기반 작업
(5) 언어 독립적인 부분의 초기화
(개요)

정원교
weongyo@hotmail.com

2004년 2월 23일

목 차

제 1 절 11 주째 강의를 시작하며

이번 주가 나오는데, 상당히 많은 시일이 걸렸습니다. 그 동안 회사일 때문에 상당히 바쁘게 지내왔는데, 계속 이와 같이 일을 하게 되면 강의를 쓰는데 많이 늦어질 듯합니다.

제 2 절 언어 독립접인 부분이란?

GCC 속에서 언어 독립적인 부분이라던가 언어 종속적인 부분과 같은 표현을 사용하게 된 이유는 GCC 의 원래 이름에서 알 수 있습니다. 즉 GNU Compiler Collection 이기 때문에 여러 언어가 GCC 라는 이름 하에 모여 살고 있습니다. 마치 건물 구조는 같지만 각 언어들이 살고 있는 방 내부 모습은 제각각인듯 말입니다. 여기서 방 내부 모습을 '언어 종속적인 부분' 이라고 생각한다면 건물 구조는 '언어 독립적인 부분'이라고 할 수 있을 것입니다. 건물 구조라는 말을 좀 풀어쓴다면 단순히 건물의 모습 뿐만 아니라 전선의 배치, 가스관의 위치 등등 여러 요소를 말할 수 있을 것입니다.

2.1 개요에 대해

이번 주는 "언어 독립적인 부분의 초기화 (개요)"에 대한 것을 하게 될 것입니다. 여기에서의 초기화의 의미는 지금까지 했던 강의와 마찬가지로 어떤 사물/사상을 담을 그릇을 미리 준비하고 닦아 놓는 과정을 말합니다. 어떤 주제에 대해서 심도있게 살펴보는 것은 나중으로 미루고 전체적인 모양새에 대해서 알아보는 과정을 하겠습니다.

아래 그림은 지금까지 GCC 모습을 간단하게 그린 것입니다. 'GCC 실행' 이 되면 '옵션 처리기' 에서 Command line 상에 입력된 option 들을 해석하게 되며 곧 이어서 '언어-독립적인 초기화' 와 '언어-종속적인 초기화'가 이루어지게 된다.

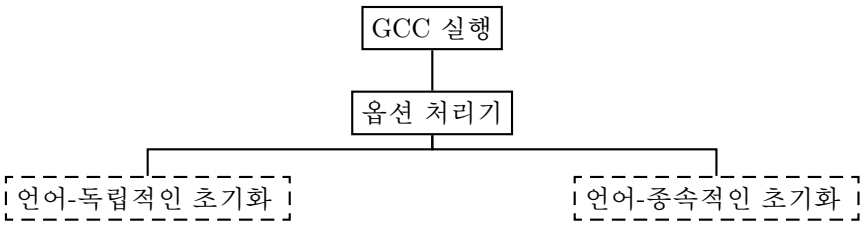

그림 1: 지금까지 살펴본 GCC 구조

이 부분은 컴파일러의 구조 중 가장 앞단에 있는 구조로써 거의 '컴파일러 구조' 를 언급할때 보이지도 않는 부분이다. 그럼 GCC 는 대략 어떤 구조를 가지고 있는지 모습만 보자. 아래 그림은 최형규씨의 글 "Porting GCC Compiler" 라는 문서에 나와 있는 것입니다.

하지만 우리가 지금 현재 보는 부분은 'For each function' 이전 부분이라고 보시면 됩니다.

제 3 절 언어 독립적인 부분

3.1 Garbage Collector

GCC 에서 처음 수행하는 언어 독립적인 부분은 Garbage-Collector (이하 GC) 부분이다. 물론 초기화 단계이기 때문에 GC 을 구현하는데 사용되는 구조체나 전역변수를 설정해 주는 부분이 전부이다.

GC 에 대한 정보를 담는 구조체 중 가장 중심에 있는 것은 $prefix/gcc/ggc-page.c 파일에 선언되어 있는

```
struct globals G;
```

구조체이다. GCC 에서 사용하는 GC 의 경우 paging 개념을 사용하기 때문에 구조체를 구성하는 구성요소들은 page_entry, page_table 과 같은 다른 구조체와 할당된 정보 등등의 것을 담고 있다.

이 구조체에 대한 자세한 설명은 나중에 설명할 것이다.

그리고 기타 다른 GC 관련 정보들을 담는 전역 변수들을 설정하게 됩니다. 여기서 설정되는 전역 변수로는

```
object_size_table objects_per_page_table size_lookup
```

가 있다. 이것으로써 GCC 내에 GC 구조체 및 전역변수 설정은 마치게 된다.

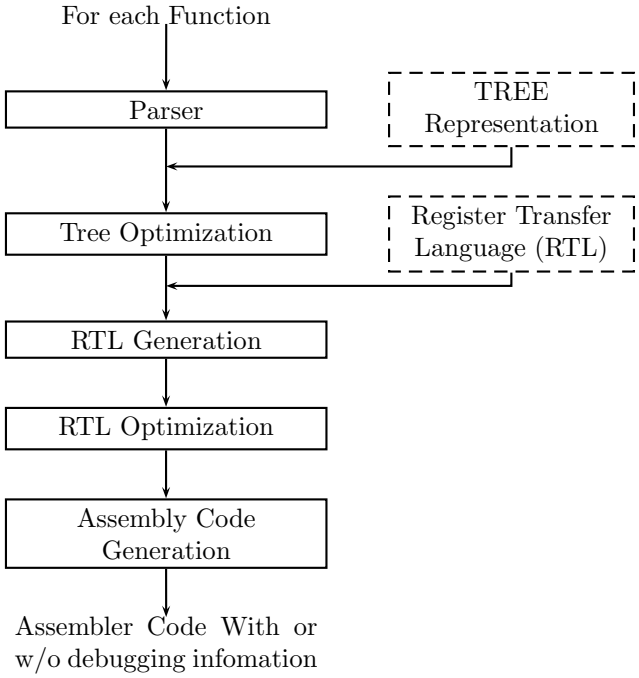

그림 2: 전체적인 GCC 구조

3.2 String Pool

String Pool 의 초기화는 다음과 같은 전역변수를 초기화하면서 간단하게 마치게 된다.

```
ident_hash
ident_hash->alloc_node
string_stack
```

String Pool 을 구성하는 구조체로는 struct ht 가 있는데, 이 구조체는 $prefix/gcc/hashtable.h 파일에 정의되어 있다. 이 구조체는 CPPlib (C++ 을 가르키는 것이 아니라 C PreProcessor 를 나타낸다.) 와 컴파일러에서 나타나게 될 식별자 (identifier) hash table 들을 가지고 있게 된다. 당연한 얘기이겠지만 식별자의 문자열을 저장하는 포인터, 식별자의 크기, 몇 개의 식별자가 저장되어 있는지 등등과 기타 테이블 사용에 관현 통계치를 보여주는 변수들로 구정되어 있다.

이 ident_hash 전역변수에 대한 할당은 ht_create () 함수에 의해 이루어지게 되며 현재 GCC 에서는 Hash Table 을 위한 크기로써 초기에 16Kbytes (2$\hat{1}$4) 를 할당하게 된다.

잠시 ht_create () 함수의 동작을 언급하고 다음으로 넘어 가겠습니다.

1. 우선 struct ht 구조체를 위한 공간을 할당합니다.
2. struct ht 구조체의 구성원인 stack 에 대한 것을 gcc_obstack_init () 함수를 사용하여 초기화하게 됩니다.
3. obstack_alignment_mask 매크로를 사용하여 이 stack 의 경우 문자열을 alignment 할 필요가 없음을 가르켜 줍니다.
4. 그리고 구성원인 entries 에 할당될 공간을 마련하고 구성원 nslots 에 현재 몇 개의 구성원이 할당되었는지를 저장합니다.

이제 또 다른 구성원인 alloc_node 를 설정하고, 전역 변수 string_stack 에 대해 gcc_obstack_init () 함수로 초기화를 시켜주게 됩니다.

마지막으로 ggc_add_root () 함수를 사용하여 indent_hash 전역 변수를 새로운 garbage collection root 로 추가합니다.

이로써 GCC 내의 String Pool 에 대한 초기화를 마루리짓게 됩니다.

GCC 에서 사용되는 hash table 구조체에 대한 자세한 설명은 다음에 알아보도록 하겠습니다.

3.3 Obstack

Obstack 이라는 말은 GCC 를 분석하다 보면 심심찮게 나오는 단어입니다. obstack 은 GCC 에서 $prefix/libiberty/obstack.h 파일과 $prefix/libiberty/obstack.c 파일들을 중심으로 구현되어 있으며, 많은 함수들과 매크로들로 구성되어 있습니다. Obstack 은 String table 을 생성 (creating) / 유지 (maintaining) 하는 간단한 방법을 제공합니다. 이것은 한 문자씩 (character-by-character) 문자열을 아주 빈번히 만들고, 때때로 그것을 유지보수 혹은 폐기할 필요가 있는 부분에 최적화가 되어 있습니다. Stack 형식이기 때문에 마지막에 할당된 것이 반드시 처음 나오는 것이 되도록 합니다. 하지만 다른것과 구분되는 '독립적인 obstack'의 경우 이러한 방식에 영향을 받을 필요가 없습니다.

GCC 코드중에서 특별히 obstack 을 초기화하는 부분은 없으며 (실제로는 $prefix/libiberty/obstack.* 파일에 초기화 함수가 구현되어 있다.) 실제로 init_obstacks () 함수에는 GCC 상에 선언된 $prefix/gcc/collect2.c 파일내의 permanent_obstack 전역변수를 초기화해주는 것만 obstack 부분과 연관이 있고 나머지 부분들은 약간 상관없는 것들이 있다. 큰 연관성이 없는 항목으로는 아래와 같은 것이 있다.

1. 'Type' Hash Table 을 담고 있는 type_hash_table 전역변수를 htab_create () 함수를 사용하여 초기화하고 이를 ggc_add_deletable_htab () 함수를 이용하여 'GC root 삭제가능 목록'에 등록합니다.
2. 별로 obstack 부분과 관련성이 없는 전역변수 global_trees 와 integer_types 를 GC root 로 등록합니다.
3. 함수포인터 lang_set_decl_assembler_name 를 set_decl_assembler_name 로 설정합니다.

GCC 내에 사용되는 **obstack**에 대한 자세한 설명은 이 강의 뒤에 마련될 것이다.

3.4 Unique RTL object 들

언어 독립적인 초기화 부분에서 init_emit_once () 함수에서 이루어지는 부분에 대해서 간단하게 살펴봅시다.

Unique RTL object 들을 생성한다는 것은 나중에 자주 사용되게 될 RTL object 들을 지금 미리 생성해 놓겠다는 뜻입니다. 여러분들이 RTL 이라는 개념에 대해서 익숙해져야 하겠지만 RTL 을 나중에 다루게 되면 그것이 GCC 가 컴파일되게 되는 machine 과 아주 많은 연관이 있습니다. 비록 사용자 입장에서 보면 단순히 C 로 짜여진 프로그램이겠지만 컴파일러 입장에서 보면 어떤 변수를 어떤 register 에 배당해야 하고 그 때 마다 PC (program counter) 는 어떤게 변경해줘야 하고, 등등 많은 부분에 대해서 컴파일러는 생각해줘야 합니다. Unique RTL object 생성부분에서는 1) 기런 기본이 되는 부분에 대해서 미리 생성함으로써 나중에 parsing 후 나타나게 될 비슷비슷한 부분에 대한 중복된 생성을 최대로 출이는 것과 2) machine 과는 상관없이 어떤 가상적인 환경 (simulation) 을 구축하는데 필요한 부분에 대해서 RTL object 를 만들게 됩니다.

다음으로는 GCC 내에서 사용될 byte_mode, word_mode, double_mode, ptr_mode 에 대한 설정을 하는 것입니다.

GCC 상에서는 machine mode 라는 것을 가지고 있는데 machine mode 는 machine level 에서의 data 의 크기와 format (형식) 을 지정합니다. 위에서 설정되는 byte_mode, word_mode, double_mode, ptr_mode 들도 현재 target 상에서 GCC 컴파일러가 사용할 byte, word, double, pointer 의 mode 를 정하는 것입니다. 'Machine Mode'에 대해서도 나중에 다시 할 것입니다.

또한 여기서는 전역 변수로 선언된 'Register RTX' 를 register number 를 할당하기도 합니다. 이같이 global 로 선언될 필요가 있는 것들에 대해서는 전역 변수 배열인 global_rtl 내에 저장되게 됩니다. register rtx 와 관련 있는 매크로는 다음과 같습니다.

```
pc_rtx
cc0_rtx
```

```
stack_pointer_rtx
frame_pointer_rtx
hard_frame_pointer_rtx
arg_pointer_rtx
virtual_incoming_args_rtx
virtual_stack_vars_rtx
virtual_stack_dynamic_rtx
virtual_outgoing_args_rtx
virtual_cfa_rtx
```

각각이 고유의 register number 를 가지고 있으며 ix386 에서 사용되는 레지스터들의 map 을 간단히 그림으로 그린다면 다음과 같습니다. 전체 Register 할당된 그림이 아니며 일부분입니다.

```
RN      Description
-----------------------------------------
2       STATIC_CHAIN_REGNUM
3       PIC_OFFSET_TABLE_REGNUM

6       HARD_FRAME_POINTER_REGNUM
7       STACK_POINTER_REGNUM

16      ARG_POINTER_REGNUM

20      FRAME_POINTER_REGNUM

<FIRST_PSEUDO_REGISTER>
53      VIRTUAL_INCOMING_ARGS_REGNUM
54      VIRTUAL_STACK_VARS_REGNUM
55      VIRTUAL_STACK_DYNAMIC_REGNUM
56      VIRTUAL_OUTGOING_ARGS_REGNUM
57      VIRTUAL_CFA_REGNUM
```

기본적인 register number 할당이 마무리되면 GC root 로써 global_rtl 전역 변수 배열을 등록합니다.

그리고 특정 rtx code 들과 operand 값들을 위한 unique rtx 들을 생성합니다. 여기서 다음과 같은 변수들이 설정됩니다.

const_int_rtx const_true_rtx

dconst0 dconst1 dconst2 dconstm1 const_tiny_rtx

여기서 const_int_rtx, const_tiny_rtx, const_true_rtx 가 GC root 로써 등록됩니다.

그리고 machine 마다 다르게 설정될 수 있는 나머지 register number 들에 대한 할당을 합니다.

```
return_address_pointer_rtx
struct_value_rtx
struct_value_incoming_rtx
static_chain_rtx
static_chain_incoming_rtx
pic_offset_table_rtx
```

위의 list 가 나머지 register number 에 해당하는 부분입니다. 그리고 마지막으로 위의 list 를 모두 GC root 로 등록하게 됩니다.

이렇게 함으로써 GCC 에서 사용되는 unique rtx 에 대한 기본적인 설정은 끝나게 됩니다.

3.5 Register set 들/ mode 들

이 부분에서는 Register set 들을 초기화하던 것을 마무리 짓고, register mode 들을 초기화하게 됩니다.
GCC 내에서 Register 들에 대한 class 가 존재합니다. 이와 같이 class 가 존재하는 이유는

1. Machine description 내에서의 레지스터 제약
2. Constant 들의 범위 정의

때문에 존재하며 현재 GCC 3.1 버전에서는 26 개의 register class 를 가지고 있습니다. 'Register set 들/ mode 들' 에서 수행되는 부분 중에 하나인 init_reg_sets_1 () 함수에서는 크게 다음과 같은 일을 하는 것이 주를 이룹니다.

1. 각 class 내의 hard register 들의 갯수를 계산
2. reg_class_subunion[I][J] 에 class I 와 J 의 합집합내에 포함되는 가장-큰-수를 가지는 Register Class 를 계산. 여기에서는 subunion (부분합집합) 을 계산합니다.
3. reg_class_superunion[I][J] 에 class I 와 J 의 합집합내에 포함되는 가장-작은-수를 가지는 Register Class 를 계산
4. 각 Register Class 의 subclass (reg_class_subclasses[I][J]) 와 superclass (reg_class_superclasses[i][j]) 들의 테이블을 초기화
5. "Constant" 테이블 초기화. 이 테이블과 연관있는 전역 변수로는 fixed_reg_set, call_used_reg_set, call_fixed_reg_set, regs_invalidated_by_call 가 있습니다. 모두 type 을 HARD_REG_SET 을 가지고 있는 것들입니다. 해당하는 비트에 적당히 설정될 것입니다.
6. Move Cost 테이블 초기화. 각 class 의 모든 부분집합을 찾아 어떤 부분집합에서 다른 집합으로 이동할 시에 드는 최대 cost 를 구합니다.

그리고 init_reg_sets_1 () 함수가 끝난 후에 수행되는 init_reg_modes () 함수에서는 전역변수 reg_raw_mode[I] 를 위한 설정이 이루어진다.

나머지 부분에서는 memory_move_secondary_cost 내에서 사용될 몇몇 가짜 stack-frame MEM reference 들을 만듭니다.

이로써 GCC 에서 사용될 "Register set 들/ mode 들"에 대한 기본적인 초기화는 마무리짓게 됩니다.

3.6 Alias sets

Alias sets 은 어떤 MEM 들이 다른 MEM 들을 alias 할 수 있는지를 back-end 가 결정할 수 있도록 도와주는데 사용됩니다.

GCC 에서 이 부분은 두 단계로 이루어집니다.

- Machine 상에 존재할 수 있는 register 들 중 incoming pointer argument 를 가질 수 있는지를 검사합니다. 이에 대한 결과값은 argument_registers 에 저장되게 됩니다.
- 여러 alias set entry 들을 저장하는데 사용되는 splay-tree 를 초기화합니다. 이것은 $prefix/gcc/alias.c 에 선언되어 있는 전역 변수 alias_sets 을 splay_tree_new () 함수를 사용하여 초기화하게 됩니다.

만약 splay-tree 에 대해 궁금한 분이 계시다면 다음과 같은 책을 읽어 보시기를 권합니다.

제목 : Data Structures and Their Algorithms.
저자 : Lewis, Harry R. 와 Denenberg, Larry
출판사 : Harper-Collins, Inc.
년도 : 1991

3.7 TREE → RTL 확장 (expand)

GCC 에는 컴파일러의 AST 에 해당하는 TREE 와 Low-Level IR (Intermediate Representation) 에 해당하는 RTL (Register Transfer Language) 을 가지고 있습니다. 당연한 이야기이지만 AST 를 IR 로 변환을 해야 합니다. AST 를 IR 로 변환하는 부분은 컴파일러에서는 middle-end 정도의 과정이라고 보시면 됩니다.

다음의 하위 섹션들은 그에 해당하는 분야에 대한 설명을 주로 하겠습니다.

3.7.1 list manager

List Manager 에서는 $prefix/gcc/rtl.def 에 정의되어 있는 EXPR_LIST 와
INSN_LIST 들의 cache 가능한 list 들을 유지보수하게 됩니다.

init_EXPR_INSN_LIST_cache () 함수에서는 EXPR_LIST 는 할당되었지만 현재 사용하지 않는 모든 EXPR_LIST 들을 포함하는 전역 변수 unused_expr_list 를 GC root 에 등록합니다.

3.7.2 statement

Statement 에 관해서는 'struct function' 의 stmt 구성요소에서 많은 부분을 초기화하게 됩니다. init_stmt () 함수에서는 단순히 stmt_obstack 전역 변수의 obstack 을 초기화할 뿐입니다.

3.7.3 function

GCC 의 경우 어떤 코드를 실제로 생성하는데 있어서, 하나의 파일을 전체적으로 컴파일 한 후 코드를 생성하지 않습니다. 현재 GCC 에서는 하나의 함수를 기준으로 컴파일한 후 함수 단위로 코드를 생성해 냅니다. 이점이 사람들로 부터 문제가 되는 부분이 되고 있다고 지적받고 있으며 속도 혹은 최적화에서 이런 mechanism 으로 인해 구현하지 못하는 부분이 있다고 알려져 있습니다.

init_function_once () 함수에서는 cfun, outer_function_chain 를 GC root 로 등록하게 됩니다.

그리고 전역변수 prologue 와 epilogue, sibcall_epilogue 에 대해 VARRAY 를 초기화하게 됩니다.

prologue 와 epilogue 는 prologue 와 epilogue insn 들의 INSN_UID 들을 기록하는데 사용됩니다.

나중에도 다시 살펴보게 되겠지만 GCC 에서 한 함수의 모든 정보는 'struct function' 에 모이게 됩니다.

'struct function' 을 초기화 하는 부분은 언어 종속적인 부분에서
init_dummy_function_start () 함수가 담당하는 부분입니다. 이 부분에서 RTL expansion mechanism 을 초기화합니다. 이렇게 함으로써 순차적 (sequence) 생성과 같은 간단한 것을 할 수 있습니다. 이것은 몇몇 단계들에서 global initialization 동안 context 를 제공하는데 사용됩니다.

단순히 'struct function *cfun' 구조체의 구성요소를 이제 실제로 사용할 수 있게 만드는 것이 목표입니다. 하지만 구성요소가 machine 에 따라 달라질 수 있는 부분이 존재한다.

실제 'struct function' 의 크기가 아주 크기 때문에 따로 강의를 마련할 것이다.

3.7.4 중간-레벨 하위루틴

이것은 bit-field store, extrace 부분 그리고 shift, 곱하기, 나누기들을 rtl 명령으로 변환하는 부분입니다.

언어 독립적인 초기화 부분에서는 여러 RTL 조각들의 cost 들을 초기화하게 됩니다. 초기화될 다음의 몇몇은 shift count 로 나열되어 있고 어떤 것은 mode 순으로 나열되어 있음을 아시기 바랍니다. 초기화되는 부분은 다음과 같은 것이 있습니다.

zero_cost, add_cost,
shift_cost, shiftadd_cost, shiftsub_cost,
negate_cost,
div_cost, mul_cost, mul_widen_cost, mul_highpart_cost

그 외에 다름과 같은 것이 초기화됩니다. 아래의 값이 0 이 아닐 경우 divide 들 혹은 modulus operation 들이 2 의 제곱 만큼 상대적으로 cheap 하기 때문에 branch 들을 사용하지 않는 것을 의미합니다. 대신 operation 을 emit 합니다.

sdiv_pow2_cheap, smod_pow2_cheap

3.7.5 expression

$prefix/gcc/expr.c 파일에 선언되어 있는 init_expr_once () 함수 또한 tree 표현식을 rtl 명령으로 변환하는데 사용되는 것을 초기화하게 됩니다. 이 함수에서는 메모리내에서 직접적으로 사용될 수 있는 어떤 mode 들을 설정하고 block mov optab 을 초기화하기 위해 컴파일당 한번만 수행됩니다.

이 함수에서 중심적으로 설정되는 것은 전역변수 direct_load 와 direct_store 입니다. 이 두 변수에는 각 mode 상에서 우리가 register 를 메모리내의 해당 mode 인 object 를 직접적으로 보내거나 가져올 수 있는지를 (이동할 수 있는지를) 기록하고 있습니다. 만약 우리가 그렇게 할 수 없다면 해당 mode 의 영역 (field) 을 접근할 때 직접적으로 해당 mode 를 사용할려고 시도하지 않습니다.

3.8 최적화 (optimazation)

Thomas Pittman, James Petes 가 쓴 "컴파일러 디자인"이란 책을 보면 아래와 같은 말이 나옵니다.

> 코드 최적화를 일반적인 두 범주로 나누어 생각하는 것이 보편환되어 있다. 즉, 목적 하드웨어를 고려치 않고 AST 에서 이루어질 수 있는 **기계-독립적** 부분과 구현 및 필요성을 설정하는 것까지도 목적 기계에 관한 밀접한 지식을 필요로 하는 **기계-의존적** 부분으로, 후자는 중간 코드에 대한 저급의 4 가지 요소 표현(low-level quad representation) 과 최종 목적 코드에 적용된다. 그와 같은 명확한 기계-독립적인 최적화를 상수 폴딩과 루프-상수 코드 모션으로 인식하면서도 대부분의 최적화가 정도의 차이는 있지만 목적 기계 구조에 종속된다는 보편성이 결여된 시각을 일반적으로 가지고 있기 때문에 이러한 구분이 별로 유용하지 않은 것으로 알고 있다.

최적화에 대해서 앞으로도 많은 공부가 있어야 하겠지만 GCC 에서 '언어-독립적인 초기화' 부분과 연관되어 있는 것에 대해서 언급을 한다면 init_loop () 함수에 대해서 말해야 할 것입니다.

init_loop () 함수는 $prefix/gcc/loop.c 파일에 선언되어 있습니다. loop.c 파일은 Strength reduction 를 포함하여 여러 loop 최적화를 수행하는 루틴들을 포함하고 있습니다.

여기서 수행되는 부분이 전역 변수 reg_address_cost 와 copy_cost 를 초기화하는 것이 목표입니다.

그리고 어떤 Machine 상에서 어떤 레지스터를 다룰 때 드는 cost (비용) 에 대한 개념 또한 앞으로 다루게 될 것입니다.

3.9 Reload

Reload 단계는 컴파일러에서 Register Allocation 이 이루어 진 후 실행됩니다. Reload 단계에서는 각 insn 에서 사용하는 operand 들이 실제 machine 상에서 작동할 경우 잘못된 경우가 발생할 수 있는지를 검사하게 됩니다. 위에서 언급했듯이 Register 에도 class 가 존재하는데, operand 들이 적당한 class 에 들어가 있는지를 점검함으로써 이루어지게 됩니다.

init_reload () 함수에서 이루어지는 과정을 좀 살펴보게 된다면...

RTL 의 어떤 조합이 실제로 컴파일하고자 하는 machine 에서 유효한 방법인지를 검사하게 됩니다. 이 검사에서 살펴보게 되는 부분은 세 부분에 대해서 살펴보게 됩니다.

1. 종종 (MEM (REG n)) 는 (REG n) 가 stack 상에 놓여있는데도 불구하고 여전히 유효한 경우가 있습니다. 이에 대해 조사하는 부분이 있습니다. 해당 여부에 대한 정보는 spill_indirect_levels 전역변수에 저장되게 됩니다.

2. Indirect addressing 이 (MEM (SYMBOL_REF ...)) 에 대해 유효한지를 검사합니다.

3. reg+reg 가 유효한 (offset 이 사용 가능한지) address 인지를 검사합니다.

검사가 끝난 후에 Reload 단계를 위한 obstack 을 초기화합니다. 이 정보를 가지고 있을 전역 변수는 $prefix/gcc/reload1.c 파일에 선언되어 있는 reload_obstack 입니다. 이 obstack 은 register elimination 동안 rtl 의 allocation 을 위해 사용됩니다. 할당된 storage 는 find_reloads () 함수가 명령어를 처리한 즉시 free 될 수 있습니다.

그리고 전역변수 reload_startobj 가 reload_obstack 의 시작점을 가르키게 됩니다.

마지막으로 전역 변수 spilled_pseudos 와 pseudos_counted 를 초기화하게 됩니다.

전역 변수 spilled_pseudos 는 어떤 pseudo 가 spill 되어야 할지를 기록하는데 사용되면, pseudos_counted 는 order_regs_for_reload 와 count_pseudo 사이의 대화에 사용됩니다. 하나의 pseudo 를 두번 세는 것을 피하는데 사용됩니다.

3.10 Storage Layout

Storage Layout 에 대한 수행 부분을 담은 파일은 $prefix/gcc/stor-layout.c 파일에 선언되어 있습니다. 이 파일은 type 들과 variable 들을 storage layout 을 위한 C-컴파일러 유틸리티들을 포함하고 있습니다.

Storage Layout 은 특정 data 의 크기와 alignment 를 정의하는데 사용됩니다.

init_stor_layout_once () 함수에서는 단순히 전역변수 pending_sizes 를 GC root 에 등록합니다. 여기서 pending_sizes 는 expand 되기를 기다리는 type 들과 decl 들의 크기들을 위한 SAVE_EXPR 들을 가지고 있습니다.

3.11 Varasm

$prefix/gcc/varasm.c 파일을 함수의 명령어 (instruction) 들을 제외한 모든 어셈블리 code 의 생성에 대해 다루는 파일입니다.

init_varasm_once () 함수에서 하는 일은 전역변수 const_str_htab 와 in_named_htab 의 Hash Table 을 초기화하고 const_hash_table 와 const_str_htab, weak_decls 을 GC root 로 등록합니다.

마지막으로 전역변수 const_alias_set 을 설정합니다.

3.12 Caller Save

Caller Save 에 대해서는 나중에 살펴보도록 하자.

3.13 기타 사항들

lang_independent_init () 함수에서 초기화되지만 큰 묶음이 아니라 따로 묶지 않았던 전역 변수에 대한 설정을 살펴보도록 합니다.

묶지 못한 부분은 모두 세개의 전역 변수입니다. 그에 대해 알아 봅시다.

decl_printable_name Declaration 을 출력하는데 사용되는 이름을 계산하는 함수를 가르키는 포인터입니다. 여기서는 decl_name () 함수의 포인터로 등록하게 됩니다.

lang_expand_expr Language-Specific tree code 를 위한 rtl 을 계산하는 함수를 가르키는 포인터입니다. 여기서는 일단 do_abort () 함수 포인터로 설정해 놓습니다.

tree_code_length 언어-의존적인 identifier 크기를 설정합니다.

제 4 절 11 주 강의를 마치며

휴... 정말 오랜만에 강의를 내놓게 되는군요. 그동안 회사일 때문에 제대로 쓰지 못했는데 조금씩 조금씩 써서 이렇게 완성하게 되었습니다. 강의를 쓰는 것은 그렇게 어렵지 않지만 GCC 를 분석하는 일이 가장 어려운 듯합니다. 제가 제대로 알지 못하는 개념일 경우 자료조사시 시간이 많이 걸립니다. 다음 강의에서 뵙겠습니다.