GCC源码分析(十七) — rtl expand之后

ps: 这是源码分析系列最后一篇文章, 从<GCC源码分析(一) — 输入参数解析>到本文只是大体介绍了整个cc1编译的基本流程, 而实际上其中的细节十分复杂,尤其各个优化的pass更是gcc的精髓,因笔者了解有限,故这些内容在此系列中尚未涉及,后续有机会再加以补充.

由前可知, 函数expand过程中执行了all_passes链表中的所有pass,其中 pass_expand负责将gimple指令序列expand为rtl指令序列, pass_expand是一级pass中的倒数第二个, 而最后一个则为**pass_rest_of_compilation, 此pass中的subpass负责处理后续的所有操作**, all_passes的关键pass如下:

```
1. INSERT_PASSES_AFTER (all_passes)
2.     NEXT_PASS (pass_fixup_cfg);
3.     ......                                          /* 这里有一系列 gimple pass 先忽略 */
4.     NEXT_PASS (pass_cleanup_cfg_post_optimizing);   /* pass "optimized", 其作用是在RTL展开之前fixup CFG 并清除无用的BB */
5.     NEXT_PASS (pass_expand);                        /* pass "expand", 此pass负责 rtl 的expand,此pass执行后,当前函数的所有语义均由rtl指令序列表示,gim
6.     NEXT_PASS (pass_rest_of_compilation);
7.         NEXT_PASS (pass_instantiate_virtual_regs);  /* 此pass负责将虚拟寄存器转换为硬件寄存器 */
8.         NEXT_PASS (pass_jump);                      /* 此pass负责删除一些没用指令,删除后清理一遍不可达的bb */
9.         ........
10.        NEXT_PASS (pass_sched);                     /* 负责对rtl指令进行指令调度(消除结构,数据,控制相关的重排) */
11.        NEXT_PASS (pass_ira);                       /* 统一寄存器分配,这里主要是伪寄存器/虚拟寄存器 => 硬件寄存器的处理 */
12.        NEXT_PASS (pass_reload);                    /* 也是统一寄存器分配相关的pass, 这里主要是对于需要保存到栈中的伪寄存器,做伪寄存器 => 内存栈的处理
13.        ......
14.        NEXT_PASS (pass_thread_prologue_and_epilogue); /* 此pass负责发射函数的prologue/epilogue指令 */
15.        ......
16.        NEXT_PASS (pass_reorder_blocks);            /* pass "bbro", 此pass负责基本块顺序的的重排 */
17.        ......
18.        NEXT_PASS (pass_sched2);                    /* 统一寄存器分配之后再次执行指令调度(重排) */
19.        ......
20.        NEXT_PASS (pass_late_compilation);
21.            NEXT_PASS (pass_free_cfg);              /* 清除rtl指令序列和bb关系的指针, 此pass之后cfg和rtl指令序列不再保持同步 */
22.            NEXT_PASS (pass_machine_reorg);         /* 平台相关pass,  在arm64平台没有开启  */
23.            ......
24.            NEXT_PASS (pass_final);                 /* pass "final", 此pass负责此函数最终的汇编代码输出 */
25.            ......
26.            NEXT_PASS (pass_clean_state);           /* 对当前一些全局变量做重置,以便于下一个函数的分析 */
```

pass_expand之后的pass完成了包括 虚拟寄存器转硬件寄存器、rtl指令调度、统一寄存器分配、pro/epilogue插入、基本块重排、以及最终汇编代码输出、全局结构体重置等多种操作, 这里只记录对编译过程理解最关键的几个pass.

一、虚拟寄存器转硬件寄存器——pass_instantiate_virtual_regs

二、统一寄存器分配——pass_ira/pass_reload

三、rtl pro/epilogue的发射——pass_thread_prologue_and_epilogue

四、最终汇编代码输出——pass_final

一、虚拟寄存器转硬件寄存器——pass_instantiate_virtual_regs

pass_instantiate_virtual_regs:execute函数为 instantiate_virtual_regs,此函数将非USE/CLOBBER/ASM_INPUT之外的所有指令中的**虚拟寄存器都替换成了对应的硬件寄存器,同时对非调试类指令确定其指令模板编号**,虚拟寄存器的替换规则如:

```
1.     virtual_incoming_args_rtx => arg_pointer_rtx (65)
2.     virtual_stack_vars_rtx => frame_pointer_rtx (64)
3.     virtual_stack_dynamic_rtx => stack_pointer_rtx (31)
4.     virtual_outgoing_args_rtx => stack_pointer_rtx (31)
```

此函数的定义如下:

```
unsigned int instantiate_virtual_regs (void)
{
  rtx_insn *insn;
  /* 遍历当前函数的整个rtx指令序列 */
  for (insn = get_insns (); insn; insn = NEXT_INSN (insn))
    if (INSN_P (insn)) {    /* 遍历所有 INSN/JUMP_INSN/CALL_INSN/DEBUG_INSN 指令 */
     /* 若此指令的PATTEN(子rtx表达式)为USE/CLOBBER/ASM_INPUT/DEBUG_MARKER,则其中不会包含虚拟寄存器,直接pass */
     if (GET_CODE (PATTERN (insn)) == USE || GET_CODE (PATTERN (insn)) == CLOBBER
        || GET_CODE (PATTERN (insn)) == ASM_INPUT || DEBUG_MARKER_INSN_P (insn))
      continue;
     else if (DEBUG_BIND_INSN_P (insn))
      instantiate_virtual_regs_in_rtx (INSN_VAR_LOCATION_PTR (insn));
     else
      instantiate_virtual_regs_in_insn (insn);    /* 非调试类指令都走这里,这里将指令中所有虚拟寄存器都替换为物理寄存器,并确定此指令的指令模板编号 */
     ......
   }

  virtuals_instantiated = 1;
  return 0;
}
```

```
/* instantiate_virtual_regs_in_insn 函数将RTL指令序列中使用的虚拟寄存器全部替换为对应的硬件寄存器(若有偏移则可能导致插入指令),
   不论是否发生了替换,此RTL指令对应的指令模板编号都会在此函数中确定 */
void instantiate_virtual_regs_in_insn (rtx_insn *insn)
{
  ......
  if(...
  else {
      extract_insn (insn);             /* 这里设置后面的  recog_data.n_operands */
      insn_code = INSN_CODE (insn);    /* 获取指令的指令模板号 */
  }

  for (i = 0; i < recog_data.n_operands; ++i) {    /* 遍历此指令PATTEN中所有操作数 */
    x = recog_data.operand[i];
    switch (GET_CODE (x)) {
    case MEM:
      rtx addr = XEXP (x, 0);    /* 获取MEM表达式的子表达式 */
      /* 将rtx表达式addr中所有的虚拟寄存器替换为对应的物理寄存器, 若发生了替换则此函数返回true,否则返回false;
         虚拟寄存器=>物理寄存器的转换通常在这一步就完成了 */
      if (!instantiate_virtual_regs_in_rtx (&addr)) continue;
      ......
      break;
    case REG:
      /* 确认rtx(REG)表达式是哪个虚拟寄存器,并返回其对应的硬件寄存器(若有),如若x为virtual_incoming_args_rtx,
         则new_rtx返回arg_pointer_rtx,而offset则返回虚拟寄存器和硬件寄存器的地址差(通常为0) */
      new_rtx = instantiate_new_reg (x, &offset);
      if (new_rtx == NULL) continue;            /* 若x非虚拟寄存器的编号,则new_rtx返回空, 代表此表达式无需替换,直接返回 */
      /* 若发生了虚拟寄存器=>物理寄存器的转换,但二者的offset差值为0, 则x直接返回虚拟寄存器对应的硬件寄存器的rtx表达式*/
      if (known_eq (offset, 0)) x = new_rtx;
      else ...
      break;
    case SUBREG: ......
    default: continue;
   }
   /* 将此函数的第i个操作数重置为x,对于MEM表达式来说实际上没有修改x(若修改则只是修改了其子表达式),对于REG来说则是将虚拟寄存器的rtx表达式替换为硬件寄存器的rtx
   *recog_data.operand_loc[i] = recog_data.operand[i] = x;
    any_change = true;  /* 只要当前rtl指令有任何操作数中出现了虚拟寄存器=>硬件寄存器的替换,就会标记any_change为true(没有替换前面直接continue) */
  }

  if (any_change) {          /* 只要rtl指令操作数出现了虚拟寄存器=>硬件寄存器的替换,就会将已查找到的指令模板编号重置为-1 */
      .......
      INSN_CODE (insn) = -1;
  }

  if (asm_noperands (PATTERN (insn)) >= 0) ......
  else {
      if (recog_memoized (insn) < 0)    /* 重新为rtl指令确定指令模板,并记录到INSN_CODE(insn)中,若找不到则报错 */
        fatal_insn_not_found (insn);
  }
}
```

二、统一寄存器分配——pass_ira/pass_reload

统一寄存器分配主要分为两步:

- pass_Ipa主要负责将伪寄存器=>硬件寄存器
- pass_reload主要负责将 伪寄存器 => 内存

二者的细节没有仔细看[TODO],在二者执行完毕后 gcc中应该就不能再分配伪寄存器了

三、rtl pro/epilogue的发射——pass_thread_prologue_and_epilogue

此函数在函数的指令序列开始/结束的位置插入prologue和epilogue的代码,其中最主要的就是当前函数返回到父函数的**rtl指令(jump_insn return)就是通过这里的epilogue插入的**:

```
unsigned int rest_of_handle_thread_prologue_and_epilogue (void)
{
  ......
  thread_prologue_and_epilogue_insns ();
  ......
  return 0;
}

/*
   此函数负责为当前函数插入 prologue和epilogue的代码, 函数返回的 (jump_insn return)指令就是在这里的epilogue中插入的,
 需要注意的是,对于不经过exit_bb而直接跳转到EXIT_BB的sibling_call,这里会为其单独插入 sibling_epilogue.
*/
void thread_prologue_and_epilogue_insns (void)
{
  ......
  edge entry_edge = single_succ_edge (ENTRY_BLOCK_PTR_FOR_FN (cfun));        /* 获取函数入口的唯一后继边 */
  rtx_insn *prologue_seq = make_prologue_seq ();    /* 生成函数 PROLOGUE的指令,这里包括PAC和栈溢出的输出,以及输出 NOTE_INSN_PROLOGUE_END */
  rtx_insn *epilogue_seq = make_epilogue_seq ();    /* 生成函数 EPILOGUE的指令,包括 EPILOGUE_BEGIN, PACIASP(若需要), (jump_insn return)指令 */
  ......
  /* 找到当前函数fallthru 到EXIT_BB的那个bb,正常应该非空且是rtl expand时的exit_bb(除非后续又插入了新bb),这里暂也称为exit_bb */
  edge exit_fallthru_edge = find_fallthru_edge (EXIT_BLOCK_PTR_FOR_FN (cfun)->preds);

  if (exit_fallthru_edge) {
      if (epilogue_seq) {
        insert_insn_on_edge (epilogue_seq, exit_fallthru_edge);    /* epilogue的指令序列直接插入 exit_bb的边指令中 */
        /* 发射所有边的边指令序列, epilogue被发射到 exit_bb => EXIT_BB之间(实际上是 exit_bb的最后), 但需要注意的是此时直接跳转到EXIT_BB的返回控制流中并未
        commit_edge_insertions ();
        /* 找到代表前面 exit_bb => EXIT_BB的fallthru边, 由于 epilogue中通常插入了return语句,故这里去掉此边的fallthru属性 */
        FOR_EACH_EDGE (e, ei, EXIT_BLOCK_PTR_FOR_FN (cfun)->preds)
          if (((e->flags & EDGE_FALLTHRU) != 0) && returnjump_p (BB_END (e->src)))
            e->flags &= ~EDGE_FALLTHRU;
      }
      ......
  }

  if (...|| prologue_seq) {
      .......
      insert_insn_on_edge (prologue_seq, entry_edge);        /* 将prologue指令序列(若非空)插入到ENTRY_BB的唯一后继边的边指令中 */
      commit_edge_insertions ();                             /* 再次发射所有边指令, prologue_seq被发射到 ENTYR_BB => init_bb的执行过程中 */
  }
  /*
     遍历EXIT_BB的所有前驱节点,正常情况下源码中的所有返回指令(return)最终都会导致控制流跳转到exit_bb,然后由 exit_bb fallthru到 EXIT_BB,
     但也有例外如 SIBLING CALL, 其不跳转到exit_bb,而是直接跳转到EXIT_BB,这是因为sibling call中已经有了return表达式,如
       (call_insn (parallel (call ..., return))), 此call_insn中有 ret(RETURN),故无需跳转到exit_bb.
     而此函数的作用是在所有 SIBLING_CALL 到EXIT_BB的指令前面,为SIBLING_CALL添加单独的退出代码(sibcall_epilogue)
  */
  for (ei = ei_start (EXIT_BLOCK_PTR_FOR_FN (cfun)->preds); (e = ei_safe_edge (ei)); ei_next (&ei)) {
     if (e->flags & EDGE_IGNORE) {    /* 有 EDGE_IGNORE 标记的边直接pass */
       e->flags &= ~EDGE_IGNORE;
       continue;
     }
     rtx_insn *insn = BB_END (e->src);                               /* 获取此边所在bb的最后一条指令 */
     if (!(CALL_P (insn) && SIBLING_CALL_P (insn))) continue;        /* 只有是 sibiling call的情况才处理,其他直接pass */

     /*
        对于sibcall,在其前面插入gen_sibcall_epilogue生成的指令,此指令和epilogue_seq 的区别在于其不再次发射return指令
        sibcall_epilogue 会被插入到 sibcall这条CALL_INSN之前(插入到之后没有用,在 call_insn中就已经有return了,会先返回)
     */
     if (rtx_insn *ep_seq = targetm.gen_sibcall_epilogue ()) {
       start_sequence ();
       emit_note (NOTE_INSN_EPILOGUE_BEG);
       emit_insn (ep_seq);
       rtx_insn *seq = get_insns ();
       end_sequence ();
       ......
       emit_insn_before (seq, insn);     /* 将sibcall_epilogue的代码插入到 CALL_INSN之前 */
     }
  }
  .......
  epilogue_completed = 1;            /* 代表 epilogue插入完毕 */
  .......
}
```

四、最终汇编代码输出——pass_final

pass_final虽然不是在函数expand执行的all_passes队列中的最后一个pass,但实际上函数expand的工作在pass_final中就完成了,其后的1,2个pass都是负责首尾工作的. pass_final的作用是将当前函数的rtl指令序列转换为汇编代码并最终输出到汇编文件(asm_out_file)中,其大体流程如下:

```
unsigned int rest_of_handle_final (void)
{
```

```
  /* 到此函数之前,当前函数的函数体已经全部转换为rtl格式了,但通常当前函数尚没有对应的rtl表达式(DECL_RTL(current_function_decl)为空, 此函数会:
     * 先通过make_decl_rtl为当前函数生成rtl表达式 (mem (symbol_ref "func_name")), 并记录到DECL_RTL中, 此表达式的内容实际上是对函数名的符号引用
     * 之后返回此表达式中当前函数的函数名字符串 "func_name" */
  const char *fnname = get_fnname_from_decl (current_function_decl);
  .......
  /*
    在汇编文件中输出函数头,如:
       .align  2
       .global main
       .type   main, %function
       main:
  */
  assemble_start_function (current_function_decl, fnname);

  rtx_insn *first = get_insns ();    /* 获取当前函数指令序列首指令地址 */
  /* 这里会执行 targetm.asm_out.function_prologue(但在aarch64为空,先pass),输出如:
      .LFB0:
        .cfi_startproc
  */
  final_start_function_1 (&first, asm_out_file, &seen, optimize);
  final_1 (first, asm_out_file, seen, optimize);     /* 遍历当前指令序列(first)中所有指令,并按照指令模板将此指令的汇编代码依次输出到汇编文件(asm_out_fi
  ......
  /* 这里会执行顺序执行 debug_hooks->end_function; targetm.asm_out.function_epilogue(实际为空); debug_hooks->end_epilogue, 并输出:
    .cfi_endproc
    .LFE0:
  */
  final_end_function ();
  /* 这里会输出unwind信息(若需要),主要输出了函数结束时的.size语句如:
    .size   main, .-main
  */
  assemble_end_function (current_function_decl, fnname);
  .......
  if (! quiet_flag) fflush (asm_out_file);                    /* 将输出刷新到汇编文件 */
  .......
  if (!DECL_IGNORED_P (current_function_decl))                /* 若此函数在符号表中不会被忽略,且存在定义,则会调用此回调 */
    debug_hooks->function_decl (current_function_decl);
  ......
  DECL_INITIAL (current_function_decl) = error_mark_node;     /* 此函数的tree_block 在这里可以释放了 */

  /* 若存在构造函数和析构函数则在这里有函数需调用*/
  if (DECL_STATIC_CONSTRUCTOR (current_function_decl) && targetm.have_ctors_dtors)
    targetm.asm_out.constructor (XEXP (DECL_RTL (current_function_decl), 0), decl_init_priority_lookup (current_function_decl));
  if (DECL_STATIC_DESTRUCTOR (current_function_decl) && targetm.have_ctors_dtors)
    targetm.asm_out.destructor (XEXP (DECL_RTL (current_function_decl), 0), decl_fini_priority_lookup(current_function_decl));
  return 0;
}
```

```
/* 此函数的输出如:
      sub      sp, sp, #16
      .cfi_def_cfa_offset 16
      str      w0, [sp, 12]
      mov      w0, 0
      add      sp, sp, 16
      .cfi_def_cfa_offset 0
      ret
*/
void final_1 (rtx_insn *first, FILE *file, int seen, int optimize_p)
{
  rtx_insn *insn, *next;
  .......
  for (insn = first; insn;) {    /* 遍历当前指令序列中所有指令,并按照指令模板将此指令的汇编代码依次输出到汇编文件中 */
      ......
      insn = final_scan_insn (insn, file, optimize_p, 0, &seen);
  }
  ......
}

rtx_insn * final_scan_insn (rtx_insn *insn, FILE *file, int optimize_p, int nopeepholes, int *seen)
{
  .......
  rtx_insn *ret = final_scan_insn_1 (insn, file, optimize_p, nopeepholes, seen);
  .......
  return ret;
}

/*
  此函数将指令分为4种情况,note/barrier/code_label/default:
    * NOTE: note指令大部分是没有输出的
    * BARRIER: 是直接忽略的,没有任何输出
    * CODE_LABEL: 会向汇编代码中输出一个标签定义,如 .L6, 6为标签的ID,并不会输出标签字符串名
    * default: 其他输出都在这里, 其对于汇编是直接拿汇编字符串做类似printf的输出; 对于其他指令则是先找到此rtx指令的指令模板和确定所有操作数,最终也是一个类似p
  最终指令的汇编代码都被输出到汇编文件(asm_out_file)中.
*/
rtx_insn * final_scan_insn_1 (rtx_insn *insn, FILE *file, int optimize_p ATTRIBUTE_UNUSED, int nopeepholes ATTRIBUTE_UNUSED, int *seen)
{
  .......
  if (insn->deleted ()) return NEXT_INSN (insn);    /*  若当前insn已经被删除了, 则直接返回下一条指令的指针, 此指令不输出 */
  switch (GET_CODE (insn)) {

    case NOTE:            /* 对于note 指令, 大部分case 都是没有输出的, pass */
     switch (NOTE_KIND (insn))  { ..... }
```

```
     break;
    case BARRIER:        /* 对于barrier指令, 不作任何输出,其只在rtl中用来标记此处代码后的代码fallthru不可达 */
     break;
    case CODE_LABEL:     /* 对于标签语句,这里输出的是 .L+标签在编译单元内的id, 如.L6: */
     .......
     if (!DECL_IGNORED_P (current_function_decl) && LABEL_NAME (insn))    /* 若此标签在符号表中有对应名字,则调用此回调 */
        debug_hooks->label (as_a <rtx_code_label *> (insn));
     .......
        targetm.asm_out.internal_label (file, "L", CODE_LABEL_NUMBER (insn));    /* 这里直接向汇编文件输出如 .L6: 这样的标签定义, 这里的数字为标签的编
     break;
    default:             /* 其余case都走这里 */
    {
     rtx body = PATTERN (insn);        /* 获取当前rtx指令中具体的的子表达式, 如(jump_insn (set pc ...)) 中的(set pc ...) */
     .......
     if (GET_CODE (body) == USE || GET_CODE (body) == CLOBBER) break;    /* USE和CLOBBER不对应汇编指令,没有任何输出 */
     .......
     /* 对于ASM_INPUT表达式,则直接将字符串原样输出到汇编代码中,但实际上其前后还有一些注释,如:
           #APP
           // 11 "1.c" 1
              mov x0, x0
           // 0 "" 2
           #NO_APP
     */
     if (GET_CODE (body) == ASM_INPUT)     {
        const char *string = XSTR (body, 0);        /* 获取ASM_INPUT表达式中的汇编字符串 */
        if (string[0]) {
          .......
          loc = expand_location (ASM_INPUT_SOURCE_LOCATION (body));    /* 获取汇编的源码位置 */
          if (*loc.file && loc.line)    /* 输出汇编的源码行号信息, 如 ;# __LINE__ __FILE__ 1 */
            fprintf (asm_out_file, "%s %i \"%s\" 1\n", ASM_COMMENT_START, loc.line, loc.file);
          fprintf (asm_out_file, "\t%s\n", string);    /* 输出汇编语句, 即 ASM_INPUT的整个字符串 */
        }
        break;
     }
     if (asm_noperands (body) >= 0)    /* 若是一个有操作数的复杂汇编(ASM_OPERANS),则格式化输出 */
     {
        unsigned int noperands = asm_noperands (body);    /* 获取当前 ASM_OPERANS的操作数个数 */
        ......

        string = decode_asm_operands (body, ops, NULL, NULL, NULL, &loc);    /* 此汇编表达式中操作数信息都收集到 ops中, 而用户输入的汇编字符串则作为st
        .......
        output_asm_insn (string, ops);        /* 格式化输出ASM_OPERANDS, ops是各个参数, 此函数类似printf */
        break;
     }
     ......
     insn_code_number = recog_memoized (insn);    /* 非汇编输出,USE/CLOBBER的情况,则先获取指令的模板编号,这里通常是获取,模板编号通常在instantiate_virtu
     cleanup_subreg_operands (insn);               /* 根据指令模板编号(INSN_CODE(insn)),将此指令中所有操作数信息都记录到全局recog_data中 */
     ......
     /* 根据模板编号(insn_code_number),先获取此指令在模板中的输出模板,如indirect_jump的模板编码为1(insn_data[1]),对应的输出模板为字符串 "br\t%0" */
     templ = get_insn_template (insn_code_number, insn);
     ......
     /* 根据指令模板中的输出模板和各个操作数的信息向汇编文件中输出此指令的汇编代码,其中: templ是输出模板字符串(如"br\t%0"),recog_data.operand记录所有操作数
     output_asm_insn (templ, recog_data.operand);
     ......
    }
   }
   return NEXT_INSN (insn);    /* 返回下一条指令,重入后继续处理下一条指令 */
}
```