기반 작업 (7)
Struct function 이란?

정원교

2004년 5월 4일

목 차

제 1 절 20 주 문서를 시작하며

이번 문서에서는 struct function 에 대해 간단하게 언급한 후 넘어가도록 하겠습니다.

제 2 절 struct function

struct function 구조체는 GCC 가 C 언어의 한 함수를 처리하기 위해, 그 함수에 대한 정보를 담기 위해 가지는 구조체이다. GCC 는 현재 하나의 함수를 기준으로 컴파일이 진행되는데, *cfun 이라는 전역 변수가 현재 GCC 가 처리하고 있는 함수에 대한 정보를 가지고 있다.

```
struct function *cfun;
```

이 구조체는 GCC 내에 포함되어 있는 다른 구조체 보다 큰 구조체이며, 그를 구성하는 요소들이 상당히 많고 포함하는 정보 또한 많다. 즉, 이 구조체는 현재 함수의 상태를 설명하는 모든 중요한 global, static 변수들을 저장할 수 있다. 우선 이 구조체의 전체적인 구조체를 살펴 본 후 내부 요소에 대해서 하나 하나 설명해 나가는 방식으로 나아가겠다.

```
struct function
{
  struct eh_status *eh;
  struct stmt_status *stmt;
  struct expr_status *expr;
  struct emit_status *emit;
  struct varasm_status *varasm;

  const char *name;

  tree decl;
  struct function *outer;

  int pops_args;
  int args_size;
  int pretend_args_size;
  int outgoing_args_size;
  rtx arg_offset_rtx;

  CUMULATIVE_ARGS args_info;

  rtx return_rtx;
  rtx internal_arg_pointer;

  const char *cannot_inline;

  struct initial_value_struct *hard_reg_initial_vals;

  int x_function_call_count;

  tree x_nonlocal_labels;

  rtx x_nonlocal_goto_handler_slots;
  rtx x_nonlocal_goto_handler_labels;
  rtx x_nonlocal_goto_stack_level;
  rtx x_cleanup_label;
```

```
rtx x_return_label;
rtx x_save_expr_regs;
rtx x_stack_slot_list;
tree x_rtl_expr_chain;
rtx x_tail_recursion_label;
rtx x_tail_recursion_reentry;
rtx x_arg_pointer_save_area;
rtx x_clobber_return_insn;

HOST_WIDE_INT x_frame_offset;

tree x_context_display;
tree x_trampoline_list;

rtx x_parm_birth_insn;
rtx x_last_parm_insn;

unsigned int x_max_parm_reg;

rtx *x_parm_reg_stack_loc;

struct temp_slot *x_temp_slots;

int x_temp_slot_level;
int x_var_temp_slot_level;
int x_target_temp_slot_level;

struct var_refs_queue *fixup_var_refs_queue;

int inlinable;
int no_debugging_symbols;

void *original_arg_vector;

tree original_decl_initial;

rtx inl_last_parm_insn;

int inl_max_label_num;

int profile_label_no;

struct machine_function *machine;

int stack_alignment_needed;
int preferred_stack_boundary;

struct language_function *language;

rtx epilogue_delay_list;

unsigned int returns_struct : 1;
```

```
  unsigned int returns_pcc_struct : 1;
  unsigned int returns_pointer : 1;
  unsigned int needs_context : 1;
  unsigned int calls_setjmp : 1;
  unsigned int calls_longjmp : 1;
  unsigned int calls_alloca : 1;
  unsigned int calls_eh_return : 1;
  unsigned int has_nonlocal_label : 1;
  unsigned int has_nonlocal_goto : 1;
  unsigned int contains_functions : 1;
  unsigned int has_computed_jump : 1;
  unsigned int is_thunk : 1;
  unsigned int instrument_entry_exit : 1;
  unsigned int profile : 1;
  unsigned int limit_stack : 1;
  unsigned int varargs : 1;
  unsigned int stdarg : 1;
  unsigned int x_whole_function_mode_p : 1;
  unsigned int x_dont_save_pending_sizes_p : 1;
  unsigned int uses_const_pool : 1;
  unsigned int uses_pic_offset_table : 1;
  unsigned int uses_eh_lsda : 1;
  unsigned int arg_pointer_save_area_init : 1;
};
```

아래에서 각 요소들의 역활에 대해서 나열하도록 하겠다.

struct eh_status *eh;

각 함수를 위한 exception status 를 저장하는데 사용된다.

struct stmt_status *stmt;

각 함수를 위한 Statment status 를 저장하는데 사용되며, 다른 섹션에서 이 구조체에 대해서 더 자세히 다룰 것이다.

struct expr_status *expr;

각 함수를 위한 Expression status 를 저장한다. 다른 섹션에서 이 구조체에 대해서 더 자세히 다룰 것이다.

struct emit_status *emit;

각 함수를 위한 명령어 방출 관련 정보를 저장하고 있다. 다른 섹션에서 이 구조체에 대해서 더 자세히 다룰 것이다.

struct varasm_status *varasm;

아직 정확한 설명이 없음.

const char *name;

이 함수의 이름.

tree decl;

이 함수의 FUNCTION_DECL 를 가르킨다.

struct function *outer;

존재할 경우, 이 함수를 포함하는 function.

int pops_args;

함수의 return 으로 컴파일된 것으로, 함수에 의해 pop 되어지는 arg 들의 byte 들의 number. pop 되어지는 바이트가 없을 경우 값 0 을 가집니다.
Return insn 의 혹은 function epilogue 의 compilation 이 영향을 받을 수 있다.

int args_size;

만약 function 의 arg 가 fixed size 를 가지고 있다면, 이것은 바이트 크기로의 바로 그 크기이며, 그렇지 않을 경우, -1 이다.
Return insn 의 혹은 function epilogue 의 compilation 이 영향을 받을 수 있다.

int pretend_args_size;

Prologue 가 push 해야 하고, caller 가 그들을 push 했다고 가정해야하는 # byte 들. Prologue 는 반드시 이것을 해야하지만, 만약 parm 들이 register 를 통해 건네질 수 있을 경우에만 해당한다.

int outgoing_args_size;

Outgoing argument 들의 byte 들의 #. 만약 ACCUMULATE_OUTGOING_ARGS 가 정의되어 있다면, 필요한 공간이 prologue 에 의해 push 됩니다.

rtx arg_offset_rtx;

이것은 arg pointer 부터 첫번째 anonymous arg 가 발견된 장소까지의 offset 이다.

CUMULATIVE_ARGS args_info;

현재 함수의 arg 들에 사용되는 여러 종류의 register 들의 용량.

rtx return_rtx;

만약 0 이 아닐 경우, 현재 function 이 그것의 result 를 반환하는 위치에 대한 RTL 표현식이다. 만약 현재 함수가 그것의 result 를 register 내에 반환한다면, current_function_return_rtx 는 항상 result 를 포함하는 hard register 일 것이다.

rtx internal_arg_pointer;

arg pointer hard register 혹은 복사본이 포함된 pseudo.

const char *cannot_inline;

현재 함수가 왜 inline 화 할 수 없는지에 대한 language-specific 이유

struct initial_value_struct *hard_reg_initial_vals;

get_hard_reg_initial_val 와 has_hard_reg_initial_val 에 의해 사용되는 불투명한 (Opaque) pointer. (integrate.[hc] 를 보라.)

int x_function_call_count;

현재 함수내에서 지금까지 보였던 함수 호출 수.

tree x_nonlocal_labels;

이 함수내의 모든 nonlocal label 들 (nested function 들로 부터 nonlocal goto 로 갈 수 있는 label 들) 을 위한 LABEL_DECL 들의 list (TREE_LIST 의 chain).

rtx x_nonlocal_goto_handler_slots;

Non-local goto 들을 위한 현재 handler 들을 잡고 있는 stack slot 들의 list (EXPR_LIST 의 chain). 함수내의 모든 nonlocal label 를 위한 것이 존재합니다; 이 list 는 nonlocal_labels 내의 하나와 매칭합니다. 함수가 nonlocal label 들을 가지고 있지 않을 때 0 입니다.

rtx x_nonlocal_goto_handler_labels;

Nonlocal goto 들을 위한 현재 handler 들을 이끄는 label 의 List (EXPR_LIST 의 연결).

rtx x_nonlocal_goto_stack_level;

Non-local goto 에 대하여 복구할 stack pointer 의 값을 가지고 있는 stack slot 을 위한 RTX. 함수가 nonlocal label 들을 가지고 있지 않다면 0 입니다.

rtx x_cleanup_label;

존재할 경우, Parm cleanup code 가 자리잡을 label.
Parameter 들을 위한 cleanup code 를 실행하기 위해서는 이 label 로 jump 하면 되는데, 그럴 경우 code 는 반드시 실행되어야 한다. 이 code 뒤에 오는 것은 logical return label 이다.

rtx x_return_label;

Function eiplogue 가 자리 잡을 label.
이 label 로의 jump 는 모든 return 들에서 epilogue 의 실행을 요구하는 machine 들 상에서 “return” instruction 로써 작동한다.

rtx x_save_expr_regs;

SAVE_EXPR 의 pseudo-reg 들의 list (EXPR_LIST 들의 chain). 그래서 만약 nonopt 이면 우리는 함수의 끝에 모두 live 로 mark 할 수 있습니다.

rtx x_stack_slot_list;

이 함수내에서의 모든 stack slot 들에 관한 list (EXPR_LIST 들의 chain). unshare_all_rtl 의 이익을 위해 만듬.

tree x_rtl_expr_chain;

RTL_EXPR 들내 insn 를 가지고 있는 모든 RTL_EXPR 들의 chain

rtx x_tail_recursion_label;

Tail recursion 에서 뒤로 jump 해야하는 label, 만약 이 함수에 대해 이것이 아직 필요하지 않는다면 값이 0 이다.

rtx x_tail_recursion_reentry;

필요할 경우, tail_recursion_label 를 삽입하기 위한 곳 뒤에 자리잡는다.

rtx x_arg_pointer_save_area;

> 만약 참고할 필요가 있을 경우, argument pointer 를 저장하는 곳의 위치. 이것이 수행되는 두 경우가 존재한다: 만약 nonlocal goto 들이 존재할 경우, 혹은 argument pointer 로 부터 offset 에 저장된 var 들이 inner routine 들에 의해 필요하게 될 경우이다.

rtx x_clobber_return_insn;

> 만약 function 이 non-void 를 반환할 경우, 사용자가 적당한 값을 반환하는 것 없이 끝부분을 자를 경우에만 return register 그룹의 clobber 를 emit 한다. 이것이 그 insn 이다.

HOST_WIDE_INT x_frame_offset;

> Stack frame 에 할당된 area 의 끝 부분 관련 offset.
> 만약 stack 이 점점 작아진다면, 이것은 마지막으로 할당된 stack slot 의 주소이다.
> 만약 stack 이 점점 커진다면, 이것은 다음 slot 에의 address 이다.

tree x_context_display;

> Containing function 들을 위한 static chain 들의 목록 (TREE_LIST 들의 연결). 각 link 는 TREE_PURPOSE 내 하나의 FUNCTION_DECL 를 가지고 있고, TREE_VALUE 내 RTL_EXPR 에는 reg rtx 를 가지고 있다.

tree x_trampoline_list;

> Nested function 들을 위한 trampoline 들의 list. (TREE_LIST 들의 chain). Trampoline 는 static chain 을 설정하고 function 으로 jump 한다. 우리는 function 의 주소가 요구될 때 trampoline 의 주소를 공급한다.
>
> 각 link 는 TREE_PURPOSE 내 하나의 FUNCTION_DECL 를 가지고 있고, TREE_VALUE 내 RTL_EXPR 에는 reg rtx 를 가지고 있다.

rtx x_parm_birth_insn;

> 만약 nonopt 일 경우, Register parm 들과 SAVE_EXPR 들이 태어났던 곳 뒤의 insn.

rtx x_last_parm_insn;

> 일거리가 그들의 nominal home 내에 parm 들을 놓는 것이었던 어떤 마지막 insn.

unsigned int x_max_parm_reg;

> 1 + 마지막 pseudo register number 는 이 함수의 parameter 의 복사본을 loading 하는데 사용될 가능성이 있습니다.

rtx *x_parm_reg_stack_loc;

> REGNO 에 따라 나열된 vector. 만약 우리가 그 parm 이 반드시 stack 내에 들어가야 한다는 것을 발견할 경우 이것은 명목상 pseudo register REGNO 내에 자리잡을 parm 을 stack 에 넣게 되는데, 이에 따른 위치를 포함한다. 이 vector 내 가장 큰 element 는 위의 MAX_PARM_REG 보다 작을 것이다.

struct temp_slot *x_temp_slots;

> 할당된 모든 temporary 들의 list, 이용 가능한 거와 사용중인 거 양쪽 다 가지고 있습니다.

int x_temp_slot_level;

temporary 들을 위한 현재 nesting level.

int x_var_temp_slot_level;

Block 내의 variable 들을 위한 현재 nesting level.

int x_target_temp_slot_level;

TARGET_EXPR 들에 의해 임시적인 것이 생성될 때, 그들은 temp_slot_level 의 레벨에서 생성되는데, 그래서 그들이 더 이상 필요없을 때까지 할당된 것을 남겨놓을 수 있다.
CLEANUP_POINT_EXPR 들은 TARGET_EXPR 들의 lifetime 을 정의합니다.

struct var_refs_queue *fixup_var_refs_queue;

이 slot 은 0 으로 초기화되며 nested function 동안 추가된다.

int inlinable;

integrate.c 를 위해.

int no_debugging_symbols;

integrate.c 를 위해.

void *original_arg_vector;

이것은 사실 rtvec 이다.

tree original_decl_initial;

아직 정확한 설명이 없음

rtx inl_last_parm_insn;

일거리가 그들의 nominal home 내에 parm 들을 놓는 것이었던 어떤 마지막 insn.

int inl_max_label_num;

현재 function 내 가장 큰 label number.

int profile_label_no;

Profile label number.

struct machine_function *machine;

Stack 상에 할당된 가장 큰 slot 의 alignment.

int stack_alignment_needed;

Stack 상에 할당된 가장 큰 slot 의 alignment.

int preferred_stack_boundary;

Stack frame 의 끝의 우선시되는 alignment.

struct language_function *language;

Language-specific code 는 이것을 마음대로 저장하는데 사용할 수 있다.

rtx epilogue_delay_list;

만약 몇몇 insn 들이 epilogue 의 delay slot 에 연기될 수 있다면, 그것들을 위한 delay list 가 여기에 기록됩니다.

unsigned int returns_struct : 1;

만약 컴파일된 function 이 값이 저장되어야 하는 address 가 주어질 필요가 있을 경우 0 이 아닌 값.

unsigned int returns_pcc_struct : 1;

만약 컴파일 진행중인 함수가 structure value 를 어디에 놓는지에 대한 address 정보를 반환할 필요가 있다면 값이 0 이 아님.

unsigned int returns_pointer : 1;

만약 현재 function 이 pointer type 을 반환할 경우 0 이 아닌 값.

unsigned int needs_context : 1;

만약 컴파일된 function 이 static chain 을 건네질 필요가 있을 경우 0 이 아닌 값.

unsigned int calls_setjmp : 1;

만약 컴파일되는 중인 함수가 setjmp 를 call 할 수 있다면 값이 0 이 아님.

unsigned int calls_longjmp : 1;

만약 컴파일되는 중인 함수가 longjmp 를 call 할 수 있다면 값이 0 이 아님.

unsigned int calls_alloca : 1;

만약 컴파일되는 중인 함수가 alloca 를 subroutine 이든 builtin 이든 호출 할 수 있으면 값이 0 이 아님.

unsigned int calls_eh_return : 1;

만약 함수가 __builtin_eh_return 를 호출한다면 0 이 아닌 값.

unsigned int has_nonlocal_label : 1;

만약 컴파일 진행중인 함수가 nested function 으로부터 non-local goto 를 받았을 때, 0 이 아닌 값을 가짐.

unsigned int has_nonlocal_goto : 1;

만약 컴파일 진행중인 함수가 부모 함수로의 non-local goto 들을 가지고 있다면, 0 이 아닌 값을 가짐.

unsigned int contains_functions : 1;

만약 컴파일 진행중인 함수가 nested function 들을 포함한다면 값이 0 이 아님.

unsigned int has_computed_jump : 1;

만약 컴파일된 function 이 computed jump 를 발생한다.

unsigned int is_thunk : 1;

만약 현재 함수가 thunk (전달된 argument 중 하나를 고치고 가른 함수로 전달하기만 하는 lightweight function) 라서 우리가 찾을 수 있는 귀퉁이들을 짜르길 시도할려면 값이 0 이 아님.

unsigned int instrument_entry_exit : 1;

만약 function entry 와 exit 를 위한 instrumentation call 들이 생성되어야 한다면 0 이 아닌 값.

unsigned int profile : 1;

만약 Profiling code 가 생성되어야 한다면 0 이 아닌 값.

unsigned int limit_stack : 1;

만약 stack limit checking 이 현재 function 내에서 활성화되어야 한다면 0 이 아닌 값.

unsigned int varargs : 1;

만약 현재 function 이 varargs.h 혹은 동일한 것을 사용할 경우 값이 0 이 아님. stdarg.h 를 사용하는 funztion 들에 대해서는 값이 0.

unsigned int stdarg : 1;

만약 현재 function 이 stdarg.h 혹은 동일한 것을 사용할 경우 값이 0 이 아님. varargs.h. 를 사용하는 funztion 들에 대해서는 값이 0.

unsigned int x_whole_function_mode_p : 1;

만약 이 function 이 function-at-a-time mode 로 처리되고 있다면 0 이 아닌 값. 다른 말로 해서, 이 function 을 위한 모든 tree structure , BLOCK tree 를 포함해서, 가 RTL generation 이 시작되기 전에 생성된 것을 말한다.

unsigned int x_dont_save_pending_sizes_p : 1;

만약 back-end 가 variable-sized object 들의 크기를 결정하는 표현식들의 나열을 유지할 필요가 없을 경우 0 이 아닌 값. 보통 그러한 표현식들은 어쨌든 저장되고 다음 function 이 시작될 때 확장되어지는 것이 보통이다. 예를 들어, 만약 parameter 가 variable-sized type 을 가지고 있다면, parameter 의 크기는 function body 에 들어갔을 때 계산된다. 하지만 몇몇 front-end 들은 그 행동을 요구하지 않기도 한다.

unsigned int uses_const_pool : 1;

만약 현재 함수가 constant pool 를 사용한다면 값이 0 이 아님.

unsigned int uses_pic_offset_table : 1;

만약 현재 함수가 pic_offset_table_rtx 를 사용한다면 값이 0 이 아님.

unsigned int uses_eh_lsda : 1;

만약 현재 function 이 exception handling 를 위한 lsda 가 필요할 경우 0 이 아닌 값.

unsigned int arg_pointer_save_area_init : 1;

만약 arg_pointer_save_area 를 초기화하기 위한 code 가 이미 emit 되었다면 0 이 아닌 값.

2.1 struct eh_status

```
struct eh_status
{
  struct eh_region *region_tree;
  struct eh_region **region_array;
  struct eh_region *cur_region;
  struct eh_region *try_region;
  tree protect_list;

  rtx filter;
  rtx exc_ptr;

  int built_landing_pads;
  int last_region_number;

  varray_type ttype_data;
  varray_type ehspec_data;
  varray_type action_record_data;

  struct call_site_record
  {
    rtx landing_pad;
    int action;
  } *call_site_data;
  int call_site_data_used;
  int call_site_data_size;

  rtx ehr_stackadj;
  rtx ehr_handler;
  rtx ehr_label;

  rtx sjlj_fc;
  rtx sjlj_exit_after;
};
```

struct eh_region *region_tree;

이 함수를 위한 모든 region 들의 tree.

struct eh_region **region_array;

Indexable array 와 같은 정보.

struct eh_region *cur_region;

가장 최근 open region.

struct eh_region *try_region;

이것은 우리가 처리하고 있는 catch block 들을 위한 region 이다.

tree protect_list;

Handler 들의 list 들의 stack (TREE_LIST). 각 node 의 TREE_VALUE 는 아직 close 되지 않은 region 들을 위한 handler 들의 TREE_CHAIN 화된 list 자신이다. 각 entry 의 TREE_VALUE 는 ehstack 상 대응하는 entry 를 위한 handler 를 포함하고 있다.

2.2 struct stmt_status

```
struct stmt_status
{
  struct nesting *x_block_stack;
  struct nesting *x_stack_block_stack;
  struct nesting *x_cond_stack;
  struct nesting *x_loop_stack;
  struct nesting *x_case_stack;
  struct nesting *x_nesting_stack;

  int x_nesting_depth;
  int x_block_start_count;

  tree x_last_expr_type;
  rtx x_last_expr_value;

  int x_expr_stmts_for_value;

  const char *x_emit_filename;
  int x_emit_lineno;

  struct goto_fixup *x_goto_fixup_chain;
};
```

struct nesting *x_block_stack;

모든 pending binding contour 들의 chain.

struct nesting *x_stack_block_stack;

Stack level 들을 복구하거나 cleanup 들을 가지는 모든 pending binding contour 들의 chain.

struct nesting *x_cond_stack;

모든 pending conditional statement 들에 대한 chain.

struct nesting *x_loop_stack;

모든 pending loop 들에 대한 chain.

struct nesting *x_case_stack;

모든 pending case 혹은 switch statement 들에 대한 chain.

struct nesting *x_nesting_stack;

위의 모든 것을 포함하는 분리된 chain. 'all' field 를 통해 chain 되어 있다.

int x_nesting_depth;

지금 nesting_stack 상에 있는 entry 들의 갯수.

int x_block_start_count;

이 함수내에서 지금까지 시작된 binding contour 들의 갯수.

tree x_last_expr_type;

매번 우리는 expression-statement 를 expand 하고, expr 의 type 과 그것의 RTL 값을 여기 기록합니다.

rtx x_last_expr_value;

아직 정확한 설명이 없음.

int x_expr_stmts_for_value;

각 expr-stmt 가 마지막일 경우 항상 그 값을 반드시 계산해야 하는 (...) grouping 내에 있다면 0 이 아닌 값을 가짐.

const char *x_emit_filename;

우리가 실제로 그것을 emit 했든 안했든, 마지막 line-number note 의 파일 이름과 줄번호.

int x_emit_lineno;

우리가 실제로 그것을 emit 했든 안했든, 마지막 line-number note 의 파일 이름과 줄번호.

struct goto_fixup *x_goto_fixup_chain;

아직 정확한 설명이 없음

2.3 struct expr_status

```
struct expr_status
{
  int x_pending_stack_adjust;
  int x_inhibit_defer_pop;
  int x_stack_pointer_delta;

  rtx x_saveregs_value;
  rtx x_apply_args_value;
  rtx x_forced_labels;
  rtx x_pending_chain;
};
```

int x_pending_stack_adjust;

Stack 상에서 우리가 결국 pop off 시켜야 하는 unit 들의 갯수. 이것은 이미 반환된 function call 들로의 argument 들이다.

int x_inhibit_defer_pop;

몇몇 ABI 하에서는 function call 들을 위해 push 된 argument 들을 pop 하는 것은 caller 의 책임입니다. Naive implementation 는 단순히 각 call 후에 즉히 argument 들을 모두 pop 합니다. 하지만, 만약 여러 function call 들이 한 줄로 구성된다면, 모든 call 들이 완료한 후 모든 argument 들을 pop 하는 것이 전통적으로 비용이 싼데, 그것은 single pop instruction 이 사용될 수 있기 때문이다. 그렇기 때문에 GCC 는 절대적으로 pop 할 필요가 있을 때 까지 argument 들을 pop 하는 것을 연기하고자 한다. (예를 들면, 조건분의 끝에서, argument 들은 반드시 pop 되어야 하는데, 조건분 바깥의 code 가 argument 들이 pop 될 필요가 있는지 아닌지를 알지 못하기 때문이다.)

하지만, INHIBIT_DEFER_POP 가 0 이 아닐 경우, 컴파일러는 pops 를 연기할 시도를 하지 않는다. 대신, stack 은 각 call 후에 즉시 pop 되어진다. 직접적으로 이 변수를 세팅하기 보다는 NO_DEFER_POP 와 OK_DEFER_POP 를 사용하라.

int x_stack_pointer_delta;

만약 PREFERRED_STACK_BOUNDARY 와 PUSH_ROUNDING 가 정의되어 있다면 Stack boundary 가 argument 들을 push 하는 동안 순간적으로 unaligned 될 수 있습니다. Nested function call 들이 올바르게 작동하기 위해, stack alignment 를 얻는 과정에서 여기에 최근 있었던 aligned boundary 뒤에 delta 를 기록한다.

rtx x_saveregs_value;

값이 0 이 아닐 경우 __builtin_saveregs 가 이 함수내에서 이미 수행 되었음을 의미합니다. 값은 반환된 값 __builtin_saveregs 을 포함하는 pseudoreg 이다.

rtx x_apply_args_value;

__builtin_apply_args 와 비슷합니다.

rtx x_forced_labels;

절대 삭제되어서는 안되는 label 들의 list.

rtx x_pending_chain;

여전히 Expand 될 필요가 있는 postincrement 들.

2.4 struct emit_status

```
struct emit_status
{
  int x_reg_rtx_no;
  int x_first_label_num;

  rtx x_first_insn;
  rtx x_last_insn;

  tree sequence_rtl_expr;

  struct sequence_stack *sequence_stack;

  int x_cur_insn_uid;
  int x_last_linenum;
  const char *x_last_filename;

  int regno_pointer_align_length;
  unsigned char *regno_pointer_align;

  tree *regno_decl;

  rtx *x_regno_reg_rtx;
};
```

int x_reg_rtx_no;

이 변수는 각 함수의 시작 부분에서 LAST_VIRTUAL_REGISTER + 1 로 재설정됩니다. rtl 생성 후, 사용된 가장 큰 register number 에 1 을 더합니다.

int x_first_label_num;

현재 함수내에서의 lowest label number.

rtx x_first_insn;
rtx x_last_insn;

현재 함수를 위한 rtl 의 이중 연결 chain 의 양끝. 아래 둘다 함수를 위한 rtl 생성의 시작 부분에서 NULL 로 재설정됩니다.

start_sequence 는 'sequence_rtl_expr' 에 따라 'sequence_stack' 에 이것들을 모두 저장하고 새로운 것, insn 들내 nested sequence, 을 시작한다.

tree sequence_rtl_expr;

현재 sequence 가 어디에 놓일지에 대한 RTL_EXPR. RTL_EXPR 가 emit 되는 후 까지 sequence 내 어떠한 temporary 들의 재사용을 막는데 사용한다.

struct sequence_stack *sequence_stack;

'start_sequence' 에 의해 저장된 pending (불완전한) sequence 들의 stack. 각 element 는 하나의 pending sequence 를 표현합니다. Main insn-chain 은 chain 이 비어있지 않을 경우, chain 의 마지막 element 에 저장됩니다.

int x_cur_insn_uid;

emit 되는 다음 insn 를 위한 INSN_UID. 각 함수가 컴파일될 때 마다 1 로 reset 됩니다.

int x_last_linenum;
const char *x_last_filename;

Emit 된 마지막 line-number NOTE 의 line number 와 source file. 이 변수는 중복 생성을 피하는데 사용됩니다.

int regno_pointer_align_length;

regno_pointer_align 과 regno_decl, x_regno_reg_rtx 벡터들의 길이. 이 vector 들은 expansion phase 가 이루어지는 동안 function 내 모든 register 들의 갯수가 아직 알려지지 않을 때, 필요하기 때문에 vector 들은 필요할 때 복사되거나 커진다.

unsigned char *regno_pointer_align;

Pseudo register number 순으로 나열되어 있으며 값이 0 이 아닐 경우 해당 pseudo 에 대한 알려진 alignment 를 줍니다. (만약 REG_POINTER 가 x_regno_reg_rtx 내에 설정되어 있을 경우). x_regno_reg_rtx 와 평형으로 할당됩니다.

tree *regno_decl;

Pseudo register number 순으로 나열되어 있으며, 만약 0 이 아닐 경우 그 register 와 대응하는 decl 을 줍니다.

rtx *x_regno_reg_rtx;

Pseudo register number 순으로 나열되어 있으며, 그 pseudo 를 위한 rtx 을 줍니다.
regno_pointer_align 와 평형으로 할당됩니다.

2.5 struct varasm_status

```
struct varasm_status
{
  struct constant_descriptor **x_const_rtx_hash_table;
  struct pool_constant **x_const_rtx_sym_hash_table;

  struct pool_constant *x_first_pool, *x_last_pool;

  HOST_WIDE_INT x_pool_offset;

  rtx x_const_double_chain;
};
```

struct constant_descriptor **x_const_rtx_hash_table;

> Constant rtl-expression 들로부터 memory-constant 들을 만들기 위한 hash facility. Immediate integer argument 들과 constant address 들이 제한되어서 그러한 constant 들이 반드시 메모리 내에 저장되어야 하는 RISC machine 상에서 사용된다.
>
> 이 constant 들의 poll 은 각 function 마다 재초기화되는데, 그래서 각 function 은 오기 바로 전 자신의 constants-pool 을 얻는다.

struct pool_constant **x_const_rtx_sym_hash_table;

struct pool_constant *x_first_pool, *x_last_pool;

> Pool 에서 처음과 끝 constant 로의 pointer 들.

HOST_WIDE_INT x_pool_offset;

> Constant pool (어떠한 machine-secific header 를 포함하고 있지 않는) 내에서의 현재 offset.

rtx x_const_double_chain;

> 현재 함수에서 구성된 모든 CONST_DOUBLE rtx 들의 chain.
> 그들은 CONST_DOUBLE_CHAIN 을 통해 chain 되어 있습니다.