처리된 option 들의 무결성 검사

정원교

2004년 2월 23일

목 차

제 1 절 8 주 강의를 시작하며

이번 주는 5 주에 했던 “개발자 측면에서 바라본 GCC 속의 option 들” 글에 이어 작성하고자 합니다. 앞 강의에서 했던 부분은

1. GCC 속에 존재하는 옵션들
2. 최적화에 따른 옵션의 변화
3. 각 옵션이 미치는 전역 변수들

에 관한 내용을 하였습니다. GCC 내에서 option 들의 무결성 검사를 할 시점은 옵션을 모두 처리한 후 기본적인 내부 설정을 이루기 전입니다. 이번 강의에서는 옵션 처리를 완전히 마무리하는 시점에서의 option 조합이 정확한지 검사를 GCC 에서 어디서 하는지와 우리가 살펴보고 있는 C 언어에서는 어떤 제약을 하는지를 보게 될 것입니다.

제 2 절 무결성 검사는 어느 시점에서 그리고 어디서 하는가?

지금까지 우리는 GCC 의 시작부터 옵션이 처리되는 부분까지 살펴보았습니다. 이제 옵션을 처리하는 부분의 마지막에 와 있습니다. 지금까지 toplev_main () 함수 내부의 parse_options_and_default_flags () 함수는 모두살펴보았습니다. 이제 실질적으로 컴파일러로써 작동을 하게 되는 do_compile () 함수로 넘어가게 되는데 이 함수의 처음 부분에서 process_options () 함수에서 마무리 command line switch 처리 과정을 가지게 됩니다. 즉 여기에서 무결성 검사를 하게 됩니다. 이번 강의에서 살펴볼 내용은 이 함수내의 수행 과정과 아직 살펴보지 못한 변수들을 설명하도록 하겠습니다.

제 3 절 process_options () 함수

이제 process_options () 함수 내에서 어떻게 수행이 되는지, 그리고 어떤 매크로를 사용하고 어떤 전역 변수를 사용하며 어떤 규칙을 가지고 행동하는지에 대해서 살펴보도록 하겠습니다.

3.1 OVERRIDE_OPTIONS 란?

같은 언어라도 GCC 는 각 machine 에 대해서 다른 수행을 할 경우가 있습니다. 몇몇 machine 들은 option 들의 특정 조합을 거절할 수 있습니다. OVERRIDE_OPTIONS 매크로 또한 그러한 설정을 할 수 있도록 하는 것이며 이것의 수행 여부 또한 설정될 수 있습니다.

이 매크로는, 만약 정의가 되어 있다면, 모든 command option 들이 처리된 후에 한번 호출됩니다.

이 매크로는 '-O' 와 같은 여러 다른 최적화를 활성화하는 사용하지 마십시오. 그것을 위해서는 'OPTIMIZATION_OPTIONS' 가 준비되어 있습니다.

우리가 살펴보고 있는 target 은 i386 이므로 $prefix/gcc/config/i386/ 에 대해서 살펴보면 이 매크로는 i386.h 에 선언되어 있다는 것을 알 수 있습니다. 아래와 같이 선언되어 있습니다.

```
#define OVERRIDE_OPTIONS override_options ()
```

위의 실제 함수 override_options () 함수는 같은 디렉토리의 i386.c 파일에 선언되어 있습니다. 이 함수는 상당히 복잡하고 살펴봐야 할 부분도 많기 때문에 아래에 따로 섹션을 마련하도록 하겠습니다.

3.2 align_*_log 변수

align_*_log 변수들을 설정합니다. 만약 아직도 설정되어 있지 않다면 그것을 1 로 기본값을 줍니다.

3.3 "Unrolling all loops"와 "Loop unrolling"

Unrolling all loops 가 설정되어 있다는 것은 표준
loop unrolling 또한 수행되어야 함을 말합니다. 그래서
flag_unroll_all_loops 가 선언되어 있다면 flag_unroll_loops 를 1 로 설정합니다.

그리고 flag_unroll_loops 가 설정되어 있는 것은 다른 요구 사항들을 필요로 하는데, Loop unrolling 는 strength_reduction 의 수행 또한 요구합니다. 그렇기 때문에 아직 활성화가 되어 있지 않다면 여기서 설정합니다. 또한 loop unrolling code 는 loop 후에 cse 가 실행될 것으로 가정하기 때문에 그것 또한 활성화시켜줍니다.

3.4 user_label_prefix

만약 기본 prefix 가 "" 혹은 "_" 보다 복잡한 형태이면 이에 대해 업근하고 이 옵션을 무시합니다.

3.5 debug_hooks

이제 우리는 write_symbols 를 알고 있으므로 그것에 기반하여 debug hooks 를 설정합니다. 기본값으로 debug output 는 아무 일도 하지 않습니다.

3.6 auxiliary 정보 생성

만약 auxiliary 정보 생성이 필요하다면 output file 를 엽니다. 이것은 다른 모든 output file 과는 다른게 source file 와 같은 디렉토리에서 수행이 이루어집니다.

3.7 기타 무결성 체크

다른 무결성 체크를 하게 됩니다. 그에 대해서는 자세히 언급하지 않도록 하겠습니다.

제 4 절 override_options () 함수

위에서 말했듯이 이제 i386.c 파일에 선언되어 있는 override_options () 함수에 대해서 살펴보도록 하겠습니다.

4.1 struct ptt

이 함수를 처음 바라보면 무지막지한 구조체하나가 우리를 기다리고 있습니다. 이름하여 'struct ptt' 라는 녀석입니다. 이 구조체의 원형은 아래와 같습니다.

```
static struct ptt
    {
      const struct processor_costs *cost;
      const int target_enable;
      const int target_disable;
      const int align_loop;
      const int align_loop_max_skip;
      const int align_jump;
      const int align_jump_max_skip;
      const int align_func;
      const int branch_cost;
    }
  const processor_target_table[PROCESSOR_max] =
    {
      {&i386_cost, 0, 0, 4, 3, 4, 3, 4, 1},
      {&i486_cost, 0, 0, 16, 15, 16, 15, 16, 1},
      {&pentium_cost, 0, 0, 16, 7, 16, 7, 16, 1},
      {&pentiumpro_cost, 0, 0, 16, 15, 16, 7, 16, 1},
      {&k6_cost, 0, 0, 32, 7, 32, 7, 32, 1},
      {&athlon_cost, 0, 0, 16, 7, 64, 7, 16, 1},
      {&pentium4_cost, 0, 0, 0, 0, 0, 0, 0, 1}
    };
```

각 구성요소에 대한 설명을 하면 아래와 같습니다.

- `cost`
  Processor cost 관련
- `target_enable`
  활성화 할 target flag 들.
- `target_disable`
  비활성화 할 target flags.
- `align_loop`
  기본 alignment 들.

- align_loop_max_skip
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- align_jump
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- align_jump_max_skip
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- align_func
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.
- branch_cost
  아직 정확한 설명이 없습니다. 추후 UPDATE 하겠습니다.

여기서 살펴보아야 할 것이 또 존재하는데, 내부에 포함된 구조체
struct processor_costs 입니다. 이에 대해서 좀 살펴 봅시다.

4.1.1 struct processor_costs

이 구조체는 $prefix/gcc/config/i386/i386.h 에 선언되어 있습니다. 이 구조체는 주어진 cpu 에 따른 특정 cost 를 정의합니다. 원형을 보고 각 구성요소에 대한 설명을 보도록 합시다.

```
struct processor_costs {
  const int add;
  const int lea;
  const int shift_var;
  const int shift_const;
  const int mult_init;
  const int mult_bit;
  const int divide;
  int movsx;
  int movzx;
  const int large_insn;
  const int move_ratio;
  const int movzbl_load;
  const int int_load[3];
  const int int_store[3];
  const int fp_move;
  const int fp_load[3];
  const int fp_store[3];
  const int mmx_move;
  const int mmx_load[2];
  const int mmx_store[2];
  const int sse_move;
  const int sse_load[3];
  const int sse_store[3];
  const int mmxsse_to_integer;
  const int prefetch_block;
  const int simultaneous_prefetches;
};
```

각 구성요소에 대한 설명을 하겠습니다.

- add
  add 명령어의 cost

- `lea`
  lea 명령어의 cost
- `shift_var`
  변수 shift 의 cost
- `shift_const`
  상수 shift 의 cost
- `mult_init`
  곱셈 시작의 cost
- `mult_bit`
  각 bit 집합당 곱셈의 cost
- `divide`
  divide/mod 의 cost
- `movsx`
  movsx 연산의 cost
- `movzx`
  movzx 연산의 cost
- `large_insn`
  이 cost 보다 큰 명령어들
- `move_ratio`
  memory-to-memory move insn 들의 scalar 수의 시발점.
- `movzbl_load`
  movzbl 를 사용한 loading 의 cost
- `int_load[3]`
  reg-reg move (2) 와 관련하여 QImode 와 HImode SImode 에서 integer 레지스터를 loading 시 cost.
- `int_store[3]`
  QImode 와 HImode SImode 에서 integer 레지스터를 storing 시 cost
- `fp_move`
  reg,reg fld/fst 의 cost
- `fp_load[3]`
  SFmode 와 DFmode, XFmode 에서 FP 레지스터 loading 시 cost
- `fp_store[3]`
  SFmode 와 DFmode, XFmode 에서 FP 레지스터 storing 시 cost
- `mmx_move`
  MMX 레지스터 이동시 cost
- `mmx_load[2]`
  SImode 와 DImode 에서 MMX 레지스터 loading 시 cost
- `mmx_store[2]`
  SImode 와 DImode 에서 MMX 레지스터 storing 시 cost
- `sse_move`
  SSE 레지스터 이동의 cost

- `sse_load[3]`
  SImode 와 DImode, TImode 에서 SSE 레지스터 loading 시 cost
- `sse_store[3]`
  SImode 와 DImode, TImode 에서 SSE 레지스터 storing 시 cost
- `mmxsse_to_integer`
  mmxsse 레지스를 integer 로 옮길 때 cost, 그 반대도 적용
- `prefetch_block`
  prefetch 를 위해 cache 되어 이동된 byte 들.
- `simultaneous_prefetches`
  병렬 prefetch 수행의 수.

이 구조체는 i386 target 으로는 현재 7 가지 형태를 가지고 있습니다. 아래에 리스트되어 있습니다.

1. i386_cost
2. i486_cost
3. pentium_cost
4. pentiumpro_cost
5. k6_cost
6. athlon_cost
7. pentium4_cost

각각의 machine 에 맞는게 설정될 것입니다. 위 7 가지 형태중 i386_cost 와 pentium_cost 가 어떻게 설정되는지를 한번 봅시다. 원형은 아래에 있습니다. 먼저 i386_cost 입니다.

```
static const
struct processor_costs i386_cost = {
  1,                    /* add 명령어의 cost */
  1,                    /* lea 명령어의 cost */
  3,                    /* 변수 shift 의 cost */
  2,                    /* 상수 shift 의 cost */
  6,                    /* 곱셈 시작의 cost */
  1,                    /* 각 bit 집합당 곱셈의 cost */
  23,                   /* divide/mod 의 cost */
  3,                    /* movsx 연산의 cost */
  2,                    /* movzx 연산의 cost */
  15,                   /* "large" insn */
  3,                    /* MOVE_RATIO */
  4,                    /* movzbl 를 사용해 QImode 를
                           loading 시 cost */
  {2, 4, 2},            /* reg-reg move (2) 와
                           관련하여 QImode 와 HImode
                           SImode 에서 integer
                           레지스터를 loading 시 cost. */
  {2, 4, 2},            /* integer 레지스터를 storing
                           시 cost */
  2,                    /* reg,reg fld/fst 의 cost */
  {8, 8, 8},            /* SFmode 와 DFmode, XFmode
```

```
                          에서 FP 레지스터 loading 시
                          cost */
  {8, 8, 8},            /* integer 레지스터들을
                          loading 시 cost */
  2,                    /* MMX 레지스터 이동시 cost */
  {4, 8},               /* SImode 와 DImode 에서 MMX
                          레지스터 loading 시 cost */
  {4, 8},               /* SImode 와 DImode 에서 MMX
                          레지스터 storing 시 cost */
  2,                    /* SSE 레지스터 이동의 cost */
  {4, 8, 16},           /* SImode 와 DImode, TImode
                          에서 SSE 레지스터 loading
                          시 cost */
  {4, 8, 16},           /* SImode 와 DImode, TImode
                          에서 SSE 레지스터 storing
                         시 cost */
  3,                    /* MMX 혹은 SSE 레지스터를
                          integer 로 */
  0,                    /* prefetch block 의 크기 */
  0,                    /* 병렬 prefetch 들의 수 */
};
```

그럼 pentium_cost 입니다.

```
static const
struct processor_costs pentium_cost = {
  1,                    /* add 명령어의 cost */
  1,                    /* lea 명령어의 cost */
  4,                    /* 변수 shift 의 cost */
  1,                    /* 상수 shift 의 cost */
  11,                   /* 곱셈 시작의 cost */
  0,                    /* 각 bit 집합당 곱셈의 cost */
  25,                   /* divide/mod 의 cost */
  3,                    /* movsx 연산의 cost */
  2,                    /* movzx 연산의 cost */
  8,                    /* "large" insn */
  6,                    /* MOVE_RATIO */
  6,                    /* movzbl 를 사용해 QImode 를
                          loading 시 cost */
  {2, 4, 2},            /* reg-reg move (2) 와
                          관련하여 QImode 와 HImode
                          SImode 에서 integer
                          레지스터를 loading 시 cost. */
  {2, 4, 2},            /* integer 레지스터를 storing
                          시 cost */
  2,                    /* reg,reg fld/fst 의 cost */
  {2, 2, 6},            /* SFmode 와 DFmode, XFmode
                          에서 FP 레지스터 loading 시
                          cost */
  {4, 4, 6},            /* integer 레지스터들을
                          loading 시 cost */
  8,                    /* MMX 레지스터 이동시 cost */
  {8, 8},               /* SImode 와 DImode 에서 MMX
```

```
                              레지스터 loading 시 cost */
  {8, 8},                  /* SImode 와 DImode 에서 MMX
                              레지스터 storing 시 cost */
  2,                       /* SSE 레지스터 이동의 cost */
  {4, 8, 16},              /* SImode 와 DImode, TImode
                              에서 SSE 레지스터 loading
                              시 cost */
  {4, 8, 16},              /* SImode 와 DImode, TImode
                              에서 SSE 레지스터 storing
                              시 cost */
  3,                       /* MMX 혹은 SSE 레지스터를
                              integer 로 */
  0,                       /* prefetch block 의 크기 */
  0,                       /* 병렬 prefetch 들의 수 */
};
```

4.2 struct pta

다른 구조체 struct pta 도 존재합니다. 원형은 아래와 같습니다.

```
static struct pta
  {
    const char *const name;           /* 프로세스 이름 혹은 별명.  */
    const enum processor_type processor;
    const enum pta_flags
      {
        PTA_SSE = 1,
        PTA_SSE2 = 2,
        PTA_MMX = 4,
        PTA_PREFETCH_SSE = 8,
        PTA_3DNOW = 16,
        PTA_3DNOW_A = 64
      } flags;
  }
const processor_alias_table[] =
  {
    {"i386", PROCESSOR_I386, 0},
    {"i486", PROCESSOR_I486, 0},
    {"i586", PROCESSOR_PENTIUM, 0},
    {"pentium", PROCESSOR_PENTIUM, 0},
    {"pentium-mmx", PROCESSOR_PENTIUM, PTA_MMX},
    {"i686", PROCESSOR_PENTIUMPRO, 0},
    {"pentiumpro", PROCESSOR_PENTIUMPRO, 0},
    {"pentium2", PROCESSOR_PENTIUMPRO, PTA_MMX},
    {"pentium3", PROCESSOR_PENTIUMPRO, PTA_MMX | PTA_SSE
                                     | PTA_PREFETCH_SSE},
    {"pentium4", PROCESSOR_PENTIUM4, PTA_SSE | PTA_SSE2 |
                                   PTA_MMX | PTA_PREFETCH_SSE},
    {"k6", PROCESSOR_K6, PTA_MMX},
    {"k6-2", PROCESSOR_K6, PTA_MMX | PTA_3DNOW},
    {"k6-3", PROCESSOR_K6, PTA_MMX | PTA_3DNOW},
    {"athlon", PROCESSOR_ATHLON, PTA_MMX | PTA_PREFETCH_SSE | PTA_3DNOW
                               | PTA_3DNOW_A},
    {"athlon-tbird", PROCESSOR_ATHLON, PTA_MMX | PTA_PREFETCH_SSE
                                     | PTA_3DNOW | PTA_3DNOW_A},
    {"athlon-4", PROCESSOR_ATHLON, PTA_MMX | PTA_PREFETCH_SSE | PTA_3DNOW
```

```
                                  | PTA_3DNOW_A | PTA_SSE},
  {"athlon-xp", PROCESSOR_ATHLON, PTA_MMX | PTA_PREFETCH_SSE
                                   | PTA_3DNOW | PTA_3DNOW_A | PTA_SSE},
  {"athlon-mp", PROCESSOR_ATHLON, PTA_MMX | PTA_PREFETCH_SSE
                                   | PTA_3DNOW | PTA_3DNOW_A | PTA_SSE},
};
```

이에 대한 자세한 설명은 별도로 하지 않겠습니다.

4.3 SUBTARGET_OVERRIDE_OPTIONS

이 매크로는 현재 저의 컴파일러 환경에서 사용하지 않는 것이기 때문에 설명을 해 드리지 못하겠습니다.

4.4 이 함수의 내부 수행

여기서 언급되는 매크로, 변수 중 알 수 없는 부분에 대해서는 이 강의의 부록을 참조하시기 바라며 본격적으로 내부 수행에 대해서 간단 명료하게 언급하도록 하겠습니다.

이 함수의 과정은 아래와 같은 절차로 이루어지게 됩니다.

1. ix86_cpu_string 와 ix86_arch_string 전역 변수에 맞는 적당한 값을 설정하게 됩니다. ix86_cpu_string 는 내부에 선언되어 있는 cpu_names 변수에 영향을 받을 수 있으며 ix86_arch_string 는 TARGET_64BIT 가 어떻게 설정되어 있느냐에 따라 “athon-4” 혹은 “i386”으로 설정될 수 있습니다.

2. ix86_cmodel 에 적당한 값을 넣는 동작이 이루어집니다. 이 값은 사용자에 의해서 달라질 수 있는데, 사용자는

   -mcmodel=

   를 이용하여 현재 컴파일하고자 하는 code model 을 지정해 줄 수 있습니다. 사용자가 지정한 값은 컴파일러가 실행된 후 옵션을 처리하는 부분에서 ix86_cmodel_string 에 저장되게 됩니다. 존재하는 종류에 대해서는 부록에 설명해 놓았습니다.

3. ix86_asm_dialect 를 무엇으로 할 것인가를 결정하게 됩니다. 어셈블리 방언의 종류에는 두 가지가 존재한다고 앞에서 말했으며 그의 종류는 “ASM_INTEL”와 “ASM_ATT” 가 있다는 사실을 여러분도 알고 계실 것입니다. 그리고 이것은 사용자가 아래와 같이 설정함으로써 이 값을 변화시킬 수 있습니다.

   -masm=intel 혹은 -masm=att

   설정한 값은 ix86_asm_string 문자열에 저장되게 됩니다.

4. 기본 CPU 를 아키텍쳐에 맞게 조절하는 단계를 거칩니다. 설정된 ix86_arch_string 값을 기반으로하여 processor_alias_table 에 설정되어 있는 이 아키텍쳐에 맞는 값들을 정보로 하여 target_flags 를 설정하게 됩니다. target_flags 변수는 -m 스위치에 대한 mask 값들을 가지게 됩니다. 이 과정에서 ix86_arch 의 값이 설정되게 됩니다.

5. ix86_cpu_string 문자열 변수와 processor_alias_table 의 요소들 간의 비교로 ix86_cpu 에 들어갈 적당한 값을 구합니다.

6. 모든 함수들을 위한 i386_stack_locals 를 설정합니다. 여기서 언급되는 init_machine_status 와 mark_machine_status 와 free_machine_status 는 모두 함수 포인터입니다.

7. -mregparm= 값을 확인합니다. 사용자가 입력한 값은 ix86_regparm_string 에 저장되며 이 스위치가 가질 수 있는 값은 0 과 REGPARM_MAX 사이의 값을 가질 수 있습니다.

8. -malign_loops, -malign-jumps, -malign-functions 옵션에 대한 처리를 하게 됩니다. 만약 사용자가 어떤 -malign-* 옵션들을 제공하였다면 그것에 대해서 경고를 하고 -falign-* 가 설정되지 않는 값만 사용합니다. 이 코드는 GCC 3.2 이후 버전에서는 제거된다고 합니다.

9. 프로세서 테이블에서 기본 align_* 값을 구합니다.
10. -mpreferred-stack-boundary= 값을 확인하거나 기본값을
    설정합니다. Pentium III 의 SSE __m128 의 기본값은 128 비트이지만 code 크기를 최적화할 때 스택을 align 되도록 유지하는 추가적인 코드를 원하지 않을 수도 있습니다.
11. -mbranch-cost= 값을 확인하고 기본값을 제공합니다.
12. 만약 TARGET_OMIT_LEAF_FRAME_POINTER 가 true 이라면 nonleaf frame 포인터들을 유지합니다.
13. flag_unsafe_math_optimizations 를 확인합니다. 만약 우리가 fast math 를 수행하고 있지 않다면 비교 순어 wrt NaNs 에 관해 고려하지 않습니다. 이것은 짧은 시간내 비교하여 순서화할 수 있도록 합니다.
14. architecture 가 FPU 를 가지고 있는지 확인합니다. 만약 architecture 가 항상 FPU 를 가지고 있다면 명령어들이 emulation 할 필요가 없기 때문에 NO_FANCY_MATH_387 를 비활성화합니다.
15. TARGET_SSE 를 검사합니다. 단순히 SSE builtins 를 요구하는 것 소용이 없습니다, 그래서 MMX 또한 -msse 로 켜질 수 있습니다.
16. TARGET_3DNOW 를 검사합니다. 만약 3DNow! 라면 또한 MMX 도 가지고 있습니다. 그래서 MMX 는 또한 -m3now 로 켜질수 있습니다
17. Prefix 로 어떤 ASM_GENERATE_INTERNAL_LABEL 를 build 할지를 결정합니다.

제 5 절 8 주 강의를 마치며

흠.. 이번 주는 상당히 바쁜 주였던 같군요. 회사일이 때문에 많이 바빴습니다. 그리고 주제도 약간 모호한 성격을 가지고 있는 것이라 글을 쓰면서도 주제 선정을 잘 못하지 않았나 걱정을 했구요. 제목은 매력적일 지 몰라도 쓰는 입장에서는 참 난감하더군요. 자세히 쓰자니 소스 코드보는게 더 낳구 그렇다고 포괄적으로 쓰자니 쓸 내용이 없으니..

제 6 절 부록: 이 함수 내부에서 사용되는 전역 변수와 매크로, 열거자

여기에서는 $prefix/gcc/config/i386/ 내에서 두루 사용되는 전역 변수, 매크로, 열거자 들에 대해서 살펴보도록 하겠습니다. 이 것은 따로 분리하여 다른 강의로 작성하는 것 또한 바람직해 보이지만 나중에 필요성이 대두되었을 때 따로 분리하도록 하겠습니다.

6.1 전역 변수

이 함수를 본격적으로 살펴보기 앞서 언급해야 할 몇 가지 사항들이 있는 것 같습니다. 우선 내부적으로 사용되는 전역 변수의 역활에 대해서 설명하는 것입니다. 여기서 살펴보는 전역 변수는 GCC 를 구성하는 많은 전역 변수중에서 i386 target 일 때만 사용될 전역 변수입니다. 이것은 대부분은 $prefix/gcc/config/i386/ 내에 존재하는 파일에 선언되어 있습니다. 이제 각각에 대해서 열거식으로 나열하여 봅시다. 알파벳순으로 나열되어 있습니다.

- `internal_label_prefix`
  ASM_GENERATE_INTERNAL_LABEL 에 의해 설정된 prefix.
- `internal_label_prefix_len`
  internal_label_prefix 에 포함되어 있는 내용의 길이.
- `ix86_align_funcs_string`
  함수들을 위한 2 제곱의 alignment.
- `ix86_align_jumps_string`
  non-loop jump 를 위한 2 제곱의 alignment.
- `ix86_align_loops_string`
  loop 를 위한 2 제곱의 alignment.
- `ix86_arch`
  어떤 명령어 집합 구조를 사용할 것인가 를 가지고 있는 변수.
- `ix86_arch_string`
  -march=¡xxx¿ 옵션을 위해 사용됩니다. 어떤 CPU 를 사용할지에 대한 문자열을 가지고 있습니다.
- `ix86_asm_string`
  Asm 방언. 방언이라고 말하면 좀 우습게 들리겠지만, 사투리를 가르키는 방언이라고 말하는 것이 좀 정확할 것 같군요. 방언이라고 말하는 이유는 어셈블러가 통일이 되어 있지 않고 조금씩 서로 형태나 모양이 다르기 때문에 이렇게 부릅니다.
- `ix86_branch_cost`
- `ix86_branch_cost_string`
  1-5 사이의 값: jump.c 파일을 보십시오.
- `ix86_cmodel_string`
  사용자로부터 전달받은 code model 옵션.
- `ix86_cpu_string`
  -mcpu=¡xxx¿ 옵션을 위해 사용됩니다. 어떤 명령어 집합 구조를 사용할 지에 대한 문자열을 가지고 있습니다.
- `ix86_fpmath_string`
  -mfpmath=¡xxx¿ 옵션을 위해 사용됩니다.
- `ix86_asm_dialect` 어셈블러의 출력형태를 가지고 있는 변수 입니다. 기본값은 AT&T 문법을 가르키는 ASM_ATT 를 가지고 있습니다.

- ix86_cmodel
  이 값은 전역 변수 ix86_cmodel_string 가 해석된 후의 값을 가지게 됩니다.
- ix86_cost
  선택한 cost 의 주소값을 가지고 있습니다. 기본 값으로 pentium_cost 를 가지고 있습니다.9
- ix86_cpu
  우리가 스케줄링을 하고자 하는 CPU 가 어떤 것인가에 대한 값을 가지고 있는 변수.
- ix86_fpmath
  floating point math 를 unit 을 사용하여 생성할 것인가.
- ix86_preferred_stack_boundary
  bit 크기로 나타낸 stack boundary 를 위한 우선(preferred) alignment.
- ix86_preferred_stack_boundary_string
  바이트로 표현된 stack boundary 를 위한 2 제곱의 alignment.
- ix86_regparm
  숫자화된 ix86_regparm_string
- ix86_regparm_string
  인자들을 넘겨줄 때 사용하는 레지스터의 #
- target_flags
  이 변수는 $prefix/gcc/rtlanal.c 파일에 정의되어 있습니다. 우리가 컴파일하고자 하는 machine sutype 를 지정한 bit flag 들. Bit 들은 tm.h 파일에 정의되어 있는 TARGET_.... 매크로를 사용하여 테스트 할 수 있으며 '-m...' 스위치에 의해 설정됩니다. rtlanal.c 에 정의되어 있어야 합니다.
- x86_prefetch_sse
  만약 sse prefetch instruction 가 NOOP 이 아니라면 true.

프로세서 cost 에 관한 정보를 가지는 전역 변수에 대해서 좀 살펴 보도록 하겠습니다. 이것 또한 group 으로 처리되어야 할 것 같아 따로 분리하였습니다.

- i386_cost
- i486_cost
- pentium_cost
- pentiumpro_cost
- k6_cost
- athlon_cost
- pentium4_cost
- size_cost

또 다른 그룹으로써 target specific, per-function data 구조체를 생성, 제거, 등록등을 하는 함수 포인터를 알아 보도록 하겠습니다.

- init_machine_status
  다음의 변수들은 target specific, per-function data 구조체들을 생성하는데 사용하는 함수의 포인터를 가지고 있습니다.

- free_machine_status
  다음의 변수들은 target specific, per-function data 구조체들을 제거하는 사용하는 함수의 포인터를 가지고 있습니다.

- mark_machine_status
  이 변수는 garbage collection 에 필요할 수 있는 target specific,
  per-function data 구조체내에 어떤 data item 들을 등록하는데 사용하는 함수의 포인터를 가지고 있습니다.

프로세서 기능/최적화에 대한 bitmask 들을 가지고 있는 전역 변수 그룹이 존재합니다. 이에 대한 원형을 보도록 하겠습니다.

```
#define m_386 (1<<PROCESSOR_I386)
#define m_486 (1<<PROCESSOR_I486)
#define m_PENT (1<<PROCESSOR_PENTIUM)
#define m_PPRO (1<<PROCESSOR_PENTIUMPRO)
#define m_K6  (1<<PROCESSOR_K6)
#define m_ATHLON  (1<<PROCESSOR_ATHLON)
#define m_PENT4  (1<<PROCESSOR_PENTIUM4)

const int x86_use_leave = m_386 | m_K6 | m_ATHLON;
const int x86_push_memory = m_386 | m_K6 | m_ATHLON | m_PENT4;
const int x86_zero_extend_with_and = m_486 | m_PENT;
const int x86_movx = m_ATHLON | m_PPRO | m_PENT4 /* m_386 | m_K6 */;
const int x86_double_with_add = ~m_386;
const int x86_use_bit_test = m_386;
const int x86_unroll_strlen = m_486 | m_PENT | m_PPRO | m_ATHLON | m_K6;
const int x86_cmove = m_PPRO | m_ATHLON | m_PENT4;
const int x86_3dnow_a = m_ATHLON;
const int x86_deep_branch = m_PPRO | m_K6 | m_ATHLON | m_PENT4;
const int x86_branch_hints = m_PENT4;
const int x86_use_sahf = m_PPRO | m_K6 | m_PENT4;
const int x86_partial_reg_stall = m_PPRO;
const int x86_use_loop = m_K6;
const int x86_use_fiop = ~(m_PPRO | m_ATHLON | m_PENT);
const int x86_use_mov0 = m_K6;
const int x86_use_cltd = ~(m_PENT | m_K6);
const int x86_read_modify_write = ~m_PENT;
const int x86_read_modify = ~(m_PENT | m_PPRO);
const int x86_split_long_moves = m_PPRO;
const int x86_promote_QImode = m_K6 | m_PENT | m_386 | m_486;
const int x86_single_stringop = m_386 | m_PENT4;
const int x86_qimode_math = ~(0);
const int x86_promote_qi_regs = 0;
const int x86_himode_math = ~(m_PPRO);
const int x86_promote_hi_regs = m_PPRO;
const int x86_sub_esp_4 = m_ATHLON | m_PPRO | m_PENT4;
const int x86_sub_esp_8 = m_ATHLON | m_PPRO | m_386 | m_486 | m_PENT4;
const int x86_add_esp_4 = m_ATHLON | m_K6 | m_PENT4;
const int x86_add_esp_8 = m_ATHLON | m_PPRO | m_K6 | m_386 | m_486
                        | m_PENT4;
const int x86_integer_DFmode_moves = ~(m_ATHLON | m_PENT4);
const int x86_partial_reg_dependency = m_ATHLON | m_PENT4;
const int x86_memory_mismatch_stall = m_ATHLON | m_PENT4;
const int x86_accumulate_outgoing_args = m_ATHLON | m_PENT4 | m_PPRO;
const int x86_prologue_using_move = m_ATHLON | m_PENT4 | m_PPRO;
const int x86_epilogue_using_move = m_ATHLON | m_PENT4 | m_PPRO;
```

```
const int x86_decompose_lea = m_PENT4;
const int x86_arch_always_fancy_math_387 = m_PENT|m_PPRO|m_ATHLON
                                           |m_PENT4;
```

6.2 매크로

위에서는 전역 변수에 대해서 살펴보았습니다. 이제 매크로에 대해서 좀 살펴보고 넘어가도록 하겠습니다. 나열 순서는 위와 같습니다.

-m 스위치들을 위한 mask 들을 위한 매크로들을 먼저 보도록 하겠습니다. 이 매크로들은 서로 묶여있는 것이기에 따로 분리해서 설명드립니다.

- MASK_80387
  하드웨어 부동 소수점
- MASK_RTD
  Arg 들을 pop 하는 ret 를 사용합니다
- MASK_ALIGN_DOUBLE
  2 word boundary 를 double 로 align 합니다
- MASK_SVR3_SHLIB
  bss 내의 지역변수들을 초기화하지 않음
- MASK_IEEE_FP
  IEEE fp 비교문들
- MASK_FLOAT_RETURNS
  st(0) 에서 float 를 반환
- MASK_NO_FANCY_MATH_387
  sin, cos, sqrt 를 비활성화
- MASK_OMIT_LEAF_FRAME_POINTER
  Leaf frame pointer 생략한다
- MASK_STACK_PROBE
  Stack probing 을 활성화한다
- MASK_NO_ALIGN_STROPS
  string ops 의 aligning 활성화한다
- MASK_INLINE_ALL_STROPS
  모든 경우 stringops 를 inline 한다
- MASK_NO_PUSH_ARGS
  push 명령어들을 사용한다
- MASK_ACCUMULATE_OUTGOING_ARGS
  Outgoing args 를 모은다
- MASK_ACCUMULATE_OUTGOING_ARGS_SET
- MASK_MMX
  MMX regs/builtins 를 지원한다
- MASK_MMX_SET
- MASK_SSE
  SSE regs/builtins 를 지원한다

- MASK_SSE_SET
- MASK_SSE2
  SSE2 regs/builtins 를 지원한다
- MASK_SSE2_SET
- MASK_3DNOW
  3Dnow builtins 을 지원한다
- MASK_3DNOW_SET
- MASK_3DNOW_A
  Athlon 3Dnow builtins 을 지원한다.
- MASK_3DNOW_A_SET
- MASK_128BIT_LONG_DOUBLE
  long double 크기가 128bit 입니다
- MASK_64BIT
  64bit code 를 생산합니다
- MASK_NO_RED_ZONE
  Red zone 을 사용하지 않습니다

이제 다른 매크로들을 살펴봅시다.

- ASM_GENERATE_INTERNAL_LABEL
  이것은 문자열 LABEL 을 어떻게 저장할 것인가에 대한 내용입니다. 위치를 가르키는 내부 label(숫자로 매겨지는) 의 symbol_ref 이름입니다. PREFIX 는 label 의 class 이고 NUM 은 class 내에서의 수입니다. 이것은 'assemble_name' 함수와 관련해서 output 하기에 적당합니다.

  대부분의 svr4 시스템을 위해서 period 로 시작하는 어떤 symbol 은 어셈블러에 의해 linker symbol table 내에 놓지 않는게 관례입니다.

  저의 컴파일 환경에서는 이 매크로는 $prefix/gcc/config/elfos.h 파일에 선언되어 있는 것을 사용합니다.
- BITS_PER_UNIT
  addressable storage unit 에서의 비트수.
- MAX_CODE_ALIGN
  여기서 사용되는 이 매크로는 $prefix/gcc/config/i386/i386.c 파일에 정의되어 있습니다. final.c 에서 왔습니다.
- REGPARM_MAX
  레지스터로 넘겨지는 인자들의 최대값. 만약 이것이 3 보다 크다면 우리는 ebx (레지스터 #4) 에서 문제를 만날 수 있는데, 이것은 이 레지스터가 caller saver register 이고 또한 ELF 에서 pic register 로써 사용되기 때문입니다. 그래 지금은 레지스터로 건내지는 것은 3 개 이상은 허락하지 않습니다.
- TARGET_ALIGN_DOUBLE
  Double 을 two word boundary 로 align 합니다. 이것은 double 을 포함하는 구조체를 위한 published ABI 와의 호환성이 깨질 수 있습니만 펜티엄상에서는 좀 더 빠른 code 생산해 냅니다.
- TARGET_CPU_DEFAULT
  기본 TARGET CPU 가 설정되어 있지 않다면 다음 값으로 합니다. configure 는 이 값을 486 으로 강제 설정하게끔 2 로 만들 수 있습니다.

- TARGET_CPU_DEFAULT_NAMES
  Target CPU 의 이름들을 가지고 있습니다. 현재 가지고 있는 요소는 아래와 같습니다.

```
#define TARGET_CPU_DEFAULT_NAMES
                 {"i386", "i486", "pentium", "pentium-mmx",\
                  "pentiumpro", "pentium2", "pentium3", \
                  "pentium4", "k6", "k6-2", "k6-3",\
                  "athlon", "athlon-4"}
```

- TARGET_64BIT
  64 비트 Sledgehammer 모드로 수행할 지에 대한 값을 가지고 있습니다. 이 값은 다른 매크로 TARGET_64BIT_DEFAULT 에 의해 결정되게 됩니다.

- TARGET_OMIT_LEAF_FRAME_POINTER
  Leaf 함수들을 위한 frame 포인터들을 생성하지 않습니다.

- TARGET_RTD
  인자들을 pop 하는 ret 명령어를 사용하여 컴파일합니다. 하지만 인자의 수가 변화할 수 있는 모든 함수들에 대해 적어도 여러분이 prototype 을 사용하지 않으면 작동하지 않을 것입니다.

6.3 열거자

이제 enum 으로 선언된 것에 대해서 관심을 가지도록 합시다.

- asm_dialect 어셈블러의 출력 형태를 정의하는 열거자입니다. 원형은 아래와 같습니다.

```
enum asm_dialect {
  ASM_ATT,
  ASM_INTEL
};
```

  ASM_ATT 는 AT&T 문법을 가르키고, ASM_INTEL 은 INTEL 문법을 가르킵니다.

- cmodel
  이 것은 아래와 같은 열거자들을 가지는 것입니다. 우선 원형을 보고 설명을 계속하도록 하겠습니다.

```
enum cmodel {
  CM_32,
  CM_SMALL,
  CM_KERNEL,
  CM_MEDIUM,
  CM_LARGE,
  CM_SMALL_PIC
};
```

  각각에 대한 설명은 아래와 같습니다.

  - CM_32
    32 비트 ABI 에서 사용됩니다.
  - CM_SMALL
    모든 code 와 data 가 address 공간의 처음 31 비트에 맞춰져 있는 것으로 가정하는 small model 입니다.

- CM_KERNEL
  모든 code 와 data 가 address 공간의 negative 31 비트들로 맞춰져 있는 것으로 가정하는 model 입니다.
- CM_MEDIUM
  address 공간의 처음 31 비트에 code 가 맞춰져 있는 것으로 가정하는 model 입니다. data 의 크기는 무제한입니다.
- CM_LARGE
  특정 section 들의 크기에 관한 가정을 가지고 있지 않는 model 입니다.
- CM_SMALL_PIC
  code+data+got/plt 테이블들이 address 공간의 처음 31 비트에 있다고 가정하고, PIC 라이브러리를 위한 model 입니다.

- fpmath_unit 부동소수점 math 관련 unit 에 대한 열거자를 가지고 있습니다.

```
enum fpmath_unit
{
  FPMATH_387 = 1,
  FPMATH_SSE = 2
};
```

- processor_type
  이 열거자는 어떤 프로세스를 스케줄을 할것인가에 대한 CPU 목록을 가지고 있습니다. CPU attribute 에 관한 정의 목록은 이것을 미러링합니다. 그래서 이 목록은 변경시 i386.md 또한 동시에 수정되어져야만 합니다. 이 원형은 아래와 같습니다.

```
enum processor_type
{
  PROCESSOR_I386,       /* 80386 */
  PROCESSOR_I486,       /* 80486DX, 80486SX, 80486DX[24] */
  PROCESSOR_PENTIUM,
  PROCESSOR_PENTIUMPRO,
  PROCESSOR_K6,
  PROCESSOR_ATHLON,
  PROCESSOR_PENTIUM4,
  PROCESSOR_max
};
```