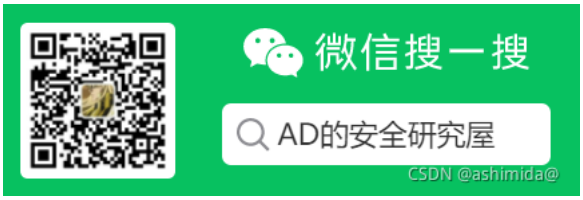
GCC源码分析(十五) — gimple转RTL(pass_expand)(上)



由前可知,函数的expand(cgraph_node::expand)过程实际上是执行了all_passes中的所有passes,这些pass中:

- 首先执行了一系列的GIMPLE_PASS(如优化)
- 然后通过pass "expand" 将gimple指令序列转化为rtl指令序列
- 之后执行了一系列RTL_PASS
- 最终通过RTL_PASS "final" 将函数的RTX指令序列输出为汇编代码

而本文主要记录在 **pass "expand" 中是如何将gimple指令序列转化为rtl指令序列的:**

一、pass_expand的基本流程

二、pass_expand-局部,临时变量的展开与空间分配

三、pass_expand-caller的参数传入和callee的参数接收

四、pass_expand-gimple指令序列的展开

五、pass_expand-init_block和exit_block的展开

六、pass_expand-边指令的插入和硬件寄存器初值的保存

一、pass_expand的基本流程

将gimple指令序列转换为rtl指令序列的pass为 pass_expand, **在此pass执行完成后当前函数的整个cfg中就不再存在gimple指令序列了,整个gimple指令序列全部转换为了rtl指令序列**, pass_expand::execute函数的大体流程如下:

```
1. /*
2.   每个函数都是通过这个pass，从 gimple转换为rtl形式的,此函数中的主要步骤包括:
3. 1. 变量展开:
4.     调用 expand_used_vars()对当前函数中所有的临时/局部/SSA_NAME变量进行分析并生成对应的RTX， 此RTX表达式代表在虚拟寄存器或堆栈中为对应变量预留(分配)的空间.
5. 2. 参数和返回值存储位置的确定:
6.     调用 expand_function_start(current_function_decl); 确定当前函数的参数和返回值的存储位置,
7.     * 此函数会按照AAPCS64标准,确定当前函数被调用时(作为callee),其参数应该来自caller的哪些硬件寄存器(如R0-R7)或栈内存偏移,这些信息均以rtx表达式的形式记录在c
8.     * 函数返回值也会按照AAPCS64标准确定其应该返回到哪里,通常是R0,此信息直接记录到RESULT_DECL节点的rtx表达式中
9. 3. 初始块的处理:
10.     调用 construct_init_block(); 创建初始块(init_block),其被插入到ENTRY_BB和 first_bb(函数体的第一个bb),在1,2阶段生成的所有rtx指令都插入到此bb中,这些指令
11.    真正执行前需要被执行的指令(包括mcount和栈溢出保护的代码)
12. 4. 基本块的展开:
13.     对函数体中除初始块之后的每个基本块的GIMPLE指令序列逐个展开为rtl指令序列(expand_gimple_basic_block),此过程可能导致新的基本块的产生
14. 5. 退出块处理:
15.     调用 construct_exit_block(); 创建退出块(exit_bb),其被插入到EXIT_BB的前面并接收EXIT_BB的所有边,所有的return指令最终都会跳转到exit_bb来统一处理,此bb中
16.    return_label标签指令(rtx(CODE_LABEL)),以及mov R0,...的将函数返回值保存到R0寄存器的指令.
17.     但需要注意的是,退出块并没有生成函数返回指令,其最终只是确保返回值被保存到R0.
18. 6. 其他处理包括:
19.    1) 将过程中所有边指令都发射到新的bb中,并将边关系调整为 e->src => bb => e->dest,即只有在此边的控制流转移过程中才执行此指令序列
20.    2) 确认边,bb和bb中指令的一致性:
21.       在之前的一些pass中,可能有插入/修改/删除指令而没有修复bb和边关系的情况,故这里再做一次统一的处理,以确保三者关系的一致.
22.       * 再次遍历所有bb,发现标签或控制流则将bb拆分为新的bb(find_many_sub_basic_blocks),防止前面某些pass插入了指令但没有拆分bb
23.       * 遍历所有bb的最后一条语句,若发现当前bb有死边则将其删除(如前面修改条件跳转为无条件跳转,但没有处理边关系,就会出现一条不可达的死边)
24.       * 优化并删除一些没有指令的bb(cleanup_cfg (CLEANUP_NO_INSN_DEL)),此过程会导致最终*.c.233r.expand与前面的分析结果差别较大
25.    3) 若整个expand过程中,用到了函数入口的某些硬件寄存器的初始值(如LR),则这些值需要被临时保存,以供指令序列中使用,emit_initial_value_sets 函数在最后会在函数入
26.       init_bb前再新建一个bb,并将这些保存硬件寄存器初值的指令发射到此bb中
27. */
28. //这里涉及SSA_NAME的部分先略过,后面单独总结
29. unsigned int pass_expand::execute (function *fun)
30. {
```

```
  basic_block bb, init_block;
  edge e;
  rtx_insn *var_seq, *var_ret_seq;

  emit_note (NOTE_INSN_DELETED);  /* 在全局的rtx_insn指令序列中(ctrl->emit.seq) 先发射一条 (note NOTE_INSN_DELETED)指令,每个函数的开始都要发射此指令 *

  start_sequence ();  /* 在ctrl->emit.seq中新开一个指令序列, 记录expand_used_vars过程中产生的新的指令序列(原有指令序列类似栈指针被保存起来) */
  /*
     此函数负责expand当前函数中所有的变量(统称为vars),包括所有SSA_NAME(cfun->gimple_df->ssa_names)和临时/局部变量(cfun->local_decls), 实际上就是为这些变
    生成代表其内存位置的rtx(MEM)或rtx(REG)表达式, 若:
     * 变量被分配在伪寄存器中,则会为此变量生成一个rtx(REG)表达式,此表达式内记录一个此函数内唯一的伪寄存器编号(ORIGINAL_REGNO),并在全局 regno_reg_rtx 数组
       中记录此函数使用了此伪寄存器
     * 变量被分配在函数栈中,则会为此变量生成一个rtx(MEM)表达式,此表达式通过栈基址+偏移的方式记录此变量在内存中的位置,如(mem (plus virtual_stack_vars_rtx,
       其中的offset是一个常量的rtx表达式,代表位此变量动态分配的栈偏移(在栈向低地址增长的情况下offset是个负数)
     最终:
     * 代表局部/临时变量位置的rtx表达式会被记录到DECL_RTL(var).rtl中
     * 代表SSA_NAME变量位置的rtx表达式会被记录到SA.partition_topseudo[i]中

     在此过程中:
     * 如果产生了要在此函数返回前执行的指令(收尾工作) ,则会通过返回值的方式返回到 var_ret_seq 指令序列中
     * 如果产生了在函数进入,函数体代码执行前就要执行的指令,则直接发射到当前的指令序列中,并最终记录到 var_seq指令序列中
  */
  var_ret_seq = expand_used_vars ();
  var_seq = get_insns ();    /* 获取当前指令序列中所有指令,实际上就是expand_used_vars发射的需要在函数体之前执行的指令(大多数情况下 var_ret_seq/var_seq都
  /* 关闭当前指令序列,同时恢复之前的指令序列[start_sequence, end_sequence]结束后,二者之间发射的指令序列无法通过get_insns获取,必须在end_sequence之前显示保
  end_sequence ();

  /*
    此函数根据AAPCS64标准确定:
     * 函数返回值的存储位置(通常是用rtx(REG)代表的R0硬件寄存器)
     * 函数的各个传入参数来自哪个硬件寄存器(R0~R7)或栈内存(也都是通过rtx(REG)表示)
    同时在函数栈/伪寄存器中为传入参数预留空间,并(在函数体指令序列之前)发射指令将传入参数复制到对应的函数栈/伪寄存器中
    最后此函数发射(note NOTE_INSN_FUNCTION_BEG)指令,代表后续指令属于函数体;若编译器指定了如-pg等参数,则在此后还会发射对_mcount函数的调用指令
    */
  expand_function_start (current_function_decl);
  /*
    parm_birth_insn在expand_function_start中指向 emit_note (NOTE_INSN_FUNCTION_BEG) 指令位置,
    var_seq指令序列(局部变量相关的指令序列)被插入到 expand_function_start 发射的传入参数保存指令序列之后,NOTE_INSN_FUNCTION_BEG指令之前(同样是_mcount之前
    */
  if (var_seq)
    {
      emit_insn_before (var_seq, parm_birth_insn);
      parm_birth_insn = var_seq;
    }

  /* 若当前是main函数, 且定义了INVOKE__main,则发起一个到 __main的调用, 以运行全局初始化 */
  if (DECL_NAME (current_function_decl) && MAIN_NAME_P (DECL_NAME (current_function_decl)) && DECL_FILE_SCOPE_P (current_function_decl))
    expand_main_function ();

  /* 为 stack_protect_guard 发射的指令序列 */
  if (crtl->stack_protect_guard && targetm.stack_protect_runtime_enabled_p ())
    stack_protect_prologue ();

  /* 将cfg_hooks修改为 rtl_cfg_hooks, 在各个不同阶段,修改控制流图需要修改的内容不同, 这里修改cfg_hooks, 以确保后续对控制流图的修改会调用到 rtl相关的hook函
  rtl_register_cfg_hooks ();

  /* 为此前已发射的代码构建一个新的初始化基本块(INIT_BB), 此基本块被插入到函数的入口基本块(ENTRY_BB)和第一个真正的代码基本块(FIRST_BB)之间(汇编代码顺序),
    插入后当前函数入口的流程就由 entry_bb => first_bb 变为 entry_bb => init_bb => first_bb
     当前 get_current_sequence 中已经发射的所有insns,都归于init_block中, 这些insns就是当前函数的初始化代码. */
  init_block = construct_init_block ();

  /* rtx_code_label是rtx格式的标签位置表达式, 在某指令expand过程中若引用到了尚未expand的bb,则会先为目标bb生成rtx(CODE_LABEL)并保存在此hash表中(若需要) */
  lab_rtx_for_bb = new hash_map<basic_block, rtx_code_label *>;

  /*
    从init_block->next_bb(也就是first_bb)开始,遍历当前函数中其余所有bb(遍历顺序是后续代码的生成顺序,遍历范围为[first_bb, EXIT_BB)),每个bb通过expand_gimp
   函数将其gimple指令序列expand为rtx指令序列.
    此函数返回后,当前bb中所有的gimple语句(bb->il.gimple.seq)都expand为rtl格式了,bb->il.gimple.seq被设置为空;
    当前bb的rtl指令序列链接在此函数的rtl指令序列中,作为当前函数rtl指令序列的一部分,而对于当前bb来说,属于其自身的rtl指令序列范围为[BB_HEAD,BB_END]
    var_ret_seq 是 expand_used_vars 的返回值, 代表在此函数执行完毕是否还有收尾指令,若有首尾指令,则任何bb中都禁止尾调用
  */
  FOR_BB_BETWEEN (bb, init_block->next_bb, EXIT_BLOCK_PTR_FOR_FN (fun), next_bb)
    bb = expand_gimple_basic_block (bb, var_ret_seq != NULL_RTX);

  /* 在所有的bb expand之后, 标记当前指令序列已经不再是ssa 模式了 */
  fun->gimple_df->in_ssa_p = false;

  /* 释放基本块的标签表达式映射hash, 这个只在前面expand_gimple_basic_block中使用, 之后不用了 */
  delete lab_rtx_for_bb;

  /*
    在EXIT_BB的代码顺序前插入一个新的bb(称为exit_bb),此bb接管了所有原本跳转到EXIT_BB的边,最终发起一个fallthru的边到EXIT_BB,同时将此函数的首尾指令发射到exi
    * 发射return_label(rtx(CODE_LABEL)), 之前所有return指令expand后都将跳转到此标签位置
    * 根据AAPCS64标准,发射如mov r0,...的指令将函数返回值保存到AAPCS64标准下的硬件寄存器(R0)
    但需要注意的是此函数并没有插入使当前函数返回其父函数的指令,函数返回指令(如ret)并非在pass_expand中发射的,而是在后续的pro_and_epilogue pass中发射的
    */
  construct_exit_block ();

  if (var_ret_seq)  /* 若在局部/临时/SSA_NAME变量expand时生成了需要在函数返回前执行的指令序列,则这些指令序列会被插入到exit_bb的return_label之后(NOTE BAS
    {
      rtx_insn *after = return_label;                                /* 获取return_label在指令序列中的位置 */
      rtx_insn *next = NEXT_INSN (after);
```

```
123.       if (next && NOTE_INSN_BASIC_BLOCK_P (next)) after = next;      /* 获取 note basic_block 之后的第一条指令 */
124.       emit_insn_after (var_ret_seq, after);                          /* 将 var_ret_seq 发射到其后 */
125.     }
126.
127.   rebuild_jump_labels (get_insns ());     /* 确认指令序列中哪些标签需要被保留 */
128.
129.   /*
130.      遍历所有bb的所有边,若发现了边指令的边(e->insns.r),则将此边的边指令序列插入到此边的 e->src => e->dest 控制流的转移过程中,此插入要确保
131.     不能影响src的其他后继和dest其他前驱的控制流. 通常这里会做个边分裂,并构建一个src => new_bb => dest的控制流转译,并将insns.r加入到new_bb中,
132.     以确保 new_bb只会在此控制流中执行.
133.   */
134.   FOR_BB_BETWEEN (bb, ENTRY_BLOCK_PTR_FOR_FN (fun), EXIT_BLOCK_PTR_FOR_FN (fun), next_bb)
135.     {
136.       for (ei = ei_start (bb->succs); (e = ei_safe_edge (ei)); )  /* 遍历当前bb的所有出边 */
137.       {
138.         /* 若此边中有指令序列,则这些指令序列是要被插入到此边的 src => dest 的指令流程中的 */
139.         if (e->insns.r)
140.         {
141.                   ......
142.           commit_one_edge_insertion (e);
143.         }
144.         else
145.           ei_next (&ei);
146.       }
147.     }
148.
149.   /* 遍历所有bb,若发现某bb的rtx指令序列除最后一条指令外中间还有有标签或控制流转移指令,则将其拆分. 正常来说应该是没有的,此过程是防止某些pass并没有很好的处理关
150.   auto_sbitmap blocks (last_basic_block_for_fn (fun));
151.   bitmap_ones (blocks);
152.   find_many_sub_basic_blocks (blocks);
153.
154.   purge_all_dead_edges ();    /* 遍历所有bb,通过判断指令序列来清除所有死边(根据bb的最后一条指令删除实际不可达的边,保证bb的边关系与指令关系的一致性) */
155.
156.   /* 整理可删除的bb和边,这里会删除很多无用的bb, 导致 x.c.233r.expand的结果和此过程分析看起来差别较大 */
157.   cleanup_cfg (CLEANUP_NO_INSN_DEL);
158.
159.   /*
160.      在代码中可能需要使用到函数入口时硬件寄存器的原始值,此函数负责在函数的入口发射指令(ENTRY_BB之后,init_bb之前再插入一个new_bb),将此函数执行过程中使用到的
161.     硬件寄存器的原始值复制到某个伪寄存器中(伪寄存器编号记录在hard_reg_initial_vals[x].pesudo_reg中),以确保即使函数执行过程中硬件寄存器已经被破坏,
162.     get_hard_reg_initial_val也可以通过pesudo_reg获取到硬件寄存器的原始值(如LR)
163.      故init_bb可能未必是此函数入口第一个有指令的bb,这取决于函数执行过程中是否有硬件寄存器需要暂存.
164.   */
165.   emit_initial_value_sets ();
166.
167.   /* 标记此函数已经被编译过了 */
168.   TREE_ASM_WRITTEN (current_function_decl) = 1;
169.
170.   /* 清空 return_label这个rtx(CODE_LABEL), 此标签在construct_exit_block中被插入到 exit_bb指令序列的第一条,在每条 greturn语句expand时都会跳转到此标签以表
171.   return_label = NULL;
172.   ...
173.   return 0;
174. }
```

二、pass_expand-局部,临时变量的展开与空间分配

在pass_expand大体流程中可知.当前函数的局部.临时变量以及SSA_NAME的展开都是通过expand_used_vars函数完成的:

```
1. /*
2.    此函数负责为当前函数的所有SSA_NAME,局部/临时变量生成代表其位置节点的rtx表达式,并将其记录到变量的DECL_RTL成员中,如果:
3.    * 变量因为绑定了硬件寄存器最终被分配到某硬件寄存器中,则其rtx表达式为 (reg num), 其中num是当前变量绑定的硬件寄存器编号
4.    * 变量被分配到伪寄存器中,则其rtx表达式同样为(reg num),但num是一个动态分配的伪寄存器编号,此表达式同时记录到全局寄存器数组的regno_reg_rtx[num]位置
5.    * 变量直接或延迟分配到函数栈中,则其rtx表达式为(mem (plus virtual_stack_vars_rtx offset)), 其中offset为当前变量在栈中基于virtual_stack_vars_rtx的偏移
6.      若栈向低地址方向增长,则此偏移为一个负数
7. */
8. //这里同样先忽略SSA_NAME的展开,后续统一介绍
9. static rtx_insn * expand_used_vars (void)
10. {
11.    tree var, outer_block = DECL_INITIAL (current_function_decl);    /* 获取函数体所在的那个tree_block */
12.   ......
13.   /* 这里忽略了判断哪些变量需要展开的代码,只简单描述此函数关键步骤(这里判断逻辑较复杂,可直接参考源码) */
14.   for (i = 0; i < SA.map->num_partitions; i++)
15.     {
16.       if (bitmap_bit_p (SA.partitions_for_parm_default_defs, i)) continue;
17.       tree var = partition_to_var (SA.map, i);    /* 获取每个SSA_NAME partition节点对应变量树节点  */
18.       expand_one_ssa_partition (var);    /* 实际上很多var节点(有SSA_NAME的) 都在这里expand 了,不考虑ssa_name的情况下此函数和expand_one_var类似,这里先不
19.     }
20.
21.   /* 获取临时变量和显式声明的局部变量个数 */
22.   len = vec_safe_length (cfun->local_decls);
23.
24.   /* 这里主要负责expand gcc生成的临时变量 */
25.   FOR_EACH_LOCAL_DECL (cfun, i, var)
26.     {
27.       bool expand_now = false;
28.       if(...) expand_now = true;    /* 对于前面判断需要展开的变量,这里设置expand_now = true */
29.       .......
30.       if (expand_now) expand_one_var (var, true, true);
```

```
      ......
    }

  expand_used_vars_for_block (outer_block, true);    /* 这里主要调用 expand_one_var expand源码中显示声明的局部变量 */

  /* 栈溢出保护在这里插入了代码 */
  switch (flag_stack_protect) {
      case ...:
      create_stack_guard (); break;
  }

  /* 在expand_one_var时,很多栈变量都不是立即展开的,而是放到这里统一展开,此函数对要统一分配的变量先进行排序以优化存储空间 */
  if (stack_vars_num > 0) partition_stack_vars ();

  /* 对排序后的要分配到栈中的变量(记录在stack_vars中)统一展开,实际上是生成代表变量在栈空间位置的rtx表达式,并将其记录到变量的DECL_RTL节点中 */
  if (stack_vars_num > 0) {
      struct stack_vars_data data;
      /* 最终生成的rtx表达式为 (mem (plus virtual_stack_vars_rtx offset)), 在栈向低地址方向增长时,offset为一个负数, 每个变量的offset是递减的 */
      expand_stack_vars (NULL, &data);
  }

  return var_end_seq;      /* 返回此函数中生成的需要在函数返回前执行的指令序列*/
}
```

其中主要的变量展开函数为 expand_one_var:

```
static poly_uint64 expand_one_var (tree var, bool toplevel, bool really_expand)
{
  if (TREE_TYPE (var) != error_mark_node && VAR_P (var))    /* 静态或全局变量是在output_in_order => varpool_node::assemble_decl 中展开的,这里直接返回
      if (is_global_var (var)) return 0;

  if (!VAR_P (var) && TREE_CODE (origvar) != SSA_NAME);
  else if (DECL_EXTERNAL (var));                             /* 外部声明不处理 和静态变量不在这里展开 */
  else if (TREE_STATIC (var));
  else if ...
  else if (VAR_P (var) && DECL_HARD_REGISTER (var))        /* 若当前边变量是显式指定了的某个硬件寄存器,则直接基于硬件寄存器展开 */
      /* 变量绑定了硬件寄存器(如R0),则代表其位置的rtx表达式就相当于已经确认了(reg r0),此函数并未动态为其分配存储位置,其expand的rtx表达式只是用来表达已确认的
      if (really_expand) expand_one_hard_reg_var (var);
  else if (use_register_for_decl (var))                    /* 如果当前变量可以通过寄存器展开(如指定了register关键字,或开启优化时大部分没有副作用的变量),
      /* 为变量var新分配一个全局为寄存器(rtx(REG),伪寄存器编号为reg_rtx_no),并将其记录在全局寄存器数组regno_reg_rtx[reg_rtx_no]和在此变量的DECL_RTL(var)
      if (really_expand) expand_one_register_var (origvar);
  else if (defer_stack_allocation (var, toplevel))
    /* defer_stack_allocation 返回false 则此变量可合并分配(否则此变量就必须被立即分配),可合并分配的则直接增加到 stack_vars 队列中即可,合并分配的好处是合并
    add_stack_var (origvar);
  else  {                                                  /* 不能在寄存器中分配,也不能合并分配的,则在栈上立即为其分配空间 */
      if (really_expand) expand_one_stack_var (origvar);
      return size;                                         /* 若变量被立即分配在栈上,则此函数返回为此变量分配了的空间大小 */
  }
  return 0;                                                /* 没有立即分配的情况都直接返回 0 */
}
```

1) 绑定了硬件寄存器的变量通过函数expand_one_hard_reg_var来展开:

```
/* 此函数负责expand一个绑定了硬件寄存器的变量, 对于绑定了硬件寄存器的变量,此变量的位置信息相当于已经确定了,此函数只是调用 make_decl_rtl,
 来将此关系以(reg num)的形式记录下来,此rtx表达式会被保存到DECL_RTL(var)中,后续在rtl指令序列中可通过DECL_RTL(var)引用变量var的位置信息 */
static void expand_one_hard_reg_var (tree var)
{
  rest_of_decl_compilation (var, 0, 0);
}

//这里只记录了此处使用到的代码逻辑
void rest_of_decl_compilation (tree decl, int top_level, int at_end)
{
  ......
  /* 对于标记了register关键字,并制定了具体硬件寄存器的声明节点(如 int register X0 asm("x0");)生成rtx表达式 */
  if (HAS_DECL_ASSEMBLER_NAME_P (decl) && DECL_ASSEMBLER_NAME_SET_P (decl) && DECL_REGISTER (decl))
    make_decl_rtl (decl);
  ......
}

/*
   此函数负责为全局的函数声明(FUNCTION_DECL),static/public/external的变量(全局变量与声明,以及所有需要分配全局空间的变量)
 以及绑定硬件寄存器的变量生成代表其位置的rtl表达式,主要分为两种情况:
  1) 对于全局的函数声明(FUNCTION_DECL)和static/public/external的变量:
     此函数获取其汇编名(此类变量都有汇编名和全局符号名),并生成一个mem (symbol_ref name)表达式记录在变量的DECL_RTL(decl)中
  2) 对于绑定寄存器的变量:
     此函数将其汇编名转换为对应的硬件寄存器编号,并生成一个 (reg number)表达式记录此变量绑定的硬件寄存器
   此函数的参数只有一个decl节点,而通过此函数可通过此函数生成代表decl位置的rtx表达式,那么这些decl的位置实际上必须是提前确定的,
  故对于普通的局部变量/参数声明/返回值/标签节点来说,由于其位置是在栈中生成的,故肯定不能通过此函数来确定位置(而是要通过
  expand_one_register_var/expand_one_stack_var 来确定其位置.
*/
void make_decl_rtl (tree decl)
{
  const char *name = 0;
  int reg_number;
  tree id;
```

```
  rtx x;

  /* 此函数通常只处理FUNCTION_DECL, 或static/public/external/register的变量节点 */
  gcc_assert (TREE_CODE (decl) != PARM_DECL && TREE_CODE (decl) != RESULT_DECL);
  gcc_assert (TREE_CODE (decl) != TYPE_DECL && TREE_CODE (decl) != LABEL_DECL);
  gcc_assert (!VAR_P (decl) || TREE_STATIC (decl) || TREE_PUBLIC (decl) || DECL_EXTERNAL (decl) || DECL_REGISTER (decl));

  /* 若当前声明节点已经有rtx表达式了,则此函数直接返回 */
  if (DECL_RTL_SET_P (decl)) { ......; return; }

    ......

  /* 否则先获取当前节点对应的汇编名字符串, 正常全局符号(函数声明,静态/外部/public的变量)都会有汇编名,
     指定了特定硬件寄存器的局部变量节点也会有对应的汇编名(如int register tt asm("x0");), 后续代码也主要处理这两种情况 */
  id = DECL_ASSEMBLER_NAME (decl);
  name = IDENTIFIER_POINTER (id);

  /* 指定了特定硬件寄存器的局部变量的汇编名都是*开头的(如上面tt的汇编名就位 "*x0"), 这里判断所有非函数声明的寄存器变量如果不是*开头的则报错 */
  if (name[0] != '*' && TREE_CODE (decl) != FUNCTION_DECL && DECL_REGISTER (decl))
      error ("register name not specified for %q+D", decl);
  else if (TREE_CODE (decl) != FUNCTION_DECL && DECL_REGISTER (decl))     /* 对于指定了register关键字的,以"*"开头的非函数声明节点,即为一个绑定了硬件寄
    {
      const char *asmspec = name+1;              /* 获取此变量汇编名对应的寄存器名,这里获取的如"x0" */
      machine_mode mode = DECL_MODE (decl);
      reg_number = decode_reg_name (asmspec);    /* 解析此寄存器名对应的寄存器编号("x0"对应0) */
      /* 创建一个rtx(REG)表达式,代表字符串中的硬件寄存器,由于默认认为所有硬件寄存器在函数内都会使用(都已经提前加入了当前函数的全局寄存器数组 regno_reg_rtx [
        故这里新创建的rtx(REG)不用也没法放到全局寄存器数组中 */
      SET_DECL_RTL (decl, gen_raw_REG (mode, reg_number));
      ORIGINAL_REGNO (DECL_RTL (decl)) = reg_number;
      REG_USERVAR_P (DECL_RTL (decl)) = 1;      /* 标记此flag的 rtx(REG)直接对应到源码中用户声明的某个变量 */
      return;                                   /* 对指定硬件寄存器编号的变量这里就直接返回了 */
    }

  /* 对于FUNCTION_DECL或static/public/external的变量节点则从这里开始处理 */

  /* 正常未初始化的全局变量都是要放到COMMON段的, 但对于指定了section的变量(DECL_SECTION_NAME (decl) != NULL)则不能放到common段 */
  if (VAR_P (decl) && (TREE_STATIC (decl) || DECL_EXTERNAL (decl))
      && DECL_SECTION_NAME (decl) != NULL && DECL_INITIAL (decl) == NULL_TREE && DECL_COMMON (decl))
    DECL_COMMON (decl) = 0;

  /* 弱符号同样也不妨到COMMON段 */
  if (VAR_P (decl) && DECL_WEAK (decl)) DECL_COMMON (decl) = 0;

  machine_mode address_mode = Pmode;
  x = gen_rtx_SYMBOL_REF (address_mode, name);    /* 为符号name创建一个符号标签rtx(SYMBOL_REF) 节点 */
  SYMBOL_REF_WEAK (x) = DECL_WEAK (decl);
  SET_SYMBOL_REF_DECL (x, decl);                  /* 设置符号标签对应的声明节点 */
  x = gen_rtx_MEM (DECL_MODE (decl), x);          /* 构建一个 mem (symbol_ref name) 节点,代表对符号name的引用 */
  SET_DECL_RTL (decl, x);    /* 对于FUNCTION_DECL或static/public/external这样的全局变量节点, 其DECL_RTL中记录的是此节点的符号引用表达式 */
}
```

2) 没绑定硬件寄存器,但可以分配到寄存器中的变量(如指定了register关键字,或开启优化时大部分没有副作用的变量)则通过expand_on_register_var函数展开:

```
/* 此函数负责为变量var新分配一个全局为寄存器(rtx(REG),伪寄存器编号为reg_rtx_no),并将其记录在当前函数的全局寄存器数组regno_reg_rtx[reg_rtx_no]和在此变量的D
   若变量var为指针类型,或是用户源码中的变量,则在此rtx(REG)表达式中也要同时做对应标记 */
static void expand_one_register_var (tree var)
{
  /* 若var是个SSA_NAME,则这里直接返回其rtx节点 */
  ......
  tree decl = var;                      /* 到这里说明当前var非SSA_NAME,此时的var应该是个变量的声明节点 */
  tree type = TREE_TYPE (decl);         /* 获取变量的类型节点 */

  machine_mode reg_mode = promote_decl_mode (decl, NULL);    /* 根据声明节点确定其提升后的机器模式 */
  /* 新分配一个伪寄存器表达式rtx(REG)作为此变量的存储空间,此伪寄存器mode为reg_mode, 在当前函数内的全局编号为reg_rtx_no,
     并将表达式的指针记录到全局寄存器数组regno_reg_rtx[reg_rtx_no]中,代表当前被分析函数又使用了一个新的伪寄存器*/
  rtx x = gen_reg_rtx (reg_mode);

  set_rtl (var, x);             /* 将代表变量存储空间的伪寄存器表达式rtx(REG)记录到此变量的DECl_RTL中 */

  if (!DECL_ARTIFICIAL (decl))    /* 对于非编译器生成的变量,则在此伪寄存器中标记其代表的是用户源码中的某个变量 */
    mark_user_reg (x);

  if (POINTER_TYPE_P (type))      /* 若变量的类型是指针类型,则在此伪寄存器中标记其中记录了一个指针 */
    mark_reg_pointer (x, get_pointer_alignment (var));
}
```

3) 若此变量不能被分配到寄存器中,则要先看其是否需要立刻展开(defer_stack_allocation),对于不需要立即展开的变量,则通过**add_stack_var先保存到全局数组stack_vars中，后续在expand_user_vars的最后通过expand_stack_vars函数一并展开**:

```
/* stack_vars数组中的的每一个元素都代表一个后续要在栈上展开的变量的信息,通过 add_stack_var 函数可将一个要延迟分配的变量增加到此数组中,
   并在 expand_used_vars的最后通过 expand_stack_vars统一分配栈空间 */
static struct stack_var *stack_vars;
struct stack_var
```

```
5. {
6.   tree decl;                /* 要存储到栈上的变量的声明节点 */
7.   poly_uint64 size;         /* 此变量需要占用栈空间的大小 */
8.   unsigned int alignb;      /* 此变量的对齐要求 */
9.   size_t representative;    /* The partition representative.  */
10.  size_t next;              /* The next stack variable in the partition, or EOC.  */
11.  bitmap conflicts;         /* The numbers of conflicting stack variables.  */
12. };
13.
14. /* 此函数负责在全局的 stack_vars数组中增加当前变量的信息(声明节点,需要分配空间大小,对齐粒度等), stack_vars中的变量在后续  expand_stack_vars中会统一在当前函
15. static void add_stack_var (tree decl)
16. {
17.   struct stack_var *v;
18.   if (stack_vars_num >= stack_vars_alloc)      /* 若当前stack_vars中已经有的存储空间不够了,则重新分配空间 */
19.   {
20.       if (stack_vars_alloc) stack_vars_alloc = stack_vars_alloc * 3 / 2;
21.       else stack_vars_alloc = 32;
22.       stack_vars = XRESIZEVEC (struct stack_var, stack_vars, stack_vars_alloc);
23.   }
24.
25.   v = &stack_vars[stack_vars_num];            /* 获取 stack_vars中下一个变量信息的存储位置 */
26.   v->decl = decl;                             /* 记录要延迟分配的变量的声明节点 */
27.   tree size = TREE_CODE (decl) == SSA_NAME ? TYPE_SIZE_UNIT (TREE_TYPE (decl)) : DECL_SIZE_UNIT (decl);
28.   v->size = tree_to_poly_uint64 (size);       /* 记录此变量占用栈空间的大小 */
29.   v->alignb = align_local_variable (decl);    /* 变量的对齐要求 */
30.   set_rtl (decl, pc_rtx);                     /* 需要延迟展开的先都设置为 pc_rtx,后续统一修正 */
31.   stack_vars_num++;                           /* 更新stack_vars已记录的变量信息的个数 */
32. }
33.
34. /* 此函数在 expand_used_vars的最后执行,其负责为 stack_vars中记录的所有变量统一计算并分配栈空间, 并将生成的rtx表达式记录到变量的DECL_RTL节点中,
35.   最终生成的rtx表达式为  (mem (plus virtual_stack_vars_rtx offset)), 在栈向低地址方向增长时,offset为一个负数, 每个变量的offset是递减的 */
36. static void expand_stack_vars (bool (*pred) (size_t), struct stack_vars_data *data)
37. {
38.   size_t si, i, j, n = stack_vars_num;    /* 获取 stack_vars中变量数量 */
39.   tree decl;
40.   .......
41.
42.   for (si = 0; si < n; ++si)              /* 遍历 stack_vars中所有变量 */
43.     {
44.       poly_int64 offset;
45.       rtx base;
46.       /* stack_vars_sorted中记录的也是stack_vars的index,只不过其按照大小顺序重新排序的,排序是在 expand_used_vars => partition_stack_vars 中完成的 */
47.       i = stack_vars_sorted[si];
48.       .......
49.       decl = stack_vars[i].decl;          /* 获取当前要expand到栈上的变量的声明节点 */
50.       /* 对于SSA_NAME,其rtx表达式不能为空; 对于变量节点,其rtx必须是 pc_rtx(这是之前在add_vars中设置的 */
51.       if (TREE_CODE (decl) == SSA_NAME ? SA.partition_to_pseudo[var_to_partition (SA.map, decl)] != NULL_RTX: DECL_RTL (decl) != pc_rtx)
52.         continue;
53.
54.       alignb = stack_vars[i].alignb;      /* 获取变量对齐粒度 */
55.       base = virtual_stack_vars_rtx;      /* 变量在栈中的空间实际上是一个基于 virtual_stack_vars_rtx + offset的表达式, virtual_stack_vars_rtx作为变量
56.       ......
57.       /* 根据size,alignb,计算出一个offset并返回;此offset是相对virtual_stack_vars_rtx的一个偏移,代表当前变量在当前函数栈中的存储位置
58.          (全局变量frame_offset记录之前已经分配的offset,故对于每次变量分配,offset都是不同的,在栈向低地址方向增长时,此offset为一个负数)
59.       offset = alloc_stack_frame_space (stack_vars[i].size, alignb);
60.            ......
61.       /* 生成一个 (mem (plus virtual_stack_vars_rtx offset)) 表达式,代表此变量在函数栈中的位置, 并将其记录到变量的 DECL_RTL 中 */
62.       expand_one_stack_var_at (stack_vars[j].decl, base, base_align, offset);
63.     }
64. }
```

4）若变量需要立即展开到栈中,则调用expand_one_stack_var直接对变量展开:

```
1. /* 此函数的逻辑和expand_one_stack_var_at基本相同,其为变量var计算并分配栈空间,并将生成的rtx表达式记录到变量的DECL_RTL节点中,
2.   最终生成的rtx表达式为  (mem (plus virtual_stack_vars_rtx offset)),在栈向低地址方向增长时,offset为一个负数, 每个变量的offset是递减的 */
3. static void expand_one_stack_var (tree var)
4. {
5.   if (TREE_CODE (var) == SSA_NAME)
6.     {
7.       .......
8.       return;
9.     }
10.  return expand_one_stack_var_1 (var);
11. }
12.
13. static void expand_one_stack_var_1 (tree var)
14. {
15.   poly_uint64 size;
16.   poly_int64 offset;
17.   unsigned byte_align;
18.
19.   size = tree_to_poly_uint64 (DECL_SIZE_UNIT (var));    /* 确定变量占用空间大小 */
20.   byte_align = align_local_variable (var);              /* 确定变量对齐粒度 */
21.
22.   offset = alloc_stack_frame_space (size, byte_align);    /* 确定变量在当前函数栈中的偏移(栈向低地址增长时是基于virtual_stack_vars_rtx的一个负数 */
23.   /* 生成一个 (mem (plus virtual_stack_vars_rtx offset)) 表达式,代表此变量在函数栈中的位置, 并将其记录到变量的 DECL_RTL 中 */
24.   expand_one_stack_var_at (var, virtual_stack_vars_rtx, crtl->max_used_stack_slot_alignment, offset);
25. }
```

三、pass_expand-caller的参数传入和callee的参数接收

在函数调用中,调用函数称为caller, 被调用函数称为callee,一个函数调用的过程可描述为:

1. caller为callee准备传入参数
2. caller中的指令需要按照函数调用标准(如aarch64中的AAPCS64)将传入参数复制到标准中规定的硬件寄存器(如R0-R7)即栈上(如>8个参数)
3. caller跳转到callee执行被调用函数
4. callee中的指令需要按照函数调用标准,将传入参数(尤其是寄存器传入参数)暂存到伪寄存器或函数栈中,以确保这些硬件寄存器在callee后续指令中可用
5. callee函数体的执行
6. callee中的指令需要按照函数调用标准,将函数返回值保存到特定位置(若函数有返回值的话, 如R0寄存器中),然后执行函数返回(ret,这一步不是在pass_expand中实现的,而是在pro_and_epilogue中实现的)
7. caller中的指令需要在返回后接收函数的返回值(若在caller中函数返回值需要保存在某变量中)

由上述流程可知,整个函数的调用过程caller和callee都需要根据函数调用标准(AAPCS64)来确定传入参数的来源,并且都需要发射指令来完成此过程:

- **此过程中caller中的一系列操作,是在pass_expand分析caller函数时解析到gcall指令时,调用expand_all函数完成的**
- **此过程中callee中的一系列操作,是在pass_expand分析callee函数时通过调用expand_function_start完成的**

故整个函数的调用是需要分成这两个部分来分析的

1. caller函数的参数传入过程

在pass_expand解析gimple指令序列时(见后续的expand_gimple_basic_block),若解析到了gcall指令,则会将此gcall指令重新构建成一个CALL_EXPR表达式,并调用expand_call函数完成此函数调用对应指令的生成:

```
/*
   此函数传入的是exp是一个CALL_EXPR(由gcall转换而来的),记录此次调用的所有信息; 若caller调用后保存了函数的返回值,则target为保存返回值的变量的rtx节点,否则tar
   此函数为此函数调用的实参发射指令,将这些实参保存到AAPCS64标准中其对应的硬件寄存器或传入参数栈位置(caller的栈);发射函数调用指令;并最终发射指令将函数返回值复
   1) 按照AAPCS64标准确定各个实参对应的传入位置(如硬件寄存器R0-R7,或某个基于virtual_outgoing_args_rtx的栈偏移)
   2) 分别为硬件寄存器和栈传入参数发射指令,此指令负责将实参赋值到对应的传入参数位置
   3) 发射函数调用指令,此指令中会按照AAPCS64标准,将函数返回值保存到硬件寄存器R0中(若需要)
   4) 发射指令将函数返回值从硬件寄存器R0复制到变量target中,并返回target
*/
rtx expand_call (tree exp, rtx target, int ignore)
{
  ......
  tree addr = CALL_EXPR_FN (exp);    /* 获取代表调用函数的指针表达式节点(ADDR_EXPR)*/
  fndecl = get_callee_fndecl (exp);  /* 这获取函数指针节点中记录的被调用函数节点(若被调用函数是确定的则可获取,若被调用函数是未决的则返回空) */

  rettype = TREE_TYPE (exp);         /* 这是 CALL_EXPR 的类型,也就是函数返回值的类型 */
  funtype = TREE_TYPE (addr);        /* 获取ADDR_EXPR的类型节点 */
  funtype = TREE_TYPE (funtype);     /* 获取被调用函数的类型节点(FUNCTION_TYPE) 节点 */
  ......
  type_arg_types = TYPE_ARG_TYPES (funtype);    /* 获取函数的参数类型链表 */

  /* 计算传入的实参的个数(在gcc C语言中由于参数不能有默认值, 故函数的实参个数是>=形参个数的,可能出现>的情况是不定参数) */
  int num_actuals = call_expr_nargs (exp) + num_complex_actuals + structure_value_addr_parm;
  n_named_args = num_actuals;

  /* 初始化args_so_far_v结构体, 此结构体后续负责记录按照AAPSC64已经分配的传入参数信息 */
  INIT_CUMULATIVE_ARGS (args_so_far_v, funtype, NULL_RTX, fndecl, n_named_args);
  /* pack_cumulative_args 只是通过一个结构体对args_so_far_v进行了一层包裹 */
  args_so_far = pack_cumulative_args (&args_so_far_v);

  /* 分配一个arg_data类型的数组,此数组用来存储每个实参的信息, 其个逆序,args[0]记录最后一个参数的信息, args[num_artuals] 记录第一个参数的信息 */
  args = XCNEWVEC (struct arg_data, num_actuals);

  /*
     此函数根据AAPCS64标准,计算出每个实参应该如何传给callee:
     * 若参数通过硬件寄存器传入则args[i].reg为一个非空的rtx(REG)表达式,代表此传入参数来自哪个硬件寄存器
     * 若参数通过caller栈传入,则args[i].reg为空,args[i].locate记录此参数在caller栈上基于virtual_outgoing_args_rtx的偏移
     args_size返回此次调用的实参总共需要在栈中基于virtual_outgoing_args_rtx分配多少栈偏移
  */
  initialize_argument_information (num_actuals, args, &args_size, n_named_args, exp, structure_value_addr_value, fndecl, fntype,
                   args_so_far, reg_parm_stack_space, &old_stack_level, &old_pending_adj, &must_preallocate, &flags, &try_tail_call, CALL_FROM_TH

  /* 若根据AAPCS64标准,若有实参需要被分配到函数栈上,则这里must_preallcate设置为1 */
  must_preallocate = finalize_must_preallocate (must_preallocate, num_actuals, args, &args_size);

  ......
  {
      rtx_insn *insns, *before_call, *after_args;
      rtx next_arg_reg;

      start_sequence ();                /* 新起一个指令序列,用来保存此CALL_EXPR导致生成的rtx指令 */

      if (must_preallocate) {
          ......
```

```
          argblock = virtual_outgoing_args_rtx;    /* 若给callee的传入参数有需要通过栈传入的,则argblock 指向传入参数的栈基地址 */
      }
      /* 遍历所有实参,对于要通过函数栈传入的参数(args[i].regs==0的参数),为其生成一个 rtx (mem (plus argblock offset))表达式此参数在函数传入栈中的位置,
        此表达式最终记录到此参数的args[i].stack中 (实际生成的表达式通常为(mem (plus virtual_outgoing_args_rtx args[i].locate.offset))) */
      compute_argument_addresses (args, argblock, num_actuals);

      /* 前面 initialize_argument_information/compute_argument_addresses 分别确定了硬件寄存器参数实参的存储位置(args[i].regs)和栈参数的存储位置(args[i
        而此函数则负责expand 硬件寄存器参数对应的实参(args[i].tree_value), 并最终将代表此实参值的rtx表达式最终被记录到 args[i].value中,只要此过程中有寄存
        处理了,则reg_parm_seen就返回1 */
      precompute_register_parameters (num_actuals, args, &reg_parm_seen);

      /* 若fndecl非空(被调用函数是确定的),则则返回代表被调用函数的符号标签表达式rtx(symbol_ref), 否则expand addr表达式,并返回代表被调用函数地址的rtx表达式
      funexp = rtx_for_function_call (fndecl, addr);

      /* 对于AAPCS64标准中需要通过函数栈传入的参数,通过store_on_arg发射指令将此实参(args[i].value)复制到对应的函数栈中 */
      for (i = 0; i < num_actuals; i++) {
        if (args[i].reg == 0 || args[i].pass_on_stack) {
          ......
          /* 发射指令,将caller实参保存到其对应的传入栈位置,对于每个实参发射的伪代码如: *(argblock + args[i].locate.offset) = args[i].value; */
          store_one_arg (&args[i], argblock, flags, adjusted_args_size.var != 0, reg_parm_stack_space);
        }
        ......
      }

      /* 确定保存返回值的硬件寄存器的表达式,在AAPCS64标准中此表达式为rtx(REG),对应寄存器为R0*/
      valreg = 0;
      if (TYPE_MODE (rettype) != VOIDmode && ! structure_value_addr) {
        ......
        valreg = hard_function_value (rettype, fndecl, fntype, (pass == 0));
      }

      after_args = get_last_insn ();    /* after_args 指向已发射的最后一条将实参复制到对应传入栈的指令 */

      /* 发射指令将caller所有寄存器传入参数复制其对应硬件寄存器中(栈参数赋值指令在前面store_one_arg中已经发射了) */
      load_register_parameters (args, num_actuals, &call_fusage, flags, pass == 0, &sibcall_failure);

           before_call = get_last_insn ();   /* before_call 指向已发射的最后一条将实参复制到对应硬件寄存器的指令 */

      /* 向指令序列发射对目标函数的调用,funexp是此函数中代表被调用函数地址的rtx表达式,对应确定的函数调用,其为一个符号标签(symbol_ref)节点.
        若calle函数有返回值,且caller需要保存返回值(valreg非空),则此处发射的指令会同时将函数返回值复制到valreg对应的rtx表达式节点中(如代表硬件寄存器R0),如:
        (call_insn 11 10 0 (parallel [
                (set (reg:SI 0 x0)                                          //函数调用的返回值被复制到R0寄存器
                     (call (mem:DI (symbol_ref:DI ("funx1") ...) )))  ))          //被调用函数确定的情况下,call命令直接调用被调用函数的符号标签
      */
      emit_call_1 (funexp, ..., valreg, ...);

      if ((flags & ECF_NORETURN) || pass == 0) {        /* 若函数永远不返回,则需要在call指令后面插入barrier指令 */
          rtx_insn *last = get_last_insn ();
          emit_barrier_after (last);
      }

     /* 处理函数的返回值,前面caller发起的函数调用只是按照AAPCS64标准将函数返回值复制到了硬件寄存器R0中, 这里判断caller的调用中是否通过某变量接收了函数返回值(
       来决定是否要再发射指令将硬件寄存器R0中的返回值复制到对应变量中 */
     if (TYPE_MODE (rettype) == VOIDmode || ignore)    /* 若CALL_EXPR最终不需要通过变量保存被调用函数的值,或被调用函数的返回值是void,则最终target设置为
       target = const0_rtx;
     else if(...) ......
     else if (target && GET_MODE (target) == TYPE_MODE (rettype) && GET_MODE (target) == GET_MODE (valreg))
        emit_move_insn (target, valreg);    /* 若target非空,则CALL_EXPR需要将返回值保存到变量target中,此时在call指令之后需要发射指令将函数返回值 从 valr
     else ...;

     insns = get_insns ();                /* insns记录此函数中发射的所有指令序列 */
     end_sequence ();                     /* pop当前指令序列 */
     .......
     normal_call_insns = insns;           /* 获取此函数发射的所有指令序列 */
   }

   emit_insn (normal_call_insns);         /* 将此指令序列重新发射到当前指令序列中 */
   return target;                         /* 返回此CALL_EXPR表达式最终的返回结果 */
}
```

2. callee函数的参数解析过程

在caller函数中,需要通过遍历指令序列的方式来发现每一条gcall函数调用指令并对其调用expand_call函数进行分析. 而对于一个函数来说,当其被执行到时,就是作为callee出现时候, 故对callee函数的参数解析过程的分析,是在pass_expand expand所有变量后就直接调用expand_function_start函数完成的:

```
/*
   此函数主要操作为:
   1. 为返回标签(return_label)生成rtx(CODE_LABEL)节点,作为后续greturn语句expand时的跳转位置
   2. 为函数返回值(RESULT_DECL)树节点生成rtx(REG)伪寄存器节点代表其存储位置
   3. 调用assign_parms函数完成:
      1) 按照AAPCS64标准,根据当前函数(作为callee)的每个形参(parm),确定其每个传入参数的来源(硬件寄存器或栈空间,记录在parm.incoming_rtl中)
      2) 为每个传入参数在当前函数(callee)栈中(局部变量后面)分配存储空间(或分配伪寄存器,取决于编译选项和参数类型),并记录在DECL_RTL(parm)中
      3) 向指令序列中发射指令(若需要),此指令负责将传入参数复制到当前函数的栈空间(mov parm.incoming_rtl DECL_RTL(parm))
      4) 根据AAPCS64标准,确定当前函数返回值存储位置(如R0寄存器),并将此rtx(REG)表达式记录到 ctrl->result中,但返回值设置到R0的操作并非此函数发出的(而是const
   4. 发射 (NOTE NOTE_INSN_FUNCTION_BEG) 指令代表参数expand的结束,以及函数体指令序列解析的开始
```

```
   5. 若gcc开启了-pg编译选项,则发起对_mocunt函数的调用
*/
void expand_function_start (tree subr)
{
  /* crtl->profile决定是否要对当前函数插mcount桩, 若gcc编译选项指定了 -pg 且当前函数没有指定 no_instrument_function 属性,则需要对当前函数插 mcount桩 */
  crtl->profile = (profile_flag && ! DECL_NO_INSTRUMENT_FUNCTION_ENTRY_EXIT (subr));

  /* 构建一个 rtx(CODE_LABEL) 表达式, 作为函数返回前(greturn语句expand时候)的跳转位置, 此标签位置对应的rtx指令会执行函数的收尾工作,
    此收尾指令在pass_expand后续的construct_exit_block函数中发射 */
  return_label = gen_label_rtx ();

  /* 获取并展开函数的返回值树节点(RESULT_DECL),实际上是此返回值树节点分配了代表其存储空间的rtx表达式(内存或伪寄存器) */
  tree res = DECL_RESULT (subr);
  if (aggregate_value_p (res, subr)) {                /* 对于聚合类型的返回值节点,则构建一个rtx(MEM)代表返回值存储位置 */
      ......
      x = gen_rtx_MEM (DECL_MODE (res), x);
      set_parm_rtl (res, x);
    }
  else if (DECL_MODE (res) == VOIDmode)               /* 如函数不需要返回值,则直接设置返回值节点的 rtx为 NULL_RTX */
    set_parm_rtl (res, NULL_RTX);
  else {                                              /* 对于非聚合类(返回值可以直接存储在寄存器中返回),则构建一个伪寄存器rtx(REG)表达式作为返回值节点的
      tree return_type = TREE_TYPE (res);
      ......
      /* 获取AAPCS64标准中函数返回值存储到的硬件寄存器, 获取此寄存器的作用是为了获取其机器模式 */
      rtx hard_reg = hard_function_value (return_type, subr, 0, 1);
      /* 根据AAPCS64标准中返回值寄存器的机器模式,为函数返回值节点(RESULT_DECL)创建代表其存储位置的伪寄存器表达式 */
      set_parm_rtl (res, gen_reg_rtx (GET_MODE (hard_reg)));
      DECL_REGISTER (res) = 1;                       /* 标记返回值节点最终分配到了寄存器中 */
   }

  /* 根据AAPCS64标准和函数形参链表,确认每个传入参数的来源(硬件寄存器或caller的栈空间)并将此信息记录到每个参数的parm->incoming_rtl中,
     同时在callee为这些参数分配临时栈(局部变量之后)/全局伪寄存器存储空间,并将此信息记录到每个参数的DECL_RTL(parm)中,
     最终向指令序列发射指令(若需),将传入参数从 parm->incoming_rtl复制到 DECL_RTL(parm)中,
     并确定此函数返回值的硬件寄存器(R0,记录到ctrl->result中),但此函数并未发射将返回值保存到R0的指令(此指令是在construct_exit_block中发射的) */
  assign_parms (subr);

  emit_note (NOTE_INSN_FUNCTION_BEG);    /* 发射一条标记语句,代表此后发射的指令来自此函数的函数体 */

  parm_birth_insn = get_last_insn ();    /* 当前指令之前的指令是为函数参数传递生成的指令 */

  if (crtl->profile)
#ifdef PROFILE_HOOK
      /* 若开启了-pg选线,则这里发起一个(call_insn (call (mem (symbol_ref:DI ("_mcount"))) ... 指令,在函数体解析之前先发起对_mcount函数的调用 */
      PROFILE_HOOK (current_function_funcdef_no);
#endif

  /* 如果指定了-stack-check,则这里插入一个note节点来标记stack-check的检查点以供后续使用 */
  if (flag_stack_check == GENERIC_STACK_CHECK)
    stack_check_probe_note = emit_note (NOTE_INSN_DELETED);
}
```

其中函数assign_parms主要内容如下:

```
/* 此函数负责:
   1) 按照AAPCS64标准,根据当前函数(作为callee)的每个形参(parm),确定其每个传入参数的来源(硬件寄存器或栈空间,记录在parm.incoming_rtl中)
   2) 为每个传入参数在当前函数(callee)栈中(局部变量后面)分配存储空间(或分配伪寄存器,取决于编译选项和参数类型),并记录在DECL_RTL(parm)中
   3) 向指令序列中发射指令(若需要),此指令负责将传入参数复制到当前函数的栈空间(mov parm.incoming_rtl DECL_RTL(parm))
   4) 根据AAPCS64标准,确定当前函数返回值存储位置(如R0寄存器),并将此rtx(REG)表达式记录到 ctrl->result中,但返回值设置到R0的操作并非此函数发出的(而是construc
   这里需要注意的是:
   1) 此函数并未处理不定参数,函数若有不定参数则是在函数体代码中显式获取的
   2) 若传入参数来自caller的栈中,则其rtx(MEM)表达式通常为  (mem (plus virtual_incoming_args_rtx, offset)),在栈向低地址增长时候,此offset为正数
   3) 若传入参数最终被复制到callee的栈中,则其空间是在callee的局部变量之后,其rtx(MEM)表达式通常为 mem(plus(virtual_stack_vars_rtx, frame_offset)),
      在栈向低地址增长时,此frame_offset为负数
*/
static void assign_parms (tree fndecl)
{
  struct assign_parm_data_all all;    /* 此结构体记录当前函数的所有形参信息 */
  tree parm;
  vec<tree> fnargs;

  crtl->args.internal_arg_pointer = targetm.calls.internal_arg_pointer ();   /* 指向 virtual_incoming_args_rtx 表达式, 此虚拟寄存器在callee栈中作为
  assign_parms_initialize_all (&all);                 /* 清空 all结构体,并初始化 all->args_so_far all->reg_parm_stack_space */

  fnargs = assign_parms_augmented_arg_list (&all);    /* fnargs返回一个数组,此数组记录此函数参数链表中的每个形参的树节点 (不定参数最后的...不对应树节点

  FOR_EACH_VEC_ELT (fnargs, i, parm)                  /* 遍历此函数的所有形参树节点 */
    {
      struct assign_parm_data_one data;               /* data保存每个参数的信息 */
      assign_parm_find_data_types (&all, parm, &data);    /* 确定当前形参树节点(parm)的类型,机器模式并保存到data中 */
      /* 根据当前形参的类型信息(data)和已处理的形参个数,确定根据AAPCS64标准此形参的传入参数位置,传入参数可能来自硬件寄存器或caller栈空间:
        * 若参数应来自硬件寄存器,则data->entry_parm 返回代表硬件寄存器编号的rtx(REG)副本
        * 若参数应来自caller参数栈,则data->locate 返回计算出的此参数基于virtual_incoming_args_rtx的偏移(此时data->entry_parm为NULL_RTX,作为判断是否为寄
      assign_parm_find_entry_rtl (&all, &data);

      if (assign_parm_is_stack_parm (&all, &data))    /* 若当前传入参数来自caller的函数栈,则返回true; 否则返回false */
      {
          /* 为此参数构建代表其传入位置的rtx(MEM)表达式 (mem (plus virtual_incoming_args_rtx, offset)) 并将其记录到data->stack_parm中 (这里的offset总
          assign_parm_find_stack_rtl (parm, &data);
```

```
36.           /* 设置 data->entry_parm = data->stack_parm, entry_parm最终代表此传入参数的位置 */
37.           assign_parm_adjust_entry_rtl (&data);
38.       }
39.       /* 记录此形参的传入参数的来源位置(parm.incoming_rtl),需要注意的是这里要区分其和DECL_RTL(parm),后者是此形参传入后保存在当前函数栈中的位置. */
40.       set_decl_incoming_rtl (parm, data.entry_parm, false);
41.
42.       /* 这里为当前传入参数确定其在callee栈中的存储位置,并记录在DECL_RTL(parm)中(对比前面的parm.incoming_rtl),对于二者不是同一个内存位置的情况,则向指令序列
43.         mov parm.incoming_rtl DECL_RTL(parm)的指令. 传入参数在callee进入时需要立即被保存以确保AAPCS64标准占用的硬件寄存器被free出来可供后续统一寄存器分配
44.         传入参数可以被保存在callee的伪寄存器中或callee函数栈内(局部变量之后),具体存储在哪里主要取决于编译的优化选项 */
45.       if (...) .......;
46.       else if (data.passed_pointer || use_register_for_decl (parm))
47.         /* 若开启了-Ox优化则通常会走这里, 则此函数会先为传入参数(不论硬件寄存器还是栈参数)创建一个伪寄存器作为其在callee中的存储位置,此伪寄存器被记录到参数的
48.            之后此函数会向指令序列中发射指令,此指令负责将传入参数复制到callee的伪寄存器中 */
49.         assign_parm_setup_reg (&all, parm, &data);
50.       else
51.         /* 若未开启编译优化,则通常会走这里(无法通过寄存器存储的变量也会走这里,如有副作用的参数等), 此函数首先负责为传入参数确定在callee栈中的一个存储位置,若
52.            * 来自硬件寄存器,则此函数会在callee中为其分配一个栈空间(在局部变量之后), 如mem(plus(virtual_stack_vars_rtx, frame_offset)), 其中frame_offset
53.              将此rtx(MEM)表达式保存data->stack_parm中,且同时保存到参数的DECL_RTL(parm)中
54.            * 来自caller栈,则通常不需要向指令序列中发射指令,而是直接将此传入参数在caller栈中的位置(data->stack_parm)直接记录到参数的DECL_RTL(parm)中即可 *
55.         assign_parm_setup_stack (&all, parm, &data);
56.       .......
57.     }
58.   fnargs.release ();      /* 到此所有传入参数使用的硬件寄存器/栈地址,在callee栈的保存位置均已确定,二者之间的复制指令(若需)也已经发射到指令序列中了 */
59.
60.   ......
61.   real_decl_rtl = targetm.calls.function_value (TREE_TYPE (decl_result),fndecl, true);    /* aarch64_function_value 这里返回R0_REGNUM的副本rtx(REG
62.   REG_FUNCTION_VALUE_P (real_decl_rtl) = 1;                    /* 标记此rtx表达式存储此函数最终的返回值 */
63.   crtl->return_rtx = real_decl_rtl;                            /* crtl->return_rtx 记录按照AAPCS64标准, 此函数的返回值应该返回到哪个硬件寄存器中 */
64. }
```

需要注意的是expand_function_start中完成了前面描述callee中的大部分步骤,除了将最终函数的返回值保存到R0寄存器中,这一步是在construct_exit_block中完成的,这里先单独列举此复制流程:

```
1. static void construct_exit_block (void) {
2.     ......
3.     expand_function_end ();
4.     ......
5. }
6.
7. void expand_function_end (void) {
8.     ......
9.     if (DECL_RTL_SET_P (DECL_RESULT (current_function_decl))) {
10.         tree decl_result = DECL_RESULT (current_function_decl);
11.         /* 获取代表函数返回值的rtx表达式节点,在 greturn指令expand时会确保将函数返回值保存到 DECL_RTL(decl_result)中,见 expand_gimple_stmt_1,case GIMPLE
12.         rtx decl_rtl = DECL_RTL (decl_result);
13.         ......
14.         rtx real_decl_rtl = crtl->return_rtx;                    /* 获取AAPCS64标准中函数返回值应该存储到的rtx(REG)表达式 */
15.         emit_move_insn (real_decl_rtl, decl_rtl);                /* 发射指令,将函数返回值赋值到 R0寄存器 */
16.     }
17. }
```