GCC TREE
정수와 실수

정원교

2004년 6월 5일

목 차

제 1 절 26 주 문서를 시작하며

이 문서에서는 GCC TREE 에서 정수와 실수가 어떻게 다루어지는지에 대해 알아보도록 하겠습니다.

제 2 절 정수

GCC 에서는 정수의 경우 HOST_WIDE_INT 를 기준으로 정의되고 사용되는데, 이것은 각 machine 에 따라 달라질 수 있는 정수의 크기를 고려하여 그 조건에 맞게 컴파일시 정해지는 매크로 이다. 이것이 어떻게 정의되는지 아래에서 살펴보자.

2.1 HOST_WIDE_INT

이 매크로는 $prefix/gcc/hwint.h 파일에 정의되어 있다. HOST_BITS_PER_INT 와 HOST_BITS_PER_LONG 에 의존하여 매크로를 위한 정의들을 제공하고 있는데, 실질적으로 HOST_WIDE_INT 매크로의 정의를 담당하는 부분에서 고려되는 것은 MAX_LONG_TYPE_SIZE 와 LONG_TYPE_SIZE 매크로 이다.

결과적으로 'long long' 이 선언되는 host 가 있을 수 있고, 'long' 이 선언되는 host 가 있을 수 있으며, 'int' 가 선언되는 곳이 있다. 필자의 경우 'int' 로 선언이 된다.

hwint.h 파일을 보면 알겠지만, 이 파일이 제대로 정의가 되기 위해서는 autoconf 가 $prefix/gcc/auto-host.h 파일에 SIZEOF_* 관련 매크로를 지정해 주어야 한다. 즉 auto-host.h 파일은 $prefix/configure 가 실행되는 동안 생성되게 된다.

그리고 HOST_WIDE_INT 의 출력과 관련하여 여러 매크로를 또한 정의하는데, 정의되는 것들을 살펴보면 아래와 같은 것들이 존재한다.

- HOST_WIDE_INT_PRINT_DEC
- HOST_WIDE_INT_PRINT_UNSIGNED
- HOST_WIDE_INT_PRINT_HEX
- HOST_WIDE_INT_PRINT_DOUBLE_HEX

부과적으로는 HOST_WIDEST_INT 매크로에 대해서도 선언을 하는데, 아래와 같은 것이 선언되게 된다.

- HOST_BITS_PER_WIDEST_INT
- HOST_WIDEST_INT
- HOST_WIDEST_INT_PRINT_DEC
- HOST_WIDEST_INT_PRINT_UNSIGNED
- HOST_WIDEST_INT_PRINT_HEX

2.2 정수 상수

정수 상수의 경우, TREE node 로는 INTEGER_CST 로 정의한다는 것을 앞에서 이미 언급하였다. 그리고 실제 해당 node 의 경우, $prefix/gcc/tree.h 의 struct tree_int_cst 구조체에 정이되어 있다. 이 구조체에 대해 언급을 하면 아래와 같은 모습이다.

```
struct tree_int_cst
{
  struct tree_common common;
  rtx rtl;
  struct {
    unsigned HOST_WIDE_INT low;
```

```
    HOST_WIDE_INT high;
  } int_cst;
};
```

이미 앞 문서에서 언급을 한 구조체로써, 실제 정수가, int_cst 의 low, high 에 각각 정의된다는 것을 확인할 수 있을 것이다. 그리고 이러한 TREE node 에 대해 accessor 매크로에 대해서 이미 말하였다.

실제 TREE node 의 정수의 경우, low, high 부분으로 나누어 관리한다는 것을 알 수 있다.

비록 필자의 컴퓨터의 경우, int 두개를 사용하여 low 와 high 를 표현하지만, Constant folding 을 수행할 때는 4 개의 int 를 사용한다. INTEGER_CST node 들에 대해 constant folding 를 수행하기 위해서는 two-word arithmetic 이 요구되는데 우리는 그것을 4 word 에 two-word integer 를 표현함으로써, 양수처럼 각 word 에 HOST_BITS_PER_WIDE_INT / 2 bit 만 저장함으로써 이를 나타낸다. 결과적으로 Word 의 값은 LOWPART + HIGHPART * BASE 로 표현하는 것이다. 이러한 각 HOST_WIDE_INT 를 계산하기 위해서 몇몇 macro 를 정의하고 있는데, 아래와 같다.

```
#define LOWPART(x) \
  ((x) & (((unsigned HOST_WIDE_INT) 1 << (HOST_BITS_PER_WIDE_INT / 2)) - 1))
#define HIGHPART(x) \
  ((unsigned HOST_WIDE_INT) (x) >> HOST_BITS_PER_WIDE_INT / 2)
#define BASE ((unsigned HOST_WIDE_INT) 1 << HOST_BITS_PER_WIDE_INT / 2)
```

또한, 정수의 overflow 를 검사하기 위한 macro 또한 존재하는데,아래와 같다.

```
#define OVERFLOW_SUM_SIGN(a, b, sum) ((~((a) ^ (b)) & ((a) ^ (sum))) < 0)
```

> 우리는 A1 + B1 = SUM1 이 2 의 보수 연산을 사용하고 overflow 를 무시한다는 것을 안다. A 와, B, SUM 은 A1 와 B1, SUM1 과 같은 sign 을 각각 가지고 있다고 가정한다. 그럼 이것의 값이 0 이 아닐 때, addition 이 일어나는 동안 overflow 가 발생했다는 것을 알린다.
>
> Overflow 는 만약 A 와 B 가 같은 sign 을 가지고 있지만, A 와 SUM 의 부호가 다를 때 발생한다. sign 이 같은지 다른지 검사하기 위해서는 '^' 를 사용하고 sign 를 제거하려면 '< 0' 을 사용한다.

4 개의 int 를 사용한 constant folding 내에서의 정수 연산에 대해서 좀 더 자세히 보기 위해서는 fold-const.c 파일에 선언되어 있는 encode (), decode () 함수 및 *_double () 함수를 참조하기 바란다.

간단한 예제를 언급한다면, encode, decode 함수로 들 수 있으며 아래와 같은 모습의 operation 을 수행한다.

```
/* two-word integer 를 4 word 로 unpack 한다.
   LOW 와 HI 는 정수이며 두 'HOST_WIDE_INT' 조각이다.
   WORDS 는 HOST_WIDE_INT 들의 배열을 가르키고 있다.  */

static void
encode (words, low, hi)
     HOST_WIDE_INT *words;
     unsigned HOST_WIDE_INT low;
     HOST_WIDE_INT hi;
{
  words[0] = LOWPART (low);
  words[1] = HIGHPART (low);
  words[2] = LOWPART (hi);
  words[3] = HIGHPART (hi);
}

/* 4 word 들의 배열을 two-word integer 로 pack 한다.
   WORDS 는 word 들의 배열을 가르킨다.
```

```
   정수는 두 'HOST_WIDE_INT' 조각으로 각각 *LOW 와 *HI 에 저장된다.  */

static void
decode (words, low, hi)
     HOST_WIDE_INT *words;
     unsigned HOST_WIDE_INT *low;
     HOST_WIDE_INT *hi;
{
  *low = words[0] + words[1] * BASE;
  *hi = words[2] + words[3] * BASE;
}
```

제 3 절 실수

GCC 의 TREE node 에서 실수를 나타내는 node 는 REAL_CST node 로써 struct tree_real_cst 구조체에 정의되어 있으며, 결과적으로 실수값은 REAL_VALUE_TYPE real_cst 에 들어가게 된다.
우선 REAL_VALUE_TYPE 매크로에 대해서 살펴보도록 하자.

3.1 REAL_VALUE_TYPE

REAL_VALUE_TYPE 는 real.h 에 정의되어 있는 매크로로써 실제로는 realvaluetype 라는 구조체를 나타내는 형태이다. 이것은 각 machine 에 따라 실수형을 software 적으로 구현하기 위해서 사용되는 구조체로써 $prefix/gcc/real.h 파일을 살펴볼 경우 이 구조체에 대해 살펴볼 수 있다. 실수형이 가질 수 있는 크기는 각 machine 마다 크기가 달라지게 되는데, realvaluetype 의 크기에 영향을 주는 것은 MAX_LONG_DOUBLE_TYPE_SIZE 매크로라고 할 수 있겠다. i386 의 경우, 이 매크로의 값은 $prefix/gcc/config/i386/i386.h 파일에 정의되어 있으며, 값은 128 로 되어 있다. 그래서 필자의 컴퓨터에서 정의되는 하나의 실수형은 아래와 같은 구조체 모양을 가지게 된다.

```
#define REAL_IS_NOT_DOUBLE
#define REAL_ARITHMETIC
typedef struct {
  HOST_WIDE_INT r[(19 + sizeof (HOST_WIDE_INT))/(sizeof (HOST_WIDE_INT))];
} realvaluetype;
#define REAL_VALUE_TYPE realvaluetype
```

앞에서 정수형 HOST_WIDE_INT 에 대해 보았으며, 필자의 컴퓨터에서는 int 이기 때문에, 위의 내부 변수 r 은 5 의 크기를 가진다는 것을 알 수 있다.

3.2 UEMUSHORT

위에서 언급한 REAL_VALUE_TYPE 는 TREE node 에서의 모습이라고 할 수 있지만 실제 실수형 계산을 할 경우는, 이 type 을 사용하는 것이 아니라, UEMUSHORT 라는 매크로가 정의하는 type 을 사용한다. 또한 이 type 을 사용하여, 해당 크기를 나타내기 위해서 NE 라는 매크로로 정의를 하는데, 아래와 같이 선언한다고 하면 간단하겠다.

```
    UEMUSHORT e[NE];
```

UEMUSHORT 가 어떻게 선언되는지는 $prefix/gcc/real.c 파일에 자세하게 나와 있는데,
HOST_BITS_PER_* 매크로의 크기에 따라 이 매크로가 가지는 type 이 달라지게 되는 것이다. 필자의 경우, 다음과 같이 선언되어

```
#define EMUSHORT short
#define EMUSHORT_SIZE HOST_BITS_PER_SHORT
#define EMULONG_SIZE (2 * HOST_BITS_PER_SHORT)
```

결과적으로는 아래와 같이 UEMUSHORT 이 선언되게 된다.

```
#define UEMUSHORT unsigned EMUSHORT
```

또한 NE 매크로도 $prefix/gcc/real.c 파일에서 그 선언을 찾을 수 있는데, NE 의 크기를 결정하는 것은 MAX_LONG_DOUBLE_TYPE_SIZE 매크로와 INTEL_EXTENDED_IEEE_FORMAT 매크로의 설정에 따라 결정된다. 필자의 컴퓨터는 MAX_LONG_DOUBLE_TYPE_SIZE 는 128 로
INTEL_EXTENDED_IEEE_FORMAT 는 1 로 선언되어 있기 때문에 아래와 같이 NE 가 선언되게 된다.

```
# define NE 6
# define MAXDECEXP 4932
# define MINDECEXP -4956
# define GET_REAL(r,e)  memcpy ((e), (r), 2*NE)
# define PUT_REAL(e,r)                                          \
        do {                                                    \
          memcpy ((r), (e), 2*NE);                              \
          if (2*NE < sizeof (*r))                               \
            memset ((char *) (r) + 2*NE, 0, sizeof (*r) - 2*NE);        \
        } while (0)
```

위를 보면, NE 말고도 몇 개의 정의들을 언급해 놓았는데, 실제로 REAL_VALUE_TYPE 변수를 UEMUSHORT 로 get 하고 put 하는 매크로들이다. 즉, GET_REAL 매크로를 통해서 REAL_VALUE_TYPE 의 변수를 UEMUSHORT 로 넣고, PUT_REAL 매크로를 통해서 UEMUSHORT 의 값을 REAL_VALUE_TYPE 에 놓는 것이다.

real.c 파일의 내용을 보면 알겠지만, 이러한 매크로의 목적은 같을 수 있지만, machine 의 환경에 따라 다르게 정의되어 있을 수 있음을 알기 바란다.

3.3 실수 연산을 위한 함수

아래의 내용은 $prefix/gcc/real.c 파일에 선언되어 있는 함수들로써, IBM, DEC, VAX 등등의 각기 다른 floating point 연산과 관련된 것을 제외한 필자의 machine 에서 필요한 함수들에 대해서 정리한 함수 목록이다.

물론 아래의 함수들을 포함해서, 다른 $prefix/gcc/real.h 에 정의되어 있는 많은 macro 들이 있지만, 여기서는 언급하지 않을 것이며, 아래의 함수들은 그 함수 이름 자체로 호출되는 경우도 있겠지만, real.h 에 정의되어 있는 매크로의 이름으로 호출되는 경우가 많으며, 한번쯤은 꼭 real.h 파일의 내용을 읽어보기 바란다.

```
static void
endian (e, x, mode)
     const UEMUSHORT e[];
     long x[];
     enum machine_mode mode;
```

포함된 32-bit number 를 16-bit number 를 포함하는 array 에서 long 의 array output 으로 복사한다. 필요시 끝에서 swap 한다. 결과는 보통 ASM_OUTPUT_ 매크로에 의해 fprintf 로 건네진다.

```
void
earith (value, icode, r1, r2)
     REAL_VALUE_TYPE *value;
     int icode;
     REAL_VALUE_TYPE *r1;
     REAL_VALUE_TYPE *r2;
```

이것은 REAL_ARITHMETIC macro 의 기능이다.

```
REAL_VALUE_TYPE
etrunci (x)
     REAL_VALUE_TYPE x;
```

REAL_VALUE_TYPE 를 signed HOST_WIDE_INT 가 되도록 0 쪽으로 자른다.
REAL_VALUE_RNDZINT (x) (etrunci (x)) 로 수행한다.

```
REAL_VALUE_TYPE
etruncui (x)
     REAL_VALUE_TYPE x;
```

REAL_VALUE_TYPE 를 unsigned HOST_WIDE_INT 가 되도록 0 쪽으로 자른다.
REAL_VALUE_UNSIGNED_RNDZINT (x) (etruncui (x)) 로 수행한다.

```
REAL_VALUE_TYPE
ereal_atof (s, t)
     const char *s;
     enum machine_mode t;
```

이것은 REAL_VALUE_ATOF 함수입니다. 10 진수 혹은 16 진수 문자열을 binary 로 변환하며 machine_mode 인자에 의해 지시된 곳까지 반올림입니다.
그런후 반올림된 값을 REAL_VALUE_TYPE 로 진행시킵니다.

```
REAL_VALUE_TYPE
ereal_negate (x)
     REAL_VALUE_TYPE x;
```

REAL_NEGATE 의 확장.

```
HOST_WIDE_INT
efixi (x)
     REAL_VALUE_TYPE x;
```

실수가 HOST_WIDE_INT 가 되도록 0 쪽으로 round 한다. REAL_VALUE_FIX (x) 로 수행한다.

```
unsigned HOST_WIDE_INT
efixui (x)
     REAL_VALUE_TYPE x;
```

실수가 unsigned HOST_WIDE_INT 가 되도록 0 쪽으로 round 한다.
REAL_VALUE_UNSIGNED_FIX (x) 로 수행되는데, 음수는 0 이 반환된다.

```
void
ereal_from_int (d, i, j, mode)
     REAL_VALUE_TYPE *d;
     HOST_WIDE_INT i, j;
     enum machine_mode mode;
```

REAL_VALUE_FROM_INT macro.

```
void
ereal_from_uint (d, i, j, mode)
     REAL_VALUE_TYPE *d;
     unsigned HOST_WIDE_INT i, j;
     enum machine_mode mode;
```

REAL_VALUE_FROM_UNSIGNED_INT macro.

```
void
ereal_to_int (low, high, rr)
     HOST_WIDE_INT *low, *high;
     REAL_VALUE_TYPE rr;
```

REAL_VALUE_TO_INT macro.

```
REAL_VALUE_TYPE
ereal_ldexp (x, n)
     REAL_VALUE_TYPE x;
     int n;
```

REAL_VALUE_LDEXP macro.

```
int
target_isinf (x)
     REAL_VALUE_TYPE x ATTRIBUTE_UNUSED;
```

REAL_VALUE_TYPE 내 infinity 를 검사한다.

```
int
target_isnan (x)
     REAL_VALUE_TYPE x ATTRIBUTE_UNUSED;
```

REAL_VALUE_TYPE item 이 NaN 인지 아닌지를 검사한다.

```
int
target_negative (x)
     REAL_VALUE_TYPE x;
```

Negative REAL_VALUE_TYPE number 를 검사한다. 이것은 단수히 sign bit 만 검사해서 -0 은 negative 로 계산한다.

```
REAL_VALUE_TYPE
real_value_truncate (mode, arg)
     enum machine_mode mode;
     REAL_VALUE_TYPE arg;
```

REAL_VALUE_TRUNCATE 의 확장. 결과는 floating point 로써, nearest 혹은 even 으로 round 된다.

```
int
exact_real_inverse (mode, r)
     enum machine_mode mode;
     REAL_VALUE_TYPE *r;
```

R 을 machine mode MODE 내 그것의 정확한 multiplicative inverse 로 변경을 시도한다. 만약 성공시, 0 이 아닌 함수값을 반환한다.

```
void
debug_real (r)
     REAL_VALUE_TYPE r;
```

사람이 읽을 수 있는 format 으로 R 의 값을 stderr 로 debugging–print 하는데 사용한다.

```
void
etartdouble (r, l)
     REAL_VALUE_TYPE r;
     long l[];
```

R 을 128-bit long double precision value 로 변환한다. Output array L 은 메모리에서 보기에 순서대로 네 32-bit 조각의 결과를 포함한다.

```
void
etarldouble (r, l)
     REAL_VALUE_TYPE r;
     long l[];
```

R 을 double extended precision value 으로 변환한다. Output array L 은 메모리에서 보기에 순서대로 세 32-bit 조각의 결과를 포함한다.

```
void
etardouble (r, l)
     REAL_VALUE_TYPE r;
     long l[];
```

R 을 double precision value 로 변환한다. Output array L 은 메모리에서 보기에 순서대로 두 32-bit 조각의 결과를 포함한다.

```
long
etarsingle (r)
     REAL_VALUE_TYPE r;
```

R 을 'long' 의 least-significant bit 에 저장되어 있는 single precision float value 로 변환한다.

```
void
ereal_to_decimal (x, s)
     REAL_VALUE_TYPE x;
     char *s;
```

X 를 assembly language file 로의 output 을 위해 decimal ASCII string S 로 변환한다. infinity 혹은 NaN 을 판독할 표현 방법이 존재하지 않기 때문에 이 값들은 tm.h 매크로에서 특별한 취급이 요구될 수 있음을 참고하라.

```
int
ereal_cmp (x, y)
     REAL_VALUE_TYPE x, y;
```

X 와 Y 를 비교한다. 만약 $X > Y$ 이면 1 을, $X == Y$ 이면 0 을, $X < Y$ 이면 -1 을, NaN 일 경우 -2 를 반환한다.

```
int
ereal_isneg (x)
     REAL_VALUE_TYPE x;
```

만약 X 의 sign bit 가 설정되어 있을 경우 1 을, 그렇지 않을 경우 0 을 반환한다.

```
static void
eclear (x)
     UEMUSHORT *x;
```

전체 e-type number X 를 깨끗히 한다.

```
static void
emov (a, b)
     const UEMUSHORT *a;
     UEMUSHORT *b;
```

E-type number 를 A 에서 B 로 옮긴다.

```
static void
eneg (x)
     UEMUSHORT x[];
```

e-type number X 를 negate 한다.

```
static int
eisneg (x)
     const UEMUSHORT x[];
```

E-type number X 의 sign bit 가 0 이 아닐 경우, 1 을 반환하고 그렇지 않다면 0 을 반환한다.

```
static int
eisinf (x)
     const UEMUSHORT x[];
```

E-type number X 가 infinity 라면 1 을 반환하고, 아닐 경우 0 을 반환한다.

```
static int
eisnan (x)
     const UEMUSHORT x[] ATTRIBUTE_UNUSED;
```

E-type number 가 number 인지를 검사한다. Bit pattern 은 우리가 정의한 것이기 때문에, 어떻게 탐지할지는 우리가 잘 알고 있다.

```
static void
einfin (x)
     UEMUSHORT *x;
```

E-type number X 를 infinity pattern (IEEE) 혹은 largest possible number (non-IEEE) 로 채운다.

```
static void
enan (x, sign)
     UEMUSHORT *x;
     int sign;
```

e-type NaN 를 output 한다. 이것은 extended real 를 위해 Intel 의 quiet NaN pattern 을 생성한다. Exponent 는 7fff 이고 leading mantissa word 는 c000 이다.

```
static void
emovi (a, b)
     const UEMUSHORT *a;
     UEMUSHORT *b;
```

e-type number A 를 exploded e-type B 로 변환해서 옮긴다.

```
static void
emovo (a, b)
     const UEMUSHORT *a;
     UEMUSHORT *b;
```

exploded e-type number A 를 e type B 로 변환해서 옮긴다.

```
static void
ecleaz (xi)
     UEMUSHORT *xi;
```

exploded e-type number XI 를 깨끗히 한다.

```
static void
ecleazs (xi)
     UEMUSHORT *xi;
```

exploded e-type XI 를 깨끗히 하지만 sign 은 만지지 않는다.

```
static void
emovz (a, b)
     const UEMUSHORT *a;
     UEMUSHORT *b;
```

exploded e-type number 를 A 에서 B 로 옮긴다.

```
static void
einan (x)
     UEMUSHORT x[];
```

exploded e-type NaN 을 생성한다. 이것을 위한 explicit pattern 은 maximum exponent 와 top two significant bits set 이다.

```
static int
eiisnan (x)
     const UEMUSHORT x[];
```

exploded e-type X 가 NaN 이면 0 이 아닌 값을 반환한다.

```
static int
eiisneg (x)
     const UEMUSHORT x[];
```

만약 exploded e-type X 의 sign 이 0 이 아닐 경우 0 이 아닌 값을 반환한다.

```
static int
eiisinf (x)
     const UEMUSHORT x[];
```

만약 exploded e-type X 가 infinite 라면 0 이 아닌 값을 반환한다.

```
static int
ecmpm (a, b)
     const UEMUSHORT *a, *b;
```

internal exploded e-type format 의 number 의 significands 을 비교한다. Guard word 들은 comparison 에 포함된다.

만약 a > b 라면 +1 을, a == b 이면 0 을, a < b 이면 −1 을 반환한다.

```
static void
eshdn1 (x)
     UEMUSHORT *x;
```

exploded e-type X 의 significand 를 밑으로 1 만큼 shift 한다.

```
static void
eshup1 (x)
     UEMUSHORT *x;
```

exploded e-type X 의 significand 를 위로 1 만큼 shift 한다.

```
static void
eshdn8 (x)
     UEMUSHORT *x;
```

Exploded e-type X 의 significand 를 밑으로 8 만큼 shift 한다.

```
static void
eshup8 (x)
     UEMUSHORT *x;
```

Exploded e-type X 의 significand 를 위로 8 만큼 shift 한다.

```
static void
eshup6 (x)
     UEMUSHORT *x;
```

Exploded e-type X 의 significand 를 위로 16 만큼 shift 한다.

```
static void
eshdn6 (x)
     UEMUSHORT *x;
```

Exploded e-type X 의 significand 를 밑으로 16 만큼 shift 한다.

```
static void
eaddm (x, y)
     const UEMUSHORT *x;
     UEMUSHORT *y;
```

exploded e-type X 와 Y 의 significand 들을 더한다. X + Y 는 Y 에 들어간다.

```
static void
esubm (x, y)
     const UEMUSHORT *x;
     UEMUSHORT *y;
```

exploded e-type X 와 Y 의 significand 들을 뺀다. Y - X 는 Y 에 들어간다.

```
static void
m16m (a, b, c)
     unsigned int a;
     const UEMUSHORT b[];
     UEMUSHORT c[];
```

e-type number B 의 significand 를 16-bit quantity A 로 곱한다. e-type result 를 C 에 실어 반환한다.

```
static int
edivm (den, num)
     const UEMUSHORT den[];
     UEMUSHORT num[];
```

exploded e-types NUM / DEN 의 significand 들을 나눈다. numerator NUM 도 denominator DEN 도 0 이 아닌 그것의 high guard word 를 가지도록 허락되지 않는다.

```
static int
emulm (a, b)
     const UEMUSHORT a[];
     UEMUSHORT b[];
```

exploded e-type A 와 B 의 significand 들을 곱한다. B 에 결과가 들어간다.

```
static void
emdnorm (s, lost, subflg, exp, rcntrl)
     UEMUSHORT s[];
     int lost;
     int subflg;
     EMULONG exp;
     int rcntrl;
```

Normalize 와 round off.

Round 되어질 internal format number 는 S 이다. Input LOST 는 만약 값이 정확할 경우 0 이다. 이것은 so-called sticky bit 이다.

Input SUBFLG 는 숫자가 subtraction operation 와 연관있는지를 가르킨다. 그러한 경우 만약 LOST 가 0 이 아닐 경우, 숫자는 지정된 것 보다 약간 작다.

Input EXP 는 biased exponent 인데, 아마 음수일 것이다. S 의 exponent field 는 무지되지만 normalization 와 rounding 에 의해 맞춰진 EXP 에 의해 대체되어진다.

Input RCNTRL 는 rounding control 이다. 만약 0 이 아닐 경우, 반환된 값은 RNDPRC bit 들로 round 되어질 것이다.

```
static void
esub (a, b, c)
     const UEMUSHORT *a, *b;
     UEMUSHORT *c;
```

빼기. C= B - A, 모두 e type number 들이다.

```
static void
eadd (a, b, c)
     const UEMUSHORT *a, *b;
     UEMUSHORT *c;
```

더하기. C = A + B, 모두 e type 이다.

```
static void
eadd1 (a, b, c)
     const UEMUSHORT *a, *b;
     UEMUSHORT *c;
```

덧셈과 뺄셈의 공통적인 산술 연산.

```
static void
ediv (a, b, c)
     const UEMUSHORT *a, *b;
     UEMUSHORT *c;
```

나누기. C = B/A, 모두 e type 이다.

```
static void
emul (a, b, c)
     const UEMUSHORT *a, *b;
     UEMUSHORT *c;
```

e-types A 와 B 를 곱한다. e-type product C 를 반환한다.

```
static void
e53toe (pe, y)
     const UEMUSHORT *pe;
     UEMUSHORT *y;
```

Double precision PE 를 e-type Y 로 변환합니다.

```
static void
e64toe (pe, y)
     const UEMUSHORT *pe;
     UEMUSHORT *y;
```

double extended precision float PE 를 e type Y 로 변환한다.

```
static void
e24toe (pe, y)
     const UEMUSHORT *pe;
     UEMUSHORT *y;
```

single precision float PE 를 e type Y 로 변환한다.

```
static void
etoe64 (x, e)
     const UEMUSHORT *x;
     UEMUSHORT *e;
```

e-type X 를 IEEE double extended format E 로 변환한다.

```
static void
toe64 (a, b)
     UEMUSHORT *a, *b;
```

이미 64-bit precision 으로 round 되어진 exploded e-type X 를 IEEE double extended format Y 로 변환한다.

```
static void
etoe53 (x, e)
     const UEMUSHORT *x;
     UEMUSHORT *e;
```

e-type X 를 IEEE double E 로 변환한다.

```
static void
toe53 (x, y)
     UEMUSHORT *x, *y;
```

이미 53-bit precision 으로 round 되어진 e-type X 를 IEEE double Y 로 변환한다.

```
static void
etoe24 (x, e)
     const UEMUSHORT *x;
     UEMUSHORT *e;
```

e-type X 를 IEEE float E 로 변환한다. DEC float 는 IEEE float 와 같다.

```
static void
toe24 (x, y)
     UEMUSHORT *x, *y;
```

이미 float precision 으로 round 되어진 exploded e-type X, 를 IEEE float Y 로 변환한다.

```
static int
ecmp (a, b)
     const UEMUSHORT *a, *b;
```

두 e type 숫자를 비교한다. 만약 a > b 이면 +1, a == b 이면 0, a < b 이면 −1, a 혹은 b 가 NaN 이면 -2 를 반환한다.

```
static void
ltoe (lp, y)
     const HOST_WIDE_INT *lp;
     UEMUSHORT *y;
```

HOST_WIDE_INT LP 를 e type Y 로 변환한다.

```
static void
ultoe (lp, y)
     const unsigned HOST_WIDE_INT *lp;
     UEMUSHORT *y;
```

unsigned HOST_WIDE_INT LP 를 e type Y 로 변환한다.

```
static void
eifrac (x, i, frac)
     const UEMUSHORT *x;
     HOST_WIDE_INT *i;
     UEMUSHORT *frac;
```

Signed HOST_WIDE_INT integer I 와 e-type (packed internal format) floating point input X 의 floating point fractional part FRAC 를 찾는다.
integer output I 는 만약 FIXUNS_TRUNC_LIKE_FIX_TRUNC 가 설정 되어 있을 경우 positive overflow 가 허락된다는 것을 제외하곤, input 의 sign 을 가진다. output e-type fraction FRAC 는 abs (X) 의 positive fractional part 이다.

```
static void
euifrac (x, i, frac)
     const UEMUSHORT *x;
     unsigned HOST_WIDE_INT *i;
     UEMUSHORT *frac;
```

unsigned HOST_WIDE_INT integer I 와 e-type X 의 floating point fractional part FRAC 를 찾는다. 음수 input 는 integer output = 0 이지만 올바른 fraction 이라는 것을 알린다.

```
static int
eshift (x, sc)
     UEMUSHORT *x;
     int sc;
```

exploded e-type X 의 significand 를 SC bit 만큼 위 혹은 아래로 shift 한다.

```
static int
enormlz (x)
     UEMUSHORT x[];
```

exploded e-type X 의 significand area 인 normalize 를 shift 한다. shift count (up = positive) 를 반환한다.

```
static void
etoasc (x, string, ndigs)
     const UEMUSHORT x[];
     char *string;
     int ndigs;
```

e-type X 를 decimal point 뒤 NDIGS digit 들로 구성된 ASCII string STRING 으로 변환한다.

```
static void
asctoe24 (s, y)
     const char *s;
     UEMUSHORT *y;
```

ASCII string S 를 single precision float value Y 로 변환한다.

```
static void
asctoe53 (s, y)
     const char *s;
     UEMUSHORT *y;
```

ASCII 문자열 S 를 double precision 값 Y 로 변환합니다.

```
static void
asctoe64 (s, y)
     const char *s;
     UEMUSHORT *y;
```

ASCII string S 를 double extended value Y 로 변환한다.

```
static void
asctoe (s, y)
     const char *s;
     UEMUSHORT *y;
```

ASCII string S 를 e type Y 로 변환한다.

```
static void
asctoeg (ss, y, oprec)
     const char *ss;
     UEMUSHORT *y;
     int oprec;
```

ASCII 문자열 SS 를 e type Y 로 변환합니다. 단 지정된 OPREC 비트 만큼의 rounding precision 을 따지게 됩니다. BASE 는 C99 hexadecimal floating constant 를 위해서는 16 입니다.

```
static void
efloor (x, y)
     const UEMUSHORT x[];
     UEMUSHORT y[];
```

(Minus infinity 쪽으로 truncate 되어진) X 보다 크지 않는 범위에서 가장 큰 정수 = Y 를 반환한다.

```
static void
eldexp (x, pwr2, y)
     const UEMUSHORT x[];
     int pwr2;
     UEMUSHORT y[];
```

e type Y = X * 2^PWR2 를 반환한다.

```
static void
eiremain (den, num)
     UEMUSHORT den[], num[];
```

EQUOT 내 exploded e-types NUM / DEN 의 quotient, NUM 내 remainder 를 반환한다.

```
static void
mtherr (name, code)
     const char *name;
     int code;
```

function NAME 가 만난 error condition CODE 를 보고한다.

Mnemonic	Value	Significance
DOMAIN	1	argument domain error
SING	2	function singularity
OVERFLOW	3	overflow range error
UNDERFLOW	4	underflow range error
TLOSS	5	total loss of precision
PLOSS	6	partial loss of precision
INVALID	7	NaN - producing operation
EDOM	33	Unix domain error code
ERANGE	34	Unix range error code

다음 메세지들의 나타나는 순서는 위에서 정의된 error code 들과 연관성을 가진다.

```
static void
make_nan (nan, sign, mode)
     UEMUSHORT *nan;
     int sign;
     enum machine_mode mode;
```

Taget machine 의 format 으로 binary NaN bit pattern 을 output 한다.

```
REAL_VALUE_TYPE
ereal_unto_float (f)
     long f;
```

이것은 REAL_VALUE_TO_TARGET_SINGLE 과 연관된 함수 'etarsingle' 의 반대역이다.

```
REAL_VALUE_TYPE
ereal_unto_double (d)
     long d[];
```

이것은 REAL_VALUE_TO_TARGET_DOUBLE 와 연관된 함수 'etardouble' 와 반대역이다.

```
REAL_VALUE_TYPE
ereal_from_float (f)
     HOST_WIDE_INT f;
```

SFmode target 'float' value 를 REAL_VALUE_TYPE 로 변환한다. 이것은 다소 ereal_unto_float 와 비슷하지만 이것을 위한 input type 들이 다르다.

```
REAL_VALUE_TYPE
ereal_from_double (d)
     HOST_WIDE_INT d[];
```

DFmode target 'double' value 를 REAL_VALUE_TYPE 로 변환한다. 이것은 다소 ereal_unto_double 와 비슷하지만 이것을 위한 input type 들이 다르다.

DFmode 는 target 의 data format 에서의 HOST_WIDE_INT 의 배열로 저장 되며, bit packing 에서 hole 을 가지고 있지 않다. Input array 의 첫번째 element 는 target computer 의 메모리에 첫번째로 오는 bit 들을 가지고 있다.

```
unsigned int
significand_size (mode)
     enum machine_mode mode;
```

주어진 floating point mode 를 위한 significand 의 binary precision 를 반환한다. Mode 는 어떠한 bit 들을 잃어버리는 일없이 많은 bit 들을 담을 수 있는 좀 더 넓은 integer value 을 가질 수 있다.