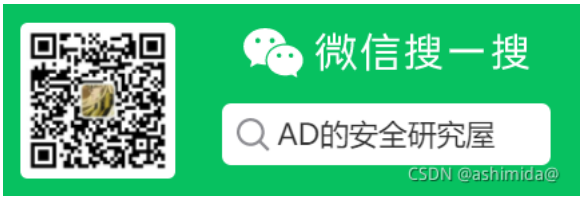
GCC源码分析(十四) — rtx结构体,指令与栈分配

更多内容可关注微信公众号



一、rtx表达式

RTL(Register Transfer Language)全称叫寄存器传输语言,其采用类似LISP语言的列表形式描述了每一条指令的语义动作. RTL语言中最基础的数据结构体就是RTX(struct rtx_def)，类似于树节点(tree)在AST中的作用,RTX(rtx_def)表达式节点是RTL语言中最基本的数据类型(RTX 是 RTL eXpression).

在gcc中,rtx结构体实际上是一个指向rtx_def结构体的指针,其类似AST中树节点(tree)是用来指向tree_node的指针一样,其定义如下:

```
typedef struct rtx_def *rtx;
```

和树节点(tree)类似,rtx节点也是有一个code字段的,所有的rtx节点都是通过不同的code来区分的,其全部定义在rtl.def中,如:

```
## ./gcc/rtl.def
## DEF_RTL_EXPR(RTL_CODE,NAME,PRINT_FORMAT,RTX_CLASS)
DEF_RTL_EXPR(UNKNOWN, "UnKnown", "*", RTX_EXTRA)
DEF_RTL_EXPR(VALUE, "value", "0", RTX_OBJ)
DEF_RTL_EXPR(EXPR_LIST, "expr_list", "ee", RTX_EXTRA)
DEF_RTL_EXPR(INSN_LIST, "insn_list", "ue", RTX_EXTRA)
DEF_RTL_EXPR(INT_LIST, "int_list", "ie", RTX_EXTRA)
......
```

此定义中包含四部分内容,解析后则分别分配到了rtx_code, rtx_name,rtx_format,rtx_class 4个枚举类型之中:

```
enum rtx_code    /* rtx_code 是个枚举类型,记录rtx中不同表达式的代码号 */
{
  NKNOWN,
  ALUE,
  EBUG_EXPR,
  XPR_LIST,
  NSN_LIST,
  ......
}
static const char *const rtx_name[NUM_RTX_CODE] = {    /* rtx_name 记录此rtx表达式输出名 */
  "UnKnown",
  "value",
  "debug_expr",
  "expr_list",
  "insn_list",
  "int_list",
  ......
}
static const char *const rtx_format[NUM_RTX_CODE] = {  /* rtx_format 记录此rtx表达式的输出格式,以及参数个数, 格式的含义见rtx_format定义 */
  "*",
  "0",
  "0",
  "ee",
  "ue",
  ......
}

enum rtx_class            /* rtx_class 是个枚举类型,记录rtx_code中这些代码的分配 */
    RTX_COMPARE,
    RTX_COMM_COMPARE,
    RTX_BIN_ARITH,
    RTX_COMM_ARITH,
    RTX_UNARY,
    RTX_EXTRA,
    RTX_MATCH,
    RTX_INSN,
    RTX_OBJ,
    RTX_CONST_OBJ,
    RTX_TERNARY,
    RTX_BITFIELD_OPS,
    RTX_AUTOINC
}
```

其中rtx_format中输出的格式归纳起来有5种,如:

- 'i'代表输出整数操作数
- 'w'代表输出宽整数操作数
- 'e' 代表输出rtx表达式
- 'E' 代表输出rtx向量
- 's' 代表输出字符串

类似AST中的树节点(tree),在rtl中最基本的节点类型是一个 **struct rtx_def**结构体,其指针类型为 **rtx,定义如下**:

```
typedef struct rtx_def *rtx;
struct rtx_def {
  ENUM_BITFIELD(rtx_code) code: 16;            /* 当前rtx表达式的code */
  ENUM_BITFIELD(machine_mode) mode : 8;        /* 此表达式对应的机器模式(决定占用空间大小的) */
  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned frame_related : 1;
  unsigned return_val : 1;

  union {
    unsigned int original_regno;
    int insn_uid;
    unsigned int symbol_ref_flags;
    enum var_init_status var_location_status;
    unsigned int num_elem;
    struct
    {
      unsigned int npatterns : 16;
      unsigned int nelts_per_pattern : 8;
      unsigned int unused : 8;
    } const_vector;
  }  u2;              //-------------------------------------------- 此前为rtx_def 结构体的头部

  union u {           //-------------------------------------------- 此后为rtx_def 结构体的操作数部分(可变长)m其以数组的形式保存当前rtx指令中所有操
    rtunion fld[1];     /* 最长用的是此字段,保存操作数信息 */
    HOST_WIDE_INT hwint[1];
    struct reg_info reg;
    struct block_symbol block_sym;
    struct real_value rv;
    struct fixed_value fv;
    struct hwivec_def hwiv;
    struct const_poly_int_def cpi;
  }  u;
}
```

整个rtx_def结构体可以分为两部分:

- 在结构体u之前的部分称为rtx_def头部(RTX Header),所有RTX指令的头部长度都是相同的,可通过 RTX_HDR_SIZE来获取其大小.
- 结构体u实际上是rtx_def中第一个操作数, 这段空间的大小则与当前rtx_def的具体指令码和操作数个数有关(如两个操作数实际分配的空间可能是fld[2])

rtx_code_size数组中记录了每个rtx表达式占用的空间大小,其定义如下:

```
// 根据不同的 rtx_code 和操作数个数(rtx_format中元素个数,如sizeof("ee")就是2个), rtx_code_size数组中记录每一种rtx表达式占用的空间大小
const unsigned char rtx_code_size[NUM_RTX_CODE] = {
#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS)                \
  (RTX_CODE_HWINT_P (ENUM)                         \
   ? RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (HOST_WIDE_INT)    \
   : (ENUM) == REG                                 \
   ? RTX_HDR_SIZE + sizeof (reg_info)                    \
   : RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (rtunion)),

#include "rtl.def"
#undef DEF_RTL_EXPR
};
```

在gcc中则通过以下宏可访问rtx_def中的操作数:

```
/* 获取RTX的第N个操作数,返回一个rt_int类型的值, RTL_CHECK2同时根据RTX->code检查此RTX表达式的第N个操作数是否应该是个rt_int类型 */
#define XINT(RTX, N)    (RTL_CHECK2 (RTX, N, 'i', 'n').rt_int)
#define XUINT(RTX, N)   (RTL_CHECK2 (RTX, N, 'i', 'n').rt_uint)    /* 获取RTX的第N个操作数,返回一个 rt_uint类型的值, 以下同理 */
#define XSTR(RTX, N)    (RTL_CHECK2 (RTX, N, 's', 'S').rt_str)
#define XEXP(RTX, N)    (RTL_CHECK2 (RTX, N, 'e', 'u').rt_rtx)
#define XVEC(RTX, N)    (RTL_CHECK2 (RTX, N, 'E', 'V').rt_rtvec)
#define XMODE(RTX, N)    (RTL_CHECK1 (RTX, N, 'M').rt_type)
#define XTREE(RTX, N)   (RTL_CHECK1 (RTX, N, 't').rt_tree)
#define XBBDEF(RTX, N)    (RTL_CHECK1 (RTX, N, 'B').rt_bb)
#define XTMPL(RTX, N)    (RTL_CHECK1 (RTX, N, 'T').rt_str)
#define XCFI(RTX, N)    (RTL_CHECK1 (RTX, N, 'C').rt_cfi)
```

二、rtx中的machine mode

rtx中所有的machine mode 最终可通过 enum machine_mode结构体查看,如下:

```
./build/gcc/insn-modes.h
enum machine_mode
{
  E_VOIDmode,              /* machmode.def:189 */
#define HAVE_VOIDmode
#ifdef USE_ENUM_MODES
#define VOIDmode E_VOIDmode
#else
#define VOIDmode ((void) 0, E_VOIDmode)
#endif
  E_BLKmode,               /* machmode.def:193 */
#define HAVE_BLKmode
#ifdef USE_ENUM_MODES
#define BLKmode E_BLKmode
#else
#define BLKmode ((void) 0, E_BLKmode)
#endif
  E_CCmode,                /* machmode.def:231 */
  ......
}
```

但需要注意的是,这个结构体是在编译过程中生成的, 其本身是来自于:

- ./gcc/machmode.def: 这里面记录了gcc中默认支持的所有机器mode
- ./gcc/config/aarch64-modes.def(以aarch64平台为例): 这里面记录了平台相关的机器mode

二者在编译器期动态的生成了 ./build/gcc/insn-modes.h 中的 enum machine_mode结构体,作为此目标平台最终的machine_mode

三、基于RTX的派生类

rtx_def是rtl中RTX指令的基类,但此类还存在一些派生类，rtx_def所有的派生类关系如图所示:

```
## 见 gcc/coretypes.h
struct rtx_def
  => class rtx_expr_list : public rtx_def
  => class rtx_insn_list : public rtx_def
  => class rtx_sequence : public rtx_def
  => class rtx_insn : public rtx_def
    => class rtx_note : public rtx_insn
    => class rtx_debug_insn : public rtx_insn
    => class rtx_nonjump_insn : public rtx_insn
    => class rtx_jump_insn : public rtx_insn
    => class rtx_call_insn : public rtx_insn
    => class rtx_jump_table_data : public rtx_insn
    => class rtx_barrier : public rtx_insn
    => class rtx_code_label : public rtx_insn
```

这些**所有的派生类中都没有在rtx_def的基础上增加成员变量,而只是增加了一些成员函数**；其各自独有的数据结构都存在了rtx_def的N个操作数字段中了.

四、rtx指令

rtx指令是rtx派生类中的一种,所有rtx指令的基类均为rtx_insn:

```
class  rtx_insn : public rtx_def {
public:
  bool deleted () const { return volatil; }
  void set_deleted () { volatil = true; }
  void set_undeleted () { volatil = false; }
};
```

可见**rtx指令(rtx_insn)实际上只是rtx节点的一个子类,其与rtx节点的区别在于**,rtx_insn以及其子类:

- 都具有prev/next指针可用来链接其他指令(rtx_insn.u.fld[0]/[1]分别为prev/next指针,对应访问宏为PREV_INSN/NEXT_INSN)
- 都具有指针记录此指令所属基本块(rtx_insn.u.fld[2]记录当前insn所在基本块,对应访问宏为 BLOCK_FOR_INSN)
- 都具有指针记录此指令的指令模板(rtx_insn.u.fld[3],对应宏PATTERN)
- 都具有指针记录此指令对应的源码位置(rtx_insn.u.fld[4],对应宏INSN_LOCATION)
- 都具有指针记录此指令对应的指令模板索引(rtx_insn.u.fld[5],对应宏 INSN_CODE)

rtx_insn派生自 rtx_def,但实际上二者fld字段的大小是不同的,这是在每次rtx结构体分配时根据其具体code来决定的,在gcc中rtx_insn以及其派生类包括:

```
class rtx_insn : public rtx_def
class rtx_debug_insn : public rtx_insn
```

```
class rtx_nonjump_insn : public rtx_insn    //指令码为INSN
class rtx_jump_insn : public rtx_insn
class rtx_call_insn : public rtx_insn
class rtx_jump_table_data : public rtx_insn
class rtx_barrier : public rtx_insn
class rtx_code_label : public rtx_insn
class rtx_note : public rtx_insn
```

而这些类的结构体大小记录在rtl.def中:

```
//结构体大小取决于节点的FORMAT字段, format字符串中有几个字符,则其fld数组中就有几个元素
//#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS) ...
DEF_RTL_EXPR(DEBUG_INSN, "debug_insn", "uuBeiie", RTX_INSN)
DEF_RTL_EXPR(INSN, "insn", "uuBeiie", RTX_INSN)
DEF_RTL_EXPR(JUMP_INSN, "jump_insn", "uuBeiie0", RTX_INSN)
DEF_RTL_EXPR(CALL_INSN, "call_insn", "uuBeiiee", RTX_INSN)
DEF_RTL_EXPR(JUMP_TABLE_DATA, "jump_table_data", "uuBe0000", RTX_INSN)
....
```

也就是说,**在gcc中一共只有8种rtx节点可以作为rtx指令**,其分别是INSN,DEBUG_INSN,JUMP_INSN,CALL_INSN,JUMP_TABLE_DATA,BARRIER,CODE_LABEL,NOTE, **任何函数的rtl指令序列就是由这些类的节点链接而成的,其他类型的rtx节点则只能作为这些指令节点的操作数(子节点出现)**

五、rtx_code中节点的含义

前面提到rtx节点根据code不同代表的含义不同,rtx节点所有的可用code记录在rtx_code结构体中, 这里介绍一些常用的code对应的rtx节点的含义:

1. 8个rtx指令节点:

1) **DEBUG_INSN:** 代表这是一条调试指令,通常来自GIMPLE_DEBUG的转换

2) **INSN:** 代表这是一条非跳转和非函数调用的指令, 如一个加法节点(PLUS)就会被包裹在一个INSN指令中(PLUS不是指令不能单独出现在指令序列中,必须通过一条指令包裹),如:

```
(insn (set (mem/c:SI (plus:DI (reg/f:DI 85 virtual-stack-vars) (const_int -4 [0xfffffffffffffffc])) (reg:SI 0 x0 [ x0 ])))
```

3) **JUMP_INSN:** 代表一条可能导致跳转的指令,其可能是条件跳转指令,也可能是无条件跳转指令

4) **CALL_INSN:** 代表一个函数调用指令,其中通常包裹一个CALL节点来代表被调用函数,而CALL_INSN还会记录此函数导致的额外副作用(如对硬件寄存器的修改)

5) **JUMP_TABL_DATA:** ...

6) **BARRIER:** 代表这是一个不可达指令, 若代码中出现了此指令则代表控制流在BARRIER的上一条指令不应该fallthru到当前指令,fallthru到当前指令就代表出错,且需要注意的是barrier指令不属于任何bb(basic_block),其上一条指令是上一个基本块的结束,下一条指令是下一个基本块的开始.

7) **CODE_LABEL:** 此指令记录一个标签在rtx指令序列中的位置, 当跳转到一个标签时,实际上就是跳转到此标签的CODE_LABEL所在的位置,一个标签只有唯一的一个CODE_LABEL(记录在DECL_RTL(tree_label_decl)中),其在指令序列的位置,就代表跳转到此标签实际上应该跳转到的位置. 一个BB中如果有CODE_LABEL指令,则其一定是此bb中的第一条指令.

8) **NOTE:** NOTE指令是给后续优化或其他pass处理时用的提示指令,其包含多种子类型,子类型的类型吗记录在 insn_note枚举类型中,如下:

```
//此枚举类型来自 insn-notes.def,这里只介绍几个用到的note
enum insn_note
{
  /* NOTE_INSN_DELETED 类型的note只代表有代码在此处插入了一个标记,同时其获取了此指令的指针,后续代码可以通过此指针定位代码中的当前位置,
     此指令码的含义是,在rtx指令序列的后续优化中不能删除这样的note,否则可能会导致有些代码找不到其标记的note而出错, 在最终转换为汇编代码时所有的note都会被忽略 */
  NOTE_INSN_DELETED,
  NOTE_INSN_DELETED_LABEL,    /* 代表一个代码中位置的标签指令, 通过LABEL_REF引用到此note */
  NOTE_INSN_DELETED_DEBUG_LABEL,
  NOTE_INSN_BLOCK_BEG,
  NOTE_INSN_BLOCK_END,
  /* 每个函数都有一个 NOTE_INSN_FUNCTION_BEG note指令,此note之后的rtx指令序列是此函数函数体expand(也就是gimple => rtl后)生成的的指令序列, 此note
    之前的指令序列,都是负责如函数参数保存,寄存器保存,结构体转换等gcc自动生成的指令 */
  NOTE_INSN_FUNCTION_BEG,
  NOTE_INSN_PROLOGUE_END,
  NOTE_INSN_EPILOGUE_BEG,
  NOTE_INSN_EH_REGION_BEG,
  NOTE_INSN_EH_REGION_END,
  NOTE_INSN_VAR_LOCATION,
  NOTE_INSN_BEGIN_STMT,
  NOTE_INSN_INLINE_ENTRY,
  /* 此note代表代表一个基本块的开始, 每一个基本块的开始都会有一条 NOTE_INSN_BASIC_BLOCK note,但如果此bb有标签的话,那么 NOTE_INSN_BASIC_BLOCK note会
    在标签指令(CODE_LABEL)的后面,作为此bb的第二条指令出现 */
  NOTE_INSN_BASIC_BLOCK,
  NOTE_INSN_SWITCH_TEXT_SECTIONS,
  NOTE_INSN_CFI,                /* intel CET CFI保护的标签 */
  NOTE_INSN_CFI_LABEL,
  NOTE_INSN_UPDATE_SJLJ_CONTEXT,
  NOTE_INSN_MAX
};
```

除了以上这8个节点类型外,其他的节点类型都不属于指令,都不会直接出现在指令序列中,若出现则必须通过某个指令来包裹,为了区分,后续的rtx节点都称为表达式节点.

2. 汇编代码表达式

1) **ASM_INPUT:** 记录一个纯字符串的汇编表达式, 其中记录的字符串最终会原样输出到汇编代码中,**需要注意的是,此表达式中的字符串中的指令若**

对其后代码有副作用,则gcc无法感知到.
2) **ASM_OPERANDS:** 代表一个可能有参数的汇编rtx表达式,和前者的区别除了可以引用gcc变量外，还可以告知gcc此表达式引发的副作用

3. 内存读写表达式
1) **SET:** 此表达式如(set lval x) ,其含义是将x的值存储到lval代表的内存位置中,其中:

- lval,x都是rtx表达式
- lval必须是一个可用于存储的左值节点
- 若lval是rtx(PC),则此表达式可代表控制流的跳转

2) **MEM:** 此表达式如(mem addr) ,代表对内存地址的一个引用,其操作数addr代表被引用的主内存地址
3) **LABEL_REF:** 此表达式代表对**指令**中的一个标签位置的引用,其只有一个操作数,必须为一个rtx(CODE_LABEL)或者一个子类型为NOTE_INSN_DELETED_LABEL的rtx(NODE).
4) **SYMBOL_REF:** 和LABEL_REF类似,区别是此表达式是**对数据**标签的引用(对符号的引用), 如通过此表达式可以引用一个函数/变量.

4. 函数调用与返回表达式
1) **CALL:** 此表达式代表对一个函数的调用,其格式为(call function nargs),其通常作为CALL_INSN的子节点出现
2) **RETURN:** 代表从当前函数返回的一个表达式
3) **SIMPLE_RETURN:** 和RETURN类似,但其是真正就一个简单的返回,而RETURN还可以包裹一些复杂的表达式

5. 寄存器表达式
1) **PC:** 代表当前的程序计数器, 在gcc中通常初始化到pc_rtx变量中
2) **CC0:** 代表当前的程序状态寄存器,在gcc中会认为比较表达式(rtx(COMPARE))的结果会被硬件直接保存到CC0寄存器中
3) **REG: 代表其他的寄存器,包括硬件寄存器和伪寄存器**, 一个rtx(REG)表达式并不是唯一对应一个寄存器的,两个rtx(REG)表达式只要其ORIGINAL_REGNO相同就表示其代表的寄存器是相同的. 在每个函数cgraph_node::expand的过程中(也就是所有rtl pass的执行过程中),会有一个全局数组 regno_reg_rtx 一直用来保存此函数expand以及优化过程中用到的所有硬件寄存器(默认全用)和伪寄存器(动态生成)

6. 比较与条件表达式
1) **COMPARE:** COMPARE表达式用来比较两个操作数, 其没有任何子条件码,gcc认为COMPARE表达式的比较结果会被硬件记录到CC0寄存器中
2) **IF_THEN_ELSE:** 是一个条件跳转表达式, 其有三个操作数:

- op0是比较条件,通常是基于cc和比较码的,如(le (reg:CC 66 cc) (const_int 0 [0]))
- op1是代表then分支成立时要跳转到的目标,其可以是个对标签的引用(label_ref x),也可以是如rtx(PC)的表达式
- op1和op0类似,其是else分支成立时要跳转的目标

7. 算术与比较操作符表达式
1) **PLUS/MINUS/...:** 这些为算术表达式,如(plus x y) 通常代表 x+y的值
2) **NE/EQ/GE/...:** 这些为比较操作符表达式，其通常用来测试CC0的某个bit位的状态,而CC0通常是通过COMPARE表达式来修改的,如((le (reg:CC 66 cc) 0)

六、rtl中的寄存器

1. 平台寄存器信息的获取
平台寄存器信息在后面的多个步骤中都有用到, gcc中平台相关的硬件寄存器信息存储在全局变量 this_target_hard_regs 结构体指针对应的结构体中,其定义如下:

```
struct target_hard_regs *this_target_hard_regs = &default_target_hard_regs;
struct target_hard_regs default_target_hard_regs;
//这里只记录部分使用到的字段
struct target_hard_regs {
    ......
  /* 此数组中的每个下标对应一个硬件寄存器, 其初值来自 FIXED_REGISTERS; 若某寄存器对应的值为1,则此寄存器不参与gcc的通用寄存器分配;若值为0则对应寄存器参与gcc
     -ffixed-reg 参数在 handle_common_deferred_options => fix_register 函数中解析,其会设置对应的下标为1,代码生成过程中此寄存器也不参与通用寄存器分配. */
  char x_fixed_regs[FIRST_PSEUDO_REGISTER];
  /* call_used的意思是此寄存器是可以在函数调用中被破坏的,被调用函数无需保护此寄存器。
    -ffixed-regs指定的寄存器同样也会设置为call used(同样不需保护,保护就可能导致代码生成,故 ffixed同时标记为不许保护), 其初值来自 CALL_USED_REGISTERS */
  char x_call_used_regs[FIRST_PSEUDO_REGISTER];
  char x_call_really_used_regs[FIRST_PSEUDO_REGISTER];
  ......
  /* 记录硬件寄存器名字的数组,其初值来自 REGISTER_NAMES */
  const char *x_reg_names[FIRST_PSEUDO_REGISTER];
  ......
};
```

this_target_hard_regs 是在编译初始化阶段初始化的(init_reg_sets),其从平台相关的配置文件中获取硬件寄存器相关信息(**如哪些硬件寄存器是代码中不可使用的; callee在返回前需要恢复哪些寄存器; 已经每个硬件寄存器的名字等**).

```
//./build/gcc/insn-constants.h
//在编译时确定的平台的所有硬件寄存器及其编号
#define R0_REGNUM 0
......
#define P15_REGNUM 83

//./gcc/config/aarch64/aarch64.h
```

```
//FIRST_PSEUDO_REGISTER为此平台第一个伪寄存器的编号
#define FIRST_PSEUDO_REGISTER        (P15_REGNUM + 1)

// 平台配置文件,记录哪些硬件寄存器是代码中不可使用的; callee需要恢复的; 以及每个硬件寄存器的名字
#define FIXED_REGISTERS                    \
  {                              \
    0, 0, 0, 0,   0, 0, 0, 0,    /* R0 - R7 */        \
    0, 0, 0, 0,   0, 0, 0, 0,    /* R8 - R15 */        \
    0, 0, 0, 0,   0, 0, 0, 0,    /* R16 - R23 */        \
    0, 0, 0, 0,   0, 1, 0, 1,    /* R24 - R30, SP */    \
    0, 0, 0, 0,   0, 0, 0, 0,   /* V0 - V7 */          \
    0, 0, 0, 0,   0, 0, 0, 0,   /* V8 - V15 */        \
    0, 0, 0, 0,   0, 0, 0, 0,   /* V16 - V23 */         \
    0, 0, 0, 0,   0, 0, 0, 0,   /* V24 - V31 */         \
    1, 1, 1, 1,           /* SFP, AP, CC, VG */    \
    0, 0, 0, 0,   0, 0, 0, 0,   /* P0 - P7 */          \
    0, 0, 0, 0,   0, 0, 0, 0,   /* P8 - P15 */          \
  }

/* [R0, R18] 需要被恢复, [R19, R28] 不需要被恢复 */
#define CALL_USED_REGISTERS                \
  {                              \
    1, 1, 1, 1,   1, 1, 1, 1,    /* R0 - R7 */        \
    1, 1, 1, 1,   1, 1, 1, 1,    /* R8 - R15 */        \
    1, 1, 1, 0,   0, 0, 0, 0,    /* R16 - R23 */        \
    0, 0, 0, 0,   0, 1, 1, 1,    /* R24 - R30, SP */    \
    1, 1, 1, 1,   1, 1, 1, 1,    /* V0 - V7 */        \
    0, 0, 0, 0,   0, 0, 0, 0,    /* V8 - V15 */        \
    1, 1, 1, 1,   1, 1, 1, 1,   /* V16 - V23 */         \
    1, 1, 1, 1,   1, 1, 1, 1,   /* V24 - V31 */         \
    1, 1, 1, 1,           /* SFP, AP, CC, VG */    \
    1, 1, 1, 1,   1, 1, 1, 1,    /* P0 - P7 */        \
    1, 1, 1, 1,   1, 1, 1, 1,    /* P8 - P15 */        \
  }

#define REGISTER_NAMES                        \
  {                                \
    "x0",  "x1",  "x2",  "x3",  "x4",  "x5",  "x6",  "x7",    \
    "x8",  "x9",  "x10", "x11", "x12", "x13", "x14", "x15",    \
    "x16", "x17", "x18", "x19", "x20", "x21", "x22", "x23",    \
    "x24", "x25", "x26", "x27", "x28", "x29", "x30", "sp",    \
    "v0",  "v1",  "v2",  "v3",  "v4",  "v5",  "v6",  "v7",    \
    "v8",  "v9",  "v10", "v11", "v12", "v13", "v14", "v15",    \
    "v16", "v17", "v18", "v19", "v20", "v21", "v22", "v23",    \
    "v24", "v25", "v26", "v27", "v28", "v29", "v30", "v31",    \
    "sfp", "ap",  "cc",  "vg",                    \
    "p0",  "p1",  "p2",  "p3",  "p4",  "p5",  "p6",  "p7",    \
    "p8",  "p9",  "p10", "p11", "p12", "p13", "p14", "p15",    \
  }

//./gcc/hard-reg-set.h
// 记录平台硬件寄存器的初始配置
static const char initial_fixed_regs[] = FIXED_REGISTERS;
static const char initial_call_used_regs[] = CALL_USED_REGISTERS;
static const char *const initial_reg_names[] = REGISTER_NAMES;


//main => toplev::main => general_init => init_reg_sets
//将平台硬件寄存器信息,复制到 this_target_hard_regs 对应结构体中
void init_reg_sets (void)
{
  ......
  memcpy (fixed_regs, initial_fixed_regs, sizeof fixed_regs);
  memcpy (call_used_regs, initial_call_used_regs, sizeof call_used_regs);
    ......
  memcpy (reg_names, initial_reg_names, sizeof reg_names);
  ......
}
```

2. 全局统一的寄存器的rtx表达式的生成

在rtl格式下, 寄存器是通过一个rtx(REG)表达式来表示的, 一个rtx(REG)表达式中的主要信息就是寄存器的编号,故两个rtx(REG)只要编号相同,就代表同一个寄存器.

在词法分析之前,gcc就为所有函数都会用到的硬件寄存器,虚拟寄存器生成了rtx(REG)表达式并保存在this_target_rtl中;

```
struct target_rtl *this_target_rtl = &default_target_rtl;
struct target_rtl default_target_rtl;
struct target_rtl {
  /* 这里面存的是gcc中一些逻辑寄存器的rtx表达式, 如global_rtl[GR_STACK_POINTER]代表栈指针寄存器的rtx(REG)表达式,此表达式中的ORIGINAL_REGNO则记录栈指针真i
  rtx x_global_rtl[GR_MAX];
  rtx x_pic_offset_table_rtx;
  rtx x_return_address_pointer_rtx;
  /* 这里存的是代表此平台各个硬件寄存器编号的rtx(REG)的表达式, x_initial_regno_reg_rtx[i] 中的ORIGINAL_REGNO 就是 i */
  rtx x_initial_regno_reg_rtx[FIRST_PSEUDO_REGISTER];
  rtx x_top_of_stack[MAX_MACHINE_MODE];
  rtx x_static_reg_base_value[FIRST_PSEUDO_REGISTER];
  struct mem_attrs *x_mode_mem_attrs[(int) MAX_MACHINE_MODE];
  bool target_specific_initialized;
}

/* global_rtl[GR_STACK_POINTER]保存当前栈指针寄存器的rtx表达式, 在init_emit_regs 函数中会将其初始化为编号为 STACK_POINTER_REGNUM(31) 的硬件寄存器(就是S
#define stack_pointer_rtx       (global_rtl[GR_STACK_POINTER])
```

```
#define frame_pointer_rtx       (global_rtl[GR_FRAME_POINTER])         /* 同上，对应硬件寄存器编号 FRAME_POINTER_REGNUM (64) */
#define hard_frame_pointer_rtx  (global_rtl[GR_HARD_FRAME_POINTER])    /* 同上，对应硬件寄存器编号 HARD_FRAME_POINTER_REGNUM (29) */
#define arg_pointer_rtx         (global_rtl[GR_ARG_POINTER])           /* 同上，对应硬件寄存器编号 ARG_POINTER_REGNUM (65) */

//main => toplev::main => do_compile => backend_init => init_emit_regs
void init_emit_regs (void)
{
  /* 确定所有硬件寄存器能存储的最大机器模式(一维数组reg_raw_mode),以及确定指定机器模式从指定硬件寄存器开始需要多少个字节来存储(二维数组hard_regno_nregs) */
  init_reg_modes_target ();
  /* 为 this_target_rtl中所有global_rtl中的寄存器生成RTX(REG)表达式，代表这些寄存器在平台的编号，在真正编译开始之前(此函数中)统一初始化一次,后面每个函数ex
    自身全局寄存器数组时,会从 this_target_rtl中复制每个函数相同的 硬件寄存器/global_rtl/虚拟寄存器表达式(直接复制其编号即可) */
  stack_pointer_rtx = gen_raw_REG (Pmode, STACK_POINTER_REGNUM);
  frame_pointer_rtx = gen_raw_REG (Pmode, FRAME_POINTER_REGNUM);
  hard_frame_pointer_rtx = gen_raw_REG (Pmode, HARD_FRAME_POINTER_REGNUM);
  arg_pointer_rtx = gen_raw_REG (Pmode, ARG_POINTER_REGNUM);
  virtual_incoming_args_rtx = gen_raw_REG (Pmode, VIRTUAL_INCOMING_ARGS_REGNUM);
  virtual_stack_vars_rtx = gen_raw_REG (Pmode, VIRTUAL_STACK_VARS_REGNUM);
  virtual_stack_dynamic_rtx = gen_raw_REG (Pmode, VIRTUAL_STACK_DYNAMIC_REGNUM);
  virtual_outgoing_args_rtx = gen_raw_REG (Pmode, VIRTUAL_OUTGOING_ARGS_REGNUM);
  virtual_cfa_rtx = gen_raw_REG (Pmode, VIRTUAL_CFA_REGNUM);
  virtual_preferred_stack_boundary_rtx = gen_raw_REG (Pmode, VIRTUAL_PREFERRED_STACK_BOUNDARY_REGNUM);

  /* 为 this_target_rtl中所有硬件寄存器生成RTX(REG)表达式,其使用流程和global_rtl相同 */
  for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
    initial_regno_reg_rtx[i] = gen_raw_REG (reg_raw_mode[i], i);
    .......
}
```

3.每个函数expand之前的此函数全局寄存器数组(regno_reg_rtx)的初始化

在每个函数expand之前,都要为此函数构建一个全局rtx寄存器数组(regno_reg_rtx), 此数组中保存了当前函数中用到的所有寄存器信息(以rtx(REG)表达式形式保存),因为硬件寄存器和虚拟寄存器每个函数都认为会默认使用到的,故在每个函数expand之前都会先将这部分寄存器的rtx表达式复制到此函数的全局寄存器数组中:

```
//main => toplev::main => do_compile => compile_file => symbol_table::finalize_compilation_unit
// => symbol_table::compile => output_in_order => cgraph_node::expand => init_function_start => prepare_function_start => init_emit
void init_emit (void)
{
    ......
  /* 为当前函数的全局寄存器指针数组(regno_reg_rtx)初始大小，默认预留 100个伪寄存器的位置,不够再分配 */
  crtl->emit.regno_pointer_align_length = LAST_VIRTUAL_REGISTER + 101;

  /* 为全局寄存器数组分配空间,其中每个元素都是一个指向rtx_def表达式的指针 */
  regno_reg_rtx = ggc_cleared_vec_alloc<rtx> (crtl->emit.regno_pointer_align_length);

  /* 从this_target_rtl中复制当前平台的硬件寄存器的rtx(REG)指针信息到当前函数的全局寄存器指针数组 regno_reg_rtx中(rtx(REG)中关键的是寄存器编号,故直接复制即
  memcpy (regno_reg_rtx, initial_regno_reg_rtx, FIRST_PSEUDO_REGISTER * sizeof (rtx));

  /* 复制6个虚拟寄存器的全局rtx(REG)表达式到regno_reg_rtx 中*/
  init_virtual_regs ();
   ......
}
```

初始化之后当前函数的全局寄存器数组 regno_reg_rtx 内容如下:

```
rtx * regno_reg_rtx {
    [0]                //|
    [...]              //|=>  aarch64中一共83个硬件寄存器
    [P15_REGNUM]       //|
    [VIRTUAL_INCOMING_ARGS_REGNUM] = virtual_incoming_args_rtx;                       //|
    [VIRTUAL_STACK_VARS_REGNUM] = virtual_incoming_args_rtx                           //|
    [VIRTUAL_STACK_DYNAMIC_REGNUM] = virtual_stack_dynamic_rtx                        //| => aarch64中一共6个虚拟寄存器
    [VIRTUAL_OUTGOING_ARGS_REGNUM] = virtual_outgoing_args_rtx                        //|
    [VIRTUAL_CFA_REGNUM] = virtual_cfa_rtx                                            //|
    [VIRTUAL_PREFERRED_STACK_BOUNDARY_REGNUM] = virtual_preferred_stack_boundary_rtx  //|
    [LAST_VIRTUAL_REGISTER]                 //在aarch64中编号为89
    [...]                                   //之后是函数expand过程中动态生成的伪寄存器
}
```

总结rtl中使用的寄存器:

1. 在rtl中,寄存器可以分为三部分,分别是硬件寄存器,虚拟寄存器和伪寄存器,其中:
   1) 硬件寄存器是平台相关的,默认认为每个函数都会使用到这些寄存器
   2) 虚拟寄存器是gcc逻辑相关的,虽然其最后通常都会对应到一个硬件寄存器上,但在gcc解析过程中需要单独使用它来确定语义.
   3) 伪寄存器在逻辑寄存器,在rtl 最开始的pass中, 会认为伪寄存器是无限使用的，而直到统一寄存器分配时才会将其对应到某个硬件寄存器中
2. 由于硬件寄存器是平台相关,且每个函数都会使用的:
   1) 首先硬件寄存器是来自于平台的, 在编译的开始 initial_fixed_regs/initial_call_used_regs/initial_reg_names等结构体中记录了此平台的硬件寄存器信息
   2) 而在后端初始化过程中,会为这些硬件寄存器统一生成rtx(REG)表达式,代表其对应的寄存器编号,并保存在this_target_rtl
   3) 在每个函数expand之前,其全局寄存器数组regno_reg_rtx要从 this_target_rtl中将这些硬件寄存器的信息复制过来,代表当前函数默认使用了这些硬件寄存器

七、栈空间分配与函数调用标准

在讨论栈分配之前,首先是要确定当前函数的栈是如何构建的,主要需要确定的内容包括栈的增长方向,出入栈的方式,局部变量扩展方向,参数扩展方向,在gcc中这4点是分别通过四个宏开关控制的:

* **STACK_GROWS_DOWNWARD:** 此宏决定当前函数栈的增长方向,为1时代表栈向低地址方向增长,为0代表向高地址方向增长, 这里需要注意的是, 不要一讨论栈就认为其默认其是向低地址方向增长的(虽然这种情况占了大多数)

* **STACK_PUSH_CODE/STACK_POP_CODE:** 此宏决定栈的出入栈方式, PUSH和POP的选项是相对的,可选项包括: PRE_DEC/PRE_INC/POST_DEC/POST_INC,这四个选项决定出入栈时是先移动sp还是先存储; sp移动方向是加还是减:

  - PRE_DEC: 先做sp=sp-4; 再push/pop
  - PRE_INC: 先做 sp = sp+4; 再push/pop
  - POST_DEC: 先push/pop;再做 sp = sp-4;
  - POST_INC: 先push/pop; 再做 sp = sp+4;

虽说push/pop只要选择匹配的选项即可,但在gcc中 push的选项只有 PRE_DEC/PRE_INC, 而pop则只有对应的POST_DEC/POST_INC,也就是说**gcc编译的代码中,push操作应该总是先移动sp,再存入值; pop操作应该总是先读取值再移动sp;且当栈向低地址增长时,push总要对应的做DEC操作; 当栈向高地址增长时,push总要对应的做INC操作**.

* **FRAME_GROWS_DOWNWARD:** 此宏决定函数局部变量的扩展方向, 为1时局部变量向低地址方向扩展; 为0时局部边变量向高地址方向扩展;

* **ARGS_GROW_DOWNWARD:** 此宏决定函数的传入栈参数(若有)在caller栈中的扩展方向,为1时候函数的传入栈参数向低地址增长,为0时函数的传入栈参数向高地址增长;

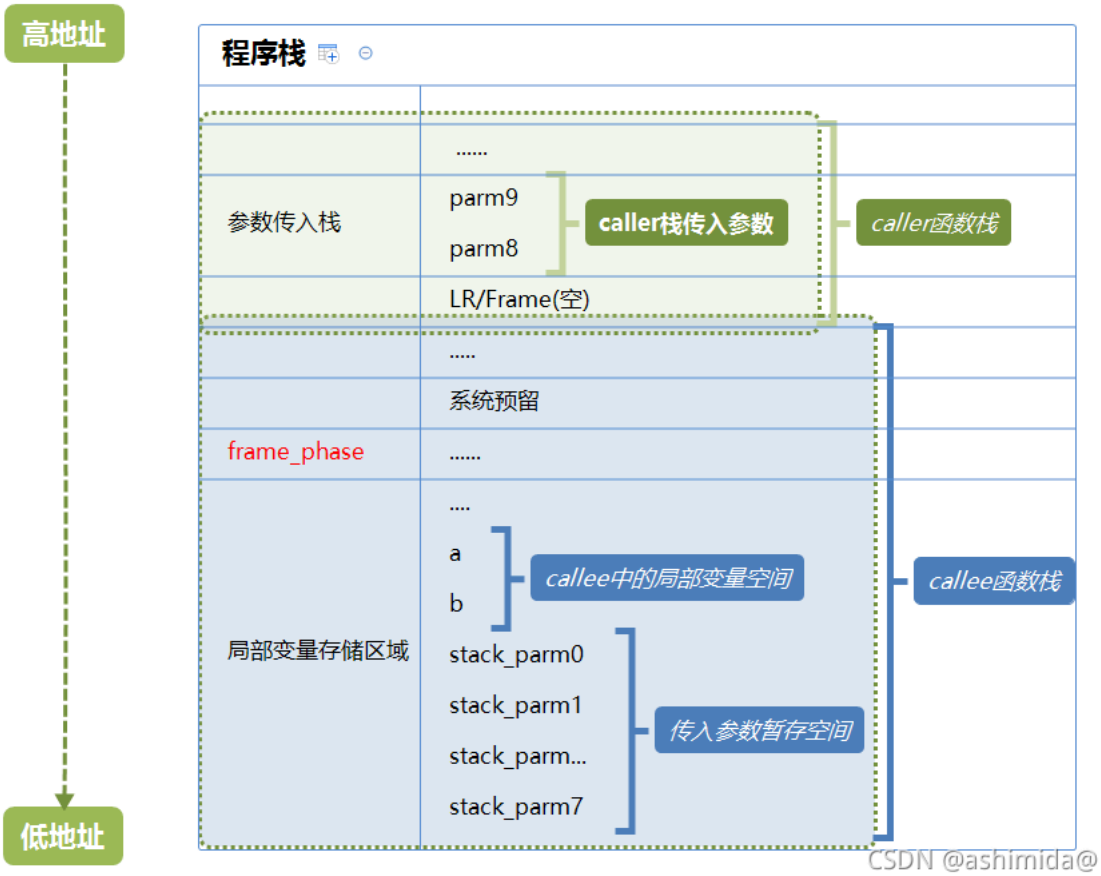
而在当前AARCH64的测试平台上,以上宏的值为:

```
1. /* 代表aarch64栈向低地址方向增长 */
2. #define STACK_GROWS_DOWNWARD    1
3.
4. /* PUSH/POP如何操作由STACK_GROWS_DOWNWARD决定 */
5. #if STACK_GROWS_DOWNWARD
6. #define STACK_PUSH_CODE PRE_DEC              /* push操作为 sp=sp-4; store; */
7. #define STACK_POP_CODE POST_INC              /* pop操作为 read; sp=sp+4; */
8. #else ...
9.
10. #define ARGS_GROW_DOWNWARD     0             /* 函数传入的栈参数向高地址方向增长 */
11. #define FRAME_GROWS_DOWNWARD    1            /* 函数局部变量向低地址方向增长 */
12.
```

根据此配置,在aarch64的整个栈布局就可以基本确定了,以函数caller调用了函数callee为例:

```
1. int callee(int x0, int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9)
2. {
3.     int a;
4.     int b;
5.     callee_codes;
6. }
7.
8. int caller()
9. {
10.     callee(0,1,2,3,4,5,6,7,8,9);
11. }
```

在callee函数体中的代码(callee_codes)真正执行前, 当前的函数栈如下:



从此函数栈可以看出:

1. 当前函数栈的增长方向是向低地址方向增长的
2. 函数传入参数是向高地址增长的(parm8/9的分配顺序)
3. 函数局部变量的分配是向低地址增长的(a/b的分配顺序)

除了栈分配方向之外,此函数栈的生成还与函数的调用标准有关,以当前测试的aarch64平台使用的AAPCS64标准为例,其要求(大多数情况下):

- 函数的前8个实数参数由寄存器R0-R7传入
- 超过8个参数的其余参数则通过栈传入

caller和callee同时满足AAPCS64标准,且使用相同的栈空间分配方式(生长方向,分配方向), 故:

- 在caller中是完全可以直接按照约定将参数放置到指定的硬件寄存器/栈偏移中
- 在callee中也是完全可以直接按照约定将传入参数从指定的硬件寄存器/栈偏移中取出的

作为caller部分,当caller调用callee时,caller需要负责给callee传递实参,其:

1. 首先要通过AAPCS64标准计算出每个实参应该通过哪个硬件寄存器/栈偏移传入
2. 在调用callee前先生成代码(实际上是在rtl中发射指令序列), 将实参复制到对应的硬件寄存器或栈偏移中
3. 发起对callee的调用
4. 按照AAPCS64标准,若函数有返回值则通常通过R0寄存器返回,若caller需要callee的返回值,则从R0中接收返回值

需要注意的是, 在gcc c中参数是不能有默认值的, 故caller传给callee的实参个数只能 >= callee的形参个数(大于的情况发生在不定参数函数的调用)

作为callee部分,当其中的指令序列被执行时就代表其被调用到了,其:

1. 实际上是先要在callee栈上为callee中的局部变量预留空间(如这里为局部变量a,b预留的空间)
2. 之后才是通过AAPCS64标准计算出每个传入参数(caller传递给callee的参数称为传入参数)应该通过哪个硬件寄存器/栈偏移获取到
3. 然后在callee栈中为传入参数预留临时存储空间(若开启优化则是在伪寄存器中预留),如这里为形参x0-x7预留的栈临时存储空间stack_parm0-7
4. 在函数体代码前生成代码(发射rtl指令序列), 此代码负责将callee的所有寄存器参数复制到其对应的临时栈空间中(若开启优化则复制到伪寄存器中)
    1. 之所以在callee中需要将寄存器传入参数复制到临时栈中,是因为如果不复制的话,x0-x7寄存器对于callee来说相当于暂不可用了;
    2. 而传入的栈参数通常不用复制(除非是因为优化暂存到伪寄存器中)，因为栈参数并不会影响硬件寄存器的正常使用.

故在上图中可以看到三部分栈空间的使用:

- caller栈的传入栈参数区域(parm8/9)
- callee栈的局部变量区域(a/b)
- callee栈的传入参数暂存区域(stack_parm0-7)

栈分配的更多细节见后续 caller/callee的代码分析