - KILL(N))$
  $IN(N) = \bigcup_{P \in predecessor(N)} OUT(P)$

性能　我们将结合一个例子来比较以下三种算法：

- 假设某个指针可能指向所有变量，即遇到指针访问就同步所有的变量
- 扫描一遍代码，记录某个指针可能指向哪些变量，只同步可能指向的变量
- 基于数据流分析的指针分析

```
1  int f()
2  {
3      int a,b,c,*p,*q;
4      ...
5      if(some_condition)
6          p = &a;
7      else
8          p = &b;
```

```
9       *p = 1;
10      while(some_condition){
11          q = p;
12          *q = 1;
13          q = &c;
14          *q = 2;
15      }
16      ...
17
18  }
```

第一种算法将在第 9 行、第 12 行和第 14 行将a，b，c 全部同步；第二种算法在第 9 行同步a，b，在第 12 行和第 14 行同步a，b，c；第三种算法将在第 12 行同步a，b，在第 12 行同步a，b，c，在第 14 行同步c。
可以看出，基于数据流分析的指针分析具有更好的精确性，因此能够节省更多的内存操作的次数。经过指针分析后将得出：

- 每一程序点上每个指针变量可能指向的变量
- 每一程序点上每个变量是否被指针指到

这两部分数据将在后面章节中，寄存器保护与同步时被使用。⚓ 有关指针分析的代码，请参阅：pointer_analysis.c

2.3.2.4　一些对数据流分析的改进设想

指针活跃性分析　在进行函数调用时，需要决定保护哪些寄存器，这时如果仅利用指针指向的信息，可能会将调用之后不活跃的指针所指向的变量一并保护，产生了一些冗余，如果能够再进行一次指针活跃性分析，并用分析结果和指向信息的交集来指到寄存器保护，将会得到更好的效果。

全局活跃变量分析　当前 MiniC 的活跃变量分析算法只能得到函数内的活跃变量信息。改进的可能同样来自函数调用：如果能够得到被调用的函数将占用哪些寄存器，就能够在调用前只保护这些寄存器，从而节省一些访存语句。这有赖于全局的（跨函数的）活跃变量分析。

2.4　谬误与陷阱

中间表示设计的小缺陷将会引起许多不便

AST 的修剪　MiniC 建立的 AST 没有经过任何修剪，从图2.1的示例中可以看出，一些无用的语法符号（例如括号、逗号）仍然留在 AST 上，并且在声明和语句等子树上存在着大量的左递归子树。
在设计之初，我们忽略了这些多余的节点和复杂的左递归结构可能带来的问题。但当使用这棵 AST 生成了符号表并且进行了类型检查后，我们发现如果事先对 AST 进行一些简单的预处理，消除掉冗余的部分，替换掉复杂的结构，就能够避免使用 AST 时复杂的条件分支，也能够避免因为复杂的结构而造成的代码逻辑上的错误。

三元式，四元式　在设计之初，我们认为三元式和四元式在表达能力上是等价的，并且发现三元式有良好的结构使得在其上构建 DAG 更方便，于是 MiniC 采用了三元式作为第二重中间表示。
然而，三元式的特点也是它的缺点：因为标号同时也代表临时变量，因此在三元式表示中，临时变量只能赋值一次。这个缺陷影响了后面的两部分工作：

- 在翻译布尔表达式时，必须允许某个标志变量被赋值两次，否则布尔表达式的赋值（如`a=b&&c`）将无法实现。因此我们向三元式中添加了`setrb` 和`getrb` 两条语句。虽然在目标代码生成阶段没有直接翻译这两条语句，而是转换成了效率更高，更易读的汇编代码，这种表示仍然给我们带来了不少麻烦。
- 在进行基于三元式的优化时，由于发现了可用表达式但无法在多处插入给同一个临时变量赋值的三元式（见 [1] 第 216 页），我们只好放弃性能更为强大的可用表达式 -复写传播来消除冗余中间代码，改为使用可用表达式传播的算法。

第三章　目标代码

在对源文件进行了足够的检查、变换并且提取了足够的信息后，应当具备了以下条件：

- 确保源文件语法、语义正确
- 有一个同目标汇编码接近的中间表示
- 有一个能够查询标识符（包括源文件中的符号和中间表示中的临时变量）信息的接口

接下来就可以开始生成目标代码了，我们将在这一章介绍编译器观点下的作为目标机器的 Unicore32 体系结构、MiniC 将三元式表示翻译成 Unicore32 汇编语言的过程以及 MiniC 在目标代码上进行的机器相关优化。

3.1　编译器 -目标机器二进制接口

本节介绍编译器视角下的 Unicore32 体系结构，主要包括寄存器使用情况和内存的维护（包括栈帧和全局区）。

⚓ 有关 *Unicore32* 指令系统，请参阅:《*Unicore32* 处理器 *ISA*（子集）介绍》

3.1.1　寄存器使用情况

下表对照了 UniCore32 的寄存器使用规范和 MiniC 的寄存器使用情况：

寄存器	Unicore32 寄存器使用规范	MiniC
r0-r3	传递参数；r0 保存返回值	传递参数；r0 保存返回值；在函数内用于无寄存器变量的装入和运算
r4-r15	caller save	caller save，个数可调 *
r17-r25	callee save	callee save，个数可调 *
r26	静态基址	装入全局变量地址
r27	栈帧基址	栈帧基址
r28	调用者 SP	传参时用于装入和运算（由于此时 r0-r3 不能用于运算）；生成立即数
r29	栈基址	栈基址
r30	返回地址	返回地址
r31	PC	PC

*`register_stat.h` 用于调整可用的 *caller save* 和 *callee save* 寄存器的个数

注意到 r28 的用法和规范有差异，但这不影响 MiniC 生成的目标文件同其它 Unicore32 编译器生成的目标文件的链接。

3.1.2　内存的维护

3.1.2.1　栈帧的维护

在程序运行之初，装载器给 r29 赋初值后，维护栈帧的工作交由程序自己来进行。因此编译器需要在目标代码中添加相关代码。
栈帧维护的汇编语句将在翻译表示函数调用的三元式（`param` 和`call`）时产生。
下图左图展示了函数调用时栈帧的情况，右图展示了局部数组的保存方法：

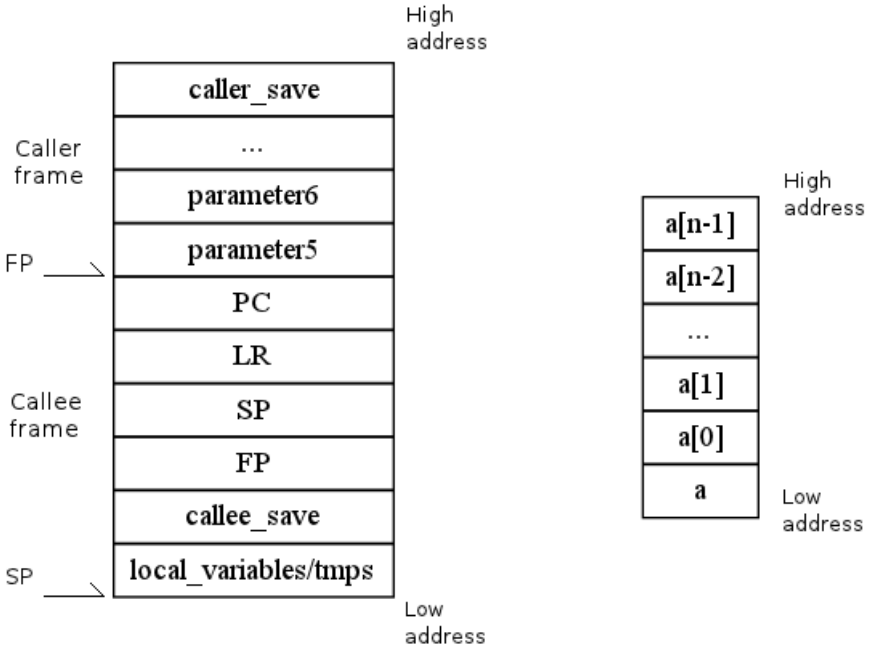


图 3.1: 栈帧与局部数组示意图

FP 以上（高地址方向）是调用者的部分栈帧（略去部分和被调用者栈帧相似），其中保存了 call save 寄存器以及传给被调用者的从第 5 个到最后一个参数（如果被调用者参数小于等于 4，那么省去这一段）。FP 到 SP 的一段是被调用者的栈帧，其中保存了调用时的 PC，返回地址，调用者的栈顶地址和栈帧基址以及 callee save 寄存器的值。同时，每一个局部变量和没有分配寄存器的临时变量在栈帧中均有位置。对于局部数组，数组的头指针存放在数组起始的第一个字节，即不考虑对齐的情况下一个 $4n$ 字节的int数组将占用 $4n+1$ 的空间。

3.1.2.2　全局区的维护以及大立即数的处理

我们将全局变量（或数组）、字符串常量保存在全局区中。由于全局区可能较大，使用基址和偏移量的寻址方式很可能出现偏移量过大不能使用带立即数的访存指令的问题。为此，我们将每个函数需要用到的全局变量的指针保存在该函数的代码段末尾，并分配给它们一个标号。当使用这些标号寻址时，汇编器会自动将寻址方式转换成 PC 相对寻址。
对于无法使用立即数寻址的三元式中的立即数，我们也将它保存在函数的代码段末尾，通过 PC 相对寻址将其读入。
例如下述代码：

```
1 int a,b;
2 int f()
3 {
4     a=25500;
```

```
5     b=25500;
6 }
```

我们将生成如下的汇编代码：

```
.comm b, 4, 4
.comm a, 4, 4
.text
.global main
.type main,function
main:
@Create stack frame
.L1:
ldw r3, .L3+8
ldw r1, .L3+0
stw r3, [r1+], #0
ldw r3, .L3+8
ldw r1, .L3+4
stw r3, [r1+], #0
@End of program
.L3:
.word a
.word b
.word 25500
```

3.2　寄存器分配

Unicore32 采用 RISC 结构，具有 31 个通用寄存器，所以对于大部分程序，几乎所有变量都可以保存在这些寄存器中，以减少对内存的读写。因此需要一个方法来给每个变量关联一个寄存器，该方法需要保证能用最节约的方式分配，同时变量之间不能互相污染。

MiniC 采用了启发式的图染色寄存器分配算法。该算法依赖于在三元式上进行的活跃变量分析的结果，即需要在某个程序点上每个变量的活跃信息，然后建立一个干涉图 $G(V,E)$，其中 V 是变量的集合，$(v_1,v_2)\in E$ 当且仅当存在某个程序点，使得在此处 v_1 和 v_2 同时活跃。给变量分配寄存器就相当于给这张干涉图着色。

举例如下，对于左图的三元式序列（花括号中是该句处的活跃变量），干涉图为右图:

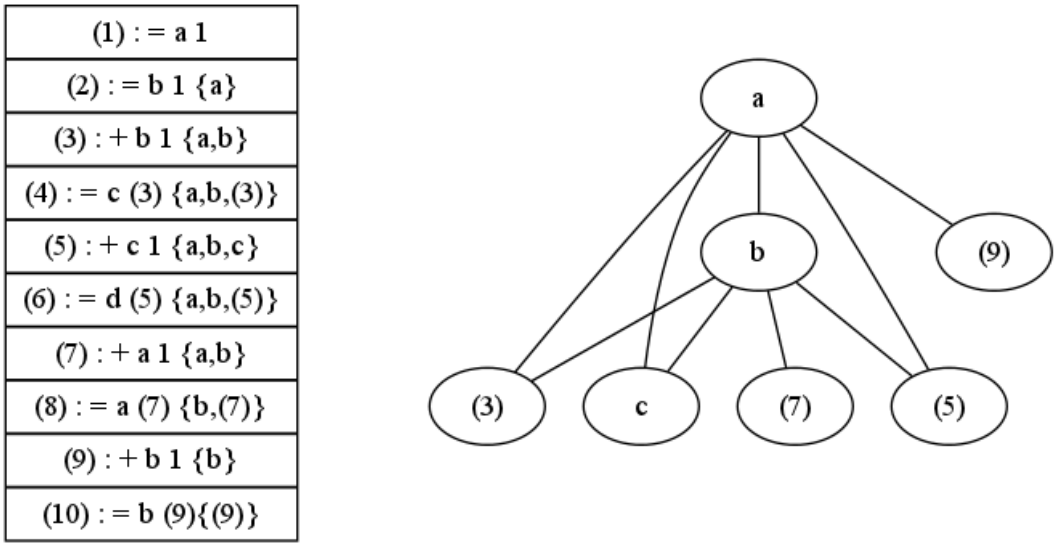


图 3.2: 寄存器分配示例

下表给出了不同的可分配寄存器数目的寄存器分配的结果：

变量	2 个可用寄存器	≥3 个可用寄存器
a	保存在内存	r6
b	r5	r5
c	r4	r4
(3)	r4	r4
(5)	r4	r4
(7)	r4	r4
(9)	r4	r4

表 3.1: 寄存器分配示例

注意到在只有 2 个可用寄存器时，出现了寄存器不足的情况。对于这种情况 MiniC 会选择将某些变量保存在内存（即在干涉图中删除该顶点）。这些保存在内存的变量每次引用都需要读取内存；每次赋值都需要写入内存。在 [1] 中介绍的算法采用的选择策略是总选择度数最大的点删除，但是在一些对数组频繁操作的程序实例（如排序算法）中，采用这种策略将会使得数组基址被保存在内存中，使得每次读取数组元素之前要先读取基址，大大影响了性能。我们认为可以采用一种更为折衷的办法：综合考虑变量的使用频率和它的干涉边数，给出一个权衡后的解。但由于时间关系，这一想法没有实现。

⚓ 有关寄存器分配和活跃变量分析的代码，请参阅：`register_allocation.c`, `live_var_anal.c`

3.3　将三元式转换为目标代码

3.3.1　控制流语句的翻译

3.3.2　寄存器保护与同步

3.3.3　立即数的处理

⚓ 有关三元式转换为目标汇编码的代码，请参阅：`gen_target_code.c`

3.4　机器相关优化

在生成目标代码的过程中，我们已经对一些潜在的代码冗余进行了清理，但是受到中间表示和翻译方法的限制，生成出的目标代码仍有改进的余地，下面介绍我们在目标代码上，配合 Unicore32 指令系统进行的优化。

3.4.1　窥孔优化

⚓ 有关目标代码上的窥孔优化的代码，请参阅：`peephole.c`

3.4.1.1　合并访存语句

如`a[i] = 1` 这样的 MiniC 语句翻译成的三元式如下：

(1) : = i 4
(2) : add_shift a i
(3) : *= (2) 1

图 3.3: 合并访存语句示例

根据三元式生成的目标代码为（假设 r4 中保存着 a 的基址，r5 中保存着 i 的值）：

```
add r5, r4, r5<<#2
mov r3, #1
stw r3, [r5+], #0
```

由于 Unicore32 提供了“寄存器位移读取（写入）”的指令，这个操作事实上只需要:

```
mov r3, #1
stw r3, [r4+],r5<<#2
```

由于`add` 和`stw` 可能不连续，逐句生成目标代码时，发现合并的可能较为困难，因此我们扫描生成出的代码，将所有符合上面例子中的情况都进行了合并。
这项简单的优化在循环地对数组访问的情况下对性能的提升比较明显。

3.4.1.2　消除冗余的`mov`

由于三元式的一些局限，翻译得到的目标代码在读取内存操作的附近可能会有一些冗余的`mov` 语句，本优化类似于复写传播，目的是将这些`mov` 删除。
能被删除的`mov` 具有以下条件：

- 从 mov 指令的源操作数最近的一次定值开始，到 mov 指令之间没有对 mov 的目标操作数引用。
- 从 mov 指令开始，将 mov 指令的源操作数替换为目标操作数时不会产生冲突。

算法

1. 对目标代码划分基本块（以跳转指令为标志），并在基本块上做一个简单的跳转关系图，每个基本块中记录每个它接下来可能执行的基本块。

2. 扫描代码。对每个 mov 语句，其目标寄存器为 Rd_{mov}，源寄存器为 Rm_{mov}。从该条语句向上寻找最近一条语句 i，语句 i 的目标寄存器为 Rm_{mov}。（语句 i 对 Rm_{mov} 定值）

3. 判断语句 i 是否可以被删除。原则有：

   - 若 mov 语句到语句 i 之间有对 Rd_{mov} 的引用，则不可删除。
   - 从语句 i 开始，根据跳转关系图向后扫描代码（深度优先的遍历）。若进入循环块则停止，语句 i 不可删除。
   - 扫描过程中，若某调语句引用了 Rm_{mov}，则将它替换为 Rd_{mov}。若某条语句对 Rm_{mov} 定值，则这条扫描路径终止。若某条语句对 Rd_{mov} 定值，并在这之后又引用 Rm_{mov}，则扫描停止，语句 i 不可删除。

4. 若上述扫描完成后，没有出现语句 i 不可删除的情况，则删除 mov 语句，将语句 i 的目标寄存器改为 Rd_{mov}。

示例　注意到在进行消除后，使用`r4` 作为源寄存器的`add` 语句改为使用`r5`。

```
ldw r4, [r5+], #0          ldw r6, [r5+], #0
mov r6, r4                 add r5, r6, 1
add r5, r4, 1              ldw r4, [r27+], #0
ldw r4, [r27+], #0
```

3.4.2　尾递归优化

尾递归优化能将符合下面条件的函数调用转化成迭代：

- 递归调用
- 该调用语句是本函数除返回语句以外的最后一条语句
- 该函数没有返回值

例如下面的快速排序代码：

```
1  void qsort(int* data, int begin, int end)
2  {
3      int i,j,tmp;
4      if(end <= begin + 1)
5          return;
6      /* partition routine */
7      ...
8      qsort(data, begin, j - 1);
9      qsort(data, j, end);
10     return;
11 }
```

注意到第 9 句的调用符合尾递归条件，因此我们将迭代地处理本次调用：将`data, j, end` 分别放在分配给对应参数的寄存器中，然后无条件跳转到函数的开始部分，建立栈帧的语句之后。下面两图对比了生成的目标代码：

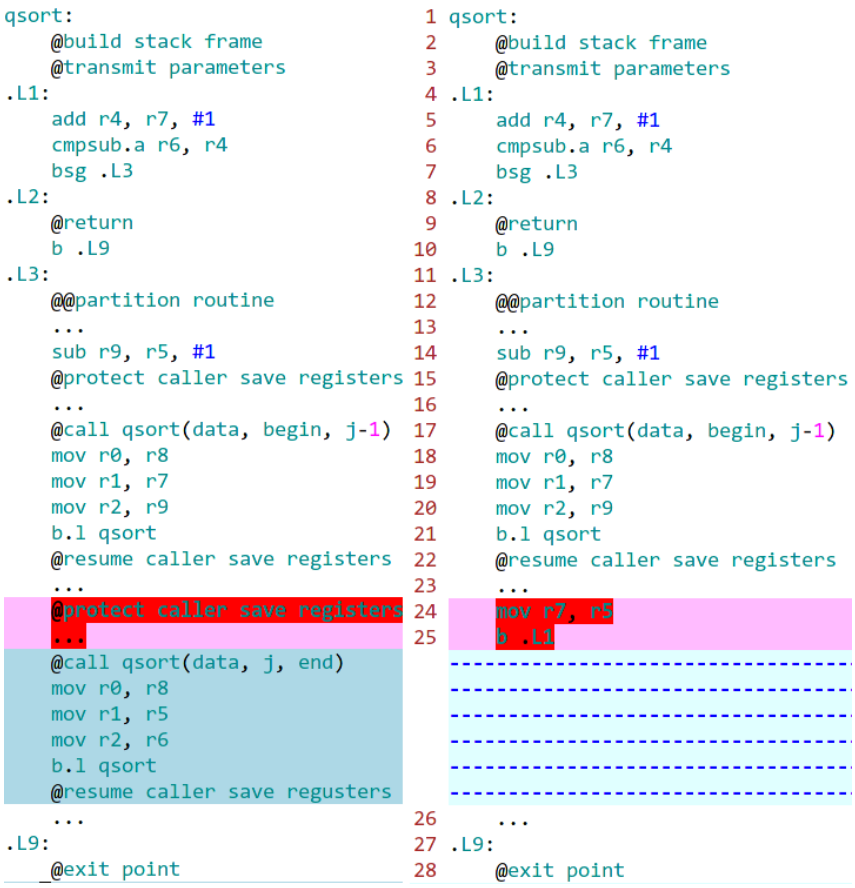


图 3.4: 尾递归示例

可以看出，在第二个`qsort` 的调用时（24 行 -30 行），进行尾递归优化的程序（右）直接将新值`j` 传入被修改的参数`begin` 所在的寄存器`r7` 中，然后无条件跳转到了函数的正文标号`.L1`。节省了保存、恢复现场以及新调用建立栈帧的指令。

⚓ 有关尾递归优化的代码，请参阅：`gen_target_code.c`

3.4.3　指令调度

根据 Unicore32 指令系统体系结构，除了载入和运算指令可能产生的数据相关无法转发，需要等待一个 cycle 以外，其它指令的数据相关均已通过转发解决。指令调度的目的是减少甚至消除因载入指令而造成的数据相关。

算法

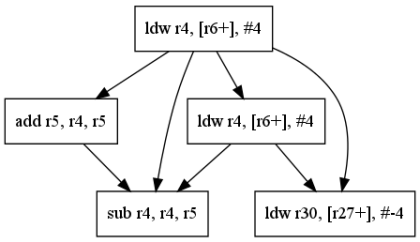
1. 在目标代码上划分基本块（以跳转指令和标签作为标记）。指令调度在基本块内完成。
2. 对每个基本块，根据指令间的数据依赖关系，建立数据依赖图。若指令 i 的源操作数依赖于指令 j 的执行结果，则建立 j 到 i 的一条边。
3. 根据数据依赖图，进行变种的拓扑排序：扫描块中所有指令。遇到`ldw` 指令，则开始执行所有`ldw` 指令依赖的指令（即在图上可达这条`ldw` 指令的指令）；之后，从所有本块中未执行过的指令中取出一条入度为 0 的指令（不依赖于任何指令），排在这条`ldw` 指令后面。特别地，这条指令的源操作数应该尽量和`ldw` 指令的目标操作数不相等。

4. 扫描一遍ldw 指令之后，所有可能调开的ldw 指令都已经被调开。再对未执行的指令进行拓扑排序即可。

示例　下面是一段汇编指令：

```
ldw r4, [r6+], #4
add r5, r4, r5
ldw r4, [r6+], #4
sub r4, r4, r5
ldw r30, [r27+], #-4
```

它的依赖关系图如下：



调度后的指令序列：

```
ldw r4, [r6+], #4
add r5, r4, r5
ldw r4, [r6+], #4
ldw r30, [r27+], #-4
sub r4, r4, r5
```

⚓ 有关指令调度的代码，请参阅：instruction_dispatch.c

3.5　谬误与陷阱

编译优化与结果正确性

教科书上对编译优化的要求是：以正确为前提，尽可能地提升目标代码的效率。然而在实践中，存在着一些对“正确”定义不清的灰色地带，如果一味地追求绝对正确，将大大损失性能。例如对如下的一段C 代码：

```
1  void f(int* p)
2  {
3      *p = 1;
4  }
5  int main()
6  {
7      int a,b,c,*p;
8      p = &b;
9      p += 1;
10     //*p = c
11     f(p);
12 }
```

按照原则上来说，如果一个程序员对他所面对的目标机器足够了解的话，`p += 1` 的使用也许是为了指向a（或c），因此在调用函数f 的时候应当将a（或c）的值同步到内存中以保证“正确”。
然而unicore32-linux-gcc 使用-O2 编译参数的结果却是错误的：它没有做任何的访存操作来保护内存和寄存器的同步。
原因是gcc 在性能和“绝对正确”之间做出了权衡：应该由程序员而不是编译器应该保证这种极为特殊的情况的正确性。
这样做所换来的性能提升也是可观的：在不考虑指针计算的情况下，简单的数据流指针分析就能够精确地得到应该同步的变量。
为了在绝大部分情况下提高性能，MiniC 也要求这种特殊情况由程序员自行保证。（去掉上面代码中的注释即可）
当然，在“保证绝对正确”的编译选项下，情况又不同了，这时候就算牺牲性能也要保证没有任何差错。

第四章　MiniC-Unicore32 模拟器性能实验

与 MiniC 配套地，我们还设计了 Unicore32 模拟器，利用模拟器测试不同的编译参数对程序性能的影响。实验选择了两个测试程序：对 128 个倒序数的快速排序和 40×40 矩阵乘法。我们分别测试了在 21 个通用寄存器寄存器（Unicore32 标准配置）和 6 个通用寄存器下，单独应用目标代码上的窥孔优化[1]、指令调度、可用表达式传播、尾递归优化后的实验结果，同时给出了打开所有优化后的实验结果；并且，在 21 个通用寄存器的情况下，我们还利用`Unicore32-linux-gcc -O2` 参数的结果作为参照。
以下是实验数据。

21 个通用寄存器

项目	gcc(O2)	无优化	仅窥孔	仅指令调度	仅可用表达式	仅尾递归	全部优化
快速排序							
总 CPU 周期数	139698	187459	135922	183103	187078	184031	132615
总指令数	112203	143927	96768	143929	143546	142022	94865
CPI	1.239	1.298	1.397	1.268	1.299	1.291	1.39
CPU 空闲周期数	26981	43018	38640	38660	43018	41495	37236
I cache 访问次数	138568	171322	124163	171324	170941	169163	122006
I cache 未命中次数	14	19	16	19	19	19	16
D cache 访问次数	26821	38143	38143	38143	38143	37254	37254
D cache 未命中次数	130	431	412	412	412	267	267
矩阵乘法							
总 CPU 周期数	141919	255302	219093	255302	284896	255302	218680
总指令数	75734	168026	131863	168026	167626	168026	131463
CPI	1.646	1.368	1.457	1.368	1.369	1.368	1.458
CPU 空闲周期数	55722	68733	68687	68733	68727	68733	68674
I cache 访问次数	97390	191622	155459	191622	191222	191622	155059
I cache 未命中次数	15	27	24	27	27	27	24
D cache 访问次数	18423	37862	37862	37862	37862	37862	37862
D cache 未命中次数	5203	5621	5620	5621	5620	5621	5619

表 4.1: 21 个通用寄存器的情况下的实验数据

[1] 由于三元式上的窥孔优化是保证编译正确性所必须的，所以默认打开

6 个通用寄存器

项目	无优化	仅窥孔	仅指令调度	仅可用表达式	仅尾递归	全部优化
快速排序						
总 CPU 周期数	281682	230416	272226	280405	275260	223217
总指令数	191465	144690	191211	190449	188925	141642
CPI	1.467	1.587	1.42	1.468	1.453	1.57
CPU 空闲周期数	89703	85212	80501	89942	85821	81061
I cache 访问次数	69941	172085	218606	217844	216066	168783
I cache 未命中次数	11902	19	22	21	22	18
D cache 访问次数	81843	81843	81843	81589	80065	79811
D cache 未命中次数	2253	2253	2253	2253	1771	1771
矩阵乘法						
总 CPU 周期数	306553	275396	282225	304184	306553	249460
总指令数	175458	146575	175458	174658	175458	145375
CPI	1.58	1.668	1.457	1.574	1.58	1.525
CPU 空闲周期数	112552	110278	88552	110983	112552	85870
I cache 访问次数	199054	170171	199054	198254	199054	168971
I cache 未命中次数	27	24	27	27	27	24
D cache 访问次数	51292	51292	51292	50892	51292	50892
D cache 未命中次数	7581	7432	7581	7413	7581	7430

表 4.2: 6 个通用寄存器的情况下的实验数据

参考文献

[1] 孙家骝，编译原理，北京大学出版社，2008