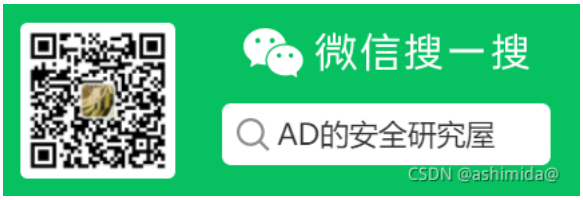
GCC源码分析(十六) — gimple转RTL(pass_expand)(下)

更多内容可关注微信公众号



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

四、pass_expand-gimple指令序列的展开

在pass_expand中,当前函数的所有函数体bb都是通过expand_gimple_basic_block将其中的gimple指令序列expand为RTL指令序列的:

```
unsigned int pass_expand::execute (function *fun) {
    .......
    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);
    .......
}
/*
  此函数负责遍历并依次expand当前bb的所有gimple指令,并将生成的rtx指令发射到指令序列中,当前bb的[BB_HEAD, BB_END]之中记录指令序列中属于当前bb的这一部分
  此函数中除解析GIMPLE_COND和尾调用的函数时可能导致边的修改外,其他gimple语句均通过expand_gimple_stmt函数统一展开
  此函数中还保证各个bb指令序列中最后一条语句和此bb的边在语义上的一致性
*/
static basic_block expand_gimple_basic_block (basic_block bb, bool disable_tail_calls)
{
  ......
  /* 获取当前bb的首指令(链表),并清空当前bb的 gimple指令队列,此函数之后bb的rtl指令序列唯一代表此bb中的指令 */
  stmts = bb_seq (bb);
  bb->il.gimple.seq = NULL;
  bb->il.gimple.phi_nodes = NULL;

  init_rtl_bb_info (bb);         /* 为 bb->il.x.rtl 分配结构体空间 */
  bb->flags |= BB_RTL;           /* 标记当前bb已经变为(正在处理成) RTL格式 */
  gsi = gsi_last (stmts);        /* 获取当前gimple语句列表的最后一条语句 */

  /* 若当前bb的最后一条指令为greturn语句,且没有返回值(return;的情况), 且其代码顺序的下一个bb就是EXIT_BB, 则删除此greturn语句,并将此bb唯一的边标记为fallthr
     以确保其gimple指令序列和边语义上的一致性 */
  if (!gsi_end_p (gsi) && gimple_code (gsi_stmt (gsi)) == GIMPLE_RETURN)
    {
      greturn *ret_stmt = as_a <greturn *> (gsi_stmt (gsi))
      if (bb->next_bb == EXIT_BLOCK_PTR_FOR_FN (cfun) && !gimple_return_retval (ret_stmt)) {
        gsi_remove (&gsi, false);                       /* 从指令序列中移除此greturn 语句 */
        single_succ_edge (bb)->flags |= EDGE_FALLTHRU;    /* 标记当前bb为fallthru */
      }
    }

  gsi = gsi_start (stmts);     /* 获取当前bb的第一条语句 */

  /* stmt记录当前bb中第一条标签指令(glabel)(若有,否则stmt为NULL), 当前指令序列非空时若有glabel则必定为此bb的第一条指令 */
  if (!gsi_end_p (gsi)) {
      stmt = gsi_stmt (gsi);
      if (gimple_code (stmt) != GIMPLE_LABEL) stmt = NULL;
    }

  /* 获取当前bb在全局map中已经生成的标签(若有), 当前bb expand之前,其他bb若引用到了当前bb(如GIMPLE_COND指令的解析过程中要跳转到当前bb),
```

```
    且当前bb中没有glabel指令,则此时会先为当前bb生成一个rtx(CODE_LABEL)标签保存到全局map lab_rtx_for_bb 中 */
  rtx_code_label **elt = lab_rtx_for_bb->get (bb);

  /* 若当前bb的第一条是glabel指令, 或之前其他bb expand时已经发射了此bb的 rtx(CODE_LABEL)标签到 lab_rtx_for_bb,则都会先在此bb的rtl指令序列中发射标签指令 *
  if (stmt || elt)
    {
      last = get_last_insn ();         /* 获取当前rtx指令队列中的最后一条rtx指令 */
      if (stmt) {
        /* expand当前bb中的glabel指令, 实际上是为其生成rtx(CODE_LABEL)指令,记录到DECL_RTL(gimple_label_label(stmt))中(若尚没有),并同时将其发射到当前rtx
        expand_gimple_stmt (stmt);
        gsi_next (&gsi);               /* 此指令处理完毕,后续处理一条指令 */
      }

      if (elt) emit_label (*elt);         /* 若全局中发现了当前bb的 rtx_label_code,则直接发射到指令序列中 */

      BB_HEAD (bb) = NEXT_INSN (last);    /* 设置前面发射的标签指令为当前bb的首指)  */
      ......
      /* 在当前指令序列中发射 (note NOTE_INSN_BASIC_BLOCK)指令,标记此bb的开始,若当前bb有标签则bb的第一条指令是标签指令,note是其第二条指令 */
      note = emit_note_after (NOTE_INSN_BASIC_BLOCK, BB_HEAD (bb));
    }
  else
    BB_HEAD (bb) = note = emit_note (NOTE_INSN_BASIC_BLOCK);    /* 若当前bb没有也不需要rtx标签,则 直接发射note 作为此bb的第一条rtx语句 */

  if (note) NOTE_BASIC_BLOCK (note) = bb;    /* 设置note指令所属基本块 */

  for (; !gsi_end_p (gsi); gsi_next (&gsi))    /* 遍历当前bb中的剩余所有gimple指令 */
    {
      basic_block new_bb;
      stmt = gsi_stmt (gsi);    /* 获取当前处理的gimple指令 */
      ......
      /* 判断当前要处理的gimple指令是否为 GIMPLE_CONDE 指令,是则调用expand_gimple_cond直接展开, 由于GIMPLE_COND可能导致边的修改,故没有在expand_gimple_s
      if (gimple_code (stmt) == GIMPLE_COND) {
        /* 此函数负责expand当前bb的最后一条GIMPLE_COND语句, 若GIMPLE_COND的then/else分支都存在且不会fallthru到next bb,则会导致生成一个new_bb,此bb在汇编
          (bb->next, 一个bb中只能有一条跳转指令,then/else分支没有fallthru则必定是两条跳转指令),此时当前bb最后一条rtx指令要么跳转(如then分支),要么fallthru
          而new_bb中的rtx指令序列则是无条件跳转到另一分支(如else).
           函数返回后若非无条件跳转,则当前bb一定拥有一条fallthru属性的边使其可跳转到到next_bb,且同时有一条跳转到其他bb的非fallthru属性的边(边的属性和rtx表
        */
        new_bb = expand_gimple_cond (bb, as_a <gcond *> (stmt));
        /* GIMPLE_COND若then/else分支都存在且不会fallthru到next bb,则会导致生成一个new_bb,此bb在汇编代码顺序上在当前bb之后(bb->next, 一个bb中只能有一条跳
          fallthru则必定是两条跳转指令),此时当前bb最后一条rtx指令要么跳转(如then分支),要么fallthru到new_bb; 而new_bb中的rtx指令序列则是无条件跳转到另一分
          next_bb都已经expand完成了, 故此函数返回new_bb, 之后会按照代码顺序继续处理new_bb->next_bb */
        if (new_bb) return new_bb;
      }
      else if (is_gimple_debug (stmt)) { ... }    /* debug指令在这里处理 */
      else
      {
          /* 尝试强转gcall指令, 若非gcall指令则call_stmt为NULL */
          gcall *call_stmt = dyn_cast <gcall *> (stmt);
          /* 若当前是个尾调用的gcall指令,但当前不允许尾调用(如函数结束会插入代码),则去掉此函数可以尾调用的标记 */
          if (call_stmt && gimple_call_tail_p (call_stmt) && disable_tail_calls)
            gimple_call_set_tail (call_stmt, false);

          /* 若当前是尾调用的gcall指令,则调用expand_gimple_tailcall单独expand,同样因为其可能导致边的修改 */
          if (call_stmt && gimple_call_tail_p (call_stmt)) {
              new_bb = expand_gimple_tailcall (bb, call_stmt, &can_fallthru);
              ......
          }
          else {        /* 除了GIMPLE_COND和尾调用外,其他gimple表达式都是直接通过 expand_gimple_stmt 函数 expand */
              def_operand_p def_p;    /* 这里是SSA_NAME相关的处理,先pass */
              def_p = SINGLE_SSA_DEF_OPERAND (stmt, SSA_OP_DEF);
              if (def_p != NULL) {
                  if (SA.values && bitmap_bit_p (SA.values, SSA_NAME_VERSION (DEF_FROM_PTR (def_p)))) {  continue; }
              }

              last = expand_gimple_stmt (stmt);    /* 所有不会导致边修改的gimple语句都通过此函数expand */
          }
      }
    }

  /* 这里应该是修复部分优化pass中可能出现的错误修改? 其遍历当前bb的所有后继边, 若其有fallthru属性的边的dest没有进入next_bb,
     则发射一条跳转到 dest的rtx指令,并去掉此边的fallthru属性 */
  FOR_EACH_EDGE (e, ei, bb->succs) {
      if ((e->flags & EDGE_FALLTHRU) && e->dest != bb->next_bb) {
        emit_jump (label_rtx_for_bb (e->dest));    /* 发射指令让当前bb跳转到dest bb */
        e->flags &= ~EDGE_FALLTHRU;    /* 当前边已经不是fallthrough了, 去掉此flag */
      }
  }
  last = get_last_insn ();      /* 获取当前已经发射的最后一条指令 */


  /* barrier指令不属于任何bb,当当前bb最后一条指令时无条件跳转时,会发射一个barrier指令代表此处代码不可达(防止此bb被fallthru),
    其不属于当前bb也不属于下一个bb,故若最后一条指令是barrier则当前bb最后一条指令前移 */
  if (BARRIER_P (last))  last = PREV_INSN (last);
  ......
  BB_END (bb) = last;          /* 设置last为当前bb的最后一条指令 */
  update_bb_for_insn (bb);     /* BB_HEAD 到 BB_END 之间的所有指令都属于当前bb,标记这些指令所属基本块为当前bb */
  return bb;
}
```

其中GIMPLE_COND指令由于会导致边的修改,故单独通过expand_gimple_cond函数来expand:

```
static basic_block expand_gimple_cond (basic_block bb, gcond *stmt)
{
  code = gimple_cond_code (stmt);      /* 获取 GIMPLE_COND 语句的subcode(比较的条件码,如EQ) */
  op0 = gimple_cond_lhs (stmt);        /* 获取比较操作两侧值的树节点 */
  op1 = gimple_cond_rhs (stmt);

  last2 = last = get_last_insn ();     /* 获取当前指令序列中已发射的最后一条指令 */

  extract_true_false_edges_from_block (bb, &true_edge, &false_edge);    /* 从当前bb抽取then/else两个分支的两个 edge 结构体*/

  true_edge->flags &= ~EDGE_TRUE_VALUE;    /* 在rtl格式中边的 TRUE/FALSE flag不再使用了,直接去掉*/
  false_edge->flags &= ~EDGE_FALSE_VALUE;

  /* 若else分支代码在当前bb下一个bb中(代码顺序),也就是说其应该是fallthru的,则当前bb最后一条rtx指令会条件跳转到then分支,并设置else分支的边为fallthru的(和下
  if (false_edge->dest == bb->next_bb) {
      .......
      return NULL;
  }
  /* 若true分支的代码在当前bb的下一个bb(通常是这里,因为正常来说,true分支的指令总是fallthru的
     这符合if() ...; else ...; 的代码顺序,而false分支总是非fallthru的 */
  if (true_edge->dest == bb->next_bb) {
      /*
         label_rtx_for_bb 先获取dest bb rtx格式下的标签指令rtx(CODE_LABEL)的指针, 若此时dest bb尚未expand且其没有glabel指令,
        则会先为dest bb生成rtx(CODE_LABEL)指令(称为false_label)并记录在全局数组lab_rtx_for_bb中
         jumpifnot_1会先expand op0/op1表达式(可能导致指令发射),之后则发射一条比较表达式,并发起一条跳转指令, 此跳转指令会在比较不成立的情况下跳转到false_la
         1) (insn (set cc (compare op0 op1))        /* compare 指令比较op0/op1两个操作数并将结果保存到cc寄存器中 */
         2) (jump_insn (set pc (if_then_else (!code CC 0)(false_label)(pc)))) /* 根据比较码(code) 查看cc中的比较结果是否成立, 如果成立则跳转到pc(也就
                                                                                 不成立则跳转到false_label) */
      */
      jumpifnot_1 (code, op0, op1, label_rtx_for_bb (false_edge->dest), false_edge->probability);
      /* 由于最终expand后,true分支成立时不跳转而是直接fallthru到下一个bb, 故 true_edge加上fallthru属性,与前面插入的rtx指令保持一致 */
      true_edge->flags |= EDGE_FALLTHRU;
      /* 若已发射的最后一条指令是一条barrier指令,则代表此GIMPLE_COND最后一条语句被expand成了一个无条件跳转到else分支的指令,则此时then分支不可达, 直接删除tl
      maybe_cleanup_end_of_block (true_edge, ...);
      return NULL;
    }

  /* 如果当前bb的then/else分支都没有fallthru到其代码顺序的下一个bb的话,则走这里; 此时首先发射指令:1)条件判断,2)如果为true跳转到true分支所在的标签 */
  jumpif_1 (code, op0, op1, label_rtx_for_bb (true_edge->dest), true_edge->probability);
  last = get_last_insn ();    /* 获取jumpif_1发射的最后一条指令 */

  emit_jump (label_rtx_for_bb (false_edge->dest));     /* 在其之后发射跳转到else分支的指令,这是个无条件跳转  1)*/
  BB_END (bb) = last;        /* then分支的最后一条指令就是此bb的最后一条指令 */

  /* 若then分支的最后一条是barrier,说明此GIMPLE_COND最终解析为了无条件跳转, bb_end向前变为那条jump指令(barrier不属于任何bb) */
  if (BARRIER_P (BB_END (bb))) BB_END (bb) = PREV_INSN (BB_END (bb));
  update_bb_for_insn (bb);    /* 标记当前bb中除barrier指令外的所有指令属于当前bb */

  /* 在bb的后面(代码顺序)创建new_bb,从else分支的(last之后发射的)第一条指令开始,到指令序列的最后一条指令都归到new_bb, 注意不论then分支是否为无条件跳转,else
  new_bb = create_basic_block (NEXT_INSN (last), get_last_insn (), bb);

  dest = false_edge->dest;    /* then分支跳转的目标bb */
  redirect_edge_succ (false_edge, new_bb);      /* 将then分支的边重新指向到new_bb,并将其变为fallthru的边 */
  false_edge->flags |= EDGE_FALLTHRU;
  .......

  make_single_succ_edge (new_bb, dest, 0);      /* 构建从new_bb到 dest的单后继的边,指令在前面的1处已经发射了,这里只设置边的关系 */

  /* 若new_bb 最后一条是barrier(通常都是),则此barrier指令不属于此bb的指令序列 */
  if (BARRIER_P (BB_END (new_bb))) BB_END (new_bb) = PREV_INSN (BB_END (new_bb));
  update_bb_for_insn (new_bb);                  /* 跟新new_bb中的指令序列所述bb */

  return new_bb;                         /* 返回新创建的bb,作为下一个继续分析的bb */
}
```

其他除尾调用的函数外,均通过expand_gimple_stmt函数展开:

```
/* 此函数负责除了GIMPLE_COND和尾调用函数外所有gimple指令的expand */
static rtx_insn * expand_gimple_stmt (gimple *stmt)
{
  rtx_insn *last = get_last_insn ();
  ......
  expand_gimple_stmt_1 (stmt);
  return last;
}

static void expand_gimple_stmt_1 (gimple *stmt)
{
  tree op0;
  switch (gimple_code (stmt))
  {
    case GIMPLE_GOTO:
      op0 = gimple_goto_dest (stmt);        /* 获取ggoto语句的跳转目标 */
      if (TREE_CODE (op0) == LABEL_DECL)    /* 若ggoto的跳转目标为标签节点 */
        expand_goto (op0);                  /* 先expand op0代表的标签节点(若需要), 并发射跳转到op0的无条件跳转指令(jump_insn (set pc_rtx, label_ref(l
      else
```

```
20.         expand_computed_goto (op0);         /* 先expand op0, 然后将其作为地址发射一个无条件跳转到op0的间接跳转指令(jump_insn (set pc_rtx, ...))*/
21.       break;
22.     case GIMPLE_LABEL:
23.       /* 此函数为glabel中的标签label_decl生成rtx(CODE_LABEL)指令,并记录在DECL_RTL(label_decl)中(若需要),
24.         同时向指令序列发射一条 (code_label ...)指令, 标签label_decl的位置因此确定 */
25.       expand_label (gimple_label_label (as_a <glabel *> (stmt)));
26.       break;
27.     case GIMPLE_NOP:            /* NOP和PREDICT 指令是不需要展开的,直接pass,注意NOP指令实际上到rtl是空的 */
28.     case GIMPLE_PREDICT:
29.       break;
30.     case GIMPLE_SWITCH:
31.       {
32.         gswitch *swtch = as_a <gswitch *> (stmt);
33.         if (gimple_switch_num_labels (swtch) == 1)    /* 若switch的case 为1个,则应该为default case, 直接发起goto语句 */
34.           expand_goto (CASE_LABEL (gimple_switch_default_label (swtch)));
35.         else
36.           expand_case (swtch);                        /* 否则 ... */
37.       }
38.       break;
39.     case GIMPLE_ASM:
40.       /* 将gasm expand 为 rtx(ASM_INPUT/ASM_OPERANDS)表达式(若gasm指令拥有属性则expand为 ASM_INPUT指令):
41.         * rtx(ASM_INPUT),最终发射为 (INSN (ASM_INPUT ....))
42.         * rtx(ASM_OPERANDS),最终根据是否引用了标签,发射为(JUMP_INSN/INSN (...))
43.         二者的区别在于,ASM_INPUT对于编译器来说就是一个字符串,会被原样写入到汇编代码中,编译器不会感知此指令中的副作用,此指令也无法引用gcc中的变量节点;
44.       */
45.       expand_asm_stmt (as_a <gasm *> (stmt));
46.       break;
47.     case GIMPLE_CALL:
48.       /* 对于gcc内置函数则调用expand_internal_call展开,否则此函数会为此gcall指令重新构建一个CALL_EXPR表达式:
49.          * 此时若此gcall指令需要保存返回值,则调用 expand_assignment展开;
50.          * 若gcall指令不需要返回值,则调用 expand_expr展开;
51.          二者最终均调用到 expand_call 函数,在此函数中发射对目标函数调用的指令(见<三、pass_expand-caller的参数传入和callee的参数接收>),最终生成的指令序列如
52.         .......        //传入参数复制到callee栈或callee的全局伪寄存器的指令
53.         (call_insn (parallel[ (call (mem (symbol_ref:DI ("_mcount")) ...
54.       */
55.       expand_call_stmt (as_a <gcall *> (stmt));
56.       break;
57.     case GIMPLE_RETURN:        /* GIMPLE_RETURN指令对应源码中的 return; 或return exp;语句,若是后者,则其发射的rtx指令最终要保证将函数返回值保存到当前函数
58.                                   的返回值节点DECL_RTL(DECL_RESULT (current_function_decl))中, 以确保 construct_exit_block 函数中可以从函数返回值节点直
59.       {
60.         op0 = gimple_return_retval (as_a <greturn *> (stmt));
61.         if (op0 && op0 != error_mark_node)
62.         {
63.           tree result = DECL_RESULT (current_function_decl);    /* 获取当前函数返回值的声明节点 */
64.           /* 若return的不是函数默认的返回值节点,则要构建一个MODIFY_EXPR将变量复制到函数的返回值节点(RETURN_DECL),此MODIRY_EXPR作为新的op0 */
65.           if (op0 != result) op0 = build2 (MODIFY_EXPR, TREE_TYPE (result), result, op0);
66.         }
67.         /* 若当前return语句没有返回值节点(也就是 return;的情况),则直接发射指令跳转到 return_label(完成收尾工作)即可, return_label的位置是在 construct_ex
68.           发射的指令序列为: (jump_insn (set pc_rtx, label_ref(void, return_label)))
69.         */
70.         if (!op0) expand_null_return ();
71.         else
72.           /* 若return指令有返回值,则先expand return语句的返回值表达式,若需要则这里会先发射指令序列, 此指令序列需确保最终确保函数的返回值存于函数的DECL_RTL(
73.              (也就是确保在函数返回前可通过访问DECL_RTL(result_decl) 获取到当前函数的返回值), 之后同样调用expand_null_return 跳转到return_label */
74.           expand_return (op0);
75.       }
76.       break;
77.     case GIMPLE_ASSIGN:
78.       {
79.         gassign *assign_stmt = as_a <gassign *> (stmt);
80.         tree lhs = gimple_assign_lhs (assign_stmt);        /* 赋值表达式的左值节点 */
81.         /* 若此赋值表达式的左值不是SSA_NAME,或subcode为单独的对象则走这里 */
82.         if (TREE_CODE (lhs) != SSA_NAME || get_gimple_rhs_class (gimple_expr_code (stmt)) == GIMPLE_SINGLE_RHS)
83.         {
84.             tree rhs = gimple_assign_rhs1 (assign_stmt);        /* 获取此赋值表达式的右值节点 */
85.             ......
86.             expand_assignment (lhs, rhs, gimple_assign_nontemporal_move_p (assign_stmt));        /* expand赋值表达式 */
87.         }
88.         else { /* 复杂情况下,如 gimple_assign <plus_expr, ...> 走这里 */
89.             ......
90.         }
91.       }
92.       break;
93.     default: gcc_unreachable ();
94.   }
95. }
```

五、pass_expand-init_block和exit_block的展开

在gimple低端化过程中将函数体的gimple指令序列划分到了一个个基本块(bb)中, 在划分基本块之前编译器先为每个函数都预先生成了两个特殊的bb:

- ENTRY_BB作为函数代码顺序上的第一个bb,代表函数的开始;
- EXIT_BB作为函数代码顺序上的最后一个bb,代表函数执行的结束;

在二者之间的bb记录的都是当前函数体的gimple指令序列, 其中代码顺序上的第一个bb称为 first_bb;

而实际上一个函数的汇编代码除了函数体部分,还需要在:

- 函数体执行之前完成如局部变量处理过程中生成的指令,传入参数的保存,桩指令(mcount,栈溢出保护)插入等操作:

在rtl expand阶段会生成这部分操作的rtx指令序列,并在函数construct_init_block中为这些指令分配一个新的基本块(称为init_bbl),并将此基本块插入到[ENTRY_BB, first_bb]之间,以确保在函数体执行前此部分代码得以执行.

- 函数体执行之后,栈溢出保护桩指令的插入,按照标准函数调用规则完成将函数返回值保存到硬件寄存器(如R0)中等操作:

在rtl expand阶段同样会生成这部分操作的rtx指令序列,并在函数construct_exit_block中为这些指令分配一个新的基本块(称为exit_bb),并将此基本块插入到EXIT_BB之前,并将原有跳转到EXIT_BB的所有边都重定位到exit_bb;

需要注意的是:

- 在rtl expand过程中,所有的greturn语句(代表函数返回)最终都要跳转到exit_bb完成收尾工作
- exit_bb中并没有发射函数返回指令,只是将函数的返回值保存到硬件寄存器R0中(返回指令是在后续pro_and_epilogue pass中发射的

这两个过程在 pass_expand中的流程为:

```
unsigned int pass_expand::execute (function *fun)
{
  ......
  init_block = construct_init_block ();        /* init_bb在此生成 */
  ......
  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);                          /* 当前函数体的所有bb在此expand为rtl指令 */
  ......
  construct_exit_block ();                     /* exit_bb在此生成 */
  ......
}
```

其中 construct_init_block负责init_bb的生成:

```
/* 此函数之后是对函数体的expand(rtx指令序列生成),此函数将函数体expand之前指令序列中已经发射的所有指令都放到一个新的init_bb中,并将其插入到 ENTRY_BB, first_
static basic_block construct_init_block (void)
{
  .......
  init_rtl_bb_info (ENTRY_BLOCK_PTR_FOR_FN (cfun));    /* 为当前函数的ENTRY_BB,EXIT_BB分配保存rtx指令序列结构体 (bb->il.x.rtl),但这两个BB中实际上是没
  init_rtl_bb_info (EXIT_BLOCK_PTR_FOR_FN (cfun));

  ENTRY_BLOCK_PTR_FOR_FN (cfun)->flags |= BB_RTL;      /* flags增加 BB_RTL 表示当前基本块已经转换为(正在转换成)RTL格式 */
  EXIT_BLOCK_PTR_FOR_FN (cfun)->flags |= BB_RTL;

  e = EDGE_SUCC (ENTRY_BLOCK_PTR_FOR_FN (cfun), 0);    /* 获取ENTRY_BB的唯一出向边,ENTRY_BB有且只有一个出向边 */

  if (e && e->dest != ENTRY_BLOCK_PTR_FOR_FN (cfun)->next_bb) { ... }    /* ENTRY_BB的下一个BB(first_bb)不是代码顺序的下一个BB,这里通常不会发生,pass
  else
    flags = EDGE_FALLTHRU;    /* first_bb是ENTRY_BB代码顺序的下一个bb,则标记ENTRY_BB的边为fallthru的 */

  /*
    create_basic_block在gimple/rtl阶段都可调用,但二者的hook函数是不同的,这里最终调用的是rtl_create_basic_block生成新的bb
    get_last_insn()获取之前已经发射的指令序列,这些指令序列主要是在前面的 expand_used_vars/expand_function_start 函数中发射的, 包括将函数传入参数复制
   到函数栈的指令,以及_mcount桩函数调用指令等.
    此函数负责分配init_bb,并在代码顺序上将其插入到 ENTRY_BB之后(正常是first_bb之前), 此前向指令序列中发射的所有指令都归属于init_bb.
    */
  init_block = create_basic_block (NEXT_INSN (get_insns ()), get_last_insn (), ENTRY_BLOCK_PTR_FOR_FN (cfun));
  ......

    /* 正常ENTRY_BB都是有且仅有唯一的出向边的, 这里将此边重定向到init_bb, 并发起 init_bb => first_bb的边, 函数开始的控制流由 ENTRY_BB => first_bb
    变为 ENTRY_BB => init_bb => first_bb (让边语义与rtl指令序列语义保持一致) */
  if (e) {
      first_block = e->dest;                /* 获取此边的dest bb,也就是first_bb */
      redirect_edge_succ (e, init_block);    /* 将边e由 <entry_bb, first_block> 调整为 <entry_bb, init_block> */

      e = make_single_succ_edge (init_block, first_block, flags);     /* 再做一条 <init_block, first_block> 的边 */
   }

  update_bb_for_insn (init_block);    /* 修改 init_block中所有insn所在的基本块为 init_block */
  return init_block;
}
```

其中construct_exit_block 负责exit_bb的生成:

```
/* 此函数负责生成exit_bb作为函数体执行完毕前最后必须执行的bb, 所有greturn指令最终都要跳转到此bb执行,其主要向指令序列中发射了:
     * greturn语句跳转的返回标签指令(return_label),
     * 将函数返回值复制到AAPCS64标准返回寄存器(R0)
     * 栈溢出保护的桩指令
     * 动态栈分配的释放指令
   并将这些指令放置到新分配的exit_bb中, 代码结束的顺序由 ... => EXIT_BB, 变为 ... => exit_bb => EXIT_BB, EXIT_BB所有的入向边都被重定向到exit_bb中
*/
static void construct_exit_block (void)
{
  rtx_insn *head = get_last_insn ();            /* 获取当前指令序列中已发射的最后一条指令 */
```

```
  basic_block prev_bb = EXIT_BLOCK_PTR_FOR_FN (cfun)->prev_bb;        /* 获取汇编代码顺序上,exit_bb前面的那个bb(prev_bb) */

  /* 此函数负责向指令序列中发射:
     * greturn语句跳转的返回标签指令(return_label),
     * 将函数返回值复制到AAPCS64标准返回寄存器(R0)
     * 栈溢出保护的桩指令
     * 动态栈分配的释放指令
  */
  expand_function_end ();

  end = get_last_insn ();    /* 再次获取当前指令序列中最后一条指令, [head,end]之间的指令是prev_bb之后的指令,其中包括expand_function_end函数发射的所有指令

  if (head == end)    /* 若expand_function_end没有生成任何指令,则直接返回即可 */
    return;
  ......
  /* 顺着指令序列遍历,直到head为return_label */
  if (NEXT_INSN (head) != return_label) ...;

  /* 创建exit_bb, 属于此bb的指令序列为 return_label到end之间的所有指令, 此bb代码顺序上插入到 prev_bb的后面(EXIT_BB前面) */
  exit_block = create_basic_block (NEXT_INSN (head), end, prev_bb);

  while (ix < EDGE_COUNT (EXIT_BLOCK_PTR_FOR_FN (cfun)->preds))    /* 将 EXIT_BB的所有入向边的dest都重定向到exit_bb(abnormal的边不处理 */
    {
      e = EDGE_PRED (EXIT_BLOCK_PTR_FOR_FN (cfun), ix);
      if (!(e->flags & EDGE_ABNORMAL))
        redirect_edge_succ (e, exit_block);
      else ix++;
    }

  e = make_single_succ_edge (exit_block, EXIT_BLOCK_PTR_FOR_FN (cfun), EDGE_FALLTHRU);    /* 为exit_bb生成唯一后继到 EXIT_BLOCK,类型为fallthru的边
  .......

  update_bb_for_insn (exit_block);    /* 标记exit_bb中所有rtx指令所属的bb */
}

/*
   此函数负责发射:
   * greturn语句跳转的返回标签指令(return_label)
   * 将函数返回值复制到AAPCS64标准返回寄存器(R0)
   * 栈溢出保护的桩指令
   * 动态栈分配的释放指令
*/
void expand_function_end (void)
{
  ......
  /* return_label是在 expand_function_start 中生成的rtx(CODE_LABEL)指令, 这里将其发射到指令序列,以确定所有greturn语句跳转的目标标签在指令序列中的位置 */
  emit_label (return_label);

  if (crtl->stack_protect_guard && ...)        //栈溢出检查的结束代码插桩
    stack_protect_epilogue ();

  /* 这里发射的指令如 (insn (set real_decl_rtl(R0) decl_rtl) */
  if (DECL_RTL_SET_P (DECL_RESULT (current_function_decl))) {
      tree decl_result = DECL_RESULT (current_function_decl);        /* 获取函数返回值的树节点 */
      /* 获取函数返回值的rtl表达式, greturn语句若返回了值,则在其expand过程中都会发射指令将返回值保存到此rtl表达式中 */
      rtx decl_rtl = DECL_RTL (decl_result);
      if (REG_P (decl_rtl) ? REGNO (decl_rtl) >= FIRST_PSEUDO_REGISTER: DECL_REGISTER (decl_result)) {    /* 若函数返回值节点是个伪寄存器, 或是个reg
        rtx real_decl_rtl = crtl->return_rtx;          /* 获取AAPCS64标准中,返回值应该存储到的寄存器(R0) */
        if (...
        else emit_move_insn (real_decl_rtl, decl_rtl);    /* 发射mov指令将函数返回值复制到硬件寄存器(R0)中,以满足AAPCS64标准 */
      }
  }
  .......
  if (! EXIT_IGNORE_STACK && cfun->calls_alloca)    /* 若函数可动态分配栈,则这里发射释放指令 */
    ......
}
```

六、pass_expand-边指令的插入和硬件寄存器初值的保存

rtl expand中最后一部分就是边指令的发射,在保存硬件寄存器初值的函数emit_initial_value_sets中就是通过边指令发射的方式向函数中插入的指令序列,故二者放到一起分析.

在gcc中通过函数get_hard_reg_initial_val可以获取函数入口进入时,代表某个硬件寄存器初值的伪寄存器rtx(REG)表达式:

```
/* 此函数传入一个硬件寄存器编号regno,返回的是一个记录此硬件寄存器在函数入口初值的伪寄存器rtx(REG)表达式(没有则增加表项)*/
rtx get_hard_reg_initial_val (machine_mode mode, unsigned int regno)
{
  struct initial_value_struct *ivs;
  rtx rv;

  /* 在 crtl->hard_reg_initial_vals; 数组中查找是否有类型为mode,节点编号为regno的rtx(REG)表达式,若有则返回代表此硬件寄存器的rtx(REG)表达式 */
  rv = has_hard_reg_initial_val (mode, regno);
  if (rv) return rv;

  ivs = crtl->hard_reg_initial_vals;
  ......

```

```
  ivs->entries[ivs->num_entries].hard_reg = gen_rtx_REG (mode, regno);    /* 生成代表编号为regno的硬件寄存器的rtx(REG)表达式 */
  ivs->entries[ivs->num_entries].pseudo = gen_reg_rtx (mode);             /* 生成一个伪寄存器,后续用于保存某硬件寄存器在函数入口时的初始值 */

  return ivs->entries[ivs->num_entries++].pseudo;                        /* 返回代表硬件寄存器初值的伪寄存器的rtx(REG)表达式 */
}
```

其实际上是为每个被引用到初值的硬件寄存器在全局map crtl->hard_reg_initial_vals中生成了一个<hard_reg, pseudo> 对, 此对中的hard_reg代表被引用初值的硬件寄存器, pseudo代表保存了此硬件寄存器初值的伪寄存器rtx(REG)表达式. **在rtl expand的过程中, 只要需要引用硬件寄存器的值就会调用到get_hard_reg_initial_val函数,此函数若发现全局map中没有,就会新增代表被引用的硬件寄存器的<hard_reg, pseudo> 对, 并返回代表硬件寄存器初值的pseudo表达式**. 而在rtl expand的最后, emit_initial_value_sets函数的工作就是在函数入口<ENTRY_BB,init_bb>之间再增加一个bb(若需要),并向其中发射指令以确保全局map中所有硬件寄存器的初始值都被复制到了对应的伪寄存器pseudo中:

```
unsigned int emit_initial_value_sets (void)
{
  struct initial_value_struct *ivs = crtl->hard_reg_initial_vals;    /* 此函数中保存了当前函数expand过程中所有被使用到初值的硬件寄存器对 <hard_reg, ps
  if (ivs == 0) return 0;    /* 如果此函数中没有代码需要使用硬件寄存器的初始值(如lr),则直接返回 */

  start_sequence ();
  for (i = 0; i < ivs->num_entries; i++)      /* 发射一个指令序列, 此指令序列将代码中每个被使用到的硬件寄存器的当前值复制到其对应的伪寄存器中 */
    emit_move_insn (ivs->entries[i].pseudo, ivs->entries[i].hard_reg);
  seq = get_insns ();
  end_sequence ();

  /* 将insn指令序列插入到当前函数的 ENTRY_BB => init_bb的执行流程中, 由于这是函数入口即执行的指令,故其可以将硬件寄存器的初值赋值到伪寄存器中以供后续使用
    此函数还会同时将当前函数中所有边中的边指令(若有)发射到此边的执行过程中 */
  emit_insn_at_entry (seq);
  return 0;
}

/* 此函数以边指令的形式将insn 插入到 ENTRY_BB => init_bb 的执行流程中, 并遍历当前函数的所有边,如果有边中存在边指令,则将此边指令插入到此边的控制流执行过程中
void emit_insn_at_entry (rtx insn)
{
  edge_iterator ei = ei_start (ENTRY_BLOCK_PTR_FOR_FN (cfun)->succs);    /* 获取函数的ENTRY_BB的唯一后继边 */
  edge e = ei_safe_edge (ei);
  gcc_assert (e->flags & EDGE_FALLTHRU);

  insert_insn_on_edge (insn, e);                /* 将insn指令序列添加到 e->insns.r中(此边的边指令) */
  /* 遍历cfg中所有边,若发现某个边有边指令(即e->insn.r非空),则将此边的边指令发射到一个new_bb中,并确保此new_bb只会在e->src => e->dest的执行过程中执行(
    控制流变为 src => new_bb => dest), new_bb不会在除了此流程外的任何流程中(如src的其他后继,或dest的其他前驱的执行过程)中执行 */
  commit_edge_insertions ();
}
```