GNU编译器内部实现

For *gcc* version 4.4.5

Richard M. Stallman and the *GCC* Developer Community

Short Contents

Table of Contents

简介

该手册介绍了GNU编译器的内部实现，包括怎样移植到新的目标机上， 以及一些如何为新语言编写前端的信息。手册基于编译器（ GCC ） 4.4.5. GNU编译器的使用方法在另一个手册中介绍，参见 Section ``Introduction'' in Using the GNU Compiler Collection (GCC)。

该手册主要是一个参考手册而不是教程。它讨论了如何帮助GCC (参见 Chapter 1 [贡献], page 1) ， GCC所支持的主机和目标机的机器特征 (参见 Chapter 2 [可移植性], page 1)，GCC与这些系统的ABI是 如何联系的 (参见 Chapter 3 [接口], page 2)，GCC前端所支持的语言特征 (参见 Chapter 5 [语言], page 45)。然后 描述了GCC源文件目录结构和构建系统，GCC前端的一些接口，以及GCC如何实现对目标系统 的支持。

更多参考资料的链接在http://gcc.gnu.org/readings.html上。

1 向GCC开发提供帮助

如果你愿意帮助预测试GCC发行版，以确保它们正常工作，则可以通过SVN (参见 http://gcc.gnu.org/svn.html) 来获得当前的开发源码。源代码和二进制 程序的快照也可以从FTP上获得（参见 http://gcc.gnu.org/snapshots.html ）。

如果你愿意从事改进GCC的工作，请阅读这些链接上的建议：

http://gcc.gnu.org/contribute.html
http://gcc.gnu.org/contributewhy.html

上面给出了如何提供有益帮助以及如何避免重复工作的信息。 在http://gcc.gnu.org/projects/上列出了被推荐的项目。

2 GCC与可移植性

GCC自身的目标是可以移植到任何 int 类型最小为32位的机器上。它是针对具有平 坦（不是分段的）字节寻址的数据地址空间的目标机器（代码地址空间可以单独分开）。 目标机器的ABI可以具有8，16，32或者64位的 int 类型。 char 可以大于 8位。

GCC通过机器描述来获得目标机器的大部分信息。机器描述给出了每条机器指令的代数方程 式表示。这是一种非常整洁的描述目标机器的方法。但是当编译器所需要的信息难以通过这 种形式来表示的时候，便需要为机器描述定义一些临时的（ad-hoc）参数。可移植性的目的 是为了减少编译器的总的工作量，对本身没有什么用处。

GCC不含有机器相关的代码，但是它确实含有依赖于机器参数的代码，例如大小端（在一个 字中，最高有效位是位于地址最高的字节中还是最低的字节中）以及是否可以自动递增寻 址。在RTL生成过程，经常需要有多种策略来针对特定类别的语法树生成代码。策略适用于 不同的参数组合。通常，并不是所有可能的情况都会被覆盖到，而只是最常见的一些或者是 已经遇到的一些。这样的话，一个新的目标机可能需要额外的策略。当这种情况发生时， 编译器便会调用 abort。幸运的是一个新的策略可以通过机器无关的方式添加， 并且只会影响到需要它们的目标机器。

3 GCC的输出接口

通常情况下GCC被配置为使用与目标系统上相同的函数调用约定。 这是通过机器描述宏来实现的（参见 Chapter 17 [目标宏], page 282 ）。

但是，结构体和联合体的值的返回在一些目标机上采用了不同的方式。于是，使用PCC编译 的返回这些类型的函数就不能够被使用GCC编译的代码调用，反之亦然。但是，这并没有造 成麻烦，因为很少有Unix库函数是返回结构体或联合体的。

GCC代码使用存放 `int` 或者 `double` 返回值的寄存器来返回1，2，4或者8个 字节长的结构体和联合体。（GCC还通常将这样类型的变量分配在寄存器中。）其它大小的 结构体和联合体在返回时，将它们存放在调用者传给的一个地址中（通常在一个寄存器中）。目标钩子（target hook）`TARGET_STRUCT_VALUE_RTX` 告诉GCC从哪里来传递该 地址。

相比之下，PCC在大多目标机上返回任何大小的结构体和联合体时，都是通过将数据复制到 一个静态存储区域，然后将那个存储地址当作指针值返回。调用者必须将数据从那个内存区 域复制到需要的地方。这比GCC使用的方法要慢，而且无法重入（reentrant）。

在一些目标机上，例如RISC机器和80386，标准的系统约定是将返回值的地址传给子程序。 在这些机器上，当使用这种方法时，GCC被配置为与标准编译器兼容。这可能会对于1，2，4 或者8字节的结构体不兼容。

GCC使用系统的标准约定来传递参数。在一些机器上，是前几个参数通过寄存器传递；在另 一些机器上，是所有的都通过栈传递。本来是可以在所有机器上都使用寄存器来传递参数的 ，而且这样还有可能显著提高性能。但是，这样就会与使用标准约定的代码完全不兼容了。 所以这种改变只有在你将GCC作为系统的唯一C编译器时才实用。当我们拥有一套完整的GNU 系统，能够用GCC来编译库时，我们可以在特定机器上实现寄存器参数传递。

在一些机器上（特别是SPARC），一些类型参数通过“隐匿引用”（invisible reference）来 传递。这意味着值是存储在内存中，将内存地址传给子程序。

如果使用 `longjmp`，则需要注意自动变量。ISO C规定了没有声明为 `volatile` 的自动变量在 `longjmp` 之后其值未定义。这也是GCC所承诺的， 因为很难正确的恢复寄存器变量的值，而且GCC的一个特点是可以在未作要求的情况下也可 以将变量放在寄存器中。

4 GCC低级运行时库

GCC在一些平台上提供了一个低级运行时库，`libgcc.a' 或者 `libgcc_s.so.1'。每当需要执行某些过于复杂而无法通过生成 内嵌代码来实现的操作时，GCC便会自动生成对该库中的例程的调用。

大多数 `libgcc` 中的例程用来处理目标处理器不能直接执行的 算术运算。这包括一些机器上的整数乘除，以及其它一些机器上的所 有浮点和定点运算。`libgcc` 还包括异常处理，以及少数其它 操作。

这些例程中有一些能够用机器独立的C来定义。其它的必须为需要它们 的处理器使用汇编语言手写。

在一些情况下，GCC还会生成对C库例程的调用，像 `memcpy` 和 `memset`。GCC可能使用的例程集在 Section ``Other Builtins'' in Using the GNU Compiler Collection (GCC) 中有介绍。

这些例程接受参数并且返回特定机器模式的值，而不是特定的C类型。 关于这方面的概念解释，参见Section 10.6 [机器模式], page 118。为了便于说明，在 这章中浮点类型 `float` 被假设为对应于 `SFmode`； `double` 对应于 `DFmode`；以及 `long double` 对应于 `TFmode` 和 `XFmode`。类似的，整数类型 `int` 和 `unsigned int` 对应于 `SImode`； `long` 和 `unsigned long` 对应于DImode；以及 `long long` 和 `unsigned long long` 对应于 `TImode`。

4.1 整数算术例程

整数算术例程用于那些对一些机器模式的算术运算不提供硬件支持的平台上。

4.1.1 算术函数

`int __ashlsi3 (int a, int b)` [Runtime Function]
`long __ashldi3 (long a, int b)` [Runtime Function]
`long long __ashlti3 (long long a, int b)` [Runtime Function]
这些函数返回 a 左移 b 位后的结果。

`int __ashrsi3 (int a, int b)` [Runtime Function]
`long __ashrdi3 (long a, int b)` [Runtime Function]
`long long __ashrti3 (long long a, int b)` [Runtime Function]
这些函数返回 a 算术右移 b 位后的结果。

`int __divsi3 (int a, int b)` [Runtime Function]
`long __divdi3 (long a, long b)` [Runtime Function]
`long long __divti3 (long long a, long long b)` [Runtime Function]
这些函数返回 a 和 b 的有符号除法的商。

`int __lshrsi3 (int a, int b)` [Runtime Function]
`long __lshrdi3 (long a, int b)` [Runtime Function]
`long long __lshrti3 (long long a, int b)` [Runtime Function]
这些函数返回将 a 逻辑右移 b 位后的结果。

`int __modsi3 (int a, int b)` [Runtime Function]
`long __moddi3 (long a, long b)` [Runtime Function]
`long long __modti3 (long long a, long long b)` [Runtime Function]
这些函数返回 a 和 b 的有符号除法的余数。

`int __mulsi3 (int a, int b)` [Runtime Function]
`long __muldi3 (long a, long b)` [Runtime Function]
`long long __multi3 (long long a, long long b)` [Runtime Function]
这些函数返回 a 和 b 的乘积。

`long __negdi2 (long a)` [Runtime Function]
`long long __negti2 (long long a)` [Runtime Function]
这些函数返回 a 的负数。

`unsigned int __udivsi3 (unsigned int a, unsigned int b)` [Runtime Function]
`unsigned long __udivdi3 (unsigned long a, unsigned long b)` [Runtime Function]
`unsigned long long __udivti3 (unsigned long long a, unsigned long long b)` [Runtime Function]
这些函数返回 a 和 b 的无符号除法的商。

`unsigned long __udivmoddi3 (unsigned long a, unsigned long b, unsigned long *c)` [Runtime Function]
`unsigned long long __udivti3 (unsigned long long a, unsigned long long b, unsigned long long *c)` [Runtime Function]
这些函数计算 a 和 b 的无符号除法的商和余数。返回值为商， 并且余数放在 c 所指向的变量中。

unsigned int __umodsi3 (unsigned int a, unsigned int b) [Runtime Function]
unsigned long __umoddi3 (unsigned long a, unsigned long b) [Runtime Function]
unsigned long long __umodti3 (unsigned long long a, unsigned long long b) [Runtime Function]

这些函数返回 a 和 b 的无符号除法的余数。

4.1.2 比较函数

以下函数实现了整数比较。这些函数实现了低层次的比较，在这之上可以构建 更高级别的比较运算符（像小于，大于，或者等于）。返回值位于0到2的范围， 使得高级别的比较运算符可以通过测试有符号比较或者无符号比较的返回值来 实现。

int __cmpdi2 (long a, long b) [Runtime Function]
int __cmpti2 (long long a, long long b) [Runtime Function]

这些函数执行一个 a 和 b 的有符号比较。如果 a 小于 b，则返回0；如果 a 大于 b，则返回2；如果 a 和 b 相等则返回1。

int __ucmpdi2 (unsigned long a, unsigned long b) [Runtime Function]
int __ucmpti2 (unsigned long long a, unsigned long long b) [Runtime Function]

这些函数执行一个a和b的无符号比较。如果 a 小于 b，则返回0； 如果 a 大于 b，则返回2；如果 a 和 b 相等则返回1。

4.1.3 可产生异常的算术函数

以下函数实现了可产生异常的算术运算。这些函数在出现有符号算术溢出时， 会调用libc函数 abort。

int __absvsi2 (int a) [Runtime Function]
long __absvdi2 (long a) [Runtime Function]

这些函数返回 a 的绝对值。

int __addvsi3 (int a, int b) [Runtime Function]
long __addvdi3 (long a, long b) [Runtime Function]

这些函数返回 a 和 b 的和；也就是 a + b。

int __mulvsi3 (int a, int b) [Runtime Function]
long __mulvdi3 (long a, long b) [Runtime Function]

这些函数返回 a 和 b 的积；也就是a * b。

int __negvsi2 (int a) [Runtime Function]
long __negvdi2 (long a) [Runtime Function]

这些函数返回 a 的负数；也就是 -a。

int __subvsi3 (int a, int b) [Runtime Function]
long __subvdi3 (long a, long b) [Runtime Function]

这些函数返回 a 和 b 的差；也就是 a - b。

4.1.4 位运算

int __clzsi2 (int a) [Runtime Function]
int __clzdi2 (long a) [Runtime Function]
int __clzti2 (long long a) [Runtime Function]

这些函数返回 a 中从最高有效位开始的前导0位的数目。 如果 a 为0，则结果未定义。

int __ctzsi2 (int a) [Runtime Function]
int __ctzdi2 (long a) [Runtime Function]
int __ctzti2 (long long a) [Runtime Function]
这些函数返回 a 中从最低有效位开始的尾部0位的数目。 如果 a 为0，则结果未定义。

int __ffsdi2 (long a) [Runtime Function]
int __ffsti2 (long long a) [Runtime Function]
这些函数返回 a 中最低有效1位的索引，或者 a 为0时返回0。最低有效位的索引为1。

int __paritysi2 (int a) [Runtime Function]
int __paritydi2 (long a) [Runtime Function]
int __parityti2 (long long a) [Runtime Function]
如果 a 中置位的数目是偶数，则这些函数返回0，否则返回1。

int __popcountsi2 (int a) [Runtime Function]
int __popcountdi2 (long a) [Runtime Function]
int __popcountti2 (long long a) [Runtime Function]
这些函数返回在 a 中置位的数目。

int32_t __bswapsi2 (int32_t a) [Runtime Function]
int64_t __bswapdi2 (int64_t a) [Runtime Function]
这些函数返回 a 的字节交换。

4.2 浮点模拟例程

软件浮点库用于不支持硬件浮点计算的机器上。也用于通过 `-msoft-float' 禁止生成浮点指令的场合。（并不是琐有的目标机都支持该开关选项。）

为了和其它编译器兼容，可以使用 DECLARE_LIBRARY_RENAMES 来 重命名浮点模拟例程（see Section 17.13 [库调用], page 338）。在这一章，使用了缺省 的名字。

目前库不支持在一些机器上用于 long double 的 XFmode。

4.2.1 算术函数

float __addsf3 (float a, float b) [Runtime Function]
double __adddf3 (double a, double b) [Runtime Function]
long double __addtf3 (long double a, long double b) [Runtime Function]
long double __addxf3 (long double a, long double b) [Runtime Function]
这些函数返回 a 和 b 的和。

float __subsf3 (float a, float b) [Runtime Function]
double __subdf3 (double a, double b) [Runtime Function]
long double __subtf3 (long double a, long double b) [Runtime Function]
long double __subxf3 (long double a, long double b) [Runtime Function]
这些函数返回 b 和 a 的差，也就是 = a − b。

float __mulsf3 (float a, float b) [Runtime Function]
double __muldf3 (double a, double b) [Runtime Function]
long double __multf3 (long double a, long double b) [Runtime Function]
long double __mulxf3 (long double a, long double b) [Runtime Function]
这些函数返回 a 和 b 的积。

float __divsf3 (float a, float b) [Runtime Function]
double __divdf3 (double a, double b) [Runtime Function]
long double __divtf3 (long double a, long double b) [Runtime Function]
long double __divxf3 (long double a, long double b) [Runtime Function]
这些函数返回 a 和 b 的商，也就是= a=b。

float __negsf2 (float a) [Runtime Function]
double __negdf2 (double a) [Runtime Function]
long double __negtf2 (long double a) [Runtime Function]
long double __negxf2 (long double a) [Runtime Function]
这些函数返回 a 的负数。它们只是简单的反转符号位，因此能够产生负0和负NaN。

4.2.2 转换函数

double __extendsfdf2 (float a) [Runtime Function]
long double __extendsftf2 (float a) [Runtime Function]
long double __extendsfxf2 (float a) [Runtime Function]
long double __extenddftf2 (double a) [Runtime Function]
long double __extenddfxf2 (double a) [Runtime Function]
这些函数将a扩展为它们返回类型的模式。

double __truncxfdf2 (long double a) [Runtime Function]
double __trunctfdf2 (long double a) [Runtime Function]
float __truncxfsf2 (long double a) [Runtime Function]
float __trunctfsf2 (long double a) [Runtime Function]
float __truncdfsf2 (double a) [Runtime Function]
这些函数将a截短为它们返回类型的模式，并向0方向舍入。

int __fixsfsi (float a) [Runtime Function]
int __fixdfsi (double a) [Runtime Function]
int __fixtfsi (long double a) [Runtime Function]
int __fixxfsi (long double a) [Runtime Function]
这些函数将a转换为有符号整数，并向0方向舍入。

long __fixsfdi (float a) [Runtime Function]
long __fixdfdi (double a) [Runtime Function]
long __fixtfdi (long double a) [Runtime Function]
long __fixxfdi (long double a) [Runtime Function]
这些函数将a转换为有符号长整数，并向0方向舍入。

long long __fixsfti (float a) [Runtime Function]
long long __fixdfti (double a) [Runtime Function]
long long __fixtfti (long double a) [Runtime Function]
long long __fixxfti (long double a) [Runtime Function]
这些函数将a转换为有符号long long整数，并向0方向舍入。

unsigned int __fixunssfsi (float a) [Runtime Function]
unsigned int __fixunsdfsi (double a) [Runtime Function]
unsigned int __fixunstfsi (long double a) [Runtime Function]

`unsigned int __fixunsxfsi (long double a)` [Runtime Function]

这些函数将a转换为无符号整数，并向0方向舍入。将负值都变为0。

`unsigned long __fixunssfdi (float a)` [Runtime Function]
`unsigned long __fixunsdfdi (double a)` [Runtime Function]
`unsigned long __fixunstfdi (long double a)` [Runtime Function]
`unsigned long __fixunsxfdi (long double a)` [Runtime Function]

这些函数将a转换为无符号长整数，并向0方向舍入。将负值都变为0。

`unsigned long long __fixunssfti (float a)` [Runtime Function]
`unsigned long long __fixunsdfti (double a)` [Runtime Function]
`unsigned long long __fixunstfti (long double a)` [Runtime Function]
`unsigned long long __fixunsxfti (long double a)` [Runtime Function]

这些函数将a转换为无符号long long整数，并向0方向舍入。将负值都变为0。

`float __floatsisf (int i)` [Runtime Function]
`double __floatsidf (int i)` [Runtime Function]
`long double __floatsitf (int i)` [Runtime Function]
`long double __floatsixf (int i)` [Runtime Function]

这些函数将有符号整数i转换为浮点数。

`float __floatdisf (long i)` [Runtime Function]
`double __floatdidf (long i)` [Runtime Function]
`long double __floatditf (long i)` [Runtime Function]
`long double __floatdixf (long i)` [Runtime Function]

这些函数将有符号长整数i转换为浮点数。

`float __floattisf (long long i)` [Runtime Function]
`double __floattidf (long long i)` [Runtime Function]
`long double __floattitf (long long i)` [Runtime Function]
`long double __floattixf (long long i)` [Runtime Function]

这些函数将有符号long long整数i转换为浮点数。

`float __floatunsisf (unsigned int i)` [Runtime Function]
`double __floatunsidf (unsigned int i)` [Runtime Function]
`long double __floatunsitf (unsigned int i)` [Runtime Function]
`long double __floatunsixf (unsigned int i)` [Runtime Function]

这些函数将无符号整数i转换为浮点数。

`float __floatundisf (unsigned long i)` [Runtime Function]
`double __floatundidf (unsigned long i)` [Runtime Function]
`long double __floatunditf (unsigned long i)` [Runtime Function]
`long double __floatundixf (unsigned long i)` [Runtime Function]

这些函数将无符号长整数i转换为浮点数。

`float __floatuntisf (unsigned long long i)` [Runtime Function]
`double __floatuntidf (unsigned long long i)` [Runtime Function]
`long double __floatuntitf (unsigned long long i)` [Runtime Function]
`long double __floatuntixf (unsigned long long i)` [Runtime Function]

这些函数将无符号long long整数i转换为浮点数。

4.2.3 比较函数

有两组基本的比较函数。

int __cmpsf2 (float a, float b) [Runtime Function]
int __cmpdf2 (double a, double b) [Runtime Function]
int __cmptf2 (long double a, long double b) [Runtime Function]
这些函数计算 = a <=> b。也就是，如果 a 小于 b，则返回-1； 如果 a 大于 b，则返回1；如果 a 与 b 相等，则返回0。 如果有任一参数为NaN，则返回1，但你不要依赖于这点；如果可以出现NaN，则使用 高级的比较函数。

int __unordsf2 (float a, float b) [Runtime Function]
int __unorddf2 (double a, double b) [Runtime Function]
int __unordtf2 (long double a, long double b) [Runtime Function]
这些函数返回一个非0值，如果任一参数为NaN，否则返回0。

还有一个直接对应于比较运算符的高级别函数的完备群。它们实现了浮点运算 的ISO C语义，将NaN考虑了进来。要仔细注意每组的返回值定义。（Under the hood），所有这些函数都被实现为：

```
if (__unordXf2 (a, b))
  return E;
return __cmpXf2 (a, b);
```

其中 E 是一个常数，被选来给出对于NaN的合适行为。因此， 返回值的意义对于每组都不同。不要依赖于这些实现；只有在下面 被说明的语义才有担保。

int __eqsf2 (float a, float b) [Runtime Function]
int __eqdf2 (double a, double b) [Runtime Function]
int __eqtf2 (long double a, long double b) [Runtime Function]
如果参数都不为NaN，并且 a 和 b 相等，则这些函数返回0。

int __nesf2 (float a, float b) [Runtime Function]
int __nedf2 (double a, double b) [Runtime Function]
int __netf2 (long double a, long double b) [Runtime Function]
如果有任一参数为NaN，或者 a 和 b 不相等，则这些函数返回非0。

int __gesf2 (float a, float b) [Runtime Function]
int __gedf2 (double a, double b) [Runtime Function]
int __getf2 (long double a, long double b) [Runtime Function]
如果参数都不为NaN，并且 a 大于或等于 b，则这些函数返回一个大于或等于0的值。

int __ltsf2 (float a, float b) [Runtime Function]
int __ltdf2 (double a, double b) [Runtime Function]
int __lttf2 (long double a, long double b) [Runtime Function]
如果参数都不为NaN，并且 a 严格小于 b，则这些函数返回一个小于0的值。

int __lesf2 (float a, float b) [Runtime Function]
int __ledf2 (double a, double b) [Runtime Function]
int __letf2 (long double a, long double b) [Runtime Function]
如果参数都不为NaN，并且 a 小于或等于 b，则这些函数返回一个小于或等于0的值。

`int __gtsf2 (float a, float b)` [Runtime Function]
`int __gtdf2 (double a, double b)` [Runtime Function]
`int __gttf2 (long double a, long double b)` [Runtime Function]
如果参数都不为NaN，并且 a 严格大于 b，则这些函数返回一个大于0的值。

4.2.4 其它浮点函数

`float __powisf2 (float a, int b)` [Runtime Function]
`double __powidf2 (double a, int b)` [Runtime Function]
`long double __powitf2 (long double a, int b)` [Runtime Function]
`long double __powixf2 (long double a, int b)` [Runtime Function]
这些函数求得 a 的 b 次幂。

`complex float __mulsc3 (float a, float b, float c, float d)` [Runtime Function]
`complex double __muldc3 (double a, double b, double c, double d)` [Runtime Function]
`complex long double __multc3 (long double a, long double b, long double c, long double d)` [Runtime Function]
`complex long double __mulxc3 (long double a, long double b, long double c, long double d)` [Runtime Function]
这些函数返回 = a + ib 和 = c + id 的乘积，遵从C99附录G的规则。

`complex float __divsc3 (float a, float b, float c, float d)` [Runtime Function]
`complex double __divdc3 (double a, double b, double c, double d)` [Runtime Function]
`complex long double __divtc3 (long double a, long double b, long double c, long double d)` [Runtime Function]
`complex long double __divxc3 (long double a, long double b, long double c, long double d)` [Runtime Function]
这些函数返回 = a + ib 和 = c + id 的商（即 = (a + ib)÷(c + id)），遵从C99附录G的规则。

4.3 十进制浮点模拟例程

软件十进制浮点库实现了IEEE 754-2008十进制浮点算术运算， 并且只在选定的目标机上起作用。

软件十进制浮点库支持DPD 编码(Densely Packed Decimal，密集十进制数) 或 BID编码 (Binary Integer Decimal，用二进制整数表示的十进制数)， 可以在配置时进行选择。

4.3.1 算术函数

`_Decimal32 __dpd_addsd3 (_Decimal32 a, _Decimal32 b)` [Runtime Function]
`_Decimal32 __bid_addsd3 (_Decimal32 a, _Decimal32 b)` [Runtime Function]
`_Decimal64 __dpd_adddd3 (_Decimal64 a, _Decimal64 b)` [Runtime Function]
`_Decimal64 __bid_adddd3 (_Decimal64 a, _Decimal64 b)` [Runtime Function]
`_Decimal128 __dpd_addtd3 (_Decimal128 a, _Decimal128 b)` [Runtime Function]
`_Decimal128 __bid_addtd3 (_Decimal128 a, _Decimal128 b)` [Runtime Function]
这些函数返回 a 和 b 的和。

`_Decimal32 __dpd_subsd3 (_Decimal32 a, _Decimal32 b)` [Runtime Function]
`_Decimal32 __bid_subsd3 (_Decimal32 a, _Decimal32 b)` [Runtime Function]
`_Decimal64 __dpd_subdd3 (_Decimal64 a, _Decimal64 b)` [Runtime Function]
`_Decimal64 __bid_subdd3 (_Decimal64 a, _Decimal64 b)` [Runtime Function]

_Decimal128 __dpd_subtd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]
_Decimal128 __bid_subtd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]

这些函数返回 a 和 b 的差，也就是 = a − b。

_Decimal32 __dpd_mulsd3 (_Decimal32 a, _Decimal32 b) [Runtime Function]
_Decimal32 __bid_mulsd3 (_Decimal32 a, _Decimal32 b) [Runtime Function]
_Decimal64 __dpd_muldd3 (_Decimal64 a, _Decimal64 b) [Runtime Function]
_Decimal64 __bid_muldd3 (_Decimal64 a, _Decimal64 b) [Runtime Function]
_Decimal128 __dpd_multd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]
_Decimal128 __bid_multd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]

这些函数返回 a 和 b 的积。

_Decimal32 __dpd_divsd3 (_Decimal32 a, _Decimal32 b) [Runtime Function]
_Decimal32 __bid_divsd3 (_Decimal32 a, _Decimal32 b) [Runtime Function]
_Decimal64 __dpd_divdd3 (_Decimal64 a, _Decimal64 b) [Runtime Function]
_Decimal64 __bid_divdd3 (_Decimal64 a, _Decimal64 b) [Runtime Function]
_Decimal128 __dpd_divtd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]
_Decimal128 __bid_divtd3 (_Decimal128 a, _Decimal128 b) [Runtime Function]

这些函数返回 a 和 b 的商，也就是 = a=b。

_Decimal32 __dpd_negsd2 (_Decimal32 a) [Runtime Function]
_Decimal32 __bid_negsd2 (_Decimal32 a) [Runtime Function]
_Decimal64 __dpd_negdd2 (_Decimal64 a) [Runtime Function]
_Decimal64 __bid_negdd2 (_Decimal64 a) [Runtime Function]
_Decimal128 __dpd_negtd2 (_Decimal128 a) [Runtime Function]
_Decimal128 __bid_negtd2 (_Decimal128 a) [Runtime Function]

这些函数返回 a 的负数。它们只是简单的反转符号位，因此能够产生负0和负NaN。

4.3.2 转换函数

_Decimal64 __dpd_extendsddd2 (_Decimal32 a) [Runtime Function]
_Decimal64 __bid_extendsddd2 (_Decimal32 a) [Runtime Function]
_Decimal128 __dpd_extendsdtd2 (_Decimal32 a) [Runtime Function]
_Decimal128 __bid_extendsdtd2 (_Decimal32 a) [Runtime Function]
_Decimal128 __dpd_extendddtd2 (_Decimal64 a) [Runtime Function]
_Decimal128 __bid_extendddtd2 (_Decimal64 a) [Runtime Function]
_Decimal32 __dpd_truncddsd2 (_Decimal64 a) [Runtime Function]
_Decimal32 __bid_truncddsd2 (_Decimal64 a) [Runtime Function]
_Decimal32 __dpd_trunctdsd2 (_Decimal128 a) [Runtime Function]
_Decimal32 __bid_trunctdsd2 (_Decimal128 a) [Runtime Function]
_Decimal64 __dpd_trunctddd2 (_Decimal128 a) [Runtime Function]
_Decimal64 __bid_trunctddd2 (_Decimal128 a) [Runtime Function]

这些函数将a的值从十进制浮点类型转换为其它类型。

_Decimal64 __dpd_extendsfdd (float a) [Runtime Function]
_Decimal64 __bid_extendsfdd (float a) [Runtime Function]
_Decimal128 __dpd_extendsftd (float a) [Runtime Function]
_Decimal128 __bid_extendsftd (float a) [Runtime Function]
_Decimal128 __dpd_extenddftd (double a) [Runtime Function]

_Decimal128 __bid_extenddftd (double a) [Runtime Function]
_Decimal128 __dpd_extendxftd (long double a) [Runtime Function]
_Decimal128 __bid_extendxftd (long double a) [Runtime Function]
_Decimal32 __dpd_truncdfsd (double a) [Runtime Function]
_Decimal32 __bid_truncdfsd (double a) [Runtime Function]
_Decimal32 __dpd_truncxfsd (long double a) [Runtime Function]
_Decimal32 __bid_truncxfsd (long double a) [Runtime Function]
_Decimal32 __dpd_trunctfsd (long double a) [Runtime Function]
_Decimal32 __bid_trunctfsd (long double a) [Runtime Function]
_Decimal64 __dpd_truncxfdd (long double a) [Runtime Function]
_Decimal64 __bid_truncxfdd (long double a) [Runtime Function]
_Decimal64 __dpd_trunctfdd (long double a) [Runtime Function]
_Decimal64 __bid_trunctfdd (long double a) [Runtime Function]

这些函数将的值从a二进制浮点类型转换为不同大小（size）的十进制浮点类型。

float __dpd_truncddsf (_Decimal64 a) [Runtime Function]
float __bid_truncddsf (_Decimal64 a) [Runtime Function]
float __dpd_trunctdsf (_Decimal128 a) [Runtime Function]
float __bid_trunctdsf (_Decimal128 a) [Runtime Function]
double __dpd_extendsddf (_Decimal32 a) [Runtime Function]
double __bid_extendsddf (_Decimal32 a) [Runtime Function]
double __dpd_trunctddf (_Decimal128 a) [Runtime Function]
double __bid_trunctddf (_Decimal128 a) [Runtime Function]
long double __dpd_extendsdxf (_Decimal32 a) [Runtime Function]
long double __bid_extendsdxf (_Decimal32 a) [Runtime Function]
long double __dpd_extendddxf (_Decimal64 a) [Runtime Function]
long double __bid_extendddxf (_Decimal64 a) [Runtime Function]
long double __dpd_trunctdxf (_Decimal128 a) [Runtime Function]
long double __bid_trunctdxf (_Decimal128 a) [Runtime Function]
long double __dpd_extendsdtf (_Decimal32 a) [Runtime Function]
long double __bid_extendsdtf (_Decimal32 a) [Runtime Function]
long double __dpd_extendddtf (_Decimal64 a) [Runtime Function]
long double __bid_extendddtf (_Decimal64 a) [Runtime Function]

这些函数将a的值从十进制浮点类型转换为不同大小的二进制浮点类型。

_Decimal32 __dpd_extendsfsd (float a) [Runtime Function]
_Decimal32 __bid_extendsfsd (float a) [Runtime Function]
_Decimal64 __dpd_extenddfdd (double a) [Runtime Function]
_Decimal64 __bid_extenddfdd (double a) [Runtime Function]
_Decimal128 __dpd_extendtftd (long double a) [Runtime Function]
_Decimal128 __bid_extendtftd (long double a) [Runtime Function]
float __dpd_truncsdsf (_Decimal32 a) [Runtime Function]
float __bid_truncsdsf (_Decimal32 a) [Runtime Function]
double __dpd_truncdddf (_Decimal64 a) [Runtime Function]
double __bid_truncdddf (_Decimal64 a) [Runtime Function]
long double __dpd_trunctdtf (_Decimal128 a) [Runtime Function]
long double __bid_trunctdtf (_Decimal128 a) [Runtime Function]

这些函数将a的值在相同大小的十进制浮点类型和二进制浮点类型之间转换。

int __dpd_fixsdsi (_Decimal32 a) [Runtime Function]
int __bid_fixsdsi (_Decimal32 a) [Runtime Function]
int __dpd_fixddsi (_Decimal64 a) [Runtime Function]
int __bid_fixddsi (_Decimal64 a) [Runtime Function]
int __dpd_fixtdsi (_Decimal128 a) [Runtime Function]
int __bid_fixtdsi (_Decimal128 a) [Runtime Function]

这些函数将 a 转换为有符号整数。

long __dpd_fixsddi (_Decimal32 a) [Runtime Function]
long __bid_fixsddi (_Decimal32 a) [Runtime Function]
long __dpd_fixdddi (_Decimal64 a) [Runtime Function]
long __bid_fixdddi (_Decimal64 a) [Runtime Function]
long __dpd_fixtddi (_Decimal128 a) [Runtime Function]
long __bid_fixtddi (_Decimal128 a) [Runtime Function]

这些函数将 a 转换为有符号长整数。

unsigned int __dpd_fixunssdsi (_Decimal32 a) [Runtime Function]
unsigned int __bid_fixunssdsi (_Decimal32 a) [Runtime Function]
unsigned int __dpd_fixunsddsi (_Decimal64 a) [Runtime Function]
unsigned int __bid_fixunsddsi (_Decimal64 a) [Runtime Function]
unsigned int __dpd_fixunstdsi (_Decimal128 a) [Runtime Function]
unsigned int __bid_fixunstdsi (_Decimal128 a) [Runtime Function]

这些函数将 a 转换为无符号整数。将负值都变为0。

unsigned long __dpd_fixunssddi (_Decimal32 a) [Runtime Function]
unsigned long __bid_fixunssddi (_Decimal32 a) [Runtime Function]
unsigned long __dpd_fixunsdddi (_Decimal64 a) [Runtime Function]
unsigned long __bid_fixunsdddi (_Decimal64 a) [Runtime Function]
unsigned long __dpd_fixunstddi (_Decimal128 a) [Runtime Function]
unsigned long __bid_fixunstddi (_Decimal128 a) [Runtime Function]

这些函数将 a 转换为无符号长整数。将负值都变为0。

_Decimal32 __dpd_floatsisd (int i) [Runtime Function]
_Decimal32 __bid_floatsisd (int i) [Runtime Function]
_Decimal64 __dpd_floatsidd (int i) [Runtime Function]
_Decimal64 __bid_floatsidd (int i) [Runtime Function]
_Decimal128 __dpd_floatsitd (int i) [Runtime Function]
_Decimal128 __bid_floatsitd (int i) [Runtime Function]

这些函数将有符号整数 i 转换为十进制浮点数。

_Decimal32 __dpd_floatdisd (long i) [Runtime Function]
_Decimal32 __bid_floatdisd (long i) [Runtime Function]
_Decimal64 __dpd_floatdidd (long i) [Runtime Function]
_Decimal64 __bid_floatdidd (long i) [Runtime Function]
_Decimal128 __dpd_floatditd (long i) [Runtime Function]
_Decimal128 __bid_floatditd (long i) [Runtime Function]

这些函数将有符号长整数 i 转换为十进制浮点数。

_Decimal32 __dpd_floatunssisd (unsigned int i) [Runtime Function]
_Decimal32 __bid_floatunssisd (unsigned int i) [Runtime Function]
_Decimal64 __dpd_floatunssidd (unsigned int i) [Runtime Function]
_Decimal64 __bid_floatunssidd (unsigned int i) [Runtime Function]
_Decimal128 __dpd_floatunssitd (unsigned int i) [Runtime Function]
_Decimal128 __bid_floatunssitd (unsigned int i) [Runtime Function]

这些函数将无符号整数 i 转换为十进制浮点数。

_Decimal32 __dpd_floatunsdisd (unsigned long i) [Runtime Function]
_Decimal32 __bid_floatunsdisd (unsigned long i) [Runtime Function]
_Decimal64 __dpd_floatunsdidd (unsigned long i) [Runtime Function]
_Decimal64 __bid_floatunsdidd (unsigned long i) [Runtime Function]
_Decimal128 __dpd_floatunsditd (unsigned long i) [Runtime Function]
_Decimal128 __bid_floatunsditd (unsigned long i) [Runtime Function]

这些函数将无符号长整数 i 转换为十进制浮点数。

4.3.3 比较函数

int __dpd_unordsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_unordsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_unorddd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_unorddd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_unordtd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_unordtd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果有任一参数为NaN，则这些函数返回非0，否则返回0。

还有一个直接对应于比较运算符的高级函数的完备群。它们按照ISO C语义实现 了浮点数比较运算，将NaN考虑了进来。要仔细注意每组函数的返回值定义。究 其原因，所有这些函数都采用以下方式实现：

```
if (__bid_unordXd2 (a, b))
 return E;
return __bid_cmpXd2 (a, b);
```

其中 E 是一个常数，被选来给出针对NaN的适当行为。因此，对于每组 函数的返回值其意义都有所不同。所以不要依赖于这些函数实现；只有在下文 明确说明的语义才是可以保证的。

int __dpd_eqsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_eqsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_eqdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_eqdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_eqtd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_eqtd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果参数都不为NaN，并且 a 和 b 相等，则这些函数返回0。

int __dpd_nesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_nesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_nedd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_nedd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_netd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_netd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果有任一参数为NaN，或者 a 和 b 不相等，则这些函数返回非0。

int __dpd_gesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_gesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_gedd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_gedd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_getd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_getd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果参数都不为NaN，并且 a 大于或等于 b，则这些函数返回一个大于或等于0的值。

int __dpd_ltsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_ltsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_ltdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_ltdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_lttd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_lttd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果参数都不为NaN，并且 a 严格小于 b，则这些函数返回一个小于0的值。

int __dpd_lesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_lesd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_ledd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_ledd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_letd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_letd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果参数都不为NaN，并且 a 小于或等于 b，则这些函数返回一个小于或等于0的值。

int __dpd_gtsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __bid_gtsd2 (_Decimal32 a, _Decimal32 b) [Runtime Function]
int __dpd_gtdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __bid_gtdd2 (_Decimal64 a, _Decimal64 b) [Runtime Function]
int __dpd_gttd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]
int __bid_gttd2 (_Decimal128 a, _Decimal128 b) [Runtime Function]

如果参数都不为NaN，并且 a 严格大于 b，则这些函数返回一个大于0的值。

4.4 定点小数模拟例程

软定点库实现了定点小数运算，并且只在选择的目标机上起作用。

为了便于理解，fract为_Fract类型的别名， accum为_Accum的别名，并且sat为_Sat的别名。

出于解说的目的，在该章节中， 定点小数类型short fract被假设为对应于机器模式QQmode；unsigned short fract对应于UQQmode； fract对应于HQmode； unsigned fract对应于UHQmode；long fract对应于SQmode； unsigned long fract对应于USQmode； long long fract对应于DQmode；以及unsigned long long fract对应于UDQmode。 类似的，定点累加类型short accum对应于HAmode；unsigned short accum对应于UHAmode； accum对应于SAmode； unsigned accum对应于USAmode；long accum对应于DAmode； unsigned long accum对应于UDAmode； long long accum对应于TAmode；以及unsigned long long accum对应于UTAmode；

4.4.1 算术函数

short fract __addqq3 (short fract a, short fract b) [Runtime Function]
fract __addhq3 (fract a, fract b) [Runtime Function]

long fract __addsq3 (long fract a, long fract b) [Runtime Function]
long long fract __adddq3 (long long fract a, long long fract b) [Runtime Function]
unsigned short fract __adduqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]
unsigned fract __adduhq3 (unsigned fract a, unsigned fract b) [Runtime Function]
unsigned long fract __addusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]
unsigned long long fract __addudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
short accum __addha3 (short accum a, short accum b) [Runtime Function]
accum __addsa3 (accum a, accum b) [Runtime Function]
long accum __addda3 (long accum a, long accum b) [Runtime Function]
long long accum __addta3 (long long accum a, long long accum b) [Runtime Function]
unsigned short accum __adduha3 (unsigned short accum a, unsigned short accum b) [Runtime Function]
unsigned accum __addusa3 (unsigned accum a, unsigned accum b) [Runtime Function]
unsigned long accum __adduda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]
unsigned long long accum __adduta3 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的和。

short fract __ssaddqq3 (short fract a, short fract b) [Runtime Function]
fract __ssaddhq3 (fract a, fract b) [Runtime Function]
long fract __ssaddsq3 (long fract a, long fract b) [Runtime Function]
long long fract __ssadddq3 (long long fract a, long long fract b) [Runtime Function]
short accum __ssaddha3 (short accum a, short accum b) [Runtime Function]
accum __ssaddsa3 (accum a, accum b) [Runtime Function]
long accum __ssaddda3 (long accum a, long accum b) [Runtime Function]
long long accum __ssaddta3 (long long accum a, long long accum b) [Runtime Function]

这些函数返回a和b的和，并带有有符号的饱和（saturation）。

unsigned short fract __usadduqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]
unsigned fract __usadduhq3 (unsigned fract a, unsigned fract b) [Runtime Function]
unsigned long fract __usaddusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]
unsigned long long fract __usaddudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
unsigned short accum __usadduha3 (unsigned short accum a, unsigned short accum b) [Runtime Function]
unsigned accum __usaddusa3 (unsigned accum a, unsigned accum b) [Runtime Function]
unsigned long accum __usadduda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]
unsigned long long accum __usadduta3 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的和，并带有无符号的饱和（saturation）。

short fract __subqq3 (short fract a, short fract b) [Runtime Function]
fract __subhq3 (fract a, fract b) [Runtime Function]
long fract __subsq3 (long fract a, long fract b) [Runtime Function]
long long fract __subdq3 (long long fract a, long long fract b) [Runtime Function]
unsigned short fract __subuqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]
unsigned fract __subuhq3 (unsigned fract a, unsigned fract b) [Runtime Function]
unsigned long fract __subusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]
unsigned long long fract __subudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
short accum __subha3 (short accum a, short accum b) [Runtime Function]
accum __subsa3 (accum a, accum b) [Runtime Function]
long accum __subda3 (long accum a, long accum b) [Runtime Function]
long long accum __subta3 (long long accum a, long long accum b) [Runtime Function]
unsigned short accum __subuha3 (unsigned short accum a, unsigned short accum b) [Runtime Function]
unsigned accum __subusa3 (unsigned accum a, unsigned accum b) [Runtime Function]
unsigned long accum __subuda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]
unsigned long long accum __subuta3 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的差，也就是a - b。

short fract __sssubqq3 (short fract a, short fract b) [Runtime Function]
fract __sssubhq3 (fract a, fract b) [Runtime Function]
long fract __sssubsq3 (long fract a, long fract b) [Runtime Function]
long long fract __sssubdq3 (long long fract a, long long fract b) [Runtime Function]
short accum __sssubha3 (short accum a, short accum b) [Runtime Function]
accum __sssubsa3 (accum a, accum b) [Runtime Function]
long accum __sssubda3 (long accum a, long accum b) [Runtime Function]
long long accum __sssubta3 (long long accum a, long long accum b) [Runtime Function]

这些函数返回a和b的差，也就是a - b，并带有有符号的饱和（saturation）。

unsigned short fract __ussubuqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]
unsigned fract __ussubuhq3 (unsigned fract a, unsigned fract b) [Runtime Function]
unsigned long fract __ussubusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]
unsigned long long fract __ussubudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
unsigned short accum __ussubuha3 (unsigned short accum a, unsigned short accum b) [Runtime Function]
unsigned accum __ussubusa3 (unsigned accum a, unsigned accum b) [Runtime Function]
unsigned long accum __ussubuda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]

`unsigned long long accum __ussubuta3` (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的差，也就是a－b，并带有无符号的饱和（saturation）。

`short fract __mulqq3` (short fract a, short fract b) [Runtime Function]
`fract __mulhq3` (fract a, fract b) [Runtime Function]
`long fract __mulsq3` (long fract a, long fract b) [Runtime Function]
`long long fract __muldq3` (long long fract a, long long fract b) [Runtime Function]
`unsigned short fract __muluqq3` (unsigned short fract a, unsigned short fract b) [Runtime Function]
`unsigned fract __muluhq3` (unsigned fract a, unsigned fract b) [Runtime Function]
`unsigned long fract __mulusq3` (unsigned long fract a, unsigned long fract b) [Runtime Function]
`unsigned long long fract __muludq3` (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
`short accum __mulha3` (short accum a, short accum b) [Runtime Function]
`accum __mulsa3` (accum a, accum b) [Runtime Function]
`long accum __mulda3` (long accum a, long accum b) [Runtime Function]
`long long accum __multa3` (long long accum a, long long accum b) [Runtime Function]
`unsigned short accum __muluha3` (unsigned short accum a, unsigned short accum b) [Runtime Function]
`unsigned accum __mulusa3` (unsigned accum a, unsigned accum b) [Runtime Function]
`unsigned long accum __muluda3` (unsigned long accum a, unsigned long accum b) [Runtime Function]
`unsigned long long accum __muluta3` (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的乘积。

`short fract __ssmulqq3` (short fract a, short fract b) [Runtime Function]
`fract __ssmulhq3` (fract a, fract b) [Runtime Function]
`long fract __ssmulsq3` (long fract a, long fract b) [Runtime Function]
`long long fract __ssmuldq3` (long long fract a, long long fract b) [Runtime Function]
`short accum __ssmulha3` (short accum a, short accum b) [Runtime Function]
`accum __ssmulsa3` (accum a, accum b) [Runtime Function]
`long accum __ssmulda3` (long accum a, long accum b) [Runtime Function]
`long long accum __ssmulta3` (long long accum a, long long accum b) [Runtime Function]

这些函数返回a和b的乘积，并带有有符号的饱和（saturation）。

`unsigned short fract __usmuluqq3` (unsigned short fract a, unsigned short fract b) [Runtime Function]
`unsigned fract __usmuluhq3` (unsigned fract a, unsigned fract b) [Runtime Function]
`unsigned long fract __usmulusq3` (unsigned long fract a, unsigned long fract b) [Runtime Function]
`unsigned long long fract __usmuludq3` (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
`unsigned short accum __usmuluha3` (unsigned short accum a, unsigned short accum b) [Runtime Function]
`unsigned accum __usmulusa3` (unsigned accum a, unsigned accum b) [Runtime Function]

unsigned long accum __usmuluda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]

unsigned long long accum __usmuluta3 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的乘积，并带有无符号的饱和（saturation）。

short fract __divqq3 (short fract a, short fract b) [Runtime Function]

fract __divhq3 (fract a, fract b) [Runtime Function]

long fract __divsq3 (long fract a, long fract b) [Runtime Function]

long long fract __divdq3 (long long fract a, long long fract b) [Runtime Function]

short accum __divha3 (short accum a, short accum b) [Runtime Function]

accum __divsa3 (accum a, accum b) [Runtime Function]

long accum __divda3 (long accum a, long accum b) [Runtime Function]

long long accum __divta3 (long long accum a, long long accum b) [Runtime Function]

这些函数返回a和b的有符号除法的商。

unsigned short fract __udivuqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]

unsigned fract __udivuhq3 (unsigned fract a, unsigned fract b) [Runtime Function]

unsigned long fract __udivusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]

unsigned long long fract __udivudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]

unsigned short accum __udivuha3 (unsigned short accum a, unsigned short accum b) [Runtime Function]

unsigned accum __udivusa3 (unsigned accum a, unsigned accum b) [Runtime Function]

unsigned long accum __udivuda3 (unsigned long accum a, unsigned long accum b) [Runtime Function]

unsigned long long accum __udivuta3 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的无符号除法的商。

short fract __ssdivqq3 (short fract a, short fract b) [Runtime Function]

fract __ssdivhq3 (fract a, fract b) [Runtime Function]

long fract __ssdivsq3 (long fract a, long fract b) [Runtime Function]

long long fract __ssdivdq3 (long long fract a, long long fract b) [Runtime Function]

short accum __ssdivha3 (short accum a, short accum b) [Runtime Function]

accum __ssdivsa3 (accum a, accum b) [Runtime Function]

long accum __ssdivda3 (long accum a, long accum b) [Runtime Function]

long long accum __ssdivta3 (long long accum a, long long accum b) [Runtime Function]

这些函数返回a和b的有符号除法的商，并带有有符号的饱和（saturation）。

unsigned short fract __usdivuqq3 (unsigned short fract a, unsigned short fract b) [Runtime Function]

unsigned fract __usdivuhq3 (unsigned fract a, unsigned fract b) [Runtime Function]

unsigned long fract __usdivusq3 (unsigned long fract a, unsigned long fract b) [Runtime Function]

unsigned long long fract __usdivudq3 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]

`unsigned short accum __usdivuha3` (unsigned short accum a, unsigned short accum b) [Runtime Function]
`unsigned accum __usdivusa3` (unsigned accum a, unsigned accum b) [Runtime Function]
`unsigned long accum __usdivuda3` (unsigned long accum a, unsigned long accum b) [Runtime Function]
`unsigned long long accum __usdivuta3` (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数返回a和b的无符号除法的商，并带有无符号的饱和（saturation）。

`short fract __negqq2` (short fract a) [Runtime Function]
`fract __neghq2` (fract a) [Runtime Function]
`long fract __negsq2` (long fract a) [Runtime Function]
`long long fract __negdq2` (long long fract a) [Runtime Function]
`unsigned short fract __neguqq2` (unsigned short fract a) [Runtime Function]
`unsigned fract __neguhq2` (unsigned fract a) [Runtime Function]
`unsigned long fract __negusq2` (unsigned long fract a) [Runtime Function]
`unsigned long long fract __negudq2` (unsigned long long fract a) [Runtime Function]
`short accum __negha2` (short accum a) [Runtime Function]
`accum __negsa2` (accum a) [Runtime Function]
`long accum __negda2` (long accum a) [Runtime Function]
`long long accum __negta2` (long long accum a) [Runtime Function]
`unsigned short accum __neguha2` (unsigned short accum a) [Runtime Function]
`unsigned accum __negusa2` (unsigned accum a) [Runtime Function]
`unsigned long accum __neguda2` (unsigned long accum a) [Runtime Function]
`unsigned long long accum __neguta2` (unsigned long long accum a) [Runtime Function]

这些函数返回a的负数。

`short fract __ssnegqq2` (short fract a) [Runtime Function]
`fract __ssneghq2` (fract a) [Runtime Function]
`long fract __ssnegsq2` (long fract a) [Runtime Function]
`long long fract __ssnegdq2` (long long fract a) [Runtime Function]
`short accum __ssnegha2` (short accum a) [Runtime Function]
`accum __ssnegsa2` (accum a) [Runtime Function]
`long accum __ssnegda2` (long accum a) [Runtime Function]
`long long accum __ssnegta2` (long long accum a) [Runtime Function]

这些函数返回a的负数，并带有有符号的饱和（saturation）。

`unsigned short fract __usneguqq2` (unsigned short fract a) [Runtime Function]
`unsigned fract __usneguhq2` (unsigned fract a) [Runtime Function]
`unsigned long fract __usnegusq2` (unsigned long fract a) [Runtime Function]
`unsigned long long fract __usnegudq2` (unsigned long long fract a) [Runtime Function]
`unsigned short accum __usneguha2` (unsigned short accum a) [Runtime Function]
`unsigned accum __usnegusa2` (unsigned accum a) [Runtime Function]
`unsigned long accum __usneguda2` (unsigned long accum a) [Runtime Function]
`unsigned long long accum __usneguta2` (unsigned long long accum a) [Runtime Function]

这些函数返回a的负数，并带有无符号的饱和（saturation）。

`short fract __ashlqq3` (short fract a, int b) [Runtime Function]
`fract __ashlhq3` (fract a, int b) [Runtime Function]

long fract __ashlsq3 (long fract a, int b) [Runtime Function]
long long fract __ashldq3 (long long fract a, int b) [Runtime Function]
unsigned short fract __ashluqq3 (unsigned short fract a, int b) [Runtime Function]
unsigned fract __ashluhq3 (unsigned fract a, int b) [Runtime Function]
unsigned long fract __ashlusq3 (unsigned long fract a, int b) [Runtime Function]
unsigned long long fract __ashludq3 (unsigned long long fract a, int b) [Runtime Function]
short accum __ashlha3 (short accum a, int b) [Runtime Function]
accum __ashlsa3 (accum a, int b) [Runtime Function]
long accum __ashlda3 (long accum a, int b) [Runtime Function]
long long accum __ashlta3 (long long accum a, int b) [Runtime Function]
unsigned short accum __ashluha3 (unsigned short accum a, int b) [Runtime Function]
unsigned accum __ashlusa3 (unsigned accum a, int b) [Runtime Function]
unsigned long accum __ashluda3 (unsigned long accum a, int b) [Runtime Function]
unsigned long long accum __ashluta3 (unsigned long long accum a, int b) [Runtime Function]

这些函数返回a左移b位的结果。

short fract __ashrqq3 (short fract a, int b) [Runtime Function]
fract __ashrhq3 (fract a, int b) [Runtime Function]
long fract __ashrsq3 (long fract a, int b) [Runtime Function]
long long fract __ashrdq3 (long long fract a, int b) [Runtime Function]
short accum __ashrha3 (short accum a, int b) [Runtime Function]
accum __ashrsa3 (accum a, int b) [Runtime Function]
long accum __ashrda3 (long accum a, int b) [Runtime Function]
long long accum __ashrta3 (long long accum a, int b) [Runtime Function]

这些函数返回a算术右移b位的结果。

unsigned short fract __lshruqq3 (unsigned short fract a, int b) [Runtime Function]
unsigned fract __lshruhq3 (unsigned fract a, int b) [Runtime Function]
unsigned long fract __lshrusq3 (unsigned long fract a, int b) [Runtime Function]
unsigned long long fract __lshrudq3 (unsigned long long fract a, int b) [Runtime Function]
unsigned short accum __lshruha3 (unsigned short accum a, int b) [Runtime Function]
unsigned accum __lshrusa3 (unsigned accum a, int b) [Runtime Function]
unsigned long accum __lshruda3 (unsigned long accum a, int b) [Runtime Function]
unsigned long long accum __lshruta3 (unsigned long long accum a, int b) [Runtime Function]

这些函数返回a逻辑右移b位的结果。

fract __ssashlhq3 (fract a, int b) [Runtime Function]
long fract __ssashlsq3 (long fract a, int b) [Runtime Function]
long long fract __ssashldq3 (long long fract a, int b) [Runtime Function]
short accum __ssashlha3 (short accum a, int b) [Runtime Function]
accum __ssashlsa3 (accum a, int b) [Runtime Function]
long accum __ssashlda3 (long accum a, int b) [Runtime Function]
long long accum __ssashlta3 (long long accum a, int b) [Runtime Function]

这些函数返回a左移b位的结果，并带有有符号的饱和（saturation）。

unsigned short fract __usashluqq3 (unsigned short fract a, int b) [Runtime Function]
unsigned fract __usashluhq3 (unsigned fract a, int b) [Runtime Function]

unsigned long fract __usashlusq3 (unsigned long fract a, int b) [Runtime Function]
unsigned long long fract __usashludq3 (unsigned long long fract a, int b) [Runtime Function]
unsigned short accum __usashluha3 (unsigned short accum a, int b) [Runtime Function]
unsigned accum __usashlusa3 (unsigned accum a, int b) [Runtime Function]
unsigned long accum __usashluda3 (unsigned long accum a, int b) [Runtime Function]
unsigned long long accum __usashluta3 (unsigned long long accum a, int b) [Runtime Function]

这些函数返回a左移b位的结果，并带有无符号的饱和（saturation）。

4.4.2 比较函数

下列函数实现了定点比较。这些函数实现了低层次的比较，可以在此基础上构造高层次的比较运算符（例如小于，大于或者等于）。返回值的范围为0到2，这样高层次的比较运算符可以使用有符号或者无符号比较，通过测试返回结果来实现。

int __cmpqq2 (short fract a, short fract b) [Runtime Function]
int __cmphq2 (fract a, fract b) [Runtime Function]
int __cmpsq2 (long fract a, long fract b) [Runtime Function]
int __cmpdq2 (long long fract a, long long fract b) [Runtime Function]
int __cmpuqq2 (unsigned short fract a, unsigned short fract b) [Runtime Function]
int __cmpuhq2 (unsigned fract a, unsigned fract b) [Runtime Function]
int __cmpusq2 (unsigned long fract a, unsigned long fract b) [Runtime Function]
int __cmpudq2 (unsigned long long fract a, unsigned long long fract b) [Runtime Function]
int __cmpha2 (short accum a, short accum b) [Runtime Function]
int __cmpsa2 (accum a, accum b) [Runtime Function]
int __cmpda2 (long accum a, long accum b) [Runtime Function]
int __cmpta2 (long long accum a, long long accum b) [Runtime Function]
int __cmpuha2 (unsigned short accum a, unsigned short accum b) [Runtime Function]
int __cmpusa2 (unsigned accum a, unsigned accum b) [Runtime Function]
int __cmpuda2 (unsigned long accum a, unsigned long accum b) [Runtime Function]
int __cmputa2 (unsigned long long accum a, unsigned long long accum b) [Runtime Function]

这些函数执行一个a和b的有符号或者无符号比较（取决于所选择的机器模式）。 如果a小于b，则返回0；如果a大于b，则返回2； 如果a和b相等，则返回1。

4.4.3 转换函数

fract __fractqqhq2 (short fract a) [Runtime Function]
long fract __fractqqsq2 (short fract a) [Runtime Function]
long long fract __fractqqdq2 (short fract a) [Runtime Function]
short accum __fractqqha (short fract a) [Runtime Function]
accum __fractqqsa (short fract a) [Runtime Function]
long accum __fractqqda (short fract a) [Runtime Function]
long long accum __fractqqta (short fract a) [Runtime Function]
unsigned short fract __fractqquqq (short fract a) [Runtime Function]
unsigned fract __fractqquhq (short fract a) [Runtime Function]
unsigned long fract __fractqqusq (short fract a) [Runtime Function]

unsigned long long fract __fractqqudq (short fract a) [Runtime Function]
unsigned short accum __fractqquha (short fract a) [Runtime Function]
unsigned accum __fractqqusa (short fract a) [Runtime Function]
unsigned long accum __fractqquda (short fract a) [Runtime Function]
unsigned long long accum __fractqquta (short fract a) [Runtime Function]
signed char __fractqqqi (short fract a) [Runtime Function]
short __fractqqhi (short fract a) [Runtime Function]
int __fractqqsi (short fract a) [Runtime Function]
long __fractqqdi (short fract a) [Runtime Function]
long long __fractqqti (short fract a) [Runtime Function]
float __fractqqsf (short fract a) [Runtime Function]
double __fractqqdf (short fract a) [Runtime Function]
short fract __fracthqqq2 (fract a) [Runtime Function]
long fract __fracthqsq2 (fract a) [Runtime Function]
long long fract __fracthqdq2 (fract a) [Runtime Function]
short accum __fracthqha (fract a) [Runtime Function]
accum __fracthqsa (fract a) [Runtime Function]
long accum __fracthqda (fract a) [Runtime Function]
long long accum __fracthqta (fract a) [Runtime Function]
unsigned short fract __fracthquqq (fract a) [Runtime Function]
unsigned fract __fracthquhq (fract a) [Runtime Function]
unsigned long fract __fracthqusq (fract a) [Runtime Function]
unsigned long long fract __fracthqudq (fract a) [Runtime Function]
unsigned short accum __fracthquha (fract a) [Runtime Function]
unsigned accum __fracthqusa (fract a) [Runtime Function]
unsigned long accum __fracthquda (fract a) [Runtime Function]
unsigned long long accum __fracthquta (fract a) [Runtime Function]
signed char __fracthqqi (fract a) [Runtime Function]
short __fracthqhi (fract a) [Runtime Function]
int __fracthqsi (fract a) [Runtime Function]
long __fracthqdi (fract a) [Runtime Function]
long long __fracthqti (fract a) [Runtime Function]
float __fracthqsf (fract a) [Runtime Function]
double __fracthqdf (fract a) [Runtime Function]
short fract __fractsqqq2 (long fract a) [Runtime Function]
fract __fractsqhq2 (long fract a) [Runtime Function]
long long fract __fractsqdq2 (long fract a) [Runtime Function]
short accum __fractsqha (long fract a) [Runtime Function]
accum __fractsqsa (long fract a) [Runtime Function]
long accum __fractsqda (long fract a) [Runtime Function]
long long accum __fractsqta (long fract a) [Runtime Function]
unsigned short fract __fractsquqq (long fract a) [Runtime Function]
unsigned fract __fractsquhq (long fract a) [Runtime Function]
unsigned long fract __fractsqusq (long fract a) [Runtime Function]
unsigned long long fract __fractsqudq (long fract a) [Runtime Function]
unsigned short accum __fractsquha (long fract a) [Runtime Function]
unsigned accum __fractsqusa (long fract a) [Runtime Function]

short __fracthahi (short accum a) [Runtime Function]
int __fracthasi (short accum a) [Runtime Function]
long __fracthadi (short accum a) [Runtime Function]
long long __fracthati (short accum a) [Runtime Function]
float __fracthasf (short accum a) [Runtime Function]
double __fracthadf (short accum a) [Runtime Function]
short fract __fractsaqq (accum a) [Runtime Function]
fract __fractsahq (accum a) [Runtime Function]
long fract __fractsasq (accum a) [Runtime Function]
long long fract __fractsadq (accum a) [Runtime Function]
short accum __fractsaha2 (accum a) [Runtime Function]
long accum __fractsada2 (accum a) [Runtime Function]
long long accum __fractsata2 (accum a) [Runtime Function]
unsigned short fract __fractsauqq (accum a) [Runtime Function]
unsigned fract __fractsauhq (accum a) [Runtime Function]
unsigned long fract __fractsausq (accum a) [Runtime Function]
unsigned long long fract __fractsaudq (accum a) [Runtime Function]
unsigned short accum __fractsauha (accum a) [Runtime Function]
unsigned accum __fractsausa (accum a) [Runtime Function]
unsigned long accum __fractsauda (accum a) [Runtime Function]
unsigned long long accum __fractsauta (accum a) [Runtime Function]
signed char __fractsaqi (accum a) [Runtime Function]
short __fractsahi (accum a) [Runtime Function]
int __fractsasi (accum a) [Runtime Function]
long __fractsadi (accum a) [Runtime Function]
long long __fractsati (accum a) [Runtime Function]
float __fractsasf (accum a) [Runtime Function]
double __fractsadf (accum a) [Runtime Function]
short fract __fractdaqq (long accum a) [Runtime Function]
fract __fractdahq (long accum a) [Runtime Function]
long fract __fractdasq (long accum a) [Runtime Function]
long long fract __fractdadq (long accum a) [Runtime Function]
short accum __fractdaha2 (long accum a) [Runtime Function]
accum __fractdasa2 (long accum a) [Runtime Function]
long long accum __fractdata2 (long accum a) [Runtime Function]
unsigned short fract __fractdauqq (long accum a) [Runtime Function]
unsigned fract __fractdauhq (long accum a) [Runtime Function]
unsigned long fract __fractdausq (long accum a) [Runtime Function]
unsigned long long fract __fractdaudq (long accum a) [Runtime Function]
unsigned short accum __fractdauha (long accum a) [Runtime Function]
unsigned accum __fractdausa (long accum a) [Runtime Function]
unsigned long accum __fractdauda (long accum a) [Runtime Function]
unsigned long long accum __fractdauta (long accum a) [Runtime Function]
signed char __fractdaqi (long accum a) [Runtime Function]
short __fractdahi (long accum a) [Runtime Function]
int __fractdasi (long accum a) [Runtime Function]
long __fractdadi (long accum a) [Runtime Function]

long long __fractdati (long accum a) [Runtime Function]
float __fractdasf (long accum a) [Runtime Function]
double __fractdadf (long accum a) [Runtime Function]
short fract __fracttaqq (long long accum a) [Runtime Function]
fract __fracttahq (long long accum a) [Runtime Function]
long fract __fracttasq (long long accum a) [Runtime Function]
long long fract __fracttadq (long long accum a) [Runtime Function]
short accum __fracttaha2 (long long accum a) [Runtime Function]
accum __fracttasa2 (long long accum a) [Runtime Function]
long accum __fracttada2 (long long accum a) [Runtime Function]
unsigned short fract __fracttauqq (long long accum a) [Runtime Function]
unsigned fract __fracttauhq (long long accum a) [Runtime Function]
unsigned long fract __fracttausq (long long accum a) [Runtime Function]
unsigned long long fract __fracttaudq (long long accum a) [Runtime Function]
unsigned short accum __fracttauha (long long accum a) [Runtime Function]
unsigned accum __fracttausa (long long accum a) [Runtime Function]
unsigned long accum __fracttauda (long long accum a) [Runtime Function]
unsigned long long accum __fracttauta (long long accum a) [Runtime Function]
signed char __fracttaqi (long long accum a) [Runtime Function]
short __fracttahi (long long accum a) [Runtime Function]
int __fracttasi (long long accum a) [Runtime Function]
long __fracttadi (long long accum a) [Runtime Function]
long long __fracttati (long long accum a) [Runtime Function]
float __fracttasf (long long accum a) [Runtime Function]
double __fracttadf (long long accum a) [Runtime Function]
short fract __fractuqqqq (unsigned short fract a) [Runtime Function]
fract __fractuqqhq (unsigned short fract a) [Runtime Function]
long fract __fractuqqsq (unsigned short fract a) [Runtime Function]
long long fract __fractuqqdq (unsigned short fract a) [Runtime Function]
short accum __fractuqqha (unsigned short fract a) [Runtime Function]
accum __fractuqqsa (unsigned short fract a) [Runtime Function]
long accum __fractuqqda (unsigned short fract a) [Runtime Function]
long long accum __fractuqqta (unsigned short fract a) [Runtime Function]
unsigned fract __fractuqquhq2 (unsigned short fract a) [Runtime Function]
unsigned long fract __fractuqqusq2 (unsigned short fract a) [Runtime Function]
unsigned long long fract __fractuqqudq2 (unsigned short fract a) [Runtime Function]
unsigned short accum __fractuqquha (unsigned short fract a) [Runtime Function]
unsigned accum __fractuqqusa (unsigned short fract a) [Runtime Function]
unsigned long accum __fractuqquda (unsigned short fract a) [Runtime Function]
unsigned long long accum __fractuqquta (unsigned short fract a) [Runtime Function]
signed char __fractuqqqi (unsigned short fract a) [Runtime Function]
short __fractuqqhi (unsigned short fract a) [Runtime Function]
int __fractuqqsi (unsigned short fract a) [Runtime Function]
long __fractuqqdi (unsigned short fract a) [Runtime Function]
long long __fractuqqti (unsigned short fract a) [Runtime Function]
float __fractuqqsf (unsigned short fract a) [Runtime Function]
double __fractuqqdf (unsigned short fract a) [Runtime Function]

short fract __fractuhqqq (unsigned fract a) [Runtime Function]
fract __fractuhqhq (unsigned fract a) [Runtime Function]
long fract __fractuhqsq (unsigned fract a) [Runtime Function]
long long fract __fractuhqdq (unsigned fract a) [Runtime Function]
short accum __fractuhqha (unsigned fract a) [Runtime Function]
accum __fractuhqsa (unsigned fract a) [Runtime Function]
long accum __fractuhqda (unsigned fract a) [Runtime Function]
long long accum __fractuhqta (unsigned fract a) [Runtime Function]
unsigned short fract __fractuhquqq2 (unsigned fract a) [Runtime Function]
unsigned long fract __fractuhqusq2 (unsigned fract a) [Runtime Function]
unsigned long long fract __fractuhqudq2 (unsigned fract a) [Runtime Function]
unsigned short accum __fractuhquha (unsigned fract a) [Runtime Function]
unsigned accum __fractuhqusa (unsigned fract a) [Runtime Function]
unsigned long accum __fractuhquda (unsigned fract a) [Runtime Function]
unsigned long long accum __fractuhquta (unsigned fract a) [Runtime Function]
signed char __fractuhqqi (unsigned fract a) [Runtime Function]
short __fractuhqhi (unsigned fract a) [Runtime Function]
int __fractuhqsi (unsigned fract a) [Runtime Function]
long __fractuhqdi (unsigned fract a) [Runtime Function]
long long __fractuhqti (unsigned fract a) [Runtime Function]
float __fractuhqsf (unsigned fract a) [Runtime Function]
double __fractuhqdf (unsigned fract a) [Runtime Function]
short fract __fractusqqq (unsigned long fract a) [Runtime Function]
fract __fractusqhq (unsigned long fract a) [Runtime Function]
long fract __fractusqsq (unsigned long fract a) [Runtime Function]
long long fract __fractusqdq (unsigned long fract a) [Runtime Function]
short accum __fractusqha (unsigned long fract a) [Runtime Function]
accum __fractusqsa (unsigned long fract a) [Runtime Function]
long accum __fractusqda (unsigned long fract a) [Runtime Function]
long long accum __fractusqta (unsigned long fract a) [Runtime Function]
unsigned short fract __fractusquqq2 (unsigned long fract a) [Runtime Function]
unsigned fract __fractusquhq2 (unsigned long fract a) [Runtime Function]
unsigned long long fract __fractusqudq2 (unsigned long fract a) [Runtime Function]
unsigned short accum __fractusquha (unsigned long fract a) [Runtime Function]
unsigned accum __fractusqusa (unsigned long fract a) [Runtime Function]
unsigned long accum __fractusquda (unsigned long fract a) [Runtime Function]
unsigned long long accum __fractusquta (unsigned long fract a) [Runtime Function]
signed char __fractusqqi (unsigned long fract a) [Runtime Function]
short __fractusqhi (unsigned long fract a) [Runtime Function]
int __fractusqsi (unsigned long fract a) [Runtime Function]
long __fractusqdi (unsigned long fract a) [Runtime Function]
long long __fractusqti (unsigned long fract a) [Runtime Function]
float __fractusqsf (unsigned long fract a) [Runtime Function]
double __fractusqdf (unsigned long fract a) [Runtime Function]
short fract __fractudqqq (unsigned long long fract a) [Runtime Function]
fract __fractudqhq (unsigned long long fract a) [Runtime Function]
long fract __fractudqsq (unsigned long long fract a) [Runtime Function]

long long fract __fractudqdq (unsigned long long fract a) [Runtime Function]
short accum __fractudqha (unsigned long long fract a) [Runtime Function]
accum __fractudqsa (unsigned long long fract a) [Runtime Function]
long accum __fractudqda (unsigned long long fract a) [Runtime Function]
long long accum __fractudqta (unsigned long long fract a) [Runtime Function]
unsigned short fract __fractudquqq2 (unsigned long long fract a) [Runtime Function]
unsigned fract __fractudquhq2 (unsigned long long fract a) [Runtime Function]
unsigned long fract __fractudqusq2 (unsigned long long fract a) [Runtime Function]
unsigned short accum __fractudquha (unsigned long long fract a) [Runtime Function]
unsigned accum __fractudqusa (unsigned long long fract a) [Runtime Function]
unsigned long accum __fractudquda (unsigned long long fract a) [Runtime Function]
unsigned long long accum __fractudquta (unsigned long long fract a) [Runtime Function]
signed char __fractudqqi (unsigned long long fract a) [Runtime Function]
short __fractudqhi (unsigned long long fract a) [Runtime Function]
int __fractudqsi (unsigned long long fract a) [Runtime Function]
long __fractudqdi (unsigned long long fract a) [Runtime Function]
long long __fractudqti (unsigned long long fract a) [Runtime Function]
float __fractudqsf (unsigned long long fract a) [Runtime Function]
double __fractudqdf (unsigned long long fract a) [Runtime Function]
short fract __fractuhaqq (unsigned short accum a) [Runtime Function]
fract __fractuhahq (unsigned short accum a) [Runtime Function]
long fract __fractuhasq (unsigned short accum a) [Runtime Function]
long long fract __fractuhadq (unsigned short accum a) [Runtime Function]
short accum __fractuhaha (unsigned short accum a) [Runtime Function]
accum __fractuhasa (unsigned short accum a) [Runtime Function]
long accum __fractuhada (unsigned short accum a) [Runtime Function]
long long accum __fractuhata (unsigned short accum a) [Runtime Function]
unsigned short fract __fractuhauqq (unsigned short accum a) [Runtime Function]
unsigned fract __fractuhauhq (unsigned short accum a) [Runtime Function]
unsigned long fract __fractuhausq (unsigned short accum a) [Runtime Function]
unsigned long long fract __fractuhaudq (unsigned short accum a) [Runtime Function]
unsigned accum __fractuhausa2 (unsigned short accum a) [Runtime Function]
unsigned long accum __fractuhauda2 (unsigned short accum a) [Runtime Function]
unsigned long long accum __fractuhauta2 (unsigned short accum a) [Runtime Function]
signed char __fractuhaqi (unsigned short accum a) [Runtime Function]
short __fractuhahi (unsigned short accum a) [Runtime Function]
int __fractuhasi (unsigned short accum a) [Runtime Function]
long __fractuhadi (unsigned short accum a) [Runtime Function]
long long __fractuhati (unsigned short accum a) [Runtime Function]
float __fractuhasf (unsigned short accum a) [Runtime Function]
double __fractuhadf (unsigned short accum a) [Runtime Function]
short fract __fractusaqq (unsigned accum a) [Runtime Function]
fract __fractusahq (unsigned accum a) [Runtime Function]
long fract __fractusasq (unsigned accum a) [Runtime Function]
long long fract __fractusadq (unsigned accum a) [Runtime Function]
short accum __fractusaha (unsigned accum a) [Runtime Function]
accum __fractusasa (unsigned accum a) [Runtime Function]

long accum __fractusada (unsigned accum a) [Runtime Function]
long long accum __fractusata (unsigned accum a) [Runtime Function]
unsigned short fract __fractusauqq (unsigned accum a) [Runtime Function]
unsigned fract __fractusauhq (unsigned accum a) [Runtime Function]
unsigned long fract __fractusausq (unsigned accum a) [Runtime Function]
unsigned long long fract __fractusaudq (unsigned accum a) [Runtime Function]
unsigned short accum __fractusauha2 (unsigned accum a) [Runtime Function]
unsigned long accum __fractusauda2 (unsigned accum a) [Runtime Function]
unsigned long long accum __fractusauta2 (unsigned accum a) [Runtime Function]
signed char __fractusaqi (unsigned accum a) [Runtime Function]
short __fractusahi (unsigned accum a) [Runtime Function]
int __fractusasi (unsigned accum a) [Runtime Function]
long __fractusadi (unsigned accum a) [Runtime Function]
long long __fractusati (unsigned accum a) [Runtime Function]
float __fractusasf (unsigned accum a) [Runtime Function]
double __fractusadf (unsigned accum a) [Runtime Function]
short fract __fractudaqq (unsigned long accum a) [Runtime Function]
fract __fractudahq (unsigned long accum a) [Runtime Function]
long fract __fractudasq (unsigned long accum a) [Runtime Function]
long long fract __fractudadq (unsigned long accum a) [Runtime Function]
short accum __fractudaha (unsigned long accum a) [Runtime Function]
accum __fractudasa (unsigned long accum a) [Runtime Function]
long accum __fractudada (unsigned long accum a) [Runtime Function]
long long accum __fractudata (unsigned long accum a) [Runtime Function]
unsigned short fract __fractudauqq (unsigned long accum a) [Runtime Function]
unsigned fract __fractudauhq (unsigned long accum a) [Runtime Function]
unsigned long fract __fractudausq (unsigned long accum a) [Runtime Function]
unsigned long long fract __fractudaudq (unsigned long accum a) [Runtime Function]
unsigned short accum __fractudauha2 (unsigned long accum a) [Runtime Function]
unsigned accum __fractudausa2 (unsigned long accum a) [Runtime Function]
unsigned long long accum __fractudauta2 (unsigned long accum a) [Runtime Function]
signed char __fractudaqi (unsigned long accum a) [Runtime Function]
short __fractudahi (unsigned long accum a) [Runtime Function]
int __fractudasi (unsigned long accum a) [Runtime Function]
long __fractudadi (unsigned long accum a) [Runtime Function]
long long __fractudati (unsigned long accum a) [Runtime Function]
float __fractudasf (unsigned long accum a) [Runtime Function]
double __fractudadf (unsigned long accum a) [Runtime Function]
short fract __fractutaqq (unsigned long long accum a) [Runtime Function]
fract __fractutahq (unsigned long long accum a) [Runtime Function]
long fract __fractutasq (unsigned long long accum a) [Runtime Function]
long long fract __fractutadq (unsigned long long accum a) [Runtime Function]
short accum __fractutaha (unsigned long long accum a) [Runtime Function]
accum __fractutasa (unsigned long long accum a) [Runtime Function]
long accum __fractutada (unsigned long long accum a) [Runtime Function]
long long accum __fractutata (unsigned long long accum a) [Runtime Function]
unsigned short fract __fractutauqq (unsigned long long accum a) [Runtime Function]

unsigned fract __fractutauhq (unsigned long long accum a) [Runtime Function]
unsigned long fract __fractutausq (unsigned long long accum a) [Runtime Function]
unsigned long long fract __fractutaudq (unsigned long long accum a) [Runtime Function]
unsigned short accum __fractutauha2 (unsigned long long accum a) [Runtime Function]
unsigned accum __fractutausa2 (unsigned long long accum a) [Runtime Function]
unsigned long accum __fractutauda2 (unsigned long long accum a) [Runtime Function]
signed char __fractutaqi (unsigned long long accum a) [Runtime Function]
short __fractutahi (unsigned long long accum a) [Runtime Function]
int __fractutasi (unsigned long long accum a) [Runtime Function]
long __fractutadi (unsigned long long accum a) [Runtime Function]
long long __fractutati (unsigned long long accum a) [Runtime Function]
float __fractutasf (unsigned long long accum a) [Runtime Function]
double __fractutadf (unsigned long long accum a) [Runtime Function]
short fract __fractqiqq (signed char a) [Runtime Function]
fract __fractqihq (signed char a) [Runtime Function]
long fract __fractqisq (signed char a) [Runtime Function]
long long fract __fractqidq (signed char a) [Runtime Function]
short accum __fractqiha (signed char a) [Runtime Function]
accum __fractqisa (signed char a) [Runtime Function]
long accum __fractqida (signed char a) [Runtime Function]
long long accum __fractqita (signed char a) [Runtime Function]
unsigned short fract __fractqiuqq (signed char a) [Runtime Function]
unsigned fract __fractqiuhq (signed char a) [Runtime Function]
unsigned long fract __fractqiusq (signed char a) [Runtime Function]
unsigned long long fract __fractqiudq (signed char a) [Runtime Function]
unsigned short accum __fractqiuha (signed char a) [Runtime Function]
unsigned accum __fractqiusa (signed char a) [Runtime Function]
unsigned long accum __fractqiuda (signed char a) [Runtime Function]
unsigned long long accum __fractqiuta (signed char a) [Runtime Function]
short fract __fracthiqq (short a) [Runtime Function]
fract __fracthihq (short a) [Runtime Function]
long fract __fracthisq (short a) [Runtime Function]
long long fract __fracthidq (short a) [Runtime Function]
short accum __fracthiha (short a) [Runtime Function]
accum __fracthisa (short a) [Runtime Function]
long accum __fracthida (short a) [Runtime Function]
long long accum __fracthita (short a) [Runtime Function]
unsigned short fract __fracthiuqq (short a) [Runtime Function]
unsigned fract __fracthiuhq (short a) [Runtime Function]
unsigned long fract __fracthiusq (short a) [Runtime Function]
unsigned long long fract __fracthiudq (short a) [Runtime Function]
unsigned short accum __fracthiuha (short a) [Runtime Function]
unsigned accum __fracthiusa (short a) [Runtime Function]
unsigned long accum __fracthiuda (short a) [Runtime Function]
unsigned long long accum __fracthiuta (short a) [Runtime Function]
short fract __fractsiqq (int a) [Runtime Function]
fract __fractsihq (int a) [Runtime Function]

long fract __fractsisq (int a) [Runtime Function]
long long fract __fractsidq (int a) [Runtime Function]
short accum __fractsiha (int a) [Runtime Function]
accum __fractsisa (int a) [Runtime Function]
long accum __fractsida (int a) [Runtime Function]
long long accum __fractsita (int a) [Runtime Function]
unsigned short fract __fractsiuqq (int a) [Runtime Function]
unsigned fract __fractsiuhq (int a) [Runtime Function]
unsigned long fract __fractsiusq (int a) [Runtime Function]
unsigned long long fract __fractsiudq (int a) [Runtime Function]
unsigned short accum __fractsiuha (int a) [Runtime Function]
unsigned accum __fractsiusa (int a) [Runtime Function]
unsigned long accum __fractsiuda (int a) [Runtime Function]
unsigned long long accum __fractsiuta (int a) [Runtime Function]
short fract __fractdiqq (long a) [Runtime Function]
fract __fractdihq (long a) [Runtime Function]
long fract __fractdisq (long a) [Runtime Function]
long long fract __fractdidq (long a) [Runtime Function]
short accum __fractdiha (long a) [Runtime Function]
accum __fractdisa (long a) [Runtime Function]
long accum __fractdida (long a) [Runtime Function]
long long accum __fractdita (long a) [Runtime Function]
unsigned short fract __fractdiuqq (long a) [Runtime Function]
unsigned fract __fractdiuhq (long a) [Runtime Function]
unsigned long fract __fractdiusq (long a) [Runtime Function]
unsigned long long fract __fractdiudq (long a) [Runtime Function]
unsigned short accum __fractdiuha (long a) [Runtime Function]
unsigned accum __fractdiusa (long a) [Runtime Function]
unsigned long accum __fractdiuda (long a) [Runtime Function]
unsigned long long accum __fractdiuta (long a) [Runtime Function]
short fract __fracttiqq (long long a) [Runtime Function]
fract __fracttihq (long long a) [Runtime Function]
long fract __fracttisq (long long a) [Runtime Function]
long long fract __fracttidq (long long a) [Runtime Function]
short accum __fracttiha (long long a) [Runtime Function]
accum __fracttisa (long long a) [Runtime Function]
long accum __fracttida (long long a) [Runtime Function]
long long accum __fracttita (long long a) [Runtime Function]
unsigned short fract __fracttiuqq (long long a) [Runtime Function]
unsigned fract __fracttiuhq (long long a) [Runtime Function]
unsigned long fract __fracttiusq (long long a) [Runtime Function]
unsigned long long fract __fracttiudq (long long a) [Runtime Function]
unsigned short accum __fracttiuha (long long a) [Runtime Function]
unsigned accum __fracttiusa (long long a) [Runtime Function]
unsigned long accum __fracttiuda (long long a) [Runtime Function]
unsigned long long accum __fracttiuta (long long a) [Runtime Function]
short fract __fractsfqq (float a) [Runtime Function]

fract __fractsfhq (float a) [Runtime Function]
long fract __fractsfsq (float a) [Runtime Function]
long long fract __fractsfdq (float a) [Runtime Function]
short accum __fractsfha (float a) [Runtime Function]
accum __fractsfsa (float a) [Runtime Function]
long accum __fractsfda (float a) [Runtime Function]
long long accum __fractsfta (float a) [Runtime Function]
unsigned short fract __fractsfuqq (float a) [Runtime Function]
unsigned fract __fractsfuhq (float a) [Runtime Function]
unsigned long fract __fractsfusq (float a) [Runtime Function]
unsigned long long fract __fractsfudq (float a) [Runtime Function]
unsigned short accum __fractsfuha (float a) [Runtime Function]
unsigned accum __fractsfusa (float a) [Runtime Function]
unsigned long accum __fractsfuda (float a) [Runtime Function]
unsigned long long accum __fractsfuta (float a) [Runtime Function]
short fract __fractdfqq (double a) [Runtime Function]
fract __fractdfhq (double a) [Runtime Function]
long fract __fractdfsq (double a) [Runtime Function]
long long fract __fractdfdq (double a) [Runtime Function]
short accum __fractdfha (double a) [Runtime Function]
accum __fractdfsa (double a) [Runtime Function]
long accum __fractdfda (double a) [Runtime Function]
long long accum __fractdfta (double a) [Runtime Function]
unsigned short fract __fractdfuqq (double a) [Runtime Function]
unsigned fract __fractdfuhq (double a) [Runtime Function]
unsigned long fract __fractdfusq (double a) [Runtime Function]
unsigned long long fract __fractdfudq (double a) [Runtime Function]
unsigned short accum __fractdfuha (double a) [Runtime Function]
unsigned accum __fractdfusa (double a) [Runtime Function]
unsigned long accum __fractdfuda (double a) [Runtime Function]
unsigned long long accum __fractdfuta (double a) [Runtime Function]

These functions convert from fractional and signed non-fractionals to fractionals and signed non-fractionals, without saturation.

fract __satfractqqhq2 (short fract a) [Runtime Function]
long fract __satfractqqsq2 (short fract a) [Runtime Function]
long long fract __satfractqqdq2 (short fract a) [Runtime Function]
short accum __satfractqqha (short fract a) [Runtime Function]
accum __satfractqqsa (short fract a) [Runtime Function]
long accum __satfractqqda (short fract a) [Runtime Function]
long long accum __satfractqqta (short fract a) [Runtime Function]
unsigned short fract __satfractqquqq (short fract a) [Runtime Function]
unsigned fract __satfractqquhq (short fract a) [Runtime Function]
unsigned long fract __satfractqqusq (short fract a) [Runtime Function]
unsigned long long fract __satfractqqudq (short fract a) [Runtime Function]
unsigned short accum __satfractqquha (short fract a) [Runtime Function]
unsigned accum __satfractqqusa (short fract a) [Runtime Function]

unsigned long accum __satfractqquda (short fract a) [Runtime Function]
unsigned long long accum __satfractqquta (short fract a) [Runtime Function]
short fract __satfracthqqq2 (fract a) [Runtime Function]
long fract __satfracthqsq2 (fract a) [Runtime Function]
long long fract __satfracthqdq2 (fract a) [Runtime Function]
short accum __satfracthqha (fract a) [Runtime Function]
accum __satfracthqsa (fract a) [Runtime Function]
long accum __satfracthqda (fract a) [Runtime Function]
long long accum __satfracthqta (fract a) [Runtime Function]
unsigned short fract __satfracthquqq (fract a) [Runtime Function]
unsigned fract __satfracthquhq (fract a) [Runtime Function]
unsigned long fract __satfracthqusq (fract a) [Runtime Function]
unsigned long long fract __satfracthqudq (fract a) [Runtime Function]
unsigned short accum __satfracthquha (fract a) [Runtime Function]
unsigned accum __satfracthqusa (fract a) [Runtime Function]
unsigned long accum __satfracthquda (fract a) [Runtime Function]
unsigned long long accum __satfracthquta (fract a) [Runtime Function]
short fract __satfractsqqq2 (long fract a) [Runtime Function]
fract __satfractsqhq2 (long fract a) [Runtime Function]
long long fract __satfractsqdq2 (long fract a) [Runtime Function]
short accum __satfractsqha (long fract a) [Runtime Function]
accum __satfractsqsa (long fract a) [Runtime Function]
long accum __satfractsqda (long fract a) [Runtime Function]
long long accum __satfractsqta (long fract a) [Runtime Function]
unsigned short fract __satfractsquqq (long fract a) [Runtime Function]
unsigned fract __satfractsquhq (long fract a) [Runtime Function]
unsigned long fract __satfractsqusq (long fract a) [Runtime Function]
unsigned long long fract __satfractsqudq (long fract a) [Runtime Function]
unsigned short accum __satfractsquha (long fract a) [Runtime Function]
unsigned accum __satfractsqusa (long fract a) [Runtime Function]
unsigned long accum __satfractsquda (long fract a) [Runtime Function]
unsigned long long accum __satfractsquta (long fract a) [Runtime Function]
short fract __satfractdqqq2 (long long fract a) [Runtime Function]
fract __satfractdqhq2 (long long fract a) [Runtime Function]
long fract __satfractdqsq2 (long long fract a) [Runtime Function]
short accum __satfractdqha (long long fract a) [Runtime Function]
accum __satfractdqsa (long long fract a) [Runtime Function]
long accum __satfractdqda (long long fract a) [Runtime Function]
long long accum __satfractdqta (long long fract a) [Runtime Function]
unsigned short fract __satfractdquqq (long long fract a) [Runtime Function]
unsigned fract __satfractdquhq (long long fract a) [Runtime Function]
unsigned long fract __satfractdqusq (long long fract a) [Runtime Function]
unsigned long long fract __satfractdqudq (long long fract a) [Runtime Function]
unsigned short accum __satfractdquha (long long fract a) [Runtime Function]
unsigned accum __satfractdqusa (long long fract a) [Runtime Function]
unsigned long accum __satfractdquda (long long fract a) [Runtime Function]
unsigned long long accum __satfractdquta (long long fract a) [Runtime Function]

short fract __satfracthaqq (short accum a) [Runtime Function]
fract __satfracthahq (short accum a) [Runtime Function]
long fract __satfracthasq (short accum a) [Runtime Function]
long long fract __satfracthadq (short accum a) [Runtime Function]
accum __satfracthasa2 (short accum a) [Runtime Function]
long accum __satfracthada2 (short accum a) [Runtime Function]
long long accum __satfracthata2 (short accum a) [Runtime Function]
unsigned short fract __satfracthauqq (short accum a) [Runtime Function]
unsigned fract __satfracthauhq (short accum a) [Runtime Function]
unsigned long fract __satfracthausq (short accum a) [Runtime Function]
unsigned long long fract __satfracthaudq (short accum a) [Runtime Function]
unsigned short accum __satfracthauha (short accum a) [Runtime Function]
unsigned accum __satfracthausa (short accum a) [Runtime Function]
unsigned long accum __satfracthauda (short accum a) [Runtime Function]
unsigned long long accum __satfracthauta (short accum a) [Runtime Function]
short fract __satfractsaqq (accum a) [Runtime Function]
fract __satfractsahq (accum a) [Runtime Function]
long fract __satfractsasq (accum a) [Runtime Function]
long long fract __satfractsadq (accum a) [Runtime Function]
short accum __satfractsaha2 (accum a) [Runtime Function]
long accum __satfractsada2 (accum a) [Runtime Function]
long long accum __satfractsata2 (accum a) [Runtime Function]
unsigned short fract __satfractsauqq (accum a) [Runtime Function]
unsigned fract __satfractsauhq (accum a) [Runtime Function]
unsigned long fract __satfractsausq (accum a) [Runtime Function]
unsigned long long fract __satfractsaudq (accum a) [Runtime Function]
unsigned short accum __satfractsauha (accum a) [Runtime Function]
unsigned accum __satfractsausa (accum a) [Runtime Function]
unsigned long accum __satfractsauda (accum a) [Runtime Function]
unsigned long long accum __satfractsauta (accum a) [Runtime Function]
short fract __satfractdaqq (long accum a) [Runtime Function]
fract __satfractdahq (long accum a) [Runtime Function]
long fract __satfractdasq (long accum a) [Runtime Function]
long long fract __satfractdadq (long accum a) [Runtime Function]
short accum __satfractdaha2 (long accum a) [Runtime Function]
accum __satfractdasa2 (long accum a) [Runtime Function]
long long accum __satfractdata2 (long accum a) [Runtime Function]
unsigned short fract __satfractdauqq (long accum a) [Runtime Function]
unsigned fract __satfractdauhq (long accum a) [Runtime Function]
unsigned long fract __satfractdausq (long accum a) [Runtime Function]
unsigned long long fract __satfractdaudq (long accum a) [Runtime Function]
unsigned short accum __satfractdauha (long accum a) [Runtime Function]
unsigned accum __satfractdausa (long accum a) [Runtime Function]
unsigned long accum __satfractdauda (long accum a) [Runtime Function]
unsigned long long accum __satfractdauta (long accum a) [Runtime Function]
short fract __satfracttaqq (long long accum a) [Runtime Function]
fract __satfracttahq (long long accum a) [Runtime Function]

long fract __satfracttasq (long long accum a) [Runtime Function]
long long fract __satfracttadq (long long accum a) [Runtime Function]
short accum __satfracttaha2 (long long accum a) [Runtime Function]
accum __satfracttasa2 (long long accum a) [Runtime Function]
long accum __satfracttada2 (long long accum a) [Runtime Function]
unsigned short fract __satfracttauqq (long long accum a) [Runtime Function]
unsigned fract __satfracttauhq (long long accum a) [Runtime Function]
unsigned long fract __satfracttausq (long long accum a) [Runtime Function]
unsigned long long fract __satfracttaudq (long long accum a) [Runtime Function]
unsigned short accum __satfracttauha (long long accum a) [Runtime Function]
unsigned accum __satfracttausa (long long accum a) [Runtime Function]
unsigned long accum __satfracttauda (long long accum a) [Runtime Function]
unsigned long long accum __satfracttauta (long long accum a) [Runtime Function]
short fract __satfractuqqqq (unsigned short fract a) [Runtime Function]
fract __satfractuqqhq (unsigned short fract a) [Runtime Function]
long fract __satfractuqqsq (unsigned short fract a) [Runtime Function]
long long fract __satfractuqqdq (unsigned short fract a) [Runtime Function]
short accum __satfractuqqha (unsigned short fract a) [Runtime Function]
accum __satfractuqqsa (unsigned short fract a) [Runtime Function]
long accum __satfractuqqda (unsigned short fract a) [Runtime Function]
long long accum __satfractuqqta (unsigned short fract a) [Runtime Function]
unsigned fract __satfractuqquhq2 (unsigned short fract a) [Runtime Function]
unsigned long fract __satfractuqqusq2 (unsigned short fract a) [Runtime Function]
unsigned long long fract __satfractuqqudq2 (unsigned short fract a) [Runtime Function]
unsigned short accum __satfractuqquha (unsigned short fract a) [Runtime Function]
unsigned accum __satfractuqqusa (unsigned short fract a) [Runtime Function]
unsigned long accum __satfractuqquda (unsigned short fract a) [Runtime Function]
unsigned long long accum __satfractuqquta (unsigned short fract a) [Runtime Function]
short fract __satfractuhqqq (unsigned fract a) [Runtime Function]
fract __satfractuhqhq (unsigned fract a) [Runtime Function]
long fract __satfractuhqsq (unsigned fract a) [Runtime Function]
long long fract __satfractuhqdq (unsigned fract a) [Runtime Function]
short accum __satfractuhqha (unsigned fract a) [Runtime Function]
accum __satfractuhqsa (unsigned fract a) [Runtime Function]
long accum __satfractuhqda (unsigned fract a) [Runtime Function]
long long accum __satfractuhqta (unsigned fract a) [Runtime Function]
unsigned short fract __satfractuhquqq2 (unsigned fract a) [Runtime Function]
unsigned long fract __satfractuhqusq2 (unsigned fract a) [Runtime Function]
unsigned long long fract __satfractuhqudq2 (unsigned fract a) [Runtime Function]
unsigned short accum __satfractuhquha (unsigned fract a) [Runtime Function]
unsigned accum __satfractuhqusa (unsigned fract a) [Runtime Function]
unsigned long accum __satfractuhquda (unsigned fract a) [Runtime Function]
unsigned long long accum __satfractuhquta (unsigned fract a) [Runtime Function]
short fract __satfractusqqq (unsigned long fract a) [Runtime Function]
fract __satfractusqhq (unsigned long fract a) [Runtime Function]
long fract __satfractusqsq (unsigned long fract a) [Runtime Function]
long long fract __satfractusqdq (unsigned long fract a) [Runtime Function]

short accum __satfractusqha (unsigned long fract a) [Runtime Function]
accum __satfractusqsa (unsigned long fract a) [Runtime Function]
long accum __satfractusqda (unsigned long fract a) [Runtime Function]
long long accum __satfractusqta (unsigned long fract a) [Runtime Function]
unsigned short fract __satfractusquqq2 (unsigned long fract a) [Runtime Function]
unsigned fract __satfractusquhq2 (unsigned long fract a) [Runtime Function]
unsigned long long fract __satfractusqudq2 (unsigned long fract a) [Runtime Function]
unsigned short accum __satfractusquha (unsigned long fract a) [Runtime Function]
unsigned accum __satfractusqusa (unsigned long fract a) [Runtime Function]
unsigned long accum __satfractusquda (unsigned long fract a) [Runtime Function]
unsigned long long accum __satfractusquta (unsigned long fract a) [Runtime Function]
short fract __satfractudqqq (unsigned long long fract a) [Runtime Function]
fract __satfractudqhq (unsigned long long fract a) [Runtime Function]
long fract __satfractudqsq (unsigned long long fract a) [Runtime Function]
long long fract __satfractudqdq (unsigned long long fract a) [Runtime Function]
short accum __satfractudqha (unsigned long long fract a) [Runtime Function]
accum __satfractudqsa (unsigned long long fract a) [Runtime Function]
long accum __satfractudqda (unsigned long long fract a) [Runtime Function]
long long accum __satfractudqta (unsigned long long fract a) [Runtime Function]
unsigned short fract __satfractudquqq2 (unsigned long long fract a) [Runtime Function]
unsigned fract __satfractudquhq2 (unsigned long long fract a) [Runtime Function]
unsigned long fract __satfractudqusq2 (unsigned long long fract a) [Runtime Function]
unsigned short accum __satfractudquha (unsigned long long fract a) [Runtime Function]
unsigned accum __satfractudqusa (unsigned long long fract a) [Runtime Function]
unsigned long accum __satfractudquda (unsigned long long fract a) [Runtime Function]
unsigned long long accum __satfractudquta (unsigned long long fract a) [Runtime Function]
short fract __satfractuhaqq (unsigned short accum a) [Runtime Function]
fract __satfractuhahq (unsigned short accum a) [Runtime Function]
long fract __satfractuhasq (unsigned short accum a) [Runtime Function]
long long fract __satfractuhadq (unsigned short accum a) [Runtime Function]
short accum __satfractuhaha (unsigned short accum a) [Runtime Function]
accum __satfractuhasa (unsigned short accum a) [Runtime Function]
long accum __satfractuhada (unsigned short accum a) [Runtime Function]
long long accum __satfractuhata (unsigned short accum a) [Runtime Function]
unsigned short fract __satfractuhauqq (unsigned short accum a) [Runtime Function]
unsigned fract __satfractuhauhq (unsigned short accum a) [Runtime Function]
unsigned long fract __satfractuhausq (unsigned short accum a) [Runtime Function]
unsigned long long fract __satfractuhaudq (unsigned short accum a) [Runtime Function]
unsigned accum __satfractuhausa2 (unsigned short accum a) [Runtime Function]
unsigned long accum __satfractuhauda2 (unsigned short accum a) [Runtime Function]
unsigned long long accum __satfractuhauta2 (unsigned short accum a) [Runtime Function]
short fract __satfractusaqq (unsigned accum a) [Runtime Function]
fract __satfractusahq (unsigned accum a) [Runtime Function]
long fract __satfractusasq (unsigned accum a) [Runtime Function]
long long fract __satfractusadq (unsigned accum a) [Runtime Function]
short accum __satfractusaha (unsigned accum a) [Runtime Function]

accum __satfractusasa (unsigned accum a) [Runtime Function]
long accum __satfractusada (unsigned accum a) [Runtime Function]
long long accum __satfractusata (unsigned accum a) [Runtime Function]
unsigned short fract __satfractusauqq (unsigned accum a) [Runtime Function]
unsigned fract __satfractusauhq (unsigned accum a) [Runtime Function]
unsigned long fract __satfractusausq (unsigned accum a) [Runtime Function]
unsigned long long fract __satfractusaudq (unsigned accum a) [Runtime Function]
unsigned short accum __satfractusauha2 (unsigned accum a) [Runtime Function]
unsigned long accum __satfractusauda2 (unsigned accum a) [Runtime Function]
unsigned long long accum __satfractusauta2 (unsigned accum a) [Runtime Function]
short fract __satfractudaqq (unsigned long accum a) [Runtime Function]
fract __satfractudahq (unsigned long accum a) [Runtime Function]
long fract __satfractudasq (unsigned long accum a) [Runtime Function]
long long fract __satfractudadq (unsigned long accum a) [Runtime Function]
short accum __satfractudaha (unsigned long accum a) [Runtime Function]
accum __satfractudasa (unsigned long accum a) [Runtime Function]
long accum __satfractudada (unsigned long accum a) [Runtime Function]
long long accum __satfractudata (unsigned long accum a) [Runtime Function]
unsigned short fract __satfractudauqq (unsigned long accum a) [Runtime Function]
unsigned fract __satfractudauhq (unsigned long accum a) [Runtime Function]
unsigned long fract __satfractudausq (unsigned long accum a) [Runtime Function]
unsigned long long fract __satfractudaudq (unsigned long accum a) [Runtime Function]
unsigned short accum __satfractudauha2 (unsigned long accum a) [Runtime Function]
unsigned accum __satfractudausa2 (unsigned long accum a) [Runtime Function]
unsigned long long accum __satfractudauta2 (unsigned long accum a) [Runtime Function]
short fract __satfractutaqq (unsigned long long accum a) [Runtime Function]
fract __satfractutahq (unsigned long long accum a) [Runtime Function]
long fract __satfractutasq (unsigned long long accum a) [Runtime Function]
long long fract __satfractutadq (unsigned long long accum a) [Runtime Function]
short accum __satfractutaha (unsigned long long accum a) [Runtime Function]
accum __satfractutasa (unsigned long long accum a) [Runtime Function]
long accum __satfractutada (unsigned long long accum a) [Runtime Function]
long long accum __satfractutata (unsigned long long accum a) [Runtime Function]
unsigned short fract __satfractutauqq (unsigned long long accum a) [Runtime Function]
unsigned fract __satfractutauhq (unsigned long long accum a) [Runtime Function]
unsigned long fract __satfractutausq (unsigned long long accum a) [Runtime Function]
unsigned long long fract __satfractutaudq (unsigned long long accum a) [Runtime Function]
unsigned short accum __satfractutauha2 (unsigned long long accum a) [Runtime Function]
unsigned accum __satfractutausa2 (unsigned long long accum a) [Runtime Function]
unsigned long accum __satfractutauda2 (unsigned long long accum a) [Runtime Function]
short fract __satfractqiqq (signed char a) [Runtime Function]
fract __satfractqihq (signed char a) [Runtime Function]
long fract __satfractqisq (signed char a) [Runtime Function]
long long fract __satfractqidq (signed char a) [Runtime Function]
short accum __satfractqiha (signed char a) [Runtime Function]
accum __satfractqisa (signed char a) [Runtime Function]

long accum __satfractqida (signed char a) [Runtime Function]
long long accum __satfractqita (signed char a) [Runtime Function]
unsigned short fract __satfractqiuqq (signed char a) [Runtime Function]
unsigned fract __satfractqiuhq (signed char a) [Runtime Function]
unsigned long fract __satfractqiusq (signed char a) [Runtime Function]
unsigned long long fract __satfractqiudq (signed char a) [Runtime Function]
unsigned short accum __satfractqiuha (signed char a) [Runtime Function]
unsigned accum __satfractqiusa (signed char a) [Runtime Function]
unsigned long accum __satfractqiuda (signed char a) [Runtime Function]
unsigned long long accum __satfractqiuta (signed char a) [Runtime Function]
short fract __satfracthiqq (short a) [Runtime Function]
fract __satfracthihq (short a) [Runtime Function]
long fract __satfracthisq (short a) [Runtime Function]
long long fract __satfracthidq (short a) [Runtime Function]
short accum __satfracthiha (short a) [Runtime Function]
accum __satfracthisa (short a) [Runtime Function]
long accum __satfracthida (short a) [Runtime Function]
long long accum __satfracthita (short a) [Runtime Function]
unsigned short fract __satfracthiuqq (short a) [Runtime Function]
unsigned fract __satfracthiuhq (short a) [Runtime Function]
unsigned long fract __satfracthiusq (short a) [Runtime Function]
unsigned long long fract __satfracthiudq (short a) [Runtime Function]
unsigned short accum __satfracthiuha (short a) [Runtime Function]
unsigned accum __satfracthiusa (short a) [Runtime Function]
unsigned long accum __satfracthiuda (short a) [Runtime Function]
unsigned long long accum __satfracthiuta (short a) [Runtime Function]
short fract __satfractsiqq (int a) [Runtime Function]
fract __satfractsihq (int a) [Runtime Function]
long fract __satfractsisq (int a) [Runtime Function]
long long fract __satfractsidq (int a) [Runtime Function]
short accum __satfractsiha (int a) [Runtime Function]
accum __satfractsisa (int a) [Runtime Function]
long accum __satfractsida (int a) [Runtime Function]
long long accum __satfractsita (int a) [Runtime Function]
unsigned short fract __satfractsiuqq (int a) [Runtime Function]
unsigned fract __satfractsiuhq (int a) [Runtime Function]
unsigned long fract __satfractsiusq (int a) [Runtime Function]
unsigned long long fract __satfractsiudq (int a) [Runtime Function]
unsigned short accum __satfractsiuha (int a) [Runtime Function]
unsigned accum __satfractsiusa (int a) [Runtime Function]
unsigned long accum __satfractsiuda (int a) [Runtime Function]
unsigned long long accum __satfractsiuta (int a) [Runtime Function]
short fract __satfractdiqq (long a) [Runtime Function]
fract __satfractdihq (long a) [Runtime Function]
long fract __satfractdisq (long a) [Runtime Function]
long long fract __satfractdidq (long a) [Runtime Function]
short accum __satfractdiha (long a) [Runtime Function]

accum __satfractdisa (long a) [Runtime Function]
long accum __satfractdida (long a) [Runtime Function]
long long accum __satfractdita (long a) [Runtime Function]
unsigned short fract __satfractdiuqq (long a) [Runtime Function]
unsigned fract __satfractdiuhq (long a) [Runtime Function]
unsigned long fract __satfractdiusq (long a) [Runtime Function]
unsigned long long fract __satfractdiudq (long a) [Runtime Function]
unsigned short accum __satfractdiuha (long a) [Runtime Function]
unsigned accum __satfractdiusa (long a) [Runtime Function]
unsigned long accum __satfractdiuda (long a) [Runtime Function]
unsigned long long accum __satfractdiuta (long a) [Runtime Function]
short fract __satfracttiqq (long long a) [Runtime Function]
fract __satfracttihq (long long a) [Runtime Function]
long fract __satfracttisq (long long a) [Runtime Function]
long long fract __satfracttidq (long long a) [Runtime Function]
short accum __satfracttiha (long long a) [Runtime Function]
accum __satfracttisa (long long a) [Runtime Function]
long accum __satfracttida (long long a) [Runtime Function]
long long accum __satfracttita (long long a) [Runtime Function]
unsigned short fract __satfracttiuqq (long long a) [Runtime Function]
unsigned fract __satfracttiuhq (long long a) [Runtime Function]
unsigned long fract __satfracttiusq (long long a) [Runtime Function]
unsigned long long fract __satfracttiudq (long long a) [Runtime Function]
unsigned short accum __satfracttiuha (long long a) [Runtime Function]
unsigned accum __satfracttiusa (long long a) [Runtime Function]
unsigned long accum __satfracttiuda (long long a) [Runtime Function]
unsigned long long accum __satfracttiuta (long long a) [Runtime Function]
short fract __satfractsfqq (float a) [Runtime Function]
fract __satfractsfhq (float a) [Runtime Function]
long fract __satfractsfsq (float a) [Runtime Function]
long long fract __satfractsfdq (float a) [Runtime Function]
short accum __satfractsfha (float a) [Runtime Function]
accum __satfractsfsa (float a) [Runtime Function]
long accum __satfractsfda (float a) [Runtime Function]
long long accum __satfractsfta (float a) [Runtime Function]
unsigned short fract __satfractsfuqq (float a) [Runtime Function]
unsigned fract __satfractsfuhq (float a) [Runtime Function]
unsigned long fract __satfractsfusq (float a) [Runtime Function]
unsigned long long fract __satfractsfudq (float a) [Runtime Function]
unsigned short accum __satfractsfuha (float a) [Runtime Function]
unsigned accum __satfractsfusa (float a) [Runtime Function]
unsigned long accum __satfractsfuda (float a) [Runtime Function]
unsigned long long accum __satfractsfuta (float a) [Runtime Function]
short fract __satfractdfqq (double a) [Runtime Function]
fract __satfractdfhq (double a) [Runtime Function]
long fract __satfractdfsq (double a) [Runtime Function]
long long fract __satfractdfdq (double a) [Runtime Function]

short accum __satfractdfha (double a) [Runtime Function]
accum __satfractdfsa (double a) [Runtime Function]
long accum __satfractdfda (double a) [Runtime Function]
long long accum __satfractdfta (double a) [Runtime Function]
unsigned short fract __satfractdfuqq (double a) [Runtime Function]
unsigned fract __satfractdfuhq (double a) [Runtime Function]
unsigned long fract __satfractdfusq (double a) [Runtime Function]
unsigned long long fract __satfractdfudq (double a) [Runtime Function]
unsigned short accum __satfractdfuha (double a) [Runtime Function]
unsigned accum __satfractdfusa (double a) [Runtime Function]
unsigned long accum __satfractdfuda (double a) [Runtime Function]
unsigned long long accum __satfractdfuta (double a) [Runtime Function]

The functions convert from fractional and signed non-fractionals to fractionals, with saturation.

unsigned char __fractunsqqqi (short fract a) [Runtime Function]
unsigned short __fractunsqqhi (short fract a) [Runtime Function]
unsigned int __fractunsqqsi (short fract a) [Runtime Function]
unsigned long __fractunsqqdi (short fract a) [Runtime Function]
unsigned long long __fractunsqqti (short fract a) [Runtime Function]
unsigned char __fractunshqqi (fract a) [Runtime Function]
unsigned short __fractunshqhi (fract a) [Runtime Function]
unsigned int __fractunshqsi (fract a) [Runtime Function]
unsigned long __fractunshqdi (fract a) [Runtime Function]
unsigned long long __fractunshqti (fract a) [Runtime Function]
unsigned char __fractunssqqi (long fract a) [Runtime Function]
unsigned short __fractunssqhi (long fract a) [Runtime Function]
unsigned int __fractunssqsi (long fract a) [Runtime Function]
unsigned long __fractunssqdi (long fract a) [Runtime Function]
unsigned long long __fractunssqti (long fract a) [Runtime Function]
unsigned char __fractunsdqqi (long long fract a) [Runtime Function]
unsigned short __fractunsdqhi (long long fract a) [Runtime Function]
unsigned int __fractunsdqsi (long long fract a) [Runtime Function]
unsigned long __fractunsdqdi (long long fract a) [Runtime Function]
unsigned long long __fractunsdqti (long long fract a) [Runtime Function]
unsigned char __fractunshaqi (short accum a) [Runtime Function]
unsigned short __fractunshahi (short accum a) [Runtime Function]
unsigned int __fractunshasi (short accum a) [Runtime Function]
unsigned long __fractunshadi (short accum a) [Runtime Function]
unsigned long long __fractunshati (short accum a) [Runtime Function]
unsigned char __fractunssaqi (accum a) [Runtime Function]
unsigned short __fractunssahi (accum a) [Runtime Function]
unsigned int __fractunssasi (accum a) [Runtime Function]
unsigned long __fractunssadi (accum a) [Runtime Function]
unsigned long long __fractunssati (accum a) [Runtime Function]
unsigned char __fractunsdaqi (long accum a) [Runtime Function]
unsigned short __fractunsdahi (long accum a) [Runtime Function]

unsigned int __fractunsdasi (long accum a) [Runtime Function]
unsigned long __fractunsdadi (long accum a) [Runtime Function]
unsigned long long __fractunsdati (long accum a) [Runtime Function]
unsigned char __fractunstaqi (long long accum a) [Runtime Function]
unsigned short __fractunstahi (long long accum a) [Runtime Function]
unsigned int __fractunstasi (long long accum a) [Runtime Function]
unsigned long __fractunstadi (long long accum a) [Runtime Function]
unsigned long long __fractunstati (long long accum a) [Runtime Function]
unsigned char __fractunsuqqqi (unsigned short fract a) [Runtime Function]
unsigned short __fractunsuqqhi (unsigned short fract a) [Runtime Function]
unsigned int __fractunsuqqsi (unsigned short fract a) [Runtime Function]
unsigned long __fractunsuqqdi (unsigned short fract a) [Runtime Function]
unsigned long long __fractunsuqqti (unsigned short fract a) [Runtime Function]
unsigned char __fractunsuhqqi (unsigned fract a) [Runtime Function]
unsigned short __fractunsuhqhi (unsigned fract a) [Runtime Function]
unsigned int __fractunsuhqsi (unsigned fract a) [Runtime Function]
unsigned long __fractunsuhqdi (unsigned fract a) [Runtime Function]
unsigned long long __fractunsuhqti (unsigned fract a) [Runtime Function]
unsigned char __fractunsusqqi (unsigned long fract a) [Runtime Function]
unsigned short __fractunsusqhi (unsigned long fract a) [Runtime Function]
unsigned int __fractunsusqsi (unsigned long fract a) [Runtime Function]
unsigned long __fractunsusqdi (unsigned long fract a) [Runtime Function]
unsigned long long __fractunsusqti (unsigned long fract a) [Runtime Function]
unsigned char __fractunsudqqi (unsigned long long fract a) [Runtime Function]
unsigned short __fractunsudqhi (unsigned long long fract a) [Runtime Function]
unsigned int __fractunsudqsi (unsigned long long fract a) [Runtime Function]
unsigned long __fractunsudqdi (unsigned long long fract a) [Runtime Function]
unsigned long long __fractunsudqti (unsigned long long fract a) [Runtime Function]
unsigned char __fractunsuhaqi (unsigned short accum a) [Runtime Function]
unsigned short __fractunsuhahi (unsigned short accum a) [Runtime Function]
unsigned int __fractunsuhasi (unsigned short accum a) [Runtime Function]
unsigned long __fractunsuhadi (unsigned short accum a) [Runtime Function]
unsigned long long __fractunsuhati (unsigned short accum a) [Runtime Function]
unsigned char __fractunsusaqi (unsigned accum a) [Runtime Function]
unsigned short __fractunsusahi (unsigned accum a) [Runtime Function]
unsigned int __fractunsusasi (unsigned accum a) [Runtime Function]
unsigned long __fractunsusadi (unsigned accum a) [Runtime Function]
unsigned long long __fractunsusati (unsigned accum a) [Runtime Function]
unsigned char __fractunsudaqi (unsigned long accum a) [Runtime Function]
unsigned short __fractunsudahi (unsigned long accum a) [Runtime Function]
unsigned int __fractunsudasi (unsigned long accum a) [Runtime Function]
unsigned long __fractunsudadi (unsigned long accum a) [Runtime Function]
unsigned long long __fractunsudati (unsigned long accum a) [Runtime Function]
unsigned char __fractunsutaqi (unsigned long long accum a) [Runtime Function]
unsigned short __fractunsutahi (unsigned long long accum a) [Runtime Function]
unsigned int __fractunsutasi (unsigned long long accum a) [Runtime Function]
unsigned long __fractunsutadi (unsigned long long accum a) [Runtime Function]

unsigned long long __fractunsutati (unsigned long long accum a) [Runtime Function]
short fract __fractunsqiqq (unsigned char a) [Runtime Function]
fract __fractunsqihq (unsigned char a) [Runtime Function]
long fract __fractunsqisq (unsigned char a) [Runtime Function]
long long fract __fractunsqidq (unsigned char a) [Runtime Function]
short accum __fractunsqiha (unsigned char a) [Runtime Function]
accum __fractunsqisa (unsigned char a) [Runtime Function]
long accum __fractunsqida (unsigned char a) [Runtime Function]
long long accum __fractunsqita (unsigned char a) [Runtime Function]
unsigned short fract __fractunsqiuqq (unsigned char a) [Runtime Function]
unsigned fract __fractunsqiuhq (unsigned char a) [Runtime Function]
unsigned long fract __fractunsqiusq (unsigned char a) [Runtime Function]
unsigned long long fract __fractunsqiudq (unsigned char a) [Runtime Function]
unsigned short accum __fractunsqiuha (unsigned char a) [Runtime Function]
unsigned accum __fractunsqiusa (unsigned char a) [Runtime Function]
unsigned long accum __fractunsqiuda (unsigned char a) [Runtime Function]
unsigned long long accum __fractunsqiuta (unsigned char a) [Runtime Function]
short fract __fractunshiqq (unsigned short a) [Runtime Function]
fract __fractunshihq (unsigned short a) [Runtime Function]
long fract __fractunshisq (unsigned short a) [Runtime Function]
long long fract __fractunshidq (unsigned short a) [Runtime Function]
short accum __fractunshiha (unsigned short a) [Runtime Function]
accum __fractunshisa (unsigned short a) [Runtime Function]
long accum __fractunshida (unsigned short a) [Runtime Function]
long long accum __fractunshita (unsigned short a) [Runtime Function]
unsigned short fract __fractunshiuqq (unsigned short a) [Runtime Function]
unsigned fract __fractunshiuhq (unsigned short a) [Runtime Function]
unsigned long fract __fractunshiusq (unsigned short a) [Runtime Function]
unsigned long long fract __fractunshiudq (unsigned short a) [Runtime Function]
unsigned short accum __fractunshiuha (unsigned short a) [Runtime Function]
unsigned accum __fractunshiusa (unsigned short a) [Runtime Function]
unsigned long accum __fractunshiuda (unsigned short a) [Runtime Function]
unsigned long long accum __fractunshiuta (unsigned short a) [Runtime Function]
short fract __fractunssiqq (unsigned int a) [Runtime Function]
fract __fractunssihq (unsigned int a) [Runtime Function]
long fract __fractunssisq (unsigned int a) [Runtime Function]
long long fract __fractunssidq (unsigned int a) [Runtime Function]
short accum __fractunssiha (unsigned int a) [Runtime Function]
accum __fractunssisa (unsigned int a) [Runtime Function]
long accum __fractunssida (unsigned int a) [Runtime Function]
long long accum __fractunssita (unsigned int a) [Runtime Function]
unsigned short fract __fractunssiuqq (unsigned int a) [Runtime Function]
unsigned fract __fractunssiuhq (unsigned int a) [Runtime Function]
unsigned long fract __fractunssiusq (unsigned int a) [Runtime Function]
unsigned long long fract __fractunssiudq (unsigned int a) [Runtime Function]
unsigned short accum __fractunssiuha (unsigned int a) [Runtime Function]
unsigned accum __fractunssiusa (unsigned int a) [Runtime Function]

unsigned long accum __fractunssiuda (unsigned int a) [Runtime Function]
unsigned long long accum __fractunssiuta (unsigned int a) [Runtime Function]
short fract __fractunsdiqq (unsigned long a) [Runtime Function]
fract __fractunsdihq (unsigned long a) [Runtime Function]
long fract __fractunsdisq (unsigned long a) [Runtime Function]
long long fract __fractunsdidq (unsigned long a) [Runtime Function]
short accum __fractunsdiha (unsigned long a) [Runtime Function]
accum __fractunsdisa (unsigned long a) [Runtime Function]
long accum __fractunsdida (unsigned long a) [Runtime Function]
long long accum __fractunsdita (unsigned long a) [Runtime Function]
unsigned short fract __fractunsdiuqq (unsigned long a) [Runtime Function]
unsigned fract __fractunsdiuhq (unsigned long a) [Runtime Function]
unsigned long fract __fractunsdiusq (unsigned long a) [Runtime Function]
unsigned long long fract __fractunsdiudq (unsigned long a) [Runtime Function]
unsigned short accum __fractunsdiuha (unsigned long a) [Runtime Function]
unsigned accum __fractunsdiusa (unsigned long a) [Runtime Function]
unsigned long accum __fractunsdiuda (unsigned long a) [Runtime Function]
unsigned long long accum __fractunsdiuta (unsigned long a) [Runtime Function]
short fract __fractunstiqq (unsigned long long a) [Runtime Function]
fract __fractunstihq (unsigned long long a) [Runtime Function]
long fract __fractunstisq (unsigned long long a) [Runtime Function]
long long fract __fractunstidq (unsigned long long a) [Runtime Function]
short accum __fractunstiha (unsigned long long a) [Runtime Function]
accum __fractunstisa (unsigned long long a) [Runtime Function]
long accum __fractunstida (unsigned long long a) [Runtime Function]
long long accum __fractunstita (unsigned long long a) [Runtime Function]
unsigned short fract __fractunstiuqq (unsigned long long a) [Runtime Function]
unsigned fract __fractunstiuhq (unsigned long long a) [Runtime Function]
unsigned long fract __fractunstiusq (unsigned long long a) [Runtime Function]
unsigned long long fract __fractunstiudq (unsigned long long a) [Runtime Function]
unsigned short accum __fractunstiuha (unsigned long long a) [Runtime Function]
unsigned accum __fractunstiusa (unsigned long long a) [Runtime Function]
unsigned long accum __fractunstiuda (unsigned long long a) [Runtime Function]
unsigned long long accum __fractunstiuta (unsigned long long a) [Runtime Function]

These functions convert from fractionals to unsigned non-fractionals; and from unsigned non-fractionals to fractionals, without saturation.

short fract __satfractunsqiqq (unsigned char a) [Runtime Function]
fract __satfractunsqihq (unsigned char a) [Runtime Function]
long fract __satfractunsqisq (unsigned char a) [Runtime Function]
long long fract __satfractunsqidq (unsigned char a) [Runtime Function]
short accum __satfractunsqiha (unsigned char a) [Runtime Function]
accum __satfractunsqisa (unsigned char a) [Runtime Function]
long accum __satfractunsqida (unsigned char a) [Runtime Function]
long long accum __satfractunsqita (unsigned char a) [Runtime Function]
unsigned short fract __satfractunsqiuqq (unsigned char a) [Runtime Function]
unsigned fract __satfractunsqiuhq (unsigned char a) [Runtime Function]

unsigned long fract __satfractunsqiusq (unsigned char a) [Runtime Function]
unsigned long long fract __satfractunsqiudq (unsigned char a) [Runtime Function]
unsigned short accum __satfractunsqiuha (unsigned char a) [Runtime Function]
unsigned accum __satfractunsqiusa (unsigned char a) [Runtime Function]
unsigned long accum __satfractunsqiuda (unsigned char a) [Runtime Function]
unsigned long long accum __satfractunsqiuta (unsigned char a) [Runtime Function]
short fract __satfractunshiqq (unsigned short a) [Runtime Function]
fract __satfractunshihq (unsigned short a) [Runtime Function]
long fract __satfractunshisq (unsigned short a) [Runtime Function]
long long fract __satfractunshidq (unsigned short a) [Runtime Function]
short accum __satfractunshiha (unsigned short a) [Runtime Function]
accum __satfractunshisa (unsigned short a) [Runtime Function]
long accum __satfractunshida (unsigned short a) [Runtime Function]
long long accum __satfractunshita (unsigned short a) [Runtime Function]
unsigned short fract __satfractunshiuqq (unsigned short a) [Runtime Function]
unsigned fract __satfractunshiuhq (unsigned short a) [Runtime Function]
unsigned long fract __satfractunshiusq (unsigned short a) [Runtime Function]
unsigned long long fract __satfractunshiudq (unsigned short a) [Runtime Function]
unsigned short accum __satfractunshiuha (unsigned short a) [Runtime Function]
unsigned accum __satfractunshiusa (unsigned short a) [Runtime Function]
unsigned long accum __satfractunshiuda (unsigned short a) [Runtime Function]
unsigned long long accum __satfractunshiuta (unsigned short a) [Runtime Function]
short fract __satfractunssiqq (unsigned int a) [Runtime Function]
fract __satfractunssihq (unsigned int a) [Runtime Function]
long fract __satfractunssisq (unsigned int a) [Runtime Function]
long long fract __satfractunssidq (unsigned int a) [Runtime Function]
short accum __satfractunssiha (unsigned int a) [Runtime Function]
accum __satfractunssisa (unsigned int a) [Runtime Function]
long accum __satfractunssida (unsigned int a) [Runtime Function]
long long accum __satfractunssita (unsigned int a) [Runtime Function]
unsigned short fract __satfractunssiuqq (unsigned int a) [Runtime Function]
unsigned fract __satfractunssiuhq (unsigned int a) [Runtime Function]
unsigned long fract __satfractunssiusq (unsigned int a) [Runtime Function]
unsigned long long fract __satfractunssiudq (unsigned int a) [Runtime Function]
unsigned short accum __satfractunssiuha (unsigned int a) [Runtime Function]
unsigned accum __satfractunssiusa (unsigned int a) [Runtime Function]
unsigned long accum __satfractunssiuda (unsigned int a) [Runtime Function]
unsigned long long accum __satfractunssiuta (unsigned int a) [Runtime Function]
short fract __satfractunsdiqq (unsigned long a) [Runtime Function]
fract __satfractunsdihq (unsigned long a) [Runtime Function]
long fract __satfractunsdisq (unsigned long a) [Runtime Function]
long long fract __satfractunsdidq (unsigned long a) [Runtime Function]
short accum __satfractunsdiha (unsigned long a) [Runtime Function]
accum __satfractunsdisa (unsigned long a) [Runtime Function]
long accum __satfractunsdida (unsigned long a) [Runtime Function]
long long accum __satfractunsdita (unsigned long a) [Runtime Function]
unsigned short fract __satfractunsdiuqq (unsigned long a) [Runtime Function]

unsigned fract __satfractunsdiuhq (unsigned long a) [Runtime Function]
unsigned long fract __satfractunsdiusq (unsigned long a) [Runtime Function]
unsigned long long fract __satfractunsdiudq (unsigned long a) [Runtime Function]
unsigned short accum __satfractunsdiuha (unsigned long a) [Runtime Function]
unsigned accum __satfractunsdiusa (unsigned long a) [Runtime Function]
unsigned long accum __satfractunsdiuda (unsigned long a) [Runtime Function]
unsigned long long accum __satfractunsdiuta (unsigned long a) [Runtime Function]
short fract __satfractunstiqq (unsigned long long a) [Runtime Function]
fract __satfractunstihq (unsigned long long a) [Runtime Function]
long fract __satfractunstisq (unsigned long long a) [Runtime Function]
long long fract __satfractunstidq (unsigned long long a) [Runtime Function]
short accum __satfractunstiha (unsigned long long a) [Runtime Function]
accum __satfractunstisa (unsigned long long a) [Runtime Function]
long accum __satfractunstida (unsigned long long a) [Runtime Function]
long long accum __satfractunstita (unsigned long long a) [Runtime Function]
unsigned short fract __satfractunstiuqq (unsigned long long a) [Runtime Function]
unsigned fract __satfractunstiuhq (unsigned long long a) [Runtime Function]
unsigned long fract __satfractunstiusq (unsigned long long a) [Runtime Function]
unsigned long long fract __satfractunstiudq (unsigned long long a) [Runtime Function]
unsigned short accum __satfractunstiuha (unsigned long long a) [Runtime Function]
unsigned accum __satfractunstiusa (unsigned long long a) [Runtime Function]
unsigned long accum __satfractunstiuda (unsigned long long a) [Runtime Function]
unsigned long long accum __satfractunstiuta (unsigned long long a) [Runtime Function]

These functions convert from unsigned non-fractionals to fractionals, with saturation.

4.5 语言无关的异常处理例程

document me!

```
_Unwind_DeleteException
_Unwind_Find_FDE
_Unwind_ForcedUnwind
_Unwind_GetGR
_Unwind_GetIP
_Unwind_GetLanguageSpecificData
_Unwind_GetRegionStart
_Unwind_GetTextRelBase
_Unwind_GetDataRelBase
_Unwind_RaiseException
_Unwind_Resume
_Unwind_SetGR
_Unwind_SetIP
_Unwind_FindEnclosingFunction
_Unwind_SjLj_Register
_Unwind_SjLj_Unregister
_Unwind_SjLj_RaiseException
_Unwind_SjLj_ForcedUnwind
_Unwind_SjLj_Resume
__deregister_frame
__deregister_frame_info
__deregister_frame_info_bases
__register_frame
__register_frame_info
```

```
__register_frame_info_bases
__register_frame_info_table
__register_frame_info_table_bases
__register_frame_table
```

4.6 其它运行时库例程

4.6.1 Cache控制函数

`void __clear_cache (char *beg, char *end)` [Runtime Function]

该函数清除 beg 和 end 之间的指令缓存。

5 GCC中的语言前端

GCC中的语言前端接口，特别是 `tree` 结构（参见 Chapter 9 [Trees], page 79），起初是为C 设计的，在许多方面仍然有些偏向于C和类C的语言。尽管如此，这种设计也相当适合于其它的过程语言，而且GCC已经拥有了许多这样的语言前端。

为GCC写一个前端编译器，而不是直接编译成汇编或者生成C代码然后再用GCC编译， 具有多个优势：

- GCC前端可以受益于GCC中已有的对众多不同目标机器的支持。
- GCC前端可以受益于GCC中的所有优化。其中一些优化，例如别名分析， 在GCC直接编译源代码时要比编译生成的C代码效果更好。
- 直接编译源代码能够比编译中间生成的C代码产生更好的调试信息。

正是由于为GCC编写一个前端编译器有这些优势，使得也有与GCC设计差异很大的语言前端 被创建，例如声明式的逻辑/功能语言（declarative logic/functional language）Mercury。由于这些原因， 实现针对特殊目的（例如，作为研究项目的一部分）编译器作为GCC前端也是有用的。

6 源目录结构和构建系统

这一章描述了GCC的源文件目录结构，以及GCC是如何被构建的。关于构建 和安装GCC的用户文档在另一个单独的手册里作介绍（`http://gcc.gnu.org/install/`），这里假设你对此已经有所了 解。

6.1 配置术语和历史

配置和构建过程有着悠久灿烂的历史，如果不清楚其缘由，很容易使人迷惑不解。 虽然有另外的文献详细描述了配置过程，这里还是给出了一些从事GCC工作都应该 了解的内容。

构建程序要知道三个系统名称：构建时所使用的机器（build），构建完成 后将使用的机器（host），以及GCC未来生成代码所要运行的机器（target）。 在配置GCC时，通过 `--build=', `--host=', `--target=' 来指定这些系统名称。

要避免只指定host而不指定build，因为 `configure` 程序可能会认为 你所指定的host和build相同（曾经发生过），而实际上可能并非如此。

我们把build, host和target都相同的情况叫做 native（本地的）。如果 build和host相同，但target不同，就叫做 cross（交叉的）。 build, host和target都不同的情况则被称为 canadian（加拿大的——用来暗指加 拿大政党状况与当时从事构建工作的人的背景类似）。如果host和 target相

同， 但build不同，则表明你在使用交叉编译器来为一个不同的系统构建本地编译器。 有些人把这称为 host-x-host，crossed native（交叉的本地的）， 或 cross-built native(交叉构建的本地的）。如果build和target相同， 但host不同，则表明你在使用交叉编译器来构建一个产生构建时所在机器代码的 交叉编译器。这种情况很少见，所以没有通用的方式来描述它。有人建议称之为 crossback。

如果build和host相同，则正要构建的GCC还会被用于构建目标库（比如 `libstdc++`）。如果build和host不同，那么必须事先构建和安装一个交叉 编译器，用于构建目标库（如果使用的配置为 `` `--target=foo-bar' ``，这个 编译器就叫做 `foo-bar-gcc`）。

对于目标库的情况，你所构建的目标机器就是通过 `` `--target' `` 指定的 机器。所以，build就是在上面进行构建的机器（这没有什么不同），host就是 为其构建的机器（目标库是为target构建的，所以host也就是所指定的target）， 同时无需使用target（因为不是在构建编译器，而是在构建库）。configure/make 过程会在必要时调整这些变量。它还会把 `$with_cross_host` 设置成 `` `--host' `` 的初始值，以供不时之需。

支持库 `libiberty` 最多会被构建三次：一次针对host，一次针对target （即使二者相同），如果build和host不同，还会针对build再构建一次。这样支 持库libiberty就可以被构建过程中生成的所有程序使用了。

6.2 顶层源文件目录

在GCC发行版本中，顶层源目录下包含了几个与其它软件发行版本，比如 GNU Binutils，共享的文件和目录。另外还包含几个子目录，里面包含了 GCC的各个部分以及运行时库。

`` `boehm-gc' `` Boehm保守垃圾收集器，作为Java运行时库的一部分。

`` `contrib' `` 贡献的脚本，可以用来和GCC一起使用。其中，`` `contrib/texi2pod.pl' `` 作为GCC构建过程的一部分，可以用来将Texinfo手册生成为man页。

`` `fastjar' `` `jar` 命令的一个实现，和Java前端一起使用。

`` `fixincludes' ``
用于修订系统头文件，使得可以和GCC一起工作。更多信息参见 `` `fixincludes/README' ``。通过这种机制修订的头文件被安装在 `` `libsubdir/include-fixed' ``下。`` `README-fixinc' ``也一起被安装，作为 `` `libsubdir/include-fixed/README' ``。

`` `gcc' `` GCC本身的主要源文件（运行时库除外），包括优化器，不同目标体系结构 的支持，语言前端和测试包。 参见Section 6.3 [`gcc' 子目录], page 47。

`` `include' `` `libiberty` 库的头文件。

`` `intl' `` GNU `libintl`，来自GNU `gettext`，用于在libc中不包含它的系统上。

`` `libada' `` Ada运行时库。

`` `libcpp' `` C预处理器库。

`` `libgfortran' ``
Fortran运行时库。

`` `libffi' `` `libffi` 库，作为Java运行时库的一部分。

`` `libiberty' `` `libiberty` 库，实现了一些常用的数据结构和算法，用于提高可移植性。 see Section ``Introduction'' in *gnu* libiberty。

`` `libjava' `` Java运行时库。

`libmudflap'
: libmudflap 库用于instrument指针和数组的dereferencing操作。

`libobjc'
: Objective-C和Objective-C++运行时库。

`libstdc++-v3'
: C++运行时库。

`maintainer-scripts'
: gcc.gnu.org 上的 gccadmin 帐户使用的脚本。

`zlib'
: zlib zlib压缩库，用于Java前端，作为Java运行库的一部分。

顶层目录的构建系统，包括子目录递归的工作方式，以及构建各种库的运行时库的 处理方式，这些在GNU Binutils的一个单独手册中有介绍。 see Section ``GNU configure and build system'' in The GNU configure and build system

6.3 `gcc' 子目录

`gcc' 目录包含了GCC的许多C源文件，用于配置和构建过程的其 它文件，以及文档和测试包子目录。GCC的源文件在一个单独的章节里 进行介绍。参见Chapter 8 [编译器的Passes和相关文件], page 69。

6.3.1 `gcc'的子目录

`gcc' 目录包含了下列子目录：

`language'
: 各种语言的子目录。包含了文件 `config-lang.in' 的目录是语言子目录。 子目录 `cp' (C++), `objc' (Objective-C) 和 `objcp' (Objective-C++)的内容在该手册中有介绍（参见 Chapter 8 [Passes and Files of the Compiler], page 69 ）；其它语言的没有。 关于这些目录下的文件的详细信息，参见 Section 6.3.8 [语言前端剖析], page 53。

`config'
: 支持的体系结构和操作系统的配置文件。关于这个目录下的文件的详情， 参见 Section 6.3.9 [目标后端剖析], page 56。

`doc'
: GCC的Texinfo文档，和自动生成的man页，以及对将安装手册转换成HTML的支持。参见 Section 6.3.7 [文档], page 51。

`ginclude'
: GCC安装的，那些主要由独立实现C标准所需要的系统头文件。 关于什么时候安装这些以及其它头文件的详情， 参见 Section 6.3.6 [GCC安装的头文件], page 51。

`po'
: 用于把GCC产生的信息翻译成各种语言（如德语、西班牙语等）的信息目录，`language.po'。这个目录还包含了信息目录模板 `gcc.pot', 封装 gettext 以提取GCC信息并创建 `gcc.pot' 的 `exgettext' （通过命令 `make gcc.pot' 执行），以及不需要提取信息的文件列表 `EXCLUDES'。

`testsuite'
: GCC测试包（运行时库的除外）。参见 Section 6.4 [测试包], page 57。

6.3.2 `gcc'目录下的配置

`gcc' 目录是通过Autoconf生成的脚本`configure'来配置的。 `configure'脚本由`configure.ac'和`aclocal.m4'生成。 Autoheader通过`configure.ac'和`acconfig.h'来生成文件`config.in'。 文件`cstamp-h.in'作为时间戳使用。

6.3.2.1 `configure'使用的脚本

`configure'使用一些其它的脚本帮助其来完成工作：

- 使用了位于顶层目录的标准GNU `config.sub'和`config.guess'文件。
- 文件`config.gcc'被用来处理针对特定目标机器的配置。 文件`config.build'被用来处理针对特定构建机器的配置。 文件`config.host'被用来处理针对特定主机的配置。（通常，这些只用于那些无法通过Autoconf测试来得到的那些特征。）关于这些文件的详细内容，参见 Section 6.3.2.2 [`config.build'，`config.host'和`config.gcc'文件], page 48。
- 每个language子目录有一个`language/config-lang.in'，用来进行前端特定的配置。关于该文件的详情，参见 Section 6.3.8.2 [The Front End `config-lang.in' File], page 56。
- 辅助脚本`configure.frag'，被`configure'用来产生输出。

6.3.2.2 `config.build'，`config.host'和`config.gcc'文件

`config.build'文件包含了针对构建GCC的特殊系统的特定规则。 通常尽量不使用这个文件，因为autoconf可以检测出构建系统的行为。

`config.host'文件包含了将要运行GCC的特殊系统的特定规则。该文件很少被用到。

`config.gcc'文件包含了GCC生成代码所针对特殊系统的特定规则。通常需要该文件。

每一个文件的开始部分都有一个设置的shell变量列表及其描述。

FIXME：介绍这些文件的内容，以及需要设置什么变量来控制build,host和target配置。

6.3.2.3 由configure创建的文件

我们在这里说明了在`gcc'目录下哪些文件将会由`configure'创建。 其它一些文件是在配置过程中生成的临时文件，而且在后续的构建过程中没有被用到，就不在这里介绍了。

- `Makefile'是由`Makefile.in'、`config'目录下的主机与目标机片断（see Chapter 19 [Makefile片段], page 403）`t-target'和`x-host'共同构造出来的，根据情况可能还会用到Makefile语言片断`language/Make-lang.in'。
- `auto-host.h'包含了由`configure'确定的主机信息。 如果主机和构建机不同，则`suto-build.h'也会被创建，其包含了关于构建机的信息。
- `config.status'脚本，可以用来运行并重新创建当前配置。
- `configargs.h'头文件，包含了传给`configure'来配置GCC的详细参数，以及使用的线程模式。
- `cstamp-h'作为时间戳使用。
- `fixinc/Makefile'由`fixinc/Makefile.in'创建。
- `gccbug', 用来报告GCC bugs的脚本，由`gccbug.in'创建。
- `intl/Makefile'由`intl/Makefile.in'创建。
- 如果语言相关的`config-lang.in'文件（参见 see Section 6.3.8.2 [The Front End `config-lang.in' File], page 56）设置了outputs，则在outputs中列出的文件也会被创建。

下列配置头文件是由Makefile通过使用`mkconfig.sh'，而不是直接通过`configure'来创建的。`config.h'，`bconfig.h'和`tconfig.h'都包含`xm-machine.h'头文件，如果存在的话，分别对应于host, build和target机器。 对于target，是一些配置头文件和一些定义；对于host和build机器，这些包括由`configure'生成的自动配置头文件。 其它配置头文件由`config.gcc'来确定。 它们也包含rtx，rtvec和tree的typedef定义。

- `config.h', 供运行在host机器上的程序使用。
- `bconfig.h', 供运行在build机器上的程序使用。
- `tconfig.h', 供运行在target机器上的程序和库使用。
- `tm_p.h', 包括了含有target机器上`.c' 文件中的函数原形的头文件`machine-protos.h'。 FIXME：为什么需要这样一个独立的头文件？

6.3.3 `gcc'目录下的构建系统

FIXME: 描述构建系统，包括什么阶段构建了什么。同时列出在构建过程中使用的， 不属于GCC本身的而因此没有在下面记录的，各种源文件（参见 Chapter 8 [Passes], page 69）。

6.3.4 Makefile工作目标

这些工作目标可以从`gcc'目录下获得：

`all` 这是缺省工作目标。它会根据你对build/host/target的配置， 来协调构建所有需要的事物。

`doc` 生成info格式的文档和man页。实质上它是调用了`make man'和`make info'。

`dvi` 生成DVI格式的文档。

`pdf` 生成PDF格式的文档。

`html` 生成HTML格式的文档。

`man` 生成man页。

`info` 生成info格式的页。

`mostlyclean`
删除构建编译器中创建的文件。

`clean` 由`make all'创建的所有其它文件。

`distclean` 由configure创建的所有文件。

`maintainer-clean`
Distclean，以及任何由其它文件生成的文件。 注意可能会需要，除了构建gcc所需的，额外的工具。

`srcextra` 在源目录生成不在CVS中存在的，但是属于发行tar包的文件。 由CVS源文件`gcc/java/parse.y'生成的`gcc/java/parse.c'是一个例子。

`srcinfo`
`srcman` 将info格式的和manpage文档复制到源目录，用于生成发行tar包。

`install` 安装gcc。

`uninstall` 删除安装的文件。

`check` 运行测试包。这将创建一个`testsuite'子目录， 其中具有各种包含测试结果的`.sum'和`.log'文件。 可以使用，例如`make check-gcc'的方式，来运行一个测试子集。 可以通过设定RUNTESTFLAGS为`.exp'文件名， （对于某些测试）后面可选的一个等号和一个文件通配符，来指定特定的测试。如：

```
make check-gcc RUNTESTFLAGS="execute.exp=19980413-*"
```

注意可能需要安装额外的工具，像TCL或dejagnu，来运行测试包。

开始编译GCC的顶层树不是GCC目录，而是使用一个复杂的Makefile来协调构建的各个步骤，包括自举（bootstrap）编译器，以及使用新的编译器来构建目标库。

当GCC被配置为本地配置时，make的缺省动作是执行完整的三阶段自举。 这意味着GCC将被构建三次，一次是使用本地编译器， 一次是使用刚由本地编译器构建的编译器，一次是使用第二次构建的编译器。 理论上，最后两次应该产生相同的结果，这可以使用`make compare'来检验。 每个阶段都被单独的配置和编译到独立的目录里， 以尽可能减少由于本地编译器和GCC之间的ABI不兼容所带来的问题。

如果做了改动，重建工作将还会从第一阶段开始，并且将改动贯穿三个阶段。 每个阶段都在它的构建目录下开始（如果先前曾被构建过）， 重建，并且复制到它的子目录。这将允许你，在修订了引起第二阶段构建崩溃的bug之后， 可以继续进行自举。这虽然对编译器没有提供像从头进行自举那样好的覆盖效果， 但却能保证新的代码在语法上是正确的（例如，没有错误使用GCC的扩展功能）， 并避免了不合逻辑的自举比较失败[1]。

其它由顶层可以获得的目标包括：

`bootstrap-lean`
: 类似bootstrap，除了各个阶段当不再需要的时候，将被移除。 这可以节省磁盘空间。

`bootstrap2`
`bootstrap2-lean`
: 只执行前两个阶段的自举。不像三阶段自举，这将不执行测试编译器运行正常的比较。注意使用“lean”的自举所需要的磁盘空间几乎是与阶段数目无关的。

`stageN-bubble (N = 1...4)`
: 使用适当的标记，来重建所有的阶段，只到N，并将按照上面描述的来传播改动。

`all-stageN (N = 1...4)`
: 假设阶段N已经被构建，使用适当的标记来重建它。这个很少用到。

`cleanstrap` 移除所有（`make clean'）并重建（`make bootstrap'）。

`compare` 比较阶段2和3的结果。这用于确保编译器运行正常，因为不管它本身是如何被编译的，都应该产生相同的目标文件。

`profiledbootstrap`
: 构建带有profile反馈信息的编译器。更多信息，参见 Section ``Building with profile feedback'' in Installing GCC。

`restrap` 重新启动自举，使得任何没有使用系统编译器构建的将被重建。

`stageN-start (N = 1...4)`
: 对于被自举的每个package，重命名目录，使得例如， 当使用N-1阶段的（stageN-1）GCC编译时，gcc指向N阶段的（stageN）GCC。

  如果需要测试或调试N阶段的GCC时，你将使用该目标。 如果只需要执行GCC（但不需要运行`make'，也不需要重建或运行测试包）， 你应该能够在`stageN-gcc'目录下工作。 这使得很容易进行并行的调试多个阶段。[2]。

`stage` 对于被自举的每个package，重定位它的构建目录来指示它的阶段。 例如，如果`gcc'目录指向阶段2的GCC，则执行该目标之后， 将被重命名为`stage2-gcc'。

---

[1] 除非编译器有bug，以及一些没有被修改的文件被错误编译了。 在这种情况下，最好使用make restrap。

[2] 习惯上，系统编译器也被称为0阶段（`stage0'）GCC。

如果在编译阶段2和阶段3的编译器时，想使用非缺省的GCC标记， 则在执行`make'时，在命令行上设置BOOT_CFLAGS。

通常，第一阶段只构建编写编译器的语言：通常是C，以及可能会有Ada。 如果你在调试一个其它的阶段2前端的错误编译（例如，Fortran前端）， 你可能想要在第一阶段也有其它语言的前端。如果这样， 在执行`make'时在命令行中设置STAGE1_LANGUAGES。

例如，在前述情况中，要调试由阶段1编译器造成的Fortran前端的错误编译， 可能需要一个命令如

```
make stage2-bubble STAGE1_LANGUAGES=c,fortran
```

另外，可以使用每个语言的目标，来构建和测试没有在阶段1启用的语言。 例如，make f951将在stage1构建目录下构建一个Fortran编译器。

6.3.5 在`gcc'目录下的库源文件和头文件

FIXME：列出并说明在`gcc'目录下的不被构建到GCC可执行程序中， 而是作为运行时库的一部分或者目标文件的所有C源文件和头文件。 例如`crtstuff.c'和`unwind-dw2.c'。 关于`ginclude'目录的详细介绍，参见 Section 6.3.6 [GCC安装的头文件], page 51。

6.3.6 GCC安装的头文件

通常，GCC希望所使用的头文件大部分由系统C库提供。但是在需要的时候， GCC会修改那些头文件以便能够工作，并且会安装一些需要单独实现的头文件。 这些头文件被安装在`libsubdir/include'下。 GCC也安装了些非C运行时库的头文件；这些没有在这里列出。 （FIXME：在某处给出这些的文档。）

GCC所安装一些头文件来自`ginclude'目录下。 这些头文件`iso646.h', `stdarg.h', `stdbool.h', `stddef.h'将被安装在`libsubdir/include'下，除非通过在目标Makefile片段中（参见 Section 19.1 [目标机片段], page 404）设置USER_H来改变。

除了这些头文件，以及那些GCC为了能够正常工作而修改的系统头文件， `libsubdir/include'下还可能安装了其它头文件。 `config.gcc'可以设置extra_headers， 用来指出将`config'目录下的其它头文件安装在一些系统上。

GCC使用`ginclude/float.h'，来安装自己版本的<float.h>， 并通过拷贝命令行选项来改变浮点数的表示。

GCC还安装它自己版本的<limits.h>；它是从`glimits.h'中生成的， 以及`limitx.h'和`limity.h'，如果系统还具有自己版本<limits.h>。 （GCC提供自己的头文件，是因为ISO C独立实现的需要， 但是该头文件还需要包含系统头文件， 是因为其它标准像POSIX指定的额外的值在<limits.h>中有定义）系统的<limits.h>头文件通过`libsubdir/include/syslimits.h'来使用， 它是从`gsyslimits.h'复制过来的，如果其不需要修改为与GCC一起工作； 如果需要修改，`syslimits.h'为修改后的副本。

当`config.gcc'设置use_gcc_tgmath为yes时， GCC还将安装<tgmath.h>。

6.3.7 构建文档

主要的GCC文档使用了Texinfo格式的手册形式。这些按照info格式安装； DVI版本的可以通过`make dvi'来生成，PDF版本的通过`make pdf'， 以及HTML版本的通过make html。另外，一些man页通过Texinfo手册来生成， 还有一些其它各种文档的文本文件， 以及运行时库具有`gcc'目录之外的它们自己的文档。 FIXME：列出运行时库的文档。

6.3.7.1 Texinfo手册

GCC整体的以及C和C++前端的手册，在文件`doc/*.texi'中。 其它前端自己的手册在文件`language/*.texi'中。 通用文件`doc/include/*.texi'被提供用来在多个手册中包含； 下列文件在`doc/include'中：

`fdl.texi' GNU自由文档许可。

`funding.texi'
"资助自由软件"章节。

`gcc-common.texi'
手册的通用定义。

`gpl.texi'
`gpl_v3.texi'
GNU通用公共许可。

`texinfo.tex'
与GCC手册一起工作的`texinfo.tex'副本。

DVI格式的手册通过`make dvi'生成， 其使用了texi2dvi（通过Makefile宏$(TEXI2DVI)）。 PDF格式的手册通过`make pdf'生成， 其使用了texi2pdf（通过Makefile宏$(TEXI2PDF)）。 HTML格式的手册通过make html生成。 info手册通过`make info'生成（作为bootstrap的一部分运行）； 这个是在源目录中生成手册，使用makeinfo， 通过Makefile宏$(MAKEINFO)，它们被包含在发现版本中。

手册在GCC网站上也有提供，HTML和PostScript格式的。 这是通过脚本`maintainer-scripts/update_web_docs'生成的。 在线提供的每个手册都必须在那个文件中的MANUALS定义中列出； 文件`name.texi'必须在源树中只出现一次， 输出手册必须与源文件具有相同的名字。（但是，手册中包含的其它不是根文件的Texinfo文件， 其名字可能会在源树中出现多次。）手册文件`name.texi'应该只包含它目录下或者`doc/include'下的其它文件。HTML手册将通过`makeinfo --html'生成， PostScript手册将通过texi2dvi和dvips， PDF手册通过texi2pdf。为了使生成的在线手册可以工作， 所有属于手册一部分的Texinfo文件都必须放入CVS，即使它们是被生成的文件。

安装手册，`doc/install.texi'，也在GCC网站上有提供。 HTML版本通过脚本`doc/install.texi2html'生成。

6.3.7.2 生成Man Page

出于用户的需求，除了完整的Texinfo手册以外，还提供了从那些手册中提取的man页。 这些man页使用`contrib/texi2pod.pl'和pod2man， 从Texinfo手册中生成。（g++,`cp/g++.1'的man页， 只是包含了一个对`gcc.1'的`.so'引用， 但所有其它man页是从Texinfo手册生成的。）

因为许多系统可能没有安装生成man页所需的工具， 所以它们只有当`configure'脚本检测到新近安装了足够的工具时才会被生成， 并且Makefile允许在生成man页失败时，而不会终止构建。 Man页也包含在发行版本中。它们在源目录下生成。

Texinfo文件中起始于`@c man'的Magic注释， 用来控制Texinfo文件中哪些部分放到man页中。`texi2pod.pl'只支持Texinfo的一个子集，并且当生成新的man页时， 可能需要增加对Texinfo特征的更多支持。 为了提高man页输出， 在`doc/include/gcc-common.texi'中提供了一些`texi2pod.pl'可以理解的特定的Texinfo宏：

@gcctabopt 选项表，以`@table @gcctabopt'形式使用， 用于`@code'的打印输出效果比`@option'的好， 但是对于man页，却想要不同的效果的情况。

`@gccoptlist`
: 用于手册中的选项列表。

`@gol`
: 用在`@gccoptlist'中每行的结尾。 这可以避免由于不同的Texinfo格式化对`@gccoptlist'宏的不同处理所带来的问题。

FIXME: 更加详细的描述`texi2pod.pl'输入语言和magic注释。

6.3.7.3 其它文档

除了GCC安装的正式文件以外，还有一些其它杂项文档的文本文件：

`ABOUT-GCC-NLS'
: GCC本地语言支持的笔记。FIXME：这应该为手册的一部分，而不是单独的文件。

`ABOUT-NLS'
: 自由翻译项目的笔记。

`COPYING'
: GNU通用公共许可。

`COPYING.LIB'
: GNU宽松通用公共许可。

`*ChangeLog*'
`*/ChangeLog*'
: GCC各部分的更新日志。

`LANGUAGES'
: GCC前端接口的一些改动的详细情况。 FIXME：该文件中信息应该为本手册中前端接口通用文档的一部分。

`ONEWS'
: 旧版本GCC中的新特征信息。（对于近期版本，信息在GCC网站上。）

`README.Portability'
: 关于在GCC中写代码时的可移植性的信息。 FIXME：为什么这部分不作为本手册中的GCC代码约定？

FIXME: 记载子目录中的这样的文件， 起码像`config', `cp', `objc', `testsuite'。

6.3.8 语言前端剖析

GCC的语言前端具有下列部分：

- `gcc'下的目录`language'包含了那个前端的源文件。 详情参见Section 6.3.8.1 [前端`language'目录], page 54。
- 在`gcc/doc/install.texi'的所支持的语言列表中对该语言的记载。
- 在`gcc/doc/install.texi'文档中， 记载了在哪个名字下能够被`--enable-shared=package'识别的语言的运行时库。
- 在文档`gcc/doc/install.texi'中对构建前端所需的任何前提条件的记载。
- 在`gcc/doc/contrib.texi'中那个前端的贡献者的详情。 如果详情是在那个前端自己的手册中， 则在`contrib.texi'中应该有一个指向那个手册列表的链接。
- 在`gcc/doc/frontends.texi'中关于对那个语言的支持的信息。
- 在`gcc/doc/standards.texi'中关于那个语言的标准和前端对其的支持的信息。 这可能是一个指向前端自己手册中这些信息的一个链接。
- 在`gcc/doc/invoke.texi'中， 那个语言的源文件后缀详请和所支持的`-x lang'选项。

- 在`gcc.c'的default_compilers里，那个语言的源文件后缀的入口项。
- 比较适合的测试包，可能在`gcc/testsuite'下或者运行时库目录下。 FIXME：在某处记载如何写测试包harnesses。
- 在`gcc'目录外，可能有一个该语言的运行时库。FIXME：详细描述一下。
- 在`gcc/doc/sourcebuild.texi'中关于任何运行时库的目录的详情。

如果前端被增加到GCC CVS库中，则还需要下列：

- 至少有一个关于那个前端和运行时库的bug的Bugzilla component。 该类别需要在`gcc/gccbug.in'中有记载，同时被添加到Bugzilla数据库中。
- 通常，在`MAINTAINERS'中列出那个前端的一个或多个维护者。
- 在GCC网站上的`index.html'和`frontends.html'中的一些记载， 以及在`readings.html'上的任何相关链接。 （不是GCC官方部分的前端还可以在`frontends.html'中列出，并带有相关链接。）
- 在`index.html'中的一条消息， 并且可能在gcc-announce@gcc.gnu.org邮件组中的一个声明。
- 前端手册应该在`maintainer-scripts/update_web_docs' (see Section 6.3.7.1 [Texinfo手册], page 52 ) 中有记载， 并且来自`onlinedocs/index.html'的在线手册应该有相应链接。
- 在GCC包含该前端之前的任何旧的发行版本或CVS库版本， 应该可以在GCC FTP站点ftp://gcc.gnu.org/pub/gcc/old-releases/上获得。
- 发行版和快照脚本`maintainer-scripts/gcc_release'应该被更新为可以生成该前端的合适的tar包。 相关的`maintainer-scripts/snapshot-README'和`maintainer-scripts/snapshot-index.html'文件应该被更新为可以列出该前端的tar包和diffs。
- 如果该前端包含自己的版本文件，其包含了当前日期， 则`maintainer-scripts/update_version'应该相应的被更新。

6.3.8.1 前端`language'目录

前端`language'目录包含了该前端的源文件（不过对于运行时库除外， 它们应该放到`gcc'目录外面），包括文档或者随着前端一起构建的辅助程序。 有些文件是专用的，并且编译器的其它部分会依赖它们的名字：

`config-lang.in'
: 所有的language子目录都需要这个文件。 详情参见 Section 6.3.8.2 [前端`config-lang.in'文件], page 56。

`Make-lang.in'
: 所有的language子目录都需要这个文件。 它包含了目标lang.hook（lang是在`config-lang.in'中对language的设置），即下面的hook的值，以及构建这些目标的其它Makefile规则(如果需要的话， 可以使用在`config-lang.in'中通过outputs指定的其它Makefile， 但是不赞成这种方式)。 它还向变量lang_checks增加了任何可以使用`gcc/Makefile.in'中的 标准规则的测试包目标。

  all.cross
  start.encap
  rest.encap FIXME: 确切的描述这些目标。

  tags 在源树的语言子目录中构建一个etags `TAGS'文件。

`info` 在构建目录下构建前端的info文档。 这个目标只被`make bootstrap'在有合适版本的`makeinfo`时调用， 所以不需要进行检查，并且如果错误发生时，应该失败。

`dvi` 在构建目录下构建前端的DVI文档。这应该使用`$(TEXI2DVI)`， 以及合适的指向要包含文件目录的`-I'参数，来完成。

`pdf` 在构建目录下构建前端的PDF文档。这应该使用`$(TEXI2PDF)`， 以及合适的指向要包含文件目录的`-I'参数，来完成。

`html` 在构建目录下构建前端的HTML文档。

`man` 在构建目录下从Texinfo手册中构建前端的man页（参见Section 6.3.7.2 [生成Man Page], page 52）。 该目标只在所需的工具可用时才被调用，当应该忽视错误从而不会在错误发生时停止构建； man页是可选的，并且所涉及到的工具可能被安装in a broken way。

`install-common`
安装前端的所有部分， 除了在`config-lang.in'中列出的`compilers`可执行程序以外。

`install-info`
安装前端的info文档，如果源目录中存在的话。 该目标应该与要安装的info文件有依赖关系。

`install-man`
安装前端的man页。该目标应该能够忽略错误。

`srcextra` 将它的依赖复制到源目录中。通常用于生成的文件，像Bison输出文件，其不存在于CVS中， 但应该放到任何的发行tar包中。 该目标将在自举过程中当`--enable-generated-files-in-srcdir'被指定为一个`configure'选项时被执行。

`srcinfo`
`srcman` 将它的依赖复制到源目录中。 该目标将在自举过程中当`--enable-generated-files-in-srcdir'被指定为一个`configure'选项时被执行。

`uninstall` 卸载通过安装编译器所安装的文件。目前还不支持，所以不要做任何事情。

`mostlyclean`
`clean`
`distclean`
`maintainer-clean`
标准GNU`*clean'目标的语言部分。关于标准目标的详情， 参见Section ``Standard Targets for Users'' in GNU Coding Standards。 对于GCC，`maintainer-clean`应该删除所有在源目录下生成的没有放入CVS的文件， 但不要删除任何放入CVS的文件。

`Make-lang.in' must also define a variable `lang_OBJS` to a list of host object files that are used by that language.

`lang.opt'` 该文件注册了前端所接受的命令行中的选项开关集，以及它们的`--help'文本。 参见Chapter 7 [选项], page 67。

`lang-specs.h'
该文件提供了在`gcc.c'中的default_compilers的入口，用于覆盖当该语言的编译器没有安装时报错的缺省行为。

`language-tree.def'
该文件，不需要存在，定义了任何语言特定的树代码。

6.3.8.2 前端`config-lang.in'文件

每一个语言子目录都包含一个`config-lang.in'文件。另外主目录下包含一个`c-config-lang.in'，里面含有C语言的限制信息。这个文件是一个shell脚本，可以用来定义一些描述语言的变量。

language 必须定义，它给出了语言的名字用于作为`--enable-languages'的参数或其它用途。

lang_requires
如果定义，该变量列出了（由空格分开）该前端需要被支持的除了C的语言前端（使用所设置的language名字）。例如，Java前端依赖于C++前端，所以要设为`lang_requires=c++'。

subdir_requires
如果定义，该变量列出了（由空格分开）该前端需要存在的除了C的前端目录。例如，Objective-C++前端使用了C++和Objective-C前端的源文件，所以要设为`subdir_requires="cp objc"'。

target_libs
如果定义，该变量列出了（由空格分开）在顶层`Makefile'中为该语言构建运行时库的目标，例如target-libobjc。

lang_dirs 如果定义，该变量列出了（由空格分开）只有构建该前端时才应被配置的除了运行时库之外的顶层目录（与`gcc'并列）。

build_by_default
如果定义为`no'，则该语言前端只有在`--enable-languages'中指定参数时才被构建。否则，会按照缺省方式来构建前端，并且受到`configure.ac'中特定逻辑的影响（正如目前，如果Ada编译器没有安装则禁止Ada前端）。

boot_language
如果定义为`yes'，则该前端在自举阶段1中会被构建。这个只与用它们自己的语言写的前端相关。

compilers 如果定义，则为空格分隔的编译器可执行程序列表，其将被驱动调用运行。这里的名字将以`\$(exeext)'结尾。

outputs 如果定义，则为空格分隔的文件列表，其将被`configure'通过值替换来生成。这种机制可以用来从`language/Makefile.in'中创建一个`language/Makefile'文件，但不赞成这样，应该从单个`gcc/Makefile'中来构建所有。

gtfiles 如果定义，则为空格分隔的文件列表，其将被gengtype.c扫描来生成垃圾搜集表，和该语言的程序。这不包括所有前端公用的文件。参见Chapter 22 [类型信息], page 407。

6.3.9 目标机后端剖析

GCC中与目标体系结构相关的后端包括下面的部分：

- 位于`gcc/config'下面的目录`machine'，包括一个机器描述文件`machine.md'（参见Chapter 16 [机器描述], page 201），头文件`machine.h'和`machine-protos.h'，以及一个源文件`machine.c'（参见Chapter 17 [目标描述的宏和函数], page 282），可能会有一个目标Makefile片段`t-machine'（参见Section 19.1 [目标Makefile片段], page 404），以及其它文件。可以在`config.gcc'中通过显示指出来改变文件的缺省名字。
- 如果需要的话，在`machine'目录下会有一个文件`machine-modes.def'，包含了附加的机器模式用来表示条件代码。详情参见Section 17.16 [条件代码], page 344。
- 在`machine'目录下，有一个可选的`machine.opt'文件，包含了一个目标特定的选项列表。你也可以通过在`config.gcc'中使用extra_options变量来增加其它的选项文件。参见Chapter 7 [选项], page 67。
- 在`config.gcc'（参见Section 6.3.2.2 [`config.gcc' 文件], page 48）中，该目标体系结构与系统相关的条目。
- 在`gcc/doc/invoke.texi'中关于该目标所支持的任何命令行选项的文档（参见运行时目标规定）。这意味着在选项的汇总表中以及单个选项的详细描述中都要有相关的条目。
- 在`gcc/doc/extend.texi'中关于所支持的目标特定属性的文档（参见__attribute__的目标特定用法），包括在哪些地方已经被其它一些目标所支持的相同属性，以及哪些在手册中已经被列举。
- 在`gcc/doc/extend.texi'中关于支持的所有目标特定pragma的文档。
- 在`gcc/doc/extend.texi'中关于支持的所有目标特定built-in函数的文档。
- 在`gcc/doc/extend.texi'中关于支持的所有目标特定格式检测风格的文档。
- 在`gcc/doc/md.texi'中关于所有目标特定的约束字母的文档（参见Section 16.8.5 [机器约束], page 217）。
- 在`gcc/doc/contrib.texi'中记录下由谁提供的目标支持。
- 在`gcc/doc/install.texi'中关于该目标体系结构支持的所有目标三元参数，给出该目标安装所需要的特殊注解或者注明没有。
- 可能在`gcc'目录之外会有其它运行库的支持。FIXME：关于这方面的参考文档。这个工作需要安装libstdc++移植手册或者将其作为本手册的一章。

如果是添加到GCC官方CVS库中的后端，还需要下面的内容：

- 在GCC网站上的`readings.html'中有一条关于目标体系结构的记录，并带有任何相关的链接。
- 在GCC网站上的`backends.html'中有关于后端和目标体系结构的详细特性。
- 在GCC网站上的`index.html'中有一条关于提供目标体系结构支持的消息公告。
- 通常的，会在`MAINTAINERS'中列出该目标的一个或多个维护者。虽然有些现存的体系结构可能没有被维护了，但是通常是不会为没有维护者的目标提供支持的。

6.4 测试包

GCC包含了几个测试包用来确保编译器的质量。大部分运行时库和语言前端在GCC中都有测试包。目前在这里只讲述了C语言的测试包；FIXME：介绍其它的测试包。

6.4.1 测试包代码中使用的习惯用法

通常，C 测试用例以`-n.c'结尾，并且从`-1.c'开始，以便于以后增加其它具有类似名字的测试用例。如果是测试一些明确定义的特征，则测试的名字应该指出这个特征，例如`feature-1.c'。如果不是测试一个明确定义的特征，而只是检验在编译器中存在的，并且是在GCC bug库中归档的bug，

则可以使用`prbug-number-1.c'这样的名字形式。 否则（对于在GCC bug库中没有归档的各种bug），测试用例根据它们被添加的日期来命名，这种情况在以前更加常见。 这样使人们能够一眼看出一个测试失败是由于一个新发现并且还没有被修复的bug造成的， 还是由于一个回退错误造成的，但它并没有给出关于bug的其它信息， 以及从哪里可以找到相关的讨论。一些其它语言的测试包也遵守类似的惯例。

在`gcc.dg'测试包中，通常需要测试一个错误确实是硬件错误， 而不只是一个警告——例如，在C标准中的violatile限定， 在有`-pedantic-errors'的时候必须为一个错误。为此， 可以使用下面的习惯用法，其中第一行为产生错误的文件的行line。

```
/* { dg-bogus "warning" "warning in place of error" } */
/* { dg-error "regexp" "message" { target *-*-* } line } */
```

可能需要检查一个表达式为整数常量表达式，并且具有一个特定的值。 要检查E具有值V，可以使用类似下面的习惯用法：

```
char x[((E) == (V) ? 1 : -1)];
```

在`gcc.dg'测试中，__typeof__有时被用于表达式类型的断言。 例如，可以参见`gcc.dg/c99-condexpr-1.c'。 更加巧妙的用法依靠了C标准中条件表达式类型的确切规则； 例如，可以参见`gcc.dg/c99-intconst-1.c'。

如果能够测试优化被做的很适当会很有帮助。这并不能在所有情况下都能做到， 但对于可以使得代码被优化掉的情况（例如，流分析或别名分析应该显示那样的代码不会被调用），或者函数将不被调用，因为它们已经被扩展为内建的函数时，是可以做到的。 这样的测试在`gcc.c-torture/execute'中。 在将要被优化掉的代码的地方，可以插入一个像link_failure () 这样的对一个不存在的函数的调用；并且还需要如下定义，

```
#ifndef __OPTIMIZE__
void
link_failure (void)
{
 abort ();
}
#endif
```

从而使得当测试在没有优化而运行时，连接依然成功。 当对一个内建函数的所有调用都已经被优化，并且不会剩下对函数的非内建版本的调用时， 那个函数可以定义为static，并且调用abort () （虽然将函数声明为静态的可能不会在所有的目标上工作）。

所有测试用例都必须是可移植的。 目标特定的测试用例必须具有适当的代码来避免在不支持的系统上引起失败； 不幸的是，这种机制随目录有所不同。

FIXME: 讨论一下非C的测试包。

6.4.2 DejaGnu测试中使用的指令

测试指令出现在测试源文件的注释中，并且起始于dg-。 其中一些是在DejaGnu中定义的，其它的是局限于GCC测试包自己的。

测试中测试指令出现的顺序很重要： 局限于GCC的指令有时会覆盖DejaGnu的指令使用的信息，并且其对GCC指令一无所知，所以DejaGnu指令必须在GCC指令之前。

个别测试指令包含了选择器， 其通常由关键字target或xfail打头。 一个选择器为：一个或多个目标三元组，可能包含通配符； 单个的有效目标关键字；或者一个逻辑表达式。 取决于上下文，选择器指定了是否测试被跳过并报告为不支持， 或者预期为失败。使用`*-*-*'来匹配任何目标。 有效目标关键字在GCC测试包的`target-supports.exp'中定义。

选择器表达式出现在大括号中， 并且使用单个的逻辑操作符：`!',`&&', 或`||'。 操作数为另一个选择器表达式，一个有效目标关键字， 单个的目标三元组，或者一个双引号或大括号包裹的目标三元组列表。 例如：

```
{ target { ! "hppa*-*-* ia64*-*-*" } }
{ target { powerpc*-*-* && lp64 } }
{ xfail { lp64 || vect_no_align } }
```

{ dg-do do-what-keyword [{ target/xfail selector }] }

do-what-keyword指定了测试如何被编译，以及是否被执行。其为：

preprocess 使用`-E'编译，从而只运行预处理器。

compile 使用`-S'编译，从而生成汇编代码文件。

assemble 使用`-c'编译，从而生成可重定位的目标文件。

link 编译，汇编，并连接，从而生成一个可执行文件。

run 生成并运行一个可执行文件，并期望其返回的退出代码为0。

缺省情况为compile。 可以通过在那些测试的.exp文件中重定义 dg-do-what-default来覆盖这个值。

如果指令包含了可选的`{ target selector } '， 则除非目标系统被包含在目标三元组列表中，或者匹配有效目标关键字， 否则测试将被跳过。

如果运行了`do-what-keyword'并且指令包含了可选的`{ xfail selector }'并且选择器匹配，则测试期望为失败。 对于`do-what-keyword'的其它值，xfail子句会被忽略； 那些测试可以使用指令dg-xfail-if。

{ dg-options options [{ target selector }] }

该DejaGnu指令提供了一个编译器选项列表， 用来在目标系统匹配selector的时候被使用， 以替换这个测试集的缺省选项。

{ dg-add-options feature ... }

增加任何访问特定特征所需要的编译器选项。 该指令对使用缺省方式启用特征或者根本不提供特征的目标不做任何事情。 其必须在所有dg-options指令之后。

支持的feature为：

c99_runtime

目标的C99运行时（包括头文件和库）。

mips16_attribute

mips16函数属性。只有MIPS目标支持该特征，并且是在特定的模式下。

{ dg-timeout n [{target selector }] }

将编译和执行测试程序的时间限制，设置为指定的秒数。

{ dg-timeout-factor x [{ target selector }] }

将测试程序的编译和执行的通常时间限制，乘以指定的浮点因子。 通常的时间限制，以妙为单位，按照下列顺序来查找：

- 由之前在测试程序中dg-timeout指令定义
- 由测试程序集定义的变量tool_timeout
- 在目标板子上（target board）设置的gcc,timeout
- 300

{ dg-skip-if comment { selector } { include-opts } { exclude-opts } }

如果测试系统被包含在selector中， 并且在include-opts中的每个选项都在用于编译测试的选项集中， 并且在exclude-opts中的每个选项都不在用于编译测试的选项集中，则跳过该测试。

空的include-opts列表使用`"*"'， 空的exclude-opts列表使用`""'。

{ dg-xfail-if comment { selector } { include-opts } { exclude-opts } }

如果符合条件（与dg-skip-if相同），则期望测试失败。

{ dg-xfail-run-if comment { selector } { include-opts } { exclude-opts } }

Expect the execute step of a test to fail if the conditions (which are the same as for dg-skip-if) and dg-xfail-if) are met.

{ dg-require-support args }

如果目标没有提供需要的支持，则跳过测试； 实际的指令参见GCC测试包中的`gcc-dg.exp'。 这些指令必须出现在任何dg-do指令之后， 并且是任何dg-additional-sources指令之前。 它们至少需要一个参数，如果特定的程序不检查参数，则其可以为空字符串。

{ dg-require-effective-target keyword }

如果测试目标，包括目前的multilib标记，没有相应的有效目标关键字，则跳过测试。 该指令必须出现在任何dg-do指令之后， 并且是任何dg-additional-sources directive 指令之前。

{ dg-shouldfail comment { selector } { include-opts } { exclude-opts } }

如果条件满足（与dg-skip-if相同）， 则期望测试可执行程序返回非零的退出状态。

{ dg-error regexp [comment [{ target/xfail selector } [line] }]] }

该DejaGnu指令出现在期望能获得一个错误消息的源行中，或者指定源行与消息相关联。 如果那个line没有产生消息，或者如果消息文本不匹配regexp， 则检测失败并且comment被包括在FAIL消息中。 检测并不查看字符串`"error"'，除非它是regexp的一部分。

{ dg-warning regexp [comment [{ target/xfail selector } [line] }]] }

该DejaGnu指令出现在期望能获得一个警告消息的源行中，或者指定源行与消息相关联。 如果那个line没有产生消息，或者如果消息文本不匹配regexp， 则检测失败并且comment被包括在FAIL消息中。 检测并不查看字符串`"warning"'，除非它是regexp的一部分。

{ dg-message regexp [comment [{ target/xfail selector } [line] }]] }

line被期望能获得一个错误或者警告消息。如果如果那个行没有产生消息， 或者如果消息文本不匹配regexp， 则检测失败并且comment被包括在FAIL消息中。

{ dg-bogus regexp [comment [{ target/xfail selector } [line] }]] }

该DejaGnu指令出现在一个不应该获得匹配regexp的消息的源行， 或者指定源行与一个虚假消息相关联。其通常和`xfail'一起使用， 来指示消息为一个特定目标集的已知问题。

{ dg-excess-errors comment [{ target/xfail selector }] }

该DejaGnu指令指示该测试由于编译器消息没有被`dg-error', `dg-warning'或`dg-bogus'处理，从而期望失败。 对于这个指令，`xfail'与`target'具有相同效果。

{ dg-output regexp [{ target/xfail selector }] }

该DejaGnu指令将regexp与测试执行程序写入`stdout'和`stderr'的组合输出作比较。

{ dg-prune-output regexp }
删除测试输出中匹配regexp的消息。

{ dg-additional-files "filelist" }
指定必须复制到编译器运行的系统上的额外文件，不是源文件。

{ dg-additional-sources "filelist" }
指定在编译命令行中出现在主测试文件之后的额外的源文件。

{ dg-final { local-directive } }
该DejaGnu指令被放在源文件中任何地方的注释中，并且在测试编译和运行后被处理。多个`dg-final'命令将按照它们在源文件中出现的顺序来处理。

GCC测试包定义了下列指令用于dg-final中。

cleanup-coverage-files
移除该测试生成的覆盖数据文件。

cleanup-repo-files
移除该测试使用`-frepo'生成的文件。

cleanup-rtl-dump suffix
移除该测试生成的匹配suffix的RTL转储文件。

cleanup-tree-dump suffix
移除该测试生成的匹配suffix的树转储文件。

cleanup-saved-temps
移除该测试使用`--save-temps'生成的文件。

scan-file filename regexp [{ target/xfail selector }]
如果regexp匹配filename文本，则测试通过。

scan-file-not filename regexp [{ target/xfail selector }]
如果regexp不匹配filename文本，则测试通过。

scan-hidden symbol [{ target/xfail selector }]
如果symbol在测试的汇编输出中被定义为隐含符号，则测试通过。

scan-not-hidden symbol [{ target/xfail selector }]
如果symbol在测试的汇编输出中没有被定义为隐含符号，则测试通过。

scan-assembler-times regex num [{ target/xfail selector }]
如果regex匹配在测试的汇编输出中的确切num次数，则测试通过。

scan-assembler regex [{ target/xfail selector }]
如果regex匹配测试的汇编输出的文本，则测试通过。

scan-assembler-not regex [{ target/xfail selector }]
如果regex不匹配测试的汇编输出的文本，则测试通过。

scan-assembler-dem regex [{ target/xfail selector }]
如果regex匹配测试的汇编输出的demangled文本，则测试通过。

scan-assembler-dem-not regex [{ target/xfail selector }]
如果regex不匹配测试的汇编输出的demangled文本，则测试通过。

`scan-tree-dump-times` regex num suffix [{ target/xfail selector }]
如果regex在具有suffix后缀的转储文件中出现确切的num次数， 则测试通过。

`scan-tree-dump` regex suffix [{ target/xfail selector }]
如果regex匹配具有suffix后缀的转储文件的文本，则测试通过。

`scan-tree-dump-not` regex suffix [{ target/xfail selector }]
如果regex不匹配具有suffix后缀的转储文件的文本，则测试通过。

`scan-tree-dump-dem` regex suffix [{ target/xfail selector }]
如果regex匹配具有suffix后缀的转储文件的demangled文本，则测试通过。

`scan-tree-dump-dem-not` regex suffix [{ target/xfail selector }]
如果regex不匹配具有suffix后缀的转储文件的demangled文本，则测试通过。

`output-exists` [{ target/xfail selector }]
如果编译器输出文件存在，则测试通过。

`output-exists-not` [{ target/xfail selector }]
如果编译器输出文件不存在，则测试通过。

`run-gcov` sourcefile
在gcov测试中检查行数。

`run-gcov` [branches] [calls] { opts sourcefile }
在gcov测试中除了行数外，还检查分支和/或调用数。

6.4.3 Ada语言测试包

Ada测试包包括了来自ACATS 2.5测试包的可执行测试， 其在http://www.adaic.org/compilers/acats/2.5上可公开获得。

这些测试被集成在GCC测试包中，在`gcc/testsuite/ada/acats'目录下， 并且如果配置GCC时设置了Ada语言，则运行make check时会自动执行。

你还可以单独运行Ada测试包，使用make check-ada，或测试的子集， 通过指定运行那些章节，例如

```
$ make check-ada CHAPTERS="c3 c9"
```

测试通过目录组织起来，每个目录对应于Ada参考手册的一个章节。 所以，例如c9对应于第9章，关于语言任务特征的。

还有一个额外的章节叫做`gcc'，包含了用来创建新的可执行测试的模版。

测试使用两个sh脚本来运行：`run_acats'和`run_all.sh'。 如果要使用模拟器或者交叉目标来运行测试， 参见`run_all.sh'顶部的small customization部分。

这些测试使用构建树来运行：它们可以在没有执行make install的情况下被运行。

6.4.4 C语言测试包

GCC在`gcc/testsuite'目录下包含下列C语言测试包：

`gcc.dg'　其包含了C编译器特定特征的测试，使用了较为现代的`dg' harness。 如果可能的话，对各种编译器特征的正确性的测试应该放在这里。

魔术注释（Magic comments）决定了文件是否被预处理，编译，连接或运行。 在这些测试中，错误和警告消息文本用来跟在注释中给出的预期文本或正规表达式作比较。 这些测试使用选项`-ansi -pedantic'来运行，除非给出了其它选项。 除了下面标注的以外，它们使用多个优化选项来运行。

`gcc.dg/compat'
该子目录包含了使用`compat.exp'的二进制兼容性测试， 并且是使用语言无关的支持（参见Section 6.4.8 [对测试二进制兼容性的支持], page 65）。

`gcc.dg/cpp'
该子目录包含了预处理器的测试。

`gcc.dg/debug'
该子目录包含了调试格式的测试。该子目录下的测试用于编译器支持的每个调试格式。

`gcc.dg/format'
该子目录包含了`-Wformat'格式检测的测试。 该目录下的测试分别使用和不使用`-DWIDE'来运行。

`gcc.dg/noncompile'
该子目录包含了不被编译并且不需要任何特定编译选项的测试。 它们使用多个优化选项来运行，因为有时编译器的优化会使得代码崩溃。

`gcc.dg/special'
FIXME: 对其进行描述。

`gcc.c-torture'
这包含了历史上很容易出问题的特定代码片断。这些测试使用多个优化选项来运行， 所以测试只在某些优化级别出问题的特征也属于这里。 这还包含了检测特定优化发生的测试。 或许值得去将正确性测试从代码质量测试中完全分开出来，但目前还没有做。

`gcc.c-torture/compat'
FIXME: 对其进行描述。

该目录应该不会被用于新的测试。

`gcc.c-torture/compile'
该测试包包含了应该编译，但不需要连接和运行的测试用例。 这些测试用例使用多个不同优化选项组合来编译。所有的警告都被禁止， 所以如果你想测试编译器警告是否存在，则这个目录不适合。虽然可以设置特定的选项， 并且测试不是用于特定的平台，但大多数这些测试用例不应该包含平台依赖。 FIXME：论述如何使用像NO_LABEL_VALUES和STACK_SIZE这样的定义。

`gcc.c-torture/execute'
该测试包包含了应该被编译，连接和运行的测试用例；否则跟`gcc.c-torture/compile'相同的注释将会使用。

`gcc.c-torture/execute/ieee'
这包含了IEEE浮点特定的测试。

`gcc.c-torture/unsorted'
FIXME: 对其进行描述。

该目录应该不会被用于新的测试。

`gcc.c-torture/misc-tests'
该目录包含了需要特殊处理的C测试。其中的一些测试具有单独的expect文件，另一些则共享特殊用途的expect文件：

`bprob*.c' 使用`bprob.exp'来测试`-fbranch-probabilities'，并且是使用通用的，语言无关的框架（参见Section 6.4.7 [对测试profile指导的优化的支持], page 65）。

`dg-*.c' 使用`dg-test.exp'来测试测试包本身。

`gcov*.c' 使用`gcov.exp'来测试gcov输出，并且是使用语言无关的支持（参见Section 6.4.6 [对测试gcov的支持], page 64）。

`i386-pf-*.c'
使用`i386-prefetch.exp'来测试i386特定的对数据预提取的支持。

FIXME: 合并到`testsuite/readme.gcc'中，并进一步论述测试用例的格式和魔术注释。

6.4.5 Java库测试包

运行时测试通过在构建树的`target/libjava/testsuite'目录下运行`make check'来执行。额外的运行时测试可以放在这个测试包中。

Mauve测试包还覆盖了libgcj的核心包的回归测试。Mauve项目为Java类库开发了测试。这些测试作为libgcj测试的一部分来运行，通过在`libjava/testsuite/libjava.mauve/mauve的libjava测试包源中放置Mauve树，或者当调用`make'时，通过`make MAUVEDIR=~/mauve check'来指定该树的位置。

为了检测回退，`mauve.exp'中有一个机制用来比较一个测试的失败与在源层次结构`libjava/testsuite/libjava.mauve/xfails'中的期望的失败列表。当向Mauve中增加一个新的失败测试时，或者修改libgcj中的bug造成了Mauve测试失败时，需要更新该文件。

我们鼓励开发者将测试用例贡献给Mauve。

6.4.6 对gcov测试的支持

expect文件`gcov.exp'，提供了对测试gcov和检查分支profile是否产生预期值的语言无关的支持。gcov测试还依赖于`gcc.dg.exp'的程序，来编译和运行测试程序。一个典型的gcov测试包含下列在注释中的DejaGnu命令：

```
{ dg-options "-fprofile-arcs -ftest-coverage" }
{ dg-do run { target native } }
{ dg-final { run-gcov sourcefile } }
```

对gcov输出的检测可以包括行数，分支百分比和调用返回百分比。所有这些检测都由在测试源文件中注释里出现的命令来请求。缺省情况下处理检测行数的命令。检测分支百分比和调用返回百分比的命令，当run-gcov命令具有相应的参数branches或calls时，才会被处理。例如，下面指定了两者都检测，同时还传给gcov一个`-b'：

```
{ dg-final { run-gcov branches calls { -b sourcefile } } }
```

行数命令出现在注释中，其源行期望能获得指定的数，并具有形式count(cnt)。测试应该只检测对于任意体系结构都会得到同一数值的行数。

检测分支百分比（branch）和调用返回百分比（returns）的命令彼此很相似。起始命令出现在将报告百分比的一个行范围的第一行或者之前，结束命令跟在行范围之后。起始命令可以包括一个百分比列表，其为在范围内期望获得的。范围由同种类型的下一个命令来终止。命令branch(end)或returns(end)标记了范围的结束，而不起是一个新的。例如：

```
if (i > 10 && j > i && j < 20) /* branch(27 50 75) */
                    /* branch(end) */
 foo (i, j);
```

对于调用返回百分比，指定的值为调用报告返回的百分比。对于分支百分比， 值或者为期望的百分比，或者为100减去那个值， 因为分支的方向可以根据目标机或优化级别而不同。

并不是所有的分支和调用都需要被检测。 测试不应检测可能被优化掉的或者被断言指令替代的分支。 不要检测编译器插入的调用或者可能被inline或优化掉的。

单个测试可以检测行数，分支百分比和调用返回百分比的组合。 检测行数的命令必须出现在将会报告行数的行中， 但是检测分支百分比和调用返回百分比的命令，可以将报告的行括起来。

6.4.7 对profile-directed优化测试的支持

文件`profopt.exp'提供了对检测， 使用profile指导的优化来构建的测试是否正确执行的支持。 该测试需要测试程序被构建和执行两次。第一次用来编译生成profile数据， 第二次用来使用第一次执行生成的数据来编译。 第二次执行用来验证测试产生了预期的结果。

要检查优化确实生成了更好的代码，测试可以被构建并运行第三次， 使用标准的来验证性能是否比profile指导的优化更好。 `profopt.exp'提供了这种最初的支持。

`profopt.exp'为profile指导的优化提供了通用的支持。 每套测试都使用了其提供的关于特定优化的信息：

`tool`
: 被测试的工具，如gcc。

`profile_option`
: 用于生成profile数据的选项。

`feedback_option`
: 使用profile数据进行优化的选项。

`prof_ext`
: profile数据文件的后缀。

`PROFOPT_OPTIONS`
: 运行每个测试所使用的选项列表，类似于torture测试的列表。

6.4.8 对二进制兼容性测试的支持

文件`compat.exp'提供了对二进制兼容性测试的语言无关的支持。 它支持测试遵循相同ABI的两个编译器，或者编译器选项的多个集合， 在一起工作时不影响二进制兼容性。它是打算用来补充ABI测试包的。

该框架支持的测试具有三部分，分别在不同的源文件中：主程序， 和两个相互作用来拆分要测试的功能的部分。

`testname_main.suffix'
: 包含了主程序，其调用了文件`testname_x.suffix'中的函数。

`testname_x.suffix'
: 包含了至少一个对`testname_y.suffix'中的函数的调用。

`testname_y.suffix'
: 与`testname_x.suffix'共享数据，或者从中获得参数。

每个测试中，主程序和一个功能部分由测试GCC编译。 另一部分可以由一个候选编译器来编译。如果没有指定候选编译器， 则所有的这三个源文件都由测试 GCC编译。你可以指定一个编译器选项

对。 每一对中第一个元素指定了测试GCC使用的选项，第二个元素用于候选编译器。 每个测试都是用这些选项来编译。

`compat.exp'定义了缺省的一对编译器选项。 这些可以通过定义环境变量COMPAT_OPTIONS来取代，如：

```
COMPAT_OPTIONS="[list [list {tst1} {alt1}]
 ...[list {tstn} {altn}]]"
```

其中tsti和alti是选项列表，要测试的编译器使用tsti， 候选的编译器使用alti。例如， 对于[list [list {-g -O0} {-O3}] [list {-fpic} {-fPIC -O2}]]， 测试会首先被要测试的编译器使用`-g -O0'来编译， 并且被候选编译器使用`-O3'来编译。测试在第二次的时候， 测试编译器会使用`-fpic'，候选编译器会使用`-fPIC -O2'。

候选编译器通过将环境变量定义为安装的编译的全路径名来指定；对于C， 定义ALT_CC_UNDER_TEST，对于C++ ，定义ALT_CXX_UNDER_TEST。 这些将被写入DejaGnu使用的`site.exp'文件中。缺省情况，是由要测试的编译器， 使用COMPAT_OPTIONS中每个编译选项对的第一个，来构建每个测试。 当ALT_CC_UNDER_TEST或ALT_CXX_UNDER_TEST为same时， 每个测试将使用测试编译来构建，但是使用COMPAT_OPTIONS中的选项组合。

若要对要测试的编译器和另一版本的使用指定编译选项的GCC， 只运行C++兼容性测试包，则在`objdir/gcc'下执行下列命令：

```
rm site.exp
make -k \
 ALT_CXX_UNDER_TEST=${alt_prefix}/bin/g++ \
 COMPAT_OPTIONS="lists as shown above" \
 check-c++ \
 RUNTESTFLAGS="compat.exp"
```

如果一个测试当源文件由不同编译器编译的时候执行失败， 而当文件由同一编译器编译的时候执行成功，则表明生成代码或运行时支持不兼容。 如果一个测试对于候选编译器执行失败，但对于测试编译器却执行成功， 则可能是因为一个bug在测试编译器中已经被修改好，但还存在于候选编译器中。

二进制兼容性测试支持少数在测试文件的注释中出现的测试框架命令。

dg-require-*
: 这些命令可以用于`testname_main.suffix'中， 使得当目标机上没有特定支持的时候跳过测试。

dg-options
: 指定选项用于编译该特定的源文件，将选项添加到COMPAT_OPTIONS中。 当该命令出现在`testname_main.suffix'中时， 选项还用于链接测试程序的时候。

dg-xfail-if
: 该命令可以用在第二个源文件中，来指定在特定目标机上对于特定选项，编译将会失败。

6.4.9 对使用多个选项进行torture测试的支持

在整个编译器测试包中，有几个目录，它们的测试程序被运行多次， 每次使用不同的选项集合。这些被称为torture测试。 `gcc/testsuite/lib/torture-options.exp'定义了建立这些列表的程序：

torture-init
: 初始化对torture列表的使用。

set-torture-options
: 设置用于测试程序的torture选项列表。 可选的，可以将torture选项集合与其它选项集合组合一起， 正如Objective-C运行时选项所用的方式。

`torture-finish`
结束对torture列表的使用。

使用torture选项的用于测试集的文件`.exp'，必须包含对这三个程序的调用，如果：

- 其调用了`gcc-dg-runtest`并且覆写了`DG_TORTURE_OPTIONS`
- 其调用了`${tool}-torture`或者`${tool}-torture-execute`，其中tool为`c`, `fortran`, 或者`objc`.
- 其调用了`dg-pch`.

如果测试程序将使用在`gcc-dg.exp'中定义的`DG_TORTURE_OPTIONS`中的列表，则文件`.exp'不必通过调用`gcc-dg-runtest`来调用torture程序。

大多数对torture选项的使用，可以通过定义`TORTURE_OPTIONS`或者通过定义`ADDITIONAL_TORTURE_OPTIONS`增加到缺省列表中，来覆写缺省列表。在文件`.dejagnurc'中定义这些，或者将它们增加到文件`site.exp'中；例如

```
set ADDITIONAL_TORTURE_OPTIONS [list \
  { -O2 -ftree-loop-linear } \
  { -O2 -fpeel-loops } ]
```

7 选项描述文件

大多数GCC命令行选项由特定的选项定义文件来描述，按照惯例命名为`.opt`。这章描述了这些文件的格式。

7.1 选项文件格式

选项文件是一个简单的记录列表，记录中的每个域独自占有一行，记录之间由空行分隔。注释独自占有一行，可以出现在文件的任何地方，并且由分号开头。分号前允许有空格。

文件中可以包含下列类型的记录：

- 语言定义记录。这些记录有两个域：字符串`Language'和语言的名字。一旦通过这种方式声明了一个语言，则可以作为选项属性来使用。参见 Section 7.2 [选项属性], page 68。
- A target specific save record to save additional information. These records have two fields: the string `TargetSave', and a declaration type to go in the `cl_target_option` structure.
- 选项定义记录。这些记录有如下域：
  1. 选项的名字，去掉前导符“-”
  2. 空格分隔的选项属性列表 (参见 Section 7.2 [选项属性], page 68)
  3. 帮助文本，用于`--help'（如果第二个域包含`Undocumented`属性，则会忽略该域）

  缺省的，所有以“f”，“W”或者“m”开头的选项被隐式的假设会有一个“no-”形式。该形式不必再单独列出来。如果以这些字母开头的选项没有“no-”形式，可以使用`RejectNegative`属性来去掉。

  帮助文本在显示前会被自动换行。正常情况下，选项的名字会被打印在输出的左边，帮助文本被打印在右边。但是，如果帮助文本包含一个tab字符，则tab左边的文本会被用来替代选项的名字，tab右边的文本作为帮助文本。这样就可以用来详尽的阐述选项会使用什么类型的参数。
- 目标掩码记录。这些记录有一个域，形式为`Mask(x)'。选项处理脚本将会自动在target_flags（参见运行时目标）中为每个掩码名字x分配一个位，并且将`MASK_x`宏对应的位置位。同时，会声明一个`TARGET_x`宏，当`MASK_x`位被置位时其值为1，否则为0。

它们最初是为了声明与用户选项没有联系的目标掩码，或者是因为这些掩码表示内 部的开关，或者是因为这些选项不在所有的配置中，但是还需要定义掩码的。

7.2 选项属性

选项记录的第二个域可以指定下列属性：

`Common` 选项对所有语言和目标都有效。

`Target` 选项对所有语言都有效，但是目标特定的。

language 当编译给定语言时，选项有效。

可以为多个不同的语言指定同一个选项。每个语言 language 必须 已经被之前的 `Language` 记录声明过。参见 Section 7.1 [选项文件格式], page 67。

`RejectNegative`

选项没有“no-”形式。所有由“f” ，“W”或者“m”开始的选项都被假设具有“no-”形式， 除非使用这个属性。

`Negative(`othername`)`

选项将会关掉另一个选项 othername，这是去掉前导符“-”的选项名字。 这会通过 `Negative` 属性来传播一连串的选项关闭行为。

`Joined`
`Separate` 选项接受一个强制参数。`Joined` 指示选项和参数可以被包含在同一个 `argv` 项中（例如 `-mflush-func=`name）。 `Separate` 指示选项和参数可以为分开的 `argv` 项（如 `-o`）。一个选项允许同时具有这两个属性。

`JoinedOrMissing`

选项接受一个可选参数。如果参数给出，则会作为选项本身的 `argv` 项 的一部分。

该属性不能和 `Joined` 或 `Separate` 一起使用。

`UInteger` 选项的参数是一个非负整数。选项解析器将会在传给选项处理前检测并转换参数。

`Var(`var`)` 该选项的状态将被存储在变量 var 中。存储状态的方式取决于选项的类型：

- 如果使用了 `Mask` 或者 `InverseMask` 属性，则 var 为包含 mask的整数变量。
- 如果选项是一个正常的on/off开关，则 var 为整数变量，并且当启用该选 项时其值为非零。当使用选项的正面形式时，选项解析器会把变量置为1，当使用 “no-” 形式的时候，置为0。
- 如果选项接受一个参数，并且具有 `UInteger` 属性，则 var 为整数 变量，并且存储了参数的值。
- 否则，如果选项接受一个参数，则 var 为指向参数字符串的指针。如果参数 是可选的并且没有给出，则该指针将为null。

选项处理脚本通常会在 `options.c' 中声明 var，并且在起动时间将 其初始化为0。你可以使用 `VarExists` 和 `Init` 来修改这种行为。

`Var(`var, set`)`

选项控制一个整数变量 var，并且当 var 等于 set 时起作用。 当使用选项的正面形式时，选项解析器会把变量置为 set，当使用“no-”形式 的时候，置为 !set。

var 的声明方式与上面描述的单一参数形式的具有相同的方式。

VarExists 由 Var 属性指定的变量已经存在。这样，就不会有任何定义被增加到 `options.c' 中。应该只有在 `options.c' 之外声明了该变量的时候，才使用这个属性。

Init(value)
由属性 Var 指定的变量应该被静态初始化为 value。

Mask(name)
选项与 target_flags 变量（see Section 17.3 [运行时目标机], page 289）中的一个位相关联，并且当该位被置位时才起作用。你还可以指定 Var 去选择一 个变量，而不只是 target_flags。

选项处理脚本将会自动为选项分配一个唯一的位。如果选项与 `target_flags' 关联，则脚本会将宏 MASK_name 设为合适的位掩码。它还会声明一个 TARGET_name 宏，当选项起作用时其值为1，否则为0。如果使用 Var 将选项关联到不同的变量上，则相关的宏分别叫做 OPTION_MASK_name 和 OPTION_name。

可以使用 MaskExists 来禁止自动位分配。

InverseMask(othername)
InverseMask(othername, thisname)
选项是具有 Mask(othername)属性的另一个选项的反面。如果给出了 thisname，则选项处理脚本会声明一个 TARGET_thisname 宏， 当选项起作用时其为1，否则为0。

MaskExists 由 Mask 属性指定的掩码已经存在。这样，就不会有 MASK 和 TARGET 定义被增加到 `options.h' 中。

该属性的主要目的用来支持同义选项。第一个选项应该使用 `Mask(name)' 并且其它的应该使用 `Mask(name) MaskExists'。

Report 选项的说明应该通过 `-fverbose-asm' 来打印。

Undocumented
选项有意的不提供文档，并且不应该包括在 `--help' 输出中。

Condition(cond)
选项应该只在预处理程序条件 cond 为真时才被接受。注意即使 cond 为 假时，任何与选项相关的C声明也会存在； cond只是简单的控制选项是否被接受， 以及是否在 `--help' 输出中被打印。

Save Build the cl_target_option structure to hold a copy of the option, add the functions cl_target_option_save and cl_target_option_restore to save and restore the options.

8 编译器的Passes和相关文件

这章综述了编译器的优化和代码产生过程。

8.1 语法分析过程

语言前端只被调用一次，通过lang_hooks.parse_file， 用来解析整个输入。语言前端可以使用任何被认为合适的中间语言表示。 C前端使用了GENERIC树，以及在`c-common.def'中定义的（double handful of）语言特定的树代码。 Fortran前端使用了完全不同的私有表示。

在某个地方， 前端必须将其使用的表示转换为编译器中语言独立的部分能够理解的表示。 目前的实现采用了两种形式。C前端在函数编译完之前， 手动的对每个函数调用gimplifier， 并且使用

gimplifier回调函数将语言特定的树代码直接转换为GIMPLE。 Fortran前端将私有表示转换为GENERIC，之后当函数编译完时， 再降低为GIMPLE。选择哪种途径可能取决于GENERIC（及其扩展）是否能够很好的匹配源语言，以及是否需要解析数据结构。

BUG ： Gimplification必须在nested function lowering之前进行， 并且nested function lowering必须在将数据传给cgraph之前，由前端完成。

TODO：Cgraph应该控制nested function lowering。并且只会在确定最外层函数被使用时才调用。

TODO：Cgraph需要一个gimplify_function回调函数。 并且在下列情况下会被调用：(1)确定函数被使用， (2)为了兑现用户指定的警告选项，需要多次的编译， (3)语言本身表明了在gimplification进行前，语义分析会不完整。 嗯。。。听起来有点过度复杂。或许我们应该总是进行前端的gimplify； 大多数情况，这只是一个函数调用。

前端需要将所有函数的定义和顶层的声明传给中端， 以至于它们能被编译和生成目标文件。对于一个简单的程序语言， 顶层的每个声明和定义都能找到，因此这样做非常方便。 另外，对于生成函数代码和生成完全的调试信息，也有差别。 对于函数代码，唯一必须的是将函数和数据定义传给中端。 对于完全的调试信息，函数，数据和类型的声明也都需要被传递。

任何情况下，如果前端需要每个完全的顶层函数或数据声明， 则每个数据定义应该传给rest_of_decl_compilation。 每个完全的类型定义应该传给rest_of_type_compilation。 每个函数定义应给传给cgraph_finalize_function。

中端根据自己的选择，将会立即生成函数和数据的定义， 或者放入队列中以便后面的处理。

8.2 Gimplification过程

Gimplification是一个离奇的术语， 用来表示将函数的中间表示转换为GIMPLE语言的过程。 The term stuck，所以像“gimplification”，“gimplify”，“gimplifier” 等类似的单词会分布在这部分代码中。

当然，尽管前端可以选择直接生成GIMPLE，但如果这样， 则处理起来可能会有些复杂，除非前端使用的中间语言非常简单。 通常生成GENERIC树以及其扩展会相对容易些， 并且让语言独立的gimplifier来多大部分的工作。

这个过程的主入口点是gimplify_function_tree， 位于`gimplify.c'中。我们从这里处理整个函数， 依次的对每条语句进行gimplify。 这个过程的主要工作是gimplify_expr。 几乎每个处理都需要经过这里一次， 并且我们是从这里来调用lang_hooks.gimplify_expr回调的。

回调函数应该考虑检查表达式， 并且当表达式不是一个需要注意的语言特定的结构， 则返回GS_UNHANDLES。 否则，应该通过某种方式修改表达式使得可以产生合法的GIMPLE。 如果回调函数确定已经完全转换，并且表达式是合法的GIMPLE， 则应该返回GS_ALL_DONE。 否则，应该返回GS_OK，这将会使得表达式会被再次处理。 如果回调函数在转换过程中遇到一个错误（因为前端依赖于gimplification处理来完成语义检测）， 则应该返回GS_ERROR。

8.3 过程管理器

过程管理器在`passes.c'， `tree-optimize.c'和`tree-pass.h'中。 它的工作是按照正确的顺序来运行所有单独的过程， 并且处理应用到每个过程的标准薄记（standard bookkeeping）。

工作原理是每个过程定义了一个结构体， 用来表示我们需要知道的关于该过程的每件事情——应该什么运行， 应该如何运行，需要什么中间语言或者附加的数据结构。 我们按照某种特定的运行顺序来注册过程， 过程管理器来安排所有的事情都按正确的顺序发生。

目前实际的实现并不是完全做到了理论上所描述的。 命令行开关和timevar_id_t枚举还必须定义在其它地方。 过程管理器。。。不管怎样，现在的实现是有用的，比没有强。

TODO：描述由过程管理器建立的全局变量，以及对一个新的过程应该如何使用。

8.4 Tree-SSA过程

下面简要描述了经过gimplification之后的树优化过程，以及所在的源文件。

- 删除无用语句（Remove useless statements）
  该过程对gimple代码进行非常简单的扫描，识别出明显的死代码并删除。我们在这里做的一些事情包括，简化具有不变条件的if语句，删除对显然不会抛出异常的代码所做的异常处理，删除不含有变量的词法绑定（lexical bindings），以及其它各种简单的清除。这是为了能够快速的去掉一些显而易见的东西，而不是等到后面去花费更多的功夫。该过程在`tree-cfg.c'中，并且由pass_remove_useless_stmts来描述。
- Mudflap声明注册（Mudflap declaration registration）
  如果启用了mudflap（参见 Section ``-fmudflap -fmudflapth -fmudflapir'' in Using the GNU Compiler Collection (GCC)）我们便产生代码来记录一些使用mudflap运行时的变量声明。特别的，运行时会跟踪这些变量声明的生命期，将它们的地址记录下来，或者哪些边界在编译时不知道（extern）。该过程生成新的异常处理结构（try/finally），因此必须在它们下降之前运行。另外，该过程enqueque生命期扩展为整个程序的静态变量声明。过程位于`tree-mudflap.c'中，并由pass_mudflap_1来描述。
- OpenMP下降（OpenMP lowering）
  如果启用了OpenMP生成（`-fopenmp'），该过程将OpenMP结构下降为GIMPLE。
  OpenMP结构下降涉及到为使用数据共享子句映射的局部变量创建替代表达式，揭示最可能同步指令的控制流，以及增加region标记来帮助控制流图的创建。该过程位于`omp-low.c'中，并由pass_lower_omp来描述。
- OpenMP扩展（OpenMP expansion）
  如果启用了OpenMP生成（`-fopenmp'），该过程将并行region扩展为由线程库调用的它们自己的函数。过程位于`omp-low.c'中，并由pass_expand_omp来描述。
- 控制流下降（Lower control flow）
  该过程压平（flatten）if语句（COND_EXPR），并将词法绑定（BIND_EXPR）移到行外。在该过程之后，所有if语句将会有确切的两条goto语句在then和else处。每条语句的词法绑定信息将在TREE_BLOCK中找到，而不是由BIND_EXPR下的它的位置来推算出。该过程可以在`gimple-low.c'中找到，并由pass_lower_cf来描述。
- 异常处理控制流下降（Lower exception handling control flow）
  该过程将高级别的异常处理结构（TRY_FINALLY_EXPR和TRY_CATCH_EXPR）转换为能显示表示控制流相关的形式。该过程之后，lookup_stmt_eh_region将会为任何可能具有EH控制流语义的语句返回一个非负数；对于确切的语义可以检查tree_can_throw_internal或tree_can_throw_external。确切的控制流可以从foreach_reachable_handler中提取。EH region嵌套树在`except.h'和`except.c'中定义。下降过程本身在`tree-eh.c'中，并由pass_lower_eh来描述。
- 构建控制流图（Build the control flow graph）
  该过程将函数分解为基本块，并创建所有相连的边。它位于`tree-cfg.c'中，并由pass_build_cfg描述。
- 找到所有被引用的变量（Find all referenced variables）
  该过程遍历整个函数，并将函数中所有被引用的变量搜集到一个数组中，referenced_vars。每个变量在数组中的索引被用作函数中这个变量的UID。SSA重写程序需要用到该数据。过程位于`tree-dfa.c'中，并由pass_referenced_vars来描述。

- 进入静态单赋值形式（Enter static single assignment form）

  该过程将函数重写为SSA形式。该过程之后，所有is_gimple_reg变量将通过SSA_NAME来引用，并且所有其它变量将由VDEFS和VUSES来注解；对于每个基本块，PHI节点将会在需要的时候被插入。该过程位于`tree-ssa.c'中，并由pass_build_ssa来描述。

- 未初始化变量警告（Warn for uninitialized variables）

  该过程扫描函数，寻找使用缺省定义的SSA_NAME。对于非参数变量，这样的使用是未初始化的。该过程运行两次，优化前和优化后。第一次过程中，我们只警告肯定是未初始化的；在第二次过程中，我们警告可能未初始化的。过程位于`tree-ssa.c'中，并由pass_early_warn_uninitialized和pass_late_warn_uninitialized定义。

- 死代码消除（Dead code elimination）

  该过程扫描函数来寻找没有副作用，且结果没有被使用的语句。它不进行内存活跃分析，所以任何存储在内存中值都被认为是被使用的。该过程在整个优化处理中被运行多次。它位于`tree-ssa-dce.c'中，并由pass_dce来描述。

- dominator优化（Dominator optimizations）

  该过程执行平凡的基于dominator的复制和常量传播，表达式简化，以及跳转线程化。它在整个优化处理中被运行多次。它位于`tree-ssa-dom.c'中，并由pass_dominator来描述。pass_dominator.

- 单用变量向前传播（Forward propagation of single-use variables）

  该过程尝试移除冗余计算，通过将只使用一次的变量替换为使用它们的表达式，并查看是否得到的结果可以被简化。它位于`tree-ssa-forwprop.c'中，并由pass_forwprop来描述。

- 复制重名命（Copy Renaming）

  该过程尝试改变涉及复制操作的编译器临时对象的名字，例如SSA->normal。当编译器临时对象是用户变量复制时，它还将编译器临时对象重命名为用户变量，使得可以更好的使用用户符号。它位于`tree-ssa-copyrename.c'中，并由pass_copyrename来描述。

- PHI节点优化（PHI node optimizations）

  该过程识别可以被表示为条件表达式的PHI输入，并将它们重写成线形的代码。它位于`tree-ssa-phiopt.c'中，并由pass_phiopt来描述。

- 可能别名优化（May-alias optimization）

  该过程执行一个流敏感基于SSA指向的分析。所得的may-alias, must-alias和escape分析 信息用来将变量从内存中可寻址的对象提升为可以被重命名为SSA形式的无别名变量。我们还为非可命名的聚合体更新VDEF/VUSE内存标记，使得可以获得较少的错误。过程位于`tree-ssa-alias.c'中，并由pass_may_alias来描述。

  进程间的指向信息位于`tree-ssa-structalias.c'中，并由pass_ipa_pta来描述。

- Profiling

  该过程重写函数，用于搜集运行时块和评估profiling数据。这些数据可以反馈给随后的编译器运行，这样就可以进行基于预期执行频率的优化。过程位于`predict.c'中，并由pass_profile来描述。

- 复数算术运算下降（Lower complex arithmetic）

  该过程将复数算术运算重写为各部分的标量算术运算。过程位于`tree-complex.c'中，并由pass_lower_complex来描述。

- 聚合体标量替换（Scalar replacement of aggregates）
  该过程将适当的非别名局部聚合体变量重写为一个标量集合。 所得的标量变量被重写成SSA形式，这样就允许后面的优化过程来做更好的工作。 过程位于`tree-sra.c'中，并由pass_sra来描述。
- 死存储消除（Dead store elimination）
  该过程消除死存储，即存储到内存中，而该内存被随后的另一个存储操作重新写入， 并且之间没有加载操作。过程位于`tree-ssa-dse.c'中， 并由pass_dse来描述。
- 尾递归消除（Tail recursion elimination）
  该过程将所有的尾递归转换到一个循环中。 它位于`tree-tailcall.c'中， 并由pass_tail_recursion来描述。
- 向前存储移动（Forward store motion）
  该过程将存储和赋值操作下沉到流图中接近它的使用点。 过程位于`tree-ssa-sink.c'中，并由pass_sink_code来描述。
- 部分冗余消除（Partial redundancy elimination）
  该过程消除部分冗余计算，同时执行加载移动。过程位于`tree-ssa-pre.c'中， 并由pass_pre来描述。
  如果设置了`-funsafe-math-optimizations'，则在部分冗余消除前， GCC尝试通过倒数方式将除法转换为乘法。过程位于`tree-ssa-math-opts.c'中， 并由pass_cse_reciprocal来描述。
- 完全冗余消除（Full redundancy elimination）
  这是一个较简单的PRE形式，只消除在所有路径上产生的冗余。 它位于`tree-ssa-pre.c'中，并由pass_fre来描述。
- 循环优化（Loop optimization）
  该过程的主驱动程序位于`tree-ssa-loop.c'中，并且由pass_loop来描述。
  该过程执行的优化为：
  循环不变量移动。该过程只移动在rtl级难以处理的不变量（函数调用， 扩展成非平凡insn序列的操作）。使用`-funswitch-loops'时， 它还将不变的条件操作数移到循环外面， 使得我们能够在循环外提过程中只需要进行平凡不变量分析。该过程还包括存储移动。 该过程在`tree-ssa-loop-im.c'中实现。
  正规归纳变量创建。该过程为循环迭代次数创建一个简单计数器， 并使用它来替换循环的退出条件，以用于当一个复杂的分析需要确定迭代次数的时候。 之后的优化便可以容易的确定迭代次数。 该过程在`tree-ssa-loop-ivcanon.c'中实现。
  规约变量优化。该过程执行标准的规约变量优化，包括强度缩减，规约变量合并， 以及规约变量消除。该过程在`tree-ssa-loop-ivopts.c'中实现。
  循环外提。该过程将不变的条件跳转移到循环外面。为了达到这一点， 对于每种可能的条件跳转结果都会创建一个循环副本。 该过程在`tree-ssa-loop-unswitch.c'中实现。 该过程应该最终替代在`loop-unswitch.c'中的rtl级的循环外提， 但是目前rtl级的过程还不是完全多余的，是因为还缺少tree级的别名分析。
  这些优化还用到了`tree-ssa-loop-manip.c', `cfgloop.c', `cfgloopanal.c'和`cfgloopmanip.c'中的各种函数。
  向量化。该过程将循环由标量类型操作转换为向量类型操作。跨越循环迭代的数据并行被利用，将数据元素从连续的迭代中组合成一个向量，对它们并行的操作。取决于可用的目标机的支持，循环在概念上按照因子VF（vectorization factor）被展开。
  自动并行化。该过程将循环迭代空间拆分到几个线程来运行。该过程在tree-parloops.c中实现。

- 用于向量化的Tree级if转换（Tree level if-conversion for vectorizer）

  该过程应用if转换来简化循环，以助于向量化。我们识别可以if转换的循环，并将基本块合并到一个大块中。想法是将循环表现为这样的形式，使得向量化能够对语句和可用的向量操作进行一一映射。该patch在GIMPLE级重新引入了COND_EXPR。该过程位于`tree-if-conv.c'中，并由pass_if_conversion来描述。

- 条件常量传播（Conditional constant propagation）

  该过程松弛一个点阵值用于识别那些即使在条件分支中也肯定是常数的。该过程位于`tree-ssa-ccp.c'中，并由pass_ccp来描述。

  一个相关的工作于内存加载和存储，而不只是寄存器值的过程，位于`tree-ssa-ccp.c'中，并由pass_store_ccp来描述。

- 条件复制传播（Conditional copy propagation）

  这类似于常量传播，不过点阵值是与“copy-of”相关的。它消除代码中的冗余复制。该过程位于`tree-ssa-copy.c'中，并由pass_copy_prop来描述。

  一个相关的工作于内存复制而不只是寄存器复制的过程，位于`tree-ssa-copy.c'中，并由pass_store_copy_prop来描述。

- 值范围传播（Value range propagation）

  该转换类似于常量传播，只不过它是传播已知值的范围，而不是传播单个常数值。该实现基于Patterson的范围传播算法（Accurate Static Branch Prediction by Value Range Propagation, J. R. C. Patterson, PLDI '95）。相对于Patterson的算法，该实现没有传播分支可能性，也没有对SSA名使用多个范围。这意味着现在的实现不能用于分支预测（虽然并不难实现）。该过程位于`tree-vrp.c'中，并由pass_vrp来描述。

- 折叠built-in函数（Folding built-in functions）

  该过程适当的简化built-in函数，使用常量参数或者可推算出的字符串长度。它位于`tree-ssa-ccp.c'中，并由pass_fold_builtins来描述。

- 拆分临界边（Split critical edges）

  该过程识别出临界边，并插入空基本块来将其转换为非临界的。该过程位于`tree-cfg.c'，并由pass_split_crit_edges描述。

- 控制依赖死代码消除（Control dependence dead code elimination）

  该过程是死代码消除的较强形式，能够消除不必要的控制流程语句。它位于`tree-ssa-dce.c'中，并由pass_cd_dce来描述。

- 尾调用消除（Tail call elimination）

  该过程识别可以被重写为跳转的函数调用。这里没有进行实际的代码转换，不过却解决了数据流和控制流的问题。代码转换需要目标机支持，因此被推迟到RTL级。同时，CALL_EXPR_TAILCALL被设置，以用来指示可能性。该过程位于`tree-tailcall.c'中，并且由pass_tail_calls来描述。RTL转换由`calls.c'中的fixup_tail_calls来处理。

- 对函数没有返回值的警告（Warn for function return without value）

  对于非void型的函数，该过程定位没有指定一个值的返回语句，并产生一个警告。这样的语句可能是在函数结束处。该过程在最后运行，这样我们能够更多可能的去检验这些语句是不可达的。其位于`tree-cfg.c'中，并由pass_warn_function_return来描述。

- Mudflap语句注解（Mudflap statement annotation）

  如果启用了mudflap，我们便重写一些内存访问代码以确保内存访问是正确的。特别的，涉及到指针废除的表达式（INDIRECT_REF, ARRARY_REF等等）被替代为检查选择地址范围 的代码

，而不是mudflap运行时数据库的有效域。该检查包括一个内联的对直接映射缓存 的查找，基于对指针值的shift/mask操作，和对运行时的回滚函数调用。 该过程位于`tree-mudflap.c'中，并由pass_mudflap_2来描述。

- 离开静态单赋值形式（Leave static single assignment form）

  该过程重写函数使得其处于正常形式。同时，我们尽可能的消去单一使用的临时对象， 这样中间语言就不再是GIMPLE了，而是GENERIC。该过程位于`tree-outof-ssa.c'中， 并且由pass_del_ssa来描述。

- 合并PHI节点（Merge PHI nodes that feed into one another）

  这是CFG清除过程的一部分。它试图将PHI节点从前部CFG块合并到另一个带有PHI节点的块。该过程位于`tree-cfgcleanup.c'中，并由pass_merge_phi来描述。

- 返回值优化（Return value optimization）

  如果函数总是返回同一局部变量，并且那个局部变量是一个聚合类型， 则变量将由函数返回值来替换（即函数的DECL_RESULT）。 这相当于作用于GIMPLE的C++命名返回值优化。该过程位于`tree-nrv.c'中， 并且由pass_nrv来描述。

- 返回槽优化（Return slot optimization）

  如果函数返回一个内存对象，并且像var = foo()这样被调用， 该过程尝试改变调用，使得var的地址传送给调用者，以避免一次额外的内存复制。 该过程位于tree-nrv.c中，并由pass_return_slot来描述。

- 优化调用__builtin_object_size（Optimize calls to __builtin_object_size）

  这是一个类似于CCP的传播过程，其试图移除对__builtin_object_size的调用， 当对象的大小能够在编译时计算出的时候。该过程位于`tree-object-size.c'中， 并有pass_object_sizes来描述。

- 循环不变量移动（Loop invariant motion）

  该过程将昂贵的循环不变量计算移出循环。该过程位于`tree-ssa-loop.c'中， 并由pass_lim来描述。

- 循环嵌套优化（Loop nest optimizations）

  这是一类工作于循环嵌套的循环转换。它包括循环变换（loop interchange），scaling，skewing和逆转（reversal），并且它们用来配合。 该过程位于`tree-loop-linear.c'中，并由pass_linear_transform来描述。

- 空循环移除（Removal of empty loops）

  该过程移除不含代码的循环。该过程位于`tree-ssa-loop-ivcanon.c'中， 并由pass_empty_loop来描述。

- 小循环展开（Unrolling of small loops）

  该过程将迭代次数很少的循环完全展开。该过程位于`tree-ssa-loop-ivcanon.c'中， 并由pass_complete_unroll来描述。

- 预测公约（Predictive commoning）

  该过程使代码可以重用先前循环迭代的计算，特别是对内存的加载和存贮。 该过程位于`tree-predcom.c'中，并由pass_predcom来描述。

- 数组预取（Array prefetching）

  该过程为循环中的数组引用产生预提取指令。 过程位于`tree-ssa-loop-prefetch.c'中，并由pass_loop_prefetch来描述。

- 重组（Reassociation）

  该过程将算术表达式重写为可以进行优化的形式，例如冗余消除和向量化。过程位于`tree-ssa-reassoc.c'中，并由pass_reassoc来描述。

- 优化stdarg函数

  该过程设法避免在stdarg函数入口处将寄存器参数保存到栈中。如果函数不使用任何va_start宏，则没有寄存器需要被保存。如果使用了va_start宏，va_list变量的使用范围不超出该函数，则只需要保存将在va_arg宏中使用的寄存器。例如，如果va_arg在函数中只用于整数类型，则不需要保存浮点寄存器。该过程位于tree-stdarg.c中，并由pass_stdarg来描述。

8.5 RTL过程

下面简要描述了tree优化之后所运行的rtl生成和优化过程。

- RTL生成

  实现RTL生成的源文件包括`stmt.c',`calls.c',`expr.c',`explow.c',`expmed.c',`function.c',`optabs.c'和`emit-rtl.c'。该过程还用到了由genemit程序通过机器描述生成的`insn-emit.c'文件。该过程使用头文件`expr.h'来交互信息。

  由程序genflags和gencodes通过机器描述来生成的头文件`insn-flags.h'和`insn-codes.h'，告诉了该过程哪些标准名字可用，以及哪些模式与它们对应。

- 生成异常处理着陆架（Generate exception handling landing pads）

  该过程生成用来管理异常处理库程序和函数中的异常处理器之间通讯的粘合机制（glue）。由异常处理库调用的函数的入口点，被称作着陆架（landing pads）。该过程的代码位于`except.c'中。

- 清除控制流图（Cleanup control flow graph）

  该过程去除不可达代码，对跳转到下一条指令（jumps to next），连续跳转（jumps to jump），交叉跳转（jumps across jumps）等情况进行简化。该过程被运行多次。出于历史原因，该过程有时被称为“跳转优化过程”。该过程的主要代码在`cfgcleanup.c'中，还有一些辅助程序在`cfgrtl.c'和`jump.c'中。

- 单定义值的向前传播（Forward propagation of single-def values）

  该过程尝试通过替换来自单一定义的变量，并观察结果是否能够被简化的方式，来去除冗余计算。它执行了复制传播和寻址模式选择。该过程运行两次，并只在第二次的时候将值传播到循环中。它位于`fwprop.c'中。

- 公共子表达式消除（Common subexpression elimination）

  该过程去除基本块中的冗余计算，并且根据代价来优化寻址模式。该过程运行两次。源代码位于`cse.c'中。

- 全局公共子表达式消除（Global common subexpression elimination）

  该过程执行两种不同类型的GCSE，取决于你是否在优化代码大小（基于LCM的GCSE趋向于通过增加代码大小来获得速度，而基于Morel- Renvoise的GCSE则不是）。当优化代码大小时，使用Morel-Renvoise Partial Redundancy Elimination（部分冗余消除）来做GCSE，并不尝试将不变量移到循环之外——这留到循环优化过程。如果进行MR PRE，则还会进行代码提升（code hoisting），（也称为code unification），还有加载移动（load motion）。如果你在优化速度，则会进行基于LCM（lazy code motion）的GCSE。LCM是基于Knoop, Ruthing和Steffen的工作。基于LCM的GCSE也会进行循环不变量代码移动。当优化速度时，我们还执行加载和存储移动。不管使用哪一种类型的GCSE，该过程都还执行全局常量传播和复制传播。该过程的源代码为`gcse.c'，LCM程序在`lcm.c'中。

- 循环优化（Loop optimization）

  该过程执行几个循环相关的优化。源文件`cfgloopanal.c'和`cfgloopmanip.c' 包含了通用的循环分析和操作代码。循环结构体的初始化和完成（finalization）由`loop-init.c'处理。循环不变量移动过程在`loop-invariant.c'中实现。 基本块级的优化——— unrolling,peeling和unswitching——在`loop-unswitch.c'和 `loop-unroll.c'中实现。`loop-doloop.c'是关于使用特定的机器相关结构来 替代循环退出条件的处理。

- 跳转迂回（Jump bypassing）

  该过程是GCSE的激进形式，通过传播常数到条件分支指令中来转换函数的控制流图。 该过程的源文件为`gcse.c'。

- If转换（If conversion） 该过程尝试使用产生比较指令和条件移送指令的算术的布尔值，来替换条件分支和附近的赋值。在重载之后最近的调用中，当目标机支持的时候，其将生成断言指令。该过程位于`ifcvt.c'。

- Web构造（Web construction）

  该过程拆分为独立的使用每个伪寄存器。这能够提高其它转换过程的效率， 例如CSE或者寄存器分配。源文件为`web.c'。

- 生命期分析（Life analysis）

  该过程计算在程序的每个点上哪些伪寄存器是活跃的， 并且使第一条指令使用一个值来指向计算值的指令。 然后它删除结果从来不会被使用的计算， 并且将内存引用和加减指令组合为自动增量或者自动减量寻址。 该过程位于`flow.c'中。

- 指令合并（Instruction combination）

  该过程尝试去将数据流相关的两条或者三条指令组合并为单一指令。它通过替代， 使用代数简化结果的方式来为指令合并RTL表达式，然后尝试去将结果跟机器描述匹配。 该过程位于`combine.c'。

- 寄存器移动（Register movement）

  该过程寻找这样的情况，即当匹配约束条件时会迫使指令需要重载， 并且这个重载为一个寄存器到寄存器的move操作。 然后它尝试改变指令使用的寄存器来避免move指令。该过程位于`regmove.c'中。

- 优化模式转换（Optimize mode switching）

  该过程寻找这样的指令，即需要处理器处于特定的模式（mode），然后将模式改变的数目减到最少。这些模式是什么以及应用于什么完全是目标机特定的。 源代码位于`mode-switching.c'中。

- 模调度（Modulo scheduling）

  该过程查看内部循环并且通过复合不同的迭代来重排它们的指令。 模调度在指令调度之后立即被执行。该过程位于`modulo-sched.c'。

- 指令调度（Instruction scheduling）

  该过程寻找这样的指令，其输出在后来的指令中不会用到。在RISC机器上， 内存加载和浮点指令经常会有这样的特征。 它重新排序基本块中的指令以尝试将定义和使用分开，从而避免引起流水线阻塞。 该过程执行两次，分别在寄存器分配之前和之后。 该过程位于`haifa-sched.c', `sched-deps.c', `sched-ebb.c', `sched-rgn.c'和`sched-vis.c'中。

- 寄存器分配（Register allocation）

  这些过程确保所有伪寄存器都被去除，或者通过给它们分配硬件寄存器， 或者使用等价表达式来替换（例如常数），或者将它们放在栈中。这由几个子过程来完成：

- 寄存器移送优化。该过程进行一些简单的RTL代码转换，来改进后续的寄存器分配。源文件为`regmove.c'。
- 集成寄存器分配(IRA)。其被称作集成的，是因为合并，寄存器活跃范围拆分，以及硬件寄存器优先选定，都在图着色中被执行。其还更好的与重载过程集合在一起。寄存器分配或者重载所溢出的伪寄存器还有机会来获得硬件寄存器，如果重载从硬件寄存器中赶出了一些伪寄存器。寄存器分配可以帮助选择更好的伪寄存器来溢出，基于它们的活跃返回，并且合并为溢出的伪寄存器所分配的栈槽。IRA为一个区域的寄存器分配，如果只有一个区域，则其被转换成Chaitin-Briggs寄存器分配。缺省的IRA使用寄存器压力来选择区域，不过用户可以强制其使用一个区域或者对应于所有循环的多个区域。

  寄存器分配的源文件为`ira.c', `ira-build.c', `ira-costs.c', `ira-conflicts.c', `ira-color.c', `ira-emit.c', `ira-lives'，以及头文件`ira.h'和`ira-int.h'，用于在分配器和编译器其它部分，以及IRA文件之间的交互。
- 重载（reloading）。该过程使用分配的硬件寄存器编号来重新为伪寄存器编号。 没有获得硬件寄存器的伪寄存器被分配到栈槽中。然后寻找那些由于无法放入寄存器， 或者放入错误类型寄存器所引起的无效指令。 通过将有问题的值临时重载到寄存器中来修复这些指令，并生成辅助的指令进行复制。

  重载过程还可选的消除帧指针， 以及插入指令来保存和恢复调用破坏掉的（call-clobbered）寄存器。

  源文件为`reload.c'和`reload1.c'， 还有用于信息交互的头文件`reload.h'。

- 基本块重新排序（Basic block reordering）

  该过程实现了profile指导的代码安置（code positioning）。如果profile信息不可用， 便会执行不同类型的静态分析来作出通常通过profile反馈（IE执行频率，分支可能性等） 而得出的预测。其在`bb-reorder.c'中实现，不同的预测程序在`predict.c'中。

- 变量跟踪（Variable tracking）

  该过程计算变量在代码中的每个位置（position）所被存储的到的地方， 并生成注解到RTL代码中来描述变量位置(location)。 如果调试信息格式支持位置列表(location lists)的话， 便会根据这些注解来生成位置列表到调试信息中。

- 延迟分支调度（Delayed branch scheduling）

  该可选的过程尝试去找到能够放在其它指令，通常是跳转或者调用指令， 的延迟槽中的指令。源文件名为`reorg.c'。

- 分支缩短（Branch shortening）

  许多RISC机器上，分支指令有一个限制范围。因此，较长的指令序列必须用于长分支。 在这个过程中，编译器计算出指令间的距离有多长， 并且对于每个分支是否使用普通指令或者交长的指令序列。

- 寄存器到栈的转换（Register-to-stack conversion）

  将一些硬件寄存器的使用转换为一个寄存器栈的使用可以在这里完成。目前， 该过程只支持Intel 80387协处理器的浮点寄存器。源文件名为`reg-stack.c'。

- Final过程

  该过程输出函数的汇编代码。源文件为`final.c'和`insn-output.c'， 后者由工具`genoutput'通过机器描述自动生成。 头文件`conditions.h'用于这些文件间的信息交互。如果启用了mudflap， 延迟声明和可寻址常量（如字符串文字）的队列将由mudflap_finish_file处理 成一个调用mudflap运行时的综合构造器函数。

- 调试信息输出

  该过程在Final过程之后运行， 是因为它必须为没有获得硬件寄存器的伪寄存器输出栈槽偏移量。源文件包括， 用于DBX符号表格式的`dbxout.c'，用于SDB符号表格式的`sdbout.c'， 用于DWARF 符 号 表 格 式 的 `dwarfout.c'， 用 于 DWARF2 符 号 表 格 式 的 `dwarf2out.c'和`dwarf2asm.c'， 以及用于VMS调试符号表格式的`vmsdbgout.c'。

9 Trees: C和C++前端使用的中间表示

这一章记述了GCC用来表示C和C++源程序的内部表示。当源程序为C或者C++时， GCC解析程序，执行语义分析（包括生成错误消息）， 然后产生在这里描述的中间表示。 该表示包含了一个对前端输入的整个转换单元的完全表示。 该表示然后由代码生成器处理，从而产生机器代码， 但是还可以用来创建源浏览器，智能编辑器，自动文本生成器，解析器， 以及任何其它处理C或C++代码所需的程序。

这一章解释了内部表示。特别是，记录了C和C++源结构的内部表示， 以及能够用来访问这些结构的宏，函数和变量。 C++的表示很大程度上为C前端使用的表示的超集。 只有一种C中使用的结构没有出现在C++前端，即GNU“嵌套函数”扩展。 许多这里记录的宏不在C中出现，因为相应的语言结构不出现在C 中。

如果你正在开发一个“后端” ，一个代码生成器或一些其它工具， 使用了该表示，你可能偶然发现需要询问一些问题， 并且这些问题不好通过这里列出的函数和宏来回答。如果是这种情况， 可能GCC已经支持了你想要的功能，只不过接口并没有记录在这里。 在这种情况下，你应该询问GCC维护者（通过发邮件给gcc@gcc.gnu.org）， 关于记录你想要的功能。 同样，如果你发现你自己写的函数并不是直接处理你的后端， 而是可能对其他使用GCC前端的人有帮助，你应该将你的patch提交纳入GCC。

9.1 不足之处

该文档中有许多地方不完整并且有错误。目前，还只是一个*初步的*文档。

9.2 概述

内部表示使用的主要数据结构体为tree。 这些节点，即所有C类型的tree，有许多种类。 tree是一个指针类型，但是它所指向的对象可能会有不同的类型。 从现在开始，除非是在谈论实际的C类型tree， 我们提到的树都是指普通类型的。

可以通过TREE_CODE宏来得知特定的树是什么类型的节点。 许多宏以树作为输入，并以树作为输出返回。 然而，大多数宏需要特定的一种树节点作为输入。 换句话说，是有一个树的类型系统，但没有反应在C的类型系统中。

出于安全考虑，使用`--enable-checking'来配置GCC会很有用。 虽然这会导致显著的性能亏损（因为所有tree类型都会在运行时被检查）， 并且因此在发行版中不合适，但在开发阶段会非常有帮助。

许多宏作为判定条件使用。许多并不是所有的这些判定条件都结束于`_P'。 不要依赖于这些宏的结果类型会是特定的类型。 但是，你可以依赖于类型可以与0相比较，这个事实。所以，像这样的语句

```
if (TEST_P (t) && !TEST_P (y))
  x = 1;
```

和

```
int i = (TEST_P (t) != 0);
```

是合法的。现在返回int值的宏，将来可能被改为返回tree值， 或者其它指针。即使那些继续返回int的，也可能会由先前只返回0和1， 改为返回多个非0的代码。因此，不要将代码写成

```
if (TEST_P (t) == 1)
```

因为这样的代码不保证将来会正确工作。

不要使用这里描述的宏或者函数的返回值的地址。 特别是，不保证这些值是左值。

一般来说，宏的名字都是大写形式，而函数的名字都是完全小写的。 很少有例外的。 你应该假设任何完全由大写字母组成的宏或函数可能会对它的参数进行多次求值。 你可以假设完全由小写字母组成的宏或函数将会对它的参数只求一次值。

error_mark_node是一个特殊的树。它的树代码为ERROR_MARK， 但由于只有一个节点具有那个代码， 所以通常的做法是将树与error_mark_node进行比较。（该测试只是一个指针相等的测试。）如果在前端处理中，有一个错误发生， 则标识errorcount将会被设置。如果前端遇到无法处理的代码，则会给用户发送一个消息，并设置sorrycount。当这些标识被设置时， 则通常返回特定类型树的宏和函数，可能会替代的返回error_mark_node。 因此，如果你打算进行任何错误代码处理， 则必须准备好跟error_mark_node打交道。

有时，一个特定的树插槽slot（就像表达式的操作数，或声明里的特定域）将被称为“为后端保留”。这些插槽用于存储RTL，当树被转换为RTL，被GCC后端使用时。 然而，如果没有进行那种处理（例如，如果前端被钩子转向给一个智能编辑器）， 那么这些插槽就可以被使用了。

如果你遇到的情况不符合这个文档，如没有在这里提到的树节点的类型， 或记载的宏返回了不同的某一类型实体，那么你发现了一个bug ，要么是前端的， 要么是该文档的。请报告这些bug，以及是否有任何其他的bug。

9.2.1 Trees

目前还没有这一节的内容。

9.2.2 标识符

IDENTIFIER_NODE表示了比标准C或C++关于标识符更略为一般的概念。 特别是，IDENTIFIER_NODE可以包含一个`$'，或其它额外的字符。

不会有两个不同的IDENTIFIER_NODE表示同一个标识符。 因此，你可以使用指针相等的方式来比较IDENTIFIER_NODE， 而不必使用像strcmp这样的函数。

可以使用下列宏来访问标识符：

IDENTIFIER_POINTER
: 标识符所表示的字符串，为一个char*。 该字符串总是以NUL结束，并且不包含嵌入的NUL字符。

IDENTIFIER_LENGTH
: 由IDENTIFIER_POINTER返回的字符串的长度，不包括结尾的NUL。 IDENTIFIER_LENGTH (x)的值总是与strlen (IDENTIFIER_POINTER (x))相同。

IDENTIFIER_OPNAME_P
: 该断言当标识符表示的是重载操作符的名字时有效。这种情况下， 不要依赖IDENTIFIER_POINTER或IDENTIFIER_LENGTH的内容。

IDENTIFIER_TYPENAME_P
: 该断言当标识符表示的是用户定义的转换操作符的名字时有效。这种情况下，IDENTIFIER_NODE的TREE_TYPE持有转换操作符转换后的类型。

9.2.3 容器

有两种通用容器数据结构可以直接用树节点表示。 TREE_LIST是一个单向链表，每个节点包含两个树。 每个节点都有TREE_PURPOSE和TREE_VALUE。（很多时候，TREE_PURPOSE包含了某种tag，或补充信息， 而TREE_VALUE包含了大部分的有效载荷。在其他情况下， TREE_PURPOSE只是一个NULL_TREE，而还有一些情况， TREE_PURPOSE和TREE_VALUE是处于相等的状况。）给定一个TREE_LIST节点，可以沿着TREE_CHAIN来找到下一个节点。 如果TREE_CHAIN为NULL_TREE时，则表示到达链表的结尾了。

TREE_VEC是一个简单的向量。TREE_VEC_LENGTH是一个整数（不是树），给出了向量中节点的数目。节点本身通过宏TREE_VEC_ELT来访问， 其接受两个参数。第一个为要查询的TREE_VEC； 第二个为整数，指定了向量中的元素。元素索引从0开始。

9.3 类型

所有的类型都有相应的树节点。 但是不要假设一个树节点就是正好对应于一个类型。 经常有多个节点对应于相同的类型。

在大多数情况下，不同种类的类型具有不同的树代码。（例如，指针类型使用POINTER_TYPE代码，而数组使用ARRAY_TYPE代码。）但是， 指向成员函数的指针使用RECORD_TYPE代码。 因此，当写与特定类型相关联的switch语句时， 应该在RECORD_TYPE case标签下小心处理指向成员函数的指针。

在C++中，数组类型没有被限定，而是数组元素的类型被限定。 这种情况反映在中间表示中。这里描述的宏在应用到数组类型时， 将总是检验元素类型的限定符。（如果元素类型本身是一个数组，则会递归执行只到找到非数组类型，并检验该类型的限定符）所以， 例如，CP_TYPE_CONST_P当表示具有七个int的数组时， 将持有const int ()[7]类型。

下列函数和宏处理cv-qualification的类型：

CP_TYPE_QUALS
: 该宏返回应用到该类型的类型限定符集。 如果没有应用限定符则该值为TYPE_UNQUALIFIED。 如果类型是const的，则会设置TYPE_QUAL_CONST位。 如果类型是volatile的，则会设置TYPE_QUAL_VOLATILE位。 如果类型是restrict的，则会设置TYPE_QUAL_RESTRICT位。

CP_TYPE_CONST_P
: 该宏当类型是const时有效。

CP_TYPE_VOLATILE_P
: 该宏当类型是volatile时有效。

CP_TYPE_RESTRICT_P
: 该宏当类型是restrict时有效。

CP_TYPE_CONST_NON_VOLATILE_P
: 该断言当类型是const的，但不是volatile的时有效。 其它cv-qualifiers会被忽略，只测试const。

TYPE_MAIN_VARIANT
: 该宏类型未限定的版本。可以用于一个未限定的类型，但这种情况下， 并不总是标识符函数。

一些其它的宏和函数可用于所有的类型：

TYPE_SIZE
: 类型表示所需要的位数，为一个INTEGER_CST。 对于不完全类型，TYPE_SIZE将为NULL_TREE。

TYPE_ALIGN 类型的对齐位数，为一个int。

TYPE_NAME 该宏返回类型的一个声明（按照TYPE_DECL的型式）。（注意该宏不返回IDENTIFIER_NODE）你可以查看TYPE_DECL的DECL_NAME来获得类型的实际的名字。TYPE_NAME将为NULL_TREE，对于不是内建类型的，typedef的，或者命名的class类型。

CP_INTEGRAL_TYPE
该断言有效，如果类型为一个整数类型。注意在C++中，枚举不是整数类型。

ARITHMETIC_TYPE_P
该断言有效，如果类型为一个整数类型（按照c++的观点）或者一个浮点类型。

CLASS_TYPE_P
该断言有效，对于一个class类型。

TYPE_BUILT_IN
该断言有效，对于一个内建类型。

TYPE_PTRMEM_P
该断言有效，如果类型为一个指向数据成员的指针。

TYPE_PTR_P 该断言有效，如果类型为一个指针，而指向者不是一个数据成员。

TYPE_PTRFN_P
该断言有效，对于一个执行函数类型的指针。

TYPE_PTROB_P
该断言有效，对于一个指向object类型的指针。注意其对于指向object类型void *的通用指针无效。你可以使用TYPE_PTROBV_P来测试指针是指向object类型，同时也是void *。

TYPE_CANONICAL
该宏为给定的类型节点返回“正则”类型。正则类型用于C++和Objective-C的前端，使得在same_type_p中，可以对两个类型进行高效的比较，从而提高性能：如果类型的TYPE_CANONICAL值相等，则类型是等价的；否则，类型不相等。关于正则类型的相等的概念，与在语言中类型相等的概念是一样的。例如，

当TYPE_CANONICAL为NULL_TREE，则对于给定的类型节点，没有正则类型。这种情况下，比较该类型和任何其它类型，需要编译器执行深入的，“结构化的”比较，来看两个类型节点是否具有相同的形式和属性。

节点的正则类型，在等价的类型类别中，总是最根本的类型。例如，int为其自己的正则类型。int的自定义类型I，将int作为它的正则类型。类似的，I*和自定义类型IP（定义为I*），将int*作为它们的正则类型。当构建一个新的类型节点时，要记着将TYPE_CANONICAL设置为合适的正则类型。如果新类型为一个复合类型（从其它类型中构建的），并且那些类型中的任意一个需要进行结构化相等，则使用SET_TYPE_STRUCTURAL_EQUALITY来确保新的类型也需要结构化相等。最后，如果出于某种原因，你不能确保TYPE_CANONICAL能指向正则类型，则使用SET_TYPE_STRUCTURAL_EQUALITY来确保新的类型，任何基于它构建的类型，则需要结构化相等。如果你怀疑正则类型系统进行了错误的类型比较，则将--param verify-canonical-types=1传给编译器，或者使用--enable-checking来configure，强制编译器基于结构化比较来验证它的正则类型比较；如果正则类型比较有出入，则编译器将打印出警告信息。

TYPE_STRUCTURAL_EQUALITY_P
: 当节点需要进行结构相等检查的时候，该断言成立，例如，当TYPE_CANONICAL为NULL_TREE时。

SET_TYPE_STRUCTURAL_EQUALITY
: 该宏规定给定的类型节点需要进行结构相等检查，例如，其将TYPE_CANONICAL设为NULL_TREE

same_type_p
: 该断言接受两个类型作为输入，如果它们为相同的类型，则判断成立。例如，如果一个类型为另一个类型的typedef，或者这两个都为同一个类型的typedef。如果作为输入给定的两个tree，只是简单的为另一个的复制，则该断言也成立；即，它们在源代码级别没有差别，但是，出于某种原因，在表示的时候进行了复制。不要用==（pointer equality）来比较类型；要用same_type_p。

下面详细介绍了各种类型，以及可以用来访问它们的宏。虽然有其它类型是在G++中用到，这里描述的类型将只是当你检查中间表示时会遇到的。

VOID_TYPE 用于表示void类型。

INTEGER_TYPE
: 用来表示跟中整数类型，包括char，short，int，long和long long。该代码不用于枚举类型和bool类型。TYPE_PRECISION为用于表示该类型的位数，是一个unsigned int。（注意，通常情况它的值与TYPE_SIZE不相同；假设有一个24位的整数类型，但是ABI要求32位的对齐方式。则，TYPE_SIZE将为一个INTEGER_CST，32，而TYPE_PRECISION为24。）如果TYPE_UNSIGNED成立，则整数类型为无符号的；否则为有符号的。

  TYPE_MIN_VALUE为一个INTEGER_CST，是该类型可以表示的最小整数。类似的，TYPE_MAX_VALUE为一个INTEGER_CST，是该类型可以表示的最大整数。

REAL_TYPE 用来表示类型float，double和long double。跟INTEGER_TYPE的情况类型，用于浮点表示的位数由TYPE_PRECISION给出。

FIXED_POINT_TYPE
: 用来表示类型short _Fract, _Fract, long _Fract, long long _Fract, short _Accum, _Accum, long _Accum, 和 long long _Accum。跟INTEGER_TYPE的情况类型，用于定点表示的位数由TYPE_PRECISION给出。可能会有填充位，小数位和整数位。小数的位数由TYPE_FBIT给出，整数的位数由TYPE_IBIT给出。如果TYPE_UNSIGNED成立，则定点类型为无符号的；否则为有符号的。

  如果TYPE_SATURATING成立，则定点类型为饱和的；否则不是饱和的。。

COMPLEX_TYPE
: 用来表示GCC内建的__complex__数据类型。TREE_TYPE为实部和虚部的类型。

ENUMERAL_TYPE
: 用于表示枚举类型。TYPE_PRECISION给出了用于表示该类型的位数（为一个int）。如果没有负数的枚举常量，则TYPE_UNSIGNED成立。最小和最大的枚举常量，可以分别使用TYPE_MIN_VALUE和TYPE_MAX_VALUE获得；每个宏都返回一个INTEGER_CST。

  实际的枚举常量可以通过查看TYPE_VALUES来获得。该宏将返回一个TREE_LIST，包含了常量；TREE_VALUE将为一个INTEGER_CST，给出了赋予那个常量的值。这些常量将按照它们被声明的顺序出现。每个常量的TREE_TYPE，将为枚举类型本身的类型。

BOOLEAN_TYPE
: 用来表示bool类型。

POINTER_TYPE

用来表示指针类型，以及指向数据成员的指针类型。TREE_TYPE给出了所指向的类型。如果类型为一个指向数据成员的指针，则TYPE_PTRMEM_P成立。对于一个指向`T X::*'形式的数据成员类型的指针，TYPE_PTRMEM_CLASS_TYPE将为类型X，而TYPE_PTRMEM_POINTED_TO_TYPE为类型T。

REFERENCE_TYPE

用来表示引用类型。TREE_TYPE给出了所引用的类型。

FUNCTION_TYPE

用来表示非成员函数和静态成员函数的类型。TREE_TYPE给出了函数的返回值类型。TYPE_ARG_TYPES为一个TREE_LIST，参数类型列表。该列表上的每个节点的TREE_VALUE为相应参数的类型；TREE_PURPOSE如果存在，则为缺省参数值的表达式。如果列表中的最后一个节点为void_list_node（一个TREE_LIST节点，其TREE_VALUE为void_type_node），则该类型的函数不接受可变参数。否则，它们是接受可变数目的参数。

注意，在C（而不是C++）中，一个声明如void f()的函数，为一个无函数原型的函数，其接受可变数目的参数；这样的函数的TYPE_ARG_TYPES将为NULL。

METHOD_TYPE

用来表示非静态成员函数的类型。像FUNCTION_TYPE一样，返回值由TREE_TYPE给出。*this的类型，即，这个函数成员所在的类的类型，由TYPE_METHOD_BASETYPE给出。TYPE_ARG_TYPES为参数列表，就像FUNCTION_TYPE一样，包含this参数。

ARRAY_TYPE 用于表示数组类型，TREE_TYPE给出了数组元素的类型。如果数组边界在类型中存在，则TYPE_DOMAIN为一个INTEGER_TYPE，它的TYPE_MIN_VALUE和TYPE_MAX_VALUE将分别为数组的下界和上界。TYPE_MIN_VALUE将总是为INTEGER_CST，0，而TYPE_MAX_VALUE将为数组元素数目减1，即可以用来索引数组元素的最大值。

RECORD_TYPE

用来表示struct和class类型，以及指向成员函数和其它语言中类似结构的指针。TYPE_FIELDS包含了该类型中包含的项，其可以为FIELD_DECL, VAR_DECL, CONST_DECL或TYPE_DECL。你不能假设类型中的域之间的顺序，以及它们是否有重叠。如果TYPE_PTRMEMFUNC_P成立，则该类型为一个指向成员（pointer-to-member）的类型。这种情况下，TYPE_PTRMEMFUNC_FN_TYPE为一个POINTER_TYPE，指向一个METHOD_TYPE。METHOD_TYPE为由pointer-to-member函数指向的函数的类型。如果TYPE_PTRMEMFUNC_P不成立，则该类型为一个class类型。更多信息，参见see Section 9.4.2 [类], page 86。

UNION_TYPE 用来表示union类型。类似于RECORD_TYPE，只不过在TYPE_FIELD中的所有FIELD_DECL起始于0位。

QUAL_UNION_TYPE

用来表示Ada中的可变记录（variant record）的一部分。类似于UNION_TYPE，只不过每个FIELD_DECL具有一个DECL_QUALIFIER域，其包含了一个布尔表达式，用来指示该域是否在对象中存在。该类型将只有一个域，所以只有当TYPE_FIELDS中，先前的域中的表达式都不为零时，每个域的DECL_QUALIFIER才被求值。通常，这些表达式将使用PLACEHOLDER_EXPR引用外部对象的一个域。

UNKNOWN_TYPE

该节点用来表示一个类型，其信息不足以进行合理的处理。

OFFSET_TYPE
: 该节点用于表示一个pointer-to-data成员。对于一个数据成员X::m，则TYPE_OFFSET_BASETYPE为X，TREE_TYPE为m类型.

TYPENAME_TYPE
: 用于表示typename T::A。TYPE_CONTEXT为T；TYPE_NAME为A的IDENTIFIER_NODE。如果类型通过模板id指定，则TYPENAME_TYPE_FULLNAME产生一个TEMPLATE_ID_EXPR。如果节点是被隐式的生成，用来支持隐式类型名扩展，则TREE_TYPE不为NULL；这种情况下，TREE_TYPE为基类的类型节点。

TYPEOF_TYPE
: 用于表示__typeof__扩展。TYPE_FIELDS为被表示的类型的表达式。

有些变量，它们的值表示基本的类型。这包括：

void_type_node
: void节点。

integer_type_node
: int节点。

unsigned_type_node.
: unsigned int节点。

char_type_node.
: char节点。

有时使用same_type_p将这些变量和手头的类型进行比较会很有帮助。

9.4 作用域

整个中间表示的根是变量global_namespace。这是在C++源代码中由::描述的命名空间。所有其它命名空间，类型，变量，函数等，都能够从这里开始查找而获得。

除了命名空间以外，C++中另一个高层次的作用域结构是类。（在该手册中，术语class用来表示ANSI/ISO C++标准中的class类型；这包括用class, struct和union关键字定义的类型。）

9.4.1 命名空间

命名空间由NAMESPACE_DECL节点表示。

然而，除了实际上是作为根表示以外，全局命名空间与其它命名空间没有区别。因此，在下文中，我们描述的是普遍的命名空间，而不是特定的全局命名空间。

下列宏和函数可以用于NAMESPACE_DECL:

DECL_NAME
: 该宏用于获得IDENTIFIER_NODE相应的命名空间的未限定名（参见 Section 9.2.2 [标识符], page 80）。全局命名空间的名字为`::'，虽然在C++中全局命名空间是没有名字的。然而，你应该使用与global_namespace比较的方式，而不是用DECL_NAME来确定命名空间是否为全局的。一个未命名的命名空间，其DECL_NAME等于anonymous_namespace_name。在一个单独的转换单元中，所有未命名空间将具有同一名字。

DECL_CONTEXT
: 该宏返回闭包的（enclosing）命名空间。global_namespace的DECL_CONTEXT为NULL_TREE。

DECL_NAMESPACE_ALIAS

如果该声明是一个命名空间的别名，则DECL_NAMESPACE_ALIAS为该别名所针对的命名空间。

不要对别名命名空间尝试使用cp_namespace_decls。相反的，沿着DECL_NAMESPACE_ALIAS链直到一个普通的，非别名的命名空间，然后在那里调用cp_namespace_decls。

DECL_NAMESPACE_STD_P

该断言当命名空间为特殊的::std命名空间时有效。

cp_namespace_decls

该函数将返回包含在命名空间中的声明，包括类型，重载函数，其它命名空间等等。如果没有声明，该函数将返回NULL_TREE。声明通过它们的TREE_CHAIN域连在一起。

虽然这个链表中的大多数实体将为声明，但也可能会出现TREE_LIST。这种情况下，TREE_VALUE将为一个OVERLOAD。TREE_PURPOSE的值未指定；后端应该忽略这个值。至于由cp_namespace_decls返回的其它种类的声明，TREE_CHAIN将会指向该链表中的下一个声明。

关于可以出现在该链表中的各种声明的更多信息，参见 Section 9.5 [声明], page 87。一些声明将不会出现在该链表中。特别是，FIELD_DECL, LABEL_DECL和PARM_DECL节点。

该函数不能用于设置了DECL_NAMESPACE_ALIAS的命名空间。

9.4.2 类

类的类型被表示为RECORD_TYPE或者UNION_TYPE。使用union标签声明的类由UNION_TYPE来表示，而使用struct或者class标签声明的类由RECORD_TYPE来表示。可以使用CLASSTYPE_DECLARED_CLASS宏来判定一个特定类型是class的还是struct的。该宏只在类使用class标签声明时才为真。

几乎所有非函数的成员都在TYPE_FIELDS列表中。给出一个成员，可以通过TREE_CHAIN来找到下一个。不要依赖于在该链表中出现的域的顺序。链表中的所有节点将为`DECL'节点。FIELD_DECL用于表示非静态数据成员，VAR_DECL用于表示静态数据成员，而TYPE_DECL用于表示类型。注意如果在类中声明了枚举类型，则用于枚举常量的CONST_DECL将会出现在该链表中。（当然，枚举类型的TYPE_DECL也会出现。）链表中没有基类的实体。特别是，对于一个对象的基类部分“base-class portion”，没有FIELD_DECL。

TYPE_VFIELD是编译器生成的域用于指向虚函数表。它有可能会出现在TYPE_FIELDS链表中。然而，后端应该处理TYPE_VFIELD，就像TYPE_FIELDS链表中所有其它实体一样。

函数成员在TYPE_METHODS链表中。同样，后续成员可以通过TREE_CHAIN域来找到。如果函数是重载的，每个重载函数都会出现；OVERLOAD节点不会出现在TYPE_METHODS链表上。隐式声明的函数（包括缺省构造函数，复制构造函数，赋值操作和析构函数）也会出现在该链表中。

每个类都具有一个相关的binfo，其能够通过TYPE_BINFO获得。Binfos用于表示基类。由TYPE_BINFO给出的binfo是退化情况，让每个类被考虑为它自己的基类。

对基类型的访问可以通过BINFO_BASE_ACCESS。这将产生access_public_node, access_private_node或access_protected_node。如果基类总是public的，BINFO_BASE_ACCESSES可以为NULL。

BINFO_VIRTUAL_P用于指定binfo是否为虚继承。其它标识，BINFO_MARKED_P和BINFO_FLAG_1到BINFO_FLAG_6可以用于语言特定用途。

下列宏可以用在表示class类型的树节点上。

LOCAL_CLASS_P
: 该断言当是局部类时有效，即在函数体内声明的类。

TYPE_POLYMORPHIC_P
: 该断言当类至少具有一个虚函数（声明的或者继承的）时有效。

TYPE_HAS_DEFAULT_CONSTRUCTOR
: 该断言当参数表示具有缺省构造函数的class类型时有效。

CLASSTYPE_HAS_MUTABLE
TYPE_HAS_MUTABLE_P
: 这些断言表示了一个class-type具有mutable数据成员。

CLASSTYPE_NON_POD_P
: 该断言表示了不是POD的class-types。

TYPE_HAS_NEW_OPERATOR
: 该断言表示一个class-type定义了operator new。

TYPE_HAS_ARRAY_NEW_OPERATOR
: 该断言表示一个class-type定义了operator new[]。

TYPE_OVERLOADS_CALL_EXPR
: 该断言表示class-type重载了函数调用operator()。

TYPE_OVERLOADS_ARRAY_REF
: 该断言表示class-type重载了operator[]。

TYPE_OVERLOADS_ARROW
: 该断言表示class-type重载了operator->。

9.5 声明

这一节涵盖了出现在内部表示中的各种声明。 函数声明（由FUNCTION_DECL节点表示）除外，其在Section 9.6 [函数], page 91一节中描述。

9.5.1 关于声明的操作

一些宏可以用于任何种类的声明。这包括：

DECL_NAME
: 该宏返回一个IDENTIFIER_NODE，给出了实体的名字。

TREE_TYPE
: 该宏返回被声明的实体的类型。

TREE_FILENAME
: 该宏返回被声明的实体所在的文件的名字，作为一个char*。对于由编译器隐式声明的实体（比如__builtin_memcpy），这将为字符串"<internal>"。

TREE_LINENO
: 该宏返回被声明的实体所在的行号，作为一个int。

DECL_ARTIFICIAL
: 该断言用来表示声明是否为编译器隐式产生的。例如，该断言可以用来存放一个隐式声明的成员函数，或者为一个class类型隐式生成的TYPE_DECL。回想一下在C++代码中：

```
struct S {};
```

大体上等价于C代码：

```
struct S {};
typedef struct S S;
```

隐式生成的typedef声明由TYPE_DECL表示，在DECL_ARTIFICIAL中存放。

DECL_NAMESPACE_SCOPE_P
: 该断言用来表示实体是否在一个命名空间中被声明。

DECL_CLASS_SCOPE_P
: 该断言用来表示实体是否在一个class作用域中被声明。

DECL_FUNCTION_SCOPE_P
: 该断言用来表示实体是否在一个函数体中被声明。

各种类型的声明：

LABEL_DECL
: 这些节点用于表示函数体内的标号。更多信息，可以参考Section 9.6 [函数], page 91。这些节点只出现在块作用域（block scope）。

CONST_DECL
: 这些节点用于表示枚举常量。常量的值由DECL_INITIAL给出，为一个INTEGER_CST，并且和CONST_DECL的TREE_TYPE具有相同的类型，即ENUMERAL_TYPE。

RESULT_DECL
: 这些节点表示函数的返回值。当RESULT_DECL被赋予一个值的时候，这表明该值将被函数通过按位拷贝的方式返回。就像对于VAR_DECL一样，你可以在RESULT_DECL上使用DECL_SIZE和DECL_ALIGN。

TYPE_DECL
: 这些节点表示typedef声明。TREE_TYPE为被声明的类型，其名字由DECL_NAME给出。有些情况下，没有相关联的名字。

VAR_DECL
: 这些节点表示具有命名空间或者块作用域的变量，以及静态数据成员。DECL_SIZE和DECL_ALIGN，类似于TYPE_SIZE和TYPE_ALIGN。对于声明，你应该总是使用DECL_SIZE和DECL_ALIGN，而不是由TREE_TYPE给定的TYPE_SIZE和TYPE_ALIGN，因为，可能会对变量应用了特定的属性，使其具有特定的大小和对齐方式。你可以使用断言DECL_THIS_STATIC或DECL_THIS_EXTERN来测试是否使用了存储类别说明符static或extern来声明一个变量。

  如果该变量被初始化（并需要一个构造者），则DECL_INITIAL将为初始化者的表达式。初始化者将被求值，并按位复制到变量中。如果DECL_INITIAL为error_mark_node，则表明存在一个初始化者，只不过它由之后代码中的显式的语句给出；这将不需要进行按位复制。

  GCC提供了一个扩展，允许自动变量或者全局变量，被放到特定的寄存器中。如果DECL_REGISTER存放了VAR_DECL，并且DECL_ASSEMBLER_NAME不等于DECL_NAME，则VAR_DECL是使用了该扩展。这种情况下，DECL_ASSEMBLER_NAME为存放变量的寄存器的名字。

PARM_DECL
: 用于表示一个函数的参数。这些节点可以作为VAR_DECL节点来看待。这些节点只出现在FUNCTION_DECL的DECL_ARGUMENTS中。

  PARM_DECL的DECL_ARG_TYPE为，当值传给函数时实际使用的类型。其可以为一个比参数的TREE_TYPE更宽的类型；例如，原始类型可能为short，而DECL_ARG_TYPE为int。

FIELD_DECL
: 这些节点表示非静态数据成员。DECL_SIZE和DECL_ALIGN的行为，跟VAR_DECL节点的一样。在父记录（parent record）中的域的位置，由三个属性组合指定。DECL_FIELD_OFFSET为按字节计数的位置，

如果DECL_C_BIT_FIELD有效，则该域是一个位域。在位域中，DECL_BIT_FIELD_TYPE还包含了原始指定的类型，而DECL_TYPE可能是根据位域的大小，修改后具有更少精度的类型。

NAMESPACE_DECL
See Section 9.4.1 [命名空间], page 85.

TEMPLATE_DECL
这些节点用于表示类，函数和变量（静态数据成员）模板。DECL_TEMPLATE_SPECIALIZATIONS为一个TREE_LIST。列表中每个节点的TREE_VALUE为一个TEMPLATE_DECL或者FUNCTION_DECL，表示该模板的特列（包括实例）。后端可以安全的忽略TEMPLATE_DECL，但是应该检查特例列表中的FUNCTION_DECL，就像是普通的FUNCTION_DECL节点一样。

对于类模板，DECL_TEMPLATE_INSTANTIATIONS列表包含了实例。每个节点的TREE_VALUE为类的一个实例。DECL_TEMPLATE_SPECIALIZATIONS包含了类的部分特例。

USING_DECL 后端可以安全的忽略掉这些节点。

9.5.2 内部结构

DECL节点在内部被表示为层次结构体。

9.5.2.1 目前的结构层次

struct tree_decl_minimal
这是用于继承的最小结构体，从而使得DECL宏更加通用。 所包含的域为一个唯一的ID，源位置，上下文和名字。

struct tree_decl_common
该结构体继承于struct tree_decl_minimal。 包含了大多DECL节点需要的域，像存储对齐信息，机器模式，大小和属性的域。

struct tree_field_decl
该结构体继承于struct tree_decl_common。用于表示FIELD_DECL。

struct tree_label_decl
该结构体继承于struct tree_decl_common。用于表示LABEL_DECL。

struct tree_translation_unit_decl
该结构体继承于struct tree_decl_common。 用于表示TRANSLATION_UNIT_DECL。

struct tree_decl_with_rtl
该结构体继承于struct tree_decl_common。 包含了一个存储与DECL节点相关的低层次RTL。

struct tree_result_decl
该结构体继承于struct tree_decl_with_rtl。 用于表示RESULT_DECL。

struct tree_const_decl
该结构体继承于struct tree_decl_with_rtl。 用于表示CONST_DECL。

struct tree_parm_decl
该结构体继承于struct tree_decl_with_rtl。 用于表示PARM_DECL。

struct tree_decl_with_vis
: 该结构体继承于struct tree_decl_with_rtl。 包含了存储可视性信息所需要的域，还有一个section名和汇编名。

struct tree_var_decl
: 该结构体继承于struct tree_decl_with_vis。 用于表示VAR_DECL。

struct tree_function_decl
: 该结构体继承于struct tree_decl_with_vis。 用于表示FUNCTION_DECL。

9.5.2.2 添加新的DECL节点类型

增加一个新的DECL树包含下列步骤

为DECL节点增加一个新的树代码
: 对于语言特定的DECL节点，在每个前端目录下有一个`.def'文件，用来添加树代码。对于属于中端一部分的DECL节点，代码应该添加到`tree.def'中。

为DECL节点创建一个新的结构体类型
: 这些结构体应该继承于现有的层次结构体，方法是使用该结构体作为第一个成员。

```
struct tree_foo_decl
{
  struct tree_decl_with_vis common;
}
```

将会创建一个名为tree_foo_decl的结构体，继承于struct tree_decl_with_vis。

对于语言特定的DECL节点，这个新的结构体类型应该放在合适的`.h'文件中。对于属于中端一部分的DECL节点，结构体类型应该在`tree.h'中。

向树结构枚举中增加一个节点成员
: 出于垃圾搜集和动态检查的目的，每个DECL节点结构体类型需要具有一个唯一的枚举值用来指定它。对于语言特定的DECL节点，该新的枚举值应该在合适的.def文中。对于属于中端的DECL节点，枚举值在`treestruct.def'中指定。

更新union tree_node
: 为了使得新的结构体类型可用，其必须被添加到union tree_node中。对于语言特定的DECL节点，一个新的项应该被添加到合适的`.h'文件中，形式如下

```
 struct tree_foo_decl GTY ((tag ("TS_VAR_DECL"))) foo_decl;
```

对于属于中端的DECL节点，额外的成员直接放在`tree.h'里的union tree_node中。 For DECL nodes that are part of the middle-end, the additional

更新动态检查信息
: 为了能够检查访问union tree_node的一个命名部分是否合法，以及特定的DECL节点是否包含了枚举的DECL节点结构体，我们使用了一个简单的查找表。该查找表需要随着树结构层次一起更新，否则用于检查和包含的宏将会失败。

对于语言特定的DECL节点，它们是在合适的`.c'文件中的init_ts函数，用于初始化查找表。为新的DECL节点建立表格的代码应该添加到这里。对于每个表示继承层次的成员的DECL树代码和枚举值，如果树代码（直接或间接）继承于那个成员，则表应该包含1。因此，一个源于struct decl_with_rtl的，枚举值为TS_FOO_DECL的FOO_DECL节点，会使用下列方式来建立。

```
tree_contains_struct[FOO_DECL][TS_FOO_DECL] = 1;
tree_contains_struct[FOO_DECL][TS_DECL_WRTL] = 1;
```

```
tree_contains_struct[FOO_DECL][TS_DECL_COMMON] = 1;
tree_contains_struct[FOO_DECL][TS_DECL_MINIMAL] = 1;
```

对于属于中端的DECL节点，建表代码在`tree.c'中。

增加访问任何新的域和标识的宏

每个增加的域或标识，都应有一个宏用来访问它，并且执行适当的检查以保证访问的是正确类型的DECL。

这些宏通常采用下面的形式

```
#define FOO_DECL_FIELDNAME(NODE) FOO_DECL_CHECK(NODE)->foo_decl.fieldname
```

但是，如果结构体只是更多结构体的一个基类，有时会使用下面的形式

```
#define BASE_STRUCT_CHECK(T) CONTAINS_STRUCT_CHECK(T, TS_BASE_STRUCT)
#define BASE_STRUCT_FIELDNAME(NODE) \
  (BASE_STRUCT_CHECK(NODE)->base_struct.fieldname
```

9.6 函数

函数由FUNCTION_DECL节点表示。 重载函数集有时通过一个OVERLOAD节点来表示。

OVERLOAD节点不是一个声明。 所以没有`DECL_'宏用于OVERLOAD。 OVERLOAD节点类似于一个TREE_LIST。 使用OVL_CURRENT来获得OVERLOAD节点关联的函数； 使用OVL_NEXT来获得重载函数列表中下一个OVERLOAD节点。 宏OVL_CURRENT和OVL_NEXT实际上是多态的； 你可以用它们工作于FUNCTION_DECL节点上，就像在重载上一样。 在FUNCTION_DECL的情况下，OVL_CURRENT将总是返回函数本身， OVL_NEXT将总是为NULL_TREE。

要确定函数的作用域，可以使用DECL_CONTEXT宏。 该宏将返回函数作为其成员的类（或者RECORD_TYPE或者UNION_TYPE）或命名空间（NAMESPACE_DECL）。对于虚函数， 该宏返回函数被实际定义的类，而不是其声明所在的基类。

如果友函数定义在类的作用域里， 宏DECL_FRIEND_CONTEXT可以用来判定其定义所在的类。例如，在

```
class C { friend void f() {} };
```

中，f的DECL_CONTEXT将为global_namespace， 而C的DECL_FRIEND_CONTEXT将为RECORD_TYPE。

在C中，函数的DECL_CONTEXT可能为另一个函数。 这表示正在使用GNU嵌套函数扩展功能。关于嵌套函数语法的详细内容， 参见GCC手册。嵌套函数可以引用其所包含的函数的局部变量。 这样的引用没有在树结构体里被显示的标记。 如果被引用VAR_DECL的DECL_CONTEXT与当前被处理的函数不相同， 并且DECL_EXTERNAL和DECL_STATIC都没有持有内容， 则该引用是针对包含的函数的局部变量，后端必须采取合适的行为。

9.6.1 函数基础

下列宏和函数能够用于FUNCTION_DECL:

DECL_MAIN_P

该断言判断一个函数是否为程序的入口点::code。

DECL_NAME 该宏返回函数未限定的名字，为一个IDENTIFIER_NODE。 对于一个函数模版的实例，DECL_NAME为模版的未限定名字， 而不是类似f<int>的东西。当用在编译器隐式生成的构造函数， 析构函数，重载操作符，或者类型转换符，或者任何函数时， DECL_NAME的值未定义。关于可以用来区分这些情况的宏，参见下面。

DECL_ASSEMBLER_NAME

该宏返回函数的mangled名字，也是一个IDENTIFIER_NODE。该名字没有包含前导的下划线。mangled名字在所有平台上按照相同的方式来计算；如果在特定的平台上，需

要对目标文件格式进行特殊的处理，则后端需要负责执行那些修改。（当然，后端不应该修改DECL_ASSEMBLER_NAME）

使用DECL_ASSEMBLER_NAME将使得额外的内存被分配（用于实体的mangled名字），所以其应该只当生成汇编代码时被使用。其不应该在优化器中被使用，用于确定两个声明是否相同，即使一些现有的优化器确实采用了这种方式。这些使用将随着时间被移除。

DECL_EXTERNAL
: 该断言判断函数是否未定义。

TREE_PUBLIC
: 该断言判断函数是否具有外部连接。

DECL_LOCAL_FUNCTION_P
: 该断言判断函数是否声明在块作用域中，即使具有全局作用域。

DECL_ANTICIPATED
: 该断言判断函数是否为built-in函数，并且函数原形没有显示声明。

DECL_EXTERN_C_FUNCTION_P
: 该断言判断函数是否声明为`extern "C"'函数。

DECL_LINKONCE_P
: 该宏用来判断该函数是否会在不同的转换单元产生多个副本。 合并不同副本是连接器的责任。 模版实例是使用DECL_LINKONCE_P最常见的例子。 G++会在需要它们的所有转换单元里实例化模版， 然后依赖连接器去移除重复的实例。

  FIXME: 该宏还没有实现。

DECL_FUNCTION_MEMBER_P
: 该宏用来判断函数是否为一个类的成员，而不是命名空间的成员。

DECL_STATIC_FUNCTION_P
: 该断言用来判断函数是否为一个静态成员函数。

DECL_NONSTATIC_MEMBER_FUNCTION_P
: 该宏用来判断是否为一个非静态成员函数。

DECL_CONST_MEMFUNC_P
: 该断言用来判断是否为const成员函数。

DECL_VOLATILE_MEMFUNC_P
: 该断言用来判断是否为volatile成员函数。

DECL_CONSTRUCTOR_P
: 该宏用来判断函数是否为一个构造函数。

DECL_NONCONVERTING_P
: 该断言用来判断构造函数是否为一个非转换构造函数。

DECL_COMPLETE_CONSTRUCTOR_P
: 该断言用来判断函数是否为完全类型的对象的构造函数。

DECL_BASE_CONSTRUCTOR_P
: 该断言用来判断函数是否为一个基类的子对象的构造函数。

DECL_COPY_CONSTRUCTOR_P
: 该断言用来判断函数是否为一个复制构造函数。

DECL_DESTRUCTOR_P
: 该宏用来判断函数是否为一个析构函数。

DECL_COMPLETE_DESTRUCTOR_P
: 该断言用来判断函数是否为一个完全类型的对象的析构函数。

DECL_OVERLOADED_OPERATOR_P
: 该宏用来判断函数是否为一个重载操作符。

DECL_CONV_FN_P
: 该宏用来判断函数是否为一个类型转换操作符。

DECL_GLOBAL_CTOR_P
: 该断言用来判断函数是否为一个文件作用域的初始化函数。

DECL_GLOBAL_DTOR_P
: 该断言用来判断函数是否为一个文件作用域的结束化函数。

DECL_THUNK_P
: 该断言用来判断函数是否为一个thunk。

  这些函数表示stub代码，用来调整this指针，然后跳转到另一个函数中。当从被跳转的函数中返回时，控制被直接传给调用者，不需要返回到thunk中。thunk的第一个参数总是为this指针；thunk应该将该值加上THUNK_DELTA。（THUNK_DELTA是一个int，而不是INTEGER_CST。）

  然后，如果THUNK_VCALL_OFFSET（一个INTEGER_CST）是非零的，则被调整的this必须再次被调整。下面的伪代码给出了完整的计算：

```
this += THUNK_DELTA
if (THUNK_VCALL_OFFSET)
 this += (*((ptrdiff_t **) this))[THUNK_VCALL_OFFSET]
```

  最终，thunk应该跳转到由DECL_INITIAL给出的位置；这将总是一个函数地址的表达式。

DECL_NON_THUNK_FUNCTION_P
: 该断言用来判断函数*不是*一个thunk函数。

GLOBAL_INIT_PRIORITY
: 如果DECL_GLOBAL_CTOR_P或者DECL_GLOBAL_DTOR_P有效，则该宏给出了函数的初始优先级。连接器将设法安排DECL_GLOBAL_CTOR_P所保存的所有的函数在递增的顺序下运行，在main被调用之前。当程序退出时，DECL_GLOBAL_DTOR_P所保存的所有函数按照相反的顺序执行。

DECL_ARTIFICIAL
: 该宏保存了函数是否由编译器隐式的生成，而不是被显式的生成。 除了隐式的生成的类成员函数以外，该宏还保存了创建的特定函数， 用来实现静态初始化和析构，来计算运行时信息等等。

DECL_ARGUMENTS
: 该宏返回函数第一个参数的PARM_DECL。 后续的PARM_DECL节点可以通过TREE_CHAIN来获得。

DECL_RESULT
: 该函数返回函数的RESULT_DECL。

TREE_TYPE
: 该宏返回函数的FUNCTION_TYPE或METHOD_TYPE。

TYPE_RAISES_EXCEPTIONS
: 该宏返回（成员）函数可以引起的异常列表。返回的列表，如果不是NULL，则为TREE_VALUE代表一个类型的节点组成。

TYPE_NOTHROW_P
: 该断言用来判断是否是使用`()'形式来指定异常的参数。

DECL_ARRAY_DELETE_OPERATOR_P
: 该断言用来判断函数是否为一个重载的operator delete[]。

DECL_FUNCTION_SPECIFIC_TARGET
: 该宏返回一个tree节点，存放了用于编译该特定函数的目标机选项，或者为NULL_TREE，如果是使用命令行中指定的目标机选项来编译该函数。

DECL_FUNCTION_SPECIFIC_OPTIMIZATION
: 该宏返回一个tree节点，存放了用于编译该特定函数的优化选项，或者为NULL_TREE，如果是使用命令行中指定的优化选项来编译该函数。

9.6.2 函数体

在当前转换单元中定义的函数将会有一个非NULL的DECL_INITIAL。 但是，后端不应该使用DECL_INITIAL给出的该特定值。

宏DECL_SAVED_TREE将会给出完整的函数体。

9.6.2.1 语句

C和C++前端，使用了对应于所有源级的语句构造的树节点。 这些都列举在这里，并附上能够用来获得它们的信息的各种宏的列表。 有一些宏能用于所有的语句：

STMT_IS_FULL_EXPR_P
: 在C++里，语句通常构成“充分表达式（full expressions）”；在语句中创建的临时事物会在声明完成时被销毁。 然而，G++有时通过语句来表示表达式； 这些语句将不会设置STMT_IS_FULL_EXPR_P。 在这样的语句中创建的临时事物将会在最内层设置了STMT_IS_FULL_EXPR_P的语句退出时被销毁。

这里有一个各种语句节点的列表，以及访问它们的宏。 该文档描述了在非模板函数（包括模板函数的实例）中这些节点的用法。 在模板函数里，使用相同的节点，但是有时方法略有不同。

许多语句具有子语句。例如，一个while循环将会有一个循环体， 其本身也是一个语句。如果子语句是NULL_TREE， 则被认为相当于一个;组成的语句，即一个表达式语句，其中表达式为空。 一个子语句实际上可以为一个语句列表，通过它们的TREE_CHAIN连接一起。 所以，你应该总是通过循环所有的子语句来处理语句树，像这样：

```
void process_stmt (stmt)
   tree stmt;
{
 while (stmt)
  {
   switch (TREE_CODE (stmt))
    {
    case IF_STMT:
     process_stmt (THEN_CLAUSE (stmt));
     /* More processing here. */
     break;

    ...
```

```
        }

      stmt = TREE_CHAIN (stmt);
    }
  }
```

换句话说，虽然C++中if语句的then子句可以只是一个语句， 但中间表示有时会使用多个语句连接在一起。

ASM_EXPR 用来表示一条内联的汇编语句。一条内联汇编语句形如：

```
asm ("mov x, y");
```

ASM_STRING宏将会为"mov x, y"返回一个STRING_CST节点。 如果原始的语句使用了扩展汇编语法，则ASM_OUTPUTS,ASM_INPUTS和ASM_CLOBBERS为用STRING_CST表示的语句的输出， 输入和clobber。扩展汇编语法形如：

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

第一个字符串是ASM_STRING，包含指令模板。 接下来的两个字符串分别是输出和输入。该语句没有clobbers。 这个例子表明，普通的汇编语句只是扩展汇编语句的一个特例； 它们没有限定符，输出，输入或者clobbers。 所有的字符串都为NUL结尾，并且不包含嵌入的NUL字符。

如果汇编语句被声明为volatile，或者语句不是扩展汇编语句， 因此是一个隐式的volatile， 则断言ASM_VOLATILE_P将会保存ASM_EXPR。

BREAK_STMT 用来表示一条break语句。没有额外的域。

CASE_LABEL_EXPR

用来表示一个case标号，case标号的范围或者一个default标号。 如果CASE_LOW是NULL_TREE，则为一个default标号。 否则，如果CASE_HIGH是NULL_TREE，则为一个普通的case标号。 这种情况下，CASE_LOW是一个表达式，给出了标号的值。 CASE_LOW和CASE_HIGH都是INTEGER_CST节点。 这些值跟在switch语句中的条件表达式具有相同的类型。

否则，如果同时定义了CASE_LOW和CASE_HIGH，则语句为一个case标号的范围。这样的语句源于允许用户使用如下形式的扩展：

```
case 2 ... 5:
```

第一个值为CASE_LOW，第二个为CASE_HIGH。

CLEANUP_STMT

用来表示在封闭作用域的出口上执行的动作。 通常，这些动作是为局部对象调用析构函数，但是后端不能依靠这个事实。 如果这些节点确实是表示了这样的析构函数，CLEANUP_DECL将为销毁的VAR_DECL。 否则CLEANUP_DECL为NULL_TREE。 无论哪种情况，CLEANUP_EXPR都是要被执行的表达式。 清除工作在作用域的出口被执行， 并且按照所遇到的CLEANUP_STMT的相反顺序进行。

CONTINUE_STMT

用来表示一条continue语句。没有额外的域。

CTOR_STMT 用于标记构建函数体的起始（CTOR_BEGIN_P）或结尾（CTOR_END_P）。关于如何使用这些节点的更多信息，参见SUBOBJECT。

DECL_STMT 用来表示一个局部声明。宏DECL_STMT_DECL可以用来获得整个声明。 该声明可以为一个LABEL_DECL，表示声明了一个局部标号。 （作为扩展，GCC允许声明具有作用域的标号。 ）在C中，该声明可以为一个FUNCTION_DECL，表示使用GCC嵌套函数扩展。 更多信息，参见 Section 9.6 [函数], page 91。

DO_STMT 用来表示do循环。循环体由DO_BODY给出，终止条件由DO_COND给出。do语句的条件总是一个表达式。

EMPTY_CLASS_EXPR
用来表示类的临时对象。（所有这样的对象都是可互换的。）TREE_TYPE表示对象的类型。

EXPR_STMT 用来表示表达式语句。使用EXPR_STMT_EXPR来获得表达式。

FOR_STMT 用来表示一条for语句。FOR_INIT_STMT是循环的初始语句。FOR_COND是终止条件。FOR_INIT_STMT是在每次循环迭代FOR_COND之前执行的表达式，该表达式常常是增加计数器。循环体由FOR_BODY给出。注意FOR_INIT_STMT和FOR_BODY返回语句，而FOR_COND和FOR_EXPR返回表达式。

GOTO_EXPR 用来表示一条goto语句。GOTO_DESTINATION通常为一个LABEL_DECL。然而，如果使用了扩展的“computed goto”，将为一个随机表达式用来指示目的地。该表达式总是具有一个指针类型。

HANDLER 用来表示C++ catch块。HANDLER_TYPE是由该处理者捕捉的异常类型。如果该处理者是针对所有类型的，则等于（指针等于）NULL。HANDLER_PARMS是catch参数，是一个DECL_STMT。HANDLER_BODY是块本身的代码。

IF_STMT 用来表示一条if语句。IF_COND是表达式。如果条件是一个TREE_LIST，则TREE_PURPOSE是一条语句（通常为DECL_STMT）。每次评估条件的时候，都要执行该语句。然后，TREE_VALUE应该作为条件表达式本身来使用。该表示用来处理C++代码，如：

```
if (int i = 7) ...
```

其中，在条件中声明了一个（或多个）新的局部变量。

LABEL_EXPR 用来表示一个标号。可以通过LABEL_EXPR_LABEL宏获得该语句声明的LABEL_DECL。可以通过LABEL_DECL的DECL_NAME获得给出的标号名字。

RETURN_STMT
用来表示一条return语句。RETURN_EXPR是返回的表达式，其将会返回NULL_TREE，如果语句只是

```
return;
```

SUBOBJECT 在构造函数中，这些节点用来标记在某一点子对象被完全构造。如果，这一点之后，在遇到设置了CTOR_END_P的CTOR_STMT之前，有异常抛出，则必须执行SUBOBJECT_CLEANUP。清除工作必须按照它们出现的顺序反向执行。

SWITCH_STMT
用来表示一个switch语句。SWITCH_STMT_COND是发生switch的表达式。更多关于条件表示的信息，参见IF_STMT文档。SWITCH_STMT_BODY是switch语句主体。SWITCH_STMT_TYPE是源代码中给出的switch表达式的，在任何编译器转换之前的原始类型。

TRY_BLOCK 用来表示一个try块。try块的主体由TRY_STMTS给出。每个catch块都是一个HANDLER节点。第一个handler由TRY_HANDLERS给出。后续的handlers可以通过TREE_CHAIN获得。handler的主体由HANDLER_BODY给出。

如果CLEANUP_P持有TRY_BLOCK，则TRY_HANDLERS将不是一个HANDLER节点。相反的，其将会是一个表达式，并且如果在try块中抛出异常时则被执行。其必须在执行完那些代码之后再重新抛出异常。并且，如果当表达式正在执行的时候，如果有异常抛出，则必须终止。

USING_STMT 用来表示using指示符。命名空间为一个NAMESPACE_DECL，由USING_STMT_NAMESPACE给出。该节点在模板函数内部需要，用来在实例化时实现using指示符。

WHILE_STMT 用来表示一个while循环。WHILE_COND是循环的终止条件。关于用来表示条件的更多信息，参见IF_STMT的文档。

WHILE_BODY是循环体。

9.7 树中的属性

使用关键字__attribute__指定的属性，在内部作为TREE_LIST来表示。TREE_PURPOSE，作为一个IDENTIFIER_NODE，是属性的名字。如果有参数的话，TREE_VALUE是一个属性参数的TREE_LIST，或者在没有参数时，为NULL_TREE。参数作为列表中的TREE_VALUE后继项存储，并且可以为标识符或者表达式。属性的TREE_CHAIN是在属性列表中应用到同一声明或类型的下一个属性，或者为NULL_TREE如果列表中没有更多的属性。

属性可以附加到声明和类型上；这些属性可以通过下列宏来访问。所有的属性都通过这种方式存储，并且许多还对声明和类型，或者其它内部编译器数据结构体，引起其它的变化。

tree DECL_ATTRIBUTES (tree decl) [Tree Macro]

该宏返回声明decl上的属性。

tree TYPE_ATTRIBUTES (tree type) [Tree Macro]

该宏返回类型type上的属性。

9.8 表达式

表达式的内部表示大多都很简单明了。但是，也有一些事实必须牢记。尤其是，表达式“tree”实际上是一个有向无环图。（例如，整个源程序中可能会有许多对常整数0的引用；这些将会由同一个表达式节点来表示。）当然，你不要以此就认为某些种类的节点是共享的，也不要认为某些种类的节点没有被共享。

下列宏可以用于所有表达式节点：

TREE_TYPE 返回表达式的类型。该值可能不是与原始程序中给出的表达式相同精度的类型。

在下文中，有些节点可能总是期望具有bool类型，但是被记载为具有整形或bool型。将来的某个时刻，C前端可能也会使用该相同的中间表示，那时，这些节点当然会具有整数类型。当然，这并不意味着暗示C++前端中这些节点不具有，或者不会具有整数类型。

下面，我们列出了各种表达式节点。除了特别注明的以外，表达式的操作数都通过TREE_OPERAND宏来访问。例如，要访问二元加法表达式expr的第一个操作数，使用：

```
TREE_OPERAND (expr, 0)
```

正如这个例子所示，操作数是从0开始索引的。

所有起始于OMP_的表达式，表示使用的是OpenMp API http://www.openmp.org/的directives和clauses。

下面的列表格首先介绍了常数，接着是一元表达式，然后是二元表达式，以及各种其它类型的表达式：

INTEGER_CST

这些节点表示整数常量。注意这些常量的类型通过TREE_TYPE来获得；并不总是int型的。特别是，char型常量使用INTEGER_CST节点表示。整数常量e的值通过下面的方式给出

```
((TREE_INT_CST_HIGH (e) << HOST_BITS_PER_WIDE_INT)
+ TREE_INST_CST_LOW (e))
```

HOST_BITS_PER_WIDE_INT在所有的平台上最少为32。 TREE_INT_CST_HIGH和TREE_INT_CST_LOW都返回一个HOST_WIDE_INT。 一个INTEGER_CST的值根据常量的类型而被解析为有符号或无符号的数。 一般来说，上面给出的表达式将会溢出，因此不要用来计算常量的值。

变量integer_zero_node是一个值为0的整数常量。类似的， integer_one_node是值为1的整数常量。 变量size_zero_node和size_one_node比较类似， 只是具有size_t类型，而不是int。

函数tree_int_cst_lt是一个断言，当第一个参数小于第二个时有效。 两个常量被假设为具有相同的符号性（即，要么都是有符号的，要么都是无符号的。 ）在作比较时，使用常量的全部宽度；并忽略掉通常的类型提升和转换规则。 类似的，tree_int_cst_equal在两个常熟相等时有效。 函数tree_int_cst_sgn返回常数的符号。根据常数是大于，等于，或小于0， 而返回值1,0或-1。此外，会顾及到常数类型的符号性； 无符号常数是永远小于0的，不论它的位模式如何。

REAL_CST

FIXME: 讨论如何获得该常量的表示，如何进行比较等等。

FIXED_CST

这些节点表示定点常数。这些常量的类型通过TREE_TYPE获得。TREE_FIXED_CST_PTR指向struct fixed_value；TREE_FIXED_CST返回结构体本身。Struct fixed_value包含了具有2个HOST_BITS_PER_WIDE_INT大小的data，以及与data关联的定点机器模式mode。

COMPLEX_CST

这些节点用于表示复数常量，即__complex__，其组成部分为常数节点。TREE_REALPART和TREE_IMAGPART返回相应的实部和虚部。

VECTOR_CST 这些节点用于表示向量常数，其组成部分为常量节点。每个单独的常量节点或者是一个常整数节点，或者是一个双精度的常数节点。第一个操作数为常数节点的TREE_LIST，并可以通过TREE_VECTOR_CST_ELTS来访问。

STRING_CST 这些节点表示字符串常量。 TREE_STRING_LENGTH返回int型的字符串长度。TREE_STRING_POINTER是一个char*型，包含了字符串本身。字符串可以不是NUL结尾的，并且可以包含嵌入的NUL字符。因此，如果字符串的结尾存在NUL，则TREE_STRING_LENGTH也包括了结尾的NUL。

对于宽字符串常量， TREE_STRING_LENGTH为字符串的字节数，并且TREE_STRING_POINTER指向在目标系统上表示的，字符串的字节数组（即，符合目标大小端的整数系列）。宽字符串和非宽字符串常量，只区别于STRING_CST的TREE_TYPE。

FIXME: 当目标系统的字节与主机系统的字节宽度不同时，字符串的格式没有被很好的定义。

PTRMEM_CST 这些节点用来表示指向成员的指针（ pointer-to-member ）常量。PTRMEM_CST_CLASS是类的类型（或者为RECORD_TYPE，或者为UNION_TYPE），PTRMEM_CST_MEMBER是指向的对象的声明。注意PTRMEM_CST_MEMBER的DECL_CONTEXT一般有别于PTRMEM_CST_CLASS的。例如，给定：

```
struct B { int i; };
struct D : public B {};
int D::*dp = &D::i;
```

虽然PTRMEM_CST_MEMBER的DECL_CONTEXT是B，但是由于B::i是B的成员，而不是D的，所以&D::i的PTRMEM_CST_CLASS为D。

VAR_DECL

这些节点表示变量，包括静态数据成员。更多信息，参见see Section 9.5 [声明], page 87。

NEGATE_EXPR

这些节点表示对单个,整数或浮点类型的操作数，进行一元取负运算。取负运算结果的类型可以通过查看表达式的类型来决定。

该操作在有符号算术溢出时的行为，由flag_wrapv和flag_trapv变量来控制。

ABS_EXPR 这些节点表示单个操作数，整数和浮点类型的，的绝对值。这通常用于实现整数类型的内建abs，labs和llabs，以及浮点类型的fabs，fabsf和fabsl。abs操作的类型可以通过查看表达式的类型来决定。

该节点不用于复数类型。要表示复数的模或者复数abs，使用内建的BUILT_IN_CABS, BUILT_IN_CABSF或BUILT_IN_CABSL，这些被用于实现C99的内建函数cabs, cabsf和cabsl。

BIT_NOT_EXPR

这些节点表示按位求补运算，并总是具有整数型。唯一的操作数是要被求补的值。

TRUTH_NOT_EXPR

这些节点表示逻辑非，并总是具有整数（或布尔）类型。操作数是要求非的值。操作数的，以及结果的类型总是BOOLEAN_TYPE或INTEGER_TYPE。

PREDECREMENT_EXPR
PREINCREMENT_EXPR
POSTDECREMENT_EXPR
POSTINCREMENT_EXPR

这些节点表示递增和递减表达式。单操作数的值将被计算，并且操作数递增或递减。在PREDECREMENT_EXPR和PREINCREMENT_EXPR的情况下，表达式的值是递增或递减之后的结果；在POSTDECREMENT_EXPR和POSTINCREMENT_EXPR的情况下，表达式的值是递增或递减发生前的值。操作数的值，跟结果的一样，将会是整数，布尔，或浮点的。

ADDR_EXPR 这些节点用于表示对象的地址。（这些表达式将总是具有指针或引用类型。）操作数或者为表达式，或者可以是一个声明。

作为扩展，GCC运行用户使用标号的地址。这种情况下，ADDR_EXPR的操作数将为LABEL_DECL。这样的表达式的类型是void*。

如果求址的对象不是左值，则创建一个临时的，并使用临时对象的地址。

INDIRECT_REF

这些节点用来表示由指针指向的对象。操作数是被dereferenced的指针；其总是具有指针或引用类型。

FIX_TRUNC_EXPR

这些节点表示浮点值到整数的转换。单操作数将具有一个浮点类型，完整的表达式将具有整数（或布尔）类型。操作数向0方向舍入。

FLOAT_EXPR 这些节点表示整数（或布尔）值向浮点值的转换。单操作数将具有整数类型，而完整的表达式将具有浮点类型。

FIXME: 操作数是如何被舍入的？这是不是取决于`-mieee'？

COMPLEX_EXPR

这些节点用于表示通过两个相同类型（整数或实数）的表达式构造的复数。第一个操作数是实部，第二个操作数是虚部。

CONJ_EXPR 这些节点表示它们的操作数的共轭（conjugate）。

REALPART_EXPR

IMAGPART_EXPR

这些节点表示复数的相应实数和虚数部分。

NON_LVALUE_EXPR

这些节点指示它们有且仅有的一个操作数不是左值的。后端可以将其作为单操作数来对待。

NOP_EXPR 这些节点用于表示不需要任何代码生成的转换。例如，由char*到int*不需要任何代码生成；这样的转换被表示为一个NOP_EXPR。单操作数为要转换的表达式。从指针到引用的转换也被表示为NOP_EXPR。

CONVERT_EXPR

这些节点类似于NOP_EXPR，不过用于可能会有代码生成的情况。例如，如果int*被转换为int，则可能会在一些平台上需要生成代码。这些节点从来不被用于C++特定的转换，例如在一个继承体系中不同的类的指针间的转换。这种情况下的任何调整，总是需要被显式的指出。类似的，用户定义的转换也不使用CONVERT_EXPR表示；相反的，而是显式的调用函数。

FIXED_CONVERT_EXPR

这些节点用于表示涉及定点值的转换。例如，从一个定点值到另一个定点值，从一个整数到一个定点值，从一个定点值到一个整数，从一个浮点值到一个定点值，或者从一个定点值到一个浮点值。

THROW_EXPR 这些节点表示throw表达式。单操作为一个表达式，其代码将被执行用于抛出异常。但是，在表达式中有一个隐式的动作没有表示出来；即调用 __throw。这个函数没有参数。如果使用了setjmp/longjmp异常，则相应的调用函数__sjthrow。通常GCC后端使用函数 emit_throw来生成该代码；你可以检查该函数看看都需要做什么。

LSHIFT_EXPR

RSHIFT_EXPR

这些节点分别表示左移和右移。第一个操作数为要移动的值；其将总是为整数类型。第二个操作数为表示移动位数的表达式。右移将作为算术移动，即，当表达式具有无符号类型则高位填充0，当表达式具有有符号类型则高位填充符号位。注意如果第二个操作数大于或等于第一个操作数的类型大小，则结果无定义。

BIT_IOR_EXPR

BIT_XOR_EXPR

BIT_AND_EXPR

这些节点分别表示位运算符“或”，“异或”，和“与”。所有操作数将总是为整数类型。

TRUTH_ANDIF_EXPR

TRUTH_ORIF_EXPR

这些节点分别表示逻辑“与”和“或”。这些操作符不是严格的；即第二个操作数只在通过求值第一个操作数无法确定表达式的值的时候，才被计算求值。操作数和结果的类型总是为BOOLEAN_TYPE或INTEGER_TYPE。

TRUTH_AND_EXPR
TRUTH_OR_EXPR
TRUTH_XOR_EXPR
: 这些节点表示逻辑与，或和异或。它们为严格的方式；所有参数都总是被计算求值。这在C或C++中没有对应的运算符，但是前端如果可以断定严格的方式没关系，则有时将会生成这些表达式。操作数和结果的类型总是为BOOLEAN_TYPE或INTEGER_TYPE。

POINTER_PLUS_EXPR
: 该节点表示指针算术运算。第一个操作数总是为一个指针/引用类型。第二个操作数总是为一个与sizetype兼容的无符号整数类型。这是唯一的可以操作指针类型的二元算术运算。

PLUS_EXPR
MINUS_EXPR
MULT_EXPR
: 这些节点表示不同的二元算术运算。分别为加法，减法和乘法。它们的操作数可以为整数或者浮点类型，但不会为一个是浮点类型的，而另一个是整数类型。

  这些运算在有符号算术溢出时的行为，由变量flag_wrapv和flag_trapv来控制。

RDIV_EXPR
: 该节点表示一个浮点除法运算。

TRUNC_DIV_EXPR
FLOOR_DIV_EXPR
CEIL_DIV_EXPR
ROUND_DIV_EXPR
: 这些节点表示返回整数结果的整数除法运算。TRUNC_DIV_EXPR向0方向舍入，FLOOR_DIV_EXPR向负无穷大舍入，CEIL_DIV_EXPR向正无穷大舍入，ROUND_DIV_EXPR向最近的整数舍入。C和C++中的整数除法为截断方式，即TRUNC_DIV_EXPR。

  这些运算在有符号算术溢出时的行为，由变量flag_wrapv和flag_trapv来控制。

TRUNC_MOD_EXPR
FLOOR_MOD_EXPR
CEIL_MOD_EXPR
ROUND_MOD_EXPR
: 这些节点表示整数类型的求余或求模运算。两个操作数a和b的整型的模被定义为a - (a/b)*b，其中使用相应的除法操作符进行除法运算。因此，对于TRUNC_MOD_EXPR的定义，是假设使用了向零方向舍去的除法，即TRUNC_DIV_EXPR。C和C++中的整型求余，使用了舍去除法，即TRUNC_MOD_EXPR。

EXACT_DIV_EXPR
: EXACT_DIV_EXPR用来表示整数除法，即分子已知为分母的确切的倍数。这使得后端可以从TRUNC_DIV_EXPR, CEIL_DIV_EXPR和FLOOR_DIV_EXPR中，为当前的目标机选择更快的运算。

ARRAY_REF
: 这些节点表示对数组的访问。第一个操作数为数组；第二个为索引。要计算被访问内存的地址，你必须要根据比例，用数组元素的类型大小来乘以索引。这些表达式的类型必须为数组元素的类型。第三和第四个操作数在gimplification之后使用，来表示下界和元素大小；但是不要直接使用它们，相应的，调用array_ref_low_bound和array_ref_element_size。

ARRAY_RANGE_REF

这些节点表示对数组的一个范围（或者说切片）的访问。操作数与ARRAY_REF相同，并具有相同的含义。这些表达式的类型必须是一个数组，其元素的类型与第一个操作数的类型相同。数组类型的范围决定了这些表达式访问的数据数目。

TARGET_MEM_REF

这些节点表示内存访问，并且其地址直接映射到目标体系结构的寻址模式。第一个参数为TMR_SYMBOL，并且必须为具有固定地址的对象的VAR_DECL。第二个参数为TMR_BASE，第三个为TMR_INDEX。第四个参数为TMR_STEP，并且必须为INTEGER_CST。第五个参数为TMR_OFFSET，并且必须为INTEGER_CST。如果相应的部分没有在地址中出现，则参数可以为NULL。TARGET_MEM_REF的地址通过下列方法来确定。

```
&TMR_SYMBOL + TMR_BASE + TMR_INDEX * TMR_STEP + TMR_OFFSET
```

第六个参数为对原始内存访问的引用，其被保留下来，用于RTL别名分析。第七个参数为一个标记，表示tree级的别名分析的结果。

LT_EXPR
LE_EXPR
GT_EXPR
GE_EXPR
EQ_EXPR
NE_EXPR

这些节点表示小于，小于或等于，大于，大于或等于，等于，和不等于的比较运算符。第一个和第二个操作数或者都为整数类型，或者都为浮点类型。这些表达式的结果类型将总是为整数或布尔类型。如果为假，则这些运算返回值0，如果为真，则返回值1。

对于浮点类型比较运算，如果我们使用了IEEE NaN，并且任意一个操作数为NaN，则NE_EXPR总是返回真，而其余的运算符总是返回假。在一些目标机上，对于IEEE NaN，除了等于和不等于以外的其它比较运算，可能会生成一个浮点异常。

ORDERED_EXPR
UNORDERED_EXPR

这些节点表示non-trapping的有序和无序的比较运算。这些运算接受两个浮点操作数，并确定它们之间是有序的，还是无序的。如果有一个操作数为IEEE NaN，则它们的比较被定以为无序的，否则为有序的。这些表达式的结果类型将总是为整数或者布尔类型。如果为假，则这些运算返回值0，如果为真，则返回值1。

UNLT_EXPR
UNLE_EXPR
UNGT_EXPR
UNGE_EXPR
UNEQ_EXPR
LTGT_EXPR

这些节点表示无序比较运算符。这些运算接受两个浮点操作数，并分别确定它们之间是否为无序的，小于，小于或等于，大于，大于或等于，或者等于。例如，如果一个操作数为IEEE NaN，或者第一个操作数小于第二个，则UNLT_EXPR返回真。除了LTGT_EXPR可能会产生异常，其它的运算都保证不会产生浮点异常。这些表达式的结果类型将总是为整数或者布尔类型。如果为假，则这些运算返回值0，如果为真，则返回值1。

MODIFY_EXPR

这些节点表示赋值。左手边为第一个操作数，右手边为第二个操作数。左手边为一个VAR_DECL, INDIRECT_REF, COMPONENT_REF，或者其它左值。

这些节点不仅用来表示使用`='进行赋值，也用来表示复合赋值（像`+='），并将其转换成`='赋值。换句话说，对`i += 3'的表示，看起来就像是对`i = i + 3'的表示。

INIT_EXPR 这些节点就像MODIFY_EXPR一样，只不过用于一个变量被初始化的时候，而不是后续的赋值。这意味着，我们可以假设初始化的目标，没有在右手边被用于计算它自己的值；任何在右手边的计算中，对左手边的引用，其行为将是未定义。

COMPONENT_REF

这些节点表示对non-static数据成员的访问。第一个操作数为对象（而不是指向它的指针）；第二个操作数为数据成员的FIELD_DECL。第三个操作数表示域的字节偏移量，但不要直接使用；相应的，调用component_ref_field_offset。

COMPOUND_EXPR

这些节点表示逗号表达式。第一个操作数为表达式，其值被计算，并在求出第二个操作数的值之前丢掉。整个表达式的值为第二个操作数的值。

COND_EXPR 这些节点表示?:表达式。第一个操作数是布尔或者整数类型。如果其求解为非零值，则第二个操作数将被求值，并返回表达式的值。否则，第三个操作数将被求值，并将表达式的值返回。

第二个操作数必须与整个表达式具有相同的类型，除非它是要无条件的抛出一个异常或者调用一个不返回的函数，这种情况下，其将是void类型。第三个操作数也具有同样的约束。这使得数组的边界检查可以被方便的表示为(i >= 0 && i < 10) ? i : abort()。

作为GNU扩展，C语言前端允许?:运算符的第二个操作数可以在源程序中省略掉。例如，x ? : 3等价于x ? x : 3，假设x是一个没有副作用的表达式。但是，在tree的表示中，第二个操作数总是存在的，并且，如果第一个参数确实产生副作用的话，则其可能通过SAVE_EXPR来保护。

CALL_EXPR 这些节点用来表示对函数的调用，包括non-static成员函数。CALL_EXPR被实现为一个具有可变数目操作数的表达式节点。不要用TREE_OPERAND来获取这些操作数，最好是用针对CALL_EXPR节点的特定的访问宏和函数。

CALL_EXPR_FN返回一个调用函数的指针；其总是一个类型为POINTER_TYPE的表达式。

调用函数的参数数目由call_expr_nargs来返回，而参数本身可以使用CALL_EXPR_ARG宏来访问。参数从零开始，从左向右进行索引。你可以使用FOR_EACH_CALL_EXPR_ARG来迭代参数，例如：

```
tree call, arg;
call_expr_arg_iterator iter;
FOR_EACH_CALL_EXPR_ARG (arg, iter, call)
 /* arg is bound to successive arguments of call. */
 ...;
```

对于non-static成员函数，将会有一个对应于this指针的操作数。所有的参数都会有相应的表达式，即使函数使用缺省的参数声明，并且在调用的地方，一些参数没有被显式的提供。

CALL_EXPR还有一个CALL_EXPR_STATIC_CHAIN操作数，用于实现嵌套函数。如果没有嵌套函数，则为null。

STMT_EXPR 这些节点用于表示GCC的语句表达式扩展。语句表达式扩展，允许代码像这样：

```
int f() { return ({ int j; j = 3; j + 7; }); }
```

还句话说，一个语句序列可以出现在通常单个表达式出现的地方。STMT_EXPR节点表示这样的表达式。The STMT_EXPR_STMT给出了表达式中包含的语句。表达式的值为最后一条语句的值。更确切的说，为嵌套在BIND_EXPR, TRY_FINALLY_EXPR, 或TRY_CATCH_EXPR中的最后一条语句所计算出来的值。例如，

```
({3;})
```

值为3，而:

```
({if(x){3;}})
```

没有值。如果STMT_EXPR没有产生一个值，则类型为void。

BIND_EXPR 这些节点表示局部块。第一个操作数为变量的列表，通过它们的TREE_CHAIN域链接。这些将从不会要求被清除。这些变量的作用域就是BIND_EXPR的主体。BIND_EXPR的主体为第二个操作数。

LOOP_EXPR 这些节点表示“无限”循环。LOOP_EXPR_BODY表示循环体。其将被永远执行，除非遇到EXIT_EXPR。

EXIT_EXPR 这些节点表示从最近包含的LOOP_EXPR中条件退出。单个操作数为条件；如果非零，则循环将被退出。EXIT_EXPR将只是出现在LOOP_EXPR中。

CLEANUP_POINT_EXPR

这些节点表示full-expression。单个操作数为被求值的表达式。任何在表达式求值中通过创建临时对象所引起的析构调用，都应该在表达式求值之后立刻执行。

CONSTRUCTOR

这些节点表示大括号括起的，对结构体或者数组的初始化。第一个操作数被保留，用于后端。第二个操作数为TREE_LIST。如果CONSTRUCTOR的TREE_TYPE为一个RECORD_TYPE或者UNION_TYPE，则TREE_LIST中每个节点的TREE_PURPOSE将为一个FIELD_DECL，并且每个节点的TREE_VALUE将为初始化该域的表达式。

如果CONSTRUCTOR的TREE_TYPE为一个ARRAY_TYPE，则TREE_LIST中每个节点的TREE_PURPOSE将为一个INTEGER_CST，或者两个INTEGER_CST的RANGE_EXPR。单个INTEGER_CST指出了数组（从0开始索引）的哪个元素将被赋值。RANGE_EXPR指出了包含端点元素的一个范围将被初始化。这两种情况下，TREE_VALUE都对应初始化者。其值将对于RANGE_EXPR的每个元素都重新计算一次。如果TREE_PURPOSE是NULL_TREE，则初始化是针对下一个可用的数组元素。

在前端，你不要认为域是按照特定的顺序出现。但是，在中端，域必须按照声明的顺序出现。你不应该假设所有的域都被表示了。没有表示的域将被设置为0。

COMPOUND_LITERAL_EXPR

这些节点表示复合文字。COMPOUND_LITERAL_EXPR_DECL_STMT为一个DECL_STMT，包含了一个由复合文字表示的未命名对象的匿名VAR_DECL；VAR_DECL的DECL_INITIAL是一个CONSTRUCTOR用来表示在复合文字中大括号包围的初始值列表。匿名的VAR_DECL还可以通过COMPOUND_LITERAL_EXPR_DECL宏直接访问。

SAVE_EXPR SAVE_EXPR表示一个被多次使用的表达式（可能会有副作用）。副作用应该只在表达式第一次被求值时发生。后续的使用应该只是重用计算所得的值。SAVE_EXPR的第一个操作数是要求值的表达式。副作用应该在深度优先前续遍历表达式树，第一次遇到SAVE_EXPR时被执行。

TARGET_EXPR

TARGET_EXPR表示一个临时对象。第一个操作数是临时变量VAR_DECL。第二个操作数是临时变量的初始值。初始值将被求值，并且如果不是void型的，则（按位）复制到临时变量中。如果初始值是void的，意味着将会自己执行初始化。

很多时候，TARGET_EXPR会出现在赋值的右边，或者作为逗号表达式的第二个操作数。这种情况下，我们说TARGET_EXPR是“normal”的；否则，我们说它是“orphaned”

。 对于一个正常的TARGET_EXPR，临时变量应被视为赋值的左端的一个别名， 而不是一个新的临时变量。

TARGET_EXPR的第三个操作数，如果存在的话， 是临时变量的清理表达式（即析构调用）。如果该表达式是孤儿的， 则该表达式必须当包含它的语句是完整的时候被执行。这些清理必须总是按照相反的顺序执行。 注意如果临时变量是在条件操作符的分支上创建的（即， COND_EXPR的第二个或第三个操作数）， 则清理必须只有在该分支实际被执行时才运行。

关于运行这些清理的更多信息，参见STMT_IS_FULL_EXPR_P。

AGGR_INIT_EXPR

AGGR_INIT_EXPR表示作为函数调用的返回值或者作为构造函数的结果的初始化。AGGR_INIT_EXPR只作为充分表达式出现， 或作为TARGET_EXPR的第二个操作数。AGGR_INIT_EXPR具有跟CALL_EXPR类似的表示。 可以使用AGGR_INIT_EXPR_FN和AGGR_INIT_EXPR_ARG宏来访问调用的函数， 以及传递的参数。

如果AGGR_INIT_VIA_CTOR_P持有AGGR_INIT_EXPR， 则初始化是通过一个构造函数进行的。AGGR_INIT_EXPR_SLOT操作数的地址， 其总是一个VAR_DECL，将被接受，并且该值替代参数列表中的第一个参数。

在这两种情况下，表达式都是void的。

VA_ARG_EXPR

该节点用来实现对C/C++可变参数列表机制的支持。 它表示了像va_arg (ap, type)这样的表达式。 它的TREE_TYPE用来产生type的树表示， 唯一的参数用来产生对ap的表示。

CHANGE_DYNAMIC_TYPE_EXPR

指出了C++ placement new的特定别名需求。有两个操作数：类型和位置。 它表示该位置的动态类型被转换为指定的类型。 别名分析代码在做基于类型的别名分析时，需要考虑到这一点。

OMP_PARALLEL

表示#pragma omp parallel [clause1 ... clauseN]。具有四个操作数：

操作数OMP_PARALLEL_BODY在GENERIC和High GIMPLE形式中是有效的。 它包含了被所有线程执行的代码体。在GIMPLE下降过程中， 这个操作数变为NULL并且代码体被线性的输出在OMP_PARALLEL之后。

操作数OMP_PARALLEL_CLAUSES为与指令相关的子句列表。

操作数OMP_PARALLEL_FN由pass_lower_omp创建， 它包含了将要包含并行区域体的函数FUNCTION_DECL。

操作数OMP_PARALLEL_DATA_ARG也由pass_lower_omp创建。 如果有共享变量用于子线程间通讯，则该操作数将包含VAR_DECL， 其包含了所有共享的值和变量。

OMP_FOR

表示#pragma omp for [clause1 ... clauseN].其具有5个操作数：

操作数OMP_FOR_BODY包含了循环体。

操作数OMP_FOR_CLAUSES为与指令相关的子句列表。

操作数OMP_FOR_INIT为VAR = N1形式的循环初始化代码。

操作数OMP_FOR_COND为VAR {<, >, <=, >=} N2形式的循环条件表达式。

操作数OMP_FOR_INCR为VAR {+=, -=} INCR形式的循环索引增量。

操作数OMP_FOR_PRE_BODY包含了来自操作数OMP_FOR_INIT, OMP_FOR_COND和OMP_FOR_INC的副作用代码。这些副作用为OMP_FOR块的一部分，但是必须在开始循环体之前被计算求值。

循环索引变量VAR必须为单个整数变量，其隐式的归每个线程私有。边界N1和N2，以及增量表达式INCR需要为循环不变量整数表达式，其不需要同步就可以被计算求值。按照标准，计算求值的顺序，频率和副作用都没有被指定。

OMP_SECTIONS

表示#pragma omp sections [clause1 ... clauseN]。

操作数OMP_SECTIONS_BODY包含了section主体， 其依次包含了一个OMP_SECTION节点集合， 每个并发的section通过#pragma omp section来划分。

操作数OMP_SECTIONS_CLAUSES为与指令相关的子句列表。

OMP_SECTION

OMP_SECTIONS的Section定界符。

OMP_SINGLE

表示#pragma omp single

操作数OMP_SINGLE_BODY包含了被单个线程执行的代码体。

操作数OMP_SINGLE_CLAUSES为与指令相关的子句列表。

OMP_MASTER

表示#pragma omp master。

操作数OMP_MASTER_BODY包含了被主控线程执行的代码体。

OMP_ORDERED

表示#pragma omp ordered。

操作数OMP_ORDERED_BODY包含了按照由循环索引变量所指示的顺序序列来执行的代码体。

OMP_CRITICAL

表示#pragma omp critical [name]。

操作数OMP_CRITICAL_BODY为临界section。

操作数OMP_CRITICAL_NAME为可选的用来标记临界section的标识符。

OMP_RETURN

这个并不表示任何OpenMP指令，它为一个人为标记用来指示OpenMP主体的结束。其被用于流图（tree-cfg.c）和OpenMP区域构建代码（omp-low.c）。

OMP_CONTINUE

类似的，该指令不表示OpenMP指令，它被OMP_FOR和OMP_SECTIONS用于 标记代码需要循环到下一个迭代（例如OMP_FOR）或者下一个section （例如OMP_SECTIONS）的地方。有一些情况， OMP_CONTINUE被放在紧挨着OMP_RETURN之前。 但是，如果在循环体之后需要出现cleanups， 则它将被生成在OMP_CONTINUE和OMP_RETURN之间。

OMP_ATOMIC

表示#pragma omp atomic。

操作数0是要被执行的原子操作的地址。

操作数1是要计算求值的表达式。gimplifier会尝试三种可供选择的代码生成策略。只要可能，则会使用内建的原子更新。如果失败，则会尝试进行比较-交换（compare-and-swap）循环。如果还是失败，则会使用表达式附近的一个常规临界section。

OMP_CLAUSE

表示与OMP_指令相关的子句。子句使用`tree.h'中定义的子代码单独表示。子句代码可以为：OMP_CLAUSE_PRIVATE, OMP_CLAUSE_SHARED, OMP_CLAUSE_FIRSTPRIVATE, OMP_CLAUSE_LASTPRIVATE, OMP_CLAUSE_COPYIN, OMP_CLAUSE_COPYPRIVATE, OMP_CLAUSE_IF, OMP_CLAUSE_NUM_THREADS, OMP_CLAUSE_SCHEDULE, OMP_CLAUSE_NOWAIT, OMP_CLAUSE_ORDERED, OMP_CLAUSE_DEFAULT, 和OMP_CLAUSE_REDUCTION。每个代码表示了相应的OpenMP子句。

与同一指令相关的子句通过OMP_CLAUSE_CHAIN链接在一起。那些接受一个变量列表的子句被限制为只有一个，使用OMP_CLAUSE_VAR来访问。因此，同一子句C下的多个变量需要被多个链接在一起的C子句表示。这样可以有助于在编译过程中增加新的子句。

VEC_LSHIFT_EXPR
VEC_RSHIFT_EXPR

这些节点相应的表示整个向量的左移和右移。第一个操作数为要移动的向量；其将总是为向量类型。第二个操作数是一个表达式，表示要移动的位数。注意如果第二个操作数大于或等于第一个操作数的类型大小，则结果未定义。

VEC_WIDEN_MULT_HI_EXPR
VEC_WIDEN_MULT_LO_EXPR

这些节点分别表示两个输入向量的高部和低部的加宽向量乘法。它们的操作数为包含同一整数类型的同一数目（N）元素的向量。结果为一个包含整数类型的元素大小为两倍宽数目为一半的向量。对于VEC_WIDEN_MULT_HI_EXPR，两个向量的高N/2个元素相乘得到N/2个积的向量。对于VEC_WIDEN_MULT_LO_EXPR，两个向量的低N/2个元素相乘得到N/2个积的向量。

VEC_UNPACK_HI_EXPR
VEC_UNPACK_LO_EXPR

这些节点分别表示拆分输入向量的高部和低部。单操作数为一个包含同一整数或浮点类型的N个元素的向量。结果为包含整数或者浮点类型的元素大小为两倍宽数目为一半的向量。对于VEC_UNPACK_HI_EXPR，向量的高N/2个元素被提取并扩展（提升）。对于VEC_UNPACK_LO_EXPR，向量的低N/2个元素被提取并扩展（提升）。

VEC_UNPACK_FLOAT_HI_EXPR
VEC_UNPACK_FLOAT_LO_EXPR

这些节点分别表示拆分输入向量的高部和低部，并将值由定点转换为浮点。单操作数为包含同一整数类型的N个元素的向量。结果为包含浮点类型的元素大小为两倍宽数目为一半的向量。对于VEC_UNPACK_HI_EXPR，向量的高N/2个元素被提取，转换并扩展。对于VEC_UNPACK_LO_EXPR，向量的低N/2个元素被提取，转换并扩展。

VEC_PACK_TRUNC_EXPR

该节点表示将两个输入向量的截断元素打包成输出向量。输入操作数是包含同一整数或者浮点类型的相同数目元素的向量。结果为包含整数或者浮点类型的元素大小为一半数目为两倍的向量。两个向量的元素合并成输出向量。

VEC_PACK_SAT_EXPR

该节点表示使用饱和方式（saturation）将两个输入向量的元素打包成输出向量。 输入操作数是包含了同一整数类型的相同数目元素的向量。 结果为一个包含整数类型的元素大小为一半数目为两倍的向量。 两个向量的元素合并成输出向量。

VEC_PACK_FIX_TRUNC_EXPR

该节点表示将将两个输入向量的元素打包成输出向量，并将值由浮点转换为定点。 输入操作数是包含浮点类型的相同数目元素的向量。 结果为包含整数类型元素大小一半数目为两倍的向量。两个向量的元素合并成输出向量。

VEC_EXTRACT_EVEN_EXPR
VEC_EXTRACT_ODD_EXPR

这些节点分别表示提取两个输入向量的偶数/奇数个元素。 它们的操作数和结果为包含同一类型的相同数目元素的向量。

VEC_INTERLEAVE_HIGH_EXPR
VEC_INTERLEAVE_LOW_EXPR

这些节点分别表示交错合并两个输入向量的高/低元素。 操作数和结果为包含同一类型的相同数目（N）元素的向量。 对于VEC_INTERLEAVE_HIGH_EXPR，第一个输入向量的高N/2个元素被第二个输入向量的高N/2个元素替换。 对于VEC_INTERLEAVE_LOW_EXPR，第一个输入向量的低N/2个元素被第二个输入向量的低N/2个元素替换。

10 RTL表示

编译器的大部分工作都是基于一种中间表示，叫做寄存器传送语言（register transfer language）。在该语言中，描述了将要输出的指令，并且差不多是按照字母顺序一个一个的来描述指令的行为。

RTL的灵感来自Lisp列表。它同时具有一个内部形式，由指向结构体的结构体组成，以及一个文本形式，用在机器描述和打印的调试输出中。文本形式使用嵌套的括号，来表示内部形式中的指针。

10.1 RTL对象类型

RTL使用五种对象：表达式、整数、宽整数、字符串和向量。其中，最重要的是表达式。RTL 表达式（简称RTX）是一个C结构体，通常用指针来引用它。这种引用它的指针的类型定义名为rtx。

整数就是C中的int，书写形式使用十进制表示。宽整数是HOST_WIDE_INT类型的一个整数对象，其书写形式也用十进制表示。

字符串为一串字符，在存储器中以C的char *形式表示且按C语法规定书写。然而，RTL中的字符串决不会为空值。若机器描述中有一空字符串，它在存储器中则表示成一个空指针而不是通常意义上的指向空字符的指针。在某些上下文中，允许用这种空指针表示空字符串。在RTL代码中，字符串经常出现在symbol_ref表达式中，但也出现在某些机器描述的RTL表达式中。

对于字符串，还有一种特殊的语法，用于在机器描述中嵌入C代码。只要字符串可以出现的地方，都可以书写一个C风格的大括号代码块。整个大括号代码块，包括最外面的一对括号，被作为字符串常量看待。括号里面的双引号字符不是特殊字符。因此，如果你在C代码中书写字符串常量，则不需要使用反斜杠来转义每个引号字符。

向量包含任意数目的指向表达式的指针。向量中元素的个数，在向量中显式的存在。向量的书写形式为，方括号（`[...]'），里面是元素，并使用空格分隔。长度为0的向量不会被创建；而是使用空指针来替代。

表达式根据expression codes来划分类别（也称作RTX代码）。表达式代码为在`rtl.def'中定义的一个名字，其也是一个（大写的）C枚举常量。合理的表达式代码以及它们的含义，是机器无关的。RTX的代码可以使用宏GET_CODE (x)来抽取，以及使用PUT_CODE (x, newcode)来修改。

表达式代码决定了表达式包含了多少个操作数，以及它们都是什么对象。在RTL中，不像Lisp，你不能通过查看一个操作数来得知它是什么对象。替代的，你必须通过它的上下文来知道——通过所包含的表达式的表达式代码。例如，在一个表达式代码为subreg的表达式中，第一个操作数被作为一个表达式，第二个操作数为一个整数。在代码为plus的表达式中，有两个操作数，都作为表达式。在symbol_ref表达式中，有一个操作数，作为一个字符串。

表达式被书写为，一对括号，包含了表达式类型的名字，它的标记和机器模式（如果存在的话），然后是表达式的操作数（通过空格分隔）。

表达式代码名，在`md'文件中按小写形式书写，但在C代码中出现时按大写形式书写。在这个手册里，它们按照如下形式表示：const_int。

在一些上下文中，表达式通常会需要一个空指针。这种书写形式为(nil)。

10.2 RTL类别和格式

不同的表达式代码被分为几个类别（classes），其有单个字符表示。你可以使用宏GET_RTX_CLASS (code)来确定RTX代码的类别。当前，`rtl.def'定义了这些类别：

RTX_OBJ
: 一个RTX代码，表示一个实际的对象，例如一个寄存器(REG)或者一个内存位置(MEM, SYMBOL_REF)。也包括LO_SUM)；但是，SUBREG和STRICT_LOW_PART不在这个类别中，而是在x类别中。

RTX_CONST_OBJ
: 一个RTX代码，表示一个常量对象。HIGH也包含在该类别中。

RTX_COMPARE
: 一个RTX代码，针对一个非对称的比较，例如GEU或LT。

RTX_COMM_COMPARE
: 一个RTX代码，针对一个对称（可交换）比较，例如，例如EQ或ORDERED。

RTX_UNARY
: 一个RTX代码，针对一元算术运算，例如NEG，NOT或者ABS。这个类别还包括值扩展（符号扩展或者零扩展），以及整数和浮点之间的转换。

RTX_COMM_ARITH
: 一个RTX代码，针对可交换的二元运算，例如PLUS或者AND。NE和EQ为比较运算，所以它们具有类别<。

RTX_BIN_ARITH
: 一个RTX代码，针对不可交换的二元运算，例如MINUS，DIV或者ASHIFTRT。

RTX_BITFIELD_OPS
: 一个RTX代码，针对位域运算。当前只有ZERO_EXTRACT和SIGN_EXTRACT。这些有三个输入，并且为左值。See Section 10.11 [位域运算], page 132。

RTX_TERNARY
: 一个RTX代码，针对其它有三个输入的运算。当前只有IF_THEN_ELSE和VEC_MERGE。

RTX_INSN
: 一个RTX代码，针对整个指令：INSN，JUMP_INSN和CALL_INSN。See Section 10.18 [Insns], page 139。

RTX_MATCH
: 一个RTX代码，针对在insn中的一些匹配，例如MATCH_DUP。这些只出现在机器描述中。

RTX_AUTOINC
: 一个RTX代码，针对一个自动增量寻址模式，例如POST_INC。

RTX_EXTRA
: 所有其它的RTX代码。这个类别包括只在机器描述(DEFINE_*等)中使用的其它RTX代码。其还表示所有表述副作用的RTX代码（SET，USE，CLOBBER等），以及在insn链中可能出现的非insn，例如NOTE，BARRIER和CODE_LABEL。SUBREG也属于该类。

对于每个表达式代码，`rtl.def'使用称作表达式代码格式（format）的字符序列，来说明所包含的对象数目，以及它们的种类。例如，subreg的格式为`ei'。

这些是最常用的格式字符：

e
: 一个表达式（实际是一个表达式指针）

i
: 一个整数。

w
: 一个宽整数。

s
: 一个字符串。

E
: 一个表达式向量。

还有一些其它的格式字符有时会被用到：

u
: `u'等价于`e'，只不过是在调试转储中的打印有所区别。其用于insn指针。

n
: `n'等价于`i'， 只不过是在调试转储中的打印有所区别。其用于note insn的行号和代码号。

S
: `S'表示一个可选的字符串。在内部的RTX对象中，`S'等价于`s'，但当对象从`md'文件中读取的时候，该操作数的字符串值可以被忽略。被忽略的字符串被当作一个空字符串。

V
: `V'表示一个可选的向量。在内部的RTX对象中，`V'等价于`E'，但是当对象从`md'文件中读取的时候，该操作数的向量值可以被忽略。被忽略的向量被当作一个没有元素的向量。

B
: `B'表示一个指向基本块结构体的指针。

0
: `0'表示一个插槽，其内容不使用任何常规的类别。`0'插槽根本不在转储中打印，通常在编译器中用于特定的方式。

这些是获得操作数数目和表达式代码格式的宏：

GET_RTX_LENGTH (code)
: 代码为code的RTX的操作数个数。

GET_RTX_FORMAT (code)
: 代码为code的RTX的格式，为C字符串。

一些RTX代码的类别总是具有相同的格式。例如，可以安全的假设所有的比较运算都具有格式ee。

1
: 所有该类别的代码都具有格式e。

<
c
2
: 所有这些类别的代码都具有格式ee。

b
3
: 所有这些类别的代码都具有格式eee。

i
: 所有该类别的代码具有的格式都起始于iuueiee。 See Section 10.18 [Insns], page 139。注意，并不是所有被链接到insn链表中的RTX对象都属于类别i。

o
m
x
: 你可以不去假设这些代码的格式。

10.3 访问操作数

表达式的操作数用宏XEXP、XINT、XWINT和XSTR访问。 所有这些宏都有两个参数：一个为表达式指针（RTX），另一个为操作数序号（从0开始计算）。如：

```
XEXP (x, 2)
```

表示以表达式方式访问表达式x的第2个操作数。

```
XINT (x, 2)
```

表示以整数方式访问x的第2个操作数。 XSTR表示以字符串方式访问。

任何一个操作数都能以整数方式、表达式方式或字符串方式来访问， 但必须根据存贮在操作数中的实际值选择正确的访问方式。 这可根据表达式的代码而获得，同样也可表达式代码获得操作数的个数。

例如：若x是subreg表达式，通过表达式代码可知它有二个操作数， 这两个操作数的访问应该是XEXP (x, 0) 和XINT (x, 1)， 若写成XINT (x, 0)，那么，你得到的表达式地址将被强制成整数， 偶尔可能会需要这样做，但在这种情况下用(int) XEXP (x, 0)表示要更好。 同样写成XEXP (x, 1) 也不会导致编译错误， 它将返回强制为表达式指针而实际为整数的1号操作数，在运行中访问该指针时，这可能会导致出错。同样，你也可以写成XEXP (x, 28)， 但这超出了此表达式的存贮边界，所得到的将是一个预料不到的结果。

对向量操作数的访问较为复杂些，可用XVEC宏来获取向量指针本身， XVECEXP和XVECLEN宏访问一个向量的元素和长度。

XVEC (exp, idx)
: 获得exp中第idx个操作数的向量指针。

XVECLEN (exp, idx)
: 获得exp中第idx个操作数（为向量操作数）的向量长度（元素个数）， 其值是int。

XVECEXP (exp, idx, eltnum)
: 访问exp中第idx个操作数（为向量操作数）的第eltnum个元素， 其值是一个RTX。
需要由你来确保eltnum不为负，并且小于XVECLEN (exp, idx)。

在本节中所定义的所有宏定义都被扩展成左值， 因而也可用于对操作数、长度和向量元素赋值。

10.4 访问特殊操作数

一些RTL节点具有与它们相关联的特殊的注解。

MEM
: MEM_ALIAS_SET (x)
: 如果为0，则x不在任何别名集中，并可能为任何对象的别名。否则，x只能为在冲突别名集中的MEM的别名。该值在前端使用语言相关的方式来设置，并且不能在后端修改。在一些前端中，这些可以通过某种方式对应到类型，或者其它语言级的实体，但是不要求非要这样，所以在后端不要做这样的假设。这些集合编号使用alias_sets_conflict_p来测试。

MEM_EXPR (x)
如果该寄存器被已知为存放了一些用户级的声明的值，则为那个tree节点。其也可以为COMPONENT_REF，这种情况下，其为某个域的引用，并且TREE_OPERAND (x, 0)包含了声明，或者另一个COMPONENT_REF，或者如果没有编译时对象相关引用，则为空。

MEM_OFFSET (x)
从MEM_EXPR起始的偏移量，为一个CONST_INT rtx。

MEM_SIZE (x)
以字节为单位的内存引用长度，为一个CONST_INT rtx。这主要与BLKmode引用相关，否则机器模式已经隐含了长度。

MEM_ALIGN (x)
内存引用的已知的对齐方式，以位为单位。

REG

ORIGINAL_REGNO (x)
该域存放了寄存器原先具有的编号；对于伪寄存器放入到一个硬寄存器中，其将存放旧的伪寄存器编号。

REG_EXPR (x)
如果该寄存器被已知存放了某个用户级的声明的值，则其为那个tree节点。

REG_OFFSET (x)
如果该寄存器被已知存放了某个用户级的声明的值，则其为相对那个逻辑存储的便宜量。

SYMBOL_REF

SYMBOL_REF_DECL (x)
如果是针对VAR_DECL或FUNCTION_DECL创建的symbol_ref x，则那个tree被记录在这里。如果该值为空，则x由后端代码生成例程创建，并不与前端的符号表实体相关联。

SYMBOL_REF_DECL也可以指向'c'类别的tree，即某种常量。这种情况下，symbol_ref为每个文件的常量池中的实体；同样，不与前端的符号表实体相关联。

SYMBOL_REF_CONSTANT (x)
如果`CONSTANT_POOL_ADDRESS_P (x)'为真，则为x的常量池实体。否则为空。

SYMBOL_REF_DATA (x)
一个不透明类型的域，用来存储SYMBOL_REF_DECL或SYMBOL_REF_CONSTANT。

SYMBOL_REF_FLAGS (x)
在一个symbol_ref中，其用于传达关于符号的各种断言。它们中的一些可以使用通用的代码来计算，一些是特定于目标机器的。通用的位：

SYMBOL_FLAG_FUNCTION
用来表示符号引用一个函数。

SYMBOL_FLAG_LOCAL
: 用来表示符号局部于该模块。参见TARGET_BINDS_LOCAL_P。

SYMBOL_FLAG_EXTERNAL
: 用来表示该符号不在该转换单元中定义。注意，其并不是SYMBOL_FLAG_LOCAL的反转。

SYMBOL_FLAG_SMALL
: 用来表示符号位于小数据段。参见TARGET_IN_SMALL_DATA_P。

SYMBOL_REF_TLS_MODEL (x)
: 这是多个位的域访问，其返回用于线程局部存储符号的tls_model。对于非线程局部符号，其返回0。

SYMBOL_FLAG_HAS_BLOCK_INFO
: 用来表示符号具有SYMBOL_REF_BLOCK和SYMBOL_REF_BLOCK_OFFSET域。

SYMBOL_FLAG_ANCHOR
: 用来表示符号作为section anchor。``Section anchors''为在object_block中具有一个已知位置的符号，并且可以用来访问该块中的附近成员。它们用来实现`-fsection-anchors'。

  如果该标记被设置，则SYMBOL_FLAG_HAS_BLOCK_INFO也被设置。

起始于SYMBOL_FLAG_MACH_DEP的位，可供目标机器使用。

SYMBOL_REF_BLOCK (x)
: 如果`SYMBOL_REF_HAS_BLOCK_INFO_P (x)'，则其为该符号所属的`object_block'结构体，或者如果其没有被分配给一个块，则为NULL。

SYMBOL_REF_BLOCK_OFFSET (x)
: 如果`SYMBOL_REF_HAS_BLOCK_INFO_P (x)'，则其为x的偏移量，从`SYMBOL_REF_BLOCK (x)'中的第一个对象开始。如果x还没有被分配给一个块，或者其还没有在那个块中给定一个偏移量，则值为负数。

10.5 RTL表达式中的标记

RTL表达式包含几个标记（位域），用于特定类型的表达式。通常它们使用下列的宏来访问，并被扩展为左值。

CONSTANT_POOL_ADDRESS_P (x)
: 位于symbol_ref中，如果其引用了当前函数的常量池中的一部分，则非零。对于大多数目标机器，这些地址在.rodata段中，与函数完全分离，但是对于有些目标机器，这些地址是靠近函数起始处。不管哪种情况，GCC都假设这些地址可以被直接寻址，或者通过基址寄存器。其被存储在unchanging域，打印输出为`/u'。

RTL_CONST_CALL_P (x)
: 位于call_insn中，表明该insn表示一个对const函数的调用。存储在unchanging域，打印输出为`/u'。

RTL_PURE_CALL_P (x)

位于call_insn中，表明该insn表示一个对pure函数的调用。存储在return_val域，打印输出为`/i'。

RTL_CONST_OR_PURE_CALL_P (x)

位于call_insn中，如果RTL_CONST_CALL_P或者RTL_PURE_CALL_P为真，则其为真。

RTL_LOOPING_CONST_OR_PURE_CALL_P (x)

位于call_insn中，表明该insn表示一个对const或者pure函数，可能是无限循环的调用。存储在call域，打印输出为`/c'。只有当RTL_CONST_CALL_P或者RTL_PURE_CALL_P为真时，其才为真。

INSN_ANNULLED_BRANCH_P (x)

位于jump_insn，call_insn或者insn中，表明该分支跳转被取消。参见下面关于sequence的讨论。存储在unchanging域，打印输出为`/u'。

INSN_DELETED_P (x)

位于insn, call_insn, jump_insn, code_label, barrier或note中，如果该insn被删除掉，则非零。存储在volatil域，打印输出为`/v'。

INSN_FROM_TARGET_P (x)

位于insn，jump_insn或者call_insn中，在分支延迟槽中，表明该insn是来自分支跳转的目标。如果分支insn设置了INSN_ANNULLED_BRANCH_P，则该insn只有当进行分支跳转的时候，才被执行。对于被取消的分支，如果是清除了INSN_FROM_TARGET_P，则insn只有当没有进行分支跳转的时候，才被执行。当INSN_ANNULLED_BRANCH_P没有被设置，该insn将总是被执行。存储在in_struct域，打印输出为`/s'。

LABEL_PRESERVE_P (x)

位于code_label或者note中，表明被代码或者数据引用的该标号，对于给定的函数RTL不可见。通过非局部goto引用的标号，将设置该位。存储在in_struct域中，打印输出为`/s'。

LABEL_REF_NONLOCAL_P (x)

位于label_ref和reg_label表达式中，如果其为对一个非局部标号的引用，则非零。存储在volatil域中，打印输出为`/v'。

MEM_IN_STRUCT_P (x)

位于mem表达式中，对于引用一整个结构体，联合体，数组，或者是它们的一部分，则非零。如果是引用一个标量变量，或者是标量指针，则为零。如果该标记和MEM_SCALAR_P都被清除，则我们无法知道该mem是否在一个结构体中。这两个标记不要被同时设置。存储在in_struct域中，打印输出为`/s'。

MEM_KEEP_ALIAS_SET_P (x)

位于mem表达式中，如果为1，则表明当访问一个部件时，应该保持该mem的别名集合不变。例如，当在一个聚合体的不可寻址部件中的时候，将其设为1。存储在jump域中，打印输出为`/j'。

MEM_SCALAR_P (x)

位于mem表达式中，如果引用的标量，被已知不是结构体，联合体或者数组的成员，则为非零。如果是这样的引用，或者通过指针的间接引用，即便是指向标量类型，则为零，如果该标记和MEM_IN_STRUCT_P都被清除，则我们无法知道该mem是否为一个结构体。这两个标记不要被同时设置。存储在return_val域中，打印输出为`/i'。

MEM_VOLATILE_P (x)

位于mem, asm_operands和asm_input表达式中，对于volatile内存引用，为非零。存储在volatil域中，打印输出为`/v'。

MEM_NOTRAP_P (x)

位于mem中，对于不会产生陷阱的内存引用，为非零。存储在call域中，打印输出为`/c'。

MEM_POINTER (x)

位于mem中，如果内存引用存放了一个指针，则非零。存储在frame_related域中，打印输出为`/f'。

REG_FUNCTION_VALUE_P (x)

位于reg中，如果其为存放函数返回值的地方，则非零。（这只发生在硬件寄存器中。）存储在return_val域中，打印输出为`/i'。

REG_POINTER (x)

位于reg中，如果寄存器存放一个指针，则非零。存储在frame_related域中，打印输出为`/f'。

REG_USERVAR_P (x)

位于reg中，如果其对应于用户源代码中出现的一个变量，则非零。对于编译器内部生成的临时对象，则为零。存储在volatil域中，打印输出为`/v'。

RTX_FRAME_RELATED_P (x)

位于insn, call_insn, jump_insn, barrier, 或者set中，如果其为函数序言的一部分，用来设置栈指针，帧指针，或者保存寄存器，则为非零。对于设置用于帧指针的临时寄存器的指令，该标记也应被设置。存储在frame_related域中，打印输出为`/f'。

特别的，在一些RISC目标机器上，对于立即数常量的大小有限制，有时不能直接通过栈指针来到达寄存器的保存区域。这种情况下，一个足够接近寄存器保存区域的临时寄存器被使用，并且正则帧地址（Canonical Frame Address）寄存器，即DWARF2的逻辑帧指针寄存器，必须（临时的）被改成该临时寄存器。所以，设置该临时寄存器的指令必须被标记为RTX_FRAME_RELATED_P。

如果被标记的指令过于复杂（跟据dwarf2out_frame_debug_expr能否处理，而定义的术语），则你还必须要创建一个REG_FRAME_RELATED_EXPR注解，并附加在该指令上。该注解应该包含一个该指令执行计算的简单表达式，即dwarf2out_frame_debug_expr可以处理的。

在带有RTL序言的目标机器上，提供异常处理的支持时，会用到该标记。

MEM_READONLY_P (x)

位于mem中，如果内存是静态分配并且只读的，则非零。

在该上下文中只读，意味着在程序的生命周期中不会被修改，但是不必要是在ROM或者不可写的页中。对于后者，一个常见的例子，是共享库的全局偏移表。该表由运行时加载器初始化，所以内存在技术上是可写的，但是当控制从运行时加载器转移给应用程序时，该内存将不再被修改。

存储在unchanging域中，打印输出为`/u'。

SCHED_GROUP_P (x)

在指令调度过程中，位于insn, call_insn或者jump_insn中，表明前一个insn必须与该insn一起调度。这用来确保特定的指令组不会被指令调度过程分隔开。例如，在

call_insn之前的use insn不可以从call_insn中分开。存储在in_struct域中，打印输出为`/s'。

SET_IS_RETURN_P (x)
对于set，如果是针对一个return，则非零。存储在jump域中，打印输出为`/j'。

SIBLING_CALL_P (x)
对于call_insn，如果该insn为一个sibling call，则非零。存储在jump域中，打印输出为`/j'。

STRING_POOL_ADDRESS_P (x)
对于一个symbol_ref表达式，如果其为对该函数的字符串常量池的寻址，则非零。存储在frame_related域中，打印输出为`/f'。

SUBREG_PROMOTED_UNSIGNED_P (x)
如果subreg对于SUBREG_PROMOTED_VAR_P为非零，并且被引用的对象为零扩展，则返回一个大于零的值；如果保持为符号扩展，则为零；如果是通过ptr_extend指令，进行某种其它方式的扩展，则小于零。存储在unchanging域和volatil域中，打印输出为`/u'和`/v'。该宏只用于获得值，不能用于修改值。使用SUBREG_PROMOTED_UNSIGNED_SET来修改值。

SUBREG_PROMOTED_UNSIGNED_SET (x)
设置subreg中的unchanging和volatil域，来反映零扩展，符号扩展，或其它扩展。如果volatil为零，然后如果unchanging为非零，则意味着零扩展，如果为零，则意味着符号扩展。如果volatil为非零，则通过ptr_extend指令使用了其它某种扩展。

SUBREG_PROMOTED_VAR_P (x)
位于subreg中，如果当访问一个被提升为符合机器描述宏PROMOTED_MODE的(see Section 17.5 [存储布局], page 291)，宽机器模式的对象时，则非零。这种情况下，subreg的机器模式为对象被声明的机器模式，SUBREG_REG的机器模式为存放该对象的寄存器的机器模式。被提升的变量，在每个赋值中，总是被符号扩展或者零扩展成宽机器模式。存储在in_struct域中，打印输出为`/s'。

SYMBOL_REF_USED (x)
位于symbol_ref中，表明x已经被使用。这通常只用于确保x只在外部被声明一次。存储在used中。

SYMBOL_REF_WEAK (x)
位于symbol_ref中，表明x已经被声明为weak。存储在return_val域中，打印输出为`/i'。

SYMBOL_REF_FLAG (x)
位于symbol_ref中，用于机器特定目的的标记。存储在volatil域中，打印输出为`/v'。
大多对SYMBOL_REF_FLAG的使用，是历史性的，并且可以通过SYMBOL_REF_FLAGS来归类。当然，如果目标机器需要多于一个位的存储时，对SYMBOL_REF_FLAGS的使用是强制的。

这些是上面的宏所引用的域：

call
在mem中，1表示该内存引用不会有陷阱。
在call中，1表示该pure或者const调用，可能为无限循环。
在RTL转储中，该标记被表示为`/c'。

`frame_related`

在insn或者set表达式中，1表示其为函数序言的一部分，设置栈寄存器，设置帧寄存器，保存寄存器，或者设置一个用于帧寄存器的临时寄存器。

在reg表达式中，1表示该寄存器存放一个指针。

在mem表达式中，1表示该内存引用存放一个指针。

在symbol_ref表达式中，1表示该引用是对函数的字符串常量池的寻址。

在RTL转储中，该标记被表示为`/f'。

`in_struct` 在mem表达式中，如果所引用的内存数据为整个结构体或者数组，或者一部分，其为1；如果为（或者可能为）一个标量变量。则为0。通过C指针的引用，为0，因为指针可以指向一个标量变量。该信息允许编译器来确定别名的可能情况。

在reg表达式中，如果寄存器整个生命期都包含在某个循环的测试表达式中，则为1。

在subreg表达式中，1表示subreg在访问一个从更宽的机器模式进行提升的对象。

在label_ref表达式中，1表示被引用的标号位于包含发现label_ref的insn的最内层循环的外面。

在code_label表达式中，如果标号不能被删除，则为1。这用于其为非局部goto的目标的标号。对于已经被删除的这样的标号，使用类型为NOTE_INSN_DELETED_LABEL的note来替换。

在insn中，在死代码消除阶段，1表示该insn为死代码。

在insn或者jump_insn中，在针对分支延迟槽中insn的reorg阶段，1表示该insn来自分支跳转的目标。

在insn中，在指令调度阶段，1表示该insn必须与之前的insn一起进行调度。

在RTL转储中，该标记被表示为`/s'。

`return_val` 在reg表达式中，1表示寄存器包含了当且函数的返回值。对于在寄存器中传递参数的机器上，同一个寄存器编号也可以被用作参数，但是这种情况下，该标记不被设置。

在mem中，1表示内存引用为一个已知不为结构体，联合体，数组的成员的标量。

在symbol_ref表达式中，1表示被引用的符号为weak。

在call表达式中，1表示调用是pure。

在RTL转储中，该标记被表示为`/i'。

`jump` 在mem表达式中，1表示当访问一个部件时，应该保持该mem的别名集不变。

在set中，1表示其为一个return。

在call_insn中，1表示其为一个sibling call。

在RTL转储中，该标记被表示为`/j'。

`unchanging` 在reg和mem表达式中，1表示表达式的值不会改变。

在subreg表达式中，如果subreg引用了机器模式已经被提升为一个宽模式的无符号对象，则为1。 references an

在分支指令延迟槽中的insn或jump_insn中，1表示将使用一个被取消的分支。

在symbol_ref表达式中，1表示该符号对函数的常量池进行寻址。

在call_insn中，1表示该指令为对const函数的调用。

在RTL转储中，该标记被表示为`/u'。

used 该标记在函数的RTL生成阶段的结尾被直接使用（不通过访问宏），来计数表达式在insns中出现的次数。出现次数大于一的表达式，根据共享结构的规则（see Section 10.20 [共享], page 146），被复制。

对于reg，其被叶子寄存器重编号代码直接使用（不通过访问宏），来确保每个寄存器只被重编号一次。

在symbol_ref中，其表示该符号的外部声明已经被书写了。

volatil 在mem, asm_operands或者asm_input表达式中，如果内存引用是volatile的，则为1。volatile的内存引用不可以被删除，重排或者合并。

在symbol_ref表达式中，其用于机器特定的目的。

在reg表达式中，如果值为用户级的变量，则为1。0表示为内部的编译器临时对象。

在insn中，1表示该insn已经被删除。

在label_ref和reg_label表达式中，1表示对非局部标号的引用。

在RTL转储中，该标记被表示为`/v'。

10.6 机器模式

机器模式描述数据对象的大小及其表示。在C代码中，机器模式表示成枚举类型enum machine_mode。此类型定义在`machmode.def'中。每个RTL表达式都有机器模式域。一些树结构如变量定义、类型等也有机器模式域。

在调试信息及机器描述中，RTL表达式的机器模式紧跟在RTL代码之后，其间用冒号隔开。每一种机器模式名末尾的字母省缺为`mode'。例如：(reg:SI 38)是一个reg表达式，其机器模式为SImode。如果方式为VOIDmode，表达式中完全不出现此模式。

以下是一个机器模式表，这里的“字节”是指具有BITS_PER_UNIT个存储位的对象（see Section 17.5 [存储布局], page 291）。

BImode “Bit”模式，表示一位，用于断言寄存器。

QImode “Quarter-Integer”模式，表示一个一字节的整数。

HImode “Half-Integer”模式，表示一个两字节的整数。

PSImode “Partial Single Integer”模式，表示一个占有四个字节但并不真正使用全部四个字节的整数。

SImode “Single Integer”模式，表示一个四字节的整数。

PDImode “Partial Double Integer”模式，表示一个占有八个字节但并不真正使用全部八个字节的整数。

DImode “Double Integer”模式，表示一个8字节的整数。

TImode “Tetra Integer”模式，表示一个16字节的整数。

OImode “Octa Integer”模式，表示一个32字节的整数。

QFmode “Quarter-Floating”模式，表示一个四分之一精度（单字节）浮点数。

HFmode “Half-Floating”模式，表示一个二分之一精度（双字节）浮点数。

TQFmode “Three-Quarter-Floating”模式，表示一个四分之三精度（单字节）浮点数。

SFmode “Single Floating”模式，表示一个单精度（4个字节）浮点数。

DFmode “Double Floating”模式，表示一个双精度（8 字节）浮点数。

XFmode “Extended Floating”模式，表示一个三精度（12 字节）浮点数。本方式用来表示 IEEE 扩展浮点类型。

SDmode ``Single Decimal Floating''模式，表示一个四字节十进制浮点数（区别于常规的二进制浮点）。

DDmode ``Double Decimal Floating''模式，表示一个八字节十进制浮点数。

TDmode ``Tetra Decimal Floating''模式，表示一个十六字节十进制浮点数，所有128位都有含义。

TFmode “Tetra Floating”模式，表示一个四精度（16 字节）浮点数。

QQmode ``Quarter-Fractional''模式，表示一个单字节的有符号小数。缺省格式为``s.7''。

HQmode ``Half-Fractional''模式，表示一个双字节的有符号小数。缺省格式为``s.15''。

SQmode ``Single Fractional''模式，表示一个四字节的有符号小数。缺省格式为``s.31''。

DQmode ``Double Fractional''模式，表示一个八字节的有符号小数。缺省格式为``s.63''。

TQmode ``Tetra Fractional''模式，表示一个十六字节有符号小数。缺省格式为``s.127''。

UQQmode ``Unsigned Quarter-Fractional''模式，表示一个单字节的无符号小数。缺省格式为``.8''。

UHQmode ``Unsigned Half-Fractional''模式，表示一个双字节的无符号小数。缺省格式为``.16''。

USQmode ``Unsigned Single Fractional''模式，表示一个四字节的无符号小数。缺省格式为``.32''。

UDQmode ``Unsigned Double Fractional''模式，表示一个八字节的无符号小数。缺省格式为``.64''。

UTQmode ``Unsigned Tetra Fractional''模式，表示一个十六字节的无符号小数。缺省格式为``.128''。

HAmode ``Half-Accumulator''模式，表示一个双字节的有符号累加器。缺省格式为``s8.7''。

SAmode ``Single Accumulator''模式，表示一个四字节的有符号累加器。缺省格式为``s16.15''。

DAmode ``Double Accumulator''模式，表示一个八字节的有符号累加器。缺省格式为``s32.31''。

TAmode ``Tetra Accumulator''模式，表示一个十六字节的有符号累加器。缺省格式为``s64.63''。

UHAmode ``Unsigned Half-Accumulator''模式，表示一个双字节的无符号累加器。缺省格式为``8.8''。

USAmode ``Unsigned Single Accumulator''模式，表示一个四字节的无符号累加器。缺省格式为``16.16''。

UDAmode ``Unsigned Double Accumulator''模式，表示一个八字节的无符号累加器。缺省格式为``32.32''。

UTAmode ``Unsigned Tetra Accumulator''模式，表示一个十六字节的无符号累加器。缺省格式为``64.64''。

CCmode “condition code”模式，表示条件代码的值。其中条件代码是一组与机器相关的位集合，用来表示比较的结果。在使用了 cc0 的机器上，不用CCmode。

BLKmode “Block”模式，表示其它模式都不适用的聚合值。在 RTL 中，只有内存引用才能有此方式，并且仅当出现在字符串移动指令或向量指令中时，才能有此方式。若机器没有上述指令，则BLKmode将不出现在 RTL中。

VOIDmode 意味着模式不出现或一个不确定的模式。例如：const_int表达式的模式就是VOIDmode，因为此类表达式可认为具有其上下文所要求的任何模式。在RTL的调试输出中，VOIDmode表示成没有任何模式出现。

QCmode, HCmode, SCmode, DCmode, XCmode, TCmode
这些方式模式由一对浮点数组成的复数。其中，浮点数分别具有QFmode、HFmode, SFmode, DFmode, XFmode和TFmode。

CQImode, CHImode, CSImode, CDImode, CTImode, COImode
这些模式代表由一对整数组成的复数。整数值分别具有方式QImode, HImode,SImode, DImode, TImode和OImode。

机器描述定义Pmode为一个C宏，其扩展为用于寻址的机器模式。通常这是一个在32位机器上，大小为BITS_PER_WORD, SImode模式。

机器描述唯一必须要支持的机器模式为QImode，以及对应于BITS_PER_WORD，FLOAT_TYPE_SIZE和DOUBLE_TYPE_SIZE的机器模式。编译器将尝试使用DImode，表示8字节的结构体和联合体，不过这可以通过重写MAX_FIXED_MODE_SIZE的定义来阻止。替换的，你可以让编译器使用TImode表示16字节的结构体和联合体。同样，你可以使得C类型short int避免使用HImode。

编译器中，有很少的对机器模式显式的引用，并且这些引用将不久被移除掉。替代的，机器模式被分成机器模式类别。这些由定义在`machmode.h'中的枚举类型enum mode_class来表示。可能的机器类别有：

MODE_INT 整数模式。缺省情况下，它们是BImode, QImode, HImode, SImode, DImode, TImode和OImode。

MODE_PARTIAL_INT
部分整数模式，PQImode, PHImode, PSImode和PDImode。

MODE_FLOAT 浮点模式。缺省情况下，这些是QFmode, HFmode, TQFmode, SFmode, DFmode, XFmode和TFmode。

MODE_DECIMAL_FLOAT
十进制浮点模式。缺省的，这些是SDmode, DDmode和TDmode。

MODE_FRACT 有符号小数模式。缺省的，这些是QQmode, HQmode, SQmode, DQmode和TQmode。

MODE_UFRACT
无符号小数模式。缺省的，这些是UQQmode, UHQmode, USQmode, UDQmode和UTQmode。

MODE_ACCUM 有符号累加器模式。缺省的，这些HAmode, SAmode, DAmode和TAmode。

MODE_UACCUM
无符号累加器模式。缺省的，这些是UHAmode, USAmode, UDAmode和UTAmode。

MODE_COMPLEX_INT
: 复数整数模式。（当前还没有被实现。）

MODE_COMPLEX_FLOAT
: 复数浮点模式。缺省情况下，为QCmode, HCmode, SCmode, DCmode, XCmode和TCmode。

MODE_FUNCTION
: Algol或者Pascal函数变量，包括一个静态链。（这些目前还没有被实现。）

MODE_CC
: 表示条件码的值的模式。为CCmode加上在`machine-modes.def'中列出的任何CC_MODE模式。See Section 16.12 [跳转指令模式], page 254, 同时参见Section 17.16 [条件代码], page 344。

MODE_RANDOM
: 这是所有不适合上面的类别的模式。目前VOIDmode和BLKmode包括在MODE_RANDOM中。

这些是与机器模式相关的C宏：

GET_MODE (x)
: 返回RTX x的机器模式。

PUT_MODE (x, newmode)
: 将RTX x的机器模式修改为newmode。

NUM_MACHINE_MODES
: 表示目标机器上可用的机器模式的个数。比最大的机器模式数值大1。

GET_MODE_NAME (m)
: 返回机器模式m的字符串名字。

GET_MODE_CLASS (m)
: 返回机器模式m的类别。

GET_MODE_WIDER_MODE (m)
: 返回下一个宽的自然的机器模式。例如，表达式GET_MODE_WIDER_MODE (QImode)返回HImode。

GET_MODE_SIZE (m)
: 返回机器模式m的字节数。

GET_MODE_BITSIZE (m)
: 返回机器模式m位数。

GET_MODE_IBIT (m)
: 返回定点机器模式m的整数位数。

GET_MODE_FBIT (m)
: 返回定点机器模式m的小数位数。

GET_MODE_MASK (m)
: 返回一个位掩码。该宏只能用于位长度小于或等于HOST_BITS_PER_INT的机器模式。

GET_MODE_ALIGNMENT (m)
: 对于模式为m的对象，返回所要求的对齐方式，以位数为单位。

GET_MODE_UNIT_SIZE (m)
: 返回模式为m的数据的子单元大小，以字节为单位。这与GET_MODE_SIZE相同，除了复数模式。对于它们，单元大小为实部或者虚部的大小。

GET_MODE_NUNITS (m)
返回在一个模式中包含的单元数目，即GET_MODE_UNIT_SIZE除以GET_MODE_SIZE。

GET_CLASS_NARROWEST_MODE (c)
返回机器模式类别c中的最窄的模式。

全局变量byte_mode和word_mode包含了类别为MODE_INT，并且位大小分别是BITS_PER_UNIT或BITS_PER_WORD的机器模式。在32位机器上，这些分别是QImode和SImode。

10.7 常量表达式类型

最简单的RTL表达式是那些对常数值的表示。

(const_int i)
这类表达式表示整数值i。i通常通过宏INTVAL来访问，INTVAL (exp)相当于XWINT (exp, 0)，为模式所生成的常量如果位数小于HOST_WIDE_INT，则必须符号扩展为全部宽度（例如，gen_int_mode）。

只有一个表达式对象表示整数值0；是变量const0_rtx的值。同样的，整数值1的唯一表达式为const1_rtx，整数值2的唯一表达式为const2_rtx，负1的唯一表达式为constm1_rtx。任何试图去创建值为0，1，2或者-1的const_int都会返回相应的const0_rtx, const1_rtx, const2_rtx或者constm1_rtx。

类似的，只有一个对象表示值为STORE_FLAG_VALUE的整数，其为const_true_rtx。如果STORE_FLAG_VALUE为1，则const_true_rtx和const1_rtx将会指向同一对象。如果STORE_FLAG_VALUE为-1，const_true_rtx和constm1_rtx将会指向同一对象。

(const_double:m i0 i1 ...)
表示或者为一个模式为m的浮点常量，或者为一个超过HOST_BITS_PER_WIDE_INT位的，但是小于其两倍的位数的整数常量（GCC并没有提供表示更大的常量的机制）。对于后者，m将为VOIDmode。

如果m为VOIDmode，则值的位数存储在i0和i1中。i0使用宏CONST_DOUBLE_LOW来访问，i1使用CONST_DOUBLE_HIGH。

如果常量为浮点（不管是什么精度），则用于存储值得整数数目取决于REAL_VALUE_TYPE的大小（see Section 17.23 [浮点], page 382）。整数表示一个浮点数，但是不如target机器的或者host机器的浮点格式那么精确。要将它们转换为target机器使用的精确的位模式，使用宏REAL_VALUE_TO_TARGET_DOUBLE等（see Section 17.21.2 [数据输出], page 360）。

(const_fixed:m ...)
表示一个机器模式为m定点常量。操作数是一个类型为struct fixed_value数据结构，并使用宏CONST_FIXED_VALUE来访问。数据的高部分使用CONST_FIXED_VALUE_HIGH来访问；低部分使用CONST_FIXED_VALUE_LOW来访问。

(const_vector:m [x0 x1 ...])
表示一个向量常量。方括号代表向量包含的常量元素。x0，x1等等为const_int，const_double或者const_fixed元素。

在const_vector中的单元数可以通过宏CONST_VECTOR_NUNITS来获得，形如CONST_VECTOR_NUNITS (v)。

在向量常量中的单个元素使用宏CONST_VECTOR_ELT来访问，形如CONST_VECTOR_ELT (v, n)，其中v为向量常量，n为要访问的元素编号。

(const_string str)
表示一个具有值str的常量字符串。目前这只用于insn属性（see Section 16.19 [Insn属性], page 266），因为C中的常量字符串存放在内存中。

(symbol_ref:mode symbol)
表示引用数据的汇编标号的值。symbol为一个字符串，描述了汇编标号的名字。如果其起始于`*'，则标号为symbol不包含`*'的其余部分。否则，标号为symbol，通常具有前缀`_'。

symbol_ref包含一个机器模式，其通常为Pmode。通常这是唯一的使得符号有效的机器模式。

(label_ref:mode label)
表示引用代码的汇编标号的值。其包含一个操作数，一个表达式，其必须为一个code_label或者一个NOTE_INSN_DELETED_LABEL类型的note，其出现在指令序列中，来标识标号应该处于的位置。

对于代码标号引用，使用一个不同的表达式类型的原因是跳转优化可以区分它们。

label_ref包含了一个机器模式，其通常为Pmode。通常这是唯一的使得标号有效的机器模式。

(const:m exp)
表示一个常量，其为汇编时算术计算的结果。操作数exp，为一个表达式，只包含了plus和minus组合的常量（const_int, symbol_ref 和 label_ref表达式）。然而，并不是所有的组合都是有效的，因为汇编器不能对重定位符号做任意的算术运算。

m应该为Pmode。

(high:m exp)
表示exp的高位，通常为一个symbol_ref。位数是机器相关的并且通常为初始化一个寄存器的高位的指令所指定的位数。其和lo_sum一起使用来表示典型的用于RISC机器的双指令序列来引用一个全局内存位置。

m应该为Pmode。

宏CONST0_RTX (mode)指定一个具有值0，机器模式为mode的表达式。如果mode为MODE_INT类别，则返回const0_rtx。如果mode为 MODE_FLOAT类别，则返回模式为mode的表达式CONST_DOUBLE。否则，其返回一个模式为mode的表达式CONST_VECTOR。类似的，宏CONST1_RTX (mode)指定一个具有值1，机器模式为mode的表达式，类似的对于CONST2_RTX。宏CONST1_RTX和CONST2_RTX对于向量模式没有定义。

10.8 寄存器和内存

这些是描述访问机器寄存器和内存的RTL表达式类型。

(reg:m n) 对于值小（那些小于FIRST_PSEUDO_REGISTER）的整数n，这表示对机器寄存器号为n的引用。对于值大的n，它表示一个临时的值或者伪寄存器（pseudo register）。编译器的策略是，在生成代码时假设有无限数目的伪寄存器，并在之后将它们转换为硬件寄存器（hard register）或者内存的引用。

m为引用的机器模式。指定机器模式是有必要的，因为机器通常可以使用多种机器模式来引用每个寄存器。例如，一个寄存器可以包含一个整字，但是可以有指令来作为半字或者一个单独的字节来引用，同样，也有指令可以作为不同精度的浮点来引用。

即使对于机器只能使用一种机器模式来引用的寄存器，mode也必须被指定。

符号FIRST_PSEUDO_REGISTER被机器描述定义，由于机器的硬件寄存器数目是一个不变的特征。然而要注意，并不是所有的机器寄存器都必须是通用寄存器。所有的可以用于存储数据的机器寄存器都为被给定硬件寄存器编号，即使那些只可以被用于特定指令或者只能存放特定类型数据。

一个硬件寄存器可以在整个函数中按照多种机器模式来访问，但是每个伪寄存器都被给定一个自然的机器模式并且只能按照那个机器模式来访问。当需要描述一个使用非自然机器模式来访问伪寄存器的时候，则使用一个subreg表达式。

一个reg表达式具有的机器模式，如果指定了多于一个字的数据，则实际上代表了多个连续的寄存器。如果寄存器编号还指定了一个硬件寄存器，则其实际上表示起始于指定寄存器的多个连续的硬件寄存器。

每个在函数的RTL代码中使用的伪寄存器编号，使用一个唯一的reg表达式来表示。

一些伪寄存器编号，处于FIRST_VIRTUAL_REGISTER到LAST_VIRTUAL_REGISTER之间，只在RTL生成阶段出现并且在优化阶段之前被消除。这些表示在栈帧中的位置，并且直到函数的RTL生成完成后才能确定。下列虚寄存器编号被定义：

VIRTUAL_INCOMING_ARGS_REGNUM
: 这指向栈上传递的参数的第一个字。通常这些参数由调用者存放，但是被调用者可能已经将之前在寄存器中传递的参数压入栈中。

  当RTL生成完成时，该虚寄存器由ARG_POINTER_REGNUM给定的寄存器和FIRST_PARM_OFFSET的值的和替换。

VIRTUAL_STACK_VARS_REGNUM
: 如果FRAME_GROWS_DOWNWARD被定义为非0的值，则该宏指向栈上第一个变量的上一个位置。否则，其指向栈上的第一个变量。

  VIRTUAL_STACK_VARS_REGNUM由FRAME_POINTER_REGNUM给定的寄存器和STARTING_FRAME_OFFSET的值的和替换。

VIRTUAL_STACK_DYNAMIC_REGNUM
: 该宏指向栈指针根据内存需要已经被调整后的栈上动态分配内存的位置。

  该虚寄存器由STACK_POINTER_REGNUM给定的寄存器和STACK_DYNAMIC_OFFSET的值的和替换。

VIRTUAL_OUTGOING_ARGS_REGNUM
: 其指向栈中，当预先压栈时（使用push insn压栈的参数应该总是使用STACK_POINTER_REGNUM），书写输出参数的位置。

  该虚拟寄存器，被替换成由STACK_POINTER_REGNUM给定的寄存器与值STACK_POINTER_OFFSET的和。

(subreg:m1 reg:m2 bytenum)
: subreg表达式用于按照自然的机器模式之外的，其它机器模式来引用一个寄存器，或者引用有多个寄存器组成的reg的其中一个寄存器。

  每个伪寄存器都具有一个自然的机器模式。如果需要按照不同的机器模式来对其操作，则寄存器必须用subreg进行包含。

  目前对于subreg的第一个操作数，有三种被支持的类型：

  - pseudo registers 这是最常见的情况。大多subreg将伪寄存器reg作为它们的第一个操作数。

- mem mem的subreg，在早期版本的GCC中比较常见，现在仍被支持。在重载过程中，这些被普通的mem替换掉。在不进行指令调度的机器上，仍然使用mem的subreg，但是不推荐这样。在重载过程之前和过程之中，这样的subreg被考虑成register_operand，而不是memory_operand。因此，调度过程无法对具有mem的subreg这样的指令进行合适的调度。所以，对于进行调度的机器，不要使用mem的subreg。为此，当INSN_SCHEDULING被定义的时候，合并过程和recog过程，具有显式的代码来禁止创建mem的subreg。

  在重载过程之后使用mem的subreg，将难以理解，应该避免这样。编译器中还有一些代码支持这些，但是这些代码可能已经过时了。这种subreg的用法不被推荐，将来很可能不被支持。
- hard registers 很少有必要在subreg中包裹硬件寄存器；这样的寄存器通常应该被缩减为一个单独的reg rtx。这种subreg的用法不被推荐，将来可能不被支持。

subreg的subreg不被支持。推荐使用simplify_gen_subreg来避免这种问题。

subreg有两种不同的风格，分别具有自己的用法和规则：

Paradoxical subregs

当m1严格宽于m2的时候，subreg表达式被称作反常的（paradoxical）。对该类别的subreg的正规测试为：

```
GET_MODE_SIZE (m1) > GET_MODE_SIZE (m2)
```

反常的subreg可以用于左值和右值。当用于左值时，源值的低位被存储在reg中，高位被丢弃。当用作右值时，subreg的低位来自reg，而高位可以被定义，也可以未被定义。

右值的高位有以下几种情况：

- subregs of mem 当m2小于一个字的时候，宏LOAD_EXTEND_OP可以控制高位如何被定义。
- subreg of regs 当SUBREG_PROMOTED_VAR_P为真时，高位被定义。SUBREG_PROMOTED_UNSIGNED_P描述了高位的内容。这样的subreg通常表示已经被提升为更宽的机器模式的局部变量，寄存器变量以及参数伪变量。

对于反常的subreg，bytenum总是为零，即使在大端的目标机上。

例如反常的subreg:

```
(set (subreg:SI (reg:HI x) 0) y)
```

在x中存储了y的低位2个字节，并丢弃高位2个字节。接着：

```
(set z (subreg:SI (reg:HI x) 0))
```

将会把z的低位2个字节设置成x，并将高位两个字节设置为未知的值，假定SUBREG_PROMOTED_VAR_P为假。

Normal subregs

当m1最多跟m2一样宽的时候，subreg表达式被称作正常的（normal）。

正常的subreg被限定为reg的特定位。有两种情况。如果m1比一个字小，则subreg指的是一个reg字的最小有效部分。如果m1为字大小，或者更大，则subreg指的是一个或者更多个完整的字。

当用作左值时，subreg为一个基于字的访问。对subreg进行存储，会修改reg中所有与subreg重叠的字，并将reg中的其它字保持不变。

当对小于一个字的正常subreg进行存储的时候，被引用的字的其它位通常处于未定义的状态。这种松弛的方式易于对这样的指令产生高效的代码。要表示保持subreg之外的所有位的指令，在subreg周围使用strict_low_part或者zero_extract。

bytenum必须标识从reg的起始，subreg的第一个字节的偏移量，假设reg按照内存的顺序布局。字节的内存顺序通过两个目标宏定义，WORDS_BIG_ENDIAN和BYTES_BIG_ENDIAN：

- WORDS_BIG_ENDIAN，如果设为1，则说明第零个字节为最大有效字的部分；否则，为最小有效字的部分。
- BYTES_BIG_ENDIAN，如果设为1，则说明第零个字节为字中的最高有效字节；否则，为字中的最低有效字节。

在一些目标机上，FLOAT_WORDS_BIG_ENDIAN与WORDS_BIG_ENDIAN不一致。然而，编译器的大部分地方会将浮点值看作它们与整数值具有相同的大小端。这是因为只将它们作为整数值的集合来处理，没有特定的数值。只有real.c和运行时库关心FLOAT_WORDS_BIG_ENDIAN。

因此，

```
(subreg:HI (reg:SI x) 2)
```

在一个BYTES_BIG_ENDIAN，`UNITS_PER_WORD == 4'的目标机上，等同于

```
(subreg:HI (reg:SI x) 0)
```

在一个小端，`UNITS_PER_WORD == 4'的目标机上。两个subreg都是访问寄存器x的低两个字节。

MODE_PARTIAL_INT机器模式的行为就好像其与相对应的MODE_INT机器模式一样宽，只不过其具有未知数目的未定义的位。例如：

```
(subreg:PSI (reg:SI 0) 0)
```

访问整个`(reg:SI 0)'，但是PSImode值和SImode值的确切关系没有被定义。如果我们假设`UNITS_PER_WORD <= 4'，则下面两个subreg：

```
(subreg:PSI (reg:DI 0) 0)
(subreg:PSI (reg:DI 0) 4)
```

表示对`(reg:DI 0)'的两个部分进行无关的四个字节访问。每个subreg都具有未知数目的未定义位。

如果`UNITS_PER_WORD <= 2'，则这两个subreg

```
(subreg:HI (reg:PSI 0) 0)
(subreg:HI (reg:PSI 0) 2)
```

表示无关的两个字节访问，一起贯穿整个`(reg:PSI 0)'。对第一个subreg进行存储不影响第二个的值，反之亦然。`(reg:PSI 0)'具有未知数目的未定义位，所以赋值：

```
(set (subreg:HI (reg:PSI 0) 0) (reg:HI 4))
```

不保证`(subreg:HI (reg:PSI 0) 0)'具有值`(reg:HI 4)'。

上面的规则应用于伪寄存器reg和硬件寄存器reg。如果对于m1，m2和硬件寄存器reg的特定组合，其语义不正确，则目标机特定的代码必须确保这些组合不会被用到。例如：

```
CANNOT_CHANGE_MODE_CLASS (m2, m1, class)
```

必须为真，对于每个包含reg的类别class。

subreg表达式的第一个操作数通常使用SUBREG_REG宏来访问，第二个操作数通常使用SUBREG_BYTE宏来访问。

BYTES_BIG_ENDIAN不等于WORDS_BIG_ENDIAN的平台是在很多年前被测试的。对于希望在将来支持这样一个平台的人们，可能会面对一些过时的代码。

(scratch:m)

这表示一个scratch寄存器，其在单个指令的执行中用到，并随后不再被使用。其被局部寄存器分配或者重载过程，转换成一个reg。

scratch通常位于clobber操作中。(see Section 10.15 [副作用], page 134)。

(cc0) 为机器的条件代码寄存器。其没有参数，并可以没有机器模式。有两种使用它的方式：

- 表示一个完整的条件代码标记的集合。这在大多机器上是最好的方式，每个比较都会设置整个标记系列。

  使用这种技术，(cc0)只在两种上下文中有效：为一个赋值的对象（在测试和比较指令中）和在跟零进行比较的比较运算符中（值为零的const_int；也就是说，const0_rtx）。

- 表示单个标记，为单个条件的结果。这用于只有一个标记位，比较指令必须指定要测试的条件的机器上。

  使用这种技术，(cc0)只在两种上下文中有效：为一个赋值的对象（在测试和比较指令中），其中源操作数为一个比较运算符，以及if_then_else的第一个参数（在条件分支中）。

只有一个代码为cc0的表达式对象；其为变量cc0_rtx的值。任何尝试创建一个代码为cc0的表达式，将返回cc0_rtx。

指令可以隐式的设置条件代码。在许多机器上，几乎所有的指令根据它们计算或者存储的值来设置条件码。没有必要在RTL中显式的记录这些行为，因为机器描述包含一个对策，用于识别这样做的指令（通过宏NOTICE_UPDATE_CC）。See Section 17.16 [条件代码], page 344. 只有目的纯粹是设置条件码的指令，以及使用条件码的指令，才需要提及(cc0)。

在一些机器上，条件码寄存器被给定一个寄存器编号，并且一个reg用于替代(cc0)。这通常为更好的方式，如果只有一个小的指令子集修改条件码。其它机器将条件码存储在通用寄存器中；这种情况下应该使用伪寄存器。

一些机器，例如SPARC和RS/6000，具有两个算术指令集合，一个设置条件码，另一个不设置。可以通常情况下生成不设置条件码的指令，并创建一个同时执行算术运算并设置条件码寄存器（这种情况下将不会是(cc0)）的指令模式。例如，搜一下`sparc.md'中的`addcc'和`andcc'。

(pc) 表示机器的程序计数器。其没有操作数并可能没有机器模式。(pc)只在跳转指令的特定上下文中使用。

只有一个代码为pc的表达式对象；其为变量pc_rtx的值。任何尝试创建一个代码为pc的表达式，将返回pc_rtx。

所有不进行跳转的指令会隐式的通过递增的方式改变程序计数器，但是不需要在RTL中提起这些。

(mem:m addr alias)

该RTX表示对表达式addr所表示的地址的主内存进行引用。m描述了被访问的内存的单元大小。alias描述了该引用的别名集合。总得来说，两个项如果不引用相同的内存地址，则在不同的别名集合里。

结构(mem:BLK (scratch))被认为是所有其它内存的别名。因此其可以在函数尾声的栈销毁中用作内存栅栏。

(concatm rtx rtx)
该RTX表示对两个其它RTX的连结。这用于复数值。其应该只出现在附加在声明中的RTL中，以及RTL生成中。不应该出现在普通的insn链上。

(concatnm [rtx ...])
该RTX表示将所有的rtx进行连结，生成一个单个的值。类似concat，其应该只出现在声明中，不应该出现在insn链上。

10.9 RTL算术运算表达式

除非其它规定，所有算术表达式的操作数必须对模式m有效。一个操作数对模式m有效，是指当它具有模式m，或者如果它是一个const_int或者const_double，并且m是一个MODE_INT类的模式。

对于可交换的二进制操作，常量应该放到第二个操作数的位置。

(plus:m x y)
(ss_plus:m x y)
(us_plus:m x y)
这三个表达式都表示x和y所表示的值的和，机器模式为m。它们在整数机器模式的溢出方面有所不同。plus以m的宽度求模进行环绕；ss_plus饱和为m可表示的有符号最大值；us_plus饱和为无符号最大值。

(lo_sum:m x y)
该表达式表示x与y低位的和。其跟high (see Section 10.7 [常数], page 122)一起使用，来表示在RISC机器中通常使用的两个指令序列，来引用一个全局内存位置。

低位的位数是机器相关的，但通常为Pmode中的位数减去high所设置的位数。

m应该为Pmode。

(minus:m x y)
(ss_minus:m x y)
(us_minus:m x y)
这三个表达式表示从x中减去y的结果，机器模式为m。在溢出方面的行为与plus的三种版本相同（参见上面）。

(compare:m x y)
表示从x中减去y的结果，用于进行比较。计算结果不产生溢出，就好像是有无限的精度一样。

当然，机器不会真的进行无限精度的减法。然而，它们可以假定这样做，当只使用结果的正负符号时，这样情况下，结果被存放在条件代码中。并且，这是这种表达式唯一可以被使用的方式：作为值存储在条件代码中，或者(cc0)，或者一个寄存器。See Section 10.10 [比较运算], page 131.

机器模式m与x和y的机器模式没有关联，而是条件代码值的机器模式。如果使用(cc0)，则为VOIDmode，否则为类别MODE_CC中的某个模式，通常为CCmode。See Section 17.16 [条件代码], page 344. 如果m为VOIDmode或者CCmode，则运算会返回足够的信息，使得任何比较运算符可以被应用到COMPARE运算的结果上。对于类别MODE_CC中的其它机器模式，运算只返回信息的子集。

通常，x和y必须具有相同的机器模式。否则，compare只有当x的机器模式在类别MODE_INT中，并且y为一个机器模式为VOIDmode的const_int或者const_double，这时才有效。x的机器模式决定了比较按照什么机器模式进行；因此其不能为VOIDmode。

如果其中一个操作数为常量，则其应该被放在第二个操作数的位置，并且相应的调整比较代码。

指定两个VOIDmode常量的compare是无效的，因为无法知道比较要按照什么机器模式进行；比较必须或者在编译过程中被折叠，或者第一个操作数必须被加载到机器模式已知的寄存器中。

(neg:m x)
(ss_neg:m x)
(us_neg:m x)

这两个表达式表示x所表示的值的负数（零减去该值），机器模式为m。它们在整数机器模式的溢出行为上有所不同。对于neg，操作数的负数可以为无法用机器模式m表示的数，这种情况下，其被截取为m。ss_neg和us_neg确保超出边界的结果饱和为最大或者最小的有符号或者无符号值。

(mult:m x y)
(ss_mult:m x y)
(us_mult:m x y)

表示x和y所表示的值的有符号乘积，机器模式为m。ss_mult和us_mult确保超出边界的结果饱和为最大或者最小的有符号或者无符号值。

一些机器支持产生比操作数更宽的乘积。则指令模式可以写成

```
(mult:m (sign_extend:m x) (sign_extend:m y))
```

其中m比x和y的机器模式更宽。

对于无符号的加宽的乘法，使用相同的语句，只不过把sign_extend替换成zero_extend。

(div:m x y)
(ss_div:m x y)

表示x有符号除以y的商，机器模式为m。如果m为一个浮点机器模式，则表示确切的商；否则为整数化的商。ss_div确保超出边界的结果饱和为最大或者最小的有符号值。

一些机器具有的除法指令，其操作数和商的宽度不全相同；你应该使用truncate和sign_extend来表示这样的指令，

```
(truncate:m1 (div:m2 x (sign_extend:m2 y)))
```

(udiv:m x y)
(us_div:m x y)

类似div，不过表示无符号除法。us_div确保超出边界的结果饱和为最大或者最小的无符号值。

(mod:m x y)
(umod:m x y)

类似div和udiv，不过表示余数。

(smin:m x y)
(smax:m x y)

表示x和y的较小值（smin）或者较大值（smax），按照机器模式为m的有符号值解析。当用于浮点，如果两个操作数都为零，或者其中一个为NaN，则没有规定哪一个操作数被作为结果返回。

(umin:m x y)
(umax:m x y)

类似smin和smax，不过值被解析为无符号整数。

(not:m x) 表示对x所表示的值进行按位求补，机器模式为m，且必须为一个定点机器模式。

(and:m x y)
表示对x和y所表示的值按位进行逻辑与，机器模式为m，且必须为一个定点机器模式。

(ior:m x y)
表示对x和y所表示的值按位进行逻辑或，机器模式为m，且必须为一个定点机器模式。

(xor:m x y)
表示对x和y所表示的值按位进行逻辑异或，机器模式为m，且必须为一个定点机器模式。

(ashift:m x c)
(ss_ashift:m x c)
(us_ashift:m x c)
这三个表达式用来表示对x进行向左算术移位c。它们在整数机器模式的溢出方面有所不同。ashift运算是一个普通的移位，当符号位有改变时，其没有特殊的行为；ss_ashift和us_ashift，饱和为可表示的最小或者最大值，如果任何被移出的位与最终的符号位不同。
x具有机器模式m，一个定点机器模式。c为一个定点机器模式或者一个模式为VOIDmode的常量。

(lshiftrt:m x c)
(ashiftrt:m x c)
类似于ashift，不过是向右移位。不像向左移位的情况，这两种运算是有区别的。

(rotate:m x c)
(rotatert:m x c)
类似的，只不过是表示向左和向右旋转。如果c为常量，则使用rotate。

(abs:m x) 表示x的绝对值，按照机器模式m来计算。

(sqrt:m x)
表示x的平方根，按照机器模式m来计算。m通常为浮点机器模式。

(ffs:m x) 表示在x中，最低有效，位为1的索引加上1，为一个模式m的整数。（如果x为零，则值为零。）x的机器模式不需要为m；取决于目标机器，可以有不同的机器模式的组合。

(clz:m x) 表示x中，从最高有效位开始，起始处为0的位数，为一个模式m的整数。如果x为零，则值由CLZ_DEFINED_VALUE_AT_ZERO (see Section 17.30 [其它], page 389)来确定。注意，。x的机器模式通常为一个整数模式。

(ctz:m x) 表示x中，从最低有效位开始，结尾处为0的位数，为一个模式m的整数。如果x为零，则值由CTZ_DEFINED_VALUE_AT_ZERO (see Section 17.30 [其它], page 389)来确定。除此之外，ctz(x)等价于ffs(x) - 1。x的机器模式通常为一个整数模式。

(popcount:m x)
表示x中为1的位数，为一个模式m的整数。x的机器模式通常为一个整数模式。

(parity:m x)
表示x中为1的位数对2进行求模，为一个模式m的整数。x的机器模式通常为一个整数模式。

(bswap:m x)
表示将x值的字节顺序进行反转，结果为m机器模式，其必须为一个定点机器模式。

10.10 比较运算

比较运算符测试两个操作数的关系，对于结果具有`MODE_INT'机器模式的比较运算，如果关系成立，则表示成机器相关的非零值，其由STORE_FLAG_VALUE (see Section 17.30 [其它], page 389)描述，但是不需要相等，如果不成立，则为零。对于结果为浮点值的比较运算，如果关系成立，则为FLOAT_STORE_FLAG_VALUE (see Section 17.30 [其它], page 389)，否则为零。对于返回向量结果的比较运算，如果关系成立，则为VECTOR_STORE_FLAG_VALUE (see Section 17.30 [其它], page 389)，否则为零向量。比较运算的机器模式独立于被比较的数据的机器模式。如果正在测试比较运算（例如，if_then_else的第一个操作数），则机器模式必须为VOIDmode。

有两种方式可以被比较运算使用。比较运算符可以用于将条件代码(cc0)与零进行比较，型如(eq (cc0) (const_int 0))。这种结构实际上是用到了先前指令的结果，条件代码在那里被设置。设置条件代码的指令必须邻接于使用条件代码的指令；只有note insn可以分开它们。

替换的，比较运算可以直接比较两个数据对象。比较运算的机器模式由操作数来决定；它们必须对一个共同的机器模式有效。对两个操作数都为常量的比较，将是无效的，因为不能从中推导出机器模式，不过这样的比较不会出现在RTL中，因为常数折叠。

在上面的例子中，如果(cc0)最后被设置为(compare x y)，则比较运算等价于(eq x y)。通常，在一个特定的机器上，只支持一种风格的比较。但是，合并过程将尝试合并运算，从而产生eq。

不等式比较有两种，有符号和无符号。因此，对于有符号和无符号的大于，有两个不同的表达式代码gt和gtu。对于相同的整数值，这些可以产生不同的结果：例如，1有符号大于−1，但是并不无符号大于，因为−1被作为无符号时，实际为0xffffffff，其大于1。

有符号比较也用于浮点值。浮点比较通过操作数的机器模式来区分。

(eq:m x y) 如果x和y所表示的值相等，则为STORE_FLAG_VALUE，否则为0。

(ne:m x y) 如果x和y所表示的值不相等，则为STORE_FLAG_VALUE，否则为0。

(gt:m x y) 如果x比y大，则为STORE_FLAG_VALUE。如果它们为定点，则按照有符号比较。

(gtu:m x y)
: 类似于gt，不过进行无符号比较，只用于定点数。

(lt:m x y)
(ltu:m x y)
: 类似于gt和gtu，不过测试“小于”。

(ge:m x y)
(geu:m x y)
: 类似于gt和gtu，不过测试“大于或等于”。

(le:m x y)
(leu:m x y)
: 类似于gt和gtu，不过测试“小于或等于”。

(if_then_else cond then else)
: 这不是比较运算，但是被列在这里，因为其总是与比较运算结合使用。确切的说，cond为一个比较表达式。该表达式表示一个根据cond，在then所表示的值和else所表示的值之间的选择，

  在大多数机器上，if_then_else表达式只用于表示条件跳转。

(cond [test1 value1 test2 value2 ...] default)
: 类似于if_then_else，不过更普通。每个test1，test2，...被依次执行。表达式的结果为对应于第一个非零测试的value，或者如果测试都为零，则为default。

这目前在指令模式中不可用，只在insn属性中被支持。See Section 16.19 [Insn属性], page 266.

10.11 位域

有专门的表达式代码来表示位域指令。

(sign_extract:m loc size pos)

这表示了对在loc处包含的或者起始的符号扩展位域的引用（内存或者寄存器引用）。位域为size个位数宽并且在位pos处起始。编译选项BITS_BIG_ENDIAN指明了pos从内存单元的那个端开始。

如果loc在内存中，则它的机器模式必须为一个单个字节的整数机器模式。如果loc在寄存器中，则使用的机器模式是通过insv或者extv指令模式的操作数来指定的（see Section 16.9 [标准名字], page 235）并且通常为一个全字的整数机器模式，这当没有任何指定的时候为缺省的。

pos的机器模式为机器特定的并且总是在insv或者extv指令模式中被指定。

机器模式m与loc所使用的相同，如果它是在寄存器中。

在RTL中，sign_extract不可以作为左值或者是其中的一部分出现。

(zero_extract:m loc size pos)

类似sign_extract，但是指向一个无符号或者零扩展的位域。相同的位序列被抽取，但是它们被填充到一个整字中，并使用零扩展而不是符号扩展。

不像sign_extract，该表达式的类型可以在RTL中为左值；它们可以出现在一个赋值的左边，来表明在一个指定的位域插入一个值。

10.12 向量运算

所有普通的RTL表达式都能够作为向量模式使用；它们被解析为对向量的每个部分进行独立的运算。另外，有一些新的表达式来描述特定的向量运算。

(vec_merge:m vec1 vec2 items)

这描述了两个向量间的合并操作。结果为机器模式为m的向量；它的元素来自vec1或者vec2。那些元素被选择是通过items来描述，其为一个由const_int表示的位掩码；0位指示相应的元素在结果向量中是来自vec2，而1指示其来自vec1。

(vec_select:m vec1 selection)

这描述了选择一个向量的一部分的操作。vec1为源向量，selection为一个parallel其包含了一个const_int，来描述结果向量的子部分，给出了源向量的子部分应该被存放进去。

(vec_concat:m vec1 vec2)

描述了一个向量连接操作。结果为向量vec1和vec2的连接；其长度为两个输出向量的长度之和。

(vec_duplicate:m vec)

该操作将一个小向量转换为一个大一点的，通过复制输入值。输出向量的机器模式必须和输入向量的相同，并且输出部分的编号必须为输入部分的编号的整数倍。

10.13 转换

所有机器模式之间的转换都必须使用显示的转换符来表示。例如，一个表示字节和全字之和的表达式就不能写成(plus:SI (reg:QI 34) (reg:SI 80))，因为plus操作符需要两个具有相同机器模式的操作符。因此，字节长度的操作数被封装在一个转换操作中，如

```
(plus:SI (sign_extend:SI (reg:QI 34)) (reg:SI 80))
```

转换符并不仅仅是一个形式上的占位符，因为可能会有多种方式将给出的最初模式转换为期望的最终模式。转换符指出了如何进行这种操作。

对于所有的转换操作，x不能为VOIDmode，因为这样就无法知道如何进行转换操作。转换必须在编译时进行或者x必须被放入寄存器中。

(sign_extend:m x)
: 表示将x的值符号扩展为机器模式m后的结果。m必须是一个定点模式，并且x是一个比m模式窄的定点值。

(zero_extend:m x)
: 表示将x的值零扩展为机器模式m后的结果。m必须是一个定点模式，并且x是一个比m模式窄的定点值。

(float_extend:m x)
: 表示将x的值扩展为机器模式m后的结果。m必须是一个浮点模式，并且x是一个比m模式窄的浮点值。

(truncate:m x)
: 表示将x的值截短为机器模式m后的结果。m必须是一个定点模式，并且x是一个比m模式宽的定点值。

(ss_truncate:m x)
: 表示将x的值截短为机器模式m后的结果，并且在溢出时作为有符号数处理。m和x的模式都必须是定点模式。

(us_truncate:m x)
: 表示将x的值截短为机器模式m后的结果，并且在溢出时作为无符号数处理。m和x的模式都必须是定点模式。

(float_truncate:m x)
: 表示将x的值截短为机器模式m后的结果。m必须是一个浮定点模式，并且x是一个比m模式宽的浮点值。

(float:m x)
: 表示将定点值x转换为有符号的浮点模式m后的结果。

(unsigned_float:m x)
: 表示将定点值x转换为无符号的浮点模式m后的结果。

(fix:m x)
: 当m是一个浮点模式时，表示将浮点值x（对模式m有效）转换为整形，仍然使用浮点模式m表示，只不过是向零方向进行舍入。

  当m是一个定点模式时，表示将浮点值x转换为有符号的模式m的结果。具体如何舍入没有做出规定。所以，这个操作可能只是被用在编译C代码时的整数值的操作数。

(unsigned_fix:m x)
: 表示将浮点值x转换为无符号的定点模式m。具体如何舍入没有做出规定。

(fract_convert:m x)

表示将定点值转换成定点机器模式m，将有符号整数值x转换成定点机器模式m，将浮点值x转换成定点机器模式m，将定点值x转换成有符号整数机器模式m，或者将浮点值x转换成浮点机器模式的结果。当发生溢出或者下溢，则结果未定义。

(sat_fract:m x)

表示将定点值x转换为浮点模式m，将有符号整数值x转换为定点模式m，或者将浮点值x转换为定点模式m的结果。当发生溢出或者下溢，则结果被饱和为最大值或者最小值。

(unsigned_fract_convert:m x)

表示将定点值x转换为无符号整数模式m，或者将无符号整数值x转换为定点模式m的结果。当发生溢出或者下溢，则结果未定义。

(unsigned_sat_fract:m x)

表示将无符号整数值x转换为定点模式m的结果。当发生溢出或者下溢，则结果被饱和为最大值或者最小值。

10.14 声明

声明表达式代码并不表示算术运算，而是关于它们的操作数状态的断言。

(strict_low_part (subreg:m (reg:n r) 0))

这个表达式代码只用在一种上下文中：作为set表达式的目标操作数。另外，这个表达式的操作数必须是一个non-paradoxical subreg表达式

这里strict_low_part指出寄存器中对于模式n有意义，但对于模式m却无意义的那一部分，是不能被修改的。通常，对于这样的subreg进行赋值，当m小于一个字时，是允许对寄存器的其它部分有未定义的影响。

10.15 副作用表达式

目前为止，所描述的表达式代码都是用来表示一个值，而不是操作。但是机器指令是不会产生值的，而只是通过副作用来改变机器状态。特定的表达式代码被用来表示副作用。

一条指令的主体，总是这些副作用代码之一；上面描述的表示值的代码，只是作为操作数出现在其中。

(set lval x)

表示将x的值存放到由lval表示的地方。lval必须是表示可以用来存放的地方的表达式：reg（或者subreg，strict_low_part或者zero_extract），mem，pc，parallel或者cc0。

如果lval是一个reg，subreg或者mem，其具有一个机器模式；则x必须对这种模式有效。

如果lval是一个subreg的strict_low_part，则由subreg的机器模式所指定的寄存器的那部分被赋予值x，而寄存器的其它部分不变。

如果lval是一个zero_extract，则由zero_extract指定的相关位域（内存或者寄存器相关的），被赋予值x，而其它位域不变。注意sign_extract不能出现在lval中。

如果lval是(cc0)，其没有机器模式，并且x可以为一个compare表达式或者任意模式的值。后者情况表示是一个“test”指令。表达式(set (cc0) (reg:m n))等价于(set (cc0) (compare (reg:m n)))。在编译过程中可以使用前一个表达式来节省空间。

如果lval是一个parallel，其用来表示一个函数通过多个寄存器来返回一个结构体的情况。parallel中的每一个元素是一个expr_list，其第一个操作数是一个reg，并且第二

个操作数是一个const_int，表示相应寄存器的数据在结构体中的偏移量（以字节为单位）。第一个元素也可能为null，用来指示结构体也有一部分是在内存中传递的。

如果lval是(pc)，则为一个跳转指令，并且x只有几种可能。其可能为一个label_ref表达式（无条件跳转）。可能为一个if_then_else（条件跳转），这种情况下，第二个或者第三个操作数必须是(pc)（用于不进行跳转的情况），并且另外两个必须是一个label_ref（用于进行跳转的情况）。x也可以是一个mem或者(plus:SI (pc) y)，其中y可以为一个reg或者mem；这些独特的模式用来表示通过分支表来进行跳转。

如果lval即不是(cc0)也不是(pc)，则lval的模式一定不是VOIDmode，并且x的模式必须对于lval的模式有效。

lval通常通过SET_DEST宏来访问，x通常使用SET_SRC宏。

(return) 在指令模式中作为单独的表达式，表示从当前函数的一个返回，在一些机器上，可以使用一条指令来完成，例如VAXen。在一些机器上，为了从函数中返回，包括多条指令的尾声必须被执行，则返回操作，通过跳转到一个位于尾声之前的标号来完成，并且不使用return表达式代码。

在if_then_else表达式中，表示放在pc中的，返回给调用者的值。

注意，指令模式为(return)的insn，在逻辑上等价于(set (pc) (return))，但是不使用后者的形式。

(call function nargs)

表示一个函数调用。function为一个mem表达式，其地址为被调用的函数的地址。nargs为一个表达式，其可以用于两个目的：在一些机器上，其表示栈参数的字节数目；在其它机器上，其表示参数寄存器的数目。

每个机器具有一个标准的，function必须具有的机器模式。机器描述定义了宏FUNCTION_MODE，来扩展为需要的模式名。在一些机器上，所允许的寻址方式取决于被寻址的机器模式，则该机器模式的用途是来描述，允许什么样的寻址。

(clobber x)

表示一个不可预期的存储或者可能的存储，将不可描述的值存储到x，其必须为一个reg，scratch, parallel 或者 mem表达式。

可以用在字符串指令中，将标准的值存储到特定的硬件寄存器中。不需要去描述被存储的值，只用来告诉编译器寄存器被修改了，以免其尝试在字符串指令中保持数据。

如果x为(mem:BLK (const_int 0))或者(mem:BLK (scratch))，则意味着所有的内存位置必须假设被破坏。如果x为一个parallel，其具有与set表达式中的parallel相同的含义。

注意，机器描述将特定的硬件寄存器归类为“call-clobbered”。所有函数调用指令都被假设为，缺省的，会破坏这些寄存器，所以不需要使用clobber表达式来表示这些。而且，每个函数调用都被假设为潜在的修改任何内存位置，除非函数被声明为const。

如果在parallel中的最后一组表达式为clobber表达式，其参数为reg或者match_scratch（see Section 16.4 [RTL模板], page 203）表达式，则合并阶段可以向构建的insn中增加适当的clobber表达式，当这样可以使得指令模式被匹配。

例如，该特点可以用在，乘法和加法指令不使用MQ寄存器，但具有一个加法累加指令，而且破坏MQ寄存器的机器上。类似的，被合并的指令可能需要临时的寄存器，而成员指令则不需要。

当寄存器的clobber表达式，出现在具有其它副作用的parallel中，如果是硬件寄存器，则寄存器分配者来确保在insn之前和之后，该寄存器都不会被占用。对于伪寄存器的

破坏，寄存器分配者和重载过程，不对clobber分配相同的硬件寄存器，以及输入操作数。你可以破坏一个特定的硬件寄存器，一个伪寄存器，或者一个scratch表达式；在后两种情况下，GCC将会分配一个硬件寄存器，临时使用。

对于需要临时寄存器的指令，应该使用scratch，而不是伪寄存器，因为这将使得合并阶段可以在需要的时候增加clobber。方式为(clobber (match_scratch ...))。如果确实是破坏了一个伪寄存器，则使用没有出现在其它地方的伪寄存器，每次生成一个新的。否则，你可能会使CSE（公共子表达式消除）迷惑。

还有一种在parallel中破坏伪寄存器的用法：当insn的输入操作数也被insn破坏。这种情况下，使用相同的伪寄存器。

(use x) 表示对x值的使用。其表示x中的值在程序的这个点上是被需要的，即使可能不清楚为什么。因此，如果先前的执行的作用只是将一个值存储在x中，则编译器将不会尝试将其删除。x必须为一个reg表达式。

在一些情况下，可能会想到，在parallel中增加一个对寄存器的use，来描述特定寄存器的值将会影响指令的行为。一个假定的例子为，对于一个加法指令模式，其可以根据特定的控制寄存器的值来执行环绕或者饱和加法：

```
(parallel [(set (reg:SI 2) (unspec:SI [(reg:SI 3)
                                       (reg:SI 4)] 0))
          (use (reg:SI 1))])
```

这将不会工作，一些优化器将只查看局部的表达式；很可能如果你有多个具有针对unspec相同输入的insn，它们将被优化掉，即使寄存器1中间有所改变。

这意味着，use只能被用于描述寄存器是活跃的。在增加use语句时，你应该多思考一下，通常，你将会使用unspec来替代。use RTX最常用于描述一个隐式的用于insn的固定寄存器。还可以安全的用于，编译器知道整个指令模式的结果是可变的，这样的指令模式中，例如`movmemm'或者`call'。

在重载阶段，具有use指令模式的insn可以附带一个reg_equal注解。这些use insn将在重载阶段退出之前被删除。

在延迟分支调度阶段，x可以为一个insn。这表示x之前曾经在该位置被定位，它的数据依赖需要被考虑。这些use insn将在延迟分支调度阶段退出之前被删除。

(parallel [x0 x1 ...]) 表示并行执行多个副作用。方括号表示一个向量；parallel的操作数为向量表达式。x0, x1等等为单独的副作用表达式，set, call, return, clobber 或 use。

“并行”意味着，首先所有在单个副作用中使用的值将被计算，然后，所有实际的副作用被执行。例如，

```
(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1)))
          (set (mem:SI (reg:SI 1)) (reg:SI 1))])
```

清楚的说明了，将硬件寄存器1的值与其所寻址的内存中的值进行交换。在(reg:SI 1)作为内存地址出现的两个地方，其都是使用执行insn之前，在寄存器1中的值。

从而，如果使用parallel，并且期望set的值，可以用于下一个set，则是不正确的。例如，人们有时候尝试用这种方式来表示，为零则跳转的指令：

```
(parallel [(set (cc0) (reg:SI 34))
          (set (pc) (if_then_else
                    (eq (cc0) (const_int 0))
                    (label_ref ...)
                    (pc)))])
```

但这是不正确的，因为其说明了跳转条件取决于，该指令之前的条件代码的值，而不是被该指令设置后的新值。

与最后的汇编代码输出一起执行的窥孔优化，可以产生由parallel组成的insn，其元素为需要输出汇编代码的操作数，通常为reg, mem或者常量表达式。这在其它编译阶段，将不是一个好的RTL形式，但是在这里是可以的，因为已经没有其它的优化了。然而，宏NOTICE_UPDATE_CC的定义，如果存在，如果定义了窥孔优化，则需要处理这样的insn。

(cond_exec [cond expr])

表示一个条件执行表达式。只有当cond为非零时，expr才被执行。cond表达式不能具有副作用，但是expr可以。

(sequence [insns ...])

表示一个insn序列。每个出现在向量中的insns，都适合出现在insn链中，所以其必须为insn, jump_insn, call_insn, code_label, barrier 或 note。

在RTL生成过程中，不会在实际的insn中放入sequence RTX。其表示define_expand产生的insn序列，用来传递给emit_insn，从而将它们插入到insn链中。当实际被插入的时候，单独的子insn将被分离出来，sequence将被忽略掉。

当延迟槽调度完成之后，insn和所有位于其延迟槽中的insn被组成一个sequence。需要延迟槽的insn为向量中的第一个insn；后续的insn为将被放在延迟槽中的insn。

INSN_ANNULLED_BRANCH_P用来表示分支insn将会有条件的取消延迟槽中的insn的效果。这种情况下，INSN_FROM_TARGET_P表示insn是来自分支的目标，并且只有当进行分支时，其才被执行；否则，insn只有当不进行分支时才被执行。See Section 16.19.7 [延迟槽], page 272.

这些表达式代码出现在副作用的地方，作为insn的主体，虽然严格的讲，它们并不总是描述副作用：

(asm_input s)

表示文字的汇编代码，通过字符串s来描述。

(unspec [operands ...] index)
(unspec_volatile [operands ...] index)

表示一个机器特定的针对operands的操作。index在多个机器特定的操作之间进行选择。unspec_volatile用于volatile操作，并且可以有陷阱；unspec用于其它操作。

这些代码可以出现在insn的pattern中，parallel中，或者表达式中。

(addr_vec:m [lr0 lr1 ...])

表示跳转地址表。向量元素lr0等等，为label_ref表达式。机器模式m描述了为每个地址给定了多少空间；通常m为Pmode。

(addr_diff_vec:m base [lr0 lr1 ...] min max flags)

表示一个跳转地址表，表示为base的偏移量。向量元素lr0等等，为label_ref表达式，base也是。机器模式m描述了为每个地址偏移给定的空间大小。min和max由分支缩短过程设置，分别存放了一个具有最小地址和最大地址的标号。详情参见rtl.def。

(prefetch:m addr rw locality)

表示对地址为addr的内存进行预取。如果预取的数据将被写，则操作数为rw，否则为0；不支持写预取的目标机，应该将其作为一个普通的预取。操作数locality描述了时间局部性的数量；如果没有，则为0，否则按照时间局部性的递增级别，依次为1，2或者3；不支持局部性暗示的目标机，应该忽略该项。

该insn用于最小化cache-miss的延迟，通过在访问数据之前将其移送到cache中。其应该只用于非故障的数据预取指令。

10.16 地址中嵌入的副作用

有六个特定的副作用表达式代码作为内存地址出现。

(pre_dec:m x)
表示该副作用为，x递减一个标准的数量，并且还表示了递减后的x的值。x必须为一个reg或者mem，但是大多数机器只允许reg。m必须为机器所使用的指针的机器模式。x被递减的数量为，所包含的内存引用的机器模式的长度，以字节为单位。这里有一个关于用法的例子：

```
(mem:DF (pre_dec:SI (reg:SI 39)))
```

这说明将伪寄存器39递减一个DFmode值的长度，并将结果用来对一个DFmode值进行寻址。

(pre_inc:m x)
类似的，用来说明递增x。

(post_dec:m x)
表示与pre_dec相同的副作用，但表示不同的值。这里表示的值为递减之前的x的值。

(post_inc:m x)
类似的，用来说明递增x。

(post_modify:m x y)
表示该副作用为，将x设置为y，并且表示x被修改之前的值。x必须为一个reg或者mem，但是大多数机器只允许reg。m必须为机器所使用的指针的机器模式。

表达式y必须为下列三种形式之一：(plus:m x z)，(minus:m x z)，或者(plus:m x i)，其中z为一个索引寄存器，i为一个常量。

这里为一个有关用法的例子：

```
(mem:SF (post_modify:SI (reg:SI 42) (plus (reg:SI 42)
                                          (reg:SI 48))))
```

这说明在使用了伪寄存器42曾经指向的值之后，将伪寄存器42修改为，加上其伪寄存器48的内容，

(pre_modify:m x expr)
类似的，表示副作用在使用之前开始有效。

这些嵌入的副作用表达式在使用时要小心。指令模式可以不使用它们。在到达编译器的`flow' pass之前，它们可能只出现在用于表示压栈。`flow' pass查找寄存器在一条指令中被递增或递减，并且在之前或者之后被作为地址使用的情况；这些情况然后被转换成使用前增（减）或后增（减）。

如果这些表达式中作为操作数的寄存器，在一个insn中的另一个地址中使用，则会使用寄存器的原始的值。在地址之外使用寄存器是不被允许的，因为这样的insn在不同的机器上行为是不同的，因此会有歧义。

可以被表示成具有嵌入副作用的指令，也可以被表示成使用parallel，包含一个额外的set来描述地址寄存器如何被修改。

10.17 作为表达式的汇编指令

RTX代码asm_operands表示由用户特定的汇编指令所产生的值。其用来表示带有参数的asm语句。一个具有单个输出操作数的asm语句，如下：

```
asm ("foo %1,%2,%0" : "=a" (outputvar) : "g" (x + y), "di" (*z));
```

其通过一个单个的asm_operands RTX来表示，其表示了存储在outputvar中的值：

```
(set rtx-for-outputvar
  (asm_operands "foo %1,%2,%0" "a" 0
        [rtx-for-addition-result rtx-for-*z]
        [(asm_input:m1 "g")
         (asm_input:m2 "di")]))
```

这里，asm_operands RTX的操作数为汇编模板字符串，输出操作数的约束，在指定的输出操作数中的索引编号，一个输入操作数RTX向量，以及一个输出操作数机器模式和约束的向量。机器模式m1为x+y的和的机器模式；m2为*z的机器模式。

当asm语句具有多个输出值时，它的insn具有多个这样的set RTX，并位于一个parallel中。每个set包括了一个asm_operands；所有这些共享相同的汇编模板和向量，但是每个包含了相应的输出操作数的约束。它们也是通过输出操作数索引编号来区分的，即0，1...连续的输出操作数。

10.18 Insns

一个函数的代码的RTL表示是一个被称作insns对象的双向链表。insn只不过是具有特定代码的表达式。有些insn是实际的指令；有些用来表示switch语句的派遣表。有些用来表示要调转的标号或者不同类别的声明信息。

除了本身特定的数据，每个insn必须有一个唯一的id号用来区别当前函数中其它的insn（经过分支延迟调度之后，具有相同id号的一个insn 的拷贝，可能会出现在一个函数中的多个地方，但是这些拷贝总是同样的，并且只是出现在一个sequence中），以及指向前面和后面insn的链表指针。这三个域在每个insn中占有相同的位置，并且独立于insn的表达式代码。它们可以通过XEXP和XINT来访问，不过，有三个特定的宏经常会被使用：

`INSN_UID (i)`

访问insn i的唯一id。

`PREV_INSN (i)`

访问指向i之前的insn的链表指针。如果i是第一个insn，则是一个null指针。

`NEXT_INSN (i)`

访问指向i之后的insn的链表指针。如果i是最后一个insn，则是一个null指针。

链表中的第一个insn可以通过调用get_insns获得；最后一个insn可以通过调用get_last_insn来获得。在由这些insn界定的链中，NEXT_INSN和PREV_INSN指针必须总是相当：如果insn 不是第一个insn，则

```
NEXT_INSN (PREV_INSN (insn)) == insn
```

总是真，并且如果insn不是最后一个insn，则

```
PREV_INSN (NEXT_INSN (insn)) == insn
```

总是真。

在延迟槽调度之后，在链中的一些insn可能为sequence表达式，其包含了一个insn向量。这个向量中除了最后一个insn之外，其它insn的NEXT_INSN的值都是向量中的下一个insn；向量中的最后一个insn的NEXT_INSN的值，等于包含sequence的insn的NEXT_INSN的值。对于PREV_INSN，也有类似的规则。

这意味着上面的恒等式，对于在sequence表达式中的insn不需要成立。特别是，如果insn为sequence中的第一个insn，则NEXT_INSN (PREV_INSN (insn))为包含sequence表达式的insn，同样如果insn为sequence中的最后一个insn，则PREV_INSN (NEXT_INSN (insn))的值也是如此。你可以使用这些表达式来查找包含sequence的insn。

每个insn都具有下列六种表达式代码中的一个：

insn 表达式代码insn用于不进行跳转和函数调用的指令。sequence表达式总是包含在表达式代码为insn的insn中，即使它们中的一个insn是跳转或者函数调用。

表达式代码为insn的insn，除了上面列出的三个必须的域以外，还具有四个额外的域。这四个域在后面的表中有描述。

jump_insn 表达式代码jump_insn用于可能执行跳转（或者，更一般的讲，指令中可能包含了label_ref表达式，并用其来设置pc）的指令。如果有一条从当前函数返回的指令，则其被记录为jump_insn。

jump_insn具有跟insn相同的额外的域，并使用同样的方式来访问，除此之外，还包含了一个域JUMP_LABEL，其当执行完跳转优化后被定义。

对于简单的条件跳转和无条件跳转，该域包含了该insn将（可能有条件的）分支跳转到的code_label。在更复杂的跳转中，JUMP_LABEL记录了insn引用的其中一个标号；其它跳转目标标号作为REG_LABEL_TARGET注解来记录。addr_vec和addr_diff_vec是例外的情况，对此，JUMP_LABEL为NULL_RTX，而只有扫描整个insn体干才能找到标号。

返回指令insn作为跳转看待，但由于它们并不引用任何标号，所以它们的JUMP_LABEL为NULL_RTX。

call_insn 表达式代码call_insn用于可能执行函数调用的指令。区分这些指令是很重要的，因为它们意味着特定的寄存器和内存位置可以被不可预知的方式改变。

call_insn具有与insn相同的额外的域，并使用相同的方式访问，除此之外，还包含一个域CALL_INSN_FUNCTION_USAGE，其包含了一个列表（expr_list表达式链），包含了use和clobber表达式，表示了被调用函数使用和破坏的硬件寄存器和MEM。

一个MEM通常指向一个栈槽，参数在其中按照引用方式（see Section 17.10.7 [寄存器参数], page 325）传递给libcall。如果参数是caller-copied（see Section 17.10.7 [寄存器参数], page 325），则栈槽会在CLOBBER和USE中被提到；如果是callee-copied，则只会出现USE，并且MEM可能指向不是栈槽的地址。

在列表中，被CLOBBER的寄存器，增加了在CALL_USED_REGISTERS中描述的寄存器（see Section 17.7.1 [寄存器基础], page 303）。

code_label code_label insn表示一个跳转insn可以跳转到的标号。除了三个标准的域以为，其还包含两个特定的域。CODE_LABEL_NUMBER用于存放label number，在编译过程中，唯一标识该标号。最终，标号在汇编输出中作为汇编标号来表示，通常的形式为`Ln'，其中n为标号编号。

当code_label出现在RTL表达式中，其通常出现在label_ref中，其表示了标号的地址，为一个编号。

除了作为code_label以外，标号还可以作为类型为NOTE_INSN_DELETED_LABEL的note来表示。

域LABEL_NUSES只当完成跳转优化过程后才被定义。其包含了在当前函数中，该标号被引用的次数。

域LABEL_KIND用来区分四种不同类型的标号：LABEL_NORMAL，LABEL_STATIC_ENTRY，LABEL_GLOBAL_ENTRY和LABEL_WEAK_ENTRY。唯一不具有类型LABEL_NORMAL的标号，为当

前函数的alternate entry points。这些可以为static（只在当前转换单元中可见），global（对所有的转换单元可见）或者weak（全局的，但是可以被另一个具有相同名字的符号覆盖）。

编译器大多将所有四种标号同等对待。有些地方需要知道标号是否为候选入口点；为此，提供了宏LABEL_ALT_ENTRY_P。其等价于测试是否`LABEL_KIND (label) == LABEL_NORMAL'。除了前端创建static，global和weak alternate entry points的代码以外，其它唯一关心它们的区别的地方是`final.c'文件中的函数output_alternate_entry_point。

使用宏SET_LABEL_KIND来设置标号的种类。

barrier 栅栏被放在指令流中，控制无法经过的地方。它们被放在无条件跳转指令的后面，表示跳转是无条件的，以及对volatile函数的调用之后，表示不会返回（例如，exit）。除了三个标准的域以外，不包含其它信息。

note note insns用于表示额外的调试和说明信息。它们包含两个非标准的域，一个使用宏NOTE_LINE_NUMBER访问的整数，以及一个使用NOTE_SOURCE_FILE访问的字符串。

如果NOTE_LINE_NUMBER是正的，则注解表示源文件行号，并且NOTE_SOURCE_FILE为源文件名。这些注解控制在汇编输出中的生成行号数据。

否则，NOTE_LINE_NUMBER不是一个行号，而是一个具有下列值之一的代码（并且NOTE_SOURCE_FILE必须包含一个空指针）：

NOTE_INSN_DELETED
: 这样的注解被完全忽略掉。编译器的一些过程会通过将insn修改成这种类型的注解，来删除insn。

NOTE_INSN_DELETED_LABEL
: 标记了曾经为code_label，但现在只用于获得其地址，并且没有代码会跳转到这里。

NOTE_INSN_BLOCK_BEG
NOTE_INSN_BLOCK_END
: 这些类型的注解表示处于变量名作用域的起始和结束。它们控制调试信息的输出。

NOTE_INSN_EH_REGION_BEG
NOTE_INSN_EH_REGION_END
: 这些类型的注解表示处于异常处理作用域的起始和结束。NOTE_BLOCK_NUMBER标识了哪一个类型为NOTE_INSN_DELETED_LABEL的CODE_LABEL或note与给定的区域相关联。

NOTE_INSN_LOOP_BEG
NOTE_INSN_LOOP_END
: 这些类型的注解表示处于while或者for循环的起始和结束。它们使得循环优化可以快速的发现循环。

NOTE_INSN_LOOP_CONT
: 出现在循环中continue语句跳转的地方。

NOTE_INSN_LOOP_VTOP
: 该注解表示循环中退出测试（exit test）起始的地方，并且退出测试在循环中被复制。当考虑循环不变量时，该位置为循环的另一个虚拟起始点。

NOTE_INSN_FUNCTION_BEG
出现在函数序言之后，函数体的起始处。

在调试转储中，这些代码被符号化的打印。

insn的机器模式通常为VOIDmode，但有些阶段出于不同的目的而使用其它机器模式。

公共子表达式消除过程将一个insn的机器模式设为QImode，当其为已经被处理过的块中的第一个insn时。

第二次Haifa调度过程中，对于可以多发射的目标机，当insn被认为是一个发射组合中的起始指令时，将其机器模式设为TImode。也就是说，该指令不能和之前的指令同时发射。这可以在后面的过程中用到，特别是机器特定的reorg。

下面的表中列出了insn, jump_insn和call_insn的其它域：

PATTERN (i) 一个表达式，为该insn执行的副作用。必须为下列代码中的一个：set, call, use, clobber, return, asm_input, asm_output, addr_vec, addr_diff_vec, trap_if, unspec, unspec_volatile, parallel, cond_exec或sequence。如果其为parallel，则parallel中的每个元素必须是这些代码中的一个，并且，parallel表达式不能被嵌套，addr_vec和addr_diff_vec不允许在parallel表达式中。

INSN_CODE (i)
一个整数，说明机器描述中的哪一个指令模式匹配该insn，或者，如果还没有进行匹配，则为−1。

对于指令模式由单个use, clobber, asm_input, addr_vec 或 addr_diff_vec表达式组成的insn，则不会进行这样的匹配，并且该域保持为−1。

对于来自asm语句的insn，也不会进行指令模式匹配。这些至少包含了一个asm_operands表达式。函数asm_noperands为这样的insn返回一个非负的值。

在调试输出中，该域被打印成一个数字，紧随一个符号表示，用来定位在`md'中的指令模式，数字表示相对命名指令模式的正的或者负的偏移量。

LOG_LINKS (i)
一个列表（insn_list表达式链），给出了基本块中指令之间的依赖信息。相关联的insn之间不会有跳转或者标号。这些只被用于指令调度和组合。这是一个不被推荐的数据结构。现在推荐使用def-use和use-def链。

REG_NOTES (i)
一个列表（expr_list和insn_list表达式链），给出了insn的其它信息。通常为从属于该insn使用的寄存器的信息。

insn的LOG_LINKS域为insn_list表达式链。每一个都具有两个操作数：第一个为insn，第二个为另一个insn_list表达式（链中的下一个）。链中的最后一个insn_list的第二个操作数为空指针。对于表达式链，重要的是有哪些insn（insn_list表达式的第一个操作数）。它们的顺序并不重要。

该列表最初由流分析过程建立；在此之前还只是空指针。流分析只将那些可以用于指令合并的数据依赖，加入到列表中。

insn的REG_NOTES域是一个类似于LOG_LINKS域的链，不过除了insn_list表达式，其还包含expr_list表达式。有多种寄存器注解，其通过机器模式区分。注解的第一个操作数op的含义依赖注解的种类。

宏REG_NOTE_KIND (x)返回寄存器注解的种类。宏PUT_REG_NOTE_KIND (x, newkind)将x的寄存器注解类型设置为newkind。

寄存器注解有三种类别：可以用来说明insn的输入，可以用来说明insn的输出，或者可以用来创建两个insn之间的连接。还有一个值集，只用于LOG_LINKS中。

这些注解用来说明insn的输入：

REG_DEAD op中的值在该insn中死掉；也就是说，紧接这个insn之后，修改该值将不会影响程序将来的行为。

这并不是说从该insn之后，寄存器op就没有有用的值了。而是说，后续的指令不会用到op的内容。

REG_UNUSED 被该insn设置的寄存器op，将不会在后续的insn中使用。这与REG_DEAD注解不同，后者表示输入中的值将不会被后续insn使用。这两个注解是不相关的；可能会都出现在同一个寄存器中。

REG_INC 寄存器op由于insn中嵌入的副作用，而被递增（或递减）。这意味着其出现在post_inc, pre_inc, post_dec或pre_dec表达式中。

REG_NONNEG 寄存器op在到达该insn的时候，被已知为具有一个非负的值。对于递减并分支跳转，直到为零的指令，例如m68k dbra，可以用来进行匹配。

REG_NONNEG注解，只有当机器描述具有`decrement_and_branch_until_zero'指令模式的时候，才被加到insn中。

REG_LABEL_OPERAND

该insn使用op，一个类型为NOTE_INSN_DELETED_LABEL的code_label或者note，但是不为jump_insn。或者，其为一个将操作数作为普通操作数的jump_insn。标号最终也可以为跳转目标，但这是在后续insn的间接跳转中。该注解使得跳转优化知道op实际上被使用了，从而流优化可以创建一个精确的流图。

REG_LABEL_TARGET

该insn为一个jump_insn，但不是addr_vec和addr_diff_vec。其使用op，一个code_label，作为直接或间接跳转的目标。其用途与REG_LABEL_OPERAND类似。该注解只存在于当insn具有多个目标的时候；insn中的最后一个标号（在最高编号的insn域中），放到JUMP_LABEL域中，并且没有REG_LABEL_TARGET。See Section 10.18 [Insns], page 139.

REG_CROSSING_JUMP

该insn为一个分支指令（无条件跳转或者间接跳转），其穿越了热代码段和冷代码段，并可能潜在的位于可执行程序中非常远的部分。该注解用来指示其它优化，表示该分支指令不应该被折叠为简单的分支结构。其用于当优化将基本块分成热代码段和冷代码段的时候。

REG_SETJMP 附加在每个针对setjmp或者相关的函数的CALL_INSN上。

下列注解描述了有关insn的输出的属性：

REG_EQUIV

REG_EQUAL 该注解只用在只设置一个寄存器的insn上，用来表示那个寄存器在运行时等价于op；该等值的作用域根据两种类型的注解而有所不同。insn显式的复制进寄存器的值可能看起来与op不同，但它们将在运行时相等。如果单个set的输出为一个strict_low_part表达式，则注解是用于subreg表达式SUBREG_REG所包含的寄存器。

对于REG_EQUIV，在整个函数中，寄存器都等价于op，并且可以在其所有出现的地方被op有效替换。（有效，这里是指程序的数据流；简单的替换可能会使得某些insn无效

。）例如，当一个常量被加载到一个寄存器中，并且寄存器不再被赋予任何其它值，则会使用这种注解。

当在函数入口处，一个参数被复制到一个伪寄存器中时，这种的注解会用来记录该寄存器等价于传递参数的栈槽。虽然，这种情况下，寄存器可能被其它的insn设置，其也可以在整个函数中被栈槽来替换。

REG_EQUIV注解还用于，在函数入口处，将一个寄存器参数复制到一个伪寄存器中的指令，如果存在一个参数本来应该被存放的栈槽。虽然其它insn可以设置该伪寄存器，但编译器还是可以在整个函数中，使用栈槽来替换伪寄存器，假设编译器可以确保栈槽被适当的初始化。这被用于调用约定为寄存器参数分配栈空间的机器上。参见Section 17.10.6 [栈参数], page 324中的REG_PARM_STACK_SPACE。

对于REG_EQUAL的情况，被该insn设置的寄存器，将在运行时，在该insn的结尾处，但不必要是函数的其它地方，等价与op。这种情况下，op通常为一个算术表达式。例如，当一个库调用的insn序列，被用在一个算术运算上，则该类的注解将被附加在产生或者复制最终值的insn上。

这两个注解在编译器过程中，按照不同的方法来使用。REG_EQUAL用于寄存器分配之前的过程中（例如公共子表达式消除和循环优化），来告诉它们如何考虑那个值。REG_EQUIV注解用于寄存器分配，来表示存在一个可用的替换表达式（为栈上一个参数位置的常量或者mem表达式），其可以用在没有足够寄存器的地方。

除了为参数提供地方的栈以外，其它所有等值最初都是通过附加一个REG_EQUAL注解来表示。在寄存器分配的早期阶段，如果op是一个常量并且insn只表示对其目的寄存器进行设置，则REG_EQUAL被改变成REG_EQUIV注解。

因此，寄存器分配之前的编译过程，只需要检查REG_EQUAL注解，而之后的编译过程只需要检查REG_EQUIV注解。

这些注解描述了insn之间的联系。它们成对的出现：一个insn具有一对注解，其中之一用来指向第二个insn，并且第二个insn也由一个反过来指向第一个insn的注解。

REG_CC_SETTER
REG_CC_USER

在使用cc0的机器上，设置和使用cc0的insns是相邻的。然而，当做完分支延迟槽填充之后，就不一定是这样的了。这种情况下，REG_CC_USER注解将被放在设置cc0的insn上，来指向使用cc0的insn，并且REG_CC_SETTER注解将被放在使用cc0的insn上，来指向设置cc0的insn。

这些值只用在LOG_LINKS域，用来表示每个链接表示的依赖类型。表示一个数据依赖（写后读依赖）的链接，不使用任何代码，它们只是简单的具有VOIDmode模式，并在打印输出中没有任何描述文本。

REG_DEP_TRUE

这表示一个真依赖（写后读依赖）。

REG_DEP_OUTPUT

这表示一个输出依赖（写后写依赖）。

REG_DEP_ANTI

这表示一个反依赖（读后写依赖）。

这些注解描述了从gcov profile数据中搜集的信息。它们作为expr_list存储在insn的REG_NOTES域中。

REG_BR_PROB
: 用于指定分支跳转率，根据profile数据。值位于0和REG_BR_PROB_BASE之间；较大的值表示该分支更可能会被执行。

REG_BR_PRED
: 这些注解在JUMP insn中，并出现在延迟分支调度之后。它们表示JUMP的方向和可能性。格式为ATTR_FLAG_*值的掩码。

REG_FRAME_RELATED_EXPR
: 用在RTX_FRAME_RELATED_P insn上，其附加的表达式被用在实际的insn模式上。这用于指令模式过于复杂或者产生误解的情况。

为方便起见，在insn_list或者expr_list中的机器模式，在调试转储中使用这些符号化的代码来打印。

表达式代码insn_list和expr_list之间的唯一区别是，insn_list的第一个操作数被假设为一个insn，并在调试转储中作为insn的唯一id来打印；而expr_list的第一个操作数作为表达式，按照普通的方式来打印。

10.19 函数调用insns的RTL表示

调用子程序的Insn具有RTL表达式代码call_insn。这些insn必须满足特别的规则，并且它们的主体必须使用特定的RTL表达式代码call。

call表达式有两个操作数，如下：

```
(call (mem:fm addr) nbytes)
```

这里nbytes操作数表示传递给子程序的参数的字节数，fm是一个机器模式（其必须与在机器描述中定义的FUNCTION_MODE相等），addr表示子程序的地址。

对于子程序没有返回值的，上面所示的call表达式是insn的整个主体，除了insn可能还会包含use或clobber表达式。

对于子程序返回不是BLKmode模式的值的，值通过硬件寄存器返回。如果该寄存器号为r，则call insn的主体看起来是这样的：

```
(set (reg:m r)
  (call (mem:fm addr) nbytes))
```

该RTL表达式很清楚的说明了（对于优化阶段），在该insn中有一个适当的寄存器用来接受一个有用的值。

当子程序返回BLKmode值时，将会通过传递给子程序用来存储返回值的地址来处理。因次，call insn本身不返回任何值，具有和没有返回值一样的RTL。

在一些机器上，调用指令本身会破坏一些寄存器，例如包含了返回地址。这些机器上的call_insn应该有一个parallel主体，包含了call表达式和clobber表达式，用来指示哪些寄存器会被破坏。类似的，如果调用指令需要栈指针之外的一些寄存器，并且没有在其RTL中显示提到的，则应该用use子表达式来指出。

被调用的函数被假设为会修改列在配置宏CALL_USED_REGISTERS（see Section 17.7.1 [寄存器基础], page 303）中的所有寄存器，并且除了const函数和库函数调用外，被假设为会修改所有的内存。

直接在call_insn之前的只是包含了use表达式的insn，用来指示哪些寄存器用来存放函数的输入。类似的，如果不在CALL_USED_REGISTERS中那些寄存器会被所调用的函数破坏，紧跟在call之后的包含了单独的clobber的insn，用来指出这些寄存器。

10.20 结构共享假设

编译器假设某些类型的RTL表达式是唯一的；不会存在两个不同的对象表示相同的值。对于其它情况，有相反的假设：在被包含的结构体中，不会在多个地方出现某一类型的RTL表达式对象。

这些假设针对于一个单独的函数；除了描述全局变量和外部函数的RTL对象，一些标准对象，例如小整形常数以外，没有其它RTL对象可以在两个函数中共用。

- 每一个伪寄存器只有一个单独的reg对象来表示，因此也只有一种机器模式。
- 对于任何标号，只有一个symbol_ref对象关联。
- 所有具有相同值的const_int表达式被共享。
- T只有一个pc表达式。
- 只有一个cc0表达式。
- 对于每一种浮点模式，只有一个const_double表达式其值为0。同样对于值1，2。
- 对于每一种向量模式，只有一个const_vector表达式其值为0。其为整数或者双精度常量向量。
- 在RTL结构体中不会在多个地方出现label_ref或scratch；换句话说，对函数中所有insn进行树遍历时，可以认为每次遇到的label_ref或者scratch都与在其它地方遇到的不同。
- 对于每个静态变量或者栈槽，通常只创建一个mem对象，所以这些对象在它们出现的所有地方被共享。然而，有时会为这些变量创建单独的，但是相等的对象。
- 当一个单独的asm语句具有多个输出操作数时，会为每一个输出数创建一个不同的asm_operands表达式。然后，这些表达式都共享包含着输入操作数序列的向量。这是为了之后用于测试两个asm_operands表达式是否来自同一语句，所以，所有的优化当进行复制整个向量时，必须仔细保持共享。
- 除了上面描述的以外，在RTL结构体中没有其它RTL对象会出现多次。编译器的许多遍扫描，都是依赖于这样的假设，即它们能在一个地方修改RTL对象，并且不会对其它insn产生不需要的副作用。
- 在最初的RTL生成过程中，可以随意使用共享结构。当一个函数的所有RTL都被生成之后，所有的共享结构体都被`emit-rtl.c'中的unshare_all_rtl进行复制，之后，将保证上面的规则会被遵循。
- 在合并阶段，共享结构体可以在insn中临时存在。但是，在insn的合并完成之前，共享结构会被复制，通过调用unshare_all_rtl的子程序copy_rtx_if_shared。

10.21 读取RTL

若要从文件中读取RTL对象，可以调用read_rtx。 它接受一个参数，stdio标准输入输出流，并且返回一个RTL对象。 该函数在`read-rtl.c'中定义。 它只在通过机器描述来生成编译器后端的各种程序中使用，编译器本身并没有用到。

人们经常想到使用以文本方式存储在文件中的RTL来作为语言前端和GCC主干之间的接口。 这种想法其实是不可行的。

GCC一度被设计为只是在内部使用RTL。对于给定的程序， RTL正确与否与特定的目标机器非常有关系。而且RTL并不包含程序的所有信息。

对于GCC和一个新语言前端接口的恰当方式， 是使用在文件`tree.h'和`tree.def'中描述的 “tree ” 数据结构。 关于该结构的文档（参见 Chapter 9 [Trees], page 79）不是很全面。

11 GENERIC

GENERIC的目的是简单的提供一个使用tree来表示整个函数的语言无关的方式。为此，之前需要为后端增加一些新的tree代码，但是大多数东西都已经存在。如果你可以使用gcc/tree.def中的代码来表示，则其即为GENERIC。

早期，有一个很大的争论，关于如何在tree IL级别来考虑语句。在GENERIC中，一条语句被定义为任意的表达式，其值，如果存在，被忽略。语句总是设置TREE_SIDE_EFFECTS，不过一个非语句的表达式也可以具有副作用。例如，CALL_EXPR。

对于一些局部优化，可以在函数的GENERIC形式上进行工作；的确，被改写的tree内联在GENERIC上可以很好的工作，不过目前编译器是在下降到GIMPLE（一个严格的形式，在下节有描述）之后，才执行内联的。确实，目前前端是在移交给tree_rest_of_compilation之前，来执行该下降，不过这看起来不太优雅。

如果有必要，前端可以在其GENERIC表示中使用一些语言相关的tree代码，只要其提供一个钩子，将它们转换成GIMPLE，并且不要期望它们可以与任何（假象的）运行在转换成GIMPLE之前的优化器一起工作。当解析C和C++时使用的中间表示，看起来非常像GENERIC，不过C和C++ gimplifier钩子会将其作为输入，并输出GIMPLE。

11.1 语句

大多数GIMPLE语句为赋值语句，由GIMPLE_ASSIGN来表示。没有其它C表达式可以出现在语句级别；对一个volatile对象的引用被转换成一个GIMPLE_ASSIGN。

还有几个复杂语句的变体。

11.1.1 块

块作用域和它们声明的变量，在GENERIC中使用BIND_EXPR代码来表示。这在之前的GCC版本中被主要表示为C语句表达式扩展。

块中的变量按照声明的顺序被搜集到BIND_EXPR_VARS中。任何运行时的初始化被从DECL_INITIAL中移出，并移送到控制块中的一条语句。当从C或者C++进行gimplifying的时候，该初始化用来替换DECL_STMT。

可变长度的数组（VLA）使得该处理变得复杂，因为它们的大小经常是一个块中早前被初始化的变量。为了进行处理，我们目前将块在那个点进行拆分，将VLA移送到一个新的，内部的BIND_EXPR。该策略在将来可能会改变。

C++程序通常包含比源代码中语法块更多的BIND_EXPR，因为多个C++构造函数具有隐式的与它们相关联的作用域。另一方面，虽然C++前端使用伪作用域来处理析构函数对对象的清除，这些并不被转换成GIMPLE形式；在相同级别上的多个声明使用相同的BIND_EXPR。

11.1.2 语句序列

同一嵌套级别的多个语句被搜集到一个STATEMENT_LIST中。语句列表使用`tree-iterator.h'中的接口来进行修改和遍历。

11.1.3 空语句

没有作用效果的语句会尽可能的被丢弃。但是如果它们嵌套在另一个结构中，并且出于某种原因该结构不能被丢弃，则使用空语句来替换，通过build_empty_stmt来生成。起初，所有的空语句是共享的，但是这在实际中产生了许多麻烦。

空语句被表示为(void)0。

11.1.4 跳转

其它的跳转由GOTO_EXPR或者RETURN_EXPR表示。

GOTO_EXPR的操作数必须为一个标号或者一个包含跳转地址的变量。

RETURN_EXPR的操作数为NULL_TREE，RESULT_DECL，或者MODIFY_EXPR，其用来设置返回值。将MODIFY_EXPR移送到一个单独的语句会好些，不过expand_return中的特定的return语义使得有些困难。这在将来可能会发生，可能会通过将大部分逻辑移送到expand_assignment中。

11.1.5 清除

对于C++局部对象的析构，以及类似的动态清除操作在GIMPLE中通过一个TRY_FINALLY_EXPR来表示。TRY_FINALLY_EXPR有两个操作数，均为要执行的语句序列。第一个序列会被执行。当其执行完毕时，第二个序列会被执行。

第一个序列可以按照下列方式来执行完毕：

1. 执行了序列中的最后一条语句，并结束。
2. 执行一个goto语句(GOTO_EXPR)，跳到序列之外的一个普通标号。 t
3. 执行一个return语句(RETURN_EXPR)。
4. 抛出一个异常。这在当前没有用GIMPLE显式的表示。

如果第一个序列通过调用setjmp或exit，或者其它不返回的函数，来执行完毕，则第二个序列不会被执行。如果第一个序列通过一个非局部goto或者一个计算goto（总的来说，编译器不知道这样一个goto语句是否会退出第一个序列，所以我们假设其没有退出）来执行完毕，则第二个序列也不会被执行。

第二个序列被执行完之后，如果其正常的执行到结尾，并结束，则只要第一个序列被继续执行，其也会被继续。

TRY_FINALLY_EXPR使流图变得复杂，因为清除工作需要在流出控制块的每条边上都出现；这就减少了将代码跨越这些边进行移动的自由。因此，运行于大多数优化过程之前的EH下降过程，通过显式的增加对每个边的清除操作，来消除这些表达式。再次抛出异常使用RESX_EXPR来表示。

12 GIMPLE

GIMPLE为一个三地址表示，通过将GENERIC表达式分解成不超过3个操作数（有些情况例外，比如函数调用）的元组。GIMPLE在很大程度上受McGill大学的McCAT编译器项目中使用的SIMPLE IL的影响。我们也做了一些改变，例如SIMPLE不支持（GIMPLE支持的）goto。

临时对象被引入，用来存放计算复杂表达式所需要的中间值。另外，GENERIC中所有的控制结构被下降为条件跳转，词法作用域被移除，异常区域被转换成一个异常区域tree。

将GENERIC转换成GIMPLE的编译器过程，被称作`gimplifier'。`gimplifier'按递归的方式进行工作，从原始的GENERIC表达式生成GIMPLE元组。

早期用于GIMPLE表示的实现策略为，使用与前端表示解析树相同的内部数据结构。这会简化实现，因为我们可以利用现存的功能和接口。然而，与抽象语法树（AST）相比，GIMPLE是一个更加严格的表示，因此其不需要tree数据结构所提供的完整复杂的结构。

函数的GENERIC表示被存放在所关联的FUNCTION_DECL tree结点的DECL_SAVED_TREE域。其通过调用gimplify_function_tree来转换成GIMPLE。

如果前端想在tree表示中包含语言特定的tree代码，并提供给后端，则其必须提供一个LANG_HOOKS_GIMPLIFY_EXPR的定义，其知道如果将前端的tree转换成GIMPLE。通常这样的钩子会涉

及许多相同的代码，用来将前端tree扩展成RTL。该函数可以返回被完全下降的GIMPLE，或者可以返回GENERIC tree并让主gimplifier将它们下降；这通常会更简单些。没有被完全下降的GIMPLE被称为``High GIMPLE''，由pass_lower_cf过程之前的IL组成。High GIMPLE包含一些容器语句，例如词法作用域（由GIMPLE_BIND来表示），以及嵌套表达式（例如，GIMPLE_TRY）。而``Low GIMPLE''将所有隐式的控制跳转或者异常表达式都直接暴露成IL和EH区域tree。

C和C++前端目前直接从前端tree转换成GIMPLE，并将其交给后端，而不是首先转换成GENERIC。它们的gimplifier钩子知道所有的_STMT结点，以及如何将它们转成GENERIC形式。在genericization过程中有一些工作，应该首先被运行，但是STMT_EXPR的存在意味着，为了将所有的C语句转换成GENERIC，则需要遍历整个tree，所以一起下降会更简单些。如果有人写了一个优化过程，其在更高级别的tree上会工作的更好，则这在将来可能会有改变，但是目前所有的优化都是在GIMPLE上进行的。

你可以使用选项`-fdump-tree-gimple'来转储一个类C的GIMPLE表示形式。

12.1 元组表示

GIMPLE指令为可变大小的元组，并由两部分组成：一个描述指令和位置的头，一个具有所有操作数的可变长度的身体。元组被组织成一个层次结构，并有3个主要类别。

12.1.1 gimple_statement_base (gsbase)

这是层次结构的根，其存放了大多GIMPLE语句所需要的基本信息。有一些域并不与所有的GIMPLE语句相关，但是被挪到基础结构中是为了利用其它域剩下的空位（从而使得结构体更加紧凑）。结构体在64位主机上占用4个字（32个字节）：

Field	Size (bits)
code	8
subcode	16
no_warning	1
visited	1
nontemporal_move	1
plf	2
modified	1
has_volatile_ops	1
references_memory_p	1
uid	32
location	32
num_ops	32
bb	64
block	63
Total size	32 bytes

- code GIMPLE指令的主要标识
- subcode 用来区分相同基本指令的不同变体，或者提供使用于给定代码的标记。subcode标记域具有不同的用法，并取决于指令的代码，但是其主要是用来区分相同家族的指令。该域最突出的用法是在赋值中，其子代码指出了在赋值的右手边所进行的操作。例如，a = b + c被编码为GIMPLE_ASSIGN <PLUS_EXPR, a, b, c>。
- no_warning 位标记，用来指出是否在该语句上已经产生了一个警告。
- visited 通用目的的“访问”标记。由每个编译过程根据需要来设置和清除。

- nontemporal_move 位标记，用在赋值中，用来表示非临时的移动。虽然该位标记只用于赋值，但其被放到这里是为了利用先前域所剩下的空位。
- plf 编译过程局部标记。该2个位的掩码可以由任何编译过程用作通用的标记。编译过程负责相应的清除和设置这两个标记。
- modified 位标记，用来指出语句是否被修改。主要由操作数扫描器来使用，用来确定什么时候重新扫描一条语句的操作数。
- has_volatile_ops 位标记，用来指出语句是否包含被标记为volatile的操作数。
- references_memory_p 位标记，用来指出语句是否包含内存引用（即，其操作数为全局变量，或者指针解引用，或者任何必须在内存中的）。
- uid 为无符号整数，由想要为每条语句分配ID的编译过程使用。这些ID必须由每个编译过程来分配和使用。
- location 为一个location_t标识符，用来指定该语句的源代码位置。其从前端继承下来。
- num_ops 该语句具有的操作数个数。这描述了元组中嵌套的操作数向量的大小。只在一些元组中使用，但其声明在基础元组中是为了利用先前语所剩下的32位空位。
- bb 包含该语句的基本块。
- block 包含该语句的词法块。还用于调试信息的生成。

12.1.2 gimple_statement_with_ops

该元组实际分成两部分：gimple_statement_with_ops_base 和 gimple_statement_with_ops。这是为了适应操作数向量的分配方法。操作数向量被定义为有1个元素的数组。所以，要分配动态数目的操作数，内存分配器(gimple_alloc)只是简单的分配足够的内存来存放结构体本身，以及在结构体尾部加上N－1个操作数。例如，要为有3个操作数的元组分配空间，gimple_alloc预留了sizeof (struct gimple_statement_with_ops) + 2 * sizeof (tree)个字节。

另一方面，该元组中的一些域需要与gimple_statement_with_memory_ops元组共享。所以，这些公共域被放在gimple_statement_with_ops_base中，然后由其它两个元组来继承。

gsbase	256
addresses_taken	64
def_ops	64
use_ops	64
op	num_ops * 64
Total size	56 + 8 * num_ops bytes

- gsbase 继承自struct gimple_statement_base。
- addresses_taken 位图，存放了所有VAR_DECL的UID，该语句使用了这些VAR_DECL的地址。例如，形式为p = &b的语句将在该集合中具有符号b的UID。
- def_ops 指针数组，指向操作数数组，指出该包含语句写入的变量的所有插槽。该数组还用于立即使用链。注意，是可以不依赖该数组的，但是这种实现会很具有入侵性。
- use_ops 类似于def_ops，不过是针对语句读取的变量。
- op 具有num_ops插槽的tree数组。

12.1.3 gimple_statement_with_memory_ops

该元组本质上等同于gimple_statement_with_ops，除了其包含4个额外的域，来存放与内存存储和加载相关的向量。类似于先前的情况，结构体被分成两部分，用来容纳操作数向量(gimple_statement_with_memory_ops_base和gimple_statement_with_memory_ops)。

Field	Size (bits)
gsbase	256
addresses_taken	64
def_ops	64
use_ops	64
vdef_ops	64
vuse_ops	64
stores	64
loads	64
op	num_ops * 64
Total size	88 + 8 * num_ops bytes

- vdef_ops 类似于def_ops，不过用于VDEF操作符。这是该语句写入的内存符号的一个实体。这用于维护内存SSA use-def和def-def链。
- vuse_ops 类似于use_ops，不过用于VUSE操作数。这是该语句加载的内存符号的一个实体。这用于维护内存SSA use-def链。
- stores 位集合，该语句写入的符号的所有UID。这与vdef_ops不同之处是，所有被影响的符号都在该集合中被提到。如果开启了内存划分，则vdef_ops向量将指向内存划分。而且，该集合中不存放SSA信息。
- loads 类似于stores，不过用于内存加载。（注意，这里有一些冗余，应该可以通过移除这些集合来减少内存使用）。

所有其它元组按照这三个基本元组来定义。每个元组会增加一些域。gimple类型被定义成所有这些结构体的联合体（为了清晰，省略掉了GTY标记）：

```
union gimple_statement_d
{
 struct gimple_statement_base gsbase;
 struct gimple_statement_with_ops gsops;
 struct gimple_statement_with_memory_ops gsmem;
 struct gimple_statement_omp omp;
 struct gimple_statement_bind gimple_bind;
 struct gimple_statement_catch gimple_catch;
 struct gimple_statement_eh_filter gimple_eh_filter;
 struct gimple_statement_phi gimple_phi;
 struct gimple_statement_resx gimple_resx;
 struct gimple_statement_try gimple_try;
 struct gimple_statement_wce gimple_wce;
 struct gimple_statement_asm gimple_asm;
 struct gimple_statement_omp_critical gimple_omp_critical;
 struct gimple_statement_omp_for gimple_omp_for;
 struct gimple_statement_omp_parallel gimple_omp_parallel;
 struct gimple_statement_omp_task gimple_omp_task;
 struct gimple_statement_omp_sections gimple_omp_sections;
 struct gimple_statement_omp_single gimple_omp_single;
 struct gimple_statement_omp_continue gimple_omp_continue;
 struct gimple_statement_omp_atomic_load gimple_omp_atomic_load;
 struct gimple_statement_omp_atomic_store gimple_omp_atomic_store;
};
```

12.2 GIMPLE指令集

下面的表格简短地描述了GIMPLE指令集。

指令	高层GIMPLE	低层GIMPLE

GIMPLE_ASM	x	x
GIMPLE_ASSIGN	x	x
GIMPLE_BIND	x	
GIMPLE_CALL	x	x
GIMPLE_CATCH	x	
GIMPLE_CHANGE_DYNAMIC_TYPE	x	x
GIMPLE_COND	x	x
GIMPLE_EH_FILTER	x	
GIMPLE_GOTO	x	x
GIMPLE_LABEL	x	x
GIMPLE_NOP	x	x
GIMPLE_OMP_ATOMIC_LOAD	x	x
GIMPLE_OMP_ATOMIC_STORE	x	x
GIMPLE_OMP_CONTINUE	x	x
GIMPLE_OMP_CRITICAL	x	x
GIMPLE_OMP_FOR	x	x
GIMPLE_OMP_MASTER	x	x
GIMPLE_OMP_ORDERED	x	x
GIMPLE_OMP_PARALLEL	x	x
GIMPLE_OMP_RETURN	x	x
GIMPLE_OMP_SECTION	x	x
GIMPLE_OMP_SECTIONS	x	x
GIMPLE_OMP_SECTIONS_SWITCH	x	x
GIMPLE_OMP_SINGLE	x	x
GIMPLE_PHI		x
GIMPLE_RESX		x
GIMPLE_RETURN	x	x
GIMPLE_SWITCH	x	x
GIMPLE_TRY	x	

12.3 异常处理

其它异常处理结构使用GIMPLE_TRY_CATCH来表示。GIMPLE_TRY_CATCH有两个操作数。第一个操作数为一个要执行的语句序列。如果执行这些语句并没有抛出异常，则第二个操作数被忽略。否则，如果有异常被抛出，则GIMPLE_TRY_CATCH的第二个操作数将被检查。第二个操作数可以具有以下形式：

1. 一个要执行的语句序列。当发生异常时，这些语句被执行，然后异常被重新抛出。
2. 一个GIMPLE_CATCH语句序列。每个GIMPLE_CATCH有一个可适用的异常类型列表和处理代码。如果被抛出的异常匹配其中一个类型，则相关的处理代码被执行。如果处理代码执行到结尾并结束，则在最初的GIMPLE_TRY_CATCH之后继续执行。
3. 一条GIMPLE_EH_FILTER语句。具有一个允许的异常类型列表，和当匹配失败时的处理代码。如果被抛出的异常不匹配所允许的类型之一，则相关的匹配失败代码会被执行。如果抛出的异常确实匹配，则继续查找下一个处理。

目前抛出异常并不直接用GIMPLE来表示，而是通过调用一个函数来实现。将来的某个时候，我们将增加某种方式来表示抛出已知类型的异常的调用。

就在运行优化器之前，编译器将高级别的EH结构下降为一组`goto'，魔术标号，以及EH区域。

12.4 Temporaries

当gimplification遇到一个过于复杂的子表达式的时候，会创建一个新的临时变量来存放子表达式的值，并且在当前语句之前，增加一条新的语句对其初始化。这些特殊的临时对象被称作`expression temporaries'，并使用get_formal_tmp_var来分配。编译器总是尝试将相等的表达式放到同一个临时对象中，来简化冗余计算消除。

只有当我们知道在使用表达式临时对象的值之前，其不会被重新求值的时候，才可以使用，否则其将不能被修改[3]。其它临时对象可以使用get_initialized_tmp_var或create_tmp_var来分配。

目前，像a = b + 5这样的表达式没有被进一步简化。我们曾经尝试将其转换成型如

```
T1 = b + 5;
a = T1;
```

的样子。但这会使表示变得膨胀，而无法获益。然而，必须在内存中的变量不能出现在表达式中；其值先被显式的加载到一个临时对象中。类似的，将表达式的值存放到内存变量中，也要通过一个临时对象。

12.5 操作数

总得来说，GIMPLE表达式由一个运算和适当数目的简单操作数组成；这些操作数必须或者为GIMPLE右值(is_gimple_val)，即一个常量，或者一个寄存器变量。更复杂的操作数被分解到临时对象中，所以，

```
a = b + c + d
```

会变成

```
T1 = b + c;
a = T1 + d;
```

对于GIMPLE_CALL的参数也是同样的规则。

一个赋值的目标通常为一个变量，但是也可以为一个INDIRECT_REF或者一个由下面描述的复合左值。

12.5.1 复合表达式

C逗号表达式的左手边被简单的移送到一个独立的语句中。

12.5.2 复合左值

目前涉及到数组和结构体域引用的复合左值，没有被分解；像a.b[2] = 42这样的表达式不再被简化（虽然是复杂的数组下标）。这种限制可以解决之后的优化器的局限性；如果我们要将其转换成

```
T1 = &a.b;
T1[2] = 42;
```

则别名分析无法记住对T1[2]的引用是来自a.b，所以，其会认为该赋值会与a的另一个成员有别名关系；这会使struct-alias-1.c运行失败。将来对优化器的改进可以不再需要限制。

12.5.3 条件表达式

C ?: 表达式被转换成一条if语句，每个分支被分配给相同的临时对象。所以，

```
a = b ? c : d;
```

会变成

[3] 这些限制源自Morgan 4.8.

```
if (b == 1)
 T1 = c;
else
 T1 = d;
a = T1;
```

GIMPLE级别的if-conversion过程在适当的时候，重新引入了?:表达式。其用于向量化循环。

注意在GIMPLE中，if语句通过GIMPLE_COND来表示，正如下面所描述。

12.5.4 逻辑运算符

除非它们出现在GIMPLE_COND的条件操作数中，否则逻辑的`and'和`or'操作符将按照下列方式进行简化：a = b && c变成

```
T1 = (bool)b;
if (T1 == true)
 T1 = (bool)c;
a = T1;
```

注意该例子中的T1不能为表达式临时对象，因为其具有两个不同的赋值。

12.5.5 操作操作数

所有的gimple操作数都是tree类型的。不过只有特定类型的tree可以被用作操作数元组。函数get_gimple_rhs_class可以进行基本的验证，其给定一个tree代码，返回一个enum，为下列enum gimple_rhs_class类型的值

- GIMPLE_INVALID_RHS 该tree不能用作GIMPLE操作数。
- GIMPLE_BINARY_RHS 该tree为一个有效的GIMPLE二元运算。
- GIMPLE_UNARY_RHS 该tree为一个有效的GIMPLE一元运算。
- GIMPLE_SINGLE_RHS 该tree为单个对象，不能被拆分成更简单的操作数（例如，SSA_NAME, VAR_DECL, COMPONENT_REF等等）。

  该操作数类别还作为转义通口，对于那些可以被平整到操作数向量中，但是右手边会需要多于两个插槽的tree节点。例如，(a op b) ? x : y的COND_EXPR表达式，会被平整到使用4个插槽的操作数向量中，但是其还需要额外的处理来从c = a op b ? x : y中判断c = a op b。对于ASSERT_EXPR，也有类似的情况。这些特殊情况的tree表达式应该被平整到操作数向量中。

对于在GIMPLE_BINARY_RHS和GIMPLE_UNARY_RHS类别中的tree节点，它们不能被直接存放在元组中。需要首先被平整，分隔到独立的部分。例如，给定GENERIC表达式

```
a = b + c
```

其tree表示为:

```
MODIFY_EXPR <VAR_DECL <a>, PLUS_EXPR <VAR_DECL <b>, VAR_DECL <c>>>
```

这种情况下，该语句的GIMPLE形式逻辑上等同于它的GENERIC形式，但是在GIMPLE中，赋值语句的右手边PLUS_EXPR，不被表示成一个tree，替代的，PLUS_EXPR的两个操作数子树被拿出来，并平整到GIMPLE元组中，如下：

```
GIMPLE_ASSIGN <PLUS_EXPR, VAR_DECL <a>, VAR_DECL <b>, VAR_DECL <c>>
```

12.5.6 操作数向量分配

操作数向量被存放在三元组结构的底部。这意味着，取决于给定语句的代码，其操作数向量相对于基本结构体的偏移量会不同。使用下列方法来访问元组操作数

unsigned gimple_num_ops (gimple g) [GIMPLE function]

返回语句G中的操作数个数。

tree gimple_op (gimple g, unsigned i) [GIMPLE function]
返回语句G的第I个操作数。

tree *gimple_ops (gimple g) [GIMPLE function]
返回指向语句G的操作数向量的指针。这通过内部称作gimple_ops_offset_[]的表来计算。该表的索引为G的gimple代码。

当编译器被构建时，将gimple.def中定义的每个语句代码，所对应的结构体大小来填充该表。因为操作数向量在结构体的底部，所以对于gimple代码C，其偏移量被计算为sizeof (struct-of C) - sizeof (tree)。

该机制对于使用gimple_op()的每次访问，都增加了一个内存重定向，如果这会变成瓶颈，则编译过程可以选择记住gimple_ops()的结果，并使用它来访问操作数。

12.5.7 操作数有效性

当为gimple语句增加一个新的操作数，将根据每个元组在它操作数向量中可以接受的情况来验证该操作数。这些断言由gimple_<name>_set_...()调用。元组会使用下列断言（注意，该列表并不全）：

is_gimple_operand (tree t) [GIMPLE function]
这是条件最宽的断言。其实质上是检查t是否具有GIMPLE_SINGLE_RHS的gimple_rhs_class。

is_gimple_val (tree t) [GIMPLE function]
返回真，如果t为一个“GIMPLE值”，其为所有非寻址的栈变量（is_gimple_reg返回真的变量）和常量（is_gimple_min_invariant返回真的表达式）。

is_gimple_addressable (tree t) [GIMPLE function]
返回真，如果t为一个符号，或者内存引用，其地址可以被使用。

is_gimple_asm_val (tree t) [GIMPLE function]
类似于is_gimple_val，不过其还接受硬件寄存器。

is_gimple_call_addr (tree t) [GIMPLE function]
返回真，如果t为一个有效的表达式，被作用由GIMPLE_CALL调用的函数。

is_gimple_constant (tree t) [GIMPLE function]
返回真，如果t为一个有效的gimple常量。

is_gimple_min_invariant (tree t) [GIMPLE function]
返回真，如果t为一个有效的最小不变量。这与常量不同，其特定的值在编译的时候可能不已知，但是知道其不会改变（例如，函数局部变量的地址）。

is_gimple_min_invariant_address (tree t) [GIMPLE function]
返回真，如果t为一个ADDR_EXPR，其在程序运行时不会改变。

12.5.8 语句有效性

is_gimple_assign (gimple g) [GIMPLE function]
返回真，如果g的代码为GIMPLE_ASSIGN。

is_gimple_call (gimple g) [GIMPLE function]
返回真，如果g的代码为GIMPLE_CALL。

gimple_assign_cast_p (gimple g) [GIMPLE function]
返回真，如果g为一个GIMPLE_ASSIGN并执行一个类型转换操作。

12.6 操作GIMPLE语句

这章介绍了所有可用于处理每个GIMPLE指令的函数。

12.6.1 通用访问方法

下列为对gimple语句的通用访问。

enum gimple_code gimple_code (gimple g) [GIMPLE function]
返回语句G的代码。

basic_block gimple_bb (gimple g) [GIMPLE function]
返回语句G所属的基本块。

tree gimple_block (gimple g) [GIMPLE function]
返回包含语句G的词法作用域。

tree gimple_expr_type (gimple stmt) [GIMPLE function]
返回STMT所计算的主表达式。如果STMT不做任何计算，则返回void_type_node。这将只为GIMPLE_ASSIGN, GIMPLE_COND 和 GIMPLE_CALL返回一些有意义的东西。对于所有其它元组代码，其将返回void_type_node。

enum tree_code gimple_expr_code (gimple stmt) [GIMPLE function]
返回STMT所计算的表达式的tree代码。这将只为GIMPLE_CALL, GIMPLE_ASSIGN和GIMPLE_COND返回一些有意义的东西。如果STMT为GIMPLE_CALL，其将返回CALL_EXPR。对于GIMPLE_COND，其返回比较断言的代码。对于GIMPLE_ASSIGN，其返回赋值语句的右手边所执行的操作代码。

void gimple_set_block (gimple g, tree block) [GIMPLE function]
将G的词法作用域块设置为BLOCK。

location_t gimple_locus (gimple g) [GIMPLE function]
返回语句G的locus信息。

void gimple_set_locus (gimple g, location_t locus) [GIMPLE function]
为语句G设置locus信息。

bool gimple_locus_empty_p (gimple g) [GIMPLE function]
返回真，如果G不具有locus信息。

bool gimple_no_warning_p (gimple stmt) [GIMPLE function]
返回真，如果对于语句STMT不会产生警告。

void gimple_set_visited (gimple stmt, bool visited_p) [GIMPLE function]
在语句STMT上将访问状态设置成VISITED_P。

bool gimple_visited_p (gimple stmt) [GIMPLE function]
返回语句STMT上的访问状态。

void gimple_set_plf (gimple stmt, enum plf_mask plf, bool val_p) [GIMPLE function]
将语句STMT上的编译过程局部标记PLF设置为VAL_P。

unsigned int gimple_plf (gimple stmt, enum plf_mask plf) [GIMPLE function]
返回语句STMT上的编译过程局部标记PLF。

bool gimple_has_ops (gimple g) [GIMPLE function]
返回真，如果语句G具有寄存器或内存操作数。

bool gimple_has_mem_ops (gimple g) [GIMPLE function]
返回真，如果语句G具有内存操作数。

unsigned gimple_num_ops (gimple g) [GIMPLE function]
返回语句G的操作数个数。

tree *gimple_ops (gimple g) [GIMPLE function]
返回语句G的操作数数组。

tree gimple_op (gimple g, unsigned i) [GIMPLE function]
返回语句G的操作数I。

tree *gimple_op_ptr (gimple g, unsigned i) [GIMPLE function]
返回语句G的操作数I的指针。

void gimple_set_op (gimple g, unsigned i, tree op) [GIMPLE function]
将语句G的操作数I设置为OP。

bitmap gimple_addresses_taken (gimple stmt) [GIMPLE function]
返回STMT使用过其地址的符号集合。

struct def_optype_d *gimple_def_ops (gimple g) [GIMPLE function]
返回语句G的DEF操作数集合。

void gimple_set_def_ops (gimple g, struct def_optype_d *def) [GIMPLE function]
将语句G的DEF操作数集合设置为DEF。

struct use_optype_d *gimple_use_ops (gimple g) [GIMPLE function]
返回语句G的USE操作数集合。

void gimple_set_use_ops (gimple g, struct use_optype_d *use) [GIMPLE function]
将语句G的USE操作数集合设置为USE。

struct voptype_d *gimple_vuse_ops (gimple g) [GIMPLE function]
返回语句G的VUSE操作数集合。

void gimple_set_vuse_ops (gimple g, struct voptype_d *ops) [GIMPLE function]
将语句G的VUSE操作数集合设置为OPS。

struct voptype_d *gimple_vdef_ops (gimple g) [GIMPLE function]
返回语句G的VDEF操作数集合。

void gimple_set_vdef_ops (gimple g, struct voptype_d *ops) [GIMPLE function]
将语句G的VDEF操作数集合设置为OPS。

bitmap gimple_loaded_syms (gimple g) [GIMPLE function]
返回语句G所加载的符号集合。集合中的每个元素为对应符号的DECL_UID。

bitmap gimple_stored_syms (gimple g) [GIMPLE function]
返回语句G所存储的符号集合。集合中的每个元素为对应符号的DECL_UID。

bool gimple_modified_p (gimple g) [GIMPLE function]
返回真，如果语句G具有操作数并且被修改域已经被设置。

bool gimple_has_volatile_ops (gimple stmt) [GIMPLE function]
返回真，如果语句STMT包含volatile操作数。

void gimple_set_has_volatile_ops (gimple stmt, bool volatilep) [GIMPLE function]
返回真，如果语句STMT包含volatile操作数。

void update_stmt (gimple s) [GIMPLE function]
将语句S标记为已经被修改，并对其进行更新。

void update_stmt_if_modified (gimple s) [GIMPLE function]
更新语句S，如果其已经被标记为被修改。

gimple gimple_copy (gimple stmt) [GIMPLE function]
返回语句STMT的一个深度复制。

12.7 元组特定访问方法

12.7.1 GIMPLE_ASM

gimple gimple_build_asm (const char *string, ninputs, noutputs, [GIMPLE function]
nclobbers, ...)
构建一条GIMPLE_ASM语句。该语句用于内联的汇编结构。STRING为汇编代码。NINPUT为寄存器输入的数目。NOUTPUT为寄存器输出的数目。NCLOBBERS为被破坏的寄存器的数目。剩下的参数tree对应于每个输入，输出和被破坏的寄存器。

gimple gimple_build_asm_vec (const char *, VEC(tree,gc) *, [GIMPLE function]
VEC(tree,gc) *, VEC(tree,gc) *)
等同于gimple_build_asm，不过参数在VEC中传递。

gimple_asm_ninputs (gimple g) [GIMPLE function]
返回GIMPLE_ASM G的输入操作数的数目。

gimple_asm_noutputs (gimple g) [GIMPLE function]
返回GIMPLE_ASM G的输出操作数的数目。

gimple_asm_nclobbers (gimple g) [GIMPLE function]
返回GIMPLE_ASM G的破坏操作数的数目。

tree gimple_asm_input_op (gimple g, unsigned index) [GIMPLE function]
返回GIMPLE_ASM G的索引为INDEX的输入操作数。

void gimple_asm_set_input_op (gimple g, unsigned index, tree in_op) [GIMPLE function]
将IN_OP设置为GIMPLE_ASM G的索引为INDEX的输入操作数。

tree gimple_asm_output_op (gimple g, unsigned index) [GIMPLE function]
返回GIMPLE_ASM G的索引为INDEX的输出操作数。

void gimple_asm_set_output_op (gimple g, unsigned index, tree out_op) [GIMPLE function]
将OUT_OP设置为GIMPLE_ASM G的索引为INDEX的输出操作数。

tree gimple_asm_clobber_op (gimple g, unsigned index) [GIMPLE function]
返回GIMPLE_ASM G的索引为INDEX的破坏操作数。

void gimple_asm_set_clobber_op (gimple g, unsigned index, tree clobber_op) [GIMPLE function]
将CLOBBER_OP设置为GIMPLE_ASM G的索引为INDEX的破坏操作数。

const char *gimple_asm_string (gimple g) [GIMPLE function]
返回GIMPLE_ASM G中的字符串表示的汇编指令。

bool gimple_asm_volatile_p (gimple g) [GIMPLE function]
返回真，如果G为一个标记为volatile的asm语句。

void gimple_asm_set_volatile (gimple g) [GIMPLE function]
将asm语句G标记为volatile。

void gimple_asm_clear_volatile (gimple g) [GIMPLE function]
从asm语句G中移除volatile标记。

12.7.2 GIMPLE_ASSIGN

gimple gimple_build_assign (tree lhs, tree rhs) [GIMPLE function]
构建一条GIMPLE_ASSIGN语句。左手边为lhs中传递的左值。右手边可以为一个一元或者二元tree表达式。表达式tree rhs将被平整，其操作数赋值给新语句中相应的操作数插槽中。该函数可用于，你已经有一个tree表达式，并想将其转成元组的时候。然而，不用为了调用该函数，而特意构建表达式tree。如果操作数已经是在独立的tree中，则最好使用gimple_build_assign_with_ops。

gimple gimplify_assign (tree dst, tree src, gimple_seq *seq_p) [GIMPLE function]
构建一个新的GIMPLE_ASSIGN元组，并将其追加到*SEQ_P的结尾。

DST/SRC分别为目的和源。你可以在DST或SRC中传递ungimplified tree，它们会在需要的时候被转换成gimple操作数。

该函数返回新创建的GIMPLE_ASSIGN元组。

gimple gimple_build_assign_with_ops (enum tree_code subcode, tree lhs, tree op1, tree op2) [GIMPLE function]
该函数类似于gimple_build_assign，不过是当赋值的右手边操作数已经被拆分成不同操作数的时候，用来构建一个GIMPLE_ASSIGN语句。

左手边为在lhs中传递的左值。subcode为赋值的右手边的tree_code。op1和op2为操作数。如果op2为null，则subcode必须为一个一元表达式的

enum tree_code gimple_assign_rhs_code (gimple g) [GIMPLE function]
返回赋值语句G的RHS上的表达式代码。

enum gimple_rhs_class gimple_assign_rhs_class (gimple g) [GIMPLE function]
返回赋值语句G右手边的表达式代码的gimple rhs类别。这个永远不会返回GIMPLE_INVALID_RHS。

tree gimple_assign_lhs (gimple g) [GIMPLE function]
返回赋值语句G的LHS。

tree *gimple_assign_lhs_ptr (gimple g) [GIMPLE function]
返回指向赋值语句G的LHS的指针。

tree gimple_assign_rhs1 (gimple g) [GIMPLE function]
返回指向赋值语句G的RHS的第一个操作数。

tree *gimple_assign_rhs1_ptr (gimple g) [GIMPLE function]
返回指向赋值语句G的RHS的第一个操作数的地址。

tree gimple_assign_rhs2 (gimple g) [GIMPLE function]
返回指向赋值语句G的RHS的第二个操作数。

tree *gimple_assign_rhs2_ptr (gimple g) [GIMPLE function]
返回指向赋值语句G的RHS的第二个操作数的地址。

void gimple_assign_set_lhs (gimple g, tree lhs) [GIMPLE function]
将LHS设置为赋值语句G的LHS操作数。

void gimple_assign_set_rhs1 (gimple g, tree rhs) [GIMPLE function]
将RHS设置为赋值语句G的RHS的第一个操作数。

tree *gimple_assign_rhs2_ptr (gimple g) [GIMPLE function]
返回指向赋值语句G的RHS的第二个操作数的指针。

void gimple_assign_set_rhs2 (gimple g, tree rhs) [GIMPLE function]
将RHS设置为赋值语句G的RHS的第二个操作数。

bool gimple_assign_cast_p (gimple s) [GIMPLE function]
返回真，如果S为一个有类型转换的赋值。

12.7.3 GIMPLE_BIND

gimple gimple_build_bind (tree vars, gimple_seq body) [GIMPLE function]
构建一条GIMPLE_BIND语句，使用VARS中的变量列表和BODY序列中的语句体。

tree gimple_bind_vars (gimple g) [GIMPLE function]
返回在GIMPLE_BIND语句G中声明的变量。

void gimple_bind_set_vars (gimple g, tree vars) [GIMPLE function]
将VARS设置为GIMPLE_BIND语句G中的声明变量集。

void gimple_bind_append_vars (gimple g, tree vars) [GIMPLE function]
将VARS追加到GIMPLE_BIND语句G中的声明变量集中。

gimple_seq gimple_bind_body (gimple g) [GIMPLE function]
返回在GIMPLE_BIND语句G中包含的GIMPLE序列。

void gimple_bind_set_body (gimple g, gimple_seq seq) [GIMPLE function]
将SEQ设置为GIMPLE_BIND语句G中包含的序列。

void gimple_bind_add_stmt (gimple gs, gimple stmt) [GIMPLE function]
追加一条语句到GIMPLE_BIND的主体的结尾。

void gimple_bind_add_seq (gimple gs, gimple_seq seq) [GIMPLE function]
追加一个语句序列到GIMPLE_BIND的主体的结尾。

tree gimple_bind_block (gimple g) [GIMPLE function]
返回与GIMPLE_BIND语句G相关联的TREE_BLOCK节点。这类似于tree中的BIND_EXPR_BLOCK域。

void gimple_bind_set_block (gimple g, tree block) [GIMPLE function]
将BLOCK设置为与GIMPLE_BIND语句G相关联的TREE_BLOCK节点。

12.7.4 GIMPLE_CALL

gimple gimple_build_call (tree fn, unsigned nargs, ...) [GIMPLE function]
构建一条对函数FN的GIMPLE_CALL语句。参数FN必须为一个FUNCTION_DECL或者一个由is_gimple_call_addr确定的gimple调用地址。NARGS为参数的数目。其余的参数在参数NARGS之后，必须为可以在gimple中作为右值的tree（即，每个操作数使用is_gimple_operand验证有效）。

gimple gimple_build_call_from_tree (tree call_expr) [GIMPLE function]
根据CALL_EXPR节点构建一个GIMPLE_CALL。参数和函数直接取自表达式。该函数假设call_expr已经是GIMPLE形式。也就是说，其操作数为GIMPLE值，并且函数调用不需要进一步的简化。call_expr中所有的调用标记被复制到新的GIMPLE_CALL中。

gimple gimple_build_call_vec (tree fn, VEC(tree, heap) *args) [GIMPLE function]
等同于gimple_build_call，不过参数是存储在VEC()中。

tree gimple_call_lhs (gimple g) [GIMPLE function]
返回调用语句G的LHS。

tree *gimple_call_lhs_ptr (gimple g) [GIMPLE function]
返回指向调用语句G的LHS的指针。

void gimple_call_set_lhs (gimple g, tree lhs) [GIMPLE function]
将LHS设置为调用语句G的LHS操作数。

tree gimple_call_fn (gimple g) [GIMPLE function]
返回调用语句G所调用的tree节点表示的函数。

void gimple_call_set_fn (gimple g, tree fn) [GIMPLE function]
将FN设置为调用语句G所调用的函数。这必须是一个gimple值，描述了被调用函数的地址。

tree gimple_call_fndecl (gimple g) [GIMPLE function]
如果给定的GIMPLE_CALL的调用者为一个FUNCTION_DECL，则将其返回。否则返回NULL。该函数类似于GENERIC中的get_callee_fndecl。

tree gimple_call_set_fndecl (gimple g, tree fndecl) [GIMPLE function]
将被调用的函数设置为FNDECL。

tree gimple_call_return_type (gimple g) [GIMPLE function]
返回调用语句G所返回的类型。

tree gimple_call_chain (gimple g) [GIMPLE function]
返回调用语句G的静态链。

void gimple_call_set_chain (gimple g, tree chain) [GIMPLE function]
将CHAIN设置为调用语句G的静态链。

gimple_call_num_args (gimple g) [GIMPLE function]
返回调用语句G的参数个数。

tree gimple_call_arg (gimple g, unsigned index) [GIMPLE function]
返回调用语句G在位置INDEX上的参数。第一个参数的索引为0。

tree *gimple_call_arg_ptr (gimple g, unsigned index) [GIMPLE function]
返回指向调用语句G在位置INDEX上的参数的指针。

void gimple_call_set_arg (gimple g, unsigned index, tree arg) [GIMPLE function]
将ARG设置为调用语句G在位置INDEX上的参数。

void gimple_call_set_tail (gimple s) [GIMPLE function]
将调用语句S标记为一个尾调用（即，就在exit函数之前的调用）。这些调用为尾调用优化的候选。

bool gimple_call_tail_p (gimple s) [GIMPLE function]
返回真，如果GIMPLE_CALL S被标记为尾调用。

void gimple_call_mark_uninlinable (gimple s) [GIMPLE function]
将GIMPLE_CALL S标记为不可内联的。

bool gimple_call_cannot_inline_p (gimple s) [GIMPLE function]
返回真，如果GIMPLE_CALL S不能被内联。

bool gimple_call_noreturn_p (gimple s) [GIMPLE function]
返回真，如果S为一个noreturn调用。

gimple gimple_call_copy_skip_args (gimple stmt, bitmap args_to_skip) [GIMPLE function]
构建一个GIMPLE_CALL，等同于STMT，不过跳过由ARGS_TO_SKIP集标记的位置参数。

12.7.5 GIMPLE_CATCH

gimple gimple_build_catch (tree types, gimple_seq handler) [GIMPLE function]
构建一个GIMPLE_CATCH语句。TYPES为该catch所处理的tree类型。HANDLER是一个语句序列，为处理代码。

tree gimple_catch_types (gimple g) [GIMPLE function]
返回由GIMPLE_CATCH语句G所处理的类型。

tree *gimple_catch_types_ptr (gimple g) [GIMPLE function]
返回指向由GIMPLE_CATCH语句G所处理的类型的指针。

gimple_seq gimple_catch_handler (gimple g) [GIMPLE function]
返回表示GIMPLE_CATCH语句G的处理者主体的GIMPLE序列。

void gimple_catch_set_types (gimple g, tree t) [GIMPLE function]
将T设置为GIMPLE_CATCH语句G处理的类型集。

void gimple_catch_set_handler (gimple g, gimple_seq handler) [GIMPLE function]
将HANDLER设置为GIMPLE_CATCH G的主体。

12.7.6 GIMPLE_CHANGE_DYNAMIC_TYPE

gimple gimple_build_cdt (tree type, tree ptr) [GIMPLE function]
构建一个GIMPLE_CHANGE_DYNAMIC_TYPE语句。TYPE为位置PTR的新类型。

tree gimple_cdt_new_type (gimple g) [GIMPLE function]
返回由GIMPLE_CHANGE_DYNAMIC_TYPE语句G设置的新类型。

tree *gimple_cdt_new_type_ptr (gimple g) [GIMPLE function]
返回一个指针，指向由GIMPLE_CHANGE_DYNAMIC_TYPE语句G设置的新类型。

void gimple_cdt_set_new_type (gimple g, tree new_type) [GIMPLE function]
将NEW_TYPE设置为GIMPLE_CHANGE_DYNAMIC_TYPE语句G返回的类型。

tree gimple_cdt_location (gimple g) [GIMPLE function]
返回GIMPLE_CHANGE_DYNAMIC_TYPE语句G所影响的位置。

tree *gimple_cdt_location_ptr (gimple g) [GIMPLE function]
返回一个指针，指向GIMPLE_CHANGE_DYNAMIC_TYPE语句G所影响的位置。

void gimple_cdt_set_location (gimple g, tree ptr) [GIMPLE function]
将PTR设置为GIMPLE_CHANGE_DYNAMIC_TYPE语句G所影响的位置。

12.7.7 GIMPLE_COND

gimple gimple_build_cond (enum tree_code pred_code, tree lhs, tree rhs, tree t_label, tree f_label) [GIMPLE function]
构建一个GIMPLE_COND语句。A GIMPLE_COND语句比较LHS 和 RHS，如果PRED_CODE中的条件为真，则跳到t_label中的标号上去，否则跳到f_label中的标号上去。PRED_CODE为关系操作符tree节点，比如EQ_EXPR, LT_EXPR, LE_EXPR, NE_EXPR等等。

gimple gimple_build_cond_from_tree (tree cond, tree t_label, tree f_label) [GIMPLE function]
跟条件表达式 treeCOND，构建一个 GIMPLE_COND语句。T_LABEL 和 F_LABEL与gimple_build_cond中的一样。

enum tree_code gimple_cond_code (gimple g) [GIMPLE function]
返回条件语句G计算的断言代码。

void gimple_cond_set_code (gimple g, enum tree_code code) [GIMPLE function]
将CODE设置为条件语句G的断言代码。

tree gimple_cond_lhs (gimple g) [GIMPLE function]
返回条件语句G要计算的断言的LHS操作数。

void gimple_cond_set_lhs (gimple g, tree lhs) [GIMPLE function]
将LHS设置为条件语句G要计算的断言的LHS操作数。

tree gimple_cond_rhs (gimple g) [GIMPLE function]
返回条件语句G要计算的断言的RHS操作数。

void gimple_cond_set_rhs (gimple g, tree rhs) [GIMPLE function]
将RHS设置为条件语句G要计算的断言的RHS操作数。

tree gimple_cond_true_label (gimple g) [GIMPLE function]
返回条件语句G当其断言求值为真时使用的标号。

void gimple_cond_set_true_label (gimple g, tree label) [GIMPLE function]
将LABEL设为条件语句G当其断言求值为真时使用的标号。

void gimple_cond_set_false_label (gimple g, tree label) [GIMPLE function]
将LABEL设为条件语句G当其断言求值为假时使用的标号。

tree gimple_cond_false_label (gimple g) [GIMPLE function]
返回条件语句G当其断言求值为假时使用的标号。

void gimple_cond_make_false (gimple g) [GIMPLE function]
将条件COND_STMT设置为'if (1 == 0)'的形式。

void gimple_cond_make_true (gimple g) [GIMPLE function]
将条件COND_STMT设置为'if (1 == 1)'的形式。

12.7.8 GIMPLE_EH_FILTER

gimple gimple_build_eh_filter (tree types, gimple_seq failure) [GIMPLE function]
构建一个GIMPLE_EH_FILTER语句。TYPES为过滤器的类型。FAILURE为一个序列，为过滤器的失败动作。

tree gimple_eh_filter_types (gimple g) [GIMPLE function]
返回GIMPLE_EH_FILTER语句G处理的类型。

tree *gimple_eh_filter_types_ptr (gimple g) [GIMPLE function]
返回一个指针，指向GIMPLE_EH_FILTER语句G处理的类型。

gimple_seq gimple_eh_filter_failure (gimple g) [GIMPLE function]
返回当GIMPLE_EH_FILTER语句失败时执行的语句序列。

void gimple_eh_filter_set_types (gimple g, tree types) [GIMPLE function]
将TYPES设置为GIMPLE_EH_FILTER语句G处理的类型集。

void gimple_eh_filter_set_failure (gimple g, gimple_seq failure) [GIMPLE function]
将FAILURE设置为GIMPLE_EH_FILTER语句失败时执行的语句序列。

bool gimple_eh_filter_must_not_throw (gimple g) [GIMPLE function]
返回EH_FILTER_MUST_NOT_THROW标记。

void gimple_eh_filter_set_must_not_throw (gimple g, bool mntp) [GIMPLE function]
设置EH_FILTER_MUST_NOT_THROW标记。

12.7.9 GIMPLE_LABEL

gimple gimple_build_label (tree label) [GIMPLE function]
构建一个GIMPLE_LABEL语句，对应于tree标号LABEL。

tree gimple_label_label (gimple g) [GIMPLE function]
返回GIMPLE_LABEL语句G使用的LABEL_DECL节点。

void gimple_label_set_label (gimple g, tree label) [GIMPLE function]
将LABEL设置为GIMPLE_LABEL语句G使用的LABEL_DECL节点。

gimple gimple_build_goto (tree dest) [GIMPLE function]
构建一个到标号DEST的GIMPLE_GOTO语句。

tree gimple_goto_dest (gimple g) [GIMPLE function]
返回无条件跳转G的目的。

void gimple_goto_set_dest (gimple g, tree dest) [GIMPLE function]
将DEST设置为无条件跳转G的目的。

12.7.10 GIMPLE_NOP

gimple gimple_build_nop (void) [GIMPLE function]
构建一个GIMPLE_NOP语句。

bool gimple_nop_p (gimple g) [GIMPLE function]
返回TRUE，如果语句G为一个GIMPLE_NOP。

12.7.11 GIMPLE_OMP_ATOMIC_LOAD

gimple gimple_build_omp_atomic_load (tree lhs, tree rhs) [GIMPLE function]
构建一个GIMPLE_OMP_ATOMIC_LOAD语句。LHS为赋值的左手边。RHS为赋值的右手边。

void gimple_omp_atomic_load_set_lhs (gimple g, tree lhs) [GIMPLE function]
设置原子加载的LHS。

tree gimple_omp_atomic_load_lhs (gimple g) [GIMPLE function]
获得原子加载的LHS。

void gimple_omp_atomic_load_set_rhs (gimple g, tree rhs) [GIMPLE function]
设置原子加载的RHS。

tree gimple_omp_atomic_load_rhs (gimple g) [GIMPLE function]
获得原子加载的RHS。

12.7.12 GIMPLE_OMP_ATOMIC_STORE

gimple gimple_build_omp_atomic_store (tree val) [GIMPLE function]
构建一个GIMPLE_OMP_ATOMIC_STORE语句。VAL为要存储的值。

void gimple_omp_atomic_store_set_val (gimple g, tree val) [GIMPLE function]
设置在原子存储中要存储的值。

tree gimple_omp_atomic_store_val (gimple g) [GIMPLE function]
返回在原子存储中要存储的值。

12.7.13 GIMPLE_OMP_CONTINUE

gimple gimple_build_omp_continue (tree control_def, tree control_use) [GIMPLE function]
构建一个GIMPLE_OMP_CONTINUE语句。CONTROL_DEF为控制变量的定义。CONTROL_USE为对控制变量的使用。

tree gimple_omp_continue_control_def (gimple s) [GIMPLE function]
返回S中GIMPLE_OMP_CONTINUE的控制变量的定义。

tree gimple_omp_continue_control_def_ptr (gimple s) [GIMPLE function]
与上面相同，不过是返回指针。

tree gimple_omp_continue_set_control_def (gimple s) [GIMPLE function]
设置S中GIMPLE_OMP_CONTINUE的控制变量的定义。

tree gimple_omp_continue_control_use (gimple s) [GIMPLE function]
返回S中GIMPLE_OMP_CONTINUE的对控制变量的使用。

tree gimple_omp_continue_control_use_ptr (gimple s) [GIMPLE function]
与上面相同，不过是返回指针。

tree gimple_omp_continue_set_control_use (gimple s) [GIMPLE function]
设置S中GIMPLE_OMP_CONTINUE的对控制变量的使用。

12.7.14 GIMPLE_OMP_CRITICAL

gimple gimple_build_omp_critical (gimple_seq body, tree name) [GIMPLE function]
构建一个GIMPLE_OMP_CRITICAL语句。BODY为只有一个线程可以执行的语句序列。NAME为可选的该临界块的标识。

tree gimple_omp_critical_name (gimple g) [GIMPLE function]
返回OMP_CRITICAL语句G关联的名字。

tree *gimple_omp_critical_name_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP临界语句G的名字。

void gimple_omp_critical_set_name (gimple g, tree name) [GIMPLE function]
设置NAME为OMP临界语句G的名字。

12.7.15 GIMPLE_OMP_FOR

gimple gimple_build_omp_for (gimple_seq body, tree clauses, tree index, tree initial, tree final, tree incr, gimple_seq pre_body, enum tree_code omp_for_cond) [GIMPLE function]
构建一个GIMPLE_OMP_FOR语句。BODY为for循环中的语句序列。CLAUSES为OMP循环结构的从句：private, firstprivate, lastprivate, reductions, ordered, schedule, 和 nowait。PRE_BODY为循环不变的语句序列。INDEX为索引变量。INITIAL为INDEX的初始值。FINAL为INDEX的最终值。OMP_FOR_COND为断言，用于比较INDEX和FINAL。INCR为递增表达式。

tree gimple_omp_for_clauses (gimple g) [GIMPLE function]
返回OMP_FOR G相关联的从句。

tree *gimple_omp_for_clauses_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_FOR G。

void gimple_omp_for_set_clauses (gimple g, tree clauses) [GIMPLE function]
将CLAUSES设置为OMP_FOR G相关联的从句列表。

tree gimple_omp_for_index (gimple g) [GIMPLE function]
返回OMP_FOR G的索引变量。

tree *gimple_omp_for_index_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_FOR G的索引变量。

void gimple_omp_for_set_index (gimple g, tree index) [GIMPLE function]
将INDEX设置为OMP_FOR G的索引变量。

tree gimple_omp_for_initial (gimple g) [GIMPLE function]
返回OMP_FOR G的初始值。

tree *gimple_omp_for_initial_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_FOR G的初始值。

void gimple_omp_for_set_initial (gimple g, tree initial) [GIMPLE function]
将INITIAL设置为OMP_FOR G的初始值。

tree gimple_omp_for_final (gimple g) [GIMPLE function]
返回OMP_FOR G的最终值。

tree *gimple_omp_for_final_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_FOR G的最终值。

void gimple_omp_for_set_final (gimple g, tree final) [GIMPLE function]
将FINAL设置为OMP_FOR G的最终值。

tree gimple_omp_for_incr (gimple g) [GIMPLE function]
返回OMP_FOR G的递增值。

tree *gimple_omp_for_incr_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_FOR G的递增值。

void gimple_omp_for_set_incr (gimple g, tree incr) [GIMPLE function]
将INCR设置为OMP_FOR G的递增值。

gimple_seq gimple_omp_for_pre_body (gimple g) [GIMPLE function]
返回在OMP_FOR语句G开始之前执行的语句序列。

void gimple_omp_for_set_pre_body (gimple g, gimple_seq pre_body) [GIMPLE function]
将PRE_BODY设置为OMP_FOR语句G开始之前执行的语句序列。

void gimple_omp_for_set_cond (gimple g, enum tree_code cond) [GIMPLE function]
将COND设置为OMP_FOR G的条件代码。

enum tree_code gimple_omp_for_cond (gimple g) [GIMPLE function]
返回OMP_FOR G关联的条件代码。

12.7.16 GIMPLE_OMP_MASTER

gimple gimple_build_omp_master (gimple_seq body) [GIMPLE function]
构建一个GIMPLE_OMP_MASTER语句。BODY为只被master执行的语句序列。

12.7.17 GIMPLE_OMP_ORDERED

gimple gimple_build_omp_ordered (gimple_seq body) [GIMPLE function]
构建一个GIMPLE_OMP_ORDERED语句。

BODY为循环中顺序执行的语句序列。

12.7.18 GIMPLE_OMP_PARALLEL

gimple gimple_build_omp_parallel (gimple_seq body, tree clauses, tree child_fn, tree data_arg) [GIMPLE function]
构建一个GIMPLE_OMP_PARALLEL语句。

BODY为并行执行的语句序列。CLAUSES为OMP并行结构从句。CHILD_FN为创建的并行线程执行的函数。DATA_ARG 为共享的数据参数。

bool gimple_omp_parallel_combined_p (gimple g) [GIMPLE function]
返回真，如果OMP并行语句G设置了GF_OMP_PARALLEL_COMBINED标记。

void gimple_omp_parallel_set_combined_p (gimple g) [GIMPLE function]
设置OMP并行语句G的GF_OMP_PARALLEL_COMBINED域。

gimple_seq gimple_omp_body (gimple g) [GIMPLE function]
返回OMP语句G的主体。

void gimple_omp_set_body (gimple g, gimple_seq body) [GIMPLE function]
将BODY设置为OMP语句G的主体。

tree gimple_omp_parallel_clauses (gimple g) [GIMPLE function]
返回OMP_PARALLEL G关联的从句。

tree *gimple_omp_parallel_clauses_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_PARALLEL G关联的从句。

void gimple_omp_parallel_set_clauses (gimple g, tree clauses) [GIMPLE function]
将CLAUSES设置为OMP_PARALLEL G关联的从句列表。

tree gimple_omp_parallel_child_fn (gimple g) [GIMPLE function]
返回用于存放OMP_PARALLEL主体的子函数。

tree *gimple_omp_parallel_child_fn_ptr (gimple g) [GIMPLE function]
返回一个指针，指向用于存放OMP_PARALLEL G的主体的子函数。

void gimple_omp_parallel_set_child_fn (gimple g, tree child_fn) [GIMPLE function]
将CHILD_FN设置为OMP_PARALLEL G的子函数。

tree gimple_omp_parallel_data_arg (gimple g) [GIMPLE function]
返回在OMP_PARALLEL G中用于从父线程到子线程发送变量和值的人工参数。

tree *gimple_omp_parallel_data_arg_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_PARALLEL G的数据参数。

void gimple_omp_parallel_set_data_arg (gimple g, tree data_arg) [GIMPLE function]
将DATA_ARG设置为OMP_PARALLEL G的数据参数。

bool is_gimple_omp (gimple stmt) [GIMPLE function]
返回真，当gimple语句STMT为OpenMP类型的。

12.7.19 GIMPLE_OMP_RETURN

gimple gimple_build_omp_return (bool wait_p) [GIMPLE function]
构建一个GIMPLE_OMP_RETURN语句。WAIT_P为真，如果这是一个非等待的返回。

void gimple_omp_return_set_nowait (gimple s) [GIMPLE function]
设置GIMPLE_OMP_RETURN语句S的nowait标记。

bool gimple_omp_return_nowait_p (gimple g) [GIMPLE function]
返回真，如果OMP返回语句G设置了GF_OMP_RETURN_NOWAIT标记。

12.7.20 GIMPLE_OMP_SECTION

gimple gimple_build_omp_section (gimple_seq body) [GIMPLE function]
构建一个GIMPLE_OMP_SECTION语句。

BODY为段中的语句序列。

bool gimple_omp_section_last_p (gimple g) [GIMPLE function]
返回真，如果OMP段语句G设置了GF_OMP_SECTION_LAST标记。

void gimple_omp_section_set_last (gimple g) [GIMPLE function]
设置G的GF_OMP_SECTION_LAST标记。

12.7.21 GIMPLE_OMP_SECTIONS

gimple gimple_build_omp_sections (gimple_seq body, tree clauses) [GIMPLE function]
构建一个GIMPLE_OMP_SECTIONS语句。BODY为段语句序列。CLAUSES为任意OMP段结构的从句：private, firstprivate, lastprivate, reduction, 和 nowait。

gimple gimple_build_omp_sections_switch (void) [GIMPLE function]
构建一个GIMPLE_OMP_SECTIONS_SWITCH语句。

tree gimple_omp_sections_control (gimple g) [GIMPLE function]
返回G中GIMPLE_OMP_SECTIONS相关联的控制变量。

tree *gimple_omp_sections_control_ptr (gimple g) [GIMPLE function]
返回一个指针，指向G中GIMPLE_OMP_SECTIONS相关联的控制变量。

void gimple_omp_sections_set_control (gimple g, tree control) [GIMPLE function]
将CONTROL设置为G中GIMPLE_OMP_SECTIONS相关联的控制变量。

tree gimple_omp_sections_clauses (gimple g) [GIMPLE function]
返回G中GIMPLE_OMP_SECTIONS相关联的从句。

tree *gimple_omp_sections_clauses_ptr (gimple g) [GIMPLE function]
返回一个指针，指向G中GIMPLE_OMP_SECTIONS相关联的从句。

void gimple_omp_sections_set_clauses (gimple g, tree clauses) [GIMPLE function]
将CLAUSES设置为G中GIMPLE_OMP_SECTIONS相关联的从句。

12.7.22 GIMPLE_OMP_SINGLE

gimple gimple_build_omp_single (gimple_seq body, tree clauses) [GIMPLE function]
构建一个GIMPLE_OMP_SINGLE语句。BODY为只被执行一次的语句序列。CLAUSES为任何OMP单结构的从句：private, firstprivate, copyprivate, nowait。

tree gimple_omp_single_clauses (gimple g) [GIMPLE function]
返回OMP_SINGLE G关联的从句。

tree *gimple_omp_single_clauses_ptr (gimple g) [GIMPLE function]
返回一个指针，指向OMP_SINGLE G关联的从句。

void gimple_omp_single_set_clauses (gimple g, tree clauses) [GIMPLE function]
将CLAUSES设置为OMP_SINGLE G关联的从句。

12.7.23 GIMPLE_PHI

gimple make_phi_node (tree var, int len) [GIMPLE function]
构建一个PHI节点，对于变量var有len个参数槽。

unsigned gimple_phi_capacity (gimple g) [GIMPLE function]
返回GIMPLE_PHI G支持的最大参数数目。

unsigned gimple_phi_num_args (gimple g) [GIMPLE function]
返回GIMPLE_PHI G中的参数数目。这必须总是为包含G的基本块的输出边的个数。

tree gimple_phi_result (gimple g) [GIMPLE function]
返回由GIMPLE_PHI G创建的SSA名字。

tree *gimple_phi_result_ptr (gimple g) [GIMPLE function]
返回一个指针，指向由GIMPLE_PHI G创建的SSA名字。

void gimple_phi_set_result (gimple g, tree result) [GIMPLE function]
将RESULT设置为由GIMPLE_PHI G创建的SSA名字。

struct phi_arg_d *gimple_phi_arg (gimple g, index) [GIMPLE function]
返回对应于GIMPLE_PHI G的输入边索引为INDEX的PHI参数。

void gimple_phi_set_arg (gimple g, index, struct phi_arg_d * phiarg) [GIMPLE function]
将PHIARG设置为对应于GIMPLE_PHI G的输入边索引为INDEX的参数。

12.7.24 GIMPLE_RESX

gimple gimple_build_resx (int region) [GIMPLE function]
构建一个GIMPLE_RESX语句。该语句是_Unwind_Resume的占位，在我们知道是否需要函数调用或者分支之前。REGION为异常区域。

int gimple_resx_region (gimple g) [GIMPLE function]
返回GIMPLE_RESX G的区域编号。

void gimple_resx_set_region (gimple g, int region) [GIMPLE function]
将REGION设置为GIMPLE_RESX G的区域编号。

12.7.25 GIMPLE_RETURN

gimple gimple_build_return (tree retval) [GIMPLE function]
构建一个GIMPLE_RETURN语句，其返回值为retval。

tree gimple_return_retval (gimple g) [GIMPLE function]
返回GIMPLE_RETURN G的返回值。

void gimple_return_set_retval (gimple g, tree retval) [GIMPLE function]
将RETVAL设置为GIMPLE_RETURN G的返回值。

12.7.26 GIMPLE_SWITCH

gimple gimple_build_switch (nlabels, tree index, tree default_label, ...) [GIMPLE function]
构建一个GIMPLE_SWITCH语句。NLABELS为不包括缺省标号的标号数目。缺省标号在DEFAULT_LABEL中传递。其余的参数为表示标号的tree。每个标号为一个代码为CASE_LABEL_EXPR的tree。

gimple gimple_build_switch_vec (tree index, tree default_label, VEC(tree,heap) *args) [GIMPLE function]
该函数为一个候选的方式，用来构建GIMPLE_SWITCH语句。INDEX和DEFAULT_LABEL与gimple_build_switch中的一样。ARGS为一个包含了标号的CASE_LABEL_EXPR tree向量。

unsigned gimple_switch_num_labels (gimple g) [GIMPLE function]
返回与switch语句G相关联的标号数目。

void gimple_switch_set_num_labels (gimple g, unsigned nlabels) [GIMPLE function]
将switch语句G的标号数目设置为NLABELS。

tree gimple_switch_index (gimple g) [GIMPLE function]
返回switch语句G的索引变量。

void gimple_switch_set_index (gimple g, tree index) [GIMPLE function]
将INDEX设置为switch语句G的索引变量。

tree gimple_switch_label (gimple g, unsigned index) [GIMPLE function]
返回编号为INDEX的标号。缺省标号为0，接着是swith语句中的其它标号。

void gimple_switch_set_label (gimple g, unsigned index, tree label) [GIMPLE function]
将LABEL的编号设置为INDEX。0总是为缺省编号。

tree gimple_switch_default_label (gimple g) [GIMPLE function]
返回switch语句的缺省标号。

void gimple_switch_set_default_label (gimple g, tree label) [GIMPLE function]
为switch语句设置缺省标号。

12.7.27 GIMPLE_TRY

gimple gimple_build_try (gimple_seq eval, gimple_seq cleanup, unsigned int kind) [GIMPLE function]
构建一个GIMPLE_TRY语句。EVAL为要求值的表达式序列。CLEANUP为在清除时运行的语句序列。KIND为枚举值GIMPLE_TRY_CATCH，如果该语句表示一个try/catch结构，或者GIMPLE_TRY_FINALLY，如果该语句表示一个try/finally结构。

enum gimple_try_flags gimple_try_kind (gimple g) [GIMPLE function]
返回GIMPLE_TRY G表示的try块的种类。这是GIMPLE_TRY_CATCH 或 GIMPLE_TRY_FINALLY。

bool gimple_try_catch_is_cleanup (gimple g) [GIMPLE function]
返回GIMPLE_TRY_CATCH_IS_CLEANUP标记。

gimple_seq gimple_try_eval (gimple g) [GIMPLE function]
返回GIMPLE_TRY G的主体使用的语句序列。

gimple_seq gimple_try_cleanup (gimple g) [GIMPLE function]
返回GIMPLE_TRY G的清除体使用的语句序列。

void gimple_try_set_catch_is_cleanup (gimple g, bool catch_is_cleanup) [GIMPLE function]
设置GIMPLE_TRY_CATCH_IS_CLEANUP标记。

void gimple_try_set_eval (gimple g, gimple_seq eval) [GIMPLE function]
将EVAL设置为GIMPLE_TRY G的主体使用的语句序列。

void gimple_try_set_cleanup (gimple g, gimple_seq cleanup) [GIMPLE function]
将CLEANUP设置为GIMPLE_TRY G的清除体使用的语句序列。

12.7.28 GIMPLE_WITH_CLEANUP_EXPR

gimple gimple_build_wce (gimple_seq cleanup) [GIMPLE function]
构建一条GIMPLE_WITH_CLEANUP_EXPR语句。CLEANUP为一个清除表达式。

gimple_seq gimple_wce_cleanup (gimple g) [GIMPLE function]
返回清除语句G的清除序列。

void gimple_wce_set_cleanup (gimple g, gimple_seq cleanup) [GIMPLE function]
将CLEANUP设置为G的清除序列。

bool gimple_wce_cleanup_eh_only (gimple g) [GIMPLE function]
返回WCE元组的CLEANUP_EH_ONLY标记。

void gimple_wce_set_cleanup_eh_only (gimple g, bool eh_only_p) [GIMPLE function]
为WCE元组设置CLEANUP_EH_ONLY标记。

12.8 GIMPLE序列

GIMPLE序列等价于GENERIC中使用的STATEMENT_LIST。它们用于将语句链接在一起，当和序列迭代器一起使用的时候，可以提供一个迭代语句的框架。

GIMPLE序列的类型为struct gimple_sequence。序列指针的类型为gimple_seq，其与struct gimple_sequence *相同。当声明一个局部序列时，你可以定义一个类型为struct gimple_sequence的局部变量。当声明一个分配在垃圾搜集堆中的序列时，使用下面介绍的函数gimple_seq_alloc。

在标题为序列迭代器的章节中，有一些便利的函数用于在序列中进行迭代。

下面是一个函数列表，用来操作和查询序列。

void gimple_seq_add_stmt (gimple_seq *seq, gimple g) [GIMPLE function]
如果G不为NULL，将一条gimple语句链接到序列*SEQ的结尾。如果*SEQ为NULL，则在链接之前分配一个序列。

void gimple_seq_add_seq (gimple_seq *dest, gimple_seq src) [GIMPLE function]
如果SRC不为NULL，则将序列SRC追加到序列*DEST的结尾。如果*DEST为NULL，则在追加之前分配一个新的序列。

gimple_seq gimple_seq_deep_copy (gimple_seq src) [GIMPLE function]
对序列SRC执行深度复制，并返回结果。

gimple_seq gimple_seq_reverse (gimple_seq seq) [GIMPLE function]
反转序列SEQ中语句的顺序。返回SEQ。

gimple gimple_seq_first (gimple_seq s) [GIMPLE function]
返回序列S中的第一条语句。

gimple gimple_seq_last (gimple_seq s) [GIMPLE function]
返回序列S中的最后一条语句。

void gimple_seq_set_last (gimple_seq s, gimple last) [GIMPLE function]
将序列S中的最后一条语句设置为LAST中的语句。

void gimple_seq_set_first (gimple_seq s, gimple first) [GIMPLE function]
将序列S中的第一条语句设置为FIRST中的语句。

void gimple_seq_init (gimple_seq s) [GIMPLE function]
将序列S初始化为空序列。

gimple_seq gimple_seq_alloc (void) [GIMPLE function]
在可以被垃圾搜集的存储中分配一个新的序列，并将其返回。

void gimple_seq_copy (gimple_seq dest, gimple_seq src) [GIMPLE function]
将序列SRC复制到序列DEST。

bool gimple_seq_empty_p (gimple_seq s) [GIMPLE function]
如果序列S为空，则返回真。

gimple_seq bb_seq (basic_block bb) [GIMPLE function]
返回BB中的语句序列。

void set_bb_seq (basic_block bb, gimple_seq seq) [GIMPLE function]
将BB中的语句序列设置成SEQ。

bool gimple_seq_singleton_p (gimple_seq seq) [GIMPLE function]
确定序列SEQ中是否只包含一条语句。

12.9 序列迭代器

序列迭代器为一些便利的结构，用于在序列中迭代语句。给定序列SEQ，典型的对gimple序列迭代器的使用为：

```
gimple_stmt_iterator gsi;

for (gsi = gsi_start (seq); !gsi_end_p (gsi); gsi_next (&gsi))
 {
  gimple g = gsi_stmt (gsi);
  /* Do something with gimple statement G.  */
 }
```

也可以向后迭代：

```
for (gsi = gsi_last (seq); !gsi_end_p (gsi); gsi_prev (&gsi))
```

在基本块上进行前向和后向迭代可以通过配合使用gsi_start_bb和gsi_last_bb。

在下面的介绍中，我们有时会用到enum gsi_iterator_update。对于该枚举的有效操作有：

- GSI_NEW_STMT 只有当增加单个语句被时才有效。将迭代器移动到该处。
- GSI_SAME_STMT 将迭代器放在相同的语句处。
- GSI_CONTINUE_LINKING 将迭代器移动到在相同方向上，适合链接其它语句的位置。

下面为一个函数列表，用于操作和使用语句迭代器。

gimple_stmt_iterator gsi_start (gimple_seq seq) [GIMPLE function]
返回一个新的迭代器，指向序列SEQ的第一个语句。如果SEQ为空，则迭代器的基本块为NULL。当迭代器总是需要正确设置基本块的时候，使用gsi_start_bb。

gimple_stmt_iterator gsi_start_bb (basic_block bb) [GIMPLE function]
返回一个新的迭代器，指向基本块BB中的第一条语句。

gimple_stmt_iterator gsi_last (gimple_seq seq) [GIMPLE function]
返回一个新的迭代器，初始化为指向序列SEQ中的最后一条语句。如果SEQ为空，则迭代器的基本块为NULL。当迭代器总是需要正确设置基本块的时候，使用gsi_last_bb。

gimple_stmt_iterator gsi_last_bb (basic_block bb) [GIMPLE function]
返回一个新的迭代器，指向基本块BB中的最后一条语句。

bool gsi_end_p (gimple_stmt_iterator i) [GIMPLE function]
如果位于I的结尾，则返回TRUE。

bool gsi_one_before_end_p (gimple_stmt_iterator i) [GIMPLE function]
如果是I的结尾之前的一个语句，则返回TRUE。

void gsi_next (gimple_stmt_iterator *i) [GIMPLE function]
将迭代器前进到下一个gimple语句。

`void gsi_prev (gimple_stmt_iterator *i)` [GIMPLE function]
将迭代器前进到前一个gimple语句。

`gimple gsi_stmt (gimple_stmt_iterator i)` [GIMPLE function]
返回当前的stmt。

`gimple_stmt_iterator gsi_after_labels (basic_block bb)` [GIMPLE function]
返回一个块语句迭代器，指向块BB中的第一个非标号的语句。

`gimple *gsi_stmt_ptr (gimple_stmt_iterator *i)` [GIMPLE function]
返回指向当前stmt的指针。

`basic_block gsi_bb (gimple_stmt_iterator i)` [GIMPLE function]
返回与该迭代器关联的基本块。

`gimple_seq gsi_seq (gimple_stmt_iterator i)` [GIMPLE function]
返回与该迭代器关联的序列。

`void gsi_remove (gimple_stmt_iterator *i, bool remove_eh_info)` [GIMPLE function]
从序列中移除当前stmt。迭代器被更新为指向下一条语句。当REMOVE_EH_INFO为真，则我们将迭代器I指向的语句从EH表中移除。否则我们不修改EH表。通常当语句将从IL中被移除，并不被插入到其它地方的时候，REMOVE_EH_INFO应该为真。

`void gsi_link_seq_before (gimple_stmt_iterator *i, gimple_seq seq, enum gsi_iterator_update mode)` [GIMPLE function]
将语句序列SEQ链接在由迭代器I指向的语句之前。MODE指出了插入操作之后，迭代器要做什么（参见上面的`enum gsi_iterator_update`）。

`void gsi_link_before (gimple_stmt_iterator *i, gimple g, enum gsi_iterator_update mode)` [GIMPLE function]
将语句G链接在由迭代器I指向的语句之前。根据MODE来更新迭代器I。

`void gsi_link_seq_after (gimple_stmt_iterator *i, gimple_seq seq, enum gsi_iterator_update mode)` [GIMPLE function]
将序列SEQ链接在由迭代器I指向的语句之后。MODE与在`gsi_insert_after`中的相同。

`void gsi_link_after (gimple_stmt_iterator *i, gimple g, enum gsi_iterator_update mode)` [GIMPLE function]
将语句G链接在由迭代器I指向的语句之后。MODE与在`gsi_insert_after`中的相同。

`gimple_seq gsi_split_seq_after (gimple_stmt_iterator i)` [GIMPLE function]
将I之后的所有语句移送到新的序列中。返回该新的序列。

`gimple_seq gsi_split_seq_before (gimple_stmt_iterator *i)` [GIMPLE function]
将I之前的所有语句移送到新的序列中。返回该新的序列。

`void gsi_replace (gimple_stmt_iterator *i, gimple stmt, bool update_eh_info)` [GIMPLE function]
将由I指向的语句替换为STMT。如果UPDATE_EH_INFO为真，则原来语句的异常处理信息被移送到新的语句中。

void gsi_insert_before (gimple_stmt_iterator *i, gimple stmt, enum gsi_iterator_update mode) [GIMPLE function]

在由迭代器I指向的语句之前插入语句STMT，更新STMT的基本块并扫描新的操作数。MODE描述了插入操作之后，如何更新迭代器I(参见enum gsi_iterator_update)。

void gsi_insert_seq_before (gimple_stmt_iterator *i, gimple_seq seq, enum gsi_iterator_update mode) [GIMPLE function]

类似于gsi_insert_before，不过是对于SEQ中的所有语句。

void gsi_insert_after (gimple_stmt_iterator *i, gimple stmt, enum gsi_iterator_update mode) [GIMPLE function]

在由迭代器I指向的语句之后插入语句STMT，更新STMT的基本块并扫描新的操作数。MODE描述了插入操作之后，如何更新迭代器I(参见enum gsi_iterator_update)。

void gsi_insert_seq_after (gimple_stmt_iterator *i, gimple_seq seq, enum gsi_iterator_update mode) [GIMPLE function]

类似于gsi_insert_after，不过是对于SEQ中的所有语句。

gimple_stmt_iterator gsi_for_stmt (gimple stmt) [GIMPLE function]

查找STMT的迭代器。

void gsi_move_after (gimple_stmt_iterator *from, gimple_stmt_iterator *to) [GIMPLE function]

将语句移送到FROM处，使得其正好位于TO处的语句之后。

void gsi_move_before (gimple_stmt_iterator *from, gimple_stmt_iterator *to) [GIMPLE function]

将语句移送到FROM处，使得其正好位于TO处的语句之前。

void gsi_move_to_bb_end (gimple_stmt_iterator *from, basic_block bb) [GIMPLE function]

将FROM处的语句移送到基本块BB的结尾。

void gsi_insert_on_edge (edge e, gimple stmt) [GIMPLE function]

将STMT增加到边E的待定列表中。直到调用gsi_commit_edge_inserts()之前，不会进行实际的插入操作。

void gsi_insert_seq_on_edge (edge e, gimple_seq seq) [GIMPLE function]

将SEQ中语句序列增加到边E的待定列表中。直到调用gsi_commit_edge_inserts()之前，不会进行实际的插入操作。

basic_block gsi_insert_on_edge_immediate (edge e, gimple stmt) [GIMPLE function]

类似于gsi_insert_on_edge+gsi_commit_edge_inserts。如果需要创建一个新的块，则将其返回。

void gsi_commit_one_edge_insert (edge e, basic_block *new_bb) [GIMPLE function]

提交在边E上进行的插入操作。如果创建了新的基本块，则将NEW_BB设置为该块，否则将其设置为NULL。

void gsi_commit_edge_inserts (void) [GIMPLE function]

该函数将提交所有要进行的边插入操作，并在需要的时候创建新的基本块。

12.10 增加一个新的GIMPLE语句代码

增加一个新的GIMPLE语句代码，第一步是修改文件gimple.def，其包含了所有的GIMPLE代码。然后，你必须增加一个相应的结构体，以及union gimple_statement_d中的一个实体，这些都在gimple.h中。这将要求你在gsstruct.def中增加一个相应的GTY标记，以及在gss_for_code中增加处理该标记的代码，这位于gimple.c中。

为了让垃圾搜集器知道你在gimple.h中创建的结构体的大小，你需要在gimple_size增加一个case来处理你的新的GIMPLE语句，位于gimple.c中。

你可能想创建一个函数来构建新的gimple语句，在gimple.c中。该函数应该被称作gimple_build_<NEW_TUPLE_NAME>，并返回类型为gimple的新的元组。

如果你的新语句需要对其成员或者操作数进行访问的代码，则在gimple.h中放入简单的inline访问代码，以及在gimple.c中任何不平凡的访问代码，并在gimple.h中有相应的函数原型。

12.11 语句和操作数遍历

有两个函数可以用于遍历语句和序列：分别为walk_gimple_stmt和walk_gimple_seq。还有第三个函数用于遍历语句中的操作数：walk_gimple_op。

tree walk_gimple_stmt (gimple_stmt_iterator *gsi, walk_stmt_fn callback_stmt, walk_tree_fn callback_op, struct walk_stmt_info *wi) [GIMPLE function]

该函数用于在GSI中遍历当前语句，并可选的使用WI中存放的遍历状态。如果WI为NULL，则在遍历中不保存状态。

回调函数CALLBACK_STMT被调用。如果CALLBACK_STMT返回真，则意味着回调函数已经处理了语句的所有操作数，并且无需遍历它的操作数。

如果CALLBACK_STMT为NULL或者返回假，则CALLBACK_OP会在语句的每个操作数上被调用，通过walk_gimple_op。如果walk_gimple_op对任意操作数返回了非NULL，则剩下的操作数将不被扫描。

返回值为对walk_gimple_op最后调用所返回的值，或者如果没有指定CALLBACK_OP则返回NULL_TREE。

tree walk_gimple_op (gimple stmt, walk_tree_fn callback_op, struct walk_stmt_info *wi) [GIMPLE function]

使用该函数来遍历语句STMT的操作数。每个操作数通过walk_tree来遍历，并使用WI中可选的状态信息。

CALLBACK_OP在STMT的每个操作数上被调用，通过walk_tree。walk_tree的其它参数必须存放在WI中。对于每个操作数OP，walk_tree被调用为：

```
walk_tree (&OP, CALLBACK_OP, WI, WI->PSET)
```

如果CALLBACK_OP对于一个操作数返回非NULL，则剩下的操作数不再被扫描。返回值为对walk_tree最后调用所返回的值，或者如果没有指定CALLBACK_OP则返回NULL_TREE。

tree walk_gimple_seq (gimple_seq seq, walk_stmt_fn callback_stmt, walk_tree_fn callback_op, struct walk_stmt_info *wi) [GIMPLE function]

该函数遍历序列SEQ中的所有语句，在每个语句上调用walk_gimple_stmt。WI跟walk_gimple_stmt中的一样。如果walk_gimple_stmt返回非NULL，则停止遍历，并返回值。否则，所有语句都被遍历并返回NULL_TREE。

13 分析和优化GIMPLE元组

在编译过程中，GCC使用了三种主要的中间语言来表示程序：GENERIC，GIMPLE和RTL。GENERIC是一种由每个前端生成的语言无关的表示。它用来作为语法分析器和优化器之间的接口。GENERIC是一种通用表示，能够表示使用GCC支持的所有语言编写的程序。

GIMPLE和RTL用于优化程序。GIMPLE用于目标和语言无关的优化（例如，内联，常数传播，尾调用消除，冗余消除等）。与GENERIC比较相似，GIMPLE是一种语言无关的树型表示。不过，与GENERIC不同的是GIMPLE的语法有更多的限制：表达式不包含3个以上的操作数（函数调用除外），它没有控制流程结构，并且具有副作用的表达式只允许出现在赋值语句的右端。详情参见描述GENERIC和GIMPLE的章节。

本章描述在GIMPLE优化器（也称为“树优化器”或者“中端”）中使用的数据结构和函数。特别是侧重于所有的宏，数据结构，函数和实现GIMPLE优化过程所需要的编程架构。

13.1 Annotations

优化器需要在优化过程中将属性和变量关联起来。例如，我们需要知道一条语句属于哪个基本块，或者一个变量是否具有别名。所有这些属性被存储在叫做注解（annotation）的数据结构中，并被连接到 `struct tree_common` 的 `ann` 域中。

目前，我们定义了变量的注解 (`var_ann_t`)。注解在 `tree-flow.h' 中有定义和文档描述。

13.2 SSA操作数

几乎每条GIMPLE语句都会包含对变量或者内存地址的引用。由于语句的形状和大小不同，它们的操作数也将会位于语句树中的不同位置。为了便于访问语句的操作数，它们被组织到与语句的注解（annotation）相关联的一个列表中。操作数列表中的每个元素都是一个指向`VAR_DECL`, `PARM_DECL` 或 `SSA_NAME`树结点的指针。这就为检查和替换操作数提供了一种非常方便的方法。

数据流分析和优化是在所有表示变量的树结点上完成的。扫描语句操作数时，将会考虑所有`SSA_VAR_P`返回非零的节点。但是，并不是所有的`SSA_VAR_P`变量都使用同一种方式来处理。出于优化的目的，我们需要区分对局部标量的引用和对全局变量，静态变量，结构体，数组，别名变量的引用，等等。原因很简单，一方面，编译器能够为局部标量收集完整的数据流信息；另一方面，全局变量可能会被函数调用所修改，另外，可能无法追踪数组所有的元素或结构体所有的域的信息，等等。

操作数扫描器搜集两类操作数：实的（real）和虚的（virtual）。`is_gimple_reg`返回真的操作数被认为是实操作数，否则为一个虚操作数。我们还区分了它们的使用和定义。如果操作数的值被语句加载（例如，操作数在赋值的右边），则为使用。如果语句给操作数赋于了一个新的值（例如，操作数在赋值语句的左边），则为定义。

虚操作数和实操作数还具有不同的数据流属性。实操作数是对它们表示的完整对象的明确引用。例如，给定

```
{
 int a, b;
 a = b
}
```

由于`a`和`b`为非别名的局部变量，语句`a = b`将具有一个实定义和一个实使用，因为变量`a`完全被变量`b`的内容修改了。实定义还被称作为killing definition（杀死定义）。类似的，对`b`的使用是读取了它的所有位。

与此相反，虚操作数用于具有部分或者不明确引用的变量。这包括结构体，数组，全局变量和别名变量。这些情况下，我们具有两种类型的定义。对于全局变量，结构体和数组，我们能够从语句

中确定这些类型的变量是否具有一个killing definition（杀死定义）。如果具有， 则语句被标记为具有那个变量的必然定义（must definition）。但是， 如果语句只是定义了变量的一部分（即，结构体中的一个域）， 或者如果我们知道语句可能会定义变量，但是不确定， 则我们将那条语句标记为具有一个可能定义（may definition）。例如，给定

```
{
 int a, b, *p;

 if (...)
  p = &a;
 else
  p = &b;
 *p = 5;
 return *p;
}
```

赋值*p = 5可能为a或者b的定义。 如果我们不能静态地确定在存储操作的时候p的指向， 我们便创建一个虚定义来标记那条语句为一个a和b的潜在的定义。 内存加载也类似的使用虚操作数进行标记。 虚操作数在树转储（dump）中显示在包含它们的语句前面。 要获得带有虚操作数的树转储，使用`-fdump-tree'的`-vops'选项：

```
{
 int a, b, *p;

 if (...)
  p = &a;
 else
  p = &b;
 # a = VDEF <a>
 # b = VDEF <b>
 *p = 5;

 # VUSE <a>
 # VUSE <b>
 return *p;
}
```

注意VDEF操作数具有被引用变量的两个副本。 这表明不是一个那个变量的killing definition（杀死定义）。在这种情况下， 我们称它为一个可能定义（may definition）或者 别名存储（aliased store）。 当函数被转换为SSA形式的时候，VDEF操作数的第二个变量副本将会变得很重要。 其将用于链接所有的非killing definition（杀死定义），用来防止优化对它们做错误的假设。

当语句完成时，便会立刻通过调用update_stmt来更新操作数。 如果语句元素通过SET_USE或SET_DEF被改变， 则不需要进一步的动作（即，那些宏会处理好语句更新）。 如果改变是通过直接操作语句的树，则必须在完成时调用update_stmt。 调用bsi_insert例程中的任何一个，或者bsi_replace， 都会隐式的调用update_stmt。

13.2.1 操作数迭代器和访问例程

与操作数相关的代码都在`tree-ssa-operands.c'中。 操作数被存储在每条语句的注解中并且可以通过操作数迭代器或者访问例程来访问。

下列访问例程可以用来检查操作数：

1. SINGLE_SSA_{USE,DEF,TREE}_OPERAND: 这些访问例程将会返回NULL，除非恰好有一个操作数匹配指定的标志。 如果恰好存在一个操作数，则操作数被作为tree,def_operand_p或者use_operand_p返回。

```
tree t = SINGLE_SSA_TREE_OPERAND (stmt, flags);
```

```
use_operand_p u = SINGLE_SSA_USE_OPERAND (stmt, SSA_ALL_VIRTUAL_USES);
def_operand_p d = SINGLE_SSA_DEF_OPERAND (stmt, SSA_OP_ALL_DEFS);
```

2. ZERO_SSA_OPERANDS: 该宏返回真，如果没有操作数匹配指定的标志。

```
if (ZERO_SSA_OPERANDS (stmt, SSA_OP_ALL_VIRTUALS))
 return;
```

3. NUM_SSA_OPERANDS: 该宏返回匹配'flags'的操作数数目。其实际上是执行了一个循环来进行统计，所以最好只有在真正需要的时候才使用它。

```
int count = NUM_SSA_OPERANDS (stmt, flags)
```

如果你想迭代一些或者所有操作数， 使用FOR_EACH_SSA_{USE,DEF,TREE}_OPERAND迭代器。 例如，要打印语句的所有操作数：

```
void
print_ops (tree stmt)
{
 ssa_op_iter;
 tree var;

 FOR_EACH_SSA_TREE_OPERAND (var, stmt, iter, SSA_OP_ALL_OPERANDS)
  print_generic_expr (stderr, var, TDF_SLIM);
}
```

如何选择合适的迭代器：

1. 确定你是否需要看到操作数指针，或者只是树，并选择合适的宏

```
Need      Macro:
----      -------
use_operand_p  FOR_EACH_SSA_USE_OPERAND
def_operand_p  FOR_EACH_SSA_DEF_OPERAND
tree      FOR_EACH_SSA_TREE_OPERAND
```

2. 你需要声明一个你感兴趣的类型的变量，和一个用作循环控制变量的ssa_op_iter结构体
3. 确定你想使用哪些操作数，并指定你所感兴趣的那些操作数的标志。 这些标志在`tree-ssa-operands.h'中声明：

```
#define SSA_OP_USE        0x01   /* Real USE operands. */
#define SSA_OP_DEF        0x02   /* Real DEF operands. */
#define SSA_OP_VUSE       0x04   /* VUSE operands. */
#define SSA_OP_VMAYUSE      0x08   /* USE portion of VDEFS. */
#define SSA_OP_VDEF       0x10   /* DEF portion of VDEFS. */

/* These are commonly grouped operand flags. */
#define SSA_OP_VIRTUAL_USES   (SSA_OP_VUSE | SSA_OP_VMAYUSE)
#define SSA_OP_VIRTUAL_DEFS   (SSA_OP_VDEF)
#define SSA_OP_ALL_USES     (SSA_OP_VIRTUAL_USES | SSA_OP_USE)
#define SSA_OP_ALL_DEFS     (SSA_OP_VIRTUAL_DEFS | SSA_OP_DEF)
#define SSA_OP_ALL_OPERANDS   (SSA_OP_ALL_USES | SSA_OP_ALL_DEFS)
```

所以，如果你想查看所有USE和VUSE操作数的use指针， 则可以使用类似下面的方法：

```
use_operand_p use_p;
ssa_op_iter iter;

FOR_EACH_SSA_USE_OPERAND (use_p, stmt, iter, (SSA_OP_USE | SSA_OP_VUSE))
 {
  process_use_ptr (use_p);
 }
```

宏TREE基本上与宏USE和DEF相同， 除了通过USE_FROM_PTR (use_p)和DEF_FROM_PTR (def_p)进行的use或def的解引用。因为我们不会使用操作数指针，所以可以混合use和def标志。

```
tree var;
ssa_op_iter iter;

FOR_EACH_SSA_TREE_OPERAND (var, stmt, iter, SSA_OP_VUSE)
 {
  print_generic_expr (stderr, var, TDF_SLIM);
 }
```

VDEF被分解为两个标记，一个是DEF部分（SSA_OP_VDEF），一个是USE部分（SSA_OP_VMAYUSE）。 如果你只是想要查看合在一起的VDEF，则可以使用第四个迭代器， 其返回语句中每个VDEF的def_operand_p和use_operand_p。 注意该宏不需要任何标记。

```
use_operand_p use_p;
def_operand_p def_p;
ssa_op_iter iter;

FOR_EACH_SSA_MAYDEF_OPERAND (def_p, use_p, stmt, iter)
 {
  my_code;
 }
```

代码中也有很多例子，同时在`tree-ssa-operands.h'中也有文档。

还有一些stmt迭代器是用于处理PHI节点的。

FOR_EACH_PHI_ARG跟FOR_EACH_SSA_USE_OPERAND非常类似， 只不过它是工作于PHI参数，而不是语句操作数。

```
/* Look at every virtual PHI use. */
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_VIRTUAL_USES)
{
 my_code;
}

/* Look at every real PHI use. */
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_USES)
 my_code;

/* Look at every PHI use. */
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_ALL_USES)
 my_code;
```

FOR_EACH_PHI_OR_STMT_{USE,DEF}与 FOR_EACH_SSA_{USE,DEF}_OPERAND非常类似， 只不过它是作用于语句或者PHI节点。 这些宏可以在合适的时候使用，但是它们比单独使用FOR_EACH_PHI和FOR_EACH_SSA例程的效率要低。

```
FOR_EACH_PHI_OR_STMT_USE (use_operand_p, stmt, iter, flags)
 {
  my_code;
 }

FOR_EACH_PHI_OR_STMT_DEF (def_operand_p, phi, iter, flags)
 {
  my_code;
 }
```

13.2.2 立即使用

现在immediate use（这个短语咋翻译？）信息总是可以被获得。 使用immediate use迭代器，你可以检查任意SSA_NAME的所有使用。 例如，要将ssa_var的所有使用改为ssa_var2， 并且之后在每个stmt上调用fold_stmt：

```
use_operand_p imm_use_p;
imm_use_iterator iterator;
tree ssa_var, stmt;


FOR_EACH_IMM_USE_STMT (stmt, iterator, ssa_var)
 {
  FOR_EACH_IMM_USE_ON_STMT (imm_use_p, iterator)
   SET_USE (imm_use_p, ssa_var_2);
  fold_stmt (stmt);
 }
```

这里有两个可以使用的迭代器。 FOR_EACH_IMM_USE_FAST用于当immediate use没有被改变的情况下，即， 只是进行查看use，但不设置它们。

如果确实要做改变，则必须要考虑到迭代器下没有被改变的事物，这时， 可以使用FOR_EACH_IMM_USE_STMT和FOR_EACH_IMM_USE_ON_STMT迭代器。 它们试图通过将语句的所有use移动到一个被控制的位置并对它们进行迭代的方式， 来保证use列表的完整性。然后优化就能够在所有的use被处理完后来操作stmt。 这比FAST版本的有点慢，因为它增加了一个占位元素并且必须对每条语句的列表进行排序。 如果循环被提前终止，则该占位元素还必须被移除。 宏BREAK_FROM_IMM_USE_SAFE用于做这个：

```
FOR_EACH_IMM_USE_STMT (stmt, iterator, ssa_var)
 {
  if (stmt == last_stmt)
   BREAK_FROM_SAFE_IMM_USE (iter);

  FOR_EACH_IMM_USE_ON_STMT (imm_use_p, iterator)
   SET_USE (imm_use_p, ssa_var_2);
  fold_stmt (stmt);
 }
```

在verify_ssa中有一些检测用来验证immediate use列表是最新的， 同时还检测一个优化是否没有使用该宏而中断循环。 在FOR_EACH_IMM_USE_FAST遍历中，直接使用'break'语句是安全的。

一些有用的函数和宏：

1. has_zero_uses (ssa_var)：如果没有ssa_var的使用，则返回真。
2. has_single_use (ssa_var)：如果只有ssa_var的单个使用，则返回真。
3. single_imm_use (ssa_var, use_operand_p *ptr, tree *stmt)：如果只有ssa_var的单个使用，则返回真， 并且还在第二和第三个参数中返回使用指针和所在的语句。
4. num_imm_uses (ssa_var)：返回ssa_var的immediate use的数目。最好不要使用该宏， 因为它只是简单的使用循环来统计use。
5. PHI_ARG_INDEX_FROM_USE (use_p)：给定一个在PHI节点中的use，返回use的索引数。 如果use不位于PHI节点中，则会触发一个断言。
6. USE_STMT (use_p)：返回use所在的语句。

注意在语句通过bsi_*程序被实际插入指令流中之前， use是不被放入immediate use列表中的。

还可以使用懒散的语句更新方式，不过这应该在确实需要的时候才使用。 别名分析和dominator优化目前都采用了这种方式。

当使用懒散更新（lazy updating）时，immediate use信息是过时的，不能被信赖。 懒散更新简单的调用mark_stmt_modified来标记语句被修改了， 而不使用update_stmt。当不再需要进行懒散更新时， 所有修改的语句都必须调用update_stmt来保持更新。 这必须在优化完成之前进行，否则verify_ssa将触发abort 异常中断。

这是通过对指令流进行简单的循环来实现的：

```
block_stmt_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
 {
  for (bsi = bsi_start (bb); !bsi_end_p (bsi); bsi_next (&bsi))
   update_stmt_if_modified (bsi_stmt (bsi));
 }
```

13.3 静态单赋值

大多数树优化器都依赖于静态单赋值（SSA）形式所提供的数据流信息。 我们是按照R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991中的描述来实现SSA形式的。

SSA形式基于的前提是程序变量只在程序中的一个位置被赋值。 对同一变量的多次赋值将创建那个变量的新的版本。 实际的程序最初自然很少是SSA形式的，因为变量一般会被赋值多次。 编译器修改程序表示，使得代码中每次变量被赋值的时候，便会创建一个新版本的变量。 不同版本的同一变量通过变量名字的版本号作为下标来区分开。 在表达式右端使用的变量被重命名，使得它们的版本号匹配最近的赋值。

我们使用SSA_NAME节点来表示变量版本。 `tree-ssa.c'中的重命名程序将每个实操作数和虚操作数，用包含了版本号和创建SSA_NAME的语句的SSA_NAME节点包裹起来。 只有定义和虚定义可能会创建新的SSA_NAME节点。

有时，控制流使得无法确定变量的最近版本是多少。这种情况下，编译器插入一个那个变量的人造定义，称作PHI function或者PHI node。 这个新的定义将变量的所有可能引入的版本合并一起，以创建一个新的名字。例如，

```
if (...)
 a_1 = 5;
else if (...)
 a_2 = 2;
else
 a_3 = 13;

# a_4 = PHI <a_1, a_2, a_3>
return a_4;
```

由于不可能确定在运行时，将运行三个分支中的哪一个，所以我们不知道在return语句中要使用a_1,a_2或a_3中的哪一个。 因此，SSA重命名将会创建一个新的版本a_4，其被赋值为“合并”a_1, a_2和a_3的结果。 因此，PHI节点意味着“这些操作数中的一个，我不知道是哪一个”。

下面的宏可以用来检查PHI节点。

PHI_RESULT (phi) [Macro]

返回由PHI节点phi(即, phi's LHS)创建的SSA_NAME。

PHI_NUM_ARGS (phi) [Macro]

返回phi中的参数个数。这个数目就是持有phi的基本块所引入的边的数目。

PHI_ARG_ELT (phi, i) [Macro]

返回phi的第i个参数的tuple表示。 tuple中的每个元素包含了一个SSA_NAME var和var借以流向的引入边。

PHI_ARG_EDGE (phi, i) [Macro]

返回phi的第i个参数对应的引入边。

PHI_ARG_DEF (phi, i) [Macro]

返回phi的第i个参数的SSA_NAME。

13.3.1 保持SSA形式

一些优化过程会改变函数并使得不再具有SSA特性。 这可能会发生在当一个过程增加了新的符号或者改变了程序使得变量不再被别名的时候。 不管什么时候发生类似的情况，受到影响的符号必须被再次重命名为SSA形式。 产生新代码或者替代存在的语句的转换也需要更新SSA形式。

由于GCC为寄存器和虚变量实现了两种不同的SSA形式， 所有保持SSA形式的更新取决于你是否正在更新寄存器或者虚名字。 这两种情况对于不断的SSA更新的背后思想是类似的：当新的SSA名字被创建时， 它们通常意味着要替换程序中的其它存在的名字。

例如，给定下列代码：

```
1 L0:
2 x_1 = PHI (0, x_5)
3 if (x_1 < 10)
4   if (x_1 > 7)
5     y_2 = 0
6   else
7     y_3 = x_1 + x_7
8   endif
9   x_5 = x_1 + 1
10  goto L0;
11 endif
```

假设我们插入了新的名字x_10和x_11（第4行和第8行）。

```
1 L0:
2 x_1 = PHI (0, x_5)
3 if (x_1 < 10)
4   x_10 = ...
5   if (x_1 > 7)
6     y_2 = 0
7   else
8     x_11 = ...
9     y_3 = x_1 + x_7
10  endif
11  x_5 = x_1 + 1
12  goto L0;
13 endif
```

我们想使用x_10和x_11的新的定义来替换x_1的所有使用。 注意将要被替换的使用只在行5, 9和11中。而且， 第9行x_7的使用不应被替换（这就是为什么我们不能仅仅标记符号x为重命名）。

另外，我们可能需要在第11行插入一个PHI节点， 因为有一个x_10和x_11的合并点。 所以x_1在第11行的使用将用新的PHI节点来替换。 PHI节点的插入是可选的。它们并不完全必要用于保持SSA形式， 并且取决于调用者的插入内容，它们可能对优化器没有用处。

更新SSA形式分为两步。首先，过程必须分别出哪些名字需要被更新， 以及哪些符号需要被重命名为SSA形式。当新的名字被引入以替换程序中现存的名字时， 新旧名字之间的映射通过调用register_new_name_mapping来注册（注意如果你的过程通过复制基本块创建了新的代码， 对tree_duplicate_bb的调用将会自动建立所需的映射）。另一方面， 如果你的过程使得一个新的符号需要为SSA形式， 则新符号需要使用mark_sym_for_renaming来注册。

在替换映射被注册完，并且新符号被标记了要重命名后， 将会调用update_ssa来按照注册的进行改变。 这可以通过显示的调用或者为你的过程在tree_opt_pass结构体中创建 TODO标记来完成。 这里有几个TODO标记用于控制update_ssa的行为：

- TODO_update_ssa. 采用为新出现的符号插入PHI节点，以及虚名字进行标记的方式更新SSA形式。 当更新实名字时，只为0_j的所有新旧定义所到达的块中的实名字0_j 插入PHI节点。如果0_j的迭代的dominance边界没有被截枝， 我们可以在块中危机具有一个或多个没有即来定义的0_j结束插入PHI节点。 这将导致对0_j符号的未初始化警告。
- TODO_update_ssa_no_phi. 不使用插入任何新PHI节点的方式来更新SSA形式。这被用于要自己插入所有PHI节点的 过程或者只需要更新use-def和def-def链的虚名字的过程（例如，DCE）。
- TODO_update_ssa_full_phi. 在任何需要的地方都插入PHI节点。不进行IDF的截枝。 这被过程用于需要0_j的PHI节点的情况 （例如，pass_linear_transform）。

  警告: 如果你需要使用这个标记，则有可能你的过程是在做一些错误的事情。 为一个旧名字插入PHI节点可能会导致沉默的codegen错误或者虚假的未初始化警告。
- TODO_update_ssa_only_virtuals. 自己更新SSA的过程可能想要使用虚名字更新来代表通用的更新。因为FUD链易于维护， 所有这简化了他们所需的工作。注意：如果使用了该标记， 则任何实名字OLD->NEW的映射将被显式的破坏，只有标记为重命名的符合被处理。

13.3.2 保持虚SSA形式

虚SSA形式比非虚SSA形式要难以保持，主要是因为语句的虚操作数集可能会意外的改变。 通常，语句修改应该被对push_stmt_changes和pop_stmt_changes的调用 所包裹。例如，

```
munge_stmt (tree stmt)
{
  push_stmt_changes (&stmt);
  ... rewrite STMT ...
  pop_stmt_changes (&stmt);
}
```

对push_stmt_changes的调用保存了语句操作数的当前状态， 对pop_stmt_changes的调用比较保存的状态和现在的， 并对适当的符号标记为SSA重命名。

当处理一个语句栈时，通过使用LIFO顺序来调用push_stmt_changes和 pop_stmt_changes，可以一次修改多条语句。

另外，如果过程在调用push_stmt_changes后发现它不需要改变语句， 它可以通过调用discard_stmt_changes来简单的丢弃最顶层的缓存。 这将避免用来确定是否符合需要被标记为重命名所需的昂贵的操作数重扫描操作和缓存比较。

13.3.3 检验SSA_NAME节点

下面的宏可以用来检查SSA_NAME节点

SSA_NAME_DEF_STMT (var) [Macro]

返回创建SSA_NAME var的语句s。 如过s是空语句（即，IS_EMPTY_STMT (s)返回true），则意味着对该变量的第一个引用是一个USE或者VUSE。

SSA_NAME_VERSION (var) [Macro]

返回SSA_NAME对象var的版本号。

13.3.4 遍历use-def链

void walk_use_def_chains (var, fn, data) [Tree SSA function]

对use-def链的遍历起始于SSA_NAME节点var。 对每一个发现的可达定义调用函数fn。函数fn接受三个参数：var， 它的定义语句（def_stmt）和一个通用指针指向fn可能想要维护的任何状态 信息（数据）。函数fn可以通过返回true来停止遍历，否则要继续遍历， fn应该返回false。

注意，如果def_stmt是一个PHI节点，则语法有点不同。 对PHI节点的每个参数arg，该函数将：

1. 为arg遍历use-def链
2. 调用FN (arg, phi, data).

注意不管fn的第一个是否还是最初的变量var，目前都会检测PHI的参数。 如果fn想获得var，则应该调用PHI_RESULT (phi)。

13.3.5 遍历支配树

void walk_dominator_tree (walk_data, bb) [Tree SSA function]

该函数遍历当前CFG的支配树， 并调用在`domwalk.h'中struct dom_walk_data里定义的一系列回调函数。 你所需要定义的回调函数可以用于在遍历过程中的不同点执行自定义的代码：

1. 当处理bb和它的孩子（children）时，在初始化所需要的任何局部数据的时候。 该局部数据被压入一个内部的栈中，该栈在遍历支配树时会被自动的压入和弹出。
2. 在遍历bb中的所有语句之前。
3. 对于bb中的每条语句。
4. 当遍历过所有语句之后，并在递归到bb的支配孩子之前。
5. 然后递归到bb的所有支配孩子。
6. 在递归到bb的所有支配孩子之后，可选的， 重新遍历bb中的每条语句（即，重复步骤2和3）。
7. 当遍历完bb和bb的支配孩子中的所有语句之后。这时，块局部数据栈被弹出。

13.4 别名分析

别名分析经历4个主要阶段：

1. 结构体的别名分析。

   该过程遍历结构体类型的变量，并确定哪些域可以使用域的偏移量和大小来重叠。 对于每个域，一个称作“结构体域标签”（SFT）的“子变量”被创建， 其使用独立的变量来表示那个域。对于给定域， 所有可能会重叠的访问将具有那个域的SFT的虚操作数。

```
struct foo
{
 int a;
 int b;
}
struct foo temp;
int bar (void)
{
 int tmp1, tmp2, tmp3;
 SFT.0_2 = VDEF <SFT.0_1>
 temp.a = 5;
```

```
  SFT.1_4 = VDEF <SFT.1_3>
  temp.b = 6;

  VUSE <SFT.1_4>
  tmp1_5 = temp.b;
  VUSE <SFT.0_2>
  tmp2_6 = temp.a;

  tmp3_7 = tmp1_5 + tmp2_6;
  return tmp3_7;
}
```

如果你出于某种原因为一个变量复制符合标签，则有可能还要复制它的子变量。

2. Points-to和escape分析

   该过程遍历SSA web中的use-def链，查看三件事情：

   - P_i = &VAR形式的赋值
   - P_i = malloc()形式的赋值
   - 逃逸当前函数的指针和ADDR_EXPR

   逃逸的概念跟Java世界中使用的相同。当一个指针或者一个ADDR_EXPR逃逸， 指的是它已经被暴露在当前函数之外。所以，全局变量的赋值，函数参数， 以及返回指针都是逃逸的地点。

   这是我们目前所限制的。因为并不是所有都被重命名到SSA，例如， 当指针被隐藏在一个结构体的域中的时候，我们就丢失了逃逸属性。 在那些情况下，我们假设指针是逃逸的。

   我们使用逃逸分析来确定是否变量为call-clobbered。简单的说， 如果一个ADDR_EXPR逃逸，则变量是call-clobbered。如果一个指针P_i逃逸， 则所有P_i指向的变量（以及它的内存标签）也逃逸。

3. 计算流敏感别名

   我们有两类内存标签。内存标签与程序中的指针所指向的数据类型相关。 这些标签称为“符号内存标签”（SMT）。另一类是那些与SSA_NAME相关的， 称作“名字内存标签”（NMT）。基本的想法是，当为一个INDIRECT_REF *P_i增加操作数时， 我们将先检查P_i是否具有名字标签，如果有的话我们就使用， 因为那将具有更加精确的别名信息。否则，我们使用标准的符号标签。

   在这一阶段，我们遍历在points-to分析中发现的所有指针， 并为与每个指针P_i关联的名字内存标签创建别名集。如果P_i逃逸， 我们标记它指向的变量和它的标签为call-clobbered。

4. 计算流不敏感别名

   该过程将比较每个符号内存标签的别名集与程序中发现的每个可寻址的变量。 给定一个符号内存标签SMT和一个可寻址变量V。 如果SMT的别名集和V冲突（通过may_alias_p来计算获得）， 则V被标记为一个别名标签并被增加到SMT的别名集中。

   每个希望执行语言相关的别名分析的语言应该定义一个函数，给定一个tree结点， 能够计算结点的别名集。在不同别名集中的节点不允许进行别名。例如， 参见C语言前端函数c_get_alias_set。

例如，考虑下面的函数：

```
foo (int i)
{
 int *p, *q, a, b;

 if (i > 10)
  p = &a;
```

```
 else
  q = &b;

 *p = 3;
 *q = 5;
 a = b + 2;
 return *p;
}
```

在别名分析完成之后，指针p的符号内存标签将具有两个别名， 变量a和b。当每次指针p被dereference时， 我们想要标记操作为一个对a和b的潜在的引用。

```
foo (int i)
{
 int *p, a, b;

 if (i_2 > 10)
  p_4 = &a;
 else
  p_6 = &b;
 # p_1 = PHI <p_4(1), p_6(2)>;

 # a_7 = VDEF <a_3>;
 # b_8 = VDEF <b_5>;
 *p_1 = 3;

 # a_9 = VDEF <a_7>
 # VUSE <b_8>
 a_9 = b_8 + 2;

 # VUSE <a_9>;
 # VUSE <b_8>;
 return *p_1;
}
```

在一些情况下，一个指针的可能别名列表可能会变得很大。 这会造成代码中插入的虚操作数的数目的膨胀，使得内存消耗和编译时间增大。

当表示别名加载和存储所需要的虚操作数数目增长的太大的时候（可以使用`--param max-aliased-vops'来配置）， 别名集被分组以避免严重的编译时间下降和内存消耗。别名分组的heuristic如下：

1. 将指针列表按照分配的虚操作数数目进行递减排序。
2. 从列表中取出第一个指针并反转内存标签和别名的脚色。通常， 不管什么时候被别名的变量Vi被发现与内存标签T别名， 我们都将Vi增加到T的可能别名集中。这意味着，别名分析之后，我们将得到：

   ```
   may-aliases(T) = { V1, V2, V3, ..., Vn }
   ```

   这意味着每条引用T的语句，将得到Vi标签的n个虚拟操作数。但是， 当启用了别名分组时，我们将T标记为别名标签并将其增加到所有Vi变量的别名集中：

   ```
   may-aliases(V1) = { T }
   may-aliases(V2) = { T }
   ...
   may-aliases(Vn) = { T }
   ```

   这有两个效果：(a)引用T的语句将只得到一个单独的虚拟操作数， (b)所有变量Vi现在将显示为相互别名。所以， 我们为了提高编译时间而失去了别名精确性。但是，理论上， 像这样使用高层次的别名的程序，应该不是把优化放在第一位。

3. 由于变量可以在多个内存标签的别名集中，所以在步骤(2)中所作的分组工作需要被扩展 为针对所有与标签T的可能别名集具有非空交集的内存标签。例如， 如果我们最初具有这些可能别名集：

   ```
   may-aliases(T) = { V1, V2, V3 }
   may-aliases(R) = { V2, V4 }
   ```

   在步骤(2)中，我们将会反转T的别名为：

   ```
   may-aliases(V1) = { T }
   may-aliases(V2) = { T }
   may-aliases(V3) = { T }
   ```

   但是注意现在V2不再与R别名了。我们本应该将R增加到may-aliases(V2)中， 但是我们是在处理别名分组以减少虚拟操作数，所以我们所要做的是将V4增加到分组中， 以获得：

   ```
   may-aliases(V1) = { T }
   may-aliases(V2) = { T }
   may-aliases(V3) = { T }
   may-aliases(V4) = { T }
   ```

4. 如果由于别名产生的虚拟操作数总数依然超出max-alias-vops设置的门槛， 则回到步骤(2)。

14 循环分析和表示

GCC提供了大量的基础结构，用来处理普通的循环结构， 也就是CFG中只有一个入口块的强连通部分。 这章描述了GCC中对于循环的GIMPLE和RTL表示， 以及循环相关的分析的接口（归纳变量分析和迭代次数分析）。

14.1 循环表示

这章描述了GCC中循环的表示，以及可以用来构建，修改和分析这些表示的函数。 大多数接口和数据结构都在`cfgloop.h'中声明。目前， 只是由处理循环的优化过程来分析这些循环结构和更新这些信息， 不过正在做一些努力，使得其在大多数优化过程中都可用。

通常，一个自然的循环会具有一个入口块（header）， 以及可能多个的从循环内部通向header的回边（latch）。 如果多个循环共享单个 header，或者在循环中间有个分支跳转， 则可能会出现带有多个latch的循环。然而GCC中对循环的表示只允许具有单个latch。 在循环分析过程中，为了消除循环结构的歧义，这样的循环的header会被拆分， 并创建前向的块。基于profile信息的heuristic，以及循环中的归纳变量的结构被用来判定latches是否与子循环相关， 还是与单个循环中的控制流相关。这意味着分析有时候会改变CFG， 并且如果你在一个优化过程的中间运行了该分析，则必须能够处理新的块。 可以通过传递LOOPS_MAY_HAVE_MULTIPLE_LATCHES标记来避免CFG改变， 但是要注意，对于具有多个latch边的循环， 大多其它的循环操作函数将无法正确工作（只有查询块成员与循环和子循环关系的，或者枚举和测试循环出口的函数能够工作）。

循环体是由header支配的一组基本块，并且可以通过回边沿着CFG中边的方向达到。 循环使用树的层次结构来组织，直接包含在循环L中的所有循环在树中都为L的子节点。 该树由struct loops结构体表示。该树的根是一个假循环，包含了函数中的所有块。 每个循环都由struct loop结构体表示。 每个循环都被赋予一个索引（struct loop结构体的num域）， 并且指向循环的指针被存在struct loops结构体中的larray向量的对应域里。 索引不必是连续的，larray中可能会有空项（NULL），是由删除循环产生的。 而且不保证索引的数字与循环和子循环有关系。循环的索引不会改变。

不要直接访问larray域中的项。函数get_loop返回给定索引的循环描述。 number_of_loops函数返回函数中的循环数目。要遍历所有的循环， 使用FOR_EACH_LOOP宏。宏的标记参数用来决定遍历的方向和要访问的循环集。 不管循环树是否变化，以及在遍历过程中循环是否被移除，每个循环都保证

只被访问一次。 新创建的循环将不会被访问到，如果需要访问，这必须在它们创建之后单独进行。 FOR_EACH_LOOP宏会分配临时变量，如果使用break或者goto终止了FOR_EACH_LOOP， 它们将不会被释放；因此必须使用FOR_EACH_LOOP_BREAK宏。

每个基本块包含了对其所属的最内层循环的引用（loop_father）。 基于这个原因，对每个CFG只可能有一个struct loops结构体在同一时间被初始化。 全局变量current_loops包含了struct loops结构体。 许多循环操作函数都假设dominance信息是最新的。

通过loop_optimizer_init函数来分析循环。 该函数的参数是一个标记集，使用整数位掩码表示。这些标记指定了循环结构体的其它哪些属性将在之后被计算/赋予，并且保留：

- LOOPS_MAY_HAVE_MULTIPLE_LATCHES: 如果设置了该标记，循环分析将不会改变CFG，特别的，具有多个回边的循环将不会被消除歧义。如果循环具有多个回边， 它的回边块被设为NULL。对于这种形式，大多循环操作函数将无法工作。
- LOOPS_HAVE_PREHEADERS: 创建前驱块的方法为，每个循环只有一个入口边，另外，这个入口边的源块只有一个后继。 这就创建了一个自然的位置，使得代码能够被移出循环， 并且保证循环的入口边由它的直接外循环进来。
- LOOPS_HAVE_SIMPLE_LATCHES: 创建前驱块，从而使得每个循环的回边块只有一个后继。 这就保证了循环的回边不属于任何子循环，并且使得对循环的操作变得非常容易。 许多循环操作函数都假设循环是处于这种形式的。注意使用该标记时， 其中没有任何控制流，且只有一个出口的“正常”循环，将包括两个基本块。
- LOOPS_HAVE_MARKED_IRREDUCIBLE_REGIONS: 强连通组件中的基本块和边，如果不是自然循环（具有多个入口块）， 将被BB_IRREDUCIBLE_LOOP和EDGE_IRREDUCIBLE_LOOP标记。 在这样的不可消减区域中的块和边，如果属于自然循环的， 则不被标记（但是会为进入和离开该区域的入口边和出口边做标记）。
- LOOPS_HAVE_RECORDED_EXITS: 为每个循环记录并更新出口列表。 这使得一些函数（如get_loop_exit_edges）更加有效。 一些函数（如single_exit）只有在出口列表被记录的情况下才能用。

这些属性也可以在之后使用函数create_preheaders, force_single_succ_latches,mark_irreducible_loops 和 record_loop_exits来求得/赋予。

循环结构体占用的内存应该在loop_optimizer_finalize函数中被释放。

CFG操作函数通常不更新循环结构体。在GIMPLE上，如果设置了current_loops， 则cleanup_tree_cfg_loop可以被用来在清除CFG的同时，更新循环结构体。

14.2 循环查询

查询循环信息的函数声明在`cfgloop.h'中。一些信息可以直接从结构体中获得。 每个基本块的loop_father域都包含了它所属的最内层的循环。 最有用的（总是保持更新的）循环结构有：

- header, latch: 循环的Header和latch基本块。
- num_nodes: 循环中的基本块数目（包括子循环的基本块）。
- depth: 该循环在循环树中的深度，也就是，外层循环的数目。
- outer, inner, next: 外部循环，第一个子循环，以及循环树中的下一个循环。

在循环结构体中还有其它一些域，它们有许多是只被一些过程使用， 或者在改变CFG时并没有被更新；总之，它们应该不要去直接访问。

用来查询循环结构体的最重要的函数有：

- flow_loops_dump: 将循环的信息转储到文件中。

- verify_loop_structure: 检查循环结构体的一致性。
- loop_latch_edge: 返回循环的latch边。
- loop_preheader_edge: 如果循环有preheaders, 则返回循环的preheader边。
- flow_loop_nested_p: 测试循环是否为另一个循环的子循环。
- flow_bb_inside_loop_p: 测试基本块是否属于一个循环（包括它的子循环）。
- find_common_loop: 找到两个循环的公共外循环。
- superloop_at_depth: 返回给定深度的外循环。
- tree_num_loop_insns, num_loop_insns: 分别在GIMPLE和RTL之上，评估循环中的insn数目。
- loop_exit_edge_p: 测试是否为循环的出口边。
- mark_loop_exit_edges: 使用EDGE_LOOP_EXIT来标记所有循环的出口边。
- get_loop_body, get_loop_body_in_dom_order, get_loop_body_in_bfs_order: 分别为，在反向的CFG中使用深度优先顺序，dominance顺序，以及宽度优先顺序， 来列举基本块。
- single_exit: 或者返回循环的单一出口边，或者当循环具有多个出口时返回NULL。 只有在使用了LOOPS_HAVE_MARKED_SINGLE_EXITS属性时，才能使用这个函数。
- get_loop_exit_edges: 列举循环的出口边。
- just_once_each_iteration_p: 如果基本块在循环的每次迭代中都只被执行一次，返回true （也就是，其不属于一个子循环，并且dominate循环的latch）。

14.3 循环操作

可以使用下列函数来操作循环树：

- flow_loop_tree_node_add: 向树中增加一个节点。
- flow_loop_tree_node_remove: 从树中移除一个节点。
- add_bb_to_loop: 向循环增加一个基本块。
- remove_bb_from_loops: 从循环中移除一个基本块。

大多数低级别的CFG函数会自动更新循环。下列函数用来处理一些比较复杂的CFG操作情况：

- remove_path: 移除一个边，以及它支配的所有块。
- split_loop_exit_edge: 拆分循环的出口边，以确保PHI节点参数保留在循环中（确保使用了循环封闭SSA形式）。 只用于GIMPLE。

最后，有一些高层的循环转换被实现。虽然其中一些能够工作于非最内层循环， 但是大多数并没有对这种情况进行过测试，目前，它们只对最内层循环可靠：

- create_iv: 创建一个新的归约变量。只工作在GIMPLE上。 standard_iv_increment_position能够被用来为iv增量找到合适的位置。
- duplicate_loop_to_header_edge, tree_duplicate_loop_to_header_edge: 这些函数（在RTL和GIMPLE之上）在进入循环头的边上对循环体复制规定的次数， 从而执行了循环展开或loop peeling。对于复制的循环， can_duplicate_loop_p（GIMPLE之上为can_unroll_loop_p）必须为真。
- loop_version, tree_ssa_loop_version: 这些函数创建一个循环的复本，以及在它们之前的一个分支跳转， 从而可以根据规定的条件来选择它们。这对需要在运行时检验一些假设的优化很有帮助（循环的一个复本通常不做变换，而另一个通过某种方式进行转换）。
- tree_unroll_loop: 展开循环，包括剥离额外的迭代，从而使得迭代次数可以由展开因子划分，更新出口条件，以及移除不会经过的出口。只在GIMPLE上工作。

14.4 循环封闭的SSA形式

在整个树级的循环优化过程中，SSA形式需要有一个额外的加强条件： 没有SSA名字在它所定义的循环之外被使用。 满足这样条件的SSA形式称作“循环封闭的SSA形式”-LCSSA。 为了形成LCSSA，对于在循环外使用SSA名字的，必须在循环的出口创建PHI节点。 为了节省内存，在LCSSA中只支持实际的操作数（不包括虚拟SSA名）。

LCSSA有许多好处：

- 许多优化（值范围分析，最终值替换）都对在循环中定义的而在循环外使用的值感兴趣， 即，对那些我们创建的新PHI节点。
- 在归约变量分析中， 没有必要指定将要执行分析的循环——标量演化分析总是返回SSA名字被定义的循环。
- 它使得循环转换中更新SSA形式变得更简单。没有LCSSA的话， 像循环展开这样的操作可能会强迫创建距离循环任意远的PHI节点，然而使用LCSSA的话， SSA形式能够被局部更新。不过，由于我们只是在LCSSA中保持真实的操作书， 所以我们不能使用这个好处（我们本来能够局部更新真实操作数， 但是这样并不比使用通用SSA形式更新更有效；对SSA的改动是相同的）。

不管怎样，这还意味着LCSSA必须被更新。这通常是很直白的， 除非你在循环中创建了一个新值，并在外面使用， 或者除非你操作了循环出口边（有函数被提供使得这些操作变得简单）。 `rewrite_into_loop_closed_ssa`用来将SSA形式重写为 LCSSA， `verify_loop_closed_ssa`用来检查LCSSA所保持的不变性。

14.5 标量演化

标量演化（SCEV）用来表示在GIMPLE之上的归纳变量分析结果。 它使得我们能够通过简单一致的方式来表示具有复杂性为的变量（我们只使用它来表示多项 式归纳变量的值，但是是可以进一步扩展的）。 SCEV分析的接口声明在`tree-scalar-evolution.h'中。要使用标量演化分析， 则必须使用`scev_initialize`。要停止使用SCEV，则使用`scev_finalize`。 为了节省时间和内存，SCEV分析会缓存结果。但是这些缓存会被大多数循环转换变为无效， 包括代码移除。如果执行了这样的转换，则必须调用`scev_reset`来清除缓存。

给定一个SSA名字， 能够使用`analyze_scalar_evolution`函数来分析它在循环中的行为。 然而返回的SCEV不需要被完全分析， 并且它可以包含对其它定义在循环中的SSA名字的引用。 必须使用`instantiate_parameters`或者`resolve_mixers`函数来解决这些（潜在的递归）引用。当你将SCEV的结果只用于某种分析时， 并且一次工作于整个循环嵌套时，`instantiate_parameters`会很有用。 它将尝试替换所有的SSA名字，用它们在所有循环中的SCEV，包括当前循环的外层循环， 因此提供了在循环嵌套中的变量行为的完全信息。当你一次只工作于一个循环， 并且可能需要根据归约变量的值来创建代码时，`resolve_mixers`会很有用。 它会只解决定义在当前循环中的SSA名字，而保留外面定义的SSA名字不变， 即使它们在外循环中的演化是已知的。

SCEV是一个标准的树表达式，除去实际上它可以包含多个特定的树节点。 `SCEV_NOT_KNOWN`为其中之一，用于值无法被表示的SSA名字。 另一个是`POLYNOMIAL_CHREC`。多项式chrec有三个参数——base，step和loop（base和step都可以进一步包含多项式chrecs）。表达式，base和step的类型必须相同。 在下面的例子中，如果变量（在特定的循环中）等于`x_1`， 则具有演化`POLYNOMIAL_CHREC(base, step, loop)`。

```
while (...)
 {
  x_1 = phi (base, x_2);
  x_2 = x_1 + step;
 }
```

注意这包括操作数上的语言限制。例如，如果我们编译C代码， 并且x具有有符号类型，那么加法溢出将会产生未定义行为， 并且我们可以假设这并没有发生。因此，SCEV的值不能溢出。

许多情况下，只是想放射归约变量。这时，额外的SCEV的表达式幂就没有用处， 并且可能会使优化变得复杂。这时， 可以使用simple_iv函数来分析一个值——结果为循环不变量base和step。

14.6 RTL上的IV分析

RTL之上的归约变量很简单，并且只允许一次在一个循环中，仿射归约变量的分析。 接口在`cfgloop.h'中声明。在循环L中分析归约变量之前， 必须在L上调用iv_analysis_loop_init函数。分析完成之后（可能会为多个循环调用iv_analysis_loop_init）， 应该调用iv_analysis_done。下面的函数能够被用来访问分析结果：

- iv_analyze: 分析在给定insn中的单个寄存器。如果在该insn中没有使用寄存器，则扫描下一个insn， 因此该函数能够在通过get_condition返回的insn上被调用。
- iv_analyze_result: 分析给定insn的赋值结果。
- iv_analyze_expr: 分析一个更加复杂的表达式。其所有操作数都通过iv_analyze来分析， 因此它们必须用在特定insn中，或者后面insn的其中之一。

归约变量的描述在struct rtx_iv中。为了处理子寄存器，该表示有些复杂； 如果extend域的值不是UNKNOWN，则归约变量在第i次迭代时的值为

```
delta + mult * extend_{extend_mode} (subreg_{mode} (base + i * step)),
```

不过下面的情况例外：如果first_special为真， 则在首次迭代时（当i为0时），值为delta + mult * base。 然而，如果extend等于UNKNOWN，则first_special必须为假， delta为0，mult为1，并且在第i次迭代时的值为

```
subreg_{mode} (base + i * step)
```

函数get_iv_value可以用来执行这些计算。

14.7 迭代次数分析

在GIMPLE和RTL之上，都有函数可以用来判定循环的迭代次数，并且具有相似的接口。 在GCC中，循环的迭代次数被定义为循环latch的执行次数。许多情况下， 是不可能无条件的判定出迭代次数——判定的迭代次数只有在满足一些假设时才正确。 分析尝试使用包含在程序中的信息来检验这些条件；如果失败了， 则条件和结果一起被返回。下面的信息和条件由分析提供：

- assumptions: 如果条件为假，则其余的信息无效。
- RTL上的noloop_assumptions, GIMPLE上的may_be_zero: 如果该条件为真，则循环在第一次迭代中便退出。
- infinite: If this condition is true, the loop is infinite. This condition is only available on RTL. On GIMPLE, conditions for finiteness of the loop are included in assumptions.
- RTL上的niter_expr , GIMPLE上的niter: 该表达式给出迭代次数。迭代次数被定义为循环latch的执行次数。

在GIMPLE和RTL之上，都需要初始化归约变量分析框架（GIMPLE上为SCEV， RTL上为loop-iv）。在GIMPLE上，结果存储在struct tree_niter_desc中。 可以通过number_of_iterations_exit函数来判定循环在通过给定出口退出之前的 迭代次数。在RTL上，结果返回在struct niter_desc结构体中， 相应的函数名为check_simple_exit。还有一些函数遍历循环的所有出口， 并尝试找到容易判定迭代次数的一个——它们是GIMPLE上的find_loop_niter和 RTL上的find_simple_exit。最后，还有一些函数提供相同的信息， 只不过还额外的对其进行缓存， 使得反复调用迭代次数的代价不会很高——它们是GIMPLE上的 number_of_latch_executions和RTL上的get_simple_loop_desc。

注意这些函数中的一些可能与其它的在行为上有些不同——有些只返回迭代次数的表达式。 函数number_of_latch_executions只工作于单个出口的循环。 函数number_of_cond_exit_executions能被用来判定一个单出口循环在退出条件下 的执行次数。（即，number_of_latch_executions加1）。

14.8 数据依赖分析

用于数据相关性分析的代码可以在`tree-data-ref.c'中找到， 接口和数据结构在`tree-data-ref.h'中描述。 用来计算对于给定循环的所有数组和指针引用的数据相关性的函数为compute_data_dependences_for_loop。 该函数目前被用于线性循环转换和向量化过程。在调用该函数之前，必须分配两个向量： 第一个向量将会包含在被分析的循环体中的数据引用集， 第二个将会包含数据引用之间的依赖关系。因此，如果数据引用向量的大小为n， 则包含依赖关系的向量将包含n*n个元素。但是，如果被分析的循环包含副作用， 例如对数据引用有潜在干扰的调用，则分析会在扫描循环体中的数据引用时停住， 并在依赖关系数组中插入一个chrec_dont_know。

数据相关性是在扫描循环体时，按照特定顺序发现的：循环体按照执行顺序分析， 每条语句的数据引用被压入数据引用数组的尾部。两个数据相关性在程序中的语法位置， 和在数据相关性数组具有相同的顺序。这种语法顺序在一些经典数据相关性测试中很重要， 并且将这中顺序映射到数组的元素可以避免对循环体表示的昂贵查询开销。

目前处理了三种类型的数据相关性：ARRAY_REF, INDIRECT_REF和COMPONENT_REF。 数据相关性的数据结构体为data_reference， 其中data_reference_p为指向数据相关性结构体的指针名。结构体包含了以下元素：

- base_object_info: 提供了关于数据引用的基本对象的信息， 以及访问函数代表了数据引用在循环中相对于它的基的演变。这些访问函数。例如， 对于一个引用a.b[i][j]，基本对象为a.b，其中一个针对每个数组下标的访问函数为： {i_init, +i_step}_1, {j_init, +, j_step}_2。
- first_location_in_loop: 提供了循环中由数据引用访问的第一个位置， 以及用来表示相对于该位置的演化的访问函数。该数据用来支持指针， 而不是数组（具有基对象的）。 指针访问被表示为从循环中的第一个位置开始的一维访问。例如:

```
for1 i
  for2 j
   *((int *)p + i + j) = a[i][j];
```

  对于p + i，指针访问的访问函数是{0, +4B}_for2。 对于a，数组的访问函数是{i_init, +i_step}_for1和 {j_init, +, j_step}_for2。

  通常，指针指向的对象或者是不可知的， 或者是我们不能证明访问被限制在这些对象的边界中。

  两个数据引用只有在最起码有一个表示的所有域都适合于这两个数据引用时， 才能够进行比较。

  目前测试数据相关性的策略为：如果a和b都由数组来表示， 则比较a.base_object和b.base_object；如果它们相等， 则应用相关性测试（使用基于base_objects的访问函数）。如果a和b都由指针表示， 则比较a.first_location和b.first_location； 如果它们相等，则应用相关性测试（使用基于第一位置的访问函数）。 但是，如果a和b的表示不同，只能尝试去证明它们的基肯定不相同。
- 别名信息。
- 对齐信息。

描述两个数据引用之间关系的结构体是data_dependence_relation， 指向这种结构体的指针的简短名字为ddr_p。该结构体包含：

- 每个数据引用的指针，
- 一个树节点are_dependent，如果分析证明了两个数据引用之间没有相关性， 则设置为chrec_known；如果分析不能判定任何有用的结果， 并且这些数据引用可能存在相关性，则设为

chrec_dont_know； 如果数据引用间存在相关性，则设置为NULL_TREE， 并且该相关性的描述在dir_vects和dist_vects数组的下标中给出。

- 一个布尔值，用来判定依赖关系是否能被表示为经典的距离向量，
- 一个数组subscripts，包含了数据引用的每个下标的描述。给出两个数组访问， 下标为对于给定维数的访问组合。例如， 给定A[f1][f2][f3]和B[g1][g2][g3]， 则有三个下标：(f1, g1), (f2, g2), (f3, g3)。
- 两个数组dir_vects和dist_vects， 包含了使用方向和距离依赖向量的数据相关典型表示。
- 循环数组loop_nest，包含了距离和方向向量指向的循环。

有一些函数可以很好的打印由数据相关分析抽出的信息： dump_ddrs打印最详尽的数据依赖关系数组， dump_dist_dir_vectors只打印数据依赖关系数组的典型的距离和方向向量， dump_data_references打印数据引用数组中的详细的数据引用。

14.9 线性循环转换框架

Lambda是一个允许循环转换使用基于迭代空间和循环边界的非退化矩阵的框架。 这允许组合使用skewing，scaling，interchange，和reversal转换。 这些转换常用来提高cache行为， 或者移除内部循环依赖使得可以进行并行化或者向量化。

为了执行这些转换，Lambda需要loopnest转化为可以被很容易的进行矩阵转换的内部形式。 函数gcc_loopnest_to_lambda_loopnest用来做这种转换。 如果循环不能够使用lambda转换，这个函数将返回NULL。

一旦通过转换函数获得lambda_loopnest， 便可以使用lambda_loopnest_transform来进行各种转换处理， 其接受一个转换矩阵。 注意这是由调用者来检验转换矩阵是否合法，即可以应用到循环上的。 Lambda只是简单的应用提供给它的矩阵。可以扩展为使用任何非退化矩阵之外的矩阵， 不过现在还没有实现。 可以使用lambda_transform_legal_p来检验给定loopnest的矩阵的合法性。

给定一个转换过的loopnest， 可以通过lambda_loopnest_to_gcc_loopnest来将其转化回到gcc IR。 该函数会修改循环使得它们匹配转换过的loopnest。

14.10 Omega 一种对线性规划问题的求解

数据相关性分析包含多个求解器，从不太复杂的到比较复杂的。 为了确保这些求解器的结果的一致性， 实现了一个基于不同求解器的数据相关性检查过程。 已经被集成到GCC中的第二种方法是基于Omega相关求解器， 由William Pugh和David Wonnacott在1990年编写。 数据相关性测试能够通过使用Presburger算法的子集来公式化， 从而可以转化为线性约束系统。然后这些线性约束系统能够使用Omega求解器求解。

Omega求解器使用Fourier-Motzkin算法进行变量消除： 一个包含n个变量的线性约束系统被消减为包含n-1个变量的线性约束系统。 Omega求解器还能够用来解决其它能被表示为线性等式和不等式系统形式的问题。 Omega求解器有一个公认的指数最坏情况，即文献上称之的“omega 恶梦”， 不过实际上，众所周知omega测试对于公用数据相关性测试是有效的。

Omega求解器所使用的描述线性规划问题接口在`omega.h'中， 求解器为omega_solve_problem。

15 控制流图

控制流图（CFG）是一个建立在中间代码（RTL或者tree指令流）之上的数据结构， 对正在编译的函数的控制流行为的抽象。CFG是一个有向图，顶点表示基本块， 边表示从一个基本块到另一个的控制流的可能转换。 用来表示控制流图的数据结构定义在`basic-block.h'中。

15.1 基本块

基本块是一段直线性的代码序列，并且只有一个入口和一个出口。 在GCC中，基本块使用basic_block数据类型来表示。

结构体basic_block的两个指针成员，指针next_bb和prev_bb， 用来维持与底层指令流顺序相同的基本块双向链表。 基本块之间的链，由给定的操作CFG的API，通过透明的方式进行更新。 宏FOR_EACH_BB可以用来按照词典顺序（lexicographical order） 访问所有基本块。 也可以使用walk_dominator_tree，进行支配遍历（dominator traversal）。 给定两个基本块A和B，如果A总是在B之前被执行， 则基本块A支配（dominate）基本块B。

BASIC_BLOCK数组包含了所有的基本块，并且顺序不固定。 每一个basic_block结构体都有一个域， 用来保留一个唯一的整数标识符index， 作为该基本块在BASIC_BLOCK数组中的索引。 函数中基本块的总数为n_basic_blocks。 由于中间过程（passes）可以重排，创建，复制和销毁基本块， 所以基本块的索引和总数在编译过程中都可能改变。 任何基本块块的索引都不应该比last_basic_block的大。

有专门的基本块来表示一个函数的可能的入口和出口。 这些基本块被称作ENTRY_BLOCK_PRT和EXIT_BLOCK_PTR。 这些基本块不包含任何代码，并且不是BASIC_BLOCK数组的成员。 因此它们被赋予了唯一的负数索引。

每个basic_block还包含了指针，用来指向基本块中第一条指令（head）和最后一条指令（tail）， 或者在基本块中包含的指令流的结尾（end）。 实际上，由于basic_block数据类型在GCC的两个主要中间表示（tree和RTL）中都被用来表示基本块， 因此具有针对这两种表示的指针，用来指向基本块的头和尾。

对于RTL，这些指针是rtx head, end。 在RTL函数表示中，头指针总是指向NOTE_INSN_BASIC_BLOCK或者CODE_LABEL。 在RTL函数表示中，指令流不仅包含“真正”的指令， 而且还有注解（notes）。 任何移动或者复制基本块的函数都需要注意更新这些注解。 许多这些注解都期望指令流是由线性区域组成的，所以这使得更新比较困难。 NOTE_INSN_BASIC_BLOCK注解是唯一类型的， 可以出现在基本块内包含的指令流中。 一个基本块的指令流总是跟随一个NOTE_INSN_BASIC_BLOCK， 但是基本块注解之前可以有0个或多个CODE_LABEL节点。 基本块结束于一条控制流指令， 或者后面是紧随CODE_LABEL或者NOTE_INSN_BASIC_BLOCK的最后一条指令。 CODE_LABEL不能出现在基本块中的指令流里。

除了注解之外，跳转表向量也被表示为insn流中的“伪指令”。 这些向量从不出现在基本块中，并应该总是被放在引用它们的 表跳转指令（tabel jump instructions）的后面。 在移除table-jump之后，通常很难消除计算地址和引用向量的代码， 所以对这些向量的清除工作被推迟到活跃分析之后。 这样，跳转表向量可能会在insn流中出现，但未被引用，没有任何用图。 在将任何边（edge）作为fall-thru之前， 都需要调用can_fallthru函数来检查这种构造方式是否可以。

对于tree的表示，基本块的头和尾由stmt_list域指向， 但是，决不要直接引用这些特定的tree。 替代的，在树级别上， 使用抽象容器和迭代器来访问基本块中的语句和表达式。 这些迭代器被称作块语句迭代器（block statement iterators, BSI）。 可以在各种`tree-*'文件中使用grep来查找^bsi。 下面的片段可以打印（pretty-print）使用GIMPLE表示的程序的所有语句。

```
FOR_EACH_BB (bb)
 {
  block_stmt_iterator si;

  for (si = bsi_start (bb); !bsi_end_p (si); bsi_next (&si))
   {
    tree stmt = bsi_stmt (si);
    print_generic_stmt (stderr, stmt, 0);
   }
 }
```

15.2 边

边表示从某个基本块A的结束到另一个基本块B的开头的可能的控制流转换。 我们称A是B的前驱，B是A的后继。在GCC中，边由edge数据类型表示。 每个edge作为两个基本块之间的链接： 一个edge的src成员指向前驱dest基本块。 数据类型basic_block的成员preds和succs， 指向块的前驱和后继们的边的type-safe向量。

当在一个边向量中访问边时，应该使用边迭代器。 边迭代器由edge_iterator数据结构和一些可以使用的操作方法构成：

ei_start 该函数初始化一个指向边向量中第一个边的edge_iterator。

ei_last 该函数初始化一个指向边向量中最后一个边的edge_iterator。

ei_end_p 如果edge_iterator表示边向量中的最后一个边，则该断言为true。

ei_one_before_end_p
如果edge_iterator表示边向量中的倒数第二个边，则该断言为true。

ei_next 该函数接受一个指向edge_iterator的指针，并使其指向序列中的下一个边。

ei_prev 该函数接受一个指向edge_iterator的指针，并使其指向序列中的上一个边。

ei_edge 该函数返回由edge_iterator当前指向的edge。

ei_safe_safe
该函数返回由edge_iterator当前指向的edge， 但是如果迭代器指向序列的结尾时，则返回NULL。 该函数是为现有的代码提供的，即代码假设用NULL边来表示序列的结尾。

宏FOR_EACH_EDGE可以方便的用来访问前驱边或后继边序列。 当在遍历中会移除元素时，不要使用该宏，否则会错过这些元素。 这里有一个如何使用该宏的例子：

```
edge e;
edge_iterator ei;

FOR_EACH_EDGE (e, ei, bb->succs)
 {
  if (e->flags & EDGE_FALLTHRU)
   break;
 }
```

有许多原因会导致控制流从一个块传递到另一个。 一种可能是某条指令，例如CODE_LABEL，在一个线形的指令流中， 总是起始一个新基本块。在这种情况下， 一个fall-thru边将基本块与随后的第一个比本块相连。 但是有许多其它原因会导致边被创建。 edge的数据类型的flags域用于存储我们处理的边的类型信息。 每个边都具有下列类型之一：

jump 与跳转指令相关的边没有被设置类型标识。这些边用于无条件或有条件跳转， 以及RTL中还有表跳转。它们是最容易操作的，因为当流图不为SSA形式的时候， 可以自由重定向。

fall-thru Fall-thru边存在于当基本块不需要分支而是继续执行随后的块的时候。 这些边的标志设为EDGE_FALLTHRU。不像其它类型的边， 这些边必须直接进入基本块的指令流中。 函数force_nonfallthru可以用于在需要重定向时插入一个无条件跳转。 注意这可能需要创建一个新基本块。

exception handling
异常处理边表示可能的控制转移，从一个陷门指令到一个异常处理器。 关于“trapping”定义不尽相同。在C++中，只有函数调用能够抛出异常， 但是对于Java，像

除0或者段错误都被定义为异常，并且因此每条指令都可能抛出这种需要处理的异常。异常边设置了EDGE_ABNORMAL和EDGE_EH标识。

当更新指令流时，能够容易的将可能trapping的指令转换成non-traaping，通过简单的将异常边移除。相反的转换比较困难，但是是不会发生的。可以通过调用purge_dead_edges来消除边。

在RTL表示中，异常边的目的地由附加在insn上的注解REG_EH_REGION来指定。在trapping调用的情况下，还设置了EDGE_ABNORMAL_CALL标识。在tree表示中，该额外的标识没有被设置。

在RTL表示中，断言may_trap_p可以用来检测指令是否还可能trap。对于tree表示，可以用tree_could_trap_p，不过该断言只检测可能的内存trap，像在废除一个无效的指针地址。

sibling calls 兄弟调用或者尾调用以非标准的方式终止函数，并且因此必须存在一个引向出口的边。EDGE_SIBCALL和EDGE_ABNORMAL在这种情况下被设置。这些边只存在于RTL表示中。

computed jumps

计算跳转包含了引向函数中代码引用的所有标号的边。所有这些边都设置了EDGE_ABNORMAL标识。用来表示计算跳转的边通常会造成编译时间性能问题，因为函数有许多标号组成，许多计算跳转可能具有密集的流图，所以这些边需要特别仔细的处理。在编译过程的早期阶段，GCC尝试避免这样的密集流图，通过因子化计算跳转。例如，给定下列跳转，

```
goto *x;
[...]

goto *x;
[...]

goto *x;
[...]
```

将计算跳转提取公因子，会产生具有比较简单流图的代码序列：

```
goto y;
[...]

goto y;
[...]

goto y;
[...]

y:
 goto *x;
```

但是，这种转换的典型问题是产生的结果代码具有运行时代价：一个额外的跳转。因此计算跳转在编译器之后的过程里被un-factored。当你工作于这些过程上时，需要注意。曾有许多已存的例子，即对未公因子化的计算跳转编译时造成的头痛之事。

nonlocal goto handlers

GCC允许嵌套函数使用goto到一个通过参数传给被调用者的标号的方式来返回到调用者那里。传给嵌套函数的标号包含了特定的代码用来在函数调用之后进行清理工作。这段代码被称为“nonlocal goto receivers”。如果一个函数包含这样的非局部goto接受者，一个从调用到标号的边被创建，并设置了EDGE_ABNORMAL和EDGE_ABNORMAL_CALL标识。

function entry points
: 根据定义，函数执行起始于基本块0， 所以总有一个边从ENTRY_BLOCK_PTR到基本块0。 目前，对备用入口点没有tree表示。 在RTL里， 备用入口点通过定义了LABEL_ALTERNATE_NAME的CODE_LABEL指定。 这能够被后端用于为通过不同上下文调用函数而生成备用prologues。 将来，Fortran90定义的多入口函数的完全支持需要被实现。

function exits
: 在pre-reload表示中，函数终止于insn链中的最后一条指令， 并且没有显示的返回指令。这对应于由fall-thru引向出口块。 reload之后，最佳的RTL epilogues被用于显示的（有条件的）返回指令中。

15.3 Profile信息

在许多情况下，编译器必须对是否由一块代码的速度来换取另一块的速度， 或者由代码的大小来换取速度，来作出选择。这种情况下， 知道给定块将会被执行几次这样的信息会很有帮助。 这就是在流程图中维护profile的目的。 GCC能够处理通过profile feedback获得的profile信息， 但也能够根据统计和启发来估计分支跳转的可能性。

基于反馈的profile是通过编译测量程序来产生的，在训练运行中执行， 并且在重新编译程序产生最终可执行程序时，读取基本块和边的执行数目。 该方法使得程序花费大量的时间在训练运行上，从而提供了非常精确的信息。 信息是否匹配平均运行取决于选择的训练数据集， 但是个别研究表现程序的行为通常会由于稍微不同的数据集就会变化。

当profile反馈不可用时，编译器可以被请求尝试使用heuristics集（详情参见`predict.def'）来进行预测程序中每个分支的行为， 并且通过在图中传播可能性来计算每个基本块的评估频率。

每个basic_block包含两个整数域来表示profile信息： frequency和count。 frequency是对函数中的基本块每隔多久被执行的评估。 其被表示为一个整数标量，范围从0到BB_FREQ_BASE。 函数中执行频率最高的基本块被初始化为BB_FREQ_BASE， 其余的frequency相应的进行刻画。优化过程中，执行频率最高的基本块的frequency能够减少（例如由循环展开造成的）或增加（例如由交叉跳转优化造成的），所以有时需要执行多次度量。

count包含了硬计数的执行数目，在训练运行中测算出的， 并且只有profile反馈可用时为非0。该值被表示为主机的宽整数（一般为64位整数），特定类型gcov_type。

大多数优化过程只能使用基本块的frequency信息， 但是一些过程可能想知道硬执行次数。在度量之后，频率应该总是匹配计数， 但是在更新profile信息的过程中，数值误差可能会积累到十分大的错误。

每个边还包含一个分支可能性域： 一个范围从0到REG_BR_PROB_BASE的整数。 其表示将控制从src基本块传递到dest基本块的可能性， 即控制流向该边的可能性。 EDGE_FREQUENCY宏可用于计算给定边会被接受的频率。 同时每个边还有一个count域，用来表示与基本块相同的信息。

基本块频率不在指令流中表示，但是在RTL表示中， 边频率用来表示条件跳转（通过REG_BR_PROB宏）， 因为它们用在将指令输出到汇编文件中的时候，并且流图不在被维护。

控制流通过给定边到达目的基本块的可能性被称作反向可能性， 并且没有直接表示，但是可以容易的从基本块的频率中计算获得。

不幸的是，更新profile信息是一个精致的任务， 这使得很难集成到CFG操作API中。许多修改CFG的函数和钩子， 像redirect_edge_and_branch， 都不具有足够的信息来容易的修改profile， 所以更新多半情况是留给调用者的。很难找到profile更新代码中的bug， 因为它们只是体现在产生了更糟的代码， 并且检测profile一致性是不可能的，因为数值误差积累。 因此在每个修改CFG的过程中，应该特别注意这个问题。

必须指出REG_BR_PROB_BASE和BB_FREQ_BASE被设为足够低， 才有可能在流图中计算任何频率或可能性作为指数的2的幂运算。

15.4 维护CFG

每个编译器过程都具有的一个重要任务是保持控制流图和所有profile信息更新。 在每个过程之后都重建控制流图是不可能的，因为这样代价会很高， 而且丢失的profile信息是根本无法重建的。

GCC有两个主要的中间表示， 并且它们都使用basic_block和edge数据类型来表示控制流。 两种表示都尽可能多的共享CFG维护的代码。对于每一种表示， 都定义了一套hooks，以便于需要的时候可以提供自己的CFG维护函数的实现。 这些钩子定义在`cfghooks.h'中。这些钩子提供了几乎所有普通的CFG操作， 包括块分割和合并，边重定向，以及创建和删除基本块。 这些钩子应该提供所有需要的维护和操作RTL和tree表示下的CFG。

目前，基本块的边界在修改指令时会被透明的维护，因此很少需要手动移动它们（比如当有人想要显式的输出基本块外面的指令的时候）。 将CFG看作指令链的组成部分，比看作建立在之上的结构，往往要更好些。 但是原则上，对于树表示的控制流图并不是数表示的必须部分。 函数树可以在不需要首先创建树表示的流图的情况下就被扩展。 这种情况在没有进行任何树优化的编译时会发生。当进行树优化时， 并且指令流被重写为SSA形式，CFG就和指令流非常紧密的联系起来了。 特别在语句插入和移除时要注意。实际上，如果没有同时对CFG进行恰当的维护， 整个树表示就很难使用和维护。

在RTL表示里，每条指令有一个BLOCK_FOR_INSN值用来表示指向包含该指令的基本块。 在tree表示里，函数bb_for_stmt返回一个指向包含所查询语句的基本块。

在tree表示里，当需要对函数进行改动时， 应该使用块语句迭代器（block statement iterators）。 这些迭代器提供了流程图和指令流的整体抽象。 块语句迭代器由block_stmt_iterator数据结构和一些修改函数构成， 包括下面的：

bsi_start 该函数初始化一个block_stmt_iterator，使其指向基本块中第一条非空语句。

bsi_last 该函数初始化一个block_stmt_iterator，使其指向基本块中最后一条语句。

bsi_end_p 如果block_stmt_iterator表示基本块的结束，则为true。

bsi_next 该函数接受一个block_stmt_iterator，并使其指向它的后继。

bsi_prev 该函数接受一个block_stmt_iterator，并使其指向它的前驱。

bsi_insert_after
该函数在block_stmt_iterator所在位置之后插入一条语句。 最后一个参数决定是否将语句迭代器更新指向新插入的语句， 还是保留指向原来的语句。

bsi_insert_before
该函数在block_stmt_iterator所在位置之前插入一条语句。 最后一个参数决定是否将语句迭代器更新指向新插入的语句， 还是保留指向原来的语句。

bsi_remove 该函数移除block_stmt_iterator所在位置的语句， 并且如果基本块中还有语句，则将剩余的语句重新链接。

在RTL表示里，宏BB_HEAD和BB_END可以用来获得基本块的 起始rtx和结束rtx。没有抽象迭代器被定义用来遍历insn链， 不过可以使用NEXT_INSN和PREV_INSN替代。参见 Section 10.18 [Insns], page 139。

通常一个代码操作过程将会简化指令流和控制流，也可能消除一些边。 例如当一个条件跳转被替换为非条件跳转，甚至在编译java时， 将可能的trapping指令简化为non-trapping。边的更新是不

透明的， 每个优化过程都要求手动进行。不过，实际中这种情况很少发生。如果存在的话， 过程可以针对给定的基本块调用purge_dead_edges来移除多余的边。

另一个常见的情景是分支指令的重定向。 不过由于可以非常好的建模为控制流图里的边重定向，因此应尽量使用redirect_edge_and_branch，而不是其它底层函数， 例如只是操作RTL链的redirect_jump。 定义在`cfghooks.h'中的CFG钩子应该提供了操作和维护CFG所需要的全部API。

有时候，一个过程可能不得不要向基本块的中间插入控制流指令，这样的话， 就在基本块中间产生一个入口点。根据定义，这是不可能的， 因此必须要将块分开以确保只含有一个入口点，也就是基本块的头。 当基本块中间的指令必须成为跳转或分支指令的目标时， 可以使用CFG钩子split_block。

对一个全局优化，一个常用的操作是在流图中将边拆分，并插入指令。 在RTL表示里，可以很容易的实现， 通过使用insert_insn_on_edge函数来生成一条暂存的“on the edge”指令， 以便之后的commit_edge_insertions调用来将插入的指令从边上移到基本块的指令流里。 如果需要的话，还会生成新的基本块。在tree表示里， 等价的函数为bsi_insert_on_edge，用来在边上插入一个块语句迭代器， 以及bsi_commit_edge_inserts，将指令挪到实际的指令流里。

在调试优化过程时， 函数verify_flow_info可能有助于发现在控制流图的更新代码中的bug。

注意，目前在由树的表示扩展到RTL时，控制流的表示会被丢弃。 长远的看，CFG应给被维持并随着函数树本身被扩展到RTL表示。

15.5 活跃信息

活跃信息有助于决定在程序的给定点是否一些寄存器是“活跃”的， 即其包含的值可能在程序之后的地方被使用。 例如，这些信息被使用在寄存器分配过程， 伪寄存器只有在活跃的时候才需要被分配给唯一的硬件寄存器或者栈存储单元。 当一个寄存器无用的时候，硬件寄存器和栈存储单元可以被随意重用于其它值。

在后端从pass_df_initialize起始到pass_df_finish结束之间， 活跃信息是有效的。有三种活跃分析：LR，能够确定在函数的任意点P， 寄存器是否会在从P到函数结束之间的某处被使用。 UR，能够确定是否从函数的起始到P之间定义了变量。 LIVE是LR和UR的交集，变量在P点是活跃的， 如果同时从函数的开始到现在存在一个赋值， 并且从P到函数的结束之间存在对其的使用。

通常这三种信息里，LIVE最有帮助。 宏DF_[LR,UR,LIVE]_[IN,OUT]可以用来访问这些信息。 这些宏接受一个基本块号，并返回一个以寄存器号为索引的位图。 该信息只保证截至在调用df_analyze之后是最新的。 关于使用数据流的详细信息参见`df-core.c'文件。

活跃信息部分存在RTL指令流里，部分存在流程图里。局部信息存在指令流中： 每条指令可以包含REG_DEAD注解（note）， 来表示给定寄存器的值已经不再被需要了，或者REG_UNUSED注解， 来表示由指令计算所得的值从来没有被使用。第二个可以有助于指令一次计算多值。

16 机器描述

机器描述包括两个部分：指令模式文件（`.md' file）和宏定义C头文件。

目标机器的`.md' 文件包含了目标机器支持的（或者最起码值得告诉编译器的）每条指令的模式。还可能包含注释。分号使得一行的剩余部分为注释，引号标注的字符串中的分号除外。

关于C头文件的信息，参见下一章。

16.1 概述机器描述是如何被使用的

编译器中有三个主要的转换：

1. 前端读取源代码并建立解析树。
2. 基于命名的指令模式，解析树被用来生成RTL insn列表。
3. insn列表被用来匹配RTL模板，产生汇编代码。

生成过程，只与insn的名字有关系，包括命名的 `define_insn` 或者 `define_expand`。编译器会选择恰当名字的模式，并且根据这章后面的文档来使用 操作数，而不需要关心RTL模板或者操作数constraint。注意，编译器所寻找的名字是被硬编码 进编译器中的——它将忽略未命名的模式和名字无法识别的模式，但是，如果你没有提供所 需要的命名模式，它将异常中断（abort）。

如果使用了 `define_insn` ，所给出的模版将会被插入到insn列表中。如果使用了 `define_expand` ，将会发生三种情况之一，取决于条件逻辑。条件逻辑可以手动为 insn列表创建一个新的insn，并且调用 `DONE`。对于某些命名模式，它可以调用 `FAIL` 来告诉编译器使用一种备用方式完成任务。如果既没有调用DONE也没有调用 `FAIL`，在模式中所给出的模版将会被插入，就像 `define_insn`一样。

一旦生成insn列表，各种优化过程便在insn列表中转换，替代和重排insn。例如， `define_split` 和 `define_peephole` 模式便在这里被使用。

最后，insn列表的RTL被用来匹配 `define_insn` 模式中的RTL模版，并且那些模式 被用来生成最终的汇编代码。这时，由于不需要关心名字，所以每个命名的 `define_insn`跟没有命名没有区别。

16.2 指令模式的方方面面

每个指令模式包含了一个不完全的RTL表达式，和之后要被填充的部分；操作数constraint，用来 限制如何填充那些部分；以及一个输出模式或者C代码来生成汇编输出。所有这些都由一个 `define_insn` 表达式包裹起来。

`define_insn` 是一个RTL表达式，包含了四或五个操作数：

1. 一个可选的名字。存在名字表明该指令模式能够为编译器的RTL生成过程，执行一个确定的 标准工作。这个过程知道确定的名字，并且如果在机器描述中定义了这些名字，则会使用 它们的指令模式。

   空字符串表示不存在名字。没有命名的指令模式是不会被用来生成RTL代码的，但是之后它 们能够用来组合多个简单insn。

   不识别的并且因此不在RTL生成中使用的名字，没有任何作用，就像没有命名一样。

   出于调试编译器的目的，你可能还需要指定一个名字起始于 `*' 字符。这样的名字只 被用来标识RTL dump中的指令，其它的都与没有命名的模式一样。

2. RTL template（see Section 16.4 [RTL模板], page 203）是一个不完全的RTL表达式向量，展示了这 条指令的样子。 所谓不完全是因为它可以包含 `match_operand`, `match_operator` 和 `match_dup` 表达式，用来表示指令的操作数。

   如果向量只有一个元素，则那个元素为指令模式的模版。如果向量有多个元素，则指令模式是一个 `parallel` 表达式，包含了所描述的元素。

3. 一个条件。这是一个字符串，包含了一个C表达式用来最终测试，判定一个insn实体是否匹 配该模式。

   对于命名模式，条件（如果存在）可能不取决于要被匹配的insn的数据，而只是取决于 目标-机器-类型标记（target-machine-type flag）。编译器需要在初始化时测试这些条 件，以至于能够确切的知道在这一次运行中，哪些命名指令是可用的。

对于没有命名的模式，条件只用来匹配一个单独的insn，并且只在insn已经匹配了模式的识 别模版。insn的操作数可以为vector `operands`。对于一旦条件匹配的insn，它便不能被用来控制寄存器分配，例如用来排除某个硬件寄存器，或者硬件寄存器组合。

4. output template：一个字符串，说明了如何将匹配的insn输出为汇编代码。字符串 中的 `%' 指定了替换一个操作数值的地方。see Section 16.5 [输出模板], page 206。

   当简单替换无法满足需求的时候，你可以指定一块C代码来计算输出。 see Section 16.6 [输出语句], page 207。
5. 一个可选的向量，包含了匹配该模式的insn的属性值。 see Section 16.19 [Insn属性], page 266。

16.3 有关`define_insn`的例子

这个指令模式的例子来自68000/68020中。

```
(define_insn "tstsi"
 [(set (cc0)
    (match_operand:SI 0 "general_operand" "rm"))]
 ""
 "*
{
 if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
  return \"tstl %0\";
 return \"cmpl #0,%0\";
}")
```

还可以写成如下形式：

```
(define_insn "tstsi"
 [(set (cc0)
    (match_operand:SI 0 "general_operand" "rm"))]
 ""
 {
 if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
  return "tstl %0";
 return "cmpl #0,%0";
})
```

这是一条根据通用操作数来设置条件代码的指令。这个模式不需要条件（条件为空字符串），所以任何insn，如果RTL描述的形式相符，则可以根据这个模式来处理。名字 `tstsi' 表示“test a `SImode` value”，并且告诉了RTL生成过程，当其需要测 试这样的值时，可以使用这个模式来构造一条指令。

输出控制字符串是一部分C代码，用来根据操作数的类别和CPU的特定类型选择输出模版。

`"rm"' 是一个操作数constraint。后面将会解释它的含义。

16.4 RTL模板

RTL模板用来定义哪些insn匹配特定的模式，以及如何找到它们的操作数。对于命名的模 式，RTL模板还说明了如何根据特定的操作数来构建一个insn。

构建insn涉及到替换指定操作数到模板。匹配insn涉及到测定被匹配insn的操作数值。这 些匹配和替换操作数的行为都是由专门的表达式类型来控制。

`(match_operand:`m n predicate constraint`)`

该表达式用来代表insn中的第 n 个操作数。当构建insn时，操作数编号 n 将 在此处被替换。当匹配insn时，凡是在insn中该位置出现的将被当作操作数编号 n； 但是其必须满足 predicate，否则该指令模式将根本不匹配。

每个指令模式中的操作数编号必须从0开始连续的选择。在指令模式中，可以对每个操作数 编号只是用一个 match_operand 表达式。 通常操作数按照在 match_operand 表达式中出现的顺序被编号。对于 define_expand，任何 使用的操作数编号，只有在 match_dup 表达式中才会具有比其它操作数编号更高 的值。

predicate 为一个字符串，为一个函数的名字，其接受两个参数，一个表达式和一个 机器模式。see Section 16.7 [断言], page 208。在匹配过程中，函数将会被调用，使用假定的操作数作为 表达式并且 m 作为机器模式参数（如果 m 没有被指定，则使用 VOIDmode，这通常会使得 predicate 可以接受任何机器模式）。如果其返 回0，则该指令模式匹配失败。predicate 可以为一个空字符串；这意味着不对操作 数作测试，这样出现在该位置的任何都是有效的。

大多时候，predicate 将会拒绝 m 之外的机器模式——但并不总是这样。例 如，predicate address_operand 使用 m 作为内存引用的机器模式。许多predicate 接受 const_int 节点，即使它们的机器模式为 VOIDmode。

constraint 控制重载以及针对一个值选择最好的寄存器类别来使用，将在后面解释（see Section 16.8 [约束], page 212）。如果constraint为空字符串，则可以忽略掉。

人们经常弄不清楚constraint和predicate的区别。predicate帮助决定一个给定的insn是否匹配指令模式。constraint在该决定中不发挥作用；替代的，其控制已经匹配的insn的各种决定。

(match_scratch:m n constraint)

该表达式也是操作数编号 n 的占位符，并且指示操作数必须为一个 scratch 或者 reg 表达式。

当在匹配指令模式时，其相当于

```
(match_operand:m n "scratch_operand" pred)
```

但是，当在生成RTL时，其产生一个(scratch:m)表达式。

如果在一个 parallel 中的最后几个表达式为 clobber 表达式，其操作数 为一个硬寄存器或者 match_scratch，则组合器可以在需要的时候增加或删除它们。see Section 10.15 [副作用], page 134。

(match_dup n)

该表达式也为操作数编号 n 的占位符。其用于当操作数需要在insn中出现多次的 情况。

在构建过程中，match_dup 的作用就跟 match_operand 一样。操作数被 替换到正在被构建的insn中。但是在匹配时，match_dup 的行为就有所不同了。 其假设操作数编号 n 已经由在识别模板中之前出现的 match_operand 确 定了，其只匹配相同的表达式。

注意 match_dup 不要用来告诉编译器特定寄存器被用于两个操作数（例如： add 将一个寄存器加到另一个之上；第二个寄存器即为输入操作数，同样也为输 出操作数）。可以为此使用匹配constraint（see Section 16.8.1 [简单约束], page 212）。match_dup 是用于一个操作数在模板中的两个地方被使用的情况，例如一条指令同时计算商和余数，其中操作码接受两个输入操作数，但是RTL模板不得不引用它们两次；一次用于求商指令 模式，一次用于求余数指令模式。

(match_operator:m n predicate [operands...])

该指令模式为一个可变RTL表达式代码的一种占位符。

当构造一个insn时，其代表RTL表达式，其表达式代码取自操作数 n，并且其操作 数从指令模式 operands 中构造。

当匹配一个表达式时，其匹配一个表达式，如果函数 predicate 对于该表达式返 回非零，并且 指令模式 operands 匹配表达式的操作数。

假设函数 commutative_operator 被如下定义，来匹配任何表达式，其操作符为 RTL中可交换的算术操作符，并且其机器模式为 mode：

```
int
commutative_integer_operator (x, mode)
   rtx x;
   enum machine_mode mode;
{
 enum rtx_code code = GET_CODE (x);
 if (GET_MODE (x) != mode)
  return 0;
 return (GET_RTX_CLASS (code) == RTX_COMM_ARITH
      || code == EQ || code == NE);
}
```

那么下列指令模式将匹配任何RTL表达式，其由一个可交换操作符和两个通用操作数组成：

```
(match_operator:SI 3 "commutative_operator"
 [(match_operand:SI 1 "general_operand" "g")
  (match_operand:SI 2 "general_operand" "g")])
```

这里的向量 [operands...] 包含了两个指令模式，因为要匹配的表达 式都是包含两个操作数。

当该指令模式确实匹配时，可交换操作符的两个操作数被记录为insn的操作数1和2。（这 由 match_operand 的两个实例完成）。insn的操作数3将为整个可交换表达式：使用 GET_CODE (operands[3]) 来查看使用了哪个可交换操作符。

match_operator 的机器模式 m 的作用与 match_operand 的类似： 其被作为第二个参数传递给predicate函数，并且函数专门负责决定被匹配的表达式是否具有那 个机器模式。

当构造insn时，gen-function 的参数3将会指定要构造的表达式的操作（即，表达式代码）。其应该为一个RTL表达式，其表达式代码被复制到一个新的表达式中，新表达式的 操作数为 gen-function的参数1和2。参数3的子表达式不被使用；只与它的表达式代码 有关。

当 match_operator 被用于指令模式中来匹配insn时，通常最好让 match_operator 的操作数编号高于insn的实际操作数。这将提高寄存器分配， 因为寄存器分配者通常查看insn的操作数1和2，来看是否它可以做寄存器绑定（ register tying）。

无法指定在 match_operator 中的constraint。对应于 match_operator 的 insn的操作数，不具有任何constraint，因为它从来不作为一个整体被重载。但是，如果它 的 operands 的一部分被 match_operand 指令模式匹配，那些部分可 以具有它们自己的 constraint。

(match_op_dup:m n[operands...])

类似 match_dup，除了其应用于操作符而不是操作数。当构造insn时，操作数 编号 n 将在这一点被替代。但是在匹配时，match_op_dup 的行为有所 不同。其假设操作数编号 n 已经被在识别模板中先前出现的 match_operator 所确定，并且其只匹配 identical-looking的表达式。

(match_parallel n predicate [subpat...])

该指令模式为一个insn的占位符，该insn由一个具有可变数目元素的 parallel 表达式组成。该表达式应该只在insn指令模式的顶层出现。

当构造insn时，操作数编号 n 将在该处被替换。当匹配一个insn时，其当insn 的主体为一个 parallel 表达式，其具有至少跟向量 subpat 表达式同 样多数目元素，并且函

数 predicate 返回非零时才匹配。 predicate负责判定在 match_parallel 中的 parallel 的元素是否有效。

match_parallel 的一个典型用法是，匹配加载和存储多个表达式，其可以在 parallel 中包含一个可变数目的元素。例如，

```
(define_insn ""
 [(match_parallel 0 "load_multiple_operation"
   [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
      (match_operand:SI 2 "memory_operand" "m"))
    (use (reg:SI 179))
    (clobber (reg:SI 179))])]
 ""
 "loadm 0,0,%1,%2")
```

这个例子来自 `a29k.md'。函数 load_multiple_operation 在 `a29k.c' 中定义，其检查在 parallel 中的序列元素，是否与在 指令模式中的 set 相同，除非它们在引用后续的寄存器和内存位置。

匹配该指令模式的insn可能看起来像：

```
(parallel
 [(set (reg:SI 20) (mem:SI (reg:SI 100)))
  (use (reg:SI 179))
  (clobber (reg:SI 179))
  (set (reg:SI 21)
     (mem:SI (plus:SI (reg:SI 100)
              (const_int 4))))
  (set (reg:SI 22)
     (mem:SI (plus:SI (reg:SI 100)
              (const_int 8))))])
```

(match_par_dup n [subpat...])

与 match_op_dup 类似，但是针对于 match_parallel，而不是 match_operator。

16.5 输出模板和操作数替换

output template是一个字符串，其指定了如何为一个指令模式输出汇编代码。 大多数模板都是一个固定的字符串，可以按照字面直接输出。符号 `%' 用来指 定替换操作数；

最简单的情况下，一个 `%' 后面跟一个数字 n 表示在字符串中的这个 位置输出操作数 n。

`%' 后面跟一个字母和一个数字表示用备用方式来输出操作数。有四个字母具 有标准内嵌的含义，将在下面描述。机器描述宏 PRINT_OPERAND 能够定义 其它的非标准含义的字母。

`%cdigit' 可以用来替换一个常数值的操作数。

`%ndigit' 类似于 `%cdigit'，只不过是对要打印的常数 值进行取反操作（negated）。

`%adigit' 可以用来替换一个操作数，并把它当作一个内存引用，实际 的操作数则被视为地址。这在输出一个加载地址的指令时很有帮助，因为对于这样 的指令，汇编器语法经常需要你将操作数写成一个内存引用的形式。

`%ldigit' 用来将一个 label_ref 替换到跳转指令中。

`%=' 输出一个在整个编译过程中，对每一条指令都是唯一的编号。这可以用来 在一个生成多个汇编指令的单一模板中， 使得局部标号能够被多次引用。

`%' 后面跟一个标点符号指定了一个不使用操作数的替换。只用一种情况是标 准的：`%%' 用来输出一个 `%' 到汇编代码中。其它非标准的情况可以被 定义在 PRINT_OPERAND 宏中。你必须还要通过宏 PRINT_OPERAND_PUNCT_VALID_P 来定义哪些标点符号是有效的。

模板可以生成多条汇编指令。这些指令文本之间使用 `\;' 隔开。

‘%’ 之后跟非标准字母或者标点符号的一种用途是区分同一机器的不同汇编语言；例如，68000的Motorola语法和MIT语法。 Motorola语法要求大多数的操作名都含有句点，而MIT语法中没有。比如，MIT语法中的操作码 ‘movel’ 在Motorola语法中为 ‘move.l’ 。两种输出法语使用了同一个模式文件，只不过在Motorola语法需要句点的地方使用了字符序列 ‘%.’ 。Motorola语法的宏PRINT_OPERAND定义了这个序列用于输出一个句点；MIT语法的宏将其定义为不做任何事情。

作为一种特殊情况，只包含一个单独的字符#的模板会指示编译器首先拆分insn，然后分别输出所得的指令。这样有助于减少输出模板的重复内容。如果你有一个define_insn，其需要输出多个汇编指令，并且有一个匹配的define_split已经被定义，则你可以简单的使用#作为输出模板，而不用将模板写成输出多个汇编指令的样子。

如果定义了宏ASSEMBLER_DIALECT，则可以在模板中使用'{option0|option1|option2}'的形式。这些形式描述了多种汇编语言语法。参见指令输出。

当RTL包含两个，约束要求必须互相匹配的操作数时，输出模板必须只引用低编号的操作数。匹配的操作数并不总是相同的，编译器的其余部分会将要打印的合适的RTL表达式放到低编号的操作数中。

紧随`%'之后的非标准字母或者标点符号的一种用法是，为同一机器区别不同的汇编语言；例如，68000的Motorola语法与MIT语法。Motorola语法要求大多操作码的名字中带有句号，而MIT语法不是这样。例如，在MIT语法中的操作码`movel'，在Motorola语法中为`move.l'。这两种输出语法都在同一个文件中的指令模式中实现，只不过字符序列`%.'用在需要句号的Motorola语法中。Motorola语法的PRINT_OPERAND宏，将该序列定义成输出一个句号；而MIT语法则将其定义为不做任何事情。

作为一个特殊情况，由一单个#字符组成的模板，指示编译器首先拆分insn，然后分别输出所得的指令。这有助于在输出模板中消除冗余。如果你有一个define_insn，其需要输出多个汇编指令，并且有一个已经定义的匹配的define_split，则你可以简单的使用#作为输出模板，来替代书写输出多个汇编指令的输出模板。

如果宏ASSEMBLER_DIALECT被定义，你可以在模板中使用形如`{option0|option1|option2}'的结构。这些描述了多个汇编语言语法的变体。See Section 17.21.7 [指令输出], page 371.

16.6 用于汇编输出的C语句

经常，单个固定的模板字符串，不能够为单个指令模式所识别的所有情况都能产生正确， 有效的汇编代码。例如，操作码可以依赖于操作数类别；或者一些不适宜的操作数组合 可能需要额外的机器指令。

如果输出控制字符串起始于`@'，则其实际为一系列模板，每一个单独一行。（空行，以及开头的空格和tab被忽略掉。）这些模板对应于模式的各个constraint（see Section 16.8.2 [多个可选项], page 215）。例如，如果一个目标机有一个二址（two-address）加法指令`addr'相加到寄存器中，另外还有一个`addm'将寄存器的值相加 到内存中，你可能会这样写模式：

```
(define_insn "addsi3"
 [(set (match_operand:SI 0 "general_operand" "=r,m")
    (plus:SI (match_operand:SI 1 "general_operand" "0,0")
         (match_operand:SI 2 "general_operand" "g,r")))]
 ""
 "@
  addr %2,%0
  addm %2,%0")
```

如果输出控制字符串起始于`*'，则其不是一个输出模板，而是一个C程序片段并且 能够计算出一个模板。其应该执行一个 return 语句来返回你想要的模板字符串。 大多数这样的模板使用C字符串文字，需要用双引号包含起来。如果要在字符串中包含这 些双引号，可以在前面加上`\'。

如果输出控制串写成一个花括号块，而不是双引号的字符串，则其被自动认为是C代码。 这种情况下，则不必要有的起始的星号，以及转义C字符串文字中的双引号。

操作数可以为数组 operands，其C数据类型为 rtx []。

一种常见的情况是，根据立即数是否在一个特定范围内来选择生成汇编代码的方式。在做 这种事情的时候要仔细，因为 INTVAL 的结果是一个主机上的整形。如果主机的 int 比目标机上的具有更多的位，则从 INTVAL 中的得到的一些位将会是 多余的。要得到正确的结果，必须仔细的忽视掉那些位的值。

有可能输出一个汇编指令，然后使用子程序 output_asm_insn 来继续输出或者计 算更多的。其接收两个参数：一个模板字符串和一个操作数向量。向量可以是 operands，或者是另一个声明为局部的并且自己初始化的 rtx 数组。

当一个insn模式有多个可选择的constraint时，则汇编代码经常主要是由所匹配的constraint选择来决 定。如果是这样，C代码可以测试变量 which_alternative，其为实际满足条件的 constraint选择的序号（0为第一个，1位第二个选择，以此类推）。

例如，假设有两个操作码来存储0，`clrreg' 用于寄存器，`clrmem' 用于内存 地址。这个模式实现了如何能够使用 `which_alternative' 来选择它们：

```
(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r,m")
       (const_int 0))]
 ""
 {
 return (which_alternative == 0
     ? "clrreg %0" : "clrmem %0");
 })
```

对于上面的例子，要生成的汇编代码 只是 由alternative来决定，则还可以写成 如下形式，使用起始于 `@' 的输出控制串：

```
(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r,m")
       (const_int 0))]
 ""
 "@
  clrreg %0
  clrmem %0")
```

16.7 断言

断言用于确定一个match_operand或者 match_operator表达式是否匹配， 以及周围的指令模式是否会被用于那些操作数的组合。 GCC有许多机器无关的predicate，并且你可以根据需要来定义机器特定的断言。 按照惯例，与match_operand一起使用的predicate的名字以`_operand'结尾， 与match_operator一起使用的predicate的名字以`_operator'结尾。

所有断言（从数学意义上）都是具有两个参数的布尔函数： 指令模式中在那个位置上被考虑的RTL表达式， 以及match_operand或match_operator所指定的机器模式。 在这一节里，第一个参数被称为op，第二个参数被称为mode。 predicate可以作为普通的具有两个参数的函数，从C中调用； 这在输出模板或者其它机器特定的代码中，很有用处。

操作数断言可以允许硬件实际上无法接受的操作数， 只要约束能够为reload提供能力，来修复它们（see Section 16.8 [约束], page 212）。 然而，只要断言指定的机器指令需求尽可能的严密，

GCC通常便会生成更好的代码。reload不能修复必须为常量的操作数（立即数）；你必须使用只允许常量的断言，或者使用额外的条件来加强必要条件。

大多数predicate使用统一的方式来处理它们的mode参数。如果mode为VOIDmode（未加指明的），则op可以具有任意的模式。如果mode为其它情况，则op必须具有相同的机器模式，除非op是一个CONST_INT或整数CONST_DOUBLE。这些RTL表达式总是具有VOIDmode，所以检测它们的模式匹配反而会适得其反。替代的，接受CONST_INT和/或整数CONST_DOUBLE的predicate，可以检测存储在常量中的值是否适合所要求的机器模式。

具有这种行为的predicate被称为常规的。genrecog能够根据常规predicate如何处理机器模式的知识来优化指令识别器。它还能够诊断使用常规predicate所出现的一些常见错误。例如，使用常规predicate而没有指定机器模式几乎总是错误的。

对mode参数进行不同方式处理的predicate被称为特殊的。通用predicate address_operand和pmode_register_operand是特殊predicate。当使用特殊predicate的时候，genrecog不做任何的优化或诊断。

16.7.1 机器无关的predicate

这些是通用predicate，适用于所有后端。它们定义在`recog.c'中。第一类predicate只允许常量或立即数。

immediate_operand [Function]
该predicate允许适合相应mode的任何类别的常量。适合用于操作数必须为常量的指令。

const_int_operand [Function]
该predicate允许适合相应mode的任何CONST_INT表达式。适合用于不是符号（symbol）或标号（label）的立即数。

const_double_operand [Function]
该predicate接受任何确实为mode的CONST_DOUBLE表达式。如果mode为VOIDmode，则其还接受CONST_INT。它是用于浮点立即数的。

第二类predicate只允许某种类别的机器寄存器。

register_operand [Function]
该predicate允许适合相应mode的任何REG或SUBREG表达式。通常适合于RISC机器上的算术指令操作数。

pmode_register_operand [Function]
这与register_operand略为不同，其对机器描述的读入器有些限制。

当机器描述读入器接受`:P'机器模式后缀时，

```
(match_operand n "pmode_register_operand" constraint)
```

与

```
(match_operand:P n "register_operand" constraint)
```

将具有完全相同的含义。不幸的是，这样不行，应为Pmode是其它机器模式的别名，并且可能随着机器特定选项的不同而改变。参见Section 17.30 [其它], page 389。

scratch_operand [Function]
该predicate允许硬件寄存器和SCRATCH表达式，但不允许伪寄存器。其由match_scratch在内部使用；而不应该被直接使用。

第三类predicate只允许某种内存引用。

memory_operand [Function]
该predicate允许任何对内存中机器模式mode的一定数量的有效引用，并通过GO_IF_LEGITIMATE_ADDRESS的弱形式来确定（参见Section 17.14 [寻址模式], page 340）。

address_operand [Function]
该predicate有些不常用；其允许任何为机器模式mode的一定数量的地址有效表达式操作数，同样通过GO_IF_LEGITIMATE_ADDRESS的弱形式来确定。首先，如果`(mem:mode (exp))'被memory_operand接受，则exp被address_operand接受。注意exp不必具有机器模式mode。

indirect_operand [Function]
这是一个memory_operand的更严格形式，其只允许将general_operand作为地址表达式的内存引用。不鼓励对该predicate的新的使用，因为general_operand的条件非常宽，所以很难说清对于indirect_operand什么是被允许的，什么是不被允许的。如果目标机对不同指令的内存操作数具有不同的要求，则最好定义目标机特定的predicate，以显式的加强硬件的要求。

push_operand [Function]
该predicate允许适合将值压入栈中的内存引用。这将为一个MEM，其引用stack_pointer_rtx，且在其地址表达式中具有一个副作用（参见Section 10.16 [Incdec], page 138）；其由宏STACK_PUSH_CODE来确定（参见Section 17.10.1 [帧布局], page 315）。

pop_operand [Function]
该predicate允许适合将值弹出栈中的内存引用。同样，这将为一个MEM，其引用stack_pointer_rtx，且在其地址表达式中具有一个副作用；不过，这次是STACK_POP_CODE。

第四类predicate允许上面的操作数的某种组合。

nonmemory_operand [Function]
该predicate允许任何对于mode有效的立即数，或寄存器操作数。

nonimmediate_operand [Function]
该predicate允许任何对于mode有效的寄存器，或内存操作数。

general_operand [Function]
该predicate允许任何对于mode有效的立即数，寄存器，或内存操作数。

最后，有一个通用操作符predicate。

comparison_operator [Function]
该predicate匹配任何执行一个基于mode的算术比较表达式；即，COMPARISON_P对于表达式代码为真。

16.7.2 定义机器特定的predicate

许多机器对操作数的要求无法使用通用的predicate来精确表达。你可以使用表达式define_predicate和define_special_predicate来定义额外的predicate。这些表达式具有三个操作数：

- predicate的名字，其将在match_operand或match_operator表达式中被引用。
- 一个RTL表达式，如果predicate允许op操作数，则值为真，否则为假。该表达式只能使用下列RTL代码：

MATCH_OPERAND
: 当用于predicate表达式中时， 表达式MATCH_OPERAND在predicate允许op时为真。 操作数编号和constraint被忽略。由于genrecog中的限制， 你只能用于引用通用的predicate和已经被定义的predicate。

MATCH_CODE
: 该表达式为真， 如果op或一个指定的op的子表达式具有给定RTX代码列表中的一个RTX代码。

  该表达式的第一个操作数为一个字符串常量， 包含了逗号分割的RTX代码名字（小写形式）列表。这些是MATCH_CODE为真的代码。

  第二个操作数为一个字符串常量，其指示op的什么子表达式需要被检查。 如果没有或者为空字符串，则检查op本身。 否则，字符串常量必须为一个数字和/或小写字母的序列。 每个字符指示从当前表达式中抽取的子表达式； 第一个字符为op，第二个和后续字符，其为先前字符的结果。 数字n用于抽取`XEXP (e, n)'； 字母l抽取`XVECEXP (e, 0, n)'， 其中n为l的字母顺序（ 0为`a'，1为'b'，等等）。 MATCH_CODE然后检查完整字符串所抽取的子表达式的RTX代码。

MATCH_TEST
: 该表达式具有一个操作数，一个包含了一个C表达式的字符串常量。 在C表达式中可以使用predicate的参数，op和mode。 当C表达式为非0值时，MATCH_TEST为真。 MATCH_TEST表达式必须不具有副作用。

AND
IOR
NOT
IF_THEN_ELSE
: 基本的`MATCH_'表达式可以使用这些逻辑操作符组合， 其分别具有C操作符`&&', `||', `!'和`? :'的语义。 正如在Common Lisp中，可以给AND或IOR表达式任意数目的参数； 这跟写成两个参数的AND或IOR表达式链具有相同的效果。

- 一个可选的C代码块，其应该在发现predicate匹配时执行`return true'， 不匹配时执行`return false'。其一定不要具有副作用。 predicate参数，op和mode，是有效的。

  如果代码块存在于predicate定义中，则对于predicate允许的操作数， RTL表达式必须求值为true并且代码块必须执行`return true'。 RTL表达式被首先求值，不要重复检查代码块中的在RTL表达式中曾经被检查过的任何事情。

程序genrecog扫描define_predicate和 define_special_predicate表达式来决定什么RTX代码可能被允许。 你应该使其在RTL predicate表达式中总是显式的， 使用MATCH_OPERAND和MATCH_CODE。

这里有一个简单的定义predicate的例子，来自IA64机器描述：

```
;; True if op is a SYMBOL_REF which refers to the sdata section.
(define_predicate "small_addr_symbolic_operand"
 (and (match_code "symbol_ref")
   (match_test "SYMBOL_REF_SMALL_ADDR_P (op)")))
```

另一个例子，展示了C块的使用。

```
;; True if op is a register operand that is (or could be) a GR reg.
(define_predicate "gr_register_operand"
 (match_operand 0 "register_operand")
{
 unsigned int regno;
 if (GET_CODE (op) == SUBREG)
  op = SUBREG_REG (op);

 regno = REGNO (op);
 return (regno >= FIRST_PSEUDO_REGISTER || GENERAL_REGNO_P (regno));
})
```

使用define_predicate编写的predicate会自动包含一个测试，用来测试mode为VOIDmode，或者op具有与mode相同的机器模式，或者op为CONST_INT或CONST_DOUBLE。它们不专门检查整数CONST_DOUBLE，也不测试每种常量的值是否适合所需求的机器模式。这是因为接受常量的目标机特定的predicate，通常必须做更严厉的值检查。如果你需要确切的通用predicate提供的对CONST_INT或CONST_DOUBLE的对待，则可以使用MATCH_OPERAND子表达式来调用const_int_operand, const_double_operand或者immediate_operand。

使用define_special_predicate编写的predicate不做任何自动的机器模式检查，并且genrecog将其作为具有特定的机器模式处理来对待。

程序genpreds负责生成代码来测试predicate。其还编写了一个包含所有机器特定predicate的函数声明的头文件。所以不需要在cpu-protos.h中声明这些predicate。

16.8 操作数的约束

在指令模式中的每个match_operand都可以指定操作数所允许的约束。约束允许你在由断言所允许的操作数集里进行微调匹配。

约束可以告诉一个操作数是否可以在寄存器中，以及哪种寄存器；操作数是否可以为内存引用，以及哪种寻址方式；操作数是否可以为立即数，以及可以具有什么值。约束还可以要求两个操作数要匹配。

16.8.1 简单约束

最简单的约束种类是一个由字母组成的字符串，每个字母描述一种所允许的操作数。这里是所允许的字母：

whitespace
: 空格字符将被忽略，并且可以插到除了起始处的任何地方。这使得机器描述中，不同操作数的每个可选项可以被可视化的对齐，即使它们具有不同数目的约束和修饰符。

`m'
: 内存操作数将被允许，包括机器支持的任何寻址方式。

`o'
: 内存操作数将被允许，但只有当地址为偏移表的时候。这意味着可以对地址加上一个小的整数(实际上,是为操作数的数个字节宽度, 这由它的机器模式决定),其结果也为一个有效的内存地址。

  例如，地址为常数的为一个偏移表；所以地址为一个寄存器和常数（只要常数在机器所支持 的地址偏移范围）的和；但是递增或者递减地址不是偏移表。更加复杂的间接/索引地址可 能是或者可能不是偏移表，这取决于机器支持的其它寻址模式。

`V'
: 一个不是offsettable的内存操作数。换句话说，任何适合`m'约束但不是`o'约束的。

`<'
: 允许具有自动减量寻址（先减或者后减）的内存操作数。

`>'
允许具有自动增量寻址（先增或者后增）的内存操作数。

`r'
允许为通用寄存器的寄存器操作数。

`i'
立即数（具有常数值）将被允许。这包括符号常量， 其值将在汇编时候或者更晚的时候才被知道。

`n'
立即数，其具有已知的数值。 许多系统不支持汇编时间常量作为小于一个字的宽度的操作数。 这些操作数的约束应该为`n'而不是`i'。

`I', `J', `K', ... `P'
从`I'到`P'的其它字母可以被定义为机器特定的， 用来运行立即数具有显示指定范围的整数值。例如，在68000上， `I'被定义为代表1到8的值。这是在移位指令中被允许作为移位数的范围。

`E'
浮点立即数（表达式代码为const_double）， 但是必须target浮点格式与host机器（编译器运行的机器）的相同才行。

`F'
浮点立即数（表达式代码为const_double或者const_vector）。

`G', `H'
`G'和`H'可以被定义为机器特定的方式来允许浮点立即数具有特定范围的值。

`s'
整数立即数，其值不是一个显式的整数。

这可能有点奇怪；如果insn允许常量操作数具有在编译时不可知的值， 它当然必须允许任何可知的值。所以为什么用`s'，而不是`i'能？有时候， 它会允许生成更好的代码。

例如，在68000上的全字指令，有可能使用一个立即数操作数； 但是如果立即数的值是处于-128和127之间，更好的代码是将值加载到寄存器中， 使用寄存器。这是因为加载到寄存器中可以由`moveq'指令来完成。 我们对此通过定义字母`K'来表示任意范围超出-128和127的整数， 然后在操作数约束中指定`Ks'。

`g'
任何寄存器，内存或整数立即数，除了不是通用寄存器的寄存器。

`X'
任何操作数都被允许，即使其不满足general_operand。 这通常用于match_scratch的约束中，当一些的可选项实际上不需要scratch寄存器的时候。

`0', `1', `2', ... `9'
匹配指定操作数编号的操作数。如果数字与字母一起使用，则数字应该放在最后。

该编号允许多于单个数字。如果多个数字连续的在一起， 则它们被解析为一个单独的十进制整数。很少会因此产生不明确，因为到目前为止， 还没有想要将`10'解析为匹配操作数1或者0的。如果有这样的需要， 则可以使用多个可选项来替代。

这被称为匹配约束，其实际上是指汇编器只有一个单独的操作数， 却在RTL insn中扮演两个角色。例如，add insn在RTL中具有两个输入操作数和一个输出操作数， 但是多数CISC机器上，add指令实际上只有两个操作数，其中一个为输入输出操作数：

```
addl #35,r12
```

匹配约束被用于这些情况。更确切的说， 匹配的两个操作数必须包括一个只作输入的操作数和一个只作输出的操作数。

`p'
允许一个为有效内存地址的操作数。这用于“加载地址”和“地址压栈”指令。

约束中的`p'必须由match_operand中的作为断言的address_operand协同工作。该断言将match_operand中指定的机器模式解析为地址有效的内存引用的机器模式。

other-letters
其它字母可以采用机器相关的方式被定义，用于代表寄存器的特定类别或者其它任意的操作数类型。`d', `a'和`f'在68000/68020被定义用来代表数据，地址和浮点寄存器。

为了具有有效的汇编代码，每个操作数必须满足它的约束。但是如果不满足的话，也不会阻止将该指令模式应用到insn上。替代的，它会指示编译器去修改代码以至于约束将被满足。通常，这是通过将一个操作数复制到寄存器中来完成的。

因此，对比下面的两条指令模式：

```
(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r")
    (plus:SI (match_dup 0)
         (match_operand:SI 1 "general_operand" "r")))]
 ""
 "...")
```

其具有两个操作数，其中一个必须出现在两个位置，

```
(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r")
    (plus:SI (match_operand:SI 1 "general_operand" "0")
         (match_operand:SI 2 "general_operand" "r")))]
 ""
 "...")
```

其具有三个操作数，其中两个通过约束被要求为是相同的。如果我们考虑如下形式的一条insn

```
(insn n prev next
 (set (reg:SI 3)
    (plus:SI (reg:SI 6) (reg:SI 109)))
 ...)
```

第一个指令模式将根本不会被应用，因为该insn不在合适的地方包含两个相同的子表达式。指令模式会说“这看起来不像是一条加法指令；试一下其它模式”。第二个指令模式将会说，“是的，这是一条加法指令，但是有些问题”。它将指使编译器的重载过程生成额外的insn，使得约束为真。结果可能看起来像：

```
(insn n2 prev n
 (set (reg:SI 3) (reg:SI 6))
 ...)

(insn n n2 next
 (set (reg:SI 3)
    (plus:SI (reg:SI 3) (reg:SI 109)))
 ...)
```

你必须确保每个操作数，在每个指令模式中，具有能够处理可能会出现的任何RTL表达式的约束。（当使用多个可选项时，每个指令模式，对于每个可能的操作数表达式组合，必须至少具有一个可选项可以处理该操作数的组合。）约束不需要允许任何可能的操作数——如果是这种情况，它们就不做约束了——但是它们必须至少指出可以加载任何可能操作数，使得适合约束的方法。

- 如果约束接受断言允许的任何操作数，则没有问题：重载对于该操作数将没有必要。

  例如，操作数对于它的约束允许除了寄存器以外的任何事物，如果它的断言不接受寄存器的话，这样是安全的。

  操作数对于断言只接受常量时，如果它的约束包含字母`i'，则是安全的。如果任何可能的常量都被接受，则可以使用`i'；如果断言具有更多的选择性，则约束也可以具有更多的选择性。

- 任何操作数表达式可以通过复制到寄存器中来进行重载。所以如果一个操作数的约束允许某种寄存器，其当然是安全的。它不需要允许所有类型的寄存器；编译器知道为了使指令有效，如何将一个寄存器复制到另一个合适类别的寄存器中。

- 非偏移表的内存引用可以通过将地址复制到寄存器中来重载。所以如果约束使用字母`o'，则所有内存引用将被照顾到。

- 常量操作数可以通过在内存中分配空间作为预先被初始化的数据来重载。然后可以使用内存引用。所以如果约束使用字母`o'或者`m'，则常量操作数不是问题。
- 如果约束允许的常量和伪寄存器没有被分配到硬件寄存器中，并且等价于一个常量，则寄存器将被常量替换。如果断言不允许常量并且insn出于某种原因被识别了，则编译器将会崩溃。因此断言必须总是能够识别任何被约束允许的对象。

如果操作数的断言能够识别寄存器，但是约束不允许它们，则能够使编译器崩溃。当该操作数正好是寄存器时，重载过程将被打乱，因为它不知道如何将寄存器临时复制到内存中。

如果断言接受一元操作符，约束将被应用到操作数上。例如，MIPS处理器在ISA3级时，支持一条指令，其将两个SImode的寄存器相加产生一个 DImode的结果，但是必须寄存器能够被正确的符号扩展。该断言对于输入操作数接受一个SImode寄存器的sign_extend。将约束写成指示寄存器的类型需要为sign_extend的操作数。

16.8.2 多个可选的约束

有时单个指令具有多个可选的操作数集。例如，在68000上， 一个逻辑或指令可以将寄存器或者立即数的值组合到内存中， 或者可以组合任何类型的操作数到寄存器中；但是不能将一个内存位置组合到另一个中。

这些约束作为多个可选项来表示。 一个可选项可以通过针对每个操作数的一系列字母来描述。 一个操作数的总的约束由该操作数第一个可选项的字母，逗号， 该操作数的第二个可选项，逗号，等等直到最后一个可选项组成。 这里有一个在68000上全字逻辑或的表示：

Here is how it is done for fullword logical-or on the 68000:

```
(define_insn "iorsi3"
 [(set (match_operand:SI 0 "general_operand" "=m,d")
    (ior:SI (match_operand:SI 1 "general_operand" "%0,0")
        (match_operand:SI 2 "general_operand" "dKs,dmKs")))]
 ...)
```

第一个可选项具有操作数0的`m' (memory)，操作数1的`0'（意味着其必须匹配 操作数0）和操作数2的`dKs'。第二个可选项具有操作数0的`d' (data register)，操作数1的`0'和操作数2的`dmKs'。 约束中的`='和`%'应用于所有的可选项； 它们的含义在下一节介绍（see Section 16.8.3 [类别优先选择], page 215）。

如果所有操作数适合任意一个可选项，则指令为有效的。否则，对于每个可选项， 编译器计算要复制操作数使得可选项可以使用所需要增加的指令个数。 需要的复制最少的可选项将被选中。如果两个可选项需要相同数目的复制， 则选择前面的。这些选择可以通过字符`?'和`!'来改变：

`?` 轻微降低`?'出现的可选项，当没有可选项被确切应用时，才将其作为选择。 编译器将该可选项的花销认为高出一个单元。

`!` 严格降低`!'出现的可选项，当该选项没有reloading的时候还可以被使用， 但是如果需要reloading，则将使用其它可选项。

当insn指令模式在其约束中具有多个可选项时， 经常会出现通过哪个可选项被匹配而决定使用使用什么汇编代码的情况。这时， 输写汇编代码的C代码可以使用变量which_alternative， 其为实际被满足的可选项的顺序编号（0对应第一个，1对应第二个，等等）。See Section 16.6 [输出语句], page 207。

16.8.3 寄存器类别优先选择

操作数约束还具有另一个功能：它们使编译器可以决定为伪寄存器分配哪种硬件寄存器。 编译器检查应用到使用伪寄存器的insn的约束， 查看机器相关的指定寄存器类别的字母像`d'和`a'。 伪寄存器

被放在获得最多“票数”的类别中。约束字母`g'和`r'也要投票： 它们在通用寄存器方面进行投票。机器描述告诉哪种寄存器被认为是通用的。

当然，在一些机器上所有寄存器都是等价的，并且没有定义寄存器类别。 那么就相应的没有这么复杂了。

16.8.4 constraint修饰符

这里是约束修饰符。

`=' 意味着该指令的该操作数为只写的：先前的值将被丢弃并且由输出数据替换。

`+' 意味着该操作数可以由指令读和写。

当编译器修订操作数来满足约束时， 它需要知道哪些操作数为指令的输入以及哪些为它的输出。`='表示一个输出； `+'表示一个操作数同时为输入和输出；所有其它操作数将被认为只是输入。

如果你指定了`='或者`+'，你要将它作为约束字符串的第一个字符。

`&' 意味着（在一个特别的可选项中）该操作数为一个earlyclobber操作数， 其在指令完成使用输入操作数之前就被修改了。 因此该操作数可能不在被用作输入操作数或者用作任何内存地址的一部分的寄存器中。

`&'只应用于其所在的可选项。在具有多个可选项的约束中， 有时一个可选项需要`&'，而其它的不需要。例如，参见68000的`movdf' insn。

一个输入操作数可以被限定为一个earlyclobber操作数， 如果它唯一的作为输入的使用发生在早期结果被写出之前。 增加这种形式的可选项经常可以允许GCC来产生更好的代码， 当只有一些输出可以被earlyclobber影响时。例如，参见ARM的`mulsi3' insn。

`&'不排除对`='的需要。

`%' 声明指令对于该操作数和随后的操作数是可交换的。这意味着编译器可以交换两操作数， 如果有更廉价的方式来使得所有操作数都适合约束。 这经常被用于实际上只有两个操作数的加法指令中：结果必须放在一个参数中。 这里有个例子，是68000半字加指令如何被定义的：

```
(define_insn "addhi3"
 [(set (match_operand:HI 0 "general_operand" "=m,r")
   (plus:HI (match_operand:HI 1 "general_operand" "%0,0")
        (match_operand:HI 2 "general_operand" "di,g")))]
 ...)
```

GCC只能处理在asm中的一个可交换对；如果你有更多的，编译器将会失败。 注意如果两个可选项严格相同，则不需要使用该修饰符；这只会在重载过程浪费时间。 该修饰符在寄存器分配之后，是不可操作的， 所以在重载之后执行的define_peephole2和define_split的结果不能依赖`%'来进行insn匹配。

`#' 表示所有后续的字符，直到下一个逗号，作为约束都被忽略掉。 它们只对选择寄存器优先时有意义。

`*' 表示后续字符在选择寄存器优先时应该被忽略掉。`*'对于重载没有影响。

这里有一个例子：68000有一条指令，用于在数据寄存器中符号扩展一个半字， 并且还可以通过将其复制到一个地址寄存器中来符号扩展一个值。 当每种寄存器都可以被接受时，对于地址寄存器的约束相对不是很严格， 所以最好是寄存器分配将地址寄存器作为其目标。 因此，`*'被使用，以至于`d'约束字母（数据寄存器）被忽略， 当计算寄存器优先时。

```
(define_insn "extendhisi2"
 [(set (match_operand:SI 0 "general_operand" "=*d,a")
    (sign_extend:SI
     (match_operand:HI 1 "general_operand" "0,g")))]
 ...)
```

16.8.5 机器特定的约束

只要可能，就应该在asm参数中使用通用目的的约束字母， 因为它们可以向阅读你的代码的人们传达更加可读的意思。如果无法做到， 则使用在不同体系结构中通常具有非常相似的意思的约束字母。最通用的约束为`m'和`r'（分别用于内存和通用寄存器； see Section 16.8.1 [简单约束], page 212 “ ）， 以及`I'，通常用于指示最常见的立即数常量格式。

每个体系结构定义了额外的约束。这些约束被编译器本身使用，用于指令生成， 同时也用于asm语句；因此，一些约束对于asm并不是很有用处。 这里有一个总结，关于在一些特定机器上使用的机器相关的约束； 包括对asm有用处的和没有用处的。 在表中提到的针对每个体系结构的编译器原文件，是该体系结构的约束的定义参考。

ARM family---`config/arm/arm.h'

f	浮点寄存器
w	VFP浮点寄存器
F	浮点常量0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0或者10.0
G	浮点常量，其负数满足约束`F'
I	可以在数据处理指令中作为立即数的整数。即，0到255范围的整数，被2的倍数进行旋转
J	范围在－4095到4095的整数
K	当倒置后，满足约束`I'的整数
L	当求负后，满足约束`I'的整数
M	范围在0到32的整数
Q	一个内存引用，其确切的地址在单个寄存器中（``m''可以用于asm语句）
R	在常量池中的项
S	在当前文件中text段中的符号
Uv	适于VFP加载/存储insn（寄存器+常量 偏移量）的内存引用
Uy	适于iWMMXt加载/存储指令的内存引用
Uq	适于ARMv4 ldrsb指令的内存引用

AVR family---`config/avr/constraints.md'

l	寄存器r0到r15
a	寄存器r16到r23
d	寄存器r16到r31
w	寄存器r24到r31。这些寄存器可以用于`adiw'命令
e	指针寄存器(r26--r31)

b 基指针寄存器(r28--r31)

q 栈指针寄存器(SPH:SPL)

t 临时寄存器r0

x 寄存器对X (r27:r26)

y 寄存器对Y (r29:r28)

z 寄存器对Z (r31:r30)

I 常量，大于−1，小于64

J 常量，大于−64，小于1

K 常量整数2

L 常量整数0

M 8位的常量

N 常量整数−1

O 常量整数8，16或24

P 常量整数1

G 浮点常量0.0

R 整数常量，范围在-6 . . . 5

Q 一个内存地址，基于Y或者Z指针，加上一个位移

CRX Architecture---`config/crx/crx.h'

b 寄存器从r0到r14（不含栈指针）

l 寄存器r16（64位累加器lo寄存器）

h 寄存器r17（64位累加器hi寄存器）

k 寄存器对r16-r17（64位累加器lo-hi寄存器对）

I 3位的常量

J 4位的常量

K 5位的常量

L 常量-1, 4, -4, 7, 8, 12, 16, 20, 32, 48

G 浮点常量，合法的可以用于存储的立即数。

Hewlett-Packard PA-RISC---`config/pa/pa.h'

a 通用寄存器1

f 浮点寄存器

q 移位数量寄存器

x 浮点寄存器(不推荐的)

y 高位部分浮点寄存器（32位），浮点寄存器（64位）

`Z` 任何寄存器

`I` 有符号的11位整数常量

`J` 有符号的14位整数常量

`K` 可以使用指令`zdepi`存放的整数常量

`L` 有符号的5位整数常量

`M` 整数常量0

`N` 可以使用`ldil`指令加载的整数常量

`O` 整数常量，其值加上1便为2的幂

`P` 可以用于在`depi`和`extru`指令中`and`运算的整数常量

`S` 整数常量31

`U` 整数常量63

`G` 浮点常量0.0

`A` 一个`lo_sum` data-linkage-table内存操作数

`Q` 可以用作整数存储指令的目的操作数的内存操作数

`R` 缩放或者未缩放的索引内存操作数

`T` 浮点加载和存储的内存操作数

`W` 寄存器间接内存操作数

picoChip family---`picochip.h'

`k` 栈寄存器

`f` 指针寄存器。可以用于访问内存，无需提供偏移量。任何其它寄存器可以用于访问内存，但需要一个常量偏移量。当偏移量为零的时候，使用指针寄存器会更有效，因为这将减少代码大小。

`t` 成对寄存器。相邻的两个寄存器，用来创建一个32位寄存器。

`a` 任何绝对内存地址（例如，符号常量，符号常量+偏移量）

`I` 4位有符号整数

`J` 4位无符号整数

`K` 8位有符号整数

`M` 任何绝对值不大于4位的常量

`N` 10位有符号整数

`O` 16位有符号整数

PowerPC and IBM RS6000---`config/rs6000/rs6000.h'

`b` 基址寄存器

`f` 浮点寄存器

`v` 向量寄存器

h	`MQ', `CTR'或者`LINK'寄存器
q	`MQ'寄存器
c	`CTR'寄存器
l	`LINK'寄存器
x	`CR'寄存器(条件寄存器)编号0
y	`CR'寄存器(条件寄存器)
z	`FPMEM'栈内存，用于FPR-GPR传送
I	有符号16位常量
J	无符号16位常量，向左移16位（使用`L'来替代常量）
K	无符号16位常量
L	有符号16位常量，向左移16位
M	大于31的常量
N	2的幂
O	零
P	常量，其负数为有符号的16位常量
G	浮点常量，可以使用一个字的指令将其加载到寄存器中
H	整数/浮点常量，可以使用三条指令将其加载到寄存器中
Q	内存操作数，相对于寄存器的偏移量(`m'适用于asm语句)
Z	内存操作数，来自寄存器的一个索引或者间接访问(`m'适用于asm语句)
R	AIX TOC项
a	地址操作数，来自寄存器的一个索引或者间接访问(`p'适用于asm语句)
S	适于作64位掩码操作数的常量
T	适于作32位掩码操作数的常量
U	System V Release 4对小数据区域的引用
t	AND掩码，可以通过两条rldic{l, r}指令执行
W	不需要内存的向量常量

Intel 386---`config/i386/constraints.md'

R	遗留的寄存---八个在所有i386处理上都可用的寄存器(a, b, c, d, si, di, bp, sp)。
q	任何可以作为rl访问的寄存器。在32位机器模式中，为a, b, c和d; 在64位机器模式中，为任何整数寄存器。
Q	任何可以作为rh来访问的寄存器：a, b, c和d。
l	任何在基址+索引的内存访问中，可以作为索引的寄存器：即除了栈指针以外的任何通用寄存器。

a	寄存器a
b	寄存器b
c	寄存器c
d	寄存器d
S	寄存器si
D	寄存器di
A	寄存器a和d，作为一对（用于在一个寄存器中返回结果的一半，另一个寄存器中返回另一半的指令）
f	任何80387浮点（栈）寄存器
t	80387浮点栈顶（%st(0)）
u	从80387浮点栈顶起始的第二项(%st(1))
y	任何MMX寄存器
x	任何SSE寄存器
Yz	第一个SSE寄存器(%xmm0)
Y2	任何SSE寄存器，当SSE2被启用
Yi	任何SSE寄存器，当SSE2和inter-unit move被启用
Ym	任何MMX寄存器，当inter-unit move被启用
I	整数常量，范围在0 . . . 31，用于32位移位
J	整数常量，范围在0 . . . 63，用于64位移位
K	有符号8位整数常量
L	0xFF或者0xFFFF，用于andsi，作为零扩展move
M	0, 1, 2, 或者 3 (用于lea指令的移位)
N	无符号8位整数常量（用于in和out指令）
O	整数常量，范围在0 . . . 127，用于64位移位
G	标准的80387浮点常量
C	标准的SSE浮点常量
e	32位有符号整数常量，或者已知适合该范围的符号引用（用于有符号扩展x86-64指令的立即数）
Z	32位无符号整数常量，或者已知适合该范围的符号引用（用于零扩展x86-64指令的立即数）

Intel IA-64---`config/ia64/ia64.h'

a	通用寄存器r0到r3，用于addl指令
b	分支寄存器
c	断言寄存器(`c' as in ``conditional'')

d 在M-unit中的应用寄存器

e 在I-unit中的应用寄存器

f 浮点寄存器

m 内存操作数，记住在IA-64上，`m'允许使用`%Pn'打印的后增和后减方式。使用`S'来禁止后增和后减。

G 浮点常量0.0或者1.0

I 14位有符号整数常量

J 22位有符号整数常量

K 8位有符号整数常量，用于逻辑指令

L 8位被调整的有符号整数常量，用于比较伪操作

M 6位无符号整数常量，用于移位计数

N 9位有符号整数常量，用于后增方式的加载和存储

O 常量零

P 0 或者 −1，用于dep指令

Q 非volatile内存，用于浮点加载和存储

R 整数常量范围在1到4，用于shladd指令

S 除了后增和后减以外的内存操作数

FRV---`config/frv/frv.h'

a 类别ACC_REGS中的寄存器(acc0 到 acc7).

b 类别EVEN_ACC_REGS中的寄存器(acc0 到 acc7).

c 类别CC_REGS中的寄存器(fcc0 到 fcc3 以及 icc0 到 icc3).

d 类别GPR_REGS中的寄存器(gr0 到 gr63).

e 类别EVEN_REGS中的寄存器(gr0 到 gr63)。奇数寄存器不在该类中，但是可以使用大于4个字节的机器模式来使用。

f 类别FPR_REGS中的寄存器(fr0 到 fr63).

h 类别FEVEN_REGS中的寄存器(fr0 到 fr63)。奇数寄存器不在该类中，但是可以使用大于4个字节的机器模式来使用。

l 类比LR_REG中的寄存器(lr寄存器)。

q 类别QUAD_REGS中的寄存器(gr2 到 gr63)。无法被4整除的寄存器编号不在该类中，但是可以使用大于8个字节的机器模式来使用。

t 类别ICC_REGS中的寄存器(icc0 到 icc3).

u 类别FCC_REGS中的寄存器(fcc0 到 fcc3).

v 类别ICR_REGS中的寄存器(cc4 到 cc7).

w 类别FCR_REGS中的寄存器(cc0 到 cc3).

x 类别QUAD_FPR_REGS中的寄存器(fr0 到 fr63)。无法被4整除的寄存器编号不在该类中，但是可以使用大于8个字节的机器模式来使用。

z 类别SPR_REGS中的寄存器(lcr 和 lr).

A 类别QUAD_ACC_REGS中的寄存器(acc0 到 acc7).

B 类别ACCG_REGS中的寄存器(accg0 到 accg7).

C 类别CR_REGS中的寄存器(cc0 到 cc7).

G 浮点常量零

I 6位有符号整数常量

J 10位有符号整数常量

L 16位有符号整数常量

M 16位无符号整数常量

N 12位有符号整数常量，且为负---即，范围在−2048 到 −1

O 常量零

P 12有符号整数常量，且大于零---即，范围在1到2047

Blackfin family---`config/bfin/constraints.md'

a P寄存器

d D寄存器

z 被函数调用破坏的P寄存器

qn 单个寄存器。如果n在范围0到7中，则对应D寄存器。如果为A，则是寄存器P0。

D 偶数编号的D寄存器

W 奇数编号的D寄存器

e 累加寄存器

A 偶数编号的累加寄存器

B 奇数编号的累加寄存器

b I寄存器

v B寄存器

f M寄存器

c 用于循环缓冲的寄存器，即I, B或者L寄存器。

C CC寄存器

t LT0或者LT1.

k LC0或者LC1.

u LB0或者LB1.

x	任何D, P, B, M, I 或者L寄存器。
y	通常只用于函数序言和尾声的额外寄存器：RETS, RETN, RETI, RETX, RETE, ASTAT, SEQSTAT 和 USP。
w	除了累加器和CC以外的任何寄存器。
Ksh	有符号16位整数（范围在-32768到32767）
Kuh	无符号16位整数(范围在0到65535)
Ks7	有符号7位整数(范围在-64 到 63)
Ku7	无符号7位整数(范围在0到127)
Ku5	无符号5位整数(范围在0到31)
Ks4	有符号4位整数(范围在-8到7)
Ks3	有符号3位整数(范围在-3到4)
Ku3	无符号3位整数(范围在0到7)
Pn	常量n，为一单个数字的常量，范围在0到4。
PA	一个整数，等于MACFLAG_XXX常量中的一个，适合用于每个累加器。
PB	一个整数，等于MACFLAG_XXX常量中的一个，只适合用于累加器A1。
M1	常量255
M2	常量65535
J	整数常量，只有一个位被设置。
L	整数常量，只有一个位没有被设置。
H	
Q	任何SYMBOL_REF

M32C---`config/m32c/m32c.c'

Rsp Rfb Rsb	`$sp', `$fb', `$sb'.
Rcr	任何控制寄存器，16位宽的时候
Rcl	任何控制寄存器，24位宽的时候
R0w R1w R2w R3w	$r0, $r1, $r2, $r3.
R02	$r0 或 $r2, 或用于32位值的$r2r0
R13	$r1 或 $r3, 或用于32位值的$r3r1
Rdi	可以存放64位值的寄存器
Rhl	$r0 或 $r1 (具有可寻址的高 / 低字节的寄存器)

R23	$r2 或 $r3
Raa	地址寄存器
Raw	地址寄存器，16位宽的时候
Ral	地址寄存器，24位宽的时候
Rqi	可以存放QI值的寄存器
Rad	可以和偏移一起使用的寄存器($a0, $a1, $sb)
Rsi	可以存放32位值的寄存器
Rhi	可以存放16位值的寄存器
Rhc	可以存放16位值的寄存器，包括所有的控制寄存器
Rra	$r0到R1，加上$a0和$a1
Rfl	标记寄存器
Rmm	基于内存的伪寄存器，$mem0到$mem15
Rpi	可以存放指针的寄存器(16位寄存器r8c, m16c; 24位寄存器m32cm, m32c)
Rpa	并行的匹配多个寄存器，形成一个大的寄存器。用于匹配函数返回值
Is3	-8 . . . 7
IS1	-128 . . . 127
IS2	-32768 . . . 32767
IU2	0 . . . 65535
In4	-8 . . . -1 或 1 . . . 8
In5	-16 . . . -1 或 1 . . . 16
In6	-32 . . . -1 或 1 . . . 32
IM2	-65536 . . . -1
Ilb	一个8位的值，且只有一个位被设置
Ilw	一个16位的值，且只有一个位被设置
Sd	普通的src/dest内存寻址模式
Sa	使用$a0或$a1的寻址模式
Si	带有立即数地址的寻址内存
Ss	使用栈指针($sp)寻址的内存
Sf	使用帧基址寄存器($fb)寻址的内存
Ss	使用小基址寄存器($sb)寻址的内存
S1	$r1h

MIPS---`config/mips/constraints.md'

d	地址寄存器。与r等价，只不过是生成MIPS16代码

f	浮点寄存器（如果可用）
h	之前为hi寄存器。该约束不再被支持。
l	lo寄存器。使用该寄存器来存放不大于一个字的值。
x	hi 和 lo寄存器的结合。使用该寄存器来存放双字的值。
c	适用于间接跳转的寄存器。对于`-mabicalls'，其将总是为$25。
v	寄存器$3。不要在新的代码中使用该约束；保留它只是为了与glibc兼容。
y	等价于r; 保留它是为了向后兼容。
z	浮点条件代码寄存器
I	有符号16位常量(用于算术指令)
J	整数零
K	无符号16位常量(用于逻辑指令)
L	有符号32位常量，其中低16位为零。这样的常量可以使用lui来加载。
M	不可以使用lui, addiu或ori加载的常量。
N	常量，范围在-65535 到 -1 (含)。
O	有符号15位常量
P	常量，范围在1到65535 (含)
G	浮点零
R	可以用于非宏的加载和存储中的地址。

Motorola 680x0---`config/m68k/constraints.md'

a	地址寄存器
d	数据寄存器
f	68881浮点寄存器，如果可用
I	整数，范围在1到8
J	16位有符号数字
K	有符号数字，大于0x80
L	整数，范围在−8 到 −1
M	有符号数字，大于0x100
N	范围在24到31，rotatert:SI 8 到 1，表示为rotate
O	16 (使用swap的rotate)
P	范围在8到15，rotatert:HI 8 到 1，表示为rotate
R	mov3q可以处理的数字
G	浮点常量，且不是68881常量
S	操作数，当-mpcrel有效时，满足'm'

T	操作数，当-mpcrel无效时，满足's'
Q	地址寄存器，简介寻址模式
U	寄存器偏移寻址
W	const_call_operand
Cs	symbol_ref 或 const
Ci	const_int
C0	const_int 0
Cj	不适合16位的有符号数的范围不适合16位
Cmvq	用于mvq的整数
Capsw	整数，用于moveq后面跟一个swap
Cmvz	用于mvz的整数
Cmvs	用于mvs的整数
Ap	push_operand
Ac	允许在clr中使用的非寄存器操作数

Motorola 68HC11 & 68HC12 families---`config/m68hc11/m68hc11.h'

a	寄存器`a'
b	寄存器`b'
d	寄存器`d'
q	8位寄存器
t	临时软寄存器_.tmp
u	软寄存器_.d1 到 _.d31
w	栈指针寄存器
x	寄存器`x'
y	寄存器`y'
z	伪寄存器`z' (在后来被`x' 或 `y'替换)
A	地址寄存器：x, y 或 z
B	地址寄存器：x 或 y
D	寄存器(x:d)，形成一个32位的值
L	常量，范围在−65536 到 65535
M	常量，低16位为零
N	常整数1 或 −1
O	常整数16
P	常量，范围在−8 到 2

SPARC---`config/sparc/sparc.h'

f	SPARC-V8体系结构上的浮点寄存器，以及SPARC-V9体系结构上的低浮点寄存器。
e	浮点寄存器。在SPARC-V8体系结构上等价于`f'，在SPARC-V9体系结构上包含低位和高位的浮点寄存器。
c	浮点条件代码寄存器。
d	低位浮点寄存器。只用于SPARC-V9体系结构上，当虚拟指令集可用的时候。
b	浮点寄存器。只用于SPARC-V9体系结构上，当虚拟指令集可用的时候。
h	64位global或out寄存器，用于SPARC-V8+体系结构。
D	向量常量
I	有符号13位常量
J	零
K	32位常量，低12位被清零（可以使用sethi指令加载的常量）
L	movcc指令所支持的范围内的常量
M	movrcc指令所支持的范围内的常量
N	与`K'相同，只不过，其会验证不在低32位范围的位全是零。对于机器模式宽于SImode的，比需使用其，而不是`K'
O	常量4096
G	浮点零
H	有符号13位常量，符号扩展到32或64位
Q	浮点常量，其整数表示可以使用单个sethi指令，被移送到整数寄存器中。
R	浮点常量，其整数表示可以使用单个mov指令，被移送到整数寄存器中。
S	浮点常量，其整数表示可以使用一个high/lo_sum指令序列，被移送到整数寄存器中。
T	内存地址，对齐到8字节的边界
U	偶数寄存器
W	内存地址，用于`e'约束寄存器
Y	向量零

SPU---`config/spu/spu.h'

a	立即数，可以使用il/ila/ilh/ilhu指令加载。const_int被当作64位值。
c	立即数，用于and/xor/or指令。const_int被当作64位值。
d	立即数，用于iohl指令。const_int被当作64位值。
f	立即数，可以使用fsmbi指令加载。

A 立即数，可以使用il/ila/ilh/ilhu指令加载。const_int被当作32位值。

B 立即数，用于大多算术指令。const_int被当作32位值。

C 立即数，用于and/xor/or指令。const_int被当作32位值。

D 立即数，用于iohl指令。const_int被当作32位值。

I 常量，范围为[-64, 63]，用于shift/rotate指令。

J 无符号7位常量，用于conversion/nop/channel指令。

K 有符号10位常量，用于大多算术指令。

M 有符号16位立即数，用于stop。

N 无符号16位常量，用于iohl和fsmbi。

O 无符号7位常量，其3个最小有效位是0。

P 无符号3位常量，用于16字节的rotate和shift。

R 调用操作数，寄存器，用于间接调用。

S 调用操作数，符号，用于相关调用。

T 调用操作数，const_int，用于绝对调用。

U 立即数，可以使用il/ila/ilh/ilhu指令加载。const_int被扩展为128位。

W 立即数，用于shift和rotate指令。const_int被当作32位值。

Y 立即数，用于and/xor/or指令。const_int被有符号扩展为128位。

Z 立即数，用于iohl指令。const_int被有符号扩展为128位。

S/390 and zSeries---`config/s390/s390.h'

a 地址寄存器(通用目的寄存器，除了r0)

c 条件代码寄存器

d 数据寄存器（任意通用寄存器）

f 浮点寄存器

I 无符号8位常量(0--255)

J 无符号12位常量(0--4095)

K 有符号16位常量(－32768--32767)

L 适合作偏移量的值

(0..4095) 短偏移

(-524288..524287)
长偏移

M 常量整数，值为0x7fffffff

N 多个字母约束，跟随4个参数字母

0..9: 部分的编号，从最大有效到最小有效计数

H,Q: 部分的机器模式

D,S,H: 包含的操作数的机器模式

0,F: 其它部分的值(F---所有位都被设置)

如果常量所指定的部分具有与其它部分不同的值，则约束匹配

Q 没有索引寄存器，但是有短偏移的内存引用

R 具有索引寄存器和短偏移的内存引用

S 没有索引寄存器，但是有长偏移的内存引用

T 具有索引寄存器和长偏移的内存引用

U 具有短偏移的指针

W 具有长偏移的指针

Y 移位计数操作数

Score family---`config/score/score.h'

d 寄存器r0到r32

e 寄存器r0到r16

t r8---r11 或 r22---r27 寄存器

h hi寄存器

l lo寄存器

x hi + lo 寄存器

q cnt寄存器

y lcb寄存器

z scb寄存器

a cnt + lcb + scb寄存器

c cr0---cr15寄存器

b cp1寄存器

f cp2寄存器

i cp3寄存器

j cp1 + cp2 + cp3寄存器

I 高16位常量(32位常量，16位最小有效位为零)

J 无符号5位整数(范围从0到31)

K 无符号16位整数(范围从0到65535)

L 有符号16位整数(范围从－32768到32767)

M 无符号14位整数(范围从0到16383)

N 有符号14位整数(范围从－8192到8191)

Z 任何SYMBOL_REF

Xstormy16---`config/stormy16/stormy16.h'

a 寄存器r0

b 寄存器r1

c 寄存器r2

d 寄存器r8

e 寄存器r0到r7

t 寄存器r0和r1

y 进位寄存器

z 寄存器r8和r9

I 常量，在0和3之间

J 常量，只有一位被设置

K 常量，只有一位被清零

L 常量，在0和255之间

M 常量，在−255和0之间

N 常量，在−3和0之间

O 常量，在1和4之间

P 常量，在−4和−1之间

Q 内存引用，为一个栈压入

R 内存引用，为一个栈弹出

S 内存引用，表示一个已知的常量地址

T 通过Rx指示的寄存器（还没实现）

U 不在2和15之间的常量

Z 常量0

Xtensa---`config/xtensa/constraints.md'

a 通用32位寄存器

b 一位布尔寄存器

A MAC16 40位累加器

I 有符号12位整数常量，用在MOVI指令中

J 有符号8位整数常量，用在ADDI指令中

K 整数常量，用于BccI指令

L 无符号常量，用于BccUI指令

16.8.6 使用enabled属性来禁止insn可选项

insn属性enabled，可以出于机器特定的原因，用来禁止特定的insn的可选项。这用于，当为现有的指令模式，增加新的指令，且其只用于使用-march=选项指定的特定cpu体系结构级别。

如果insn的可选项被禁止，则其将不被使用。编译器将被禁止的可选项的约束视为不被满足。

为了能够使用enabled属性，后端必须在机器描述文件中增加：

1. 对enabled insn属性的定义。该属性通常使用define_attr命令来定义。该定义应该基于其它insn属性以及/或者目标机标记。enabled属性为数字属性，并且对于被启用的可选项应该求值为(const_int 1)，否则为(const_int 0)。
2. 另一个insn属性的定义，用于描述为什么一个insn可选项可用或不可用。例如，下面例子中的cpu_facility。
3. 对每个insn定义，赋予第二个属性。（注意：显然只是针对那些具有多个可选项的定义。其它insn模式应该使用insn条件来禁止或开启。）

例如，下面两个指令模式可以容易的使用enabled属性合并在一起：

```
(define_insn "*movdi_old"
 [(set (match_operand:DI 0 "register_operand" "=d")
    (match_operand:DI 1 "register_operand" " d"))]
 "!TARGET_NEW"
 "lgr %0,%1")

(define_insn "*movdi_new"
 [(set (match_operand:DI 0 "register_operand" "=d,f,d")
    (match_operand:DI 1 "register_operand" " d,d,f"))]
 "TARGET_NEW"
 "@
  lgr  %0,%1
  ldgr %0,%1
  lgdr %0,%1")
```

合并成:

```
(define_insn "*movdi_combined"
 [(set (match_operand:DI 0 "register_operand" "=d,f,d")
    (match_operand:DI 1 "register_operand" " d,d,f"))]
 ""
 "@
  lgr  %0,%1
  ldgr %0,%1
  lgdr %0,%1"
 [(set_attr "cpu_facility" "*,new,new")])
```

其中enabled属性的定义为：

```
(define_attr "cpu_facility" "standard,new" (const_string "standard"))

(define_attr "enabled" ""
 (cond [(eq_attr "cpu_facility" "standard") (const_int 1)
     (and (eq_attr "cpu_facility" "new")
        (ne (symbol_ref "TARGET_NEW") (const_int 0)))
     (const_int 1)]
    (const_int 0)))
```

16.8.7 定义机器特定的约束

机器特定的约束分为两类：寄存器约束和非寄存器约束。在后者中， 如果约束允许所有可能的内存或地址操作数，则应该被专门标记出来， 以便给reload更多信息。

机器特定的约束可以给定任意长度的名字，但是它们全部由字母，数字， 下划线（`_'）和三角括号（`<>'）组成。跟C标识符类似， 它们必须起始于字母或者下划线。

为了避免操作数约束字符串的混淆，约束的名字不能起始于任何其它约束的名字。 例如，如果x被定义为一个约束名，则不可以定义xy，反之依然。 按照这个规则，所有约束都不能起始于通用约束字母：`E F V X g i m n o p r s'。

寄存器约束直接对应于寄存器类别。参见Section 17.8 [寄存器类别], page 307。 因此它们的定义没有太多的灵活性。

define_register_constraint name regclass docstring [MD Expression]

这三个参数都是字符串常量。name为约束的名字， 将在match_operand表达式中出现。如果name为多个字母的约束， 则它的长度应该与所有起始与同一字母的约束相同。 regclass可以为相应的寄存器类别的名字（参见Section 17.8 [寄存器类别], page 307）， 或者一个C表达式，其值为合适的寄存器类别。如果为表达式，其必须不具有副作用， 并且不能查看操作数。通常使用表达式是为了当寄存器类别对于给定的子体系结构无效时， 将一些寄存器约束映射为NO_REGS。

docstring为一条语句，介绍了约束的含义。这将在下面做进一步的解释。

非寄存器的约束更加像断言：约束定义给出一个布尔表达式，其指示是否约束匹配。

define_constraint name docstring exp [MD Expression]

name和docstring参数与define_register_constraint的相同， 但是注意docstring直接跟随name之后。 exp为一个RTL表达式， 遵循在断言定义中相同的规则。详情参见Section 16.7.2 [定义predicate], page 210。如果求得为真， 则约束匹配；如果求得为假，则不匹配。约束表达式应该指示出它们可能匹配的RTL， 就像断言表达式一样。

C表达式match_test，可以访问下列变量：

op 定义操作数的RTL对象。

mode op的机器模式。

ival `INTVAL (op)', 如果op为const_int。

hval `CONST_DOUBLE_HIGH (op)', 如果op为整数const_double。

lval `CONST_DOUBLE_LOW (op)', 如果op为整数const_double。

rval `CONST_DOUBLE_REAL_VALUE (op)', 如果op为浮点const_double。

变量*val应该只在表达式的其它部分已经验证了op为合适类型的RTL对象时， 才被使用。

大多数非寄存器约束应该使用define_constraint来定义。 其余的两个定义表达式只适合当约束匹配失败时，应该由reload单独处理的约束。

define_memory_constraint name docstring exp [MD Expression]

使用该表达式来定义匹配所有内存操作数的子集的约束：也就是， reload能够通过将操作数转换为`(mem (reg X))'的形式使得它们 匹配。其中X为基址寄存器（通过BASE_REG_CLASS指定的寄存器类别，see Section 17.8 [寄存器类别], page 307）。

例如，在S/390上，一些指令不接受任意的内存引用，只接受那些不使用索引寄存器的。 约束字母`Q'被定义用来表示这个类型的内存地址。 如果`Q'使用define_memory_constraint定义，则`Q'约束可以处理任意内存操作数，因为reload知道在需要的时候， 它可以简单的将内存地址复制到基址寄存器中。 这与`o'约束可以处理任意内存操作数的方式类似。

语法和语义在其它方面都与define_constraint相同。

define_address_constraint name docstring exp [MD Expression]

使用该表达式来定义匹配所有地址操作数的子集的约束：也就是， reload能够通过将操作数转换为`(reg X)'的形式使得它们匹配。 同样X为基址寄存器。

使用define_address_constraint定义的约束只能用于address_operand断言 ，或者机器特定的同样方式的断言。它们与通用的`p'约束类似。

语法和语义在其它方面都与define_constraint相同。

由于历史的原因， 起始于字母`G H'的名字被保留为只匹配const_double的约束， 起始与字母`I J K L M N O P'被保留为只匹配const_int的约束。 这在将来可能会改变。暂时的，这些名字的约束必须使用固定形式来书写， 以便genpreds能够辨别出你在做正确的事情：

```
(define_constraint "[GHIJKLMNOP]..."
 "doc..."
 (and (match_code "const_int") ; const_double for G/H
   condition...))       ; usually a match_test
```

可以使用起始于其它字母的名字来定义匹配const_double或const_int的约束。

在约束定义中的每个docstring应该是一条或多条完整的语句，使用Texinfo格式来标记。 它们目前没有被使用。在将来，它们将被复制到 GCC手册中，在机器约束这一章节， 用来替换手工维护的表格。而且，将来编译器可以使用其来给出更多有帮助的诊断信息， 当过少的asm约束选择造成重载失败时。

如果你在docstring的起始出放入伪Texinfo指令`@internal'， 则（ 在将来 ）其将只出现在internals手册版本的机器特定约束表中。 这可以用于不应该出现在asm语句中的约束。

16.8.8 从C中测试约束

有时从C代码中测试约束要比隐式的通过match_operand中的约束字符串有用处。 生成文件`tm_p.h'声明了一些接口，用于机器特定的约束。 这些接口都没有用于在Section 16.8.1 [简单约束], page 212中描述的通用约束。 这在将来可能会有所改变。

警告：`tm_p.h'可能声明了其它操作约束的函数，除了在这里列的以外。 不要在机器独立的代码中使用那些函数。它们只是为了实现旧的约束接口。 它们在将来将会有变动或者消失。

一些有效的约束名字不是有效的C标志符， 所以这里有一个mangling框架用于从C中引用它们。不包含三角括号或者下划线的约束名保持不变。下划线改写成两次， 每个`<'被`_l'替换，每个`>'被`_g'替换。这里有些例子：

Original	Mangled
x	x
P42x	P42x
P4_x	P4__x
P4>x	P4_gx
P4>>	P4_g_g
P4_g>	P4__g_g

在该章节中，变量c或者为一个抽象的约束， 或者为来自enum constraint_num的常量； 变量m为一个mangled约束名字（通常作为一个大标志符的一部分）。

`constraint_num` [Enum]

对于每个机器特定的约束，有一个对应的枚举常量：`CONSTRAINT_'加上约束的mangled名字。 函数接受一个`enum constraint_num`作为参数。

机器独立的约束不具有相关的常量。这在将来可能会有改变。

`inline bool satisfies_constraint_m (rtx exp)` [Function]

对于每个机器特定的，非寄存器约束m，有一个这样函数；其返回`true`， 如果exp满足约束。这些函数只有当`rtl.h'被包含在`tm_p.h'之前时才可见。

`bool constraint_satisfied_p (rtx exp, enum constraint_num c)` [Function]

类似`satisfies_constraint_m`函数，只是被测试的约束作为参数给出， c。如果c指定一个寄存器约束，该函数将总是返回`false`。

`enum reg_class regclass_for_constraint (enum constraint_num c)` [Function]

返回与c关联的寄存器类别。如果c不是寄存器约束， 或者那些寄存器对于当前选择的子target无效，则返回`NO_REGS`。

这里有一个使用`satisfies_constraint_m`的例子。 在窥孔优化（参见Section 16.18 [窥孔定义], page 263）中，操作数约束字符串将被忽略， 所以如果有相应的约束，它们必须在C条件中被测试。在例子中，优化将被采用， 如果操作数2不满足`K'约束。（这是从 i386机器描述中的窥孔定义简化的版本。）

```
(define_peephole2
 [(match_scratch:SI 3 "r")
  (set (match_operand:SI 0 "register_operand" "")
       (mult:SI (match_operand:SI 1 "memory_operand" "")
                (match_operand:SI 2 "immediate_operand" "")))]

 "!satisfies_constraint_K (operands[2])"

 [(set (match_dup 3) (match_dup 1))
  (set (match_dup 0) (mult:SI (match_dup 3) (match_dup 2)))]

 "")
```

16.9 用于生成的标准指令模式名

这里有一个在编译器的RTL生成过程中有意义的指令名称表，在指令模式中给定这些名字中的一个，则告诉RTL生成过程，其可以使用该指令模式来完成一个特定的任务。

`movm'

这里m表示一个两字母的机器模式名字，小写。该指令模式将那种机器模式的数据从操作数1移送到操作数0。例如`movsi'移送整字数据。

如果操作数0为一个寄存器的`subreg`，机器模式为m，寄存器自己的机器模式比m更宽，则该指令的效果是将指定的值存储在寄存器的对应于机器模式m的部分。m之外，且与`subreg`在同一个目标字中的位，为未定义。目标字以外的位保持不变。

这类指令模式有几处特别的地方。首先，每个这些直到整字大小的名字，包括整字大小的，必须被定义，因为没有其它方式来从一个地方将数据复制到另一个地方。如果有接受更大机器模式的操作数的指令模式，则必须为那些大小的整数机器模式定义`movm'。

第二，这些指令模式不仅用在RTL生成过程。甚至重载过程可以生成move insn将值从栈槽复制到临时寄存器中。当这样的时候，其中一个操作数为硬件寄存器，另一个为需要被重载到寄存器中的操作数。

因此，当给定这样一对操作数时，指令模式必须生成不需要重载，并且不需要临时寄存器的RTL。例如，如果你使用一个define_expand来支持该指令模式，则这种情况下，define_expand一定不能调用force_reg或者任何其它可能生成新的伪寄存器的函数。

甚至对于在RISC机器上的子字机器模式，从内存中获取这些机器模式通常需要多个insn和一些临时变量，该要求也存在。

重载过程中，具有无效地址的内存引用可以被作为操作数。这样的地址将在重载过程的后面被替换成有效地址。这种情况下，可能对地址没有做任何事情，而只是使用它。如果其被复制，则将无法使用有效的地址进行替换。不要尝试将这样的地址变成有效的地址。注意，general_operand当应用到这样的地址的时候将会失败。

全局变量reload_in_progress（其必须被显式的声明）可以用来确定是否需要这样的特殊的处理。

需要进行重载的操作数的种类取决于机器描述的其余部分，不过通常在RISC机器上，只有那些没有获得硬件寄存器的伪寄存器，而在其它机器上，显式的内存引用也有可能需要进行重载。

如果一个scratch寄存器，被需要用来将一个对象与内存之间进行移送，则其可以活跃分析之前，使用gen_reg_rtx来进行分配。

如果在重载过程中，或者之后，有需要scratch寄存器的情况，则你必须提供一个适当的secondary_reload目标钩子。

宏can_create_pseudo_p可以用来确定创建一个新的伪寄存器是否不安全。如果该变量为非零，则调用gen_reg_rtx来分配一个新的伪寄存器是不安全的。

`movm'上的约束必须允许将任何硬件寄存器移送到任何其它硬件寄存器上，假设HARD_REGNO_MODE_OK在两个寄存器上都允许机器模式m，并且REGISTER_MOVE_COST应用到它们的类别上返回值2。

必须提供浮点`movm'指令，用于任何可以存放定点值的寄存器，因为联合体和结构体（具有机器模式SImode或DImode）可以在那些寄存器中，并且它们可以具有浮点成员。

还需要支持定点`movm'指令，用于浮点寄存器。不幸的是，我忘了为什么要这样，并且不知道这是否还是真的。如果HARD_REGNO_MODE_OK排斥在浮点寄存器中的定点值，则定点`movm'指令的约束必须被设计成，避免尝试重载到一个浮点寄存器。

`reload_inm'
`reload_outm'
这些命名指令模式已经被目标钩子secondary_reload废弃。

类似`movm'，不过用于当需要使用scratch寄存器在操作数0和操作数1之间移送的时候。操作数2描述scratch寄存器。参见在see Section 17.8 [寄存器类别], page 307中对SECONDARY_RELOAD_CLASS宏的讨论。

这些指令模式中的match_operand的形式，有一些特殊的限制。首先，只有重载操作数的断言才被检查，即reload_in检查操作数1，而不检查操作数0和2。第二，在约束中只能有一个可选项。第三，约束只能使用单个寄存器类别字母；后续的约束字母都被忽略。一个例外是，空的约束字符串匹配ALL_REGS寄存器类别。这可以减轻后端为这些指令模式定义ALL_REGS约束字母的负担。

`movstrictm'
类似`movm'，只不过，如果操作数0为一个寄存器的机器模式为m的subreg，且寄存器的自然机器模式是较宽的，则`movstrictm'指令保证不会修改属于机器模式m之外的寄存器的任何部分。

`movmisalignm'
该move指令模式的变体，被设计为从没有与其机器模式自然对齐的内存地址中加载和存储值。对于存储，内存将在操作数0中；对于加载，内存将在操作数1中。其它操作数保证不为内存，所以容易判别是加载还是存储。

该指令模式用于向量化，当展开MISALIGNED_INDIRECT_REF表达式的时候。

`load_multiple'
将多个连续的内存位置加载到连续的寄存器中。操作数0为连续寄存器中的第一个，操作数1为第一个内存位置，操作数2为一个常量：连续寄存器的数目。

只有当目标机器确实具有这样指令的时候才定义该指令模式；如果将内存加载到连续寄存器的最有效的方式，是每次加载一个，则 不用定义该指令模式。

在一些机器上，对于哪些连续寄存器可以存储到内存中，会有一些限制，例如特定的起始寄存器或者结尾寄存器的编号，或者一个有效范围。对于那些机器，使用define_expand (see Section 16.15 [定义扩展], page 258)，并当不符合限制的时候，将指令模式变成失败。

将生成的insn写成一个parallel，其元素为一个从适当内存位置到寄存器的set（可能还需要use或元素clobber）。使用match_parallel (see Section 16.4 [RTL 模板], page 203)来识别insn。关于使用该insn模式的例子，可以参见`rs6000.md'。

`store_multiple'
类似于`load_multiple'，不过是将多个连续的寄存器存储到连续的内存位置。操作数0为连续内存位置的第一个，操作数1为第一个寄存器，操作数2为常量：连续寄存器的数目。

`vec_setm' 设置向量中给定的域。操作数0为要修改的向量，操作数1为域的新值，操作数2指定了域的索引。

`vec_extractm'
从向量中抽取给定的域。操作数1为向量，操作数2指定了域索引，操作数0为存放值的地方。

`vec_extract_evenm'
从输入向量（操作数1和2）中，抽取偶数元素。操作数2的偶数元素按照它们原来的顺序，连结到操作数1的偶数元素后面。结果存储在操作数0中。输出和输入向量应该具有相同的模式。

`vec_extract_oddm'
从输入向量（操作数1和2）中，抽取奇数元素。操作数2的奇数元素按照它们原来的顺序，连结到操作数1的奇数元素后面。结果存储在操作数0中。输出和输入向量应该具有相同的模式。

`vec_interleave_highm'
将两个输入向量的高部分元素合并到输入向量中。输出和输入向量应该具有相同的模式(N个元素)。第一个输入向量的高N/2个元素被第二个输入向量的高N/2个元素交错的插入。

`vec_interleave_lowm'
将两个输入向量的低部分元素合并到输入向量中。输出和输入向量应该具有相同的模式(N个元素)。第一个输入向量的低N/2个元素被第二个输入向量的低N/2个元素交错的插入。

`vec_initm'
将向量初始化为给定的值。操作数0为要初始化的向量，操作数1并行的包含每个域的值。

`pushm1' 输出一个push指令。操作数0是压栈的值。只有当PUSH_ROUNDING被定义时，才被使用。出于历史原因，该指令模式可以缺失，这种情况下使用mov扩展来替代，并用MEM表达式来形成压栈运算。mov扩展方法不被推荐。

`addm3' 操作数2加上操作数1，将结果存储在操作数0中。所有操作数必须具有机器模式m。这也可以用于两地址机器上，通过约束来要求操作数1和0为相同的位置。

`ssaddm3', `usaddm3'
`subm3', `sssubm3', `ussubm3'
`mulm3', `ssmulm3', `usmulm3'
`divm3', `ssdivm3'
`udivm3', `usdivm3'
`modm3', `umodm3'
`uminm3', `umaxm3'
`andm3', `iorm3', `xorm3'
类似的，用于其它算术运算。

`sminm3', `smaxm3'
有符号的最小值和最大值运算。当用于浮点，如果两个操作数都为零，或者有一个为NaN，则没有明确指定哪个操作数作为结果返回。

`reduc_smin_m', `reduc_smax_m'
查找向量的有符号最小/最大元素。向量为操作数1，标量结果存放在操作数0（也是一个向量）的最小有效位。输出和输入向量应该具有相同的模式。

`reduc_umin_m', `reduc_umax_m'
查找向量的无符号最小/最大元素。向量为操作数1，标量结果存放在操作数0（也是一个向量）的最小有效位。输出和输入向量应该具有相同的模式。

`reduc_splus_m'
计算向量的有符号元素的和。向量为操作数1，标量结果存放在操作数0（也是一个向量）的最小有效位。输出和输入向量应该具有相同的模式。

`reduc_uplus_m'
计算向量的无符号元素的和。向量为操作数1，标量结果存放在操作数0（也是一个向量）的最小有效位。输出和输入向量应该具有相同的模式。

`sdot_prodm'
`udot_prodm'
计算两个有符号/无符号元素乘积的和。操作数1和2为相同的模式。它们的乘积，为一个宽的模式，被计算并增加到操作数3上。操作数3的机器模式等于或宽于乘积的机器模式。结果被放在操作数0上，其与操作数3具有相同的机器模式。

`ssum_widenm3'
`usum_widenm3'
操作数0和2具有相同的机器模式，其比操作数1的宽。将操作数1加上操作数2，并将加宽的结果放在操作数0中。（这用于表示元素累加到一个更宽模式的累加器中）

`vec_shl_m', `vec_shr_m'
整个向量向左/向右移位。操作数1为被移位的向量。操作数2为移位的位数。操作数0为移位后的结果向量被存储的地方。输出和输入向量应该具有相同的模式。

`vec_pack_trunc_m'
变窄（降级）并合并两个向量的元素。操作数1和2为具有相同机器模式，N个，大小为S的整数或者浮点元素的向量。操作数0为结果向量，通过使用截取的方式，将它们变窄并连接成，2×N个大小为N/2的元素。

`vec_pack_ssat_m', `vec_pack_usat_m'
变窄（降级）并合并两个向量的元素。操作数1和2为具有相同机器模式，N个，大小为S的整数元素的向量。操作数0为结果向量，通过使用有符号/无符号饱和算术的方式，将它们变窄并连接成。

`vec_pack_sfix_trunc_m', `vec_pack_ufix_trunc_m'
将两个向量的元素变窄，转成有符号/无符号整数类型并合并。操作数1和2为具有相同机器模式，N个，大小为S的浮点元素的向量。操作数0为结果向量，由2×N个大小为N/2的元素连接而成。

`vec_unpacks_hi_m', `vec_unpacks_lo_m'
抽取并变宽（提升），具有有符号整数或浮点元素的，向量的高/低部分。输入向量（操作数1）具有N个大小为S的元素。使用有符号或者浮点扩展，将向量的高/低元素进行变宽（提升），并将结果，N/2个大小为2*S的值，放在输出向量（操作数0）中。

`vec_unpacku_hi_m', `vec_unpacku_lo_m'
抽取并变宽（提升），具有无符号整数元素的，向量的高/低部分。输入向量（操作数1）具有N个大小为S的元素。使用零扩展，将向量的高/低元素进行变宽（提升），并将结果，N/2个大小为2*S的值，放在输出向量（操作数0）中。

`vec_unpacks_float_hi_m', `vec_unpacks_float_lo_m'
`vec_unpacku_float_hi_m', `vec_unpacku_float_lo_m'
抽取具有有符号/无符号整数元素的，向量的高/低部分，并转换成浮点类型。输入向量（操作数1）具有N个大小为S的元素。使用浮点转换，将向量的高/低元素进行转换，并将结果，N/2个大小为2*S的值，放在输出向量（操作数0）中。

`vec_widen_umult_hi_m', `vec_widen_umult_lo_m'
`vec_widen_smult_hi_m', `vec_widen_smult_lo_m'
有符号/无符号加宽乘法。两个输入（操作数1和2）为，N个大小为S的有符号/无符号元素的向量。将两个向量的高/低元素相乘，并将N/2个大小为2×S的乘积放在输出向量（操作数0）中。

`mulhisi3' 机器模式为HImode的操作数1和2相乘，并将SImode乘积放在操作数0中。

`mulqihi3', `mulsidi3'
类似的其它宽度的加宽乘法指令。

`umulqihi3', `umulhisi3', `umulsidi3'
类似的加宽乘法指令，进行无符号乘法。

`usmulqihi3', `usmulhisi3', `usmulsidi3'
类似的加宽乘法指令，将第一个操作数解析为无符号的，第二个为有符号的，然后进行有符号乘法。

`smulm3_highpart'
对机器模式为m的操作数1和2进行有符号乘法，并将乘积的最高有效的一半放在操作数0中。乘积的最低有效的一半被丢弃。

`umulm3_highpart'
类似的，只不过乘法是无符号的。

`maddmn4' 将操作数1和2进行相乘，有符号扩展成机器模式n，加上操作数3，并将结果存放在操作数0中。操作数1和2具有机器模式m，操作数0和3具有机器模式n。两种机器模式必须都为整数或者浮点模式，并且n必须为m的两倍大小。

换句话说，maddmn4类似于mulmn3，只不过其还加上操作数3。

这些指令不允许执行FAIL。

`umaddmn4'
类似maddmn4，只不过零扩展乘法操作数，而不是有符号扩展它们。

`ssmaddmn4'
类似maddmn4，不过所有的运算都必须是有符号饱和的。

`usmaddmn4'
类似umaddmn4，不过所有的运算都必须是无符号饱和的。

`msubmn4' 将操作数1和2相乘，有符号扩展为机器模式n，减去操作数3，并将结果存放在操作数0中。操作数1和2具有机器模式m，操作数0和3具有机器模式n。两种机器模式必须都为整数或者浮点模式，并且n必须为m的两倍大小。

换句话说，msubmn4类似于mulmn3，只不过其还减去操作数3。

这些指令不允许执行FAIL。

`umsubmn4'
类似msubmn4，不过零扩展乘法操作数，而不是有符号扩展它们。

`ssmsubmn4'
类似msubmn4，不过所有的运算都必须是有符号饱和的。

`usmsubmn4'
类似umsubmn4，不过所有的运算都必须是无符号饱和的。

`divmodm4' 有符号除法，同时产生商和余数。操作数1被操作数2除，产生商存储在操作数0中，余数存储在操作数3中。

对于具有同时产生商和余数的指令的机器，提供`divmodm4'指令模式，但不要提供`divm3'和`modm3'。这使得当商和余数都被计算的时候，可以优化成相对常见的情况。

如果存在只产生商或者余数的指令，并且比都产生的指令更有效，则将`divmodm4'的输出例程写成调用find_reg_note，查看商或者余数的REG_UNUSED注解，来产生适当的指令。

`udivmodm4'
类似的，不过进行无符号除法。

`ashlm3',`ssashlm3',`usashlm3'
将操作数1向左算术移位，左移位数由操作数2指定，将结果存储在操作数0中。这里m为操作数0和1的机器模式；操作数2的机器模式通过指令模式来指定，编译器会在生成指令之前，将操作数转换成该模式。超出范围的

移位数目的含义，可以通过TARGET_SHIFT_TRUNCATION_MASK来指定。 See [TARGET_SHIFT_TRUNCATION_MASK], page 391. 操作数2总是一个标量类型。

`ashrm3', `lshrm3', `rotlm3', `rotrm3'
其它移位或者旋转指令，类似于ashlm3指令。操作数2总是一个标量类型。

`vashlm3', `vashrm3', `vlshrm3', `vrotlm3', `vrotrm3'
向量移位和旋转指令，操作数2为向量，而不是标量类型。

`negm2', `ssnegm2', `usnegm2'
对操作数1求负，并将结果存放在操作数0中。

`absm2' 将操作数1的绝对值存放在操作数0中。

`sqrtm2' 将操作数1的平方根存储在操作数0中。

sqrt内建C函数，总是使用对应于C数据类型double的机器模式，sqrtf内建函数使用对应于C数据类型float的机器模式。

`fmodm3' 将操作数1除以操作数2的余数，存储在操作数0中，并向零方向舍入为整数。

fmod内建C函数总是使用对应于C数据类型double的机器模式，fmodf内建函数使用对应于C数据类型float的机器模式。

`remainderm3'
将操作数1除以操作数2的余数，存储在操作数0中，并舍入为最接近的整数。

remainder内建C函数总是使用对应于C数据类型double的机器模式，remainderf内建函数使用对应于C数据类型float的机器模式。

`cosm2' 将操作数1的余弦存放在操作数0中。

cos内建C函数总是使用对应于C数据类型double的机器模式，cosf内建函数使用对应于C数据类型float的机器模式。

`sinm2' 将操作数1的正弦存放在操作数0中。

sin内建C函数总是使用对应于C数据类型double的机器模式，sinf内建函数使用对应于C数据类型float的机器模式。

`expm2' 将操作数1的幂存放在操作数0中。

exp内建C函数总是使用对应于C数据类型double的机器模式，expf内建函数使用对应于C数据类型float的机器模式。

`logm2' 将操作数1的自然对数存放在操作数0中。

log内建C函数总是使用对应于C数据类型double的机器模式，logf内建函数使用对应于C数据类型float的机器模式。

`powm3' 将操作数1的，指数为操作数2的幂值存放在操作数0中

pow内建C函数总是使用对应于C数据类型double的机器模式，powf内建函数使用对应于C数据类型float的机器模式。

`atan2m3' 将操作数1除以操作数2的反正切，存放在操作数0中，使用两个参数的正负符号来确定结果的商。

atan2内建C函数总是使用对应于C数据类型double的机器模式，atan2f内建函数使用对应于C数据类型float的机器模式。

`floorm2' 存储不大于参数的最大整数值。

floor内建C函数总是使用对应于C数据类型double的机器模式，floorf内建函数使用对应于C数据类型float的机器模式。

`btruncm2' 存储将参数向零方向舍入的整数。

trunc内建C函数总是使用对应于C数据类型double的机器模式，truncf内建函数使用对应于C数据类型float的机器模式。

`roundm2' 存储将参数向远离零的方向舍入的整数。

round内建C函数总是使用对应于C数据类型double的机器模式，roundf内建函数使用对应于C数据类型float的机器模式。

`ceilm2' 存储将参数向远离零的方向舍入的整数。

ceil内建C函数总是使用对应于C数据类型double的机器模式，ceilf内建函数使用对应于C数据类型float的机器模式。

`nearbyintm2'

将参数根据缺省的舍入模式，舍入为整数。

nearbyint内建C函数总是使用对应于C数据类型double的机器模式，nearbyintf内建函数使用对应于C数据类型float的机器模式。

`rintm2' 将参数根据缺省的舍入模式，舍入为整数，并且当结果与参数的值不同的时候，抛出不精确异常。

rint内建C函数总是使用对应于C数据类型double的机器模式，rintf内建函数使用对应于C数据类型float的机器模式。

`lrintmn2'

将操作数1（对于浮点模式m有效）转换成定点机器模式n，作为有符号数，根据当前的舍入模式，并存储在操作数0（具有机器模式n）中。

`lroundm2' 将操作数1（对于浮点模式m有效）转换成定点机器模式n，舍入到最近的，远离零方向的有符号数，并存储在操作数0（具有机器模式n）中。

`lfloorm2' 将操作数1（对于浮点模式m有效）转换成定点机器模式n，向下舍入成有符号数，并存储在操作数0（具有机器模式n）中。

`lceilm2' 将操作数1（对于浮点模式m有效）转换成定点机器模式n，向上舍入成有符号数，并存储在操作数0（具有机器模式n）中。

`copysignm3'

将操作数1的数量级和操作数的符号组成的值，存放在操作数0中。

copysign内建C函数总是使用对应于C数据类型double的机器模式，copysignf内建函数使用对应于C数据类型float的机器模式。

`ffsm2' 将操作数1的最小有效，置1的位的索引，加上1，存放在操作数0中。如果操作数1为零，则存储零。m为操作数0的机器模式；操作数1的机器模式由指令模式指定，编译器会在生成指令之前，将操作数转成该机器模式。

ffs内建C函数总是使用对应于C数据类型int的机器模式。

`clzm2' 将x中，从最高有效位开始，起始处置0的位的数目，存放在操作数0中。如果x为0，则CLZ_DEFINED_VALUE_AT_ZERO (see Section 17.30 [其它], page 389)宏定义了结果是否为未定义或者一个有用的值。m为操作数0的机器模式；操作数1的机器模式由指令模式指定，编译器会在生成指令之前，将操作数转成该机器模式。

`ctzm2' 将x中，从最小有效位开始，结尾处置0的位的数目，存放在操作数0中。如果x为0，则CTZ_DEFINED_VALUE_AT_ZERO (see Section 17.30 [其它], page 389)宏定义了结果是否为未定义或者一个有用的值。m为操作数0的机器模式；操作数1的机器模式由指令模式指定，编译器会在生成指令之前，将操作数转成该机器模式。

`popcountm2' 将x中置1的位的数目，存放在操作数0中。m为操作数0的机器模式；操作数1的机器模式由指令模式指定，编译器会在生成指令之前，将操作数转成该机器模式。

`paritym2' 将x的奇偶校验存放在操作数0中，即：x中置1的位数对2求模。m为操作数0的机器模式；操作数1的机器模式由指令模式指定，编译器会在生成指令之前，将操作数转成该机器模式。

`one_cmplm2' 对操作数1进行按位求补，并存放在操作数0中。

`cmpm' 比较操作数0和1，并设置条件代码。RTL指令模式应该像这样：

```
(set (cc0) (compare (match_operand:m 0 ...)
                    (match_operand:m 1 ...)))
```

`tstm' 将操作数0与零进行比较，并设置条件码。RTL指令模式应该像这样：

```
(set (cc0) (match_operand:m 0 ...))
```

`tstm'指令模式不应该为不使用(cc0)的机器定义。这样做会使得编译器变得迷惑，因为其将会不清楚哪一个set操作为比较。应该使用`cmpm'。

`movmemm' 块移动指令。内存的目的块和源块为前两个操作数，都为地址是Pmode的mem:BLK。

要移动的字节数为第三个操作数，机器模式为m。通常，你会将m指定为word_mode。然而，如果你可以生成更好的代码，知道有效长度的范围比一整个字要小，则你应该提供一个指令模式，其机器模式对应于你可以更有效的处理的值的范围（例如，QImode对于范围0--127；注意我们回避了负数），并且一个使用word_mode的指令模式。

第四个操作数为已知的源和目的的共享对齐，形式为一个const_int rtx。因此，如果编译器知道源和目的都是字对齐的，则其可以为该操作数提供值4。

可选的操作数5和6，分别指定了期望的对齐方式和块的大小。期望的对齐方式不同于操作数4中的对齐方式，块并不要求所有的情况下都按照这样对齐。期望的对齐方式也是以字节为单位，类似于操作数4。期望的大小，当不知道的时候，被设置为(const_int -1)。

描述多个movmemm指令模式，只有当对于更小的机器模式的指令模式，对操作数1，2，4具有更少限制的时候，才会获利。注意movmemm中的机器模式m不对块中单独的被移动的数据单元的机器模式做任何限制。

这些指令模式不需要对源和目的可能重叠的情况，进行特殊的考虑。

`movstr' 字符串复制指令，具有stpcpy的语义。操作数0为输出操作数，机器模式为Pmode。目的字符串和源字符串的地址为操作数1和2，都是地址为Pmode的mem:BLK。对该指令模式的执行，应该将地址存放在操作数0中，其中NUL终结符存放在目标字符串中。

`setmemm' 块设置指令。目的字符串为第一个操作数，作为一个mem:BLK，其地址的机器模式为Pmode。被设置的字节的数目是第二个操作数，机器模式为m。用于初始化内存的值为第三个操作数。只支持清空内存的目标机应该拒绝任何不为常数0的值。关于对机器模式选择的讨论，参见`movmemm'。

第四个操作数为目标的已知对齐方式，形式为const_int rtx。因此，如果编译器知道目的操作数是字对齐的，则其可以为该操作数提供值4。

可选的操作数5和6，分别指定了期望的对齐方式和块的大小。期望的对齐方式不同于操作数4中的对齐方式，块并不要求所有的情况下都按照这样对齐。期望的对齐方式也是以字节为单位，类似于操作数4。期望的大小，当不知道的时候，被设置为(const_int -1)。

对多个setmemm的使用，类似于movmemm

`cmpstrnm' 字符串比较指令，有5个操作数。操作数0为输出，机器模式为m。剩下的4个操作数类似于`movmemm'的操作数。两个指定的内存块按字节来进行比较，按照字典顺序，从每个字符串的起始处开始。指令不允许一次取多个字节，因为每个字符串都可能在第一个字节中终止，读取后面的字节可能会访问一个无效的页或者段，并产生一个缺失。该指令的效果是将值存放在操作数0中，其符号表示了比较的结果。

`cmpstrm' 字符串比较指令，不知道最大的长度。操作数0为输出，机器模式为m。第二个和第三个操作数为被比较的内存块；都是机器模式为Pmode的mem:BLK。

第四个操作数为源和目的的已知共享的对齐方式，形式为const_int rtx。因此，如果编译器知道源和目的都是字对齐的，则其可以为该操作数提供值4。

两个指定的内存块按字节来进行比较，按照字典顺序，从每个字符串的起始处开始。指令不允许一次取多个字节，因为每个字符串都可能在第一个字节中终止，读取后面的字节可能会访问一个无效的页或者段，并产生一个缺失。该指令的效果是将值存放在操作数0中，其符号表示了比较的结果。

`cmpmemm' 块比较指令，这5个操作数类似于`cmpstrm'。两个指定的内存块按字节来进行比较，按照字典顺序，从每个字符串的起始处开始。不像`cmpstrm'，该指令可以在两个内存块中取任意个字节。该指令的效果是将值存放在操作数0中，其符号表示了比较的结果。

`strlenm' 计算字符串的长度，有3个操作数。操作数0为结果（机器模式为m），操作数1为一个mem，指出字符串的第一个字符，操作数2为要查找的字符（通常为零），操作数3为一个常量，描述了字符串起始处的已知对齐方式。

`floatmn2'
将有符号整数，操作数1（对于定点机器模式m有效），转换成浮点机器模式n，并存放在操作数0（机器模式为n）中。

`floatunsmn2'
将无符号整数，操作数1（对于定点机器模式m有效），转换成浮点机器模式n，并存放在操作数0（机器模式为n）中。

`fixmn2' 将操作数1（对于浮点机器模式m有效），转换成定点机器模式n，作为一个有符号数并存放在操作数0（机器模式为n）中。该指令的结果，只有当操作数1的值为整数时，才被定义。

如果机器描述定义了该指令模式，则其还需要定义ftrunc指令模式。

`fixunsmn2'
将操作数1（对于浮点机器模式m有效），转换成定点机器模式n，作为一个无符号数并存放在操作数0（机器模式为n）中。该指令的结果，只有当操作数1的值为整数时，才被定义。

`ftruncm2' 将操作数1（对于浮点机器模式m有效），转换成整数值，仍按照浮点机器模式m来表示，并存放在操作数0（对于浮点机器模式m有效）中。

`fix_truncmn2'
类似于`fixmn2'，不过工作于，将任意机器模式为m的浮点值，转换成整数。

`fixuns_trunc*mn*2'
类似`fixuns*mn*2'，不过工作于，将任意机器模式为m的浮点值，转换成整数。

`trunc*mn*2'
将操作数1（对机器模式m有效），截取为机器模式n，并存放在操作数0（机器模式为n）中。两个机器模式都必须同为定点的或者浮点的。

`extend*mn*2'
将操作数1（对机器模式m有效）符号扩展成机器模式n，并存放在操作数0（机器模式为n）中。两个机器模式都必须同为定点的或者浮点的。

`zero_extend*mn*2'
将操作数1（对机器模式m有效）零扩展成机器模式n，并存放在操作数0（机器模式为n）中。两个机器模式都必须同为定点的或者浮点的。

`fract*mn*2'
将机器模式为m的操作数1，转换成机器模式n，并存放在操作数0（机器模式为n）中。机器模式m和n可以为定点到定点，有符号整数到定点，定点到有符号整数，浮点到浮点，或者定点到浮点。当发生溢出时，结果未定义。

`satfract*mn*2'
将机器模式为m的操作数1，转换成机器模式n，并存放在操作数0（机器模式为n）中。机器模式m和n可以为定点到定点，有符号整数到定点，或者浮点到定点。当发生溢出时，指令将结果饱和为最大或最小值。

`fractuns*mn*2'
将机器模式为m的操作数1，转换成机器模式n，并存放在操作数0（机器模式为n）中。机器模式m和n可以为无符号整数到定点，或者定点到无符号整数。当发生溢出时，结果未定义。

`satfractuns*mn*2'
将机器模式为m的，无符号整数，操作数1，转换成定点机器模式n，并存放在操作数0（机器模式为n）中。当发生溢出时，指令将结果饱和为最大或最小值。

`extv'
从操作数1（寄存器或者内存操作数）中抽取一个位域，其中操作数2指定了宽度，按位为单位，操作数3为起始位，并将结果存放在操作数0中。操作数0必须具有机器模式word_mode。操作数1可以具有机器模式byte_mode 或 word_mode；通常word_mode只允许用于寄存器。操作数2和3必须对word_mode有效。

RTL生成过程，生成的该指令，操作数2和3为常量，并且对于操作数2，常量不为零。

位域的值，在存放到操作数0之前，被有符号扩展为一整个字的整数。

`extzv'
类似`extv'，只不过位域的值被零扩展。

`insv'
将操作数3（必须对word_mode有效）存储到操作数0中的位域，其中操作数1指定了位宽，操作数2指定了起始位。操作数0可以具有机器模式byte_mode 或 word_mode；通常word_mode只允许用于寄存器。操作数1和2必须对word_mode有效。

RTL生成过程，生成的该指令，操作数1和2为常量，并且对于操作数1，常量不为零。

`mov*mode*cc'
根据对操作数1的比较，有条件的将操作数2或者3移送到操作数0中。如果比较为真，则操作数2被移送到操作数0中，否则操作数3被移送。

操作数的机器模式不需要与被移送的操作数的相同。一些机器，例如sparc64，具有可以根据浮点条件码，条件移送整数值的指令，以及相反的指令。

如果机器没有条件移送指令，则不要定义这些指令模式。

`addmodecc'
类似于`movmodecc'，不过是条件加法。根据在操作数1中的比较，条件性的将操作数2或者(操作数2 + 操作数3)，移送到操作数0中。如果比较为真，则操作数2被移送到操作数0中，否则(操作数2 + 操作数3)被移送。

`scond'
根据条件码将零或者非零存放在操作数中。当且仅当条件cond为真时，存储的值才为非零。cond为一个比较运算表达式代码的名字，例如eq, lt或leu。

当书写match_operand表达式的时候，你来指定操作数必须具有的机器模式。编译器自动的查看你使用的机器模式，并提供那个机器模式的操作数。

对于条件为真时所存储的值，其低位必须为1，不然必须为负。否则，指令就不适合，你应该从机器描述中将其去掉。你可以通过定义宏STORE_FLAG_VALUE (see Section 17.30 [其它], page 389)，来描述哪个值被存放。如果不能找到一个用于所有`scond'指令模式的描述，则你应该从机器描述中去掉这些操作。

这些操作可以失败，但应该只在相对不常见的情况下这样做；如果它们对于常见的情况，包括整数比较，会失败，则最好去掉这些指令模式。

如果这些操作被去掉，则编译器通常会生成，将常量复制到目标，并在将零赋值给目标的语句附近进行分支跳转。如果这样的代码比用于`scond'模式的指令，后面跟着需要将结果转成SImode的1或者零的的指令，更有效，则你应该从机器描述中去掉`scond'操作。

`bcond'
条件分支指令。操作数0为一个label_ref指出要跳转到的标号。如果条件码符合条件cond则跳转。

一些机器不遵循这里假设的模型，即一个比较指令，跟随一个条件跳转指令。那种情况下，`cmpm' (和 `tstm')指令模式，应该简单的将操作数存放开，并在define_expand (see Section 16.15 [定义扩展], page 258)中为条件分支操作，生成所有需要的insn。所有对扩展`bcond'指令模式的调用，都会立即优先执行对扩展`cmpm'或者`tstm'的调用。

对条件代码值使用伪寄存器的，或者用于比较的机器模式取决于被测试的条件的机器，也应该使用上面的机制。See Section 16.12 [跳转指令模式], page 254.

上面的讨论也应用在`movmodecc'和`scond'指令模式上。

`cbranchmode4'
条件分支指令，结合一个比较指令。操作数0为比较运算符。操作数1和2分别为比较运算的第一个和第二个操作数。操作数3为一个label_ref，指出了跳转的标号。

`jump'
函数内部跳转；无条件分支。操作数0为一个label_ref，指出了跳转的标号。该指令模式名在所有机器上都是强制必须的。

`call'
没有返回值的子程序调用指令。操作数0为调用的函数；操作数1为压栈的参数的字节数，为一个const_int；操作数2为用作操作数的寄存器数目。

在大多机器上，操作数2没有被实际存放在RTL模式中。提供它是出于安全考虑，一些RISC机器需要将该信息放到汇编代码中；它们可以将其放在RTL中，而不是操作数1中。

操作数0应该为一个mem RTX，其地址为函数的地址。然而注意，该地址可以为一个symbol_ref表达式，即使其在目标机器上可能不是一个合法的内存地址。如果其也不是调用指令的有效参数，则该操作的指令模式应该为一个define_expand (see Section 16.15 [定义扩展], page 258)，其将地址放入寄存器中，并在调用指令中使用寄存器。

`call_value'

有返回值的子程序调用指令。操作数0为硬件寄存器，存放返回值。还有三个操作数，与`call'指令相同（只不过将编号加一）。

返回 BLKmode对象的子程序，使用`call' insn。

`call_pop', `call_value_pop'

类似于`call' 和 `call_value'，只不过用于其被定义，并且RETURN_POPS_ARGS为非零的时候。它们应该生成一个 parallel，包含函数调用和一个set，来指示对帧指针的调整。

对于RETURN_POPS_ARGS可以为非零的机器，使用这些指令模式可以增加帧指针被消除掉的函数的数目。

`untyped_call'

返回一个任意类型的值的子函数调用指令。操作数0为调用的函数；操作数1为内存位置，存放调用函数后的结果；操作数2为一个parallel表达式，其中每个元素都为一个set表达式，用来指示将函数返回值保存到结果块中。

该指令模式应该被定义，来支持__builtin_apply，在一些机器上，需要特殊的指令来调用一个具有任意参数的子程序，或者将返回值保存。在具有多个寄存器，可以存放一个返回值（即FUNCTION_VALUE_REGNO_P对多个寄存器都为真）的机器上，需要该指令模式。

`return' 子程序返回指令。该指令模式名应该只有当，单个指令可以做从函数中返回时的所有工作的时候，才被定义。

类似`movm'指令模式，该指令模式也在RTL生成阶段之后被使用。这种情况下，其用来支持一些机器，从函数中返回通常需要多个指令，但是某些类别的函数只需要一条指令来实现返回。通常，可以适用的函数为那些不需要保存任何寄存器或者分配栈空间的函数。

对于这样的机器，该指令模式中指定的条件，应该只有当reload_completed为非零的时候才为真，并且函数的尾声应该只为一单个指令。对于有寄存器窗口的机器，例程leaf_function_p可以用来确定是否需要对寄存器窗口压栈。

具有条件性返回指令的机器，应该将指令模式定义成

```
(define_insn ""
 [(set (pc)
    (if_then_else (match_operator
            0 "comparison_operator"
            [(cc0) (const_int 0)])
           (return)
           (pc)))]
 "condition"
 "...")
```

其中condition通常为，在`return'指令模式中指定的相同的条件。

`untyped_return'

未定义类型的子程序返回指令。该指令模式应该被定义，来支持__builtin_return，在一些机器上，需要特殊的指令来返回一个任意类型的值。

操作数0为一个内存位置，存放使用__builtin_apply调用函数的结果；操作数1为一个parallel表达式，每个元素都是一个set表达式，指示了从结果块中恢复函数的返回值。

`nop' 空操作指令。该指令模式名应该总是被定义，用来在汇编代码中输出一个no-op。(const_int 0)将作为一个RTL指令模式。

`indirect_jump'
一个指令，跳转到操作数0表示的地址。该指令模式名在所有机器上都必须存在。

`casesi' 通过派遣表进行跳转的指令，包括边界检查。该指令接受五个操作数：

1. 派遣的索引，具有机器模式SImode。
2. 表中索引的较低边界，一个整数常量。
3. 表中索引的整个范围---最大索引减去最小的。
4. 位于表之前的标号。
5. 一个标号，如果索引值超出边界，则跳转到该地方。

表为jump_insn中的一个addr_vec 或 addr_diff_vec。表中的元素个数为一加上上界和下界的差。

`tablejump' 跳转到一个可变地址的指令。这是一个低级别的能力，可以用来实现一个派遣表，当没有`casesi'指令模式的时候。

该指令模式需要两个操作数：地址或偏移量，以及一个标号，其直接位于跳转表的前面。如果宏CASE_VECTOR_PC_RELATIVE求值为一个非零值，则第一个操作数为一个偏移量，其从表的地址开始计算；否则，其为一个跳转的绝对地址。这两种情况下，第一个操作数都为Pmode。

`tablejump' insn总是其使用的跳转表之前的最后一个insn。其汇编代码通常不需要用到第二个操作数，但是你应该在RTL指令模式中包含它，使得跳转优化不会将表作为不可到达代码删除。

`decrement_and_branch_until_zero'
条件分支指令，递减一个寄存器并且如果寄存器非零则跳转。操作数0为递减并测试的寄存器；操作数1为如果寄存器非零，则跳转的标号。See Section 16.13 [循环指令模式], page 255.

该可选的指令模式只用于合并器，通常被循环优化器使用，当启动强度消减的时候。

`doloop_end'
条件分支指令，递减一个寄存器，并且如果寄存器非零则跳转。该指令接受五个操作数：操作数0是用来递减和测试的寄存器；操作数1是循环迭代的次数，为一个const_int，或者如果直到运行时才能确定，则为const0_rtx；操作数2为实际的或者估算的最大迭代数，为一个const_int；操作数3为被包含的循环数，为一个const_int（最内层循环的值为1）；操作数4为如果寄存器非零，要跳转的标号。See Section 16.13 [循环指令模式], page 255.

该可选的指令模式应该为，具有低开销循环指令的机器定义，循环优化器会尝试修改合适的循环来利用它。如果不支持嵌套的低开销循环，则使用define_expand (see Section 16.15 [定义扩展], page 258)，并如果操作数3不为const1_rtx，则使得指令模式失败。类似的，如果实际的或者估算的最大迭代数目对于该指令来说太大，则使其失败。

`doloop_begin'
与doloop_end成套的指令，被用于需要执行一些初始化的机器，例如加载用于低开销循环指令中的特定寄存器。如果初始化insn不总是需要被生成，则使用define_expand (see Section 16.15 [定义扩展], page 258)，并使其失败。

`canonicalize_funcptr_for_compare'
正规化操作数1中的函数指针，并将结果存放在操作数0中。

操作数0总是一个reg，并具有机器模式Pmode；操作数1可以为一个reg, mem, symbol_ref, const_int等等，也具有机器模式Pmode。

正规化一个函数指针，通常涉及到计算函数的地址，该函数指针用在间接调用中。

只有当目标机器上，对于函数指针可以有不同的值，但是当在间接调用的时候，其还是调用相同的函数的时候，才定义该指令模式。

`save_stack_block'
`save_stack_function'
`save_stack_nonlocal'
`restore_stack_block'
`restore_stack_function'
`restore_stack_nonlocal'

大多数机器用来保存和恢复栈指针的方式，是通过将其复制到一个机器模式为Pmode的对象。不要在这样的机器上定义这些指令模式。

一些机器要求对栈指针的保存和恢复，进行特殊的处理。在那些机器上，根据非标准的情况来定义指令模式，使用define_expand (see Section 16.15 [定义扩展], page 258)来产生要求的insn。三种保存和恢复类型：

1. `save_stack_block'将栈指针存放在用来分配可变大小的对象块的起始处，`restore_stack_block'当退出块的时候恢复栈指针。
2. `save_stack_function' 和 `restore_stack_function'为函数最外层的快做类似的工作，并用于当函数分配可变大小的对象或者调用alloca的时候。只有尾声使用被恢复的栈指针，这使得在一些机器上，可以有更简单的保存或恢复序列。
3. `save_stack_nonlocal'用在包含嵌套函数分支跳转标号的函数中。其保存栈指针的方式为，最内层函数可以使用`restore_stack_nonlocal'来恢复栈指针。编译器生成代码，用来恢复帧寄存器和参数指针寄存器，但是一些机器要求保存和恢复额外的数据，例如寄存器窗口信息或者栈后退链。在这些指令模式中放置保存和恢复这些要求的数据的insn。

当保存栈指针时，操作数0是保存区域，操作数1是栈指针。用于分配保存区域的机器模式缺省为Pmode，不过你可以通过定义STACK_SAVEAREA_MODE宏(see Section 17.5 [存储布局], page 291)来覆盖该选择。你必须指定一个整数机器模式，或者VOIDmode，如果对于特定的类型不需要保存区域（或者因为没有需要保存的，或者因为可以使用机器特定的保存区域）。操作数0为栈指针，操作数1为用于恢复操作的保存区域。如果`save_stack_block'被定义，则操作数0一定不能为VOIDmode，因为这些保存操作数可以被任意的嵌套。

当栈指针被保存，是用于非局部goto，则保存区域为一个mem，为一个相对于virtual_stack_vars_rtx的常量偏移，其它两种情况下，保存区域为一个reg。

`allocate_stack'

从栈指针中减去（或者增加，如果STACK_GROWS_DOWNWARD未定义）操作数1，来为动态分配的数据创建空间。将由此产生的指向该空间的指针存放在操作数0中。如果你是从主栈中分配空间，则可以通过生成一个insn，将virtual_stack_dynamic_rtx复制到操作数0中。如果你是从其它地方分配空间，则可以生成将该空间的位置复制到操作数0中的代码。对于后者情况，你必须确保该空间当主栈中对应的空间被释放的时候，其也被释放。

如果所有需要做的事情只是减法操作，则不用定义该指令模式。一些机器还要求其它的操作，例如栈探测，或者维护后向链。定义该指令模式除了更新栈指针之外的，来生成那些操作。

`check_stack'
如果在你的系统上，不能通过使用加载或者存储指令(see Section 17.10.3 [栈检查], page 320)探测栈，从而进行栈检查，则定义该指令模式来执行所需要的检查，并且如果栈已经溢出则产生一个错误信号。有单个操作数，为栈中从当前栈指针开始，最远的栈位置。通常，在需要该指令模式的机器上，你将从一个全局的或者线程特定的变量或者寄存器中获得栈的限制。

`nonlocal_goto'
生成产生一个非局部goto的代码，例如，从一个函数跳转到一个外部函数的标号。该指令模式有四个参数，每个参数表示一个在跳转中用到的值。第一个参数被加载到帧指针中，第二个为分支跳转的地址，第三个为栈被保存的地址，最后一个为标号的地址，放在静态链中。

在大多上机器上，你不需要定义该指令模式，因为GCC会产生正确的代码，用来加载帧指针和静态链，恢复栈（使用`restore_stack_nonlocal'指令模式，如果定义），并间接跳转。你只有当该代码在你的机器上不工作的情况下，才需定义该指令模式。

`nonlocal_goto_receiver'
该指令模式，如果定义，包含了非局部goto的目标处所需要的代码。通常不需要定义该指令模式。通常需要该指令模式的原因是，如果一些值，例如全局表的指针，必须在帧指针被恢复的时候，其也被恢复。注意，非局部goto，只出现在一个转换单元中，所以被给定模块的所有函数共享的全局表指针，不需要被恢复。该指令模式没有参数。

`exception_receiver'
该指令模式，如果被定义，包含了在一个异常处理的地方所需要的代码，其在非局部goto的地方不需要。通常你不需要定义该指令模式。一个典型的，你可以需要定义该指令模式的原因是，如果某个值，例如指向全局表的指针，必须在控制流分支跳转到异常处理之后，被恢复。该指令模式没有参数。

`builtin_setjmp_setup'
该指令模式，如果被定义，则包含了需要初始化jmp_buf的代码。你通常不需要定义该指令模式。一个典型的，你可以需要定义该指令模式的原因是，如果某个值，例如指向全局表的指针，必须被恢复。尽管如此，还是推荐指针值如果可能（例如，给定一个标号的地址），则应被重新计算。有一单个操作数，

`builtin_setjmp_receiver'
该指令模式，如果被定义，包含了在内建setjmp的地方，并且在非局部goto的地方不需要的代码。你通常不需要定义该指令模式。一个典型的，你可以需要定义该指令模式的原因是，如果某个值，例如指向全局表的指针，必须被恢复。其接受一个参数，为builtin_longjmp将控制转出的标号；该指令模式可以被生成为对于标号的一个小的偏移。

`builtin_longjmp'
该指令模式，如果被定义，则执行整个longjmp动作。你通常不需要定义该指令模式，除非你还定义了builtin_setjmp_setup。单个操作数为指向jmp_buf的指针。

`eh_return' 该指令模式，如果被定义，则影响__builtin_eh_return的方式，并且调用帧异常处理库函数会被建立。其用于处理异常返回路径所需要的非平凡的动作。

函数应该返回的异常处理的地址，被作为操作数传给该指令模式。其通常需要被指令模式复制到某个特定的寄存器或者内存位置。如果该指令模式需要确定目标调用帧的位置，则可以使用EH_RETURN_STACKADJ_RTX。

如果该指令模式没有被定义，缺省的动作为简单的将返回地址复制给EH_RETURN_HANDLER_RTX。或者宏，或者该指令模式，应该被定义，如果使用了调用帧异常处理。

`prologue' 该指令模式，如果被定义，用来产生函数的入口RTL。函数入口负责设置栈帧，初始化帧指针寄存器，保存被调用者需要保存的寄存器，等等。

使用一个序言指令模式，通常的方式为定义TARGET_ASM_FUNCTION_PROLOGUE来产生序言的汇编代码。

prologue指令模式对于执行指令调度的目标机尤其有用。

`epilogue' 该指令模式为函数的出口生成RTL。函数出口负责撤销栈帧的分配，恢复被调用者所保存的寄存器，并产生返回指令。

使用尾声指令模式，通常的方式为定义TARGET_ASM_FUNCTION_EPILOGUE，来产生尾声的汇编代码。

prologue指令模式对于执行指令调度的，或者它们的返回指令具有延迟槽的目标机，尤其有用。

`sibcall_epilogue'

该指令模式，如果被定义，产生一个函数的出口RTL，并且最终不分支跳转会到调用函数。该指令模式将在任何兄弟调用（即尾调用）地点之前被产生。

sibcall_epilogue指令模式一定不能破坏任何用于传递的参数，或者用于传给当前函数的参数的栈槽。

`trap' 该指令模式，如果被定义，则会发射一个错误信号。在其它地方，其被Java前端使用，来发射“无效的数组索引”异常信号。

`conditional_trap'

条件陷阱指令。操作数0为执行比较的RTL。操作数1为陷阱代码，为一个整数。

典型的conditional_trap指令模式型如：

```
(define_insn "conditional_trap"
 [(trap_if (match_operator 0 "trap_operator"
      [(cc0) (const_int 0)])
      (match_operand 1 "const_int_operand" "i"))]
 ""
 "...")
```

`prefetch'

该指令模式，如果被定义，则产生无故障的数据预取指令代码。操作数0为预取的内存地址。操作数1为常量1，如果预取打算去写一个内存地址，否则为常量0。操作数2为数据的时间局部性的等级，值在0和3之间。0意味着数据没有时间局部性，所以在访问之后不需要留在缓存中；3意味着数据具有高等级的时间局部性，应该尽可能的留在所有级别的缓存中；1和2分别意味着，低等级和中等级的时间局部性。

不支持写预取或者局部性暗示的目标机，可以忽略操作数1和2的值。

`blockage'

该指令模式定义了一个伪insn，用来阻止指令调度器将指令跨越所定义的insn块边界进行移动。通常为一个UNSPEC_VOLATILE指令模式。

`memory_barrier'

如果目标机内存模型不完全同步，则该指令模式应该被定义为一条指令，在期望进行加载和存储的指令之前。该指令模式没有操作数。

`sync_compare_and_swapmode'

该指令模式，如果被定义，产生一个比较并交换的原子操作代码。操作数1为执行原子操作的内存。操作数2为，与当前内存位置的内容进行比较的“旧”值。操作数3为，如果比较成功存放在内存中的“新”值。如果比较成功，其当然为对操作数2的一个复制。

该指令模式必须同时显示出操作数0和1被修改。

该指令模式必须产生内存栅栏指令，使得在原子操作之前的所有内存操作，都在原子操作之前发生，所有在原子操作之后的内存操作，都在原子操作之后发生。

`sync_compare_and_swap_ccmode'

该指令模式与sync_compare_and_swapmode类似，除了其比较和交换的比较部分就好像是通过cmpm来发出的。该比较只与EQ和NE分支跳转，以及setcc操作一起使用。

一些目标机确实是通过状态标记来暴露比较并交换操作的成功或失败。理想的，我们不需要一个单独的命名指令模式来利用该特性，但是合并过程无法处理具有多个set的指令模式，而这正是定义sync_compare_and_swapmode所需要的。

`sync_addmode', `sync_submode'
`sync_iormode', `sync_andmode'
`sync_xormode', `sync_nandmode'

该指令模式产生一个在内存上进行原子操作的代码。操作数0为进行原子操作的内存。操作数1为二元操作符的第二个操作数。

``nand''运算为~op0 & op1。

该指令模式必须产生内存栅栏指令，使得在原子操作之前的所有内存操作，都在原子操作之前发生，所有在原子操作之后的内存操作，都在原子操作之后发生。

如果这些指令模式没有被定义，则操作将通过一个比较并交换操作，如果定义，来构建。

`sync_old_addmode', `sync_old_submode'
`sync_old_iormode', `sync_old_andmode'
`sync_old_xormode', `sync_old_nandmode'

这些指令模式产生在内存上的原子操作代码，并且返回操作之前内存中的值。操作数0为结果值，操作数1为执行原子操作的内存，操作数2为二元操作的第二个操作数。

该指令模式必须产生内存栅栏指令，使得在原子操作之前的所有内存操作，都在原子操作之前发生，所有在原子操作之后的内存操作，都在原子操作之后发生。

如果这些指令模式没有被定义，则操作将通过一个比较并交换操作，如果定义，来构建。

`sync_new_addmode', `sync_new_submode'
`sync_new_iormode', `sync_new_andmode'
`sync_new_xormode', `sync_new_nandmode'

这些指令模式类似于sync_old_op所对应的指令模式，除了它们返回操作之后内存位置中存在的值，而不是操作之前。

`sync_lock_test_and_setmode'

该指令模式根据目标机的能力，可以接受两种形式。两种情况下，操作数0为结果操作数，操作数1为执行原子操作的内存，操作数2为在锁中设置的值。

理想的情况下，该操作为一个原子交换操作，内存操作数中之前的值被复制到结果操作数中，值操作数被保存在内存操作数中。

对于能力差些的目标机，任何不为常量1的值操作数，将使用FAIL进行拒绝。这种情况下，目标机可以使用一个原子的测试并置位操作。结果操作数应该包含1，如果该位在之前被设置，或者为0，如果该位在之前被清空。内存操作数的真实内容由实现来定义。

该指令模式必须产生内存栅栏指令，使得在原子操作之前的所有内存操作，都在原子操作之前发生，所有在原子操作之后的内存操作，都在原子操作之后发生。

如果这些指令模式没有被定义，则操作将通过一个比较并交换操作，如果定义，来构建。

`sync_lock_releasemode'

该指令模式，如果被定义，释放由sync_lock_test_and_setmode设置的锁。操作数0为包含锁的内存；操作数1为存放在锁中的值。

如果目标机没有实现sync_lock_test_and_setmode的完整语义，则任何不是常量0的值操作数将使用FAIL来拒绝，内存操作数的真实内容由实现来定义。

该指令模式必须产生内存栅栏指令，使得在原子操作之前的所有内存操作，都在原子操作之前发生，所有在原子操作之后的内存操作，都在原子操作之后发生。

如果这些指令模式没有被定义，则会产生一个memory_barrier指令模式，紧跟一个将值存储到内存操作数的操作。

`stack_protect_set'

该指令模式，如果被定义，将内存操作数1中的Pmode值移送到内存操作数0中，并在之后不将该值留在寄存器中。这避免在某处泄露该值，从而使得攻击者用来重写栈保护槽。

如果该指令模式没有被定义，则生成一个普通的move指令模式。

`stack_protect_test'

该指令模式，如果被定义，比较内存操作数1和内存操作数0中的Pmode值，在之后不将该值留在寄存器中，并且如果值不等，则分支跳转到操作数2。

如果该指令模式没有被定义，则使用一个普通的比较和条件分支指令模式。

`clear_cache'

该指令模式，如果被定义，刷新一个内存区域的缓存。该区域的界限由操作数0（包含）和操作数1（不包含）中的Pmode指针界定。

如果该指令模式没有被定义，则使用对库函数__clear_cache的一个调用。

16.10 指令模式的顺序问题

有时一条insn可以匹配不止一个指令模式。 则在机器描述中首先出现的指令模式将被使用。因此，更加详细的指令模式（匹配更少事物的）和更快的指令 （在匹配时将会产生更好的代码）将通常放在描述中的前面。

有些情况下，指令模式的顺序效果可以用于隐藏无效的指令模式。 例如，68000有一条将全字转换为浮点的指令，和一条将字节转换为浮点的指令。 则将整数转换为浮点的指令将两者都可以匹配。 我们将转换全字的指令模式放在前面，从而确保使用前一种方式。 （否则将可能产生一个大的整数，以作为单个字节的立即数，这样可能无法工作）。 除了使用该指令模式顺序，还可能将转换字节的指令模式作的更加巧妙些， 以能够合适的处理任何常数值。

16.11 指令模式的相互依赖性

每个机器描述必须针对每个名为`bcond'的条件分支， 具有一个命名的指令模式。识别模板必须总是具有如下形式

```
(set (pc)
  (if_then_else (cond [(cc0) (const_int 0)])
        (label_ref (match_operand 0 "" ""))
        (pc)))
```

另外，每个机器描述必须针对每个可能的逆转条件分支具有一个匿名的指令模式。 它们的模板的形式为

```
(set (pc)
  (if_then_else (cond [(cc0) (const_int 0)])
        (pc)
        (label_ref (match_operand 0 "" ""))))
```

之所以需要这些，是因为jump优化会将直接条件（direct-conditional）分支， 转换为逆转条件（reverse-conditional）分支。

使用match_operator结构来减少必须为分支指定的指令模式的编号 通常是很方便的。例如，

```
(define_insn ""
 [(set (pc)
    (if_then_else (match_operator 0 "comparison_operator"
                    [(cc0) (const_int 0)])
          (pc)
          (label_ref (match_operand 1 "" ""))))]
 "condition"
 "...")
```

有些情况，机器支持除了一个或多个操作数的机器模式不同的相同指令。 例如，可以有指令“sign-extend halfword” 和“sign-extend byte” ，其指令模式为

```
(set (match_operand:SI 0 ...)
  (extend:SI (match_operand:HI 1 ...)))

(set (match_operand:SI 0 ...)
  (extend:SI (match_operand:QI 1 ...)))
```

常整数不指定机器模式，所以扩展常量值的指令可以两个指令模式都匹配。 实际匹配的指令模式将为文件中首先出现的指令模式。为了获得正确的结果， 其必须为尽可能宽的模式（这里为HImode）。 如果指令模式匹配QImode，则当常量值实际不适合该模式时，结果将不正确。

像这样扩展常数的指令很少被生成，因为它们会被优化掉， 不过在不优化的编译中确实偶尔会出现。

如果在指令模式中constraint允许为一个常量，reload过程可以用constraint允许的常量来替换寄存器。 类似的对于内存引用。由于存在这种替换， 所以不要为递增和递减指令提供单独的指令模式。 替换的，它们应该由同一个指令模式生成。

16.12 定义跳转指令模式

对于大多数机器，GCC假设该机器具有一个条件码。 比较insn根据给定操作数的有符号和无符号比较的结果来设定条件码。 单独的分支insn测试条件代码，并根据它的值进行分支跳转。 分支insn分为不同的有符号和无符号的。 许多通用机器，像VAX，68000和32000都按这种方式工作。

一些机器具有截然不同的有符号和无符号比较指令，并且只有一套条件分支指令。 处理这些机器最容易的方法为保持它们不变，直到最后写汇编代码的阶段。 这时，当输出比较指令的代码时，查看一下使用next_cc0_user (insn)的分支。 （ ）如果RTL得出其为一个无符号分支，则输出一个无符号比较； 否则输出一个有符号比较。当分支本身被输出时，你可以将有符号和无符号分支视为等同的。

之所以可以这样做，是因为GCC总是生成一对连续的RTL insn， 可能由note insn分隔，一个用于设置条件代码，一个用于测试， 并保持这对insn不被改变，直到最后。

要使用该技术，你必须定义机器描述宏NOTICE_UPDATE_CC， 来做CC_STATUS_INIT；换句话说，没有多余的比较指令。

一些机器具有比较分支指令，并且没有条件码。对它们可以使用类似的技术。 当该要“输出”一个比较指令时，将它的操作数记录在两个静态变量中。 当输出随后的条件码分支指令时，实际上输出了一个使用已记录的操作数的比较分支指令。

它还用于定义比较分支指令的指令模式。在优化编译中， 比较和分支指令对将根据这些指令模式被组合。但是如果优化不要求的时候， 这是不会发生的。所以你必须针对你定义的任何特定指令模式额外使用上面的一种解决方式。

在许多RISC机器上，大多数指令不影响条件码，并且甚至会没有一个单独的条件码寄存器。 在这些机器上，限制条件码的定义和使用为邻近的insn是不必要的， 并且还会阻止一些重要的优化。例如，在IBM RS/6000上，对于分支将会有一个延迟， 除非条件码寄存器在条件分支的三条指令前被设置。 如果不允许将条件码寄存器的定义和使用分开，则指令调度器将无法执行该优化。

在这些机器上，不要使用(cc0)，而是使用寄存器来表示条件代码。 如果该机器有一个特定的条件代码寄存器，则使用硬件寄存器。 如果条件代码或者比较结果可以被放在任意的通用寄存器中， 或者有多个条件寄存器，则使用伪寄存器。

在一些机器上，所生成的分支指令类型可以依赖于条件代码所产生的方式； 例如，在68k和SPARC上，直接从加法或减法指令来设置条件代码， 这并不像测试指令那样，不会清除溢出位， 所以不同的分支指令必须用于某些条件分支。对于使用(cc0)的机器， 对条件代码的设置和使用必须是邻近的（只有note insn分隔）， 以允许在cc_status中的标记被使用。（参见条件码）并且， 比较分支insn可以互相定位，通过使用函数prev_cc0_setter和next_cc0_user。

但是，这在不使用(cc0)的机器上是不一样的。 在这些机器上，并不会假设比较分支指令是邻近的，上面的方法不可用。 替换的，我们使用条件码寄存器的机器模式来记录条件码寄存器的不同格式。

用于存储条件码值的寄存器应该具有MODE_CC类别的一个机器模式。 通常，其为CCmode。如果需要额外的机器模式（正如上面提到的SPARC中加法例子）， 则在`machine-modes.def'中定义它们（参见条件码）。 还要定义SELECT_CC_MODE来选择给定比较操作数的机器模式。

如果知道在RTL生成过程中，将需要不同的机器模式（例如，如果机器具有单独的比较指令，针对有符号和无符号）， 则它们可以在那个时候被指定。

如果是在指令合成时需要不同的机器模式， 则宏SELECT_CC_MODE用来确定哪个机器模式作为比较结果。 指令模式应该使用该模式来书写。要支持上面讨论的SPARC的加法， 我们具有指令模式

```
(define_insn ""
 [(set (reg:CC_NOOV 0)
       (compare:CC_NOOV
        (plus:SI (match_operand:SI 0 "register_operand" "%r")
                 (match_operand:SI 1 "arith_operand" "rI"))
        (const_int 0)))]
 ""
 "...")
```

SPARC的宏SELECT_CC_MODE为参数是plus的比较运算返回一个CC_NOOVmode。

16.13 定义循环指令模式

一些机器具有特定的跳转指令，可以使循环更为有效。 常见的例子为68000的`dbra'指令，其执行一个寄存器的递减， 并且如果结果大于0，则进行一个分支。其它机器，特别是数字信号处理器（DSP），具有特定的块重复指令，以提供低成本的循环支持。 例如TI TMS320C3x/C4x DSP具有一个块

重复指令，其加载特定的寄存器来标记一个循环的顶部和底部，并计算循环迭代的次数。这就避免了对`dbra'这样的指令的取指和执行的需要，并避免了和跳转相关的流水线阻塞。

GCC具有三个特定的命名指令模式，来支持低开销循环。它们为`decrement_and_branch_until_zero',`doloop_begin'和`doloop_end'。第一个指令模式`decrement_and_branch_until_zero'，在RTL生成过程中没有被产生，但可以在指令合并阶段被产生。这需要循环优化器的帮助，使用在strength reduction过程中所搜集的信息，来将一个循环转换为向下计数到0。一些target还要求循环优化器增加一个REG_NONNEG注解，来指示迭代计数总是为正的。这在target执行一个有符号循环终止测试的时候被需要。例如，68000为它的dbra指令使用了下面类似的指令模式：

```
(define_insn "decrement_and_branch_until_zero"
 [(set (pc)
    (if_then_else
     (ge (plus:SI (match_operand:SI 0 "general_operand" "+d*am")
           (const_int -1))
       (const_int 0))
     (label_ref (match_operand 1 "" ""))
     (pc)))
  (set (match_dup 0)
    (plus:SI (match_dup 0)
        (const_int -1)))]
 "find_reg_note (insn, REG_NONNEG, 0)"
 "...")
```

注意，由于该insn为jump insn并且具有一个输出，所以其必须处理自己的重载，因此要使用`m' constraint。还要注意，由于该insn是在指令组合过程中，通过将两个连序insn组合成一个隐式并行的insn而生成的，所以迭代计数器需要。注意下面类似的指令模式将不会被编译器匹配。

```
(define_insn "decrement_and_branch_until_zero"
 [(set (pc)
    (if_then_else
     (ge (match_operand:SI 0 "general_operand" "+d*am")
       (const_int 1))
     (label_ref (match_operand 1 "" ""))
     (pc)))
  (set (match_dup 0)
    (plus:SI (match_dup 0)
        (const_int -1)))]
 "find_reg_note (insn, REG_NONNEG, 0)"
 "...")
```

另外两个特定的循环指令模式`doloop_begin'和`doloop_end'，由循环优化器为一些具有有限循环迭代，使用强度消减所搜集的信息的良好行为的循环所生成的。

`doloop_end'指令模式描述了实际的循环指令（或者隐式的循环操作），`doloop_begin'指令模式为一个可选的配套指令模式，可以用于一些低开销循环指令的初始化需要。

注意有些机器需要在循环顶部生成实际的循环指令（例如TMS320C3x/C4x DSP）。在循环顶部生成真正的RTL循环指令，会对流分析造成问题。所以，替换的，一个假`doloop' insn在循环的结尾处被生成。机器相关的reorg过程会检查该`doloop' insn的存在，然后向后搜寻插入了真正循环insn（假设在循环中没有会造成问题的指令）的循环顶部。任何额外的标号都可以在该点被生成。另外，如果所需要的特定迭代技术器寄存器没有被分配，则该机器相关的reorg过程能够生成一个传统的比较跳转指令对。

指令模式`decrement_and_branch_until_zero'和`doloop_end'的本质区别为，循环优化器会为后者分配一个额外的伪寄存器作为迭代计数器。该伪寄存器不能用在循环中（即，通用的规约变量不能由此生成），但是，许多情况下循环规约变量可能变成冗余的并且被后续的过程移除掉。

16.14 指令规范化

经常会有多个RTL表达式可以表示由单个机器指令所执行的运算。该情况对于逻辑，分支和乘累加指令最常见。对于这样的情况，编译器尝试将这些多个RTL表达式转换为一个规范的形式，以减少对insn指令模式的需求数。

除了进行代数简化以外，还执行了下面的规范化：

- 对于可交换指令和比较指令，总是将常量作为第二个操作数。如果机器只支持常量作为第二个操作数，则只需要提供匹配将常量作为第二个操作数的指令模式。
- 对于结合性操作符，操作符序列总是向左方向链接；例如，一个整数plus，只有它的左操作数本身可以为一个plus。当应用到整数时，and, ior, xor, plus, mult, smin, smax, umin和umax为可结合的，对于浮点，这些有时为可结合的。
- 对于这些操作符，如果只有一个为neg, not,mult, plus 或 minus表达式的操作数，则其将为第一个操作数。
- 对于neg, mult, plus和minus的组合中，neg操作（如果存在）将被尽可能的移到内部。例如(neg (mult A B))将被规范为(mult (neg A) B)，但是(plus (mult (neg A) B) C)将被规范为(minus A (mult B C))。
- 对于compare运算符，在使用cc0（参见Section 16.12 [跳转指令模式], page 254）的机器上，常量总是为第二个操作数。在其它机器上，极少的情况下，编译器可能想使用常量作为第一个操作数来构建compare。但是，这些情况并不常见，所以不值得来提供匹配常量作为第一个操作数的指令模式，除非机器确实具有这样的指令。

  在与上面条件相同的情况下，neg, not, mult, plus 或 minus的操作数被作为第一个操作数。
- (ltu (plus a b) b)被转换为(ltu (plus a b) a)。同样，使用geu来替换ltu。
- (minus x (const_int n))被转换为(plus x (const_int -n))。
- 在地址计算中（即，在mem中），左移操作被转换为与合适的2的幂相乘。
- De Morgan法则被用于在按位‘逻辑与’或着‘逻辑或’运算中，将位置反。如果该结果为not表达式的唯一的操作数，则其为第一个。

  具有执行按位‘逻辑与’，且其中一个操作数为一个按位求反的机器，应该为该指令指定如下的指令模式

```
(define_insn ""
 [(set (match_operand:m 0 ...)
    (and:m (not:m (match_operand:m 1 ...))
           (match_operand:m 2 ...)))]
 "..."
 "...")
```

  类似的，“NAND”指令的指令模式应给被写为

```
(define_insn ""
 [(set (match_operand:m 0 ...)
    (ior:m (not:m (match_operand:m 1 ...))
           (not:m (match_operand:m 2 ...))))]
 "..."
 "...")
```

  对于这两种情况，都没必要包含许多逻辑上相同的RTL表达式。
- 涉及按位‘异或’和求反的唯一可能的RTL表达式为(xor:m x y) 和(not:m (xor:m x y))。
- 对于三项的和，其中一个为常量的，将使用如下形式

```
(plus:m (plus:m x y) constant)
```

- 在不使用cc0的机器上，(compare x (const_int 0))将被转换为x。

- 位组（通常是单个位）和0的相等比较，将使用zero_extract， 而不是等价的and或者sign_extract运算。

更多的规范化规则都定义在`gcc/rtlanal.c'里的函数commutative_operand_precedence中。

16.15 为代码生成定义RTL序列

在一些目标机上，一些用于RTL生成的标准指令模式名无法通过单个insn来处理， 但是可以用一个RTL insn序列来表示它们。对于这些目标机， 你可以写一个define_expand来指定如何生成RTL序列。

define_expand为一个RTL表达式，看起来非常像define_insn； 但是不同之处为，define_expand只用于RTL生成，并且可以产生多个RTL insn。

define_expand RTX具有4个操作数:

- 名字。每个define_expand必须具有一个名字， 因为必须通过对名字的引用才能使用它。
- RTL模板。这是一个RTL表达式向量，表示一个指令序列。
- 条件，一个字符串包含了一个C表达式。 该表达式用于表示该指令模式对于GCC运行时命令行选项所选择的什么样的target机器 子类别有效。这跟具有标准名字的define_insn的条件类似。因此， 条件（如果存在）可以不依赖于所匹配的insn的数据，而只是依赖于target机器类型标号。 编译器需要在初始化时测试这些条件，以便确切的知道在一次特定的运行时， 哪些命名指令有效。
- 准备语句，一个字符串，包含了0个或多个C语句，其将在RTL模板生成RTL代码前被执行。

每个由define_expand生成的RTL insn必须匹配机器描述中的某个 define_insn。否则，编译器在尝试为insn生成代码或者试图对其进行优化的时候， 将会崩溃。

RTL模板，除了控制RTL insn的生成，还描述了当使用该指令模式时， 所需要指定的操作数。特别是，它为每个操作数给出了断言。

需要被指定的由指令模式生成RTL的，真正的操作数， 在RTL模板中它第一次出现的位置使用match_operand来描述。 这将把对于操作数的断言信息放入记录该事情的表中。 如果操作数被引用多次，则后续的引用应该使用match_dup。

RTL模板还可以引用内部操作数， 其为只在由define_expand生成的序列中使用的临时寄存器或者标号。 内部操作数使用match_dup来替换到RTL模板中，而不是match_operand。 内部操作数的值在编译器需要使用该指令模式时，不作为参数传入。替代的， 它们在指令模式中计算，在准备语句中。 这些语句计算值并将它们存入到合适的operands元素中， 以便match_dup可以找到它们。

有两个特定的宏，用于准备语句中：DONE和FAIL。 在其后面跟随一个分号，以作为一条语句来使用。

DONE
: 使用DONE宏来结束该指令模式的RTL生成。这种情况下， 由该指令模式生成的唯一的RTL insn将为在准备语句中显示调用emit_insn生成 的insn；RTL模板将不被生成。

FAIL
: 使指令模式对于这种情况失败。当指令模式失败时，这意味着指令模式实际上无效。编译器中的调用程序将会尝试其它策略，使用其它指令模式来进行代码生成。

  目前，FAIL操作只支持二元（加法，乘法，移位等）和 位域（extv, extzv和insv）操作。

如果准备语句即没有调用DONE，也没有调用FAIL， 则define_expand的行为便跟define_insn一样，即RTL模板用于生成insn

RTL模板不用于匹配，只是用于生成最初的insn列表。 如果准备语句总是调用DONE或者FAIL， 则RTL模板可以简化为一个简单的操作数列表，例如：

```
(define_expand "addsi3"
 [(match_operand:SI 0 "register_operand" "")
  (match_operand:SI 1 "register_operand" "")
  (match_operand:SI 2 "register_operand" "")]
 ""
 "
{
 handle_add (operands[0], operands[1], operands[2]);
 DONE;
}")
```

这里有一个例子，是为SPUR芯片定义的左移位：

```
(define_expand "ashlsi3"
 [(set (match_operand:SI 0 "register_operand" "")
    (ashift:SI
     (match_operand:SI 1 "register_operand" "")
     (match_operand:SI 2 "nonmemory_operand" "")))]
 ""
 "
{
 if (GET_CODE (operands[2]) != CONST_INT
   || (unsigned) INTVAL (operands[2]) > 3)
  FAIL;
}")
```

这个例子使用了define_expand，使得当移位数在支持的0到3的范围内时， 便会生成移位RTL insn，而对于其它情况，则会失败。当其失败时， 编译器便会使用不同的指令模式（比如一个库调用）来尝试其它策略。

如果编译器能够处理具有名字的指令模式中的非平凡的条件字符串， 则对于这样情况也可以使用define_insn。这里有另一种情况（68000上的零扩展）， 其使用了define_expand的更强大的功能：

```
(define_expand "zero_extendhisi2"
 [(set (match_operand:SI 0 "general_operand" "")
    (const_int 0))
  (set (strict_low_part
     (subreg:HI
      (match_dup 0)
      0))
    (match_operand:HI 1 "general_operand" ""))]
 ""
 "operands[1] = make_safe_from (operands[1], operands[0]);")
```

这里将会生成两个RTL insn，一个用于清除整个输出操作数， 另一个用于将输入操作数复制到其低半部份。 该指令序列在输入操作数指向输出操作数（的旧值）时，是不正确的。 所以，准备语句用来确保不是这样。当其指向operands[0]时， 函数make_safe_from用来将operands[1]复制到临时寄存器中。 其通过生成另一个RTL insn来完成这件事。

最后，第三个例子显示了内部操作数的使用。 在SPUR芯片上的零扩展是通过将结果与上半字mask来完成的。 但是该mask不能通过一个const_int来表示，因为常量值太大， 无法在该机器上被合法化。所以其必须使用force_reg复制到寄存器中， 然后在and中使用该寄存器。

```
(define_expand "zero_extendhisi2"
 [(set (match_operand:SI 0 "register_operand" "")
    (and:SI (subreg:SI
         (match_operand:HI 1 "register_operand" "")
         0)
        (match_dup 2)))]
 ""
 "operands[2]
```

```
= force_reg (SImode, GEN_INT (65535)); ")
```

注意：如果define_expand被用于一个标准的二元或者一元算数运算， 或者一个位域运算，则其生成的最后的insn一定不能为一个code_label, barrier或note。其必须为一个insn, jump_insn或call_insn。如果你在结尾处不需要实际的insn， 则可以生成一条将操作数的结果复制到其本身的insn。这样的insn将不会生成代码， 但可以避免编译器中的问题。

16.16 定义如何拆分指令

有两种情况，你应该指定如何将一个指令模式拆分为多个insn。在一些机器上， 指令需要延迟槽（参见Section 16.19.7 [延迟槽], page 272）或者指令的输出对于多周期（参见Section 16.19.8 [处理器流水线描述], page 273）不可用， 则优化这些情况的编译器过程需要能够将insn移入延迟槽中。但是， 一些insn可能会生成不止一条机器指令。这些insn则不能被放入延迟槽。

通常你可以重写单个insn为单独的insn列表，每个对应于一条机器指令。 这样做的缺点是它将造成编译变慢并且需要更多的空间。如果结果insn太复杂， 则还会抑制一些优化。当编译器有理由相信可以改进指令或者延迟槽调度的时候， 则会拆分insn。

insn组合器阶段还拆分putative insns。 如果三个insn被合并到一个使用复杂表达式的insn， 其不能被某个define_insn模式匹配， 则组合器阶段尝试将复杂指令模式拆分为两个被识别的insn。通常， 它能够将复杂指令模式通过拆分某个子表达式来断开。但是，有些情况下， 像在一个RISC机器上执行一个大常量的加法，则拆分加法为两个insn的方式是机器相关的。

define_split定义告诉了编译器如何将一个复杂的insn拆分为多个简单的insn。 它的形式为：

```
(define_split
 [insn-pattern]
 "condition"
 [new-insn-pattern-1
  new-insn-pattern-2
  ...]
 "preparation-statements")
```

insn-pattern为需要被拆分的指令模式，condition为要被测试的最终条件， 跟define_insn中的一样。 当一个insn匹配insn-pattern， 并且满足条件condition， 则它由insn列表new-insn-pattern-1, new-insn-pattern-2等来替换。

preparation-statements与那些为define_expand（参见Section 16.15 [定义扩展], page 258）指定的语句类似，并且在生成新RTL之前被执行。 与define_expand中的不同之处为，这些语句不能生成任何新的伪寄存器。 一旦完成重载，它们则不能在栈帧中分配任何空间。

指令模式根据两种不同的环境来匹配insn-pattern。 如果需要为延迟槽调度或者insn调度来拆分insn， 则insn已经是有效的， 这意味着它已经被一些define_insn匹配过， 并且如果reload_completed为非0，则已经满足那个define_insn的约束。 在那种情况下，新的insn模式必须也是匹配某个define_insn的insn， 并且如果reload_completed为非0，则也必须满足那些定义的约束。

对于这种define_split用法的例子，考虑下面来自`a29k.md'的例子， 其将从HImode到SImode的sign_extend拆分为一对shift insn：

```
(define_split
 [(set (match_operand:SI 0 "gen_reg_operand" "")
    (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
 ""
 [(set (match_dup 0)
    (ashift:SI (match_dup 1)
          (const_int 16)))
  (set (match_dup 0)
```

```
    (ashiftrt:SI (match_dup 0)
          (const_int 16)))]
 "
{ operands[1] = gen_lowpart (SImode, operands[1]); }")
```

当组合器阶段尝试拆分一个insn模式时，则情况总是为， 指令模式没有被任何define_insn匹配。组合器过程首先尝试将单个set表达式拆分， 然后是在parallel中的相同的set表达式， 不过跟随一个伪寄存器的clobber，以作为scratch寄存器来使用。 这这些情况下，组合器期望能够生成两个新的insn。 它将验证这些指令模式匹配某个define_insn定义， 所以你不需要在define_split中做这些测试（ 当然，there is no point in writing a define_split that will never produce insns that match).

```
(define_split
 [(set (match_operand:SI 0 "gen_reg_operand" "")
    (plus:SI (match_operand:SI 1 "gen_reg_operand" "")
         (match_operand:SI 2 "non_add_cint_operand" "")))]
 ""
 [(set (match_dup 0) (plus:SI (match_dup 1) (match_dup 3)))
  (set (match_dup 0) (plus:SI (match_dup 0) (match_dup 4)))]
"
{
 int low = INTVAL (operands[2]) & 0xffff;
 int high = (unsigned) INTVAL (operands[2]) >> 16;

 if (low & 0x8000)
  high++, low |= 0xffff0000;

 operands[3] = GEN_INT (high << 16);
 operands[4] = GEN_INT (low);
}")
```

这里断言non_add_cint_operand匹配任何不是单个add insn的有效操作数的 const_int。

使用scratch寄存器的例子，来自同一个文件， 用来生成等价的寄存器和大常量的比较运算：

```
(define_split
 [(set (match_operand:CC 0 "cc_reg_operand" "")
    (compare:CC (match_operand:SI 1 "gen_reg_operand" "")
          (match_operand:SI 2 "non_short_cint_operand" "")))
  (clobber (match_operand:SI 3 "gen_reg_operand" ""))]
 "find_single_use (operands[0], insn, 0)
 && (GET_CODE (*find_single_use (operands[0], insn, 0)) == EQ
   || GET_CODE (*find_single_use (operands[0], insn, 0)) == NE)"
 [(set (match_dup 3) (xor:SI (match_dup 1) (match_dup 4)))
  (set (match_dup 0) (compare:CC (match_dup 3) (match_dup 5)))]
 "
{
 /* Get the constant we are comparing against, C, and see what it
    looks like sign-extended to 16 bits.  Then see what constant
    could be XOR'ed with C to get the sign-extended value.  */

 int c = INTVAL (operands[2]);
 int sextc = (c << 16) >> 16;
 int xorv = c ^ sextc;

 operands[4] = GEN_INT (xorv);
 operands[5] = GEN_INT (sextc);
}")
```

为了避免混淆，不要写这样的define_split， 其接受匹配某个define_insn的一些insn，同时也接受不匹配的insn。替代的， 可以写两个分别的define_split定义，一个针对有效的insn， 一个针对无效的insn。

允许将跳转指令拆分为一个跳转序列或者在拆分非跳转指令时创建新的跳转。 由于控制流图和分支预测信息需要更新，所以会有一些限制。

将跳转指令拆分为由另一个跳转指令覆盖的指令序列，总是有效的， 因为编译器期望新的跳转具有相同的行为。当新的序列包含多个跳转指令或新的标号时， 则需要更多的辅助。只允许创建无条件跳转，或者简单的条件跳转指令。另外， 其必须为每个条件跳转附加一个REG_BR_PROB注解。 全局变量split_branch_probability保存了原始分支的可能性。 为了简化边频率的重新计算，新的序列要求只具有向前跳转。

对于通常的情况， define_split的模式完全匹配define_insn的模式， 则可以使用define_insn_and_split。其形式为：

```
(define_insn_and_split
 [insn-pattern]
 "condition"
 "output-template"
 "split-condition"
 [new-insn-pattern-1
  new-insn-pattern-2
  ...]
 "preparation-statements"
 [insn-attributes])
```

insn-pattern, condition, output-template和 insn-attributes跟在define_insn中的用法一样. new-insn-pattern 向量和 preparation-statements 跟在 define_split中的用法一样。split-condition也跟在define_split中的用法一样, 不同之处是如果condition开始于`&&', 则用于拆分的条件将被构造为split condition和insn condition的逻辑“and”运算。 例如，在i386.md中:

```
(define_insn_and_split "zero_extendhisi2_and"
 [(set (match_operand:SI 0 "register_operand" "=r")
   (zero_extend:SI (match_operand:HI 1 "register_operand" "0")))
  (clobber (reg:CC 17))]
 "TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
 "#"
 "&& reload_completed"
 [(parallel [(set (match_dup 0)
          (and:SI (match_dup 0) (const_int 65535)))
      (clobber (reg:CC 17))])]
 ""
 [(set_attr "type" "alu1")])
```

在这种情况下，实际的split condition将为`TARGET_ZERO_EXTEND_WITH_AND && !optimize_size && reload_completed'。

define_insn_and_split结构提供了与两个单独的define_insn和define_split指令模式相同的功能. 其形式紧凑。

16.17 在机器描述中包含指令模式

模板include告诉编译器工具， 从哪里寻找在其它文件中而不是文件`.md'中的模板。 这只在构建时候使用，并且不允许进行预处理操作。

其形如：

```
(include
 pathname)
```

例如：

```
(include "filestuff")
```

其中pathname为一个指定文件位置的字符串，指定了包含文件为`gcc/config/target/filestuff'。目录`gcc/config/target'被当作缺省目录。

机器描述可以被分割成容易管理的小章节并放在子目录下。

通过指定：

```
(include "BOGUS/filestuff")
```

包含文件被指定为`gcc/config/target/BOGUS/filestuff'。

为包含文件指定一个绝对路径，像：

```
(include "/u2/BOGUS/filestuff")
```

是被允许的，但不鼓励这么做。

16.17.1 用于目录搜索的RTL生成工具选项

选项`-Idir'指定了搜索机器描述的目录。例如：

```
genrecog -I/p1/abc/proc1 -I/p2/abcd/pro2 target.md
```

目录dir将被增加到搜索头文件的目录列表的头部。这可以用来覆盖系统的机器定义文件，而替换为你自己的版本，因为这些目录将在缺省机器描述文件目录之前被搜索。如果你使用了多个`-I'选项，则目录按照从左到右的顺序被扫描；之后为标准的缺省目录。

16.18 机器特定的窥孔优化

除了指令模式，`md'文件还可以包含关于机器特定的窥孔优化的定义。

当程序中的数据流没有建议进行尝试的时候，组合器便不会注意某些可能存在的窥孔优化。例如，有时两个连续的insn是可以被组合的，虽然第二个没有显示出要使用在第一个中所计算的寄存器。机器特定的窥孔优化器可以检测出这样的机会。

有两种窥孔定义形式可以使用。最初的define_peephole运行于汇编输出时，用于匹配insn和替换汇编文本。不赞成使用define_peephole。

较新的define_peephole2用来匹配insn和替换新的insn。peephole2过程运行于寄存器分配之后，调度之前，其使进行调度的target获得更好的代码。

16.18.1 RTL到文本的窥孔优化器

定义的形式如下：

```
(define_peephole
 [insn-pattern-1
  insn-pattern-2
  ...]
```

```
  "condition"
  "template"
  "optional-insn-attributes")
```

如果没有在该机器描述中使用任何机器特定的信息， 则可以省略掉最后的字符串操作数。 如果有，则其必须遵守在define_insn中相同的规则。

该结构中，insn-pattern-1等为匹配连续insn的指令模式。 当insn-pattern-1匹配第一个insn，insn-pattern-2匹配下一个，等等依次类推的情况时， 则会将优化应用到该insn序列。

每个由窥孔匹配的insn也必须匹配一个define_insn。 窥孔只在代码生成前的最后阶段被检查，并且只是可选的。 因此，在一个未优化的编译中，或者不同的优化阶段中， 任何匹配窥孔但是不匹配define_insn的将会在代码生成时造成崩溃。

和通常一样，insn的操作数使用match_operands, match_operator和match_dup来匹配。 不同的是，操作数编号应用在定义的所有insn指令模式中。所以， 你可以通过一个insn中的match_operand和另一个insn中的match_dup， 来在两个insn中检查相同的操作数。

用于match_operand指令模式的操作数constraint对窥孔的适用性没有任何直接的影响， 不过它们将会在后面被验证，所以要确信当窥孔匹配时，你的constraint要足够通用。 如果窥孔匹配，但constraint却不满足，则编译器将崩溃。

将窥孔中的所有操作数的constraint省略掉是安全的； 或者你可以编写constraint作为之前测试过的标准的二次检查。

一旦insn序列匹配指令模式，则condition将被检查。 这是一个C表达式，用于对是否执行优化来做最后的决定（如果表达式非0时，我们这样做）。 如果condition被省略掉（换句话说，字符串为空）则优化会被应用到每个匹配指令模式的insn序列。

定义的窥孔优化在寄存器分配完成之后应用。 因此，窥孔定义可以只是查看操作数，便能检查哪些操作数结束于哪种寄存器。

在条件中引用操作数的方式为对操作数编号i编写operands[i]（匹配于(match_operand i...)）。使用变量insn来引用正在被匹配的insns的最后一个insn； 使用prev_active_insn来找到先前的insns。

当正在优化中间结果计算时， 你可以使用条件来匹配只有当中间结果不在其它地方被使用的情况。 使用C表达式dead_or_set_p (insn, op)， 其中insn为你所期望其值为最后一次被使用的insn，以及op为中间值（来自operands[i]）。

应用优化，意味着将insn序列替换为新的insn。 template控制了针对该组合insn的最终汇编代码输出。 就像define_insn模板所做的一样。 该模板中的操作数编号与用于要匹配的原始insn序列中的相同。

被定义的窥孔优化器的结果不需要匹配机器描述中的任何insn模式； 它甚至没有机会来匹配它们。窥孔优化器定义本身是作为insn模式， 用来控制insn如何输出。

被定义的窥孔优化器被作为汇编代码运行输出， 所以它们产生的insns不再被组合或重排。

这里有一个例子，来自68000机器描述：

```
(define_peephole
 [(set (reg:SI 15) (plus:SI (reg:SI 15) (const_int 4)))
  (set (match_operand:DF 0 "register_operand" "=f")
     (match_operand:DF 1 "register_operand" "ad"))]
 "FP_REG_P (operands[0]) && ! FP_REG_P (operands[1])"
{
 rtx xoperands[2];
 xoperands[1] = gen_rtx_REG (SImode, REGNO (operands[1]) + 1);
#ifdef MOTOROLA
 output_asm_insn ("move.l %1,(sp)", xoperands);
```

```
 output_asm_insn ("move.l %1,-(sp)", operands);
 return "fmove.d (sp)+,%0";
#else
 output_asm_insn ("movel %1,sp@", xoperands);
 output_asm_insn ("movel %1,sp@-", operands);
 return "fmoved sp@+,%0";
#endif
})
```

该优化的效果是将

```
jbsr _foobar
addql #4,sp
movel d1,sp@-
movel d0,sp@-
fmoved sp@+,fp0
```

转换为

```
jbsr _foobar
movel d1,sp@
movel d0,sp@-
fmoved sp@+,fp0
```

insn-pattern-1等看起来与define_insn的第二个操作数非常相似。 不过有一个重要的不同：define_insn的第二个操作数包含了一个或多个RTX， 使用方括号包裹。通常，只有一个： 那么相同的动作则可以写成define_peephole的一个元素。 但是，当在define_insn中有多个动作时，它们被隐式的由parallel包裹。 则你必须在define_peephole中，显式的写出parallel，以及里面的方括号。 因此，如果一个insn的指令模式如下，

```
(define_insn "divmodsi4"
 [(set (match_operand:SI 0 "general_operand" "=d")
    (div:SI (match_operand:SI 1 "general_operand" "0")
        (match_operand:SI 2 "general_operand" "dmsK")))
  (set (match_operand:SI 3 "general_operand" "=d")
    (mod:SI (match_dup 1) (match_dup 2)))]
 "TARGET_68020"
 "divsl%.l %2,%3:%0")
```

则在窥孔中提及该insn的方法为：

```
(define_peephole
 [...
  (parallel
   [(set (match_operand:SI 0 "general_operand" "=d")
      (div:SI (match_operand:SI 1 "general_operand" "0")
          (match_operand:SI 2 "general_operand" "dmsK")))
    (set (match_operand:SI 3 "general_operand" "=d")
      (mod:SI (match_dup 1) (match_dup 2)))])
 ...]
 ...)
```

16.18.2 RTL到RTL的窥孔优化器

define_peephole2定义告诉了编译器如何将一个指令序列用来替换另一个序列， 可能需要那些额外的scratch寄存器，以及它们的生命期必须为什么。

```
(define_peephole2
 [insn-pattern-1
  insn-pattern-2
  ...]
 "condition"
 [new-insn-pattern-1
```

```
 new-insn-pattern-2
 ...]
"preparation-statements")
```

定义几乎与define_split（参见 Section 16.16 [Insn拆分], page 260）相同， 除了要匹配的指令模式不是一个单个指令，而是一个指令序列。

在输出模板中有可能需要用到额外的scratch寄存器。 如果没有合适的寄存器，则指令模式将简单的作为不匹配处理。

使用match_scratch描述所需要的scratch寄存器， 并放在输入指令模式的顶层。被分配的寄存器（最初的）将会在原始指令序列中需要使用的位置死掉。 如果scratch被用于多个位置， 则位于输入指令模式顶层的match_dup 指令模式用来标记在输入序列中寄存器必须为活跃的最后位置。

这里有一个来自IA-32机器描述的例子：

```
(define_peephole2
 [(match_scratch:SI 2 "r")
  (parallel [(set (match_operand:SI 0 "register_operand" "")
                  (match_operator:SI 3 "arith_or_logical_operator"
                   [(match_dup 0)
                    (match_operand:SI 1 "memory_operand" "")]))
             (clobber (reg:CC 17))])]
 "! optimize_size && ! TARGET_READ_MODIFY"
 [(set (match_dup 2) (match_dup 1))
  (parallel [(set (match_dup 0)
                  (match_op_dup 3 [(match_dup 0) (match_dup 2)]))
             (clobber (reg:CC 17))])]
 "")
```

该指令模式尝试拆分加载的使用，以希望我们能够调度内存加载延迟。 它分配了一个GENERAL_REGS("r")类别的单个的SImode寄存器， 其只需在算术运算之前的位置为活跃的。

很难找到需要延长scratch生命期的真实例子，所以这里只是一个制造的例子：

```
(define_peephole2
 [(match_scratch:SI 4 "r")
  (set (match_operand:SI 0 "" "") (match_operand:SI 1 "" ""))
  (set (match_operand:SI 2 "" "") (match_dup 1))
  (match_dup 4)
  (set (match_operand:SI 3 "" "") (match_dup 1))]
 "/* determine 1 does not overlap 0 and 2 */"
 [(set (match_dup 4) (match_dup 1))
  (set (match_dup 0) (match_dup 4))
  (set (match_dup 2) (match_dup 4))]
  (set (match_dup 3) (match_dup 4))]
 "")
```

如果我们没有在输入序列的中间增加(match_dup 4)， 则可能的情况是我们选择的寄存器会在序列的起始处被第一个或第二个set杀死。

16.19 指令属性

除了描述target机器所支持的指令以外，`md' 文件还定义了一组 attributes 以及每个属性的取值集合。每条生成的insn都为每个属性赋予一 个值。一种可能的属性是insn对于机器的条件代码所产生的影响。该属性然后可以 被 NOTICE_UPDATE_CC 使用，来跟踪条件代码。

16.19.1 定义属性以及它们的值

表达式 define_attr 用于定义目标机所需要的每个属性。其形式为：

```
(define_attr name list-of-values default)
```

name 为一个字符串，指定了被定义的属性名。

list-of-values 或者为一个字符串，指定了可以赋予属性的逗号分隔 的值的列表，或者为一个空字符串，表示属性接受一个数字值。

default 为一个属性表达式，给出了匹配指令模式，但指令模式定义中 没有显式包含该属性值的insns，所应具有的属性值。关于更多处理缺省值的 信息，see Section 16.19.4 [属性例子], page 270。关于不依赖于任何特定insn的属性， see Section 16.19.6 [常量属性], page 272。

对于每个定义的属性，都有许多定义被写入 `insn-attr.h' 文件。对于 显式指定了属性取值集合的情况，下列将被定义：

- 一个针对符号 `HAVE_ATTR_name' 的 `#define' 被写入。
- 一个枚举类别被定义，元素的形式为 `upper-name_upper-value'， 其中属性名和值首先被转换为大写的。
- 一个函数 `get_attr_name' 被定义，其传入一个insn并返回该insn 的属性值。

例如，如果在 `md' 文件中存在下列定义：

```
(define_attr "type" "branch,fp,load,store,arith"...)
```

则下面的行将被写入文件 `insn-attr.h' 中。

```
#define HAVE_ATTR_type
enum attr_type {TYPE_BRANCH, TYPE_FP, TYPE_LOAD,
                TYPE_STORE, TYPE_ARITH};
extern enum attr_type get_attr_type ();
```

如果属性接受数字值，则不会定义enum 类型，并且获得属性值的函数将 返回int。

有些属性被赋予特定的含义。这些属性不能随便用于其它目的：

length　length属性用于计算每输出的代码块的长度。这尤其在验证分支距离的时候特别重要。See Section 16.19.5 [Insn长度], page 271.

enabled　enabled属性可以被定义，用来在代码生成过程中，阻止insn定义中的特定的可选项。See Section 16.8.6 [禁止Insn可选项], page 232.

16.19.2 属性表达式

用于定义属性的RTL表达式，除了使用上面描述的代码外，还使用了下面要 讨论的一些细节。属性值表达式必须为下列形式之一：

(const_int i)

整数 i 指定了一个数字属性的值。 i 必须为非负的。

数字属性的值可以被指定为一个 const_int，或者一个在 const_string, eq_attr (参见下面), attr, symbol_ref, 简单算术表达式，和特定指令上的 set_attr (see Section 16.19.3 [给Insns打标签], page 269)中，作为字符串来表示的整数，

(const_string value)

字符串指定了一个常量属性值。如果 value 使用 `"*"' 指定， 则表示包含该表达式的insn将使用属性的缺省值。显然 `"*"' 不能用 于 define_attr 的 default 表达式。

如果属性值被指定为数值的，value 必须为一个字符串，包含了一个 非负整数（通常这种情况下会使用 const_int）。否则，其必须包含 一个有效的属性值。

(if_then_else test true-value false-value)

test 指定了一个属性测试，其格式在下面定义。如果 test 为 真时，该表达式的值为 true-value，否则为 false-value。

(cond [test1 value1 ...] default)
该表达式的第一个操作数为一个向量，包含了偶数个表达式并且由 test 和 value 表达式对组成。表达式 cond 的值为对应于第一个为 真的 test 表达式的 value。如果没有 test 表达式为真， 则 cond 表达式的值为 default 表达式。

test表达式可以具有下列形式：

(const_int i)
该测试为真，如果i非零，否则为假。

(not test)
(ior test1 test2)
(and test1 test2)
这些测试为真，如果所表示的逻辑函数为真。

(match_operand:m n pred constraints)
该测试为真，如果insn的操作数n，其属性值被确定具有机器模式m（如果m为VOIDmode，则这部分测试被忽略），并且被字符串pred指定的函数，当传递操作数n和机器模式m时，返回一个非零值（如果pred为空字符串，则这部分测试被忽略）。

constraints操作数被忽略并且应该为空字符串。

(le arith1 arith2)
(leu arith1 arith2)
(lt arith1 arith2)
(ltu arith1 arith2)
(gt arith1 arith2)
(gtu arith1 arith2)
(ge arith1 arith2)
(geu arith1 arith2)
(ne arith1 arith2)
(eq arith1 arith2)
这些测试为真，如果所表示的对两个算术表达式的比较为真。算术表达式的形式为 plus, minus, mult, div, mod, abs, neg, and, ior, xor, not, ashift, lshiftrt和ashiftrt表达式。

const_int和symbol_ref总为有效（其它的形式， see Section 16.19.5 [Insn长度], page 271）。symbol_ref为一个字符串，表示当使用`get_attr_...'程序求解时，可以生成一个int的C表达式。其通常应该为一个全局变量。

(eq_attr name value)
name为一个字符串，指定了属性的名字。

value为一个字符串，或者为一个该属性name的有效值，由逗号分隔的值列表，或者为`!'，后面跟随一个值或者列表。如果value不是起始于`!'，则该测试为真，如果当前insn的name属性的值在值列表中。如果value起始于`!'，则该测试为真，如果属性值不在指定的列表中。

例如，

```
(eq_attr "type" "load,store")
```

等价于

```
(ior (eq_attr "type" "load") (eq_attr "type" "store"))
```

如果name指定了属性`alternative'，则它是指的编译器变量which_alternative的值（see Section 16.6 [输出语句], page 207），并且值必须为小整数。例如，

```
(eq_attr "alternative" "2,3")
```

等价于

```
(ior (eq (symbol_ref "which_alternative") (const_int 2))
     (eq (symbol_ref "which_alternative") (const_int 3)))
```

注意，对于大多数属性，当被测试的属性的值已知为匹配特定模式的所有insn时，eq_attr测试将被简化。这是迄今为止最常见的情况。

(attr_flag name)

表达式attr_flag的值为真，如果由name指定的标记对于当前被调度的insn为真。

name为一个字符串，指定了要测试的标记集合。测试标记forward和backward可以确定条件分支的方向。测试标记very_likely, likely, very_unlikely和unlikely可以确定条件分支是否被接受。

如果very_likely为真，则likely标记也为真。同样对于very_unlikely和unlikely也是这样。

该例子描述了一个条件分支延迟槽，其对于被接受的（annul-true）forward分支或者没有被接受的（annul-false）的backward分支，可以置空（nullified），

```
(define_delay (eq_attr "type" "cbranch")
 [(eq_attr "in_branch_delay" "true")
  (and (eq_attr "in_branch_delay" "true")
       (attr_flag "forward"))
  (and (eq_attr "in_branch_delay" "true")
       (attr_flag "backward"))])
```

标记forward和backward为假，如果当前被调度的insn不是条件分支。

标记very_likely和likely为真，如果被调度的insn不是条件分支。标记very_unlikely和unlikely为假，如果被调度的attr_flag不是条件分支。

attr_flag只用于延迟槽调度阶段，并且跟编译器的其它过程没有关系。

(attr name)

返回另一个属性的值。这对于数值属性非常有用，因为eq_attr和attr_flag可以产生比非数值属性更加有效的代码。

16.19.3 给Insns赋予属性值

给insn的属性赋予的值，主要由该insn所匹配的模式决定（或者什么define_peephole生成的它）。每个define_insn和define_peephole可以具有可选的最后的参数，用来指定匹配insn的属性值。在特定insn中没有指定的任何属性的值，将被设为在define_attr中指定的缺省值。

可选的define_insn和define_peephole最后的参数，为一个表达式向量，每个元素定义了单个属性的值。赋属性值的最通用方式是使用set表达式，其第一个操作数为一个attr表达式，给出了要设置的属性名。第二个操作数为一个属性表达式（参见Section 16.19.2 [表达式], page 267），给出了属性值。

当属性值依赖于可选（`alternative'）属性，则可以使用set_attr_alternative表达式。其允许指定属性表达式向量，每个元素对应一个可选属性。

当不需要一般性的任意属性表达式，则可以使用简单的set_attr表达式，其允许指定一个字符串，给出单个属性值或者属性值列表，其中每个元素对应于一个可选属性。

以上的属性指定形式将在下面展示出。在每种情况中，name为一个字符串，指定了要被设置的属性。

(set_attr name value-string)

value-string或者为一个字符串，给出了期望的属性值， 或者为一个包含了逗号分隔的列表的字符串，给出了后续可选项的值。 元素的个数必须匹配在insn指令模式的约束中的可选项的个数。

注意可以为一些可选项指定`*'，这样属性将被假设它的缺省值匹配那个可选项。

(set_attr_alternative name [value1 value2 ...])

取决于insn的可选项，值将为被指定值的其中之一。 这是对`alternative'属性使用cond的简化形式。

(set (attr name) value)

第一个操作数必须为特定RTL表达式attr，其唯一的操作数是一个字符串， 给出了被设置的属性名。value为属性值。

下面展示了三种表示相同的属性指定的不同的方式：

```
(set_attr "type" "load,store,arith")

(set_attr_alternative "type"
          [(const_string "load") (const_string "store")
           (const_string "arith")])

(set (attr "type")
  (cond [(eq_attr "alternative" "1") (const_string "load")
      (eq_attr "alternative" "2") (const_string "store")]
     (const_string "arith")))
```

表达式define_asm_attributes提供了一种机制， 用来指定赋予insn的属性是由asm语句产生的。其形式为：

```
(define_asm_attributes [attr-sets])
```

其中attr-sets为与define_insn和define_peephole中的相同。

这些值通常为“最坏情况”属性值。例如，它们可能指示条件码将被破坏（clobbered）。

为length属性赋值，将被特殊处理。 计算asm insn长度的方式是将表达式define_asm_attributes中指定的长度 乘以在asm语句中指定的机器指令个数。 指令个数通过计算字符串中分号和换行符的个数来决定。因此， 在define_asm_attributes中指定的length属性值应该为单个机器指令的 最大可能长度。

16.19.4 关于属性说明的例子

要想有效的使用insn属性，巧妙的使用缺省值是很重要的。通常，insn被分为不同类别， 并使用称作type的属性来表示该值。该属性通常只用于定义其它属性的缺省值。 可以举一个例子来阐明它的用法。

假设我们有一个RISC机器，其具有一个条件码并且在寄存器中只进行全字操作。 让我们假设可以将所有的insn分为加载，存储，（整数）算术运算，浮点运算和分支。

在这里我们将关注条件码对于insn的影响，并局限在下列可能的影响： 条件码可以被不可预期的设置（clobbered），没有改变，被设为符合运算结果的值， 或者只在先前被设置的条件码已经被修改。

下面是该机器的一个样本`md'文件：

```
(define_attr "type" "load,store,arith,fp,branch" (const_string "arith"))

(define_attr "cc" "clobber,unchanged,set,change0"
       (cond [(eq_attr "type" "load")
```

```
              (const_string "change0")
            (eq_attr "type" "store,branch")
              (const_string "unchanged")
            (eq_attr "type" "arith")
              (if_then_else (match_operand:SI 0 "" "")
                    (const_string "set")
                    (const_string "clobber"))]
           (const_string "clobber")))

(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r,r,m")
    (match_operand:SI 1 "general_operand" "r,m,r"))]
 ""
 "@
  move %0,%1
  load %0,%1
  store %0,%1"
 [(set_attr "type" "arith,load,store")])
```

注意我们假设在上面的例子中，比机器字小的算术运算将会clobber条件码， 因为它们将会根据全字的结果来设置条件码。

16.19.5 计算一个Insn的长度

许多机器提供了多种类型的分支指令，针对于不同长度的分支位移。多数情况下， 汇编器会选择使用正确的指令。但是，当汇编器无法做到的时候，如果一个特殊的属性， length属性，被定义，则可以由GCC来完成。 该属性必须通过在它的define_attr中指定一个空字符串， 从而被定义成具有数字值。

对于length属性，在test表达式中允许两个额外形式的算术术语：

(match_dup n)
: 这是指当前insn的操作数n的地址，其必须为一个label_ref。

(pc)
: 这是指当前insn的地址。或许可以将其设为下一个insn的地址， 从而跟其它地方的用法一致，但是这样容易引起混淆，因为还要计算当前insn的长度。

对于通常的insn，长度将由length属性的值来确定。 对于addr_vec和addr_diff_vec的insn模式，长度通过向量数乘于每个向量的大小来计算获得。

长度按照可寻址的存储单元（字节）来度量。

下列宏可以用于改进长度计算：

ADJUST_INSN_LENGTH (insn, length)
: 如果定义，则在上下文中作为函数来使用，用于修改赋予指令insn的长度。 length为一个lvalue（左值）包含了最初计算的insn长度并将使用insn的正确长度来更新。

  该宏通常并不需要。一种使用它的情况为ROMP。在这个机器上， 一个addr_vec insn的大小必须被加2用于补偿可能需要的指令对齐。

返回get_attr_length（length属性的值）的程序， 可以被输出程序用来确定将要写入的分支指令的形式，正如下面的例子。

作为一个指定可变长度分支的例子，可以考虑一下IBM360。 如果我们采用寄存器将被设为函数起始地址这样的约定， 我们则可以使用一个4字节的指令来跳转到4K范围的标号。 否则，我们需要一个6字节的序列来从内存加载地址并然后分支到那里。

对于这样的机器，可以按照如下的方式来指定一个分支指令模式：

```
(define_insn "jump"
 [(set (pc)
    (label_ref (match_operand 0 "" "")))]
 ""
{
 return (get_attr_length (insn) == 4
     ? "b %l0" : "l r15,=a(%l0); br r15");
}
 [(set (attr "length")
    (if_then_else (lt (match_dup 0) (const_int 4096))
          (const_int 4)
          (const_int 6)))])
```

16.19.6 常量属性

一个define_attr的特殊形式，是其缺省值的表达式为一个const表达式， 这表示了对于一个给定的运行编译器的一个属性为常量。 常量熟性可以用于指定使用了哪个处理器的变种。例如，

```
(define_attr "cpu" "m88100,m88110,m88000"
 (const
  (cond [(symbol_ref "TARGET_88100") (const_string "m88100")
     (symbol_ref "TARGET_88110") (const_string "m88110")]
    (const_string "m88000"))))

(define_attr "memory" "fast,slow"
 (const
  (if_then_else (symbol_ref "TARGET_FAST_MEM")
        (const_string "fast")
        (const_string "slow"))))
```

针对常量属性生成的程序不具有任何参数，因为它不依赖于任何特定的insn。 用于定义常量属性的RTL表达式可以使用symbol_ref形式， 但是不可以使用包括insn属性的match_operand形式或者eq_attr形式。

16.19.7 延迟槽调度

如果在一个目标机上存在延迟槽的话，insn属性机制可以用于指定对延迟槽的需求。 一条指令被称为需要延迟槽， 如果在物理上位于该指令之后的一些指令将被按照它们仿佛是位于之前的情况被执行。 典型的例子是分支和调用指令，其通常在执行分支或调用之前先执行后面的指令。

在一些机器上，条件分支指令可以选择性的废除（annul）延迟槽中的指令。 这意味着该指令对于特定的分支结果将不被执行。 对于分支为真时废除指令和分支为假时废除指令，这两种方式都被支持。

延迟槽调度与指令调度的不同之处在于， 判定一条指令是否需要延迟槽只依赖于正被生成的指令的类型，而不是指令间的数据流。 关于数据相关的指令调度的讨论，参见下一个章节。

一个insn对一个或多个延迟槽的需求是通过define_delay表达式来表示的。 其具有下列形式：

```
(define_delay test
       [delay-1 annul-true-1 annul-false-1
        delay-2 annul-true-2 annul-false-2
        ...])
```

test是一个属性测试，用来表示该define_delay是否应用到特定的insn。 如果是，则所需延迟槽的数目通过作为第二个参数的向量的长度来确定。 放在延迟槽n中的insn必须满足属性测试delay-n。 annul-true-n是一个属性测试，用来指定当分支为真时哪些insn可以被废除。 如果该延迟槽不支持废除，则应该使用(nil)。

例如，通常情况下分支和调用insns都需要一个单个的延迟槽， 其可以包含任何不是分支或调用的其它insn，则下面的可以放入`md'文件中：

```
(define_delay (eq_attr "type" "branch,call")
       [(eq_attr "type" "!branch,call") (nil) (nil)])
```

可以指定多个define_delay表达式。在这种情况下， 每个这样的表达式都指定了不同的延迟槽需求， 并且在两个define_delay表达式中的test必须不能都为真。

例如，如果我们有一个机器，其对分支需要一个延迟槽，但对调用需要两个， 延迟槽不能包含分支或调用insn，并且当分支为真时， 任何在分支延迟槽中的有效insn可以被废除，则我们可以使用下列方式来表示：

```
(define_delay (eq_attr "type" "branch")
  [(eq_attr "type" "!branch,call")
   (eq_attr "type" "!branch,call")
   (nil)])

(define_delay (eq_attr "type" "call")
       [(eq_attr "type" "!branch,call") (nil) (nil)
        (eq_attr "type" "!branch,call") (nil) (nil)])
```

16.19.8 处理器流水线描述

为了获得更好的性能，大多数现代处理器（超流水线，超标量RISC， 以及VLIW处理器）都具有许多功能单元（functional units）， 可以在其上同时执行多条指令。一条指令当它的发射条件（issue conditions） 被满足时才开始执行。如果不满足，则指令会被阻塞（stalled），直到它的条件被满足。 这样的互锁（流水线）延迟(interlock (pipeline) delay) 导致对后续指令读取的中断（或者需要nop指令，例如一些MIPS处理器）。

现代处理器中有两种主要的互锁延迟。第一种为数据依赖延迟，用来确定指令延迟时间（instruction latency time）。直到所有源数据都被先前指令求得， 该指令才会开始执行（有更加复杂的情况是， 当指令开始执行时数据还不可用，但是将会在指令开始执行后的给定时间准备好）。考虑数据依赖延迟是简单的。 两个指令间的数据依赖（真依赖，输出依赖，反依赖）延迟被给定为一个常量。 大多数情况下该方法都适合。第二种互锁延迟为保留延迟（reservation delay）。 保留延迟意味着要执行的两条指令将会需要共享的处理器资源，即总线，内部寄存器， 以及/或者功能单元，而这些将被保留一段时间。考虑这种延迟是复杂的， 特别是对于现代RISC处理器。

探索更多的处理器并行的任务是由指令调度器来解决的。为了能够更好的解决该问题， 指令调度器必须具有一个处理器并行的适当描述（或者说流水线描述）。 GCC机器描述使用正规表达式来描述处理器并行和对指令组的功能单元保留。

GCC指令调度器使用流水线冒险识别器通过给定的处理器时钟周期模拟来找出可能的指令问题。流水线冒险识别器通过处理器流水线描述自动生成。 由机器描述生成的流水线冒险识别器是基于有限确定状态机（DFA）： 如果存在从一个自动机状态到另一状态的转换，则可以进行指令发射。 该算法非常快，而且它的速度不依赖于处理器的复杂度[4]。

该章节的剩余部分描述了构造一个基于自动机的处理器流水线描述的命令（directive）。 这些结构在机器描述文件中的顺序并不重要。

下面的可选结构描述了生成的自动机的名字，并用于流水线冒险识别。 有时供流水线冒险识别器使用的生成的有限状态机会非常大。 如果我们使用多个自动机并且将功能单元绑定到自动机上， 则自动机的总的大小通常会小于单个自动机的情况。 如果没有这样一个结构，则会只生成一个有限状态机。

[4] 然而，自动机的大小依赖于处理器的复杂度。为了限制这种影响， 机器描述可以将机器描述的正交部分拆分成多个自动机：但是， 由于每个这样的自动机都必须独立的执行每一步， 所以这确实会在算法性能上造成一点消减。

```
(define_automaton automata-names)
```

automata-names为一个字符串，给出了自动机的名字。 名字由逗号分隔。所有自动机应该具有唯一的名字。 自动机名用于结构define_cpu_unit和define_query_cpu_unit。

用于指令保留描述的每个处理器功能单元应该使用下列结构来描述。

```
(define_cpu_unit unit-names [automaton-name])
```

unit-names为一个字符串，给出了由逗号分隔的功能单元的名字。 不要使用名字`nothing'，它被保留用于其它目的。

automaton-name为一个字符串，给出了功能单元绑定的自动机名。 自动机应该在结构define_automaton中有描述。 如果有一个定义的自动机，则你应该给出automaton-name。

为功能单元赋予自动机，受到insn保留中对功能单元使用的限制。 最重要的constraint为：如果一个功能单元保留。 其余的constraint将在后续的结构描述中提到。

下面的结构描述了CPU功能单元，类似于define_cpu_unit。 对于这样的功能单元的保留，可以被询问自动机状态。 对于给定的自动机状态，指令调度器从来不询问功能单元的保留。 所以按照规则，你不需要该结构。 该结构可以被用于将来的代码生成目的（例如，生成VLIW insn模板）。

```
(define_query_cpu_unit unit-names [automaton-name])
```

unit-names为一个字符串，给出了由逗号分隔的功能单元名字。

automaton-name为一个字符串，给出了功能单元所绑定的自动机。

下面的结构为描述一条指令的流水线特征的主要结构。

```
(define_insn_reservation insn-name default_latency
                         condition regexp)
```

default_latency为一个数，给出了指令的延迟时间。 在旧描述和基于自动机的流水线描述中，有一个重要的不同之处。 当我们使用旧描述时，延迟时间是用于所有的依赖。 在基于自动机的流水线描述中，给定的延迟时间只用于真依赖。 反依赖的代价总为0，并且输出依赖的代价是生产者insn和消费者insn的延迟时间之差（如果差为负数，则代价被认为为0）。 你可以通过使用目标机钩子TARGET_SCHED_ADJUST_COST（参见Section 17.18 [调度], page 349），来改变任何描述的缺省代价。

insn-name为一个字符串，给出了insn的内部名字。 内部名字被用于结构define_bypass和为了调试所生成的自动机描述文件。 内部名字与define_insn中的名字没有任何关系。 使用在处理器手册中描述的insn类别，是一个很好的做法。

condition定义了什么样的RTL insns由该结构描述。 你应该记住如果对于一个insn， 两个或更多不同define_insn_reservation结构的condition都为真， 则会出问题。这种情况，该insn将使用什么保留，是未定义的。 这种情况在流水线冒险识别器生成时，是不被检查的， 因为识别两个条件具有相同值是十分困难的（特别是如果条件中包含symbol_ref）。 这在流水线识别器工作时，也不被检查，因为它将使识别器变得相当慢。

regexp为一个字符串，描述了指令对cpu的功能单元的保留。 保留通过正规表达式来描述，语法如下：

```
regexp = regexp "," oneof
       | oneof

oneof = oneof "|" allof
      | allof

allof = allof "+" repeat
      | repeat

repeat = element "*" number
```

```
        | element

element = cpu_function_unit_name
        | reservation_name
        | result_name
        | "nothing"
        | "(" regexp ")"
```

- `,'用于描述在保留中，下一周期的开始。
- `|'用于描述在保留中，第一个正规表达式**or**第二个正规表达式， **or**等等。
- `+'用于描述在保留中，第一个正规表达式**and**第二个正规表达式， **and**等等。
- `*'用于方便记述，其简单的表示一个正规表达式序列， 表达式随着周期前移被重复number次（参见`,'）。
- `cpu_function_unit_name'表示对命名功能单元的保留。
- `reservation_name' --- 参见对结构`define_reservation'的描述。
- `nothing'表示没有功能单元被保留。

有时，对于不同insn，具有共同部分的单元保留。 这样情况，你可以通过使用下面的结构来描述共同部分，以简化流水线描述。

```
(define_reservation reservation-name regexp)
```

reservation-name为一个字符串，给出了regexp的名字。 功能单元名和保留名属于同一命名空间。所以，保留名应该与功能单元名不同， 并且不能为预留名`nothing'。

下面的结构被用于描述对于给定的指令对，在延迟时间上的例外。也称之为bypass。

```
(define_bypass number out_insn_names in_insn_names
        [guard])
```

number定义了给定字符串out_insn_names的指令所产生的结果， 什么时候可以由给定字符串in_insn_names的指令使用。 字符串中的指令由逗号分隔。

guard为一个可选的字符串，给出了C函数名，其定义了bypass的额外的保护条件。 该函数将两个insn作为参数。如果函数返回0，则对于该情况bypass将被忽略。 额外的guard在识别复杂的bypass时，很有必要。 例如当消费者只是一个insn `store'的地址（而不是被存储的值）。

下面五个结构通常用于描述VLIW处理器，或者更精确的说， 来描述放入VLIW指令槽中的小指令的位置。 它们也可以用于RISC处理器。

```
(exclusion_set unit-names unit-names)
(presence_set unit-names patterns)
(final_presence_set unit-names patterns)
(absence_set unit-names patterns)
(final_absence_set unit-names patterns)
```

unit-names为一个字符串，给出了由逗号分隔的功能单元的名字。

patterns为一个字符串，给出了由逗号分隔的功能单元的模式。 目前的模式，为一个单元或者由空格分隔的单元。

第一个结构(`exclusion_set') 意味着第一个字符串中的每个功能单元不能与 第二个字符串中的功能单元同时被保留，反之亦然。例如，结构可以用于描述处理器 （例如，一些SPARC处理器）具有全流水浮点功能单元， 其只可以同时执行单浮点insn或者双浮点insn。

第二个结构(`presence_set') 意味着第一个字符串中的每个功能单元不能被保留， 除非至少一种模式的功能单元其名字在第二个字符串中且被保留。这是一个不对称关系。 例如，可以用于描述VLIW `slot1'在`slot0'保留之后被保留。 我们可以使用下列结构来描述

```
(presence_set "slot1" "slot0")
```

或者`slot1'只在`slot0'和功能单元`b0'保留之后被保留。 这种情况下，我们可以写成

```
(presence_set "slot1" "slot0 b0")
```

第三个结构(`final_presence_set') 类似于`presence_set'。 区别在于什么时候进行检查。当指令在给定自动机状态被发射时， 其将影响所有当前和计划中的单元保留，并且自动机状态被改变。 第一个状态为源状态，第二个为结果状态。 对于`presence_set'的检查是在源状态保留时进行的， 对于`final_presence_set'的检查是在结果状态下进行的。 该结构可以用于描述实际上是两个连续的保留的保留。例如，如果我们使用

```
(presence_set "slot1" "slot0")
```

下列insn将永远不会被发射（因为`slot1'需要`slot0'， 而`slot0'在源状态是空缺的）

```
(define_reservation "insn_and_nop" "slot0 + slot1")
```

但是如果我们使用类似的`final_presence_set'其就可以被发射。

第四个结构 (`absence_set') 意味着在第一个字符串中的每个功能单元， 只有在每个名字在第二个字符串中的功能单元没有被保留时才能被保留。 这是一个不对称关系（实际上`exclusion_set'与其类似，但它是对成的）。 例如，可以用于VLIW描述，来表示`slot0'不能在`slot1'或 `slot2'保留后被保留。这可以描述为

```
(absence_set "slot0" "slot1, slot2")
```

或者`slot2'不能被保留，如果`slot0'和单元`b0'被保留， 或者`slot1'和单元`b1'被保留. 这种情况下，我们可以写成

```
(absence_set "slot2" "slot0 b0, slot1 b1")
```

所有在集合（set）中提到的功能单元应属于相同的自动机。

最后一个结构(`final_absence_set')类似于`absence_set'， 但是检查是在结果（状态）保留时进行。参见`final_presence_set'的注解。

你可以使用下面的结构来控制流水线冒险识别器的生成。

```
(automata_option options)
```

options为一个字符串，给出了影响生成代码的选项。目前有下列选项：

- no-minimization不对自动机进行最小化处理。 这只在我们进行调试描述信息并且需要更加精确的查看保留状态时，才值得做。
- time意味着打印生成自动机的时间统计。
- stats意味着打印生成自动机的DFA状态，NDFA状态和arcs这样的数目统
- v意味着生成一个描述生成自动机的文件。文件具有后缀`.dfa'， 并且可以用于验证和调试描述。
- w意味着对于非关键的错误使用警告来替代。
- ndfa生成非确定有限状态机。这将影响对正规表达式中操作符`|'的对待。 通常对该操作符的处理是先尝试第一个，然后再第二个。 非确定状态机意味着尝试所有的选择，其中一些可以被后续的insn放弃。
- progress意味着输出一个进度条，来显示被处理的自动机目前生成了多少状态。 这在调试DFA描述时很有用。如果你看到太多的状态被生成， 你可以中断流水线冒险识别器的生成并尝试去弄清楚为什么会生成如此大的自动机。

作为一个例子，考虑一个超标量RISC机器， 其可以在一个周期发射三条insn（两条整数insn和一条浮点insn）， 但是只能完成两条insn。为了描述，我们定义下列功能单元。

```
(define_cpu_unit "i0_pipeline, i1_pipeline, f_pipeline")
(define_cpu_unit "port0, port1")
```

所有简单的整数insn可以在任何整数流水线中被执行，并且结果可以在两个周期获得。 简单的整数insn将被发射到第一个流水线中，除非它被保留， 否则它们将被发射到第二个流水线中。整数除和乘insn只能在第二个整数流水线中被执行， 并且它们的结果相应的在8和4个周期获得。 整数除为非流水线，即后续的整数除insn在当前的除法insn完成前不能被发射。 浮点insn为全流水的并且它们的结果在3个周期获得。 当浮点insn的结果被整数insn使用使，将会产生一个额外的周期延迟。 要描述所有这些，我们可以指定

```
(define_cpu_unit "div")

(define_insn_reservation "simple" 2 (eq_attr "type" "int")
            "(i0_pipeline | i1_pipeline), (port0 | port1)")

(define_insn_reservation "mult" 4 (eq_attr "type" "mult")
            "i1_pipeline, nothing*2, (port0 | port1)")

(define_insn_reservation "div" 8 (eq_attr "type" "div")
            "i1_pipeline, div*7, div + (port0 | port1)")

(define_insn_reservation "float" 3 (eq_attr "type" "float")
            "f_pipeline, nothing, (port0 | port1))

(define_bypass 4 "float" "simple,mult,div")
```

为了简化描述，我们可以描述下列保留

```
(define_reservation "finish" "port0|port1")
```

并在所有define_insn_reservation中使用，比如下面的结构

```
(define_insn_reservation "simple" 2 (eq_attr "type" "int")
            "(i0_pipeline | i1_pipeline), finish")
```

16.20 条件执行

许多体系结构都提供了某种形式的条件执行，或者predicate。其特点是能够使得 指令集中的大多数指令变为无效。当指令集很大并且不完全对称时，在 `.md' 文件中直接描述这些形式将会变得非常冗长。一种可替代的方式 为 define_cond_exec 模板。

```
(define_cond_exec
 [predicate-pattern]
 "condition"
 "output-template")
```

predicate-pattern 为运行时执行insn所需要的必须为真的条件，并且 应该能匹配一个相关的操作符。可以使用 match_operator 来一次匹配 多个相关的操作符。任何 match_operand 操作数必须具有不超过一个 的可选项。

condition 为一个C表达式，对于生成的指令模式必须匹配为真。

output-template 为一个类似于 define_insn 输出模板 （ see Section 16.5 [输出模板], page 206 ）的字符串，除了不应用 `*' 和 `@' 的特殊情况。这只在针对predicate的汇编文本为一个主insn的简单前缀时有用。为了 处理通用的情况，有一个全局变量 current_insn_predicate，在当前 insn被predicate时其将包含整个predicate，否则将为 NULL。

当使用 define_cond_exec 时，将会创建一个对 predicable 指令属性的隐式引用。 see Section 16.19 [Insn属性], page 266。该属性并须为布尔的（即在它的 list-of-values 中具有确切的两个元素）。甚至，其必须不能使用复杂 表达式。也就是，insn中的缺省的和所有的使用都必须为一个简单常量，不能依 赖于可选项或其它。

对于每个 predicable 属性为真的 define_insn，一个新的匹配 一个指令predicate版本的 define_insn 指令模式将被生成。例如，

```
(define_insn "addsi"
 [(set (match_operand:SI 0 "register_operand" "r")
    (plus:SI (match_operand:SI 1 "register_operand" "r")
         (match_operand:SI 2 "register_operand" "r")))]
 "test1"
 "add %2,%1,%0")

(define_cond_exec
 [(ne (match_operand:CC 0 "register_operand" "c")
   (const_int 0))]
 "test2"
 "(%0)")
```

生成一个新的指令模式

```
(define_insn ""
 [(cond_exec
  (ne (match_operand:CC 3 "register_operand" "c") (const_int 0))
  (set (match_operand:SI 0 "register_operand" "r")
     (plus:SI (match_operand:SI 1 "register_operand" "r")
          (match_operand:SI 2 "register_operand" "r"))))]
 "(test2) && (test1)"
 "(%3) add %2,%1,%0")
```

16.21 常量定义

在指令模板中使用文字常量会减小可读性并会成为一个维护问题。

要克服该问题，你可以使用 define_constants 表达式。它包含了一个 “名字-值” 成对向量。从定义处开始，任何出现在MD文件中的那些名字，都作为 相应的值被替换。你可以多次使用 define_constants；每次都为表格中 增加更多的常量。使用不同的值来重新定义一个常量将会产生一个错误。

回到a29k加载乘的例子，对于

```
(define_insn ""
 [(match_parallel 0 "load_multiple_operation"
  [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
     (match_operand:SI 2 "memory_operand" "m"))
   (use (reg:SI 179))
   (clobber (reg:SI 179))])]
 ""
 "loadm 0,0,%1,%2")
```

你可以写成:

```
(define_constants [
  (R_BP 177)
  (R_FC 178)
  (R_CR 179)
  (R_Q 180)
])

(define_insn ""
 [(match_parallel 0 "load_multiple_operation"
  [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
     (match_operand:SI 2 "memory_operand" "m"))
   (use (reg:SI R_CR))
   (clobber (reg:SI R_CR))])]
```

```
""
"loadm 0,0,%1,%2")
```

使用define_constants定义的常量也在insn-codes.h头文件中作为#defines被输出。

16.22 迭代器

后端（Ports）常常需要为多个机器模式或者多个rtx代码定义类似的指令模式。 GCC提供了一些简单的迭代机制使得该处理变的很容易。

16.22.1 机器模式迭代器

后端（Ports）经常需要为两个或多个不同机器模式定义类似的指令模式。例如：

- 如果一个处理器对单浮点和双浮点算术都具有硬件支持，则 SFmode 的 指令模式将会与 DFmode 的非常类似。
- 如果一个后端（port）在一个配置中使用 SImode 的指针，而在另一个 配置中使用 DImode 的指针，则通常会具有非常类似的操作指针的 SImode 和 DImode 的指令模式。

机器模式迭代器允许从一个 `.md' 文件模板实例化多个指令模式。它们 可以用于任何类型的基于rtx的结构，例如 define_insn, define_split 或 define_peephole2。

16.22.1.1 定义机器模式迭代器

定义一个机器模式迭代器的语法为：

```
(define_mode_iterator name [(mode1 "cond1") ... (moden "condn")])
```

这将允许后续的 `.md' 文件结构可以使用机器模式suffix :name。每个这样的结构将被扩展 n 次，一次使用 :mode1 来替换，一次使用 :mode2 来替换，等等。 在扩展 modei 时，每个C条件 condi 还要为真。

例如：

```
(define_mode_iterator P [(SI "Pmode == SImode") (DI "Pmode == DImode")])
```

定义了一个新的机器模式后缀 :P。每个使用 :P 的结构将被扩展 两次，一次由 :SI 来替换 :P，一次由 :DI 来替换 :P。其中 :SI 版本的只有当 Pmode == SImode 时才被应 用，:DI 版本的只有当 Pmode == DImode 时才被应用。

就像其它 `.md' 条件，一个空字符串被当作“总为真”。(mode "") 也可以被缩写为 mode。例如：

```
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
```

意味着 :DI 扩展只有为 TARGET_64BIT 时被应用，但 :SI 扩展却没有这样的限制。

迭代器按照它们定义的顺序被应用。这在当两个迭代器用于一个结构中需要替换时 会变的很重要。See Section 16.22.1.2 [替换], page 279。

16.22.1.2 机器模式迭代器中的替换

如果一个 `.md' 文件结构使用了机器模式迭代器，则结构的每个版本将通常 需要轻微不同的字符串或机器模式。例如：

- 当一个 define_expand 定义了多个 addm3 指令模式时（see Section 16.9 [标准名字], page 235），每个扩展将需要针对 m 的适当的机器模式。
- 当一个 define_insn 定义了多个指令模式时，每条指令将通常需要一个不 同的汇编助记符。
- 当一个 define_insn 需要操作数具有不同的机器模式时，针对一个操作数 的机器模式使用迭代器通常需要对于其它操作数使用特点的机器模式。

GCC通过“机器模式属性”系统来支持这样的变种。有两种标准属性：mode， 其为机器模式的小写名字，MODE，其为机器模式的大写名字。你可以定义其 它属性，使用：

```
(define_mode_attr name [(mode1 "value1") ... (moden "valuen")])
```

其中 name 为属性的名字，valuei 为与 modei 关联的值。

当GCC使用 :mode 来替换某个 :iterator 时，其将扫描指令模式中的 每个字符串和机器模式，按照 <iterator:attr>形式的序列， 其中attr为一个机器模式属性的名字。如果属性针对mode被定义，则整个 <...> 序列将被适当的属性值替换。

例如，假设一个 `.md' 具有：

```
(define_mode_iterator P [(SI "Pmode == SImode") (DI "Pmode == DImode")])
(define_mode_attr load [(SI "lw") (DI "ld")])
```

如果其中一个使用 :P 的指令模式包含了字符串 "<P:load>\t%0,%1"，则 SI 版本的指令模式将使用 "lw\t%0,%1" 并且 DI 版本的将使用 "ld\t%0,%1"。

这里有一个关于使用针对一个机器模式的属性的例子：

```
(define_mode_iterator LONG [SI DI])
(define_mode_attr SHORT [(SI "HI") (DI "SI")])
(define_insn ...
 (sign_extend:LONG (match_operand:<LONG:SHORT> ...)) ...)
```

iterator: 前缀可以被省略掉，这种情况下将会针对每个迭代器扩展来 尝试替换。

16.22.1.3 有关机器模式迭代器的例子

这里有一个来自MIPS后端的一个例子。其定义了下列机器模式和属性（除了别的以外）：

```
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
(define_mode_attr d [(SI "") (DI "d")])
```

并且使用下列模板来同时定义 subsi3 和 subdi3：

```
(define_insn "sub<mode>3"
 [(set (match_operand:GPR 0 "register_operand" "=d")
    (minus:GPR (match_operand:GPR 1 "register_operand" "d")
          (match_operand:GPR 2 "register_operand" "d")))]
 ""
 "<d>subu\t%0,%1,%2"
 [(set_attr "type" "arith")
  (set_attr "mode" "<MODE>")])
```

这就完全等价于：

```
(define_insn "subsi3"
 [(set (match_operand:SI 0 "register_operand" "=d")
    (minus:SI (match_operand:SI 1 "register_operand" "d")
          (match_operand:SI 2 "register_operand" "d")))]
 ""
 "subu\t%0,%1,%2"
 [(set_attr "type" "arith")
  (set_attr "mode" "SI")])

(define_insn "subdi3"
 [(set (match_operand:DI 0 "register_operand" "=d")
    (minus:DI (match_operand:DI 1 "register_operand" "d")
          (match_operand:DI 2 "register_operand" "d")))]
 ""
 "dsubu\t%0,%1,%2"
 [(set_attr "type" "arith")
  (set_attr "mode" "DI")])
```

16.22.2 代码迭代器

代码迭代器使用与机器模式迭代器类似的方法来操作。see Section 16.22.1 [机器模式迭代器], page 279。

结构：

```
(define_code_iterator name [(code1 "cond1") ... (coden "condn")])
```

定义了一个伪rtx代码 name，如果条件 condi 为真时，其可以作为 codei 被实例化。每个 codei 必须具有相同的rtx格式。 see Section 10.2 [RTL类别], page 109。

跟机器模式迭代器一样，每个使用 name 的指令模式将被扩展 n 次， 一次使用 code1 来替换，一次使用 code2，等等。 See Section 16.22.1.1 [定义机器模式迭代器], page 279.

跟机器模式一样可以为代码定义属性。有两种标准代码属性：code，代码 的小写形式名字，以及 CODE，代码的大写形式名字。其它属性使用下列 方式来定义：

```
(define_code_attr name [(code1 "value1") ... (coden "valuen")])
```

这里有一个实际使用的关于代码迭代器的例子，摘自MIPS后端（port）：

```
(define_code_iterator any_cond [unordered ordered unlt unge uneq ltgt unle ungt
                                eq ne gt ge lt le gtu geu ltu leu])

(define_expand "b<code>"
 [(set (pc)
    (if_then_else (any_cond:CC (cc0)
                               (const_int 0))
                  (label_ref (match_operand 0 ""))
                  (pc)))]
 ""
{
 gen_conditional_branch (operands, <CODE>);
 DONE;
})
```

这等价于：

```
(define_expand "bunordered"
 [(set (pc)
    (if_then_else (unordered:CC (cc0)
                                (const_int 0))
                  (label_ref (match_operand 0 ""))
                  (pc)))]
 ""
{
 gen_conditional_branch (operands, UNORDERED);
 DONE;
})

(define_expand "bordered"
 [(set (pc)
    (if_then_else (ordered:CC (cc0)
                              (const_int 0))
                  (label_ref (match_operand 0 ""))
                  (pc)))]
 ""
{
 gen_conditional_branch (operands, ORDERED);
 DONE;
})

...
```

17 目标机描述宏和函数

除了文件`machine.md'，机器描述还包括一个通常名为`machine.h'的C头文件和一个名为`machine.c'的C源文件。头文件定义了许多用来传达不适合`.md'文件框架的目标机器信息的宏。文件`tm.h'应该为`machine.h'的一个链接。头文件`config.h'包含`tm.h'，并且大多数编译器源文件包含`config.h'。源文件定义了变量targetm，其为一个包含了与目标机器相关的函数指针和数据的结构体。如果它们没有在GCC中的其它地方定义，`machine.c'还应该包含它们的定义，其它通过宏调用的函数在`.h'文件中定义。

17.1 全局变量targetm

struct gcc_target targetm [Variable]

目标机`.c'文件必须定义包含了目标机器相关的函数指针和数据的全局变量 targetm。该变量在`target.h'中声明；`target-def.h'定义了用来初始化该变量的宏TARGET_INITIALIZER，和结构体元素的缺省初始化宏。`.c'文件应该覆写这些缺省定义不合适的宏。例如：

```
#include "target.h"
#include "target-def.h"

/* Initialize the GCC target structure.  */

#undef TARGET_COMP_TYPE_ATTRIBUTES
#define TARGET_COMP_TYPE_ATTRIBUTES machine_comp_type_attributes

struct gcc_target targetm = TARGET_INITIALIZER;
```

其中，宏应该通过这种方式在`.c'文件中被定义，从而成为targetm结构体的一部分。该宏将在下面作为具有函数原型的“目标钩子”来介绍。在`.h'.h文件中定义的许多宏将来都会改为targetm结构体的一部分。

17.2 控制编译驱动器，`gcc'

你可以控制编译驱动器。

SWITCH_TAKES_ARG (char) [Macro]

一个C表达式，用来确定选项`-char'是否接受参数。值应该为那个选项接受的参数个数，对于许多选项其为0。

缺省情况下，该宏被定义为DEFAULT_SWITCH_TAKES_ARG，其可以正常处理标准的选项。除非你希望增加额外的接受参数的选项，否则不需要定义SWITCH_TAKES_ARG。任何重定义都应该先调用DEFAULT_SWITCH_TAKES_ARG，然后再检查额外的选项。

WORD_SWITCH_TAKES_ARG (name) [Macro]

一个C表达式，用来确定选项`-name'是否接受参数。值应该为那个选项接受的参数个数，对于许多选项其为0。该宏不同于SWITCH_TAKES_ARG，是用于选项名为多个字符的情况。

缺省情况下，该宏被定义为DEFAULT_WORD_SWITCH_TAKES_ARG，其可以正常处理标准的选项。除非你希望增加额外的接受参数的选项，否则不需要定义WORD_SWITCH_TAKES_ARG。任何重定义都应该先调用DEFAULT_WORD_SWITCH_TAKES_ARG，然后再检查额外的选项。

SWITCH_CURTAILS_COMPILATION (char) [Macro]

一个C表达式，用来确定选项`-char'是否在生成可执行程序之前停止编译。值为布尔型，如果选项确实停止生成可执行程序则为非0，否则为0。

缺省情况下，该宏被定义为DEFAULT_SWITCH_CURTAILS_COMPILATION，其可以正常处理标准的选项。除非你希望增加额外的影响可执行程序生成的选项，否则不需要定义SWITCH_CURTAILS_COMPILATION。任何重定义都应该先调用DEFAULT_SWITCH_CURTAILS_COMPILATION，然后再检查额外的选项。

SWITCHES_NEED_SPACES [Macro]

一个值为字符串的C表达式，用来枚举连接器在选项和其参数之间需要一个空格的那些选项。

如果该宏没有被定义，则缺省值为""。

TARGET_OPTION_TRANSLATE_TABLE [Macro]

如果定义，则为字符串对（pairs of strings）列表，其中第一个字符串为`gcc'驱动程序的一个可能的命令行目标，第二个字符串为空格分隔的（不支持tab和其它whitespace）选项列表，用来替换第一个选项。定义该列表的目标机要负责确保结果是有效的。替换选项不可以为--opt风格的，它们必须为-opt风格的。该宏的目的是为选择multilib提供一种替换机制，例如通过一个选项可以打开许多选项，其中一些用来选择multilib。例如，在下面这个无意义的定义中，`-malt-abi',`-EB'和`-mspoo'会导致不同的multilib被选择：

```
#define TARGET_OPTION_TRANSLATE_TABLE \
{"-fast", "-march=fast-foo -malt-abi -I/usr/fast-foo"}, \
{"-compat", "-EB -malign=4 -mspoo"}
```

DRIVER_SELF_SPECS [Macro]

驱动器本身的specs列表。其应该为一个字符串数组的适当的初始化值，并且不使用大括号包裹。

驱动器将这些specs应用到它自己的命令行上，位于加载缺省`specs'文件（而不是由命令行指定的）和选择multilib目录或者运行任何子命令之间。驱动器按照给定的顺序来应用它们，所以每个spec可以依赖于先前增加的选项。还可以使用通常的方式，用`%<option'来移除选项。

当一个port具有多个相互依赖的目标机选项时，该宏会有帮助。它提供了一种标准化命令行的方法，使得其它specs的书写变得容易些。

如果不需要做任何事情的话，则不要定义该宏。

OPTION_DEFAULT_SPECS [Macro]

一个specs列表，用于支持驱动器中的配置时（configure-time）缺省选项（即，`--with'选项）。其应该为一个结构体数组的适当的初始化值，每个元素包含两个字符串，并且不带有最外层的大括号对。

字符串对的第一项为缺省名字。其必须匹配目标机的`config.gcc'中的代码。第二项为一个spec，当那个缺省名字被指定的时候会被应用。在spec中，字符串`%(VALUE)'出现的所有地方都将被缺省值替换。

驱动器将这些specs应用到它自己的命令行上，位于加载缺省`specs'文件和处理DRIVER_SELF_SPECS之间，使用与DRIVER_SELF_SPECS相同的机制。

如果不需要做任何事情的话，则不要定义该宏。

CPP_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序要传给CPP的选项。其还可以指定如何将你传给GCC的选项转换成GCC传给CPP的选项。

如果不需要做任何事情的话，则不要定义该宏。

CPLUSPLUS_CPP_SPEC [Macro]

该宏类似于CPP_SPEC，只不过是用于C++而不是C。如果你不定义该宏，则会使用CPP_SPEC的值（如果存在）来替代。

CC1_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序要传给cc1,cc1plus,f771和其它语言前端的选项。 其还可以指定如何将你传给GCC的选项转换成GCC传给前端的选项。

如果不需要做任何事情的话，则不要定义该宏。

CC1PLUS_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序要传给cc1plus的选项。 其还可以指定如何将你传给GCC的选项转换成GCC传给cc1plus的选项。

如果不需要做任何事情的话，则不要定义该宏。 注意在CC1_SPEC中定义的所有选项已经被传给cc1plus， 所以不需要在CC1PLUS_SPEC中重复CC1_SPEC的内容。

ASM_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序要传给汇编器的选项。 其还可以指定如何将你传给GCC的选项转换成GCC传给汇编器的选项。 参见文件`sun3.h'中的例子。

如果不需要做任何事情的话，则不要定义该宏。

ASM_FINAL_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序如何在运行正常的汇编器之后，来运行任何清除程序。 通常，不需要该宏。参见文件`mips.h'中的例子。

如果不需要做任何事情的话，则不要定义该宏。

AS_NEEDS_DASH_FOR_PIPED_INPUT [Macro]

如果驱动器应该传给汇编器一个由单横线`-'组成的参数， 来指示它从标准输入（其将为一个与编译器输出相连接的管道）读取时， 则定义该宏，并且不需要给出定义值。 该参数在任何指定输出文件名的`-o'选项后面被给出。

如果不定义该宏，则汇编器被认为在没有传给任何非选项参数时，才从标准输入读取。 如果你的汇编器根本不能从标准输入读取，则使用`%{pipe:%e}'； 参见`mips.h'中的例子。

LINK_SPEC [Macro]

一个C字符串常量，告诉GCC驱动程序要传给连接器的选项。 其还可以指定如何将你传给GCC的选项转换成GCC传给连接器的选项。

如果不需要做任何事情的话，则不要定义该宏。

LIB_SPEC [Macro]

另一个C字符串常量，与LINK_SPEC的用法很相似。 两者的区别是LIB_SPEC用于传给连接器的命令的结尾处。

如果该宏没有被定义，则提供缺省的方式，从通常的地方来加载标准C库。 参见`gcc.c'。

LIBGCC_SPEC [Macro]

另一个C字符串常量， 告诉GCC驱动程序如何以及什么时候将对`libgcc.a'的引用放到连接器的命令行中。 该常量同时被放在LIB_SPEC的值的前面和后面。

如果该宏没有被定义，则GCC驱动器提供了一个缺省的方式， 将字符串`-lgcc'传给连接器。

REAL_LIBGCC_SPEC [Macro]

缺省情况下，如果ENABLE_SHARED_LIBGCC被定义， 则LIBGCC_SPEC不直接被驱动程序使用，而是根据命令行标记`-static',`-shared',`-static-libgcc'和`-shared-libgcc'的值， 进行修改，从而引用`libgcc.a'的不同版本。在一些目标机上，这些修改并不合适， 这样就可以定义REAL_LIBGCC_SPEC。 REAL_LIBGCC_SPEC告诉驱动器如何将对`libgcc'的引用放到连接器命令行中， 不过不像LIBGCC_SPEC，它不经过修改，被直接使用。

USE_LD_AS_NEEDED [Macro]

一个宏，用于控制在REAL_LIBGCC_SPEC中提到的对LIBGCC_SPEC的修改。 如果非0，则会生成一个spec，当不使用任何-static,-static-libgcc或-shared-libgcc来连接的时候， 将使用–as-needed和静态异常处理库所在位置的共享libgcc。

LINK_EH_SPEC [Macro]

如果定义，则该C字符串常量被增加到LINK_SPEC中。 当USE_LD_AS_NEEDED为0或者未定义时，其还影响在REAL_LIBGCC_SPEC中提到的对LIBGCC_SPEC的修改。

STARTFILE_SPEC [Macro]

另一个C字符串常量，与LINK_SPEC的用法很相似。 两者的不同之处是STARTFILE_SPEC用于传给连接器的命令的最开始处。

如果该宏没有被定义，则会提供一种缺省方式， 从通常的地方来加载标准C启动（startup）文件。参见`gcc.c'。

ENDFILE_SPEC [Macro]

另一个C字符串常量，与LINK_SPEC的用法很相似。 两者的不同之处是ENDFILE_SPEC用于传给连接器的命令的最末尾处。

如果不需要做任何事情的话，则不要定义该宏。

THREAD_MODEL_SPEC [Macro]

GCC -v将会打印GCC被配置使用的线程模式。然而，在有些平台上这是无法工作的， 像AIX 4.3。对于这样的平台，将THREAD_MODEL_SPEC定义为没有空格的字符串来 命名可识别的线程模式的名字。%*为该宏的缺省值， 将扩展为在`config.gcc'中设置的thread_file的值。

SYSROOT_SUFFIX_SPEC [Macro]

定义该宏，当GCC使用sysroot被配置时，来为目标机的sysroot增加一个suffix。 这将导致GCC在sysroot+suffix下查找usr/lib等。

SYSROOT_HEADERS_SUFFIX_SPEC [Macro]

定义该宏，当GCC使用sysroot被配置时，来为目标机的sysroot增加一个headers_suffix。这将导致GCC将更新的sysroot+headers_suffix传给CPP， 使其在sysroot+headers_suffix下查找usr/include等。

EXTRA_SPECS [Macro]

定义该宏，来提供放在`specs'文件中， 可以被各种specification，像CC1_SPEC，使用的额外的specification。

定义应该为一个结构体数组的初始化值，其包含一个字符串常量， 定义了specification的名字，以及一个字符串常量，提供相应的specification。

如果不需要做任何事情的话，则不要定义该宏。

当一种体系结构包含多个相关的目标机，所具有的各种..._SPECS彼此很相似 的时候，EXTRA_SPECS会很有用，维护者会希望有一个地方来集中保存这些定义。

例如，PowerPC System V.4的目标机使用EXTRA_SPECS， 在System V调用序列被使用时，来定义_CALL_SYSV， 在较老的基于AIX的调用序列被使用时，来定义_CALL_AIX。

`config/rs6000/rs6000.h'目标机文件这样定义:

```
#define EXTRA_SPECS \
  { "cpp_sysv_default", CPP_SYSV_DEFAULT },
```

```
#define CPP_SYS_DEFAULT ""
```

`config/rs6000/sysv.h'目标机文件这样定义:

```
#undef CPP_SPEC
#define CPP_SPEC \
"%{posix: -D_POSIX_SOURCE } \
%{mcall-sysv: -D_CALL_SYSV } \
%{!mcall-sysv: %(cpp_sysv_default) } \
%{msoft-float: -D_SOFT_FLOAT} %{mcpu=403: -D_SOFT_FLOAT}"

#undef CPP_SYSV_DEFAULT
#define CPP_SYSV_DEFAULT "-D_CALL_SYSV"
```

而`config/rs6000/eabiaix.h'目标机文件将CPP_SYSV_DEFAULT定义为:

```
#undef CPP_SYSV_DEFAULT
#define CPP_SYSV_DEFAULT "-D_CALL_AIX"
```

LINK_LIBGCC_SPECIAL_1 [Macro]

定义该宏，如果驱动程序应该找到库`libgcc.a'。 如果没有定义该宏，则驱动程序将参数`-lgcc'传给连接器，告诉连接器来查找。

LINK_GCC_C_SEQUENCE_SPEC [Macro]

指定给连接器的libgcc和libc的顺序。缺省为%G %L %G。

LINK_COMMAND_SPEC [Macro]

一个C字符串常量，给出执行连接器所需的完整的命令行。当定义时， 每次在`gcc.c'中对连接器命令行的修改，你都需要更新你的port。 因此，只有当你需要完全重定义所调用的连接器的命令行，并且没有其他方式来完成时， 才定义该宏。可以通过LINK_GCC_C_SEQUENCE_SPEC来替代该宏。

LINK_ELIMINATE_DUPLICATE_LDIRECTORIES [Macro]

一个非零值， 使得collect2从连接命令中移除重复的`-Ldirectory'搜索目录。 如果移除重复的搜索目录会改变连接器的语法，则不要定义为非零。

MULTILIB_DEFAULTS [Macro]

定义该宏为字符串数组的初始化C表达式，用来告诉驱动器程序对于该目标机， 哪些选项是缺省的，因此当使用MULTILIB_OPTIONS时，不需要单独处理。

如果在target makefile片段中没有定义MULTILIB_OPTIONS， 或者如果在MULTILIB_OPTIONS中列出的选项中没有被设为缺省的， 则不要定义该宏。参见Section 19.1 [目标机片段], page 404。

RELATIVE_PREFIX_NOT_LINKDIR [Macro]

定义该宏来告诉gcc， 如果前缀指示了一个绝对的文件名， 则其应该只将`-B'的前缀转换成`-L'连接器选项，

MD_EXEC_PREFIX [Macro]

如果定义，该宏为一个可选前缀，其在STANDARD_EXEC_PREFIX之后进行尝试。 当使用`-b'选项，或者编译器被构建为交叉编译器时， 将不搜索MD_EXEC_PREFIX。如果你定义了MD_EXEC_PREFIX， 则要确保将其增加到`configure.in'中用于查找汇编器的目录列表中。

STANDARD_STARTFILE_PREFIX [Macro]

定义该宏为C字符串常量， 如果你希望覆盖将libdir作为前缀来搜索起始文件`crt0.o'等的标准选择。 当构建为交叉编译器时，STANDARD_STARTFILE_PREFIX不被搜索。

STANDARD_STARTFILE_PREFIX_1 [Macro]

定义该宏为C字符串常量，如果你希望覆盖在缺省前缀之后， 将/lib作为前缀来搜索起始文件`crt0.o'等的标准选择。 当构建为交叉编译器时，STANDARD_STARTFILE_PREFIX_1不被搜索。

STANDARD_STARTFILE_PREFIX_2 [Macro]

定义该宏为C字符串常量，如果你希望覆盖在缺省前缀之后， 将/lib作为前缀来搜索起始文件`crt0.o'等的标准选择。 当构建为交叉编译器时，STANDARD_STARTFILE_PREFIX_2不被搜索。

MD_STARTFILE_PREFIX [Macro]

如果定义，该宏提供了额外的前缀，在标准前缀之后被尝试。 当使用`-b'选项，或者编译器被构建为交叉编译器时， 将不搜索MD_EXEC_PREFIX。

MD_STARTFILE_PREFIX_1 [Macro]

如果定义，该宏提供了另一个额外的前缀，在标准前缀之后被尝试。 当使用`-b'选项，或者编译器被构建为交叉编译器时，其将不被搜索。

INIT_ENVIRONMENT [Macro]

定义该宏为C字符串常量，如果你希望为驱动器调用的程序，例如汇编器和连接器， 来设置的环境变量。驱动器将该宏的值传给putenv来初始化需要的环境变量。

LOCAL_INCLUDE_DIR [Macro]

定义该宏为C字符串常量，如果你希望覆写当尝试搜索局部头文件时， 将`/usr/local/include'作为缺省前缀的标准选择。 在搜索顺序中，LOCAL_INCLUDE_DIR位于SYSTEM_INCLUDE_DIR之前。

交叉编译器既不搜索`/usr/local/include'，也不搜索它的替换者。

MODIFY_TARGET_NAME [Macro]

定义该宏，如果你希望定义命令行开关来修改缺省的目标机名字。

对于每个开关，你可以包含一个字符串，用来追加到配置名字的第一部分， 或者如果有的话，从配置名字中删除。该定义为一个结构体数组的初始化值。 每个数组元素具有三个元素：开关名（字符串常量，包括起始的横线）， 一个枚举常量ADD或者DELETE，来表示插入或者删除字符串， 以及要插入或者删除的字符串（字符串常量）。

例如，在一台机器上，配置名字的结尾为`64'表示其为64位目标机， 你想使用`-32'和`-64'开关来选择32位和64位目标机，则代码为

```
#define MODIFY_TARGET_NAME \
  { {"-32", DELETE, "64"}, \
    {"-64", ADD, "64"}}
```

SYSTEM_INCLUDE_DIR [Macro]

定义该宏作为C字符串常量， 如果你希望指定一个系统特定的目录在标准目录之前来搜索头文件。 在搜索顺序中，SYSTEM_INCLUDE_DIR位于STANDARD_INCLUDE_DIR之前。

交叉编译器不使用该宏，并且不搜索所指定的目录。

STANDARD_INCLUDE_DIR [Macro]

定义该宏为C字符串常量， 如果你希望覆写将`/usr/include'作为缺省前缀来尝试搜索头文件的标准选择。

交叉编译器忽略该宏，并且不搜索`/usr/include'和它的替换者。

STANDARD_INCLUDE_COMPONENT [Macro]

“component” 对应于STANDARD_INCLUDE_DIR。关于组件的描述，参见下面的INCLUDE_DEFAULTS。如果你没有定义该宏，则不使用组件。

INCLUDE_DEFAULTS [Macro]

定义该宏，如果你希望覆写include文件的全部缺省搜索路径。对于一个本地编译器，缺省搜索路径通常由GCC_INCLUDE_DIR, LOCAL_INCLUDE_DIR, SYSTEM_INCLUDE_DIR, GPLUSPLUS_INCLUDE_DIR和 STANDARD_INCLUDE_DIR组成。另外，GPLUSPLUS_INCLUDE_DIR和GCC_INCLUDE_DIR由`Makefile'自动定义，并指定为GCC的私有搜索区域。目录GPLUSPLUS_INCLUDE_DIR只用于C++程序。

该定义为一个结构体数组的初始化值。每个数组元素具有四个元素：目录名（字符串常量），组件名（也是字符串常量），一个标志用来指示只用于C++，以及一个标志用来表示当编译C++时，对该目录下include进来的代码，不需要使用extern `C'进行包裹。使用空元素来标记数组的结尾。

组件名指出了include文件属于什么GNU包，如果存在，则全部使用大写字母。例如，可能为`GCC'或`BINUTILS'。如果程序包是商家提供的操作系统的一部分，则将名字写为`0'。

例如，这是用于VAX/VMS的定义:

```
#define INCLUDE_DEFAULTS \
{                        \
 { "GNU_GXX_INCLUDE:", "G++", 1, 1},  \
 { "GNU_CC_INCLUDE:", "GCC", 0, 0},   \
 { "SYS$SYSROOT:[SYSLIB.]", 0, 0, 0}, \
 { ".", 0, 0, 0},            \
 {0, 0, 0, 0}              \
}
```

这些是尝试查找exec文件的前缀顺序：

1. 用户使用`-B'指定的前缀。
2. 环境变量GCC_EXEC_PREFIX，或者如果GCC_EXEC_PREFIX没有被设置并且编译器没有被安装到配置时的prefix处，则搜索编译器实际被安装的位置。
3. 环境变量COMPILER_PATH指定的目录。
4. 宏STANDARD_EXEC_PREFIX，如果编译器安装在配置时的prefix处。
5. 位置`/usr/libexec/gcc/'，仅当是本地编译器。
6. 位置`/usr/lib/gcc/'，仅当是本地编译器。
7. 宏MD_EXEC_PREFIX，如果定义，仅当是本地编译器。

这些是尝试查找启动文件的前缀顺序：

1. 用户使用`-B'指定的前缀。
2. 环境变量GCC_EXEC_PREFIX，或者基于工具链的安装位置被自动确定的值。
3. 由环境变量LIBRARY_PATH指定的目录（或者port特定的名字，只用于本地编译器，交叉编译器不使用）。
4. 宏STANDARD_EXEC_PREFIX，但只当工具链被安装在配置的prefix处，或者为本地编译器。
5. 位置`/usr/lib/gcc/'，仅当是本地编译器。
6. 宏MD_EXEC_PREFIX，如果定义，仅当是本地编译器。
7. 宏MD_STARTFILE_PREFIX，如果定义，仅当为本地编译器，或者具有目标机系统根目录（system root）。

8. 宏MD_STARTFILE_PREFIX_1，如果定义， 仅当为本地编译器，或者具有目标机系统根目录。
9. 宏STANDARD_STARTFILE_PREFIX，带有任何sysroot的修改。 如果该路径相关，则会加上GCC_EXEC_PREFIX前缀和机器后缀， 或者STANDARD_EXEC_PREFIX和机器后缀。
10. 宏STANDARD_STARTFILE_PREFIX_1，仅当为本地编译器， 或者具有目标机系统根目录。该宏的缺省值为`/lib/'。
11. 宏STANDARD_STARTFILE_PREFIX_2，仅当为本地编译器， 或者具有目标机系统根目录。该宏的缺省值为`/usr/lib/'。

17.3 运行时的target指定

这些是运行时的target指定。

TARGET_CPU_CPP_BUILTINS () [Macro]

该类函数的宏扩展成一块代码，其定义了target CPU内建的预处理器宏和断言， 使用函数builtin_define, builtin_define_std和 builtin_assert。当前端调用该宏时，其提供一个尾部的分号， 由于其已经结束了命令行选项处理，所以你的代码可以自由的使用那些结果。

builtin_assert接受一个字符串，按照传给命令行选项`-A'的形式， 例如cpu=mips，并且创建一个断言。builtin_define接受一个字符串， 按照传给命令行选项`-D'的形式，并且无条件的定义一个宏。

builtin_define_std接受一个字符串，来表示对象的名字。 如果其不在用户命名空间，则builtin_define_std无条件的定义它。否则， 定义一个具有两个前导下划线的版本，和另一个具有两个前导和后缀的下划线的版本。 例如传递unix，将定义__unix, __unix__以及可能的unix； 传递_mips，将定义__mips, __mips__和可能的_mips， 传递_ABI64，将指定义_ABI64。

你还可以测试被编译的C方言。 变量c_language被设为clk_c, clk_cplusplus或者clk_objective_c。注意，如果我们在预处理汇编， 则该变量将为clk_c，不过类函数的宏preprocessing_asm_p()将返回真， 所以你可能要先检查它。如果你需要检查严格的ANSI，可以使用变量flag_iso。 类函数的宏preprocessing_trad_p()可以用来检查传统的预处理。

TARGET_OS_CPP_BUILTINS () [Macro]

类似于TARGET_CPU_CPP_BUILTINS，不过该宏是可选的，用于target操作系统。

TARGET_OBJFMT_CPP_BUILTINS () [Macro]

类似于TARGET_CPU_CPP_BUILTINS，不过该宏是可选的，用于target目标文件格式。`elfos.h'使用该宏来定义__ELF__，所以你可能不需要自己定义。

extern int target_flags [Variable]

该变量在`options.h'中声明，其在任何target特定的头文件之前被包含进来。

Target Hook int TARGET_DEFAULT_TARGET_FLAGS [Variable]

该变量指定了target_flags的初始值。其缺省设置为0。

bool TARGET_HANDLE_OPTION (size_t code, const char *arg, int value) [Target Hook]

该宏当用户指定了在`.opt'定义文件（参见Chapter 7 [选项], page 67）中描述的target特定选项 时被调用。其可以做一些选项特定的处理，并且如果选项有效时应该返回真。 缺省的定义不做任何事情，只是返回真。

code指定了与选项相关的OPT_name枚举值； 这里的name只是对选项名的重写，将非字母和数字的字符替换为下划线。 arg指定了字符串参数，如果没有参数则为空。 如果选项被标记为UInteger（参见Section 7.2 [选项属性], page 68）， 则value为参数的数值。否则value为1，如果使用了正的选项， 为0如果使用了“no-”形式。

bool TARGET_HANDLE_C_OPTION (size_t code, const char *arg, int value) [Target Hook]

宏当用户指定了在`.opt'定义文件（参见“选项”）中描述的target特定的C语言家族 的选项时被调用。其可以做一些选项特定的处理，并且如果选项有效时应该返回真。 缺省的定义不做任何事情，只是返回假。

通常，你应该使用TARGET_HANDLE_OPTION来处理选项。然而， 如果选项处理只需要在C（和相关语言）前端中有效时， 你再使用TARGET_HANDLE_C_OPTION。

TARGET_VERSION [Macro]

该宏为C语句，用来在stderr上打印表示特定机器描述选择的字符串。 每个机器描述都应该定义TARGET_VERSION。例如：

```
#ifdef MOTOROLA
#define TARGET_VERSION \
 fprintf (stderr, " (68k, Motorola syntax)");
#else
#define TARGET_VERSION \
 fprintf (stderr, " (68k, MIT syntax)");
#endif
```

OVERRIDE_OPTIONS [Macro]

有时特定的命令选项组合在特定的target机器上没有意义。 你可以定义该宏OVERRIDE_OPTIONS。 如果定义，则在所有的命令选项被解析后执行。

不要使用该宏来打开`-O'额外的优化。 使用OPTIMIZATION_OPTIONS来做这件事情。

C_COMMON_OVERRIDE_OPTIONS [Macro]

类似于OVERRIDE_OPTIONS，但只用于C语言的前端（C, Objective-C, C++, Objective-C++），所以可以用于修改只存在于那些前端中的选项标志变量。

OPTIMIZATION_OPTIONS (level, size) [Macro]

一些机器可能需要改变不同优化级别所做的优化。该宏，如果定义， 将在优化级别被确定后，剩余命令选项被解析前执行。 该宏中设定的值将被作为其它命令行选项的缺省值。

level为指定的优化级别；如果指定`-O2'则为2， 如果指定`-O'则为1，如果都没指定则为0。

如果指定`-Os'则size为非零，否则为0。

不要使用该宏来改变不是机器特定的选项。 这些应该在所有支持的机器上的相同优化级别上被统一选择。 使用该宏来打开机器特定的优化。

不要在该宏中检查write_symbols！

bool TARGET_HELP (void) [Target Hook]

该钩子当用户在命令行中执行`--target-help'时被调用。 使得target可以显示在其`.opt'文件中找到的机器特定的命令行选项信息。

CAN_DEBUG_WITHOUT_FP [Macro]

定义该宏，如果没有帧指针也可以进行调试。如果定义了该宏， 则只要指定`-O'，GCC便打开`-fomit-frame-pointer'选项。

17.4 为基于每个函数的信息定义数据结构

如果target需要存储基于每个函数的信息，则GCC为此提供了一个宏和一些变量。 注意，只是使用静态变量来保存信息是一个糟糕的想法，因为GCC支持嵌套函数， 所以可能会在编码一个函数的中途遇到另一个。

GCC定义了称为`struct function`的数据结构体， 包含了特定于单个函数的所有数据。该结构体包含一个称为`machine`的域， 其类型为`struct machine_function *`， 可以被target用于指向它们自己的特定数据。

如果一个target需要基于每个函数的特定数据， 则应该定义类型`struct machine_function`， 以及宏`INIT_EXPANDERS`。 该宏将被用于初始化函数指针`init_machine_status`。 该指针将在下面说明。

一个典型的基于每个函数的target特定数据， 是用于创建一个RTX来存放含有函数返回地址的寄存器。 该RTX随后可以被用于实现`__builtin_return_address`函数。

注意，早期的GCC实现使用了单个数据区域来存放所有的基于每个函数的信息。 因此当开始处理嵌套函数时，旧式的基于每个函数的数据不得不被压入栈中， 并且当处理完成，还要出栈。 GCC曾经提供名为`save_machine_status`和`restore_machine_status` 函数指针来处理target特定信息的保存和恢复。 由于单数据区域的方法不再被使用了，这些指针也不再被支持。

`INIT_EXPANDERS` [Macro]

被调用的宏，用来初始化任何target特定信息。 该宏在任何RTL生成开始之前，基于每个函数被调用一次。 该宏的目的是允许函数指针`init_machine_status`的初始化。

`void (*) (struct function *) init_machine_status` [Variable]

如果该函数指针非空，则会在函数编译开始之前，基于每个函数被调用一次， 用于允许target来执行对`struct function`结构体的任何target特定初始化。 它将被用于初始化那个结构体的`machine`域。

结构体`struct machine_function`将期望被GC来释放。通常， 它们所引用的任何内存都必须使用`ggc_alloc`来分配，包括结构体本身。

17.5 存储布局

注意该表格中的宏定义中，对于以bit为单位的大小或对齐，不需要为常量。 它们可以为引用了静态变量的C表达式，例如`target_flags`。 参见Section 17.3 [运行时目标机], page 289。

`BITS_BIG_ENDIAN` [Macro]

定义该宏的值为1，如果字节中的最高有效位具有最低编号；否则定义其值为0。 这意味着bit-field指令从最高有效位计数。如果机器没有bit-field指令， 则该宏也需要被定义，但定义什么值都无所谓。该宏不需要为一个常量。

该宏不影响结构体域被打包成字节或者字的方式； 那是由`BYTES_BIG_ENDIAN`来控制的。

`BYTES_BIG_ENDIAN` [Macro]

定义该宏的值为1，如果字中的最高有效字节具有最低编号。该宏不需要为一个常量。

`WORDS_BIG_ENDIAN` [Macro]

定义该宏的值为1，如果在多字（multiword）对象中，最高有效字具有最低编号。 这同时应用于内存位置和寄存器中； GCC从根本上假设在内存中的字的顺序与在寄存器中的一样。该宏不需要为一个常量。

`LIBGCC2_WORDS_BIG_ENDIAN` [Macro]

定义该宏如果`WORDS_BIG_ENDIAN`不是常量。该宏必须为一个常量值， 其与`WORDS_BIG_ENDIAN`的具有相同的含义， 并只用于编译`libgcc2.c'的时候。通常该值会根据预处理器定义来设置。

FLOAT_WORDS_BIG_ENDIAN [Macro]

定义该宏值为1，如果DFmode, XFmode或TFmode浮点数被存储在 内存中，并且包含符号位的字位于最低地址；否则值为0。该宏不需要为一个常量。

你不需要定义该宏，如果顺序与多字整数相同。

BITS_PER_UNIT [Macro]

定义该宏为一个可寻址的存储单元（字节）中的位数。如果没有定义，缺省为8。

BITS_PER_WORD [Macro]

字的位数。如果没有定义，缺省为BITS_PER_UNIT * UNITS_PER_WORD。

MAX_BITS_PER_WORD [Macro]

字的最大位数。如果没有定义，缺省为BITS_PER_WORD。 否则其为一个常量，为BITS_PER_WORD在运行时可以具有的最大值。

UNITS_PER_WORD [Macro]

字中的存储单元数；通常为通用目的寄存器的大小，2的1到8次幂。

MIN_UNITS_PER_WORD [Macro]

字中的最小存储单元数。如果没有定义，缺省为UNITS_PER_WORD。否则， 其为一个常量，为UNITS_PER_WORD在运行时可以具有的最小值。

UNITS_PER_SIMD_WORD (mode) [Macro]

向量化可以产生的向量的单元数。缺省等于UNITS_PER_WORD， 因为向量化可以在即使没有专门的SIMD硬件的情况下做一些转换。

POINTER_SIZE [Macro]

指针的宽度，位数。必须指定不比Pmode宽的值。 如果其不等于Pmode的宽度，则必须定义POINTERS_EXTEND_UNSIGNED。 如果没有指定一个值，则缺省为BITS_PER_WORD。

POINTERS_EXTEND_UNSIGNED [Macro]

一个C表达式，用来确定指针应该如何从ptr_mode扩展为Pmode或者 word_mode。如果指针应该被零扩展，则其比0大，如果应该被符号扩展则为0， 如果需要其它转换方式则为负。对于最后一种情况， 扩展通过target的ptr_extend指令来完成。

你不需要定义该宏，如果ptr_mode, Pmode和word_mode都为相同的宽度。

PROMOTE_MODE (m, unsignedp, type) [Macro]

用来更新m和unsignedp，当一个类型为type并且具有特定的机器模式 的对象要被存储到寄存器中时。该宏只在type为一个标量类型时才被调用。

在大多数RISC机器上，只有作用于在整个寄存器上的运算， 定义该宏将m设为为word_mode， 如果m为一个比BITS_PER_WORD窄的整数模式。在大多数情况下， 只有整数模式应该被加宽，因为宽精度的浮点运算通常比相应的窄精度的运算代价要更高。

大多数机器，宏定义不改变unsignedp。然而， 一些机器具有优先处理特定模式的有符号或者无符号的指令。例如，在DEC Alpha上， 32位load和32位add指令会将结果有符号扩展为64位。在这样的机器上， 根据扩展的类型来设置unsignedp会更加有效。

如果从来不会修改m，则不要定义该宏。

PROMOTE_FUNCTION_MODE [Macro]

类似于PROMOTE_MODE，但应用于输出的函数参数，或者函数返回值，分别由TARGET_PROMOTE_FUNCTION_ARGS和TARGET_PROMOTE_FUNCTION_RETURN指定。

缺省为PROMOTE_MODE。

bool TARGET_PROMOTE_FUNCTION_ARGS (tree fntype) [Target Hook]

该target钩子应该返回true，如果由PROMOTE_FUNCTION_MODE描述的提升应该应用于输出的函数参数。

bool TARGET_PROMOTE_FUNCTION_RETURN (tree fntype) [Target Hook]

该target钩子应该返回true，如果由PROMOTE_FUNCTION_MODE描述的提升应该应用于函数返回值。

如果该target钩子返回true，则TARGET_FUNCTION_VALUE必须执行与PROMOTE_FUNCTION_MODE相同的提升。

PARM_BOUNDARY [Macro]

函数参数在栈上的对齐方式，位数。所有栈参数都接受这样的对齐，而不论数据类型是什么。在大多数机器上，这与整数的大小相同。

STACK_BOUNDARY [Macro]

定义该宏为硬件要求的在该机器上的栈指针的最小对齐。定义为一个C表达式，为所要的对齐（位数）。该值作为缺省值使用，如果没有定义PREFERRED_STACK_BOUNDARY。在大多数机器上，这应该与PARM_BOUNDARY相同。

PREFERRED_STACK_BOUNDARY [Macro]

定义该宏，如果你希望对栈指针维持一个特定的对齐，大于硬件要求的对齐。定义为一个C表达式，为所要的对齐（位数）。该宏必须等于或大于STACK_BOUNDARY。

INCOMING_STACK_BOUNDARY [Macro]

Define this macro if the incoming stack boundary may be different from PREFERRED_STACK_BOUNDARY. This macro must evaluate to a value equal to or larger than STACK_BOUNDARY.

FUNCTION_BOUNDARY [Macro]

函数入口点所需的对齐位数。

BIGGEST_ALIGNMENT [Macro]

该机器上任何数据类型可以需要的最大对齐位数。注意这不是所支持的最大对齐，而是如果违反该对齐则可能会造成错误。

MALLOC_ABI_ALIGNMENT [Macro]

Alignment, in bits, a C conformant malloc implementation has to provide. If not defined, the default value is BITS_PER_WORD.

ATTRIBUTE_ALIGNED_VALUE [Macro]

Alignment used by the __attribute__ ((aligned)) construct. If not defined, the default value is BIGGEST_ALIGNMENT.

MINIMUM_ATOMIC_ALIGNMENT [Macro]

如果被定义，则为最小对齐位数，可以分给一个对象并且在一个操作中被应用，而不需要干扰任何附近的对象。通常为BITS_PER_UNIT，但在没有字节或半字的存储运算的机器上可以更大些。

BIGGEST_FIELD_ALIGNMENT [Macro]

任何结构体或者联合体域在该机器上需要的最大对齐。如果被定义，这将只覆盖结构体和联合体的域的BIGGEST_ALIGNMENT，除非域对齐已经通过__attribute__((aligned(n)))设置。

ADJUST_FIELD_ALIGN (field, computed) [Macro]

一个表达式，为结构体域field的对齐方式，如果对齐方式是按照通常方式计算（包括应用BIGGEST_ALIGNMENT和BIGGEST_FIELD_ALIGNMENT）。其只覆盖没有通过__attribute__((aligned(n)))设置的域。

MAX_STACK_ALIGNMENT [Macro]

Biggest stack alignment guaranteed by the backend. Use this macro to specify the maximum alignment of a variable on stack.

If not defined, the default value is STACK_BOUNDARY.

MAX_OFILE_ALIGNMENT [Macro]

由该机器的目标文件格式所支持的最大对齐。使用该宏来限制可以使用__attribute__((aligned(n)))结构来指定的对齐。如果没有定义，则缺省值为BIGGEST_ALIGNMENT。

在使用ELF的系统上，缺省（在`config/elfos.h'中）为在32位host上可以表示的32位ELF section对齐，即`(((unsigned HOST_WIDEST_INT) 1 << 28) * 8)'。在32位ELF上，最大支持的section对齐位数是`(0x80000000 * 8)'，但这在32位host上无法表示。

DATA_ALIGNMENT (type, basic-align) [Macro]

如果定义，则为一个C表达式，来计算在静态存储中的变量的对齐。type为数据类型，basic-align为对象通常具有的对齐。该宏的值被用于替代那个对齐，并应用的对象上。

如果该宏没有定义，则使用basic-align。

该宏的一种用法是增加中等大小数据的对齐，使得可以放在最少的cache line中。另一种用法是使得字符数组按照字对齐，这样strcpy调用可以通过内联方式完成。

CONSTANT_ALIGNMENT (constant, basic-align) [Macro]

如果定义，为一个C表达式，来计算放在内存中的常量的对齐。constant为常量，basic-align为该对象通常具有的对齐。该宏的值被用于替代那个对齐，并应用的对象上。

如果该宏没有定义，则使用basic-align。

该宏的典型用法为增加字符串常量的对齐，使其为字对齐，这样strcpy调用可以通过内联方式完成。

LOCAL_ALIGNMENT (type, basic-align) [Macro]

如果定义，为一个C表达式，来计算在局部存储中的对象的对齐。type为数据类型，basic-align为对象通常的对齐。该宏的值被用于替代那个对齐，并应用的对象上。

如果该宏没有定义，则使用basic-align。

该宏的一种用法是增加中等大小数据的对齐，使得可以放在最少的cache line中。

STACK_SLOT_ALIGNMENT (type, mode, basic-align) [Macro]

If defined, a C expression to compute the alignment for stack slot. type is the data type, mode is the widest mode available, and basic-align is the alignment that the slot would ordinarily have. The value of this macro is used instead of that alignment to align the slot.

If this macro is not defined, then basic-align is used when type is NULL. Otherwise, LOCAL_ALIGNMENT will be used.

This macro is to set alignment of stack slot to the maximum alignment of all possible modes which the slot may have.

LOCAL_DECL_ALIGNMENT (decl) [Macro]

If defined, a C expression to compute the alignment for a local variable decl.

If this macro is not defined, then LOCAL_ALIGNMENT (TREE_TYPE (decl), DECL_ALIGN (decl)) is used.

One use of this macro is to increase alignment of medium-size data to make it all fit in fewer cache lines.

EMPTY_FIELD_BOUNDARY [Macro]

允许像int：0;；这样的空域的结构体位域的对齐位数。

如果PCC_BITFIELD_TYPE_MATTERS为真，则其覆盖该宏。

STRUCTURE_SIZE_BOUNDARY [Macro]

任何结构体或联合体的大小必须为该位数的倍数。 每个结构体或联合体的大小都将被舍入到该位数的一个倍数。

如果没有定义该宏，则缺省与BITS_PER_UNIT相同。

STRICT_ALIGNMENT [Macro]

定义该宏值为1，如果给定数据不在通常对齐方式上，则指令无法工作。 如果对于这种情况指令只不过是变慢，则定义该宏为0。

PCC_BITFIELD_TYPE_MATTERS [Macro]

定义该宏，如果你希望仿效许多其它C编译器处理位域和包含它们的结构体的对齐方式。

该行为是书写为命名位域（int,short或其它整数类型）的类型被实施 用于整个结构体的对齐，就好像结构体包含了一个该类型的普通的域。另外， 位域放在结构体中，使得其将适合这样的域，而不会跨越边界。

这样，大多数机器上，书写为int的命名位域将不会跨越一个四字节的边界， 并将使得整个结构体为四字节对齐。（可能不使用四字节对齐；其由其它对齐参数控制。）

一个没有命名的位域将不会影响包含结构体的对齐。

如果定义了该宏，则其定义为一个C表达式；该表达式的非0值会使用这样方式。

注意如果该宏没有定义，或者其值为0，则一些位域可能跨越多于一个的对齐边界。 编译器可以支持这种引用，如果有`insv', `extv'和`extzv' insns可以直接引用内存。

其它已知的可以使位域工作的的方式为定义STRUCTURE_SIZE_BOUNDARY和BIGGEST_ALIGNMENT一样大。

除非机器具有位域指令或者你按照那种方式定义了STRUCTURE_SIZE_BOUNDARY， 否则你必须定义PCC_BITFIELD_TYPE_MATTERS具有非0值。

如果你的目标是使得GCC使用与其它编译器相同的约定来布局位域， 则这里有一种方式可以调查其它编译器是如何做的。编译运行该程序：

```
struct foo1
{
 char x;
 char :0;
 char y;
};
```

```
struct foo2
{
 char x;
 int :0;
 char y;
};

main ()
{
 printf ("Size of foo1 is %d\n",
      sizeof (struct foo1));
 printf ("Size of foo2 is %d\n",
      sizeof (struct foo2));
 exit (0);
}
```

如果其打印2和5，则编译器的行为就是你通过PCC_BITFIELD_TYPE_MATTERS获得的效果。

BITFIELD_NBYTES_LIMITED [Macro]

跟PCC_BITFIELD_TYPE_MATTERS相似，除了它只影响结构体中的位域的对齐。

bool TARGET_ALIGN_ANON_BITFIELD (void) [Target Hook]

当PCC_BITFIELD_TYPE_MATTERS为真， 该钩子将确定未命名位域是否要影响包含它的结构体的对齐。 钩子应该返回真，如果结构体应该继承未命名位域的类型所要求的对齐。

bool TARGET_NARROW_VOLATILE_BITFIELD (void) [Target Hook]

该target钩子应该返回true，如果访问volatile位域应该尽可能使用最窄的机器模式。其应该返回false，如果这些访问应该使用位域的容器的类型。

缺省为!TARGET_STRICT_ALIGN。

MEMBER_TYPE_FORCES_BLK (field, mode) [Macro]

返回1，如果一个包含field的结构体或者数组应该使用BLKMODE模式来访问。

如果field为结构体中唯一的域，则mode为它的机器模式， 否则mode为VOIDmode。

通常，不需要该宏。

ROUND_TYPE_ALIGN (type, computed, specified) [Macro]

定义该宏为一个表达式，为一个类型（由作为树节点的type给定）的对齐， 如果按照通常方式计算的对齐方式为computed并且显示指定的对齐方式为specified。

缺省是使用specified，如果其更大； 否则使用computed和BIGGEST_ALIGNMENT中较小的。

MAX_FIXED_MODE_SIZE [Macro]

一个整数表达式，为实际应该被使用的最大的整数机器模式的位数。 所有该大小或者更小一些的整数机器模式都可以用于结构体和联合体。 如果哦没有定义该宏，则假设为GET_MODE_BITSIZE (DImode)。

STACK_SAVEAREA_MODE (save_level) [Macro]

如果定义，则为一个enum machine_mode类型的表达式， 指定名为save_stack_level的指令模式（参见Section 16.9 [标准名字], page 235）的save区域操作数的机器模式。 save_level为SAVE_BLOCK, SAVE_FUNCTION或SAVE_NONLOCAL中之一。

你不需要定义该宏，如果其总是返回Pmode。你通常将会定义该宏， 如果save_stack_level指令模式需要同时支持32和64位机器模式。

STACK_SIZE_MODE [Macro]

如果定义，为一个enum machine_mode类型的表达式， 指定名为allocate_stack的指令模式（参见Section 16.9 [标准名字], page 235）的size increment操作数的机器模式。

你不需要定义该宏，如果其总是返回word_mode。你通常将会定义该宏，如果allocate_stack指令模式需要同时支持32和64位机器模式。

enum machine_mode TARGET_LIBGCC_CMP_RETURN_MODE () [Target Hook]

该target钩子应该返回扩展为libgcc调用的比较指令的返回值的机器模式。 如果没有定义，则返回word_mode，其对于大多数target是正确的。

enum machine_mode TARGET_LIBGCC_SHIFT_COUNT_MODE () [Target Hook]

该target钩子应该返回扩展为libgcc调用的移位指令的shift count操作数的机器模式。 如果没有定义，则返回word_mode，其对于大多数target是正确的。

ROUND_TOWARDS_ZERO [Macro]

如果定义，该宏应该为真，如果舍入的模式是朝向0。

定义该宏只影响`libgcc.a'模拟浮点算术的方式。

不定义该宏，等价于其返回0。

LARGEST_EXPONENT_IS_NORMAL (size) [Macro]

该宏应该返回真，如果具有size位数的浮点不具有NaN或无穷的表示， 但是使用最大的普通数的指数来替代表示。

定义该宏只影响`libgcc.a'模拟浮点算术的方式。

该宏缺省定义对所有的size都返回假。

bool TARGET_VECTOR_OPAQUE_P (tree type) [Target Hook]

This target hook should return true a vector is opaque. That is, if no cast is needed when copying a vector value of type type into another vector lvalue of the same size. Vector opaque types cannot be initialized. The default is that there are no such types.

bool TARGET_MS_BITFIELD_LAYOUT_P (tree record_type) [Target Hook]

This target hook returns true if bit-fields in the given record_type are to be laid out following the rules of Microsoft Visual C/C++, namely: (i) a bit-field won't share the same storage unit with the previous bit-field if their underlying types have different sizes, and the bit-field will be aligned to the highest alignment of the underlying types of itself and of the previous bit-field; (ii) a zero-sized bit-field will affect the alignment of the whole enclosing structure, even if it is unnamed; except that (iii) a zero-sized bit-field will be disregarded unless it follows another bit-field of nonzero size. If this hook returns true, other macros that control bit-field layout are ignored.

When a bit-field is inserted into a packed record, the whole size of the underlying type is used by one or more same-size adjacent bit-fields (that is, if its long:3, 32 bits is used in the record, and any additional adjacent long bit-fields are packed into the same chunk of 32 bits. However, if the size changes, a new field of that size is allocated). In an unpacked record, this is the same as using alignment, but not equivalent when packing.

If both MS bit-fields and `__attribute__((packed))' are used, the latter will take precedence. If `__attribute__((packed))' is used on a single field when MS bit-fields are in use, it will take precedence for that field, but the alignment of the rest of the structure may affect its placement.

bool TARGET_DECIMAL_FLOAT_SUPPORTED_P (void) [Target Hook]
返回真，如果target支持十进制浮点。

bool TARGET_FIXED_POINT_SUPPORTED_P (void) [Target Hook]
返回真，如果target支持定点算术。

void TARGET_EXPAND_TO_RTL_HOOK (void) [Target Hook]
该钩子在扩展为rtl之前被调用，允许target在扩展前执行额外的实例化或者分析。 例如，rs6000port使用它来分配scratch栈槽，当被扩展的函数具有任何SDmode使用时， 用于在内存和浮点寄存器之间复制SDmode值。

void TARGET_INSTANTIATE_DECLS (void) [Target Hook]
该钩子允许后端执行额外的rtl实例化，这些实际上不存在于任何insn中，但在之后会有。

const char * TARGET_MANGLE_TYPE (tree type) [Target Hook]
If your target defines any fundamental types, or any types your target uses should be mangled differently from the default, define this hook to return the appropriate encoding for these types as part of a C++ mangled name. The type argument is the tree structure representing the type to be mangled. The hook may be applied to trees which are not target-specific fundamental types; it should return NULL for all such types, as well as arguments it does not recognize. If the return value is not NULL, it must point to a statically-allocated string constant.

Target-specific fundamental types might be new fundamental types or qualified versions of ordinary fundamental types. Encode new fundamental types as `u n name', where name is the name used for the type in source code, and n is the length of name in decimal. Encode qualified versions of ordinary types as `U n name code', where name is the name used for the type qualifier in source code, n is the length of name as above, and code is the code used to represent the unqualified version of this type. (See write_builtin_type in `cp/mangle.c' for the list of codes.) In both cases the spaces are for clarity; do not include any spaces in your string.

This hook is applied to types prior to typedef resolution. If the mangled name for a particular type depends only on that type's main variant, you can perform typedef resolution yourself using TYPE_MAIN_VARIANT before mangling.

The default version of this hook always returns NULL, which is appropriate for a target that does not define any new fundamental types.

17.6 源语言的数据类型布局

这些宏定义了在被编译的程序中使用的标准基础数据类型的大小和其它特征。 不像之前章节中的宏，这些是应用到C和相关语言的特定特征上， 而不是存储布局的基础方面。

INT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型int的位大小。 如果没有定义，缺省为一个字。

SHORT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型short的位大小。如果没有定义， 缺省为半个字。（如果比一个存储单元小，则会向上舍入为一个单元。）

LONG_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long的位大小。 如果没有定义，缺省为一个字。

ADA_LONG_TYPE_SIZE [Macro]
在一些机器上，本地Ada编译器使用的类型long的大小与C使用的不相同。 这种情况下，定义该宏为一个C表达式用于那个类型的大小。如果没有定义， 则缺省为LONG_TYPE_SIZE的值。

LONG_LONG_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long long的位大小。如果没有定义， 缺省为两个字。如果你想在你的机器上支持GNU Ada，则该宏的值最少必须为64。

CHAR_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型char的位大小。 如果没有定义，缺省为BITS_PER_UNIT。

BOOL_TYPE_SIZE [Macro]
一个C表达式，为在target机器上C++类型bool和C99类型_Bool的位大小。 如果没有定义，并且通常不会定义，缺省为CHAR_TYPE_SIZE。

FLOAT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型float的位大小。 如果没有定义，缺省为一个字。

DOUBLE_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型double的位大小。 如果没有定义，缺省为两个字。

LONG_DOUBLE_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long double的位大小。 如果没有定义，缺省为两个字。

SHORT_FRACT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型short _Fract的位大小。 如果没有定义，缺省为BITS_PER_UNIT。

FRACT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型_Fract的位大小。如果没有定义， 缺省为BITS_PER_UNIT＊2。

LONG_FRACT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long _Fract的位大小。 如果没有定义，缺省为BITS_PER_UNIT＊4。

LONG_LONG_FRACT_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long long _Fract的位大小。 如果没有定义，缺省为BITS_PER_UNIT＊8。

SHORT_ACCUM_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型short _Accum的位大小。 如果没有定义，缺省为BITS_PER_UNIT＊2。

ACCUM_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型_Accum的位大小。 如果没有定义，缺省为BITS_PER_UNIT＊4。

LONG_ACCUM_TYPE_SIZE [Macro]
一个C表达式，为在target机器上类型long _Accum的位大小。 如果没有定义，缺省为BITS_PER_UNIT＊8。

LONG_LONG_ACCUM_TYPE_SIZE [Macro]

一个C表达式，为在target机器上类型long long _Accum的位大小。 如果没有定义，缺省为BITS_PER_UNIT * 16。

LIBGCC2_LONG_DOUBLE_TYPE_SIZE [Macro]

定义该宏，如果LONG_DOUBLE_TYPE_SIZE不是常量或者如果你想让`libgcc2.a' 中具有大小不是LONG_DOUBLE_TYPE_SIZE的程序。如果没有定义， 缺省为LONG_DOUBLE_TYPE_SIZE。

LIBGCC2_HAS_DF_MODE [Macro]

定义该宏，如果LIBGCC2_DOUBLE_TYPE_SIZE和 LIBGCC2_LONG_DOUBLE_TYPE_SIZE都不是DFmode， 但是你还想让`libgcc2.a'中具有DFmode的程序。如果没有定义， 并且LIBGCC2_DOUBLE_TYPE_SIZE或LIBGCC2_LONG_DOUBLE_TYPE_SIZE为64， 则缺省为1，否则为0。

LIBGCC2_HAS_XF_MODE [Macro]

定义该宏，如果LIBGCC2_LONG_DOUBLE_TYPE_SIZE不是XFmode， 但是你还想让`libgcc2.a'中具有XFmode的程序。如果没有定义， 并且LIBGCC2_LONG_DOUBLE_TYPE_SIZE为80，则缺省为1，否则为0。

LIBGCC2_HAS_TF_MODE [Macro]

定义该宏，如果LIBGCC2_LONG_DOUBLE_TYPE_SIZE不是TFmode， 但是你还想让`libgcc2.a'中具有TFmode的程序。如果没有定义， 并且LIBGCC2_LONG_DOUBLE_TYPE_SIZE为128，则缺省为1，否则为0。

SF_SIZE [Macro]
DF_SIZE [Macro]
XF_SIZE [Macro]
TF_SIZE [Macro]

定义这些宏为SFmode, DFmode, XFmode和TFmode值的尾数 位大小，如果在`libgcc2.h'中的缺省定义不合适。缺省的， FLT_MANT_DIG用于SF_SIZE, LDBL_MANT_DIG用于XF_SIZE和 TF_SIZE，并且 DBL_MANT_DIG或 LDBL_MANT_DIG用于 DF_SIZE， 根据 LIBGCC2_DOUBLE_TYPE_SIZE或LIBGCC2_LONG_DOUBLE_TYPE_SIZE是否为64。

TARGET_FLT_EVAL_METHOD [Macro]

一个C表达式，为`float.h'中的FLT_EVAL_METHOD的值。 如果没有定义，则FLT_EVAL_METHOD的值将为0。

WIDEST_HARDWARE_FP_SIZE [Macro]

一个C表达式，为硬件支持的最宽浮点格式的位数。如果定义该宏， 则必须指定一个小于或等于LONG_DOUBLE_TYPE_SIZE的值。如果没有定义， 则缺省为LONG_DOUBLE_TYPE_SIZE的值。

DEFAULT_SIGNED_CHAR [Macro]

一个表达式，其值为1或者0，根据类型char缺省应该为有符号的还是无符号的。 用户总是可以使用选项`-fsigned-char'和`-funsigned-char'来覆盖该缺省定义。

bool TARGET_DEFAULT_SHORT_ENUMS (void) [Target Hook]

该target钩子应该返回真，如果编译器应该为enum类型设置为可以表示该类型值 范围的字节数。其应该返回假，如果所有的enum类型应该按照int类型的 方式来分配。

缺省为返回假。

SIZE_TYPE [Macro]

一个C表达式，为一个字符串描述了用于size值的数据类型名。 typedef名size_t使用该字符串的内容来定义。

字符串可以包含多一个的关键字。如果是这样，则将它们使用空格分开， 首先是任意长度的关键字，然后是合适unsigned 的，最后是int。 字符串必须显示的匹配文件`c-decl.c'中函数init_decl_processing中 定义的数据类型名。不可以省略掉int或者改变顺序， 这将会使编译器在启动时崩溃。

如果没有定义，缺省为"long unsigned int"。

PTRDIFF_TYPE [Macro]

一个C表达式，为一个字符串，描述了用于两个指针相减的结果的数据类型名。 typedef名ptrdiff_t使用该字符串的内容来定义。更多信息， 参见上面的SIZE_TYPE。

如果没有定义，则缺省为"long int"。

WCHAR_TYPE [Macro]

一个C表达式，为一个字符串，描述了用于宽字符的数据类型名。 typedef名wchar_t使用该字符串的内容来定义。更多信息， 参见上面的SIZE_TYPE。

如果没有定义，则缺省为"int"。

WCHAR_TYPE_SIZE [Macro]

一个C表达式，为宽字符数据类型的位数。 这用于不能使用WCHAR_TYPE的cpp中。

WINT_TYPE [Macro]

一个C表达式，为一个字符串， 描述了传递给printf并且从getwc中返回的宽字符数据类型名。 typedef名wint_t使用该字符串的内容来定义。更多信息， 参见上面的SIZE_TYPE。

如果没有定义，则缺省为"unsigned int"。

INTMAX_TYPE [Macro]

一个C表达式，为一个字符串， 描述了可以表示任何标准或者扩展的有符号整数类型值的数据类型名。 typedef名intmax_t使用该字符串的内容来定义。更多信息， 参见上面的SIZE_TYPE。

如果没有定义，则缺省为"int","long int"或"long long int"中第一个与long long int具有相同精度的字符串。

UINTMAX_TYPE [Macro]

一个C表达式，为一个字符串， 描述了可以表示任何标准或者扩展的无符号整数类型值的数据类型名。 typedef名uintmax_t使用该字符串的内容来定义。更多信息， 参见上面的SIZE_TYPE。

如果没有定义，则缺省为"unsigned int","long unsigned int"或 "long long unsigned int"中第一个与long long unsigned int 具有相同精度的字符串。

TARGET_PTRMEMFUNC_VBIT_LOCATION [Macro]

The C++ compiler represents a pointer-to-member-function with a struct that looks like:

```
struct {
 union {
  void (*fn)();
  ptrdiff_t vtable_index;
```

```
  };
  ptrdiff_t delta;
};
```

The C++ compiler must use one bit to indicate whether the function that will be called through a pointer-to-member-function is virtual. Normally, we assume that the low-order bit of a function pointer must always be zero. Then, by ensuring that the vtable_index is odd, we can distinguish which variant of the union is in use. But, on some platforms function pointers can be odd, and so this doesn't work. In that case, we use the low-order bit of the `delta` field, and shift the remainder of the `delta` field to the left.

GCC will automatically make the right selection about where to store this bit using the `FUNCTION_BOUNDARY` setting for your platform. However, some platforms such as ARM/Thumb have `FUNCTION_BOUNDARY` set such that functions always start at even addresses, but the lowest bit of pointers to functions indicate whether the function at that address is in ARM or Thumb mode. If this is the case of your architecture, you should define this macro to `ptrmemfunc_vbit_in_delta`.

In general, you should not have to define this macro. On architectures in which function addresses are always even, according to `FUNCTION_BOUNDARY`, GCC will automatically define this macro to `ptrmemfunc_vbit_in_pfn`.

`TARGET_VTABLE_USES_DESCRIPTORS` [Macro]

Normally, the C++ compiler uses function pointers in vtables. This macro allows the target to change to use ``function descriptors'' instead. Function descriptors are found on targets for whom a function pointer is actually a small data structure. Normally the data structure consists of the actual code address plus a data pointer to which the function's data is relative.

If vtables are used, the value of this macro should be the number of words that the function descriptor occupies.

`TARGET_VTABLE_ENTRY_ALIGN` [Macro]

By default, the vtable entries are void pointers, the so the alignment is the same as pointer alignment. The value of this macro specifies the alignment of the vtable entry in bits. It should be defined only when special alignment is necessary. */

`TARGET_VTABLE_DATA_ENTRY_DISTANCE` [Macro]

There are a few non-descriptor entries in the vtable at offsets below zero. If these entries must be padded (say, to preserve the alignment specified by `TARGET_VTABLE_ENTRY_ALIGN`), set this to the number of words in each data entry.

17.7 寄存器的用法

这一节说明了如何描述目标机器具有什么寄存器，以及它们（通常）可以被如何使用。

寄存器类别用来描述一个特定指令可以使用哪些寄存器；参见 Section 17.8 [寄存器类别], page 307。关于使用寄存器来访问栈帧的信息，参见 Section 17.10.4 [帧寄存器], page 321。关于使用寄存器来传值，参见 Section 17.10.7 [寄存器参数], page 325。关于使用寄存器来返回值，参见 Section 17.10.8 [标量返回], page 329。

17.7.1 寄存器的基本特征

寄存器具有不同的特征。

FIRST_PSEUDO_REGISTER [Macro]

编译器知道的硬件寄存器个数。它们包括编号0到FIRST_PSEUDO_REGISTER-1； 因此，第一个伪寄存器的编号实际被赋值为FIRST_PSEUDO_REGISTER。

FIXED_REGISTERS [Macro]

一个初始化，说明哪些寄存器在整个编译代码中都用于固定用途， 因此不能用于通用分配。这些将包括栈指针，帧指针（在当不需要帧指针时， 可以用于通用寄存器的机器上除外），在一些机器上被认为是可寻址的寄存器的程序计数器， 以及其它具有标准用法的编号寄存器。

这些信息作为一个序列编号来表示，由逗号分隔并由大括号包括。 第n个编号为1，如果寄存器n为固定的，否则为0。

从该宏初始化的表，以及由下面的初始化的表，都可以在运行时被覆盖， 或者通过执行宏CONDITIONAL_REGISTER_USAGE自动完成， 或者通过用户使用命令选项`-ffixed-reg', `-fcall-used-reg'和`-fcall-saved-reg'。

CALL_USED_REGISTERS [Macro]

类似FIXED_REGISTERS，但是对于被函数调用破坏的每个寄存器， 以及固定寄存器，值为1。因此， 该宏标识了哪些寄存器不适合用于必须活跃于整个函数调用的值的通用分配。

如果寄存器在CALL_USED_REGISTERS中具有值0， 如果寄存器在函数中被使用，则编译器会自动的在函数入口保存它， 并在函数出口恢复它。

CALL_REALLY_USED_REGISTERS [Macro]

类似CALL_USED_REGISTERS，除了该宏不需要包含整个FIXED_REGISTERS集。 （CALL_USED_REGISTERS必须为FIXED_REGISTERS的超集）。 该宏为可选的。如果没有被指定，其缺省为CALL_USED_REGISTERS的值。

HARD_REGNO_CALL_PART_CLOBBERED (regno, mode) [Macro]

一个C表达式，值为非0， 如果不允许将机器模式为mode的值存储在编号为regno的硬件寄存器中， 并且整个调用中没有破坏其某个部分。对于大多数机器，该宏不需要被定义。 其只用于一些在调用中不保护寄存器的整个内容的机器上。

CONDITIONAL_REGISTER_USAGE [Macro]

0条或者多条C语句，其可以条件修改5个变量fixed_regs, call_used_rehs, global_regs, reg_names 和 reg_class_contents，来考虑这些寄存器集对target标号的任何依赖。 前3个为char []类型（作为布尔向量来解析）。 global_regs为一个const char *[]，reg_class_contents为一个HARD_REG_SET。 在宏被调用之前，fixed_regs, call_used_regs, reg_class_contents和 reg_names已经分别通过FIXED_REGISTERS, CALL_USED_REGISTERS, REG_CLASS_CONTENTS和REGISTER_NAMES被初始化。 global_regs已经被清除，并且任何`-ffixed-reg', `-fcall-used-reg'和`-fcall-saved-reg'选项已经被应用。

如果不需要做什么工作，则不必定义该宏。

如果整个寄存器的类别的使用，取决于target标记， 则你可以通过使用该宏来指示GCC修改fixed_regs和call_used_regs为1， 对于类别中的每个不应由GCC使用的寄存器。还有就是，定义宏REG_CLASS_FROM_LETTER / REG_CLASS_FROM_CONSTRAINT来返回NO_REGS， 如果其被调用，并带有一个不应该被使用的类别字母。

（然而，如果该类没有包含在GENERAL_REGS中， 并且所有的insn指令模式的约束允许该类通过target开关来控制， 则GCC将自动避免使用这些寄存器，当target开关与它们相反时。）

INCOMING_REGNO (out) [Macro]

定义该宏，如果target机器具有寄存器窗口。 该C表达式根据调用函数能看到的寄存器编号out， 返回被调用函数所能看到的寄存器编号。 返回out，如果寄存器编号out不是发送寄存器（outbound register）。

OUTGOING_REGNO (in) [Macro]

定义该宏，如果target机器具有寄存器窗口。 该C表达式根据被调用函数能看到的寄存器编号in， 返回调用函数所能看到的寄存器编号。 返回in，如果寄存器编号in不是运入寄存器（inbound register）。

LOCAL_REGNO (regno) [Macro]

定义该宏，如果target机器具有寄存器窗口。该C表达式返回真， 如果寄存器为调用保存的，但是在寄存器窗口中。不像大多调用保存的寄存器， 这样的寄存器不需要在函数出口或者非局部调转中被显示的恢复。

PC_REGNUM [Macro]

如果程序计数器具有一个寄存器编号，则定义其为那个寄存器编号。否则不要定义它。

17.7.2 寄存器的分配顺序

寄存器按照顺序进行分配。

REG_ALLOC_ORDER [Macro]

如果定义，则为一个整数向量的初始化值，包含了硬件寄存器号， GCC将按该照顺序来使用它们（前面的优先）。

如果没有定义该宏，则寄存器按照低编号优先的方式使用。

该宏的一个用处是在一些机器上， 高编号的寄存器必须总是被保存并且save-multiple-registers指令 只支持连续序列的寄存器。在这些机器上， 可以定义REG_ALLOC_ORDER来初始化列表， 使得高编号寄存器优先分配。

ORDER_REGS_FOR_LOCAL_ALLOC [Macro]

一条C语句（无分号）， 用来选择按照什么顺序来为局部于一个基本块的伪寄存器分配硬件寄存器。

将想要的寄存器顺序存储在数组reg_alloc_order中。 元素0为先分配的寄存器；元素1为下一个；等等。

在执行宏之前，宏的内容体不应该对reg_alloc_order的内容作任何假设。

在大多数机器上，不需要定义该宏。

IRA_HARD_REGNO_ADD_COST_MULTIPLIER (regno) [Macro]

In some case register allocation order is not enough for the Integrated Register Allocator (IRA) to generate a good code. If this macro is defined, it should return a floating point value based on regno. The cost of using regno for a pseudo will be increased by approximately the pseudo's usage frequency times the value returned by this macro. Not defining this macro is equivalent to having it always return 0.0.

在大多数机器上，不需要定义该宏。

17.7.3 如何使值适合寄存器

这节讨论的宏，描述了每个寄存器可以存放哪类的值（明确的说， 是哪些机器模式的），以及对于给定的机器模式需要多少个连续的寄存器。

HARD_REGNO_NREGS (regno, mode) [Macro]

一个C表达式，为存放模式mode的值所需要的连续的硬件寄存器， 起始于寄存器编号regno。该宏不要返回0， 即使寄存器不能存放指定的mode——替代的， 使用HARD_REGNO_MODE_OK和/或CANNOT_CHANGE_MODE_CLASS。

在所有寄存器都是一个字大小的机器上，该宏的一个合适的定义为

```
#define HARD_REGNO_NREGS(REGNO, MODE)       \
  ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) \
   / UNITS_PER_WORD)
```

HARD_REGNO_NREGS_HAS_PADDING (regno, mode) [Macro]

一个C表达式，为非0，如果模式为mode的值，存储在内存中， 并由padding结尾，这使得其占有更多的空间， 比在起始于寄存器编号regno的寄存器中。缺省的为0。

例如，如果浮点值存储在三个32位寄存器中，但是在内存中占有128位，则该宏应该为非0。

该宏只有当subreg_get_info会错误的确定一个subreg可以通过寄存器编号的偏移量来表示，而实际上这样的subreg将会保存一些不应该被表示的padding时， 才需要被定义。

HARD_REGNO_NREGS_WITH_PADDING (regno, mode) [Macro]

对于HARD_REGNO_NREGS_HAS_PADDING会返回非0的regno和mode的值， 其为一个C表达式，返回保存包括任何padding的值所需要的寄存器的最大数。 在上面的例子中，值将为4。

REGMODE_NATURAL_SIZE (mode) [Macro]

定义该宏，如果存放模式mode的值的寄存器的自然大小，不是word大小。 其为一个C表达式，对于指定的mode给出以字节为单位的自然的大小。 其被寄存器分配用于尝试优化它的结果。 例如这出现在SPARC 64位机器上，其浮点寄存器的自然大小仍然是32位。

HARD_REGNO_MODE_OK (regno, mode) [Macro]

一个C表达式，其为非0， 如果允许将一个mode模式的值存储在硬件寄存器编号regno（或者起始于它的多个寄存器）中。对于所有寄存器都是等价的机器上， 一个合适的定义为

```
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1
```

你不需要包含检查固定寄存器编号的代码， 因为分配机制总是认为它们已经被占用了。

在一些机器上，双精度值必须放在偶/奇寄存器对。 你可以通过定义该宏来拒绝这样模式的奇数寄存器编号。

对于一个模式可以放在寄存器中的最小需求为， `movmode'指令模式支持在寄存器和同一类别的其它硬件寄存器之间的移动， 并且将一个值移动到寄存器中并移动回来，而不会改变。

由于用于move word_mode的同一指令，也可以用于所有更窄的整数模式， 所以HARD_REGNO_MODE_OK不必要在任何机器上对于这些模式都不同， 假定你定义了指令模式`movhi'等。

许多机器对于浮点算术具有特定的寄存器。 通常人们假设浮点机器模式只在浮点寄存器中被允许。这并不真实。 任何可以存放整数的寄存器都可以安全的存放一个浮点机器模式， 而不管是否可以在这些寄存器上进行浮点算术。 整数move指令可以用于移动这些值。

然而在一些机器上，定点机器模式不可以放在浮点寄存器中。 比如如果浮点寄存器对任何存储的值进行标准化， 因为存储一个非浮点值将会使值变得混淆。这种情况下，HARD_REGNO_MODE_OK应该拒绝定点机器模式放在浮点寄存器中。 但是，如果浮点寄存器不自

动标准化， 如果你可以存储任何位的指令模式并无需改动的获得它， 则任何机器模式都可以放在浮点寄存器中，这样你可以定义该宏来表明可以这么做。

当然，特定的浮点寄存器的主要意义是它们在浮点算术指令中可以使用。 但是，这根HARD_REGNO_MODE_OK没有关系。 你可以通过对那些执行写合适的约束来处理。

在一些机器上，浮点寄存器访问起来特别慢，所以如果浮点算术没有完成前， 最好将值存在栈帧中，而不是在这样的寄存器中。 只要浮点寄存器不在GENERAL_REGS类别中，它们将不会被使用， 除非某个指令模式的约束要求这样。

HARD_REGNO_RENAME_OK (from, to) [Macro]

一个C表达式，为非0， 如果可以将一个硬件寄存器from重命名为另一个寄存器to。

该宏的一个通用的用法是防止将一个寄存器重命名为另一个寄存器， 而其在中断处理函数的序言中没有被保存。

缺省总是为非0。

MODES_TIEABLE_P (mode1, mode2) [Macro]

一个C表达式，其为非0，如果一个模式mode1的值， 不需要复制便可以按照模式mode2来访问。

如果HARD_REGNO_MODE_OK (r, mode1)和 HARD_REGNO_MODE_OK (r, mode2)对于任何r总是相同， 则MODES_TIEABLE_P (mode1, mode2)应该为非0。 如果它们对于任何r都不同，则你应该定义该宏来返回0， 除非某个其它机制能够确保值可以按照更窄的模式来访问。

你应该定义该宏来尽可能情况的返回非0， 因为这样会使得GCC执行更好的寄存器分配。

bool TARGET_HARD_REGNO_SCRATCH_OK (unsigned int regno) [Target Hook]

This target hook should return true if it is OK to use a hard register regno as scratch reg in peephole2.

One common use of this macro is to prevent using of a register that is not saved by a prologue in an interrupt handler.

The default version of this hook always returns true.

AVOID_CCMODE_COPIES [Macro]

定义该宏，如果编译器应该避免复制从/到CCmode寄存器。 你应该只当对复制从/到CCmode寄存器的支持不完善的时候定义该宏。

17.7.4 处理叶子函数

在一些机器上，一个叶子函数（即，不做任何调用的函数），如果其不创建自己的寄存器窗口，则可以运行的更加有效。通常这意味着，其需要通过调用者传递参数的寄存器来接收它的参数，而不是它们通常到达的寄存器。

通常只有当叶子函数还满足其它条件时，才会对其进行特殊的对待；例如，通常它们可能只使用，用于它自己的变量和临时对象的那些寄存器。我们使用术语“叶子函数”来指一个适合这样特殊处理的函数，所以没有函数调用的函数并不一定是“叶子函数”。

GCC是在它知道函数是否适合作为叶子函数来对待之前，分配寄存器编号的。所以它需要重编号寄存器，以便输出一个叶子函数。下面的宏用来完成此事。

LEAF_REGISTERS [Macro]

一个char向量的名字，按照硬件寄存器编号进行索引，对于允许作为叶子函数处理的候选寄存器，其值为1。

如果叶子函数处理涉及到重编号寄存器，则这里标记的寄存器应该是重编号之前的——那些GCC通常分配的。重编号后，在汇编代码中实际被使用的寄存器，不要在该向量中被标记为1。

只有当目标机器提供了优化叶子函数处理的方法时，才定义该宏。

LEAF_REG_REMAP (regno) [Macro]

一个C表达式，当函数作为叶子函数来处理时，其值为应该对regno进行重编的寄存器编号。

如果regno为一个在重编号前不应该出现在叶子函数中的寄存器编号，则表达式的值应该为-1，这将造成编译器中断退出。

只有当目标机器提供了优化叶子函数处理的方法，并且寄存器需要被重编号的时候，才定义该宏。

TARGET_ASM_FUNCTION_PROLOGUE和TARGET_ASM_FUNCTION_EPILOGUE通常必须专门处理叶子函数。它们可以测试C变量current_function_is_leaf，其对于叶子函数为非0。current_function_is_leaf在局部寄存器分配之前被设置，并且可以用于剩余的编译器过程。它们还可以测试C变量current_function_uses_only_leaf_regs，其对于只用叶子寄存器的叶子函数为非0。current_function_uses_only_leaf_regs在所有修改指令的过程被运行完之后可用，并且只有当LEAF_REGISTERS被定义时才有用。

17.7.5 形成栈的寄存器

有一些特性用来处理计算机中形成栈的寄存器。 栈寄存器通常写成被压入一个栈中，并相对于栈顶进行编号。

目前，GCC只能处理一组类栈的寄存器，并且它们的编号必须是连续的。 而且，现存的对类栈寄存器的支持是特定于80387浮点协处理器。 如果你有一个新的体系结构使用了类栈寄存器， 你将需要在`reg-stack.c'上做大量的工作，并书写你自己的机器描述， 同时还要定义这些宏。

STACK_REGS [Macro]

如果机器具有任何类栈寄存器，则定义该宏。

FIRST_STACK_REG [Macro]

第一个类栈寄存器的编号。这是栈顶。

LAST_STACK_REG [Macro]

最后一个类栈寄存器的编号，这是栈底。

17.8 寄存器类别

在许多机器上，编号寄存器并不都是等价的。例如，一些寄存器不可以用作索引寻址； 一些寄存器不可以用于某些指令。 这些机器限制使用寄存器类别(register classes)来描述给编译器。

你定义一些寄存器类别，给出每个类别的名字并指名哪些寄存器属于它。然后， 你可以指定哪些寄存器类别对于特定的指令模式可以用作操作数。

总的来说，每个寄存器将属于多个类别。实际上， 必须有一个名为ALL_REGS的类别，包含所有的寄存器。 另外必须有一个名为NO_REGS的类别，不包含寄存器。 通常两个类别的并集将成为另一个类别；但并不这么要求。

其中一个类别必须名为GENERAL_REGS。该名字没有什么特殊的， 但是操作数约束字母`r'和`g'专门指定该类。 如果GENERAL_REGS与ALL_REGS相同， 则可以将其定义为扩展成ALL_REGS的宏。

对类别进行排序，使得如果类x包含在类y中， 则x具有比y更低的类别编号。

在操作数约束中指定GENERAL_REGS之外的类别的方法， 是通过机器相关的约束字母。你可以定义该字母来对应于不同的类别， 然后在操作数约束中使用它们。

只要有指令同时允许两个类别，就应该定义一个它们的并集的类别。 例如，如果对于一个特定的操作数， 有一条指令允许一个浮点（协处理器）寄存器或者一个通用寄存器， 则应该定义一个类别FLOAT_OR_GENERAL_REGS，其包含它们两。 否则你不会得到最优的代码。

你还必须指定关于寄存器类别的冗余信息：对于每个类别， 有哪些类别包含它以及哪些被它包含； 对于每个类别对，最大的类包含在它们的并集中。

当一个值占用多个连续的位于特定类别的寄存器时， 所有被使用的寄存器必须属于那个类别。因此， 寄存器类别不能用于要求寄存器对起始于偶数编号的寄存器。 用来指定该要求的方法是使用HARD_REGNO_MODE_OK。

用于按位与或者移位指令的操作数的寄存器类别具有特殊的要求： 对于每个定点机器模式，每个这样的类必须具有一个子类， 它的寄存器可以按照该机器模式与内存进行传送值。例如，在一些机器上， 对于单字节值的操作（QImode）被限制为特定的寄存器。这样的话， 每个用于按位与或者移位指令的寄存器类别必须具有一个子类， 组成它的寄存器可以用来加载或存储单字节值。 这使得PREFERRED_RELOAD_CLASS总是具有一个可以返回的值。

enum reg_class [Data type]

一个枚举类型，必须使用所有的寄存器类别名作为枚举值来定义。 NO_REGS必须位于最前面。ALL_REGS必须为最后一个寄存器类别， 后面再跟随一个枚举值，LIM_REG_CLASSES，其不是一个寄存器类别， 但是用来告诉有多少个类别。

每个寄存器类别具有一个编号，其为将类别名映射到int类型的值。 编号在下面描述的许多表中用作索引。

N_REG_CLASSES [Macro]

不同寄存器类别的数目，定义为:

```
#define N_REG_CLASSES (int) LIM_REG_CLASSES
```

REG_CLASS_NAMES [Macro]

一个初始化值，包含了作为C字符串常量的寄存器类别的名字。 这些名字用于书写一些调试转储。

REG_CLASS_CONTENTS [Macro]

一个初始化值，包含了寄存器类别的内容，作为位掩码的整数。 第n个整数指定了类别n的内容。整数掩码的解析方式为， 寄存器r在类别中，如果mask & (1 << r)为1。

当机器具有多于32个寄存器的时候，一个整数还不能满足。这时整数被替换为子初始化值，为括号包裹的多个整数。每个子初始化值必须适合类型HARD_REG_SET的初始化值， 其在`hard-reg-set.h'中定义。这种情况下， 每个子初始化值的第一个整数对应于寄存器0到31，第二个对应于32到63，等等。

REGNO_REG_CLASS (regno) [Macro]

一个C表达式，其值为包含了硬件寄存器regno的寄存器类别。总的来说， 会有不止一个这样的类别；选择最小的那个，这意味着没有更小的类别包含该寄存器。

BASE_REG_CLASS [Macro]

一个宏，它的定义为有效的基址寄存器必须属于的类别名字。 基址寄存器用于由寄存器的值加上一个偏移量来表示的地址中。

MODE_BASE_REG_CLASS (mode) [Macro]
这是宏BASE_REG_CLASS的变体，其允许在机器模式相关的方式下选择基址寄存器。 如果mode为VOIDmode，则其应该返回根BASE_REG_CLASS同样的值。

MODE_BASE_REG_REG_CLASS (mode) [Macro]
一个C表达式，其值为有效的基址寄存器必须属于的类别名字， 且用于一个基址寄存器加上索引寄存器的地址中。你应该定义该宏， 如果表示为基址寄存器加上索引寄存器的地址具有不同于其它基址寄存器的用法要求时。

MODE_CODE_BASE_REG_CLASS (mode, outer_code, index_code) [Macro]
一个C表达式，其值为有效的基址寄存器必须属于的类别名字。 outer_code和index_code定义了基址寄存器出现的上下文。 outer_code为。index_code为相应的索引表达式的代码，如果outer_code为PLUS；否则为SCRATCH。

INDEX_REG_CLASS [Macro]
一个C表达式，其值为有效的索引寄存器必须属于的类别名字。 索引寄存器为一个用于地址中的寄存器， 它的值用于乘于一个标量因子或者加上另一个寄存器（也可以加上一个偏移量）。

REGNO_OK_FOR_BASE_P (num) [Macro]
一个C表达式，其为非0， 如果寄存器编号num适合在操作数地址中作为基址寄存器使用。 其可以为一个合适的硬件寄存器，或者一个已经被分配了这样的硬件寄存器的伪寄存器。

REGNO_MODE_OK_FOR_BASE_P (num, mode) [Macro]
一个C表达式，类似于REGNO_OK_FOR_BASE_P， 除了表达式可以检查内存引用的机器模式mode。你应该定义该宏， 如果内存引用的机器模式影响了一个寄存器是否可以作为基址寄存器使用。 如果你定义了该宏，则编译器将使用它来替代REGNO_OK_FOR_BASE_P。 对于出现在MEM之外的地址，即作为一个address_operand， mode可以为VOIDmode。

REGNO_MODE_OK_FOR_REG_BASE_P (num, mode) [Macro]
一个C表达式，其为非0， 如果寄存器编号num适合在表示为基址加索引的地址中作为一个基址寄存器使用， 并通过模式mode来访问。其可以为一个合适的硬件寄存器， 或者一个已经被分配了这样的硬件寄存器的伪寄存器。你应该定义该宏， 如果表示为基址寄存器加上索引寄存器的地址具有不同于其它基址寄存器的用法要求时。

不赞成使用该宏；请使用更加通用的REGNO_MODE_CODE_OK_FOR_BASE_P。

REGNO_MODE_CODE_OK_FOR_BASE_P (num, mode, outer_code, index_code) [Macro]
一个C表达式，类似于REGNO_MODE_OK_FOR_BASE_P， 除了表达式可以检查寄存器出现内存引用中的上下文。outer_code为。 index_code为相应的索引表达式的代码，如果outer_code为PLUS； 否则为SCRATCH。对于出现在MEM之外的地址， 即作为一个address_operand，mode可以 为 VOIDmode。 A C expression that is just like REGNO_MODE_OK_FOR_BASE_P, except that that expression may examine the context in which the register appears in the memory reference. outer_code is the code of the immediately enclosing expression (MEM if at the top level of the address, ADDRESS for something that occurs in an address_operand). index_code is the code of the corresponding index expression if outer_code is PLUS; SCRATCH otherwise. The mode may be VOIDmode for addresses that appear outside a MEM, i.e., as an address_operand.

REGNO_OK_FOR_INDEX_P (num) [Macro]
一个C表达式，其为非0， 如果寄存器编号num适合作为索引寄存器用于操作数地址中。 其可以为一个合适的硬件寄存器或者一个已经被分配了这样的硬件寄存器的伪寄存器。

索引寄存器和基址寄存器的区别是，索引寄存器可以被标量化。 如果一个地址包含了两个寄存器的和，并且都不被标量化， 则一个可以被标签为“base” 另一个为“index”；但是使用哪个标签， 必须要适合机器的约束。编译器将尝试两种标签方式，来查找有效的一种， 并且当两种方式都无法工作时，重载一个或两个寄存器。

PREFERRED_RELOAD_CLASS (x, class) [Macro]

一个C表达式，对寄存器类别进行额外的限制， 用于当需要复制值x到类别为class的寄存器中。 值为一个寄存器类别；可能为class或者其它比class小的类别。 在许多机器上，下列定义是安全的：

```
#define PREFERRED_RELOAD_CLASS(X, CLASS) CLASS
```

有时返回一个更加限制的类别将会产生更好的代码。例如，在68000上， 当x为一个整数常量，并且在`moveq'指令的范围内， 则该宏的值总是为DATA_REGS，只要class包含数据寄存器。需要一个数据寄存器保证了`moveq'将使用。

一种PREFERRED_RELOAD_CLASS必须不返回class的情况为， 如果x为一个合法常量其不能被加载到某个寄存器类别中。 通过返回NO_REGS，你可以强迫x放入内存位置中。例如， rs6000可以加载立即数值到通用寄存器中， 但没有指令可以加载立即数值到浮点寄存器中， 所以PREFERRED_RELOAD_CLASS返回NO_REGS， 当x为一个浮点常量时。如果常量不能被加载到任何种类的寄存器中， 如果LEGITIMATE_CONSTANT_P使常量为非法的， 而不是使用PREFERRED_RELOAD_CLASS，则代码生成将会更好。

如果一个insn在寄存器分配之后具有伪寄存器， 则重载将遍历选择项并且反复调用PREFERRED_RELOAD_CLASS来找到最好的一个。 返回NO_REGS，在这种情况下，使得重载在约束前增加一个!： x86后端使用该特征来劝阻使用387寄存器，当算术在SSE寄存器中进行时（反之亦然）。

PREFERRED_OUTPUT_RELOAD_CLASS (x, class) [Macro]

类似于PREFERRED_RELOAD_CLASS，但是用于输出重载而不是输入重载。 如果没有定义该宏，则缺省为使用不变的class。

你还可以使用PREFERRED_OUTPUT_RELOAD_CLASS来劝阻使用一些可选项的重载， 类似于PREFERRED_RELOAD_CLASS。

LIMIT_RELOAD_CLASS (mode, class) [Macro]

一个C表达式，对寄存器类别实施额外的限制， 用于当需要在一个类别为class的重载寄存器中保存机器模式为mode的值的时候。

不像PREFERRED_RELOAD_CLASS， 该宏应该用于当有特定的机器模式不能简单的放入特定的重载类别中的时候。

值为一个寄存器类别；可能为class，或者其它更小的类别。

不要定义该宏，除非target机器具有一些限制，使得宏需要做一些事情。

enum reg_class TARGET_SECONDARY_RELOAD (bool in_p, rtx x, enum reg_class reload_class, enum machine_mode reload_mode, secondary_reload_info *sri) [Target Hook]

许多机器具有一些寄存器，其不能直接和内存之间进行复制， 甚至不能和其它类型的寄存器。一个例子是`MQ'寄存器， 在大多数机器上，只能与通用寄存器直接进行复制，而不能和内存之间。 下面，我们将使用术语‘中途寄存器’ ，当一个move操作不能直接执行， 而必须首先通过将源复制到中途寄存器中，然后再从中途寄存器复制到目的。 一个中途寄存器总是具有与源和目的相同的机器模式。由于其存放了被复制的实际的值， 所以重载可能进行优化来重用一个中途寄存器，并且省略掉从源进行复制的操作， 当它可以确定中途寄存器还保留着所需要的值的时候。

另一种需要执行二次重载的情况是，在一些机器上， 其允许所有的寄存器和内存之间进行复制， 但要求一个scratch寄存器来存储一些内存的位置（例如，在RT上那些具有符号地址的，以及在SPARC上当编译PIC时那些具有特定符号地址的）。 草稿寄存器不需要与被复制的值具有相同的机器模式，并且通常保留一个不同的值。 在md文件中需要特殊的指令模式来描述复制在草稿寄存器的帮助下如何执行； 这些指令模式还描述了草稿寄存器的编号，寄存器类别和机器模式。

在一些情况下，同时需要中途寄存器和草稿寄存器。

对于输入重载，该target钩子使用非零的in_p来调用，并且x为一个rtx， 其需要被复制到一个类别为reload_class，机器模式为reload_mode的寄存器中。 对于输出重载，该target钩子使用为0的in_p调用，并且一个类别为reload_class， 需要复制到机器模式为reload_mode的rtx x中。

如果在类别为reload_class的寄存器和x直接进行复制需要一个中途寄存器， 则钩子secondary_reload应该返回一个该中途寄存器需要的寄存器类别。 如果不需要中途寄存器，则其应该返回NO_REGS。如果需要多个中途寄存器， 则描述在复制链中最近的那个。

如果需要草稿寄存器， 则还要描述如何如何在重载寄存器和这个最近的中途寄存器直接进行复制。 或者如果不需要中途寄存器，但仍然需要一个草稿寄存器， 则描述重载寄存器和重载操作数x之间的复制。

为此，你需要设置sri->icode为在md文件中执行move的指令模式的代码。 操作数0和1分别为该复制的输出和输入。从2以后的操作数为草稿操作数。 这些草稿操作数必须具有机器模式并且一个single-register-class的输出约束。

当使用中途寄存器的时候，secondary_reload钩子将会被再次调用， 来确定如何在中途寄存器和重载操作数之间进行复制， 所以你的钩子必须还要具有处理中途操作数的寄存器类别的代码。

x可以为伪寄存器或者一个伪寄存器的subreg， 其可以为一个硬件寄存器或者在内存中。使用true_regnum来查看； 其将返回-1如果伪寄存器在内存中，以及硬件寄存器编号，如果其在一个寄存器中。

在内存中的草稿操作数（约束"=m"/"=&m"）目前不被支持。 为此，目前你必须继续使用SECONDARY_MEMORY_NEEDED。

copy_cost还是要该target钩子来查找值如何被复制。 如果你想让其包含像分配草稿寄存器所需要的额外代价， 则可以设置sri->extra_cost为额外代价。 或者如果两个相关move会具有比两个单个move之和的代价要低， 则可以设置sri->extra_cost为一个负数。

SECONDARY_RELOAD_CLASS (class, mode, x) [Macro]
SECONDARY_INPUT_RELOAD_CLASS (class, mode, x) [Macro]
SECONDARY_OUTPUT_RELOAD_CLASS (class, mode, x) [Macro]

These macros are obsolete, new ports should use the target hook TARGET_SECONDARY_RELOAD instead.

These are obsolete macros, replaced by the TARGET_SECONDARY_RELOAD target hook. Older ports still define these macros to indicate to the reload phase that it may need to allocate at least one register for a reload in addition to the register to contain the data. Specifically, if copying x to a register class in mode requires an intermediate register, you were supposed to define SECONDARY_INPUT_RELOAD_CLASS to return the largest register class all of whose registers can be used as intermediate registers or scratch registers.

If copying a register class in mode to x requires an intermediate or scratch register, SECONDARY_OUTPUT_RELOAD_CLASS was supposed to be defined be defined to return the largest register class required. If the requirements for input and output reloads were the same, the macro SECONDARY_RELOAD_CLASS should have been used instead of defining both macros identically.

The values returned by these macros are often GENERAL_REGS. Return NO_REGS if no spare register is needed; i.e., if x can be directly copied to or from a register of class in mode without requiring a scratch register. Do not define this macro if it would always return NO_REGS.

If a scratch register is required (either with or without an intermediate register), you were supposed to define patterns for `reload_inm' or `reload_outm', as required (see Section 16.9 [标准名字], page 235. These patterns, which were normally implemented with a define_expand, should be similar to the `movm' patterns, except that operand 2 is the scratch register.

These patterns need constraints for the reload register and scratch register that contain a single register class. If the original reload register (whose class is class) can meet the constraint given in the pattern, the value returned by these macros is used for the class of the scratch register. Otherwise, two additional reload registers are required. Their classes are obtained from the constraints in the insn pattern.

x might be a pseudo-register or a subreg of a pseudo-register, which could either be in a hard register or in memory. Use true_regnum to find out; it will return −1 if the pseudo is in memory and the hard register number if it is in a register.

These macros should not be used in the case where a particular class of registers can only be copied to memory and not to another class of registers. In that case, secondary reload registers are not needed and would not be helpful. Instead, a stack location must be used to perform the copy and the movm pattern should use memory as an intermediate storage. This case often occurs between floating-point and general registers.

SECONDARY_MEMORY_NEEDED (class1, class2, m) [Macro]

一些机器要求某些寄存器必须使用内存才能与其它寄存器进行复制。 定义该宏在那些机器上，其为一个C表达式， 为非0如果机器模式为m的对象在类别为class1的寄存器中， 只能通过将class1的寄存器存储到内存中并且将内存位置加载到class2的寄存中。

如果其值总是为0，则不要定义该宏。

SECONDARY_MEMORY_NEEDED_RTX (mode) [Macro]

通常当SECONDARY_MEMORY_NEEDED被定义时， 编译器会分配一个栈槽为需要寄存器复制的内存位置。 如果该宏被定义，则编译器会替代的使用该宏定义的内存位置。

如果没有定义SECONDARY_MEMORY_NEEDED，则不要定义该宏。

SECONDARY_MEMORY_NEEDED_MODE (mode) [Macro]

当编译器需要一个二级内存位置在机器模式为mode的寄存器之间进行复制的时候， 其通常分配足够的内存来存放BITS_PER_WORD个位， 并且执行该位数宽度的mode的存储和加载操作。

这在大多数机器上是正确的， 因为其确保寄存器的所有位被复制并且阻止对寄存器按照较窄的机器模式来访问， 这对于浮点寄存器通常是禁止的。

然而，该缺省行为在一些机器上是不正确的，例如DEC Alpha， 其在浮点寄存器中存储short整数与在整数寄存器中是不同的。 在那些机器上，缺省的宽度将不正确，你必须定义该宏来抑制这种宽度。 详情参见`alpha.h'文件。

如果没有定义，或者如果BITS_PER_WORD个位数的宽度是正确的，则不要定义该宏。

SMALL_REGISTER_CLASSES [Macro]

在一些机器上，让硬件寄存器的活跃性跨越任意的insn是危险的。 特别是，这些机器具有要求值要在特定寄存器中的指令（像累加器），并且重载将会失败如果所要求的硬件寄存器用于其它目的。

定义SMALL_REGISTER_CLASSES为一个表达式，非0的值在这些机器上。 当该宏具有非0值，则编译器尝试最小化硬件寄存器的生命期。

定义该宏为非0值总是安全的，但是如果你没必要定义它，则你将会减少优化数目。 如果你没有定义该宏为非0，并且当其需要时， 则编译器会产生寄存器溢出并且打印一个fatal error消息。 对于大多数机器，你根本不需要定义该宏。

CLASS_LIKELY_SPILLED_P (class) [Macro]

A C expression whose value is nonzero if pseudos that have been assigned to registers of class class would likely be spilled because registers of class are needed for spill registers.

The default value of this macro returns 1 if class has exactly one register and zero otherwise. On most machines, this default should be used. Only define this macro to some other expression if pseudos allocated by `local-alloc.c' end up in memory because their hard registers were needed for spill registers. If this macro returns nonzero for those classes, those pseudos will only be allocated by `global.c', which knows how to reallocate the pseudo to another register. If there would not be another register available for reallocation, you should not change the definition of this macro since the only effect of such a definition would be to slow down register allocation.

CLASS_MAX_NREGS (class, mode) [Macro]

一个C表达式，为需要存放机器模式为mode的值所需要的连续的类别为class的寄存器的最大数。

这与宏HARD_REGNO_NREGS很相近。实际上， 宏CLASS_MAX_NREGS (class, mode)的值应该为HARD_REGNO_NREGS (regno, mode)的最大值。

该宏有助于处理多字的值，在重载过程中。

CANNOT_CHANGE_MODE_CLASS (from, to, class) [Macro]

如果被定义，为一个C表达式，其返回非0，对于一个class， 其由机器模式from到to的改变是无效的。

例如，加载32位整数或者浮点对象到浮点寄存器中，在Alpha上将被扩展为64位。 因此加载64位对象并存储为32位对象时将不保存第32位。 因此，`alpha.h'定义CANNOT_CHANGE_MODE_CLASS如下：

```
#define CANNOT_CHANGE_MODE_CLASS(FROM, TO, CLASS) \
  (GET_MODE_SIZE (FROM) != GET_MODE_SIZE (TO) \
  ? reg_classes_intersect_p (FLOAT_REGS, (CLASS)) : 0)
```

const enum reg_class * TARGET_IRA_COVER_CLASSES () [Target Hook]

Return an array of cover classes for the Integrated Register Allocator (IRA). Cover classes are a set of non-intersecting register classes covering all hard registers used

for register allocation purposes. If a move between two registers in the same cover class is possible, it should be cheaper than a load or store of the registers. The array is terminated by a LIM_REG_CLASSES element.

This hook is called once at compiler startup, after the command-line options have been processed. It is then re-examined by every call to target_reinit.

The default implementation returns IRA_COVER_CLASSES, if defined, otherwise there is no default implementation. You must define either this macro or IRA_COVER_CLASSES in order to use the integrated register allocator with Chaitin-Briggs coloring. If the macro is not defined, the only available coloring algorithm is Chow's priority coloring.

IRA_COVER_CLASSES [Macro]

See the documentation for TARGET_IRA_COVER_CLASSES.

17.9 废弃的定义约束的宏

机器特定的约束可以使用这些宏来定义，来替代在Section 16.8.7 [定义约束], page 233中描述的机器描述结构。 这种机制已经被废弃；旧的port应该转换为新的机制。

CONSTRAINT_LEN (char, str) [Macro]

对于起始于str，其起始字母为c的约束，返回其长度。 这允许你具有比单个字母更长的寄存器类别/常量/额外约束； 你不需要定义该宏，如果你只用单个字母的约束。 该宏的定义应该使用DEFAULT_CONSTRAINT_LEN，对于你不想特别处理的所有字符。 在genoutput.c中有一些合理性检查，用来为md文件检查约束的长度。

REG_CLASS_FROM_LETTER (char) [Macro]

一个C表达式，其为寄存器类别定义了机器相关的操作数约束字母。 如果char为这样的字母，则值应该为对应的寄存器类别。否则， 值应该为NO_REGS。寄存器字母`r'，对应于类别GENERAL_REGS， 将不被传给该宏；你不需要处理它。

REG_CLASS_FROM_CONSTRAINT (char, str) [Macro]

类似于REG_CLASS_FROM_LETTER，不过你还得到在str中传递的字符串， 所以你可以使用后缀来区别不同的变种。

CONST_OK_FOR_LETTER_P (value, c) [Macro]

一个C表达式，其定义了机器相关操作数约束字母（`I', `J', `K', ... `P'），指定了整数值的特定范围。如果c为那些字母中的， 则表达式应该检查value，一个整数，如果在合适的范围中则返回1，否则返回0。 如果c不是那些字母中的，则值应该为0，而不管value是多少。

CONST_OK_FOR_CONSTRAINT_P (value, c, str) [Macro]

类似CONST_OK_FOR_LETTER_P，但是你还得到在str中传递的字符串， 所以你可以使用后缀来区别不同的变种。

CONST_DOUBLE_OK_FOR_LETTER_P (value, c) [Macro]

一个C表达式，定义了机器相关操作数约束字母， 指定了const_double值的特定范围（`G'或`H'）。

如果c为那些字母中的，则表达式应该检查value， 一个代码为const_double的RTX，如果在合适的范围中则返回1，否则返回0。 如果c不是那些字母中的，则值应该为0，而不管value是多少。

const_double用于所有的浮点常量和DImode定点常量。 一个给定的字母可以接受一种或者这两种类型的值。 其可以使用GET_MODE来区别这些类型。

CONST_DOUBLE_OK_FOR_CONSTRAINT_P (value, c, str) [Macro]
类似CONST_DOUBLE_OK_FOR_LETTER_P，但是你还得到在str中传递的字符串， 所以你可以使用后缀来区别不同的变种。

EXTRA_CONSTRAINT (value, c) [Macro]
一个C表达式，定义了可选的机器相关约束字母， 其可以用于为target机器隔离特定类型的操作数，通常为内存引用。 任何没有在其它地方定义并且不被 REG_CLASS_FROM_LETTER / REG_CLASS_FROM_CONSTRAINT匹配的字母都可以使用。 通常该宏将不被定义。

如果对于特定的target机器需要该宏，则应该返回1， 如果value对应于由约束字母c表示的操作数类型。 如果c没有作为extra约束定义，则值应该为0，而不管value是多少。

例如，则ROMP上，加载指令不能将它们的输出放在r0中， 如果内存引用包含了一个符号地址。 约束字母`Q'被定义来表示不包含符号地址的内存地址。 一个可选项使用`Q'约束在输入上并且`r'在输出上来指定。 下一个可选项指定了`m'在输入上并且不包含r0的寄存器类别在输出上。

EXTRA_CONSTRAINT_STR (value, c, str) [Macro]
类似EXTRA_CONSTRAINT，但是你还得到在str中传递的字符串， 所以你可以使用后缀来区别不同的变种。

EXTRA_MEMORY_CONSTRAINT (c, str) [Macro]
一个C表达式，定义了可选的机器相关约束字母， 在那些由EXTRA_CONSTRAINT接受的字母中，其应该被重载过程作为内存约束来对待。

其应该返回1，如果由str起始，并且第一个字母为c的约束所表示的操作数 类型所包含的所有内存引用为简单的基址寄存器。这允许重载过程来重载操作数， 如果其不直接对应于操作数类型c，通过将其地址复制到基址寄存器中。

例如，在S/390上，一些指令不接受任意的内存引用，只接受那些不使用索引寄存器的。 约束字母`Q'被通过EXTRA_CONSTRAINT定义，来表示这种类型的内存地址。 如果字母`Q'被标记为 EXTRA_MEMORY_CONSTRAINT， 则一个`Q'常量可以处理任何内存操作数， 因为重载过程知道其可以通过将内存地址复制到基址寄存器中如果需要的话。 这类似于可以处理任何内存操作数的`o'约束的方式。

EXTRA_ADDRESS_CONSTRAINT (c, str) [Macro]
一个C表达式，定义了可选的机器相关约束字母，在那些由 EXTRA_CONSTRAINT / EXTRA_CONSTRAINT_STR接受的字母中， 其应该被重载过程作为地址约束来对待。

其应该返回1，如果由str起始，并且第一个字母为c的约束所表示的操作数 类型所包含的所有内存引用为简单的基址寄存器。这允许重载过程来重载操作数， 如果其不直接对应于操作数类型str，通过将其地址复制到基址寄存器中。

标记为EXTRA_ADDRESS_CONSTRAINT的约束只能与address_operand断言 一起使用。其类似于`p'约束。

17.10 栈布局和调用约定

这一节描述了栈的布局和调用约定。

17.10.1 基本的帧布局

这里是基本的栈布局。

STACK_GROWS_DOWNWARD [Macro]

定义该宏，如果将一个字压入栈中使得栈指针移向更小的地址。

当我们说“定义该宏，如果...”，这意味着编译器只是使用#ifdef来检查该宏，所以具体定义的值并没有关系。

STACK_PUSH_CODE [Macro]

该宏定义了当压栈所使用的操作。对于RTL形式，压栈操作将为(set (mem (STACK_PUSH_CODE (reg sp))) ...)。

可选择的方式为PRE_DEC, POST_DEC, PRE_INC和POST_INC。使用哪一个是正确的，取决于栈的方向和栈指针是否指向栈中的最后一项，还是指向之后的空间。

缺省为PRE_DEC，当STACK_GROWS_DOWNWARD被定义时，这大多情况下都是正确，否则为PRE_INC，这经常是错误的。

FRAME_GROWS_DOWNWARD [Macro]

定义该宏为非零值，如果局部变量槽的地址位于帧指针的负偏移处。

ARGS_GROW_DOWNWARD [Macro]

定义该宏，如果函数的连续的参数在栈上的地址是递减的。

STARTING_FRAME_OFFSET [Macro]

帧指针到第一个被分配的局部变量槽的偏移量。

如果FRAME_GROWS_DOWNWARD，则通过从STARTING_FRAME_OFFSET减去第一个 栈槽的长度来查找下一个栈槽的偏移量。否则，通过从STARTING_FRAME_OFFSET 加上第一个栈槽的长度来查找。

STACK_ALIGNMENT_NEEDED [Macro]

定义为0，来禁止在重载过程中对栈的对齐。缺省的非0值适合于大多part。

在一些port上，STARTING_FRAME_OFFSET为非0，或者在局部块之后有一块寄存器保存区域，其不需要对齐到STACK_BOUNDARY，这样禁止栈对齐并且在后端实现可能会更好。

STACK_POINTER_OFFSET [Macro]

从栈指针寄存器到第一个输出的参数所放在的位置的偏移量。如果没有指定，则缺省值0被使用。这对于大多数机器都合适。

如果ARGS_GROW_DOWNWARD，则这是输出参数位于的第一个位置的上面的位置的偏移量。

FIRST_PARM_OFFSET (fundecl) [Macro]

参数指针寄存器到第一个参数的地址的偏移量。在一些机器上，其可能依赖于函数的数据类型。

如果ARGS_GROW_DOWNWARD，则这是第一个参数的地址的上面的位置的偏移量。

STACK_DYNAMIC_OFFSET (fundecl) [Macro]

栈指针寄存器到在栈上动态分配的对象的偏移量，例如，通过alloca。

该宏的缺省值为STACK_POINTER_OFFSET加上输出参数的长度。缺省值对于大多数机器是正确的。详情参见`function.c'。

INITIAL_FRAME_ADDRESS_RTX [Macro]

一个C表达式，其值为RTL，表示初始栈帧的地址。该地址被传给RETURN_ADDR_RTX和DYNAMIC_CHAIN_ADDRESS。如果你没有定义该宏，则一个合理的缺省值将被使用。定义该宏，可以使帧指针消除在__builtin_frame_address (count)和 __builtin_return_address (count)不等于0的情况下工作。

DYNAMIC_CHAIN_ADDRESS (frameaddr) [Macro]

一个C表达式，其值为RTL，表示栈帧中的地址，指向被存储的调用者的帧。 假设frameaddr为一个栈帧本身的地址的RTL表达式。

如果你没有定义该宏，则缺省为返回frameaddr的值——也就是说， 栈帧地址也是指向之前帧的地址。

SETUP_FRAME_ADDRESSES [Macro]

如果定义，为一个C表达式，其产生机器特定的代码来建立栈，使得可以访问任意的帧。 例如，在SPARC上，我们必须刷新栈的所有寄存器窗口，在我们可以访问任意栈帧之前。 你很少会需要定义该宏。

bool TARGET_BUILTIN_SETJMP_FRAME_VALUE () [Target Hook]

该target钩子应该返回一个rtx，用于将当前帧的地址存储到内建的setjmp缓存中。 缺省值，virtual_stack_vars_rtx，对于大多数机器是正确的。 一种你可能需要定义该target钩子的原因是， 如果hard_frame_pointer_rtx在你的机器上是合适的值。

FRAME_ADDR_RTX (frameaddr) [Macro]

一个C表达式，其值为RTL，表示当前帧的帧地址。frameaddr为当前帧的帧指针。 这用于__builtin_frame_address。你只有当帧地址与帧指针不同的时候才需要定义该宏。 大多数机器不需要定义该宏。

RETURN_ADDR_RTX (count, frameaddr) [Macro]

A C expression whose value is RTL representing the value of the return address for the frame count steps up from the current frame, after the prologue. frameaddr is the frame pointer of the count frame, or the frame pointer of the $\mathrm{count} - 1$ frame if RETURN_ADDR_IN_PREVIOUS_FRAME is defined.

The value of the expression must always be the correct address when count is zero, but may be NULL_RTX if there is no way to determine the return address of other frames.

RETURN_ADDR_IN_PREVIOUS_FRAME [Macro]

定义该宏，如果一个特定的栈帧的返回地址是从之前栈帧的帧指针中访问的。

INCOMING_RETURN_ADDR_RTX [Macro]

一个C表达式，其值为RTL，表示在任何函数的起始处，在序言之前， 流入的返回地址的位置。该RTL或者为一个REG， 指示返回地址保存在`REG'中，或者一个MEM表示位于栈中。

你只在你想支持调用帧调试信息，像DWARF2提供的那样，的时候才需要定义该宏。

如果该RTL为一个REG， 你还要定义DWARF_FRAME_RETURN_COLUMN为DWARF_FRAME_REGNUM (REGNO)。

DWARF_ALT_FRAME_RETURN_COLUMN [Macro]

一个C表达式，其值为一个整数，给出了DWARF2的列号，可以用作替代的返回列。 column必须不对应于任何gcc硬件寄存器（也就是说， 其必须不在DWARF_FRAME_REGNUM的范围中）。

该宏当被设为一个通用寄存器，但是候选的column需要用于signal帧的时候会很有用。 一些target还使用了不同的帧返回列。

DWARF_ZERO_REG [Macro]

一个C表达式，其值为一个整数，给出了DWARF2寄存器编号，其被认为总是具有值0。 这应该只当target的体系结构中具有一个0寄存器并且认为使用寄存器编号来确定栈的 回溯是一个好主意的时候才被定义。新的part应该避免该宏。

void TARGET_DWARF_HANDLE_FRAME_UNSPEC (const char *label, rtx pattern, int index) [Target Hook]

该target钩子允许后端生成帧相关的insn，其包含了UNSPECs或UNSPEC_VOLATILEs。DWARF2调用帧调试信息引擎将会按照如下的形式来调用它

```
(set (reg) (unspec [...] UNSPEC_INDEX))
```

和

```
(set (reg) (unspec_volatile [...] UNSPECV_INDEX)).
```

来使后端生成调用帧指令。label为insn附带的CFI标号， pattern为insn的指令模式， index为UNSPEC_INDEX或UNSPECV_INDEX。

INCOMING_FRAME_SP_OFFSET [Macro]

一个C表达式，其值为一个整数，给出了偏移字节数， 从栈指针寄存器到任何函数的起始处，序言之前的栈帧的顶部。 帧的顶部被定义为之前帧的栈指针的值，就在call指令之前。

你只有当你想支持像DWARF2提供的那样的帧调试信息时才需要定义该宏。

ARG_POINTER_CFA_OFFSET (fundecl) [Macro]

一个C表达式，其值为一个整数，给出了偏移字节数，从参数指针到规范化帧地址（cfa）。最终的值应该与通过INCOMING_FRAME_SP_OFFSET所计算的一致。 不幸的是这在虚寄存器实例化的时候不可用。

该宏的缺省值为FIRST_PARM_OFFSET (fundecl)，其对于大多数机器是正确的； 总的来说，参数在栈帧之前被找到。注意有些情况不是这样的，一些target将寄存器保存在调用者的帧中，像SPARC和rs6000，这样的target就不需要定义该宏。

你只有当缺省是不正确的时候， 以及你想支持像DWARF2提供的那样的帧调试信息时才需要定义该宏。

FRAME_POINTER_CFA_OFFSET (fundecl) [Macro]

如果被定义，则为一个C表达式，其值为一个整数，给出了偏移字节数， 从帧指针到规范化帧地址（cfa）。 最终的值应该与通过INCOMING_FRAME_SP_OFFSET所计算的一致。

通常CFA被作为参数指针的偏移量来计算，通过ARG_POINTER_CFA_OFFSET， 但是如果参数指针是一个变量，这就不太可能了。如果该宏被定义， 它暗示了虚寄存器实例化应该基于帧指针而不是参数指针。 FRAME_POINTER_CFA_OFFSET和ARG_POINTER_CFA_OFFSET只有一个应该被定义。

CFA_FRAME_BASE_OFFSET (fundecl) [Macro]

如果定义，则为一个C表达式，其值为一个整数，给出了偏移字节数， 从规范化帧地址（cfa）到DWARF2调试信息使用的frame base。缺省为0。 不同的值可以在一些port上减少调试信息的大小。

17.10.2 对异常处理的支持

EH_RETURN_DATA_REGNO (N) [Macro]

一个C表达式，其值为第N个寄存器的编号，用于异常处理的数据， 或者为INVALID_REGNUM，如果小于N个寄存器可用。

异常处理库程序与异常处理者通过一套协定好的寄存器来通讯。理想的， 这些寄存器应该为调用破坏的；可以使用调用保存的寄存器， 但可能会对代码大小产生负影响。target必须支持至少两个数据寄存器， 但如果有足够的可用的寄存器，则应该定义为4。

你必须定义该宏，如果你想支持像DWARF 2提供的调用帧异常处理。

EH_RETURN_STACKADJ_RTX [Macro]

一个C表达式，其值为RTL，表示一个位置，用来存储栈调整，在函数返回前应用。 这用于unwind栈到一个异常处理的调用帧中。其将被赋予0在通常的返回代码路径上。

通常这是一个调用破坏的硬件寄存器，但也可以为一个栈槽。

不要定义该宏，如果栈指针在调用帧本身通过序言和尾声来保存和恢复时； 这种情况下，异常处理库函数将更新栈位置并保存。否则，你必须定义该宏， 如果你想支持调用帧异常处理，就像DWARF2提供的那样。

EH_RETURN_HANDLER_RTX [Macro]

一个C表达式，其值为一个RTL，表示一个位置，用来存储我们应该返回的异常处理的地址。其在通常的返回的代码路径上将不被赋值。

通常这是在调用帧中通常返回地址存储的位置。对于通过在栈中弹出地址的target， 这可以为一个内存地址，就在target调用帧的下面，而不是在当前调用帧中。 如果被定义，EH_RETURN_STACKADJ_RTX将已经被赋值， 所以其可以用于计算target调用帧的位置。

一些target具有更加复杂的要求，比在初始化代码生成阶段存储到地址中。 这种情况下，要替代的使用eh_return指令模式。

如果你想支持调用帧异常处理，你必须定义该宏或者eh_return指令模式。

RETURN_ADDR_OFFSET [Macro]

如果定义，则为一个整数值的C表达式，并会为此生成rtl来加上异常处理地址， 在其在异常处理表中搜索之前，并且再减去它，在用它来返回到异常处理之前。

ASM_PREFERRED_EH_DATA_FORMAT (code, global) [Macro]

该宏选择在异常处理section中嵌入的指针的解码。如果尽可能，该宏应该被定义， 这样异常处理section将不会要求进行动态重定位，并可以为只读的。

code为0，对于数据，1对于代码标号，2对于函数指针。 global为真，如果符号可以由动态重定位影响。 宏应该返回在`dwarf2.h'中可以找到的DW_EH_PE_*的组合。

如果该宏没有定义，则指针将不被解码，而是直接表示。

ASM_MAYBE_OUTPUT_ENCODED_ADDR_RTX (file, encoding, size, addr, done) [Macro]

该宏允许target生成特定的magic， 用于表示ASM_PREFERRED_EH_DATA_FORMAT选择的encoding。 通常代码考虑pc-relative和indirect解码； 如果target使用text-relative或者data-relative解码，则必须定义该宏。

这是一个C语句，如果格式被处理，则执行分支跳转。encoding为选择的格式， size为格式占用的字节数，addr为生成的SYMBOL_REF。

MD_UNWIND_SUPPORT [Macro]

一个字符串，指定了文件在unwind-dw2.c中被#include包含进来。 被包含进来的文件通常定义了MD_FALLBACK_FRAME_STATE_FOR。

MD_FALLBACK_FRAME_STATE_FOR (context, fs) [Macro]

该宏允许target增加CPU和操作系统特定代码到call-frame unwinder， 用于当没有unwind数据可用时。最常见的原因是实现该宏来通过signal帧来unwind。

该宏由`unwind-dw2.c', `unwind-dw2-xtensa.c'和`unwind-ia64.c'中的 uw_frame_state_for调用。context为一个_Unwind_Context； fs为一个_Unwind_FrameState。检查context->ra来得到被执行的 代码的地址，检查context->cfa来得到栈指针的值。如果帧可以被解码， 则寄存

器保存地址应该在fs中更新，并且宏应该求值为_URC_NO_REASON。 如果帧不能被解码，则宏应该求解为_URC_END_OF_STACK。

对于java中合适的信号处理，该宏通过MAKE_THROW_FRAME来应用，在`libjava/include/*-signal.h'中定义。

MD_HANDLE_UNWABI (context, fs) [Macro]

该宏允许target增加操作系统特定的代码到调用帧unwinder， 来处理IA-64 .unwabi unwinding伪指令，通常用于signal或者interrrupt帧。

该宏由`unwind-ia64.c'中的uw_update_context调用。 context为一个_Unwind_Context； fs为一个_Unwind_FrameState。检查fs->unwabi来得到abi。 如果.unwabi伪指令可以被处理，则寄存器保存地址应该在fs中更新。

TARGET_USES_WEAK_UNWIND_INFO [Macro]

一个C表达式，计算为真，如果target需要unwind info给定comdat linkage。 定义其为1，如果comdat linkage有必要。缺省为0。

17.10.3 指定如何进行栈检查

GCC将检查栈引用是否位于栈的边界里，如果指定了`-fstack-check'， 使用三种方式的之一：

1. 如果STACK_CHECK_BUILTIN宏的值为非0， 则GCC将假设你已经安排了在配置文件的合适的地方进行栈检查，例如， 在TARGET_ASM_FUNCTION_PROLOGUE中。GCC将不再做其它特殊的处理。
2. If STACK_CHECK_BUILTIN is zero and the value of the STACK_CHECK_STATIC_BUILTIN macro is nonzero, GCC will assume that you have arranged for static stack checking (checking of the static stack frame of functions) to be done at appropriate places in the configuration files. GCC will only emit code to do dynamic stack checking (checking on dynamic stack allocations) using the third approach below.
3. 如果上面两种方式都不是，则GCC将生成代码来周期的“探测”栈指针， 使用下面定义的宏的值。

If neither STACK_CHECK_BUILTIN nor STACK_CHECK_STATIC_BUILTIN is defined, GCC will change its allocation strategy for large objects if the option `-fstack-check' is specified: they will always be allocated dynamically if their size exceeds STACK_CHECK_MAX_VAR_SIZE bytes.

STACK_CHECK_BUILTIN [Macro]

一个非0值，如果栈检查按照机器相关的方式通过配置文件来完成。 你应该定义该宏，如果栈检查被你的机器的ABI要求， 或者你想让栈检查使用比GCC可移植方式更有效的方法。该宏的缺省值为0。

STACK_CHECK_STATIC_BUILTIN [Macro]

A nonzero value if static stack checking is done by the configuration files in a machine-dependent manner. You should define this macro if you would like to do static stack checking in some more efficient way than the generic approach. The default value of this macro is zero.

STACK_CHECK_PROBE_INTERVAL [Macro]

一个整数表示GCC必须生成栈探测指令的间隔。 你通常需要定义该宏为不大于在栈结尾处的“guard pages”的大小。 缺省值4096适合于大多数系统。

STACK_CHECK_PROBE_LOAD [Macro]
一个整数，其为非0，如果GCC应该使用加载指令来执行栈探测， 为0如果GCC应该使用存储指令。缺省为0，其在大多数系统是更有效的选择。

STACK_CHECK_PROTECT [Macro]
栈从栈溢出中恢复所需要的字节数，用于这样的恢复被支持的语言。 缺省值75个字应该适合大多数机器。

The following macros are relevant only if neither STACK_CHECK_BUILTIN nor STACK_CHECK_STATIC_BUILTIN is defined; you can omit them altogether in the opposite case.

STACK_CHECK_MAX_FRAME_SIZE [Macro]
一个栈帧的最大大小，以字节为单位。 GCC将生成探测指令在非叶子函数来确保栈中至少这么多大小的字节是可用的。 如果一个栈帧大于该大小，则栈检查将不可靠并且GCC将产生一个警告。缺省值被选择， 使得GCC只生成一条指令，在大多数系统上。你通常不应该修改该宏的缺省值。

STACK_CHECK_FIXED_FRAME_SIZE [Macro]
GCC使用该值来生成上面的警告消息。其表示函数使用的固定的帧数量， 不包括用于任何被调用者保存的寄存器，临时变量和用户变量的空间。 你只需要指定该数量的上界并且通常使用缺省值，4个字。

STACK_CHECK_MAX_VAR_SIZE [Macro]
为当用户指定`-fstack-check'时，GCC将在栈帧的固定域放入的对象的最大大小， 以字节为单位。GCC根据上面的宏来计算缺省值并且你通常不需要覆盖缺省值。

17.10.4 用于栈帧寻址的寄存器

这里讨论了用于栈帧寻址的寄存器。

STACK_POINTER_REGNUM [Macro]
栈指针寄存器的寄存器编号，其还必须为一个FIXED_REGISTERS中的固定寄存器。 在大多数机器上，硬件决定了这是哪个寄存器。

FRAME_POINTER_REGNUM [Macro]
帧指针寄存器的寄存器编号，其用于访问在栈帧中的自动变量。 在一些机器上，硬件决定了这是哪个寄存器。 在其它机器上，你可以选择任意寄存器来达到该目的。

HARD_FRAME_POINTER_REGNUM [Macro]
在一些机器上，帧指针和自动变量的起始处之间的偏移量直到寄存器分配进行完之后才知道（例如，因为保存寄存器位于这两个位置之间）。这些机器上， 定义FRAME_POINTER_REGNUM为一个特定的固定寄存器的编号，在内部使用， 直到位移已知，并且定义HARD_FRAME_POINTER_REGNUM为实际的硬件寄存器， 用于帧指针。

你只需要在非常少见的情况下定义该宏，当不可能计算帧指针和自动变量的偏移时， 并且直到寄存器分配完成。当该宏被定义， 你必须还要在你的ELIMINABLE_REGS的定义中指示如果消除 FRAME_POINTER_REGNUM为HARD_FRAME_POINTER_REGNUM或者 STACK_POINTER_REGNUM。

如果其与FRAME_POINTER_REGNUM相同，则不要定义该宏。

ARG_POINTER_REGNUM [Macro]

arg指针寄存器的寄存器编号，其用于访问函数的参数列表。在大多数机器上，这与帧指针寄存器相同。在一些机器上，硬件决定了其为那个寄存器。在其它机器上，你可以选择任意的寄存器。如果这与帧指针寄存器不同，则你必须标记其为一个固定寄存器，根据FIXED_REGISTERS或者设法能够消除它（参见Section 17.10.5 [消除], page 323）。

RETURN_ADDRESS_POINTER_REGNUM [Macro]

返回地址指针寄存器的寄存器编号，其用于访问栈中当前函数的返回地址。在一些机器上，返回地址不在帧指针或栈指针或参数指针的固定偏移处。该寄存器被定义指向栈中的返回地址，并且然后通过ELIMINABLE_REGS转换为帧指针或者栈指针。

不要定义该宏，除非没有其他的方式从栈中获得返回地址。

STATIC_CHAIN_REGNUM [Macro]

STATIC_CHAIN_INCOMING_REGNUM [Macro]

用于传递函数static链指针的寄存器编号。如果寄存器窗口被使用，则寄存器编号由被调用函数所看到是STATIC_CHAIN_INCOMING_REGNUM，而由调用者函数看到的是STATIC_CHAIN_REGNUM。如果这些寄存器是相同的，则不需要定义STATIC_CHAIN_INCOMING_REGNUM。

静态链寄存器不需要为一个固定寄存器。

如果静态链在内存中传递，则这些宏不需要定义；替代的，应该定义下面的两个宏。

STATIC_CHAIN [Macro]

STATIC_CHAIN_INCOMING [Macro]

如果静态链在内存中传递，则这些宏提供了给出它们存储的mem表达式rtx。STATIC_CHAIN和STATIC_CHAIN_INCOMING分别给出了由调用者和被调用函数 所看到的位置。通常前者将在栈指针的一个偏移处并且后者将在帧指针的一个偏移处。

变量stack_pointer_rtx, frame_pointer_rtx和arg_pointer_rtx将在使用这些宏之前被初始化，并可以引用。

如果静态链在寄存器中传递，则应该定义之前的两个宏。

DWARF_FRAME_REGISTERS [Macro]

该宏指定了可以在一个调用帧中被保存的硬件寄存器的最大数。这用于DWARF2异常处理中的size数据结构体。

在GCC3.0之前，该宏需要用来建立一个稳定的异常处理ABI。

如果该宏没有被定义，其缺省为FIRST_PSEUDO_REGISTER。

PRE_GCC3_DWARF_FRAME_REGISTERS [Macro]

该宏类似于DWARF_FRAME_REGISTERS，是为了向后兼容在GCC3.0之前编译的代码而提供。

如果没有定义该宏，其缺省为DWARF_FRAME_REGISTERS。

DWARF_REG_TO_UNWIND_COLUMN (regno) [Macro]

定义该宏，如果target对于dwarf寄存器的表示与对于unwind column的内部表示不相同。给定一个dwarf寄存器，该宏应该返回替代使用的内部unwind column编号。

例子参见PowerPC's SPE target。

DWARF_FRAME_REGNUM (regno) [Macro]

定义该宏，如果target对于用在.eh_frame或者.debug_frame的dwarf寄存器的表示与用在其它调试信息section中的不同。给定一个GCC硬件寄存器编号，该宏应该返回.eh_frame 寄存器编号。缺省为DBX_REGISTER_NUMBER (regno)。

DWARF2_FRAME_REG_OUT (regno, for_eh) [Macro]

定义该宏来影射在调用帧信息中存放的寄存器编号， 其为GCC使用DWARF_FRAME_REGNUM来搜集的应该放在.debug_frame （ for_eh为0 ）和.eh_frame （ for_eh为非0 ）中的寄存器。 缺省为返回regno。

17.10.5 消除帧指针和参数指针

这些是关于消除帧指针和参数指针的。

FRAME_POINTER_REQUIRED [Macro]

一个C表达式，其为非0，如果函数必须具有并且使用帧指针。 该表达式在重载过程被求值。 如果值为非零，则函数将具有一个帧指针。

原则上表达式可以检查当前函数并根据事实来决定，但在大多数机器上， 常量0或者1就足够了。使用0当机器允许生成的代码不具有帧指针， 这将节省一些时间或空间。使用1当避免帧指针不会带来好处。

在一些情况下，编译器不知道如何生成没有帧指针的有效的代码。 编译器识别到哪些情况，并自动的给函数一个帧指针， 而不管FRAME_POINTER_REQUIRED如何。你不需要为此担心。

在不需要帧指针的函数中，帧指针寄存器可以被分配用于普通用法， 除非你标记其为一个固定寄存器。更多信息，参见FIXED_REGISTERS。

INITIAL_FRAME_POINTER_OFFSET (depth-var) [Macro]

一条C语句，用来紧接着函数序言之后， 将帧指针和栈指针值的差存储在depth-var中。 该值应该通过像 get_frame_size ()这样的结果信息以及寄存器表 regs_ever_live和 call_used_regs中被计算。

如果ELIMINABLE_REGS被定义，则该宏将不被使用并不需要被定义。 否则，其必须被定义，即使FRAME_POINTER_REQUIRED被定义为总是为真； 这这种情况下，你可以设置depth-var为任何值。

ELIMINABLE_REGS [Macro]

如果被定义，则该宏指定了一个寄存器双对的表，用于消除不需要的指向栈帧的寄存器。 如果没有被定义，则编译器唯一尝试去做的消除是将对帧指针的引用替换为对栈指针的引用。

该宏的定义为一个结构体初始化列表，每个指定了最初的和替换后的寄存器。

在一些机器上，参数指针的位置直到编译结束时才知道。这种情况下， 一个单独的硬件寄存器必须用于参数指针。 该寄存器可以通过替换为帧指针或者参数指针来消除，这取决于帧指针是否已经被消除。

这种情况下，你可能会指定：

```
#define ELIMINABLE_REGS \
{{ARG_POINTER_REGNUM, STACK_POINTER_REGNUM}, \
 {ARG_POINTER_REGNUM, FRAME_POINTER_REGNUM}, \
 {FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM}}
```

注意首先指定的是使用栈指针来消除参数指针，因为这是首选的消除方式。

CAN_ELIMINATE (from-reg, to-reg) [Macro]

一个C表达式，其返回非0， 如果编译器被允许尝试使用寄存器编号to-reg来替换寄存器编号from-reg。 该宏只在ELIMINABLE_REGS被定义时才需要定义，并且通常为常量1， 因为大多数情况下编译器已经知道是否不该进行寄存器消除。

INITIAL_ELIMINATION_OFFSET (from-reg, to-reg, offset-var) [Macro]

该宏类似于INITIAL_FRAME_POINTER_OFFSET。 其指定了被指定的寄存器双对的初始差。该宏必须被定义， 如果ELIMINABLE_REGS被定义。

17.10.6 在栈上传递函数参数

该节的宏控制如何在栈上传递参数。关于控制在寄存器中传递特定参数的其它宏， 参见后续的章节。

bool TARGET_PROMOTE_PROTOTYPES (tree fntype) [Target Hook]

该target钩子返回true，如果在函数原型中声明的一个参数， 为整型的并且比int小，应该作为int来传递。 除了能够避免一些不匹配的错误以外，其还能在特定机器上生成更好的代码。缺省为不提升原型。

PUSH_ARGS [Macro]

一个C表达式。如果非0，则将使用push insn来传递输出参数。 如果target机器不具有push指令，则设置其为0。这将指示GCC使用替代的策略： 分配整个参数块然后将参数存进去。当PUSH_ARGS为非0时， PUSH_ROUNDING也必须被定义。

PUSH_ARGS_REVERSED [Macro]

一个C表达式。如果非0，则函数参数将按照从最后一个到第一个的顺序来求值， 而不是从第一个到最后一个。如果该宏没被定义，其缺省为PUSH_ARGS， 在栈和args按照相反的顺序进行增长的target上，否则为0。

PUSH_ROUNDING (npushed) [Macro]

一个C表达式，其为当一个指令试图压入npushed个字节时，实际压入栈中的字节数。

在一些机器上，定义

```
#define PUSH_ROUNDING(BYTES) (BYTES)
```

便可以满足。但是在其它机器上，指令压入一个字节时， 而为了保持对齐实际压入了两个字节。则定义应该为

```
#define PUSH_ROUNDING(BYTES) (((BYTES) + 1) & ~1)
```

ACCUMULATE_OUTGOING_ARGS [Macro]

一个C表达式。如果非0，则为输出参数中将被计算并放进变量current_function_outgoing_args_size所需要的空间最大数目。 对于每个调用，将不会有空间被压入栈中；替代的，函数序言应该增加栈帧的大小。

同时设置PUSH_ARGS和ACCUMULATE_OUTGOING_ARGS是不合适的。

REG_PARM_STACK_SPACE (fndecl) [Macro]

定义该宏，如果函数应该假设参数的栈空间已经被分配， 即使它们的值是在寄存器中被传递的。

该宏的值是一个fndecl表示的函数在寄存器中传递的参数的保留空间的大小， 字节为单位，其可以为0如果GCC在调用一个库函数。

该空间可以被调用者分配，或者为机器相关的栈帧的一部分： 这由OUTGOING_REG_PARM_STACK_SPACE决定。

OUTGOING_REG_PARM_STACK_SPACE (fntype) [Macro]

定义该宏为一个非0值，如果分配在寄存器中传递的参数的保留空间，是由调用者负责。

如果ACCUMULATE_OUTGOING_ARGS被定义， 则该宏控制这些参数的空间是否算在current_function_outgoing_args_size中。

STACK_PARMS_IN_REG_PARM_AREA [Macro]

定义该宏，如果REG_PARM_STACK_SPACE被定义， 但是栈参数不跳过其所指定的区域。

通常，当一个参数没有在寄存器中传递时， 其被放在REG_PARM_STACK_SPACE区域之外的栈上。 定义该宏来抑制这种行为并使得在栈上传递的参数按照它的自然位置。

RETURN_POPS_ARGS (fundecl, funtype, stack-size) [Macro]

一个C表达式，指示函数在返回时所弹出的它自己的参数的字节个数， 或者为0如果函数不弹出参数并且调用者必须在函数返回后弹出它们。

fundecl为一个C变量，其值为一个树节点，描述了被讨论的函数。 通常其为一个FUNCTION_DECL类型的节点，描述了函数的声明。 你可以从中获得函数的DECL_ATTRIBUTES。

funtype为一个C变量，其值为一个树节点，描述了被讨论的函数。 通常其为一个FUNCTION_TYPE类型的节点，描述了函数的数据类型。 从中可以获得函数值和参数的数据类型（如果知道的话）。

当正在被考虑的函数是一个库函数调用时， fundecl将包含一个库函数的标志符节点。这样，如果你需要区别不同的库函数， 则可以通过它们的名字进行。注意该上下文中的“库函数”是指用于执行算术的库函数， 其名字在编译器中是已知的并且在被编译的C代码中没有被提到。

stack-size为在栈上传递的参数的字节数。如果是传递的字节数是可变的， 则为0，并且参数弹出将总是为被调用函数的责任。

在VAX上，所有的函数总是弹出它们的参数，所以该宏的定义为stack-size。 在68000上，使用标准的调用约定，没有函数弹出它们的参数， 所以对于这种情况该宏的值总是0。但是也可以使用可选的调用约定， 这种情况下函数接受固定数目参数的将弹出它们， 其它的函数（例如printf）则不弹出（由调用者来弹出所有参数）。 当使用这种约定时，funtype被检查用来确定一个函数是否接受了固定数目的参数。

CALL_POPS_ARGS (cum) [Macro]

一个C表达式，指示一个调用序列从栈中弹出的字节数目。 其被加到RETURN_POPS_ARGS的值中，当编译一个函数调用时。

cum为一个变量，为被调用函数的所有参数的累积。

在特定的体系结构上，例如SH5，一个调用蹦床被用于弹出栈上特定的寄存器， 根据被传递给函数的参数。因为这是调用方的属性，而不是被调用函数的， 所以RETURN_POPS_ARGS不太适合。

17.10.7 在寄存器中传递参数

这节描述了让你控制不同类型的参数如何在寄存器中传递， 或者它们如何被安排在栈中的宏。

FUNCTION_ARG (cum, mode, type, named) [Macro]

— Macro: FUNCTION_ARG (cum, mode, type, named)

一个C表达式，控制函数的参数是否在寄存器中传递，以及在那个寄存器中传递。

参数为cum，其总结了所有之前的参数；mode，参数的机器模式； type，参数的数据类型，作为一个树节点或者0如果不知道（这发生于C支持库的 函数）；named，其为1对于普通参数，0对于无名参数， 对应于在被调用函数原型中的`...'。type可以为一个不完全类型，如果之前产生了语法错误。

表达式的值通常为一个在其中传递参数的硬件寄存器的reg RTX，或者0， 在栈上传递参数。

对于像VAX和68000的机器，其通常所有参数被压栈，则定义为0就行。

表达式的值也可以为一个parallel RTX。这用于当参数在多个位置传递的时候。 parallel的机器模式应该为整个参数的机器模式。 parallel保存了任意数目的expr_list对； 每一个描述了该部分参数在哪里传递， reg RTX的机器模式指示了该部分的参数有多大。 expr_list的第二个操作数为一个const_int， 其给出了该部分起始处与整个参数的偏移字节数。作为一个特例， parallel RTX中的第一个expr_list的第一个操作数可以为0。 这意味着整个参数也在栈中存储。

该宏最后一次被调用的时候，MODE == VOIDmode， 并且结果被传递给call或者call_value指令模式，分别作为其操作数2和3。

通常使ISO库`stdarg.h'在一些参数在寄存器中传递的机器上工作的方法， 是使无名参数在栈上传递。这通过使FUNCTION_ARG当named为0时返回0来实现。

你可以在该宏的定义中，使用钩子targetm.calls.must_pass_in_stack来确定该 参数是否为一个必须在栈中传递的类型。如果REG_PARM_STACK_SPACE没有定义并且 FUNCTION_ARG对于这样的参数返回非0，则编译器会abort。 如果REG_PARM_STACK_SPACE被定义，则参数会在栈中计算并且然后加载到寄存器中。

bool TARGET_MUST_PASS_IN_STACK (enum machine_mode mode, tree type) [Target Hook]

该target钩子应该返回true，如果我们不应该只在寄存器中传递type。 文件`expr.h'中有一个定义，其通常是合适的，更多的文档请参考`expr.h'。

FUNCTION_INCOMING_ARG (cum, mode, type, named) [Macro]

定义该宏，如果target机器具有“寄存器窗口” ， 这样函数看到的参数寄存器没有必要与调用者传递参数的寄存器相同。

对于这样的机器，FUNCTION_ARG计算调用者传递值的寄存器， FUNCTION_INCOMING_ARG应该按照类似的方式定义， 来告诉被调用的函数参数在哪里到来。

如果FUNCTION_INCOMING_ARG没有定义，则FUNCTION_ARG具有这两种用途。

int TARGET_ARG_PARTIAL_BYTES (CUMULATIVE_ARGS *cum, enum machine_mode mode, tree type, bool named) [Target Hook]

该target钩子返回在参数的起始处必须被放入寄存器的字节数。 值必须为0对于参数全部放在寄存器中或者全部压入栈中。

一些机器上，特定的参数必须部分在寄存器中传递并且部分在内存中传递。 在这些机器上，通常参数的起始一些字在寄存器中传递，其余的在栈上。 如果一个多字的参数（ double或者结构体）跨越了边界， 则其起始的一些字必须在寄存器中传递并且剩余的被压栈。 该宏告诉编译器这种情况什么时候发生，以及多少字节应该在寄存器中。

FUNCTION_ARG对于这些参数应该返回第一个寄存器，被调用者用于该参数； 通常FUNCTION_INCOMING_ARG用于被调用的函数。

bool TARGET_PASS_BY_REFERENCE (CUMULATIVE_ARGS *cum, enum machine_mode mode, tree type, bool named) [Target Hook]

该target钩子应该返回true， 如果cum所指示的位置的参数应该按照引用的方式来传递。

如果钩子返回真，则参数的副本在内存中产生并且指向参数的指针被替代参数本身来传递。指针按照传递该类型的指针的方式来传递。

bool TARGET_CALLEE_COPIES (CUMULATIVE_ARGS *cum, enum machine_mode mode, tree type, bool named) [Target Hook]

由该钩子的参数所描述的函数的参数已知为通过引用来传递的。钩子应该返回真， 如果函数参数应该由被调用者复制，而不是调用者。

对于任何该钩子返回真的参数，如果其可以被确定参数没有被修改，则不需要产生副本。
该钩子的缺省版本总是返回假。

CUMULATIVE_ARGS [Macro]
一个C类型，用来声明一个变量， 被用作FUNCTION_ARG的第一个参数以及其它相关的值。对于一些target机器， 类型int可以满足并且目前可以保持参数的字节数。

不需要在CUMULATIVE_ARGS中记录任何已经在栈中传递的参数的信息。 编译器有其它变量来记录。对于所有参数在栈上传递的target机器， 不需要在CUMULATIVE_ARGS中保存任何事物；然而， 数据结构体必须存在并且不能为空，因此可以使用int。

OVERRIDE_ABI_FORMAT (fndecl) [Macro]
If defined, this macro is called before generating any code for a function, but after the cfun descriptor for the function has been created. The back end may use this macro to update cfun to reflect an ABI other than that which would normally be used by default. If the compiler is generating code for a compiler-generated function, fndecl may be NULL.

INIT_CUMULATIVE_ARGS (cum, fntype, libname, fndecl, n_named_args) [Macro]
一条C语句（没有分号），用于初始化变量cum，在参数列表的起始处。 变量具有类型CUMULATIVE_ARGS。fntype的值为树节点， 为将要接受参数的函数的数据类型，或者为0如果参数是传给编译器支持库的函数。 对于直接调用，没有libcall，fndecl包含了被编译的函数。fndecl在当INIT_CUMULATIVE_ARGS被用于查找被编译的函数的参数时， 也被设置。n_named_args被设为命名参数的个数，包括一个结构体返回地址， 如果其作为参数被传递。当处理流入参数时，n_named_args被设为-1。

当处理对编译器支持库的函数的调用时， libname指示了为哪一个函数。 其为一个symbol_ref rtx，包含了函数的名字，作为字符串。libname为0， 当一个普通C函数被处理。因此，每次该宏被调用时， 或者libname或者fntype为非 0，但不会同时非0。

INIT_CUMULATIVE_LIBCALL_ARGS (cum, mode, libname) [Macro]
类似于INIT_CUMULATIVE_ARGS，但只用于流出的libcall， 其接受一个MODE参数而不是fntype。如果该宏没有定义， 则使用INIT_CUMULATIVE_ARGS (cum, NULL_RTX, libname, 0)来替代。

INIT_CUMULATIVE_INCOMING_ARGS (cum, fntype, libname) [Macro]
类似于INIT_CUMULATIVE_ARGS，但会覆盖其，用于查找被编译的函数的参数。 如果该宏没有被定义，则使用INIT_CUMULATIVE_ARGS来替代。

传递给libname的值总是为0，因为库函数具有特定的调用约定，从来不被GCC编译。 参数libname的存在是为了与INIT_CUMULATIVE_ARGS对称。

FUNCTION_ARG_ADVANCE (cum, mode, type, named) [Macro]
一条C语句（没有分号），来更新总结变量cum来在参数列表中前进一个参数。 值mode, type和named描述了那个参数。一旦执行后， 变量cum便适合分析随后的参数。

该宏不需要做任何事情，如果要询问的参数是在栈中传递的。 编译器知道如何追踪用于参数的栈空间，不需要任何特殊帮助。

FUNCTION_ARG_OFFSET (mode, type) [Macro]
If defined, a C expression that is the number of bytes to add to the offset of the argument passed in memory. This is needed for the SPU, which passes char and short arguments in the preferred slot that is in the middle of the quad word instead of starting at the top.

FUNCTION_ARG_PADDING (mode, type) [Macro]

如果被定义，则为一个C表达式，其确定是否使用额外的空间来填补参数， 以及按照什么方向。值应该为类型enum direction：或者upward， 向上填补参数，downward向下，或者none不进行填补。

填补的数目总是刚足够达到下一个FUNCTION_ARG_BOUNDARY的倍数； 该宏不进行控制。

该宏具有一个缺省定义，其对大多数系统是对的。对于小端机器，缺省为向上填补。 对于大端机器，缺省为如果参数的大小比int短则向下填补，否则向上。

PAD_VARARGS_DOWN [Macro]

如果定义，则为一个C表达式，其确定va_arg的缺省实现是否会尝试向下填补， 在读取下一个参数之前，如果那个参数比PARM_BOUNDARY所控制的对齐空间要小。 如果该宏没有定义，则所有这样的参数都被向下填补， 如果BYTES_BIG_ENDIAN为真。

BLOCK_REG_PADDING (mode, type, first) [Macro]

指定了寄存器和内存间移动的块的最后一个元素的填补。first为非0， 如果这是唯一的元素。定义该宏，允许更好的处理在大端机器上寄存器函数参数， 不使用PARALLEL rtl。特别的，MUST_PASS_IN_STACK不需要测试填充和寄存器中的类型的机器模式， 因为在寄存器中不在有“错误的”部分；例如， 一个三字节的聚合类型可能在寄存器的高部传递，如果需要的话。

FUNCTION_ARG_BOUNDARY (mode, type) [Macro]

如果定义，为一个C表达式，其给出了指定的mode和type的参数的对齐边界位数。 如果没有定义，则PARM_BOUNDARY用于所有参数。

FUNCTION_ARG_REGNO_P (regno) [Macro]

一个C表达式，其为非0，如果regno为硬件寄存器的编号， 函数参数有时在其中传递。这不包括隐式参数，像静态链和结构体值的地址。 在许多机器上，没有寄存器可以用于此目的，因为所有函数参数都被压到栈上。

bool TARGET_SPLIT_COMPLEX_ARG (tree type) [Target Hook]

该钩子应该返回真，如果参数type作为两个标量参数传递。缺省的， GCC将尝试将复数参数打包成target的字大小。 一些ABI要求复数参数要被拆分开并且作为单独的部分对待。例如，在AIX64上， 复数浮点应该在一对浮点寄存器中传递，即使复数浮点可以适合一个64位的浮点寄存器。

该钩子的缺省值为NULL，其被最为假来对待。

tree TARGET_BUILD_BUILTIN_VA_LIST (void) [Target Hook]

该钩子返回一个target的va_list的类型节点。缺省版本返回void*。

tree TARGET_FN_ABI_VA_LIST (tree fndecl) [Target Hook]

This hook returns the va_list type of the calling convention specified by fndecl. The default version of this hook returns va_list_type_node.

tree TARGET_CANONICAL_VA_LIST_TYPE (tree type) [Target Hook]

This hook returns the va_list type of the calling convention specified by the type of type. If type is not a valid va_list type, it returns NULL_TREE.

tree TARGET_GIMPLIFY_VA_ARG_EXPR (tree valist, tree type, tree *pre_p, tree *post_p) [Target Hook]

该钩子执行target特定的VA_ARG_EXPR的gimplification。 前两个参数对应于va_arg的参数；后两个作为gimplify.c:gimplify_expr。

bool TARGET_VALID_POINTER_MODE (enum machine_mode mode) [Target Hook]
定义该钩子返回非0，如果port可以处理具有机器模式mode的指针。 缺省版本对于ptr_mode和Pmode都返回真。

bool TARGET_SCALAR_MODE_SUPPORTED_P (enum machine_mode mode) [Target Hook]
定义该钩子来返回非0，如果port准备好了处理涉及标量机器模式mode的insn。 对于被考虑支持的一个标量机器模式，所有的基本算术和比较都必须能工作。

缺省版本返回真，对于任何要求处理基本C类型（被port定义）的机器模式。 包括在`optabs.c'中的代码支持的双字算术。

bool TARGET_VECTOR_MODE_SUPPORTED_P (enum machine_mode mode) [Target Hook]
定义该钩子来返回非0，如果port准备好了处理涉及向量模式mode的insn。 最起码，其必须有该机器模式的move指令模式。

17.10.8 标量函数值如何被返回

这节讨论了控制返回标量值的宏，值可以放在寄存器中。

rtx TARGET_FUNCTION_VALUE (tree ret_type, tree fn_decl_or_type, bool outgoing) [Target Hook]
定义该宏返回一个RTX，用来表示函数返回或者接受数据类型为ret_type的值的 地方。 ret_type为一个树节点，表示一个数据类型。 fn_decl_or_type为一个树节点，表示被调用的函数的FUNCTION_DECL或者 FUNCTION_TYPE。如果outgoing为假， 则钩子应该计算调用者将要看到返回值的寄存器。否则，钩子应该返回一个RTX， 其表示函数在哪儿返回一个值。

在许多机器上，只有TYPE_MODE (ret_type)是相关的。（实际上，在大多数机器上，标量值不管机器模式如何，都在同一地方返回。）表达式的值通常为一个硬件寄存器的reg RTX，为存放返回值的地方。 该值还可以为一个parallel RTX，如果返回值在多个地方。 对于parallel形式的解释，参见FUNCTION_ARG。 m68k port使用了这种parallel类型来返回指针， 在`%a0'(规范化位置) 和`%d0'中。

如果TARGET_PROMOTE_FUNCTION_RETURN返回真，如果valtype为一个标量类型， 则你必须应用在PROMOTE_MODE中指定的相同的提升规则。

如果确切的知道被调用的函数，则func为它的树节点（FUNCTION_DECL）； 否则，func为一个空指针。这是的可以使用不同的值返回约定， 对于所有调用都知道的特定的函数。

一些target机器具有“寄存器窗口”， 这使得函数返回它的值所用的寄存器与调用者看到值所在的寄存器不同。 对于这样的机器，你应该返回不同的RTX，根据outgoing。

TARGET_FUNCTION_VALUE不用于返回聚合数据类型的值， 因为这些通过其它方式返回。参见下面的TARGET_STRUCT_VALUE_RTX以及相关的宏。

FUNCTION_VALUE (valtype, func) [Macro]
该宏已经不赞成被使用。对于新的target，使用TARGET_FUNCTION_VALUE来替代。

FUNCTION_OUTGOING_VALUE (valtype, func) [Macro]
该宏已经不赞成被使用。对于新的target，使用TARGET_FUNCTION_VALUE来替代。

LIBCALL_VALUE (mode) [Macro]
一个C表达式，用来创建一个RTX，表示库函数返回模式mode的值的地方。 如果确切的知道被调用的函数，则func为它的树节点（FUNCTION_DECL）； 否则func为一个空指针。这使得可以使用不同的值返回约定， 对于所有调用都知道的特定的函数。

注意“库函数”在该上下文中意味着是编译器支持的程序，用于执行算术运算， 其名字由编译器知道并且没有在被编译的C代码中提到。

LIBRARY_VALUE的定义不需要考虑聚合数据类型，因为没有库函数返回这样的类型。

FUNCTION_VALUE_REGNO_P (regno) [Macro]

一个C表达式，为非0，如果regno为硬件寄存器的编号， 且被调用的函数可以通过它来返回值。

用于返回值的寄存器，并且其被限制为一个寄存器对（比如， 对于类型double的值）的第二个时，不需要被该宏识别。 所以对于大多数机器，该定义可以为：

```
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 0)
```

如果机器具有寄存器窗口，使得调用者和被调用函数使用不同的寄存器来返回值， 则该宏应该只是识别调用者的寄存器编号。

TARGET_ENUM_VA_LIST (idx, pname, ptype) [Macro]

This target macro is used in function c_common_nodes_and_builtins to iterate through the target specific builtin types for va_list. The variable idx is used as iterator. pname has to be a pointer to a const char * and ptype a pointer to a tree typed variable. The arguments pname and ptype are used to store the result of this macro and are set to the name of the va_list builtin type and its internal type. If the return value of this macro is zero, then there is no more element. Otherwise the IDX should be increased for the next call of this macro to iterate through all types.

APPLY_RESULT_SIZE [Macro]

定义该宏，如果`untyped_call'和`untyped_return'需要比 FUNCTION_VALUE_REGNO_P用于保存和恢复一个任意返回地址所用的更多的空间。

bool TARGET_RETURN_IN_MSB (tree type) [Target Hook]

该钩子应该返回真，如果类型type的值按照在寄存器中的最高有效位返回（换句话说，如果他们在最低有效位进行被padded）。 你可以假设该type在寄存器中被返回；调用者被要求进行该检查。

注意TARGET_FUNCTION_VALUE提供的寄存器必须能够保存完整的返回值。 例如，如果一个1，2或者3字节的结构体被返回，按照4字节寄存器中的最高有效位的方式， 则TARGET_FUNCTION_VALUE应该提供一个SImode rtx。

17.10.9 如何返回大的值

当函数值的机器模式为BLKmode（并且在一些其它情况下）， 值不根据TARGET_FUNCTION_VALUE来返回（参见 Section 17.10.8 [标量返回], page 329）。 替代的，调用者传递内存块的地址。 该地址被称为结构体地址（structure value address）。

这一节描述了如何控制在内存中返回结构体值。

bool TARGET_RETURN_IN_MEMORY (tree type, tree fntype) [Target Hook]

该target钩子应该返回一个非零值，来指明在内存中返回函数值， 正如返回大的结构体的方式。这里type为值的数据类型， fntype为函数的类型，或者NULL，如果是libcall。

注意模式BLKmode的值必须被该函数显示的处理。 而且，选项`-fpcc-struct-return'将会其作用， 而不管该宏如何定义。在大多数系统上，可能会没有定义该钩子； 这将使用一个缺省定义，其值为常数1对于BLKmode值，其它的为0。

不要使用该宏来指示结构体和联合体应该总是在内存中返回。 你应该使用DEFAULT_PCC_STRUCT_RETURN来做这件事情。

DEFAULT_PCC_STRUCT_RETURN [Macro]

定义该宏为1，如果所有的结构体和联合体返回值必须在内存中。 由于这将使得代码变慢，所以应该只有需要与其它编译器或者ABI兼容时才定义。 如果你定义了该宏为0， 则对于结构体和联合体返回值的约定则由TARGET_RETURN_IN_MEMORY target钩子来决定。

如果没有定义，将缺省为1。

rtx TARGET_STRUCT_VALUE_RTX (tree fndecl, int incoming) [Target Hook]

该target钩子应该返回结构体值的地址位置（通常为mem或者reg），或者0，如果地址作为“不可视”的第一个参数传递。 注意fndecl可以为NULL，对于libcall。 你不需要定义该target钩子，如果地址总是作为“不可视”的第一个参数传递。

在一些体系结构上， 被调用函数寻找结构体值地址的地方与调用者放入的地方不相同。 这可能是由于寄存器窗口，或者函数序言将其移动到一个不同的地方。 incoming为1或者2， 当地址在被调用函数的上下文中需要， 为0如果在调用者的上下文中需要。

如果incoming为非0并且地址是在栈中找到， 则返回一个mem，其引用帧指针。 如果incoming为2， 则结果用于在函数的起始处获取结构体值的地址。 如果你需要输出调整代码，你应该在这里进行。

PCC_STATIC_STRUCT_RETURN [Macro]

定义该宏，如果在target机器上的通常的系统约定， 对于返回结构体和联合体，为被调用函数返回包含该值的静态变量的地址。

不要定义该宏，如果通常的系统约定为调用者将地址传给子程序。

该宏具有`-fpcc-struct-return'模式下的效果， 但是当你使用`-freg-struct-return'模式时，其将不做任何事。

17.10.10 调用者保存的寄存器分配

如果你使用这种功能，GCC可以将寄存器保存在函数调用附近。 这使得可以使用调用破坏的（call-clobbered）寄存器来存放必须活跃于调用之间的变量。

CALLER_SAVE_PROFITABLE (refs, calls) [Macro]

一个C表达式来确定是否值得考虑将一个伪寄存器放在一个调用破坏的硬件寄存器中， 并在每个函数调用的附近进行保存和恢复。表达式应该为1，当值得去做，否则为0。

如果没有定义该宏，缺省值将被使用， 其在大多数机器上都是好的：4 * calls < refs。

HARD_REGNO_CALLER_SAVE_MODE (regno, nregs) [Macro]

一个C表达式指定了将伪寄存器nregs保存在调用破坏的硬件寄存器regno中， 需要哪种机器模式。如果regno不适合调用者保存，则应该返回VOIDmode。 对于大多数机器，该宏不需要被定义，因为GCC将选择最小的合适的机器模式。

17.10.11 函数入口和出口

这一章描述了输出函数入口（prologue）和出口（epilogue）代码的宏。

void TARGET_ASM_FUNCTION_PROLOGUE (FILE *file, HOST_WIDE_INT size) [Target Hook]

如果被定义，则为一个函数，其为函数的入口输出汇编代码。序言负责设置栈帧， 初始化帧指针寄存器，保存必须被保存的机器，并分配保存局部变量所需要的额外字节数 size。size为一个整数。 file为汇编代码应该被输出到的一个stdio流。

函数起始处的标号不需要被该宏输出。其已经在该宏运行时被输出了。

要确定哪些寄存器要保存，该宏可以引用数组regs_ever_live： 元素r为非零，如果硬件寄存器r在函数某处被使用。 这意味着倘若其不是调用使用的（call-used）寄存器， 则函数序言应该保存寄存器r。 (同样TARGET_ASM_FUNCTION_EPILOGUE也必须使用regs_ever_live。)

在具有“寄存器窗口”的机器上，函数入口代码不在栈中保存位于窗口中的寄存器， 即使它们认为被函数调用保留；替代的，如果在函数中使用了任何非调用使用的寄存器， 其使用适当的步骤来“压入”寄存器栈中。

在一些机器上，函数可以有帧指针，也可以没有，则函数入口代码必须相应的有所不同； 如果需要则其必须建立帧指针，否则不建立。要确定是否想要帧指针， 宏可以引用变量frame_pointer_needed。在运行时，如果函数需要帧指针， 则变量的值将被设为1。参见Section 17.10.5 [消除], page 323。

函数入口代码负责分配函数需要的任何栈空间。该栈空间包括下面列出的域。 大多数情况下，这些域按照列出的顺序被分配，最后列出的域最靠近栈顶（如果STACK_GROWS_DOWNWARD被定义，则为最低地址，如果没有定义， 则为最高地址）。你可以为一个机器使用不同的顺序， 如果这样做更加方便或者出于兼容的原因。除了由于标准或者调试器的要求之外， 没有理由GCC使用的栈布局需要适合机器上的其它编译器所使用的。

void TARGET_ASM_FUNCTION_END_PROLOGUE (FILE *file) [Target Hook]

如果被定义，则为一个函数，在序言的结尾处输出汇编代码。 这应该被用于当函数序言作为RTL输出时，并且你需要输出一些额外的汇编语言。 参见“序言指令模式”

void TARGET_ASM_FUNCTION_BEGIN_EPILOGUE (FILE *file) [Target Hook]

如果被定义，则为一个函数，在尾声的起始处输出汇编代码。 这应该被用于当函数尾声作为RTL输出时，并且你需要输出一些额外的汇编语言。 参见“尾声指令模式”

void TARGET_ASM_FUNCTION_EPILOGUE (FILE *file, HOST_WIDE_INT size) [Target Hook]

如果被定义，则为一个函数，其为函数的退出输出汇编代码。 尾声负责恢复保存的寄存器和栈指针为函数被调用时的值，并将控制返回给调用者。 该宏接受跟TARGET_ASM_FUNCTION_PROLOGUE相同的参数， 并且要恢复的寄存器按照相同的方式通过regs_ever_live和 CALL_USED_REGISTERS来确定。

在一些机器上，有一个单独的指令，可以做从函数中返回的所有工作。 在这些机器上，给出那个名为`return'的指令， 并且不要定义宏TARGET_ASM_FUNCTION_EPILOGUE。

如果你想使用TARGET_ASM_FUNCTION_EPILOGUE， 则不要定义名为`return'的指令模式。 如果你想target切换使用return指令或者尾声，则定义一个`return'指令模式， 带有一个有效性条件用来测试target的适当的切换。 如果`return'指令模式的有效性条件为假，则使用尾声。

在一些机器上，函数可以有帧指针，也可以没有，则函数的退出代码必须相应有所不同。 有时这两种情况的代码会完全不同。要确定是否需要帧指针， 该宏可以引用变量frame_pointer_needed。当编译一个需要帧指针的函数时， 变量的值将为1。

通常，TARGET_ASM_FUNCTION_PROLOGUE和TARGET_ASM_FUNCTION_EPILOGUE 必须单独处理叶子函数。对于这样的函数，C变量current_function_is_leaf为 非零。参见Section 17.7.4 [叶子函数], page 306。

在一些机器上，一些函数在退出时弹出它们的参数，而其它的则将它们留给调用者来完成。例如，68020当给定`-mrtd'时会弹出具有固定参数个数的函数的参数。

你对宏的定义决定了哪些函数弹出它们的自己的参数。 TARGET_ASM_FUNCTION_EPILOGUE需要知道这些。 称作current_function_pops_args的变量为函数应该弹出的参数的字节个数。 参见Section 17.10.8 [标量返回], page 329。

- current_function_pretend_args_size个字节大小的未初始化空间位于栈中第一个 参数的下面。（这可能不是被分配的栈域的最起始处，如果调用序列在压入栈参数时还压入 了其它东西。通常，在这样的机器上，并没有压入其它东西，因为函数序言本身来做所有的 压栈操作）该域用于参数可以部分在寄存器中传递，部分在内存中传递的机器上， 以及支持<stdarg.h>的特性的情况。
- 有一块内存用于保存函数使用的特定的寄存器。该区域的大小， 可能还包括作为返回地址和指向之前栈帧的指针的一些空间， 其为机器特定的并且通常取决于函数中已经使用了哪些寄存器。 具有寄存器窗口的机器通常不需要这样的存储区域。
- 一块至少size个字节的区域，可能舍人到一个分配边界的大小， 来保存函数的局部变量。在一些机器上，该域和保存域可以按照相反的顺序出现， 使得保存域接近于栈顶。
- 可选的，当ACCUMULATE_OUTGOING_ARGS被定义时，还有一块current_function_outgoing_args_size字节大小的区域用于函数的传出 的参数列表。参见 Section 17.10.6 [栈参数], page 324。

EXIT_IGNORE_STACK [Macro]

定义该宏为一个C表达式，其为非0，如果返回指令或者函数尾声忽略栈指针的值； 换句话说，如果在从函数中返回前，删除调整栈指针的指令是安全的。缺省为0。

注意该宏的值只于维护帧指针的函数相关。 在没有帧指针的函数中删除最后的栈调整是绝对不安全的，并且编译器知道这种情况， 而不管EXIT_IGNORE_STACK定义如何。

EPILOGUE_USES (regno) [Macro]

定义该宏为一个C表达式，其为非0，对于用于尾声或者`return'指令模式的寄存器。 栈和帧指针寄存器已经被假设需要使用。

EH_USES (regno) [Macro]

定义该宏为一个C表达式，其为非0，对于用于异常处理机制的寄存器， 所以其应该被考虑为在一个异常边的入口上是活跃的。

DELAY_SLOTS_FOR_EPILOGUE [Macro]

定义该宏，如果函数尾声包含延迟槽，并且函数其余的指令可以被移动过去。 该定义应该为一个C表达式，其值为一个整数表示有多少个延迟槽。

ELIGIBLE_FOR_EPILOGUE_DELAY (insn, n) [Macro]

一个C表达式，返回1，如果insn可以放在尾声中的延迟槽编号n中。

参数n为一个整数，其标识了目前被考虑的延迟槽（由于不同的延迟槽可以具有不同 的适任规则）。其从不为负，并且总是小于尾声延迟槽的总数（DELAY_SLOTS_FOR_EPILOGUE的返回值）。 如果你为给定的延迟槽拒绝了一个特定的insn，原则上，其可以被后续的延迟槽重新考虑。 而且，其它insn还可以（至少原则上）被目前为止还没有被填充的延迟槽考虑。

被接受填充尾声延迟槽的insn被放在一个RTL链表中，使用insn_list对象， 并存储在变量current_function_epilogue_delay_list中。 第一个延迟槽的insn位于链表中的第一个。你对宏TARGET_ASM_FUNCTION_EPILOGUE 的定义应该通过输出该链表的insn来填充延迟槽，通常是调用final_scan_insn。

你不需要定义该宏，如果你没有定义DELAY_SLOTS_FOR_EPILOGUE。

void TARGET_ASM_OUTPUT_MI_THUNK (FILE *file, tree thunk_fndecl, [Target Hook]
HOST_WIDE_INT delta, HOST_WIDE_INT vcall_offset, tree function)

一个函数，输出一个thunk函数的汇编代码，用于实现具有多继承的C++虚函数调用。 thunk作为一个虚函数的封装，用来调整隐式对象参数，在将控制移交给实函数之前。

首选，输出代码来增加整数delta为包含传递进来的第一个参数的为。 假设该参数包含一个指针，并用于在C++中传递this指针。 这是在函数序言之前的参数，例如在sparc上为`%o0'。

然后，如果vcall_offset非0，则在增加delta之后应该进行额外的调整。 特别是，如果p为被调整的指针，则应该进行如下的调整：

```
p += (*((ptrdiff_t **)p))[vcall_offset/sizeof(ptrdiff_t)]
```

加法之后，输出代码跳转到function，其为FUNCTION_DECL。 这是一个直接跳转，而不是调用，并且不触及返回地址。因此从FUNCTION中返回时， 将返回到调用当前`thunk'的地方。

其效果就好像是函数被直接调用，并使用调整后的第一个参数。 该宏负责输出thunk函数的所有代码；TARGET_ASM_FUNCTION_PROLOGUE和TARGET_ASM_FUNCTION_EPILOGUE不被调用。

thunk_fndecl是冗余的。（delta和function已经从中被抽取出来。）其可能在一些target上有用，也很可能没用。

如果你没有定义该宏，则C++前端的target无关代码将会生成一个不太有效的重量级的 thunk，其调用function而不是直接跳转过去。普通的方法不支持varargs。

bool TARGET_ASM_CAN_OUTPUT_MI_THUNK (tree thunk_fndecl, HOST_WIDE_INT delta, HOST_WIDE_INT vcall_offset, tree function) [Target Hook]

一个函数，返回真，如果TARGET_ASM_OUTPUT_MI_THUNK应该能够为其传递的参数所指定的 thunk函数输出汇编代码，否则为假。在后一种情况下，C++前端将会使用普通的方式，并具有之前提到的限制。

17.10.12 为profiling生成代码

这些宏将帮助你为profiling生成代码。

FUNCTION_PROFILER (file, labelno) [Macro]

一条C语句或者复合语句，来输出到file中一些汇编代码， 来调用profiling子程序mcount。

关于mcount期望如何被调用的细节，由你的操作系统环境来决定， 而不是GCC。要弄清楚它们，可以编译一个小程序， 使用系统安装的C编译进行profiling，并查看生成的汇编代码。

mcount的旧的实现，期望一个计数变量的地址被加载到某个寄存器中。 该变量的名字为`LP'，后面跟随数字labelno， 所以你应该生成该名字，在fprintf中使用`LP%d'。

PROFILE_HOOK [Macro]

一条C语句或者复合语句，来输出到file中一些汇编代码， 来调用profiling子程序mcount，即使target不支持profiling。

NO_PROFILE_COUNTERS [Macro]

定义该宏为一个表达式，具有一个非0的值， 如果你系统上的mcount子程序不需要为每个函数分配一个计数变量。 这对于大多数现代实现都是正确的。如果你定义了该宏， 你一定不要使用FUNCTION_PROFILER的labelno参数。

PROFILE_BEFORE_PROLOGUE [Macro]

定义该宏，如果函数profiling的代码应该位于函数序言之前。 通常profiling代码位于之后。

17.10.13 允许尾调用

bool TARGET_FUNCTION_OK_FOR_SIBCALL (tree decl, tree exp) [Target Hook]

如果可以为指定的调用表达式exp做sibling call优化，则为真。 decl为被调用的函数，或者为NULL，如果这是一个间接调用。

通常调用约定的限制不会阻止当前转换单元之外的或者PIC编译过程中的函数尾调用。 钩子用来加强这些限制，由于sibcall md模式不能fail。 成功的sibling call优化的标准可能在不同的体系结构上有很大的差别。

void TARGET_EXTRA_LIVE_ON_ENTRY (bitmap *regs) [Target Hook]

增加任何在函数入口为活跃的硬件寄存器到regs。 该钩子只需要被定义来提供不能通过检查FUNCTION_ARG_REGNO_P，callee保存的寄存器，STATIC_CHAIN_INCOMING_REGNUM，STATIC_CHAIN_REGNUM，TARGET_STRUCT_VALUE_RTX，FRAME_POINTER_REGNUM，EH_USES，FRAME_POINTER_REGNUM，ARG_POINTER_REGNUM和PIC_OFFSET_TABLE_REGNUM来发现的寄存器。

17.10.14 栈冲突保护

tree TARGET_STACK_PROTECT_GUARD (void) [Target Hook]

该钩子返回一个外部变量的DECL节点，用作栈保护者。 该变量在运行时被初始化为某个随即值，并用于初始化放在局部栈帧顶端的警卫值。 该变量的类型必须为ptr_type_node。

该钩子的缺省版本创建一个叫做`__stack_chk_guard'的变量，其通常在`libgcc2.c'中被定义。

tree TARGET_STACK_PROTECT_FAIL (void) [Target Hook]

该钩子返回一个tree表达式，用以警告运行时，栈保护者变量被修改了。 该表达式应该包括一个对无返回的（noreturn）函数的调用。

该钩子的缺省版本调用一个叫做`__stack_chk_fail'的函数，不接受任何参数。该函数通常在`libgcc2.c'中被定义。

17.11 实现Varargs宏

GCC自带了<varargs.h>和<stdarg.h>的实现，可直接用于在栈上传递参数的机器上。其它机器需要它们自己的varargs实现，并且两个机器独立的头文件必须条件包含它。

ISO <stdarg.h>与传统<varargs.h>的区别主要在va_start的调用约定上。传统的实现只接受一个参数，其为存储参数指针的变量。 ISO的实现接受额外的第二个参数。用户应该将函数的最后一个命名参数写在这里。

然而，va_start不应该使用这个参数。 发现命名参数结尾的方法为使用下面描述的内建函数。

__builtin_saveregs () [Macro]

使用该内建函数来将参数寄存器保存在内存中，使得varargs机制可以访问它们。va_start的ISO版本和传统版本都必须使用__builtin_saveregs，除非你使用TARGET_SETUP_INCOMING_VARARGS来替代（参见下面）。

在一些机器上，__builtin_saveregs为开放编码的，在target钩子TARGET_EXPAND_BUILTIN_SAVEREGS的控制下。 在其它机器上，其调用了汇编语言写的例程，可以在`libgcc2.c'中找到。

不管怎样，调用__builtin_saveregs的生成代码都出现在函数的起始处。 这是因为寄存器必须在函数开始使用它们前被保存。

__builtin_args_info (category) [Macro]

使用该内建函数来找到寄存器中第一个匿名参数。

通常，机器可以具有多个用于参数的寄存器类别，每一个用于特定的数据类型。（例如，在一些机器上，浮点寄存器用于浮点参数，而其它参数在通用寄存器中传递。）要使非varargs

函数使用合适的调用约定，你已经定义了CUMULATIVE_ARGS数据类型 来记录在每个类别中有多少寄存器已被使用。

__builtin_args_info在普通参数布局完成之后， 访问同一数据结构体CUMULATIVE_ARGS，使用category来指定访问哪个字。 因此，其值指示了在给定category中的第一个未使用的寄存器。

通常，你会在va_start的实现中使用__builtin_args_info， 访问每个类一次，并将值存储到va_list对象中。 这是因为va_list将必须更新值， 因此无法修改通过__builtin_args_info访问的值。

__builtin_next_arg (lastarg) [Macro]

这与__builtin_args_info等价，用于栈参数。其返回第一个匿名栈参数的地址， 类型为void *。如果ARGS_GROW_DOWNWARD， 其返回第一个匿名栈参数的上面的位置地址。在va_start中使用它来初始化指针， 来从栈中获得参数。同样，在va_start中使用它来验证第二个参数lastarg 为当前函数的最后一个命名参数。

__builtin_classify_type (object) [Macro]

由于每个机器具有它自己的约定，对于哪些数据类型在哪种寄存器中传递， 因此你的va_arg实现必须包含这些约定。 将指定数据类型归类的最简单方法是使用__builtin_classify_type， 加上sizeof和__alignof__。

__builtin_classify_type忽略object的值，只考虑它的数据类型。 其返回一个整数来描述什么类型为，整型，浮点，指针，结构体等。

文件`typeclass.h'定义了一个枚举， 你可以用来解析__builtin_classify_type的值。

这些机器描述宏用来帮助实现varargs：

rtx TARGET_EXPAND_BUILTIN_SAVEREGS (void) [Target Hook]

如果定义，该钩子产生机器特定代码，用于调用__builtin_saveregs。 该代码将被移动到函数的最开始处，在访问任何参数之前。 该函数的返回值应该为一个RTX，其包含了__builtin_saveregs的返回值。

void TARGET_SETUP_INCOMING_VARARGS (CUMULATIVE_ARGS *args_so_far, [Target Hook]
enum machine_mode mode, tree type, int *pretend_args_size, int second_time)

该target钩子提供了使用__builtin_saveregs和定义 TARGET_EXPAND_BUILTIN_SAVEREGS钩子的替代。 使用它来将匿名寄存器参数存储到栈中，使得所有参数都像是通过栈连续的传递。当这样做时，你可以使用varargs的标准实现，其用于将所有参数在栈上传递的机器上。

参数args_so_far指向CUMULATIVE_ARGS数据结构体， 包含了处理完命名参数之后所获得的值。 参数mode和type描述了最后一个命名参数的机器模式和作为树结点的数据类型。

该target钩子应该做两件事：第一，将所有不用于命名参数的参数寄存器压入栈中， 第二，存储数据的大小，把pretend_args_size指向的int值得变量压入。 这里你存储的值将作为额外的偏移量，用来建立栈帧。

因为你必须生成代码来将匿名参数在编译时压入，而不需要知道它们的数据类型， 所以TARGET_SETUP_INCOMING_VARARGS只在只有一种参数寄存器类别并用于所有 数据类型的机器上有用。

如果参数second_time非0，其以为着函数的参数被第二次分析。 这发生于内联函数，其直到源文件结尾才被实际编译。对于这种情况， 钩子TARGET_SETUP_INCOMING_VARARGS不应该产生任何指令。

bool TARGET_STRICT_ARGUMENT_NAMING (CUMULATIVE_ARGS *ca) [Target Hook]

定义该钩子来返回true，如果函数参数传递的位置依赖于其是否为一个命名参数。

该钩子控制对于varargs和stdarg函数，如何设置FUNCTION_ARG的named参数。 如果该钩子返回true，则named参数总是为命名参数，未命名参数总是未假。 如果返回false，但是TARGET_PRETEND_OUTGOING_VARARGS_NAMED返回 true，则所有参数都被作为命名的对待。否则所有命名参数，除了最后一个， 被作为命名的对待。

如果其总是返回0，则不需要定义该钩子。

bool TARGET_PRETEND_OUTGOING_VARARGS_NAMED [Target Hook]

如果你需要条件的改变ABI，使得一种工作于TARGET_SETUP_INCOMING_VARARGS， 另一种工作于TARGET_SETUP_INCOMING_VARARGS和 TARGET_STRICT_ARGUMENT_NAMING都没有被定义，则定义该钩子返回true， 如果使用TARGET_SETUP_INCOMING_VARARGS，否则返回false。 否则，你不需要定义该钩子。

17.12 嵌套函数的蹦床

一个蹦床trampoline为在运行时，当使用嵌套函数的地址时，创建的一小块代码。其通常驻于栈上，在包含函数的栈帧中。这些宏告诉GCC如何生成代码来分配和初始化一个蹦床。

在蹦床中的指令必须做两件事情：将一个常量地址加载到静态链寄存器中，并跳转到嵌套函数的实际地址。在CISC机器，像m68k上，这要求两条指令，一个move立即数和一个jump。然后两个地址存放在蹦床中作为字长的立即操作数。在RISC机器上，其通常需要分成两部分加载每个地址到寄存器中。然后地址的各部分形成独立的立即操作数。

用来初始化蹦床的代码必须将变量的组成部分——静态链值和函数地址——存储到指令的立即操作数中。在CISC机器上，这是简单的复制每个地址到一个内存引用，在蹦床起始处的合适偏移量上。在RISC机器上，其可能需要拿出部分地址并单独存储它们。

TRAMPOLINE_TEMPLATE (file) [Macro]

一条C语句，来在流file上，为一个包含蹦床常量部分的数据块输出汇编代码。 该代码应该不包括标号——标号被自动考虑。

如果没有定义该宏，其意味着target不需要模版。 不要在将蹦床复制到一个地方的块移动代码会比在该处生成它的代码大的系统上定义该宏。

TRAMPOLINE_SECTION [Macro]

返回蹦床模版被放入的section（参见 Section 17.19 [段], page 353 ）。 缺省值为readonly_data_section。

TRAMPOLINE_SIZE [Macro]

一个C表达式，蹦床的字节单位的大小，为整数。

TRAMPOLINE_ALIGNMENT [Macro]

蹦床需要的对齐，以位为单位。

如果没有定义该宏，则使用BIGGEST_ALIGNMENT的值来对齐蹦床。

INITIALIZE_TRAMPOLINE (addr, fnaddr, static_chain) [Macro]

一条C语句用来初始化蹦床的可变部分。addr为一个RTX，蹦床的地址； fnaddr为一个RTX，嵌套函数的地址；static_chain为一个RTX， 当其被调用时，应该传递给函数的静态链值。

TRAMPOLINE_ADJUST_ADDRESS (addr) [Macro]

一条C语句，应该执行任何机器特定的调整，对蹦床的地址。 其参数包含传给INITIALIZE_TRAMPOLINE的地址。 对于用于函数调用的地址应该不同于模版被存储的地址的情况， 应该赋给addr不同的地址。如果没有定义该宏，则addr将被用于函数调用。

如果没有定义该宏，缺省的蹦床作为一个栈槽被分配。这对于大多数机器是正确的。 例外的是一些机器，其不可能在栈区域中执行指令。在这样的机器上， 你可能必须实行一个独立的栈，使用该宏， 并结合TARGET_ASM_FUNCTION_PROLOGUE和TARGET_ASM_FUNCTION_EPILOGUE。

fp指向一个数据结构体，一个struct function， 其描述了对直接包含蹦床所对应函数的函数的编译状态。 蹦床的栈槽在该包含函数的栈帧中。其它分配策略可能也必须作一些类似的事情。

在许多机器上实现蹦床是困难的，因为它们具有独立的指令和数据缓存。 写到栈位置中使得无法清除指令缓存中的内存，所以当程序跳转到那个位置时， 其执行了旧的内容。

有两种可能的解决方法。一种是清除指令缓存的相关部分，当蹦床被建立的时候。 另一种是使所有蹦床为等同的，通过使它们跳转到一个标准的子程序中。 前者使得蹦床执行更快；后者使得初始化更快。

要在初始化蹦床时清除指令缓存，定义下列宏。

CLEAR_INSN_CACHE (beg, end) [Macro]

如果被定义，将扩展为一个C表达式，在指定的间隔处来清除指令缓存。 该宏的定义通常为一系列asm语句。beg和end都为指针表达式。

操作系统可能还需要栈被设为可执行的，在调用蹦床之前。要实现这种需求，定义下列宏。

ENABLE_EXECUTE_STACK [Macro]

定义该宏，如果在执行位于栈上的代码之前必须执行特定的操作。 宏应该扩展为一系列的C文件作用域的结构（例如函数）并提供一个唯一的如何口点名 为__enable_execute_stack。target负责生成对入口点的调用， 例如从INITIALIZE_TRAMPOLINE宏中。

要使用标准的子程序，定义下列宏。另外， 你必须确信在蹦床中的指令使用相同的指令来填充整个缓存行， 或者蹦床代码的起始处总是在缓存行的某点被对齐。查看`m68k.h'作为参考。

TRANSFER_FROM_TRAMPOLINE [Macro]

定义该宏，如果蹦床需要一个特定的子程序来做它们的工作。 该宏应该扩展为一系列的asm语句，其将由GCC来编译。 它们放在名为__transfer_from_trampoline的库函数中。

如果当你跳转到子程序时，你需要避免普通的被编译的C函数的序言代码， 你可以通过在汇编代码中放一个你自己的特定标号。 使用一条asm语句来生成汇编标号，另一条语句使得标号为global的。 然后蹦床可以使用该标号直接跳到你特定的汇编代码上。

17.13 库例程的隐式调用

这是库函数的隐式调用的说明。

DECLARE_LIBRARY_RENAMES [Macro]

该宏，如果被定义，应该扩展为一块C代码，当编译libgcc.a的函数时被扩展。 其可以被用于提供GCC内部库函数的替代名字，如果有编译应该提供的ABI名字。

void TARGET_INIT_LIBFUNCS (void) [Target Hook]

该钩子应该声明额外的库函数或者重命名存在的，使用`optabs.c'中定义的函数set_optab_libfunc和 init_one_libfunc。 init_optabs调用该宏，在初始化所有正常的库函数之后。

缺省为不作任何事情。大多数port不需要定义该钩子。

FLOAT_LIB_COMPARE_RETURNS_BOOL (mode, comparison) [Macro]

该宏应该返回true，如果实现在模式mode下的浮点比较操作符 comparison的库函数应该返回一个布尔值， 如果应该返回一个三态值则返回false。

GCC本身的浮点库从比较运算符中返回三态值，所以缺省总是返回假。 大多数port不需要定义该宏。

TARGET_LIB_INT_CMP_BIASED [Macro]

该宏应该求解为true，如果整型比较函数（像__cmpdi2） 应该返回0来指示第一个操作数比第二个小，1来指示相等， 2来指示第一个操作数大于第二个。如果该宏求解为false， 则比较函数返回-1，0和1，来替代0，1和2。如果target使用`libgcc.a'中的函数， 则不需要定义该宏。

US_SOFTWARE_GOFAST [Macro]

定义该宏，如果你的系统C库使用US Software GOFAST库来提供浮点模拟。

除了定义该宏以外，你的体系结构必须将 TARGET_INIT_LIBFUNCS设为gofast_maybe_init_libfuncs，或者从那个钩子版本中调用该函数。 其在`config/gofast.h'中定义， 且必须被你的体系结构的`cpu.c'文件包含进来。 例如参见`sparc/sparc.c'。

如果该宏被定义， 则TARGET_FLOAT_LIB_COMPARE_RETURNS_BOOL target钩子必须返回假， 对于SFmode和DFmode比较。

TARGET_EDOM [Macro]

target机器上的EDOM的值，作为一个C整型常量表达式。如果没有定义该宏， 则GCC不尝试将EDOM的值直接存放到errno中。 查看`/usr/include/errno.h'来查找你的系统上的EDOM的值。

如果没有定义TARGET_EDOM则被编译的代码通过调用库函数并使其报告错误， 来报告domain错误。如果对于这样的错误，你系统上的数学函数使用matherr， 则应该不定义TARGET_EDOM，以便matherr被正常使用。

GEN_ERRNO_RTX [Macro]

定义该宏为C表达式来创建一个rtl表达式，来引用全局“变量” errno。（在一些系统上，errno可能实际不是一个变量。）如果没有定义该宏， 则会使用一个合理的缺省。

TARGET_C99_FUNCTIONS [Macro]

当该宏非0时，GCC将隐式的优化sin调用为sinf， 类似的还有C99标准中定义的其它函数。缺省为非0，这对于大多现代系统是合适的， 然而有一些系统缺少对这些函数的运行时支持，所以它们需要该宏被重定义为0。

TARGET_HAS_SINCOS [Macro]

当该宏非0时， GCC将隐式的优化sin和cos调用为使用相同参数的sincos调用。 缺省为0。target必须提供下列函数：

```
void sincos(double x, double *sin, double *cos);
void sincosf(float x, float *sin, float *cos);
void sincosl(long double x, long double *sin, long double *cos);
```

NEXT_OBJC_RUNTIME [Macro]

定义该宏来使用NeXT系统约定为Objective-C消息发送生成代码。 该调用约定包括将对象，选择者和方法一起传递给方法查询库函数。

缺省调用约定只将对象和选择者传递给查询函数，其返回一个指向方法的指针。

17.14 寻址模式

这是关于寻址模式的宏。

HAVE_PRE_INCREMENT [Macro]
HAVE_PRE_DECREMENT [Macro]
HAVE_POST_INCREMENT [Macro]
HAVE_POST_DECREMENT [Macro]

一个C表达式，为非0，如果机器分别支持前增，前减，后增，或者后减寻址。

HAVE_PRE_MODIFY_DISP [Macro]
HAVE_POST_MODIFY_DISP [Macro]

一个C表达式，非零，如果机器支持pre-或者post-address， 除了生成内存操作数的大小以外，还有常量副作用生成。

HAVE_PRE_MODIFY_REG [Macro]
HAVE_POST_MODIFY_REG [Macro]

一个C表达式，非零，如果机器支持pre-或者post-address， 除了生成内存操作数的大小以外，还有寄存器置换的副作用生成。

CONSTANT_ADDRESS_P (x) [Macro]

一个C表达式，为1，如果RTX x为一个常量，其为一个有效地址。 在大多数机器上，这可以被定义为CONSTANT_P (x)， 但一些机器在支持哪些常量地址方面更加严格。

CONSTANT_P (x) [Macro]

CONSTANT_P，其由target无关代码定义，接受整数值表达式， 其值不被显示的知道，例如symbol_ref, label_ref，high表达式， 以及const算术表达式，const_int和const_double表达式。

MAX_REGS_PER_ADDRESS [Macro]

一个数，为可以出现在一个有效的内存地址中的最大寄存器编号。 注意需要你来指定GO_IF_LEGITIMATE_ADDRESS应该能够接受的等于该最大值的值。

GO_IF_LEGITIMATE_ADDRESS (mode, x, label) [Macro]

一个C复合语句，带有条件goto label;； 如果对于一个模式为mode的内存操作数， x（一个RTX）为一个在target机器上的合法内存地址时，被执行。

通常定义多个相对简单的宏来作为该宏的子程序。否则其可能会太复杂，难以理解。

该宏必须按照两种方式退出：一种严格的方式和一种非严格的。严格的方式用于重载阶段。其必须被定义，以便任何没有被分配硬件寄存器的伪寄存器被作为内存引用考虑。 在需要某种寄存器的上下文中，一个没有硬件寄存器的伪寄存器必须被拒绝。

非严格的方式用于其它过程。其必须被定义来接受所有伪寄存器， 在每个需要某种寄存器的上下文中。

想要使用该宏的严格方式的编译器源文件定义宏REG_OK_STRICT。 你应该使用一个#ifdef REG_OK_STRICT条件来定义严格方式，反之定义非严格的。

出于不同目的（对于基址寄存器，对于索引寄存器等等）来检查可接受的寄存器的子程序通常用于定义GO_IF_LEGITIMATE_ADDRESS。 那么只有这些子程序宏需要这两种方式；高级别的宏对于严格的或者不严格的可以相同。

通常常量地址，其为一个symbol_ref和一个整数的和， 被存储在一个const RTX中，来标记它们为一个常量。 因此，不需要专门识别这样的和是否为合法地址。 通常你应该简化识别任何合法的const。

通常PRINT_OPERAND_ADDRESS不准备处理没有标记为const的常量和。 其假设plus表示为索引。如果是这样， 你必须作为非法地址来拒绝这样的常量sum， 以便不会提供给PRINT_OPERAND_ADDRESS。

在一些机器上，一个符号地址是否为合法的，依赖于地址所引用的section。 在这些机器上，定义target钩子TARGET_ENCODE_SECTION_INFO来将信息存储 到symbol_ref中，然后在此处检查。当你遇到一个const， 你将必须查看内部，来找到symbol_ref，以便确定section。参见 Section 17.21 [汇编格式], page 357。

TARGET_MEM_CONSTRAINT [Macro]

A single character to be used instead of the default 'm' character for general memory addresses. This defines the constraint letter which matches the memory addresses accepted by GO_IF_LEGITIMATE_ADDRESS_P. Define this macro if you want to support new address formats in your back end without changing the semantics of the 'm' constraint. This is necessary in order to preserve functionality of inline assembly constructs using the 'm' constraint.

FIND_BASE_TERM (x) [Macro]

一个C表达式，用来确定地址x的base term。该宏只在两个地方使用： `alias.c'的find_base_value和 find_base_term。

不定义该宏也总是安全的。它的存在是为了别名分析可以理解机器相关的地址。

该宏的典型用法是处理在UNSPEC中包含label_ref或symbol_ref的地址。

LEGITIMIZE_ADDRESS (x, oldx, mode, win) [Macro]

一个c复合语句，对于模式为mode的操作数， 尝试使用有效的内存地址来替换x。win为代码中某处的一个c语句标号； 宏定义可以使用

```
GO_IF_LEGITIMATE_ADDRESS (mode, x, win);
```

来避免进一步的处理，如果地址已经变为合法的。

x总是为调用break_out_memory_refs的结果， oldx将为传给那个函数来生成x的操作数。

该宏生成的代码不应该修改x的子结构。 如果其将x转换为一个更加合法的形式， 其应该为x（其总为一个c变量）赋予一个新的值。

该宏不必要产生一个合法的地址。编译器具有做这件事的标准方法。 实际上省略掉该宏是安全的。但是通常机器相关的策略可以产生更好的代码。

LEGITIMIZE_RELOAD_ADDRESS (x, mode, opnum, type, ind_levels, win) [Macro]

一条C复合语句，其尝试使用一个机器模式为mode的操作数的有效内存地址， 来替换地址需要重载的x。win为代码中的一个C语句标号。 不必要定义该宏，但其可能会对性能有帮助。

例如，在i386上，有时可能通过将两个伪寄存器的和重载到一个寄存器中， 从而只使用一个重载寄存器，而不是两个。另一方面， 许多RISC处理器的偏移量是有限制的，使得经常要生成一个中间地址来寻址一个栈槽。 通过适当的定义LEGITIMIZE_RELOAD_ADDRESS， 为邻近的一些栈槽生成的中间地址可以为同一个，实现共享。

注意：该宏应该慎重使用。有必要了解重载是如何工作的，以便有效的使用该宏。

注意：该宏必须能够重载由该宏的之前调用所创建的地址。 如果不能处理这样的地址，则编译器可能会产生不正确的代码或者中断退出。

宏定义应该使用push_reload来指示需要重载的部分； opnum，type和ind_levels通常无需更改而直接传给push_reload。

该宏生成的代码必须不要修改x的子结构体。如果其将x转换成更合法的形式， 则其必须为x（其总为一个c变量）赋予一个新的值。 这也通常应用于你通过调用push_reload而间接改变的部分。

宏定义可以使用strict_memory_address_p来测试地址是否已经为合法的。

如果你只想改变x的一部分，一种标准的方法是使用copy_rtx。 但是注意，其只与rtl同一级不共享。因此，如果改变的部分不在顶层， 则你要首先替换顶层。

该宏不必要产生一个合法的地址；但是通常机器相关的策略可以产生更好的代码。

GO_IF_MODE_DEPENDENT_ADDRESS (addr, label) [Macro]

一条C语句或者复合语句，具有一个条件goto label;； 当内存地址x（一个RTX）可以具有不同的含义， 该含义取决于内存引用的机器模式时，被执行。

自动递增和自动递减地址通常具有机器模式相关的效果， 因为递增或递减的数量为被寻址的操作数的大小。 一些机器具有其它机器模式相关的地址。许多RISC机器没有机器模式相关的地址。

你可以假设addr对于机器是一个有效的地址。

LEGITIMATE_CONSTANT_P (x) [Macro]

一个C表达式，如果x对于target机器上的一个立即操作数为合法的常量， 则为非0。你可以假设x满足CONSTANT_P，所以不需要进行检查。 实际上，在任何CONSTANT_P都是有效的机器上，为该宏定义为`1'是合适的。

rtx TARGET_DELEGITIMIZE_ADDRESS (rtx x) [Target Hook]

该钩子用于撤销LEGITIMIZE_ADDRESS和LEGITIMIZE_RELOAD_ADDRESS target 宏可能造成的模糊效果。这些宏的一些后端实现， 将符号引用包含在一个UNSPEC rtx中来表示PIC或者类似的寻址模式。 该target钩子允许GCC的优化器来理解这些透明的UNSPEC的语义， 通过将它们转换回到它们最初的形式。

bool TARGET_CANNOT_FORCE_CONST_MEM (rtx x) [Target Hook]

该钩子应该返回真，如果x不能够（或不应该）被溢出到常量池中。 该钩子的缺省版本返回假。

定义该钩子的主要原因是阻止重载决定将一个不合法的常量从常量池中重载， 而不是溢出并重载一个寄存器来保存常量。对于不同的target， 该限制对于TLS符号的地址常常是真。

bool TARGET_USE_BLOCKS_FOR_CONSTANT_P (enum machine_mode mode, rtx x) [Target Hook]

该钩子应该返回真， 如果常量x的池实体（pool entries）可以放在一个object_block结构体中。 mode为x的机器模式。

缺省版本为所有的常量返回假。

tree TARGET_BUILTIN_RECIPROCAL (enum tree_code fn, bool tm_fn, bool sqrt) [Target Hook]

该钩子应该返回一个函数的decl，该函数实现了代码为fn的内建函数的倒数， 或者如果没有这样的函数，则返回NULL_TREE。 当fn为一个机器相关的内建函数的代码时，tm_fn为真。 当sqrt为真时，只对平方根函数进行额外的优化， 并且只有sqrt函数的倒数可用。

tree TARGET_VECTORIZE_BUILTIN_MASK_FOR_LOAD (void) [Target Hook]

该钩子应该返回一个函数f的decl，给定一个地址addr作为参数， 该函数返回一个掩码m，在addr没有被适当的对齐时， 其可以用于从两个向量中抽取位于addr中的相关数据。

自动向量化，当向量化一个加载操作，且地址addr可以没有对齐， 则会生成两个向量加载，从addr附件的两个对齐的地址。 其然后生成一个REALIGN_LOAD操作，来从两个加载的向量中抽取相关数据。 REALIGN_LOAD的前两个参数，v1和v2，为两个向量， 每个的大小为VS，第三个参数，OFF，定义了数据如何从这两个向量中抽取： 如果OFF为0，则返回的向量为V2 ； 否则返回的向量由v1的后VS-OFF个元素连接到v2的前OFF个元素而组成。

如果定义了该钩子，则自动向量化会生成一个对f的调用（使用该钩子返回的DECL ）并使用f的返回值作为REALIGN_LOAD的参数OFF。因此， f返回的掩码m应该遵守REALIGN_LOAD所期望的上面描述的语义。 如果该钩子没有被定义， 则addr将作为REALIGN_LOAD的参数OFF来使用， 这种情况下将会考虑addr的低log2(VS)-1位。

tree TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_EVEN (tree x) [Target Hook]

该钩子应该返回一个函数f的decl， 该函数实现了两个类型为x的输入向量作为偶数元素的加宽乘法。

如果定义了该钩子，则自动化向量当进行向量化加宽乘法时， 将会使用它和TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_ODD target钩子， 以防结果顺序不需要被保存（例如，只用于减法计算）。 否则将使用widen_mult_hi/lo。

tree TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_ODD (tree x) [Target Hook]

该钩子应该返回一个函数f的decl， 该函数实现了两个类型为x的输入向量作为奇数元素的加宽乘法。

如果定义了该钩子，则自动化向量当进行向量化加宽乘法时， 将会使用它和TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_EVEN target钩子， 以防结果顺序不需要被保存（例如，只用于减法计算）。 否则将使用widen_mult_hi/lo。

tree TARGET_VECTORIZE_BUILTIN_CONVERSION (enum tree_code code, tree type) [Target Hook]

该钩子应该返回一个函数的decl，该函数实现了类型为type的输入向量的转换。 如果type为一个整数类型，则转换结果为一个同样大小的浮点类型的向量。 如果type为浮点类型，则转换结果为一个同样大小的整数类型的向量。 code指定了如何应用转换（截断，舍入，等）。

如果定义了该钩子，则自动向量化当进行向量化转换时， 会使用TARGET_VECTORIZE_BUILTIN_CONVERSION target钩子。 否则，其应该返回NULL_TREE。

tree TARGET_VECTORIZE_BUILTIN_VECTORIZED_FUNCTION (enum built_in_function code, tree vec_type_out, tree vec_type_in) [Target Hook]

该钩子应该返回一个函数的decl， 该函数实现了代码为code的内建函数的向量化变体， 或者如果没有这样的函数则返回NULL_TREE。 向量化的函数的返回类型应该为向量类型vec_type_out， 并且参数类型应该为vec_type_in。

17.15 锚定的地址

GCC通常将每一个静态对象作为一个单独的实体来寻址。例如，如果我们有：

```
static int a, b, c;
int foo (void) { return a + b + c; }
```

foo的代码通常会计算三个独立的符号地址：a，b和c的。 在一些target上，只计算一个符号地址并且通过相对地址来访问这三个变量会更好些。 等价的伪代码可能为：

```
int foo (void)
{
 register int *xr = &x;
 return xr[&a - &x] + xr[&b - &x] + xr[&c - &x];
}
```

（这不是有效的C）。我们称像x这样的共享地址为“section anchors”。它们的用法由`-fsection-anchors'控制。

下面的钩子描述了GCC需要知道的target属性，以便有效利用section anchors。除非TARGET_MIN_ANCHOR_OFFSET或TARGET_MAX_ANCHOR_OFFSET被设为一个非0的值，否则section anchors根本不会被使用。

Target Hook HOST_WIDE_INT TARGET_MIN_ANCHOR_OFFSET [Variable]

应该应用到section anchor的最小偏移量。在大多数target上，其应该为可以应用到基址寄存器并且对每种机器模式都可以给出合法地址的最小偏移量。缺省值为0。

Target Hook HOST_WIDE_INT TARGET_MAX_ANCHOR_OFFSET [Variable]

类似TARGET_MIN_ANCHOR_OFFSET，但是为可以应用到section anchors的最大(包括)偏移量。缺省为0。

void TARGET_ASM_OUTPUT_ANCHOR (rtx x) [Target Hook]

写汇编代码来定义section anchor x，其为一个SYMBOL_REF，并且`SYMBOL_REF_ANCHOR_P (x)'为真。该钩子被调用时，汇编输出位置被设为SYMBOL_REF_BLOCK (x)的起始处。

如果ASM_OUTPUT_DEF可用，则钩子的缺省定义使用它来定义符号为`. + SYMBOL_REF_BLOCK_OFFSET (x)'。如果ASM_OUTPUT_DEF不可用，则钩子的缺省定义为NULL，其禁止了section anchors的使用。

bool TARGET_USE_ANCHORS_FOR_SYMBOL_P (rtx x) [Target Hook]

返回真，如果GCC应该尝试使用anchors来访问SYMBOL_REF x。你可以假设`SYMBOL_REF_HAS_BLOCK_INFO_P (x)'和`!SYMBOL_REF_ANCHOR_P (x)'。

缺省版本对于大多数target都正确，但是你可能需要截取该钩子来处理target特定的属性或者target特定的section。

17.16 条件代码状态

这节描述了条件代码状态。

文件`conditions.h'定义了变量cc_status，用来描述条件代码如何被计算（对于条件代码的解释取决于设置它的指令的情况）。该变量包含了条件码目前基于的RTL表达式，以及一些标准的标记。

有时额外的机器特定的标记必须被定义，在机器描述头文件中。其还可以增加额外的机器特定信息，通过定义CC_STATUS_MDEP。

CC_STATUS_MDEP [Macro]

一个数据类型的C代码，其用于声明cc_status的mdep部件。缺省为int。

该宏在不使用cc0的机器上不被使用。

CC_STATUS_MDEP_INIT [Macro]

一个C表达式，用来初始化mdep域为“空”。缺省定义不做任何事，因为大多数机器不使用该域。如果你想使用该域，则可能应该定义该宏来初始化它。

该宏在不使用cc0的机器上不被使用。

NOTICE_UPDATE_CC (exp, insn) [Macro]

一个C复合语句，用来适当的为主体为exp的insn， 设置cc_status的部件。 该宏负责识别insn将条件码设置作为副产品以及显示的set (cc0)。

该宏在不使用cc0的机器上不被使用。

如果insn不设置条件码，但修改其它机器寄存器， 则该宏必须检查它们是否使得记录条件码的表达式变为无效。例如，在68000上， 在地址寄存器上存储insn不设置条件码， 其意味着通常NOTICE_UPDATE_CC对于这样的insn可以不修改cc_status。 但是假设之前的insn将条件码设成基于位置`a4@(102)'， 并且当前insn在`a4'上存储了一个新值。虽然条件码没有被改变，但其不再为真，因为其反映了`a4@(102)'的内容。因此对于这种情况， NOTICE_UPDATE_CC必须修改cc_status，来表示条件码值不可知。

NOTICE_UPDATE_CC的定义必须要准备处理窥孔优化的结果： insn的指令模式为parallel RTXs，其包含了不同的reg， mem或者常量操作数。这些insn的RTL结构体不足以表明insn实际要做的事情。 NOTICE_UPDATE_CC应该做的是当遇到这样的，就直接运行CC_STATUS_INIT。

NOTICE_UPDATE_CC可能的定义为调用一个函数， 来查看一个属性（参见 Section 16.19 [Insn属性], page 266），例如名为`cc'。这避免了在两个地方， `md'文件和NOTICE_UPDATE_CC中具有指令模式的详细信息。

SELECT_CC_MODE (op, x, y) [Macro]

当比较运算代码op应用到rtx x和y上时， 从MODE_CC类中返回一个机器模式。例如，在SPARC上， SELECT_CC_MODE被定义为（该定义的描述，参见 Section 16.12 [跳转指令模式], page 254）

```
#define SELECT_CC_MODE(OP,X,Y) \
  (GET_MODE_CLASS (GET_MODE (X)) == MODE_FLOAT      \
   ? ((OP == EQ || OP == NE) ? CCFPmode : CCFPEmode)   \
   : ((GET_CODE (X) == PLUS || GET_CODE (X) == MINUS   \
       || GET_CODE (X) == NEG) \
      ? CC_NOOVmode : CCmode))
```

你应该定义该宏，当且仅当你在`machine-modes.def'中定义了额外的CC机器模式。

CANONICALIZE_COMPARISON (code, op0, op1) [Macro]

在一些机器上，并不是所有可能的比较都被定义， 但你可以将一个无效的比较转换为一个有效的。例如，Alpha没有GT比较， 但你可以使用LT比较来替代，并且交换操作数的顺序。

在一些机器上，定义该宏为一条C语句来做任何需要的转换。 code为初始化比较代码，op0和op1为比较的左，右操作数。 你应该根据需要来修改code，op0和op1。

GCC将不假设该宏的比较结果为有效的， 但会查看结果insn是否匹配`md'文件中的指令模式。

你不需要定义该宏，如果其不会改变比较代码或者操作数。

REVERSIBLE_CC_MODE (mode) [Macro]

一个C表达式，其值为1，如果总是可以安全的将模式为mode的比较运算逆转。 如果SELECT_CC_MODE可以为浮点不等于比较返回mode， 则REVERSIBLE_CC_MODE (mode)必须为0。

你不需要定义该宏，如果其总是返回0， 或者如果浮点格式不是IEEE_FLOAT_FORMAT。 例如，这是在SPARC上的定义，其中浮点不等于比较总是为CCFPEmode：

```
#define REVERSIBLE_CC_MODE(MODE) ((MODE) != CCFPEmode)
```

REVERSE_CONDITION (code, mode) [Macro]

一个C表达式，其值为按照CC_MODE模式进行比较的条件码的逆转。 宏只用于REVERSIBLE_CC_MODE (mode)为非0的情况。 当机器具有某种非标准的方式来反转特定条件时

，定义该宏。 例如，当所有浮点条件为非陷阱的，编译器可以自由的转换未排序的比较为排序的。 则定义可以为：

```
#define REVERSE_CONDITION(CODE, MODE) \
  ((MODE) != CCFPmode ? reverse_condition (CODE) \
  : reverse_condition_maybe_unordered (CODE))
```

REVERSE_CONDEXEC_PREDICATES_P (op1, op2) [Macro]

C表达式，返回真，如果条件执行断言op1，一个比较操作，为op2的反转， 反之亦然。定义该宏返回0，如果target具有条件执行断言，且不能被安全的反转。 不需要验证参数op1和op2相等，这已经被单独执行过。如果没有指定，该宏被定义为：

```
#define REVERSE_CONDEXEC_PREDICATES_P (x, y) \
  (GET_CODE ((x)) == reversed_comparison_code ((y), NULL))
```

bool TARGET_FIXED_CONDITION_CODE_REGS (unsigned int *, unsigned int *) [Target Hook]

对于一些target，其不使用(cc0)，而是使用硬件寄存器来存放条件码， 而不是伪寄存器，则正规的CSE过程通常不能识别硬件寄存器被设为一个普通值的情况。 使用该钩子来开启一个小的过程，来优化这种情况。该钩子应该返回真，来开启该过程， 并且应该将参数设置成指向用于条件码的硬件寄存器编号。当只有一个这样的寄存器时， 这在大多数系统上都为真，则第二个参数指向的整数应该被设为INVALID_REGNUM。

该钩子的缺省版本返回假。

enum machine_mode TARGET_CC_MODES_COMPATIBLE (enum machine_mode, enum machine_mode) [Target Hook]

对于一些target，其使用MODE_CC类别中的多个条件码机器模式， 有时比较可以对多个机器模式都有效。在这样的系统上， 定义该target钩子来接收两个mode参数并返回一个mode， 对于该模式两个比较都可以被有效执行。如果没有这样的模式，则返回VOIDmode。

该钩子的缺省版本检查模式是否相同。如果是，则返回该模式。如果不同， 则返回VOIDmode。

17.17 描述操作的相对代价

这些宏让你描述target机器上各种操作的相对速度。

REGISTER_MOVE_COST (mode, from, to) [Macro]

一个C表达式， 为从寄存器类别from到类别to移动模式为mode的数据的代价。 类别使用枚举值表示，例如GENERAL_REGS。缺省值为2；其它值相对于它来解析。

当from与to相同时，并不要求代价总是为2；在一些机器上， 如果不是通用寄存器，则寄存器之间的移动代价是昂贵的。

如果重载遇到一个insn，由两个硬件寄存器之间的单个set组成， 并且如果REGISTER_MOVE_COST应用到它们的类别上返回2， 则重载不检查insn的约束是否满足。将代价设为2以外的值将允许重载验证约束是否满足。 如果`movm'模式的约束不允许这样的复制，则你应该这样做。

MEMORY_MOVE_COST (mode, class, in) [Macro]

一个C表达式， 为在寄存器类别class和内存之间移动模式为mode的数据的代价； in为0，如果值要被写到内存中，非0，如果要从内存中读进。 该代价为REGISTER_MOVE_COST的相对值。如果在寄存器和内存间移动比两个寄存器之间更昂贵，则应该定义该宏来表示相对代价。

如果你没有定义该宏，如果需要的话， 则GCC使用缺省值4加上通过第二个重载寄存器复制的代价。 如果你的机器需要第二个重载寄存器在内存和寄存器类别class直接复制， 但是重载机制比通过中间物质复制更复杂，则定义该宏来反映move的实际代价。

GCC定义函数memory_move_secondary_cost，如果需要第二次重载。 其根据通过第二个寄存器复制来计算代价。 如果你的机器使用第二个寄存器按照约定的方式从内存中复制， 但是缺省值4对你的机器不正确，则定义该宏来增加某个其它值作为那个函数的结果。 函数的参数与该宏相同。

BRANCH_COST (speed_p, predictable_p) [Macro]
一个C表达式，为分支指令的代价。缺省值为1；其它值相对于它来解析。

这些是额外的宏，其不指定确切的相对代价，而只是指定特定的动作比GCC通常期望的要昂贵。

SLOW_BYTE_ACCESS [Macro]
定义该宏为一个C表达式，如果访问小于一个字的内存（即char或者short）不如访问一个字的内存快，即， 如果这样的访问需要多于一条的指令，并且如果字节和（对齐的）字加载的代价没有区别， 则值为非零。

当该宏没有被定义，则编译器将通过找到最小的包含对象来访问一个域；当其被定义， 如果允许对齐，则会使用全字的加载。除非字节访问比字访问快，则使用字访问比较好， 因为其可以消除后续的内存访问，如果后续的访问发生在结构体的相同字的其它域。

SLOW_UNALIGNED_ACCESS (mode, alignment) [Macro]
定义该宏的值为1， 如果由mode和alignment参数描述的内存访问比对齐的访问具有多倍的代价， 例如如果它们在陷阱处理中被模拟。

当该宏为非0时，编译器在为块移动生成代码时， 将按照STRICT_ALIGNMENT为非0的方式执行。这可以引起相当多的指令被产生。 因此如果非对齐访问只是增加一个周期或者两个，则不要设置该宏为非零。

如果该宏的值总是0，则不需要被定义。如果该宏被定义，其应该产生一个非0值， 当STRICT_ALIGNMENT非0时。

MOVE_RATIO [Macro]
标量的内存到内存的move insn的临界数，低于其值的时候， 应该生成一个insn序列，而不是字符串move insn或者库调用。 增加值将总是使得代码更快，但是会最终由于代码大小的增加而产生高的代价。

注意在一些机器上，对应的move insn为一个define_expand， 其产生一个insn序列，则该宏为该序列的个数。

如果没有定义，则会使用一个合理的缺省值。

MOVE_BY_PIECES_P (size, alignment) [Macro]
一个C表达式，用于确定是否使用move_by_pieces来复制一块内存， 或者使用其它某种块移动机制。缺省为1， 如果move_by_pieces_ninsns返回值小于MOVE_RATIO。

MOVE_MAX_PIECES [Macro]
一个C表达式， 由move_by_pieces使用用于确定load或者store用于复制内存的最大单元。 缺省为MOVE_MAX。

CLEAR_RATIO [Macro]
标量move insn的临界数，低于其值时，应该生成一个insn序列来清除内存， 而不是字符串clear insn或者库调用。增加值将总是使得代码更快， 但是会最终由于代码大小的增加而产生高的代价。

如果没有定义，则会使用一个合理的缺省值。

CLEAR_BY_PIECES_P (size, alignment) [Macro]
一个C表达式，用于确定是否使用clear_by_pieces来清除一块内存， 或者使用其它块清楚机制。缺省为1， 如果move_by_pieces_ninsns返回值小于CLEAR_RATIO。

SET_RATIO [Macro]
标量move insn的临界数，低于其值时， 应该生成一个insn序列来将内存设为一个常量值，而不是一个块设置insn或者库调用。 增加值将总是使得代码更快，但是会最终由于代码大小的增加而产生高的代价。

如果没有定义，缺省值为MOVE_RATIO。

SET_BY_PIECES_P (size, alignment) [Macro]
一个C表达式用来确定是否使用store_by_pieces来设置内存块为常量值， 或者使用其它机制。当存储非常数0的值时，由__builtin_memset使用。 缺省为1，如果move_by_pieces_ninsns返回值小于SET_RATIO。

STORE_BY_PIECES_P (size, alignment) [Macro]
一个C表达式用来确定是否使用store_by_pieces来设置内存块为常量字符串， 或者使用其它的机制。当使用常量源字符串调用时，被__builtin_strcpy使用。 缺省为1，如果move_by_pieces_ninsns返回值小于MOVE_RATIO。

USE_LOAD_POST_INCREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，后增加载是否好。 缺省值为HAVE_POST_INCREMENT。

USE_LOAD_POST_DECREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，后减加载是否好。 缺省值为HAVE_POST_DECREMENT。

USE_LOAD_PRE_INCREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，前增加载是否好。 缺省值为HAVE_PRE_INCREMENT。

USE_LOAD_PRE_DECREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，前减加载是否好。 缺省值为HAVE_PRE_DECREMENT。

USE_STORE_POST_INCREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，后增存储是否好。 缺省值为HAVE_POST_INCREMENT。

USE_STORE_POST_DECREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，后减存储是否好。 缺省值为HAVE_POST_DECREMENT。

USE_STORE_PRE_INCREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，前增存储是否好。 缺省值为HAVE_PRE_INCREMENT。

USE_STORE_PRE_DECREMENT (mode) [Macro]
一个C表达式用于确定对于给定的mode，前减存储是否好。 缺省值为HAVE_PRE_DECREMENT。

NO_FUNCTION_CSE [Macro]
定义该宏，如果调用常量函数地址要比调用保存在寄存器中的地址好些。

RANGE_TEST_NON_SHORT_CIRCUIT [Macro]
定义该宏，如果由`fold_range_test ()'产生的non-short-circuit操作为可选的。 该宏缺省为真，如果BRANCH_COST大于或等于2。

bool TARGET_RTX_COSTS (rtx x, int code, int outer_code, int *total) [Target Hook]

该target钩子描述了RTL表达式的相对代价。

代价可以依赖于表达式的确切形式，可以通过检查x来获得表达式的形式， 表达式包含的rtx的代码为outer_code。code为表达式代码，冗余的， 因为其可以使用GET_CODE (x)获得。

实现该钩子时， 你可以使用结构COSTS_N_INSNS (n)来指定代价等价于n个指令。

在钩子的入口处，*total包含了缺省的表达式代价的估值。需要的话， 钩子应该修改该值。传统的，缺省代价对于乘法为COSTS_N_INSNS (5)， 对于除法和求模为COSTS_N_INSNS (7)， 对于其它操作为COSTS_N_INSNS (1)。

当优化代码大小时，即，当optimize_size非0时， 该target钩子应该用于估计一个表达式的相对大小代价， 同样也是相对于COSTS_N_INSNS。

钩子返回真，当所有x的子表达式都被处理，当rtx_cost应该递归时返回假。

int TARGET_ADDRESS_COST (rtx address) [Target Hook]

该钩子计算包含address的寻址模式的代价。如果没有定义， 代价通过address表达式和TARGET_RTX_COST钩子来计算。

对于大多数CISC机器，缺省代价为寻址模式的真实代价的合理近似值。 然而，在RISC机器上，所有指令通常具有相同的长度和执行时间。 因此所有寻址将具有相等的代价。

对于多于一个的形式的寻址，将会使用最低代价的形式。 如果多个形式具有相同的，最低的代价，则使用最复杂的。

例如，假设地址等于寄存器和常量的和，并在同一基本块中使用两次。 当该宏没有被定义，地址将在寄存器中计算，并且内存引用将通过寄存器间接实现。 在一些机器上，包含该和的寻址模式的代价不比简单的间接引用高， 这样则会产生一条额外的指令，并且可能需要一个额外的寄存器。 对该宏进行合适的指定，会消除这样的情况。

该钩子从不会被无效地址调用。

在一些机器上，地址包括多于一个寄存器的代价， 跟只包含一个寄存器的地址计算代价一样低， 则定义TARGET_ADDRESS_COST来反映这种情况， 可以使得两个寄存器在代码域中为活跃的，如果没有定义则可能只有一个为活跃的。 这种效果在定义该宏时应该被考虑。 可能只有对于有大量寄存器的机器才可能会有相等的代价。

17.18 调整指令调度器

指令调度器可能需要一些机器特定的调整，来产生好的代码。GCC为此提供了几个target钩子。通常定义它们的一部分就足够了：先尝试该列表中最前面的。

int TARGET_SCHED_ISSUE_RATE (void) [Target Hook]

该钩子返回在target机器上同一时间可以发射的最大指令数目。缺省为1。 虽然insn调度器本身可以定义同一周期发射一个insn的可能能性， 但该值可以作为额外的约束，用于相同模拟处理器周期的insn发射（参见钩子`TARGET_SCHED_REORDER'和`TARGET_SCHED_REORDER2'）。该值在整个编译过程中必须为常量。如果你需要其依赖指令是什么而变化， 则必须使用`TARGET_SCHED_VARIABLE_ISSUE'。

int TARGET_SCHED_VARIABLE_ISSUE (FILE *file, int verbose, rtx insn, int more) [Target Hook]

该钩子在调度器从就绪列表中调度了一个insn之后被执行。 其应该返回在当前周期仍然可以被发射的insn数目。 对于CLOBBER和USE之外的insn，缺省为`more - 1'， 其通常不根据发射频率来计数。你应该定义该钩子， 如果一些insn比其它的需要更多的机器资源， 使得在同一周期它们后面可以跟随较少的insn。file或者为一个null指针， 或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。 insn为被调度的指针。

int TARGET_SCHED_ADJUST_COST (rtx insn, rtx link, rtx dep_insn, int cost) [Target Hook]

该函数根据insn和dep_insn通过依赖链接的关系来更正cost值。 其应该返回新的值。缺省为不对cost进行调整。 例如这可以用于指定调度器使用传统的流水线描述， 即输出或反向依赖不产生与数据依赖相同的代价。 如果调度器使用基于流水线描述的自动机，则反向依赖的代码为0， 输出依赖的代价为1和第一个insn与第二个insn之间的延迟时间的最大值。 如果这些值无法接受，你应该使用该钩子来修改它们。参见 Section 16.19.8 [处理器流水线描述], page 273。

int TARGET_SCHED_ADJUST_PRIORITY (rtx insn, int priority) [Target Hook]

该钩子调整insn的整数调度有限级priority。其应该返回新的priority。 增加优先级来提前执行insn，减少优先级来推迟执行insn。 如果不需要调整insn的调度优先级，则不需要定义该钩子。

int TARGET_SCHED_REORDER (FILE *file, int verbose, rtx *ready, int *n_readyp, int clock) [Target Hook]

该钩子在调度器调度完就绪列表后被执行， 以允许机器描述来重新排序（例如在`VLIW'机器上，将两个小指令合并一起）。 file或者为一个null指针，或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。 ready为指向已经被调度的指令就绪列表的指针。 n_readyp为指向在就绪列表中的元素个数的指针。 调度器按照相反的顺序读取就绪列表，从ready[*n_readyp-1]开始， 到ready[0]。clock为调度器的时钟tick。你可以修改就绪列表和insn。 返回值为这个周期可以发射的insn数；这通常只是为issue_rate。 参见`TARGET_SCHED_REORDER2'。

int TARGET_SCHED_REORDER2 (FILE *file, int verbose, rtx *ready, int *n_ready, clock) [Target Hook]

类似于`TARGET_SCHED_REORDER'，只不过在不同的时间被调用。 该函数每当调度器开始一个新的周期时被调用。其在每个周期都被调用一次， 紧跟在`TARGET_SCHED_VARIABLE_ISSUE'之后；其可以重排就绪列表并返回在同一周期被调度的insn数目。 如果常常调度一个insn会引起其他insn可以在同一周期就绪，则定义该钩子会很有用。 这样其它insn便可以被适当的考虑进来。

void TARGET_SCHED_DEPENDENCIES_EVALUATION_HOOK (rtx head, rtx tail) [Target Hook]

该钩子在由两个参数值给出的链中向前评估完insn的依赖关系之后， 但在insn链的调度之前被调用。例如，其可以被用于更好的insn分类， 如果其需要依赖分析。该钩子可以使用insn调度器的向后和向前依赖， 因为它们已经被计算好了。

void TARGET_SCHED_INIT (FILE *file, int verbose, int max_ready) [Target Hook]

该钩子在每个要被调度的指令块的起始处被调用。file或者为一个null指针， 或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。 max_ready为在当前调度域中可以同时活跃的insn最大数。 这可以用来分配需要的草稿空间，例如，`TARGET_SCHED_REORDER'。

void TARGET_SCHED_FINISH (FILE *file, int verbose) [Target Hook]

该钩子在每个要被调度的指令块的起始处被调用。 其可以用于执行清除由其它调度钩子完成的任何动作。file或者为一个null指针， 或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。

void TARGET_SCHED_INIT_GLOBAL (FILE *file, int verbose, int old_max_uid) [Target Hook]

该钩子在函数级初始化的时候被调度器执行。file或者为一个null指针， 或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。 old_max_uid为调度开始时，最大的insn uid。

void TARGET_SCHED_FINISH_GLOBAL (FILE *file, int verbose) [Target Hook]
这是一个清除钩子，对应于TARGET_SCHED_INIT_GLOBAL。 file或者为一个null指针，或者一个stdio流，用来写入调试输出。 verbose为`-fsched-verbose-n'提供的详细级别。

int TARGET_SCHED_DFA_PRE_CYCLE_INSN (void) [Target Hook]
该钩子返回一个RTL insn。流水线冒险识别器中的自动机状态， 按照当新的模拟处理器周期开始，insn被调度的样子被改变。 该钩子的用法可以简化一些VLIW处理器的自动机流水线描述。 如果钩子被定义，其只用于基于自动机的流水线描述。缺省为不改变状态， 当新的模拟处理器周期开始时。

void TARGET_SCHED_INIT_DFA_PRE_CYCLE_INSN (void) [Target Hook]
该钩子可以用于初始化先前的钩子所使用的数据。

int TARGET_SCHED_DFA_POST_CYCLE_INSN (void) [Target Hook]
该钩子与`TARGET_SCHED_DFA_PRE_CYCLE_INSN'类似，但用于改变状态， 按照当新的模拟处理器周期结束时insn被调度的方式。

void TARGET_SCHED_INIT_DFA_POST_CYCLE_INSN (void) [Target Hook]
该钩子与`TARGET_SCHED_INIT_DFA_PRE_CYCLE_INSN'类似， 但用于初始化先前的钩子所使用的数据。

void TARGET_SCHED_DFA_PRE_CYCLE_ADVANCE (void) [Target Hook]
该钩子用来通报target，当前模拟周期将要完成。 该钩子类似于`TARGET_SCHED_DFA_PRE_CYCLE_INSN'， 但用于在更复杂的情况下改变状态，例如当在一个单独的insn上前移一个状态并不足够时。

void TARGET_SCHED_DFA_POST_CYCLE_ADVANCE (void) [Target Hook]
该钩子用来通报target，新的模拟周期刚刚开始。 该钩子类似于`TARGET_SCHED_DFA_POST_CYCLE_INSN'， 但用于在更复杂的情况下改变状态，例如当在一个单独的insn上前移一个状态并不足够时。

int TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD (void) [Target Hook]
该钩子控制基于DFA的insn调度器来更好的选择一个insn， 从就绪insn队列中。通常调度器从队列中选择第一个insn。如果钩子返回一个正值， 则会有额外的调度器代码来尝试所有的`TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD ()'的排列组合， 来选择一个insn，使得发射该insn将在同一周期产生最大的insn发射数。 对于VLIW处理器， 代码实际上解决了将简单insn打包成VLIW insn的问题。 当然，如果VLIW打包规则在自动机中有描述。

该代码还能用于超标量RISC处理器。 让我们考虑一个具有3级流水的超标量RISC处理器。 一些insn可以在流水线A或B中被执行， 一些insn只能在流水线B或C中执行， 并且有一个insn可以在流水线B中被执行。处理器可以发射第一个insn到A， 第二个到B。这种情况下，第三个insn将会等待释放B，直到下一个周期。 如果调度器先发射第三个insn，则处理器可以一个周期发射所有的3个insn。

实际上该代码证明了基于自动机的流水线冒险识别器的优点。 我们最快且最容易的尝试许多insn调度，并选择最好的一种。

缺省为不进行多遍的调度。

int TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD_GUARD (rtx) [Target Hook]
该钩子控制了对于多遍insn调度，就绪insn队列中的什么样的insn将被考虑。 如果钩子返回0，对于最为参数传递的insn，则insn将不被选择发射。

缺省为所有的就绪insn都可以被选择发射。

int TARGET_SCHED_DFA_NEW_CYCLE (FILE *, int, rtx, int, int, int *) [Target Hook]
该钩子在给定的周期，在发射作为第三个参数传递的insn之前被insn调度器调用。 如果钩子返回非零，则insn在给定的处理器周期将不被发射。替代的， 处理器周期将前移。如果最后一个参数的值为0， 则insn就绪队列没有在新的周期开始时按照通常的方式被排序。 第一个参数传递了调试输出的文件。第二个参数传递了调试输出的详细级别。 第四个和第五个参数值分别对应于之前insn被发射的处理器周期，以及当前处理器周期。

bool TARGET_SCHED_IS_COSTLY_DEPENDENCE (struct dep_def *_dep, int cost, int distance) [Target Hook]
该钩子用于定义哪种依赖被target认为是具有昂贵代价的， 以至于将insn调度成依赖太近是不明智的。参数为： 第一个参数_dep为被评估的依赖。第二个参数cost为依赖的代价， 第三个参数distance为两个insn的周期距离。钩子返回true， 如果考虑两个insn间的距离，它们间的依赖被 target认为是昂贵的， 否则为false。

在多发射，乱序机器上，定义该钩子可以有用，（a）实际中是无法预测真实的数据/资源延迟， 但是（b）有一个更好的机会来预测实际要被执行的组， 并且（c）正确模拟分组会非常重要。在这样的target上， 可能想要允许发射距离较近的依赖insn，即，比依赖距离近； 但是对于“昂贵的依赖”不这样做，这样就可以定义该钩子。

void TARGET_SCHED_H_I_D_EXTENDED (void) [Target Hook]
该钩子在输出一个新的指令到指令流之后被insn调度器调用。 钩子通知target后端来延伸它的每个指令的数据结构。

void * TARGET_SCHED_ALLOC_SCHED_CONTEXT (void) [Target Hook]
Return a pointer to a store large enough to hold target scheduling context.

void TARGET_SCHED_INIT_SCHED_CONTEXT (void *tc, bool clean_p) [Target Hook]
Initialize store pointed to by tc to hold target scheduling context. It clean_p is true then initialize tc as if scheduler is at the beginning of the block. Otherwise, make a copy of the current context in tc.

void TARGET_SCHED_SET_SCHED_CONTEXT (void *tc) [Target Hook]
Copy target scheduling context pointer to by tc to the current context.

void TARGET_SCHED_CLEAR_SCHED_CONTEXT (void *tc) [Target Hook]
Deallocate internal data in target scheduling context pointed to by tc.

void TARGET_SCHED_FREE_SCHED_CONTEXT (void *tc) [Target Hook]
Deallocate a store for target scheduling context pointed to by tc.

void * TARGET_SCHED_ALLOC_SCHED_CONTEXT (void) [Target Hook]
Return a pointer to a store large enough to hold target scheduling context.

void TARGET_SCHED_INIT_SCHED_CONTEXT (void *tc, bool clean_p) [Target Hook]
Initialize store pointed to by tc to hold target scheduling context. It clean_p is true then initialize tc as if scheduler is at the beginning of the block. Otherwise, make a copy of the current context in tc.

void TARGET_SCHED_SET_SCHED_CONTEXT (void *tc) [Target Hook]
Copy target scheduling context pointer to by tc to the current context.

void TARGET_SCHED_CLEAR_SCHED_CONTEXT (void *tc) [Target Hook]
Deallocate internal data in target scheduling context pointed to by tc.

void TARGET_SCHED_FREE_SCHED_CONTEXT (void *tc) [Target Hook]
Deallocate a store for target scheduling context pointed to by tc.

int TARGET_SCHED_SPECULATE_INSN (rtx insn, int request, rtx *new_pat) [Target Hook]
该钩子当insn只有投机依赖，并因此可以被投机的调度时，被调用。 钩子用于检查insn的指令模式是否具有一个投机版本，并且如果检查成功， 则生成那个投机模式。钩子应该返回1，如果具有投机形式，或者-1，如果不具有。 request描述了请求投机的类型。如果返回值等于1， 则new_pat被赋值为生成的投机指令模式。

int TARGET_SCHED_NEEDS_BLOCK_P (rtx insn) [Target Hook]
该钩子在为insn生成恢复代码时，被insn调度器调用。其应该返回非零， 如果相应的检查指令应该分支跳转到回复代码，否则为0.

rtx TARGET_SCHED_GEN_CHECK (rtx insn, rtx label, int mutate_p) [Target Hook]
该钩子被insn调度器调用，来为恢复检查指令产生一个指令模式。 如果mutate_p为0，则insn为一个投机指令，对此应该生成检查。 label或者为一个基本快的标号，恢复代码应该被生成的地方， 或者为一个null指针，当请求的检查没有分支到恢复代码（简单的检查）。 如果mutate_p为非 0， 则由insn注解的对应于一个简单检查的指令模式应该被生成。 这种情况下，label不能为null。

int TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD_GUARD_SPEC (rtx insn) [Target Hook]
该钩子用于在就绪列表中第一个指令上没有调用`TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD_GUARD'的情况。 钩子用于丢弃从当前周期调度的起始于就绪列表的投机指令。对于非投机指令， 钩子应该总是返回非零。例如，在ia64后端，钩子用于取消数据投机insn， 当ALAT表将满时。

void TARGET_SCHED_SET_SCHED_FLAGS (unsigned int *flags, spec_info_t spec_info) [Target Hook]
该钩子被insn调度器用于查找什么特点应该被启用。 flags初始时可以设置了SCHED_RGN或SCHED_EBB位。这指示调度器过程， 应该提供什么数据。target后端应该修改flags，通过修改对应于下列特点的位： USE_DEPS_LIST, USE_GLAT, DETACH_LIFE_INFO 和DO_SPECULATION。 对于DO_SPECULATION特点，一个额外的结构体spec_info应该由target来填充。 该结构体描述了调度器中可以使用的投机类型。

int TARGET_SCHED_SMS_RES_MII (struct ddg *g) [Target Hook]
该钩子被swing modulo调度器调用，来计算基于资源的下界， 其基于在机器上可用的资源以及每个指令要求的资源。 target后端可以使用g来计算这个边界。如果没有实现该钩子， 则会使用一个非常简单的下界：指令总数除以发射率。

17.19 将输出划分到section中（Texts, Data, . . . ）

目标文件被划分到包含不同类型数据的section中。大多数情况下， 有三个section：text section，存放指令和只读数据；data section， 存放初始化的可写数据；bss section，存放未初始化的数据。 一些系统还具有其它类型的section。

`varasm.c'提供了一些已知的section，例如text_section, data_section和bss_section。 通常控制一个foo_section变量的方式是定义一个相关联的 宏FOO_SECTION_ASM_OP，正如下面将要描述的

。 宏只在`varasm.c'初始化时被读一次， 所以它们的值必须为运行时常量。不过它们可以依赖于命令行标记。

注意：一些运行时文件，例如`crtstuff.c'， 也使用FOO_SECTION_ASM_OP宏，并且将它们作为字符串文字。

一些汇编器要求每次选择section时，都要写入一个不同的字符串。 如果你的汇编器属于这类， 你应该定义TARGET_ASM_INIT_SECTIONS钩子并使用 get_unnamed_section来建立section。

你必须总是创建一个 text_section， 或者通过定义 TEXT_SECTION_ASM_OP， 或者通过在 TARGET_ASM_INIT_SECTIONS中初始化 text_section。 同样对于 data_section和 DATA_SECTION_ASM_OP。 如果你没有创建一个独立的readonly_data_section， 则缺省使用 text_section。

所有其他`varasm.c' section都是可选的，如果target不提供则为null。

TEXT_SECTION_ASM_OP [Macro]
一个C表达式，值为一个字符串，包括空格， 其包含了在指令和只读数据之前的汇编操作。通常为"\t.text"。

HOT_TEXT_SECTION_NAME [Macro]
如果定义，则为一个C字符串常量，为包含最频繁被执行的程序的函数的section名字。 如果没有定义，GCC将会提供一个缺省定义，如果target支持命名section。

UNLIKELY_EXECUTED_TEXT_SECTION_NAME [Macro]
如果定义，则为一个C字符串常量，为包含程序中不太可能被执行的函数的section名字。

DATA_SECTION_ASM_OP [Macro]
一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为可写的初始化数据的汇编操作。通常为"\t.data"。

SDATA_SECTION_ASM_OP [Macro]
如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为初始化的，可写的小数据的汇编操作。

READONLY_DATA_SECTION_ASM_OP [Macro]
一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为只读的初始化数据的汇编操作。

BSS_SECTION_ASM_OP [Macro]
如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为未初始化的， global 数据的汇编操作。 如果没有定义， 并且 ASM_OUTPUT_BSS和 ASM_OUTPUT_ALIGNED_BSS也都没有定义， 则为初始化的global数据将被输出在data section ，如果使用了`-fno-common'， 否则将使用ASM_OUTPUT_COMMON。

SBSS_SECTION_ASM_OP [Macro]
如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为未初始化的，可写的小数据的汇编操作。

INIT_SECTION_ASM_OP [Macro]
如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为初始化代码的汇编操作。如果没有定义， GCC将假设这样的section不存在。该section没有相应的init_section变量； 其完全在运行时代码中使用。

FINI_SECTION_ASM_OP [Macro]

如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为结束代码的汇编操作。如果没有定义， GCC将假设这样的section不存在。该section没有相应的fini_section变量； 其完全在运行时代码中使用。

INIT_ARRAY_SECTION_ASM_OP [Macro]

如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为.init_array（或相当的） section的一部分的 汇编操作。如果没有定义，GCC将假设这样的section不存在。 不要同时定义该宏和INIT_SECTION_ASM_OP。

FINI_ARRAY_SECTION_ASM_OP [Macro]

如果定义，则为一个C表达式，值为一个字符串，包括空格， 其包含了标识后续的数据为.fini_array（或相当的） section的一部分的 汇编操作。如果没有定义，GCC将假设这样的section不存在。 不要同时定义该宏和FINI_SECTION_ASM_OP。

CRT_CALL_STATIC_FUNCTION (section_op, function) [Macro]

如果定义，为一个ASM语句，其通过section_op来切换到不同的section， 调用function，然后切换回到text section。 这在`crtstuff.c'中使用， 如果INIT_SECTION_ASM_OP或FINI_SECTION_ASM_OP从init和 fini section中调用初始化和结束函数。缺省下，该宏使用简单的函数调用。 一些port需要手工的代码来避免在函数前奏中对寄存器初始化的依赖， 或者确保常量池在text section中不要结束的太远。

TARGET_LIBGCC_SDATA_SECTION [Macro]

如果定义，则为一个字符串， 其命名了在crtstuff和libgcc中定义的小变量应该存放的section。 这在target具有选项来优化访问小数据的时候很有用。例如， 对于具有.sdata section（像MIPS）的target， 你可以使用-G 0来编译crtstuff，使得其不需要小数据的支持， 但是使用该宏将小数据放到.sdata中， 这样你的应用程序不管是否使用小数据，都可以访问到这些变量。

FORCE_CODE_SECTION_ALIGN [Macro]

如果定义，则为一个ASM语句，其将code section对齐到某个任意的边界。 这用于使得所有.init和.fini section的fragment都具有同样的对齐， 这样就可以阻止连接器增加任何padding。

JUMP_TABLES_IN_TEXT_SECTION [Macro]

定义该宏为一个表达式，具有非零值，如果跳转表（对于tablejump insn） 应该被输出到text secton中，以及汇编指令。否则，使用只读data section。

如果没有独立的只读data section，则该宏不相关。

void TARGET_ASM_INIT_SECTIONS (void) [Target Hook]

定义该钩子，如果你需要在建立`varasm.c' section时做一些特殊的处理， 或者你的target具有一些特殊的section需要创建。

GCC在处理完命令行之后，在写任何汇编代码之前， 并在调用任何下面描述的返回section的钩子之前调用该钩子。

TARGET_ASM_RELOC_RW_MASK (void) [Target Hook]

返回一个掩码，用来描述当选择section时，应该如何对待重定位。 如果全局重定位应该放在读写section中，则应该设置位1； 如果局部重定位应该被放在读写section中，则应该设置位0。

该函数的缺省版本返回3，当`-fpic'有效时，否则返回0。 当target不支持（某种）在只读section中，甚至在可执行程序中的动态重定位时， 通常会重定义该钩子。

section * TARGET_ASM_SELECT_SECTION (tree exp, int reloc, unsigned HOST_WIDE_INT align) [Target Hook]

返回exp应该被放入的section。 你可以假设exp为VAR_DECL节点或者一个常量。 reloc指示exp的初始化值是否需要连接时重定位。 当变量只包含局部重定位时位0被设置，对于全局重定位位1被设置。 align为常量对齐位数。

该函数的缺省版本只关心将只读变量放到readonly_data_section中。

参见USE_SELECT_SECTION_FOR_FUNCTIONS.

USE_SELECT_SECTION_FOR_FUNCTIONS [Macro]

如果你希望对于FUNCTION_DECL， 将会调用TARGET_ASM_SELECT_SECTION，则定义该宏。同样对于变量和常量。

对于FUNCTION_DECL，reloc将为0，如果函数被确定有可能被调用， 非零如果其不能被调用。

void TARGET_ASM_UNIQUE_SECTION (tree decl, int reloc) [Target Hook]

构建一个唯一的section名， 使用STRING_CST节点表示， 并赋值为`DECL_SECTION_NAME (decl)'。 跟TARGET_ASM_SELECT_SECTION一样， reloc指示exp的初始化值是否需要连接时重定位。

该函数的缺省版本向ELF section名中追加一个符号名。例如， 函数foo将被放在.text.foo中。 这对于实际的target目标格式通常是可以的。

section * TARGET_ASM_FUNCTION_RODATA_SECTION (tree decl) [Target Hook]

返回与`DECL_SECTION_NAME (decl)'关联的只读data section。 该函数的缺省版本选择.gnu.linkonce.r.name， 如果函数的section为.gnu.linkonce.t.name， .rodata.name如果函数在.text.name中， 否则为通常的只读data section。

section * TARGET_ASM_SELECT_RTX_SECTION (enum machine_mode mode, rtx x, unsigned HOST_WIDE_INT align) [Target Hook]

返回具有机器模式mode的常量x应该放入的section。 你可以假设x为RTL形式的某种常量。参数mode除了const_int之外， 是冗余的。align为常量对齐位数。

该函数的缺省版本考虑将符号常量flag_pic模式的， 放在data_section中， 其它放在readonly_data_section中。

void TARGET_MANGLE_DECL_ASSEMBLER_NAME (tree decl, tree id) [Target Hook]

定义该钩子，如果你需要处理由target无关的代码生成的汇编名。 提供给该钩子的id将为被计算的名字（例如C中的DECL_NAME宏， 或者C++中的mangled name）。该钩子的返回值为一个IDENTIFIER_NODE。 该钩子的缺省实现只是返回提供的id。

void TARGET_ENCODE_SECTION_INFO (tree decl, rtx rtl, int new_decl_p) [Target Hook]

定义该钩子，如果对符号或者常量的引用必须根据符号所命名的变量或者函数来不同处理（例如其在哪个section中）。

钩子在为decl创建rtl之后立即被执行， decl可能为一个变量或者函数声明，或者常量池的入口。不要在该钩子中 使用DECL_RTL (decl)；那个域可能还没有被初始化。

对于常量，可以假设rtl为一个mem，其地址为一个 symbol_ref。 大多数decl将具有这种形式，但不被保证。全局寄存器变量，例如， 它们的rtl将具有一个reg。（对于这样不寻常的rtl通常是将其放在一边）。

参数new_decl_p将为真， 如果这是第一次对于该decl调用TARGET_ENCODE_SECTION_INFO。 对于后续的调用其将为假，这放生在复制的声明中。对于复制声明，是否需要做什么， 取决于钩子是否检查DECL_ATTRIBUTES。当钩子对于常量被调用， 则new_decl_p总为真。

该钩子通常做的事情是记录symbol_ref中的标记，使用SYMBOL_REF_FLAG或SYMBOL_REF_FLAGS。

该钩子的缺省定义，`varasm.c'中的default_encode_section_info，设置了SYMBOL_REF_FLAGS中通常有用的位。在覆盖它之前检查缺省代码是否做了你所需要的。

const char *TARGET_STRIP_NAME_ENCODING (const char *name) [Target Hook]
解析name并返回真实的名字部分，没有TARGET_ENCODE_SECTION_INFO可能加进去的字符。

bool TARGET_IN_SMALL_DATA_P (tree exp) [Target Hook]
返回真，如果exp应该被放到“小数据”section中。该钩子的缺省版本总是返回假。

Target Hook bool TARGET_HAVE_SRODATA_SECTION [Variable]
如果target将只读“小数据”放到单独的section中，则包含值为真。缺省值为假。

bool TARGET_BINDS_LOCAL_P (tree exp) [Target Hook]
返回真，如果exp命名了一个对象，其名字解析规则必须

该钩子的缺省版本实现了ELF的名字解析规则，其具有一个比目前支持的其它目标文件格式较松散的全局名字绑定模型。

Target Hook bool TARGET_HAVE_TLS [Variable]
如果target支持thread-local storage，则包含值为真。缺省值为假。

17.20 位置独立代码

这一节描述了帮助实现位置独立代码生成的宏。简单的定义这些宏并不足以生成有效的PIC；你必须还要增加对宏GO_IF_LEGITIMATE_ADDRESS和PRINT_OPERAND_ADDRESS的支持，还有LEGITIMIZE_ADDRESS。你必须修改`movsi'的定义，当源操作数包含一个符号地址时来做一些适当的处理。还可能需要修改switch语句的处理，使得它们使用相对地址。

PIC_OFFSET_TABLE_REGNUM [Macro]
用于寻址内存中静态数据地址表的寄存器的编号。一些情况下，该寄存器由处理器的“应用二进制接口”（ABI）定义。当该宏被定义时，会为该寄存器生成一次RTL，使用栈指针和帧指针寄存器。如果该宏没有被定义，则需要机器相关文件来分配这样的寄存器（如果需要的话）。注意该寄存器在使用时（即当flag_pic为真时）必须为固定的。

PIC_OFFSET_TABLE_REG_CALL_CLOBBERED [Macro]
定义该宏，如果由PIC_OFFSET_TABLE_REGNUM定义的寄存器被调用破坏。如果PIC_OFFSET_TABLE_REGNUM没有被定义，则不要定义该宏。

LEGITIMATE_PIC_OPERAND_P (x) [Macro]
一个C表达式，如果当生成位置独立代码时，x为target机器上的合法的立即数操作数，则为非零。你可以假设x满足CONSTANT_P，所以不需要进行检查。你还可以假设flag_pic为真，所以也不需要进行检查。如果当生成位置独立代码时，所有的常量（包括SYMBOL_REF）都可以为立即操作数，则不需要定义该宏。

17.21 定义汇编语言输出

这一节描述的宏，主要用于描述如何使用汇编语言书写指令，而不是指令本身。

17.21.1 汇编文件的总体框架

这章描述了汇编文件的总体框架。

void TARGET_ASM_FILE_START () [Target Hook]

将汇编器期望在文件起始处发现的任何文本输出到asm_out_file。 缺省行为由两个标志控制，将在下面介绍。除非你target的汇编器十分不常见， 如果你覆盖了缺省，则应该在你的target钩子上的某处调用default_file_start。 这将让其它target文件依赖于这些变量。

bool TARGET_ASM_FILE_START_APP_OFF [Target Hook]

如果该标记为真，则宏ASM_APP_OFF的文本将作为汇编文本的第一行被打印， 除非正在使用`-fverbose-asm'。（如果那个宏被定义为空字符串， 则该变量将不起作用。）对于ASM_APP_OFF的通常定义， 其效果是告知GNU汇编器不需要费功夫从输入中去除掉注释或额外的空格。 这将使得运行更快些。

缺省为假。不要将其设为真，除非你已经核实你的port不会产生任何额外的空格或者注释，这将使得GAS在NO_APP模式下产生错误。

bool TARGET_ASM_FILE_START_FILE_DIRECTIVE [Target Hook]

如果该标记为真，对于主源文件，output_file_directive将被调用， 就在打印了ASM_APP_OFF之后（如果启用了该功能）。 大多数ELF汇编器期望做这件事。缺省为假。

void TARGET_ASM_FILE_END () [Target Hook]

将汇编器期望在文件结尾发现的任何文本输出到中。缺省为什么也不输出。

void file_end_indicate_exec_stack () [Function]

一些系统使用通用的约定，用特定的`.note.GNU-stack' section， 来指示一个依赖栈的目标文件是否被执行。如果你的系统使用了该约定， 你应该将TARGET_ASM_FILE_END定义为该函数。 如果你需要在钩子中做其它事情，则要在你的钩子函数中调用该函数。

ASM_COMMENT_START [Macro]

一个C字符串常量，描述了如何在target汇编语言中起始一条注释。 编译器假设注释将结束于行尾。

ASM_APP_ON [Macro]

一个C字符串常量，将在每个asm语句或者一组连续的asm语句之前被输出。 通常为"#APP"，其为一条注释，对于多数汇编器不起作用， 但可以告诉GNU汇编器必须对随后的行检查所有的有效汇编结构。

ASM_APP_OFF [Macro]

一个C字符串常量，将在每个asm语句或者一组连续的asm语句之前被输出。 通常为"#NO_APP"，告诉GNU汇编器继续进行省时的假设， 认为通常的编译器输出是有效的。

ASM_OUTPUT_SOURCE_FILENAME (stream, name) [Macro]

一条C语句，输出COFF信息或者DWARF调试信息， 用来指示文件名name为当前的对于标准输入输出流stream的源文件。

该宏不需要被定义，如果输出的标准形式适用于正在使用的文件格式。

OUTPUT_QUOTED_STRING (stream, string) [Macro]

一条C语句，将字符串string输出到stdio流stream中。 如果你不在你的配置文件中调用函数output_quoted_string， 则GCC将只调用它来输出文件名到汇编源文件中。 所以你可以使用它来规范使用该宏的文件名的格式。

ASM_OUTPUT_IDENT (stream, string) [Macro]

一条C语句，用来输出一些东西到汇编文件中， 以处理包含文本string的`#ident'伪指令。如果没有定义该宏， 对于`#ident'伪指令，将不输出任何东西。

void TARGET_ASM_NAMED_SECTION (const char *name, unsigned int flags, unsigned int align) [Target Hook]

输出汇编伪指令来切换section名字。section应该具有flags指定的属性， 其为`output.h'中定义的SECTION_*标记的位掩码。如果align非零， 则其包含了用于section的对齐字节数，否则将使用target缺省的。 只有那些必须在section伪指令里指定对齐的target才需要关注align — 我们仍然使用ASM_OUTPUT_ALIGN。

bool TARGET_HAVE_NAMED_SECTIONS [Target Hook]

该标记为真，如果target支持TARGET_ASM_NAMED_SECTION。

bool TARGET_HAVE_SWITCHABLE_BSS_SECTIONS [Target Hook]

该标记为真，如果我们能够通过切换到BSS section来创建初始化为0的数据， 并且使用ASM_OUTPUT_SKIP来分配空间。这在大多数ELF target上为真。

unsigned int TARGET_SECTION_TYPE_FLAGS (tree decl, const char *name, int reloc) [Target Hook]

根据变量或函数decl，section名name，和声明的初始化是否可以包含运行时重定位， 来选择供TARGET_ASM_NAMED_SECTION使用的section属性集。 decl可以为null，这种情况下，将假设为可读写数据。

该函数的缺省版本用来处理选择代码和数据，只读数据和可读写数据， 以及flag_pic。 你应该只有在你的target具有通过__attribute__设置的特定标记时， 才需要覆盖该函数。

int TARGET_ASM_RECORD_GCC_SWITCHES (print_switch_type type, const char * text) [Target Hook]

使得target能够记录传给编译器的gcc命令行开关，和打开的选项。 参数type指定了要记录的内容。其可以为下面的值：

SWITCH_TYPE_PASSED

text为用户设置的命令行开关。

SWITCH_TYPE_ENABLED

text为开启的选项。这可能为命令行开关的直接结果，或者是因为其被缺省打开， 或者因为其被不同的命令行开关的副作用打开。例如， `-O2'开关打开了各种不同的独立的优化过程。

SWITCH_TYPE_DESCRIPTIVE

text或者为NULL，或者为一些应该被忽略的描述文本。如果texttext为NULL，则其被用来警告target钩子记录开始或者结束。 第一次， type为SWITCH_TYPE_DESCRIPTIVE并且text为NULL，这是警告记录开始了， 第二次，是警告结束了。该特征可以允许target钩子来做任何需要的准备， 在其开始记录开关的时候，并执行一些比较的整理，在其完成记录开关之后。

SWITCH_TYPE_LINE_START

该选项可以被该target钩子忽略掉。

SWITCH_TYPE_LINE_END

该选项可以被该target钩子忽略掉。

钩子的返回值必须为0。其它返回值可能在将来被支持。

缺省下，该钩子被设为NULL，但是对于基于ELF的target，有一个实现例子。叫做elf_record_gcc_switches，其记录了开关，并作为ASCII文本放在汇编输出文件中一个新的，可以字符串合并的section。新section的名字由target钩子TARGET_ASM_RECORD_GCC_SWITCHES_SECTION提供。

const char * TARGET_ASM_RECORD_GCC_SWITCHES_SECTION [Target Hook]

这是TARGET_ASM_RECORD_GCC_SWITCHES target钩子的ELF实现的例子所创建的section名字。

17.21.2 数据的输出

const char * TARGET_ASM_BYTE_OP [Target Hook]
const char * TARGET_ASM_ALIGNED_HI_OP [Target Hook]
const char * TARGET_ASM_ALIGNED_SI_OP [Target Hook]
const char * TARGET_ASM_ALIGNED_DI_OP [Target Hook]
const char * TARGET_ASM_ALIGNED_TI_OP [Target Hook]
const char * TARGET_ASM_UNALIGNED_HI_OP [Target Hook]
const char * TARGET_ASM_UNALIGNED_SI_OP [Target Hook]
const char * TARGET_ASM_UNALIGNED_DI_OP [Target Hook]
const char * TARGET_ASM_UNALIGNED_TI_OP [Target Hook]

这些钩子指定了用于创建特定类型的整数对象的汇编伪指令。TARGET_ASM_BYTE_OP伪指令创建一个字节大小的对象，TARGET_ASM_ALIGNED_HI_OP创建一个两个字节对齐的对象，等等。这些钩子都可以为NULL，这表示没有合适的伪指令。

编译器将在一个新行中的起始处打印这些字符串，随后紧跟对象的初始化值。大多数情况下，字符串应该包含一个tab，一个伪操作符，然后是另一个tab。

bool TARGET_ASM_INTEGER (rtx x, unsigned int size, int aligned_p) [Target Hook]

函数assemble_integer使用该钩子来输出一个整数对象。x为对象的值，size为它的以字节为单位的大小，aligned_p指示其是否为对齐的。函数应该返回真，如果它能够输出对象。如果返回假，则assemble_integer将尝试把对象分割为更小的部分。

该钩子的缺省实现将使用TARGET_ASM_BYTE_OP字符串家族，当相应字符串为NULL时返回假。

OUTPUT_ADDR_CONST_EXTRA (stream, x, fail) [Macro]

一条C语句用来识别output_addr_const不能处理的rtx模式，并输出汇编代码到模式x对应的stream中。这可以用来允许在常量中出现机器相关的UNSPEC。

如果OUTPUT_ADDR_CONST_EXTRA没有能够识别出指令模式，其必须goto fail，这样就会打印出一个标准错误消息。如果其本身打印了一个错误消息，例如通过调用output_operand_lossage，其可以正常的结束。

ASM_OUTPUT_ASCII (stream, ptr, len) [Macro]

一条C语句，用来输出到stdio流stream中一条汇编指令，以组合一个在ptr处包含len个字节的字符串常量。ptr将为一个char *类型的C表达式，len为一个int型的C表达式。

如果汇编器具有一个.ascii伪指令，正如在Berkeley Unix汇编器上的，则不要定义宏ASM_OUTPUT_ASCII。

ASM_OUTPUT_FDESC (stream, decl, n) [Macro]

一条C语句，用来输出decl的函数描述符的字n。这必须在定义TARGET_VTABLE_USES_DESCRIPTORS时被定义，否则将不起作用。

CONSTANT_POOL_BEFORE_FUNCTION [Macro]

你可以定义该宏为一个C表达式。你应该定义表达式具有非零值， 如果GCC应该在输出函数的代码前，输出常量池，或者定义为0， 如果GCC应该在函数后输出常量池。如果你不定义该宏，则通常情况下， GCC将在函数前输出常量池。

ASM_OUTPUT_POOL_PROLOGUE (file, funname, fundecl, size) [Macro]

一条C语句，用来输出汇编命令，以定义函数的常量池的起始。 funname为一个字符串，给定了函数的名字。如果需要函数的返回类型， 则可以通过fundecl来获得。 size为在该调用之后要立即写入的常量池的大小，以字节为单位。

通常情况下，如果不需要常量池前缀，该宏不需要被定义。

ASM_OUTPUT_SPECIAL_POOL_ENTRY (file, x, mode, align, labelno, jumpto) [Macro]

一条C语句（带有或者不带有分号），用来输出一个常量在常量池中， 如果其需要特殊的处理。（该宏对于可以正常输出的RTL表达式不需要做任何事情。）

参数file为将汇编代码输出到的标准I/O流。x为要输出的常量的RTL表达式， mode为机器模式（用于x为`const_int'时）。 align为值x所需要的对齐；你应该输出一个汇编伪指令来执行该对齐。

参数labelno为该池中实体的地址的内部标号的编号。 该宏的定义负责在合适的地方输出标号的定义。这里有一个实现的例子：

```
(*targetm.asm_out.internal_label) (file, "LC", labelno);
```

当你专门输出一个池中实体时，你应该结束于一个goto， 以跳转到标号jumpto。这将阻止相同的池中实体通过通常的方式被再一次输出。

如果不做任何事情，则不要定义该宏。

ASM_OUTPUT_POOL_EPILOGUE (file funname fundecl size) [Macro]

一条C语句，用来输出汇编命令到函数常量池的结尾。funname为一个字符串， 给出了函数的名字。如果需要函数的返回类型，可以通过fundecl来获得。 size为GCC在该调用之前立即写入的常量池的大小，以字节为单位。

通常情况下，如果不需要常量池结束语，则不需要定义该宏。

IS_ASM_LOGICAL_LINE_SEPARATOR (C, STR) [Macro]

定义该宏为一个C表达式，其为非零，如果C被汇编器用作逻辑行分隔符。 STR指向在字符串中C被发现的位置； 这可以用于行分隔符使用多个字符的时候。

如果你不定义该宏，则缺省的为只将字符`;'作为逻辑行的分隔符。

const char * TARGET_ASM_OPEN_PAREN [Target Hook]

const char * TARGET_ASM_CLOSE_PAREN [Target Hook]

这些target钩子为C字符串常量，描述了算术表达式组合的汇编语法。 如果没有被覆盖，它们缺省为通常的括号，这对大多数汇编器都是正确的。

这些宏由`real.h'提供，用于写ASM_OUTPUT_DOUBLE等的定义：

REAL_VALUE_TO_TARGET_SINGLE (x, l) [Macro]

REAL_VALUE_TO_TARGET_DOUBLE (x, l) [Macro]

REAL_VALUE_TO_TARGET_LONG_DOUBLE (x, l) [Macro]

REAL_VALUE_TO_TARGET_DECIMAL32 (x, l) [Macro]

REAL_VALUE_TO_TARGET_DECIMAL64 (x, l) [Macro]

REAL_VALUE_TO_TARGET_DECIMAL128 (x, l) [Macro]

这些将类型为REAL_VALUE_TYPE的x，转换为target的浮点表示，并将其存储在变量l中。对于REAL_VALUE_TO_TARGET_SINGLE和REAL_VALUE_TO_TARGET_DECIMAL32，该变量应该为一个简单的long int。对于其它的，其应该为一个long int的数组。该数组的元素个数由所需要的target浮点数据类型的大小决定：每个long int数组元素有32位。每个数组元素存放32位的结果，即使long int在host机器上比32位宽。

数组元素值被设计成，可以使用fprintf按照在target机器内存中的顺序来打印它们。

17.21.3 未初始化变量的输出

这一节的每个宏都是用于输出单个未初始化变量的整个工作中。

ASM_OUTPUT_COMMON (stream, name, size, rounded) [Macro]

一条C语句（没有分号），用来将大小为size，名字为name的通用标号的汇编定义，输出到stdio流stream中。变量rounded为调用者想要对齐而舍入的大小。

使用表达式assemble_name (stream, name)来输出name本身；在这之前和之后，输出额外的定义name的汇编语法，以及换行。

该宏控制如何输出未初始化的通用全局变量的汇编定义。

ASM_OUTPUT_ALIGNED_COMMON (stream, name, size, alignment) [Macro]

类似ASM_OUTPUT_COMMON，除了其接受一个alignment，作为独立，显式的参数。如果你定义了该宏，其被用于替换ASM_OUTPUT_COMMON，使得你在处理变量对齐方面变得更加灵活。aligment被指定为位的数目。

ASM_OUTPUT_ALIGNED_DECL_COMMON (stream, decl, name, size, alignment) [Macro]

类似ASM_OUTPUT_ALIGNED_COMMON，除了要被输出的变量的decl，如果存在的话，或者为NULL_TREE如果没有相应的变量。如果你定义了该宏，GCC将替换ASM_OUTPUT_COMMON和ASM_OUTPUT_ALIGNED_COMMON。当你需要看到变量的decl，以便选择如何输出时，可以定义该宏。

ASM_OUTPUT_BSS (stream, decl, name, size, rounded) [Macro]

一条C语句（没有分号），用来将名字为name，大小为size个字节的未初始化的全局decl的汇编定义输出到stdio流 stream中。变量rounded为调用者想要对齐而舍入的大小。

当定义该宏时，可以尝试使用`varasm.c'中定义的asm_output_bss。如果不行，使用表达式assemble_name (stream, name)来输出name本身；在此之前和之后，输出额外的定义name的汇编语法，以及换行。

有两种方式来处理全局BSS。一种是定义该宏或者它的对齐副本，ASM_OUTPUT_ALIGNED_BSS。另一种是让TARGET_ASM_SELECT_SECTION返回一个可切换的BSS section（参见[TARGET_HAVE_SWITCHABLE_BSS_SECTIONS], page 359）。你不需要两者都做。

一些语言不具有common数据，并且要求全局BSS为non-common的，以便高效的处理未初始化全局变量。C++就是这样的例子。然而，如果target不支持全局BSS，则前端可以选择生成全局common，以便在目标文件中节省空间。

ASM_OUTPUT_ALIGNED_BSS (stream, decl, name, size, alignment) [Macro]

类似ASM_OUTPUT_BSS，除了其接受需要的aligment作为单独，显式的参数。如果你定义了该宏，其被用于替换ASM_OUTPUT_BSS，这使得你在处理变量所需的对齐方面更加灵活。aligment被指定为位数。

当定义该宏时，尝试使用在文件`varasm.c'中定义的函数asm_output_aligned_bss。`varasm.c' when defining this macro.

ASM_OUTPUT_LOCAL (stream, name, size, rounded) [Macro]

一条C语句（没有分号），用来将名字为name，大小为size个字节的local-common标号的汇编定义输出到stdio流stream中。 变量rounded为调用者想要对齐而舍入的大小。

使用表达式assemble_name (stream, name)来输出name本身； 在此之前和之后，输出额外的定义name的汇编语法，以及换行。

该宏控制如何输出未初始化的静态变量的汇编定义。

ASM_OUTPUT_ALIGNED_LOCAL (stream, name, size, alignment) [Macro]

类似ASM_OUTPUT_LOCAL，除了其接受需要的aligment作为单独，显式的参数。 如果你定义了该宏，其被用于替换ASM_OUTPUT_LOCAL， 这使得你在处理变量所需的对齐方面更加灵活。 aligment被指定为位数。

ASM_OUTPUT_ALIGNED_DECL_LOCAL (stream, decl, name, size, alignment) [Macro]

类似ASM_OUTPUT_ALIGNED_DECL，除了要被输出的变量的decl， 如果存在的话，或者为NULL_TREE如果没有相应的变量。如果你定义了该宏， GCC将替换ASM_OUTPUT_DECL和ASM_OUTPUT_ALIGNED_DECL。 当你需要看到变量的decl，以便选择如何输出时，可以定义该宏。

17.21.4 标号的生成和输出

这节是关于标号输出的。

ASM_OUTPUT_LABEL (stream, name) [Macro]

一条C语句（没有分号）， 用来将名字为name的标号的汇编定义输出到stdio流stream中。 使用表达式assemble_name (stream, name)来输出name本身； 在此之前和之后，输出定义name的额外的汇编语法，以及换行。 该宏的缺省定义被提供，其对于多数系统都是正确的。

ASM_OUTPUT_INTERNAL_LABEL (stream, name) [Macro]

等同于ASM_OUTPUT_LABEL，除了name为已知的，引用了编译器生成的标号。 缺省定义使用assemble_name_raw，其类似于assemble_name， 只不过更加高效。

SIZE_ASM_OP [Macro]

一个C字符串，包含了适当的汇编伪指令，用于指定符号的大小，不需要任何参数。 在使用ELF的系统上，缺省为（在`config/elfos.h'中）`"\t.size\t"'； 在其它系统上，缺省为不定义该宏。

只有在你的系统上可以正确的使用ASM_OUTPUT_SIZE_DIRECTIVE和ASM_OUTPUT_MEASURED_SIZE的缺省定义时，才定义该宏。 如果对于那些宏，你需要自己特定的定义，或者如果你根本不需要显式的符号大小， 则不要定义该宏。

ASM_OUTPUT_SIZE_DIRECTIVE (stream, name, size) [Macro]

一条C语句（没有分号），用来将一条伪指令输出到stdio流stream中， 以告诉汇编器符号name的大小为size。 size为HOST_WIDE_INT。 如果你定义了SIZE_ASM_OP，则该宏的缺省定义会被提供。

ASM_OUTPUT_MEASURED_SIZE (stream, name) [Macro]

一条C语句（没有分号），用来将一条伪指令输出到stdio流stream中， 以告诉汇编器通过从当前地址减去符号name的地址，来计算符号的大小。

如果你定义了SIZE_ASM_OP，则该宏的缺省定义会被提供。 缺省定义假设汇编器可以识别特殊的`. '符号，作为引用当前地址， 并能够计算该处和其它符号的差。如果你的汇编器不识别`. '，或者不能计算差， 你需要重定义ASM_OUTPUT_MEASURED_SIZE来使用其它技术。

TYPE_ASM_OP [Macro]

一个C字符串，包含了适当的汇编伪指令，用于指定符号的类型，不需要任何参数。 在使用ELF的系统上，缺省为（在`config/elfos.h'中）`"\t.type\t"'； 在其它系统上，缺省为不定义该宏。

只有在你的系统上可以正确的使用ASM_OUTPUT_TYPE_DIRECTIVE的缺省定义时， 才定义该宏。 如果对于该宏，你需要自己特定的定义， 或者如果你根本不需要显式的符号类型，则不要定义该宏。

TYPE_OPERAND_FMT [Macro]

一个C字符串， 指定了TYPE_ASM_OP的第二个操作数的格式（使用printf语法）。 在使用ELF的系统上，缺省为（在`config/elfos.h'中）`"@%s"'； 在其它系统上，缺省为不定义该宏。

只有在你的系统上可以正确的使用ASM_OUTPUT_TYPE_DIRECTIVE的缺省定义时， 才定义该宏。 如果对于该宏，你需要自己特定的定义， 或者如果你根本不需要显式的符号类型，则不要定义该宏。

ASM_OUTPUT_TYPE_DIRECTIVE (stream, type) [Macro]

一条C语句（没有分号），用以将一条伪指令输出到stdio流stream中， 以告诉汇编器符号name的类型为type。type是一个C字符串； 目前该字符串总是`"function"'或者`"object"'， 但你不要依赖于此。

如果你定义了TYPE_ASM_OP和TYPE_OPERAND_FMT， 则该宏的缺省定义会被提供。

ASM_DECLARE_FUNCTION_NAME (stream, name, decl) [Macro]

一条C语句（没有分号），用以将任何声明被定义的函数名字name所需要的文本， 输出到stdio流stream中。 该宏负责输出标号定义（或者使用ASM_OUTPUT_LABEL）。 参数decl为表示函数的FUNCTION_DECL树结点。

如果该宏没有被定义， 则函数名被作为标号按照通常的方式来定义（使用ASM_OUTPUT_LABEL）。

你可能希望在定义该宏时使用ASM_OUTPUT_TYPE_DIRECTIVE。

ASM_DECLARE_FUNCTION_SIZE (stream, name, decl) [Macro]

一条C语句（没有分号），用以将任何声明被定义的函数的大小所需要的文本， 输出到stdio流stream中。参数name为函数的名字。 参数decl为表示函数的FUNCTION_DECL树结点。

如果该宏没有被定义，则函数大小没有被定义。

你可能希望在定义该宏时使用ASM_OUTPUT_MEASURED_SIZE。

ASM_DECLARE_OBJECT_NAME (stream, name, decl) [Macro]

一条C语句（没有分号）， 用以将任何声明被定义的初始化变量名字name所需要的文本， 输出到stdio流stream中。 该宏必须输出标号定义（可能使用ASM_OUTPUT_LABEL）。 参数decl为表示变量的VAR_DECL树结点。

如果该宏没有被定义， 则变量名被作为标号按照通常的方式来定义（使用ASM_OUTPUT_LABEL）。

你可能希望在定义该宏时使用ASM_OUTPUT_TYPE_DIRECTIVE和/或ASM_OUTPUT_SIZE_DIRECTIVE。

ASM_DECLARE_CONSTANT_NAME (stream, name, exp, size) [Macro]

一条C语句（没有分号），用以将任何声明被定义的常量名字name所需要的文本， 输出到stdio流stream中。 该宏负责输出标号定义（可能使用ASM_OUTPUT_LABEL）。 参数exp为常量的值，size为常量的大小，以字节为单位。 name为内部标号。

如果该宏没有被定义， 则name被作为标号按照通常的方式来定义（使用ASM_OUTPUT_LABEL）。

你可能希望在定义该宏时使用ASM_OUTPUT_TYPE_DIRECTIVE。

ASM_DECLARE_REGISTER_GLOBAL (stream, decl, regno, name) [Macro]

一条C语句（没有分号），用以将任何为具有名字name的全局变量声明一个寄存器regno所需要的文本，输出到stdio流stream中。

如果没有定义该宏，则相当于定义其什么都不做。

ASM_FINISH_DECLARE_OBJECT (stream, decl, toplevel, atend) [Macro]

一条C语句（没有分号），用于在编译器完全处理了初始化者之后， 来完成声明一个变量名，这样当数组的大小由初始化者控制的时候， 就有机会来确定数组的大小。这用于需要声明对象的大小的系统上。

如果没有定义该宏，则相当于定义其什么都不做。

你可能希望在定义该宏时使用ASM_OUTPUT_SIZE_DIRECTIVE和/或ASM_OUTPUT_MEASURED_SIZE。

void TARGET_ASM_GLOBALIZE_LABEL (FILE *stream, const char *name) [Target Hook]

该target钩子为一个函数，用于将一些命令输出到stdio流stream中， 从而使得标号name为全局的；也就是，可以从其它文件中引用。

缺省实现依赖于GLOBAL_ASM_OP的适当定义。

void TARGET_ASM_GLOBALIZE_DECL_NAME (FILE *stream, tree decl) [Target Hook]

该target钩子为一个函数，用于将一些命令输出到stdio流stream中， 从而使得decl相关联的名字为全局的；也就是，可以从其它文件中引用。

缺省实现使用TARGET_ASM_GLOBALIZE_LABEL target钩子。

ASM_WEAKEN_LABEL (stream, name) [Macro]

一条C语句（没有分号），用于将一些命令输出到stdio流stream中， 从而使得标号name为弱的；也就是，可以从其它文件中引用， 但只有在没有其它定义的时候。 使用表达式assemble_name (stream, name)来输出name本身； 在此之前和之后，输出使得name为弱的额外的汇编语法，以及换行。

如果没有定义该宏或者ASM_WEAKEN_DECL， GCC将不支持弱符号并且你不要定义宏SUPPORTS_WEAK。

ASM_WEAKEN_DECL (stream, decl, name, value) [Macro]

组合（并替换）了函数ASM_WEAKEN_LABEL和ASM_OUTPUT_WEAK_ALIAS， 允许访问相关的函数或变量decl。如果value不为NULL， 该C语句应该将定义弱符号name具有值value的汇编代码，输出到stdio流stream中。如果value为NULL， 其应该输出命令来使得name为弱的。

ASM_OUTPUT_WEAKREF (stream, decl, name, value) [Macro]

输出一条伪指令，使得name被用来使用弱符号语义引用符号value。 decl为name的声明。

SUPPORTS_WEAK [Macro]

一个C表达式，如果target支持弱符号，则求值为真。

如果你没有定义该宏，`defaults.h'会提供一个缺省的定义。 如果ASM_WEAKEN_LABEL或者ASM_WEAKEN_DECL被定义， 则缺省定义为`1'；否则为`0'。如果你想使用编译器标记， 例如`-melf'，来控制弱符号的支持，则定义该宏。

MAKE_DECL_ONE_ONLY (decl) [Macro]

一条C语句（没有分号），用来标记decl作为public符号生成，这样在多个转换单元中额外的副本将被连接器丢弃。如果你的目标文件格式提供了这样的支持，例如在Microsoft Windows PE/COFF格式中的`COMDAT' section标记，并且这种支持需要对decl进行改动，例如将其放到独立的section中，则定义该宏。

SUPPORTS_ONE_ONLY [Macro]

一个C表达式，如果target支持one-only语义，则其求值为真。

如果你没有定义该宏，`varasm.c'会提供一个缺省的定义。如果MAKE_DECL_ONE_ONLY被定义，则缺省定义为`1'；否则为`0'。如果你想使用编译器标记来控制one-only符号的支持，或者如果设置DECL_ONE_ONLY标记就足以标记声明被作为one-only生成时，定义该宏。

void TARGET_ASM_ASSEMBLE_VISIBILITY (tree decl, const char *visibility) [Target Hook]

该target钩子为一个函数，用来将一些命令输出到asm_out_file，其将使得与decl相关的符号具有通过visibility指定的隐藏，保护或者内部可见的属性。

TARGET_WEAK_NOT_IN_ARCHIVE_TOC [Macro]

一个C表达式，如果target的连接器期望弱符号不出现在静态归档的目录表中，则其求值为真。缺省为0。

将弱符号置于归档的目录表之外，意味着如果符号将只在一个转换单元中有一个定义，并且有从其它转换单元中进行未定义的引用，则该符号将不为弱的。定义该宏为非零，将使得这样通常为弱的符号成为非弱的。

C++ ABI要求该宏为0。当target不能完全遵守C++ ABI，并且连接器要求弱符号在静态归档的目录表外面时，定义该宏。

ASM_OUTPUT_EXTERNAL (stream, decl, name) [Macro]

一条C语句（没有分号），用于将声明在该编译中被引用但没有被定义的外部符号名字name，所需要的任何文本输出到stdio流stream中。decl的值为声明的树结点。

如果不需要输出任何东西，则不需要定义该宏。GNU汇编器和大多数Unix汇编器不需要做任何事情。

void TARGET_ASM_EXTERNAL_LIBCALL (rtx symref) [Target Hook]

该target钩子是一个函数，用于将一条汇编伪指令输出到asm_out_file中，用以声明一个库函数名字为外部的。库函数的名字由symref给出，symref为一个symbol_ref。

void TARGET_ASM_MARK_DECL_PRESERVED (tree decl) [Target Hook]

该target钩子为一个函数，用于将一条汇编伪指令输出到asm_out_file中，用以注释使用的符号。Darwin target使用.no_dead_code_strip伪指令。

ASM_OUTPUT_LABELREF (stream, name) [Macro]

一条C语句（没有分号），用于将名为name的标号的引用的汇编语法输出到stdio流stream中。这应该在name前加上`_'。该宏用于assemble_name。

ASM_OUTPUT_SYMBOL_REF (stream, sym) [Macro]

一条C语句（没有分号），用于输出对SYMBOL_REF sym的引用。如果没有定义，assemble_name将被用来输出符号的名字。该宏可以用于修改符号被引用的方式，根据TARGET_ENCODE_SECTION_INFO的信息。

ASM_OUTPUT_LABEL_REF (stream, buf) [Macro]

一条C语句（没有分号），用于输出对ASM_GENERATE_INTERNAL_LABEL的结果， buf的引用。如果没有定义， assemble_name将被用来输出符号的名字。 该宏不被调用它的output_asm_label或者%l指示符使用；

void TARGET_ASM_INTERNAL_LABEL (FILE *stream, const char *prefix, unsigned long labelno) [Target Hook]

一个函数，将标号输出到stdio流stream中， 标号的名字由字符串prefix和编号labelno组成。

当然这些标号应该与用户级别函数和变量使用的标号不同。 否则，程序将具有与内部标号的命名冲突。

通常要求内部标号不包含在目标文件的符号表中。大多数汇编器具有命名约定， 来处理这些标号。在许多系统上，位于标号的起始处的字母`L'，具有这样的效果。 你应该找到你的系统使用的约定，并遵守。

该函数的缺省版本利用ASM_GENERATE_INTERNAL_LABEL了。

ASM_OUTPUT_DEBUG_LABEL (stream, prefix, num) [Macro]

一条C语句，用来将调试信息标号输出到stdio流stream中， 标号的名字由字符串prefix和编号num组成。这对VLIW target很有用， 因为调试信息标号可能需要与分支目标标号进行不同的处理。在一些系统上， 分支目标标号必须在指令束的起始处，但是调试信息标号可以出现在指令束的中间。

如果该宏没有被定义，则会使用(*targetm.asm_out.internal_label)。

ASM_GENERATE_INTERNAL_LABEL (string, prefix, num) [Macro]

一条C语句，用于将标号存储到字符串string中， 标号的名字由字符串prefix和编号num组成。

如果字符串起始于`*'， 则assemble_name将按照不改变剩余字符串的方式被输出。 这对于ASM_GENERATE_INTERNAL_LABEL非常方便。 如果字符串不起始于`*'，则会使用ASM_OUTPUT_LABELREF来输出字符串， 并且可能会做改变。（当然，ASM_OUTPUT_LABELREF也是你的机器描述的一部分， 所以你知道它在你的机器上做了什么。）

ASM_FORMAT_PRIVATE_NAME (outvar, name, number) [Macro]

一个C表达式，给outvar（类型为char *的变量）赋值一个新分配的字符串，该字符串由字符串name和编号number组成， 并增加适当的标点符号。使用alloca为字符串获得空间。

字符串将被ASM_OUTPUT_LABELREF作为参数使用， 来产生一个名字为name的内部静态变量的汇编标号。 因此，字符串必须为有效的汇编代码。参数number在每次执行该宏时都不相同； 其使得在不同作用域下的内部静态变量的名字不会有冲突。

理想情况下，该字符串应该不是一个有效的C标识符， 以阻止任何与用户自己的符号的冲突。大多数汇编器运行点或者百分号在汇编符号中； 在名字和编号之间加入至少一个这样的字符便可以。

如果该宏没有被定义，一个缺省的定义将被提供，其在大多数系统上都是正确的。

ASM_OUTPUT_DEF (stream, name, value) [Macro]

一条C语句，用于将定义符号name具有值value的汇编代码输出到stdio流stream中。

如果定义了SET_ASM_OP，一个缺省的定义将被提供， 其在大多数系统上都是正确的。

ASM_OUTPUT_DEF_FROM_DECLS (stream, decl_of_name, decl_of_value) [Macro]

一条C语句，用于将定义树结点为decl_of_name的符号， 具有树结点decl_of_value的值的汇编代码输出到stdio流stream中。 该宏将优先于`ASM_OUTPUT_DEF'被使用，如果其被定义，并且如果树结点有效。

如果定义了SET_ASM_OP，一个缺省的定义将被提供， 其在大多数系统上都是正确的。

TARGET_DEFERRED_OUTPUT_DEFS (decl_of_name, decl_of_value) [Macro]

一条C语句，如果定义树结点为decl_of_name的符号具有树结点为 decl_of_value的值的汇编代码，应该在当前编译单元结尾处被生成， 则求值为真。缺省为不推迟定义的输出。 该宏影响`ASM_OUTPUT_DEF' 和 `ASM_OUTPUT_DEF_FROM_DECLS'的定义输出。

ASM_OUTPUT_WEAK_ALIAS (stream, name, value) [Macro]

一条C语句，用来将定义弱符号name具有值value的汇编代码输出 到stdio流stream中。如果value为NULL， 其定义name为未定义的符号。

如果target只支持弱别名时，定义该宏；否则尽量定义ASM_OUTPUT_DEF。

OBJC_GEN_METHOD_LABEL (buf, is_inst, class_name, cat_name, sel_name) [Macro]

定义该宏来覆盖缺省的用于Objective-C方法的汇编名。

缺省名为一个唯一的方法编号，跟随class的名字（例如`_1_Foo'）。 对于在category中的方法，category也包含在汇编名中（例如`_1_Foo_Bar'）。

这些名字在大多数系统上是安全的，但是使得调试变得困难， 因为方法selector不在名字中。因此一些特定的系统定义了其它计算名字的方式。

buf为char *类型的表达式，给出一个缓存来存储名字； 其长度等于class_name，cat_name和sel_name的和， 再加上额外的50个字符。

参数is_inst指定了方法是一个实例方法，还是一个类方法； class_name为类的名字；cat_name为category的名字（或者为空， 如果方法不在category中）；sel_name为selector的名字。

在汇编器可以处理带引号的名字的系统上，你可以使用该宏来提供更加可读的名字。

ASM_DECLARE_CLASS_REFERENCE (stream, name) [Macro]

一条C语句（没有分号），用于将命令输出到stdio流stream中， 来声明标号name为Objective-C class引用。 这只在连接器具有对NeXT-style运行时的特殊支持的target上需要。

ASM_DECLARE_UNRESOLVED_REFERENCE (stream, name) [Macro]

一条C语句（没有分号），用于将命令输出到stdio流stream中， 来声明标号name为未解决的Objective-C class引用。 这只在连接器具有对NeXT-style运行时的特殊支持的target上需要。

17.21.5 如何处理初始化函数

一些语言被编译后的代码中会包含构造者（也称为初始化例程）---当程序启动时， 用来初始化程序中的数据的函数。这些函数需要在程序开始前被调用---也就是说， 在调用main之前。

编译一些语言还会生成析构者（也成为终止例程），将在程序终止时被调用。

为了让初始化和终止函数工作，编译器必须在汇编代码中输出一些东西， 使得那些函数在适当的时候被调用。当你将编译器移植到一个新的系统时， 你需要指定如何做。

GCC目前支持两种主要的方式，来执行初始化和终止函数。 每种方式都有两个变体。大多数结构体对于这四个变体都是通用的。

连接器必须构建两个这些函数的链表---一个是初始化函数链表， 叫做__CTOR_LIST__，一个是终止函数，叫做__DTOR_LIST__。

每个列表总是起始于一个被忽略的函数指针（其可能为0，-1， 或者之后的函数指针的个数，这取决于具体环境）。 随后是一系列指向构造者（或析构者）的零个或多个函数指针， 然后是一个包含0的函数指针。

取决于操作系统和它的可执行文件格式，或者`crtstuff.c'， 或者`libgcc2.c'会在起始时间和退出时间遍历这些列表。 构造者按照列表相反的顺序被调用；析构者按照向前的顺序。

处理静态构造者的最好的方式， 只有在目标文件格式提供任意命名section的时候才可以工作。 这会在构造者列表旁边设置一个section，析构者列表旁边设置另一个， 通常称作`.ctors'和`.dtors'。 每个定义了初始化函数的目标文件，还在构造section中放入一个字， 以指向那个函数。 连接器将所有这些字累积放入到一个邻近的`.ctors' section中。 终止函数按照类似的方式处理。

如果定义了TARGET_ASM_NAMED_SECTION， 则该方法将被`target-def.h'作为缺省方式选择。 不支持任意section，但是支持特定的构造者和析构者section的target， 可以定义CTORS_SECTION_ASM_OP和DTORS_SECTION_ASM_OP 来达到相同的效果。

当支持任意section时，有两个变体， 取决于如何调用`crtstuff.c'中的代码。 在支持.init section（其在程序起始时被执行）的系统上， `crtstuff.c'的部分代码被编译到那个section中。 程序按照类似于下面的方式被gcc驱动连接：

```
ld -o output_file crti.o crtbegin.o ... -lgcc crtend.o crtn.o
```

函数（__init）的序言出现在`crti.o'的.init section中； 尾声出现在`crtn.o'中。同样的，函数__fini在.fini section中。 通常这些文件由操作系统或者GNU C库提供，但GCC也为一些target提供。

目标文件`crtbegin.o'和`crtend.o'是从`crtstuff.c' 中编译出来的（对于大多target）。它们包含了，在.init和.fini section中的代码片断，用于跳转到.text section中的例程中。 连接器会把section的所有部分放在一起，形成完整的__init函数， 其可以在起始处调用我们需要的例程。

要使用这个变体，你必须适当的定义INIT_SECTION_ASM_OP宏。

如果init section不可用，GCC在编译任何叫做main（或者更加确切的说， 任何被语言前端expand_main_function指定为程序入口点的函数）的函数时， 其会插入一个调用__main的程序，以作为在函数序言之后首先执行的代码。 __main函数在`libgcc2.c'中定义，并运行全局的构造者。

对于文件格式不支持任意section的，也有两个变体。对于最简单的变体， 必须使用GNU连接器（GNU ld）和'a.out'格式。 在这种情况下， TARGET_ASM_CONSTRUCTOR被定义， 用来生成一个`N_SETT'类型的.stabs条目， 来引用名字__CTOR_LIST__， 并且其值为一个包含了初始化代码的void函数地址。 GNU连接器将其识别为一个要设定的值；该值会被累积， 并最终作为一个向量放在可执行程序中， 一个前导（被忽略的）数目和一个尾部的0元素。 TARGET_ASM_DESTRUCTOR按照类似的情况被处理。 由于init section不可用，所以使得编译main来调用__main， 以开始初始化。

最后一个变体既不使用任意section，也不使用GNU连接器。这适合于， 你想进行动态连接并使用GNU连接器不支持的文件格式的时候， 例如'ECOFF'。 在这种情况下， TARGET_HAVE_CTORS_DTORS为假， 初始化和终止函数简单的通过它们的名字来识别。这要求在连接过程中的额外程序， 叫做collect2。该程序作为GCC使用的连接器； 它通过运行普通的连接器来完成工作，但是还负责包含初始化和终止函数的向量。 这些函数通过__main调用。要使用这种方式， 必须在`config.gcc'中定义use_collect2。

17.21.6 控制初始化例程的宏

这里是控制编译器如何处理初始化和终止函数的宏：

INIT_SECTION_ASM_OP [Macro]

如果定义，为一个C字符串常量，包括空格， 用于将随后的数据作为初始化代码的汇编操作。 如果没有定义， GCC将假设这样的section不存在。当你使用用于初始化和终止函数的特定的section时， 该宏还控制`crtstuff.c'和`libgcc2.c'如何运行初始化函数。

HAS_INIT_SECTION [Macro]

如果定义，main将不会调用__main。对于控制起始代码， 像OSF/1这样的系统，应该定义该宏， 对于支持INIT_SECTION_ASM_OP的系统不应该显式的定义。

LD_INIT_SWITCH [Macro]

如果定义，为一个C字符串，作为一个开关，告诉连接器后面的符号为一个初始化例程。

LD_FINI_SWITCH [Macro]

如果定义，为一个C字符串常量，作为一个开关，告诉连接器后面的符号为一个结束例程。

COLLECT_SHARED_INIT_FUNC (stream, func) [Macro]

如果定义，为一条C语句，其将写一个在加载共享库时可以被自动调用的函数。 函数应该调用func，其不接受任何参数。如果没有定义， 并且目标格式要求显示的初始化函数，则将会生成一个叫做_GLOBAL__DI的函数。

该函数和下面的一个，被collect2使用，用于连接一个需要构造者或者析构者， 或者代码中具有DWARF2异常表嵌入的共享库的时候。

COLLECT_SHARED_FINI_FUNC (stream, func) [Macro]

如果定义，为一条C语句，其将写一个在卸载共享库时可以被自动调用的函数。 函数应该调用func，其不接受任何参数。如果没有定义， 并且目标格式要求显示的初始化函数，则将会生成一个叫做_GLOBAL__DD的函数。

INVOKE__main [Macro]

如果定义，main将调用__main， 而不管INIT_SECTION_ASM_OP的存在。对于init section不被自动运行， 但是仍可以用于搜集构建者和析构者列表的系统，该宏应该被定义。

SUPPORTS_INIT_PRIORITY [Macro]

如果非零，则支持C++ init_priority属性， 并且编译器应该生成指令来控制对象初始化的顺序。 如果为0，编译器遇到init_priority属性时，将产生一条错误信息。

bool TARGET_HAVE_CTORS_DTORS [Target Hook]

该值为真，如果target支持一些搜集构造者和析构者在起始和退出时运行的本地方法。 如果我们必须使用collect2，则为假。

void TARGET_ASM_CONSTRUCTOR (rtx symbol, int priority) [Target Hook]

如果定义，为一个函数，输出汇编代码来调用在初始化时symbol引用的函数。

假定symbol为一个没有参数并没有返回值的函数的SYMBOL_REF。 如果target支持初始化优先级， priority为一个0到MAX_INIT_PRIORITY之间的值； 否则，其必须为DEFAULT_INIT_PRIORITY。

如果该宏没有被taret定义，则一个适当的缺省将被选择， 如果（1）target支持任意section名，（2）target定义了CTORS_SECTION_ASM_OP， 或者（3）没有定义USE_COLLECT2。

void TARGET_ASM_DESTRUCTOR (rtx symbol, int priority) [Target Hook]

类似于TARGET_ASM_CONSTRUCTOR，不过用于终止函数，而不是初始化函数。

如果TARGET_HAVE_CTORS_DTORS为真， 对于生成的目标文件的初始化例程将具有静态连接。

如果你的系统使用collec2作为处理构造者的方法， 则那个程序通常使用nm来扫描目标文件，寻找被调用的构造者。

在一些特定的系统上，你可以定义该宏，使得collect2工作的更快。

OBJECT_FORMAT_COFF [Macro]

定义该宏，如果系统使用COFF（Common Object File Format）目标文件， 这样collect2能够假设为该格式，并扫描直接目标文件的构造/析构函数。

该宏只有在本地编译器上才有效率； 对于交叉编译器，collect2总是使用nm。

REAL_NM_FILE_NAME [Macro]

定义该宏为一个C字符串常量，包含用来执行nm的文件名。 缺省为搜索通常的路径。

如果你的系统支持共享库，并具有一个程序能够列出给定库或可执行程序的动态依赖， 你可以定义这些宏使得能够运行共享库中的初始化和终止函数。

LDD_SUFFIX [Macro]

定义该宏为一个C字符串常量，包含程序的名字，其可以列出动态依赖， 像SunOS 4中的"ldd"。

PARSE_LDD_OUTPUT (ptr) [Macro]

定义该宏为C代码，从LDD_SUFFIX指定的程序的输出中抽取文件名。 ptr为char *类型的变量，指向LDD_SUFFIX的输出中的一行。 如果行中列出的是动态依赖，则代码必须将ptr前进到那一行的文件名起始处。 否则，其必须设置ptr为NULL。

SHLIB_SUFFIX [Macro]

定义该宏为一个C字符串常量，包含了target缺省的共享库扩展名（例如`".so"'）。 当生成全局构造者和析构者名字时，collect2从该后缀的后面剥去版本信息。 该定义只在使用collect2来处理构造者和析构者的target上需要。

17.21.7 汇编指令的输出

这章描述了汇编指令的输出。

REGISTER_NAMES [Macro]

一段C初始化程序，包含了机器寄存器的汇编名字，每个名字使用C字符串常量表示。 这用来将编译器中的寄存器编号转换成汇编语言。

ADDITIONAL_REGISTER_NAMES [Macro]

如果定义，则为结构体数组的初始化程序，结构体包含了名字和寄存器编号。 该宏定义了硬件寄存器的附加名字，这样就可以允许在声明中的asm选项， 来使用附加名引用寄存器。

ASM_OUTPUT_OPCODE (stream, ptr) [Macro]

定义该宏，如果你在使用一个不常见的汇编器，其需要不一样的机器指令名字。

定义为C语句，输出一个汇编指令代码到标准输入输出流stream中。 宏操作数ptr为类型是char *的变量，其指向内部形式中的指令码名字， 内部形式使用机器描述来表示。该定义应该输出操作码名字到stream中， 执行你想要的任何转换，并且将变量ptr增加到指向opcode的尾部， 这样其才不会被输出两次。

实际上，相对于整个指令码的名字，你的宏定义可以处理或多或少的部分； 但是如果你想处理包含`%'序列的文本，则必须小心你所做的替换。 要保证增加ptr，使得不会输出不应该被输出的文本。

如果需要查看操作数的值，它们可以作为recog_data.operand的元素被找到。

如果宏定义不做任何事情，则指令使用通常的方式来输出。

FINAL_PRESCAN_INSN (insn, opvec, noperands) [Macro]

如果定义，则为一条C语句，其就在为insn输出汇编代码之前将被执行， 用来修改被抽取的操作数，从而可以被不同方式的输出。

这里的参数opvec为一个向量，包含了从insn中抽取的操作数， noperands为向量的元素个数。该向量的内容用于将insn模板转换成汇编代码， 所以你可以通过修改向量的内容来改变汇编输出。

该宏当有多个汇编语法共用一个指令模式文件时很有用；通过定义该宏， 你可以使大量类别的指令按照不同的方式输出（例如重组操作）。 自然的，影响单个insn模式的汇编语法， 应该通过在那些指令模式中写条件输出程序来处理。

如果没有定义该宏，则其相当于一条空语句。

PRINT_OPERAND (stream, x, code) [Macro]

C复合语句，用来将指令操作数x的汇编语法输出到标准输入输出流stream中。 x为RTL表达式。

code值可以用来指定打印操作数的方式。 用于当操作数必须根据上下文进行不同的打印的时候。 code来自用于打印操作数的`%'指定语句。 如果指定语句只是`%digit'，则code为0； 如果指定语句为`%ltrdigit'，则code为ltr的ASCII码。

如果x为寄存器，则该宏应该打印寄存器的名字。 名字可以在数组reg_names中找到，数组的类型为char *[]。 reg_names通过REGISTER_NAMES来初始化。

当机器描述具有一个`%punct'指定语句时（`%'后面跟随一个标点符号字符），则该宏被调用时，x为空指针， code为标点符号字符。

PRINT_OPERAND_PUNCT_VALID_P (code) [Macro]

一个C表达式， 当code为在PRINT_OPERAND宏中使用的有效的标点符号字符时， 其计算为真。 如果没有定义PRINT_OPERAND_PUNCT_VALID_P， 则意味着不以这种方式使用标点符号字符（除了标准的`%'以外）。

PRINT_OPERAND_ADDRESS (stream, x) [Macro]

C复合语句，用来将指令操作数为内存引用，其地址为x的汇编语法， 输出到标准输入输出流stream中。x为一个RTL表达式。

在一些机器上，符号地址的语法取决于地址所引用的section。在这些机器上， 定义钩子TARGET_ENCODE_SECTION_INFO来将信息存储到symbol_ref， 并在这里进行检查。 参见 Section 17.21 [汇编格式], page 357。

DBR_OUTPUT_SEQEND (file) [Macro]

C语句，在所有的栈槽填充指令被输出之后执行。 如果需要的话， 调用dbr_sequence_length来判定在序列中被填充的栈槽数目（如果当前不是输出一个序列，则为0），用来决定输出多少个no-ops，或其它。

如果不做任何事情，就不要定义该宏，但是如果将延迟序列显示化， 则会有助于阅读汇编输出（例如，使用空格）。

注意，用于带有延迟槽的指令的输出程序， 必须准备好处理没有被作为序列输出的情况（即，当没有运行调度过程， 或者没有找到栈槽填充者）。当没有处理序列时，变量final_sequence为空，否则其包含了被输出的rtx sequence。

REGISTER_PREFIX [Macro]
LOCAL_LABEL_PREFIX [Macro]
USER_LABEL_PREFIX [Macro]
IMMEDIATE_PREFIX [Macro]

如果定义，则为C字符串表达式，用于asm_fprintf（参见`final.c'）的选项`%R', `%L', `%U'和`%I'。 这在单个`md'文件必须支持多个汇编格式时很有用。这种情况下， 不同的`tm.h'文件可以定义不同的这些宏。

ASM_FPRINTF_EXTENSIONS (file, argptr, format) [Macro]

如果定义，该宏应该被扩展为一系列case语句， 其将在asm_fprintf函数中的switch语句里被解析。 这将应允许target来定义额外的printf格式，其在生成它们的汇编语句时很有帮助。 注意，大写字母被保留用于asm_fprintf将来的通用扩展， 所以不要用于target特定代码中。输出文件由参数file给定。 varargs输出指针为argptr，格式字符串的其余部分，由format指向。

ASSEMBLER_DIALECT [Macro]

如果你的target支持多个汇编语言方言（例如不同的操作码）， 可以定义该宏作为C表达式，给出汇编语言方言的索引，0作为第一个。

如果该宏被定义，你可以在指令模式的输出模版中（参见输出模版）或者asm_fprintf的第一个参数中使用如下的结构形式

```
`{option0|option1|option2...}'
```

该结构输出`option0', `option1', `option2'等等， 如果ASSEMBLER_DIALECT的值为0，1，2，等等。 这些字符串中的任何特殊字符将保留它们通常的含义。 如果括号中的可选项多于ASSEMBLER_DIALECT的值，则什么也不输出。

如果没有定义该宏，字符`{', `|'和`}'在模版中或asm_fprintf的操作数中不具有任何特殊含义。

如果你能够通过定义宏REGISTER_PREFIX, LOCAL_LABEL_PREFIX, USER_LABEL_PREFIX和IMMEDIATE_PREFIX来表达出汇编语言语法的 不同之处，则定义这些宏。如果语法差异比较大，涉及到操作码不同或操作数顺序， 则定义ASSEMBLER_DIALECT，使用`{option0|option1}'语法。

ASM_OUTPUT_REG_PUSH (stream, regno) [Macro]

C表达式，向stream中输出汇编代码，用于将硬件寄存器编号regno压入栈中。 代码不需要为最优的，因为该宏只在profiling的时候使用。

ASM_OUTPUT_REG_POP (stream, regno) [Macro]

C表达式，向stream中输出汇编代码，用于将硬件寄存器编号regno弹出栈中。 代码不需要为最优的，因为该宏只在profiling的时候使用。

17.21.8 派遣表的输出

这些是关于派遣表的。

ASM_OUTPUT_ADDR_DIFF_ELT (stream, body, value, rel) [Macro]

一条C语句，用来将汇编伪指令输出到stdio流stream中， 以生成两个标号间的距离。 value和rel为两个内部标号的编号。 这些标号的定义通过使用(*targetm.asm_out.internal_label)来输出， 并且它们必须使用相同的方式打印。例如，

```
fprintf (stream, "\t.word L%d-L%d\n",
    value, rel)
```

在一些机器上，派遣表中的地址是相对于表自己的地址，对此你必须提供该宏。 如果定义，GCC还将在生成PIC的所有机器上使用该宏。 body为ADDR_DIFF_VEC的主体；提供它使得可以读取模式和标记。

ASM_OUTPUT_ADDR_VEC_ELT (stream, value) [Macro]

在一些机器上，派遣表中的地址为绝对地址，对此应该提供该宏。

定义应该为一条C语句，用来将汇编伪指令输出到stdio流stream中， 以生成对一个标号的引用。value为内部标号的编号， 其定义应该使用(*targetm.asm_out.internal_label)输出。例如，

```
fprintf (stream, "\t.word L%d\n", value)
```

ASM_OUTPUT_CASE_LABEL (stream, prefix, num, table) [Macro]

定义该宏，如果在跳转表之前的标号需要被特殊输出。 前三个参数跟(*targetm.asm_out.internal_label)相同； 第四个参数为随后的跳转表（一个包含addr_vec或addr_diff_vec 的jump_insn）。

该特点用于system V，来为table输出一条swbeg语句。

如果没有定义该宏，这些标号使用(*targetm.asm_out.internal_label)输出。

ASM_OUTPUT_CASE_END (stream, num, table) [Macro]

定义该宏，如果在跳转表的结尾必须输出一些特殊的东西。定义应该为一条C语句， 在写完table的汇编代码后被执行。其应该将适当的代码写入到stdio流stream中。 参数table为jump-table insn，num为前面的标号的编号。

如果没有定义该宏，在跳转表的结尾不输出任何特殊的东西。

void TARGET_ASM_EMIT_UNWIND_LABEL (stream, decl, for_eh, empty) [Target Hook]

该target钩子在每个FDE的起始处生成一个标号。在一些target上，FDE需要特殊的标号， 对此应该定义该宏。其应该为函数声明decl相关联的FDE写入适当的标号到stdio流 stream中。第三个参数，for_eh，为一个布尔值；当是异常表时为真。 第四个参数empty，为一个布尔值；当是一个省略掉的FDE的占位符标号时为真。

缺省为FDE不给出非局部标号。

void TARGET_ASM_EMIT_EXCEPT_TABLE_LABEL (stream) [Target Hook]

该target钩子在异常表的起始处生成一个标号。在一些target上， 异常表需要根据函数被分割开，对此应该定义该宏。

缺省为不生成标号。

void TARGET_UNWIND_EMIT (FILE * stream, rtx insn) [Target Hook]

该target钩子生成需要展开给定指定的汇编伪指令。 这只在设置了TARGET_UNWIND_INFO的时候才被使用。

17.21.9 用于异常区域的汇编命令

这一节描述了标记异常区域的起始和结束的命令。

EH_FRAME_SECTION_NAME [Macro]

如果定义，为一个C字符串常量，包含异常处理帧展开信息的section名字。 如果没有定义，GCC将提供一个缺省定义，如果target支持命名section。 `crtstuff.c'使用该宏来切换到适当的section。

你应该定义该符号，如果你的target支持DWARF2帧展开信息并且缺省定义不工作。

EH_FRAME_IN_DATA_SECTION [Macro]

如果定义，DWARF2帧展开信息将被放到data section，即使target支持命名section。 例如当系统连接器进行垃圾搜集并且section不能被标记为不被搜集的时候， 可能需要这样。

不要定义该宏，除非TARGET_ASM_NAMED_SECTION也被定义。

EH_TABLES_CAN_BE_READ_ONLY [Macro]

定义该宏为1，如果你的target对于没有使用non-PIC代码编码的帧展开信息， 将总是要求运行时重定位， 但是连接器可能不支持将只读和读写section合并到单独的读写section中。

MASK_RETURN_ADDR [Macro]

一个rtx，用于对通过RETURN_ADDR_RTX发现的返回地址进行掩码操作， 使得其不包含任何无关的位。

DWARF2_UNWIND_INFO [Macro]

定义该宏为0，如果你的target支持DWARF2帧展开信息， 但是其还不能与异常处理一起工作。否则， 如果你的target支持这样的信息（如果定义了`INCOMING_RETURN_ADDR_RTX'， 并且`UNALIGNED_INT_ASM_OP'和`OBJECT_FORMAT_ELF'之一）， GCC将提供缺省的定义，1。

如果定义了TARGET_UNWIND_INFO，target特定的展开者将用于所有情况。 定义该宏将使得可以生成DWARF2帧调试信息。

如果没有定义TARGET_UNWIND_INFO，并且该宏被定义为1， 则DWARF2 unwider将为缺省的异常处理机制； 否则基于setjmp/longjmp的框架将被缺省使用。

TARGET_UNWIND_INFO [Macro]

定义该宏，如果你的target具有ABI指定的unwind表。 通常这些将由TARGET_UNWIND_EMIT输出。

Target Hook bool TARGET_UNWIND_TABLES_DEFAULT [Variable]

该变量应该被设为true，如果target ABI即使不使用异常的时候也要求展开表。

MUST_USE_SJLJ_EXCEPTIONS [Macro]

该宏只有当DWARF2_UNWIND_INFO为运行时变量时才需要被定义。那种情况下， `except.h'不能正确的确定MUST_USE_SJLJ_EXCEPTIONS的相应定义， 所以target必须直接提供。

DONT_USE_BUILTIN_SETJMP [Macro]

定义该宏为1，如果基于setjmp/longjmp的框架应该使用C库中的 setjmp/longjmp， 而不是__builtin_setjmp/__builtin_longjmp。

DWARF_CIE_DATA_ALIGNMENT [Macro]

该宏只有当target可能会在函数序言中存储寄存器， 并且相对栈指针的偏移量没有对齐于UNITS_PER_WORD的时候，才需要被定义。 定义应该为负的最小对齐数，如果STACK_GROWS_DOWNWARD被定义， 否则为正的最小对齐数。参见 Section 17.22.5 [SDB和DWARF], page 381。 只有当target支持DWARF2帧展开信息的时候才有用。

Target Hook bool TARGET_TERMINATE_DW2_EH_FRAME_INFO [Variable]
值为真，如果target应该增加一个0字到Dwarf-2帧信息section的结尾， 当用于异常处理时。 缺省值为假，如果EH_FRAME_SECTION_NAME被定义，否则为真。

rtx TARGET_DWARF_REGISTER_SPAN (rtx reg) [Target Hook]
给定一个寄存器，该钩子应该返回一个并行的寄存器，来表示到哪里发现寄存器块。 定义该钩子，如果寄存器和它的机器模式在Dwarf中被表示为非连接的位置， 或者如果在Dwarf中寄存器应该被表示为多于一个寄存器。 否则该钩子应该返回NULL_RTX。如果没有定义，缺省为返回NULL_RTX。

void TARGET_INIT_DWARF_REG_SIZES_EXTRA (tree address) [Target Hook]
如果一些寄存器在Dwarf-2展开信息中按照多个块来表示， 定义该钩子在运行时来填充信息。 其将被expand_builtin_init_dwarf_reg_sizes调用。address为表的地址。

bool TARGET_ASM_TTYPE (rtx sym) [Target Hook]
该钩子被用于从帧展开表中输出一个引用到由sym标识的type_info对象中。 其应该返回true，如果引用被输出。 返回false将会造成引用使用通常的Dwarf2例程被输出。

bool TARGET_ARM_EABI_UNWINDER [Target Hook]
该钩子应该被设为true，在使用基于ARM EABI的展开库的target上， 并且在其它target上为false。这将影响展开表的格式。缺省为false。

17.21.10 用于对齐的汇编命令

这一节描述了用于对齐的命令。

JUMP_ALIGN (label) [Macro]
在label前放入的对齐数（基于2的log），label为跳转的普通目的地并且不具有fallthru入边。

如果你目前不想做任何特殊的对齐，则不需要定义该宏。 大多数机器描述目前都没有定义该宏。

除非需要检查label参数， 最好在target的OVERRIDE_OPTIONS中设置变量align_jumps。 否则应该尽量在JUMP_ALIGN实现中尊重用户在align_jumps中的选择。

LABEL_ALIGN_AFTER_BARRIER (label) [Macro]
在label前放入的对齐数（基于2的log），label后跟随一个BARRIER。

如果你目前不想做任何特殊的对齐，则不需要定义该宏。 大多数机器描述目前都没有定义该宏。

LABEL_ALIGN_AFTER_BARRIER_MAX_SKIP [Macro]
当应用LABEL_ALIGN_AFTER_BARRIER时，要跳过的最大字节个数。 这只在定义了ASM_OUTPUT_MAX_SKIP_ALIGN时才起作用。

LOOP_ALIGN (label) [Macro]
在label前放入的对齐数（基于2的log），label后跟随一个NOTE_INSN_LOOP_BEG注解。

如果你目前不想做任何特殊的对齐，则不需要定义该宏。 大多数机器描述目前都没有定义该宏。

除非需要检查label参数， 最好在target的OVERRIDE_OPTIONS中设置变量align_loops。 否则应该尽量在LOOP_ALIGN实现中尊重用户在align_loops中的选择。

LOOP_ALIGN_MAX_SKIP [Macro]
当应用LOOP_ALIGN时，要跳过的最大字节个数。 这只在定义了ASM_OUTPUT_MAX_SKIP_ALIGN时才起作用。

LABEL_ALIGN (label) [Macro]
在label前放入的对齐数（基于2的log），如果LABEL_ALIGN_AFTER_BARRIER / LOOP_ALIGN指定了不相同的对齐，则使用最大的值。

除非需要检查label参数，最好在target的OVERRIDE_OPTIONS中设置变量align_labels。 否则应该尽量在LABEL_ALIGN实现中尊重用户在align_labels中的选择。

LABEL_ALIGN_MAX_SKIP [Macro]
当应用LABEL_ALIGN时，要跳过的最大字节个数。 这只在定义了ASM_OUTPUT_MAX_SKIP_ALIGN时才起作用。

ASM_OUTPUT_SKIP (stream, nbytes) [Macro]
一条C语句，将一条汇编指令输出到stdio流stream中，使得将位置计数器前移nbytes个字节。那些字节在加载时应该为0。 nbytes将为unsigned HOST_WIDE_INT类型的C表达式。

ASM_NO_SKIP_IN_TEXT [Macro]
定义该宏，如果ASM_OUTPUT_SKIP不应该在text section中使用，因为无法在跳过的字节中放入0。这在许多Unix系统上都为，当在text section中使用时，跳过字节的伪指令会产生no-op指令，而不是0。

ASM_OUTPUT_ALIGN (stream, power) [Macro]
一条C语句，将一条汇编指令输出到stdio流stream中，使得将位置计数器前移2的power次幂个字节。 power为int类型的表达式。

ASM_OUTPUT_ALIGN_WITH_NOP (stream, power) [Macro]
类似ASM_OUTPUT_ALIGN，除了使用“nop”指令来填充，如果需要的话。

ASM_OUTPUT_MAX_SKIP_ALIGN (stream, power, max_skip) [Macro]
一条C语句，将一条汇编指令输出到stdio流stream中，使得将位置计数器前移2的power次幂个字节，但只有当需要max_skip个或者更少的字节来满足对齐要求的时候才行。 power和max_skip为int类型的表达式。

17.22 控制调试信息格式

描述了如何指定调试信息。

17.22.1 影响所有调试格式的宏

这些宏影响所有的调试格式。

DBX_REGISTER_NUMBER (regno) [Macro]

一个C表达式，返回编译器寄存器号regno的DBX寄存器号。 在提供的缺省宏中，该表达式的值将为regno本身。 但是有时候，有些寄存器编译器知道但是DBX不知道，或者相反。 对于这种情况，寄存器可能需要在编译器中具有一个编号， 而在DBX中具有另一个。

如果两个寄存器在GCC中具有连续的编号， 并且可以作为一对用来保存多字的值， 则它们在使用DBX_REGISTER_NUMBER重新编号之后 必须还具有连续的编号。 否则，调试器将无法访问这样的寄存器对， 因为调试器期望寄存器对在自己的编号方案下也是连续的。

如果你发现自己定义的DBX_REGISTER_NUMBER不能保持寄存器对， 则必须重定义实际的寄存器编号方案。

DEBUGGER_AUTO_OFFSET (x) [Macro]

一个C表达式，返回具有地址x（RTL表达式）的自动变量的整数偏移量。 缺省计算是假设x基于帧指针的，并会给出相对帧指针的偏移量。 这对于产生DBX调试输出或COFF风格的SDB调试输出， 并且在使用`-g'选项时也允许消除帧指针的目标，会有需要。

DEBUGGER_ARG_OFFSET (offset, x) [Macro]

一个C表达式，返回具有地址x（RTL表达式）的参数的整数偏移量。 名义上的偏移量为offset。

PREFERRED_DEBUGGING_TYPE [Macro]

一个C表达式，返回当用户只指定`-g'时， GCC应该产生的调试输出类型。 如果你已经使得GCC支持多种调试输出格式的时候，则定义该宏。 目前，允许的值为DBX_DEBUG, SDB_DEBUG, DWARF_DEBUG, DWARF2_DEBUG, XCOFF_DEBUG, VMS_DEBUG和VMS_AND_DWARF2_DEBUG。

当用户指定`-ggdb'时，GCC通常还使用该宏的值来选择调试输出格式， 但是有两个例外。 如果定义了DWARF2_DEBUGGING_INFO， 则GCC使用值DWARF2_DEBUG， 如果定义了DBX_DEBUGGING_INFO， 则GCC使用DBX_DEBUG。

该宏的值只影响缺省调试输出；用户总是可以通过使用`-gstabs', `-gcoff', `-gdwarf-2', `-gxcoff'或`-gvms'来获得指定类型的输出。

17.22.2 用于DBX输出的特定选项

这些是用于DBX输出的特定选项。

DBX_DEBUGGING_INFO [Macro]

定义该宏，如果对于`-g'选项，GCC应该产生DBX的调试输出。

XCOFF_DEBUGGING_INFO [Macro]

定义该宏，如果对于`-g'选项，GCC应该产生XCOFF格式的调试输出。 这是DBX格式的变体。

DEFAULT_GDB_EXTENSIONS [Macro]

定义该宏来控制GCC是否缺省的生成GDB的扩展版本DBX调试信息（假设是使用DBX格式的调试信息）。如果没有定义该宏，则缺省为1： 如果可能的话，总是生成扩展信息。

DEBUG_SYMS_TEXT [Macro]
定义该宏，如果所有.stabs命令应该输出在text section中。

ASM_STABS_OP [Macro]
一个C字符串常量，包括空格， 命名了用来替代"\t.stabs\t"的汇编伪指令， 来定义一个普通调试符号。 如果没有定义该宏，则使用"\t.stabs\t"。 该宏只用于DBX调试信息格式。

ASM_STABD_OP [Macro]
一个C字符串常量，包括空格， 命名了用来替代"\t.stabd\t"的汇编伪指令， 来定义一个普通调试符号。 如果没有定义该宏，则使用"\t.stabd\t"。 该宏只用于DBX调试信息格式。

ASM_STABN_OP [Macro]
一个C字符串常量，包括空格， 命名了用来替代"\t.stabn\t"的汇编伪指令，来定义一个普通调试符号。 如果没有定义该宏，则使用"\t.stabn\t"。该宏只用于DBX调试信息格式。

DBX_NO_XREFS [Macro]
定义该宏，如果DBX在你的系统上不支持`xstagname'结构。 在一些系统上，该结构被用于描述对名字叫tagname的结构体的向前引用。 在其它系统上，该结构根本不被支持。

DBX_CONTIN_LENGTH [Macro]
DBX格式的调试信息中的符号名被连续的处理（分成两个独立的.stabs指令）当其达到一个特定长度时（缺省为80个字符）。 在一些操作系统上，DBX 要求做这样的分割；在其它上面，则不能进行分割。 你可以通过定义该宏为0来约束分割。 你可以覆盖缺省的分割长度，通过定义该宏为一个长度的表达式。

DBX_CONTIN_CHAR [Macro]
通常通过在.stabs字符串的结尾增加一个`\'字符来表示连续的处理。 要是用不用的字符，定义该宏为一个你想使用的字符常量。 不要定义该宏，如果反斜杠在你的系统上是正确的。

DBX_STATIC_STAB_DATA_SECTION [Macro]
定义该宏，如果需要在为非全局的静态变量输出`.stabs'伪操作符之前进入data section。

DBX_TYPE_DECL_STABS_CODE [Macro]
对于typedef，.stabs指令的“code”域所使用的值。 缺省为N_LSYM。

DBX_STATIC_CONST_VAR_CODE [Macro]
对于位于text section的静态变量，.stabs指令的“code”域所使用的值。 DBX格式不提供任何正确是方式。缺省为N_FUN。

DBX_REGPARM_STABS_CODE [Macro]
对于寄存器中传递的参数，.stabs指令的“code”域所使用的值。 DBX格式不提供任何正确是方式。缺省为N_RSYM。

DBX_REGPARM_STABS_LETTER [Macro]
在DBX符号数据中使用的字母，用来标识一个符号为在寄存器中传递的参数。 DBX格式目前没有提供任何这样做的方式。缺省为'P'。

DBX_FUNCTION_FIRST [Macro]
定义该宏，如果对于函数和它的参数的DBX信息应该位于函数的汇编代码之前。 通常，在DBX格式中，调试信息完全位于汇编代码之后。

DBX_BLOCKS_FUNCTION_RELATIVE [Macro]

定义该宏为1，如果描述块（N_LBRAC或N_RBRAC）的作用域的符号的值，应该相对于函数括号的起始处。通常GCC使用绝对值。

DBX_LINES_FUNCTION_RELATIVE [Macro]

定义该宏为1，如果指示当前行（N_SLINE）的符号的值，应该相对于函数括号的起始处。通常GCC使用绝对值。

DBX_USE_BINCL [Macro]

定义该宏，如果GCC应该为被包含的头文件生成N_BINCL和N_EINCL stabs，如Sun系统。通常，GCC不生成N_BINCL或N_EINCL stabs。

17.22.3 针对DBX格式的钩子

这些是针对DBX格式的钩子。

DBX_OUTPUT_LBRAC (stream, name) [Macro]

定义该宏，以告知如何将变量名的作用域级别的起始调试信息输出到stream中。参数name为一个汇编符号的名字（使用assemble_name），其值为作用域起始的地址。

DBX_OUTPUT_RBRAC (stream, name) [Macro]

类似DBX_OUTPUT_LBRAC，不过是作用域级别的结尾。

DBX_OUTPUT_NFUN (stream, lscope_label, decl) [Macro]

定义该宏，如果target机器要求对函数decl输出一个N_FUN条目进行特殊处理。

DBX_OUTPUT_SOURCE_LINE (stream, line, counter) [Macro]

一条C语句，将DBX调试信息在当前源文件行号line的代码前，输出到stdio流stream中。counter为宏被调用的次数，包括当前调用；其用于在汇编输出中生成为一个标号。

如果缺省输出是正确的，或者其能够通过定义DBX_LINES_FUNCTION_RELATIVE而变的正确，则不要定义该宏。

NO_DBX_FUNCTION_END [Macro]

一些stab封装格式（特别是ECOFF），不能处理. stabs "",N_FUN,,0,0,Lscope-function-1 gdb dbx扩展结构。在那些机器上，定义该宏来关掉这个特点，并且不影响其它gdb扩展。

NO_DBX_BNSYM_ENSYM [Macro]

一些汇编器不能处理. stabd BNSYM/ENSYM,0,0 gdb dbx扩展结构。在那些机器上，定义该宏来关掉这个特点，并且不影响其它gdb扩展。

17.22.4 DBX格式的文件名

这一节描述了DBX格式的文件名。

DBX_OUTPUT_MAIN_SOURCE_FILENAME (stream, name) [Macro]

一条C语句，将DBX调试信息输出到标准输入输出（stdio）流stream上，其中文件name为主源文件——被指定为被编译的输入文件。该宏只被调用一次，在编译的开始处。

如果DBX调试信息输出的标准形式合适，则该宏不需要被定义。

有时可能需要引用相当与text段起始处的标号。这时可以使用`assemble_name (stream, ltext_label_name)'来完成。如果这样做，则必须还要将变量used_ltext_label_name设为true。

NO_DBX_MAIN_SOURCE_DIRECTORY [Macro]

定义该宏的值为1， 如果不让GCC在文件起始处产生对当前的编译目录和当前的源语言的指示。

NO_DBX_GCC_MARKER [Macro]

定义该宏的值为1，如果不让GCC产生该目标文件是由GCC编译的指示。 缺省情况是在每个源文件的起始处产生一个N_OPT stab， 其中字符串为`gcc2_compiled.'，值为0。

DBX_OUTPUT_MAIN_SOURCE_FILE_END (stream, name) [Macro]

一条C语句，用来在主源文件name的编译结尾输出DBX调试信息。 输出将被写入标准输入输出流stream中。

如果没有定义该宏，则在编译的结尾将不做任何特定的输出， 这对于大多数机器都是正确的。

DBX_OUTPUT_NULL_N_SO_AT_MAIN_SOURCE_FILE_END [Macro]

定义该宏而不是定义DBX_OUTPUT_MAIN_SOURCE_FILE_END， 如果要在编译结尾输出的是一个N_SO stab， 其具有空字符串，值为文件中最高的绝对text地址。

17.22.5 用于SDB和DWARF输出的宏

这些是用于SDB和DWARF输出的宏。

SDB_DEBUGGING_INFO [Macro]

定义该宏，如果对于`-g'选项，GCC应该为SDB产生COFF风格的调试输出。

DWARF2_DEBUGGING_INFO [Macro]

定义该宏，如果对于`-g'选项，GCC应该产生dwarf本版2格式的调试输出。

int TARGET_DWARF_CALLING_CONVENTION (tree function) [Target Hook]

定义该钩子，使得为个函数输出dwarf属性DW_AT_calling_convention。 返回DW_CC_标记的enum值。

要支持可选的调用帧调试信息， 你必须还要定义INCOMING_RETURN_ADDR_RTX，并如果在序言中使用RTL， 则设置RTX_FRAME_RELATED_P，或者如果没有使用RTL， 则从TARGET_ASM_FUNCTION_PROLOGUE中调用dwarf2out_def_cfa 和dwarf2out_reg_save。

DWARF2_FRAME_INFO [Macro]

定义该宏为非0值，如果GCC应该总是输出Dwarf2帧信息。 如果DWARF2_UNWIND_INFO（参见 Section 17.21.9 [异常区域输出], page 374 ）为非0， 则GCC将不管你如何定义DWARF2_FRAME_INFO，都会输出该信息。

DWARF2_ASM_LINE_DEBUG_INFO [Macro]

定义该宏为非0值，如果汇编器能够生成Dwarf2行调试信息section。

ASM_OUTPUT_DWARF_DELTA (stream, size, label1, label2) [Macro]

一条C语句，来输出汇编伪指令， 以创建一个lab1减去lab2的差，使用给定的size。

ASM_OUTPUT_DWARF_OFFSET (stream, size, label, section) [Macro]

一条C语句，来输出汇编伪指令， 以创建一个相对的section的给定label的引用， 使用给定的size。label为在给定section中定义的label。

ASM_OUTPUT_DWARF_PCREL (stream, size, label) [Macro]
一条C语句，来输出汇编伪指令， 以创建一个给定label的引用，使用给定的size。

void TARGET_ASM_OUTPUT_DWARF_DTPREL (FILE *FILE, int size, rtx x) [Target Hook]
如果定义，该target钩子为一个函数，其输出一个相对DTP的引用，对给定的TLS符号。

PUT_SDB_... [Macro]
定义这些宏来覆盖汇编语法，为特定的SDB汇编伪指令。 参见`sdbout.c'，关于这些宏和它们的参数的列表。 如果使用标准语法，你不需要定义它们。

SDB_DELIM [Macro]
一些汇编器不支持分号作为分隔符，即使在SDB汇编伪指令之间。 这种情况下，定义该宏为要使用的分隔符（通常为`\n'）。 如果只需要改变该宏，则不需要定义新的PUT_SDB_op宏集合。

SDB_ALLOW_UNKNOWN_REFERENCES [Macro]
定义该宏以允许对未知结构体，联合体和枚举标记的引用，被输出。 标准的COFF不允许处理未知的引用，MIPS ECOFF支持该特定。

SDB_ALLOW_FORWARD_REFERENCES [Macro]
定义该宏以允许对未遇到的结构体，联合体和枚举标记的引用，被输出。 一些汇编器对此会出错。

SDB_OUTPUT_SOURCE_LINE (stream, line) [Macro]
一条C语句，输出当前源文件的行号的SDB调试信息在代码前。 缺省为输出一个.ln伪指令。

17.22.6 用于VMS调试格式的宏

这是用于VMS调试格式的宏。

VMS_DEBUGGING_INFO [Macro]
定义该宏，如果GCC应该为`-g'选项产生VMS调试输出。 VMS的缺省行为是在没有`-g'时，生成可以回溯的最少调试信息， 除非使用`-g0'显示的覆盖。 该行为由OPTIMIZATION_OPTIONS和OVERRIDE_OPTIONS控制。

17.23 交叉编译和浮点

虽然所有现代机器都使用二进制补码来表示整数，但对于浮点数却有不同的表示。 这意味着在交叉编译器中， 被编译的程序中的浮点数的表示可能与执行编译的机器上的表示不相同。

因为不同的表示方式可能会提供不同的取值范围和精度， 所以所有的浮点常量必须被表示成target机器的格式。 因此，交叉编译器不能使用host机器的浮点算术；其必须模拟target的算术运算。 为了确保一致性，GCC总是使用模拟方式来处理浮点值， 即使host和target的浮点格式相同。

下列宏由`real.h'提供给编译器使用。 编译器的生成或者优化浮点计算的所有部分必须使用这些宏。 它们可能会计算操作数多次，所以操作数一定不要有副作用。

REAL_VALUE_TYPE [Macro]
C数据类型，用于存放target机器格式的浮点值。 通常为一个包含HOST_WIDE_INT型数组的结构体， 但是所有的代码应该将其作为不透明的量。

int REAL_VALUES_EQUAL (REAL_VALUE_TYPE x, REAL_VALUE_TYPE y) [Macro]
比较两个值是否相等，x和y。 如果target浮点格式支持负0和/或NaN， 则`REAL_VALUES_EQUAL (-0.0, 0.0)'为真,`REAL_VALUES_EQUAL (NaN, NaN)'为假。

int REAL_VALUES_LESS (REAL_VALUE_TYPE x, REAL_VALUE_TYPE y) [Macro]
测试x是否小于y。

HOST_WIDE_INT REAL_VALUE_FIX (REAL_VALUE_TYPE x) [Macro]
将x截取为有符号整数，向0舍入。

unsigned HOST_WIDE_INT REAL_VALUE_UNSIGNED_FIX (REAL_VALUE_TYPE x) [Macro]
将x截取为无符号整数，向0舍入。如果x为负，则返回0。

REAL_VALUE_TYPE REAL_VALUE_ATOF (const char *string, enum machine_mode mode) [Macro]
将string转换为target机器模式mode所表示的浮点数。该程序可以处理十进制和十六进制的浮点常量，使用C语言定义的语法。

int REAL_VALUE_NEGATIVE (REAL_VALUE_TYPE x) [Macro]
如果x为负数（包括负零），返回1，否则0。

int REAL_VALUE_ISINF (REAL_VALUE_TYPE x) [Macro]
确定x是否表示无穷（正的或负的）。

int REAL_VALUE_ISNAN (REAL_VALUE_TYPE x) [Macro]
确定x是否表示“NaN”（not-a-number）。

void REAL_ARITHMETIC (REAL_VALUE_TYPE output, enum tree_code code, REAL_VALUE_TYPE x, REAL_VALUE_TYPE y) [Macro]
计算一个算术运算对浮点值x和y，将结果存到output（其必须为一个变量）中。

要执行的运算由code指定。只有下列代码被支持：PLUS_EXPR, MINUS_EXPR, MULT_EXPR, RDIV_EXPR, MAX_EXPR, MIN_EXPR。

如果REAL_ARITHMETIC被要求计算除0，并且target的浮点格式不能表示无穷，则会调用abort。调用者应该首先检查这种情况，使用MODE_HAS_INFINITIES。See Section 17.5 [存储布局], page 291.

REAL_VALUE_TYPE REAL_VALUE_NEGATE (REAL_VALUE_TYPE x) [Macro]
返回浮点值x的负数。

REAL_VALUE_TYPE REAL_VALUE_ABS (REAL_VALUE_TYPE x) [Macro]
返回x的绝对值。

REAL_VALUE_TYPE REAL_VALUE_TRUNCATE (REAL_VALUE_TYPE mode, enum machine_mode x) [Macro]
将浮点值x截取为适合mode。返回值仍然是REAL_VALUE_TYPE，但是具有一个合适的位模式，其精度根据模式mode来输出浮点常量。

void REAL_VALUE_TO_INT (HOST_WIDE_INT low, HOST_WIDE_INT high, REAL_VALUE_TYPE x) [Macro]
转换浮点值x为双精度整数，存储到low和high中。如果值不是整形的，则进行截取。

void REAL_VALUE_FROM_INT (REAL_VALUE_TYPE x, HOST_WIDE_INT low, HOST_WIDE_INT high, enum machine_mode mode) [Macro]
转换在low和high中的一个双精度整数为浮点值，其被存储在x中。值被截取以适合机器模式mode。

17.24 机器模式切换指令

下列的宏用来控制模式切换优化：

OPTIMIZE_MODE_SWITCHING (entity) [Macro]

定义该宏，如果在优化编译中，port需要为机器模式切换插入额外的指令。

例如，SH4可以执行单精度和双精度的浮点运算，但是执行单精度运算时，必须清除FPSCR PR位，而执行双精度运算时，必须设置该位。改变PR位需要一个通用寄存器来作为草稿寄存器，因此这些FPSCR设置必须在重载之前被插入，即你不能将它放在指令输出或者TARGET_MACHINE_DEPENDENT_REORG阶段。

你可以具有多个具有模式切换的实体，并且在运行时选择哪些实体实际需要。对于任何需要模式切换的entity，OPTIMIZE_MODE_SWITCHING应该返回非零。如果你定义了该宏，你还必须定义NUM_MODES_FOR_MODE_SWITCHING, MODE_NEEDED, MODE_PRIORITY_TO_MODE和EMIT_MODE_SET。MODE_AFTER, MODE_ENTRY和MODE_EXIT是可选的。

NUM_MODES_FOR_MODE_SWITCHING [Macro]

如果你定义了OPTIMIZE_MODE_SWITCHING，你必须定义该宏，作为整数数组的初始化。每个初始化元素N引用一个需要模式切换的实体，并且指定了该实体可能需要被设置的不同模式的数目。初始化的位置——起始于0——确定了被用于引用有问题的模式切换实体的整数。

MODE_NEEDED (entity, insn) [Macro]

entity为一个整数指定了模式切换的实体。如果定义了OPTIMIZE_MODE_SWITCHING，则必须定义该宏来返回一个不大于在NUM_MODES_FOR_MODE_SWITCHING中相应元素的整数值，来指示entity在执行insn前必须被切换成的模式。

MODE_AFTER (mode, insn) [Macro]

如果定义该宏，其在模式切换过程中对于每个insn进行求值。其确定一个insn的结果时的模式（如果与输入时的模式不同）。

MODE_ENTRY (entity) [Macro]

如果定义该宏，其对每个需要模式切换的entity进行求值。结果为一个整数，为entity在函数入口处被假定切换成的模式。如果定义了MODE_ENTRY，则必须定义MODE_EXIT。

MODE_EXIT (entity) [Macro]

如果定义该宏，其对每个需要模式切换的实体进行求值。结果为一个整数，为实体在函数出口处被假定切换成的模式。如果定义了MODE_EXIT，则必须定义MODE_ENTRY。

MODE_PRIORITY_TO_MODE (entity, n) [Macro]

该宏指定了被处理的entity的模式顺序。0为最高优先级，NUM_MODES_FOR_MODE_SWITCHING[entity] - 1为最低。宏的值应该为一个整数，表示entity的一个模式。对于任何固定的entity，mode_priority_to_mode(entity, n))应该为0...num_modes_for_mode_switching[entity] - 1之间的双向影射。

EMIT_MODE_SET (entity, mode, hard_regs_live) [Macro]

生成一个或多个insn来将entity设为mode。hard_reg_live是在insn被插入点处的硬件寄存器活跃集。

17.25 定义目标机特定的__attribute__用法

可以为函数，数据和类型定义target特定的属性。 这些使用下列target钩子来描述；它们还需要在`extend.texi'中被记述。

const struct attribute_spec * TARGET_ATTRIBUTE_TABLE [Target Hook]

如果定义， 该目标钩子指向一个`struct attribute_spec' (在`tree.h'中定义) 数组， 用来指定该目标的机器特定的属性， 以及这些属性被应用到的实体和它们接受的参数的一些限制。

int TARGET_COMP_TYPE_ATTRIBUTES (tree type1, tree type2) [Target Hook]

如果定义，该目标钩子为一个函数， 如果type1和type2的属性不匹配则返回0，匹配则返回1，几乎匹配则返回2（这将产生一个warning）。如果没有被定义， 则机器特定的属性总被假定为匹配的。

void TARGET_SET_DEFAULT_TYPE_ATTRIBUTES (tree type) [Target Hook]

如果定义，该目标钩子为一个函数，其将缺省属性赋予新定义的类型type。

tree TARGET_MERGE_TYPE_ATTRIBUTES (tree type1, tree type2) [Target Hook]

定义该target钩子，如果合并类型属性需要进行特殊的处理。 如果定义，则结果为type1和type2的组合TYPE_ATTRIBUTES列表。 其假设comptypes总是被调用并返回1。 该函数可以调用merge_attributes来处理机器无关的合并。

tree TARGET_MERGE_DECL_ATTRIBUTES (tree olddecl, tree newdecl) [Target Hook]

定义该target钩子，如果合并decl属性需要进行特殊的处理。如果定义， 则结果为olddecl和newdecl的组合DECL_ATTRIBUTES列表。 newdecl为olddecl的拷贝。这样的例子是当一个属性覆盖另一个， 或者当一个属性被后续的属性置空的情况。 该函数可以调用merge_attributes来处理机器无关的合并。

如果唯一需要target特定的处理是Microsoft Windows target的`dllimport'，则你应该定义宏TARGET_DLLIMPORT_DECL_ATTRIBUTES为1。 然后编译器将会定义一个叫做merge_dllimport_decl_attributes的函数， 其可以被定义为TARGET_MERGE_DECL_ATTRIBUTES的扩展。 你还可以为你的port在属性表中增加handle_dll_attribute， 来执行`dllimport'和`dllexport'属性的初始化处理。 例如，在`i386/cygwin.h'和`i386/i386.c'中。

bool TARGET_VALID_DLLIMPORT_ATTRIBUTE_P (tree decl) [Target Hook]

decl为一个指定为__attribute__((dllimport))的变量或者函数。 使用该钩子，如果target需要给handle_dll_attribute增加额外的有效性检查。

TARGET_DECLSPEC [Macro]

定义该宏为非零， 如果你想将__declspec(X)与__attribute((X))等同对待。 缺省下，只有在定义了TARGET_DLLIMPORT_DECL_ATTRIBUTES的target上才可以。 目前对于__declspec的实现是通过一个内建的宏，但是你不应该依赖于实现细节。

void TARGET_INSERT_ATTRIBUTES (tree node, tree *attr_ptr) [Target Hook]

定义该target钩子，如果你想在decl被创建时，能够为其增加属性。 这在后端想要实现一个pragma，并且用到与pragma相关的属性的时候， 通常很有用。参数node是正在创建的decl。 参数attr_ptr是指向该decl的属性列表的指针。不要修改列表本身， 因为其可能与其它decl共享，但是可以将属性链接到列表的头部， 并且修改*attr_ptr以指向新的属性，或者如果需要进一步的修改， 创建一个列表的拷贝。

bool TARGET_FUNCTION_ATTRIBUTE_INLINABLE_P (tree fndecl) [Target Hook]

该target钩子返回true，如果可以将fndecl内联到当前函数中，而不管它具有的target特定属性，否则为false。缺省下，如果函数具有一个target特定属性，则不会被内联。

bool TARGET_VALID_OPTION_ATTRIBUTE_P (tree fndecl, tree name, tree args, int flags) [Target Hook]

This hook is called to parse the attribute(option("...")), and it allows the function to set different target machine compile time options for the current function that might be different than the options specified on the command line. The hook should return true if the options are valid.

The hook should set the DECL_FUNCTION_SPECIFIC_TARGET field in the function declaration to hold a pointer to a target specific struct cl_target_option structure.

void TARGET_OPTION_SAVE (struct cl_target_option *ptr) [Target Hook]

This hook is called to save any additional target specific information in the struct cl_target_option structure for function specific options. See Section 7.1 [选项文件格式], page 67.

void TARGET_OPTION_RESTORE (struct cl_target_option *ptr) [Target Hook]

This hook is called to restore any additional target specific information in the struct cl_target_option structure for function specific options.

void TARGET_OPTION_PRINT (struct cl_target_option *ptr) [Target Hook]

This hook is called to print any additional target specific information in the struct cl_target_option structure for function specific options.

bool TARGET_OPTION_PRAGMA_PARSE (target args) [Target Hook]

This target hook parses the options for #pragma GCC option to set the machine specific options for functions that occur later in the input stream. The options should be the same as handled by the TARGET_VALID_OPTION_ATTRIBUTE_P hook.

bool TARGET_CAN_INLINE_P (tree caller, tree callee) [Target Hook]

This target hook returns false if the caller function cannot inline callee, based on target specific information. By default, inlining is not allowed if the callee function has function specific target options and the caller does not use the same options.

17.26 模拟TLS

For targets whose psABI does not provide Thread Local Storage via specific relocations and instruction sequences, an emulation layer is used. A set of target hooks allows this emulation layer to be configured for the requirements of a particular target. For instance the psABI may in fact specify TLS support in terms of an emulation layer.

The emulation layer works by creating a control object for every TLS object. To access the TLS object, a lookup function is provided which, when given the address of the control object, will return the address of the current thread's instance of the TLS object.

const char * TARGET_EMUTLS_GET_ADDRESS [Target Hook]

Contains the name of the helper function that uses a TLS control object to locate a TLS instance. The default causes libgcc's emulated TLS helper function to be used.

const char * TARGET_EMUTLS_REGISTER_COMMON [Target Hook]
Contains the name of the helper function that should be used at program startup to register TLS objects that are implicitly initialized to zero. If this is NULL, all TLS objects will have explicit initializers. The default causes libgcc's emulated TLS registration function to be used.

const char * TARGET_EMUTLS_VAR_SECTION [Target Hook]
Contains the name of the section in which TLS control variables should be placed. The default of NULL allows these to be placed in any section.

const char * TARGET_EMUTLS_TMPL_SECTION [Target Hook]
Contains the name of the section in which TLS initializers should be placed. The default of NULL allows these to be placed in any section.

const char * TARGET_EMUTLS_VAR_PREFIX [Target Hook]
Contains the prefix to be prepended to TLS control variable names. The default of NULL uses a target-specific prefix.

const char * TARGET_EMUTLS_TMPL_PREFIX [Target Hook]
Contains the prefix to be prepended to TLS initializer objects. The default of NULL uses a target-specific prefix.

tree TARGET_EMUTLS_VAR_FIELDS (tree type, tree *name) [Target Hook]
Specifies a function that generates the FIELD_DECLs for a TLS control object type. type is the RECORD_TYPE the fields are for and name should be filled with the structure tag, if the default of __emutls_object is unsuitable. The default creates a type suitable for libgcc's emulated TLS function.

tree TARGET_EMUTLS_VAR_INIT (tree var, tree decl, tree tmpl_addr) [Target Hook]
Specifies a function that generates the CONSTRUCTOR to initialize a TLS control object. var is the TLS control object, decl is the TLS object and tmpl_addr is the address of the initializer. The default initializes libgcc's emulated TLS control object.

bool TARGET_EMUTLS_VAR_ALIGN_FIXED [Target Hook]
Specifies whether the alignment of TLS control variable objects is fixed and should not be increased as some backends may do to optimize single objects. The default is false.

bool TARGET_EMUTLS_DEBUG_FORM_TLS_ADDRESS [Target Hook]
Specifies whether a DWARF DW_OP_form_tls_address location descriptor may be used to describe emulated TLS control objects.

17.27 定义MIPS target的协处理器的规范

MIPS规范允许MIPS的实现最多具有4个协处理器，其中每个最多具有32个私有寄存器。 GCC支持使用汇编形式的变量对这些寄存器的访问，寄存器以及内存间的传值。例如：

```
register unsigned int cp0count asm ("c0r1");
unsigned int d;

d = cp0count + 3;
```

（“c0r1”是协处理器0的寄存器1的缺省名；可以按照下面的描述来增加可选名字， 或者可以通过SUBTARGET_CONDITIONAL_REGISTER_USAGE来覆写全部的缺省名字。）

协处理器寄存器被假设为epilogue-used；对它们的赋值将被保存， 即使在函数中的后面不会再使用该寄存器。

另一个需要注意的是：根据MIPS spec，协处理器1（如果存在）为FPU。 标准mips浮点支持的对COP1寄存器的访问，不包含在这个机制中。

下面描述的一个宏，用于定义MIPS协处理器接口， 可能在子目标（subtarget）中需要被覆写。

ALL_COP_ADDITIONAL_REGISTER_NAMES [Macro]

一个逗号分隔的列表（起始于逗号），是用于描述可选的协处理器寄存器的名字对。 每项的格式应该为

{ alternatename, register_number}

缺省情况: 空

17.28 预编译头文件有效性检查的参数

void *TARGET_GET_PCH_VALIDITY (size_t *sz) [Target Hook]

该钩子返回TARGET_PCH_VALID_P所需要的数据， 并且将`*sz'设为以字节为单位的数据大小。

const char *TARGET_PCH_VALID_P (const void *data, size_t sz) [Target Hook]

该钩子检查用于创建PCH文件的选项是否与现在的设置兼容。 如果是则返回NULL，否则为一个适当的错误消息。 错误消息将会展现给用户，所以必须使用`_(msg)'来本地化。

data为当PCH文件被创建时，TARGET_GET_PCH_VALIDITY所返回的数据， sz为以字节为单位的数据大小。是可以假设data由同一版本的编译器所创建的， 所以不需要格式检查。

default_pch_valid_p的缺省定义应该适合于大多数target。

const char *TARGET_CHECK_PCH_TARGET_FLAGS (int pch_flags) [Target Hook]

如果该钩子为非空， 则TARGET_PCH_VALID_P的缺省实现将用它来检查target_flags的兼容值。 pch_flags指定了当PCH文件被创建时， target_flags所具有的值。 返回值与TARGET_PCH_VALID_P的相同。

17.29 C++ ABI参数

tree TARGET_CXX_GUARD_TYPE (void) [Target Hook]

定义该钩子来覆盖用于guard变量的整数类型。 这些被用于实现静态对象的一次构建。缺省为long_long_integer_type_node。

bool TARGET_CXX_GUARD_MASK_BIT (void) [Target Hook]

该钩子确定如何使用guard变量。如果第一个字节应该被使用， 则应该返回false（缺省）。返回值为true表明应该使用最低有效位。

tree TARGET_CXX_GET_COOKIE_SIZE (tree type) [Target Hook]

该钩子返回cookie的大小，其中cookie为当分配一个数组其元素具有type类型， 所使用的cookie。假设已经知道需要一个cookie。 缺省为max(sizeof (size_t), alignof(type))，在2.7节IA64/Generic C++ ABI中定义。

bool TARGET_CXX_COOKIE_HAS_SIZE (void) [Target Hook]

该钩子应该返回true，如果元素大小应该被存放在数组cookie中。 缺省为返回false。

int TARGET_CXX_IMPORT_EXPORT_CLASS (tree type, int import_export) [Target Hook]
如果后端定义了该钩子，则允许覆盖对导出类type的决定。 import_export将包含1，如果类将被导出，−1如果其将被导入，否则为0. 该函数应该返回修改后的值，并执行其它需要的操作来支持后端的操作系统。

bool TARGET_CXX_CDTOR_RETURNS_THIS (void) [Target Hook]
该钩子应该返回true，如果构造者和析构者返回创建/销毁对象的地址。 缺省为返回false。

bool TARGET_CXX_KEY_METHOD_MAY_BE_INLINE (void) [Target Hook]
该钩子返回真，如果类的关键方法（即，如果方法在当前转换单元中定义， 其使得虚拟表被输出）可以为内联函数。对于标准的Itanium C++ ABI， 关键方法可以为内联函数，只要函数不在类定义中声明为内联的。 其它ABI的变体中，内联函数不能为关键方法。缺省为返回true。

void TARGET_CXX_DETERMINE_CLASS_DATA_VISIBILITY (tree decl) [Target Hook]
decl为虚拟表，类型信息对象，或其它类似的隐含类数据对象， 其将在该转换单元中作为外部链接被输出。ELF的可视性不会被显示的指定。 如果target需要指定可视性， 可以使用该钩子来设置DECL_VISIBILITY和DECL_VISIBILITY_SPECIFIED。

bool TARGET_CXX_CLASS_DATA_ALWAYS_COMDAT (void) [Target Hook]
该钩子返回真（缺省情况）如果虚拟表和其它类似的隐式类数据对象总是为COMDAT， 如果它们具有外部连接。如果该钩子返回假， 则只在一个转换单元中被输出的虚拟表的类的类数据将不是COMDAT。

bool TARGET_CXX_LIBRARY_RTTI_COMDAT (void) [Target Hook]
该钩子返回真（缺省情况）， 如果在C++运行时定义的基本类型的RTTI信息应该总是为COMDAT，否则为假。

bool TARGET_CXX_USE_AEABI_ATEXIT (void) [Target Hook]
该钩子返回真， 如果__aeabi_atexit（如被ARM EABI定义）应该用于注册静态析构者， 当`-fuse-cxa-atexit'为有效时。缺省是返回假，使用__cxa_atexit。

bool TARGET_CXX_USE_ATEXIT_FOR_CXA_ATEXIT (void) [Target Hook]
该钩子返回真， 如果target的atexit函数可以跟__cxa_atexit一样被用于注册静态析构者。这要求在共享库中atexit注册的函数要按照正确的顺序运行，当库被卸载时。 缺省为返回假。

void TARGET_CXX_ADJUST_CLASS_AT_DEFINITION (tree type) [Target Hook]
type为一个刚被定义的C++类（即RECORD_TYPE 或 UNION_TYPE）。 使用该钩子来调整类（例如tweak可视性或者执行其它target需要的修改）。

17.30 其它参数

这是一些其它参数。

HAS_LONG_COND_BRANCH [Macro]
定义该布尔值的宏用来指示你的体系结构是否具有可以跨越所有内存的条件分支。 它用于将可执行程序hot和cold基本块分割到单独的段的优化。 如果该宏被设为false，则gcc将任何穿越段的条件分支转换为无条件分支或间接跳转。

HAS_LONG_UNCOND_BRANCH [Macro]
定义该布尔值的宏用来指示你的体系结构是否具有可以跨越所有内存的无条件分支。 它用于将可执行程序hot和cold基本块分割到单独的段的优化。 如果该宏被设为false，则gcc将任何穿越段的无条件分支转换为间接跳转。

CASE_VECTOR_MODE [Macro]

机器模式名字的一个别名。这是跳转表（jump-table）的元素应该具有的机器模式。

CASE_VECTOR_SHORTEN_MODE (min_offset, max_offset, body) [Macro]

可选的：当最小值和最大值位移已知时，返回addr_diff_vec的首选机器模式。 如果定义了该宏，这使得在分支缩短中增加了额外的代码来处理addr_diff_vec。 要使其工作，还必须要定义INSN_ALIGN， 并且显示的对addr_diff_vec进行对齐。参数body被提供， 使得可以更新offset_unsigned和标量标记。

CASE_VECTOR_PC_RELATIVE [Macro]

定义该宏为一个C表达式，来指示跳转表什么时候应该包含相对地址。 你不需要定义该宏，如果跳转表从来不包含相对地址， 或者跳转表只在`-fPIC'或者`-fPIC'有效时才包含相对地址。

CASE_VALUES_THRESHOLD [Macro]

定义其为一个最小差值数，用于选择是使用跳转表来替代条件分支树。 缺省为4，对于具有casesi指令的机器，其它的为5。这对大多数机器是最好的。

CASE_USE_BIT_TESTS [Macro]

定义该宏为一个C表达式，来指示C switch语句是否可以通过位测试序列来实现。 这在可以通过寄存器中的位数来有效实现左移1位的处理器上很有利， 但不适合需要循环的target。缺省下，该宏返回true， 如果target定义了ashlsi3指令模式，否则返回false。

WORD_REGISTER_OPERATIONS [Macro]

定义该宏，如果整数机器模式的小于一个字的寄存器间的运算总是在整个寄存器中执行。 大多数RISC机器具有这个属性，大多数CISC机器不具有。

LOAD_EXTEND_OP (mem_mode) [Macro]

定义该宏为一个C表达式， 指示当insn使用比一个字窄的的整数模式的mem_mode模式读取内存时， 将读取的数据的mem_mode外的位进行符号扩展或者零扩展。 返回SIGN_EXTEND，对于那些要符号扩展的insn， 返回ZERO_EXTEND对那些零扩展的，对于其它的机器模式返回UNKNOWN。

该宏不会被非整型的，或者宽度大于等于BITS_PER_WORD的mem_mode调用， 所以对于这种情况你可以返回任何值。如果总是返回UNKNOWN，则不要定义该宏。 在定义该宏的机器上，你通常要定义其为常量SIGN_EXTEND或者ZERO_EXTEND。

你可以返回一个非UNKNOWN的值，即使对于一些硬件寄存器并没有执行符号扩展，如果对于这些硬件寄存器的REGNO_REG_CLASS，当from机器模式为mem_mode，并且to机器模式为任何大于其但是不大于word_mode的整形机器模式的时候，CANNOT_CHANGE_MODE_CLASS返回非零。

你必须返回UNKNOWN，如果一些硬件寄存器允许该机器模式，CANNOT_CHANGE_MODE_CLASS说它们不能变成word_mode，但是它们可以变成其它大于mem_mode且仍然小于word_mode的整形机器模式。

SHORT_IMMEDIATES_SIGN_EXTEND [Macro]

定义该宏，如果将short立即数加载到寄存器中要进行符号扩展。

FIXUNS_TRUNC_LIKE_FIX_TRUNC [Macro]

定义该宏，如果将浮点数转换为有符号定点数的指令， 同样可以有效的转换为无符号的。

int TARGET_MIN_DIVISIONS_FOR_RECIP_MUL (enum machine_mode mode) [Target Hook]

当`-ffast-math'有效时，GCC尝试使用相同的除数来优化除法， 通过将它们转换为乘以倒数的方式。 缺省实现返回3，如果机器具有除法指令，否则为2。

MOVE_MAX [Macro]

单个指令可以在内存和寄存器间，或者两个内存位置间快速移动的最大字节数。

MAX_MOVE_MAX [Macro]

单个指令可以在内存和寄存器间，或者两个内存位置间快速移动的最大字节数。 如果没有定义，则缺省为MOVE_MAX。 否则，其为MOVE_MAX在运行时可以具有的最大常数值。

SHIFT_COUNT_TRUNCATED [Macro]

一个C表达式，为非零，如果在该机器上，实际用于计算移位运算的位数等同于，用来表示被移位的对象大小的位数。当该宏为非零的时候，编译器将假设可以安全的忽略掉对移位运算的计数进行截取的有符号扩展，零扩展和按位与指令。在一些机器上，具有指令可以作用于可变位置的位域，其可能会包含‘位测试’指令，非零的SHIFT_COUNT_TRUNCATED还可以使得作为位域指令参数的值的截取运算。

如果指令会截取计数（对于位移运算）和位置（对于位域运算），或者如果没有可变位置的位域指令存在，则你应该定义该宏。

然而，在一些机器上，例如80386和680x0，截取操作只应用在移位运算上，而不在位域运算上。在这样的机器上，定义SHIFT_COUNT_TRUNCATED为零。可替代的，在`md'文件中增加指令模式，包含对移位指令隐式的截取操作。

如果其值总是为零，则不需要定义该宏。

int TARGET_SHIFT_TRUNCATION_MASK (enum machine_mode mode) [Target Hook]

该函数描述了标准的移位指令模式，对于mode，如何处理负的数量或者大于机器模式宽度的数量的移位。 See [shift patterns], page 240.

在许多机器上，移位指令模式将会应用一个掩码m到移位计数上，意味着将x固定宽度移位y等价于对x任意宽度移位y & m。如果这对机器模式mode为真，则函数应该返回m，否则应该返回0。返回值0意味着不保证特定的行为。

注意，不像SHIFT_COUNT_TRUNCATED，该函数不应用到通用的移位rtx上；其只应用到由命名移位指令模式生成的指令上。

该函数的缺省实现返回GET_MODE_BITSIZE (mode) - 1，如果SHIFT_COUNT_TRUNCATED，否则为0。该定义总是安全的，但是如果SHIFT_COUNT_TRUNCATED为假，并且一些移位指令模式还是截取移位计数，则你可以通过重写覆盖该宏来获得更好的代码。

TRULY_NOOP_TRUNCATION (outprec, inprec) [Macro]

一个C表达式，其为非零，如果在该机器上，将inprec个位数的整数转换成outprec个位数（outprec比inprec小），通过简单的认为其只具有outprec个位。

在许多机器上，该表达式可以为1。

int TARGET_MODE_REP_EXTENDED (enum machine_mode mode, enum machine_mode rep_mode) [Target Hook]

整型机器模式可以表示为一个值，其总是被扩展为更宽的整型模式。返回SIGN_EXTEND，如果机器模式为mode的值被表示为有符号扩展成rep_mode的形式。否则，返回UNKNOWN。（目前，没有目标机使用零扩展表示，所以不像LOAD_EXTEND_OP，TARGET_MODE_REP_EXTENDED被期望返回SIGN_EXTEND或者UNKNOWN。而且没有目标机将mode扩展为mode_rep，以至于mode_rep不是下一个最宽的整型机器模式，目前，我们利用了这个事实。）

类似于LOAD_EXTEND_OP，你可以返回一个非UNKNOWN的值，即使扩展在特定硬件寄存器上没有被执行，只要对于这些硬件寄存器的REGNO_REG_CLASS，CANNOT_CHANGE_MODE_CLASS返回非零。

注意，TARGET_MODE_REP_EXTENDED和LOAD_EXTEND_OP描述了两个相关联的属性。如果你定义了TARGET_MODE_REP_EXTENDED (mode, word_mode)，你可能还要定义LOAD_EXTEND_OP (mode)，来返回相同类型的扩展。

为了加强mode的表示，当截取成mode时，TRULY_NOOP_TRUNCATION应该返回假。

STORE_FLAG_VALUE [Macro]

一个C表达式，描述了整型机器模式的比较运算符返回的值，并且当条件为真时由存储标记指令(`scond')存储。该描述必须应用到所有的`scond'指令模式，并且所有比较运算的结果必须具有MODE_INT机器模式。

值为1或者-1，意味着实现比较运算的指令当比较为真时返回确切的1或者-1，当比较为假时返回0。否则，值会表示当比较为真时结果的哪些位保证为1。该值按照比较运算的机器模式来解析，其由`scond'指令模式中的第一个操作数的机器模式给出。目前，编译器只用到了STORE_FLAG_VALUE的低位或者符号位。

如果STORE_FLAG_VALUE不为1或者-1，则编译器将生成只依赖特定位的代码。其还可以用等价的运算来替换比较运算，如果它们会造成需要的位被设置，即使其它位没有被定义。例如，在比较运算返回一个SImode值的机器上，其STORE_FLAG_VALUE被定义为`0x80000000'，说明只有符号位是相关的，表达式

```
(ne:SI (and:SI x (const_int power-of-2)) (const_int 0))
```

可以被转换为

```
(ashift:SI x (const_int n))
```

其中n为适当的移位计数，用来将被测试的位移送到符号位。

没有办法来描述，一个机器对于真值，总是设置低顺序的位，而不保证其它位的值。但是，我们不知道是否有机器具有这样的指令。如果你正在尝试将GCC移植到这样的机器上，那么可以在比较运算的指令模式中，包含一条指令来执行将结果和1进行逻辑与，并且通过gcc@gcc.gnu.org让我们知道。

通常，机器将具有多个指令，从比较（或者条件代码）中获得一个值。这里有一些规则用来指导对STORE_FLAG_VALUE的值的选择，以及要使用的指令：

- 使用最短的序列，产生STORE_FLAG_VALUE的有效定义。对编译器来说，将值正常化（例如，将其转成1或者0）要比进行比较运算更有效，因为可能会有一些机会来合并其它的正常化运算。
- 对于等长的序列，使用值1或者 - 1。在跳转代价比较高的机器上，一般会倾向于 - 1，其它一些机器喜欢用1。
- 作为第二种选择，选择值`0x80000001'，如果存在指令，同时设置符号位和低顺序位，但是不定义其它位。
- 否则，使用值`0x80000000'。

许多机器可以同时产生供STORE_FLAG_VALUE选择的值，以及同样数量指令的取反。在那些机器上，你应该还定义这些情况的指令模式，例如，

```
(set A (neg:m (ne:m B C)))
```

一些机器还可以在条件代码值上，执行and或者plus运算，使用少于相应的`scond' insn后跟随and或者plus的指令数目。在这些机器上，需要定义适当的指令模式。分别使用名字incscc和decscc，对于在条件代码值上执行plus或者minus运算的指令模式。参见`rs6000.md'中的一些例子。可以使用GNU Superoptizer来在其它机器上查找这样的指令序列。

如果该宏没有被定义，则使用缺省的值1。你不需要定义STORE_FLAG_VALUE，如果机器没有存储标记的指令，或者如果这些指令生成的值为1。

FLOAT_STORE_FLAG_VALUE (mode) [Macro]

一个C表达式，给出一个非零的REAL_VALUE_TYPE值，当浮点比较运算的结果为真时返回该值。在一些机器上，具有返回浮点值的比较运算，可以定义该宏。如果没有这样的运算，则不要定义该宏。

VECTOR_STORE_FLAG_VALUE (mode) [Macro]

一个C表达式，给出一个rtx，表示向量比较中非零的真元素。返回的rtx应该对于mode的内部机器模式是有效的，mode为一个向量机器模式。定义该宏，在一些机器上，具有返回向量结果的向量比较运算。如果没有这样的运算，则不要定义该宏。通常，该宏被定义为const1_rtx或者constm1_rtx。该宏可以返回NULL_RTX，来阻止编译器优化给定的机器模式的向量比较运算。

CLZ_DEFINED_VALUE_AT_ZERO (mode, value) [Macro]

CTZ_DEFINED_VALUE_AT_ZERO (mode, value) [Macro]

一个C表达式，指示体系机构是否为clz或者ctz在操作数为零时，定义了值。结果为0，表示值未被定义。如果值只被定义为RTL表达式，则宏应该求值为1；如果还应用到相应的optab项（其通常情况为直接扩展为相应的RTL），则宏应该求值为2。在值被定义的情况下，value应该被设置为该值。

如果该宏没有被定义，则clz或者ctz在操作数为零时，被认为未定义。

该宏必须被定义，如果目标机对ffs的扩展，依赖于特定的值，以获得正确的结果。否则，没有必要，虽然其可以用于优化一些边角的情况，并且为ffs optab提供缺省的扩展。

注意，不论该宏是否定义，clz和ctz在操作数为0时的定义，都不会被扩展为用户可见的内建函数。因此，可以任意调整该值，来匹配对这些运算的目标机扩展，而无需担心会破坏API。

Pmode [Macro]

指针的机器模式别名。在大多数机器上，定义该宏为整型机器模式，对应于硬件指针宽度；32位机器上为SImode，64位机器上为DImode。在一些机器上，你必须定义该宏为部分整型机器模式，例如PSImode。

Pmode的宽度必须至少与POINTER_SIZE的值一样大。如果不相等，你必须定义宏POINTERS_EXTEND_UNSIGNED来描述指针如何被扩展为Pmode。

FUNCTION_MODE [Macro]

机器模式的别名，在call RTL表达式中，用于被调用函数的内存引用。在大多数CISC机器上，指令可以起始于任意字节地址，这应该为QImode。在大多RISC机器上，所有的指令都具有固定的大小和对齐方式，这应该为与机器指令字具有相同大小和对齐的机器模式，通常为SImode或者HImode。

STDC_0_IN_SYSTEM_HEADERS [Macro]

正常的操作中，预处理器会将__STDC__扩展为常量1，来表明GCC遵循ISO标准C。在一些主机上，例如Solaris，系统编译器使用不同的约定，__STDC__通常为0，但是如果用户指出要严格遵循C标准时为1。

定义STDC_0_IN_SYSTEM_HEADERS使得GNU CPP遵循主机的约定，当处理系统头文件时，但是，当处理用户文件时__STDC__将总是被扩展为1。

NO_IMPLICIT_EXTERN_C [Macro]

定义该宏，如果系统头文件支持C++，也支持C。该宏抑制了通常在C++中使用系统头文件的方法，即假设文件的内容包含在`extern "C" {...}'中。

REGISTER_TARGET_PRAGMAS () [Macro]

定义该宏，如果你想实现任何目标机特定的pragma。如果被定义，其为一个C表达式，为每个pragma使用一系列的对c_register_pragma或者c_register_pragma_with_expansion调用。该宏还可以做任何pragma所要求的设置。

定义该宏的主要原因是提供相同目标机上，与其它编译器的兼容性。大体上，我们不鼓励为GCC定义目标机特定的pragma。

如果pragma可以通过attribute属性来实现，则你也应该考虑定义目标机钩子`TARGET_INSERT_ATTRIBUTES'。

出现在pragma行中的预处理器宏不被扩展。所有不匹配被注册的`#pragma'指令，将被安静的忽略，除非用户指定`-Wunknown-pragmas'。

void c_register_pragma (const char *space, const char *name, void (*callback) (struct cpp_reader *)) [Function]

void c_register_pragma_with_expansion (const char *space, const char *name, void (*callback) (struct cpp_reader *)) [Function]

每个对c_register_pragma或者c_register_pragma_with_expansion的调用，都建立了一个pragma。当预处理器遇到一个形式为

```
#pragma [space] name ...
```

的pragma，则callback函数将被调用。

space为大小写敏感的pragma命名空间，或者为NULL，将gragma放在全局命名空间中。回调函数接受pfile作为第一个参数，如果需要其可以被传递给cpplib的函数。你可以通过调用pragma_lex，来词法分析name之后的token。没有被回调函数读入的token将被安静的忽略。行尾由类型为CPP_EOF的token来指示。宏扩展发生在使用c_register_pragma_with_expansion注册的pragma的参数上，但不在使用c_register_pragma注册的pragma的参数上。

注意，使用pragma_lex是特定于C和C++编译器的。其在Java或者Fortran编译器上，或者其它语言编译器上无法工作。因此，如果pragma_lex准备从目标机特定代码中被调用，其必须只在构建C和C++编译器的时候执行。这可以通过在`config.gcc'文件中的目标机项中，定义变量c_target_objs和cxx_target_objs来实现。这些变量应该命名目标机特定的，语言特定的对象文件，其包含了使用pragma_lex的代码。注意，还有必要增加一个规则到由tmake_file指定的makefile片断中，来显示如何构建该目标文件。

HANDLE_SYSV_PRAGMA [Macro]

定义该宏（值为1），如果你想gcc支持System V风格的pragma `#pragma pack(<n>)'和`#pragma weak <name> [=<value>]'。

pack pragma描述了结构体中域的最大对齐方式（以字节为单位），很大程度上等同于使用`__aligned__'和`__packed__' __attribute__。pack值为零，会将行为重置为缺省。

对于目标机支持的微软Visual C/C++风格的位域打包（例如-mms-bitfields），其微妙之处在于：当位域被插入到一个打包了的记录中时，则底层类型的整个大小会被一个或多个相同大小临近的位域使用（也就是说，如果为long:3，则记录会使用32位，任何额外的临近的long位域将被打包到32位块中。然而，如果大小改变了，则会分配一个新的同样大小的域）。

如果MS位域和`__attribute__((packed))'都被使用，则后者优先。如果当在使用MS位域的时候，`__attribute__((packed))'被用于一个单独的域，则对于该域是优先的，但是结构体的其它部分的对齐方式会影响它的放置。

weak pragma只有当SUPPORTS_WEAK和ASM_WEAKEN_LABEL被定义时，才工作。如果可用，其允许创建特殊命名的弱标号，以及可选的值。

HANDLE_PRAGMA_PACK_PUSH_POP [Macro]

定义该宏（值为1），如果你想支持Win32风格的pragmas `#pragma pack(push[,n])' and `#pragma pack(pop)'。`pack(push,[n])' pragma指定结构体内的域的最大对齐方式（按字节），等同于使用`__aligned__' 和 `__packed__' __attribute__。如果pack值为0，则会将行为重置为缺省情况。连续的调用该pragma会造成先前的值被压栈，所以调用`#pragma pack(pop)'会返回先前的值。

HANDLE_PRAGMA_PACK_WITH_EXPANSION [Macro]

定义该宏，以及HANDLE_SYSV_PRAGMA，如果在`#pragma pack'中的参数宏应该被扩展。

TARGET_DEFAULT_PACK_STRUCT [Macro]

如果你的目标机要求结构体缺省的打包方式不是0（意味着机器缺省方式），则定义该宏为必要的值（以字节为单位）。该值必须还对于`#pragma pack()'有效（也就是说，2的小的幂数）。

HANDLE_PRAGMA_PUSH_POP_MACRO [Macro]

定义该宏，如果你想支持Win32风格的pragmas `#pragma push_macro(macro-name-as-string)'和 `#pragma pop_macro(macro-name-as-string)'。`#pragma push_macro(macro-name-as-string)'用来保存命名的宏，并且通过`#pragma pop_macro(macro-name-as-string)'来返回先前的值。

DOLLARS_IN_IDENTIFIERS [Macro]

定义该宏，来控制C语言家族，标识符名字中对字符`$'的使用。0意味着缺省不允许使用`$'；1意味着允许。缺省为1；对于这种情况，不需要定义该宏。

NO_DOLLAR_IN_LABEL [Macro]

定义该宏，如果汇编器不接受标号名字中的`$'。缺省的，G++的构造函数和析构函数会在标识符中使用`$'。如果该宏被定义，则使用`.'来替代。

NO_DOT_IN_LABEL [Macro]

定义该宏，如果汇编器不接受标号名字中的`.'。缺省的，G++的构造函数和析构函数的名字会使用`.'。如果该宏被定义，则这些名字被重写，以避免出现`.'。

INSN_SETS_ARE_DELAYED (insn) [Macro]

定义该宏为一个C表达式，其为非零，如果对于延迟槽调度器，将指令放在insn的延迟槽中是安全的，即使它们可能会使用insn设置或者破坏的一个资源。insn总是一个jump_insn或者insn；GCC知道每个call_insn具有这种行为。在一些机器上，一些insn或者jump_insn确实为一个函数调用，因此也具有这种行为，你应该定义该宏。

如果其总是返回零，则不需要定义该宏。

INSN_REFERENCES_ARE_DELAYED (insn) [Macro]

定义该宏为一个C表达式，其为非零，如果对于延迟槽调度器，将指令放在insn的延迟槽中是安全的，即使它们可能会设置或者破坏insn使用的一个资源。insn总是一个jump_insn或者insn。在一些机器上，一些insn或者jump_insn确实为一个函数调用，并且其操作数为寄存器，实际是在其调用的子函数中使用，你应该定义该宏。这使得延迟槽调度器将复制参数到参数寄存器的指令移送到insn的延迟槽中。

如果其总是返回零，则不需要定义该宏。

MULTIPLE_SYMBOL_SPACES [Macro]

定义该宏为一个C表达式，其为非零，如果对于一些情况，没有用户的介入，一个转换单元中的全局符号可能不会被发现为另一个转换单元中的未定义的符号。例如，在Microsoft Windows下，符号必须被从共享库（DLL）中显式的导入。

如果其总是为零，则不需要定义该宏。

tree TARGET_MD_ASM_CLOBBERS (tree outputs, tree inputs, tree clobbers) [Target Hook]
该目标钩子将移植平台希望一个asm可以自动破坏的硬件寄存器增加到clobbers STRING_CST tree中。其应该返回最后一个用于增加一个破坏者的tree_cons。outputs, inputs 和 clobber为asm的相应的参数，可以用来检查以避免破坏asm的输入或者输出寄存器。你可以使用`tree.h'中声明的tree_overlaps_hard_reg_set来测试是否与asm声明的寄存器有重叠。

MATH_LIBRARY [Macro]
定义该宏为一个C字符串常量，为连接器的参数，作为系统数学库连接，或者`""'，如果目标机没有单独的数学库。

只有当缺省的`"-lm"'有错误时，才需要定义该宏。

LIBRARY_PATH_ENV [Macro]
定义该宏为一个C字符串常量，为一个环境变量，指定了连接器应该从哪里查找库。

只有当缺省的`"LIBRARY_PATH"'有错误时，才需要定义该宏。

TARGET_POSIX_IO [Macro]
定义该宏，如果目标机支持下列POSIX文件函数，access, mkdir和使用fcntl / F_SETLKW的文件加锁。其还在运行时为交叉profiling创建目录。

MAX_CONDITIONAL_EXECUTE [Macro]
一个C表达式，为通过条件执行指令来替代分支的最大指令数。值BRANCH_COST+1为缺省值，如果机器没有使用cc0，如果使用了cc0，则为1。

IFCVT_MODIFY_TESTS (ce_info, true_expr, false_expr) [Macro]
使用该宏，如果目标机需要在将基本块转为条件执行代码时，对条件进行机器相关的修改。ce_info指向一个数据结构，struct ce_if_block，其包含了关于当前被处理的块的信息。true_expr和false_expr为测试，分别用于转换then-block和else-block。如果测试不能被转换，则将true_expr或者false_expr设置为空指针。

IFCVT_MODIFY_MULTIPLE_TESTS (ce_info, bb, true_expr, false_expr) [Macro]
类似于IFCVT_MODIFY_TESTS，不过用于将更加复杂的if语句转换为由and和or运算组合的条件。bb包含的基本块，包含了当前被处理的测试，并将被转换为一个条件。

IFCVT_MODIFY_INSN (ce_info, pattern, insn) [Macro]
一个C表达式，来修改INSN的PATTERN，其将被转换为一个条件执行的格式。ce_info指向一个数据结构，struct ce_if_block，其包含了关于当前被处理的块的信息。

IFCVT_MODIFY_FINAL (ce_info) [Macro]
一个C表达式，用来执行在将代码转换为条件执行时，任何最终机器相关的修改。涉及到的基本块可以在由ce_info指向的struct ce_if_block结构体中找到。

IFCVT_MODIFY_CANCEL (ce_info) [Macro]
一个C表达式，用来取消在将代码转换为条件执行时，任何机器相关的修改。涉及到的基本块可以在由ce_info指向的struct ce_if_block结构体中找到。

IFCVT_INIT_EXTRA_FIELDS (ce_info) [Macro]
一个C表达式，来初始化任何struct ce_if_block结构体中额外的域，其通过IFCVT_EXTRA_FIELDS宏来定义。

IFCVT_EXTRA_FIELDS [Macro]

如果被定义，其应该扩展为一个域声明集合，其将被增加到struct ce_if_block结构体中。这些应该通过IFCVT_INIT_EXTRA_FIELDS宏来初始化。

void TARGET_MACHINE_DEPENDENT_REORG () [Target Hook]

如果非空，则该钩子对指令流执行目标机特定的编译过程。编译器将在所有优化级别上执行该过程，就在其通常执行延迟分支调度之前的地方。

该钩子的确切目的因目标机不同而不同。一些用来为了正确性执行必要的转换，例如布局函数常量池，或者避免硬件冒险。其它用来作为机器相关的优化。

如果没有什么可做的，则不需要实现该钩子。缺省定义为空。

void TARGET_INIT_BUILTINS () [Target Hook]

定义该钩子，如果你有任何需要被定义的机器特定的内建函数。其应该为一个函数，执行必要的设置。

机器特定的内建函数可以用于扩展特定的机器指令，否则其通常不会被生成，因为在源语言中没有等价的对应（例如，SIMD向量指令或者预取指令）。

要创建一个内建函数，调用函数lang_hooks.builtin_function，其由语言前端定义。你可以使用任何由build_common_tree_nodes和build_common_tree_nodes_2建立的类型节点；只有使用这两个函数的语言前端会调用`TARGET_INIT_BUILTINS'。

rtx TARGET_EXPAND_BUILTIN (tree exp, rtx target, rtx subtarget, enum machine_mode mode, int ignore) [Target Hook]

扩展由`TARGET_INIT_BUILTINS'建立的对一个机器特定的内建函数的调用。exp为函数调用的表达式；如果方便的话，结果应该放到target，并且具有机器模式mode。subtarget可以用作目标，来计算exp的操作数。ignore为非零，如果值将被忽略。该函数应该返回调用内建函数的结果。

tree TARGET_RESOLVE_OVERLOADED_BUILTIN (tree fndecl, tree arglist) [Target Hook]

为`TARGET_INIT_BUILTINS'建立的机器特定内建函数选择一个替身。这发生在常规的类型检查之前，因此允许目标机来实现函数重载的粗糙形式。fndecl为内建函数的声明。arglist为传递给内建函数的参数列表。结果为一个完全表达式，实现了该运算，通常为另一个CALL_EXPR。

tree TARGET_FOLD_BUILTIN (tree fndecl, tree arglist, bool ignore) [Target Hook]

将`TARGET_INIT_BUILTINS'建立的机器特定的内建函数的调用进行折叠。fndecl为内建函数的声明。arglist为传递给内建函数的参数列表。结果为另一个tree，包含了一个简化的表达式，为调用的结果。如果ignore为真，则值将被忽略。

const char * TARGET_INVALID_WITHIN_DOLOOP (rtx insn) [Target Hook]

接受一个指令insn，如果其在一个低开销循环中有效，则返回NULL，否则返回一个字符串说明为什么不能应用doloop。

许多目标机使用特定的寄存器，用于低开销循环。对于任何破坏这些的指令，该函数应该返回一个字符串，指出不能应用doloop的原因。缺省的，RTL循环优化不对包含函数调用或者表指令分支的循环，使用现有的doloop指令模式。

MD_CAN_REDIRECT_BRANCH (branch1, branch2) [Macro]

接受一个分支insn branch1和另一个branch2。返回真，如果将branch1重定向到branch2的目的地是可能的。

在一些目标机上，分支可能具有有限的范围。优化延迟槽的填充，可以导致分支可以被重定向，反过来也可能会造成一个分支的偏移量溢出。

bool TARGET_COMMUTATIVE_P (rtx x, outer_code) [Target Hook]

该目标钩子返回true，如果x被认为是可交换的。通常，这就是COMMUTATIVE_P (x)。但是HP PA不认为PLUS在MEM中是可交换的。outer_code为包含rtl的rtx代码，如果知道，否则为UNKNOWN。

rtx TARGET_ALLOCATE_INITIAL_VALUE (rtx hard_reg) [Target Hook]

当硬件寄存器的初始值已经被复制到伪寄存器中，通常没有必要为该伪寄存器分配另一个寄存器，因为可以使用原始的硬件寄存器或者被保存到的栈槽。在寄存器分配的起始处，TARGET_ALLOCATE_INITIAL_VALUE针对每个其初始值使用get_func_hard_reg_initial_val或者get_hard_reg_initial_val复制过的，硬件寄存器被调用一次。可能的值为，NULL_RTX如果你不想做任何特殊的分配，一个REG rtx——其通常为硬件寄存器本身，如果知道其不会被破坏——或者一个MEM。如果返回一个MEM，这只是一个给分配器的提示；其仍然有可能会决定用另一个寄存器。你可以在钩子中使用current_function_leaf_function，用来确定被询问的硬件寄存器是否会被破坏。该钩子的缺省值为NULL，其禁止了任何特殊的分配。

int TARGET_UNSPEC_MAY_TRAP_P (const_rtx x, unsigned flags) [Target Hook]

该目标钩子返回非零，如果x，一个unspec或者unspec_volatile运算，可能会造成一个陷阱。目标机可以使用该钩子来加强对unspec和unspec_volatile运算的分析的精确性。你可以调用may_trap_p_1来分析x的内部元素，这种情况下，也应该传递flags。

void TARGET_SET_CURRENT_FUNCTION (tree decl) [Target Hook]

编译器每当改变当前函数上下文(cfun)时，便会调用该钩子。你可以定义该函数，如果后端需要基于每个函数执行任何初始化或者重置行为。例如，其可以用来实现函数属性，影响寄存器的使用或者代码生成指令模式。参数decl为新的函数上下文的声明，可以为空，表示编译器已经离开函数上下文，要返回顶层去处理。缺省钩子函数不做任何事情。

GCC将cfun设置为一个哑的函数上下文，在初始化后端一些部分的时候。钩子函数在这种情况下不会被调用；你不需要担心钩子函数被递归调用，或者当后端处于部分初始化的状态。

TARGET_OBJECT_SUFFIX [Macro]

定义该宏为C字符串，表示在你的目标机器上，对象文件的后缀。如果没有定义该宏，则GCC会使用`.o'作为目标文件的后缀。

TARGET_EXECUTABLE_SUFFIX [Macro]

定义该宏为C字符串，表示在你的目标机器上，为可执行文件自动增加的后缀。如果没有定义该宏，GCC将为可执行文件使用空字符串作为后缀。

COLLECT_EXPORT_LIST [Macro]

如果被定义，collect2将在其命令行中扫描单独的目标文件，并为连接器创建一个导出列表。为AIX这样的系统定义该宏，并使用导出列表，其连接器会丢弃没有从main中引用的对象文件。

MODIFY_JNI_METHOD_CALL (mdecl) [Macro]

定义该宏为一个C表达式，表示方法调用mdecl的一个变种，如果Java Native Interface (JNI)方法必须通过你目标机上其它方法来调用。例如，在32位Microsoft Windows上，JNI方法必须使用stdcall调用约定来调用，该宏则被定义为如下表达式：

```
build_type_attribute_variant (mdecl,
                              build_tree_list
                              (get_identifier ("stdcall"),
                               NULL))
```

bool TARGET_CANNOT_MODIFY_JUMPS_P (void) [Target Hook]

该目标钩子在一个点之后返回true，该处应该创建新的跳转指令。在一些机器上，对每个跳转都要求使用寄存器，例如SH5的SHmedia ISA，该点通常为重载，所以该目标钩子应该被定义为这样的函数：

```
static bool
cannot_modify_jumps_past_reload_p ()
{
 return (reload_completed || reload_in_progress);
}
```

int TARGET_BRANCH_TARGET_REGISTER_CLASS (void) [Target Hook]

该目标钩子返回一个寄存器类，分支目标寄存器优化将会应用在该类别上。该类别中的所有寄存器应该是可以互换使用的。重载之后，该类别中的寄存器将被重新分配，并且加载将被悬挂在循环之外，从属于块间调度。

bool TARGET_BRANCH_TARGET_REGISTER_CALLEE_SAVED (bool after_prologue_epilogue_gen) [Target Hook]

分支目标寄存器优化，缺省的会将，被调用者保存的寄存器排除在外，其在当前函数中已经是不活跃的。如果目标钩子返回真，它们将被包含进来。目标代码必须确保在由`TARGET_BRANCH_TARGET_REGISTER_CLASS'返回的类别中的所有寄存器，如果需要保存的，会被保存。after_prologue_epilogue_gen指出是否序言和尾声已经被生成。注意，即使你只当after_prologue_epilogue_gen为假时返回真，你还可能要在INITIAL_ELIMINATION_OFFSET做出特定的预防，来为调用者保存的目标寄存器保留空间。

POWI_MAX_MULTS [Macro]

如果被定义，该宏被解析为一个有符号整型C表达式，描述了浮点乘法的最大数。

void TARGET_EXTRA_INCLUDES (const char *sysroot, const char *iprefix, int stdinc) [Macro]

该钩子用于记录目标机的任何额外的include文件。参数stdinc指示是否存在通常的include文件。参数sysroot为系统根目录。参数iprefix为gcc目录的前缀。

void TARGET_EXTRA_PRE_INCLUDES (const char *sysroot, const char *iprefix, int stdinc) [Macro]

该钩子用于记录目标机的任何先于标准头文件之前的include文件。参数stdinc指示是否存在通常的include文件。参数sysroot为系统根目录。参数iprefix为gcc目录的前缀。

void TARGET_OPTF (char *path) [Macro]

该目标机钩子用来记录目标机特殊的include路径。参数path为要记录的include。在Darwin系统上，被用于Framework include，其语义与`-I'有所不同。

bool TARGET_USE_LOCAL_THUNK_ALIAS_P (tree fndecl) [Target Hook]

该目标机钩子返回true，如果当为虚函数构造thunk时，使用局部别名是安全的，否则为false。缺省的，钩子对所有函数返回true，如果目标机支持别名（即：定义了ASM_OUTPUT_DEF），否则false。

TARGET_FORMAT_TYPES [Macro]

如果被定义，则该宏为全局变量的名字，其包含了目标机特定的格式检查信息，针对`-Wformat'选项。缺省为没有目标机特定的格式检查。

TARGET_N_FORMAT_TYPES [Macro]

如果被定义，则该宏为TARGET_FORMAT_TYPES中的项数。

TARGET_OVERRIDES_FORMAT_ATTRIBUTES [Macro]

如果被定义，则该宏为全局变量的名字，其包含了目标机特定的格式，用来覆盖`-Wformat'选项。缺省为没有目标机特定的格式覆盖。如果被定义，则TARGET_FORMAT_TYPES也必须被定义。

TARGET_OVERRIDES_FORMAT_ATTRIBUTES_COUNT [Macro]

如果被定义，该宏描述TARGET_OVERRIDES_FORMAT_ATTRIBUTES中的项数。

TARGET_OVERRIDES_FORMAT_INIT [Macro]

如果被定义，则该宏描述可选的初始化程序，用于目标机特定的系统printf和scanf格式设置。

bool TARGET_RELAXED_ORDERING [Target Hook]

如果设置为true，则意味着目标机的内存模型不保证，没有依赖关系的加载操作会按照指令流的顺序来访问主存；如果顺序很重要，那么必须使用显式的内存栅栏。这对许多现在的处理器是这样的，例如Alpha, PowerPC和ia64，其实现了关于内存一致性的``relaxed,'' ``weak,'' 或``release''策略。缺省为false。

const char *TARGET_INVALID_ARG_FOR_UNPROTOTYPED_FN (tree typelist, tree funcdecl, tree val) [Target Hook]

如果被定义，则当传递参数val给函数原型为typelist的函数funcdecl是非法的时候，该宏会返回诊断信息。

const char * TARGET_INVALID_CONVERSION (tree fromtype, tree totype) [Target Hook]

如果被定义，则当从fromtype转换成totype是无效的时候，该宏会返回诊断信息，或者返回NULL，如果有效性应该由前端来确定。

const char * TARGET_INVALID_UNARY_OP (int op, tree type) [Target Hook]

如果被定义，则当在类型为type的操作数上执行op（一元的加号通过CONVERT_EXPR来表示）是无效的时候，该宏会返回诊断信息，或者返回NULL，如果有效性应该由前端来确定。

const char * TARGET_INVALID_BINARY_OP (int op, tree type1, tree type2) [Target Hook]

如果被定义，则当在类型为type1和type2的操作数上执行op是无效的时候，该宏会返回诊断信息，或者返回NULL，如果有效性应该由前端来确定。

TARGET_USE_JCR_SECTION [Macro]

该宏定义是否使用JCR段来记录Java类。缺省的，如果SUPPORTS_WEAK和TARGET_HAVE_NAMED_SECTIONS都为真，则TARGET_USE_JCR_SECTION被定义为1，否则为0。

OBJC_JBLEN [Macro]

该宏为NeXT运行时，确定objective C跳转缓存的大小。缺省的，OBJC_JBLEN被定义为一个无害的值。

LIBGCC2_UNWIND_ATTRIBUTE [Macro]

定义该宏，如果对于`libgcc'中那些为调用栈展开（call stack unwinding）提供低级别支持的函数上，需要附加目标机特定的属性时。其被用在`unwind-generic.h'中的声明和那些函数相关的定义中。

void TARGET_UPDATE_STACK_BOUNDARY (void) [Target Hook]

如果需要，定义该宏来更新当前函数栈边界。

rtx TARGET_GET_DRAP_RTX (void) [Target Hook]
定义该宏为动态重对齐参数指针（Dynamic Realign Argument Pointer）的rtx，如果当栈被对齐时，需要用不同的参数指针寄存器来访问函数的参数列表。

bool TARGET_ALLOCATE_STACK_SLOTS_FOR_ARGS (void) [Target Hook]
当优化被禁止时，该钩子用来指示参数是否应该被分配到栈槽中。通常，GCC当不做优化时，会为参数分配栈槽，以便于调试。然而，当函数使用__attribute__((naked))声明时，将没有栈帧，因此编译器不能安全的将参数从用来传递它们的寄存器中移送到栈上。因此，该钩子通常应该返回真，但是对于裸露的函数应该返回假。缺省的实现总是返回真。

18 主机配置

大多数关于编译器实际运行的机器和系统的详细信息可以由 configure 脚本检测。有一些是不可能通过 configure 来检测到的；这些将有两种方式可以来描述，或者通过定义在名为 `xm-machine.h' 的文件中的宏，或者通过 `config.gcc' 中 的 out_host_hook_obj 变量所制定的钩子函数。(The intention is that very few hosts will need a header file but nearly every fully supported host will need to override some hooks.)

如果只需要定义一些宏，并且它们的定义很简单，可以考虑使用在 `config.gcc' 中的 xm_defines 变量来替代创建一个主机配置头 文件。参见 Section 6.3.2.2 [系统配置], page 48。

18.1 主机通用信息

有些东西不具有可移植性，甚至是在相似的操作系统之间，并且autoconf也难 以检测出。它们是通过钩子函数来实现的，这些函数放在 `config.gcc' 中 host_hook_obj 变量指定的文件中。

void HOST_HOOKS_EXTRA_SIGNALS (void) [Host Hook]
该主机钩子用于建立对额外信号的处理。最通用的事情是在这个钩子中去检测栈溢出。

void * HOST_HOOKS_GT_PCH_GET_ADDRESS (size_t size, int fd) [Host Hook]
该主机钩子返回很可能在编译器的后续调用中为空闲的某块空间地址。我们打算将 PCH数据加载到这个地址，从而使得不需要对数据进行重定位。该区域应该能够容纳 size 个字节。如果主机使用 mmap，则 fd 为一个打开文件的 描述符，可以用来做探测。

int HOST_HOOKS_GT_PCH_USE_ADDRESS (void * address, size_t size, int fd, size_t offset) [Host Hook]
该主机钩子会在将要加载PCH文件时被调用。我们要从 fd 中加载 size 字节到内存中 address 地址的 offset 偏移量出。给定的 address 为之前调用 HOST_HOOKS_GT_PCH_GET_ADDRESS 所得的结果。如果不能在 address 处分配 size 个字节，则返回 −1。如果分配了内 存但是没有加载数据，则返回0。如果该钩子完成了所有的事情，则返回1。

如果实现使用了保留地址空间，则会释放超出 size 的任何保留空间，而 不管返回值如何。如果不加载PCH，则该钩子可以使用 size 为0的方式调 用，这样所有保留地址空间将被释放。

不要试图不能被该执行程序返回的 address 值；直接返回 −1。这 些值通常表明了一个过时的PCH文件（由其它GCC可执行程序创建的），并且该PCH 文件是无法工作的。

size_t HOST_HOOKS_GT_PCH_ALLOC_GRANULARITY (void); [Host Hook]
该主机钩子返回分配虚拟内存所需的对齐大小。通常这与getpagesize相同，但是 在一些主机上，保留内存的对齐大小与供使用的内存页尺寸是不同的。

18.2 主机文件系统

GCC需要知道许多关于主机文件系统的语义方面的东西。具有Unix和MS-DOS语义的 文件系统会被自动检测。对于其它系统，你可以在 `xm-machine.h' 中定义下列宏。

HAVE_DOS_BASED_FILE_SYSTEM
: 如果主机文件系统服从MS-DOS而不是Unix定义的语义，则该宏会被 `system.h' 自动定义。DOS文件系统大小写不敏感，文件描述可以起始于一个驱动字母，并且 斜线和反斜线（`/' 和 `\'）都是目录分隔符。

DIR_SEPARATOR
DIR_SEPARATOR_2
: 如果定义，这些宏扩展为字符常量，用来指定在文件描述中的目录名的分隔符。在Unix和MS-DOS文件系统上，`system.h' 会自动给出合适的值。如果你的文件 系统不是这些，则在 `xm-machine.h' 中定义一个或者这两个合适的值。

  但是，像VMS这样的操作系统，构建路径名要比将目录名通过特定字符进行字符串连接复杂的多，对于这种情况，则不要定义这些宏。

PATH_SEPARATOR
: 如果定义，该宏将扩展为一个字符常量，用来指定搜寻路径中元素的分隔符。缺省 值为一个冒号（`:'）。基于DOS的系统，通常，并不是所有的，使用分号（`;'）。

VMS
: 如果主机系统为VMS，则定义该宏。

HOST_OBJECT_SUFFIX
: 定义该宏为一个C字符串，用来表示在你的主机上目标文件的后缀。如果没有定义该 宏，GCC将会使用 `.o' 作为目标文件的后缀。

HOST_EXECUTABLE_SUFFIX
: 定义该宏为一个C字符串，用来表示在你的主机上可执行文件的后缀。如果没有定义 该宏，GCC将会使用空字符串作为可执行文件的后缀。

HOST_BIT_BUCKET
: 一个路径名，由主机操作系统定义，可以作为一个文件被打开和写入内容，但是所 有写入的信息都将被丢弃。这就是众所皆知的 bit bucket，或者 null device。如果没有定义该宏，GCC将使用 `/dev/null' 作为 bit bucket。如果主机不支持bit bucket，则将该宏定义为一个无效的文件名。

UPDATE_PATH_HOST_CANONICALIZE (path)
: 如果定义，则为一个C语句（没有分号），当编译驱动器或者预处理器中使用的路径 需要被canonicalized时，用于执行主机相关的 canonicalization。path 是 被malloc出来的。如果C语句将 path canonicalize到一个不同的缓存中，则 旧的路径将被释放，并且新的缓存应该通过malloc被分配。

DUMPFILE_FORMAT
: 定义该宏为一个字符串，用来表示构建调试转储文件名字的索引部分的格式。结果 字符串必须适合15个字节。文件名的全称为：汇编文件名的前缀，应用该模式生成 的索引号，以及每种转储文件所特定的字符串，例如 `rtl'。

  如果没有定义该宏，GCC将会使用 `.%02d.'。应该在使用缺省方式会生成无效 文件名的情况下，定义该宏。

DELETE_IF_ORDINARY
: 定义该宏为一个C语句（没有分号），用来执行主机相关的编译驱动器产生的普通临 时文件的删除操作。

如果没有定义该宏，GCC将会使用缺省的版本。应该在缺省版本不能可靠的删除临时文件的情况下，定义该宏。例如，在VMS上，会允许一个文件有多个版本。

HOST_LACKS_INODE_NUMBERS
如果主机文件系统不在struct stat中报告有意义的inode数字时，则定义该宏。

18.3 关于主机的其它杂项

FATAL_EXIT_CODE
一个C表达式，作为当编译器发生严重错误退出时，所返回的状态码。缺省值为系统 提供的宏 `EXIT_FAILURE'，或者如果系统没有定义此宏时为 `1'。只在 这些缺省值不正确的时候，才定义该宏。

SUCCESS_EXIT_CODE
一个C表达式，作为当编译器没有发生严重错误而退出时，所返回的状态码。（警告 不属于严重错误。）缺省值为系统提供的宏 `EXIT_SUCCESS'，或者如果系统没 有定义此宏时为 `0'。只在这些缺省值不正确的时候，才定义该宏。

USE_C_ALLOCA
定义该宏，如果GCC应该使用 `libiberty.a' 提供的C实现的 alloca。 这只影响编译器本身的一些部分的内存分配。并不改变代码生成。

当GCC通过其它编译器而不是它本身来构建时，C alloca 总是被使用。这是 因为大多其它实现都具有严重的bug。应该只在基于栈的 alloca 可能无法工 作的系统上定义该宏。例如，如果系统在栈的大小上有一个小额限制，则GCC内建的 alloca 将无法可靠的工作。

COLLECT2_HOST_INITIALIZATION
如果定义，则为一个C语句（没有分号），当 collect2 被初始化时，执行主 机相关的初始化。

GCC_DRIVER_HOST_INITIALIZATION
如果定义，则为一个C语句（没有分号），当编译驱动器被初始化时，执行主机相关的初始化。

HOST_LONG_LONG_FORMAT
如果定义，则为一个字符串，用于表示像 printf 这样的函数的 long long 类型参数。缺省值为 ll。

另外，如果 configure 在 `auto-host.h' 中生成了任何不正确的宏 定义，你可以在一个主机配置头文件中覆盖那个定义。如果你需要这么做，请首先看看 是否可以去修补 configure。

19 Makefile片段

当使用 `configure' 脚本配置GCC时，将会从模版文件 `Makefile.in' 中构建 `Makefile' 文件。这个时候，会将 `config' 目录下的makefile片段合在一起。这些片段用来设置不 能被autoconf检测计算出的 Makefile 参数。要合并的片段列表由 `config.gcc'（以及偶尔由 `config.build' 和 `config.host'）设置；参见 Section 6.3.2.2 [系统配置], page 48。

片段命名为 `t-target' 或者 `x-host'，取决 于它们是否与配置GCC来产生特定目标代码相关，或者配置GCC来运行在特定主机。这里的 target 和 host 是（Here target and host

are mnemonics which usually have some relationship to the canonical system name, but no formal connection.)

如果不存在这些文件，则意味着不需要对给定目标或主机添加什么。大多 数目标机需要一些 `t-target' 片段，不过需要 `x-host' 片段的很少。

19.1 目标机Makefile片段

目标makefile片段能够设置这些Makefile变量。

LIBGCC2_CFLAGS
: 编译 `libgcc2.c' 时使用的编译器标记。

LIB2FUNCS_EXTRA
: 将被编译或汇编，并插入 `libgcc.a' 的源文件名列表。

Floating Point Emulation
: 要使GCC在 `libgcc.a' 中包括软浮点库，则使用下面的一些规则 来定义 FPBIT 和 DPBIT：

```
# We want fine grained libraries, so use the new code
# to build the floating point emulation libraries.
FPBIT = fp-bit.c
DPBIT = dp-bit.c


fp-bit.c: $(srcdir)/config/fp-bit.c
    echo '#define FLOAT' > fp-bit.c
    cat $(srcdir)/config/fp-bit.c >> fp-bit.c

dp-bit.c: $(srcdir)/config/fp-bit.c
    cat $(srcdir)/config/fp-bit.c > dp-bit.c
```

可能需要在 `fp-bit.c' 和 `dp-bit.c' 的开始处提供额外的 #define来控制目标机大小端和其它选项。

CRTSTUFF_T_CFLAGS
: 编译 `crtstuff.c' 时使用的特定标记。see Section 17.21.5 [初始化], page 368。

CRTSTUFF_T_CFLAGS_S
: 编译共享连接的 `crtstuff.c' 时使用的特定标记。用于在 EXTRA-PARTS 中使用 `crtbeginS.o' 和 `crtendS.o' 时。see Section 17.21.5 [初始化], page 368。

MULTILIB_OPTIONS
: 对于一些目标机，使用不同方式调用GCC所产生的目标对象不能被一起连接。 例如，对于一些目标机，GCC可产生大端和小端代码。对于这些目标机，必须 安排编译多个版本的 `libgcc.a'，对应于每个不相兼容的选项集。当 GCC调用连接器时，它会根据使用的命令行选项来安排连接正确版本的 `libgcc.a'。

  MULTILIB_OPTIONS 宏列出了必须构建特定版本 `libgcc.a' 的 选项集。将互不兼容的选项并排写出，并由斜线分隔。将可以一起使用的选项 由空格分开。构建程序将会构建所有兼容选项的组合。

  例如，如果将 MULTILIB_OPTIONS 设置为 `m68000/m68020 msoft-float'，`Makefile' 将会使用下列选项集来构建特定版本的 `libgcc.a'： `-m68000', `-m68020', `-msoft-float', `-m68000 -msoft-float', 和 `-m68020 -msoft-float'。

MULTILIB_DIRNAMES

如果使用了 MULTILIB_OPTIONS，该变量指定了用于存放不同库的目录名。MULTILIB_OPTIONS 中的每个元素，对应于 MULTILIB_DIRNAMES 中 的每个元素。如果没有使用 MULTILIB_DIRNAMES，缺省值将为 MULTILIB_OPTIONS，并使用斜线来替代空格。

例如，如果 MULTILIB_OPTIONS 设置为 `m68000/m68020 msoft-float'，则 MULTILIB_DIRNAMES 的缺省值为 `m68000 m68020 msoft-float'。如果需要不同的目录名时，可以指定不同的值。

MULTILIB_MATCHES

有时，相同的选项可以被写成两种不同的方式。如果一个选项在 MULTILIB_OPTIONS 式列出，GCC需要知道它的任何同义形式。这种情况下，将 MULTILIB_MATCHES 设置为 `option=option' 形式的列表来描述 所有相关的同义词。例如，`m68000=mc68000 m68020=mc68020'。

MULTILIB_EXCEPTIONS

有时，当 MULTILIB_OPTIONS 中指定了多个选项集时，会有些组合不能被 构建。这种情况下，将 MULTILIB_EXCEPTIONS 设置为所有不被构建的例外。

例如ARM处理器不能执行同时执行硬件浮点指令和缩减大小的THUMB指令，这样就没有必要构建这些选项组合的库。因此将 MULTILIB_EXCEPTIONS 设为：

```
*mthumb/*mhard-float*
```

MULTILIB_EXTRA_OPTS

有时当构建多版本 `libgcc.a' 时，有些选项需要总是被传给编译器。这种情 况下，将 MULTILIB_EXTRA_OPTS 设置为用于所有构建时的选项列表。如果设 置了该宏，则可能要将 CRTSTUFF_T_CFLAGS 设置为跟在其后的破折号。

NATIVE_SYSTEM_HEADER_DIR

如果系统头文件的缺省位置不是 `/usr/include'，则必须将该宏设置为包含头 文件的目录。该值应该匹配 SYSTEM_INCLUDE_DIR 宏的值。

SPECS 不幸的是，设置 MULTILIB_EXTRA_OPTS 并不足够，因为它并不影响目标库的 构建，最起码对于缺省multilib的构建是这样的。一种可能的方法是使用 DRIVER_SELF_SPECS 从 `specs' 文件中取得选项，就像是它们被传给了 编译器驱动程序的命令行。但是，你不想在安装工具链之后再增加这些选项，所以你 可以在安装原始的内嵌 `specs' 时，调节在构建工具链时使用的 `specs' 文件。诀窍是将 SPECS 设置为其它文件名（例如 `specs.install'），然后该文件将会由内建 `specs' 创建出来，并且引出一个 `Makefile' 规 则来生成 `specs' 文件，在构建时候使用。

T_CFLAGS These are extra flags to pass to the C compiler. They are used both when building GCC, and when compiling things with the just-built GCC. This variable is deprecated and should not be used.

19.2 主机Makefile片段

不鼓励使用 `x-host' 片段。应该只在没有其它机制可以获得所需要的时候 才使用。

20 collect2

GCC使用叫做 collect2 的工具，在几乎所有的系统上，来安排在起始 时候调用不同的初始化函数。

程序 collect2 的工作方式是通过初次连接程序，并查找连接器输出 文件中的指示为构造函数的特定名字符号；如果找到，则会创建一个新的包含 这些符号的临时文件 `.c'，编译该文件，并再次连接程序。

实际调用构造者的是叫做 __main 的子程序，其在main函数体（假若 main 是由GNU CC编译的）的开始处被（自动的）调用。调用 __main 是必需的，即使当编译C代码，并允许将C和C++目标代码连接 一起时。如果使用了 `-nostdlib'，则会得到对 __main 未解 决的引用(unresolved reference) 这样的错误，这是因为它是定义在标准GCC库 中。将 `-lgcc' 包含在编译器命令行的后面便可以解决这个引用。

程序 collect2 被作为 ld 安装在编译器过程所被安装的目录 下。当 collect2 需要找到 真正的 ld 时，它会尝试 下面的文件名：

- 编译器搜索目录下的 `real-ld'。
- 环境变量 PATH 所列出的目录下的 `real-ld'。
- 如果制定了配置宏 REAL_LD_FILE_NAME 则搜索该文件。
- 编译器搜索目录下的 `ld'，除了能导致 collect2 递归执行自己 的以外。
- PATH 下的 `ld'。

“编译器搜索目录” 是指的 gcc 针对编译器过程所搜寻的全部目录。 这包括通过 `-B' 制定的目录。

交叉编译器的搜寻方式有一些不同：

- 编译器搜索目录下的 `real-ld'。
- PATH 下的 `target-real-ld'。
- 如果制定了配置宏 REAL_LD_FILE_NAME 则搜索该文件。
- 编译器搜索目录下的 `ld'。
- PATH 下的 `target-ld'。

collect2 显示的避免使用调用 collect2 的文件名来运行 ld。实际上，它记录了一个名字列表——以防一个 collect2 版本会 找到另一个版本的 collect2。

collect2 使用上面针对 ld 的相同算法来搜寻工具 nm 和 strip。

21 标准头文件目录

GCC_INCLUDE_DIR 意味着对于本地和交叉都相同的。这是GCC存放私有 包含文件和修订过的（fixed）包含文件的地方。交叉编译的GCC会对 `$(tooldir)/include' 下的头文件运行 fixincludes。（如果 交叉编译头文件需要被修订，它们必须在构建GCC之前安装。如果交叉编译头 文件已经适合GCC，则不需要任何特定修改。）

`GPLUSPLUS_INCLUDE_DIR' 意味着对于本地和交叉都相同的。这是 g++ 首先查找头文件的地方。C++库只安装目标独立的头文件到这 里。

LOCAL_INCLUDE_DIR 只被本地编译器使用，GCC并不安装任何文件到里 面。正常情况下为 `/usr/local/include'。这是系统放置额外头文件的 地方。

CROSS_INCLUDE_DIR 只被交叉编译器使用，GCC并不安装任何文件到里 面。

TOOL_INCLUDE_DIR 被本地和交叉编译器共同使用。这是其它安装包安 装头文件以供GCC使用的地方。对于交叉编译器，这相当于 `/usr/include'。当构建交叉编译器时，fixincludes 会处理该 目录下的任何文件。

22 内存管理和类型信息

GCC使用了一些相当复杂微妙的内存管理技术，包括从GCC源代码中确定GCC的 数据结构的信息，并使用该信息来执行垃圾收集和实现预编译头文件。

使用完整的C解析器来完成这项工作会非常复杂，因此只解析C的有限子集， 并且使用特定的标记来确定源代码的哪些部分需要分析。所有的 `struct` 和 `union` 声明，如果所定义的数据结构要在垃圾收集 器的控制下进行分配，则必须被标记。所有的全局变量，如果所保存的指针是 指向垃圾收集的内存，则也必须被标记。最后，所有的全局变量，如果需要通 过预编译头文件来保存和恢复，则必须被标记。（ 预编译头文件机制只能保存 标量，复杂的数据结构必须被分配在垃圾收集内存中，从而被保存在预编译头 文件中。 ）

标记的完整格式是

```
GTY ((option) [(param)], [option] [(param)] ...))
```

不过大多数情况下，不需要选项。虽然这样，外面的双括号依然是必须的： `GTY(())`。标记可以出现在：

- 在结构体定义中，放在开括号之前；
- 在全局变量声明中，放在关键词 `static` 或者 `extern` 之后；
- 在结构体的域的定义里，在域的名字之前。

这里有一些标记简单数据结构和全局变量的例子。

```
struct tag GTY(())
{
 fields...
};

typedef struct tag GTY(())
{
 fields...
} *typename;

static GTY(()) struct tag *list;  /* points to GC memory */
static GTY(()) int counter;       /* save counter in a PCH */
```

解析器能够理解简单的typedef，例如 `typedef struct tag *name;` 和 `typedef int name;`。这些不需要被标记。

22.1 GTY(())的内部

有时候C代码不足以完全描述类型结构体，这时可以使用 `GTY` 选项和 额外标记来提供额外的信息。一些选项接受一个参数，其可以为字符串或者类 型名。如果一个选项不需要参数，则可以完全省略参数，或者提供一个空字符 串作为参数。例如，`GTY ((skip))` 和 `GTY ((skip(""))` 是等价的。

当参数为字符串时，通常为一个C代码片段。有四种特定换码符可以在字符串 中使用，用来指定被标记的数据结构体：

`%h` 当前结构体。

`%1` 直接包含当前结构体的结构体。

`%0` 包含当前结构体的最外层结构体。

`%a` `[i1][i2]...` 形式的部分表达式，用来索引当前被标记的数组项。

例如，假设有一个结构体

```
struct A {
 ...
};
struct B {
 struct A foo[12];
};
```

并且 b 是 struct B 类型的变量。当标记 `b.foo[11]' 时，%h 将扩展为 `b.foo[11]'，%0 和 %1 都会 扩展为 `b'，%a 会扩展为 `[11]'。

由于原始的C中，相邻的字符串会被连接；这对于复杂的表达式是有帮助的。

```
GTY ((chain_next ("TREE_CODE (&%h.generic) == INTEGER_TYPE"
          "?TYPE_NEXT_VARIANT (&%h.generic)"
          ":TREE_CHAIN (&%h.generic)")))
```

可用的选项：

length ("expression")

有两个地方需要显示的告诉类型机构一个数组的长度。第一种情况是当一个结 构体结束于一个可变长度数组，像这样：

```
struct rtvec_def GTY(()) {
 int num_elem;     /* number of elements */
 rtx GTY ((length ("%h.num_elem"))) elem[1];
};
```

在这种情况下，length 选项用来覆盖指定数组的长度（通常本应该为 1）。选项的参数是C代码片断用来计算长度。

第二种情况是当一个结构体或者全局变量包含一个指向数组的指针，像这样：

```
tree *
 GTY ((length ("%h.regno_pointer_align_length"))) regno_decl;
```

在这种情况下，regno_decl 已经通过类似下面的方式被分配：

```
 x->regno_decl =
  ggc_alloc (x->regno_pointer_align_length * sizeof (tree));
```

并且 length 提供了指定域的长度。

lenght 的第二种用法还包括在全局变量上，像这样：

```
static GTY((length ("reg_base_value_size")))
 rtx *reg_base_value;
```

skip

如果 skip 应用在一个域上，则类型机构将会忽略该域。这有些危险； 唯一安全的使用方式是在一个联合体中，当一个域确实不会被使用到的时候。

desc ("expression")
tag ("constant")
default

类型机构需要知道 union 的哪一个域是当前活跃的。这是通过赋给每个 域一个常数 tag 值，并且使用 desc 指定一个判别器来完成的。 由 desc 给出的表达式的值用来与每个 tag 值比较，每个 tag 值应该不同。如果没有 tag 匹配，则会使用标记为 default 的域。

在 desc 选项中，“当前结构体”是指要进行判别的联合体，可以使用 %1 来指定。 tag 选项没有换码符可用，因为其为常数。

例如,

```
struct tree_binding GTY(())
{
 struct tree_common common;
 union tree_binding_u {
  tree GTY ((tag ("0"))) scope;
  struct cp_binding_level * GTY ((tag ("1"))) level;
 } GTY ((desc ("BINDING_HAS_LEVEL_P ((tree)&%0)"))) xscope;
 tree value;
};
```

在这个例子中，当BINDING_HAS_LEVEL_P应用到 struct tree_binding* 时，其值会被假设为0或者1。如果是1，类型机制则会认为域 level 存在， 如果是0，则会认为域 scope 存在。

param_is (type)
use_param

有时候，定义某种数据结构作为通用指针（也就是 PTR），并且与特定 类型一起使用是比较方便的。param_is 指定了所指向的真正类型， use_param 说明了该类型应该放在通用数据结构的哪个地方。

例如，为了让 htab_t 指向trees，则应该像这样来写 htab_t 的定义：

```
typedef struct GTY(()) {
 ...
 void ** GTY ((use_param, ...)) entries;
 ...
} htab_t;
```

然后按这种方式声明变量：

```
 static htab_t GTY ((param_is (union tree_node))) ict;
```

paramn_is (type)
use_paramn

在更复杂的情况下，数据结构可能需要工作在多个不同类型之上，而且这些类型 也不必都是指针。对于这样的，可以使用 param1_is 到 param9_is 来指定由 use_param1 到 use_param9 标识的 实际类型域。

use_params

当结构体包含另一个参数化的结构体时，不需要做特别的处理，内部结构体会继 承外部的参数。当结构体包含指向一个参数化的结构体的指针时，类型机构不会 自动检测到（是应该可以的，只是还没有实现），所以需要告诉类型机构所指向 的结构体将使用外部结构体的相同参数。这可以通过使用 usr_params 选项来标识指针。

deletable

将 deletable 应用到全局变量上时，表示当垃圾收集运行时，不需要标记 由该变量指向的任何对象，可以只是将其设为 NULL。这可以用来维护一个 可以重用的空闲结构体列表。

if_marked ("expression")

假设你想要一些类别的对象是唯一的，并且为此你将它们放在了哈希表中。如果垃 圾搜集标记了哈希表，这些对象将永远不会被释放掉，即使最后一个引用也不存在。 对此GCC有特定的处理方式：如果你使用 if_marked 选项在一个全局哈希表 上，GCC将会对每个哈希表项调用该选项参数命名的函数。如果函数返回非0，哈希 表项将按照通常的方式被标记，如果返回0，则哈希表项将会被删除。

函数 ggc_marded_p 可以用来判断一个元素是否已经被标记。实际上，通常 的情况是使用 if_marked ("ggc_marked_p")。

mark_hook ("hook-routine-name")

如果用在结构体或者联合体类型上，给出的（双引号之间的） hook-routine-name 则为一个函数名，其在垃圾搜集器刚刚标记数据为可达 （ reachable ）时会被调用。该函数不应该改变数据，或者调用任何ggc函数。它的 唯一参数是一个指向刚刚被标记的结构体或联合体的指针。

maybe_undef

当应用到一个域时，maybe_undef 表示可以允许该域所指向的结构体没有被 定义，只要该域总是为 NULL。这可以用来避免要求后端去定义一些可选的 结构体。该选项对语言前端不起作用。

nested_ptr (type, "to expression", "from expression")

类型设备期望所有指针都指向一个对象的起始处。有时候出于抽象目的，使用指向 对象内部的指针是比较方便的。只要能够对原始对象和指针进行相互转换，这样的 指针还是可以使用的。type 是原始对象的类型，to expression 返回 给定原始对象的指针，from expression 返回给定指针的原始对象。指针可 以使用 %h 转换符得到。

chain_next ("expression")
chain_prev ("expression")
chain_circular ("expression")

让类型设备知道对象是否经常被链接在长的链表中是有帮助作用的。这可以让其 使用遍历链表的方式来替代递归调用，从而使得生成的代码使用很少的栈空间。 chain_next 是链表中的下一项，chain_prev 是前一项。对于单 向链表，只使用 chain_next；对于双向链表，两者都使用。设备要求对 一个项求 chain_prev，然后 chain_next，可以得到原始的项。

reorder ("function name")

一些数据结构依赖于相应的指针顺序。如果预编译头文件设备需要改变顺序，其将会调用由 reorder 选项指定的函数。函数必须接收 4 个参数，`void *, void *, gt_pointer_operator, void *'。第一个参数是指 向更新对象的结构体的指针，或者对象本身，如果没有包含的结构体。第二个参 数为一个cookie，目前被忽略。第三个参数是一个函数，给定指针，将会更新该 指针为正确的新值。第四个参数是一个cookie，且必须传给第二个参数。

PCH无法处理依赖于指针绝对值得数据结构。reorder 函数代价很高。在 可能的情况下，最好依赖于数据的属性，像ID号或者字符串的哈希值。

special ("name")

special 选项用来标记类型必须由特定情况的机制来处理。参数是特定 情况的名字。详细信息参见 `gengtype.c'。应避免添加新的特定情况，除 非没有别的办法。

22.2 为垃圾收集器标记Roots

除了了解类型信息，类型机构（type machinery）还需要定位全局变量 （ roots ），作为垃圾搜集器开始的地方。Roots必须按照下面的语法之一 来声明：

- extern GTY(([options])) type name;
- static GTY(([options])) type name;

语法

- GTY(([options])) type name;

是 不 被接受的。对于这样的变量，应该在某个头文件中存在一个 extern 声明——可以在那里标记，而不要在它的定义中标记。或者， 如果变量只在一个文件中使用，则将其定义为 static。

22.3 包含类型信息的源文件

只要向之前没有 GTY 标记的源文件里添加 GTY 标记，或者 创建一个新的包含 GTY 标记的源文件，那么就有三件事情需要做：

1. 需要将文件添加到类型机构需要扫描的源文件列表中。有四种情况：
   a. 对于一个后端文件，通常会自动完成。如果没有，则需要将其添加到 `config.gcc' 里的适当port条目 target_gtfiles 中。
   b. 对于所用前端共享的文件，将文件名添加到 `Makefile.in' 里的 GTFILES 变量中。
   c. 对于一个前端的文件，将文件名添加到在适当 `config-lang.in' 里定义的 gtfiles 变量中。对于C，文件为 `c-config-lang.in'。
   d. 对于一些而不是所有的前端的文件，将文件名添加到 所有 使用 它的前端的 gtfiles 变量中。
2. 如果是头文件，则需要检查是否被包含在正确的位置，使得对于生成文件 为可见的。对于一个后端文件，这应该是自动完成的。对于前端的头文件， 应该被包含 `gtype-lang.h' 的同一文件所包含。对于其它头 文件，需要被包含在 `gtype-desc.c' 中。`gtype-desc.c' 为 生成文件，所以需要将头文件名添加到 `gengtype.c' 里的 open_base_file 里的 ifiles 中。

   对于不是头文件的源文件，类型机构将会生成一个头文件，并且该头文件 应该被包含在所修改的源文件中。文件名为 `gt-path.h'，其 中 path 是相对于 `gcc' 目录的路径名，并且由 - 来替换斜线。例如，要被包含在 `cp/parser.c' 中的头文件命名为 `gt-cp-parser.c'。生成的头文件应该被包含在源文件所有其它内容 之后。不要忘记将该文件在 `Makefile' 中作为一个依赖条件！

对于语言前端，还有另一个文件需要在某处被包含。其为 `gtype-lang.h'，其中 lang 是语言子目录的名字。

22.4 如何调用垃圾收集器

The GCC garbage collector GGC is only invoked explicitly. In contrast with many other garbage collectors, it is not implicitly invoked by allocation routines when a lot of memory has been consumed. So the only way to have GGC reclaim storage it to call the ggc_collect function explicitly. This call is an expensive operation, as it may have to scan the entire heap. Beware that local variables (on the GCC call stack) are not followed by such an invocation (as many other garbage collectors do): you should reference all your data from static or external GTY-ed variables, and it is advised to call ggc_collect with a shallow call stack. The GGC is an exact mark and sweep garbage collector (so it does not scan the call stack for pointers). In practice GCC passes don't often call ggc_collect themselves, because it is called by the pass manager between passes.

资助自由软件

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers---the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, ``We will donate ten dollars to the Frobnitz project for each disk sold.'' Don't be satisfied with a vague promise, such as ``A portion of the profits are donated,'' since it doesn't give a basis for comparison.

Even a precise fraction ``of the profits from this disk'' is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is ``the proper thing to do'' when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

GNU项目和GNU/Linux

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for ``GNU's Not Unix''; it is pronounced ``guh-NEW''.) Variants of the GNU operating system, which use the kernel Linux, are now widely used; though these systems are often referred to as ``Linux'', they are more accurately called GNU/Linux systems.

For more information, see:

http://www.gnu.org/
http://www.gnu.org/gnu/linux-and-gnu.html

GNU通用公共授权

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   ``This License'' refers to version 3 of the GNU General Public License.

   ``Copyright'' also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   ``The Program'' refers to any copyrightable work licensed under this License. Each licensee is addressed as ``you''. ``Licensees'' and ``recipients'' may be individuals or organizations.

   To ``modify'' a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a ``modified version'' of the earlier work or a work ``based on'' the earlier work.

   A ``covered work'' means either the unmodified Program or a work based on the Program.

   To ``propagate'' a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To ``convey'' a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays ``Appropriate Legal Notices'' to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The ``source code'' for a work means the preferred form of the work for making modifications to it. ``Object code'' means any non-source form of a work.

   A ``Standard Interface'' means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The ``System Libraries'' of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A ``Major Component'', in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The ``Corresponding Source'' for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any

non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to ``keep intact all notices''.
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an ``aggregate'' if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
   b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing

this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A ``User Product'' is either (1) a ``consumer product'', which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, ``normally used'' refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

``Installation Information'' for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third

party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

``Additional permissions'' are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
   d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
   e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
   f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered ``further restrictions'' within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An ``entity transaction'' is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations.

If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A ``contributor'' is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's ``contributor version''.

A contributor's ``essential patent claims'' are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, ``control'' includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a ``patent license'' is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To ``grant'' such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. ``Knowingly relying'' means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate,

modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is ``discriminatory'' if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License ``or any later version'' applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM ``AS IS'' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the ``copyright'' line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.
```

```
This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an ``about box''.

You should also get your employer (if you work as a programmer) or school, if any, to sign a ``copyright disclaimer'' for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

GNU自由文档授权

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of ``copyleft'', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to

software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ``Document'', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ``you''. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A ``Modified Version'' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A ``Secondary Section'' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The ``Invariant Sections'' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The ``Cover Texts'' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

   A ``Transparent'' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not ``Transparent'' is called ``Opaque''.

   Examples of suitable formats for Transparent copies include plain *ascii* without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary

word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ``Title Page'' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ``Title Page'' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section ``Entitled XYZ'' means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as ``Acknowledgements'', ``Dedications'', ``Endorsements'', or ``History''.) To ``Preserve the Title'' of such a section when you modify the Document means that it remains a section ``Entitled XYZ'' according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque

copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
   D. Preserve all the copyright notices of the Document.
   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
   H. Include an unaltered copy of this License.
   I. Preserve the section Entitled ``History'', Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ``History'' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
   J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in

the Document for previous versions it was based on. These may be placed in the ``History'' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled ``Acknowledgements'' or ``Dedications'', Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled ``Endorsements''. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled ``Endorsements'' or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ``Endorsements'', provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or

publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ``History'' in the various original documents, forming one section Entitled ``History''; likewise combine any sections Entitled ``Acknowledgements'', and any sections Entitled ``Dedications''. You must delete all sections Entitled ``Endorsements.''

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ``aggregate'' if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled ``Acknowledgements'', ``Dedications'', or ``History'', the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or

distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ``or any later version'' applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

附录：如何为你的文档使用该授权

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) year your name.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
> Texts. A copy of the license is included in the section entitled ``GNU
> Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the ``with...Texts.'' line with this:

> with the Invariant Sections being list their titles, with
> the Front-Cover Texts being list, and with the Back-Cover Texts
> being list.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

GCC的贡献者

The GCC project would like to thank its many contributors. Without them the project would not have been nearly as successful as it has been. Any omissions in this list are accidental. Feel free to contact law@redhat.com or gerald@pfeifer.com if you have been left out or some of your contributions are not listed. Please keep this list in alphabetical order.

- Analog Devices helped implement the support for complex data types and iterators.
- John David Anglin for threading-related fixes and improvements to libstdc++-v3, and the HP-UX port.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Abramo and Roberto Bagnara for the SysV68 Motorola 3300 Delta Series port.
- Alasdair Baird for various bug fixes.
- Giovanni Bajo for analyzing lots of complicated C++ problem reports.
- Peter Barada for his work to improve code generation for new ColdFire cores.
- Gerald Baumgartner added the signature extension to the C++ front end.
- Godmar Back for his Java improvements and encouragement.
- Scott Bambrough for help porting the Java compiler.
- Wolfgang Bangerth for processing tons of bug reports.
- Jon Beniston for his Microsoft Windows port of Java.
- Daniel Berlin for better DWARF2 support, faster/better optimizations, improved alias analysis, plus migrating GCC to Bugzilla.
- Geoff Berry for his Java object serialization work and various patches.

- Uros Bizjak for the implementation of x87 math built-in functions and for various middle end and i386 back end improvements and bug fixes.
- Eric Blake for helping to make GCJ and libgcj conform to the specifications.
- Janne Blomqvist for contributions to GNU Fortran.
- Segher Boessenkool for various fixes.
- Hans-J. Boehm for his garbage collector, IA-64 libffi port, and other Java work.
- Neil Booth for work on cpplib, lang hooks, debug hooks and other miscellaneous clean-ups.
- Steven Bosscher for integrating the GNU Fortran front end into GCC and for contributing to the tree-ssa branch.
- Eric Botcazou for fixing middle- and backend bugs left and right.
- Per Bothner for his direction via the steering committee and various improvements to the infrastructure for supporting new languages. Chill front end implementation. Initial implementations of cpplib, fix-header, config.guess, libio, and past C++ library (libg++) maintainer. Dreaming up, designing and implementing much of GCJ.
- Devon Bowen helped port GCC to the Tahoe.
- Don Bowman for mips-vxworks contributions.
- Dave Brolley for work on cpplib and Chill.
- Paul Brook for work on the ARM architecture and maintaining GNU Fortran.
- Robert Brown implemented the support for Encore 32000 systems.
- Christian Bruel for improvements to local store elimination.
- Herman A.J. ten Brugge for various fixes.
- Joerg Brunsmann for Java compiler hacking and help with the GCJ FAQ.
- Joe Buck for his direction via the steering committee.
- Craig Burley for leadership of the G77 Fortran effort.
- Stephan Buys for contributing Doxygen notes for libstdc++.
- Paolo Carlini for libstdc++ work: lots of efficiency improvements to the C++ strings, streambufs and formatted I/O, hard detective work on the frustrating localization issues, and keeping up with the problem reports.
- John Carr for his alias work, SPARC hacking, infrastructure improvements, previous contributions to the steering committee, loop optimizations, etc.
- Stephane Carrez for 68HC11 and 68HC12 ports.
- Steve Chamberlain for support for the Renesas SH and H8 processors and the PicoJava processor, and for GCJ config fixes.
- Glenn Chambers for help with the GCJ FAQ.
- John-Marc Chandonia for various libgcj patches.
- Scott Christley for his Objective-C contributions.
- Eric Christopher for his Java porting help and clean-ups.
- Branko Cibej for more warning contributions.
- The GNU Classpath project for all of their merged runtime code.

- Nick Clifton for arm, mcore, fr30, v850, m32r work, `--help', and other random hacking.
- Michael Cook for libstdc++ cleanup patches to reduce warnings.
- R. Kelley Cook for making GCC buildable from a read-only directory as well as other miscellaneous build process and documentation clean-ups.
- Ralf Corsepius for SH testing and minor bug fixing.
- Stan Cox for care and feeding of the x86 port and lots of behind the scenes hacking.
- Alex Crain provided changes for the 3b1.
- Ian Dall for major improvements to the NS32k port.
- Paul Dale for his work to add uClinux platform support to the m68k backend.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Russell Davidson for fstream and stringstream fixes in libstdc++.
- Bud Davis for work on the G77 and GNU Fortran compilers.
- Mo DeJong for GCJ and libgcj bug fixes.
- DJ Delorie for the DJGPP port, build and libiberty maintenance, various bug fixes, and the M32C port.
- Arnaud Desitter for helping to debug GNU Fortran.
- Gabriel Dos Reis for contributions to G++, contributions and maintenance of GCC diagnostics infrastructure, libstdc++-v3, including `valarray<>`, `complex<>`, maintaining the numerics library (including that pesky `<limits>` :-) and keeping up-to-date anything to do with numbers.
- Ulrich Drepper for his work on glibc, testing of GCC using glibc, ISO C99 support, CFG dumping support, etc., plus support of the C++ runtime libraries including for all kinds of C interface issues, contributing and maintaining `complex<>`, sanity checking and disbursement, configuration architecture, libio maintenance, and early math work.
- Zdenek Dvorak for a new loop unroller and various fixes.
- Richard Earnshaw for his ongoing work with the ARM.
- David Edelsohn for his direction via the steering committee, ongoing work with the RS6000/PowerPC port, help cleaning up Haifa loop changes, doing the entire AIX port of libstdc++ with his bare hands, and for ensuring GCC properly keeps working on AIX.
- Kevin Ediger for the floating point formatting of num_put::do_put in libstdc++.
- Phil Edwards for libstdc++ work including configuration hackery, documentation maintainer, chief breaker of the web pages, the occasional iostream bug fix, and work on shared library symbol versioning.
- Paul Eggert for random hacking all over GCC.
- Mark Elbrecht for various DJGPP improvements, and for libstdc++ configuration support for locales and fstream-related fixes.
- Vadim Egorov for libstdc++ fixes in strings, streambufs, and iostreams.
- Christian Ehrhardt for dealing with bug reports.
- Ben Elliston for his work to move the Objective-C runtime into its own subdirectory and for his work on autoconf.

- Revital Eres for work on the PowerPC 750CL port.
- Marc Espie for OpenBSD support.
- Doug Evans for much of the global optimization framework, arc, m32r, and SPARC work.
- Christopher Faylor for his work on the Cygwin port and for caring and feeding the gcc.gnu.org box and saving its users tons of spam.
- Fred Fish for BeOS support and Ada fixes.
- Ivan Fontes Garcia for the Portuguese translation of the GCJ FAQ.
- Peter Gerwinski for various bug fixes and the Pascal front end.
- Kaveh R. Ghazi for his direction via the steering committee, amazing work to make `-W -Wall -W* -Werror' useful, and continuously testing GCC on a plethora of platforms. Kaveh extends his gratitude to the CAIP Center at Rutgers University for providing him with computing resources to work on Free Software since the late 1980s.
- John Gilmore for a donation to the FSF earmarked improving GNU Java.
- Judy Goldberg for c++ contributions.
- Torbjorn Granlund for various fixes and the c-torture testsuite, multiply- and divide-by-constant optimization, improved long long support, improved leaf function register allocation, and his direction via the steering committee.
- Anthony Green for his `-Os' contributions and Java front end work.
- Stu Grossman for gdb hacking, allowing GCJ developers to debug Java code.
- Michael K. Gschwind contributed the port to the PDP-11.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Mostafa Hagog for Swing Modulo Scheduling (SMS) and post reload GCSE.
- Bruno Haible for improvements in the runtime overhead for EH, new warnings and assorted bug fixes.
- Andrew Haley for his amazing Java compiler and library efforts.
- Chris Hanson assisted in making GCC work on HP-UX for the 9000 series 300.
- Michael Hayes for various thankless work he's done trying to get the c30/c40 ports functional. Lots of loop and unroll improvements and fixes.
- Dara Hazeghi for wading through myriads of target-specific bug reports.
- Kate Hedstrom for staking the G77 folks with an initial testsuite.
- Richard Henderson for his ongoing SPARC, alpha, ia32, and ia64 work, loop opts, and generally fixing lots of old problems we've ignored for years, flow rewrite and lots of further stuff, including reviewing tons of patches.
- Aldy Hernandez for working on the PowerPC port, SIMD support, and various fixes.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Kazu Hirata for caring and feeding the Renesas H8/300 port and various fixes.
- Katherine Holcomb for work on GNU Fortran.
- Manfred Hollstein for his ongoing work to keep the m88k alive, lots of testing and bug fixing, particularly of GCC configury code.

- Steve Holmgren for MachTen patches.
- Jan Hubicka for his x86 port improvements.
- Falk Hueffner for working on C and optimization bug reports.
- Bernardo Innocenti for his m68k work, including merging of ColdFire improvements and uClinux support.
- Christian Iseli for various bug fixes.
- Kamil Iskra for general m68k hacking.
- Lee Iverson for random fixes and MIPS testing.
- Andreas Jaeger for testing and benchmarking of GCC and various bug fixes.
- Jakub Jelinek for his SPARC work and sibling call optimizations as well as lots of bug fixes and test cases, and for improving the Java build system.
- Janis Johnson for ia64 testing and fixes, her quality improvement sidetracks, and web page maintenance.
- Kean Johnston for SCO OpenServer support and various fixes.
- Tim Josling for the sample language treelang based originally on Richard Kenner's ``toy'' language.
- Nicolai Josuttis for additional libstdc++ documentation.
- Klaus Kaempf for his ongoing work to make alpha-vms a viable target.
- Steven G. Kargl for work on GNU Fortran.
- David Kashtan of SRI adapted GCC to VMS.
- Ryszard Kabatek for many, many libstdc++ bug fixes and optimizations of strings, especially member functions, and for auto_ptr fixes.
- Geoffrey Keating for his ongoing work to make the PPC work for GNU/Linux and his automatic regression tester.
- Brendan Kehoe for his ongoing work with G++ and for a lot of early work in just about every part of libstdc++.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination and delay slot scheduling. Richard Kenner was also the head maintainer of GCC for several years.
- Mumit Khan for various contributions to the Cygwin and Mingw32 ports and maintaining binary releases for Microsoft Windows hosts, and for massive libstdc++ porting work to Cygwin/Mingw32.
- Robin Kirkham for cpu32 support.
- Mark Klein for PA improvements.
- Thomas Koenig for various bug fixes.
- Bruce Korb for the new and improved fixincludes code.

- Benjamin Kosnik for his G++ work and for leading the libstdc++-v3 effort.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Asher Langton and Mike Kumbera for contributing Cray pointer support to GNU Fortran, and for other GNU Fortran improvements.
- Jeff Law for his direction via the steering committee, coordinating the entire egcs project and GCC 2.95, rolling out snapshots and releases, handling merges from GCC2, reviewing tons of patches that might have fallen through the cracks else, and random but extensive hacking.
- Marc Lehmann for his direction via the steering committee and helping with analysis and improvements of x86 performance.
- Victor Leikehman for work on GNU Fortran.
- Ted Lemon wrote parts of the RTL reader and printer.
- Kriang Lerdsuwanakij for C++ improvements including template as template parameter support, and many C++ fixes.
- Warren Levy for tremendous work on libgcj (Java Runtime Library) and random work on the Java front end.
- Alain Lichnewsky ported GCC to the MIPS CPU.
- Oskar Liljeblad for hacking on AWT and his many Java bug reports and patches.
- Robert Lipe for OpenServer support, new testsuites, testing, etc.
- Chen Liqin for various S+core related fixes/improvement, and for maintaining the S+core port.
- Weiwen Liu for testing and various bug fixes.
- Manuel Lopez-Ibañez for improving `-Wconversion' and many other diagnostics fixes and improvements.
- Dave Love for his ongoing work with the Fortran front end and runtime libraries.
- Martin von Lowis for internal consistency checking infrastructure, various C++ improvements including namespace support, and tons of assistance with libstdc++/compiler merges.
- H.J. Lu for his previous contributions to the steering committee, many x86 bug reports, prototype patches, and keeping the GNU/Linux ports working.
- Greg McGary for random fixes and (someday) bounded pointers.
- Andrew MacLeod for his ongoing work in building a real EH system, various code generation improvements, work on the global optimizer, etc.
- Vladimir Makarov for hacking some ugly i960 problems, PowerPC hacking improvements to compile-time performance, overall knowledge and direction in the area of instruction scheduling, and design and implementation of the automaton based instruction scheduler.
- Bob Manson for his behind the scenes work on dejagnu.
- Philip Martin for lots of libstdc++ string and vector iterator fixes and improvements, and string clean up and testsuites.
- All of the Mauve project contributors, for Java test code.
- Bryce McKinlay for numerous GCJ and libgcj fixes and improvements.

- Adam Megacz for his work on the Microsoft Windows port of GCJ.
- Michael Meissner for LRS framework, ia32, m32r, v850, m88k, MIPS, powerpc, haifa, ECOFF debug support, and other assorted hacking.
- Jason Merrill for his direction via the steering committee and leading the G++ effort.
- Martin Michlmayr for testing GCC on several architectures using the entire Debian archive.
- David Miller for his direction via the steering committee, lots of SPARC work, improvements in jump.c and interfacing with the Linux kernel developers.
- Gary Miller ported GCC to Charles River Data Systems machines.
- Alfred Minarik for libstdc++ string and ios bug fixes, and turning the entire libstdc++ testsuite namespace-compatible.
- Mark Mitchell for his direction via the steering committee, mountains of C++ work, load/store hoisting out of loops, alias analysis improvements, ISO C `restrict` support, and serving as release manager for GCC 3.x.
- Alan Modra for various GNU/Linux bits and testing.
- Toon Moene for his direction via the steering committee, Fortran maintenance, and his ongoing work to make us make Fortran run fast.
- Jason Molenda for major help in the care and feeding of all the services on the gcc.gnu.org (formerly egcs.cygnus.com) machine---mail, web services, ftp services, etc etc. Doing all this work on scrap paper and the backs of envelopes would have been... difficult.
- Catherine Moore for fixing various ugly problems we have sent her way, including the haifa bug which was killing the Alpha & PowerPC Linux kernels.
- Mike Moreton for his various Java patches.
- David Mosberger-Tang for various Alpha improvements, and for the initial IA-64 port.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits and for ISO C99 support.
- Bill Moyer for his behind the scenes work on various issues.
- Philippe De Muyter for his work on the m68k port.
- Joseph S. Myers for his work on the PDP-11 port, format checking and ISO C99 support, and continuous emphasis on (and contributions to) documentation.
- Nathan Myers for his work on libstdc++-v3: architecture and authorship through the first three snapshots, including implementation of locale infrastructure, string, shadow C headers, and the initial project documentation (DESIGN, CHECKLIST, and so forth). Later, more work on MT-safe string and shadow headers.
- Felix Natter for documentation on porting libstdc++.
- Nathanael Nerode for cleaning up the configuration/build process.
- NeXT, Inc. donated the front end that supports the Objective-C language.
- Hans-Peter Nilsson for the CRIS and MMIX ports, improvements to the search engine setup, various documentation fixes and other small fixes.
- Geoff Noer for his work on getting cygwin native builds working.

- Diego Novillo for his work on Tree SSA, OpenMP, SPEC performance tracking web pages, GIMPLE tuples, and assorted fixes.
- David O'Brien for the FreeBSD/alpha, FreeBSD/AMD x86-64, FreeBSD/ARM, FreeBSD/PowerPC, and FreeBSD/SPARC64 ports and related infrastructure improvements.
- Alexandre Oliva for various build infrastructure improvements, scripts and amazing testing work, including keeping libtool issues sane and happy.
- Stefan Olsson for work on mt_alloc.
- Melissa O'Neill for various NeXT fixes.
- Rainer Orth for random MIPS work, including improvements to GCC's o32 ABI support, improvements to dejagnu's MIPS support, Java configuration clean-ups and porting work, etc.
- Hartmut Penner for work on the s390 port.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Alexandre Petit-Bianco for implementing much of the Java compiler and continued Java maintainership.
- Matthias Pfaller for major improvements to the NS32k port.
- Gerald Pfeifer for his direction via the steering committee, pointing out lots of problems we need to solve, maintenance of the web pages, and taking care of documentation maintenance in general.
- Andrew Pinski for processing bug reports by the dozen.
- Ovidiu Predescu for his work on the Objective-C front end and runtime libraries.
- Jerry Quinn for major performance improvements in C++ formatted I/O.
- Ken Raeburn for various improvements to checker, MIPS ports and various cleanups in the compiler.
- Rolf W. Rasmussen for hacking on AWT.
- David Reese of Sun Microsystems contributed to the Solaris on PowerPC port.
- Volker Reichelt for keeping up with the problem reports.
- Joern Rennecke for maintaining the sh port, loop, regmove & reload hacking.
- Loren J. Rittle for improvements to libstdc++-v3 including the FreeBSD port, threading fixes, thread-related configury changes, critical threading documentation, and solutions to really tricky I/O problems, as well as keeping GCC properly working on FreeBSD and continuous testing.
- Craig Rodrigues for processing tons of bug reports.
- Ola Ronnerup for work on mt_alloc.
- Gavin Romig-Koch for lots of behind the scenes MIPS work.
- David Ronis inspired and encouraged Craig to rewrite the G77 documentation in texinfo format by contributing a first pass at a translation of the old `g77-0.5.16/f/DOC' file.
- Ken Rose for fixes to GCC's delay slot filling code.
- Paul Rubin wrote most of the preprocessor.
- Petur Runolfsson for major performance improvements in C++ formatted I/O and large file support in C++ filebuf.

- Chip Salzenberg for libstdc++ patches and improvements to locales, traits, Makefiles, libio, libtool hackery, and ``long long'' support.
- Juha Sarlin for improvements to the H8 code generator.
- Greg Satz assisted in making GCC work on HP-UX for the 9000 series 300.
- Roger Sayle for improvements to constant folding and GCC's RTL optimizers as well as for fixing numerous bugs.
- Bradley Schatz for his work on the GCJ FAQ.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- William Schelter did most of the work on the Intel 80386 support.
- Tobias Schluter for work on GNU Fortran.
- Bernd Schmidt for various code generation improvements and major work in the reload pass as well a serving as release manager for GCC 2.95.3.
- Peter Schmid for constant testing of libstdc++---especially application testing, going above and beyond what was requested for the release criteria---and libstdc++ header file tweaks.
- Jason Schroeder for jcf-dump patches.
- Andreas Schwab for his work on the m68k port.
- Lars Segerlund for work on GNU Fortran.
- Joel Sherrill for his direction via the steering committee, RTEMS contributions and RTEMS testing.
- Nathan Sidwell for many C++ fixes/improvements.
- Jeffrey Siegal for helping RMS with the original design of GCC, some code which handles the parse tree and RTL data structures, constant folding and help with the original VAX & m68k ports.
- Kenny Simpson for prompting libstdc++ fixes due to defect reports from the LWG (thereby keeping GCC in line with updates from the ISO).
- Franz Sirl for his ongoing work with making the PPC port stable for GNU/Linux.
- Andrey Slepuhin for assorted AIX hacking.
- Trevor Smigiel for contributing the SPU port.
- Christopher Smith did the port for Convex machines.
- Danny Smith for his major efforts on the Mingw (and Cygwin) ports.
- Randy Smith finished the Sun FPA support.
- Scott Snyder for queue, iterator, istream, and string fixes and libstdc++ testsuite entries. Also for providing the patch to G77 to add rudimentary support for `INTEGER*1`, `INTEGER*2`, and `LOGICAL*1`.
- Brad Spencer for contributions to the GLIBCPP_FORCE_NEW technique.
- Richard Stallman, for writing the original GCC and launching the GNU project.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Nigel Stephens for various mips16 related fixes/improvements.
- Jonathan Stone wrote the machine description for the Pyramid computer.

- Graham Stott for various infrastructure improvements.
- John Stracke for his Java HTTP protocol fixes.
- Mike Stump for his Elxsi port, G++ contributions over the years and more recently his vxworks contributions
- Jeff Sturm for Java porting help, bug fixes, and encouragement.
- Shigeya Suzuki for this fixes for the bsdi platforms.
- Ian Lance Taylor for his mips16 work, general configury hacking, fixincludes, etc.
- Holger Teutsch provided the support for the Clipper CPU.
- Gary Thomas for his ongoing work to make the PPC work for GNU/Linux.
- Philipp Thomas for random bug fixes throughout the compiler
- Jason Thorpe for thread support in libstdc++ on NetBSD.
- Kresten Krab Thorup wrote the run time support for the Objective-C language and the fantastic Java bytecode interpreter.
- Michael Tiemann for random bug fixes, the first instruction scheduler, initial C++ support, function integration, NS32k, SPARC and M88k machine description work, delay slot scheduling.
- Andreas Tobler for his work porting libgcj to Darwin.
- Teemu Torma for thread safe exception handling support.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the VAX machine description.
- Daniel Towner and Hariharan Sandanagobalane contributed and maintain the picoChip port.
- Tom Tromey for internationalization support and for his many Java contributions and libgcj maintainership.
- Lassi Tuura for improvements to config.guess to determine HP processor types.
- Petter Urkedal for libstdc++ CXXFLAGS, math, and algorithms fixes.
- Andy Vaught for the design and initial implementation of the GNU Fortran front end.
- Brent Verner for work with the libstdc++ cshadow files and their associated configure steps.
- Todd Vierling for contributions for NetBSD ports.
- Jonathan Wakely for contributing libstdc++ Doxygen notes and XHTML guidance.
- Dean Wakerley for converting the install documentation from HTML to texinfo in time for GCC 3.0.
- Krister Walfridsson for random bug fixes.
- Feng Wang for contributions to GNU Fortran.
- Stephen M. Webb for time and effort on making libstdc++ shadow files work with the tricky Solaris 8+ headers, and for pushing the build-time header tree.
- John Wehle for various improvements for the x86 code generator, related infrastructure improvements to help x86 code generation, value range propagation and other work, WE32k port.
- Ulrich Weigand for work on the s390 port.

- Zack Weinberg for major work on cpplib and various other bug fixes.
- Matt Welsh for help with Linux Threads support in GCJ.
- Urban Widmark for help fixing java.io.
- Mark Wielaard for new Java library code and his work integrating with Classpath.
- Dale Wiles helped port GCC to the Tahoe.
- Bob Wilson from Tensilica, Inc. for the Xtensa port.
- Jim Wilson for his direction via the steering committee, tackling hard problems in various places that nobody else wanted to work on, strength reduction and other loop optimizations.
- Paul Woegerer and Tal Agmon for the CRX port.
- Carlo Wood for various fixes.
- Tom Wood for work on the m88k port.
- Canqun Yang for work on GNU Fortran.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- Kevin Zachmann helped port GCC to the Tahoe.
- Ayal Zaks for Swing Modulo Scheduling (SMS).
- Xiaoqiang Zhang for work on GNU Fortran.
- Gilles Zunino for help porting Java to Irix.

The following people are recognized for their contributions to GNAT, the Ada front end of GCC:

- Bernard Banner
- Romain Berrendonner
- Geert Bosch
- Emmanuel Briot
- Joel Brobecker
- Ben Brosgol
- Vincent Celier
- Arnaud Charlet
- Chien Chieng
- Cyrille Comar
- Cyrille Crozes
- Robert Dewar
- Gary Dismukes
- Robert Duff
- Ed Falis
- Ramon Fernandez
- Sam Figueroa
- Vasiliy Fofanov

- Michael Friess
- Franco Gasperoni
- Ted Giering
- Matthew Gingell
- Laurent Guerby
- Jerome Guitton
- Olivier Hainque
- Jerome Hugues
- Hristian Kirtchev
- Jerome Lambourg
- Bruno Leclerc
- Albert Lee
- Sean McNeil
- Javier Miranda
- Laurent Nana
- Pascal Obry
- Dong-Ik Oh
- Laurent Pautet
- Brett Porter
- Thomas Quinot
- Nicolas Roche
- Pat Rogers
- Jose Ruiz
- Douglas Rupp
- Sergey Rybin
- Gail Schenker
- Ed Schonberg
- Nicolas Setton
- Samuel Tardieu

The following people are recognized for their contributions of new features, bug reports, testing and integration of classpath/libgcj for GCC version 4.1:

- Lillian Angel for `JTree` implementation and lots Free Swing additions and bug fixes.
- Wolfgang Baer for `GapContent` bug fixes.
- Anthony Balkissoon for `JList`, Free Swing 1.5 updates and mouse event fixes, lots of Free Swing work including `JTable` editing.
- Stuart Ballard for RMI constant fixes.
- Goffredo Baroncelli for `HTTPURLConnection` fixes.
- Gary Benson for `MessageFormat` fixes.
- Daniel Bonniot for `Serialization` fixes.

- Chris Burdess for lots of gnu.xml and http protocol fixes, StAX and DOM xml:id support.
- Ka-Hing Cheung for TreePath and TreeSelection fixes.
- Archie Cobbs for build fixes, VM interface updates, URLClassLoader updates.
- Kelley Cook for build fixes.
- Martin Cordova for Suggestions for better SocketTimeoutException.
- David Daney for BitSet bug fixes, HttpURLConnection rewrite and improvements.
- Thomas Fitzsimmons for lots of upgrades to the gtk+ AWT and Cairo 2D support. Lots of imageio framework additions, lots of AWT and Free Swing bug fixes.
- Jeroen Frijters for ClassLoader and nio cleanups, serialization fixes, better Proxy support, bug fixes and IKVM integration.
- Santiago Gala for AccessControlContext fixes.
- Nicolas Geoffray for VMClassLoader and AccessController improvements.
- David Gilbert for basic and metal icon and plaf support and lots of documenting, Lots of Free Swing and metal theme additions. MetalIconFactory implementation.
- Anthony Green for MIDI framework, ALSA and DSSI providers.
- Andrew Haley for Serialization and URLClassLoader fixes, gcj build speedups.
- Kim Ho for JFileChooser implementation.
- Andrew John Hughes for Locale and net fixes, URI RFC2986 updates, Serialization fixes, Properties XML support and generic branch work, VMIntegration guide update.
- Bastiaan Huisman for TimeZone bug fixing.
- Andreas Jaeger for mprec updates.
- Paul Jenner for better `-Werror' support.
- Ito Kazumitsu for NetworkInterface implementation and updates.
- Roman Kennke for BoxLayout, GrayFilter and SplitPane, plus bug fixes all over. Lots of Free Swing work including styled text.
- Simon Kitching for String cleanups and optimization suggestions.
- Michael Koch for configuration fixes, Locale updates, bug and build fixes.
- Guilhem Lavaux for configuration, thread and channel fixes and Kaffe integration. JCL native Pointer updates. Logger bug fixes.
- David Lichteblau for JCL support library global/local reference cleanups.
- Aaron Luchko for JDWP updates and documentation fixes.
- Ziga Mahkovec for Graphics2D upgraded to Cairo 0.5 and new regex features.
- Sven de Marothy for BMP imageio support, CSS and TextLayout fixes. GtkImage rewrite, 2D, awt, free swing and date/time fixes and implementing the Qt4 peers.
- Casey Marshall for crypto algorithm fixes, FileChannel lock, SystemLogger and FileHandler rotate implementations, NIO FileChannel.map support, security and policy updates.
- Bryce McKinlay for RMI work.
- Audrius Meskauskas for lots of Free Corba, RMI and HTML work plus testing and documenting.
- Kalle Olavi Niemitalo for build fixes.

- Rainer Orth for build fixes.
- Andrew Overholt for `File` locking fixes.
- Ingo Proetel for `Image`, `Logger` and `URLClassLoader` updates.
- Olga Rodimina for `MenuSelectionManager` implementation.
- Jan Roehrich for `BasicTreeUI` and `JTree` fixes.
- Julian Scheid for documentation updates and gjdoc support.
- Christian Schlichtherle for zip fixes and cleanups.
- Robert Schuster for documentation updates and beans fixes, `TreeNode` enumerations and `ActionCommand` and various fixes, XML and URL, AWT and Free Swing bug fixes.
- Keith Seitz for lots of JDWP work.
- Christian Thalinger for 64-bit cleanups, Configuration and VM interface fixes and `CACAO` integration, `fdlibm` updates.
- Gael Thomas for `VMClassLoader` boot packages support suggestions.
- Andreas Tobler for Darwin and Solaris testing and fixing, `Qt4` support for Darwin/OS X, `Graphics2D` support, `gtk+` updates.
- Dalibor Topic for better `DEBUG` support, build cleanups and Kaffe integration. `Qt4` build infrastructure, `SHA1PRNG` and `GdkPixbugDecoder` updates.
- Tom Tromey for Eclipse integration, generics work, lots of bug fixes and gcj integration including coordinating The Big Merge.
- Mark Wielaard for bug fixes, packaging and release management, `Clipboard` implementation, system call interrupts and network timeouts and `GdkPixpufDecoder` fixes.

In addition to the above, all of which also contributed time and energy in testing GCC, we would like to thank the following for their contributions to testing:

- Michael Abd-El-Malek
- Thomas Arend
- Bonzo Armstrong
- Steven Ashe
- Chris Baldwin
- David Billinghurst
- Jim Blandy
- Stephane Bortzmeyer
- Horst von Brand
- Frank Braun
- Rodney Brown
- Sidney Cadot
- Bradford Castalia
- Robert Clark
- Jonathan Corbet
- Ralph Doncaster
- Richard Emberson

- Levente Farkas
- Graham Fawcett
- Mark Fernyhough
- Robert A. French
- Jorgen Freyh
- Mark K. Gardner
- Charles-Antoine Gauthier
- Yung Shing Gene
- David Gilbert
- Simon Gornall
- Fred Gray
- John Griffin
- Patrik Hagglund
- Phil Hargett
- Amancio Hasty
- Takafumi Hayashi
- Bryan W. Headley
- Kevin B. Hendricks
- Joep Jansen
- Christian Joensson
- Michel Kern
- David Kidd
- Tobias Kuipers
- Anand Krishnaswamy
- A. O. V. Le Blanc
- llewelly
- Damon Love
- Brad Lucier
- Matthias Klose
- Martin Knoblauch
- Rick Lutowski
- Jesse Macnish
- Stefan Morrell
- Anon A. Mous
- Matthias Mueller
- Pekka Nikander
- Rick Niles
- Jon Olson
- Magnus Persson

- Chris Pollard
- Richard Polton
- Derk Reefman
- David Rees
- Paul Reilly
- Tom Reilly
- Torsten Rueger
- Danny Sadinoff
- Marc Schifer
- Erik Schnetter
- Wayne K. Schroll
- David Schuler
- Vin Shelton
- Tim Souder
- Adam Sulmicki
- Bill Thorson
- George Talbot
- Pedro A. M. Vazquez
- Gregory Warnes
- Ian Watson
- David E. Young
- And many others

And finally we'd like to thank everyone who uses the compiler, provides feedback and generally reminds us why we're doing this work in the first place.

选项索引

GCC's command line options are indexed here without any initial `-' or `--'. Where an option has both positive and negative forms (such as `-foption' and `-fno-option'), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

(Index is nonexistent)

概念索引

(Index is nonexistent)