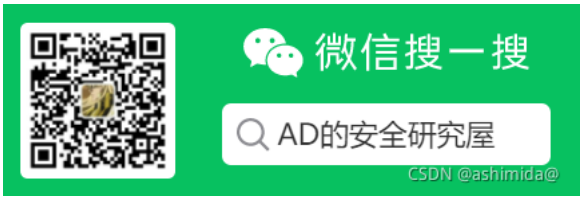
GCC源码分析(十二) — gimplify之后的基本流程



一、gimplify之后

gimplify之后的所有流程主要都是在函数 symbol_table::compile中完成的，此过程中除了IPA_PASS的执行外,还完成了:gimple=>rtl的转换, 以及最终rtl=>汇编代码的输出:

```
1. void symbol_table::finalize_compilation_unit (void)
2. {
3.   ......
4.   current_function_decl = NULL;    /* 全局的 cfun, current_function_decl被清空 */
5.   set_cfun (NULL);
6.
7.   analyze_functions (/*first_time=*/true);    /* 所有的 gimplify 都在此函数的两次调用中完成 */
8.   ......
9.   analyze_functions (/*first_time=*/false);
10.  ......
11.  compile ();                    /* gimplify之后的所有流程都在此函数中完成 */
12.  ......
13. }
```

symbol_table::compile的主要流程如下:

```
1. void symbol_table::compile (void)
2. {
3.   .......
4.   state = IPA;    /*  标记当前处于IPA pass的执行状态 */
5.
6.   if (!seen_error ()) ipa_passes ();     /* 执行所有不针对具体函数的 IPA passes(过程间优化) */
7.
8.   global_info_ready = true;
9.
10.  /* 将代表当前所在段的全局变量(in_section)切换到 text_section, 并同时输出 "\t.text" 到汇编文件,这里需要注意的是 .text 并不是汇编代码中输出的第一条指令,
11.     在 do_compile => lang_dependent_init 时打开输出文件 main_input_filename时先输出了代码:
12.      .arch armv8-a
13.      .file "1.c"
14.     而这里是在这两句之后输出了 ".text"
15.  */
16.  switch_to_section (text_section);
17.
18.  (*debug_hooks->assembly_start) ();       /* 此hook表示,从此开始gcc会将所有gimple的函数/变量转化为rtl并同时输出到汇编代码中 */
19.  ......
20.  bitmap_obstack_initialize (NULL);
21.  execute_ipa_pass_list (g->get_passes ()->all_late_ipa_passes);      /* 执行所有的late ipa passes */
22.  bitmap_obstack_release (NULL);
23.
24.  mark_functions_to_output ();    /* 标记哪些函数需要被输出到汇编代码中 */
25.  ......
26.  state = EXPANSION;            /* 标记当前处于函数/变量的gimple => rtl阶段(称为expand), 此时所有ipa pass都已经执行完毕了 */
27.
28.  /* 此函数负责处理需要按照顺序输出的函数和变量,对这些变量和函数expand 为rtl后直接输出到汇编代码 , 在未开启优化时,大部分函数都在这里输出, 开启优化 (如-O2) (!
29.  output_in_order ();
30.
31.  /* 处理 output_in_order 中没有处理的函数, 若开启优化则函数通常都在这里处理, 其处理方式也是 expand 为rtl后输出,只不过是按照全局符号表的顺序输出的 */
32.  expand_all_functions ();
33.
34.  /* 处理 output_in_order 中没有处理的变量, 同样按照其在全局符号表中的顺序输出 */
35.  output_variables ();
36.
37.  ......
38.  state = FINISHED;            /* 标记当前所有的函数/变量的输出已经结束 */
39.  output_weakrefs ();          /* 对弱引用的输出 */
40.  ......
41. }
```

二、ipa_passes/late_ipa_passes

symbol_table::compile 函数在gimplify后首先对整个编译单元执行所有的 ipa_pass,包括 all_small_ipa_passes, all_regular_ipa_passes以及 all_late_ipa_passes,.这里需要注意的是:

- IPA pass的作用是过程间优化,故其是不针对具体函数的, 其执行时全局的cfun, current_function_decl均为空, 而若IPA_PASS中有针对函数的 sub pass(如GIMPLE_PASS),则此时会遍历每个函数并对其执行此pass,每次遍历都会设置cfun, current_function_decl, 故在整个GCC编译过程中,如果是一个IPA_PASS则肯定没有函数上下文,而如果是GIMPLE_PASS/RTL_PASS则一定是有文件上下文的.
- SSA生成的pass, 负责inline的pass都是在IPA_PASS中完成的
- 所有pass的链表可参考 passes.def,或通过 编译选项 -fdump-passes查看

```
void ipa_passes (void)
{
  gcc::pass_manager *passes = g->get_passes ();        /* 获取全局的pass_manager */

  set_cfun (NULL);
  current_function_decl = NULL;            /* IPA pass是针对所有函数的,其执行时 cfun, current_function_decl均为空(除非子pass非IPA pass) */
  gimple_register_cfg_hooks ();            /* 由于此时尚未expand为rtl, 故基本块的操作还是使用 gimple_cfg_hooks 中的函数 */

  bitmap_obstack_initialize (NULL);

  invoke_plugin_callbacks (PLUGIN_ALL_IPA_PASSES_START, NULL);    /* IPA pass 开始的回调函数 */

  if (!in_lto_p)
    {
      execute_ipa_pass_list (passes->all_small_ipa_passes);       /* 所有 small_ipa_pass 都在这里执行,其中包括 ssa的创建,以及 early_inline的处理 */
      if (seen_error ()) return;
    }

  symtab->remove_unreachable_nodes (symtab->dump_file);          /* 遍历并删除符号表中不可达节点 */

  if (symtab->state < IPA_SSA)
    symtab->state = IPA_SSA;                                    /* all_small_ipa_passes中已经完成了SSA的创建,故这里将状态切换为 IPA_SSA */

  if (!in_lto_p)
    {
      ......
      execute_ipa_summary_passes((ipa_opt_pass_d *) passes->all_regular_ipa_passes);        /* 这里负责执行每个ipa pass的 generate_summary 函数 */
    }

  /* lto相关输出,先pass  */
  if (flag_generate_lto || flag_generate_offload) targetm.asm_out.lto_start ();
  ......
  if (flag_generate_lto || flag_generate_offload) targetm.asm_out.lto_end ();


  if (!flag_ltrans && ((in_lto_p && flag_incremental_link != INCREMENTAL_LINK_LTO) || !flag_lto || flag_fat_lto_objects))
    execute_ipa_pass_list (passes->all_regular_ipa_passes);     /* all_regular_ipa_passes的执行, 编译器自主inline在这里处理*/

  invoke_plugin_callbacks (PLUGIN_ALL_IPA_PASSES_END, NULL);    /* ipa pass执行完毕的回调 */

  bitmap_obstack_release (NULL);
}
```

三、函数与变量的rtl expand和汇编输出流程

IPA_PASS执行完毕后,则会依次遍历每个函数/变量节点,并在此过程中完成此函数/变量节点的gimple=>rtl以及rtl=>汇编代码, 其中:

- output_in_order 负责按照顺序处理函数/变量节点,在未开启优化时大部分函数和变量节点都是按照顺序处理的
- expand_all_functions负责按照全局符号表顺序处理 output_in_order中没有处理的函数
- output_variables负责按照全局符号表顺序处理 output_in_order中没有处理的变量
- output_weakrefs负责处理弱引用

这里不关注函数/变量输出顺序,故仅以output_in_order为例,描述遍历过程:

```
void output_in_order (void)
{
  int max = symtab->order;    /* order记录符号表中一共有多少个函数/变量/汇编符号需要按照顺序输出 */
  cgraph_order_sort * nodes = XCNEWVEC (cgraph_order_sort, max);    /* 分配数组,每个按照顺序输出的节点都会保存在此数组中 */

  FOR_EACH_DEFINED_FUNCTION (pf) {        /* 遍历所有有定义的函数节点 */
      if (pf->process && !pf->thunk.thunk_p && !pf->alias) {
        if (!pf->no_reorder) continue;    /* 未开启优化通常这里会返回 */
        i = pf->order;                    /* 获取函数节点输出顺序编号 */
        nodes[i].kind = ORDER_FUNCTION;
        nodes[i].u.f = pf;                /* 记录函数节点的cgraph_node指针 */
      }
```

```
  }

  FOR_EACH_VARIABLE (pv) {                /* 遍历所有变量节点 */
      if (!pv->no_reorder) continue;
      /* 使用硬件寄存器的变量直接pass */
      if (DECL_HARD_REGISTER (pv->decl) || DECL_HAS_VALUE_EXPR_P (pv->decl))  continue;
      i = pv->order;                      /* 获取变量节点输出顺序编号 */
      nodes[i].kind = pv->definition ? ORDER_VAR : ORDER_VAR_UNDEF;
      nodes[i].u.v = pv;                  /* 记录变量节点的varpool_node指针 */
  }

  for (pa = symtab->first_asm_symbol (); pa; pa = pa->next) {    /* 遍历所有汇编节点 */
      i = pa->order;
      nodes[i].kind = ORDER_ASM;
      nodes[i].u.a = pa;                  /* 记录汇编节点的 asm_node 指针 */
    }
  .......

  /* 遍历所有需要按序输出的节点,并对其分别处理,最终此节点对应的代码会被输出到 汇编文件(asm_out_file) */
  for (i = 0; i < max; ++i) {
    switch (nodes[i].kind) {
     case ORDER_FUNCTION:
      nodes[i].u.f->process = 0;
      nodes[i].u.f->expand ();    /* 对于当前编译单元内定义的按照顺序输出的函数, 则在这里调用 cgraph_node::expand 进行rtl展开以及最终汇编代码输出 */
      break;
     case ORDER_VAR:
      nodes[i].u.v->assemble_decl ();    /* 对于当前编译单元内定义的按照顺序输出的变量节点,则在这里进行rtl展开及汇编代码输出 */
      break;
     case ORDER_VAR_UNDEF:
      assemble_undefined_decl (nodes[i].u.v->decl);    /* 对于非当前编译单元内定义的变量,在aarch64平台此函数不做任何操作 */
      break;
     case ORDER_ASM:
      assemble_asm (nodes[i].u.a->asm_str);            /* 对于全局汇编节点,则原样输出其字符串 */
      break;
     case ORDER_UNDEFINED:               /* 对于如未定义函数, 则不处理直接返回 */
       break;
     default:
      gcc_unreachable ();
    }
  }
  symtab->clear_asm_symbols ();         /* 全局汇编节点已经处理完毕,清除后返回 */
  free (nodes);
}
```

由上可知,在所有的IPA_PASS执行完毕后, 会遍历当前遍历单元中所有需要输出的函数,变量等节点:

- 对于当前编译单元内定义的函数节点,最终通过 cgraph_node::expand 函数展开为rtl,并最终输出到汇编代码
- 对于当前编译单元内定义的变量节点,最终通过 varpool_node::assemble_decl 函数展开为rtl, 并最终输出到汇编代码

二者最终都会为此声明节点生成rtl表达式,并根据此表达式将此声明节点的代码输出到汇编文件; **但对于函数定义来说,由于涉及到函数体的展开,故整个过程是通过一个pass链表(all_passes)中的一系列pass来完成的; 而对于变量声明则是可以简单的通过assemble_variable一个函数来完成**.

四、函数的rtl expand和汇编输出——cgraph_node::expand

对于函数来说由于其函数体本身也要展开,其中还涉及优化等一系列的操作,故函数处理的整个过程是通过 all_passes中的一个个pass来完成的, rtl expand 和 最终的汇编输出则都只是all_passes链表中的一个pass:

```
void cgraph_node::expand (void)
{
  if (native_rtl_p ()) return;      /* 若此函数在源码中本来就是rtl函数,则在源码解析阶段就已经处理了, 这里不处理直接返回 */
  gcc_assert (lowered);             /* 确保当前函数已经做过了gimple低端化处理 */
  ......
  push_cfun (DECL_STRUCT_FUNCTION (decl));        /* expand的过程是针对具体函数的,这里先将当前函数设置到cfun和 current_function_decl中 */

  init_function_start (decl);                     /* rtl expand之前的初始化 */
  ......
  invoke_plugin_callbacks (PLUGIN_ALL_PASSES_START, NULL);    /* all_passes 执行之前的 plugin hook */

  execute_pass_list (cfun, g->get_passes ()->all_passes);     /* 执行all_pass中的所有pass, rtl_expand 到汇编代码的输出都是在执行此pass链表的过程中完成

  invoke_plugin_callbacks (PLUGIN_ALL_PASSES_END, NULL);      /* all_passes 执行结束的 plugin hook */
  ......

  gcc_assert (TREE_ASM_WRITTEN (decl));                       /* 确保当前函数已经正常处理完成 */

  if (cfun) pop_cfun ();         /* 恢复原有cfun current_function_decl上下文*/
  ......
}
```

all_pass中的一些重要pass记录如下:

```
INSERT_PASSES_AFTER (all_passes)
    NEXT_PASS (pass_fixup_cfg);
    ......                                          /* 这里有一系列 gimple pass 先忽略 */
    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 */
    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
    NEXT_PASS (pass_rest_of_compilation);
        NEXT_PASS (pass_instantiate_virtual_regs);  /* 此pass负责将虚拟寄存器转换为硬件寄存器 */
        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 */
        ........
        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) */
        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 */
        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
        ......
        NEXT_PASS (pass_thread_prologue_and_epilogue); /* 此pass负责发射函数的prologue/epilogue指令 */
        ......
        NEXT_PASS (pass_reorder_blocks);            /* pass "bbro", 此pass负责基本块顺序的的重排 */
        ......
        NEXT_PASS (pass_sched2);                    /* 统一寄存器分配之后再次执行指令调度(重排) */
        ......
        NEXT_PASS (pass_late_compilation);
            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 */
            NEXT_PASS (pass_machine_reorg);         /* 平台相关pass,  在arm64平台没有开启  */
            ......
            NEXT_PASS (pass_final);                 /* pass "final", 此pass负责此函数最终的汇编代码输出 */
            ......
            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 */
```

```
    在上述pass链表中:
    * pass_expand负责将函数的gimple指令序列转换为rtl指令序列
    * pass_final 负责将最终函数的rtl指令序列转换为具体汇编代码并输出到汇编文件(asm_out_file)中
```

五、变量的 rtl expand和汇编输出——varpool_node::assemble_decl

对于变量声明来说其处理过程比较简单, rtl expand和最终汇编的输出通过函数 assemble_variable即可完成:

```
bool varpool_node::assemble_decl (void)
{
  .......

  if (DECL_HARD_REGISTER (decl)) return false;    /* 硬件寄存器声明是不输出任何信息的 */

  if (!in_other_partition && !DECL_EXTERNAL (decl))    /* 非外部声明的全局声明都在这里做rtl expand 并最终输出到汇编文件 */
    {
      ......
      assemble_variable (decl, 0, 1, 0);              /* 这里负责对变量声明做rtl expand,并最终输出到汇编文件 */
      gcc_assert (TREE_ASM_WRITTEN (decl));           /* 确保当前变量已经正常处理完成 */
      ......
      return true;
    }
  return false;
}
```

其中assemble_variable:

```
void assemble_variable (tree decl, int top_level ATTRIBUTE_UNUSED, int at_end ATTRIBUTE_UNUSED, int dont_output_data)
{
  ......
  gcc_assert (VAR_P (decl));            /* 确保这是个变量节点 */

  if (DECL_EXTERNAL (decl))  return;    /* 外部声明在汇编代码中是不需要任何内容的,直接返回 */
  ......
  decl_rtl = DECL_RTL (decl);           /* 为变量生成对应的RTL节点(若没有),对于全局变量来说,其rtl表达式是一个代表其符号引用的mem表达式 (mem (symbol_ref

  TREE_ASM_WRITTEN (decl) = 1;          /* 标记此变量已经处理过了 */

  gcc_assert (MEM_P (decl_rtl));        /* 确保当前全局声明的rtl表达式是一个 (mem (symbol_ref name)) */
  gcc_assert (GET_CODE (XEXP (decl_rtl, 0)) == SYMBOL_REF);

  symbol = XEXP (decl_rtl, 0);          /* 从全局声明的rtl表达式中获取其 symbol_ref节点 */
  ......
  name = XSTR (symbol, 0);              /* 从 rtl(SYMBOL_REF)表达式中获取此全局变量的符号名字符串 */
  ......
  if ((flag_sanitize & SANITIZE_ADDRESS) && asan_protect_global (decl)) ......    /* AddressSanitizer 在这里有hook */

  align = get_variable_align (decl);    /* 获取此声明的内存对齐粒度 */

  sect = get_variable_section (decl, false);    /* 通过分析属性和声明类型来判断当前变量应该分配到哪个段(如 comm_section代表 common段),并将其返回到 sect

  /* 对于非common段的 public变量, 则这里会在源码中输出入 .global xxx 的语句 */
  if (TREE_PUBLIC (decl) && (sect->common.flags & SECTION_COMMON) == 0)
    globalize_decl (decl);
```

```
  ......
  if (sect && (sect->common.flags & SECTION_CODE) != 0)    /* 若变量最终被分配到了代码段,则这里会对其做标记(正常变量是要放到数据段的) */
    DECL_IN_TEXT_SECTION (decl) = 1;
  ......
  if(...
  /* 若变量被分配到不可切换段(tls_comm_section/lcomm_section/comm_section/bss_noswitch_section),则这里直接输出如 .comm xxx,4,4 的汇编代码 */
  else if (SECTION_STYLE (sect) == SECTION_NOSWITCH)
    assemble_noswitch_variable (decl, name, sect, align);
  else {                          /* 若变量分配到了可切换的段,则走这里 */
    ......
    switch_to_section (sect);    /* 先输出段切换汇编指令,在汇编代码中则代表当前已切换到了变量所在的段, 如输出 .bss/.data 分别代表后续变量会被分配到bss/da

    if (align > BITS_PER_UNIT)   /* 输出变量的对齐要求, 如 .align 2 */
       ASM_OUTPUT_ALIGN (asm_out_file, floor_log2 (align / BITS_PER_UNIT));
    /*
       在汇编中输出此变量的定义,如:
            .type   yyyy, %object
            .size   yyyy, 4
        yyyy:
            .zero   4
    */
    assemble_variable_contents (decl, name, dont_output_data,(sect->common.flags & SECTION_MERGE) && (sect->common.flags & SECTION_STRINGS));

    if (asan_protected) ......    /* AddressSanitizer 在这里有代码 */
  }
}
```