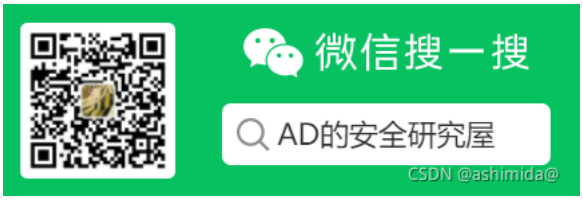
GCC源码分析(十三) — 机器描述文件

更多内容可关注微信公众号



在gcc中,各个平台都有自己的机器描述文件,机器描述文件主要包括两种:

- ${target}.[md]: 在aarch64平台为 ./config/aarch64/aarch64.md文件, 此文件中主要记录的是aarch64平台的指令模板, 此模板用来决定最终RTL是如何生成汇编代码的.
- ${target}.[ch]: 在aarch64平台中为 ./config/aarch64/aarch64.h, ./config/aarch64/aarch64.c 两个文件. 除了RTL=>汇编代码的指令模板外, 还有大量的无法用指令模板描述的目标机器信息(GIMPLE => RTL阶段也会使用),如寄存器信息,存储布局信息,硬件函数实现等,这些信息都是需要用C语言进行描述的,其中:
    - 可设置为宏定义的部分记录在aarch64.h 中
    - 目标机器相关的函数记录在aarch64.c 中

下面简单介绍机器描述文件中几个较为重要的内容:

一、targetm

targetm是用来描述目标机器的结构体,其通常在${target}.c中定义如下:

```
./gcc/config/aarch64.c
struct gcc_target targetm = TARGET_INITIALIZER;
```

此结构体内部定义十分复杂,包含目标机器上汇编代码输出,指令调度,向量化,函数调用传参返回,语言相关,TLS等相关宏定义与函数定义,在后续代码中若看到了如 targetm.xxx的引用,都要到targetm中找其对应的定义,此结构体简单记录如下:

```
struct gcc_target {
  struct asm_out{ ... };     /* 其中包含目标机器汇编语言生成相关的定义和函数 */
  struct sched { ... };      /* 其中包含于指令调度相关的函数定义 */
  struct vectorize { ... };  /* 其中包含与向量化相关的函数定义 */
  struct calls { ... };      /* 其中包含与函数调用,参数传递等相关的函数定义 */
  struct c { ... };          /* 记录与c语言前端相关的处理函数 */
  struct emutls { ... };     /* 记录与县城局部存储相关的数据定义和函数定义 */
  struct target_option_hooks { ... }; /* 记录与目标机器编译选项处理相关的钩子函数 */
  ...
}
```

以其中asm_out为例:

```
// ./build/gcc/target-hook-def.h
#ifndef TARGET_ASM_OPEN_PAREN
#define TARGET_ASM_OPEN_PAREN "("
#endif
#ifndef TARGET_ASM_CLOSE_PAREN
#define TARGET_ASM_CLOSE_PAREN ")"
#endif
#ifndef TARGET_ASM_BYTE_OP
#define TARGET_ASM_BYTE_OP "\t.byte\t"
#endif
struct asm_out
{
    const char *open_paren, *close_paren;    /* 对应 TARGET_ASM_OPEN_PAREN,  TARGET_ASM_CLOSE_PAREN, 也就是在aarch64平台汇编代码中括号的定义分别是 (
    const char *byte_op;        /* 汇编代码中字节标识符, 对应字符串为 "\t.byte\t" */
    struct asm_int_op
    {
      const char *hi;
      const char *si;
      const char *di;
      const char *ti;
    } aligned_op, unaligned_op;
      ......
   void (* function_prologue) (FILE *, HOST_WIDE_INT);    /* 为当前函数插入函数开始的prologue的代码 */
```

```
24.       ......
25. }
```

二、默认的编译选项

在gcc中可通过 gcc -dumpspecs 来查看各个文件默认的编译选项(这里只做个备忘记录)

```
1. tangyuan@ubuntu:~/compile/gcc-9.2.0/gcc/config$ gcc -dumpspecs |head
2. *asm:
3. %{m16|m32:--32}  %{m16|m32|mx32:;:--64}  %{mx32:--x32}  %{msse2avx:%{!mavx:-msse2avx}}
4.
5.
6. *asm_debug:
7. %{%:debug-level-gt(0):%{gstabs*:--gstabs}%{!gstabs*:%{g*:--gdwarf2}}} %{fdebug-prefix-map=*:--debug-prefix-map %*}
8.
9.
10. *asm_final:
11. %{gsplit-dwarf:
12.        objcopy --extract-dwo     %{c:%{o*:%*}%{!o*:%b%O}}%{!c:%U%O}      %{c:%{o*:%:replace-extension(%{o*:%*} .dwo)}%{!o*:%b.dwo}}%{!c:%b.dwo}
13.        objcopy --strip-dwo       %{c:%{o*:%*}%{!o*:%b%O}}%{!c:%U%O}     }
```

三、存储布局

存储布局(storage layout)主要包括目标机器的:

1. 位顺序和字节顺序:

位顺序和字节顺序实际上就是常说大小端,大小端分为位(bit)的大小端,和字节(byte)的大小端,在aarch64中:

```
1. ./gcc/config/aarch64/aarch64.h
2. #define BITS_BIG_ENDIAN 0    /* 代表位顺序采用小端顺序,此时数字0x1 存储形式为 0b 0000 0001(而非 0b 1000 0000) */
3. #define BYTES_BIG_ENDIAN (TARGET_BIG_END != 0)    /* 代表字节顺序的大小端,在本机测试这里为小端, 也就是数字 0x21 存储为 0b 0000 0010 0000 0001 */
4. #define WORDS_BIG_ENDIAN (BYTES_BIG_ENDIAN)    /* 多个字的存储顺序与字节序相同 */
```

2. 类型宽度

类型宽度是基本存储类型占用空间的大小,在aarch64中:

```
1. //./build/gcc/insn-modes.h    (from genmodes.c)
2. #define BITS_PER_UNIT (8)     /* 一个可寻址单元的位数(bits),通常指一个字节(byte)的宽度 */
3.
4. //./config/aarch64/aarch64.h
5. #define UNITS_PER_WORD 8      /* 一个word包含多少个可寻址的单元(bytes)*/
6.
7. //./gcc/default.h
8. #ifndef BITS_PER_WORD
9. #define BITS_PER_WORD (BITS_PER_UNIT * UNITS_PER_WORD)    /* 正常一个word的大小是二者的乘积(单位为bits) */
10. #endif
11.
12. #ifndef POINTER_SIZE
13. #define POINTER_SIZE BITS_PER_WORD        /* 指针占用空间大小, 指针占用的空间就是一个word的大小 */
14. #endif
```

3. 机器模式提升

对于aarch64来说,机器模式提升主要是将 1/2 byte的对象,提升至4byte,这样刚好可用一个寄存器存储

```
1. ./gcc/config/aarch64/aarch64.c
2. #define PROMOTE_MODE(MODE, UNSIGNEDP, TYPE) \   /* 此宏负责将aarch64中的QI(1 byte)/HI(2 byte) 提升为SI(4byte),无其他操作 */
3. .......
```

4. 存储对齐

存储对齐的目的是为了对数据的加速访问,在aarch64中存储对齐相关宏定义如下:

```
1. ./gcc/config/aarch64/aarch64.c
2. #define PARM_BOUNDARY 64          /* 函数参数在堆栈中的对齐位数,以 bit为单位; 所有堆栈中的参数至少要满足此对齐要求 */
3. #define STACK_BOUNDARY 128        /* 栈的边界地址的对齐粒度 */
4. #define FUNCTION_BOUNDARY 32      /* 默认函数入口地址的对齐粒度,在aarch64中函数的默认对齐粒度为4字节 */
5. ...
```

5. 编程语言中数据类型的存储布局

编程语言中数据类型的大小只与平台有关, 与BITS_PER_UNIT是无关的(BITS_PER_UNIT 的定义来自于genmodes.c, 其默认就是8 bit):

```
1. ./gcc/genmodes.c
2. static void
3. create_modes (void)
```

```
{
#include "machmode.def"
    .......
#ifdef BITS_PER_UNIT
  bits_per_unit = BITS_PER_UNIT;
#else
  bits_per_unit = 8;
#endif
```

而编程语言中数据类型:

```
./gcc/config/aarch64/aarch64.c
#define INT_TYPE_SIZE        32
#define LONG_TYPE_SIZE        (TARGET_ILP32 ? 32 : 64)
#define POINTER_SIZE        (TARGET_ILP32 ? 32 : 64)
#define LONG_LONG_TYPE_SIZE    64
#define FLOAT_TYPE_SIZE        32
#define DOUBLE_TYPE_SIZE    64
```

也就是**说在aarch64平台中,INT类型始终是32 bit(也就是4字节),** 但需要注意的是: **通常认为编程语言中int是4字节, long是根据32/64位平台不同而分为4/8字节,这种说法不总是正确的**(虽然通常是正确的), 在gcc中可以看到,某些平台中 int并不是 4字节(虽然大多数平台都是4字节):

```
tangyuan@ubuntu:~/compile/gcc-9.2.0/gcc/config$ grep "#define INT_TYPE_SIZE" -R ./|grep -v 32
./sparc/sparc.c:#define INT_TYPE_SIZE BITS_PER_WORD
./rl78/rl78.h:#define INT_TYPE_SIZE                    16
./msp430/msp430.h:#define INT_TYPE_SIZE                16
./avr/avr.h:#define INT_TYPE_SIZE (TARGET_INT8 ? 8 : 16)
./stormy16/stormy16.h:#define INT_TYPE_SIZE 16
```

四、机器描述文件${target}.md的基本内容

机器描述文件的作用是将目标机器的特性引入到编译器中,从知道编译器根据目标机器的特性将Insn转换为目标代码. 机器描述文件中的主要内容包括对于目标机器的:

- 指令模板(Insn Pattern)的定义
- 常量(Constant)的定义
- 属性(Attribute)的定义
- 自定义断言(User-Defined Predicte)的定义
- 自定义约束(User-Defined Constraint)的定义
- 枚举器(Iterator)的定义
- 流水线(Pipeline)的声明
- 窥孔优化(Peephole Optimization)的定义等

这里主要介绍其中的指令模板, 指令模板的格式为:

```
(define_insn 指令模板名称
    RTL模板
    条件
    输出模板
    属性
 )
```

以jump指令的指令模板为例:

```
//./config/aarch64/aarch64.md
(define_insn "jump"                                          /* 指令模板名称是一个字符串,唯一的描述了此指令模板,如 这里的"jump"*/
  [(set (pc) (label_ref (match_operand 0 "" "")))]           /* 指令模板中的RTL模板, jump的rtx指令的内容就是根据此RTL模板生成的(见 gen_jump) */
  ""                                                         /* 此条件作为rtx指令生成汇编代码时,判断rtx指令是否可用此输出模板匹配的最终判断条件
                                                               如 SIMPLE_RETURN指令是否生成则取决于aarch64_use_simple_return_insn_p的判断,见 recog*/
  "b\\t%l0"                                                  /* 此指令的输出模板,最终指令输出到汇编文件中的内容取自此模板 */
  [(set_attr "type" "branch")]                               /* 指令模板的属性,通常在流水线优化中应用 */
)
```

五、机器描述文件的信息提取

机器描述文件中的信息最终是被提取为一个个.[ch]文件,这些文件提取后也最终参与了编译,gcc在提取这些文件前会先编译出一系列二进制程序用来处理指令模板,包括:

- **gencode:** 从指令模板(如aarch64.md)中提取所有指令模板，并根据模板名为此模板生成系统唯一编号(如"jump"=>CODE_FOR_jump)并将所有指令模板编号都记录在枚举类型 insn_code 中(枚举类型元素个数为NUM_INSN_CODES),最终生成到文件./build/gcc/insn-codes.h中
- **genoutput:** 从指令模板(如aarch64.md)中提取了所有指令的操作数信息(operand_data[])和所有指令的RTL模板信息(insn_data[NUM_INSN_CODES])等, 最终生成到文件./build/gcc/insn-output.c中
- **genrecog:** 根据指令模板(如aarch64.md)中的RTL模板,生成函数recog,其函数用来为rtl阶段的一条rtx指令查找符合其指令格式的模板编号(insn_code中的编号),最终生成到文件 ./build/gcc/insn-recog.c中
- **genextract:** 根据指令模板(如aarch64.md)中的RTL模板,生成函数 insn_extract,此函数用来根据模板编号(insn_code),将一条rtx指令的操作数全部提取出来,最终生成到文件 ./build/gcc/insn-extract.c中.

- **genopinit:** 根据指令模板(如aarch64.md),生成pats数组,此数组中每个元素都记录了一个gcc指令模板编号(scode)与平台指令模板编号(icode)之间的对应关系，最终生成到文件 ./build/gcc/insn-opinit.c中.
- **gentmit:** 根据指令模板(如aarch64.md)中的RTL模板,生成gcc中各类指令的rtx指令生成函数(如jump指令的rtx指令通过生成的gen_jump函数生成)，最终生成到文件./build/gcc/insn-emit.c中.

此外根据模板gcc还生成了如一些属性相关的.[ch]文件,这里就不一一列举了.

在编译过程中,gcc先通过源码编译出了以上二进制,并通过运行以上二进制程序将指令模板转换为一个个.[ch]文件,最终在增加了这些文件的源码之上,编译出cc1等一系列最终二进制.

六、机器描述文件在gcc中的使用

前面介绍了一系列关于gcc中的机器描述文件, 实际上机器描述文件在源码中的使用主要体现在三处:

1. gimple指令转换为rtx指令时:

在gimple指令转rtx指令时,rtx指令通常是需要调用指令模板中生成的函数生成的,以ggoto语句的展开和mov语句的生成为例:

```
/* ggoto语句的展开流程为: ... => expand_goto => emit_jump => targetm.gen_jump (label) => target_gen_jump => gen_jump */

//./build/gcc/insn-emit.c
/* ../../../gcc-9.2.0/gcc/config/aarch64/aarch64.md:377 */
rtx gen_jump (rtx operand0 ATTRIBUTE_UNUSED)           /* gen_jump语句实际上是根据 aarch64.md:337行的指令模板生成的 */
{
  /* (set pc_rtx, label_ref( void, operand0)) */
  return gen_rtx_SET (pc_rtx, gen_rtx_LABEL_REF (VOIDmode, operand0));
}

//./config/aarch64.md:377
(define_insn "indirect_jump"
  [(set (pc) (match_operand:DI 0 "register_operand" "r"))]
  ""
  "br\\t%0"
  [(set_attr "type" "branch")]
)


/* gcc语义上的一条机器模式为si的mov指令的展开流程为:   ... => aarch64_emit_move => emit_move_insn_1 => gen_movsi */
//./gcc/.expr.c
rtx_insn * emit_move_insn_1 (rtx x, rtx y)
{
  machine_mode mode = GET_MODE (x);
  enum insn_code code
  /* 此函数先根据当前的gcc操作类型(mov_optab) + 机器模式(如si), 确定要生成的rtx指令的gcc指令编码(scode),然后通过pats数组
     (./build/gcc/insn-opinit.c) 查找gcc指令编码对应的指令模板编码(icode), 这里 mov_optab = 0x7D, mode = 0xe(scode = 0x7d000e),
     最终返回的icode为 5157(CODE_FOR_movsi)
  */
  code = optab_handler (mov_optab, mode);    /* 从pat数组中查找当前指令的机器模板编号 */
  if (code != CODE_FOR_nothing)
    return emit_insn (GEN_FCN (code) (x, y));    /* 调用机器模板对应的生成函数来生成rtx指令,最终执行的如 gen_movsi */
    .......
}

//./build/gcc/insn-output.c
const struct insn_data_d insn_data[] =
{
  ......
  /* ../../../gcc-9.2.0/gcc/config/aarch64/aarch64.md:1057 */
  {                                                 /* insn_data根据指令模板为每个指令生成此结构体 */
    "movsi",   { 0 },
    { (insn_gen_fn::stored_funcptr) gen_movsi },    /* gen_movsi为为movsi生成rtx指令的函数 */
    &operand_data[6512],
    .......
  },
 .......
}

//./build/gcc/insn-output.c
static const char * output_46 (rtx *operands, rtx_insn *insn )
{
  switch (which_alternative)
    {
    case 0: return "mov\t%w0, %w1";
    case 1: return "mov\t%w0, %w1";
  .......
}


/* ../../../gcc-9.2.0/gcc/config/aarch64/aarch64.md:1057 */
rtx gen_movsi (rtx operand0, rtx operand1)         /* 此函数是通过解析指令模板为movsi自动生成的其rtx指令生成函数 */
{
  rtx_insn *_val = 0;
  .......
  emit_insn (gen_rtx_SET (operand0, operand1));
  .......
}
```

2. 虚拟寄存器实例化时:

gimple=>rtl时,虽然根据某指令模板生成了rtx指令,但此时此rtl指令并没有记录其生成的那个指令模板(INSN_CODE(rtx)),这应该是因为后续优化,寄存器替换等过程中rtx指令被修改后,其对应的指令模板可能也随之改变了. 而直到虚拟寄存器实例化时(instantiate_virtual_regs_in_insn)才会为大部分rtx指令重新确定其指令模板编号,此流程为:

```
./gcc/function.c
void instantiate_virtual_regs_in_insn (rtx_insn *insn)
{
  ......
  if (recog_memoized (insn) < 0) fatal_insn_not_found (insn);
}

./gcc/recog.h
int recog_memoized (rtx_insn *insn)
{
  /* 获取此指令的子表达式(PATTERN) 对应的指令模板, 并将其记录到此指令中,
     其中 recog函数是根据指令模板自动生成的,其作用是反向查找一条rtx指令应该使用哪个指令模板
  */
  if (INSN_CODE (insn) < 0) INSN_CODE (insn) = recog (PATTERN (insn), insn, 0);
  return INSN_CODE (insn);
}
```

虚拟寄存器实例化时, 大部分指令都通过recog函数确定了其对应的指令模板,此指令模板的编号记录在 INSN_CODE(insn)中,供后续汇编代码输出时使用.

3.汇编代码输出时:

在函数rtl expand的最后(变量expand不需要走模板,直接在aarch64.c中有固定函数),会通过 pass_final将当前函数的所有指令输出到汇编代码, 对于每条指令来说,其最终的输出函数为final_scan_insn_1:

```
/*
  此函数中会根据此指令在2.中确定的指令模板编号,先提取所有操作数(提取函数insn_extract也是根据指令模板生成的),同时提取此指令模板中的输出格式字符串(来自insn_da
*/
rtx_insn * final_scan_insn_1 (rtx_insn *insn, FILE *file, ...)
{
    ......
    insn_code_number = recog_memoized (insn);                  /* 获取指令模板编号(已有不重新计算) */
    /* 这里最终调用函数 insn_extract 提取此rtx指令中所有操作数, 此函数也是根据指令模板生成的(./build/gcc/insn-extract.c) */
    cleanup_subreg_operands (insn);
    templ = get_insn_template (insn_code_number, insn);        /* 根据指令模板编号,获取输出指令的汇编模板,如 "jump" 指令对应  "br\\t%0" */
    output_asm_insn (templ, recog_data.operand);               /* 根据操作数和指令模板字符串,输出汇编代码 */
}
```

由上可知,指令模板在gcc代码中的主要作用是:

1. gimple=> rtx指令时,gimple指令应该生成何种rtx指令需要参考指令模板
2. rtx指令序列优化后, 每个rtx指令对应的指令模板(其应该以何种汇编格式输出)需要重新确定,此解析函数的生成也要依赖指令模板
3. rtx指令最终输出为汇编代码时,确定每个操作数的函数,以及汇编代码的格式(输出模板),也均来自于指令模板

其他关于机器描述文件的细节可参考<深入分析GCC>,王亚刚