GCC Machine Description
(1) Gen* 시리즈

정원교

2004년 7월 11일

목 차

제 1 절 28 주 문서를 시작하며

이제 조금씩 Machine Description 에 대해서 살펴보게 되었습니다. MD 에 대한 정의를 앞에서 하지 않아 좀 난해할 수 도 있지만, 추후에 계속 작성해 나갈 것이기 때문에 큰 혼돈은 없을 듯합니다. Gen* 시리즈의 경우 각각 하나의 프로그램(main () 함수를 독립적으로 가짐) 으로 되어 있기 때문에, 하나씩 필요한 부분만 보셔도 큰 상관은 없을 것입니다. gensupport.c 에 대해서 공통적으로 사용되기 때문에 이에 대해서 먼저 보시는 것이 편리할 것입니다.

제 2 절 Gen* 시리즈란?

GCC 에서는 Machine Description (이하 MD) 이라는 파일이 각 machine 에 따라 존재하며, 확장자는 .md 를 가지는 파일이 있다. 각 $prefix/gcc/config/[machine name]/*.md 위치에 존재하게 된다.

Gen* 시리즈라고 이름을 붙인 것은 아래의 리스트와 같이 gen 이라는 이름으로 시작하는 GCC 에 포함되어 있지만, 개별적인, 즉 MD 를 GCC 컴파일러가 사용하기 쉽도록 변환시키주는 프로그램을 말한다.

이 파일들의 위치는 $prefix/gcc/ 에 존재하게 된다.

```
-rw-r--r--    1 aso      aso         11582 12월  2  2001 genattr.c
-rw-r--r--    1 aso      aso        172308  2월 19  2002 genattrtab.c
-rw-r--r--    1 aso      aso          1700  1월  6  2002 gencheck.c
-rw-r--r--    1 aso      aso          2573 12월  2  2001 gencodes.c
-rw-r--r--    1 aso      aso          9693 12월  2  2001 genconfig.c
-rw-r--r--    1 aso      aso          2488 12월  2  2001 genconstants.c
-rw-r--r--    1 aso      aso         24545  3월 21  2002 genemit.c
-rw-r--r--    1 aso      aso         13460  3월  8  2002 genextract.c
-rw-r--r--    1 aso      aso          6492  3월 13  2002 genflags.c
-rw-r--r--    1 aso      aso          9877 12월 24  2001 gengenrtl.c
-rw-r--r--    1 aso      aso         12736 12월  2  2001 genopinit.c
-rw-r--r--    1 aso      aso         26256  3월 13  2002 genoutput.c
-rw-r--r--    1 aso      aso         11749 12월  2  2001 genpeep.c
-rw-r--r--    1 aso      aso          1886 10월  8  2001 genpreds.c
-rw-r--r--    1 aso      aso         76504  3월 21  2002 genrecog.c
-rw-r--r--    1 aso      aso         25465 12월  2  2001 gensupport.c
```

앞에서 이야기했듯이, MD 를 해석해서 그 역활에 맞는 파일을 생성하는 것이 목적으로써 이러한 MD 를 통해 만들어진 파일들은 아래와 같은 것들이 있다.

```
-rw-r--r--    1 aso      aso         17016 12월 27  2003 genrtl.c
-rw-r--r--    1 aso      aso         19608 12월 27  2003 genrtl.h
-rw-r--r--    1 aso      aso          4475 Feb 20 13:13 insn-attr.h
-rw-r--r--    1 aso      aso        947783 Feb 20 13:15 insn-attrtab.c
-rw-r--r--    1 aso      aso         16071 Feb 20 13:13 insn-codes.h
-rw-r--r--    1 aso      aso           396 Feb 20 13:12 insn-config.h
-rw-r--r--    1 aso      aso           199 Feb 20 13:12 insn-constants.h
-rw-r--r--    1 aso      aso        392610 Feb 20 13:16 insn-emit.c
-rw-r--r--    1 aso      aso         67357 Feb 20 13:16 insn-extract.c
-rw-r--r--    1 aso      aso         90100 Feb 20 13:12 insn-flags.h
-rw-r--r--    1 aso      aso         22435 Feb 20 13:16 insn-opinit.c
-rw-r--r--    1 aso      aso        397092 Feb 20 13:16 insn-output.c
-rw-r--r--    1 aso      aso           635 Feb 20 13:16 insn-peep.c
-rw-r--r--    1 aso      aso       1167870 Feb 20 13:16 insn-recog.c
-rw-r--r--    1 aso      aso          2661 12월 27  2003 tm-preds.h
```

몇몇 예외적인 것만 제외하곤, 대부분 gen* 의 뒷부분 이름에 insn- 접두사가 붙어 이름이 구성된다.

아래에서는 각 gen* 시리즈의 역활에 대해서 자세히 보도록 하겠다.

제 3 절 genattr

Machine description 으로 부터 attribute information (insn-attr.h) 를 생성한다.

3.1 목적

각 MD 내 DEFINE_ATTR 와 DEFINE_DELAY, DEFINE_FUNCTION_UNIT 를 읽어서, DEFINE_ATTR 일 경우 해당 attribute 를 위한 enumerator 를 작성하며, DEFINE_FUNCTION_UNIT 에 대해서는 여러 definition 들을 정의한다.

3.2 작동 방법

DEFINE_ATTR 와 DEFINE_DELAY, DEFINE_FUNCTION_UNIT 를 처리한다. 아래와 같은 전역 변수를 사용한다.

- all_multiplicity
- all_simultaneity
- all_ready_cost
- all_issue_delay
- all_blockage

위의 전역 변수 모두 DEFINE_FUNCTION_UNIT 를 처리하는데 사용된다.
전체 DEFINE_FUNCTION_UNIT 중 가장 큰 값을 가진다.

3.3 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 값에 대한 범위.  */

struct range
{
  int min;
  int max;
};

/* DEFINE_FUNCTION_UNIT 에서 언급된 각 function unit 에 관한 정보를
   기록한다.  */

struct function_unit
{
  char *name;                   /* Function unit 이름.  */
  struct function_unit *next;   /* 다음 function unit.  */
  int multiplicity;             /* 이 type 의 unit 들의 갯수.  */
  int simultaneity;             /* 이 function unit 상에 동시에 수행할 수 있는 insn
                                   들의 최대 갯수, 만약 0 이면 무제한.  */
  struct range ready_cost;      /* Ready cost 값의 범위.  */
  struct range issue_delay;     /* Issue delay 값의 범위.  */
};
```

제 4 절 genattrtab

Machine description 으로부터 attribute 들의 값을 계산하는 code 를 생성한다.

4.1 작동 방법

4.1.1 DEFINE_INSN 혹은 DEFINE_PEEPHOLE, DEFINE_ASM_ATTRIBUTES 의 처리

- gen_insn () 함수의 경우, 각 rtx 에 대해서 전역변수 defs 에 linked-list 로 연결시킨다.
- 전역 변수 defs 를 기반으로 전역 변수 insn_alternatives 와 insn_n_alternatives 를 구성한다.
- 이렇게 구성된 defs 들과 전역 변수는 fill_attr () 함수를 통해서 DEFINE_ATTR node 들에 반영되어진다.

4.1.2 DEFINE_ATTR 의 처리

- gen_attr () 함수에서 처리를 살펴보면 다음과 같다.
  - find_attr () 함수를 이용하여 전역변수 attrs 에 현재 처리하고 있는 rtx 와 같은 이름을 가진 struct attr_desc 구조체가 존재한다면 그것을 반환하고 없을 경우, 새로운 struct attr_desc 구조체를 할당하여 값을 초기화한 후 반환한다.
  - DEFINE_ATTR 의 두번째 인자를 처리하는데, 만약 이것이 NULL 일 경우, attr→is_numeric 가 1 로 설정되고, 아닐 경우 콤마(,)를 기준으로 분리하여 각각에 대한 CONST_STRING rtx 를 생성하여 각 struct attr_value 구조체를 할당하고, 이에 대한 적당한 값을 넣는다. 물론 해당 attr→first_value 에 이에 대한 liked-list head 부분을 넣게 된다.
  - DEFINE_ATTR 의 세번째 인자가 CONST rtx 일 경우, attr→is_const 를 1 로 설정하고, 세번째 인자 모습에서 가장 top level 에 있는 CONST node 를 제거하는데, 오직 top level 의 것만 제거되어질 수 있다. CONST rtx 가 아닐 경우, 제거 되지 않는다.
  - 마지막으로 attr→default_val 의 값, 디폴트값, 을 구하기 위해서 get_attr_value () 함수를 호출하는데, 이 함수의 경우, IF_THEN_ELSE rtx 를 COND rtx 로 변경한다. 그리고 앞에서 만든 두번째 인자에 대한 각 문자열에 대한 rtx, CONST_STRING 과 비교를 하여 attr→default_val 의 값을 설정한다. 완벽하게 구문을 해석해서 기본값을 구하는 것이 아니라, 기본값을 해석할 수 없을 경우, 해당 DEFINE_ATTR 의 세번째 인자 rtx 를 그대로 attr→default_val 의 값으로 설정한다.
- gen_attr () 함수 처리가 완료되면, 전역변수 attrs 에 축적되어 있는 attribute 들에 대한 검증을 check_attr_value () 함수를 통해서 수행한다.
- 검증이 완료되면, fill_attr () 함수를 통해서 각 attribute 의 설정에 DEFINE_INSN 설정을 반영한다. 즉 DEFINE_INSN 의 네번째 인자를 통해 지정된 attribute 가 있을 경우, 이에 대한 표현식을 각 attribute 의 설정 (ad→first_value) 에서 찾아 같은 표현식이 있을 경우 해당하는 곳에 struct insn_ent 를 추가하고, 그렇지 않을 경우 ad→default_val 에 추가하게 된다. 네번째 인자를 통해 지정된 attribute 가 있을 경우라도 ad→first_value 에 이와 같은 rtx node 들을 가진 것이 없다면, 새로운 struct attr_value 를 만들어 ad→first_value 에 연결시킨다.
- optimize_attrs () 함수를 통해서 지금까지 보인 attribute 들을 최적화한다.
- 최적화가 완료된 후 write_attr_get () 함수를 통해서 하나씩 output 한다.

4.1.3 DEFINE_DELAY 의 처리

- gen_delay () 함수의 경우, 각 rtx 에 대해서 전역변수 delays 에 linked-list 로 연결시킨다.

4.1.4 DEFINE_FUNCTION_UNIT 의 처리

```
(define_function_unit "pent_np" 1 0
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "imul"))
  11 11)
```

와 같은 define_function_unit 구문을 MD 에서 만났을 때, gen_unit () 함수를 통해서 처리하게 되는데, 첫번째 인자와 두번째 인자, 세번째 인자가 하나의 set 이 되어 하나의 struct function_unit 구조체를 구성하게 되며, 이러한 set 이 같이 선언된 된 다른 define_function_unit (아래와 같은 녀석들) 들은 해당 set 이 위의 것과 같기 때문에, unit→condexp (여기서 unit 은 struct function_unit 에 대한 포인터) 는 IOR 연산을 하게 된다.

```
;; Rep movs takes minimally 12 cycles.
(define_function_unit "pent_np" 1 0
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "str"))
  12 12)
```

그래서 unit→condexp 에 다음과 같은 RTX 가 들어가 있다고 가정하면

```
(and (eq_attr ("cpu") ("pentium"))
    (ior (eq_attr ("type") ("imul"))
        (eq_attr ("type") ("str"))))
```

새로 입력되는 op→condexp (여기서 op 는 struct function_unit_op 로의 포인터) 가 다음과 같을 경우,

```
and (eq_attr ("cpu") ("pentium"))
    (eq_attr ("type") ("idiv")))
```

아래와 같은 IOR 연산을 뜻하는 RTX 가 생성되게 된다.

```
(ior (and (eq_attr ("cpu") ("pentium"))
        (ior (eq_attr ("type") ("imul"))
            (eq_attr ("type") ("str"))))
    (and (eq_attr ("cpu") ("pentium"))
        (eq_attr ("type") ("idiv"))))
```

이를 최적화시켜서, 다음과 같은 구문을 만들게 된다.

```
(and (eq_attr ("cpu") ("pentium"))
    (ior (ior (eq_attr ("type") ("imul"))
            (eq_attr ("type") ("str")))
        (eq_attr ("type") ("idiv"))))
```

4.2 구조체들

4.2.1 DEFINE_INSN 관련 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 각 DEFINE_INSN 혹은 DEFINE_PEEPHOLE, DEFINE_ASM_ATTRIBUTES 가 마추졌을 때,
   우리는 'struct insn_def' 내에 그에 대한 정보를 모두 기록한다.  이것은 attribute
   정의들이 파일의 어느곳에서 나타날 수 있도록 도와준다.  */

struct insn_def
{
```

```
  struct insn_def *next;        /* 연결된 다음 insn.  */
  rtx def;                      /* DEFINE_...  */
  int insn_code;                /* Instruction 번호.  */
  int insn_index;               /* 파일에서 expression 번호, 오류용.  */
  int lineno;                   /* 줄번호.  */
  int num_alternatives;         /* Alternative 들의 갯수.  */
  int vec_idx;                  /* 'def' 내 attribute vector 의 index.  */
};

/* 모든 것이 읽어진 후, 우리는 각 attribute 값에 해당 값을 가지는 insn code 목록을
   저장한다.  이것은 그 해당 list 에 사용되는 구조체이다.  */

struct insn_ent
{
  struct insn_ent *next;        /* 연결된 다음.  */
  int insn_code;                /* Instruction 번호.  */
  int insn_index;               /* 파일에서 정의에 관한 index */
  int lineno;                   /* 줄번호.  */
};

static struct insn_def *defs;
```

4.2.2 DEFINE_ATTR 관련 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 각 attribute 의 값 (constant 혹은 계산되어진 것) 은 구조체를 할당하는데,
   해당 값을 가지는 insn 들의 listhead 로써 사용된다.  */

struct attr_value
{
  rtx value;                    /* Attribute 의 값.  */
  struct attr_value *next;      /* 연결된 다음 attribute 값.  */
  struct insn_ent *first_insn;  /* 이 값을 가지는 첫번째 insn.  */
  int num_insns;                /* 이 값을 가지는 insn 들의 갯수.  */
  int has_asm_insn;             /* 만약 'asm' insn 들을 위해 이 값이
                                   사용될 경우 true.  */
};

/* 각 attribute 를 위한 구조체.  */

struct attr_desc
{
  char *name;                   /* attribute 의 이름.  */
  struct attr_desc *next;       /* 다음 attribute.  */
  unsigned is_numeric   : 1;    /* 이 attribute 의 값은 numeric 이다.  */
  unsigned negative_ok  : 1;    /* Negative numeric 값을 허용한다.  */
  unsigned unsigned_p   : 1;    /* output function 이 unsigned int 로 만든다.  */
  unsigned is_const     : 1;    /* 각 run 을 위한 attribute value constant.  */
  unsigned is_special   : 1;    /* 'write_attr_set' 를 호출하지 않는다.  */
  unsigned func_units_p : 1;    /* 이것은 function_units attribute 이다.  */
  unsigned blockage_p   : 1;    /* 이것은 blockage range function 이다.  */
  struct attr_value *first_value; /* 이 attribute 의 첫번째 값.  */
```

```
  struct attr_value *default_val; /* 이 attribute 의 기본값.  */
  int lineno;                     /* 줄번호.  */
};

#define NULL_ATTR (struct attr_desc *) NULL
#define MAX_ATTRS_INDEX 256
static struct attr_desc *attrs[MAX_ATTRS_INDEX];
```

4.2.3 DEFINE_FUNCTION_UNIT 관련 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 각 DEFINE_FUNCTION_UNIT 에 관한 정보를 기록한다.  */

struct function_unit_op
{
  rtx condexp;                    /* 이용가능한 insn 를 위한 expression TRUE.  */
  struct function_unit_op *next; /* 이 function unit 을 위한 다음 operation.  */
  int num;                        /* Unit 내 이 operation type 을 위한 ordinal.  */
  int ready;                      /* Data 가 ready 할 때까지의 cost.  */
  int issue_delay;                /* Unit 이 다른 insn 를 받아들일 수
                                     있을 때까지의 cost.  */
  rtx conflict_exp;               /* Expression TRUE for insns incurring issue delay.  */
  rtx issue_exp;                  /* Issue delay 를 계산하는 표현식.  */
  int lineno;                     /* 줄 번호.  */
};

/* DEFINE_FUNCTION_UNIT 에서 언급된 각 function unit 에 관한 정보를
   기록한다.  */

struct function_unit
{
  const char *name;               /* Function unit 이름.  */
  struct function_unit *next;     /* 다음 function unit.  */
  int num;                        /* 이 unit type 의 ordinal.  */
  int multiplicity;               /* 이 type 의 unit 들의 갯수.  */
  int simultaneity;               /* 이 function unit 에 대해 simultaneous insns
                                     의 최대 갯수.  만약 제한이 없다면 0.  */
  rtx condexp;                    /* Expression TRUE for insn needing unit.  */
  int num_opclasses;              /* 다른 operation type 들의 갯수.  */
  struct function_unit_op *ops; /* 첫번째 operation type 으로의 포인터.  */
  int needs_conflict_function;  /* 만약 conflict function 가 필요할
                                     경우 0 이 아닌 값.  */
  int needs_blockage_function;  /* 만약 blockage function 가 필요할
                                     경우 0 이 아닌 값.  */
  int needs_range_function;     /* 만약 blockage range function 가
                                     필요할 경우 0 이 아닌 값.  */
  rtx default_cost;               /* 만약 constant 일 경우, Conflict cost.  */
  struct range issue_delay;     /* Issue delay 값들의 범위.  */
  int max_blockage;               /* Insn 가 unit 을 block 하는데의 최대 시간.  */
  int first_lineno;               /* 처음 보였던 줄 번호.  */
};
```

4.3 전역 변수

```
/* 각 insn code 를 위해, constraint alternative 들의 갯수를 저장한다.  */

static int *insn_n_alternatives;

/* 각 insn code 를 위해, 각 가능한 alternative 를 위한 bit 를 가진 bitmap 을
   저장한다.  */

static int *insn_alternatives;
```

4.4 함수들

4.4.1 expand_units () 함수

- 우선 gen_unit () 함수에서 해석한 DEFINE_FUNCTION_UNIT 들이 들어있는 전역 변수 units 를 살피는데, 각 struct function_unit 의 condexp 를 check_attr_test () 함수를 통해서 새롭게 변형시킨다. check_attr_test () 함수에서는 (eq_attr “att” “a1,a2”) 를 (ior (eq_attr ...) (eq_attrq ..)) 로 변환하고 (eq_attr “att” “!a1”) 를 (not (eq_attr “att” “a1”)) 로 변환한다. 후자 테스트를 먼저한다.

- 각각의 struct function_unit_op 들을 살펴보면서, issue expression 을 생성하게 된다.
  만약 DEFINE_FUNCTION_UNIT 의 7 번째 인자, 즉 conflict vector 가 주어졌을 경우,
  op→conflict_exp 를 기반으로 IF_THEN_ELSE rtx 를 생성하여, simplify_knowing () 함수를 이용하여 간단화시키게 된다. 또한 unit→issue_delay.min 과 unit→issue_delay.max 가 다를 경우, conflict function 을 생성할 필요가 있기 때문에, make_internal_attr () 함수를 이용하여 전역변수 attrs 에 추가한다. 인자는 이름과 issue expression 들이다.

- Issue expression 및 conflict function 에 대한 처리가 완료되면, unitsmask 를 생성하는데, 전역 변수 units 에 있는 모든 unit 들을 살펴보며, unit→condexp 를 이용하여 IF_THEN_ELSE 구문을 만들게 된다. conditional 이 true 일 경우, 1 << unit→num 값이, 아닐 경우 0 인 표현식이다. 만들어진 IF_THEN_ELSE 구문을 각각 merging 하게 되는데, 아래와 같은 모습일 것이다.

  ```
  (if_then_else (and (eq_attr/u/i ("cpu") ("athlon"))
          (ior/i (eq_attr/i ("memory") ("load"))
              (eq_attr/i ("memory") ("both"))))
      (const_string/i ("33554432"))
      (const_string/i ("0")))
  ```

  위 표현식은 특정 unit→condexp 를 이용하여 unit→num 을 값으로 IF_THEN_ELSE 구문을 만든 경우이다.

  ```
  (if_then_else (and (eq_attr/u/i ("cpu") ("athlon"))
          (eq_attr/i ("athlon_fpunits") ("store")))
      (const_string/i ("16777216"))
      (const_string/i ("0")))
  ```

  이러한 두 표현식을 ORX_OP 를 사용하여 묶는다. 두 표현식을 묶으면 아래와 같은 표현식이 된다.

  ```
  (ior (if_then_else (and (eq_attr/u/i ("cpu") ("athlon"))
              (ior/i (eq_attr/i ("memory") ("load"))
                  (eq_attr/i ("memory") ("both"))))
          (const_string/i ("33554432"))
          (const_string/i ("0")))
      (if_then_else (and (eq_attr/u/i ("cpu") ("athlon"))
              (eq_attr/i ("athlon_fpunits") ("store")))
          (const_string/i ("16777216"))
  ```

```
        (const_string/i ("0"))))
```

이렇게 모든 unit 들의 condexp 를 묶었으면, simplify_by_exploding () 함수를 이용하여 간단화시킨다. 그런후, FFS rtx 로 unitsmask 를 감싼다. FFS rtx 로 감싸는 이유는 나중에 output 시, 이것이 function_units_used 를 위한 것인지를 쉽게 알아보기 위해서이다. 이렇게 감싼 unitsmask 는 make_internal_attr () 함수를 이용하여 "*function_units_used" 라는 attr 를 만들게 되며, 이것의 default_val 는 unitsmask 로 설정된다.

- 이제 각 unit 을 위한 ready cost function 을 계산하기 전에, 몇몇 변수를 설정을 하게 되는데, 각 변수의 용도는 다음과 같다.
  - unit_num → 각 unit 의 번호에 따라 unit 에 대한 포인터를 가지고 있다. unit_num[x] 의 마지막은 unit 을 계산하면서 보인 총 ops 의 갯수를 가지고 있다.
  - unit_ops → 각 unit 의 번호에 따라 unit 의 ops 정보들을 가지고 있다. unit_ops[x] 의 마지막은 unit 을 계산하면서 보인 ops 들의 모든 포인터를 가지고 있다.

  이제 위의 변수를 다 설정했다면, 전체 unit 들을 하나씩 살펴보면서 아래의 ready cost 를 생성한다.
  - 먼저 각 unit 에 대한 ops 의 ready cost 가 증가하는 순으로 정렬한다.
  - 얼마나 많은 distinct non-default ready cost 값들이 존재하는지 결정한다. default ready cost value 값은 1 이다. i386.md 에서의 function_unit 중 "pent_np" 는 5 이다.
  - 위에서 계산한 distinct non-default ready cost 값들을 기반으로 readycost 표현식을 생성하는데, 각각의 ops 들을 살펴보면서 이전 ready 값과 같을 경우, IOR 표현식을 만들어 합하고, COND 표현식으로 readycost 표현식을 만든다. "pent_np" 을 위해 생성된 readycost 표현식을 보인다면 아래와 같은 모습이다.

```
(cond[
        (and (eq_attr/u/i ("cpu") ("pentium"))
            (eq_attr/i ("type") ("idiv")))
        (const_string/i ("46"))
        (and (eq_attr/u/i ("cpu") ("pentium"))
            (eq_attr/i ("type") ("str")))
        (const_string/i ("12"))
        (and (eq_attr/u/i ("cpu") ("pentium"))
            (eq_attr/i ("type") ("imul")))
        (const_string/i ("11"))
        (and (eq_attr/u/i ("cpu") ("pentium"))
            (ior (ior (and (eq_attr/i ("type") ("fmov"))
                           (and (ior/i (eq_attr/i ("memory") ("load"))
                                       (eq_attr/i ("memory") ("store")))
                                (eq_attr/i ("mode") ("XF"))))
                      (and (ior/i (eq_attr/i ("type") ("alu"))
                                  (ior/i (eq_attr/i ("type") ("alu1"))
                                         (ior/i (eq_attr/i ("type") ("negnot"))
                                                (eq_attr/i ("type") ("ishift")))))
                           (and (eq_attr/i ("pent_pair") ("np"))
                                (eq_attr/i ("memory") ("both")))))
                 (and (ior/i (eq_attr/i ("type") ("alu"))
                             (ior/i (eq_attr/i ("type") ("alu1"))
                                    (eq_attr/i ("type") ("ishift"))))
                      (and (not/i (eq_attr/i ("pent_pair") ("np")))
                           (eq_attr/i ("memory") ("both"))))))
        (const_string/i ("3"))
```

```
        (and (eq_attr/u/i ("cpu") ("pentium"))
            (ior 이하 임의생략)
        (const_string/i ("2"))
    ]
    (const_string/i ("1")))
```

- 여기서 현재 처리하고 있는 unit 이 unit_num[x] 의 마지막은 unit (총 합 unit) 이 아닐 경우, 아래와 같은 처리를 수행한다. 만약 마지막 unit (총 합 unit) 일 경우, 아래의 부분을 처리하지 않고, 단순히 위에서 생성한 readycost 를 기반으로 "*result_ready_cost" 라는 attribute 를 생성하게 된다.
  - Unit 의 ops 들을 살펴보면서, max_blockage 와 min_blockage 를 계산하고 각 ops 들을 위한 blockage function 를 전역변수 attrs 에 make_internal_attr () 함수를 이용하여 등록한다.

    max_blockage 와 min_blockage 를 계산하는 과정에 대해서는 아래의 operate_exp () 함수 설명에 자세히 나와 있으니, 참조하기 바란다. operate_exp () 함수에 의해서 계산된 rtx 는 simplify_knowning () 함수에 의해 간단화되는데, 여기서 중요한 것은 rtx 내에 포함되어 있는 CONST_STRING 으로, 이 값이 서로 다르다면 복잡하게 얽힌 max_blockage 혹은 min_blockage 표현식이 나올 것이다. 하지만 이 CONST_STRING 값이 서로 같다면, simplify_knowning () 함수에 의해서 최적화되어 단순히 (const_string ...) node 만 남게 될 것이다.
  - 위에서 계산된 max_blockage 를 이용하여 unit→max_blockage 를 계산한다. max_attr_value () 함수에서 수행되며, 각 rtx 를 방문하여 CONST_STRING 의 최대 크기를 구한다.
  - 앞에서 계산한 max_blockage 와 min_blockage 를 기반으로 unit→needs_blockage_function 와 unit→needs_range_function 가 필요한지를 계산하고 unit→needs_range_function 가 1 일 경우, blockage range function 를 계산하고 그것의 값을 쓰기 위한 attribute 를 만든다. unit→needs_blockage_function 가 판단되는 기준은 max_blockage 와 min_blockage 가 서로 다를 경우이며, 같으면 생성되지 않는다. unit→needs_range_function 에서의 range function 은 min 과 max 가 존재하는데, 하나의 int (4 바이트) 에 넣기 위해서 16 비트씩 잘라서 사용하며, High 부분은 min 이, Low 부분은 max 가 각각 16 비트씩 사용한다. 예를 들어 131128 정수값은 2 진수로 10000000000111000 이며, min 은 10, max 는 1110000 으로, 십진수로 표현하면, min = 2, max = 56 값을 나타낸다. 이 부분은 operate_exp 에서 RANGE_OP 에서 만들어진다.
- 마지막으로 앞에서 계산한 readycost 를 기반으로 해당 unit 을 위한 ready_cost function 를 위해 attribute 를 만든다.

- Conflict cost function 를 요구하는 각 unit 을 위해, insn 들을 operation number 로 매핑하는 attribute 를 만든다. 대부분 이름이 *_cases 로 끝나는 것들이다.

4.4.2 Conflict function 의 생성 조건

- Conflict function 이 생성되기 위해서는 우선 해당 unit (struct function_unit) 의 구성요소인 unit→needs_conflict_function 가 1 로 설정되어 있어야 하는데, 이것이 1 로 설정되기 위해서는 unit→issue_delay.min 와 unit→issue_delay.max 의 값이 달라야 한다. 이 두 값이 서로 다르기 위해서는 DEFINE_FUNCTION_UNIT 의 7 번째 인자에 RTX 표현식들이 추가적으로 나타내야 한다.

- 생성 조건을 만족하게 되면, 해당 unit 의 op 들을 위한 issue expression 을 생성하게 되는데, DEFINE_FUNCTION_UNIT 의 7 번째 인자가 지정되어 있을 경우, 대부분 attribute 에 issue expresion 과 이름을 추가함으로써 output 시 함수가 output 되게 하는데, (이 부분은 make_internal_attr () 함수에 의해서 이루어진다.) 이 issue expression 이 만들어지는 부분에 대해서 살펴보면 아래와 같다.
  - DEFINE_FUNCTION_UNIT 의 7 번째 인자가 지정되어 있을 경우, IF_THEN_ELSE rtx 를 생성하게 되는데, conditional 은 op→conflict_exp 이고, 조건이 true 일 경우, make_numeric_value (op→issue_delay) 가 false 일 경우, make_numeric_value (0) 형태를 가지는 rtx 구문이다. 이 IF_THEN_ELSE rtx 는 뒤에서 COND 구문으로 변경되게 된다.

– 위에서 IF_THEN_ELSE rtx 구문이 COND rtx 구문으로 변경되었을 때, unit→condexp (현재 unit 에서의 조건 표현식으로써 같은 이름을 가진 DEFINE_FUNCTION_UNIT 의 조건들을 모두 IOR 연산으로 묶은 표현식) 를 기반으로 간단화를 수행한다.

4.4.3 simplify_knowing () 함수

이 함수가 호출되어 최적화가 이루어진다는 것은, DEFINE_FUNCTION_UNIT 에 부과적인 conflict vector 가 지정되어 있다는 말이며, unit 의 구성요소인 issue_delay 의 min 과 max 의 값이 달라 conflict function 을 만들도록 한다. 이때 conflict function 을 만들도록 할 때, make_internal_attr () 함수를 통해서 전역변수 attrs 에 추가하는 형식을 하며, attr 의 default_val 가 issue expresion 으로 설정되게 된다.

- 첫번째 인자

```
(and (eq_attr/u/i ("cpu") ("athlon"))
    (ior (eq_attr/i ("athlon_decode") ("vector"))
        (eq_attr/i ("athlon_decode") ("direct"))))
```

- 두번째 인자

```
(cond[
        (eq_attr/i ("athlon_decode") ("vector"))
        (const_string/i ("1"))
    ]
    (const_string/i ("0")))
```

- 결과

```
(cond[
        (and/i (eq_attr/i ("athlon_decode") ("direct"))
            (eq_attr/u/i ("cpu") ("athlon")))
        (const_string/i ("0"))
    ]
    (const_string/i ("1")))
```

첫번째 인자가 true 이면, 두번째 인자인 IF_THEN_ELSE rtx 를 최적화시키면 결과가 나온다.

4.4.4 operate_exp () 함수

- DEFINE_FUNCTION_UNIT 을 처리하는데만 사용된다.

LEFT	RIGHT	return 결과값	기타 조건
CONST_STRING	CONST_STRING	CONST_STRING	
CONST_STRING	IF_THEN_ELSE	IF_THEN_ELSE	
CONST_STRING	COND	COND	
IF_THEN_ELSE	anything	IF_THEN_ELSE	
COND	anything	COND	
anything	IF_THEN_ELSE	IOR	op 가 ORX_OP 일 경우

이 함수의 수행과정을 살펴보면 아래와 같다.

– 이 함수의 경우 인자를 세개 받아 들이는데, operator, LEFT, RIGHT 이며, 이하는 모두 재귀호출로 이루어진다.

- 위의 표를 보면 알 수 있겠지만, 결과적으로 이야기해서 LEFT 에 존재하는 CONST_STRING 을 모두 찾아서, RIGHT 의 표현식으로 치환버린다. 이 과정에서 LEFT 에서 선택된 CONST_STRING 과 RIGHT 내에 존재하는 모든 CONST_STRING 들과 operator 연산이 이루어진다. 예를 들어 설명하면, operator 가 MAX_OP 일 경우, LEFT 의 CONST_STRING 과 RIGHT 내에 존재하는 모든 CONST_STRING 과 값을 비교해서 최대값으로 RIGHT 의 CONST_STRING 들을 바꾼다.
- 자세한 예를 든다면 아래와 같은데, (앞에 있는 번호는 줄 번호이다.) 아래에 LEFT 모습과

```
01: (if_then_else 조건1
02:    (if_then_else 조건1
03:        (if_then_else 조건1
04:            (cond[
05:                    조건2
06:                    (const_string/i ("2"))
07:                ]
08:                (if_then_else 조건1
09:                    (if_then_else 조건1
10:                        (cond[
11:                                (eq_attr/i ("pent_pair") ("np"))
12:                                (const_string/i ("1"))
13:                            ]
14:                            (const_string/i ("0")))
15:                        (const_string/i ("1")))
16:                    (const_string/i ("1"))))
17:            (const_string/i ("2")))
18:        (const_string/i ("2")))
19:    (const_string/i ("2")))
```

RIGHT 모습이 아래에 존재한다.

```
01: (if_then_else 조건1
02:     (if_then_else 조건2
03:         (cond[
04:                 (eq_attr/i ("pent_pair") ("np"))
05:                 (const_string/i ("2"))
06:             ]
07:             (const_string/i ("0")))
08:         (const_string/i ("2")))
09:     (const_string/i ("2")))
```

이제 첫번째 연산의 경우, LEFT 의 17 번째 줄부터 시작하는데, operator 가 MAX_OP 일 경우, RIGHT 에 존재하는 모든 CONST_STRING 과 비교해서 큰 값이 완성될 것인데, 위의 RIGHT 중 7 번째 줄이 0 이기 때문에, MAX 연산에 의해서 2 로 변경될 것이다. 그럼 17 번째 줄은 CONST_STRING 에서 IF_THEN_ELSE 로 변경된 모습일 것이며, 모습은 아래와 같을 것이다.

```
(if_then_else 조건1
    (if_then_else 조건1
        (if_then_else 조건1
            (cond[
                    조건2
                    (const_string/i ("2"))
                ]
                (if_then_else 조건1
                    (if_then_else 조건1
```

```
                    (cond[
                            (eq_attr/i ("pent_pair") ("np"))
                            (const_string/i ("1"))
                        ]
                        (const_string/i ("0")))
                    (const_string/i ("1")))
                (const_string/i ("1"))))
            (if_then_else 조건1
                (if_then_else 조건2
                    (cond[
                        (eq_attr/i ("pent_pair") ("np"))
                        (const_string/i ("2"))
                    ]
                        (const_string/i ("0")))
                    (const_string/i ("2")))
                (const_string/i ("2"))))
        (const_string/i ("2")))
    (const_string/i ("2")))
```

계속 LEFT 모습은 이러한 형식으로 계속 변경될 것이다. LEFT 의 마지막 CONST_STRING 을 치환할 때까지 이 operate_exp () 함수의 재귀 호출은 계속 될 것이다.

4.5 출력

4.5.1 ATTRIBUTE 들의 출력

Attribute 들의 경우는, make_internal_attr () 함수에 의해서 추가된 attribute 들도 고려해야 하는데, 하나의 attribute 는 write_attr_get () 함수에 의해서 하나씩 출력되게 된다. 아래에서는 이 write_attr_get () 함수에 대해서 좀 더 자세히 알아 보도록 하자.

- write_attr_get () 함수에서 읽게 되는, common_av 는 default 라벨에 들어갈 rtx 표현식을 가지고 있는데, find_most_used () 함수를 이용하여 가장 많이 사용되는 값을 찾는다. 가장 많이 사용되기 때문에 default 라벨로 묶는 것이 가장 효율적이기 때문이다.
- write_attr_case () 함수의 경우, SWITCH 구문에서 하나의 case 를 출력하는데 사용된다.
  - walk_attr_value () 함수를 통해서, 전역변수 must_extract, must_constrain, address_used, length_used 값을 설정하는데, 이 전역변수들은 다음과 같은 의미를 가진다.
    - must_extract 는 insn operand 들을 extract 할 필요가 있을 경우 1 로 설정된다. SYMBOL_REF, MATCH_OPERAND, EQ_ATTR, MATCH_DUP 와 같은 표현식이 있을 경우 1 로 설정되는데, 각각 조건이 있다.
    - must_constrain 는 우리가 반드시 which_alternative 를 계산할 필요가 있을 경우 1 로 설정된다. SYMBOL_REF, EQ_ATTR 와 같은 표현식이 있을 경우 1 로 설정되는데, 각각 조건이 있다.
    - address_used 는 만약 address expression 가 사용되었을 경우 1 로 설정된다. MATCH_DUP, PC 와 같은 표현식이 있을 경우 1 로 설정된다.
    - length_used 는 (eq_attr "length" ...) 형태가 사용되었을 경우 1 로 설정된다. EQ_ATTR 와 같은 표현식이 있을 경우 1 로 설정되는데, 조건이 있다.
  - write_attr_set () 함수의 재귀적인 호출을 통해서 struct attr_value 내 표현식을 C 표현식으로 옮긴다.

4.5.2 Function Unit 들의 출력

DEFINE_FUNCTION_UNIT 들에 대한 output 은 write_function_unit_info () 함수에서 이루어진다.

제 5 절 gencheck

Tree code 들을 위한 check macro 들을 생성한다.

5.1 작동 방법

$prefix/gcc/tree.def, $prefix/gcc/c-common.def, $prefix/gcc/gencheck.h 파일에 정의되어 있는 SYMBOL 들을 읽어서 아래와 같은 파일을 만든다.

5.2 $prefix/gcc/tree-check.h

```
/* This file is generated using gencheck. Do not edit. */

#ifndef GCC_TREE_CHECK_H
#define GCC_TREE_CHECK_H

#define ERROR_MARK_CHECK(t)     TREE_CHECK (t, ERROR_MARK)
#define IDENTIFIER_NODE_CHECK(t)        TREE_CHECK (t, IDENTIFIER_NODE)
#define TREE_LIST_CHECK(t)      TREE_CHECK (t, TREE_LIST)
#define TREE_VEC_CHECK(t)       TREE_CHECK (t, TREE_VEC)
...
...
(그 외 다수 존재)
```

제 6 절 gencodes

Machine description 으로 부터 아래와 같은 내용을 생성: - 각각의 정의된 표준 insn name 을 위한 insn_code_number 값을 가지는 몇몇 종류의 macro 들. CODE_FOR_... 형태의 macro.

6.1 목적

gencodes 의 목적은 insn-codes.h 파일을 작성하는데 있으며, 결과적으로는 enum insn_code 를 구성하는데 있다.

6.2 작동 방법

대단히 간단한데, init_md_reader_args () 함수를 읽은 후, *_queue 에 저장되어 있는 RTX 중 DEFINE_INSN 와 DEFINE_EXPAND 의 이름과 insn_code_number 를 읽어 출력하는데, 이 insn_code_number 는 gensupport 에서 언급된 전역 변수 sequence_num 의 값이다. (물론 MD 를 해석하면서 instruction 을 발견하면 이 변수의 값은 1 씩 증가한다.

6.3 gencodes 가 생성하는 insn-codes.h 내용

```
/* Generated automatically by the program ‘gencodes’
   from the machine description file ‘md’.  */

#ifndef GCC_INSN_CODES_H
#define GCC_INSN_CODES_H

enum insn_code {
  CODE_FOR_cmpdi_ccno_1_rex64 = 0,
  CODE_FOR_cmpdi_1_insn_rex64 = 2,
  CODE_FOR_cmpqi_ext_3_insn = 15,
  CODE_FOR_cmpqi_ext_3_insn_rex64 = 16,
  CODE_FOR_x86_fnstsw_1 = 30,
  CODE_FOR_x86_sahf_1 = 31,
  CODE_FOR_popsi1 = 42,
  CODE_FOR_movsi_insv_1 = 73,
  CODE_FOR_pushdi2_rex64 = 77,
  CODE_FOR_popdi1 = 80,
  CODE_FOR_swapxf = 105,
  CODE_FOR_swaptf = 106,
  CODE_FOR_zero_extendhisi2_and = 107,
  CODE_FOR_zero_extendsidi2_32 = 115,
  CODE_FOR_zero_extendsidi2_rex64 = 116,
  CODE_FOR_zero_extendhidi2 = 117,
...
...
(줄 임)
```

제 7 절 genconfig

Machine description 으로 부터 아래와 같은 내용을 생성: - 몇몇 #define 설정 flag 들.

7.1 목적

아래와 같은 MACRO 들의 적당한 값들을 설정하는 것이 목적이다.

- MAX_RECOG_OPERANDS
- MAX_DUP_OPERANDS
- MAX_INSNS_PER_SPLIT
- HAVE_cc0
- HAVE_conditional_move
- HAVE_conditional_execution
- HAVE_lo_sum
- HAVE_peephole
- HAVE_peephole2
- MAX_INSNS_PER_PEEP2

7.2 작동 방법

- MAX_RECOG_OPERANDS

  전역 변수 max_recog_operands 에 이 매크로의 값이 담기며, 기본값은 29 로 설정되어 있다. MATCH_OPERAND, MATCH_OPERATOR, MATCH_DUP 의 첫번째 값의 범위에 따라 이 값이 변경되게 된다.

- MAX_DUP_OPERANDS

  전역 변수 max_dup_operands 에 이 매크로의 값이 담기며, 기본값은 1 로 설정되어 있다. 이 값은 한 insn 내에 존재하는 MATCH_OP_DUP, MATCH_PAR_DUP, MATCH_DUP 의 최대 갯수를 구하는 것이다.

- MAX_INSNS_PER_SPLIT

  전역 변수 max_insns_per_split 에 이 매크로의 값이 담기며, 기본값은 1 로 설정되어 있다. 이 값은 gen_split () 함수를 처리할 때 변경될 수 있으며, 하나의 define_split 에 존재하는 insn 가 나누어질 수 있는 insn 들의 최대 갯수를 말한다. define_split 의 세번째 인자이며, vector 로 묶여 있는 것이 일반적이다.

- HAVE_cc0

  Recog pattern 에서 CC0 가 보일 경우 1

- HAVE_conditional_move

  IF_THEN_ELSE 의 두 arm 이 모두 MATCH_OPERAND 일 경우, 이 machine 이 conditional move 를 가지고 있다고 간주한다.

- HAVE_conditional_execution

Recog pattern 에서 COND_EXEC 가 보일 경우 1

- HAVE_lo_sum

  Recog pattern 에서 LO_SUM 가 보일 경우 1

- HAVE_peephole

  DEFINE_PEEPHOLE 이 정의되어 있을 경우 1

- HAVE_peephole2

  DEFINE_PEEPHOLE2 가 정의되어 있을 경우 1

- MAX_INSNS_PER_PEEP2

  DEFINE_PEEPHOLE2 의 첫번째 인자인 vector 에서 구성요소 RTX 의 code 값이 MATCH_DUP 와 MATCH_SCRATCH 를 제외한 최대 갯수. 예를 들어 아래와 같은 예제가 있다고 했을 때, 최대 갯수는 1 (SET 하나) 이다.

```
(define_peephole2
  [(match_scratch:DI 2 "r")
   (set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")
```

7.3 genconfig 가 생성하는 insn-config.h 내용

```
/* Generated automatically by the program ‘genconfig’
   from the machine description file ‘md’.  */

#ifndef GCC_INSN_CONFIG_H
#define GCC_INSN_CONFIG_H

#define MAX_RECOG_OPERANDS 30
#define MAX_DUP_OPERANDS 4
#ifndef MAX_INSNS_PER_SPLIT
#define MAX_INSNS_PER_SPLIT 4
#endif
#define HAVE_conditional_move 1
#define HAVE_peephole2 1
#define MAX_INSNS_PER_PEEP2 4

#endif /* GCC_INSN_CONFIG_H */
```

제 8 절 genconstants

Machine description 으로 부터 아래와 같은 내용을 생성: 여러 #define statement 에 관한 것을 만드는데 각 constant 이름은 (define_constants ...) pattern 으로 이루어진다.

8.1 목적

define_constants 를 처리하기 위해서 존재하며 결과적으로 insn-constants.h 파일을 생성한다.

8.2 작동 방법

MD 파일을 해석하면서 define_constants 를 만날 경우, struct md_constant *def 를 구성한 후 md_constants 에 삽입한다. 이 부분은 init_md_reader () 함수에서 수행되며, 결과로 만들어진 md_constants 는 traverse_md_constants () 함수를 통해서 내용이 출력되게 된다. (물론 출력하는 함수 포인터 print_md_constant 를 건네준다.)

8.3 전역 변수

```
static htab_t md_constants;

struct md_constant { char *name, *value; };
```

제 9 절 genemit

Machine description 으로 부터 rtl 로 insn 들을 만들어줄 code 를 생성.

9.1 목적

- DEFINE_INSN 일 경우, pattern 과 같은 RTX 표현식을 생성하는 C 코드를 작성한다. 물론 적당한 대치가 이루어진다.
- DEFINE_EXPAND 일 경우, 네번째 인자인 C 표현식을 우선 실행하고, pattern 과 같은 RTX 표현식을 emit_insn () 함수를 통해서 현재 sequence 에 emit 한다. 그런 후, gen_sequence () 함수를 통해서, 현재까지 SEQUENCE 에 들어온 것을 기반으로 RTX 를 반환한다.

9.2 작동 방법

DEFINE_INSN 와 DEFINE_EXPAND, DEFINE_SPLIT, DEFINE_PEEPHOLE2 를 처리한다.

9.2.1 DEFINE_INSN

해당 pattern 을 위한 RTX 를 생성할 C 표현식을 생성하는데, pattern 내에 CLOBBER 가 존재할 경우, struct clobber_pat 와 struct clobber_ent 를 작성해서, 전역 변수 clobber_list 에 등록할 의무를 가지고 있다.

9.2.2 DEFINE_EXPAND

DEFINE_EXPAND 의 처리는 결과적으로 아래와 같은 C 코드 생성에 있다.

```
rtx
gen_cmpdi (operand0, operand1)
     rtx operand0;
     rtx operand1;
{
  rtx _val = 0;
  start_sequence ();
  {
    rtx operands[2];
    operands[0] = operand0;
    operands[1] = operand1;
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (DImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
}
    operand0 = operands[0];
    operand1 = operands[1];
  }
  emit_insn (gen_rtx_SET (VOIDmode,
        gen_rtx_REG (CCmode,
        17),
        gen_rtx_COMPARE (CCmode,
        operand0,
        operand1)));
```

```
  _val = gen_sequence ();
  end_sequence ();
  return _val;
}
```

이는 다음과 같은 define_expand 를 해석한 결과이다.

```
(define_expand "cmpdi"
  [(set (reg:CC 17)
        (compare:CC (match_operand:DI 0 "nonimmediate_operand" "")
                    (match_operand:DI 1 "x86_64_general_operand" "")))]
  ""
{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[0] = force_reg (DImode, operands[0]);
  ix86_compare_op0 = operands[0];
  ix86_compare_op1 = operands[1];
  DONE;
})
```

이것의 네번째 인자가 사용되는 부분은 크게 고려 사항이 없지만, emit_insn () 함수를 결정하는데는 고려 사항이 존재한다. 즉 emit_insn () 함수가 사용될 수 있고, emit_jump_insn () 혹은 emit_call_insn (), emit_label (), emit (), emit_barrier () 가 사용될 수 있다는 말이다. 아래는 우선순위순이다.

- emit_jump_insn () 함수

  (SET (PC) ...) 패턴 혹은 (PARALLEL (SET (PC) ...) ...) 일 경우 선택된다.

- emit_call_insn () 함수

  (SET (CALL) ...) 패턴 혹은 (CALL ...) 패턴, (PARALLEL (SET (CALL) ...) ...), (PARALLEL (CALL) ...) ...) 일 때 선택된다.

- emit_label () 함수

  (CODE_LABEL ...) 패턴일 때 선택된다.

- emit () 함수

  MATCH_OPERAND 혹은 MATCH_DUP, MATCH_OPERATOR, MATCH_OP_DUP, MATCH_PARALLEL, MATCH_PAR_DUP, PARALLEL 일 경우 선택된다.

- emit_insn () 함수

  위의 것이 모두 해당되지 않을 경우 선택된다.

9.2.3 DEFINE_SPLIT

Expand 와 비슷하다. 하지만 DEFINE_EXPAND 의 경우, 생성되는 함수의 인자를 operand0, operand1 이런 형식으로 받지만, DEFINE_SPLIT 는 rtx *operands 이러한 형식으로 받아 들인다. 예를 들면 아래와 같은 코드는

```
(define_split
  [(set (reg:CCFP 18)
        (compare:CCFP
          (match_operand:SF 0 "register_operand" "")
          (float (match_operand:SI 1 "register_operand" ""))))]
```

```
  "0 && TARGET_80387 && reload_completed"
  [(set (mem:SI (pre_dec:SI (reg:SI 7))) (match_dup 1))
   (set (reg:CCFP 18) (compare:CCFP (match_dup 0) (match_dup 2)))
   (parallel [(set (match_dup 1) (mem:SI (reg:SI 7)))
              (set (reg:SI 7) (plus:SI (reg:SI 7) (const_int 4)))])]
  "operands[2] = gen_rtx_MEM (Pmode, stack_pointer_rtx);
   operands[2] = gen_rtx_FLOAT (GET_MODE (operands[0]), operands[2]);")
```

아래와 같은 C 코드로 변환된다. 실제 emit 되는 부분은 세번째 인자이다.

```
extern rtx gen_split_845 PARAMS ((rtx *));
rtx
gen_split_845 (operands)
      rtx *operands;
{
  rtx operand0;
  rtx operand1;
  rtx operand2;
  rtx _val = 0;
  start_sequence ();
operands[2] = gen_rtx_MEM (Pmode, stack_pointer_rtx);
   operands[2] = gen_rtx_FLOAT (GET_MODE (operands[0]), operands[2]);
  operand0 = operands[0];
  operand1 = operands[1];
  operand2 = operands[2];
  emit_insn (gen_rtx_SET (VOIDmode,
        gen_rtx_MEM (SImode,
        gen_rtx_PRE_DEC (SImode,
        gen_rtx_REG (SImode,
        7))),
        operand1));
  emit_insn (gen_rtx_SET (VOIDmode,
        gen_rtx_REG (CCFPmode,
        18),
        gen_rtx_COMPARE (CCFPmode,
        operand0,
        operand2)));
  emit (gen_rtx_PARALLEL (VOIDmode,
        gen_rtvec (2,
                gen_rtx_SET (VOIDmode,
        copy_rtx (operand1),
        gen_rtx_MEM (SImode,
        gen_rtx_REG (SImode,
        7))),
                gen_rtx_SET (VOIDmode,
        gen_rtx_REG (SImode,
        7),
        gen_rtx_PLUS (SImode,
        gen_rtx_REG (SImode,
        7),
        GEN_INT (4))))));
  _val = gen_sequence ();
  end_sequence ();
```

```
  return _val;
}
```

DEFINE_SPLIT 와 DEFINE_PEEPHOLE2 를 처리할 때, $prefix/gcc/genemit.c 파일에 존재하는 지역 변수 used 를 사용하여 gen_exp () 함수 (이 함수는 실질적인 RTX 표현식을 C 표현식으로 바꿔서 출력한다.) 를 호출하는데, used 는 RTX 표현식을 위한 C 표현식을 만들 때, 이미 앞에서 operand 가 사용되었다면, 해당 operand 에 대해서 "copy_rtx (operand%d)" 와 같은 표현식을 출력하는 것을 말한다. 예를 들어 설명하면, 아래와 같은 C 표현식은 DEFINE_SPLIT 를 처리한 후 생성된 표현식인데,

```
extern rtx gen_split_879 PARAMS ((rtx *));
rtx
gen_split_879 (operands)
      rtx *operands;
{
  rtx operand0;
  rtx _val = 0;
  start_sequence ();

  operand0 = operands[0];
  emit (gen_rtx_PARALLEL (VOIDmode,
        gen_rtvec (2,
                gen_rtx_SET (VOIDmode,
        operand0,
        gen_rtx_AND (SImode,
        copy_rtx (operand0),
        GEN_INT (65535))),
                gen_rtx_CLOBBER (VOIDmode,
        gen_rtx_REG (CCmode,
        17)))));
  _val = gen_sequence ();
  end_sequence ();
  return _val;
}
```

위 부분에서 copy_rtx() 함수를 보면, operand0 를 복사하는 것을 볼 수 있는데, 이를 사용한 이유가 위에서 operand0 을 이미 사용하였기 때문에 이러는 것이다.

9.2.4 DEFINE_PEEPHOLE2

DEFINE_SPLIT 와 비슷하지만, peephole2 pattern 에 사용되는 scratch register 들을 위해 필요한 find_free_register () 함수들을 정의하는 부분이 추가되어 있다. 이 부분은 genemit.c 파일의 output_peephole2_scratches () 함수에서 수행을 하는데, 아래와 같은 RTX 가 있다고 가정을 하였을 때,

```
(define_peephole2
  [(match_scratch:DI 2 "r")
   (set (match_operand:DI 0 "push_operand" "")
        (match_operand:DI 1 "immediate_operand" ""))]
  "TARGET_64BIT && !symbolic_operand (operands[1], DImode)
   && !x86_64_immediate_operand (operands[1], DImode)"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))]
  "")
```

이를 이용하여 아래와 같은 C 표현식을 만들어 내며, 그중 if 구문이 output_peephole2_scratches () 함수에 의해서 만들어진 부분이다.

```
extern rtx gen_peephole2_853 PARAMS ((rtx, rtx *));
rtx
gen_peephole2_853 (curr_insn, operands)
     rtx curr_insn ATTRIBUTE_UNUSED;
     rtx *operands;
{
  rtx operand0;
  rtx operand1;
  rtx operand2;
  rtx _val = 0;
  HARD_REG_SET _regs_allocated;
  CLEAR_HARD_REG_SET (_regs_allocated);
  if ((operands[2] = peep2_find_free_register (0, 0, "r", DImode,
                                               &_regs_allocated)) == NULL_RTX)
    return NULL;
  start_sequence ();

  operand0 = operands[0];
  operand1 = operands[1];
  operand2 = operands[2];
  emit_insn (gen_rtx_SET (VOIDmode,
        operand2,
        operand1));
  emit_insn (gen_rtx_SET (VOIDmode,
        operand0,
        copy_rtx (operand2)));
  _val = gen_sequence ();
  end_sequence ();
  return _val;
}
```

이는 DEFINE_PEEPHOLE2 의 첫번째 vector 를 살펴보아서, MATCH_SCRATCH 를 발견할 경우, 이에 대한 정보를 기록하게 되는데, if 구문에서 결정되는 integer 값들 (insn_nr, last_insn_nr 등등) 에 대해서는 스스로 살펴보기 바란다.

9.2.5 Clobbers

위에서 DEFINE_INSN, DEFINE_EXPAND, DEFINE_SPLIT, DEFINE_PEEPHOLE2 에 대한 정보를 모두 출력하였으면, 마지막으로 output_add_clobbers () 함수와 output_added_clobbers_hard_reg_p () 함수를 이용하여, 아래와 같은 두 함수를 만든다.

```
void
add_clobbers (pattern, insn_code_number)
     rtx pattern ATTRIBUTE_UNUSED;
     int insn_code_number;

int
added_clobbers_hard_reg_p (insn_code_number)
     int insn_code_number;
```

이 함수는 DEFINE_INSN 를 처리하면서 획득한, clobber 정보들로, 전역 변수 clobber_list 에 있는 것을 바탕으로 작성되는데, added_clobbers_hard_reg_p () 함수의 경우, clobbers[x]→has_hard_reg 를 기반으로 0 인 것과 1 인 것을 정리하여 생성하게 된다.

9.3 전역 변수

```
static int max_opno;
static int max_dup_opno;
static int max_scratch_opno;
static int register_constraints;
static int insn_code_number;
static int insn_index_number;
```

9.4 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* CLOBBER 들을 가는 insn 들의 pattern 들을 기록하기 위한 data structure.
   우리는 previously-allocated PARALLEL expression 에 이 CLOBBER
   들을 더한 function 을 output 하기 위해 이것을 사용한다.  */

struct clobber_pat
{
  struct clobber_ent *insns;
  rtx pattern;
  int first_clobber;
  struct clobber_pat *next;
  int has_hard_reg;
} *clobber_list;

/* Clobber list 를 사용하는 한 insn 를 기록한다.  */

struct clobber_ent
{
  int code_number;              /* Insn 들만 센다.  */
  struct clobber_ent *next;
};
```

제 10 절 genextract

Machine description 에서 rtl 인 insn 로부터 operand 들을 추출해줄 code 들을 생성한다.

10.1 작동 방법

DEFINE_INSN 와 DEFINE_PEEPHOLE 만을 처리한다.

10.1.1 DEFINE_INSN 인 경우

하나의 extraction method 는 하나의 struct extraction 에 저장된다. 하나의 extraction method 로 묶이기 위한 조건은 아래와 같다.

1. 현재 해석한 insn 와 struct extraction 의 연결 리스트 header 인 extractions 의 op_count 와 같아야 한다.
2. 또한 dup_count 도 같아야 한다.
3. 현재 해석한 insn 의 oplocs 와 duplocs, dupnums 도 extractions 의 것과 같아야 한다.

예를 들어 설명해 보도록 하자. 아래와 같은 define_insn 이 있다고 가정하자.

```
(define_insn "cmpdi_1_insn_rex64"
  [(set (reg 17)
        (compare (match_operand:DI 0 "nonimmediate_operand" "mr,r")
                 (match_operand:DI 1 "x86_64_general_operand" "re,mr")))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCmode)"
  "cmp{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "icmp")
   (set_attr "mode" "DI")])
```

에 대한 각각의 값은 다음과 같다.

- op_count = 2 (op_count 값을 MATCH_OPERAND, MATCH_SCRATCH, MATCH_OPERATOR, MATCH_PARALLEL 를 만났을 때 증가할 수 있는데, 선정 기준은 각각의 RTX 에 대한 첫번째 인자 (물론 정수이다.) 의 값이 현재 insn 를 해석하면서 지금까지 보인 다른 RTX 의 첫번째 인자보다 클 경우, MAX 를 구함으로써 이루어진다.)
- oplocs[0] = "10"
  oplocs[1] = "11" (walk_rtx () 함수를 통해서 path 를 추적할 때, 표현식으로 전개될 때는 0 부터 9 사이의 값이 선택되고, vector 로 전개될 때는 a 부터 z 사이의 값이 선택된다. 여기서 10 의 의미를 살펴보면, 앞의 1 은 set 의 두번째 인자라는 뜻이고, 0 은 compare 의 첫번째 인자라는 뜻이다. 즉, match_operand 의 위치를 할 수 있다. 11 을 해석하면 set 의 두번째 인자인 compare 의 두번째 match_operand 를 뜻한다.)

다른 전역 변수에 대해 설명을 하면 다음과 같다.

- dup_count (walk_rtx () 함수를 재귀적으로 실행하면서 MATCH_DUP 와 MATCH_PAR_DUP, MATCH_OP_DUP 를 만났을 경우 증가한다.)
- duplocs (위 세 *_DUP 의 위치 정보를 가지고 있다. 위 oplocs 설명을 보면 알 수 있다.)
- dupnums (위 세 *_DUP 의 첫번째 인자 값이, dup_count 순서대로 들어가 있다.)

10.1.2 DEFINE_PEEPHOLE 인 경우

i386.md 에는 DEFINE_PEEPHOLE 이 없다. 이에 작동 방법을 간단히 언급하면, 전역 변수 peepholes 에 단순히 insn code number 들을 연결한다.

10.2 전역 변수

```
/* insn_code_number 로 index 된 이름들의 배열을 가지고 있다.  */
static char **insn_name_ptr = 0;
static int insn_name_ptr_size = 0;

/* Number instruction pattern 들이 다루어지며, 첫번째는 0 에서 시작한다.  */

static int insn_code_number;

/* 이 insn 에 가장 큰 operand number 를 기록한다.  */

static int op_count;

/* 위에서 설명한 문자열 포맷을 사용하는 operand 들의 위치를 기록한다.  */

static char *oplocs[MAX_RECOG_OPERANDS];

/* 각 instruction 에 나타난 MATCH_DUP 의 발생 횟수.  첫번째 발생은
   0 으로 시작한다.  */

static int dup_count;

/* 어떤 MATCH_DUP operand 들의 위치를 기록한다.  */

static char *duplocs[MAX_DUP_OPERANDS];

/* 어떤 MATCH_DUP 들의 operand number 를 기록한다.  */

static int dupnums[MAX_DUP_OPERANDS];

/* Peephole 들을 위한 insn_codes 목록을 기록한다.  */

static struct code_ptr *peepholes;

/* 지난번 때 해석된 insn 의 이름을 기록한다.  */
static const char *last_real_name = "insn";
static int last_real_code = 0;
}
```

10.3 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 이 구조체는 extractions method 들의 하나의 집합을 표현하는데 필요한 모든
   정보를 포함한다.  각 method 는 만약 operand 들이 같은 장소에 있다면 하나의
   pattern 이상에 의해 사용될 수 있다.

   각 operand 를 위한 문자열은 operand 로의 path 를 표현하고 expression 으로
   갈 때는 '0' 부터 '9' 까지를 포함하고 vector 로 갈 때는 'a' 부터 'z' 를
   포함한다.  우리는 오직 RTL 표현식의 첫번째 operand 만 vector 라고 가정한다.
   genrecog.c 는 같은 가정을 만들고 (같은 표현을 사용한다.) 현재 true 이다.  */
```

```
struct extraction
{
  int op_count;
  char *oplocs[MAX_RECOG_OPERANDS];
  int dup_count;
  char *duplocs[MAX_DUP_OPERANDS];
  int dupnums[MAX_DUP_OPERANDS];
  struct code_ptr *insns;
  struct extraction *next;
};

/* extraction method 가 사용하는 단일 insn code 를 가지고 있다.  */

struct code_ptr
{
  int insn_code;
  struct code_ptr *next;
};

static struct extraction *extractions;
```

10.4 출력

이제 위에서 해석한 것을 어떻게 출력할 것인가에 대해서 살펴보자. 하나의 struct extration 은 하나의 method 로 간주해서 전역 변수 extractions 에 있는 것을 하나씩 출력한다.
위에서 살펴본 cmpdi_1_insn_rex64 를 살펴보면 아래와 같이 생성된다.

```
    case 37:  /* *cmpfp_iu_sse_only */
    case 36:  /* *cmpfp_iu_sse */
    case 35:  /* *cmpfp_iu */
    case 34:  /* *cmpfp_i_sse_only */
    case 33:  /* *cmpfp_i_sse */
    case 32:  /* *cmpfp_i */
    case 27:  /* *cmpfp_2u */
    case 24:  /* *cmpfp_2_tf */
    case 23:  /* *cmpfp_2_xf */
    case 21:  /* *cmpfp_2_df */
    case 19:  /* *cmpfp_2_sf */
    case 10:  /* *cmpqi_1 */
    case 9:  /* *cmpqi_ccno_1 */
    case 8:  /* *cmphi_1 */
    case 6:  /* *cmphi_ccno_1 */
    case 5:  /* *cmpsi_1_insn */
    case 3:  /* *cmpsi_ccno_1 */
    case 2:  /* cmpdi_1_insn_rex64 */
    case 0:  /* cmpdi_ccno_1_rex64 */
      ro[0] = *(ro_loc[0] = &XEXP (XEXP (pat, 1), 0));
      ro[1] = *(ro_loc[1] = &XEXP (XEXP (pat, 1), 1));
      break;
```

만약 MATCH_DUP 와 MATCH_PAR_DUP, MATCH_OP_DUP 같은 *_DUP 에 대해서는 다음과 같은 code 를 생성한다.

```
    case 106:  /* swaptf */
```

```
case 105:  /* swapxf */
case 96:  /* *swapdf */
case 91:  /* *swapsf */
case 87:  /* *swapdi_rex64 */
case 61:  /* *swapqi */
case 55:  /* *swaphi_2 */
case 54:  /* *swaphi_1 */
case 48:  /* *swapsi */
  ro[0] = *(ro_loc[0] = &XEXP (XVECEXP (pat, 0, 0), 0));
  ro[1] = *(ro_loc[1] = &XEXP (XVECEXP (pat, 0, 0), 1));
  recog_data.dup_loc[0] = &XEXP (XVECEXP (pat, 0, 1), 0);
  recog_data.dup_num[0] = 1;
  recog_data.dup_loc[1] = &XEXP (XVECEXP (pat, 0, 1), 1);
  recog_data.dup_num[1] = 0;
  break;
```

위의 코드를 생성하는 모든 정보는 하나의 extraction 에 모두 있으므로 이해가 되지 않는 부분이 있다면, 해당 코드를 살펴보기 바란다.

제 11 절 genflags

Machine description 으로 부터 아래와 같은 내용을 생성: - 간단한 표준 명령어들이 이 machine 에서 이용 가능한지를 말해주는 몇몇 HAVE_... flag 들.

11.1 작동 방법

1. init_md_reader_args () 함수를 읽는다.
2. read_md_rtx () 함수를 통해서 해석한 RTX 를 하나씩 읽는다.
3. DEFINE_INSN 와 DEFINE_EXPAND 일 경우, gen_insn () 함수를 통해서 처리한다.
   (a) 만약 DEFINE_INSN 와 DEFINE_EXPAND 의 이름이 NULL 이거나 * 로 시작할 경우 이를 처리하지 않는다.
   (b) 세번째 인자, 즉 condition 이 정의되어 있지 않다면, 아래와 같이 정의한다. 만약 swapxf 를 처리한다고 할 때.

   ```
   #define HAVE\_swapxf 1
   ```

   (c) 만약 condition 이 정의되어 있다면 아래와 같은 모습이 될 것이다.

   ```
   #define HAVE_fix_truncdi_nomemory (TARGET_80387 \
      && FLOAT_MODE_P (GET_MODE (operands[1])) \
      && (!SSE_FLOAT_MODE_P (GET_MODE (operands[1])) || !TARGET_64BIT))
   ```

4. gen_proto () 함수를 이용하여 해당 함수들을 위한 prototype 들을 작성한다. DEFINE_INSN 혹은 DEFINE_EXPAND 의 이름이 call, call_pop, sibcall, sibcall_pop 일 경우 인자가 4 개를 받아들이도록 하는 매크로를 만들고, call_value, call_value_pop, sibcall_value, sibcall_value_pop 일 경우 인자가 5 개를 받아들이도록 한다. 예를 든다면 아래와 같은 prototype 이 생성된다.

```
#define GEN_CALL_POP(A, B, C, D) gen_call_pop ((A), (B), (C), (D))
extern struct rtx_def *gen_call_pop                          PARAMS
    ((struct rtx_def *,
      struct rtx_def *, struct rtx_def *, struct rtx_def *));
#define GEN_CALL(A, B, C, D) gen_call ((A), (B), (C))
extern struct rtx_def *gen_call                              PARAMS
    ((struct rtx_def *, struct rtx_def *, struct rtx_def *));
extern struct rtx_def *gen_call_exp                          PARAMS
    ((struct rtx_def *, struct rtx_def *));
#define GEN_CALL_VALUE_POP(A, B, C, D, E)       \
    gen_call_value_pop ((A), (B), (C), (D), (E))
extern struct rtx_def *gen_call_value_pop                    PARAMS
    ((struct rtx_def *, struct rtx_def *,
      struct rtx_def *, struct rtx_def *, struct rtx_def *));
#define GEN_CALL_VALUE(A, B, C, D, E) gen_call_value ((A), (B), (C), (D))
extern struct rtx_def *gen_call_value                        PARAMS
    ((struct rtx_def *, struct rtx_def *, struct rtx_def *, struct rtx_def *));
```

제 12 절 gengenrtl

RTL 구조체들을 할당하는 code 들을 생성.

12.1 작동 방법

gengenrtl 의 -h 옵션을 통해서 header 를 만들고 없으면 code 를 만든다. genheader () 함수를 통해서 header 를 처리하고 gencode () 함수를 통해서 code 를 만든다. struct rtx_definition 구조체를 통해서 이를 처리한다.

12.2 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
struct rtx_definition
{
  const char *const enumname, *const name, *const format;
};

#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS) { STRINGX(ENUM), NAME, FORMAT },

static const struct rtx_definition defs[] =
{
#include "rtl.def"              /* rtl expressions are documented here */
};
```

제 13 절 genopinit

Machine description 으로부터 optab 들을 초기화하는 code 를 생성.

13.1 작동 방법

genopinit.c 파일의 경우, DEFINE_INSN 와 DEFINE_EXPAND 만을 처리하는데, 이를 만날 경우, gen_insn () 함수를 호출한다. 이 함수에서의 처리는 아래와 같은 방법으로 진행된다.

- 우선 가장 중요하게 생각하는 부분은 아래의 섹션에 존재하는 char * 배열인 optabs 이다. 이것이 처리하는게 있어서 기준이 된다.
- 그중에서 () 가 한쌍을 이뤄서 처리가 이루어지게 된다. 배열 optabs 에서 먼저 봐야 할 부분이 () 로 둘러싸여져 있는 부분이다.
- 현재 처리하고 있는 insn 의 이름 (예를 들면, cmpdi_ccno_1_rex64 를 들 수 있겠다.) 과 $(와 $) 사이의 이름 첫글자를 비교한다. 만약 $(cmp$a$) 를 예로 든다면, cmpdi_ccno_1_rex64 에서의 cmp 와 $(cmp$a$) 에서의 cmp 가 같기 때문에 $a 를 처리하게 된다. 여기서 $a 등등에 관한 몇몇 기호들을 살펴볼 필요가 있는다. 그에 대한 의미는 아래와 같다. 여기서의 의미는 $(와 $) 사이의 것만 의미한다.

Character	설명
a 혹은 b	하나의 MACHINE_MODE 를 가르킨다. 만들어 질 때는 소문자로 변환된다.
N	지역 변수 force_consec 를 1 로 변경한다. 만약 1 로 설정된다면, 처음 $a 의 Machine Mode 는 반드시 두번째 $b 의 Wider Mode 여야 함을 뜻한다. 즉 GET_MODE_WIDER_MODE(m1) == m2 여야 함을 의미한다. (m1 는 $a 의 Machine Mode 이고, m2 는 $b 의 Machine Mode 이다.)
I	지역 변수 force_int 를 1 로 변경한다. 이는 MACHINE_MODE 선택에 영향을 준다. 즉 MODE 의 class 가 MODE_INT 혹은 MODE_VECTOR_INT 여야 함을 의미한다.
P	지역 변수 force_partial_int 를 1 로 변경한다. 이는 MACHINE_MODE 선택에 영향을 준다. 즉 MODE 의 class 가 MODE_INT 혹은 MODE_PARTIAL_INT, MODE_VECTOR_INT 여야 함을 의미한다.
F	지역 변수 force_float 를 1 로 변경한다. 이는 MACHINE_MODE 선택에 영향을 준다. 즉 MODE 의 class 가 MODE_FLOAT 혹은 MODE_VECTOR_FLOAT 여야 함을 의미한다.
V	아무 일도 하지 않는다.
c	대부분 comparison 에서 찾아 볼 수 있으며, RTX CLASS 가 '<' 에 속하는 것을 말한다.

- 이제 optabs 에 있는 것 중 형식에 맞는 것이 있다면, 이제 이를 출력하는데, $ 로 시작하는 것은 모두 뒤바꿔줘야 한다. 아래는 실제로 출력할 때 고려하는 $x 표이다.

(,)	무시된다.
I, F, N	무시된다.
V	$a 의 Mode Class 가 MODE_FLOAT 일 경우, 'v' 를 출력한다.
a 와 b	실제 Machine Mode 의 이름을 소문자로 짧게 출력한다. (예, di, si)
A 와 B	실제 Machine Mode 의 이름을 길게 출력한다. (예, HImode, QImode)
c	RTX 의 이름을 출력한다. (예, eq, leu 등등)
C	RTX 의 이름을 대문자로 출력한다. (예, LE, LEU, NE 등등)

그럼 cmpdi_ccno_1_rex64 는 처리가 어떻게 될까? cmpdi 까지는 어느 정도 처리되는 듯 하다가, 뒤에 _ccno_1_rex64 가 오기 때문에, 탈락된다.

13.2 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

- optabs

  GCC 의 많은 부분에서 machine mode 로 index 된 배열들을 사용하고, MD 파일 내에는 해당 mode 의 operand 들 상에서 주어진 operation 을 실행할 pattern 용 insn code 들을 포함한다. 그러한 pattern 들은 MD 파일에 이름을 가지며, 또한 사용될 mode 들과 operation 의 이름을 가진다. 이 프로그램은 MD 파일 존재하는 relevant pattern 들의 모든 insn code 들에 대한 optab 들을 초기화하는 function 'init_all_optabs' 를 작성한다. 이 배열은 초기화하는데 필요한 optab 들의 list 를 포함한다. 각 문자열 내에서 매치되는 pattern 의 이름은 $(와 $) 로 경계가 정해진다. 문자열에서 $a 와 $b 는 short mode name ('mode' 를 포함하는 mode name 의 부분으로 소문자로 변환되었다.) 를 match 하는데 사용된다. Initializer 를 작성할 때, 전체 문자열이 사용된다. $A 와 $B 는 mode 의 전체 이름으로 대체된다; $a 와 $b 는 위의 short form 의 이름 으로 대체된다. 만약 $N 이 패턴에서 존재한다면, 그것은 두 mode 들이 같은 mode class 에 consecutive width 들이여 함을 의미한다. (예를 들면 QImode 와 HImode). $I 는 오직 full integer mode 들만 다음 mode 로 고려되어야 함을 의미하고 $F 는 오직 float mode 들만 다음으로 간주되어야 함을 의미한다. $P 는 두 full 과 partial integer mode 들이 고려되어야 함을 의미한다. $V 는 만약 첫번째 mode 가 MODE_FLOAT mode 일 경우 'v' 를 emit 함을 의미한다. 몇몇 optab 들을 위해 우리는 RTL code 들내에 operation 을 저장한다. 이것은 comparison 들에서만 오직 사용된다. 그러한 경우, $c 와 $C 는 각각 comparison 의 소문자와 대분자 form 들이다.

13.3 전역 변수

```
optab optab_table[OTI_MAX];

enum insn_code extendtab[MAX_MACHINE_MODE][MAX_MACHINE_MODE][2];
enum insn_code fixtab[NUM_MACHINE_MODES][NUM_MACHINE_MODES][2];
enum insn_code fixtrunctab[NUM_MACHINE_MODES][NUM_MACHINE_MODES][2];
enum insn_code floattab[NUM_MACHINE_MODES][NUM_MACHINE_MODES][2];

/* Conditional 를 위한 rtx-code (예를 들면, EQ, LT, ....) 순으로
   나열되어 있으며 gen_function 로 하여금 해당 condition 를 검사하기
   위한 branch 를 만들도록 한다.  */

rtxfun bcc_gen_fctn[NUM_RTX_CODE];

/* Conditional 를 위한 rtx-code (예를 들면, EQ, LT, ....) 순으로
   나열되어 있으며 insn code 로 하여금 해당 condition 을 검사하기
   위한 store-condition insn 을 만들도록 한다.  */

enum insn_code setcc_gen_code[NUM_RTX_CODE];

#ifdef HAVE_conditional_move
/* machine mode 순으로 나열되어 있으며, insn code 는 conditional move
   insn 를 만들도록한다.  이것은 bcc_gen_fctn 와 setcc_gen_code 와
   같이 rtx-code 순으로 나열되어 있지 않는데, 명명된 pattern 들의
   번호순으로 되어 있다.  나중에 많은 rtx code 들이 conditional 이 될
   날을 대비해야 하겠다.  (예: ARM 과 같은)  */
enum insn_code movcc_gen_code[NUM_MACHINE_MODES];
#endif
```

13.4 genopinit 가 설정하는 구성 요소

```
static const char * const optabs[] =
```

```
{ "extendtab[$B][$A][0] = CODE_FOR_$(extend$a$b2$)",
  "extendtab[$B][$A][1] = CODE_FOR_$(zero_extend$a$b2$)",
  "fixtab[$A][$B][0] = CODE_FOR_$(fix$F$a$I$b2$)",
  "fixtab[$A][$B][1] = CODE_FOR_$(fixuns$F$a$b2$)",
  "fixtrunctab[$A][$B][0] = CODE_FOR_$(fix_trunc$F$a$I$b2$)",
  "fixtrunctab[$A][$B][1] = CODE_FOR_$(fixuns_trunc$F$a$I$b2$)",
  "floattab[$B][$A][0] = CODE_FOR_$(float$I$a$F$b2$)",
  "floattab[$B][$A][1] = CODE_FOR_$(floatuns$I$a$F$b2$)",
  "add_optab->handlers[$A].insn_code = CODE_FOR_$(add$P$a3$)",
  "addv_optab->handlers[(int) $A].insn_code =\n\
    add_optab->handlers[(int) $A].insn_code = CODE_FOR_$(add$F$a3$)",
  "addv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(addv$I$a3$)",
  "sub_optab->handlers[$A].insn_code = CODE_FOR_$(sub$P$a3$)",
  "subv_optab->handlers[(int) $A].insn_code =\n\
    sub_optab->handlers[(int) $A].insn_code = CODE_FOR_$(sub$F$a3$)",
  "subv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(subv$I$a3$)",
  "smul_optab->handlers[$A].insn_code = CODE_FOR_$(mul$P$a3$)",
  "smulv_optab->handlers[(int) $A].insn_code =\n\
    smul_optab->handlers[(int) $A].insn_code = CODE_FOR_$(mul$F$a3$)",
  "smulv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(mulv$I$a3$)",
  "umul_highpart_optab->handlers[$A].insn_code = CODE_FOR_$(umul$a3_highpart$)",
  "smul_highpart_optab->handlers[$A].insn_code = CODE_FOR_$(smul$a3_highpart$)",
  "smul_widen_optab->handlers[$B].insn_code = CODE_FOR_$(mul$a$b3$)$N",
  "umul_widen_optab->handlers[$B].insn_code = CODE_FOR_$(umul$a$b3$)$N",
  "sdiv_optab->handlers[$A].insn_code = CODE_FOR_$(div$a3$)",
  "sdivv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(div$V$I$a3$)",
  "udiv_optab->handlers[$A].insn_code = CODE_FOR_$(udiv$I$a3$)",
  "sdivmod_optab->handlers[$A].insn_code = CODE_FOR_$(divmod$a4$)",
  "udivmod_optab->handlers[$A].insn_code = CODE_FOR_$(udivmod$a4$)",
  "smod_optab->handlers[$A].insn_code = CODE_FOR_$(mod$a3$)",
  "umod_optab->handlers[$A].insn_code = CODE_FOR_$(umod$a3$)",
  "ftrunc_optab->handlers[$A].insn_code = CODE_FOR_$(ftrunc$F$a2$)",
  "and_optab->handlers[$A].insn_code = CODE_FOR_$(and$a3$)",
  "ior_optab->handlers[$A].insn_code = CODE_FOR_$(ior$a3$)",
  "xor_optab->handlers[$A].insn_code = CODE_FOR_$(xor$a3$)",
  "ashl_optab->handlers[$A].insn_code = CODE_FOR_$(ashl$a3$)",
  "ashr_optab->handlers[$A].insn_code = CODE_FOR_$(ashr$a3$)",
  "lshr_optab->handlers[$A].insn_code = CODE_FOR_$(lshr$a3$)",
  "rotl_optab->handlers[$A].insn_code = CODE_FOR_$(rotl$a3$)",
  "rotr_optab->handlers[$A].insn_code = CODE_FOR_$(rotr$a3$)",
  "smin_optab->handlers[$A].insn_code = CODE_FOR_$(smin$I$a3$)",
  "smin_optab->handlers[$A].insn_code = CODE_FOR_$(min$F$a3$)",
  "smax_optab->handlers[$A].insn_code = CODE_FOR_$(smax$I$a3$)",
  "smax_optab->handlers[$A].insn_code = CODE_FOR_$(max$F$a3$)",
  "umin_optab->handlers[$A].insn_code = CODE_FOR_$(umin$I$a3$)",
  "umax_optab->handlers[$A].insn_code = CODE_FOR_$(umax$I$a3$)",
  "neg_optab->handlers[$A].insn_code = CODE_FOR_$(neg$P$a2$)",
  "negv_optab->handlers[(int) $A].insn_code =\n\
    neg_optab->handlers[(int) $A].insn_code = CODE_FOR_$(neg$F$a2$)",
  "negv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(negv$I$a2$)",
  "abs_optab->handlers[$A].insn_code = CODE_FOR_$(abs$P$a2$)",
  "abs_optab->handlers[$A].insn_code = CODE_FOR_$(abs$P$a2$)",
```

```
"absv_optab->handlers[(int) $A].insn_code =\n\
  abs_optab->handlers[(int) $A].insn_code = CODE_FOR_$(abs$F$a2$)",
"absv_optab->handlers[(int) $A].insn_code = CODE_FOR_$(absv$I$a2$)",
"sqrt_optab->handlers[$A].insn_code = CODE_FOR_$(sqrt$a2$)",
"sin_optab->handlers[$A].insn_code = CODE_FOR_$(sin$a2$)",
"cos_optab->handlers[$A].insn_code = CODE_FOR_$(cos$a2$)",
"strlen_optab->handlers[$A].insn_code = CODE_FOR_$(strlen$a$)",
"one_cmpl_optab->handlers[$A].insn_code = CODE_FOR_$(one_cmpl$a2$)",
"ffs_optab->handlers[$A].insn_code = CODE_FOR_$(ffs$a2$)",
"mov_optab->handlers[$A].insn_code = CODE_FOR_$(mov$a$)",
"movstrict_optab->handlers[$A].insn_code = CODE_FOR_$(movstrict$a$)",
"cmp_optab->handlers[$A].insn_code = CODE_FOR_$(cmp$a$)",
"tst_optab->handlers[$A].insn_code = CODE_FOR_$(tst$a$)",
"bcc_gen_fctn[$C] = gen_$(b$c$)",
"setcc_gen_code[$C] = CODE_FOR_$(s$c$)",
"movcc_gen_code[$A] = CODE_FOR_$(mov$acc$)",
"cbranch_optab->handlers[$A].insn_code = CODE_FOR_$(cbranch$a4$)",
"cmov_optab->handlers[$A].insn_code = CODE_FOR_$(cmov$a6$)",
"cstore_optab->handlers[$A].insn_code = CODE_FOR_$(cstore$a4$)",
"push_optab->handlers[$A].insn_code = CODE_FOR_$(push$a1$)",
"reload_in_optab[$A] = CODE_FOR_$(reload_in$a$)",
"reload_out_optab[$A] = CODE_FOR_$(reload_out$a$)",
"movstr_optab[$A] = CODE_FOR_$(movstr$a$)",
"clrstr_optab[$A] = CODE_FOR_$(clrstr$a$)" };
```

제 14 절 genoutput

RTL 로부터 인식된 assembler insn 들을 output 할 code 를 생성한다.

14.1 목적

아래의 결과 즉, output_* 함수 혹은 변수와, predicate 들에 대한 extern 선언 prototype, 'struct insn_data' 배열, 'struct insn_operand data' 배열들을 만드는 것이 목적이다.

14.2 작동 방법

DEFINE_INSN, DEFINE_PEEPHOLE, DEFINE_EXPAND, DEFINE_SPLIT, DEFINE_PEEPHOLE2 를 처리한다.

14.2.1 DEFINE_INSN

- 새로운 struct data * d 를 할당하고, d→code_number, id→ndex_number, d→lineno, d→name 를 설정하고, d→next 를 0 으로 설정하여 전역 변수 idata_end 에 기록한다. d→operand 또한 0 으로 초기화한다.
- scan_operands () 함수를 이용하여, 다음과 같은 rtx 들을 처리한다.
  - MATCH_OPERAND
  - MATCH_SCRATCH
  - MATCH_OPERATOR
  - MATCH_PARALLEL

    위의 rtx 들의 경우, 목적은 전역 변수 max_opno, num_dups 와 다음과 같은 operand node 들을 채우는 것이 목적이다.

    ```
    d->operand[opno].seen
    d->operand[opno].mode
    d->operand[opno].strict_low
    d->operand[opno].predicate
    d->operand[opno].constraint
    d->operand[opno].n_alternatives
    d->operand[opno].address_p
    d->operand[opno].eliminable
    ```
  - MATCH_DUP
  - MATCH_OP_DUP
  - MATCH_PAR_DUP

    위의 rtx 들의 경우, 단순히 num_dups 만 만날때 마다 1 증가 시킨다.
- 현재 INSN 에 대한 검증을 수행한 후, 변수 d 를 채웠을 경우, 이를 전역 변수 odata 에 기록하고, odata_end 를 연결시킨다. place_operands () 함수를 통해 이루어진다.
- process_template () 함수를 통해 template 들을 처리한다. * 로 시작할 경우, 바로 실행할 code 가 포함되어 있을 경우이고, @ 로 시작할 경우, 다수의 template 가 들어있을 경우에 해당한다. * 로 시작할 경우, ouput_* 에 대한 함수를 생성하게 되며, @ 로 시작할 경우, output_@ 에 대한 변수를 생성하게 된다. * 는 output_format 이 INSN_OUTPUT_FORMAT_FUNCTION 로, @ 는 output_format 이 INSN_OUTPUT_FORMAT_MULTI 로, 그 외의 모든 것은 INSN_OUTPUT_FORMAT_SINGLE 로 설정된다.

14.2.2 DEFINE_EXPAND

DEFINE_INSN 를 처리하는 것과 거의 흡사하나, 이 rtx 의 경우, template 가 존재하지 않는다. 그래서 d→output_format 이 INSN_OUTPUT_FORMAT_NONE 로 설정된다.

14.2.3 DEFINE_PEEPHOLE

DEFINE_PEEPHOLE 의 경우, DEFINE_INSN 처리와 거의 같지만 alternative 들이 존재하지 않기 때문에 이 부분에 대해서는 처리하지 않는다고 보면 되며, 나머지 부분은 같다.

14.2.4 DEFINE_SPLIT 와 DEFINE_PEEPHOLE2

DEFINE_INSN 와 유사하지만, template 가 존재하지 않고, alternative 들도 존재하지 않는다.
d→output_format 는 INSN_OUTPUT_FORMAT_NONE 로 설정된다.

14.3 출력

이 프로그램은 compiler target machine 을 위한 machine description 를 읽고 다음을 포함하는 파일을 생성한다.

1. DEFINE_INSN 와 DEFINE_PEEPHOLE 을 위한 template 들을 생성한다. output_코드번호 형식의 전역변수 혹은 함수로 선언이 되며, Template 가 * 로 시작할 경우는 함수, @ 로 시작할 경우는 변수로 선언된다.

2. Machine Description 에서 사용되는 predicate 들에 대한 extern 선언 prototype 을 생성한다.

3. Index code number 로 정렬된 'struct insn_data' 배열. 다음의 내용을 포함한다.

   - 'name' 는 해당 pattern 의 이름. 이름이 없을 경우 이름이 주어진다.
   - 'output' 는 output template 혹은 template 들의 output 배열, output function 을 잡고 있다.
   - 'genfun' 는 argument 로 주어진 operand 로 해당 pattern 의 body 를 생성하는 function 이다.
   - 'n_operands' 는 해당 insn 를 위한 pattern 내 distinct operand 들의 숫자.
   - 'n_dups' 는 insn 의 pattern 에서 나타나는 match_dup 의 갯수. 이것은 insn 가 인식된 후 얼마나 많은 'recog_data.dup_loc' 를 암시하는지 말해준다.
   - 'n_alternatives' 는 각 pattern 의 constraint 에서 alternative 들의 숫자이다.
   - 'output_format' 는 'output' 이 무슨 type 의 것인지 말한다.
   - 'operand' 는 해당 insn 를 위한 operand data 배열의 base 이다.

   insn_data 를 만들 때, 고려해야 할 사항이 존재하는데, 이름이 존재하지 않을 경우이다.
   DEFINE_INSN 혹은 DEFINE_EXPAND, DEFINE_PEEPHOLE, DEFINE_SPLIT,
   DEFINE_PEEPHOLE2 중 해당 rtx 에 대한 이름이 존재하지 않을 경우 우리는 offset 을 통해서 이름을 만드는데, 이름이 없는 insn 를 처리하기 전에 처리했던 last_name 과, 현재 이름이 없는 것 다음으로 이름이 존재하는 next_name 을 기반으로 한다.

4. 위에서 'operand' 에 의해 사용되는 'struct insn_operand data' 배열

   - 'predicate', int-valued function, 는 이 operand 를 위한 match_operand predicate 이다.
   - 'constraint' 는 이 operand 를 위한 constraint 이다. 이것은 오직 match_operand 들내 register constraint 들이 나타날 경우에만 존재한다.
   - 'address_p' 는 ADDRESS rtx 들내에 operand 들이 보인다는 것을 의미한다.
     이것은 match_operand rtx 들내에 register constraint 들이 없을 경우 존재한다.
   - 'mode' 는 operand 가 가졌으면 하는 machine mode 이다.

- 'strict_low' 는 STRICT_LOW_PART 에 포함되는 operand 들에 대해 0 이 아닌 값이다.
- 'eliminable' 는 MATCH_OPERAND 에 의해 일반적으로 match 되는 operand 들에 대해 0 이 아닌 값이다; MATCH_OPERATOR 들 같이 register elimination 동안에 변경되어지는 안되는 operand 들은 값이 0 이다.

Insn 의 code number 는 단순히 machine description 에서의 위치이다; code number 들은 각 description entry 들에 대해 순차적으로 할당되며, code number 는 0 으로 시작한다. 그래서, machine description 내 다음 entry 는

```
(define_insn "clrdf"
  [(set (match_operand:DF 0 "general_operand" "")
        (const_int 0))]
  ""
  "clrd %0")
```

25 번째 entry 로써 존재하고, insn_data[24].template 는 "clrd %0" 이고, insn_data[24].n_operands 는 1 로 설정될 것이다.

14.4 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

```
/* 모든 instruction 은 이것보다 더 많은 operand 들을 가질 수 없다. 이런
   임의의 제한에 대해서 죄송하다는 생각을 하지만, 이렇게 많은 operand 들을
   가지는 명령어가 어떤 machine 에 존재할것인가? */

#define MAX_MAX_OPERANDS 40

/* 이 chain 에 우리가 output 할 operand 들에 관한 모든 정보를 기록한다. */

struct operand_data
{
  struct operand_data *next;
  int index;
  const char *predicate;
  const char *constraint;
  enum machine_mode mode;
  unsigned char n_alternatives;
  char address_p;
  char strict_low;
  char eliminable;
  char seen;
};

/* Index 0 에서 null operand 를 가진 것으로 시작한다. */

static struct operand_data null_operand =
{
  0, 0, "", "", VOIDmode, 0, 0, 0, 0, 0
};

static struct operand_data *odata = &null_operand;
static struct operand_data **odata_end = &null_operand.next;
```

```
/* recog.h 내 constant 들과 반드시 일치해야 한다.  */

#define INSN_OUTPUT_FORMAT_NONE         0       /* abort */
#define INSN_OUTPUT_FORMAT_SINGLE       1       /* const char * */
#define INSN_OUTPUT_FORMAT_MULTI        2       /* const char * const * */
#define INSN_OUTPUT_FORMAT_FUNCTION     3       /* const char * (*)(...) */

/* 이 chain 에 우리가 output 할 모든 정보를 기록하는데, insn 의 code number
   와 결합되어 있다.  */

struct data
{
  struct data *next;
  const char *name;
  const char *template;
  int code_number;
  int index_number;
  int lineno;
  int n_operands;               /* 이 insn 가 인식하는 operand 들의 갯수 */
  int n_dups;                   /* Pattern 에서 match_dup 가 보인 횟수 */
  int n_alternatives;           /* 각 constraint 에서 alternative 들의 갯수 */
  int operand_number;           /* Big array 에서 operand index.  */
  int output_format;            /* INSN_OUTPUT_FORMAT_*.  */
  struct operand_data operand[MAX_MAX_OPERANDS];
};

/* 이 변수는 insn chain 의 첫번째 link 를 가르킨다.  */

static struct data *idata, **idata_end = &idata;
```

제 15 절 genpeep

Machine description 으로 부터 peephole 최적화를 수행할 code 를 생성한다.

15.1 작동 방법

DEFINE_PEEPHOLE 이 MD 내에 있을 경우에 처리하는데, 현재 i386.md 에는 DEFINE_PEEPHOLE 의 경우 사용하지 않고, DEFINE_PEEPHOLE2 에 대해서만 사용하기 때문에 이 결과로 생성되는 것은 존재하지 않는다.

제 16 절 genpreds

Machine description 으로 부터 아래와 같은 내용을 생성: - 각각의 정의된 표준 insn name 들을 위한 insn_code_number 값을 지정할 몇몇 CODE_FOR_... 를 만듬.

16.1 작동 방법

Target machine (여기에서는 i386.h) 에 정의되어 있는 PREDICATE_CODES 매크로를 기반으로 해서 tm-preds.h 를 생성하는데, PREDICATE_CODES 매크로는 다음과 같이 정의되어 있다.

```
static const struct {
    const char *const name;
    const RTX_CODE codes[NUM_RTX_CODE];
} predicate[] = {
    PREDICATE_CODES
};
/* i386.c 내 predicate 들에 매칭되는 code 들을 정의한다.  */

#define PREDICATE_CODES                                                 \
  {"x86_64_immediate_operand", {CONST_INT, SUBREG, REG,                 \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_nonmemory_operand", {CONST_INT, SUBREG, REG,                 \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_movabs_operand", {CONST_INT, SUBREG, REG,                    \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_szext_nonmemory_operand", {CONST_INT, SUBREG, REG,           \
                                   SYMBOL_REF, LABEL_REF, CONST}},      \
  {"x86_64_general_operand", {CONST_INT, SUBREG, REG, MEM,              \
                              SYMBOL_REF, LABEL_REF, CONST}},           \
  {"x86_64_szext_general_operand", {CONST_INT, SUBREG, REG, MEM,        \
                                  SYMBOL_REF, LABEL_REF, CONST}},       \
  {"x86_64_zext_immediate_operand", {CONST_INT, CONST_DOUBLE, CONST,    \
                                     SYMBOL_REF, LABEL_REF}},           \
  {"shiftdi_operand", {SUBREG, REG, MEM}},                              \
  {"const_int_1_operand", {CONST_INT}},                                 \
  {"symbolic_operand", {SYMBOL_REF, LABEL_REF, CONST}},                 \
  {"aligned_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,      \
                       LABEL_REF, SUBREG, REG, MEM}},                   \
  {"pic_symbolic_operand", {CONST}},                                    \
  {"call_insn_operand", {REG, SUBREG, MEM, SYMBOL_REF}},                \
  {"constant_call_address_operand", {SYMBOL_REF, CONST}},               \
  {"const0_operand", {CONST_INT, CONST_DOUBLE}},                        \
  {"const1_operand", {CONST_INT}},                                      \
  {"const248_operand", {CONST_INT}},                                    \
  {"incdec_operand", {CONST_INT}},                                      \
  {"mmx_reg_operand", {REG}},                                           \
  {"reg_no_sp_operand", {SUBREG, REG}},                                 \
  {"general_no_elim_operand", {CONST_INT, CONST_DOUBLE, CONST,          \
                         SYMBOL_REF, LABEL_REF, SUBREG, REG, MEM}},     \
  {"nonmemory_no_elim_operand", {CONST_INT, REG, SUBREG}},              \
  {"q_regs_operand", {SUBREG, REG}},                                    \
  {"non_q_regs_operand", {SUBREG, REG}},                                \
  {"fcmov_comparison_operator", {EQ, NE, LTU, GTU, LEU, GEU, UNORDERED, \
```

```
                                ORDERED, LT, UNLT, GT, UNGT, LE, UNLE, \
                                GE, UNGE, LTGT, UNEQ}},                 \
  {"sse_comparison_operator", {EQ, LT, LE, UNORDERED, NE, UNGE, UNGT,   \
                               ORDERED, UNEQ, UNLT, UNLE, LTGT, GE, GT  \
                               }},                                      \
  {"ix86_comparison_operator", {EQ, NE, LE, LT, GE, GT, LEU, LTU, GEU,  \
                               GTU, UNORDERED, ORDERED, UNLE, UNLT,     \
                               UNGE, UNGT, LTGT, UNEQ }},               \
  {"cmp_fp_expander_operand", {CONST_DOUBLE, SUBREG, REG, MEM}},        \
  {"ext_register_operand", {SUBREG, REG}},                              \
  {"binary_fp_operator", {PLUS, MINUS, MULT, DIV}},                     \
  {"mult_operator", {MULT}},                                            \
  {"div_operator", {DIV}},                                              \
  {"arith_or_logical_operator", {PLUS, MULT, AND, IOR, XOR, SMIN, SMAX, \
                                 UMIN, UMAX, COMPARE, MINUS, DIV, MOD,  \
                                 UDIV, UMOD, ASHIFT, ROTATE, ASHIFTRT,  \
                                 LSHIFTRT, ROTATERT}},                  \
  {"promotable_binary_operator", {PLUS, MULT, AND, IOR, XOR, ASHIFT}},  \
  {"memory_displacement_operand", {MEM}},                               \
  {"cmpsi_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,        \
                     LABEL_REF, SUBREG, REG, MEM, AND}},                \
  {"long_memory_operand", {MEM}},
}}}
```

16.2 $prefix/gcc/tm-preds.h

```
/* Generated automatically by the program 'genpreds'.  */

#ifndef GCC_TM_PREDS_H
#define GCC_TM_PREDS_H

#ifdef RTX_CODE

extern int x86_64_immediate_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_nonmemory_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_movabs_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_szext_nonmemory_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_general_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_szext_general_operand PARAMS ((rtx, enum machine_mode));
extern int x86_64_zext_immediate_operand PARAMS ((rtx, enum machine_mode));
extern int shiftdi_operand PARAMS ((rtx, enum machine_mode));
extern int const_int_1_operand PARAMS ((rtx, enum machine_mode));
extern int symbolic_operand PARAMS ((rtx, enum machine_mode));
extern int aligned_operand PARAMS ((rtx, enum machine_mode));
extern int pic_symbolic_operand PARAMS ((rtx, enum machine_mode));
extern int call_insn_operand PARAMS ((rtx, enum machine_mode));
extern int constant_call_address_operand PARAMS ((rtx, enum machine_mode));
extern int const0_operand PARAMS ((rtx, enum machine_mode));
extern int const1_operand PARAMS ((rtx, enum machine_mode));
extern int const248_operand PARAMS ((rtx, enum machine_mode));
extern int incdec_operand PARAMS ((rtx, enum machine_mode));
```

```
extern int mmx_reg_operand PARAMS ((rtx, enum machine_mode));
extern int reg_no_sp_operand PARAMS ((rtx, enum machine_mode));
extern int general_no_elim_operand PARAMS ((rtx, enum machine_mode));
extern int nonmemory_no_elim_operand PARAMS ((rtx, enum machine_mode));
extern int q_regs_operand PARAMS ((rtx, enum machine_mode));
extern int non_q_regs_operand PARAMS ((rtx, enum machine_mode));
extern int fcmov_comparison_operator PARAMS ((rtx, enum machine_mode));
extern int sse_comparison_operator PARAMS ((rtx, enum machine_mode));
extern int ix86_comparison_operator PARAMS ((rtx, enum machine_mode));
extern int cmp_fp_expander_operand PARAMS ((rtx, enum machine_mode));
extern int ext_register_operand PARAMS ((rtx, enum machine_mode));
extern int binary_fp_operator PARAMS ((rtx, enum machine_mode));
extern int mult_operator PARAMS ((rtx, enum machine_mode));
extern int div_operator PARAMS ((rtx, enum machine_mode));
extern int arith_or_logical_operator PARAMS ((rtx, enum machine_mode));
extern int promotable_binary_operator PARAMS ((rtx, enum machine_mode));
extern int memory_displacement_operand PARAMS ((rtx, enum machine_mode));
extern int cmpsi_operand PARAMS ((rtx, enum machine_mode));
extern int long_memory_operand PARAMS ((rtx, enum machine_mode));

#endif /* RTX_CODE */

#endif /* GCC_TM_PREDS_H */
```

제 17 절 genrecog

Machine description 으로부터 insn 들을 rtl 로 인식하는 code 를 생성.

17.1 작동 방법

DEFINE_INSN 와 DEFINE_SPLIT, DEFINE_PEEPHOLE2 를 처리한다.

17.1.1 DT_*

- MATCH_PARALLEL 일 경우, DT_veclen_ge 관련 struct decision_test 가 생성된다.
- MATCH_OPERAND
- MATCH_SCRATCH
- MATCH_OPERATOR
- MATCH_INSN

  위의 네 rtx 의 predicate 가 존재할 경우, DT_pred 관련 struct decision_test 가 생성된다. Operand 관련 rtx 이기 때문에 DT_accept_op 또한 뒤에 붙인다.
- MATCH_OP_DUP 일 경우, DT_dup 를 만든 후, DT_accept_op 를 붙인다.
- MATCH_DUP 일 경우, DT_dup 를 붙인다.
- 이 외의 rtx 들 중에서, 해당 rtx 인자들의 class 가 'i' 이고 값이 0 일 경우, DT_elt_zero_int 를 붙인다. 1 일 경우 DT_elt_one_int 를 붙인다.
- 이 외의 rtx 들 중에서, 해당 rtx 인자들의 class 가 'w' 이고 값이 0 일 경우, DT_elt_zero_wide_safe 를 붙인다. 1 일 경우 DT_elt_zero_wide 를 붙인다.
- 이 외의 rtx 들 중에서, 해당 rtx 인자들의 class 가 'E' 일 경우, DT_veclen 를 붙인다.
- 이제 현재 rtx 에 대한 하위 rtx 들을 모두 처리했고, CODE 가 UNKNOWN 이 아닐 경우, DT_code 를 붙인다. 또한 MODE 가 VOIDmode 가 아닐 경우, DT_mode 를 붙인다.
- 만약 해당 rtx 에 대한 부과적인 pattern 인식을 위한 조건인 C 표현식이 있을 경우, DT_c_test 를 붙인다.
- 해당 rtx 에 대한 모든 처리가 끝날 경우, 마지막 struct decision_test 끝에 DT_accept_insn 를 붙인다.

17.1.2 maybe_both_true () 함수의 기능

아래의 D1 은 이 함수의 첫번째 인자이고, D2 는 두번째 인자이다. 모두 struct decision_test 이다.

- D1 의 DT_* 가, D2 의 것과 같을 경우

  DT_mode, DT_code, DT_veclen, DT_elt_zero_int, DT_elt_one_int, DT_elt_zero_wide, DT_elt_zero_wide_safe 를 처리한다.
- D1 이 DT_pred 이고, D2 가 DT_mode 일 경우

  D1 의 predicate mode 와 D2 의 mode 가 같지 않고, D1 의 predicate 가 address_operand 가 아닐 경우 true 가 아님으로 0 을 반환한다.
- D1 이 DT_pred 이고, D2 가 DT_code 일 경우

  D1 의 predicate 가 가질 수 있는 코드가, D2 의 것과 같을 경우 0 이 아닌 값을 반환한다.

- D1 이 DT_pred 이고, D2 가 DT_pred 일 경우

  D1 의 predicate 가 가질 수 있는 코드와, D2 의 것과 공통되는 것이 있을 경우 0 이 아닌 값을 반환한다.
- D1 이 DT_veclen 이고, D2 가 DT_veclen_ge 일 경우

  D1 의 veclen 가 D2 보다 크거나 같으면 0 이 아닌 값을 반환한다.
- D1 이 DT_veclen_ge 이고, D2 가 DT_veclen 일 경우

  D2 의 veclen 이 D1 보다 크거나 같으면 0 이 아닌 값을 반환한다.

- **(예제 1)** 이 함수가 true 일 때는 다음과 같은 상황일 경우,

```
{pred=(const0_operand,DI) + A_op=1} 4 n -1 a -1
  {c_test="TARGET_64BIT && ..." + A_insn=(0,0)} 5 n -1 a -1
```

과

```
{pred=(x86_64_general_operand,DI) + A_op=1} 17 n -1 a -1
  {c_test="TARGET_64BIT && ..." + A_insn=(2,0)} 18 n -1 a -1
```

Merging 이 이루어진 후의 모습은 다음과 같다.

```
{code=set} 0 n -1 a -1
  {code=reg + elt0_i=17} 1 n -1 a -1
    {code=compare} 2 n -1 a -1
      {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 3 n 9 a -1
        {pred=(const0_operand,DI) + A_op=1} 4 n 17 a -1
          {c_test="TARGET_64BIT && ..." + A_insn=(0,0)} 5 n -1 a -1
        {pred=(x86_64_general_operand,DI) + A_op=1} 17 n -1 a -1
          {c_test="TARGET_64BIT && ..." + A_insn=(2,0)} 18 n -1 a -1
      {mode=DI + code=minus} 9 n -1 a -1
        {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 10 n -1 a -1
          {pred=(x86_64_general_operand,DI) + A_op=1} 11 n -1 a -1
            {code=const_int + elt0_ws=0
             + c_test="TARGET_64BIT && ..." + A_insn= (1,0)} 12 n -1 a -1
```

- **(예제 2)**

```
{pred=(reg_or_0_operand,SF) + A_op=0} 3 n -1 a -1
  {pred=(reg_or_0_operand,SF) + A_op=1} 4 n -1 a -1
    {c_test="! TARGET_SOFT_FL..." + A_insn=(0,0)} 5 n -1 a -1
```

과

```
{pred=(reg_or_0_operand,DF) + A_op=0} 9 n -1 a -1
  {pred=(reg_or_0_operand,DF) + A_op=1} 10 n -1 a -1
    {c_test="! TARGET_SOFT_FL..." + A_insn=(1,0)} 11 n -1 a -1
```

만약 maybe_both_true_1 과 maybe_both_true_2 가 -1 을 반환한다면, true 일 수 있다 라는 가정을 하기 때문에 이에 대한 정확한 판단을 위해서는 true 일 수 있다 라고 가정되는 struct decision_test 의 하위 목록 (해당 decision_test 를 D1 이라고 했을 때, D1→success.first 로 연결된 linked list 들) 을 검사함으로써 판단 여부를 가리며, 이 때는 오직 maybe_both_true_1 과 maybe_both_true_2 가 1 을 반환할 때만 true 로 여겨진다.

17.2 Merging 이 이루어지기 전

- make_insn_sequence () 함수를 통해서 처리가 이루어지는데, 최종적인 목표는 현재 insn 에 대한 한 struct decision_head 구조체를 완성하는 것이 목표이다. DEFINE_INSN 일 경우, make_insn_sequence () 함수의 두번째 인자는 RECOG 가 되며, DEFINE_SPLIT 일 경우 SPLIT, DEFINE_PEEPHOLE2 일 경우 PEEPHOLE2 가 된다.
- 위의 세 rtx (DEFINE_INSN 와 DEFINE_SPLIT, DEFINE_PEEPHOLE2) 들은 모두 pattern 들을 가지고 있는데, 우선 validate_pattern () 함수를 통해서 해당 pattern 에 대한 검증을 수행한다.
- 이제 add_to_sequence () 함수를 통해서, 해당 pattern 의 각 rtx 들을 방문하며 재귀적으로 struct decision_head 를 채우게 되는데, 최종적으로 이 함수가 반환하는 것은 마지막으로 만들어진 struct decision 구조체이며, 두번째 인자로 주는 struct decision_head 에 차례대로 읽은 것들이 들어가게 된다.
  - 이 함수는 각각의 방문하는 rtx 마다, 그에 대한 struct decision 을 할당하며, rtx 의 각 인자들이 struct decision_test 구조체를 만들 필요성이 있을 경우, 이를 할당하고 내부에 적당한 값을 집어 넣는다.
  - 그리고 하위에 있는 rtx 들에 대해 재귀적으로 반복 수행한다.
- 만약 PATTERN 이 인식되기 위해 부과적으로 만족해야 하는 C 표현식이 있을 경우, 위에서 반환된 마지막으로 만들어진 struct decision 구조체의 연결 리스트 끝에 이 C 표현식에 대한 struct decision_test 구조체를 할당하고 설정한다.
- 마지막으로 만들어진 struct decision 구조체의 연결 리스트 끝에 DT_accept_insn 관련 struct decision_test 구조체를 넣는다.

17.3 Merging 이 이루어지는 동안

- merge_trees () 함수를 통해서 merging 이 두 decision 에 대해 이루어지는데, 예로 설명을 해보자. 아래과 같이 두 decision 에 대해서 merging 과정을 설명하면 좀 더 쉽게 이해를 할 수 있을 듯 하다. 앞의 A1, A2, A3, ... 혹은 B1, B2, B3, ... 는 모두 줄 번호이다.

```
A1: {code=set} 0 n -1 a -1
A2:   {code=reg + elt0_i=17} 1 n -1 a -1
A3:     {code=compare} 2 n -1 a -1
A4:       {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 3 n -1 a -1
A5:         {pred=(const0_operand,DI) + A_op=1} 4 n -1 a -1
A6:           {c_test="TARGET_64BIT && ..." + A_insn=(0,0)} 5 n -1 a -1

B1: {code=set} 6 n -1 a -1
B2:   {code=reg + elt0_i=17} 7 n -1 a -1
B3:     {code=compare} 8 n -1 a -1
B4:       {mode=DI + code=minus} 9 n -1 a -1
B5:         {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 10 n -1 a -1
B6:           {pred=(x86_64_general_operand,DI) + A_op=1} 11 n -1 a -1
B7:             {code=const_int + elt0_ws=0
                 + c_test="TARGET_64BIT && ..." + A_insn=(1,0)} 12 n -1 a -1
```

- 결과

```
C1: {code=set} 0 n -1 a -1
C2:   {code=reg + elt0_i=17} 1 n -1 a -1
C3:     {code=compare} 2 n -1 a -1
```

```
C4:       {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 3 n 9 a -1
C5:         {pred=(const0_operand,DI) + A_op=1} 4 n -1 a -1
C6:           {c_test="TARGET_64BIT && ..." + A_insn=(0,0)} 5 n -1 a -1
C7:       {mode=DI + code=minus} 9 n -1 a -1
C8:         {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 10 n -1 a -1
C9:           {pred=(x86_64_general_operand,DI) + A_op=1} 11 n -1 a -1
C0:             {code=const_int + elt0_ws=0
                 + c_test="TARGET_64BIT && ..." + A_insn=(1,0)} 12 n -1 a -1
```

17.4 출력

17.4.1 factor_tests () 함수

- factor_tests () 함수가 호출되었을 때는, 모든 insn 가 처리가 된 상태에서 호출되며, 1 차적인 merging 이 이루어져 전역변수 recog_tree 에 DEFINE_INSN 에 관한 모든 decision 들이 들어가 있을 것이다.
- 이 함수에서는 2 차 merging 을 수행한다는데, 수행하는 방식은 다음과 같다.
  1. recog_tree 에 있는 first 부터 last 까지 for 구문을 수행하면서 처리를 하는데, 첫번째 decision 과 두번째 decision 을 recog_tree 에서 분리한다.
  2. 분리된 두 개의 decision 에서 각각 toplevel 에 decision_test 구문이 두 개 이상 있을 경우, 해당하는 test 들을 잘라내서 새로운 decision 을 만든 후, 하위 decision 목록에 추가한다. 예를 들어 설명하면

     ```
     {code=parallel + veclen=2} 269 n 276 a -1
       ....
     ```

     와 같은 decision 이 있다고 가정했을 때, (하위 decision 목록은 생략하였다.) 다음과 같은 모양으로 변경시킨다.

     ```
     {code=parallel} 269 n -1 a -1
       {veclen=2} 10828 n -1 a -1
         ....
     ```
  3. 그런 후, 이 두 decision 을 merging_trees () 함수를 이용하여 merging 을 시도한다.
  4. 이제 현재 level 에서의 모든 decision 에 대한 merging 이 이루어졌다면, 하위에 존재하는 decision 들에 대해서도 merging 을 재귀적으로 한다.

17.4.2 break_out_subroutines () 함수

- 재귀적인 호출을 통해서, 현재 HEAD (이 함수의 첫번째 인자) 의 하위 node 들의 갯수를 구하는데, 하위 node 들이 대단이 많을 때, 실제 생성되는 insn-recog.c 파일에서 분리된 하위 루틴으로 구성되도록 만든다. 분리의 기준이 되는 것이 하위 node 들의 갯수이며, SUBROUTINE_THRESHOLD 값에 따라 정해지는데 이것은 $pregix/gcc/genrecog.c 파일에 100 으로 설정되어 있다.

17.4.3 find_afterward () 함수

- 이 함수에서 decision 의 구성요소 afterward 와 need_label 을 설정하게 되는데, 설정되는 기준은 다음과 같다.
  - afterward 같은 level 에 존재하는 decision 들을 maybe_both_true () 함수로 테스트를 한 후, 이를 만족하는 것이 있을 경우 afterward 가 설정된다. (물론 p→subroutine_number 가 존재할 경우, 이를 만족하는 것이 없더라도, find_afterward () 함수의 두번째 인자로 들어온 real_afterward 값이 들어가게 된다.) 아래는 find_afterward () 함수를 실행하기 전의 어떤 같은 decision 레벨의 모습이다. 소문자 a 뒤를 잘보기 바란다.

```
{mode=HI} 138 n 145 a -1
{mode=CCFP} 145 n 197 a -1
{mode=CCFPU} 197 n 221 a -1
{mode=CC} 221 n 262 a -1
{mode=SI} 262 n 394 a -1
{mode=QI} 394 n 508 a -1
{mode=DI} 508 n 591 a -1
{mode=SF} 591 n 610 a -1
{mode=DF} 610 n 633 a -1
{mode=XF} 633 n 637 a -1
{mode=TF} 637 n 2244 a -1
{mode=TI} 2244 n 1 a -1
{code=reg + elt0_i=17} 1 n 226 a -1
{mode=CCFP} 226 n 244 a -1
{mode=CCFPU} 244 n 377 a -1
{code=strict_low_part} 377 n 1441 a -1
{code=reg} 1441 n 5583 a -1
{code=pc} 5583 n 1333 a -1
{pred=(register_operand,VOID) + A_op=0} 1333 n 7516 a -1
{A_op=0} 7516 n 7552 a -1
{mode=V4SF} 7552 n 7556 a -1
{mode=V4SI} 7556 n 7560 a -1
{mode=V8QI} 7560 n 7564 a -1
{mode=V4HI} 7564 n 7568 a -1
{mode=V2SI} 7568 n 7572 a -1
{mode=V2SF} 7572 n 7576 a -1
{mode=TI} 7576 n 7622 a -1
{mode=SI} 7622 n 7651 a -1
{mode=DI} 7651 n 7706 a -1
{mode=SF} 7706 n 8016 a -1
{mode=CCFP} 8016 n 8027 a -1
{mode=CCFPU} 8027 n 8674 a -1
{A_op=0} 8674 n 8701 a -1
{mode=V2SF} 8701 n 8719 a -1
{mode=V2SI} 8719 n 8888 a -1
{mode=V8QI} 8888 n 8922 a -1
{mode=V4HI} 8922 n -1 a -1
```

find_afterward () 함수 처리가 완료된 후, 모습이다. a 뒤의 값이 변경되었으며, a 의 값의 의미가 특정 decision 의 number 임을 알 수 있을 것이다.

```
{mode=HI} 138 n 145 a -1
{mode=CCFP} 145 n 197 a 1
{mode=CCFPU} 197 n 221 a 1
{mode=CC} 221 n 262 a 1
{mode=SI} 262 n 394 a 1
{mode=QI} 394 n 508 a 1
{mode=DI} 508 n 591 a 1
{mode=SF} 591 n 610 a 1
{mode=DF} 610 n 633 a 1
{mode=XF} 633 n 637 a 1
{mode=TF} 637 n 2244 a 1
{mode=TI} 2244 n 1 a 1
```

```
{code=reg + elt0_i=17} 1 n 226 a 226
{mode=CCFP} 226 n 244 a 377
{mode=CCFPU} 244 n 377 a 377
{code=strict_low_part} 377 n 1441 a 7516
{code=reg} 1441 n 5583 a 1333
{code=pc} 5583 n 1333 a 7516
{pred=(register_operand,VOID) + A_op=0} 1333 n 7516 a 7516
{A_op=0} 7516 n 7552 a 7552
{mode=V4SF} 7552 n 7556 a 8674
{mode=V4SI} 7556 n 7560 a 8674
{mode=V8QI} 7560 n 7564 a 8674
{mode=V4HI} 7564 n 7568 a 8674
{mode=V2SI} 7568 n 7572 a 8674
{mode=V2SF} 7572 n 7576 a 8674
{mode=TI} 7576 n 7622 a 8674
{mode=SI} 7622 n 7651 a 8674
{mode=DI} 7651 n 7706 a 8674
{mode=SF} 7706 n 8016 a 8674
{mode=CCFP} 8016 n 8027 a 8674
{mode=CCFPU} 8027 n 8674 a 8674
{A_op=0} 8674 n 8701 a 8701
{mode=V2SF} 8701 n 8719 a -1
{mode=V2SI} 8719 n 8888 a -1
{mode=V8QI} 8888 n 8922 a -1
{mode=V4HI} 8922 n -1 a -1
```

- need_label 이 값은 C 표현식이 만들어지면서 그때 상황에 따라 목적 decision node 의 값이 설정될 수 있는데, 설정되는 조건은 다음과 같다.
  - find_afterward () 함수에서 true 일 수 있는 alternative 를 찾을 경우, 해당 alternative 는 need_label 이 설정된다.
  - write_switch () 함수를 통해서 SWITCH 구문을 만들 때, CASE 혹은 DEFAULT 내 goto 문이 존재한다면, 그 해당 decision node 에 대해서는 need_label 이 1 로 설정되는 것이 필요하다.
  - write_action () 함수를 통해서 goto 구문이 생성될 때, 해당 decision node 에 대해서는 need_label 이 1 로 설정되는 것이 필요하다.

17.4.4 simplify_tests () 함수

- 각 node 들을 재귀적으로 살펴보면서, DT_pred 속성을 가진 node 들을 찾은 후, DT_pred 속성 앞에 DT_mode 혹은 DT_code 가 있으면 잘라낸다. 예를 들어 설명하면 아래와 같은 node 가 있다고 했을 때,

```
{mode=DF + pred=(nonimmediate_operand,DF) + A_op=2} 168 n -1 a 141
  {c_test="TARGET_80387" + A_insn=(22,0)} 169 n -1 a 141
```

아래와 같은 DT_mode 를 날려, "mode=DF" 는 사라졌다.

```
{pred=(nonimmediate_operand,DF) + A_op=2} 168 n -1 a 141
  {c_test="TARGET_80387" + A_insn=(22,0)} 169 n -1 a 141
```

17.4.5 write_subroutines () 함수

- 재귀적으로 write_subroutines () 함수를 호출하여, 가장 하위 node 로 이동한 후에, 밖으로 빠져 나오면서, 해당 node 의 subroutine_number 를 검사한다. 즉 head→first→subroutine_number 값을 검사

한다. 앞에서 break_out_subroutines () 함수를 이용하여 하나의 경계점을 만들어 놓았으며, 이 값이 0 보다 크다면, 즉 경계점이라면 write_subroutine () 함수를 호출한다.

- write_subroutine () 함수에서는 하나의 경계점에 대한 함수 선언과 도입부를 생성하게 되며, 각 head→first→tests 들에 대한 정보는 write_tree () 함수에서 실제로 output 된다.
  - 도입부에서 사용되는 변수의 갯수는 max_depth 와 연관이 있다.

```
static int recog_1 PARAMS ((rtx, rtx, int *));
static int
recog_1 (x0, insn, pnum_clobbers)
     rtx x0 ATTRIBUTE_UNUSED;
     rtx insn ATTRIBUTE_UNUSED;
     int *pnum_clobbers ATTRIBUTE_UNUSED;
{
  rtx * const operands ATTRIBUTE_UNUSED = &recog_data.operand[0];
  rtx x1 ATTRIBUTE_UNUSED;
  rtx x2 ATTRIBUTE_UNUSED;
  rtx x3 ATTRIBUTE_UNUSED;
  rtx x4 ATTRIBUTE_UNUSED;
  rtx x5 ATTRIBUTE_UNUSED;
  rtx x6 ATTRIBUTE_UNUSED;
  rtx x7 ATTRIBUTE_UNUSED;
  int tem ATTRIBUTE_UNUSED;
```

  - 위의 내용이 도입부이며, x1 ~ x7 이 max_depth 의 크기에 따라 결정되는 요소이며, map_depth 값이 7 일 경우, 위와 같이 7 개의 변수 생성된다.

- write_tree () 함수에서 change_state () 함수를 이용하여, 각 node 에 대한 depth 를 계산하여 output 하고, 실제 같은 level 인 node 들에 대한 출력은 write_tree_1 () 함수로 넘긴다.
  - change_state () 함수에 대해 설명하면, decision node 의 position 의 값 변화에 따라 다른데, 각 각의 경우에 따라 아래와 같은 구문이 생성된다.

“” → “0”	x1 = XEXP (x0, 0);
“” → “1”	x1 = XEXP (x0, 1);
“” → “110”	x1 = XEXP (x0, 1); x2 = XEXP (x1, 1); x3 = XEXP (x2, 0);
“” → “1a0”	x1 = XEXP (x0, 1); x2 = XVECEXP (x1, 0, 0); x3 = XEXP (x2, 0);
“0” → “1”	x1 = XEXP (x0, 1);
“1” → “0”	x1 = XEXP (x0, 0);
“1” → “1a”	x2 = XVECEXP (x1, 0, 0);
“1a” → “”0”	x1 = XEXP (x0, 0);
“1a” → “1a0”	x3 = XEXP (x2, 0);
“1a0” → “1a1”	x3 = XEXP (x2, 1);
“1a1” → “1a0”	x3 = XEXP (x2, 0);

- write_tree_1 () 함수의 경우, 만약 작성할 SWITCH 구문이 있을 경우, write_switch () 함수를 호출하며, 아닐 경우 각 node 에 대한 C 표현식을 출력한다. switch 구문은 두개 연속으로 같은 레벨에서 DT_* 가 발견될 경우, 사용한다.

- 각 node 에 대한 C 표현식은 write_node () 함수에서 완성된다. 대부분 if 구문을 생성하며, write_cond () 함수에서는 if 안에 들어간 조건을 ouput 하며, write_action () 함수에서는 해당 구문이 true 일 경우 실행할 statement 를 완성한다.

- decision node 에 afterward 가 존재할 경우, write_afterward () 함수를 이용하여 해당 branch 정보를 output 한다.

좀 더 쉬운 이해를 위해서 다음과 같은 예제에 대해서 설명을 부연하겠다.

```
1  L1106: ATTRIBUTE_UNUSED_LABEL
2   x2 = XVECEXP (x1, 0, 0);
3   if (memory_operand (x2, HImode))
4     {
5       operands[0] = x2;
6       goto L1107;
7     }
8   x1 = XEXP (x0, 0);
9   goto L10842;
```

위의 예제는 insn-recog.c 파일에서 볼 수 있는 일반적인 pattern 으로, 앞의 번호는 줄번호이다.

- 1 번째 줄은 write_tree () 함수에서 생성되며, 현재 decision 의 number 이다.
- 2 번째 줄은 write_tree () 함수에서 호출된, change_state () 함수에서 output 된다.
- 3 번째 줄은 write_tree () 함수에서 호출된, write_tree_1 함수의 write_node () 함수의 write_cond () 함수에서 output 된다.
- 4 ~ 7 번째 줄은, write_node () 함수의 write_action () 함수에서 output 된다.
- 8 ~ 9 번째 줄은 write_tree_1 함수에서 호출된 write_afterward () 함수에 output 되며, 8 번째 줄은 이 함수내에 포함된 change_state () 함수에서 output 된다.

17.5 Debugging

해당 rtx 에 대한 생성되는 decision 에 대한 debugging 정보를 이용하기 위해서는 $prefix/gcc/genrecog.c 에 선언되어 있는 debug_decision () 함수와 debug_decision_list () 함수들에 대해서 알아보기 바란다.

```
(define_insn "cmpdi_ccno_1_rex64"
  [(set (reg 17)
        (compare (match_operand:DI 0 "nonimmediate_operand" "r,?mr")
                 (match_operand:DI 1 "const0_operand" "n,n")))]
  "TARGET_64BIT && ix86_match_ccmode (insn, CCNOmode)"
  "@
   test{q}\t{%0, %0|%0, %0}
   cmp{q}\t{%1, %0|%0, %1}"
  [(set_attr "type" "test,icmp")
   (set_attr "length_immediate" "0,1")
   (set_attr "mode" "DI")])
```

위의 구문의 경우 다음과 같은 decision 이 생성된다.

```
{code=set} 0 n -1 a -1
  {code=reg + elt0_i=17} 1 n -1 a -1
    {code=compare} 2 n -1 a -1
      {mode=DI + pred=(nonimmediate_operand,DI) + A_op=0} 3 n -1 a -1
        {pred=(const0_operand,DI) + A_op=1} 4 n -1 a -1
          {c_test="TARGET_64BIT && ..." + A_insn=(0,0)} 5 n -1 a -1
```

위의 decision 은 merging 이 이루어지기 전의 모습이다.

17.6 구조체

아래와 같은 구조체가 이 프로그램에서 사용된다.

17.6.1 genrecog.c 파일

```
/* Decision tree 들의 listhead.  Node 에 대한 alternative 들은
   doublely-linked list 로 유지되어서 우리는 merging 시 적당한 장소에 쉽게
   node 를 추가할 수 있다.  */

struct decision_head
{
  struct decision *first;
  struct decision *last;
};

/* Single test.  두 accept type 들은 per-se 관련 test 가 아니지만,
   같거나 부족할 경우 tree merging 에 영향을 미치기 때문에, 여기에서
   유지하는 것이 편리하다.  */

struct decision_test
{
  /* Node 에 첨부된 test 들을 통해 연결된 linked list.  */
  struct decision_test *next;

  /* 이 type 들은 우리가 테스트하고자 하는 순으로 순서없이 나열되어 있다.  */
  enum decision_type
    {
      DT_mode, DT_code, DT_veclen,
      DT_elt_zero_int, DT_elt_one_int, DT_elt_zero_wide, DT_elt_zero_wide_safe,
      DT_veclen_ge, DT_dup, DT_pred, DT_c_test,
      DT_accept_op, DT_accept_insn
    } type;

  union
  {
    enum machine_mode mode;     /* Node 의 machine mode.  */
    RTX_CODE code;              /* Test 할 code.  */

    struct
    {
      const char *name;         /* 호출할 predicate.  */
      int index;                /* 'preds' 혹은 -1 인 index.  */
      enum machine_mode mode;   /* Node 를 위한 machine mode.  */
    } pred;

    const char *c_test;         /* 실행할 추가적인 test.  */
    int veclen;                 /* Vector 의 길이.  */
    int dup;                    /* 비교 대상인 operand 의 번호.  */
    HOST_WIDE_INT intval;       /* XWINT 용 XINT 용 값..  */
    int opno;                   /* 매치된 operand 번호.  */

    struct {
      int code_number;          /* 매치된 insn number.  */
      int lineno;               /* Insn 의 줄번호.  */
      int num_clobbers_to_add;  /* 추가된 CLOBBER 들의 갯수.  */
```

```
    } insn;
  } u;
};

/* 합법적인 insn 들을 인식하기 위한 decision tree 용 데이타 구조체.  */

struct decision
{
  struct decision_head success; /* Test 성공한 node 들.  */
  struct decision *next;        /* Test 실패한 node.  */
  struct decision *prev;        /* 해당 node 가 우리를 테스트했을 때
                                   실패한 node.  */
  struct decision *afterward;   /* Test 성공한 node 이지만,
                                   Successor 의 실패 node 들.  */

  const char *position;         /* Pattern 에서 위치를 알리는 문자열..  */

  struct decision_test *tests;  /* 이 node 용 test 들.  */

  int number;                   /* Node 번호, label 에 사용됨 */
  int subroutine_number;        /* 이 node 가 시작하는 subroutine 의 갯수 */
  int need_label;               /* Output 시 label 이 필요함.  */
};
```

위에서 next 와 prev 는 merge_trees () 함수를 실행하면서 설정되는데, 두개의 decision D1 과 D2 를 merging 하는 과정에서 처음으로 서로 맞지 않는 부분을 발견하였을 때 서로 설정되어 진다. 위에서의 Merging 이 이루어지는 동안 섹션을 참조하기 바란다.

```
/* 우리는 subroutine 으로 세가지 type 들을 쓸 수 있다.  하나는 insn 인식을 위해
   다른 하나는 slplit insn 들을 위해,  나머지 하나는 peephole-type 최적화들을
   위한 것이 그것이다.  이것은  어떤 type 이 쓰여질 지를 정의한다.  */

enum routine_type {
  RECOG, SPLIT, PEEPHOLE2
};

#define IS_SPLIT(X) ((X) != RECOG)

/* 이 table 은 recog.c 에 정의된 predicate 와 match 될 가능성이 있는 rtl code
   목록을 포함한다.  함수 'maybe_both_true' 는 tree node 들의 특정 pair 들에
   의한 match 들이 될 수 있는 표현식이 존재하지 않음을 유추하기 위해
   이것을 사용한다. 또한, 만약 predicate 가 오직 하나의 code 만 match
   될 수 있다면, 우리는 predicate 를 테스트하는 node 에 해당 code 를
   hardwire 할 수 있다.  */

static const struct pred_table
{
  const char *const name;
  const RTX_CODE codes[NUM_RTX_CODE];
} preds[] = {
  {"general_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,
                       LABEL_REF, SUBREG, REG, MEM}},
#ifdef PREDICATE_CODES
```

```
  PREDICATE_CODES
#endif
  {"address_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,
                       LABEL_REF, SUBREG, REG, MEM, PLUS, MINUS, MULT}},
  {"register_operand", {SUBREG, REG}},
  {"pmode_register_operand", {SUBREG, REG}},
  {"scratch_operand", {SCRATCH, REG}},
  {"immediate_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,
                         LABEL_REF}},
  {"const_int_operand", {CONST_INT}},
  {"const_double_operand", {CONST_INT, CONST_DOUBLE}},
  {"nonimmediate_operand", {SUBREG, REG, MEM}},
  {"nonmemory_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,
                         LABEL_REF, SUBREG, REG}},
  {"push_operand", {MEM}},
  {"pop_operand", {MEM}},
  {"memory_operand", {SUBREG, MEM}},
  {"indirect_operand", {SUBREG, MEM}},
  {"comparison_operator", {EQ, NE, LE, LT, GE, GT, LEU, LTU, GEU, GTU,
                           UNORDERED, ORDERED, UNEQ, UNGE, UNGT, UNLE,
                           UNLT, LTGT}},
  {"mode_independent_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,
                                LABEL_REF, SUBREG, REG, MEM}}
};

#define NUM_KNOWN_PREDS ARRAY_SIZE (preds)

static const char *const special_mode_pred_table[] = {
#ifdef SPECIAL_MODE_PREDICATES
  SPECIAL_MODE_PREDICATES
#endif
  "pmode_register_operand"
};

#define NUM_SPECIAL_MODE_PREDS ARRAY_SIZE (special_mode_pred_table)
```

17.6.2 i386.h 파일

```
#define PREDICATE_CODES                                                 \
  {"x86_64_immediate_operand", {CONST_INT, SUBREG, REG,                 \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_nonmemory_operand", {CONST_INT, SUBREG, REG,                 \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_movabs_operand", {CONST_INT, SUBREG, REG,                    \
                                SYMBOL_REF, LABEL_REF, CONST}},         \
  {"x86_64_szext_nonmemory_operand", {CONST_INT, SUBREG, REG,           \
                                      SYMBOL_REF, LABEL_REF, CONST}},   \
  {"x86_64_general_operand", {CONST_INT, SUBREG, REG, MEM,              \
                              SYMBOL_REF, LABEL_REF, CONST}},           \
  {"x86_64_szext_general_operand", {CONST_INT, SUBREG, REG, MEM,        \
                                    SYMBOL_REF, LABEL_REF, CONST}},     \
  {"x86_64_zext_immediate_operand", {CONST_INT, CONST_DOUBLE, CONST,    \
```

```
                                    SYMBOL_REF, LABEL_REF}},            \
  {"shiftdi_operand", {SUBREG, REG, MEM}},                              \
  {"const_int_1_operand", {CONST_INT}},                                 \
  {"symbolic_operand", {SYMBOL_REF, LABEL_REF, CONST}},                 \
  {"aligned_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,      \
                       LABEL_REF, SUBREG, REG, MEM}},                   \
  {"pic_symbolic_operand", {CONST}},                                    \
  {"call_insn_operand", {REG, SUBREG, MEM, SYMBOL_REF}},                \
  {"constant_call_address_operand", {SYMBOL_REF, CONST}},               \
  {"const0_operand", {CONST_INT, CONST_DOUBLE}},                        \
  {"const1_operand", {CONST_INT}},                                      \
  {"const248_operand", {CONST_INT}},                                    \
  {"incdec_operand", {CONST_INT}},                                      \
  {"mmx_reg_operand", {REG}},                                           \
  {"reg_no_sp_operand", {SUBREG, REG}},                                 \
  {"general_no_elim_operand", {CONST_INT, CONST_DOUBLE, CONST,          \
                         SYMBOL_REF, LABEL_REF, SUBREG, REG, MEM}},     \
  {"nonmemory_no_elim_operand", {CONST_INT, REG, SUBREG}},              \
  {"q_regs_operand", {SUBREG, REG}},                                    \
  {"non_q_regs_operand", {SUBREG, REG}},                                \
  {"fcmov_comparison_operator", {EQ, NE, LTU, GTU, LEU, GEU, UNORDERED, \
                                 ORDERED, LT, UNLT, GT, UNGT, LE, UNLE, \
                                 GE, UNGE, LTGT, UNEQ}},                \
  {"sse_comparison_operator", {EQ, LT, LE, UNORDERED, NE, UNGE, UNGT,   \
                               ORDERED, UNEQ, UNLT, UNLE, LTGT, GE, GT  \
                               }},                                      \
  {"ix86_comparison_operator", {EQ, NE, LE, LT, GE, GT, LEU, LTU, GEU,  \
                               GTU, UNORDERED, ORDERED, UNLE, UNLT,     \
                               UNGE, UNGT, LTGT, UNEQ }},               \
  {"cmp_fp_expander_operand", {CONST_DOUBLE, SUBREG, REG, MEM}},        \
  {"ext_register_operand", {SUBREG, REG}},                              \
  {"binary_fp_operator", {PLUS, MINUS, MULT, DIV}},                     \
  {"mult_operator", {MULT}},                                            \
  {"div_operator", {DIV}},                                              \
  {"arith_or_logical_operator", {PLUS, MULT, AND, IOR, XOR, SMIN, SMAX, \
                                 UMIN, UMAX, COMPARE, MINUS, DIV, MOD,  \
                                 UDIV, UMOD, ASHIFT, ROTATE, ASHIFTRT,  \
                                 LSHIFTRT, ROTATERT}},                  \
  {"promotable_binary_operator", {PLUS, MULT, AND, IOR, XOR, ASHIFT}},  \
  {"memory_displacement_operand", {MEM}},                               \
  {"cmpsi_operand", {CONST_INT, CONST_DOUBLE, CONST, SYMBOL_REF,        \
                     LABEL_REF, SUBREG, REG, MEM, AND}},                \
  {"long_memory_operand", {MEM}},
```

17.7 전역변수

```
/* insn_code_number 순으로 나열된 name 배열을 가진다.  */
static char **insn_name_ptr = 0;
static int insn_name_ptr_size = 0;

#define SUBROUTINE_THRESHOLD    100
```

```
static int next_subroutine_number;

/* Tree node 를 위해 이용가능한 다음 node 번호. */

static int next_number;

/* insn_code 로 사용할 다음 번호. */

static int next_insn_code;

/* 비슷하지만 MD 파일내 모든 표현식을 세는데, 오류 메시지에 사용된다. */

static int next_index;

/* 우리가 해석할 때 본 가장 깊은 depth 를 기록해서 우리가 만들 각 subroutine
   에서 할당할 변수가 얼마나 많은지 안다. */

static int max_depth;

/* 현재 해석되고 있는 pattern 의 시작 줄번호. */
static int pattern_lineno;

/* 오류의 갯수. */
static int error_count;

static const char *last_real_name = "insn";
static int last_real_code = 0;
```

제 18 절 gensupport

다른 gen* 시리즈들이 사용하는 여러 생성 단계들을 위한 지원 routine 들로 구성되어 있다.
이 파일에서 가장 중요한 함수는 아래의 두 함수이다.

- init_md_reader_args () 함수

  Reader 를 초기화하기 위한 entry point. 지정된 MD 파일을 읽어서, 해당 문자열로 구성된 각 node 들을 실제 rtx node 로 변환시켜, $prefix/gcc/gensupport.c 파일에 선언되어 있는 전역 변수 define_attr_queue 와 define_insn_queue, other_queue 에 linked list 를 구성한다.

- read_md_rtx 함수

  MD 파일로부터 단일 rtx 를 읽는 entry point. 이것은 앞의 init_md_reader_args () 함수에서 읽은 각 rtx node 들을 하나씩 읽어 오는 것을 말하는데, 다음과 같은 순서로 읽히게 되는 것이 중요하다.

  – define_attr_queue 가 가장 먼저 output 되며, 이것이 null 이 될 때까지 계속 진행된다.
  – 위 전역 변수를 다 읽었다면, 그 다음으로 define_insn_queue 로 넘어간다.
  – 마지막으로 other_queue 가 읽힌다.