

# GCC RTL Representation

## (2) 변수와 구조체 그리고 함수들

정원교  
weongyo@hotmail.com

2005년 9월 23일

### 목 차

제 1 절	17 주째 문서를 시작하며	3
제 2 절	전역 열거자	3
제 3 절	전역 변수와 그에 따른 정의들	9
제 4 절	구조체	10
제 5 절	접근자들	14
제 6 절	RTL 구현을 위한 함수들	28
6.1	alias.c . . . . .	28
6.2	builtins.c . . . . .	30
6.3	calls.c . . . . .	30
6.4	cfgrtl.c . . . . .	30
6.5	combine.c . . . . .	30
6.6	cse.c . . . . .	31
6.7	emit-rtl.c . . . . .	32
6.8	expow.c . . . . .	43
6.9	expmed.c . . . . .	44
6.10	expr.c . . . . .	44
6.11	flow.c . . . . .	44
6.12	fold-const.c . . . . .	45
6.13	function.c . . . . .	46
6.14	gcse.c . . . . .	48
6.15	global.c . . . . .	48
6.16	ifcvt.c . . . . .	49
6.17	list.c . . . . .	49
6.18	local-alloc.c . . . . .	49
6.19	jump.c . . . . .	50
6.20	loop.c . . . . .	54
6.21	predict.c . . . . .	54
6.22	print-rtl.c . . . . .	55
6.23	profile.c . . . . .	55
6.24	read-rtl.c . . . . .	56
6.25	recog.c . . . . .	56

---

6.26	reg-stack.c . . . . .	57
6.27	regclass.c . . . . .	57
6.28	regmove.c . . . . .	58
6.29	regrename.c . . . . .	59
6.30	reorg.c . . . . .	59
6.31	rtl.c . . . . .	59
6.32	rtlanal.c . . . . .	59
6.33	sibcall.c . . . . .	66
6.34	simplify-rtx.c . . . . .	66
6.35	stmt.c . . . . .	69
6.36	unroll.c . . . . .	69
6.37	varasm.c . . . . .	69

## 제 1 절 17 주째 문서를 시작하며

저번 주 우리는 RTL 을 구성하는 각 RTX 에 대해 살펴 보았었다. 이번 주는 GCC 내에서 이 메카니즘을 사용하기 위해서 선언한 전역변수 및 함수들을 살펴보는 시간을 갖도록 하겠다.

설명은 나열식으로 되어 있으며, 전역 변수 및 각 함수들을 섹션으로 묶여져 있다.

## 제 2 절 전역 열거자

각 RTX 의 operation 을 구현하기 위해서 내부적으로 사용하는 전역 열거자가 아래와 같이 존재한다. 각 사항에 대한 설명은 아래를 참조하기 바란다.

```
enum rtx_code {
    UNKNOWN, NIL, INCLUDE, EXPR_LIST, INSN_LIST,
    MATCH_OPERAND, MATCH_SCRATCH, MATCH_DUP, MATCH_OPERATOR,
    MATCH_PARALLEL, MATCH_OP_DUP, MATCH_PAR_DUP, MATCH_INSN,
    DEFINE_INSN, DEFINE_PEEPHOLE, DEFINE_SPLIT, DEFINE_INSN_AND_SPLIT,
    DEFINE_PEEPHOLE2, DEFINE_COMBINE, DEFINE_EXPAND, DEFINE_DELAY,
    DEFINE_FUNCTION_UNIT, DEFINE_ASM_ATTRIBUTES, DEFINE_COND_EXEC,
    SEQUENCE, ADDRESS, DEFINE_ATTR, ATTR, SET_ATTR,
    SET_ATTR_ALTERNATIVE, EQ_ATTR, ATTR_FLAG, INSN, JUMP_INSN,
    CALL_INSN, BARRIER, CODE_LABEL, NOTE, COND_EXEC, PARALLEL,
    ASM_INPUT, ASM_OPERANDS, UNSPEC, UNSPEC_VOLATILE, ADDR_VEC,
    ADDR_DIFF_VEC, PREFETCH, SET, USE, CLOBBER, CALL, RETURN,
    TRAP_IF, RESX, CONST_INT, CONST_DOUBLE, CONST_VECTOR,
    CONST_STRING, CONST, PC, VALUE, REG, SCRATCH, SUBREG,
    STRICT_LOW_PART, CONCAT, MEM, LABEL_REF, SYMBOL_REF, CCO,
    ADDRESSOF, QUEUED, IF_THEN_ELSE, COND, COMPARE,
    PLUS, MINUS, NEG, MULT, DIV, MOD, UDIV, UMOD, AND,
    IOR, XOR, NOT, ASHIFT, ROTATE, ASHIFTRT, LSHIFTRT,
    ROTATERT, SMIN, SMAX, UMIN, UMAX, PRE_DEC, PRE_INC,
    POST_DEC, POST_INC, PRE_MODIFY, POST_MODIFY,
    NE, EQ, GE, GT, LE, LT, GEU, GTU, LEU, LTU,
    UNORDERED, ORDERED,
    UNEQ, UNGE, UNGT, UNLE, UNLT, LTGT,
    SIGN_EXTEND, ZERO_EXTEND, TRUNCATE,
    FLOAT_EXTEND, FLOAT_TRUNCATE, FLOAT, FIX,
    UNSIGNED_FLOAT, UNSIGNED_FIX, ABS, SQRT, FFS,
    SIGN_EXTRACT, ZERO_EXTRACT, HIGH, LO_SUM,
    RANGE_INFO, RANGE_REG, RANGE_VAR, RANGE_LIVE,
    CONSTANT_P_RTX, CALL_PLACEHOLDER,
    VEC_MERGE, VEC_SELECT, VEC_CONCAT, VEC_DUPLICATE,
    SS_PLUS, US_PLUS, SS_MINUS, US_MINUS, SS_TRUNCATE, US_TRUNCATE,
    PHI, LAST_AND_UNUSED_RTX_CODE
};
```

위의 각 element 에 대해서는 이전 강의 16 주를 참고하면 될 것이다.

enum reg\_note 에 대해서 살펴보기로 하자. 이 열거자는 이 insn 가 여러 REG 들에게 무엇을 어떻게 영향을 미치는지에 대한 note 들의 list 를 가지고 있다. 이것은 EXPR\_LIST rtx 들의 chain 이며, 두번째 operand 는 chain pointer 이고 첫번째 operand 는 설명된 REG 이다. EXPR\_LIST 의 mode field 는 실제 machine mode 가 아닌 enum reg\_note 에서의 값을 포함하고 있다.

```
enum reg_note
```

```

{
  REG_DEAD = 1,
  REG_INC,
  REG_EQUIV,
  REG_EQUAL,
  REG_WAS_0,
  REG_RETVAL,
  REG_LIBCALL,
  REG_NONNEG,
  REG_NO_CONFLICT,
  REG_UNUSED,
  REG_CC_SETTER, REG_CC_USER,
  REG_LABEL,
  REG_DEP_ANTI, REG_DEP_OUTPUT,
  REG_BR_PROB,
  REG_EXEC_COUNT,
  REG_NOALIAS,
  REG_SAVE_AREA,
  REG_BR_PRED,
  REG_FRAME_RELATED_EXPR,
  REG_EH_CONTEXT,
  REG_EH_REGION,
  REG_SAVE_NOTE,
  REG_MAYBE_DEAD,
  REG_NORETURN,
  REG_NON_LOCAL_GOTO,
  REG_SETJMP,
  REG_ALWAYS_RETURN,
  REG_VTABLE_REF
};

```

위 각 element 들에 대한 설명은 아래와 같다.

#### REG.DEAD

REG 내 값이 이 insn 에서 죽었다. (즉, 이것은 이전 이 insn 에서 필요하지 않았다.) 만약 REG 가 이 insn 내에 설정되어 있다면, REG\_DEAD note 는, 그럴 필요지만, 생략될 수 있겠다.

#### REG.INC

REG 는 자동증가되거나 혹은 자동감소된다.

#### REG.EQUIV

전체적으로 insn 을 설명한다; insn 가 register 를 constant 값 혹은 메모리 주소와 같은 것으로 설정함을 말한다. 만약 register 가 stack 에 spill 되어지면 상수값은 그것을 대체해야만 한다. REG\_EQUIV 의 content 들은 상수값 혹은 메모리 주소인데, 이것은 비록 같은 값을 가지고 있더라도 SET 의 source 와는 다를 것이다. REG\_EQUIV note 는 register parameter 를 pseudo-register 로 복사하는 명령어상에 나타날 수도 있는데, 만약 함수의 전체 기간 동안 pseudo-register 를 잡고 있는데 사용될 수 있는 메모리 주소가 존재할 경우가 해당된다.

#### REG.EQUAL

목적지가 아주 잠깐 지정된 rtx 와 같은 것을 제외하고는 REG\_EQUIV 와 비슷하다. 그런 까닭에, 그것은 substitution 을 위해 사용될 수 없지만 cse 에서는 사용될 수 있다.

## REG\_WAS\_0

이 insn 에서의 register set 이 이 명령 전에 0 을 가지고 있었다. note 의 content 들은 0 을 저장하고 있었던 insn 이다. 만약 그 insn 가 삭제되었거나 혹은 NOTE 로 개선되었다면, REG\_WAS\_0 는 비작동적이다. REG\_WAS\_0 note 는 실제로 EXPR\_LIST 가 아닌, INSN\_LIST 이다.

## REG\_RETVAL

이 insn 는 return 값들을 위해 hard reg 의 library call 의 return-value 를 복사한다. 이 note 는 실제로 INSN\_LIST 이고 call 를 위한 argument 들을 설정하는데 연관되는 첫번째 insn 를 가르킨다. flow.c 는 그것의 결과가 dead 되었을 때 전체 library call 를 삭제하기 위해 이것을 사용한다.

## REG\_LIBCALL

REG\_RETVAL 의 반대: library call 의 첫번째 insn 에서 일어나며, REG\_RETVAL 를 가지고 있는 것을 가르킨다. 이 note 는 또한 INSN\_LIST 이다.

## REG\_NONNEG

레지스터는 containing loop 동안 항상 nonnegative 이다. 이것은 branche 들 내에서 사용되는데, 이로 인해 0 으로 끝나는 decrement 와 branch instruction 들이 매치되어질 수 있다. md 파일에 'decrement\_and\_branch\_until\_zero' 라는 이름으로 명명된 insn 패턴이 반드시 존재하여야 하며, 그렇지 않을 경우 이것은 어떠한 명령어들에게도 추가되지 않을 것이다.

## REG\_NO\_CONFLICT

!이 명령어 뒤부터는! note 내 register 와 이 insn 의 목적지사이에서는 충돌이 없을 것이다.

## REG\_UNUSED

이 insn 내 register 를 인식하고 한번도 사용되지 않은 것들을 가려낸다.

## REG\_CC\_SETTER, REG\_CC\_USER

REG\_CC\_SETTER 와 REG\_CC\_USER 는 각자 CC0 를 설정하고 사용하는 insn 들의 쌍을 연결한다. 보통, 이것들은 연속적인 insn 들이길 요구하지만, 우리는 branch 의 delay slot 내에 cc0-setting insn 를, 오직 insn 의 한 복사본만 존재할 경우에는, 놓는 것을 허락한다. 그러한 경우, 이 note 들은 어떤 다른 필요한 정보를 결정하기 위해, 그리고 적절히 CC\_STATUS 를 업데이트 하기 위해 code generation 을 허락하는 것들을 모두 가르킨다. 이 note 들은 INSN\_LIST 들이다.

## REG\_LABEL

CODE\_LABEL 를 가르킨다. non-JUMP\_INSN 들에서 사용되는데, 이 명령어들은 REG\_LABEL note 내에 포함되어 있는 CODE\_LABEL 이 insn 에 의해 사용됨을 말한다. 이 note 는 INSN\_LIST 이다.

## REG\_DEP\_ANTI, REG\_DEP\_OUTPUT

REG\_DEP\_ANTI 와 REG\_DEP\_OUTPUT 는 각자 write-after-read 와 write-after-write 의존성들을 표현하기 위해 LOG\_LINKS 내에서 사용된다. flow 에 의해 생성되는 LOG\_LINK 의 한 type 인, Data 의존성들은 0 reg note 종류로 표현된다.

## REG\_BR\_PROB

REG\_BR\_PROB 는 JUMP\_INSN 들과 CALL\_INSN 들에 붙는다. 정수값을 가지고 있으며, jump 들에 대해서 이것이 taken branch 일 probability 이다. call 들에 대해서 이것이 return 하지 않을 probability 이다.

## REG\_EXEC\_COUNT

REG\_EXEC\_COUNT 는 각 basic block 의 첫번째 insn 와 각 CALL\_INSN 뒤 첫번째 insn 에 붙는다. 이것은 이 block 이 얼마나 많이 실행되었는지를 가르킨다.

## REG\_NOALIAS

Call insn 에 붙는다; call 이 malloc-like 이며 반환된 pointer 는 다른 어떤 것을 alias 할 수 없다.

## REG\_SAVE\_AREA

SETJMP\_VIA\_SAVE\_AREA 가 true 인 target 에 대해, dynamic stack allocation 에 의해 생성된 rti 를 최적화하는데 사용된다.

## REG\_BR\_PRED

REG\_BR\_PRED 는 JUMP\_INSN 들과 CALL\_INSN 들에 붙는다. 그것은 두 정수값의 CONCAT 을 포함한다. 첫번째는 note 를 더한 branch predictor 를 나타내고 두번째는 REG\_BR\_PROB note 가 사용하는 것과 같은 format 인 branch 의 predicted hitrate 를 나타낸다.

## REG\_FRAME\_RELATED\_EXPR

RTX\_FRAME\_RELATED\_P 인 insn 들에 붙지만, DWARF 에 대해서는 그들이 무엇을 강조하는지를 해석하는데 매우 복잡함을 보인다. 붙은 rtx 는 intuition 대신 사용된다.

## REG\_EH\_CONTEXT

REG 가 함수를 위한 exception context 를 갖고 있음을 가르킨다. 이 context 는 inline 함수들에 의해 공유되는데, 그래서 실제 exception context 를 취득하기 위한 code 는 inlining 후까지 지연 되어 진다.

## REG\_EH\_REGION

INSN 가 무슨 exception region 내에 속하는지를 나타낸다. 이것은 어떤 call 이 어떤 region 으로 던져져야 하는지를 가르키는데 사용된다. REGION 0 는 call 이 전혀 던져질수 없음을 가르킨다. REGION -1 는 그것이 던져 질수 없으며, non-local goto 를 실행하지 않을 것임을 가르킨다.

## REG\_SAVE\_NOTE

scheduling 사이에 NOTE\_INSN note 들을 저장하기 위해 haifa-sched 에서 사용된다.

## REG\_MAYBE\_DEAD

이 insn (prologue 의 한 부분인) 가 나중에 사용되지 않을 수 있는 값을 계산하였으며, 그래서 그 insn 을 지워도 상관없음을 가르킨다. 보통, prologue 내 어떤 insn 를 삭제하는 것은 error 이다. 현재 parameter 는 사용되지 않으며 (const\_int 0) 로 설정된다.

## REG\_NORETURN

Call 이 return 하지 않음을 가르킨다.

## REG\_NON\_LOCAL\_GOTO

Indirect jump 가 computed goto 대신 non-local goto 임을 가르킨다.

## REG\_SETJMP

이 note 의 종류는 'setjmp' 의 각각에 대해 생성되는데, 두번 return 할 수 있는 비슷한 함수에 대해서도 생성된다.

## REG\_ALWAYS\_RETURN

항상 return 하는 call 들을 가르킨다.

## REG\_VTABLE\_REF

Memory load 가 vtable 을 참조함을 가르킨다. 표현식은  
(plus (symbol\_ref vtable.sym) (const\_int offset)) 형식의 것이다.

출번호가 아닌 note 들의 종류로써 NOTE\_LINE\_NUMBER field 에 나타날 수 있는 code 들. 우리는 특별한 note code 들을 위한 것으로 여기서는 0 값을 사용하지 않을려고 하는데 그것은 때때로 source line 이 실제로 0 이 될 수 있기 때문이다! 이것을 알고 있으라. (예를 들면) 이것은 우리가 몇몇 C++ translation unit 을 위한 per-translation-unit constructor 와 destructor 를 위한 code 를 생성할 때 발생한다.

만약 당신이 다음의 값들 중 어떤 것을 변경해야 하거나 여기에 새로운 값을 추가해야 한다. rtl.c 내 note\_insn\_name 배열을 변경하는 것을 잊지 마라.

```
enum insn_note
{
  NOTE_INSN_BIAS = -100,
  NOTE_INSN_DELETED,
  NOTE_INSN_BLOCK_BEG,
  NOTE_INSN_BLOCK_END,
  NOTE_INSN_LOOP_BEG,
  NOTE_INSN_LOOP_END,
  NOTE_INSN_LOOP_CONT,
  NOTE_INSN_LOOP_VTOP,
  NOTE_INSN_LOOP_END_TOP_COND,
  NOTE_INSN_FUNCTION_END,
  NOTE_INSN_PROLOGUE_END,
  NOTE_INSN_EPILOGUE_BEG,
  NOTE_INSN_DELETED_LABEL,
  NOTE_INSN_FUNCTION_BEG,
  NOTE_INSN_EH_REGION_BEG,
  NOTE_INSN_EH_REGION_END,
  NOTE_INSN_REPEATED_LINE_NUMBER,
  NOTE_INSN_RANGE_BEG,
  NOTE_INSN_RANGE_END,
  NOTE_INSN_LIVE,
  NOTE_INSN_BASIC_BLOCK,
  NOTE_INSN_EXPECTED_VALUE,
  NOTE_INSN_MAX
};
```

NOTE\_INSN\_BIAS = -100

이 number 들을 모두 음수로 유지한다. 필요시 적당히 조절한다.

## NOTE\_INSN\_DELETED

이 note 는 chain 의 바깥으로 insn 을 패치하는 것이 안전하지 않을 때 insn 를 제거하기 위해 사용된다.

## NOTE\_INSN\_BLOCK\_BEG

## NOTE\_INSN\_BLOCK\_END

이것들은 lexical block 의 시작과 끝을 마크하기 위해 사용된다. NOTE\_BLOCK 와 identify\_blocks, reorder\_blocks 를 보라.

NOTE\_INSN\_LOOP\_BEG  
NOTE\_INSN\_LOOP\_END

이것은 loop 의 극점들을 mark 한다.

NOTE\_INSN\_LOOP\_CONT

Loop 시 'continue' jump 가 이루어지는 장소에서 생성된다.

NOTE\_INSN\_LOOP\_VTOP

Duplicated exit test 의 시작부분에서 생성된다.

NOTE\_INSN\_LOOP\_END\_TOP\_COND

Loop 의 최상위에서의 conditional 의 끝부분에 생성된다. 이것은 실제로 loop 구조체를 이해하는 것 대신에 loop rotation 의 lame form 을 실행하는데 사용된다. 해당 note 는 rotation 이 완료된 후 폐기된다.

NOTE\_INSN\_FUNCTION\_END

이 note 의 종류는 return insn 혹은 return label 바로 전 함수의 끝 부분에 생성된다. 최적화 컴파일에서 그것이 첫번째 jump 최적화에 의해 삭제되는데, 그 시기는 return statement 없이 함수 body 의 끝을 control 이 떨어져 나갈 수 있을지 없을지를 결정하는 optimizer 를 활성화시킨 후 이다.

NOTE\_INSN\_PROLOGUE\_END

이것은 마지막 prologue insn 바로 뒤 부분의 point 를 mark 한다.

NOTE\_INSN\_EPILOGUE\_BEG

이것은 첫번째 epilogue insn 바로 앞 부분의 point 를 mark 한다.

NOTE\_INSN\_DELETED\_LABEL

우리가 삭제했었던 user-declared label 들의 장소에 생성된다.

NOTE\_INSN\_FUNCTION\_BEG

이 note 는 함수의 실제 body 의 시작을 가르키는데, 즉, 모든 parm 들 바로 뒤 point 는 그들의 home 들, 기타 등등으로 이동되어졌다.

NOTE\_INSN\_EH\_REGION\_BEG  
NOTE\_INSN\_EH\_REGION\_END

Exception handling region 들이 시작하고 끝나는 곳에서의 note 들. Question 내 region 을 확인하기 위해 NOTE\_EH\_HANDLER 를 사용한다.

NOTE\_INSN\_REPEATED\_LINE\_NUMBER

중복 line number note 가 output 될 때마다 생성된다. 예를 들면, 이것은 inline function 의 끝 뒤에 output 되는데, gcov 에서 두번 세어진 경로 부터 inline call 을 포함하는 line 을 막기 위해서 이다.

NOTE\_INSN\_RANGE\_BEG  
NOTE\_INSN\_RANGE\_END

Live range region 의 시작/끝으로써, Stack 상에 할당된 pseudo 들은 임시 register 들로 할당되어질 수 있다. NOTE\_RANGE\_INFO 를 사용한다.

## NOTE\_INSN\_LIVE

어떤 register 들이 현재 live 인지 기록한다. NOTE\_LIVE\_INFO 를 사용한다.

## NOTE\_INSN\_BASIC\_BLOCK

following basic block 을 위한 구조체를 기록한다. NOTE\_BASIC\_BLOCK 를 사용한다.

## NOTE\_INSN\_EXPECTED\_VALUE

Location 에서 register 의 예상된 값을 기록한다. NOTE\_EXPECTED\_VALUE 를 사용한다; (eq (reg) (const\_int)) 와 같이 저장된다.

### 제 3 절 전역 변수와 그에 따른 정의들

RTL 메카니즘의 구현을 위해서 내부적으로 사용하는 전역변수와 이 변수에 접근하기 위해 사용되는 accessor 를 나타내었다.

```
const unsigned char rtx_length[NUM_RTX_CODE];
#define GET_RTX_LENGTH(CODE) (rtx_length[(int) (CODE)])
```

rtx code 로 나열되어 있으며 그 code 에 대한 rtx 의 operand 들의 갯수를 알려줍니다. rtx header data 는 포함하지 않습니다. (code 와 link 들)

```
extern const char * const rtx_name[NUM_RTX_CODE];
#define GET_RTX_NAME(CODE) (rtx_name[(int) (CODE)])
```

rtx code 순으로 나열되어 있으며 C 문자열로 rtx 종류의 이름을 제공합니다.

```
extern const char * const rtx_format[NUM_RTX_CODE];
#define GET_RTX_FORMAT(CODE) (rtx_format[(int) (CODE)])
```

machine mode 순으로 나열되어 있으며 machine mode 의 이름을 제공합니다. 이 이름은 "mode" 문자를 포함하지 않습니다.

```
extern const char rtx_class[NUM_RTX_CODE];
#define GET_RTX_CLASS(CODE) (rtx_class[(int) (CODE)])
```

machine mode 순으로 나열되어 있으며 mode 의 길이를 비트 크기로 제공합니다. GET\_MODE\_BITSIZE 가 이것을 사용합니다.

```
extern const char * const reg_note_name[];
#define GET_REG_NOTE_NAME(MODE) (reg_note_name[(int) (MODE)])
```

EXPR\_LIST insn 들내 REG\_NOTE 들을 위한 이름들.

```
extern const char * const note_insn_name[NOTE_INSN_MAX - NOTE_INSN_BIAS];
#define GET_NOTE_INSN_NAME(NOTE_CODE) \
  (note_insn_name[(NOTE_CODE) - (int) NOTE_INSN_BIAS])
```

Line number 들보다는 NOTE insn 들을 위한 이름들.

## 제 4 절 구조체

이 섹션에서는 RTL 을 표현하기 위한 RTX 를 담기 위해 선언된 구조체의 모습과 각 구성요소에 대한 설명을 나열하도록 하겠다.

RTX 을 담기 위한 구조체는 크기는 두개로 나뉘지만, 더 크게 보았을 때는 하나로 표현되어진다. 두 구조체는 아래와 같으며

```
typedef struct rtx_def *rtx;
typedef struct rtvec_def *rtvec;
```

이 구조체를 통해서 RTL 이 표현되어 지게 된다. 아래에 이 두 구조체의 구성요소를 살펴보자. 우선 RTL expression (“rtx”) 구조체를 보도록 하자.

```
struct rtx_def
{
  ENUM_BITFIELD(rtx_code) code: 16;
  ENUM_BITFIELD(machine_mode) mode : 8;

  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned integrated : 1;
  unsigned frame_related : 1;

  rtunion fld[1];
};
```

각 구성요소에 대한 세부 설명은 다음과 같다.

ENUM\_BITFIELD(rtx\_code) code: 16

이 expression 의 종류.

ENUM\_BITFIELD(machine\_mode) mode : 8

이 expression 이 가지고 있는 값의 종류.

unsigned int jump : 1

INSN 에서 만약 이 함수내 제어의 흐름을 변화할 수 있다면 1.  
MEM 에서 MEM\_KEEP\_ALIAS\_SET\_P.  
INSN\_LIST 에서 LINK\_COST\_ZERO.  
SET 에서 SET\_IS\_RETURN\_P.

unsigned int call : 1

INSN 에서 만약 다른 함수를 호출할 수 있다면 1.  
INSN\_LIST 에서 LINK\_COST\_FREE.

unsigned int unchanging : 1

REG 에서 만약 이 표현식의 값이 현재 함수동안 절대 변화하지 않고, 심지어 그것이 명백하게 상수가 아닐지라도 변하지 않는다면 1.  
만약 memory 의 content 들이 상수이면 MEM 에서 값이 1. 이것은 이 expression 의 값이 받드

시 상수임을 의미하지 않습니다.

SUBREG 에서 만약 unsigned 인 promoted variable 로부터 기인했다면 1.

SYMBOL\_REF 에서 만약 per-function constants pool 에서 어떤 것을 주소화하고 있다면 1.

CALL\_INSN 에서 만약 그것이 const call 일 경우 1.

JUMP\_INSN 에서 만약 그것이 무효로 되어져야 하는 branch 일 경우 1. 컴파일의 끝까지 reorg 로부터 유효; 사용되기 전 깨끗히 됨.

unsigned int volatil : 1

만약 memory 의 content 들이 volatile 이면 MEM expression 에서 값이 1.

INSN 혹은 CALL\_INSN, JUMP\_INSN, CODE\_LABEL, BARRIER 에서 만약 그것이 삭제되었다면 1.

REG 표현식에서 만약 변수에 상응하는 것이 사용자에게 의해 선언되었다면 1. 내부적으로 생성된 임시의 것에 대해서는 0.

SYMBOL\_REF 에서 이 flag 는 machine-specific 목적들을 위해 사용된다.

LABEL\_REF 혹은 REG\_LABEL note 에서, 이것은 LABEL\_REF\_NONLOCAL\_P 이다.

unsigned int in\_struct : 1

MEM 에서 aggregate 의 field 로의 참조에 대해서는 1.

만약 MEM 가 C 에서 변수였거나 혹은 \* operator 의 결과였다면 0.

만약 그것이 C 에서 . 혹은 -> operator (구조체 상에서) 의 결과일 경우 1.

REG 에서 만약 register 가 오직 loop 중 exit code 에서만 사용될 경우 1.

SUBREG 표현식에서 만약 이 표현식이 promoted mode 를 가진 변수로부터 생성되었을 경우 1.

CODE\_LABEL 에서 만약 label 이 nonlocal goto 들을 위해 사용되거나 그것의 count 가 0 이 더라고 절대 삭제되어서는 안될 경우 1.

LABEL\_REF 에서 만약 이것이 현재 loop 밖의 label 로의 reference 이라면 1.

INSN 혹은 JUMP\_INSN, CALL\_INSN 에서 만약 이 insn 이 반드시 preceding insn 와 함께 스케줄되어야 한다면 1. 오직 sched 내에서만 유효.

INSN 혹은 JUMP\_INSN, CALL\_INSN 에서 만약 insn 이 delay slot 내에 있거나 branch 의 target 으로부터의 것이라면 1. 컴파일의 끝까지 reorg 로부터 유효; 사용되기 전 깨끗히 됨.

unsigned int used : 1

만약 이 rtx 가 사용되었다면 1. 이것은 공유된 구조체를 복사하는데 사용된다. 'unshare\_all\_rtl' 를 참조.

REG 에서, 이것은 그러한 목적으로는 필요하지 않으며 대신 'leaf\_renumber\_regs\_insn' 에서 사용된다.

SYMBOL\_REF 에서는 emit\_library\_call 가 함수로써 그것을 사용했음을 의미한다.

unsigned int integrated : 1

이 rtx 가 procedure integration 로부터 왔다면 0 이 아님.

REG 에서, 0 이 아님은 이 reg 가 현재 함수의 return 값을 참조함을 의미한다. 만약 symbol 이 weak 라면 SYMBOL\_REF 에서 1 을 가집니다.

unsigned int frame\_related : 1

INSN 혹은 SET 에서 만약 이 rtx 가 call frame 과 관련이 있을 경우 1 인데, 이 call frame 은 우리가 frame address 를 어떻게 계산할지를 변경한다거나 prologue 와 epilogue 에서 register 들을 저장하고 복구하는데 관여한다.

MEM 에서 만약 MEM 가 aggregate 의 member 보다는 scalar 를 참조할 경우 1.

REG 에서 1 일 경우 레지스터는 포인터이다.

SYMBOL\_REF 에서 만약 per-function constant string pool 에서 어떤 것을 주소화하고 있을 경우 1.

```
rtunion fld[1]
```

이 rtx 의 operand 들 중 첫번째 요소. 그들의 type 들과 operand 들의 수는 rtl.def 에 따른 ‘code’ field 에 의해 제어됩니다.

다음으로 vector 를 담기 위한 구조체이다. RTL vector. 이것들은 이것의 변수 갯수에 대한 필요성이 있을 때 RTX 들내에 보인다. Principle use 는 PARALLEL 표현식들 내 보인다. 구성요소를 이루는 num.elem 은 element 들의 갯수를 말한다.

```
struct rtvec_def {
    int num_elem;
    rtx elem[1];
};
```

rtx 구조체를 이루는 구성요소 중 rtunion 공용체에 대해서 살펴 보고, 포함하고 있는 구조체에 대해서도 나열하도록 하겠다.

```
typedef union rtunion_def
{
    HOST_WIDE_INT rtwint;
    int rtint;
    unsigned int rtuint;
    const char *rtstr;
    rtx rtx;
    rtvec rtvec;
    enum machine_mode rtttype;
    addr_diff_vec_flags rt_addr_diff_vec_flags;
    struct cselib_val_struct *rt_cselib;
    struct bitmap_head_def *rtbit;
    tree rttree;
    struct basic_block_def *bb;
    mem_attrs *rtmem;
} rtunion;
```

여기에서 addr\_diff\_vec\_flags 구조체와 mem\_attrs 구조체만이 \$prefix/gcc/rtl.h 파일에 선언되어 있으며, 다른 구조체들을 각 다른 파일에 선언되어 있다. 각 구조체에 대해 살펴보도록 rtl.h 파일 외부에 선언된 구조체에 대해서는 간단히 언급만 하겠다.

```
typedef struct
{
    unsigned min_align: 8;
    unsigned base_after_vec: 1;
    unsigned min_after_vec: 1;
    unsigned max_after_vec: 1;
    unsigned min_after_base: 1;
    unsigned max_after_base: 1;
    unsigned offset_unsigned: 1;
    unsigned : 2;
    unsigned scale : 8;
} addr_diff_vec_flags;
```

ADDR\_DIFF\_VEC 의 flag 들과 bitfield 들. BASE 는 rtl.def 에서 설명된 것과 같이 어떤 offset 들이 계산되어졌는가와 연관된 base label 이다.

각 구성요소에 대해 살펴보면 아래와 같다.

unsigned min\_align: 8

shorten\_branches 의 시작 부분에 설정 - 오직 최적화할 때만!!! - :

unsigned base\_after\_vec: 1

BASE 는 ADDR\_DIFF\_VEC 의 뒤이다.

unsigned min\_after\_vec: 1

최소 address target label 은 ADDR\_DIFF\_VEC 뒤이다.

unsigned max\_after\_vec: 1

최대 address target label 은 ADDR\_DIFF\_VEC 뒤이다.

unsigned min\_after\_base: 1

최소 address target label 은 BASE 뒤이다.

unsigned max\_after\_base: 1

최대 address target label 은 BASE 뒤이다.

unsigned offset\_unsigned: 1

실제 branch shortening process 에 의해 설정 - 오직 최적화할 때만!!! offset 들은 반드시 unsigned 로 취급되어야 한다.

unsigned scale : 8

아직 정확한 설명이 없음.

MEM 의 attribute 들을 표현하는데 사용되는 구조체. 이것들은 hash 되어서 같은 attribute 들을 가진 MEM 들은 data 구조체를 공유합니다. 이것은 그것은 그 자리에서 바로 수정될 수 없음을 의미합니다. 만약 어느 element 가 0 이 아니라면 대응하는 attribute 의 값은 알려지지 않았음을 의미합니다.

```
typedef struct
{
    HOST_WIDE_INT alias;
    tree expr;
    rtx offset;
    rtx size;
    unsigned int align;
} mem_attrs;
```

각 구성요소에 대한 설명은 아래와 같다.

HOST\_WIDE\_INT alias

Memory alias set.

tree expr

MEM 에 대응하는 expr.

rtx offset

CONST\_INT 처럼 DECL 의 시작부터의 offset.

rtx size

CONST\_INT 처럼 바이트 크기.

unsigned int align

비트로의 MEM 의 alignment.

나머지 구조체들로는 cselib\_val\_struct 와 bitmap\_head\_def, basic\_block\_def 가 있으며, 이들은 \$prefix/gcc/cselib.h 와 \$prefix/gcc/bitmap.h, \$prefix/gcc/basic-block.h 에 각각 선언되어 있다. 이에 대한 자세한 정보를 얻기 위해서는 해당 파일을 참조하기 바란다.

## 제 5 절 접근자들

위에서 구조체들의 구조와 구성요소에 대해서 살펴보았으므로 이제 각 구성요소에 접근하기 위해서 사용되는 접근자들을 살펴보도록 하자.

```
#define GET_CODE(RTX) ((enum rtx_code) (RTX)->code)
#define PUT_CODE(RTX, CODE) ((RTX)->code = (ENUM_BITFIELD(rtx_code)) (CODE))

#define GET_MODE(RTX) ((enum machine_mode) (RTX)->mode)
#define PUT_MODE(RTX, MODE) ((RTX)->mode = (ENUM_BITFIELD(machine_mode)) (MODE))

#define RTX_INTEGRATED_P(RTX) ((RTX)->integrated)
#define RTX_UNCHANGING_P(RTX) ((RTX)->unchanging)
#define RTX_FRAME_RELATED_P(RTX) ((RTX)->frame_related)
```

위 매크로들은 rtx\_def 구조체에 접근하기 위한 매크로들이다.

```
#define NULL_RTVEC (rtvec) 0

#define GET_NUM_ELEM(RTVEC) ((RTVEC)->num_elem)
#define PUT_NUM_ELEM(RTVEC, NUM) ((RTVEC)->num_elem = (NUM))
```

위 매크로들은 rtvec\_def 구조체에 접근하기 위한 매크로들이다.

```
#define REG_P(X) (GET_CODE (X) == REG)
```

만약 X 가 register 를 위한 rtl 일 경우 nonzero 를 알리는 술어.

```
#define LABEL_P(X) (GET_CODE (X) == CODE_LABEL)
```

만약 X 가 label insn 일 경우 nonzero 를 알리는 술어.

```
#define JUMP_P(X) (GET_CODE (X) == JUMP_INSN)
```

만약 X 가 jump insn 일 경우 nonzero 를 알리는 술어.

```
#define NOTE_P(X) (GET_CODE (X) == NOTE)
```

만약 X 가 note insn 일 경우 nonzero 를 알리는 술어.

```
#define BARRIER_P(X) (GET_CODE (X) == BARRIER)
```

만약 X 가 barrier insn 일 경우 nonzero 를 알리는 술어.

```
#define JUMP_TABLE_DATA_P(INSN) \
  (JUMP_P (INSN) && (GET_CODE (PATTERN (INSN)) == ADDR_VEC || \
                    GET_CODE (PATTERN (INSN)) == ADDR_DIFF_VEC))
```

만약 X 가 jump table 을 위한 data 일 경우 nonzero 를 알리는 술어.

```
#define CONSTANT_P(X) \
  (GET_CODE (X) == LABEL_REF || GET_CODE (X) == SYMBOL_REF \
   || GET_CODE (X) == CONST_INT || GET_CODE (X) == CONST_DOUBLE \
   || GET_CODE (X) == CONST || GET_CODE (X) == HIGH \
   || GET_CODE (X) == CONST_VECTOR \
   || GET_CODE (X) == CONSTANT_P_RTX)
```

만약 X 가 정수인 상수값일 경우 1.

ENABLE\_RTL\_CHECKING 기능이 활성화가 되어 있을 경우 아래와 같은 루틴을 사용하여 RTL 을 검사 하게 된다.

```
#if defined ENABLE_RTL_CHECKING && (GCC_VERSION >= 2007)
#define RTL_CHECK1(RTX, N, C1) __extension__ \
(*({ rtx _rtx = (RTX); int _n = (N); \
enum rtx_code _code = GET_CODE (_rtx); \
if (_n < 0 || _n >= GET_RTX_LENGTH (_code)) \
rtl_check_failed_bounds (_rtx, _n, __FILE__, __LINE__, \
__FUNCTION__); \
if (GET_RTX_FORMAT(_code)[_n] != C1) \
rtl_check_failed_type1 (_rtx, _n, C1, __FILE__, __LINE__, \
__FUNCTION__); \
&_rtx->fld[_n]; }))

#define RTL_CHECK2(RTX, N, C1, C2) __extension__ \
(*({ rtx _rtx = (RTX); int _n = (N); \
enum rtx_code _code = GET_CODE (_rtx); \
if (_n < 0 || _n >= GET_RTX_LENGTH (_code)) \
rtl_check_failed_bounds (_rtx, _n, __FILE__, __LINE__, \
__FUNCTION__); \
if (GET_RTX_FORMAT(_code)[_n] != C1 \
&& GET_RTX_FORMAT(_code)[_n] != C2) \
rtl_check_failed_type2 (_rtx, _n, C1, C2, __FILE__, __LINE__, \
__FUNCTION__); \
&_rtx->fld[_n]; }))

#define RTL_CHECKC1(RTX, N, C) __extension__ \
(*({ rtx _rtx = (RTX); int _n = (N); \
if (GET_CODE (_rtx) != (C)) \
rtl_check_failed_code1 (_rtx, (C), __FILE__, __LINE__, \
__FUNCTION__); \
&_rtx->fld[_n]; }))

#define RTL_CHECKC2(RTX, N, C1, C2) __extension__ \
(*({ rtx _rtx = (RTX); int _n = (N); \
enum rtx_code _code = GET_CODE (_rtx); \
if (_code != (C1) && _code != (C2)) \
rtl_check_failed_code2 (_rtx, (C1), (C2), __FILE__, __LINE__, \
__FUNCTION__); \
&_rtx->fld[_n]; }))

#define RTVEC_ELT(RTVEC, I) __extension__ \
(*({ rtvec _rtvec = (RTVEC); int _i = (I); \
if (_i < 0 || _i >= GET_NUM_ELEM (_rtvec)) \
rtvec_check_failed_bounds (_rtvec, _i, __FILE__, __LINE__, \
__FUNCTION__); \
&_rtvec->elem[_i]; }))

#else
```

```
#define RTL_CHECK1(RTX, N, C1)      ((RTX)->fld[N])
#define RTL_CHECK2(RTX, N, C1, C2) ((RTX)->fld[N])
#define RTL_CHECKC1(RTX, N, C)     ((RTX)->fld[N])
#define RTL_CHECKC2(RTX, N, C1, C2) ((RTX)->fld[N])
#define RTVEC_ELT(RTVEC, I)        ((RTVEC)->elem[I])

#endif
```

이제 검사 루틴을 보았으니, 각 구성요소 특히 rtunion 내의 구성요소로 접근하기 위한 accessor 를 살펴보고 각 accessor 의 이름에 따른 의미를 알아보도록 하자.

```
#define XWINT(RTX, N)      (RTL_CHECK1 (RTX, N, 'w').rtwint)
#define XINT(RTX, N)      (RTL_CHECK2 (RTX, N, 'i', 'n').rtint)
#define XSTR(RTX, N)      (RTL_CHECK2 (RTX, N, 's', 'S').rtstr)
#define XEXP(RTX, N)      (RTL_CHECK2 (RTX, N, 'e', 'u').rtx)
#define XVEC(RTX, N)      (RTL_CHECK2 (RTX, N, 'E', 'V').rtvec)
#define XMODE(RTX, N)     (RTL_CHECK1 (RTX, N, 'M').rttype)
#define XBITMAP(RTX, N)   (RTL_CHECK1 (RTX, N, 'b').rtbit)
#define XTREE(RTX, N)     (RTL_CHECK1 (RTX, N, 't').rttree)
#define XBBDEF(RTX, N)    (RTL_CHECK1 (RTX, N, 'B').bb)
#define XTmpl(RTX, N)     (RTL_CHECK1 (RTX, N, 'T').rtstr)

#define XVECEXP(RTX, N, M)      RTVEC_ELT (XVEC (RTX, N), M)
#define XVECLEN(RTX, N)        GET_NUM_ELEM (XVEC (RTX, N))
```

일반적인 accessor 들.

```
#define XOWINT(RTX, N)      (RTL_CHECK1 (RTX, N, 'O').rtwint)
#define XOINT(RTX, N)      (RTL_CHECK1 (RTX, N, 'O').rtint)
#define XOUINT(RTX, N)     (RTL_CHECK1 (RTX, N, 'O').rtuint)
#define XOSTR(RTX, N)      (RTL_CHECK1 (RTX, N, 'O').rtstr)
#define XOEXP(RTX, N)      (RTL_CHECK1 (RTX, N, 'O').rtx)
#define XOVEC(RTX, N)      (RTL_CHECK1 (RTX, N, 'O').rtvec)
#define XOMODE(RTX, N)     (RTL_CHECK1 (RTX, N, 'O').rttype)
#define XOBITMAP(RTX, N)   (RTL_CHECK1 (RTX, N, 'O').rtbit)
#define XOTREE(RTX, N)     (RTL_CHECK1 (RTX, N, 'O').rttree)
#define XOBDEF(RTX, N)     (RTL_CHECK1 (RTX, N, 'O').bb)
#define XOADVFLAGS(RTX, N) (RTL_CHECK1 (RTX, N, 'O').rt_addr_diff_vec_flags)
#define XOCSELIB(RTX, N)   (RTL_CHECK1 (RTX, N, 'O').rt_cselib)
#define XOMEMATTR(RTX, N) (RTL_CHECK1 (RTX, N, 'O').rtmem)

#define XCWINT(RTX, N, C)   (RTL_CHECKC1 (RTX, N, C).rtwint)
#define XCINT(RTX, N, C)   (RTL_CHECKC1 (RTX, N, C).rtint)
#define XCUINT(RTX, N, C)  (RTL_CHECKC1 (RTX, N, C).rtuint)
#define XCSTR(RTX, N, C)   (RTL_CHECKC1 (RTX, N, C).rtstr)
#define XCEXP(RTX, N, C)   (RTL_CHECKC1 (RTX, N, C).rtx)
#define XCVEC(RTX, N, C)   (RTL_CHECKC1 (RTX, N, C).rtvec)
#define XCMODE(RTX, N, C)  (RTL_CHECKC1 (RTX, N, C).rttype)
#define XCBITMAP(RTX, N, C) (RTL_CHECKC1 (RTX, N, C).rtbit)
#define XCTREE(RTX, N, C)  (RTL_CHECKC1 (RTX, N, C).rttree)
#define XCBDEF(RTX, N, C)  (RTL_CHECKC1 (RTX, N, C).bb)
#define XCADVFLAGS(RTX, N, C) (RTL_CHECKC1 (RTX, N, C).rt_addr_diff_vec_flags)
#define XCCSELIB(RTX, N, C) (RTL_CHECKC1 (RTX, N, C).rt_cselib)
```

```
#define XCVECEXP(RTX, N, M, C) RTVEC_ELT (XCVEC (RTX, N, C), M)
#define XCVECLEN(RTX, N, C) GET_NUM_ELEM (XCVEC (RTX, N, C))
```

```
#define XC2EXP(RTX, N, C1, C2) (RTL_CHECKC2 (RTX, N, C1, C2).rtx)
```

보통 type code 대신에 '0' field 를 예상한다는 것을 제외하고는 XWINT , 기타등등 모습이다.

Insn 들의 특정 field 들에 대한 ACCESS MACRO 들.

```
#define INSN_P(X) (GET_RTX_CLASS (GET_CODE(X)) == 'i')
```

X 가 insn 인지 아닌지를 결정한다.

```
#define INSN_UID(INSN) XINT (INSN, 0)
```

각 insn 를 위한 유일한 번호를 가짐. 순차적으로 증가할 필요는 없습니다.

```
#define PREV_INSN(INSN) XEXP (INSN, 1)
```

```
#define NEXT_INSN(INSN) XEXP (INSN, 2)
```

Sequence 로 묶인 insn 들의 chain.

```
#define PATTERN(INSN) XEXP (INSN, 3)
```

insn 의 body.

```
#define INSN_CODE(INSN) XINT (INSN, 4)
```

이 명령어가 인식되었을 때의 code number. -1 은 이 명령어가 아직 인식되지 않았음을 의미한다.

```
#define LOG_LINKS(INSN) XEXP(INSN, 5)
```

flow.c 에서 설정; 그 전에는 비어있음. INSN\_LIST rtx 들의 chain 을 가지고 있는 INSN\_LIST rtx 의 첫번째 operand 들은 이것에 direct data-flow connection 들을 가진 이전 insn 들을 가르킨다. 그것은 그것들의 insn 들이 변수들을 설정함을 의미한다. 그 변수의 다음 사용은 이 명령어 안에서 발생한다. 그들은 항상 이 insn 처럼 같은 basic block 에 있다.

```
#define INSN_DELETED_P(INSN) ((INSN)->volatile)
```

만약 insn 가 삭제되었었다면 1.

```
#define CONST_OR_PURE_CALL_P(INSN) ((INSN)->unchanging)
```

만약 insn 가 const 혹은 pure function 으로의 call 이면 1.

```
#define SIBLING_CALL_P(INSN) ((INSN)->jump)
```

만약 (CALL\_INSN 인 것처럼 나타난) insn 가 sibling call 이면 1.

```
#define INSN_ANNULLED_BRANCH_P(INSN) ((INSN)->unchanging)
```

만약 insn 가 무조건적으로 그것의 delay slot 들을 실행시켜서는 안되는 branch 일 경우 1, 즉, 그것이 annulled branch 일 때.

```
#define INSN_DEAD_CODE_P(INSN) ((INSN)->in_struct)
```

Insn 가 dead code 일 경우 1. Dead-code elimination phase 상에서만 유효.

```
#define INSN_FROM_TARGET_P(INSN) ((INSN)->in_struct)
```

만약 insn 가 delay slot 내에 있고 그 branch 의 target 으로부터 기인했다면 1. 만약 branch insn 가 INSN\_ANNULLED\_BRANCH\_P 설정을 가지고 있다면, 이 insn 는 branch 가 받아들여졌을 때만 실행되어야 한다. 이 bit 가 깨끗한 annulled branche 들에서, 해당 insn 는 오직 branch 가 받아들여 지지 않았을 때만 실행되어야 한다.

```
#define ADDR_DIFF_VEC_FLAGS(RTX) XOADVFLAGS(RTX, 4)
```

아직 정확한 설명이 존재하지 않음.

```
#define CSELIB_VAL_PTR(RTX) XOCSELIB(RTX, 0)
```

아직 정확한 설명이 존재하지 않음.

```
#define REG_NOTES(INSN) XEXP(INSN, 6)
```

이 insn 가 여러 REG 들에게 무엇을 어떻게 영향을 미치는지에 대한 note 들의 list 를 가지고 있다. 이것은 EXPR\_LIST rtx 들의 chain 이며, 두번째 operand 는 chain pointer 이고 첫번째 operand 는 설명된 REG 이다. EXPR\_LIST 의 mode field 는 실제 machine mode 가 아닌 enum reg\_note 에서의 값을 포함하고 있다.

```
#define REG_NOTE_KIND(LINK) ((enum reg_note) GET_MODE (LINK))
```

```
#define PUT_REG_NOTE_KIND(LINK, KIND) \
  PUT_MODE (LINK, (enum machine_mode) (KIND))
```

EXPR\_LIST 내 reg-note kind 를 추출하고 넣는 macro 들을 정의한다.

```
#define CALL_INSN_FUNCTION_USAGE(INSN) XEXP(INSN, 7)
```

이 field 는 오직 CALL\_INSN 들상에서 나타난다. 이것은 USE 와 CLOBBER 표현식들의 EXPR\_LIST 의 chain 을 잡고 있다.

- USE 표현식들은 함수로 건네진 argument 들로 채워진 register 들을 목록화하고 있다.
- CLOBBER 표현식들은 이 CALL\_INSN 에 의해 명시적으로 clobber 된 register 들을 문서화한다.

Pseudo register 들은 이 list 에서 언급될 수 없다.

```
#define CODE_LABEL_NUMBER(INSN) XINT (INSN, 5)
```

Code-label 의 label-number. 어셈블러 label 은 'L' 로부터 만들어지며 label-number 는 십진수로 출력된다. Label number 들은 컴파일에서 유일하다.

```
#define NOTE_SOURCE_FILE(INSN) XCSTR (INSN, 3, NOTE)
```

```
#define NOTE_BLOCK(INSN) XCTREE (INSN, 3, NOTE)
```

```
#define NOTE_EH_HANDLER(INSN) XCINT (INSN, 3, NOTE)
```

```
#define NOTE_RANGE_INFO(INSN) XCEXP (INSN, 3, NOTE)
```

```
#define NOTE_LIVE_INFO(INSN) XCEXP (INSN, 3, NOTE)
```

```
#define NOTE_BASIC_BLOCK(INSN) XCBDEF (INSN, 3, NOTE)
```

```
#define NOTE_EXPECTED_VALUE(INSN) XCEXP (INSN, 3, NOTE)
```

Line number 인 NOTE 에서, 이것은 line 이 어디인지에 대한 파일 이름관련 문자열이다. 우리는 NOTE\_INSN\_BLOCK\_BEG 와 NOTE\_INSN\_BLOCK\_END note 들 내에서 임시적으로 block number 들을 기록하기 위해 같은 field 를 사용한다. (만약 우리가 block number 를 위해 다른 macro 를 사용한다면 int 들과 pointer 들 사이에 엄청 많은 cast 들을 피한다.) NOTE\_INSN\_RANGE\_START,END 와 NOTE\_INSN\_LIVE note 들은 그 field 내 rtx 로써 그들의 정보를 기록한다.

```
#define NOTE_LINE_NUMBER(INSN) XCINT (INSN, 4, NOTE)
```

NOTE 가 줄번호인 것에서, 이것은 줄번호이다. NOTE 의 다른 종류들에서는 여기는 음수로 구분되어 진다.

```
#define NOTE_INSN_BASIC_BLOCK_P(INSN)          \
(GET_CODE (INSN) == NOTE                      \
 && NOTE_LINE_NUMBER (INSN) == NOTE_INSN_BASIC_BLOCK)
```

만약 INSN 가 basic block 의 시작 부분을 mark 하는 note 라면 0 이 아니다.

```
#define LABEL_NAME(RTX) XCSTR (RTX, 6, CODE_LABEL)
```

Input source code 내 explicit label 과 상응할 경우에 대한 label 의 이름.

```
#define LABEL_NUSES(RTX) XCINT (RTX, 3, CODE_LABEL)
```

jump.c 에서, 각 label 은 어떤 것을 가르키는 LABEL\_REF 들의 number 의 count 를 포함하고 있기 때문에, 사용되지 않은 label 들을 삭제될 수 있다.

```
#define LABEL_ALTERNATE_NAME(RTX) XCSTR (RTX, 7, CODE_LABEL)
```

Name 을 CODE\_LABEL 와 연합한다.

```
#define ADDRESSOF_REGNO(RTX) XCUINT (RTX, 1, ADDRESSOF)
```

이 ADDRESSOF 가 만들어졌던 원래 regno.

```
#define ADDRESSOF_DECL(RTX) XCTREE (RTX, 2, ADDRESSOF)
```

우리가 주소로 가졌던 register 내 변수.

```
#define JUMP_LABEL(INSN) XCEXP (INSN, 7, JUMP_INSN)
```

jump.c 에서, 각 JUMP\_INSN 는 jump 할 수 있는 label 을 가르킬 수 있으며 그래서 만약 JUMP\_INSN 가 삭제될 경우 label 의 LABEL\_NUSES 는 감소되어질 수 있고 label 이 삭제될 수도 있다.

```
#define LABEL_REFS(LABEL) XCEXP (LABEL, 4, CODE_LABEL)
```

이러한 basic block 들은 flow.c 에서 발견된다. 각 CODE\_LABEL 은 해당 label 로 jump 하는 모든 LABEL\_REF 들을 통과하는 chain 을 시작한다. 결국 chain 은 CODE\_LABEL 에서 결말을 낸다: 그것은 원형의 모습이다.

```
#define LABEL_NEXTREF(REF) XCEXP (REF, 1, LABEL_REF)
```

이것은 어떤 특정 label 로의 reference 들의 circular chain 이 어떻게 연결되었는지에 대한 LABEL\_REF 내 field 이다. 이 chain 은 flow.c 에서 설정된다.

```
#define CONTAINING_INSN(RTX) XCEXP (RTX, 2, LABEL_REF)
```

이러한 basci block 들은 flow.c 에서 찾아진다. 각 LABEL\_REF 는 이 field 를 가진 그것의 containing instruction 를 가르킨다.

```
#define REGNO(RTX) XCUINT (RTX, 0, REG)
#define ORIGINAL_REGNO(RTX) XOUINT (RTX, 1)
```

REG rtx 용, REGNO 는 register number 를 추출한다. ORIGINAL\_REGNO 는 register 가 원래 가지고 있었던 number 를 잡고 있다; Hard reg 로 변화한 pseudo register 에 대해 이것은 이전 pseudo register number 를 잡고 있다.

```
#define REG_FUNCTION_VALUE_P(RTX) ((RTX)->integrated)
```

REG rtx 용, REG\_FUNCTION\_VALUE\_P 는 만약 reg 가 현재 함수의 return value 일 경우 0 이 아니다.

```
#define REG_USERVAR_P(RTX) ((RTX)->volatil)
```

REG rtx 에서 만약 user 에 의해 선언된 변수와 상응하는 것일 경우 1.

```
#define REG_POINTER(RTX) ((RTX)->frame_related)
```

만약 register 가 pointer 라면 REG rtx 에서는 값이 1.

```
#define HARD_REGISTER_P(REG) (HARD_REGISTER_NUM_P (REGNO (REG)))
```

만약 주어진 register REG 가 hard register 와 상응할 경우 1.

```
#define HARD_REGISTER_NUM_P(REG_NO) ((REG_NO) < FIRST_PSEUDO_REGISTER)
```

만약 주어진 register nubmer REG\_NO 가 hard register 와 상응할 경우 1.

```
#define INTVAL(RTX) XCWINT(RTX, 0, CONST_INT)
```

CONST\_INT rtx 용, INTVAL 는 정수를 추출한다.

```
#define CONST_DOUBLE_LOW(r) XCWINT (r, 1, CONST_DOUBLE)
#define CONST_DOUBLE_HIGH(r) XCWINT (r, 2, CONST_DOUBLE)
```

CONST\_DOUBLE 용: 보통, 값을 가지고 있는 두 int 들.

DImode 용, 모두 존재한다; CONST\_DOUBLE\_LOW 는 low-order word 이고 ...\_HIGH 는 high-order 이다.

Float 용, int 들의 갯수가 변하며, CONST\_DOUBLE\_LOW 는 \*메모리 내에서\* 처음 와야하는 것이다. 그래서 int 들의 배열의 주소로써 &CONST\_DOUBLE\_LOW(r) 를 사용한다.

```
#define CONST_DOUBLE_CHAIN(r) XCEXP (r, 0, CONST_DOUBLE)
```

현재 함수내에 사용중인 모든 CONST\_DOUBLE 들의 chain 을 위한 link.

```
#define CONST_VECTOR_ELT(RTX, N) XCVECEXP (RTX, 0, N, CONST_VECTOR)
```

CONST\_VECTOR 용, Element #n 를 반환한다.

```
#define CONST_VECTOR_NUNITS(RTX) XCVECLEN (RTX, 0, CONST_VECTOR)
```

CONST\_VECTOR 용, Vector 내 element 들의 number 를 반환한다.

```
#define SUBREG_REG(RTX) XCEXP (RTX, 0, SUBREG)
#define SUBREG_BYTE(RTX) XCUINT (RTX, 1, SUBREG)
```

SUBREG rtx 용, SUBREG\_REG 는 우리가 값의 subreg 를 원할 경우 그 값을 추출한다. SUBREG\_BYTE 는 byte-number 를 추출한다.

```
#define SUBREG_PROMOTED_VAR_P(RTX) ((RTX)->in_struct)
#define SUBREG_PROMOTED_UNSIGNED_P(RTX) ((RTX)->unchanging)
```

만약 SUBREG\_REG 내 포함되어 있는 REG 가 이미 SUBREG 의 mode 부터 해당 reg 의 mode 까지 sign- 혹은 zero-extended 임이 알려졌을 경우 1. SUBREG\_PROMOTED\_UNSIGNED\_P 는 extension 의 signedness 를 준다.

LHS 로써 사용될 경우, is 는 이 extension 이 반드시 SUBREG\_REG 로 assign 할때 수행되어 여 함을 의미한다.

```
#define ASM_OPERANDS_TEMPLATE(RTX) XCSTR (RTX, 0, ASM_OPERANDS)
#define ASM_OPERANDS_OUTPUT_CONSTRAINT(RTX) XCSTR (RTX, 1, ASM_OPERANDS)
#define ASM_OPERANDS_OUTPUT_IDX(RTX) XCINT (RTX, 2, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT_VEC(RTX) XCVEC (RTX, 3, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT_CONSTRAINT_VEC(RTX) XCVEC (RTX, 4, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT(RTX, N) XCVCEXP (RTX, 3, N, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT_LENGTH(RTX) XCVECLEN (RTX, 3, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT_CONSTRAINT_EXP(RTX, N) \
  XCVCEXP (RTX, 4, N, ASM_OPERANDS)
#define ASM_OPERANDS_INPUT_CONSTRAINT(RTX, N) \
  XSTR (XCVCEXP (RTX, 4, N, ASM_OPERANDS), 0)
#define ASM_OPERANDS_INPUT_MODE(RTX, N) \
  GET_MODE (XCVCEXP (RTX, 4, N, ASM_OPERANDS))
#define ASM_OPERANDS_SOURCE_FILE(RTX) XCSTR (RTX, 5, ASM_OPERANDS)
#define ASM_OPERANDS_SOURCE_LINE(RTX) XCINT (RTX, 6, ASM_OPERANDS)
```

ASM\_OPERANDS rtx 의 여러 component 들에 대한 접근.

```
#define MEM_KEEP_ALIAS_SET_P(RTX) ((RTX)->jump)
```

MEM RTX 용, 만약 우리가 component 에 접근하였을 때 변화하지 않은 이 mem 에 대해 alias set 를 유지해야 할 경우 1. 예로써 우리가 이미 aggregate 의 non-addressable component 내에 있을 경우 1 로 설정한다.

```
#define MEM_VOLATILE_P(RTX) ((RTX)->volatil)
```

MEM rtx 용, 만약 volatile reference 일 경우 1. 또한 ASM\_OPERANDS rtx 에서도.

```
#define MEM_IN_STRUCT_P(RTX) ((RTX)->in_struct)
```

MEM rtx 용, 만약 그것이 aggregate 를 참조하거나 aggregate 자신 혹은 aggregate 의 field 로 참조할 경우 1. 만약 0 일 경우 RTX 는 그러한 reference 일수도 아닐 수도 있다.

```
#define MEM_SCALAR_P(RTX) ((RTX)->frame_related)
```

MEM rtx 용, 만약 이것이 scalar 를 참조할 경우 1. 만약 0 일 경우 RTX 는 scalar 를 참조할 수도 안 할 수도 있다.

```

#define MEM_SET_IN_STRUCT_P(RTX, VAL)          \
do {                                          \
  if (VAL)                                   \
  {                                          \
    MEM_IN_STRUCT_P (RTX) = 1;             \
    MEM_SCALAR_P (RTX) = 0;                \
  }                                          \
else                                         \
  {                                          \
    MEM_IN_STRUCT_P (RTX) = 0;             \
    MEM_SCALAR_P (RTX) = 1;                \
  }                                          \
} while (0)

```

만약 VAL 가 0 이 아니라면, RTX 내 MEM\_IN\_STRUCT\_P 를 설정하고 MEM\_SCALAR\_P 를 깨끗히 한다. 그렇지 않을 경우 반대로 작동한다. 이것을 사용할 때는 당신이 MEM 가 구조체 내에 있거나 scalar 임을 확실히 알고 있을 경우에만 이 macro 를 사용하라. VAL 는 오직 한번 검사된다.

```

#define MEM_ATTRS(RTX) XOMEMATTR (RTX, 1)

```

Memory attribute block. 우리는 block 내의 각각의 값을 위한 access macro 들을 제공하며 지정된 것이 없다면 기본값들을 제공합니다.

```

#define MEM_ALIAS_SET(RTX) (MEM_ATTRS (RTX) == 0 ? 0 : MEM_ATTRS (RTX)->alias)

```

MEM 용, alias set. 만약 0 이면 이 MEM 은 아직 어느 alias set 내에 속해 있지 아니며 어떤 것도 alias 할 수 있습니다. 그렇지 않을 경우, MEM 은 단지 같은 alias set 내에 있는 MEM 들만 alias 할 수 있습니다 이 값은 .language-dependent manner 에서 front-end 에서 설정되며 back-end 에서 변경되어서는 안됩니다. 이렇게 설정된 number 들은 0 인지 혹은 값이 같은지를 비교할 때 사용됩니다; 그외에 다른 의미를 가지고 있지 않습니다. 몇몇 front-end 에서는 이 number 들은 type 들 혹은 다른 language-level entity 들과 함께 작동할 수 있지만 그것은 필요하지 않으며 back-end 는 그러한 assumption 들을 만들지 않습니다.

```

#define MEM_EXPR(RTX) (MEM_ATTRS (RTX) == 0 ? 0 : MEM_ATTRS (RTX)->expr)

```

MEM rtx 용, 만약 DECL 의 부분을 참조하는 것으로 알려질 경우 그것을 참조하는 것으로 알려진 decl. 그것은 COMPONENT\_REF 일 수 있다.

```

#define MEM_OFFSET(RTX) (MEM_ATTRS (RTX) == 0 ? 0 : MEM_ATTRS (RTX)->offset)

```

MEM rtx 용, 만약 RTX 가 항상 CONST\_INT 인 것으로 알려졌을 경우, MEM\_EXPR 의 시작 부분으로 부터의 offset.

```

#define MEM_SIZE(RTX)                                \
(MEM_ATTRS (RTX) != 0 ? MEM_ATTRS (RTX)->size      \
 : GET_MODE (RTX) != BLKmode ? GEN_INT (GET_MODE_SIZE (GET_MODE (RTX))) \
 : 0)

```

MEM rtx 용, 만약 RTX 가 항상 CONST\_INT 인 것으로 알려졌을 경우, MEM 의 byte 들로써의 크기.

```

#define MEM_ALIGN(RTX)                                \
(MEM_ATTRS (RTX) != 0 ? MEM_ATTRS (RTX)->align     \
 : (STRICT_ALIGNMENT && GET_MODE (RTX) != BLKmode  \
 ? GET_MODE_ALIGNMENT (GET_MODE (RTX)) : BITS_PER_UNIT))

```

MEM rtx 용, 비트로 표현한 alignment. 우리는 mode 의 alignment 를 STRICT\_ALIGNMENT 일때 기본값으로써 사용할 수 있다.

```
#define MEM_COPY_ATTRIBUTES(LHS, RHS) \
(MEM_VOLATILE_P (LHS) = MEM_VOLATILE_P (RHS), \
MEM_IN_STRUCT_P (LHS) = MEM_IN_STRUCT_P (RHS), \
MEM_SCALAR_P (LHS) = MEM_SCALAR_P (RHS), \
RTX_UNCHANGING_P (LHS) = RTX_UNCHANGING_P (RHS), \
MEM_KEEP_ALIAS_SET_P (LHS) = MEM_KEEP_ALIAS_SET_P (RHS), \
MEM_ATTRS (LHS) = MEM_ATTRS (RHS))
```

RHS 부터 LHS 까지 memory location 들에 적용하는 attribute 들을 복사한다.

```
#define LABEL_OUTSIDE_LOOP_P(RTX) ((RTX)->in_struct)
```

LABEL\_REF 용, 1 은 이 reference 가 reference 를 포함하는 loop 밖의 label 로 향한것임을 의미한다.

```
#define LABEL_REF_NONLOCAL_P(RTX) ((RTX)->volatil)
```

LABEL\_REF 용, 1 은 이것이 nonlocal label 위한 것임을 의미한다. 그렇지 않으면 REG\_LABEL note 를 위한 EXPR\_LIST 내이던가.

```
#define LABEL_PRESERVE_P(RTX) ((RTX)->in_struct)
```

CODE\_LABEL 용, 1 은 항상 이 label 이 필요한 것으로 고려됨을 의미한다.

```
#define REG_LOOP_TEST_P(RTX) ((RTX)->in_struct)
```

REG, 1 은 register 가 loop 의 exit test 에서만 사용됨을 의미한다.

```
#define SCHED_GROUP_P(INSN) ((INSN)->in_struct)
```

sched 동안, insn 에 대해, insn 가 preceding insn 와 함께 반드시 스케줄 되어져야 한다면 1 을 의미한다.

```
#define LINK_COST_ZERO(X) ((X)->jump)
```

```
#define LINK_COST_FREE(X) ((X)->call)
```

sched 동안, insn 의 LOG\_LINKS 에 대해, 이것들은 dependence link 의 adjusted cost 를 cache 한다. instruction 를 실행하기 위한 cost 는 어떻게 result 들이 사용되었느냐에 기반하여 변화할 것이다. LINK\_COST\_ZERO 는 link 를 통한 cost 가 변화하거나 변화되지 않을 때 1 이다. (즉, link 는 zero additional cost 를 가지고 있다.) LINK\_COST\_FREE 는 link 를 통한 cost 가 0 일 때 (즉, link 가 cost free 를 만들 때) 1 이다. 다른 경우, cost 를 위한 adjustment 가 그것이 필요할 때 매번 재계산되어진다.

```
#define SET_DEST(RTX) XC2EXP(RTX, 0, SET, CLOBBER)
```

```
#define SET_SRC(RTX) XCEXP(RTX, 1, SET)
```

```
#define SET_IS_RETURN_P(RTX) ((RTX)->jump)
```

SET rtx 용, SET\_DEST 는 설정될 장소이고 SET\_SRC 는 설정할 값이다.

```
#define TRAP_CONDITION(RTX) XCEXP (RTX, 0, TRAP_IF)
```

```
#define TRAP_CODE(RTX) XCEXP (RTX, 1, TRAP_IF)
```

TRAP\_IF rtx 용, TRAP\_CONDITION 는 표현식이다.

```

#define COND_EXEC_TEST(RTX) XCEXP (RTX, 0, COND_EXEC)
#define COND_EXEC_CODE(RTX) XCEXP (RTX, 1, COND_EXEC)

COND_EXEC rtx 용, COND_EXEC_TEST 는 조건적으로 code 를 실행할 기반 을 위한 condi-
tion 이고, COND_EXEC_CODE 는 만약 condition 이 true 일 경우 실행할 code 이다.

#define CONSTANT_POOL_ADDRESS_P(RTX) ((RTX)->unchanging)

SYMBOL_REF 에서 만약 이것이 이 함수의 constants pool 을 주소화할 경우 1.

#define STRING_POOL_ADDRESS_P(RTX) ((RTX)->frame_related)

SYMBOL_REF 에서 만약 이것이 이 함수의 string constant pool 을 주소화할 경우 1.

#define SYMBOL_REF_FLAG(RTX) ((RTX)->volatile)

SYMBOL_REF 에 대한 machine-specific 목절들을 위한 flag.

#define SYMBOL_REF_USED(RTX) ((RTX)->used)

값이 1 일 경우 SYMBOL_REF 가 emit_library_call 에서 library function 임을 의미한다.

#define SYMBOL_REF_WEAK(RTX) ((RTX)->integrated)

1 은 SYMBOL_REF 가 weak 임을 의미합니다.

if (defined (HAVE_PRE_INCREMENT) || defined (HAVE_PRE_DECREMENT)
    || defined (HAVE_POST_INCREMENT) || defined (HAVE_POST_DECREMENT))
#define FIND_REG_INC_NOTE(INSN, REG) \
    ((REG) != NULL_RTX && REG_P ((REG)) \
     ? find_regno_note ((INSN), REG_INC, REGNO (REG)) \
     : find_reg_note ((INSN), REG_INC, (REG)))
#else
#define FIND_REG_INC_NOTE(INSN, REG) 0
#endif

REG_INC note 들을 찾는 macro 를 정의한다. 하지만 그들이 전혀 존재하지 않는 machine 상
에서는 시간을 절약한다.

RANGE_INFO 를 위한 accessor 들.

#define RANGE_INFO_NOTE_START(INSN) XCEXP (INSN, 0, RANGE_INFO)

RANGE_START,END note 들에 대해 RANGE_START note 를 반환한다.

#define RANGE_INFO_NOTE_END(INSN) XCEXP (INSN, 1, RANGE_INFO)

RANGE_START,END note 들에 대해 RANGE_START note 를 반환한다.

#define RANGE_INFO_REGS(INSN) XCVEC (INSN, 2, RANGE_INFO)
#define RANGE_INFO_REGS_REG(INSN, N) XCVECEXP (INSN, 2, N, RANGE_INFO)
#define RANGE_INFO_NUM_REGS(INSN) XCVECLEN (INSN, 2, RANGE_INFO)

RANGE_START,END note 들에 대해, range 에서 사용된 register 들을 포함하는 vector 를 반
환한다.

#define RANGE_INFO_NCALLS(INSN) XCINT (INSN, 3, RANGE_INFO)

```

RANGE\_START,END note 들에 대해, range 내 call 들의 number.

```
#define RANGE_INFO_NINSNS(INSN) XCINT (INSN, 4, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 내 insn 들의 number.

```
#define RANGE_INFO_UNIQUE(INSN) XCINT (INSN, 5, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, 이 range 를 인식하기 위한 유일한 #.

```
#define RANGE_INFO_BB_START(INSN) XCINT (INSN, 6, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 가 시작하는 basic block #.

```
#define RANGE_INFO_BB_END(INSN) XCINT (INSN, 7, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 가 끝나는 basic block #.

```
#define RANGE_INFO_LOOP_DEPTH(INSN) XCINT (INSN, 8, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 가 속한 것의 loop depth.

```
#define RANGE_INFO_LIVE_START(INSN) XCBITMAP (INSN, 9, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 의 시작에서 live register 들의 bitmap.

```
#define RANGE_INFO_LIVE_END(INSN) XCBITMAP (INSN, 10, RANGE_INFO)
```

RANGE\_START,END note 들에 대해, range 의 끝에서 live register 들의 bitmap.

```
#define RANGE_INFO_MARKER_START(INSN) XCINT (INSN, 11, RANGE_INFO)
```

RANGE\_START note 들에 대해, range 의 시작 부분의 marker #.

```
#define RANGE_INFO_MARKER_END(INSN) XCINT (INSN, 12, RANGE_INFO)
```

RANGE\_START note 들에 대해, range 의 끝 부분의 marker #.

```
#define RANGE_REG_PSEUDO(INSN,N) XCINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 0, REG)
```

Live rane note 를 위한 원래 pseudo register #.

```
#define RANGE_REG_COPY(INSN,N) XCINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 1, REG)
```

Pseudo register # original register 가 복사되었거나, -1 이다.

```
#define RANGE_REG_REFS(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 2)
```

Live range note 내 register 가 얼마나 많이 참조되었는지.

```
#define RANGE_REG_SETS(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 3)
```

Live range note 내 register 가 얼마나 많이 설정되었는지.

```
#define RANGE_REG_DEATHS(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 4)
```

Live range note 내 register 가 얼마나 많이 죽었는지.

```
#define RANGE_REG_COPY_FLAGS(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 5)
```

Range 의 시작 부분의 range register 로 원래 값이 복사될 필요가 있는지 없는지.

```
#define RANGE_REG_LIVE_LENGTH(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 6)
```

Register copy 가 live over 인 insn 들의 #.

```
#define RANGE_REG_N_CALLS(INSN,N) XINT (XCVECEXP (INSN, 2, N, RANGE_INFO), 7)
```

Register copy 가 live over 인 call 들의 #.

```
#define RANGE_REG_SYMBOL_NODE(INSN,N) XTREE (XCVECEXP (INSN, 2, N, RANGE_INFO), 8)
```

만약 register 가 user defined variable 일 경우 declaration 의 DECL\_NODE pointer

```
#define RANGE_REG_BLOCK_NODE(INSN,N) XTREE (XCVECEXP (INSN, 2, N, RANGE_INFO), 9)
```

Register 가 user defined variable 일 경우 변수가 선언되어 있는 block 으로의 BLOCK\_NODE pointer.

```
#define RANGE_VAR_LIST(INSN) (XEXP (INSN, 0))
```

변수가 속한 distinct range 들의 EXPR\_LIST.

```
#define RANGE_VAR_BLOCK(INSN) (XTREE (INSN, 1))
```

변수가 선언된 block.

```
#define RANGE_VAR_NUM(INSN) (XINT (INSN, 2))
```

변수가 선언된 distinct range 들의 #.

```
#define RANGE_LIVE_BITMAP(INSN) (XBITMAP (INSN, 0))
```

NOTE\_INSN\_LIVE note 용, 현재 live 인 register 들.

```
#define RANGE_LIVE_ORIG_BLOCK(INSN) (XINT (INSN, 1))
```

NOTE\_INSN\_LIVE note 용, 원래 basic block number.

```
#define PHI_NODE_P(X) \
((X) && GET_CODE (X) == INSN \
&& GET_CODE (PATTERN (X)) == SET \
&& GET_CODE (SET_SRC (PATTERN (X))) == PHI)
```

Insn 가 PHI node 인지를 결정한다.

```
#define CONST0_RTX(MODE) (const_tiny_rtx[0][ (int) (MODE)])
```

mode MODE 내에서의 상수 0 rtx 를 반환합니다. 정수 mode 들은 VOIDmode 처럼 취급됩니다.

```
#define CONST1_RTX(MODE) (const_tiny_rtx[1][ (int) (MODE)])
```

```
#define CONST2_RTX(MODE) (const_tiny_rtx[2][ (int) (MODE)])
```

위와 같은 상수 1 과 2 를 위한 것.

```
#ifndef HARD_FRAME_POINTER_REGNUM
```

```
#define HARD_FRAME_POINTER_REGNUM FRAME_POINTER_REGNUM
```

```
#endif
```

만약 HARD\_FRAME\_POINTER\_REGNUM 가 정의되어 있다면 special dummy reg 가 frame pointer 를 표현하기 위해 사용됩니다. 이것은 hard frame pointer 와 automatic variable 들이 register allocation 가 끝날 때까지 결정될 수 없는 amount 에 의해 나뉘어져 있기 때문이다. 우리는 이러한 경우 ELIMINABLE\_REGS 가 정의될 것이라고 가정할 수 있는데, 이 행동은 FRAME\_POINTER\_REGNUM 를 HARD\_FRAME\_POINTER\_REGNUM 로 제거하는 것이 될 수 있다.

```
#define pc_rtx                (global_rtl[GR_PC])
#define cc0_rtx              (global_rtl[GR_CC0])
```

직접적으로 사용할 수 있게 대체되는 표준 rtx 의 piece 들.

```
#define stack_pointer_rtx    (global_rtl[GR_STACK_POINTER])
#define frame_pointer_rtx   (global_rtl[GR_FRAME_POINTER])
#define hard_frame_pointer_rtx (global_rtl[GR_HARD_FRAME_POINTER])
#define arg_pointer_rtx     (global_rtl[GR_ARG_POINTER])
```

hard reg 들을 확정하는 모든 참조문들인데, (가능할 때) 그들 내에 pseudo reg 들을 할당함으로써 생성된 이것은 이러한 unique rtx object 들을 가진다.

```
#define GEN_INT(N)  gen_rtx_CONST_INT (VOIDmode, (HOST_WIDE_INT) (N))
```

Prototype 이 있던 없던 같은 결과를 얻는 다는 것을 확실히 하기 위해 여기 cast 를 필요로 합니다.

```
#define FIRST_VIRTUAL_REGISTER (FIRST_PSEUDO_REGISTER)
```

Virtual register 들은 RTL 생성동안 stack frame 내에서의 실제 위치를 RTL 생성이 완료될때 까지 알수 없을 경우 그 위치를 참고하는데 사용됩니다. routine instantiate\_virtual\_regs 는 이것을 적당한 값으로 대체 하는데, 이것들은 보통 frame,arg,stack\_pointer\_rtx 에 상수를 더한 것이다.

```
#define virtual_incoming_args_rtx (global_rtl[GR_VIRTUAL_INCOMING_ARGS])
#define VIRTUAL_INCOMING_ARGS_REGNUM (FIRST_VIRTUAL_REGISTER)
```

Caller 에 의해 전해진다고 가정되겠지만 caller 에 의해서든 callee 에 의해서든 스택상으로 건내지는 incoming argument 들의 첫번째 word 를 가르킵니다.

```
#define virtual_stack_vars_rtx (global_rtl[GR_VIRTUAL_STACK_ARGS])
#define VIRTUAL_STACK_VARS_REGNUM ((FIRST_VIRTUAL_REGISTER) + 1)
```

만약 FRAME\_GROWS\_DOWNWARD 라면 이것은 스택상의 처음 변수 바로 윗 부분을 가르킵니다. 그렇지 않다면 스택상의 처음 변수를 가르키게 됩니다.

```
#define virtual_stack_dynamic_rtx (global_rtl[GR_VIRTUAL_STACK_DYNAMIC])
#define VIRTUAL_STACK_DYNAMIC_REGNUM ((FIRST_VIRTUAL_REGISTER) + 2)
```

이것은 Stack pointer 가 요청된 양만큼 맞추어진 바로 다음에 스택상에 동적으로-할당된 메모리의 위치를 가르킵니다.

```
#define virtual_outgoing_args_rtx (global_rtl[GR_VIRTUAL_OUTGOING_ARGS])
#define VIRTUAL_OUTGOING_ARGS_REGNUM ((FIRST_VIRTUAL_REGISTER) + 3)
```

Stack 이 pre-pushed (push insn 들을 사용하여 push 된 argument 들은 항상 sp 를 사용함) 될 때, 어떤 outgoing argument 들이 쓰여져야 하는지에 대해서 stack 내 위치를 이것이 가르키고 있다.

```
#define virtual_cfa_rtx                (global_rtl[GR_VIRTUAL_CFA])
#define VIRTUAL_CFA_REGNUM            ((FIRST_VIRTUAL_REGISTER) + 4)
```

이것은 함수의 Canonical Frame Address 를 가르킵니다. 이것은 INCOMING\_FRAME\_SP\_OFFSET 에 의해 생성된 CFA 와 연동하여야 하지만 단순함을 위해 arg pointer 와 상대적으로 계산됩니다; frame\_pointer 도 stack pointer 도 reload 후 까지 CFA 와 상대적으로 고정 (fix) 되어 있을 이유는 없습니다.

```
#define LAST_VIRTUAL_REGISTER          ((FIRST_VIRTUAL_REGISTER) + 4)
```

자세한 설명이 없음.

```
#define REGNO_PTR_FRAME_P(REGNUM)    \
((REGNUM) == STACK_POINTER_REGNUM    \
 || (REGNUM) == FRAME_POINTER_REGNUM \
 || (REGNUM) == HARD_FRAME_POINTER_REGNUM \
 || (REGNUM) == ARG_POINTER_REGNUM   \
 || ((REGNUM) >= FIRST_VIRTUAL_REGISTER \
 && (REGNUM) <= LAST_VIRTUAL_REGISTER))
```

만약 REGNUM 가 stack frame 내 pointer 일 경우 0 이 아닌 값.

```
#define INVALID_REGNUM                (~(unsigned int) 0)
```

REGNUM 는 절대로 INSN stream 에 나타나지 않는다.

```
#define COSTS_N_INSNS(N) ((N) * 4)
```

아직 자세한 설명이 없음.

```
#define MAX_COST INT_MAX
```

rtl expression 의 최대 cost. 이 값은 어떠한 환경하에서 이 cost 를 가지는 rtx 를 하지 않지 않겠다는 특별한 의미를 가지고 있다.

## 제 6 절 RTL 구현을 위한 함수들

아래 RTL 관련 함수들의 설명 부분에 모두 대문자로 구성되는 영어 단어의 경우, 실제 함수의 인자값인 경우이다. 보면서 의심이 되시는 분들은 반드시 함수의 원형을 찾아 보시기 바란다.

### 6.1 alias.c

```
void clear_reg_alias_info          PARAMS ((rtx));
```

Register 에 대한 alias info 를 깨끗히 한다. 이것은 만약 RTL transformation 이 register 의 값을 변경하였을 때 사용된다. 이것은 AUTO\_INC\_DEC 최적화들로 flow 에서 사용된다. 우리는 reg\_base\_value 를 깨끗히 할 필요가 없는데, 그것은 flow 가 단지 offset 만 변경하기 때문이다.

```
rtx canon_rtx                      PARAMS ((rtx));
```

View alias analysis 의 관점에서 본 X 의 canonical version 를 반환한다. (예를 들어 만약 X 가 MEM 이고 이것의 address 가 register 이며, register 가 알려지 값 (SYMBOL\_REF 를 말함) 을 가지고 있다면, MEM 의 address 가 SYMBOL\_REF 인 MEM 이 반환된다.)

```
int true_dependence          PARAMS ((rtx, enum machine_mode, rtx,
                                     int (*)(rtx, int)));
```

True dependence: X 는 MEM 이 자리를 잡은 곳에 저장된 후 읽혀진다.

```
rtx get_addr                PARAMS ((rtx));
```

Address X 를 우리가 사용할 수 있는 것으로 변환한다. 이것은 그것이 값이 아닐 경우 변화하지 않은 것을 변환함으로써 이루어지는데 후자의 경우 우리는 좀 더 유용한 rtx 를 얻기 위해 cselib 를 호출한다.

```
int canon_true_dependence   PARAMS ((rtx, enum machine_mode, rtx,
                                     rtx, int (*)(rtx, int)));
```

Canonical true dependence: X 는 MEM 가 놓여진 장소에 저장된 후 읽혀진다. MEM 을 가정하는 true\_dependence 의 변종은 이미 canonicalize 되어졌다. (그래서 우리는 더이상 여기서 그것을 할 수 없다.) mem\_addr argument 가 더해졌으며, 그로 인해 true\_dependence 는 canonicalize 하기 전에 이 값을 계산한다.

```
int read_dependence        PARAMS ((rtx, rtx));
```

Read dependence: X 는 MEM 가 놓여진 장소에 저장된 후 읽혀진다. 양쪽 read 들이 volatile 일 경우에만 여기서 어떤 dependence 가 될 수 있다.

```
int anti_dependence        PARAMS ((rtx, rtx));
```

Anti dependence: X 는 MEM 가 자리잡은 곳에서 읽은 후 쓰여진다.

```
int output_dependence      PARAMS ((rtx, rtx));
```

Output dependence: X 는 MEM 가 자리잡은 곳에서 저장된 후 쓰여진다.

```
void mark_constant_function PARAMS ((void));
```

만약 상수 (constant) 라면 함수를 mark 합니다.

```
void init_alias_once       PARAMS ((void));
```

아직 정확한 설명이 없음.

```
void init_alias_analysis   PARAMS ((void));
```

Aliasing machinery 를 초기화합니다. REG\_KNOWN\_VALUE 배열을 초기화합니다.

```
void end_alias_analysis    PARAMS ((void));
```

아직 정확한 설명이 없음.

```
rtx addr_side_effect_eval  PARAMS ((rtx, int, int));
```

SIZE 가 memory reference 의 바이트로의 크기인 ADDR 로의 (N\_REFS + 1) 번째 memory reference 의 address 를 반환한다. 만약 ADDR 가 memory reference 에 의해 수정되지 않았다면, ADDR 가 반환된다.

## 6.2 builtins.c

```
rtx expand_builtin_expect_jump PARAMS ((tree, rtx, rtx));
```

expand\_builtin\_expect 와 비슷하지만 jump context 에서 이것을 한다. 이것은 만약 conditional 이 \_\_builtin\_expect 일 경우 do\_jump 에 의해 호출된다. jump 를 emit 하는 insn 들의 SEQUENCE 를 반환하거나 만약 우리가 \_\_builtin\_expect 를 최적화할 수 없다면 NULL 을 반환한다. 우리는 jump 시 이것을 최적화하는 것이 필요한데, PowerPC 와 같은 machine 들은 test 를 SCC operation 으로 바꾸지 않고, 0/1 에 대한 test 에 기반하여 jump 한다.

## 6.3 calls.c

```
void emit_library_call PARAMS ((rtx, enum libcall_type,
                                enum machine_mode, int,
                                ...));
```

Function FUN (SYMBOL\_REF rtx) 로 library call 을 output 한다. (NO\_QUEUE 가 0 일 때 queue 를 방출함) 이것은 mode OUTMODE 의 값을 위해서이며, NARGS 다른 argument 들을 가지고, alternating rtx value 들과 그들을 어떤 것으로 변환하기 위한 machine\_mode 들이 건네진다. rtx 값들은 이미 protect\_from\_queue 를 통해 반드시 건네졌었어야 한다. FN\_TYPE 은 'normal' call 들에 대해서 LCT\_NORMAL 혹은 'const' call 들에 대해서 LCT\_CONST, REG\_LIBCALL/REG\_RETVAL note 들로 둘러싸여져야 하는 'const' call 들에 대해서는 LCT\_CONST\_MAKE\_BLOCK, 여분의 (use (memory (scratch))) 를 가진 REG\_LIBCALL/REG\_RETVAL note 들로 둘러싸여져야 하는 'purep' call 들에 대해서는 LCT\_PURE\_MAKE\_BLOCK, library call 들의 다른 type 들에 대해서는 다른 LCT\_ value 여야 한다.

```
rtx emit_library_call_value PARAMS ((rtx, rtx, enum libcall_type,
                                    enum machine_mode, int,
                                    ...));
```

emit\_library\_call 과 비슷하지만, 나머지 argument, VALUE, 가 두번째에 오며 결과를 어디에 저장할지를 말한다. (만약 VALUE 가 0 이면, 이 함수는 값을 반환할 편리한 방법을 고른다.) 이 함수는 값이 발견된 곳에 대한 rtx 를 반환한다. 만약 VALUE 가 0 이 아닐 경우, VALUE 가 반환된다.

## 6.4 cfgrtl.c

```
rtx delete_insn PARAMS ((rtx));
```

그것을 patch out 함으로써 INSN 를 삭제한다. 다음 insn 를 반환한다.

```
void delete_insn_chain PARAMS ((rtx, rtx));
```

반드시 짝이어야 한다는 note 를 남기고, START 와 FINISH 사이 insn 들의 chain 을 unlink 한다.

## 6.5 combine.c

```
int combine_instructions PARAMS ((rtx, unsigned int));
```

Combiner 를 위한 main entry point. F 는 함수의 첫번째 insn 이다. NREGS 는 첫번째 unused pseudo-reg number 이다.

만약 combiner 가 indirect jump 명령어를 direct jump 로 바꿨다면 0 이 아닌 값을 반환한다.

```
unsigned int extended_count      PARAMS ((rtx, enum machine_mode, int));
```

MODE 의 signedness 가 UNSIGNEDP 에 의해 인식되는데, 이 MODE 내 quantity 로써 해석 될 시점에 X 내 존재하는 “extended” bit 들의 number 를 반환한다. Unsigned quantity 들을 위해, 이것은 high-order zero bit 들의 number 이다. Signed quantity 들을 위해, 이것은 sign bit minus 1 의 복사본들의 number 이다. 두 경우에서 이 함수는 “spare” bit 들의 number 를 반환한다. 예를 들어 만약 이 함수가 적어도 1 을 반환하기 위한 어떤 두 quantity 들을 더해 졌다면, 덧셈은 overflow 가 아님으로 알려진다.

이 함수는 combine 동안 호출되지 않는다면 항상 0 을 반환하는데, 이것은 반드시 define\_split 로부터 호출되어야 함을 강조한다.

```
rtx remove_death                PARAMS ((unsigned int, rtx));
```

INSN 의 dead registers list 로 부터 register number REGNO 를 제거한다.

만약 존재했을 경우에 대해 death 를 기록하는데 사용된 note 를 반환한다.

```
void dump_combine_stats        PARAMS ((FILE *));
```

아직 정확한 설명이 없음.

```
void dump_combine_total_stats  PARAMS ((FILE *));
```

아직 정확한 설명이 없음.

## 6.6 cse.c

```
rtx gen_lowpart_if_possible    PARAMS ((enum machine_mode, rtx));
```

X 가 fixed-point number 를 위한 rtx (예를 들면, MEM 혹은 REG, SUBREG) 임을 가정하고 X 의 least-significant part 를 참조하는 rtx (MEM 혹은 SUBREG, CONST\_INT) 를 반환한다. MODE 는 반환한 X 의 part 가 얼마나 큰지를 표현한다.

만약 요청된 operation 이 수행될 수 없다면, 0 을 반환한다.

이것은 emit-rtl.c 내 gen\_lowpart 와 비슷하다.

```
int rtx_cost                   PARAMS ((rtx, enum rtx_code));
```

rtx X 계산의 cost 견적을 반환합니다. 하나의 사용은 cse 에서, 어떠한 표현식을 hash table 내에 유지할 것인지를 결정하는데. 다른 하나는 rtl generation 에서, 곱셈을 하는 가장 싼 방법을 고르는데. 후자의 다른 사용은 나중에 만들어 질 수 있다.

```
int address_cost              PARAMS ((rtx, enum machine_mode));
```

Address expression X 의 cost 를 반환합니다. 주어지는 X 는 적당히 형식화된 address reference 라고 예상합니다.

```
void delete_trivially_dead_insns PARAMS ((rtx, int, int));
```

모든 insn 들을 scan 하고 죽은 것을 삭제한다; 즉, 한번도 사용되지 않은 register 를 절약하거나 자신에서 register 를 복사하는 것들을 절약한다.

Cse 혹은 loop, 다른 최적화들에 의해 분명히 죽은 것으로 판명된 insn 들을 제거하는데 사용된다. Dead quantity 들을 위한 givs 들을 만들거나 loop 들 밖으로 dead invariant 들을 옮기려 하지 않기 때문에 loop 내의 heuristic 들을 증진시킨다. 컴파일의 남은 pass 들은 또한 속도가 향상될 것이다.

```
int cse_main          PARAMS ((rtx, int, int, FILE *));
```

함수의 명령어들을 상대로 cse 를 실행한다. F 는 첫번째 명령어이다. NREGS 는 명령어에서 사용되는 가장 높은 pseudo-reg number 에 1 을 더한 것이다.

AFTER\_LOOP 는 만약 이것이 loop 최적화뒤에 수행되는 cse call 일 경우 1 이다. (아직 -frerun-cse-after-loop 경우일 때만).

만약 jump\_optimize 가 conditional jump 명령어들내 간단화로 인해 재수행되어야 할 경우 1 을 반환한다.

```
void cse_end_of_basic_block    PARAMS ((rtx,
                                     struct cse_basic_block_data *,
                                     int, int, int));
```

INSN 의 basic block 의 끝을 찾아 그것의 range 와 block 의 모든 insn 들내 SET 들의 총 갯수, block 의 마지막 insn, branch path 를 반환한다.

branch path 는 어떠한 branch 들이 따라야 하는지를 가르킨다. 만약 0 이 아닌 path 크기가 지정된다면, block 는 재 scan 되어져야 하고 다른 branch 들의 set 이 주어질 것 이다. Branch path 는 오직 FLAG\_CSE\_FOLLOW\_JUMPS 혹은 FLAG\_CSE\_SKIP\_BLOCKS 가 0 이 아닐 때만 사용된다.

DATA 는 block 을 설명하는데 사용되는, 밑에서 선언된, 구조체 cse\_basic\_block\_data 로의 pointer 이다. 현재 block 에 관한 정보로 채워져 있고 어떤 경우, incoming 구조체의 branch path 는 output branch path 를 건설하는데 사용된다.

## 6.7 emit-rtl.c

```
rtx gen_rtx          PARAMS ((enum rtx_code,
                              enum machine_mode, ...));
```

```
rtx gen_rtx (code, mode, [element1, ..., elementn])
```

이 함수는 RTX code 가 <code> 로 지정되는 크기의 RTX 를 생성합니다. RTX 구조체는 인자 <element1> 에서 <elementn> 로 초기화됩니다. 이 형식은 지정된 RTX type 의 형식에 따라 해석 됩니다. (어떠한) rtx 와 결합된 특별한 machine mode 는 <mode> 내에 지정된다.

gen\_rtx 는 그것이 생성할 lisp-like rtx 을 조합하는 방식으로 답을 수 있다. 예를 들어 다음과 같은 rtx 구조는:

```
(plus:QI (mem:QI (reg:SI 1))
         (mem:QI (plusw:SI (reg:SI 2) (reg:SI 3))))
```

...다음과 같은 C code 로 생성될 수 있습니다. -

```
gen_rtx (PLUS, QImode,
        gen_rtx (MEM, QImode,
                gen_rtx (REG, SImode, 1)),
        gen_rtx (MEM, QImode,
                gen_rtx (PLUS, SImode,
                        gen_rtx (REG, SImode, 2),
                        gen_rtx (REG, SImode, 3))));
```

```
rtvec gen_rtvec      PARAMS ((int, ...));
```

```
gen_rtvec (n, [rt1, ..., rtn])
```

이 routine 는 rtvec 을 생성하고 그것의 argument 들인 rtx 를 가르키는 pointer 를 안에 저장한다.

```
rtx copy_insn_1          PARAMS ((rtx));
```

재귀적으로 copy\_insn 를 위한 rtx 의 새로운 복사본을 생성한다. 이 함수는 copy\_rtx 와 다른데, 이것은 SCRATCH 들과 ASM\_OPERAND 들을 제대로 다룬다. 보통, 이 함수는 직접적으로 사용되지 않고, front end 로써 copy\_insn 를 사용한다. 하지만, 당신은 먼저 copy\_insn 로 insn pattern 을 복사한후 SCRATCH 들을 포함하는 어떤 REG\_NOTE 들을 제대로 복사하기 위해 나중에 이 함수를 사용하면 된다.

```
rtx copy_insn          PARAMS ((rtx));
```

Rtx 의 새 복사본을 생성한다. 이 함수는 copy\_rtx 와 다른데, 이것은 SCRATCH 들과 ASM\_OPERAND 들을 제대로 다룬다. INSN 는 실제로 완전한 INSN 이면 안되는데, 이것은 단지 패턴일 수 있다.

```
rtx copy_rtx_if_shared  PARAMS ((rtx));
```

ORIG 를 사용중으로 mark 하고 만약 그것이 이미 사용중이었다면 그것의 복사본을 반환한다. 재귀적으로 하위 표현식들에 대해서도 똑같이 한다.

```
rtvec gen_rtvec_v      PARAMS ((int, rtx *));
```

설명이 존재하지 않음

```
rtx gen_reg_rtx        PARAMS ((enum machine_mode));
```

mode MODE 의 새 pseudo register 를 위한 REG rtx 를 생성한다. 이 pseudo 는 다음 순차적인 register number 가 할당된다.

```
rtx gen_label_rtx     PARAMS ((void));
```

Unique label number 를 가지는 새로이 생성된 CODE\_LABEL 를 반환한다.

```
int subreg_hard_regno  PARAMS ((rtx, int));
```

Hard register 의 SUBREG 인 X 의 final regno 를 반환한다.

```
rtx gen_lowpart_common PARAMS ((enum machine_mode, rtx));
```

Low-order bit 들의 number 가 MODE 에 의해 주어지는 X 의 몇몇 low-order bit 들을 나타내는 값을 반환한다. floating-point 와 fixed-point 값들 사이에 conversion 이 없으며, bit representation 이 반환됨을 알기 바란다.

이 함수는 아래 gen\_lowpart 와 cse.c, combine.c 내에 동일 기능을 하는 것들 사이의 공통된 점을 다룬다. 이것은 compilation 의 모든 시점에서 안전하게 다뤄질 수 있는 경우들만 묶은 것이다.

만약 이것이 우리가 다룰 수 있는 경우가 아니라면, 0 을 반환한다.

```
rtx gen_lowpart        PARAMS ((enum machine_mode, rtx));
```

X 가 값에 대해 rtx (예를 들면, MEM 혹은 REG, SUBREG) 라고 가정하고 X 의 least-significant part 를 참조하는 rtx (MEM 혹은 SUBREG, CONST\_INT) 를 반환한다. MODE 는 반환할 X 의 part 가 얼마나 클지를 지정하며 이것은 보통 한 word 보다 커서는 안된다. 만약 X 가 MEM 이고 이것의 address 가 QUEUED 일 경우, 값 또한 그러할 것이다.

```
rtl gen_highpart          PARAMS ((enum machine_mode, rtl));
```

‘gen\_lowpart’ 와 비슷하지만 most significant part 를 참조한다. 이것은 complex number 의 imaginary part 에 접근하는데 사용된다.

```
rtl gen_highpart_mode    PARAMS ((enum machine_mode,
                                enum machine_mode, rtl));
```

gen\_highpart\_mode 와 비슷하지만 EXP 가 VOIDmode constant 일 수 있는 경우에 EXP operand 의 mode 를 받아들인다는 면에서 다르다.

```
rtl gen_realpart        PARAMS ((enum machine_mode, rtl));
```

Complex value X 의 real part (mode MODE 를 가지고 있는) 를 반환한다. 이것은 항상 memory 내 low address 에 온다.

```
rtl gen_imagpart        PARAMS ((enum machine_mode, rtl));
```

Complex value X 의 imaginary part (mode MODE 를 가지고 있는) 를 반환한다. 이것은 항상 memory 내 high address 에 온다.

```
rtl operand_subword     PARAMS ((rtl, unsigned int, int,
                                enum machine_mode));
```

Operand OP 의 subword OFFSET 을 반환한다. Word number, OFFSET 은 low-order address 로 시작하는 word number 로써 해석된다. OFFSET 0 은 만약 WORDS\_BIG\_ENDIAN 가 아닐 경우 low-order word 이고 그렇지 않을 경우 high-order word 이다.

만약 우리가 요구된 word 를 추출할 수 없다면, 0 을 반환한다. 그렇지 않을 경우, 요청된 word 와 부합하는 rtl 가 반환될 것이다.

VALIDATE\_ADDRESS 는 만약 address 가 유효성 검사가 이루어져야 한다면 0 이 아닌 값이다. reload 가 완료되기 전에, valid address 는 항상 반환될 것이다. reload 후에, 만약 valid address 가 반환될 수 없다면, 우리는 0 을 반환한다.

만약 VALIDATE\_ADDRESS 가 0 일 경우, 우리는 요청된 address 를 간단히 구성하는데 그것을 유효성 검사하는 것은 caller 의 책임이다.

MODE 는 OP 의 mode 인데, 그것이 CONST\_INT 경우에 해당한다.

```
rtl constant_subword    PARAMS ((rtl, int,
                                enum machine_mode));
```

operand\_subword 의 모든 constant 경우들에 대한 helper routine. 몇몇 곳에서 이것을 직접 포함한다.

```
rtl operand_subword_force PARAMS ((rtl, unsigned int,
                                enum machine_mode));
```

‘operand\_subword’ 와 비슷하지만, 절대 0 을 반환하지 않는다. 만약 우리가 요청된 subword 를 추출하지 못했다면, register 내에 OP 를 놓고 재시도 한다. 만약 그래도 실패하면 abort 한다. 우리는 항상 이러한 경우에 address 를 유효성 검사한다.

MODE 는 OP 의 mode 인데, 그것은 CONST\_INT 경우에 해당한다.

```
int subreg_lowpart_p    PARAMS ((rtl));
```

만약 X 가 SUBREG 의 containing reg 의 least significant part 를 참조하는 SUBREG 인 것으로 가정된다면 1 을 반환한다. 만약 X 가 SUBREG 가 아닐 경우, 항상 1 을 반환한다. (그것 자신의 low part 이다.)

```
unsigned int subreg_lowpart_offset PARAMS ((enum machine_mode,
                                           enum machine_mode));
```

Target format 내 memory 에 저장되어 있는 mode INNERMODE 의 값의 OUTERMODE low part 를 얻기 위한 바이트 크기로의 offset 을 반환한다.

```
unsigned int subreg_highpart_offset PARAMS ((enum machine_mode,
                                           enum machine_mode));
```

Target format 내 memory 에 저장되어 있는 mode INNERMODE 의 값의 OUTERMODE high part 를 얻기 위한 바이트 크기로의 offset 을 반환한다.

```
rtl make_safe_from          PARAMS ((rtl, rtl));
```

만약 필요할 경우 X 를 복사하는데, OTHER 내의 변화로 인해 변경되지 않는다. X 혹은 X 의 값이 내부에 복사된 pseudo reg 를 위한 rtl 을 반환한다. OTHER 는 반드시 SET\_DEST 로 써 유효해야 한다.

```
rtl convert_memory_address PARAMS ((enum machine_mode, rtl));
```

ptr\_mode 내 memory address 인 주어진 X 는 Pmode 인 address 로 그것을 변환 하거나, 반대로 한다. (TO\_MODE 는 어떤 방식인지를 말한다.) 우리는 변환을 통해 arithmetic operation 들을 계상하는 과정에서 pointer 들이 overflow 하는 것을 허락하지 않는 점에서 이점을 가지는데, 그 래 인해 address arithmetic insn 들이 사용될 수 있다.

```
rtl get_insns              PARAMS ((void));
```

현재 sequence 혹은 현재 function 의 첫번째 insn 을 반환한다.

```
const char *get_insn_name PARAMS ((int));
```

이것을 정의함으로써 우리는 debug\_rtl function 을 얻기 위해 print-rtl.o 와 연결할 수 있다.

```
rtl get_last_insn         PARAMS ((void));
```

현재 sequence 혹은 현재 function 에 emit 되어진 마지막 insn 을 반환한다.

```
rtl get_last_insn_anywhere PARAMS ((void));
```

Emit 되어진 마지막 insn 을 반환하는데, 그것이 이제 막 push 되어진 sequence 일지라도 변함 이 없다.

```
void start_sequence       PARAMS ((void));
```

RTL\_EXPR 내에 패키징되어질 수 있는 sequence 로 insn 들을 emit 하기 시작한다. 만약 이 sequence 가 컴파일러로 하여금 argument 들을 function call (저런 pop 들은 이전에 연기되었 을 수 있기 때문인데, 더 자세한 사항은 INHIBIT\_DEFER\_POP 를 참조하라) 들에게 pop 할 수 있도록 하는 원일을 포함할 경우, 이 함수를 호출하기 전에 do\_pending\_stack\_adjust 를 사용한 다. 그것은 deferred pop 들이 우연적으로 이 sequence 의 중간에 emit 되지 않았다는 것을 확 실히 할 것이다.

```
void push_to_sequence     PARAMS ((rtl));
```

현재 sequence 로써, 이전에 현재였던 것을 저장하고, FIRST 로 시작하는 insn chain 을 구성한 다. 이 함수를 어떻게 사용하는지에 대한 더 자세한 정보를 start\_sequence 문서에서 보라.

```
void end_sequence        PARAMS ((void));
```

현재 sequence 를 emit 한 후 저장되어 있는 이전 state 를 복구합니다.

지금 막 구성된 sequence 의 content 들을 얻기 위해서, 우리는 반드시 이것을 호출하기 \*전에\* 'gen\_sequence' 를 호출해야 합니다.

만약 컴파일러가 이 sequence 를 생성하는 동안 deferred popping argument 들을 가지고 있다면, 이 sequence 가 즉시 instruction stream 내에 삽입되지 않을 경우, gen\_sequence 을 호출하기 전에 do\_pending\_stack\_adjust 를 사용하라. 그것은 deferred pop 들이 이 sequence 에 삽입되고, instruction stream 내 임의의 장소에 삽입되지 않도록 확실히 한다. Argument 들의 deferred popping 에 관한 더 자세한 정보는 INHIBIT\_DEFER\_POP 를 본다.

```
void push_to_full_sequence      PARAMS ((rtx, rtx));
```

FIRST 에서 LAST 내에 사정된 chain 으로 부터 insn chain 을 구성한다.

```
void end_full_sequence         PARAMS ((rtx*, rtx*));
```

이것은 end\_sequence 와 같이 수행하지만, FIST 에서 LAST 내 오래된 sequence 를 기록한다는 점에서 다르다.

```
rtx gen_sequence              PARAMS ((void));
```

현재 sequence 로 이미 emit 된 insn 들을 포함하는 SEQUENCE rtx 생성한다.

이것은 DEFINE\_EXPAND 로 부터 gen... function 가 어떻게 반환될 SEQUENCE 을 구성하는지에 대해서 이다.

```
rtx emit_insn_before          PARAMS ((rtx, rtx));
```

body PATTERN 를 가지는 명령어를 만들고 instruction BEFORE 앞에 그것을 output 한다.

```
rtx emit_jump_insn_before     PARAMS ((rtx, rtx));
```

body PATTERN 와 code JUMP\_INSN 를 가지는 명령어를 만들고 instruction BEFORE 앞에 그것을 output 한다.

```
rtx emit_call_insn_before     PARAMS ((rtx, rtx));
```

body PATTERN 와 code CALL\_INSN 를 가지는 명령어를 만들고 instruction BEFORE 앞에 그것을 output 한다.

```
rtx emit_barrier_before      PARAMS ((rtx));
```

code BARRIER 의 insn 을 만들고 insn BEFORE 앞에 그것을 output 한다.

```
rtx emit_label_before        PARAMS ((rtx, rtx));
```

code BARRIER 의 insn 을 만들고 insn BEFORE 앞에 그것을 output 한다.

```
rtx emit_note_before         PARAMS ((int, rtx));
```

insn BEFORE 앞에 subtype SUBTYPE 의 note 를 방출한다.

```
rtx emit_insn_after          PARAMS ((rtx, rtx));
```

body PATTERN 를 가진 code INSN 의 insn 을 만들고 insn AFTER 뒤에 그것을 output 한다.

```
rtx emit_jump_insn_after     PARAMS ((rtx, rtx));
```

body PATTERN 를 가진 code JUMP\_INSN 의 insn 을 만들고 insn AFTER 뒤에 그것을 output 한다.

```
rtl emit_barrier_after          PARAMS ((rtl));
```

code BARRIER 의 insn 를 만들고 insn AFTER 뒤에 그것을 output 한다.

```
rtl emit_label_after           PARAMS ((rtl, rtl));
```

insn AFTER 뒤에 label LABEL 를 emit 한다.

```
rtl emit_note_after            PARAMS ((int, rtl));
```

insn AFTER 뒤에 subtype SUBTYPE 의 note 를 emit 한다.

```
rtl emit_line_note_after       PARAMS ((const char *, int, rtl));
```

insn AFTER 뒤에 FILE 과 LINE 을 위한 line note 를 emit 한다.

```
rtl emit_insn                  PARAMS ((rtl));
```

패턴이 PATTERN 이고 코드가 INSN 인 insn 을 생성합니다. 그리고 그것을 doubly-linked list 의 끝에 추가합니다. 만약 PATTERN 이 SEQUENCE 라면 그것의 element 들을 취하여 각 element 를 위한 insn 을 emit 합니다.

마지막으로 emit 된 insn 을 되돌려줍니다.

```
rtl emit_insns                 PARAMS ((rtl));
```

INSN 로 시작하는 chain 내에 insn 들을 emit 한다. emit 되어진 마지막 insn 을 반환한다.

```
rtl emit_insns_before          PARAMS ((rtl, rtl));
```

INSN 로 시작하는 chain 내에 insn 들을 emit 하고 insn BEFORE 앞에 그들을 자리 잡게 한다. emit 되어진 마지막 insn 을 반환한다.

```
rtl emit_insns_after           PARAMS ((rtl, rtl));
```

FIRST 로 시작하는 chain 내에 insn 들을 emit 하고 insn AFTER 의 뒤에 그들을 자리잡게 한다. emit 되어진 마지막 insn 을 반환한다.

```
rtl emit_jump_insn             PARAMS ((rtl));
```

pattern PATTERN 를 가진 code JUMP\_INSN 의 insn 를 만들고 doubly-linked list 의 끝에 그것을 더한다.

```
rtl emit_call_insn             PARAMS ((rtl));
```

pattern PATTERN 를 가진 code CALL\_INSN 의 insn 를 만들고 doubly-linked list 의 끝에 그것을 더한다.

```
rtl emit_label                 PARAMS ((rtl));
```

Doubly-linked list 의 끝에 label LABEL 를 더한다.

```
rtl emit_barrier               PARAMS ((void));
```

code BARRIER 의 insn 을 만들고 doubly-linked list 의 끝에 그것을 더한다.

rtl emit\_line\_note                   PARAMS ((const char \*, int));

FILE 와 LINE 에 의해 지정된 data-field 들을 가지는 code NOTE 의 insn 을 만들고 doubly-linked list 의 끝에 그것을 더한다 하지만 오직 line-number 들이 debugging info 에 요구될 때만.

rtl emit\_note                         PARAMS ((const char \*, int));

FILE 와 LINE 에 의해 지정된 data-field 들을 가지는 code NOTE 의 insn 을 만들고 doubly-linked list 의 끝에 그것을 더한다 만약 line-number NOTE 일 경우, 이전 것과 매칭될 경우 그것을 생략한다.

rtl emit\_line\_note\_force             PARAMS ((const char \*, int));

NOTE 를 emit 하는데, LINE 이 이전 note 라도 그것을 생략하지 않는다.

rtl make\_insn\_raw                    PARAMS ((rtl));

모든 slot 들을 초기화한 INSN rtl 을 생성하여 되돌려줍니다. pattern slot 에는 PATTEN 을 저장합니다.

rtl previous\_insn                    PARAMS ((rtl));

이전 insn 을 반환한다. 만약 SEQUENCE 일 경우, sequence 의 마지막 insn 을 반환한다.

rtl next\_insn                         PARAMS ((rtl));

다음 insn 을 반환한다. 만약 SEQUENCE 일 경우, sequence 의 첫 부분에 insn 을 반환한다.

rtl prev\_nonnote\_insn                PARAMS ((rtl));

NOTE 가 아닌 INSN 앞의 이전 insn 을 반환한다. 이 routine 은 SEQUENCE 들내에 보이지 않는다.

rtl next\_nonnote\_insn                PARAMS ((rtl));

NOTE 가 아닌 INSN 뒤의 다음 insn 을 반환한다. 이 routine 은 SEQUENCE 들내에 보이지 않는다.

rtl prev\_real\_insn                   PARAMS ((rtl));

INSN 앞에 있는 마지막 INSN 혹은 CALL\_INSN, JUMP\_INSN 를 반환하며 존재하지 않을 경우 0 을 반환한다. 이 routine 은 SEQUENCE 들 내에 보이지 않는다.

rtl next\_real\_insn                   PARAMS ((rtl));

INSN 뒤 있는 마지막 INSN 혹은 CALL\_INSN, JUMP\_INSN 를 반환하며 존재하지 않을 경우 0 을 반환한다. 이 routine 은 SEQUENCE 들 내에 보이지 않는다.

rtl prev\_active\_insn                 PARAMS ((rtl));

실제로 어떤 것을 수행하는 INSN 앞 마지막 insn 를 찾는다. 이 routine 은 SEQUENCE 들 내에 보이지 않는다. reload 가 완료될 때 까지 이것은 prev\_real\_insn 와 같다.

rtl next\_active\_insn                 PARAMS ((rtl));

아직 정확한 설명이 없음.

```
int active_insn_p          PARAMS ((rtx));
```

실제로 어떤 것을 수행하는 INSN 뒤 다음 insn 를 찾는다. 이 routine 은 SEQUENCE 들 내에 보이지 않는다. reload 가 완료될 때 까지 이것은 next\_real\_insn 와 같다.

```
rtx prev_label            PARAMS ((rtx));
```

insn INSN 앞 마지막 CODE\_LABEL 를 반환하는데, 없을 경우 0 을 반환한다.

```
rtx next_label            PARAMS ((rtx));
```

insn INSN 뒤 다음 CODE\_LABEL 를 반환하는데, 없을 경우 0 을 반환한다.

```
rtx next_cc0_user         PARAMS ((rtx));
```

INSN 뒤에 CC0 을 사용하고 그것을 설정할 것으로 가정되는 다음 insn 을 반환한다. 이것은 prev\_cc0\_setter 의 반대이다. (즉, 이 함수의 결과에 영향을 미쳤던 prev\_cc0\_setter 는 INSN 를 양보해야 한다.)

일반적으로, 이것은 단순히 다음 insn 이지만, 만약 REG\_CC\_USER note 가 존재한다면 그것은 CC0 을 사용하는 insn 을 포함한다.

만약 우리가 insn 을 찾을 수 없다면 0 을 반환한다.

```
rtx prev_cc0_setter       PARAMS ((rtx));
```

INSN 을 위해 CC0 을 설정하는 insn 를 찾는다. 만약 INSN 가 REG\_CC\_SETTER note 를 가지고 있지 않다면 그것은 이전 insn 이다.

```
rtx try_split             PARAMS ((rtx, rtx, int));
```

더 나은 스케줄링을 위해 분리될 수 있는 insn 들을 분리 시도한다. PAT 은 분리할 패턴들이다. TRIAL 은 PAT 을 제공하는 insn 이다. LAST 는 만약 우리가 생성된 sequence 의 마지막 insn 을 반환해야 한다면 0 이 아닌 값이다.

만약 이 routine 이 분리하는데 성공하였다면, LAST 의 값에 의존하여 처음 혹은 마지막 replacement insn 를 반환한다. 그렇지 않다면 그것은 TRIAL 을 반환한다. 만약 반환되어진 insn 가 분리될 수 있다면 그것은 수행될 것이다.

```
rtx set_unique_reg_note   PARAMS ((rtx, enum reg_note, rtx));
```

Datum 으로서 DATUIM 을 가지는 insn INSN 위에 KIND 의 note 를 자리 잡게 한다. 만약 이 type 의 note 가 이미 존재한다면, 먼저 그것을 제거한다.

```
rtx gen_rtx_CONST_DOUBLE PARAMS ((enum machine_mode,
                                   HOST_WIDE_INT, HOST_WIDE_INT));
```

CONST\_DOUBLE 들은 특별하게 다를 필요가 있는데, 그들의 길이가 오직 run-time 시에만 알 수 있기 때문이다.

```
rtx gen_rtx_CONST_INT     PARAMS ((enum machine_mode, HOST_WIDE_INT));
```

특별한 관심을 요구하는 몇몇 RTL code 들이 있습니다; generation function 들은 raw handling 을 한다. 만약 당신이 이 list 에 더한다면, gengenrtl.c 내 special\_rtx 또한 수정하라.

```
rtx gen_raw_REG           PARAMS ((enum machine_mode, int));
```

새로운 REG rtx 를 생성합니다. ORIGINAL\_REGNO 가 제대로 설정되는지, 여러 전역 rtl (frame\_pointer\_rtx 와 같은) 과 공유를 시도하지 않도록 확실하게 합니다.

```
rtl gen_rtx_REG PARAMS ((enum machine_mode, int));
```

아직 정확한 설명이 없음.

```
rtl gen_rtx_SUBREG PARAMS ((enum machine_mode, rtl, int));
```

아직 정확한 설명이 없음.

```
rtl gen_rtx_MEM PARAMS ((enum machine_mode, rtl));
```

아직 정확한 설명이 없음

```
rtl gen_lowpart_SUBREG PARAMS ((enum machine_mode, rtl));
```

만약 MODE 가 REG 의 mode 보다 작을 경우 REG 의 least-significant part 를 나타내는 SUBREG 를 생성한다. 그렇지 않다면 paradoxical SUBREG 이다.

```
int max_reg_num PARAMS ((void));
```

현재 함수에서 사용되는 가장 큰 pseudo reg number 에 1 을 더한 값을 반환한다.

```
int max_label_num PARAMS ((void));
```

현재 함수내 지금까지 사용된 가장 큰 label number + 1 을 반환한다.

```
int get_first_label_num PARAMS ((void));
```

(만약 어떤 것이 사용되었을 경우) 이 함수내 사용된 첫번째 label number 를 반환한다.

```
void delete_insns_since PARAMS ((rtl));
```

FROM 이후로 만들어진 모든 insn 들을 삭제한다. FROM 은 새로운 마지막 명령어가 된다.

```
void mark_reg_pointer PARAMS ((rtl, int));
```

가망이 있는 pointer register 로써 REG 을 인식하고 만약 ALIGN 이 0 이 아닐 경우 그것의 alignment 로써 보인다.

```
void mark_user_reg PARAMS ((rtl));
```

(CONCAT 일 수 있는) REG 를 user register 로써 인식한다.

```
void reset_used_flags PARAMS ((rtl));
```

공유된 sub-part 들을 보는데 사용된 copy\_rtx\_if\_shared 를 허락하기 위해 X 내 모든 USED bit 들을 깨끗히 한다.

```
void reorder_insns_nobb PARAMS ((rtl, rtl, rtl));
```

이 함수는 현재 비난을 받고 있다. 대신 sequence 들을 사용해 달라.

Chain 내 다른 장소로 insn 들의 consecutive bunch (연속적인 묶음) 을 이동한다. 이동된 insn 들은 FROM 와 TO 사이에 있다. 그들은 insn AFTER 뒤 새로운 위치에 자리잡을 것이다. AFTER 는 절대 FROM 혹은 TO 혹은 그 사이에 있는 어떠한 insn 면 안된다.

이 함수는 SEQUENCE 들에 대해서 알지 못하므로 delay-slot filling 이 수행된 후 호출되어서는 안될 것이다.

```
void reorder_insns PARAMS ((rtl, rtl, rtl));
```

위와 함수는 같지만, BB boundary 들을 업데이트하는 것을 돌본다.

```
int get_max_uid          PARAMS ((void));
```

이 함수의 어떠한 명령어들의 uid 보다 큰 number 를 반환한다.

```
int in_sequence_p       PARAMS ((void));
```

현재 sequence 내에서 emit 중이라면 값 1 을 반환.

```
void force_next_line_note PARAMS ((void));
```

Line number 가 변경되지 않았더라고 line note 를 다음 statement 에 발생하도록 한다. 이것은 함수의 시작 부분에 사용된다.

```
void clear_emit_caches  PARAMS ((void));
```

아직 설명이 없음.

```
void init_emit          PARAMS ((void));
```

각 함수를 위한 rtl 을 생성하기 전에 이 파일내에 존재하는 data structure 들과 variable 들을 초기화합니다.

```
void init_emit_once     PARAMS ((int));
```

모든 함수사이에 공유되는 몇몇 unique rtl object 를 생성합니다. 만약 line number 가 생성되었다면 LINE\_NUMBERS 는 0 이 아닐 것입니다.

```
void push_topmost_sequence PARAMS ((void));
```

현재 sequence 로써 outer-level insn chain 를 설정하고 이전에 현재였던 것을 저장한다.

```
void pop_topmost_sequence PARAMS ((void));
```

outer-level insn chain 을 emit 한 후, outer-level insn chain 를 업데이트하고, 이전에 저장되었던 상태를 복구한다.

```
int subreg_realpart_p   PARAMS ((rtx));
```

만약 X 가 SUBREG 로써 가정된다면 1 을 반환한다. 그것의 containing reg 내 complex value 의 real part 를 참조한다. Complex value 들은 항상 first word 내 real part 가 저장되며 이것은 WORDS\_BIG\_ENDIAN 와 상관없다.

```
void reverse_comparison PARAMS ((rtx));
```

주어진 compare instruction 로 operand 들은 swap 한다. Test instruction 은 operand 와 0 의 비교로 변경된다.

```
void set_new_first_and_last_insn PARAMS ((rtx, rtx));
```

Chain 내 첫번째 와 마지막 insn 들에 대한 새로운 pointer 들을 설치한다. 또한 사용된 마지막 보다 1 큰 cur\_insn\_uid 를 설정한다. insn chain 를 복사한 후 inline-procedure 를 위해 사용된다.

```
void set_new_first_and_last_label_num PARAMS ((int, int));
```

현재 함수에서 발견된 label number 들의 range 를 설정한다. 이것은 inline function 을 구식 방법을 사용하여 컴파일할 때 사용된다.

```
void set_new_last_label_num          PARAMS ((int));
```

현재 함수에서 발견된 label number 들의 range 를 설정한다. 이것은 inline function 을 구식 방법을 사용하여 컴파일할 때 사용된다.

```
void unshare_all_rtl_again          PARAMS ((rtx));
```

RTL insn body 들을 통과시키고 어떤 잘못된 shared structure 들을 다시 복수한다. 이것은 폐비용이 비싼편이기 때문에 알뜰하게 수행되어야 한다.

```
void set_last_insn                  PARAMS ((rtx));
```

Chain 내 마지막으로 새 insn 를 지정한다.

```
void link_cc0_insns                PARAMS ((rtx));
```

INSN 는 CC0 를 사용하고 delay slot 내로 이동되는 중이다. REG\_CC\_SETTER 와 REG\_CC\_USER note 들을 설정해 우리는 그것을 찾을 수 있다.

```
void add_insn                       PARAMS ((rtx));
```

INSN 을 doubly-linked list 의 끝에 추가합니다. INSN 는 아마도 INSN, JUMP\_INSN, CALL\_INSN, CODE\_LABEL, BARRIER 혹은 NOTE 일 것입니다.

```
void add_insn_before                PARAMS ((rtx, rtx));
```

INSN 를 insn BEFORE 앞 doubly-linked list 에 더한다. 이것과 이전 것은 insn 를 삽입하기 위해 호출되어지는 유일한 function 들이어야 하는데, 단지 그들은 SEQUENCE 를 어떻게 update 할지만 알기 때문에 1 회만 delay slots 이 채워진다.

```
void add_insn_after                 PARAMS ((rtx, rtx));
```

INSN 를 insn AFTER 뒤 doubly-linked list 에 더한다. 이것과 이전 것은 insn 를 삽입하기 위해 호출되어지는 유일한 function 들이어야 하는데, 단지 그들은 SEQUENCE 를 어떻게 update 할지만 알기 때문에 1 회만 delay slots 이 채워진다.

```
void remove_insn                   PARAMS ((rtx));
```

그것의 doubly-linked list 로 부터 insn 를 제거한다. 이 함수는 sequence 들을 어떻게 다룰지 알고 있다.

```
void reorder_insns_with_line_notes  PARAMS ((rtx, rtx, rtx));
```

reorder\_insns 와 비슷하지만 디버깅시 moved insn 들의 line number 들을 예약하기 위해 line note 들을 삽입한다. 이것은 AFTER 와 FROM 사이, TO 뒤에 다른 하나에 note 를 삽입할 것이다.

```
void emit_insn_after_with_line_notes PARAMS ((rtx, rtx, rtx));
```

emit\_insn\_after 와 비슷하지만, Line note 들은 삽입된 것인데, 이 insn 이 FROM 에 있었던 것처럼 행동하는 것이 다르다.

```
enum rtx_code classify_insn         PARAMS ((rtx));
```

어떤 insn 의 type 이 body 로써 X 를 가져야 하는지에 대한 지시를 반환한다. 값은 CODE\_LABEL 혹은 INSN, CALL\_INSN, JUMP\_INSN.

```
rtx emit                                PARAMS ((rtx));
```

Insn 의 적당한 종류로써 rtl 패턴 X 를 방출한다. 만약 X 가 label 일 경우, insn chain 내에 간단히 더해진다.

```
int force_line_numbers PARAMS ((void));
```

Query 와 clear/ 는 no\_line\_numbers 를 복구한다. 이것은 도달할 수 없는 code 에 관해 경고 메시지를 출력하는 적당한 line number 들을 주기위해 stmt.c 내에 switch / case 를 다루는 데에 사용된다.

```
void restore_line_number_status PARAMS ((int old_value));
```

아직 정확한 설명이 없음.

```
void renumber_insns                      PARAMS ((FILE *));
```

명령어들을 재 숫자화하여 어떠한 instruction UID 들도 낭비되지 않는다.

```
void remove_unnecessary_notes            PARAMS ((void));
```

불필요한 node 들을 명령어 stream 에서 제거한다.

## 6.8 explow.c

```
rtx find_next_ref                        PARAMS ((rtx, rtx));
```

INSN 뒤 REG 를 참조하는 다음 insn 를 반환하며, 만약 REG 가 참조된 다음 것의 앞에서 clobber 되었거나 code 의 straight-line piece 내 REG 를 참조하는 insn 을 찾을 수 없을 때 0 을 반환한다.

```
void set_stack_check_libfunc PARAMS ((rtx));
```

Front end 는 GCC 의 stack 을 override 하기를 원할 수 있는데, 이것은 stack 을 검사하기 위해 어떤 call 을 run-time routine 에 제공함으로써 검사되어진다. 그래서 그러한 routine 을 호출하기 위한 메카니즘을 여기서 제공한다.

```
HOST_WIDE_INT trunc_int_for_mode PARAMS ((HOST_WIDE_INT,
                                           enum machine_mode));
```

MODE 를 위한 적당한 truncate 이고 sign-extend C 인 것.

```
rtx plus_constant_wide                   PARAMS ((rtx, HOST_WIDE_INT));
```

X 와 정수 C 의 합인 rtx 를 반환합니다.

이 함수는 반드시 'plus\_constant' macro 를 통해 사용되어야 합니다.

```
void optimize_save_area_alloc          PARAMS ((rtx));
```

SETJMP\_VIA\_SAVE\_AREA 가 true 인 target 을 위해 allocate\_dynamic\_stack\_space 에 의해 생성된 RTL 을 최적화 한다. 문제점이 있는데, 이런 platform 들 상에서 사용된 dynamic stack space 는 원래 frame 과 충돌이 날 수 있으며, 이로 인해 longjmp 가 그것을 unwind 한다면 crash 를 불러 일으킬수 있다.

## 6.9 expmed.c

```
void init_expmed          PARAMS ((void));
```

아직 정확한 설명이 없음.

```
void expand_inc          PARAMS ((rtx, rtx));
```

TARGET 에 INC 를 더한다.

```
void expand_dec          PARAMS ((rtx, rtx));
```

TARGET 으로부터 DEC 를 뺀다.

```
rtx expand_mult_highpart  PARAMS ((enum machine_mode, rtx,
                                unsigned HOST_WIDE_INT, rtx,
                                int, int));
```

OP0 와 CNST1 을 곱하기 위한 code 를 방출하는데, 만약 편안한 방식일 경우 TARGET 내 결과의 high half 를 놓는다. 그리고 그 결과가 어디인지를 반환한다. 만약 operation 이 수행되어 질 수 없다면, 0 이 반환된다.

MODE 는 operation 과 result 의 mode 이다.

UNSIGNEDP 는 unsigned 곱셈임을 의미할 때 0 이 아니다.

MAX\_COST 는 확장된 RTL 를 위한 총 allowed cost 이다.

```
int ceil_log2            PARAMS ((unsigned HOST_WIDE_INT));
```

$2^{*n} \geq X$  를 만족하는 가장 작은 n 을 반환한다.

## 6.10 expr.c

```
void move_by_pieces      PARAMS ((rtx, rtx,
                                unsigned HOST_WIDE_INT,
                                unsigned int));
```

Block FROM 에서 block TO 까지 LEN byte 들을 복사하는 여러 move 명령어들을 생성한다. (이것들은 BLKmode 를 가진 MEM rtx 들이다.) Caller 는 호출하기 전 protect\_from\_queue 을 통해 반드시 FROM 과 TO 를 건네야 한다.

만약 PUSH\_ROUNDING 가 정의되어졌고 TO 가 NULL 이면, emit\_single\_push\_insn 는 stack 에 FROM 을 push 하는데 사용된다.

ALIGN 은 우리가 가정할 수 있는 최대 alignment 이다.

## 6.11 flow.c

```
rtx find_use_as_address  PARAMS ((rtx, rtx, HOST_WIDE_INT));
```

Rtx X 내에서 REG 가 memory address 로써 사용된 장소를 찾는다. 그래서 그것을 사용한 MEM rtx 을 반환한다. 만약 PLUSCONST 가 0 이 아닐 경우, 대신 memory address 와 동일한 (plus REG (const.int PLUSCONST)) 를 찾는다.

만약 그러한 address 가 보이지 않으면 0 을 반환한다. 만약 REG 가 한번 이상 나타날 경우, 혹은 address 로써가 아닌 다른 것으로 사용될 경우 (rtx) 1 을 반환한다.

```
void recompute_reg_usage PARAMS ((rtx, int));
```

Register allocation 바로 전에 register set/reference count 들을 재계산 한다.

이것은 register allocator 들에 대해 특별한 의미들을 가지고 있는 값들을 변경 하거나/되어지게 하는 set/reference count 들로 인한 문제를 피한다.

게다가, reference count 들은 hard regs 로의 allocation 을 위한 pseudos 를 우선 순위 매기기 위해 register allocator 들에 의해 사용되어지는 primary component 이다. 더욱 정확한 reference count 들은 일반적으로 더 나은 register allocation 를 이끈다.

F 는 scan 되어진 첫번째 insn 이다.

LOOP\_STEP 는 얼마나 많은 loop\_depth 가 loop nesting level 당 증가되어야 하는지를 나타내는데, 이것은 loop 내 reference 들을 위한 ref count 를 증가하기 위해서 이다.

Register allocator 들에 의해 사용되는 REG\_LIVE\_LENGTH 와 REG\_BASIC\_BLOCK, 가능한 다른 정보를 update 하는 것은 충분히 가치가 있을 것이다.

```
int initialize_uninitialized_subregs    PARAMS ((void));
```

Entry 상에 살아있는 pseudo register 들을 찾는 entry block 의 모든 immediate successor 들을 처리한다. 그것들의 첫번째 instance 가 몇몇 종류의 부분적인 register reference 인 모든 것을 찾고, entry block 뒤 그들을 0 으로 초기화한다. 이것은 register 들의 값이 알려지지 않는 것들 내에서의 bit set 들을 막고, 우리가 원하지 않는 sticky bit 들의 몇몇 종류를 포함할 수 있다.

## 6.12 fold-const.c

```
int add_double    PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                          unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                          unsigned HOST_WIDE_INT *,
                          HOST_WIDE_INT *));
```

두 doubleword 정수들을 doubleword 결과로 더한다. 각 argument 는 두 'HOST\_WIDE\_INT' 조각들로 주어진다. 한 argument 는 L1 과 H1; 다른 하나는 L2 와 H2 이다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
int neg_double    PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                          unsigned HOST_WIDE_INT *,
                          HOST_WIDE_INT *));
```

두 doubleword 정수들을 doubleword 결과로 negate 한다. 그것이 sign 으로 가정했을 때 만약 operation 이 overflow 이면, 0 이 아닌 값을 반환한다. 각 argument 는 L1 과 H1 내 두 'HOST\_WIDE\_INT' 조각들로 주어진다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
int mul_double    PARAMS ((unsigned HOST_WIDE_INT,
                          HOST_WIDE_INT,
                          unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                          unsigned HOST_WIDE_INT *,
                          HOST_WIDE_INT *));
```

두 doubleword 정수들을 doubleword 결과로 곱한다. 그것이 sign 으로 가정했을 때 만약 operation 이 overflow 이면, 0 이 아닌 값을 반환한다. 각 argument 는 두 'HOST\_WIDE\_INT' 조각들로 주어진다. 한 argument 는 L1 과 H1; 다른 하나는 L2 와 H2 이다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
void lshift_double    PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                              HOST_WIDE_INT, unsigned int,
                              unsigned HOST_WIDE_INT *,
                              HOST_WIDE_INT *, int));
```

결과의 PREC bit 들만 유지한채, L1, H2 내 Doubleword 정수를 COUNT 자리로 왼쪽 shift 한다. 만약 COUNT 가 음수이면 오른쪽 shift 한다. ARITH nonzero 는 arithmetic shift 을 의미하며 그렇지 않을 땐 logical shift 를 사용한다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
void rshift_double      PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                                HOST_WIDE_INT, unsigned int,
                                unsigned HOST_WIDE_INT *,
                                HOST_WIDE_INT *, int));
```

결과의 PREC bit 들만 유지한채, L1, H2 내 Doubleword 정수를 COUNT 자리로 오른쪽 shift 한다. COUNT 는 반드시 양수여야 한다. ARITH nonzero 는 arithmetic shift 을 의미하며 그렇지 않을 땐 logical shift 를 사용한다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
void lrotate_double    PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                                HOST_WIDE_INT, unsigned int,
                                unsigned HOST_WIDE_INT *,
                                HOST_WIDE_INT *));
```

결과의 PREC bit 들만 유지한채, L1, H2 내 Doubleword 정수를 COUNT 자리로 왼쪽 rotate 한다. 만약 COUNT 가 음수일 경우 오른쪽 rotate 한다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

```
void rrotate_double    PARAMS ((unsigned HOST_WIDE_INT, HOST_WIDE_INT,
                                HOST_WIDE_INT, unsigned int,
                                unsigned HOST_WIDE_INT *,
                                HOST_WIDE_INT *));
```

결과의 PREC bit 들만 유지한채, L1, H2 내 Doubleword 정수를 COUNT 자리로 오른쪽 rotate 한다. COUNT 는 반드시 양수여야 한다. 값은 \*LV 와 \*HV 내 두 'HOST\_WIDE\_INT' 조각으로 저장된다.

## 6.13 function.c

```
rtx assign_stack_local  PARAMS ((enum machine_mode,
                                HOST_WIDE_INT, int));
```

assign\_stack\_local\_1 주위로 둘러싸는 wrapper; 현재 함수를 위한 local stack slot 을 할당한다.

```
rtx assign_stack_temp    PARAMS ((enum machine_mode,
                                HOST_WIDE_INT, int));
```

임시적인 stack slot 을 할당하고 나중에 다시 재사용을 위해 그것을 기록한다. 먼저 3 개의 argument 들은 함수를 처리하는데 있어서 같다.

```
rtx assign_stack_temp_for_type  PARAMS ((enum machine_mode,
                                HOST_WIDE_INT, int, tree));
```

임시적인 stack slot 을 할당하고 나중에 다시 재사용을 위해 그것을 기록한다.

MODE 는 반환된 rtx 에게 주어진 machine mode 이다.

SIZE 는 요구된 space 의 unit 들내 size 이다. 우리는 여기에 rounding 을 하지 않는데, assign\_stack\_local 가 어떤 요구된 rounding 을 할 것이기 때문이다.

KEEP 은 만약 이 slot 이 free\_temp\_slots 로의 call 후에 유지되어야 하는 것일 경우 1 이다. Block 을 위한 automatic variable 들은 이 flag 를 가지고 할당 된다. KEEP 은 만약 우리가 임시적으로 longer term 을 할당할 경우, 2 를 갖는데, 이것의 lifetime 은 CLEANUP\_POINT\_EXPR 들에 의해 제어된다. KEEP 은 만약 inner level 에서 block 내 변수로써 취급되는 어떤 것을 우리가 할당하고자 할 경우 3 을 갖는다. (예를 들면, AVE\_EXPR).

TYPE 은 stack slot 을 위해 사용되는 type 이다.

```
rtx assign_temp          PARAMS ((tree, int, int, int));
```

Temporary 를 할당한다. 만약 TYPE\_OR\_DECL 가 decl 일 경우 우리는 decl 을 대신하여 그것을 하고 error 메세지들내에 사용되어야 한다. 그러한 경우에, 주어진 type 에 대한 할당을 한다. KEEP 이 assign\_stack\_temp 을 위한 것이다. MEMORY\_REQUIRED 는 만약 결과가 반드시 addressable stack memory 여야 할 경우 1 을 갖는다; 만약 register 가 OK 이면 0 이다. DONT\_PROMOTE 는 만약 우리가 wider mode 들로 register 내 값을 승진시켜야 할 경우 1 을 갖는다.

```
rtx gen_mem_addressof   PARAMS ((rtx, tree));
```

Register REG 를 위한 (MEM (ADDRESSOF (REG))) rtx 를 만드는데, 단순히 그것의 주소만 갖는 것으로 한다. DECL 은 decl 혹은 register 내 저장된 object 를 위한 SAVE\_EXPR 이고 만약 우리가 stack 내에 REG 를 강제로 넣어야 할 필요가 있을 경우 나중에 사용될 것이다. REG 는 put\_reg\_into\_stack 내에 있는 MEM 과 같은 것에 의해 덮어쓰여 진다.

```
void reposition_prologue_and_epilogue_notes PARAMS ((rtx));
```

Instruction scheduling 와 delayed branch scheduling 후에 prologue-end 와 epilogue-begin note 들을 재위치시킨다.

```
void thread_prologue_and_epilogue_insns    PARAMS ((rtx));
```

만약 machine 이 그것을 지원할 경우 prologue 와 epilogue RTL 를 생성한다. 그리고 이것을 prologue 가 어디서 끝나는지, epilogue 가 어디서 시작하는지를 가르키는 node 를 가진 곳에 thread 한다. 가능한 basic block 정보를 update 한다.

```
int prologue_epilogue_contains             PARAMS ((rtx));
```

아직 정확한 설명이 없음.

```
int sibcall_epilogue_contains              PARAMS ((rtx));
```

아직 정확한 설명이 없음.

```
void preserve_rtl_expr_result              PARAMS ((rtx));
```

X 는 RTL\_EXPR 의 결과이다. 만약 해당 RTL\_EXPR 와 결합된 임시 slot 일 경우, 현재 level 에서의 임시 slot 내로 그것을 승진시키켜서 RTL\_EXPR 내 구성된 slot 들을 free 할 때 free 되지 않도록 한다.

```
void mark_temp_addr_taken                  PARAMS ((rtx));
```

만약 X 가 임시 slot 으로의 reference 일 수 있다면 그것의 address 가 얻어질 수 있다는 사실을 기록한다.

```
void update_temp_slot_address              PARAMS ((rtx, rtx));
```

NEW 가 이전에서는 OLD 로 알려졌는 temp slot 을 참조하는 동일한 방식임을 가르킨다.

```
void purge_addressof          PARAMS ((rtx));
```

INSNS 로부터 ADDRESSOF 의 모든 발생을 제거한다. 남아있는 (MEM (ADDRESSOF)) 패턴들을 제거하고, stack 내에 필요한 어떤 register 들을 강요한다.

```
void purge_hard_subreg_sets   PARAMS ((rtx));
```

INSNS 로부터 hard subregs 의 SET 들의 모든 발생을 제거한다. 우리가 예상하는 그러한 SET 들만 남겨진 그것들이다. 이는 integrate 가 return 값 register 의 부분들의 집합을 다룰 수 없기 때문이다.

우리는 오직 hard register 들의 subregs 만 제거하길 원하기 때문에 alter\_subreg 를 사용하지 않는다.

## 6.14 gcse.c

```
int gcse_main                 PARAMS ((rtx, FILE *));
```

Global common subexpression elimination 에 대한 entry point. F 는 함수내 첫번째 명령어 이다.

## 6.15 global.c

```
void mark_elimination        PARAMS ((int, int));
```

Hard register number FROM 가 이전에 제거되어 hard register number TO 부터의 offset 으로 대체되었음을 말한다. Basic block 의 시작 부분에서 hard registers live 의 상태는 FROM 의 사용을 TO 의 사용으로 대체함으로써 update 되었다.

```
int global_alloc             PARAMS ((FILE *));
```

local\_alloc 에 의해 할당되지 않은 pseudo-register 들의 allocation 을 수행한다. FILE 은 debugging information 를 output 할 파일이며, 만약 그러한 output 이 요구되지 않을 경우 0 이다.

Return 값은 만약 reload 가 실패했고, 우리가 이 함수에 대해 더이상 어떤 것을 하지 말아야 할 때 0 이 아닌 값을 반환한다.

```
void dump_global_regs        PARAMS ((FILE *));
```

아직 정확한 설명이 없음.

```
void retry_global_alloc      PARAMS ((int, HARD_REG_SET));
```

pseudo reg REGNO 내에 넣기 위한 hard reg 를 찾기 위해 'reload' 로부터 호출된다. 아마도 이전에 hard reg 가 가치있게 보여지지 않았거나, 혹은 아마도 그것의 오래된 hard reg 가 reloads 를 위해 징발되었다. FORBIDDEN\_REGS 는 사용된 적이 없는 특정 hard reg 들, 심지어 그것이 할당된 것으로 보이지 않는 것들을 가르킨다. 만약 FORBIDDEN\_REGS 가 0 이면, 아무런 regs 가 금지되지 않았다.

```
void build_insn_chain        PARAMS ((rtx));
```

현재 함수의 insn 을 걸어가면서 reload\_insn\_chain 를 생성하고 register life information 을 기록한다.

## 6.16 ifcvt.c

```
void if_convert          PARAMS ((int));
```

모든 if-conversion 를 위한 main entry point.

## 6.17 list.c

```
void init_EXPR_INSN_LIST_cache    PARAMS ((void));
```

아직 정확한 설명이 존재하지 않음

```
void free_EXPR_LIST_list          PARAMS ((rtx *));
```

이 함수는 EXPR\_LIST node 들의 전체 list 를 자유롭게 할 것이다.

```
void free_INSN_LIST_list          PARAMS ((rtx *));
```

이 함수는 INSN\_LIST node 들의 전체 list 를 자유롭게 할 것이다.

```
void free_EXPR_LIST_node          PARAMS ((rtx));
```

이 함수는 개별적인 EXPR\_LIST node 를 자유롭게 할 것이다.

```
void free_INSN_LIST_node          PARAMS ((rtx));
```

이 함수는 개별적인 INSN\_LIST node 를 자유롭게 할 것이다.

```
rtx alloc_INSN_LIST              PARAMS ((rtx, rtx));
```

이 call 은 gen\_rtx\_INSN\_LIST 의 장소에 사용된다. 만약 이용 가능한 cached node 가 존재한다면 우리는 그것을 사용할 것이다. 그렇지 않다면 gen\_rtx\_INSN\_LIST 로의 call 이 만들어진다.

```
rtx alloc_EXPR_LIST              PARAMS ((int, rtx, rtx));
```

이 call 은 gen\_rtx\_EXPR\_LIST 의 장소에 사용된다. 만약 이용 가능한 cached node 가 존재한다면 우리는 그것을 사용할 것이다. 그렇지 않다면 gen\_rtx\_EXPR\_LIST 로의 call 이 만들어진다.

## 6.18 local-alloc.c

```
void dump_local_alloc           PARAMS ((FILE *));
```

아직 정확한 설명이 없음.

```
int local_alloc                 PARAMS ((void));
```

\$prefix/gcc/local-alloc.c 파일의 main entry point.

```
int function_invariant_p        PARAMS ((rtx));
```

만약 X 가 현재 함수에 대해 불변일 경우 0 이 아닌 값을 반환한다.

## 6.19 jump.c

```
rtx next_nondeleted_insn      PARAMS ((rtx));
```

INSN 부터 전진하다가 지워지지 않은 어떤 것에 도착하면 그것을 반환한다. 아마 INSN 자신을 반환할 것이다.

```
enum rtx_code reverse_condition  PARAMS ((enum rtx_code));
```

비교를 위해 주어진 rtx-code 는 negated comparison 를 위한 code 를 반환한다. 만약 그러한 code 가 존재하지 않을 경우 UNKNOWN 을 반환한다.

주의! reverse\_condition 는 IEEE floating point comparison 의 결과 상의 행동할 수 있는 jump 에 대해 사용하는 것은 안전하지 못한데, 그것은 comparison 들내에서 non-signaling nans 의 특별한 취급으로 인한 것이다. 대신에 reversed\_comparison\_code 를 사용하라.

```
enum rtx_code reverse_condition_maybe_unordered PARAMS ((enum rtx_code));
```

비슷하지만, 우리는 IEEE floating-point 에 대해 안전하게 그것을 만드는 unordered comparison 들을 생성하는데 허락받았다. 물론, 우리는 target 이 그들을 지원하는지 인식해야 할 것이다.

```
enum rtx_code swap_condition     PARAMS ((enum rtx_code));
```

비슷하지만, comparison 의 두 operand 들이 swap 되어졌을 때 code 를 반환한다. 이것은! IEEE floating-point 에 대해 안전하다.

```
enum rtx_code unsigned_condition PARAMS ((enum rtx_code));
```

주어진 comparison CODE 는 그에 대응하는 unsigned comparison 을 반환한다. 만약 CODE 가 equality comparison 이거나 이미 unsigned comparison 이면 CODE 가 반환된다.

```
enum rtx_code signed_condition   PARAMS ((enum rtx_code));
```

비슷하지만, comparison 의 signed version 을 반환한다.

```
void mark_jump_label          PARAMS ((rtx, rtx, int));
```

X 내에서 참조되는 모든 CODE\_LABEL 들을 찾고, 그들의 use count 들을 증가시킨다. 만약 INSN 가 JUMP\_INSN 이고 INSN 내 참조되는 CODE\_LABEL 가 적어도 하나 이상 이라면 JUMP\_LABEL (INSN) 내에 그들의 하나를 저장한다. 만약 INSN 가 INSN 혹은 CALL\_INSN 이고, INSN 내 참조되는 CODE\_LABEL 가 적어도 하나 이상일 경우, INSN 로의 해당 label 을 포함하는 REG\_LABEL note 를 더한다. 또한, 연속적인 label 들이 존재할 때, 그들 중 마지막을 canonicalize 한다.

loop-beginning note 로 분리되어 있는 두 label 들은 우리가 아직 loop-optimization 를 수행하지 않았다면 반드시 독립적으로 유지하여야 한다. 왜냐하면 그들 사이에 gap 이 loop-optimize 가 invariant code 를 이동하고자 하는 곳이기 때문이다. CROSS\_JUMP 는 loop-optimization 가 같지 수행됨을 우리에게 말해준다.

```
void cleanup_barriers         PARAMS ((void));
```

몇몇 오래된 code 는 non-fallthru insn 의 NEXT\_INSN 로써 정확히 하나의 BARRIER 가 오기를 예상한다. 이것은 일반적으로 true 가 아니지만, 다수의 barrier 들이 조금씩 있을 수 있고, BARRIER 가 하나 혹은 더 많은 NOTE 들에 의해 마지막 real insn 로부터 분리될 수 있기 때문에 이해되어야 한다.

이 simple pass 는 barrier 들을 이동시키고 중복되는 것을 제거함으로써 오래된 code 가 행복하게 한다.

```
bool squeeze_notes          PARAMS ((rtx *, rtx *));
```

START 바로 뒤부터 시작하여 START 와 END 사이에 존재하는 모든 block-beg, block-end, loop-beg, loop-cont, loop-vtop, loop-end, note 들을 옮긴다. START 와 END 는 그러한 note 들 일 것이다. 만약 원래의 것이 그런 note 일 경우 다를 수 있지만, 새로운 starting 과 ending insn 들의 값을 반환한다. 만약 그러한 note 들만 존재하지 실제 명령어들이 없을 경우 true 를 반환한다.

```
rtx delete_related_insns    PARAMS ((rtx));
```

Insn 들의 chain 으로 부터 insn INSN 를 삭제하고 label ref count 들을 update 하고 현재 도달 할 수 없는 insn 들을 삭제한다.

삭제되지 않은 INSN 뒤의 첫번째 insn 을 반환한다.

이 명령어의 사용은 반대 의견이 많다. 대신 delete\_insn 를 사용하고 만약 필요한 경우 도달할 수 없는 code 를 삭제하기 위해서는 subsequent cfg.cleanup pass 를 사용하라.

```
void delete_jump           PARAMS ((rtx));
```

만약 수행하는 모든 INSN 가 pc 를 설정할 경우, 그것을 삭제하고 만약 그것이 이전 것이었을 경우 그것을 위한 condition code 들을 설정하는 insn 를 삭제한다.

```
void delete_barrier        PARAMS ((rtx));
```

INSN 가 BARRIER 인지 확인하고 그것을 삭제한다.

```
rtx get_label_before       PARAMS ((rtx));
```

INSN 앞 label 을 반환하거나, 그곳에 새 label 을 놓는다.

```
rtx get_label_after        PARAMS ((rtx));
```

INSN 뒤 label 을 반환하거나, 그곳에 새 label 을 놓는다.

```
rtx follow_jumps          PARAMS ((rtx));
```

LABEL 에서의 어떤 unconditional jump 를 따라간다; 그러한 jump 들의 chain 에 의해 도착한 ultimate label 을 반환한다. 만약 LABEL 이 jump 에 대해 다르지 않는다면, LABEL 을 반환한다. 만약 chain loop 들 혹은 우리가 끝을 찾을 수 없다면, LABEL 을 반환한다. 이는 insn 를 변경하는 것을 비하기 위해 caller 에서 말해주기 때문이다.

만약 RELOAD\_COMPLETED 가 0 이면, 우리는 NOTE\_INSN\_LOOP\_BEG 혹은 USE 혹은 CLOBBER 사이를 연결하지 않는다.

```
int comparison_dominates_p PARAMS ((enum rtx_code, enum rtx_code));
```

만약 CODE1 이 CODE2 보다 더 strict 하다면, 즉 만약 CODE1 의 truth 가 CODE2 의 truth 를 내포한다면 0 이 아닌 값을 반환한다.

```
int condjump_p            PARAMS ((rtx));
```

만약 INSN 가 (가능한) conditional jump 이고 그 이상이 아니라면 0 이 아닌 값을 반환한다.

이 함수의 사용은 반대 의견이 있는데, 그것은 우리가 combined branch 와 compare insn 들을 지원할 필요가 있기 때문이다. 가능한 any\_condjump\_p 를 대신 사용하라.

```
int any_condjump_p        PARAMS ((rtx));
```

Insn 가 conditional jump 이면 true 를 반환한다. 이 함수는 PARALLEL 내에 설정한 PC 를 포함하는 명령어들을 위해 동작한다. 이 명령어는 여러 다른 작용을 가지고 있기 때문에 jump 를 제거하기 전에 당신은 반드시 onlyjump\_p 를 검사하여야 한다.

condjump\_p 와 달리 unconditional jump 들에 대해 false 를 반환한다는 사실을 알기 바란다.

```
int any_uncondjump_p          PARAMS ((rtx));
```

Insn 가 unconditional direct jump 이고 가능한 PARALLEL 내부에 번들로 자리잡고 있을 경우 true 를 반환한다.

```
rtx pc_set                    PARAMS ((rtx));
```

PC 의 설정을 반환하고 그렇지 않다면 NULL 을 반환한다.

```
rtx condjump_label           PARAMS ((rtx));
```

Conditional jump 의 label 을 반환한다.

```
int simplejump_p             PARAMS ((rtx));
```

만약 INSN 가 unconditional jump 이고 그 이상이 아닐 경우 0 을 반환한다.

```
int returnjump_p            PARAMS ((rtx));
```

아직 자세한 설명이 없음.

```
int onlyjump_p               PARAMS ((rtx));
```

만약 INSN 가 오직 control 만 전달하는 jump 이고 그 이상이 아닐 경우 true 을 반환한다.

```
int only_sets_cc0_p          PARAMS ((rtx));
```

만약 X 가 condition code 들만 설정하고 더이상의 영향이 없는 RTX 일 경우 0 이 아닌 값을 반환한다.

```
int sets_cc0_p               PARAMS ((rtx));
```

만약 X 가 아무것도 하지 않지만 condition code 들을 설정하고, CLOBBER 혹은 USE register 들을 설정하는 RTX 일 경우 1 을 반환한다. 만약 X 가 명시적으로 condition code 들을 설정하고 다른 것들을 수행할 경우 -1 을 반환한다.

```
int invert_jump_1            PARAMS ((rtx, rtx));
```

jump JUMP 의 condition 을 변환하고, 현재 jump 할 곳 대신에 label NLABEL 로 jump 하도록 만든다. Change group 내에 변화점을 추적한다. 만약 우리가 inversion 와 redirection 을 어떻게 실행하는지 보지 않을 경우 false 를 반환한다.

```
int invert_jump              PARAMS ((rtx, rtx, int));
```

jump JUMP 의 condition 을 변환하고, 현재 jump 할 곳 대신에 label NLABEL 로 jump 하도록 만든다. 만약 성공시 true 를 반환한다.

```
int rtx_renumbered_equal_p    PARAMS ((rtx, rtx));
```

rtx\_equal\_p 와 비슷하지만 두 REG 들이 같은 값을 renumber 할 경우 같은 것으로 고려하거나 만약 operand 들의 order 가 반대로 되었다면 같은 것으로 두 commutative operation 들을 고려한다는 점이 다르다.

```
int true_regnum          PARAMS ((rtx));
```

만약 X 가 hard register 이거나 그것과 동일한 것 혹은 그것의 subregister 일 경우 hard register number 을 반환한다. 만약 X 가 hard register 로써 할당되지 않은 pseudo register 일 경우, pseudo register number 를 반환한다. 그렇지 않을 경우 -1 을 반환한다. 어떠한 rtx 도 X 에 대해 유효하다.

```
int redirect_jump_1     PARAMS ((rtx, rtx));
```

현재 jump 하고자 하는 곳 대신 NLABEL 로의 JUMP 를 만든다. Change group 내에 수정 사항을 축적하고. 만약 우리가 그것이 어떻게 되는지 보길 원하지 않는다면 false 를 반환한다.

```
int redirect_jump       PARAMS ((rtx, rtx, int));
```

현재 jump 하고자 하는 곳 대신 NLABEL 로의 JUMP 를 만든다. 만약 old jump target label 이 결과로써 사용되지 않았다면, 그것과 그것을 따르는 code 는 아마 삭제될 것이다.

만약 NLABEL 가 zero 일 경우, 우리는 jump 를 (가능한 conditional) RETURN insn 로 변경할 것이다.

반환 값은 만약 변경 사항이 있을 경우 1 을, 만약 그렇지 않다면 0 일 것이다. (이것은 NLABEL == 0 에 대해서만 일어날 수 있다.)

```
void rebuild_jump_labels PARAMS ((rtx));
```

Jump optimizer 로의 동일한 입구. 이 entry point 는 오직 jumping insn 들내 JUMP\_LABEL field 를 재구성하고 non-jumping 명령어들내 REG\_LABEL note 들을 재구성하기 위해서 이다.

```
enum rtx_code reversed_comparison_code PARAMS ((rtx, rtx));
```

Rtx 표현식으로 COMPARISON 을 가지는 이전 함수 주위를 싸는 wrapper. 이것은 많은 caller 들을 간단화한다.

```
enum rtx_code reversed_comparison_code_parts PARAMS ((enum rtx_code,
                                                    rtx, rtx, rtx));
```

주어진 comparison (CODE ARG0 ARG1), insn 내부에 있는, INSN 들은 만약 거꾸로 비교 하는 것이 가능할 경우 그에 대한 code 를 반환한다. 그렇지 않으면 UNKNOWN 반환한다. UNKNOWN 는 우리가 CC\_MODE compare 를 가지고 있는 경우나 우리가 그것의 source 가 floating point 혹은 integer comparison 인지 할지 못할 때 반환될 것이다. Machine description 는 이러한 경우들에 대해 overhead 를 피하기 위해서 이 함수를 도와주는 뜻으로 반드시 REVERSIBLE\_CC\_MODE 와 REVERSE\_CONDITION macro 들을 선언하여야 한다.

```
void delete_for_peephole PARAMS ((rtx, rtx));
```

포괄적으로 FROM 부터 TO 까지 insn 들의 range 를 삭제한다. 이것은 peephole optimization 의 이득을 위한 것으로 이러한 insn 들이 수행되든 되지 않든 그들을 대체할 새 peephole insn 에 의해 수행될 때까지 이루어질 것이다.

```
int condjump_in_parallel_p PARAMS ((rtx));
```

만약 INSN 가 PARALLEL 내 (가능한) conditional jump 일 경우 0 이 아닌 값을 반환한다.

이 함수의 사용은 반대 의견이 있는데, 우리가 combined branch 와 compare insn 들을 지원해야 하기 때문이다. 가능한 대신 any\_condjump\_p 를 사용하라.

```
void never_reached_warning PARAMS ((rtx, rtx));
```

우리는 INSN 가 절대 도달하지 않는지를 결정하고 그것을 삭제하기 위해 이야기 한다. 만약 user 가 그것을 요철할 경우 경고 메시지를 출력한다.

이 경고를 더욱 더 유용하기 만들기 위해서 이것은 도달되지 않는 basic block 당 한번만 호출 되도록 해야 하고 basic block 이 현재 함수로 부터 한 줄이상 , 적어도 한 operation 이상을 포함하고 있을 때만 경고한다. CSE 와 inlining 는 insn 들을 중복할 수 있어서 이것으로 부터 가짜 경고들을 발생할 소지가 있다.

```
void purge_line_number_notes    PARAMS ((rtx));
```

아직 정확한 설명이 없음.

```
void copy_loop_headers          PARAMS ((rtx));
```

아직 정확한 설명이 없음.

## 6.20 loop.c

```
void init_loop                  PARAMS ((void));
```

아직 정확한 설명이 없음.

```
rtx libcall_other_reg          PARAMS ((rtx, rtx));
```

무슨 reg 들이 INSN 로 끝나는 libcall block 으로 참조되는지, 동일한 값으로써 언급된 것들로부터 검사한다. 만약 없을 경우, 0 을 반환한다. 만약 1 개 이상일 경우, 그것을 모두 포함하는 EXPR\_LIST 를 반환한다.

```
void loop_optimize              PARAMS ((rtx, FILE *, int));
```

이 file 의 entry point. 현재 함수에 대한 loop 최적화를 수행한다. F 는 함수의 첫번째 insn 이고 DUMPFILER 는 주어진 action 들의 trace 의 output 을 위한 stream 이다 ( 혹은 output 할 필요가 없다면 0.)

```
void record_excess_regs         PARAMS ((rtx, rtx, rtx *));
```

IN\_THIS 에서 언급된 모든 pseudo-regs 를 기록하기 위해 \*OUTPUT 에 element 들을 더한다. 하지만 NOT\_IN\_THIS 에서 언급되지 않은 것이어야 한다.

## 6.21 predict.c

```
void invert_br_probabilities     PARAMS ((rtx));
```

INSN 내 모든 branch prediction 들 혹은 probability note 들을 뒤집는다. 이것은 jump 에 의해 사용된 condition 을 우리가 뒤집기 위해 매번 수행할 때 필요하다.

```
bool expensive_function_p       PARAMS ((int));
```

만약 function 이 비용이 비싼것처럼 보여, code size 의 증가 비용에서 prologue 혹은 epilogue, do inlining 의 performance 를 최적화하기 위한 point 가 존재하지 않는다면 true 를 반환한다. THRESHOLD 는 명령어들의 갯수 한계인데, 함수는 expensive 로 고려되고 있지 않는 평균적인 것들을 실행할 수 있다.

## 6.22 print-rtl.c

```
void debug_rtx          PARAMS ((rtx));
```

X 가 어떻게 보이는지를 알기 위해 디버거로부터 이 함수를 호출한다.

```
void debug_rtx_list    PARAMS ((rtx, int));
```

X 상의 list 를 출력하기 위해 이 함수를 호출한다.

N 은 출력할 rtx 들의 갯수이다. 양수값들은 지정된 rtx 상에서 부터 출력한다. 음수값들은 rtx 주위 window 를 출력한다. EG: -5 는 양쪽 의 2 rtx 들을 출력한다. (지정된 rtx 를 포함하여).

```
void debug_rtx_range   PARAMS ((rtx, rtx));
```

START 에서 END 까지 포괄적인 rtx list 를 출력하기 위해 이 함수를 호출한다.

```
rtx debug_rtx_find     PARAMS ((rtx, int));
```

insn uid UID 를 가진 것을 찾기 위해 rtx list 를 검색하는데 이 함수를 호출하며 DEBUG\_RTX\_COUNT 를 사용하고, 또한 그것을 출력하기 위해 debug\_rtx\_list 를 호출한다. 찾아진 insn 는 나중에 디버깅 분석을 가능하기 위해 반환된다.

```
void print_mem_expr    PARAMS ((FILE *, tree));
```

아직 정확한 설명이 없음.

```
void print_rtl         PARAMS ((FILE *, rtx));
```

File OUTF 상에서 RTX\_FIRST 로 시작하는 insn 들의 chain 을 출력하기 위한 외부 entry point. 공백줄은 insn 들을 분리한다.

만약 RTX\_FIRST 가 insn 이 아니면, 홀로 출력되며 새 줄을 가지지 않는다.

```
void print_simple_rtl  PARAMS ((FILE *, rtx));
```

print\_rtl 와 비슷하지만 모든 자세한 내용을 출력하지 않는다; 예를 들어 만약 RTX 가 CONST\_INT 이면 그냥 decimal format 만 출력한다.

```
int print_rtl_single   PARAMS ((FILE *, rtx));
```

print\_rtx 와 비슷하지만, 파일을 지정한다. 만약 우리가 실제로 어떤 것을 출력했다면 0 이 아닌 값을 반환한다.

```
void print_inline_rtx  PARAMS ((FILE *, rtx, int));
```

FILE 의 현재 줄의 rtx 를 출력한다. 초기에 IND 문자들을 들여쓰기 한다.

## 6.23 profile.c

```
void init_branch_prob  PARAMS ((const char *));
```

Branch-prob processing 을 위한 file-level 초기화를 수행합니다.

```
void branch_prob       PARAMS ((void));
```

Program flow graph 에 기반한 instrument 와/혹은 analyze program behavior. 각 경우에 대해, 이 함수는 컴파일 되어지고 있는 함수를 위한 flow graph 를 생성한다. Flow graph 는 BB.GRAPH 내에 저장된다.

FLAG\_PROFILE\_ARCS 가 0 이 아닐 때, 이 함수는 flow graph 의 dynamic behavior 를 재건설하는데 필요한 flow graph 내 edge 들을 갱신한다.

FLAG\_BRANCH\_PROBABILITIES 가 0 이 아닐 때, 이 함수는 컴파일되었던 이전 함수의 실행으로 부터 edge count information 를 포함하는 data file 에서 추가적인 정보를 읽는다. 이 경우, Flow graph 는 실제 execution count 들로 주석이 달리는데, 이것은 나중에 최적화 목적으로 rtl 내에 전해질 것이다.

이 파일의 main entry point.

```
void end_branch_prob          PARAMS ((void));
```

Branch-prob processing 가 완료된 후 file-level cleanup 을 실행한다.

```
void output_func_start_profiler PARAMS ((void));
```

만약 이것이 이전에 수행된 적이 없을 경우 \_bb\_init\_func 와 연루될 constructor 를 위한 output code.

## 6.24 read-rtl.c

```
void traverse_md_constants    PARAMS ((int (*) (void **, void *),
                                     void *));
```

모든 constant 정의에 대해, 두 argument 로 CALLBACK 을 호출한다: Constant 정의와 INFO 를 가르키는 pointer. CALLBACK 이 0 을 반환할 경우 멈춘다.

```
int read_skip_spaces        PARAMS ((FILE *));
```

Non-whitespace char 까지 INFILE 로 부터 읽어서 그것을 반환한다. List 스타일과 C 스타일의 comment 들은 whitespace 로써 취급된다. genflags 와 같은 tool 들이 이 함수를 사용한다.

```
rtx read_rtx                PARAMS ((FILE *));
```

INFILE 에서 printed representation 형태의 rtx 를 읽고 그에 따라 만들어진 core 를 실제 rtx 형태로 반환한다. read\_rtx 는 컴파일러내에서는 제대로 사용되지 않지만 machine description 들로부터 C code 를 만드는 유틸리티 gen\*.c 에서는 많이 사용된다.

```
void fancy_abort PARAMS ((const char *, int, const char *))
```

Obstack 이 연결되어 있는 경우와 abort 가 fancy\_abort 에 정의되어 있을 때 default entry 를 제공한다.

## 6.25 recog.c

```
rtx *find_constant_term_loc  PARAMS ((rtx *));
```

주어진 rtx \*P 가 만약 integer constant term 을 포함하는 합일 경우, 해당 constant term 로의 pointer 의 location (type rtx \*) 반환한다. 그렇지 않다면, null pointer 를 반환한다.

```
int asm_operands            PARAMS ((rtx));
```

만약 BODY 가 ASM\_OPERANDS 를 사용하는 insn body 이라면, insn 내 operand 들의 (input 과 output 양쪽 다) 갯수를 반환한다. 그렇지 않다면 -1 을 반환한다.

```
const char *decode_asm_operands PARAMS ((rtx, rtx *, rtx **,
                                         const char **,
                                         enum machine_mode *));
```

BODY 가 ASM\_OPERANDS 를 사용하는 insn body 라고 가정하여, vector OPERANDS 에는 그것의 operand 들 (input 과 output 양쪽) 을 복사하고, OPERAND\_LOCS 에는 insn 내 operand 들의 위치를 복사하고, CONSTRAINTS 에는 operand 들을 위한 constraint 들을 복사한다. MODES 에는 operand 들의 mode 들을 쓴다. assembler-template 을 반환한다.

만약 MODES 혹은 OPERAND\_LOCS, CONSTRAINTS, OPERANDS 가 0 일 경우 우리는 그러한 정보를 저장하지 않는다.

```
enum reg_class reg_preferred_class PARAMS ((int));
```

Pseudo reg number REGNO 가 최적으로 할당되어 있는 reg\_class 를 반환한다. 이 함수는 info 가 계산되기 전에 때때로 호출된다. 그러한 것이 일어날 때 마다 단순히 무해한 GENERAL\_REGS 를 반환한다.

```
enum reg_class reg_alterate_class PARAMS ((int));
```

아직 자세한 설명이 없음.

```
rtx get_first_nonparm_insn PARAMS ((void));
```

'assign\_parms' 에 의해 생성되는 것들을 따르는 첫번째 insn 을 반환한다.

```
void split_all_insns PARAMS ((int));
```

함수내 모든 insn 들을 분리한다. 만약 UPD\_LIFE 이면, 끝난후 life info 를 update 한다.

```
void split_all_insns_noflow PARAMS ((void));
```

split\_all\_insns 와 같지만, 이용가능한 CFG 가 있다고 예상하지 않는다. machine dependent reorg pass 들에 의해 사용된다.

## 6.26 reg-stack.c

```
void reg_to_stack PARAMS ((rtx, FILE *));
```

"flat" register file usage 에서 "stack" register file 로 register usage 를 변환한다. FIRST 는 함수내 첫번째 insn 이고, FILE 은 사용될 경우 dump file 이다.

CFG 와 run life analysis 를 건설한다. 그런 다음, 각 insn 를 하나씩 변환한다. 만약 최적화 중일 경우, converter 가 edge 들상에 pop insn 들을 삽입했을 때 생성된 code duplication 을 제거하기 위해 마지막에 cleanup.cfg 단계를 실행한다.

## 6.27 regclass.c

```
enum machine_mode choose_hard_reg_mode PARAMS ((unsigned int,
                                                unsigned int));
```

Hard reg REGNO 에 합법적이고 nregs 를 저장하는데 충분히 큰 machine mode 를 반환합니다. 만약 우리가 적당한 것을 찾을 수 없다면 VOIDmode 를 반환합니다.

```
void free_reg_info PARAMS ((void));
```

allocate\_reg\_info 에 의해 할당된 공간을 자유롭게 한다.

```
int reg_classes_intersect_p      PARAMS ((enum reg_class, enum reg_class));
    만약 C1 과 C2 가 공통되는 register 가 있다면 0 이 아닌 값을 반환한다.

int reg_class_subset_p          PARAMS ((enum reg_class, enum reg_class));
    만약 C1 이 C2 의 subset 일 경우, 즉, 만약 C1 내 모든 register 가 C2 에 포함된다면 0 이 아닌
    값을 반환한다.

void globalize_reg              PARAMS ((int));
    Global 로써 register number I 를 기록한다.

void init_regs                  PARAMS ((void));
    Register set 들을 초기화하는 것을 마무리짓고 register mode 들을 초기화 합니다.

void init_reg_sets              PARAMS ((void));
    reg usage 에 관한 위의 데이터를 초기화 하기 위해 이 함수는 한번만 호출됩니다. 이 함수가 수
    행된 후 여러 switch 들이 덮여쓰여집니다.

void regset_release_memory      PARAMS ((void));
    Register set 들에 의해 할당된 어떠한 memory 를 풀어준다.

void regclass_init              PARAMS ((void));
    이 단계를 위한 몇몇 global data 를 초기화한다.

void regclass                    PARAMS ((rtx, int, FILE *));
    이것은 모든 명령어들을 scan 하고 각 pseudo-register 를 위한 preferred class 를 계산한다. 이
    정보는 'reg_preferred_class' 를 호출함으로써 나중에 접근될 수 있다. 이 단계는 local register
    allocation 바로 앞에 온다.

void reg_scan                    PARAMS ((rtx, unsigned int, int));
    아직 정확한 설명이 없음.

void reg_scan_update            PARAMS ((rtx, rtx, unsigned int));
    FIRST 부터 LAST 까지 insn 들을 살펴봄으로써 'regscan' 정보를 update 한다. 몇몇 새로운
    REG 들이 생성되고, OLD_MAX_REGNO 보다 큰 수를 가진 어떠한 REG 가 그러한 REG 이
    다. 우리는 단지 그러한 것 들에 대한 정보만 update 한다.

void fix_register                PARAMS ((const char *, int, int));
    NAME 으로 명명된 register 의 사용 특성을 지정한다. 만약 FIXED 이면 fixed register 이고
    CALL_USED 이면 call-used register 여야 한다.
```

## 6.28 regmove.c

```
void regmove_optimize           PARAMS ((rtx, int, FILE *));
    Register move optimization 을 위한 main entry. F 는 첫번째 명령어이다. NREGS 는 명령어에
    서 사용되는 가장 높은 pseudo-reg number 에 1 을 더한 값이다. REGMOVE_DUMP_FILE 은
    주어진 action 들의 trace 의 output 을 위한 stream 이다 (혹은 output 이 필요없다면 0).

void combine_stack_adjustments  PARAMS ((void));
    Stack adjustment combination 을 위한 main entry point.
```

### 6.29 regrename.c

```
void regrename_optimize          PARAMS ((void));
```

현재 함수상에서 register renaming 을 수행한다.

```
void copyprop_hardreg_forward    PARAMS ((void));
```

Forward copy propagation optimization 를 위한 main entry point.

### 6.30 reorg.c

```
void dbr_schedule                PARAMS ((rtx, FILE *));
```

Delay slot 들에 놓일 insn 들을 찾기 시도한다.

### 6.31 rtl.c

```
rtx rtl_alloc                   PARAMS ((RTL_CODE));
```

code CODE 를 가지는 rtl 을 할당합니다. CODE 는 rtl 에 저장됩니다; 나머지 모든 값들은 0 으로 초기화됩니다.

```
rtvec rtlvec_alloc              PARAMS ((int));
```

N element 만큼의 rtl vector 를 할당합니다. 길이를 저장하며 모든 element 들을 0 으로 초기화합니다.

```
rtx copy_rtl                    PARAMS ((rtl));
```

Rtl 의 새 복사본을 생성한다. 재귀적으로 rtl 의 operand 들을 복사하며, 공유가능한 소수의 rtl code 들에 대해서는 제외한다.

```
rtx copy_most_rtl               PARAMS ((rtl, rtl));
```

'copy\_rtl' 와 비슷하지만 만약 MAY\_SHARE 가 존재한다면, 복사된다기 보다는 바로 결과로 자리잡을 것이다. MAY\_SHARE 는 MEM 들의 EXPR\_LIST 의 MEM 이다.

```
rtx shallow_copy_rtl            PARAMS ((rtl));
```

Rtl 의 새 복사본을 생성한다. 오직 한 level 만 복사한다.

```
int rtl_equal_p                  PARAMS ((rtl, rtl));
```

만약 X 와 Y 가 identical-looking rtl 들일 경우 1 을 반환한다. 이것은 rtl argument 들에 대한 Lisp function EQUAL 이다.

### 6.32 rtlanal.c

```
int rtl_addr_can_trap_p         PARAMS ((rtl));
```

만약 MEM 내 address 로써 X 의 사용이 trap 을 유발할 수 있다면 0 을 반환한다.

```
int rtl_unstable_p              PARAMS ((rtl));
```

만약 X 의 값이 unstable (은 프로그램의 다른 관점에서 달라질 수 있다.) 일 경우 1 을 반환한다. frame pointer, arg pointer, 기타 등등은 (한 함수내에서) stable 로써 고려되고 'unchanging' 로 mark 될 것이다.

```
int rtx_varies_p          PARAMS ((rtx, int));
```

만약 X 가 프로그램의 두 실행사이에 변화될 수 있는 값을 가지고 있다면 1 을 반환한다. 0 은 X 가 특정 constant 들 혹은 near-constant 들에 대하여 각자 비교될 수 있음을 의미한다. FOR\_ALIAS 는 만약 우리가 alias analysis 로 부터 호출되었다면 0 이 아닌 값을 가지는데, 만약 값이 0 일 경우, 우리는 좀 더 conservative 하다. frame pointer 와 arg pointer 는 constant 로 고려된다.

```
int rtx_addr_varies_p    PARAMS ((rtx, int));
```

만약 X 가 memory location 의 address 가 constant address 들과 각자 비교될 수 없는 memory location 을 참조하거나 X 가 BLKmode memory object 를 참조한다면 1 을 반환한다. FOR\_ALIAS 는 만약 우리가 alias analysis 로 부터 호출되었다면 0 이 아닌 값을 가지는데, 만약 값이 0 일 경우, 우리는 좀 더 conservative 하다.

```
HOST_WIDE_INT get_integer_term  PARAMS ((rtx));
```

만약 X 내 integer term 의 값이 분명한 경우 그 값을 반환하고 그렇지 않다면 0 을 반환한다. 오직 분명한 integer term 들만 탐지되며, 이것은 'related.value' field 를 가진 cse.c 내에서 사용된다.

```
rtx get_related_value      PARAMS ((rtx));
```

만약 X 가 상수 일 경우, 분명한 integer term 없는 값을 반환하고 그렇지 않다면 0 을 반환한다. 오직 분명한 integer term 들만 탐지된다.

```
rtx get_jump_table_offset  PARAMS ((rtx, rtx *));
```

주어진 tablejump insn INSN 는 jump table 내 offset 을 위한 RTL expression 을 반환한다. 만약 offset 이 결정되어질 수 없다면 NULL\_RTX 을 반환한다.

만약 EARLIEST 가 0 아닐 경우, offset 을 위치시키는데 사용된 가장 최근의 insn 가 발견된 장수를 가르키는 pointer 이다.

```
int reg_mentioned_p      PARAMS ((rtx, rtx));
```

만약 register REG 가 IN 내의 어떤 곳에 보인다면 0 이 아닌 값을 반환한다. 또한 만약 REG 가 register 가 아닌 경우도 작동하는데, 이러한 경우, REG 로 Lisp "equal" 인 IN 의 하위표현식을 검사한다.

```
int count_occurrences    PARAMS ((rtx, rtx, int));
```

X 내에서 FIND 가 보이는 곳의 갯수를 반환한다. 만약 COUNT\_DEST 가 0 일 경우 우리는 SET 의 destination 내에서 발생하는 것은 세지 않는다.

```
int reg_referenced_p     PARAMS ((rtx, rtx));
```

만약 X, register, 의 오래된 값 이 BODY 내에서 참조되었다면 0 이 아닌 값을 반환한다. 만약 X 가 전체적으로 새로운 값으로 대체되고 오직 SET\_DEST 로써 사용될 경우 우리는 그것을 reference 로 간주하지 않는다.

```
int reg_used_between_p   PARAMS ((rtx, rtx, rtx));
```

Register REG 가 FROM\_INSN 와 TO\_INSN (이 둘 모두를 포괄하는) 사이 insn 내에서 사용되었다면 0 이 아닌 값을 반환한다.

```
int reg_referenced_between_p  PARAMS ((rtx, rtx, rtx));
```

Register REG 가 FROM\_INSN 와 TO\_INSN (이 둘 모두를 포괄하는) 사이 insn 내에서 참조되었다면 0 이 아닌 값을 반환한다. REG 의 설정은 세지 않는다.

```
int reg_set_between_p          PARAMS ((rtx, rtx, rtx));
```

만약 register REG 가 FROM\_INSN 와 TO\_INSN (이 둘 모두를 포괄하는) 사이 insn 내에서 set 되거나 clobber 되었다면 0 이 아닌 값을 반환한다.

```
int commutative_operand_precedence PARAMS ((rtx));
```

OP, commutative operation 의 operand, 가 첫번째 혹은 두번째 operand 로 우선시 되는지 아닌지를 가르키는 값을 반환한다. 값이 높으면 높을 수록, 첫번째 operand 를 우선시하는 것이 강하다. 우리는 first operand 를 위한 우선권을 지시하기 위해서는 음수를 사용하고 second operand 에 대해서는 양수를 사용한다.

```
int swap_commutative_operands_p PARAMS ((rtx, rtx));
```

만약 expression 을 canonicalize 하기 위해 commutative operation 의 operand 들을 swap 할 필요가 있을 경우 1 을 반환한다.

```
int regs_set_between_p        PARAMS ((rtx, rtx, rtx));
```

reg\_set\_between\_p 와 비슷하지만, X 내 모든 register 들을 검사하는 것이 다르다. 오직 그들 중 아무것도 START 와 END 사이에 수정되지 않았을 경우에만 0 을 반환한다. non-register 들을 다른 방식으로 고려하지는 않는다.

```
int modified_between_p        PARAMS ((rtx, rtx, rtx));
```

reg\_set\_between\_p 와 비슷하지만, X 내 모든 register 들을 검사한다. 오직 그들 중 아무것도 START 와 END 사이에 수정되지 않았을 경우에만 0 을 반환한다. 만약 X 가 MEM 을 포함할 경우에 1 을 반환하는데, 이 routine 은 어떠한 memory aliasing 를 수행하지 않는다.

```
int no_labels_between_p       PARAMS ((rtx, rtx));
```

만약 BEG 와 END 사이, 포괄적인 BEG 와 END, 에 CODE\_LABEL insn 이 없다면 1 을 반환한다.

```
int no_jumps_between_p        PARAMS ((rtx, rtx));
```

만약 BEG 와 END 사이, 포괄적인 BEG 와 END, 에 JUMP\_INSN insn 가 없다면 1 을 반환한다.

```
int insn_dependent_p          PARAMS ((rtx, rtx));
```

만약 insn X 내 어떤것이 INSN Y 내의 어떤것에 (anti,output,true) 의존할 경우 true 을 반환한다.

```
int reg_set_p                  PARAMS ((rtx, rtx));
```

reg\_set\_between\_p 의 내부 수행자.

```
int modified_in_p              PARAMS ((rtx, rtx));
```

reg\_set\_p 과 비슷하지만, X 내 모든 register 들을 검사한다. 그들중 어떤 것도 INSN 내에서 수정되지 않을 경우에만 0 을 반환한다. 이 routine 은 어떠한 memory aliasing 를 수행하지 않는다.

```
rtx single_set_2          PARAMS ((rtx, rtx));
```

주어진 INSN 에서 만약 insn 가 오직 single SET 만 가지고 있다면 SET expression 을 반환한다. 그것은 또한 CLOBBER 들 혹은 USE 들, SET 을 가질 수 있는데, 이것들의 output 은 사용되지 않을 것이며, 우리가 무시할 수 있다.

```
int multiple_sets        PARAMS ((rtx));
```

주어진 INSN 에서, 만약 하나의 SET 이상을 가지고 있다면 0 이 아닌 값을 반환하고 아니면 0 을 반환한다.

```
int set_noop_p           PARAMS ((rtx));
```

만약 SET 의 destination 이 source 와 같고 부과적인 영향이 없다면 0 이 아닌 값을 반환한다.

```
int noop_move_p          PARAMS ((rtx));
```

만약 insn 가 SET 들로만 이루어지고, 각각이 그 자신의 값들만 설정한다면 0 이 아닌 값을 반환한다.

```
rtx find_last_value      PARAMS ((rtx, rtx *, rtx, int));
```

X 가 \*PINSN 전부터 할당되었던 마지막 것을 반환한다. 만약 VALID\_TO 가 NULL\_RTX 가 아니면 object 가 VALID\_TO 까지 수정되지 않았는지를 검사한다. 만약 object 가 수정되었다면, 만약 우리가 X 로 partial assignment 를 hit 하거나 처음으로 CODELABEL 을 hit 했다면, X 를 반환한다. 만약 우리가 assignment 를 발견하면 그것을 가르키게 \*PINSN 를 업데이트 한다. ALLOW\_HWREG 는 만약 hardware register 들이 src 가 될 수 있음을 허락하면 1 로 설정된다.

```
int refers_to_regno_p    PARAMS ((unsigned int, unsigned int,
                                rtx, rtx *));
```

만약 range [REGNO, ENDREGNO) 내 register 가 X 내에서 묵시적으로나 명시적으로 내에 저장되어 있는 것 보다 나타난다면 0 이 아닌 값을 반환한다.

LOC 의 substructure 내에 포함되어 있는 reference 들은 세지 않는다. LOC 는 아마 0 일 것이고, 어떤 것도 무시하지 않는다는 의미이다.

```
int reg_overlap_mentioned_p  PARAMS ((rtx, rtx));
```

만약 X 를 수정하는 것이 IN 에게 영향 미칠 것같으면 0 이 아닌 값을 반환한다. 만약 X 가 register 혹은 SUBREG 이면, 우리는 X 내 어떠한 register number 가 relevant register number 들과 충돌하는지를 검사한다. 만약 X 가 constant 이면 0 을 반환한다. 만약 X 가 MEM 이면, IN 이 MEM 을 포함하고 있을 경우에 대해 1 을 반환한다. (우리는 memory address 들이 충돌도 나지 않을까 걱정할 수 있는데, 이러한 경우는 매우 드물다고 예상하기 때문에 검사하지 않는다.)

```
rtx set_of                PARAMS ((rtx, rtx));
```

주어진 INSN 는 PAT 을 수정하는 SET 혹은 CLOBBER expression 를 반환한다. (직접적으로나 STRICT\_LOW\_PART 와 작은 modifier 들을 통해서)

```
void note_stores          PARAMS ((rtx,
                                void (*) (rtx, rtx, void *),
                                void *));
```

X 내에 저장되어 있거나, X 에 의해 clobber 되어있는 각 register 혹은 MEM 에 대해 FUN 을 호출한다. (X 는 insn 의 pattern 일 것이다.) FUN 은 두 argument 들을 받는다:

저장되어 있거나, clobber 된 REG 혹은 MEM, CC0,  
PC, 저장 역할을 하는 SET 혹은 CLOBBER rtx.

만약 저장되어 있거나 clobber 된 item 이 hard register 의 SUBREG 이면, SUBREG 가 건네질 것이다.

```
void note_uses          PARAMS ((rtx *,
                                void (*) (rtx *, void *),
                                void *));
```

notes\_stores 와 비슷하지만 PBODY, insn 의 PATTERN 을 가르키는 pointer, 내 참조되는 각 표현식에 FUN 을 호출하는 것이 다르다. 우리는 각 표현식에 오직 FUN 만 호출하지 어떤 내부 subexpression 들에 대해서는 하지 않는다. FUN 은 표현식으로서의 pointer 을 받고 DATA 는 이 함수로 건네진다.

이것은 reg\_referenced\_p 내에서 이루어지는 것과 같은 test 가 전혀 아닌데, 그것은 우리가 하지 않지만, 부분적으로 설정되고 있을 경우, 어떤 것이 참조되었음으로 간주하기 때문이다.

```
rtx reg_set_last      PARAMS ((rtx, rtx));
```

REG 가 INSN 이전에 설정되었는데, 그 마지막 값을 반환한다. 만약 우리가 그것을 쉽게 찾을 수 없을 경우 0 을 반환한다.

우리는 오직 REG 혹은 SUBREG, constant 만 반환하는데, 그것은 만약 MEM 가 변화되지 않고 남아있는지를 검사하기 너무 힘들기 때문이다.

```
int dead_or_set_p     PARAMS ((rtx, rtx));
```

만약 X 의 old content 들이 INSN 뒤에 살아남지 못한다면 0 이 아닌 값을 반환한다. 이것은 만약 X 가 (cc0) 이거나 X 가 register 이고 X 가 INSN 내에서 죽었거나 INSN 가 전체적으로 X 를 설정하기 때문일 때, true 일 것이다.

“전체적으로 설정” 이 의미하는 것은 직접적으로 설정하지 SUBREG 혹은 ZERO\_EXTRACT, SIGN\_EXTRACT 통해 하지 않는다는 뜻이고, 그래서 old content 들의 흔적이 남지 않는다. 그와 비슷하게 REG\_INC 는 count 하지 않는다.

REG 는 hard 혹은 pseudo reg 일 것이다. Renumbering 은 account 내에서 이루어 지지 않지만, 차이점이 없이 만들어 위해서 이 사용을 하는데, 이는 reg 들이 그들의 lifetime 동안 overlap 하지 않기 때문이다. 그렇기 때문이 이 함수는 death 들이 (flow.c) 에서 계산되었던 수 어떤 시간에 사용될 것이다.

만약 REG 가 다수의 machine register 들을 차지하는 hard reg 이라면 이 함수는 만약 그 register 들의 각각이 INSN 로 대체될 경우만 1 을 반환한다.

```
int dead_or_set_regno_p PARAMS ((rtx, unsigned int));
```

개별적인 register 를 검사하기 위한 dead\_or\_set\_p 용 utility function. 또한 flow.c 로부터 호출된다.

```
rtx find_reg_note     PARAMS ((rtx, enum reg_note, rtx));
```

만약 존재할 경우 insn INSN 내 kind KIND 의 reg-note 를 반환한다. 만약 DATUM 이 0 이 아닐 경우 그것의 datum 이 DATUM 인 것을 찾는다.

```
rtx find_regno_note   PARAMS ((rtx, enum reg_note,
                                unsigned int));
```

만약 register number REGNO 로 적용하는 것이 있을 경우, insn INSN 내 kind KIND 의 reg-note 를 반환한다. 만약 그러한 reg-note 가 없다면 0 을 반환한다. 이 NOTE 의 REGNO 가 만약 REGNO 가 hard register 일 경우 REGNO 일 필요가 없음을 알기 바란다. 이것은 note 가 REGNO 을 overlap 한 경우가 되겠다.

```
rtx find_reg_equal_equiv_note    PARAMS ((rtx));
```

만약 insn 가 오직 single set 과 하나의 note 를 가지고 있다면 REG.EQUIV 혹은 REG.EQUAL note 를 반환한다.

```
int find_reg_fusage              PARAMS ((rtx, enum rtx_code, rtx));
```

만약 kind CODE 의 종류의 DATUM 혹은 DATUM 의 어떤 overlap 이 INSN 의 CALL\_INSN\_FUNCTION\_USAGE information 에서 발견될 경우 true 를 반환한다.

```
int find_regno_fusage           PARAMS ((rtx, enum rtx_code,
                                         unsigned int));
```

만약 kind CODE 의 종류의 REGNO 혹은 REGNO 의 어떤 overlap 이 INSN 의 CALL\_INSN\_FUNCTION\_USAGE information 에서 발견될 경우 true 를 반환한다.

```
int pure_call_p                 PARAMS ((rtx));
```

만약 INSN 가 pure function 로의 call 이면 true 를 반환한다.

```
void remove_note                PARAMS ((rtx, rtx));
```

INSN 의 REG\_NOTES 로부터 register note NOTE 를 제거한다.

```
int volatile_refs_p             PARAMS ((rtx));
```

만약 X 가 UNSPEC\_VOLATILE operation 들 혹은 volatile ASM\_OPERANDS expression 들을 참조하는 어떤 volatile memory 를 포함하고 있다면 0 이 아닌 값을 반환한다.

```
int side_effects_p              PARAMS ((rtx));
```

위와 비슷하지만, 그것이 또한 register pre- 와 post- incrementing 을 거절하는 것이 다르다.

```
int volatile_insn_p             PARAMS ((rtx));
```

만약 X 가 어떠한 volatile instruction 들을 포함하고 있다면 0 이 아닌 값을 반환한다. 이것들은 unpredictable machine state instruction 들을 유발할수 있는 명령어들이고 그러인해 어떠한 명령어들도 그들 사이트로 move 되거나 combine 되어서는 안된다. 이것은 오직 volatile asm 들과 UNSPEC\_VOLATILE instruction 들만 포함한다.

```
int may_trap_p                  PARAMS ((rtx));
```

만약 rtx X 를 평가하는것이 trap 을 유발할 수 있다면 0 이 아닌 값을 반환한다.

```
int inequality_comparisons_p    PARAMS ((rtx));
```

만약 X 가 EQ 혹은 NE 가 아닌, 즉 inequality, 비교를 포함하고 있다면 0 이 아닌 값을 반환한다.

```
rtx replace_rtx                 PARAMS ((rtx, rtx, rtx));
```

X 내에서 FROM 의 어떤 발생을 TO 로 대체한다. 함수는 대체를 위해 CONST\_DOUBLE 내로 진입하지 않는다.

이 복사는 이루어지지 않으며 모든 copy 들이 수정되지 않는한 X 는 반드시 공유되어서는 안된다. 참고하라.

```
rtx replace_regs          PARAMS ((rtx, rtx *, unsigned int,
                                int));
```

Rtx X 을 통해서, REG\_MAP 에 따라 많은 register 들을 대체한다. (변화된 내용을 가지는 X 일 수 있는) X 을 위한 대체물을 반환한다. REG\_MAP[R] 는 register R 을 위한 대체물이고 대체하지 않는다면 0 이다. NREGS 는 REG\_MAP 의 길이이며, regs  $i$ = NREGS 들은 map 되지 않는다.

우리는 오직 REG 혹은 SUBREG 의 REG\_MAP entry 들만 지원한다. 또한 hard register 들은 pseudo 들로 map 되어서는 안되며 반대로도 안된다. 왜냐하면 validate\_change 가 호출되지 않았기 때문이다.

만약 REPLACE\_DEST 가 1 이면, replacement 들은 또한 destination 들내에서 수행될 것이다. 그렇지 않다면 오직 source 들만 대체된다.

```
int computed_jump_p      PARAMS ((rtx));
```

만약 INSN 가 indirect jump (즉 computed jump) 일 경우 0 이 아닌 값을 반환한다.

Tablejump 들과 casesi insn 들은 indirect jump 들로 고려되지 않는데, 우리는 그들은 (use (label.ref)) 로 그들을 인식할 수 있다.

```
int for_each_rtx         PARAMS ((rtx *, rtx_function, void *));
```

Depth-first search 를 통해 X 를 탐사하는데, 각 sub-expression (X 자신도 포함하여) 마다 F 를 호출한다. F 는 또한 DATA 로도 전달될 수 있다. 만약 F 가 -1 을 반환하면, sub-expression 들을 탐사하지 않지만 tree 의 나머지 로의 탐사는 계속한다. 만약 F 가 어떤 0 이 아닌 값을 전혀 반환하지 않는다면, 탐사를 멈추고 F 에 의해 반환된 값을 반환한다. 그렇지 않다면 0 을 반환한다. 이 함수는 RTX\_EXPR 들을 포함하는 tree structure 내로 탐사하지 않고 sub-expression 들의 format code 가 실제로 RTL 인지 아닌지를 알 수 없는 경로 인해 '0' 인 sub-expression 들 내로도 탐사하지 않는다.

이 routine 은 매우 일반적이며, 이 파일의 다른 routine 들의 많은 부분을 수행하는데 사용될 수 (되어야) 있다.

```
rtx regno_use_in         PARAMS ((unsigned int, rtx));
```

REGNO 로의 어떤 참조를 위한 X 를 찾는데, 만약 발견될 경우 reference 의 rtx 을 반환하며 그렇지 않다면 NULL\_RTX 를 반환한다.

```
int auto_inc_p           PARAMS ((rtx));
```

만약 X 가 autoincrement side effect 이고 register 가 stack pointer 가 아니라면 1 을 반환한다.

```
int in_expr_list_p       PARAMS ((rtx, rtx));
```

Entry 의 첫번째 operand 가 NODE 인 것을 위한 LISTP (EXPR\_LIST) 를 찾고 발견될 경우 1 을 반환한다. 간단한 equality test 는 NODE 가 매치되는지 결정하는데 사용된다.

```
void remove_node_from_expr_list PARAMS ((rtx, rtx *));
```

Entry 의 첫번째 operand 가 NODE 인 것을 위한 LISTP (EXPR\_LIST) 를 찾고 발견될 경우 list 로부터 entry 를 제거한다.

간단한 equality test 는 NODE 가 매치되는지 결정하는데 사용된다.

```
int insns_safe_to_move_p      PARAMS ((rtx, rtx, rtx *));
```

만약 FROM 으로 시작한 instruction 들의 sequence 와 TO 을 포함하는 것들을 이동하기 안전할 경우 1 을 반환한다. 만약 NEW\_TO 가 NULL 이 아니고, sequence 가 이미 move 하기에 안전하지 않지만, 안전한 sequence 로 확장하기 쉬울 수 있을 경우, NEW\_TO 는 확장된 sequence 의 끝을 가르킬 것이다.

현재 이 함수는 오직 region 이 전체 exception region 들을 포함하는지만 검사하지만 또한 추가적인 condition 들을 검사하도록 확장되어 질 수 있다.

```
int loc_mentioned_in_p      PARAMS ((rtx *, rtx));
```

만약 IN 가 address LOC 를 가지는 rtl 의 한 조각을 포함하고 있을 경우 0 이 아닌 값을 반환한다.

```
rtx find_first_parameter_load  PARAMS ((rtx, rtx));
```

Load 된 첫번째 parameter 를 위해 backward 를 살펴본다. BOUNDARY 를 건너뛰지 않는다.

### 6.33 sibcall.c

```
void optimize_sibling_and_tail_recursive_calls PARAMS ((void));
```

주어진 (비어있을 수 있는) 잠재적 sibling 의 set 혹은 tail recursion call site 들은 이것이 최적화가 가능한지를 결정한다.

잠재적 sibling 혹은 tail recursion call 들은 CALL\_PLACEHOLDER insn 들로 mark 된다. CALL\_PLACEHOLDER insn 는 일반적인 call 혹은 sibling call, tail recursive call 를 수행하기 위한 insn 들의 chain 들을 잡고 있다.

CALL\_PLACEHOLDER 를 적당한 insn chain 로 대체한다.

```
void replace_call_placeholder  PARAMS ((rtx, sibcall_use_t));
```

CALL\_PLACEHOLDER 를 그것의 children 중 하나와 대체한다. INSN 는 CALL\_PLACEHOLDER insn 여야 한다; USE 는 어떤 child 를 사용할지 말한다.

```
int stack_regs_mentioned     PARAMS ((rtx insn));
```

만약 INSN 가 stacked register 들을 언급할 경우 0 이 아닌 값을 반환하며, 그렇지 않을 경우 0 을 반환한다.

### 6.34 simplify-rtx.c

```
rtx simplify_unary_operation  PARAMS ((enum rtx_code,
                                     enum machine_mode, rtx,
                                     enum machine_mode));
```

Unary operation CODE 의 output mode 가 input operand OP 를 가지는 MODE 일 때 이 unary operation CODE 를 간단화 작업을 한다. 여기서 input operand OP 의 mode 는 원래 OP\_MODE 였다. 만약 간단화 작업이 만들어질 수 없다면 0 을 반환한다.

```
rtx simplify_binary_operation PARAMS ((enum rtx_code,
                                     enum machine_mode, rtx,
                                     rtx));
```

result mode MODE 와 세개의 operand 들, OP0 과 OP1, OP2 를 가진 binary operation CODE 를 간단화한다. 만약 simplifications 이 가능하지 않을 경우 0 을 반환한다.

EQ 혹은 LT 와 같은 relational operation 들에 이것을 사용하지 마라. 대신 simplify\_relational\_operation 을 사용하라.

```
rtx simplify_ternary_operation  PARAMS ((enum rtx_code,
                                         enum machine_mode,
                                         enum machine_mode, rtx, rtx,
                                         rtx));
```

result mode MODE 와 세개의 operand 들, OP0 과 OP1, OP2 를 가진 operation , CODE 를 간단화한다. OP0\_MODE 는 그것이 constant 가 되기 전에 OP0 의 mode 였다. 만약 simplifications 이 가능하지 않을 경우 0 을 반환한다.

```
rtx simplify_relational_operation PARAMS ((enum rtx_code,
                                           enum machine_mode, rtx,
                                           rtx));
```

simplify\_binary\_operation 와 비슷하지만 relational operator 들을 위해 사용된다는 점에서 다르다. MODE 는 result 의 것이 아니라, operand 들의 mode 이다. 만약 MODE 가 VOIDmode 이면, 두 operand 들은 반드시 VOIDmode 이어야 하고 우리는 operand 들을 "infinite precision" 으로 비교한다.

만약 simplification 가 가능하지 않다면, 이 함수는 0 을 반환한다. 그렇지 않다면 const\_true\_rtx 혹은 const0\_rtx 을 반환한다.

```
rtx simplify_gen_binary        PARAMS ((enum rtx_code,
                                         enum machine_mode,
                                         rtx, rtx));
```

만약 표현식이 fold 인지를 보고 operand 를 적당히 순서화시킴으로써 binary operation 를 만든다.

```
rtx simplify_gen_unary        PARAMS ((enum rtx_code,
                                         enum machine_mode, rtx,
                                         enum machine_mode));
```

만약 그것이 fold 한것으로 보인다면 먼저 unary operation 를 만들고 그렇지 않을 경우 지정된 operation 을 만들어 낸다.

```
rtx simplify_gen_ternary      PARAMS ((enum rtx_code,
                                         enum machine_mode,
                                         enum machine_mode,
                                         rtx, rtx, rtx));
```

비슷하지만 ternary operation 들을 위한 것이다.

```
rtx simplify_gen_relational    PARAMS ((enum rtx_code,
                                         enum machine_mode,
                                         enum machine_mode,
                                         rtx, rtx));
```

relational operation 들과 비슷하지만 CMP\_MODE 는 mode 비교가 이루어 졌는지를 지정한다.

```
rtx simplify_subreg          PARAMS ((enum machine_mode,
                                     rtx,
                                     enum machine_mode,
                                     unsigned int));
```

SUBREG:OUTERMODE(OP:INNERMODE, BYTE) 를 간단화하고 만약 간단화가 가능하지 않을 경우 0 을 반환한다.

```
rtx simplify_gen_subreg     PARAMS ((enum machine_mode,
                                     rtx,
                                     enum machine_mode,
                                     unsigned int));
```

SUBREG operation 를 만들거나 그것이 fold 했을 경우 동일하다.

```
rtx simplify_replace_rtx   PARAMS ((rtx, rtx, rtx));
```

NEW 를 가진 X 에서 OLD 의 모든 발생들을 대체하고 결과인 RTX 를 간단화 시키려한다. 가능한 간단화된 새로운 RTX 를 반환한다.

```
rtx simplify_rtx           PARAMS ((rtx));
```

X, rtx expression 를 간단화시킨다.

간단화된 표현식을 반환하거나, 간단화가 가능하지 않을 경우 NULL 을 반환한다.

이것은 간단화 routine 들을 위한 우선되는 entry point 이지만; 우리는 여전히 좀 더 전문화된 routine 들을 호출하기 위한 단계들을 허용한다.

현재 GCC 는 code 의 통일을 위해 필요한 크게 세 부분의 (예스 3 개) RTL simplification 을 가지고 있다.

1. cse.c 파일 내 fold\_rtx. 이 code 는 RTL simplification 를 돕기 위해 여러 CSE 개별적인 정보를 사용한다
2. combine.c 파일 내 simplify\_rtx. fold\_rtx 와 비슷하지만 RTL simplification 를 돕기위한 combine 개별적인 정보를 사용한다.
3. 이 파일내 routine 들.

오랜 시간 동안 우리는 simplification code 가 하나의 body 만을 가지길 원하였다; 내가 다음의 단계에서 추천하는 이런 상태를 얻기 위해서 였다.

1. fold\_rtx & simplify\_rtx 를 엮어버리고 이러한 routine 들과 pass dependent state 가 아닌 어떤 simplification 들을 이동시킨다.
2. #1 에서 이동된 code 로써, fold\_rtx & simplify\_rtx 를 가능할 때 마다 이 code 를 사용하기 변경한다.
3. 이러한 routine 들에게 제공되어지는 pass dependent state 를 허락하고 pass dependent state 에 기반하는 simplification 들을 더한다. redundant/dead 가 되는 cse.c 와 combine.c 에서 code 를 제거한다.

비록 그것이 시간이 걸리겠지만, 결과적으로 컴파일러는 유지보수 하기 더욱 쉽고 향상된 기능을 제공해 줄 수 있을 것이다. 추가적으로 4 가지 장소에 (3 개의 RTL simplification 과 1 개의 tree simplification) simplification 을 더 더해야 한다면 그건 정말 바보같은 일 일 것이다.

```
rtx avoid_constant_pool_reference PARAMS ((rtx));
```

만약 X 각 constant pool 를 참조하는 MEM 일 경우, real 값을 반환하고 그렇지 않다면 X 를 반환한다.

### 6.35 stmt.c

```
void set_file_and_line_for_stmt PARAMS ((const char *, int));
```

현재 file 와 line 을 기록한다. emit\_line\_note 에 의해 호출된다.

```
void expand_null_return PARAMS ((void));
```

현재 함수로부터 return 하는 RTL 을 생성하는데, 값을 가지고 있지 않다. (즉, 우리는 어떤 값을 반환하는 것에 관해 어떤 것도 하지 않는다.)

```
void emit_jump PARAMS ((rtx));
```

다음 sequential instruction 로써 LABEL 에 unconditional jump 를 더한다.

```
int preserve_subexpressions_p PARAMS ((void));
```

만약 우리가 separate pseudo 들로써 sub-expression 들을 보존해야 한다면 0 이 아닌 값을 반환한다. 우리가 최적화중이 아니라면 절대 그렇게 하지 않는다. 우리는 항상 -fexpensive-optimizations 일 경우에만 한다.

그렇지 않다면, 우리는 오직 우리가 loop 의 “early” part 내에 있을 경우만 한다. 즉, loop 가 여전히 작은 것일 경우에 말이다.

### 6.36 unroll.c

```
int set_dominates_use PARAMS ((int, int, int, rtx, rtx));
```

만약 FIRST\_UID 가 REGNO 의 set 이고, FIRST\_UID 가 LAST\_UID 에 대해 우위에 있을 때 (예를 들면, FIST\_UID 는 항상 LAST\_UID 일 경우에 실행된다), 1 을 반환한다. 그렇지 않으면 0 을 반환한다. COPY\_START 는 insn 들 FIRST\_UID 와 LAST\_UID 를 찾기 시작할 수 있는 곳이다. COPY\_END 는 그러한 insn 들을 찾는 것을 멈추는 곳이다.

만약 LOOP\_START 와 FIRST\_UID 사이에 JUMP\_INSN 가 없다면, FIRST\_UID 가 반드시 LAST\_UID 에 우위에 서야 한다.

만약 FIRST\_UID 와 LAST\_UID 사이에 CODE\_LABEL 가 있다면, FIRST\_UID 는 LAST\_UID 에 우위를 서면 안될 것이다.

만약 FIRST\_UID 와 LAST\_UID 사이에 CODE\_LABEL 가 없다면, FIRST\_UID 는 반드시 LAST\_UID 에 우위를 서야 한다.

### 6.37 varasm.c

```
rtx immed_double_const PARAMS ((HOST_WIDE_INT,
                                HOST_WIDE_INT, enum machine_mode));
```

int 쌍으로 구성된 값을 위한 CONST\_DOUBLE 혹은 CONST\_INT 을 반환한다. 정수를 위해, I0 는 low-order word 이고 I1 은 high-order word 이다. 실수를 위해, I0 는 low address 를 가진 word 이고 I1 은 high address 를 가진 word 이다.

```
rtx mem_for_const_double PARAMS ((rtx));
```

주어진 constant rtx X 에서 이 constant 가 memory 내 위치한 곳을 위한 MEM 을 반환한다. 만약 아직 위치하지 않았다면 0 을 반환한다.

```
rtx force_const_mem PARAMS ((enum machine_mode, rtx));
```

주어진 constant rtx X 에서 그 값을 위한 memory constant 를 만들고 (혹은 찾고) memory 내 그것을 참조하는 MEM rtx 를 반환한다.

```
rtx get_pool_constant          PARAMS ((rtx));
```

주어진 constant pool SYMBOL\_REF 에서 대응하는 constant 를 반환한다.

```
enum machine_mode get_pool_mode_for_function  PARAMS ((struct function *, rtx));
```

자세한 설명이 없음.

```
rtx get_pool_constant_mark    PARAMS ((rtx, bool *));
```

주어진 constant pool SYMBOL\_REF 에서 대응하는 constant 와 그것이 output 되었는지 아닌지를 반환한다.

```
rtx get_pool_constant_for_function  PARAMS ((struct function *, rtx));
```

위와 비슷하지만 특정 함수의 constant pool 인 것에서 다르다.

```
enum machine_mode get_pool_mode  PARAMS ((rtx));
```

비슷하지만 mode 를 반환한다.

```
int get_pool_offset          PARAMS ((rtx));
```

비슷하지만 constant pool 내 offset 을 반환한다.

```
rtx simplify_subtraction     PARAMS ((rtx));
```

주어진 MINUS 표현식에서 양측에 같은 symbol 을 포함할 경우 그것을 간단화한다.

```
rtx output_constant_def      PARAMS ((tree, int));
```

Constant expression EXP 용으로 memory 내 constant data 으로의 reference 를 나타내는 rtx 를 반환한다.

만약 그러한 constant 를 위한 어셈블러 code 가 이미 output 되었다면, 그것을 참조하는 rtx 를 반환한다. 그렇지 않다면, memory 내 그러한 constant 를 output 하고 (혹은 나중에 위해 연기하거나) 그것을 위한 rtx 을 생성한다.

만약 DEFER 가 0 이 아니면, 문자열 constant 들의 output 은 연기되어질 수 있으며 모든 최적화가 끝난 함수내에 참조되는 것만 output 할 수 있다.

EXP 의 TREE\_CST\_RTL 는 해당 rtx 를 가르키게 설정된다. Constant 인 const\_hash\_table record 들은 이미 label string 들을 가지고 있다.

```
rtx immed_real_const        PARAMS ((tree));
```

EXP 에 의해 지정된 값을 위한 CONST\_DOUBLE rtx 을 반환하는데, 그 값은 반드시 REAL\_CST tree node 여야 한다.

```
int in_data_section         PARAMS ((void));
```

우리가 지금 data section 에 있는 지를 알아낸다.

```
void init_varasm_once       PARAMS ((void));
```

아직 정확한 설명이 없음.