

GCC 的中间语言及后端信息的转换

The Intermediate Language and the Back-End Information Translation in GCC

任珊虹 赵克佳 赵雄芳^①

Ren Shanhong, Zhao Kejia and Zhao Xiongfang

(国防科技大学计算机研究所)

(Computer Research Institute, National University of Defense Tech.)

摘要 GCC 是一个成功地支持多种高级语言和多种机器平台的系统,分析其中间语言 RTL 是扩充其前端和移植 GCC 的前提。GCC 的后端信息 MD 使用 RTL 的外部形式书写,主要用于定义机器的指令集、功能部件、延迟特性等。GCC 提供一套 MD 的转换程序 gen* 来生成一套 C 文件 insn-*, 供编译体更有效地获取与机器有关的信息;可以把 gen* 看作是 MD 的编译器。

ABSTRACT GCC is a successful system which supports a few high level languages as well as many machines. Before adding something to its front end or making it run on some other system, analysis of its intermediate language RTL becomes a prerequisite. The back end information MDs in GCC are written in textual form of RTL. MDs are used to define instruction sets, function units, delay slots, etc. GCC uses a set of programs gen* to produce a set of C files insn-*, which give the machine information to the compiler in GCC directly. Gen* can be considered as the compiler of MD.

关键词 RTL (Register Transfer Language), RTX (RTL eXpression), MD (Machine Description), MD 转换。

KEY WORDS RTL, RTX, MD, Translation of MD.

一、GCC 的中间语言 RRL

1. 简介 GCC

GCC (GNU CC Compiler) 是一个成功的支持多种高级语言和多种机器平台的系统,它的前端支持 C、C++、Objective C 等高级语言(据 GCCV2.6.0 称,对 ADA、PASCAL

^① 收稿日期:1995 年 2 月。

作者地址:410073 湖南长沙国防科技大学计算机研究所

作者简介:任珊虹,硕士研究生,研究方向是软件工程 赵克佳,副教授,负责 GCC 移植开发工作。

赵雄芳副教授,多年从事编辑系统的研究和开发。

等语言的支持正在开发中), 后端支持 SUN、SGI、SONY、MIPS、IBM、HP、DEC、APOLLO、MOTOROLA 等 34 家公司的 25 种 CPU 类型 (如 alpha、i386、m68000、mips、sh、sparc、vax) 上的 26 个系统 (如 bsd、isc、genix、sco、sunos、sysV、ultrix、unos、vms) 的不同版本。其简单的工作流程见图 1。

与语言无关和与机器无关的编译的大部分工作, 是在其中间语言 RTL (Register Transfer Language) 上完成的。

2. RTL 的作用

由 GCC 的工作流程图可以看到, 多种语言的语法分析器流向同一个语义分析器。这种一对多的映射由 RTL 充当接口: 各语法分析器输出的语法树均被转换成统一的 RTL 表示, 接下来的寄存器分配、优化、指令调度等工作均在 RTL 上完成。从 RTL 生成步以后的所有编译工作可以说是都脱离了语言特征。换句话说, GCC 使多种语言在编译原理的实现上得到了统一, 共享了既成的最优的编译方法。GCC 不追求前端语法上的统一, 而是预处理多种语言: 将它们映射成同等语义的 GCC 标准语言 RTL。RTL 本身并不便于阅读和书写, 但它是一种理想的内部语言。若要使 GCC 支持一种新的高级语言, 只要编写它的预处理程序。因此研究 RTL 是扩充 GCC 前端的前提之一。

RTL 还充当 GCC 与多种机器平台的接口: 所有的机器描述均由 RTL (的外部形式) 书写, 以便文件生成器 “gen *” 读取关于机器平台的信息。机器描述包括: 指令延迟特性的定义, 各功能部件的定义, 机器指令集的定义 (对同一操作不同机器模式的指令, 分别定义), 还包括关于本机器描述中各属性的定义。这样, 使用统一的数据结构名和宏操作名。使 GCC 对不同的机器平台生成不同的汇编代码。可见, 对 RTL 的分析也是移植 GCC 的前提之一。只要写出正确的机器描述文件, 可以很容易完成交叉编译。

GCC 使用 RTL, 体现了当今流行的软件设计和实现的思想: 将可重用部分独立出来, 通过接口实现模块间一对多映射。这样做的好处是: 提高软件 (尤其是大型系统软件) 生产的质和量, 增强系统的可维性、可扩性、可移植性和可靠性。

3. RTL 的定义

RTL 的语法借鉴 LISP 语言, 使表 (list) 成为其唯一的逻辑结构 (称之为 RTX: RTL eXpression)。RTL 有自己的语法和语义, 对 RTX 作了定义: 限制 RTX 的头元素 (称之为 RTX 代码) 的取值, 每一种 RTX 代码的取值都有特定的含义; 且 RTX 代码决定其尾元素 (称之为该代码的操作数或域) 的个数和类型。

RTL 有两种表示: 外部表示, 是类 LIST 的表文本的括号嵌套, 用于机器描述和卸出调试信息; 内部表示, 是 C 结构的指针链接, 用于编译的内部操作。两种表示可以相互转换。

RTL 使用五种基本对象: 整数、宽整数、字符串、向量和 RTX。加上机器模式 (描述对象在机器一级的大小和格式, 如 SImode 表示四字节整型), 就是 RTX 的构成元素。RTL 的元素可以是 RTX, 向量的元素也是 RTX, 这两种情况导致 RTX 嵌套。

RTL 代码决定 RTX 的类型。GCC 的 RTL 定义了九类 RTX, 每一类用于不同的目的。这九类 RTX 是:

第一类 用于构造 RTX 的元表达式 (如 “nil”)

- 第二类 用于构造表 (如 “insn_list”)
- 第三类 用于机器描述 (如 “define_insn”)
- 第四类 用于 INSN 属性 (如 “define_attr”)
- 第五类 用于指令链的 INSN (如 “insn”)
- 第六类 用于 INSN 的顶层构造 (如 “parallel”)
- 第七类 描述机器指令的行为 (如 “set”)
- 第八类 用于表达式的原始值 (如 “const_int”)
- 第九类 描述操作的结果 (如 “plus”)

[GCC编译流程简图]

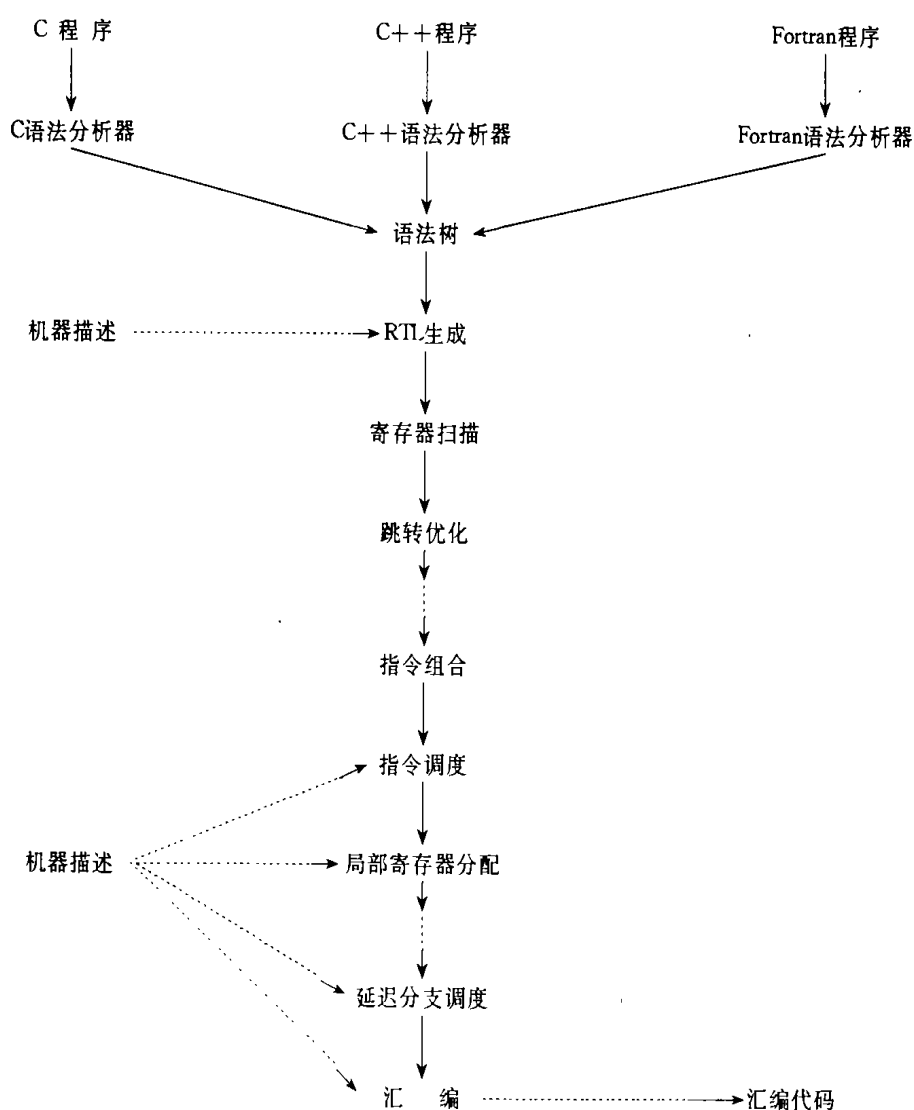


图1 GCC 工作流程

其中，第三类、第四类只用于机器描述；第五类 INSN 是编译所使用的最外层 RTX 单元，即编译的处理对象是一个 INSN 序列；第六类或第七类做 INSN 或机器描述的操作数，表示对机器状态产生“副作用”（sideeffect）的行为，有时称它们为 INSN 的体或样

板 (body or pattern); 第八类或第九类做第六类或第七类的操作数, 表示值而非行为; 第一类和第二类用做其它某些代码的操作数。

另外, 还有四个内部“副作用”表达式代码, 其作用是为寄存器在栈中定址, 不出现现在指令模板中。

每个 RTX 代码均有自己的格式: 操作数的个数和它们的类别, 如代码“insn”被定义为:

```
DEF_RTL_EXPR (INSN, "insn", "iuueiee", 'i')
```

其含义为: 该 RTX 代码的内部名是 INSN (枚举型), 外部名是“insn”, 有 7 个操作数 (域), 它们的类别依次为 iuueiee (i: 整型, u: 指向另一个 INSN 的指针, e: RTX 的指针), 该 RTX 表示一条机器指令 (‘i’)。

一个“insn”的实例是:

```
(insn 32 30 34 (set (reg: SI 75)
  (plus: SI (reg: SI 74)
    (const_int 1))) -1 (nil)
  (nil))
```

含义是: 本条 INSN 的代码为“insn”, 在 INSN 序列中的编号为 32; 其上一条 INSN 的编号为 30, 下一条 INSN 的编号为 34。其行为是: 将编号为 74 的寄存器 (伪寄存器) 的内容与整常数 1 相加, 再将结果赋给编号为 75 的寄存器 (伪寄存器), 操作都是针对机器模式 SI mode 的。

二、GCC 的后端信息 MD

1. MD 的作用和构造

GCC 使用 RTL 表示两类信息: INSN 和 MD。INSN 序列是编译体的处理对象。INSN 表示一条指令, 通常对应机器的一个动作或一个动作序列, 可能对机器状态产生影响。编译过程的简图如下:

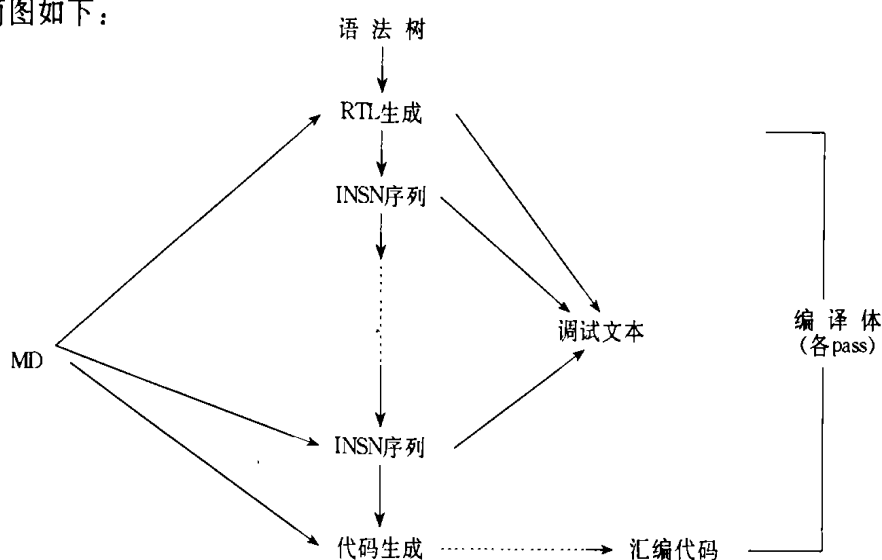


图 2 GCC 编译过程

机器描述 MD (Machine Description) 是 GCC 的后端信息。MD 的作用是:

- 1) 在 RTL 生成过程中, 提供 INSN 模板
- 2) 在代码生成过程中, 提供汇编模板
- 3) 为 INSN 提供机器特性信息, 指导指令调度等编译后期的工作

MD 包括四部分:

- 1) 描述文件 (名为 “md”)

用 RTL 书写指令集、功能部件、指令属性、指令延迟特性等信息

- 2) 机器宏文件 (名为 “tm. h”)

定义 “字节”、“字”、“字符”、“宽字”、“寄存器集” 等硬件信息

- 3) 系统宏文件 (名为 “config. h”)

说明特有函数别名、数据模式限制、是否有某些特殊的系统文件等信息

- 4) 代码输出的辅助文件 (“aux-output. c”)

GCC 支持的每类平台均有一套 MD, 安装 GCC (configure) 时, 被定义的宿主机的 MD 被逻辑链接成这四个文件名。要移植 GCC, 只需书写相应平台的 MD。

“md” 文件是 MD 的核心, 它是一条条描述构成的序列, 以下的 “MD” 特指 “md” 中的一条描述。所以, 可以说 “md” 是 MD 的序列。

2. MD 的定义

MD 采用 RTX 的外部表示来书写, 即括号嵌套的表。MD 的代码及其格式如下:

```
(define_insn 名字 RTL 模板 条件 输出模板 属性)
(define_expand 名字 RTL 模板 条件 准备语句)
(define_split RTL 模板 条件 [RTL 模板-1 RTL 模板-2...] 准备语句)
(define_peephole [RTL 模板-1 RTL 模板-2...] 条件 输出模板 属性)
(define_asm_attributes 属性)
(define_delay 测试[延迟槽测试-1 分支真删除测试-1 分支假删除测试-1
                延迟槽测试-2 分支真删除测试-2 分支假删除测试-2
                .....
                ])
(define_function_unit 名字 部件数 可并行指令数 测试
                    结果到达时间 流出起步时间 冲突表)
(define_attr 名字 值表 缺省值)
(define_combine [...] “” “”) (未定义)
```

MD 中 RTL 模板的内部还有一些特定于机器描述的代码:

```
(match_operand: M 操作数号 谓词 约束)
(match_scratch: M 操作数号 约束)
(match_operator: M 操作数号 谓词 [操作数-1 操作数-2...])
(match_parallel: M 操作数号 [RTL-1 RTL-2])
(match_dup: M 操作数号)
(match_op_dup: M 操作数号 [操作数-1 操作数-2...])
```

(match_paw_dup: M 操作数号 [操作数-1 操作数-2...])

(address RTX)

(eq_attr 名字 值)

(attr_flag 名字)

(attr 名字)

(set_attr 名字 属性)

(set_attr_alternative 名字 值表)

MD 必须定义 GCC 的所有标准名 (define_insn), 还可以根据本平台特性定义辅助类 MD (有名或无名 define_insn、define_split、define_expand、define_peephole) 和内部 MD (以标准名为前缀); 各标准名对应一个 INSN 模板和平台的一条或若干条机器指令。

机器的每个功能部件都对应若干条 MD (define_function_unit)。同一功能部件的多个 MD 主要用于定义不同指令或不同 CPU 所使用的不同的结果到达时间 (ready_delay) 和流出起步时间 (issue_delay)。

MD 还要定义指令集涉及的所有类型的延迟槽 (define_delay)。

每类平台都有自己的属性集, 其中每个属性都对应一个 MD (define_attr), 定义该属性的值集和缺省值。gen * 通过比较各 MD 对属性集的取值, 来确定它们之间的关系。

MD 的顺序一般无关紧要, 但某些平台对能同时被匹配的多个 MD 有顺序要求, 一般内涵更丰富的 MD 优先 (与 ES 的冲突解决策略类似)。

MD 表达了 GCC 标准动作集→同等语义 INSN 和 GCC 标准动作集→同等语义汇编代码的映射。

define_insn 类的 MD 是“md”的核心, 其它类 MD 都围绕着它来定义。以下就以 define_insn 为例介绍 MD。

(define_insn 名字 RTL 模板 条件 输出模板 属性)

名字 字符串, 可为空。前缀必须是 GCC 的 RTL 定义的标准名集中的元素。每个标准名都指导 GCC 执行一个标准动作, 如“addsi3”表明将操作数 2 与操作数 1 相加送入操作数 0 (三操作数指令), 模式为 SImode (单整型; 4bytes, “byte”表示多少位在“tm.h”中定义)。

RTL 模板 向量, 其元素是 RTX。若只有一个元素, 则对应于 INSN 的体: 在 RTL 生成时, 将操作数填入被匹配的 RTL 模板的拷贝中, 即构造了 INSN 的体; 否则表示一个“parallel”表达式。

条件 检查该 MD 可否被本编译器使用。

输出模板 字符串。表示匹配该 MD 中 RTL 模板的 INSN 应怎样输出代码: 将填入到 RTL 模板中的操作数相应地填入到这个字符串中特定字符 (“%操作号”) 标定的位置上, 即构造出汇编代码。

属性 向量, 其元素是 RTX。对属性集 (每个“md”都有自己的属性集) 的某些元素赋值, 不被赋值的属性取缺省值。属性用于 MD 之间的关系判断。

define_insn 类 MD 的一个例子如下 (“mips.md”中一个 MD):

(define_insn “addsf3”

```

[ (set (match_operand: SF 0 "register_operand" "=f")
      (plus: SF (match_operand: SF 1 "register_operand" "f")
                (match_operand: SF 2 "register_operand" "f"))))
  "TARGET_HARD_FLOAT"
  "add. s\t%0,%1,%2"
  [(set_attr "type" "fadd")
   (set_attr "mode" "SF")
   (set_attr "length" "1")]]

```

该 MD 匹配或构造的 INSN 所具有的体（样板）的模板如下：

```

(set 操作数 0
  (plus: SF 操作数 1
    操作数 2)

```

而它将给匹配该 RTL 模板的 INSN 提供如下汇编输出模板：

```
add. s    操作数 0  操作数 1  操作数 2
```

其中，各操作数是要被填入的类型恰当的 RTX（寄存器或存储器）；机器模式是 SF-mode。

可见，MD 在 RTL 生成、汇编代码生成过程中起着匹配和构造的作用。MD 是对编译体信息（INSN）进行前处理和后处理的重要内容。

三、GCC 后端信息的转换

1. GCC 转换其后端信息的必要性

编译体实际操作的 INSN 的内部表示，即 C 的指针链接结构；而 MD 是 RTX 的外部表示，即表文本。GCC 提供一套 MD 的预处理程序 gen*，来生成相应的一套 insn-* 文件，编译体从 insn-* 中读取 MD 信息。但 gen* 的作用决不仅仅是 RTX 形式上的转换（实际上，公用文件“rtl.c”中的 read_rtx() 函数可以很容易地实现 RTX 外部表示到内部表示的转换）。

gen*（11 个 C 程序）对 MD 作语义和语法的分析，抽取和收集各 MD 的特征信息，转换并加工成 C 的函数、数组、类型的定义或说明，写入 insn-*，即：

```

      读      生成      读
MD  →  gen*  ⇒ insn-* → 编译体

```

GCC 的 makefile 文件中有“gen* md > insn-*”的命令行，所以 GCC 安装(configure) 完毕以后，编译体直接调用 insn-*，而脱离了“md”文件。

如果把 MD 看作一个小的机器描述语言，那么 gen* 就是它的编译器，其输出是 C 文件 insn-*。也可以把 gen* 叫作 insn-* 的文件生成器。对 GCC 而言，gen* 只有一套，因读入的“md”不同而生成内容不同的 insn-*，即每个平台都被生成自己的一套 insn-*。

insn-*（11 个 C 文件）所包含的内容是“md”文件的信息被加工后的结果：MD

的构成元素被分析和整理,写入数组;MD 之间的依赖关系被判断和选择,写成函数;MD 所含的特征信息被统计、计算和比较,定义成宏;与 MD 有关的函数说明、结构说明和结构定义,被填入适当的位置。

为了分析清楚 GCC 后端如何为编译体提供信息,就要涉及三个方面:

1) MD 的内容

读 生成 读
2) MD \rightarrow gen * \Rightarrow insn - * \rightarrow 编译体的过程

3) insn - * 的内容及其何处、何时被使用

对前端或编译体的开发者而言,前两个方面可以是透明的;而后端的开发者就要全面而深入地了解这三方面内容,以修改或重写出正确的目标平台 MD。

2. gen * 对 MD 的编译

① gen * 的一些共同的特点:

1) 每个 gen * 程序均是针对某些类的 MD,来生成用于不同编译 pass 的 insn - * 文件。

2) 顺序读 MD,有两种处理方式:

(1) 边读边处理

(2) 边读边归类,再分别处理

3) 使用 insn_code,它是顺序对 define_insn、define_split、define_expand、define_peephole 四类 MD 的唯一的递增的编号;index_code,它是顺序对所有 MD 的唯一的递增的编号。

insn_code 用来做 insn - * 中的各数据结构的下标 (key),和各函数分支语句 (switch) 的判断变元 (决定转向哪个 case)。index_code 用于错误信息的输出,即若某 MD 有书写错误 (语法或语义错误),则输出它的 index_code。

注意:在 genattrtab. c \Rightarrow insn_attrtab. c 的过程中,define_asm_attributes 不参与 index_code 的统计。

4) gen * 符合 RTL 的共享约定:任何 RTX 均是唯一的,即对相同结构和内容 RTX 的引用,可以用不同指针,但它们都指向同一 RTX 实体。这样做有助于提高编译速度:判断两个 RTX 是否相同,只要比较它们的指针。

5) gen * 最大限度地减少其输出 insn - * 的信息冗长:不重写相同的判断和输出,而是通过 case 或 goto 共享 (这也是 RTX 结构唯一性的表现)。

这样做使 gen * 的代码量增加不少,信息转换速度也有所降低,但对 MD 的编译是在 makefile 中完成,所以合理。

这样做也使 insn - * 函数结构复杂 (尤以 insn-recog. c 为甚:goto 语句使它简直难以阅读),但其含义依赖于 gen * 和 MD,对后端开发者来说,它们只是 gen * 的优化过的输出结果;而前端开发者或编译体开发者只关心它们功能、输入和输出的说明。

另外,在 switch 语句中 (大多数 insn - * 函数的核心语句都是 switch 语句),将相同取值最多的情况作为 default 来输出。

6) gen * 避免生成过大的 insn - * 模块,使用自定的限制判断函数,将一个过大的

C 模块分成大小适中的子模块。而 `insn - *` 仍对外只提供原模块名，子模块之间的调用关系是透明的。

7) `gen *` 还采用很多优良的算法：如 hash 查寻；使用条件或树；简化常表达式；优化数据结构等。

`gen *` 本身就是一个书写优美的 (MD 的) 编译器。

②一个例子：`genoutput. c` \Rightarrow `insn-putput. c`

`genoutput. c` 是针对 `define_insn`、`define_split`、`define_expand`、`define_peephole` 类 MD 来编译的。它抽取 MD 的各操作数 (域)，统计其中的某些信息，分析某些域的特点，生成函数和数组，写入 `insn-output. c`。

`genoutput. c` 生成了若干数组，都以 `insn_code` 为下标，存放相应的 MD 的信息。如 `insn_name []` 存放各 MD 的名字，`insn_template []` 存放各 MD 的汇编输出模板，`insn_n_operands []` 存放各 MD 中 `match_operand` 的个数，等。

有些 MD 的输出模板是以 ‘*’ 或 ‘@’ 开头的字串，需要作相应处理后才表示汇编模板。`genoutput. c` 对它们分别生成函数 `output_N`，N 表示 `insn_code`。

③. `insn - *` 文件

`insn - *` 用于编译的各 pass，有的作为某个 pass 的重要输入内容，有的作为各 pass 的公用 include 文件。

RTL 生成、汇编代码生成以及其它涉及到后端信息的地方，都引用这一套 GCC 内部文件中的数组、函数、类型的定义或说明。

实际上，`insn - *` 的内容完全可以手工书写。但是编译体所使用的信息太分散了，关于一个 MD 的信息要写在几十个数组或函数中，若真的手写，是一件烦杂、易错且工程量很大的工作。与 `insn - *` 比较，MD 将后端信息集中、抽象，使一条 MD 的所有信息有机地表达在同一结构中，其内在联系使 MD 的书写和阅读都较容易；而 MD 之间的关系通过它们的属型关系和共有的某种特征来反映。`gen *` 对 MD 所富含的语义进行“解码”，抽取不同编译 pass 所关心的内容，以 C 程序最容易读取的方式集中写入不同的信息体，同时“显化”MD 之间的联系，预先“替”编译体做了比较、判断、选择的工作。