1. Introduction. The subroutines below are used to simulate 64-bit `MMIX` arithmetic on an old-fashioned 32-bit computer—like the one the author had when he wrote `MMIXAL` and the first `MMIX` simulators in 1998 and 1999. All operations are fabricated from 32-bit arithmetic, including a full implementation of the IEEE floating point standard, assuming only that the C compiler has a 32-bit unsigned integer type.

Some day 64-bit machines will be commonplace and the awkward manipulations of the present program will look quite archaic. Interested readers who have such computers will be able to convert the code to a pure 64-bit form without difficulty, thereby obtaining much faster and simpler routines. Meanwhile, however, we can simulate the future and hope for continued progress.

This program module has a simple structure, intended to make it suitable for loading with `MMIX` simulators and assemblers.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
  ⟨ Stuff for C preprocessor 2 ⟩
  typedef enum { false, true } bool;
  ⟨ Tetrabyte and octabyte type definitions 3 ⟩
  ⟨ Other type definitions 36 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 5 ⟩
```

2. Subroutines of this program are declared first with a prototype, as in ANSI C, then with an old-style C function definition. Here are some preprocessor commands that make this work correctly with both new-style and old-style compilers.

```
⟨ Stuff for C preprocessor 2 ⟩ ≡
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

This code is used in section 1.

3. The definition of type **tetra** should be changed, if necessary, so that it represents an unsigned 32-bit integer.

```
⟨ Tetrabyte and octabyte type definitions 3 ⟩ ≡
  typedef unsigned int tetra;    /* for systems conforming to the LP-64 data model */
  typedef struct {
    tetra h, l;
  } octa;    /* two tetrabytes make one octabyte */
```

This code is used in section 1.

4. `#define sign_bit ((unsigned) #80000000)`

```
⟨ Global variables 4 ⟩ ≡
  octa zero_octa;    /* zero_octa.h = zero_octa.l = 0 */
  octa neg_one = {-1, -1};    /* neg_one.h = neg_one.l = -1 */
  octa inf_octa = {#7ff00000, 0};    /* floating point +∞ */
  octa standard_NaN = {#7ff80000, 0};    /* floating point NaN(.5) */
```

See also sections 9, 30, 32, 69, and 75.

This code is used in section 1.

5. It's easy to add and subtract octabytes, if we aren't terribly worried about speed.

```
⟨ Subroutines 5 ⟩ ≡
  octa oplus ARGS((octa, octa));
  octa oplus(y, z)      /* compute y + z */
      octa y, z;
  { octa x;
    x.h = y.h + z.h; x.l = y.l + z.l;
    if (x.l < y.l) x.h++;
    return x;
  }
  octa ominus ARGS((octa, octa));
  octa ominus(y, z)      /* compute y − z */
      octa y, z;
  { octa x;
    x.h = y.h − z.h; x.l = y.l − z.l;
    if (x.l > y.l) x.h−−;
    return x;
  }
```

See also sections 6, 7, 8, 12, 13, 24, 25, 26, 27, 28, 29, 31, 34, 37, 38, 39, 40, 41, 44, 46, 50, 54, 60, 61, 62, 68, 82, 85, 86, 88, 89, 91, and 93.

This code is used in section 1.

6. In the following subroutine, *delta* is a signed quantity that is assumed to fit in a signed tetrabyte.

```
⟨ Subroutines 5 ⟩ +≡
  octa incr ARGS((octa, int));
  octa incr(y, delta)      /* compute y + δ */
      octa y;
      int delta;
  { octa x;
    x.h = y.h; x.l = y.l + delta;
    if (delta ≥ 0) {
      if (x.l < y.l) x.h++;
    } else if (x.l > y.l) x.h−−;
    return x;
  }
```

7. Left and right shifts are only a bit more difficult.

```
⟨ Subroutines 5 ⟩ +≡
  octa shift_left ARGS((octa, int));
  octa shift_left(y, s)      /* shift left by s bits, where 0 ≤ s ≤ 64 */
      octa y;
      int s;
  {
    while (s ≥ 32) y.h = y.l, y.l = 0, s −= 32;
    if (s) { register tetra yhl = y.h ≪ s, ylh = y.l ≫ (32 − s);
      y.h = yhl + ylh; y.l ≪= s;
    }
    return y;
  }
  octa shift_right ARGS((octa, int, int));
  octa shift_right(y, s, u)      /* shift right, arithmetically if u = 0 */
      octa y;
      int s, u;
  {
    while (s ≥ 32) y.l = y.h, y.h = (u ? 0 : −(y.h ≫ 31)), s −= 32;
    if (s) { register tetra yhl = y.h ≪ (32 − s), ylh = y.l ≫ s;
      y.h = (u ? 0 : (−(y.h ≫ 31)) ≪ (32 − s)) + (y.h ≫ s); y.l = yhl + ylh;
    }
    return y;
  }
```

8. Multiplication. We need to multiply two unsigned 64-bit integers, obtaining an unsigned 128-bit product. It is easy to do this on a 32-bit machine by using Algorithm 4.3.1M of *Seminumerical Algorithms*, with $b = 2^{16}$.

The following subroutine returns the lower half of the product, and puts the upper half into a global octabyte called *aux*.

```
⟨ Subroutines 5 ⟩ +≡
  octa omult ARGS((octa, octa));
  octa omult(y, z)
      octa y, z;
  {
    register int i, j, k;
    tetra u[4], v[4], w[8];
    register tetra t;
    octa acc;
    ⟨ Unpack the multiplier and multiplicand to u and v 10 ⟩;
    for (j = 0; j < 4; j++) w[j] = 0;
    for (j = 0; j < 4; j++)
      if (¬v[j]) w[j + 4] = 0;
      else {
        for (i = k = 0; i < 4; i++) {
          t = u[i] * v[j] + w[i + j] + k;
          w[i + j] = t & #ffff, k = t ≫ 16;
        }
        w[j + 4] = k;
      }
    ⟨ Pack w into the outputs aux and acc 11 ⟩;
    return acc;
  }
```

9. ⟨ Global variables 4 ⟩ +≡

```
  octa aux;      /* secondary output of subroutines with multiple outputs */
  bool overflow;     /* set by certain subroutines for signed arithmetic */
```

10. ⟨ Unpack the multiplier and multiplicand to u and v 10 ⟩ ≡

```
u[3] = y.h ≫ 16, u[2] = y.h & #ffff, u[1] = y.l ≫ 16, u[0] = y.l & #ffff;
v[3] = z.h ≫ 16, v[2] = z.h & #ffff, v[1] = z.l ≫ 16, v[0] = z.l & #ffff;
```

This code is used in section 8.

11. ⟨ Pack w into the outputs *aux* and *acc* 11 ⟩ ≡

```
aux.h = (w[7] ≪ 16) + w[6], aux.l = (w[5] ≪ 16) + w[4];
acc.h = (w[3] ≪ 16) + w[2], acc.l = (w[1] ≪ 16) + w[0];
```

This code is used in section 8.

12. Signed multiplication has the same lower half product as unsigned multiplication. The signed upper half product is obtained with at most two further subtractions, after which the result has overflowed if and only if the upper half is unequal to 64 copies of the sign bit in the lower half.

```
⟨ Subroutines 5 ⟩ +≡
  octa signed_omult ARGS((octa, octa));
  octa signed_omult(y, z)
      octa y, z;
  {
    octa acc;
    acc = omult(y, z);
    if (y.h & sign_bit) aux = ominus(aux, z);
    if (z.h & sign_bit) aux = ominus(aux, y);
    overflow = (aux.h ≠ aux.l ∨ (aux.h ⊕ (aux.h ≫ 1) ⊕ (acc.h & sign_bit)));
    return acc;
  }
```

13. Division. Long division of an unsigned 128-bit integer by an unsigned 64-bit integer is, of course, one of the most challenging routines needed for `MMIX` arithmetic. The following program, based on Algorithm 4.3.1D of *Seminumerical Algorithms*, computes octabytes q and r such that $(2^{64}x+y) = qz+r$ and $0 \le r < z$, given octabytes x, y, and z, assuming that $x < z$. (If $x \ge z$, it simply sets $q = x$ and $r = y$.) The quotient q is returned by the subroutine; the remainder r is stored in aux.

```
⟨ Subroutines 5 ⟩ +≡
  octa odiv ARGS((octa,octa,octa));
  octa odiv(x,y,z)
      octa x, y, z;
  {
    register int i, j, k, n, d;
    tetra u[8], v[4], q[4], mask, qhat, rhat, vh, vmh;
    register tetra t;
    octa acc;
    ⟨ Check that x < z; otherwise give trivial answer 14 ⟩;
    ⟨ Unpack the dividend and divisor to u and v 15 ⟩;
    ⟨ Determine the number of significant places n in the divisor v 16 ⟩;
    ⟨ Normalize the divisor 17 ⟩;
    for (j = 3; j ≥ 0; j--) ⟨ Determine the quotient digit q[j] 20 ⟩;
    ⟨ Unnormalize the remainder 18 ⟩;
    ⟨ Pack q and u to acc and aux 19 ⟩;
    return acc;
  }
```

14. ⟨ Check that $x < z$; otherwise give trivial answer 14 ⟩ ≡

```
  if (x.h > z.h ∨ (x.h ≡ z.h ∧ x.l ≥ z.l)) {
    aux = y; return x;
  }
```

This code is used in section 13.

15. ⟨ Unpack the dividend and divisor to u and v 15 ⟩ ≡

```
  u[7] = x.h ≫ 16, u[6] = x.h & #ffff, u[5] = x.l ≫ 16, u[4] = x.l & #ffff;
  u[3] = y.h ≫ 16, u[2] = y.h & #ffff, u[1] = y.l ≫ 16, u[0] = y.l & #ffff;
  v[3] = z.h ≫ 16, v[2] = z.h & #ffff, v[1] = z.l ≫ 16, v[0] = z.l & #ffff;
```

This code is used in section 13.

16. ⟨ Determine the number of significant places n in the divisor v 16 ⟩ ≡

```
  for (n = 4; v[n - 1] ≡ 0; n--) ;
```

This code is used in section 13.

17. We shift u and v left by d places, where d is chosen to make $2^{15} \le v_{n-1} < 2^{16}$.

```
⟨ Normalize the divisor 17 ⟩ ≡
  vh = v[n − 1];
  for (d = 0; vh < #8000; d++, vh ≪= 1) ;
  for (j = k = 0; j < n + 4; j++) {
    t = (u[j] ≪ d) + k;
    u[j] = t & #ffff, k = t ≫ 16;
  }
  for (j = k = 0; j < n; j++) {
    t = (v[j] ≪ d) + k;
    v[j] = t & #ffff, k = t ≫ 16;
  }
  vh = v[n − 1];
  vmh = (n > 1 ? v[n − 2] : 0);
```

This code is used in section 13.

18.

```
⟨ Unnormalize the remainder 18 ⟩ ≡
  mask = (1 ≪ d) − 1;
  for (j = 3; j ≥ n; j−−) u[j] = 0;
  for (k = 0; j ≥ 0; j−−) {
    t = (k ≪ 16) + u[j];
    u[j] = t ≫ d, k = t & mask;
  }
```

This code is used in section 13.

19.

```
⟨ Pack q and u to acc and aux 19 ⟩ ≡
  acc.h = (q[3] ≪ 16) + q[2], acc.l = (q[1] ≪ 16) + q[0];
  aux.h = (u[3] ≪ 16) + u[2], aux.l = (u[1] ≪ 16) + u[0];
```

This code is used in section 13.

20.

```
⟨ Determine the quotient digit q[j] 20 ⟩ ≡
  {
    ⟨ Find the trial quotient, q̂ 21 ⟩;
    ⟨ Subtract b^j q̂v from u 22 ⟩;
    ⟨ If the result was negative, decrease q̂ by 1 23 ⟩;
    q[j] = qhat;
  }
```

This code is used in section 13.

21.

```
⟨ Find the trial quotient, q̂ 21 ⟩ ≡
  t = (u[j + n] ≪ 16) + u[j + n − 1];
  qhat = t/vh, rhat = t − vh ∗ qhat;
  if (n > 1)
    while (qhat ≡ #10000 ∨ qhat ∗ vmh > (rhat ≪ 16) + u[j + n − 2]) {
      qhat−−, rhat += vh;
      if (rhat ≥ #10000) break;
    }
```

This code is used in section 20.

22. After this step, $u[j+n]$ will either equal k or $k-1$. The true value of u would be obtained by subtracting k from $u[j+n]$; but we don't have to fuss over $u[j+n]$, because it won't be examined later.

```
⟨ Subtract b^j q̂v from u 22 ⟩ ≡
  for (i = k = 0; i < n; i++) {
    t = u[i+j] + #ffff0000 - k - qhat * v[i];
    u[i+j] = t & #ffff, k = #ffff - (t ≫ 16);
  }
```

This code is used in section 20.

23. The correction here occurs only rarely, but it can be necessary—for example, when dividing the number #7fff800100000000 by #800080020005.

```
⟨ If the result was negative, decrease q̂ by 1 23 ⟩ ≡
  if (u[j+n] ≠ (tetra) k) {
    qhat--;
    for (i = k = 0; i < n; i++) {
      t = u[i+j] + v[i] + k;
      u[i+j] = t & #ffff, k = t ≫ 16;
    }
  }
```

This code is used in section 20.

24. Signed division can be reduced to unsigned division in a tedious but straightforward manner. We assume that the divisor isn't zero.

```
⟨ Subroutines 5 ⟩ +≡
  octa signed_odiv ARGS((octa, octa));
  octa signed_odiv(y, z)
      octa y, z;
  {
    octa yy, zz, q;
    register int sy, sz;
    if (y.h & sign_bit) sy = 2, yy = ominus(zero_octa, y);
    else sy = 0, yy = y;
    if (z.h & sign_bit) sz = 1, zz = ominus(zero_octa, z);
    else sz = 0, zz = z;
    q = odiv(zero_octa, yy, zz);
    overflow = false;
    switch (sy + sz) {
    case 2+1: aux = ominus(zero_octa, aux);
      if (q.h ≡ sign_bit) overflow = true;
    case 0+0: default: return q;
    case 2+0: if (aux.h ∨ aux.l) aux = ominus(zz, aux);
      goto negate_q;
    case 0+1: if (aux.h ∨ aux.l) aux = ominus(aux, zz);
    negate_q: if (aux.h ∨ aux.l) return ominus(neg_one, q);
      else return ominus(zero_octa, q);
    }
  }
```

25. Bit fiddling. The bitwise operators of `MMIX` are fairly easy to implement directly, but three of them occur often enough to deserve packaging as subroutines.

```
⟨ Subroutines 5 ⟩ +≡
  octa oand ARGS((octa, octa));
  octa oand(y, z)      /* compute y ∧ z */
      octa y, z;
  { octa x;
    x.h = y.h & z.h; x.l = y.l & z.l;
    return x;
  }
  octa oandn ARGS((octa, octa));
  octa oandn(y, z)      /* compute y ∧ z̄ */
      octa y, z;
  { octa x;
    x.h = y.h & ~z.h; x.l = y.l & ~z.l;
    return x;
  }
  octa oxor ARGS((octa, octa));
  octa oxor(y, z)      /* compute y ⊕ z */
      octa y, z;
  { octa x;
    x.h = y.h ⊕ z.h; x.l = y.l ⊕ z.l;
    return x;
  }
```

26. Here's a fun way to count the number of bits in a tetrabyte. [This classical trick is called the "Gillies–Miller method for sideways addition" in *The Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Gill, second edition (Reading, Mass.: Addison–Wesley, 1957), 191–193. Some of the tricks used here were suggested by Balbir Singh, Peter Rossmanith, and Stefan Schwoon.]

```
⟨ Subroutines 5 ⟩ +≡
  int count_bits ARGS((tetra));
  int count_bits(x)
      tetra x;
  {
    register int xx = x;
    xx = xx − ((xx ≫ 1) & #55555555);
    xx = (xx & #33333333) + ((xx ≫ 2) & #33333333);
    xx = (xx + (xx ≫ 4)) & #0f0f0f0f;
    xx = xx + (xx ≫ 8);
    return (xx + (xx ≫ 16)) & #ff;
  }
```

27. To compute the nonnegative byte differences of two given tetrabytes, we can carry out the following 20-step branchless computation:

```
⟨ Subroutines 5 ⟩ +≡
  tetra byte_diff ARGS((tetra, tetra));
  tetra byte_diff (y, z)
      tetra y, z;
  {
    register tetra d = (y & #00ff00ff) + #01000100 − (z & #00ff00ff);
    register tetra m = d & #01000100;
    register tetra x = d & (m − (m ≫ 8));
    d = ((y ≫ 8) & #00ff00ff) + #01000100 − ((z ≫ 8) & #00ff00ff);
    m = d & #01000100;
    return x + ((d & (m − (m ≫ 8))) ≪ 8);
  }
```

28. To compute the nonnegative wyde differences of two tetrabytes, another trick leads to a 15-step branchless computation. (Research problem: Can *count_bits*, *byte_diff*, or *wyde_diff* be done with fewer operations?)

```
⟨ Subroutines 5 ⟩ +≡
  tetra wyde_diff ARGS((tetra, tetra));
  tetra wyde_diff (y, z)
      tetra y, z;
  {
    register tetra a = ((y ≫ 16) − (z ≫ 16)) & #10000;
    register tetra b = ((y & #ffff) − (z & #ffff)) & #10000;
    return y − (z ⊕ ((y ⊕ z) & (b − a − (b ≫ 16))));
  }
```

29. The last bitwise subroutine we need is the most interesting: It implements MMIX's MOR and MXOR operations.

```
⟨ Subroutines 5 ⟩ +≡
  octa bool_mult ARGS((octa, octa, bool));
  octa bool_mult(y, z, xor)
      octa y, z;      /* the operands */
      bool xor;       /* do we do xor instead of or? */
  {
    octa o, x;
    register tetra a, b, c;
    register int k;
    for (k = 0, o = y, x = zero_octa; o.h ∨ o.l; k++, o = shift_right(o, 8, 1))
      if (o.l & #ff) {
        a = ((z.h ≫ k) & #01010101) * #ff;
        b = ((z.l ≫ k) & #01010101) * #ff;
        c = (o.l & #ff) * #01010101;
        if (xor) x.h ⊕= a & c, x.l ⊕= b & c;
        else x.h |= a & c, x.l |= b & c;
      }
    return x;
  }
```

30. Floating point packing and unpacking. Standard IEEE floating binary numbers pack a sign, exponent, and fraction into a tetrabyte or octabyte. In this section we consider basic subroutines that convert between IEEE format and the separate unpacked components.

```
#define ROUND_OFF 1
#define ROUND_UP 2
#define ROUND_DOWN 3
#define ROUND_NEAR 4
```

⟨ Global variables 4 ⟩ +≡

```
int cur_round;      /* the current rounding mode */
```

31. The *fpack* routine takes an octabyte f, a raw exponent e, and a sign s, and packs them into the floating binary number that corresponds to $\pm 2^{e-1076} f$, using a given rounding mode. The value of f should satisfy $2^{54} \le f \le 2^{55}$.

Thus, for example, the floating binary number $+1.0 =$ `#3ff0000000000000` is obtained when $f = 2^{54}$, $e =$ `#3fe`, and $s =$ `'+'`. The raw exponent e is usually one less than the final exponent value; the leading bit of f is essentially added to the exponent. (This trick works nicely for subnormal numbers, when $e < 0$, or in cases where the value of f is rounded upwards to 2^{55}.)

Exceptional events are noted by oring appropriate bits into the global variable *exceptions*. Special considerations apply to underflow, which is not fully specified by Section 7.4 of the IEEE standard: Implementations of the standard are free to choose between two definitions of "tininess" and two definitions of "accuracy loss." `MMIX` determines tininess *after* rounding, hence a result with $e < 0$ is not necessarily tiny; `MMIX` treats accuracy loss as equivalent to inexactness. Thus, a result underflows if and only if it is tiny and either (i) it is inexact or (ii) the underflow trap is enabled. The *fpack* routine sets `U_BIT` in *exceptions* if and only if the result is tiny, `X_BIT` if and only if the result is inexact.

```
#define X_BIT (1 ≪ 8)    /* floating inexact */
#define Z_BIT (1 ≪ 9)    /* floating division by zero */
#define U_BIT (1 ≪ 10)   /* floating underflow */
#define O_BIT (1 ≪ 11)   /* floating overflow */
#define I_BIT (1 ≪ 12)   /* floating invalid operation */
#define W_BIT (1 ≪ 13)   /* float-to-fix overflow */
#define V_BIT (1 ≪ 14)   /* integer overflow */
#define D_BIT (1 ≪ 15)   /* integer divide check */
#define E_BIT (1 ≪ 18)   /* external (dynamic) trap bit */
```

```
⟨ Subroutines 5 ⟩ +≡
  octa fpack ARGS((octa, int, int, int));
  octa fpack(f, e, s, r)
      octa f;     /* the normalized fraction part */
      int e;      /* the raw exponent */
      int s;      /* the sign */
      int r;      /* the rounding mode */
  {
    octa o;
    if (e > #7fd) e = #7ff, o = zero_octa;
    else {
      if (e < 0) {
        if (e < −54) o.h = 0, o.l = 1;
        else { octa oo;
          o = shift_right(f, −e, 1);
          oo = shift_left(o, −e);
          if (oo.l ≠ f.l ∨ oo.h ≠ f.h) o.l |= 1;    /* sticky bit */
        }
        e = 0;
      } else o = f;
    }
    ⟨ Round and return the result 33 ⟩;
  }
```

32.
```
⟨ Global variables 4 ⟩ +≡
  int exceptions;    /* bits possibly destined for rA */
```

33. Everything falls together so nicely here, it's almost too good to be true! The conditional expression in the case for ROUND_NEAR rounds towards an even number in case of a tie.

```
⟨ Round and return the result 33 ⟩ ≡
  if (o.l & 3) exceptions |= X_BIT;
  switch (r) {
  case ROUND_DOWN: if (s ≡ '-') o = incr(o, 3); break;
  case ROUND_UP: if (s ≠ '-') o = incr(o, 3);
  case ROUND_OFF: break;
  case ROUND_NEAR: o = incr(o, o.l & 4 ? 2 : 1); break;
  }
  o = shift_right(o, 2, 1);
  o.h += e ≪ 20;
  if (o.h ≥ #7ff00000) exceptions |= O_BIT + X_BIT;      /* overflow */
  else if (o.h < #100000) exceptions |= U_BIT;      /* tininess */
  if (s ≡ '-') o.h |= sign_bit;
  return o;
```

This code is used in section 31.

34. Similarly, *sfpack* packs a short float, from inputs having the same conventions as *fpack*.

```
⟨ Subroutines 5 ⟩ +≡
  tetra sfpack ARGS((octa, int, int, int));
  tetra sfpack(f, e, s, r)
      octa f;      /* the fraction part */
      int e;      /* the raw exponent */
      int s;      /* the sign */
      int r;      /* the rounding mode */
  {
    register tetra o;
    if (e > #47d) e = #47f, o = 0;
    else {
      o = shift_left(f, 3).h;
      if (f.l & #1fffffff) o |= 1;
      if (e < #380) {
        if (e < #380 − 25) o = 1;
        else { register tetra o0, oo;
          o0 = o;
          o = o ≫ (#380 − e);
          oo = o ≪ (#380 − e);
          if (oo ≠ o0) o |= 1;      /* sticky bit */
        }
        e = #380;
      }
    }
    ⟨ Round and return the short result 35 ⟩;
  }
```

35. ⟨ Round and return the short result 35 ⟩ ≡

```
if (o & 3) exceptions |= X_BIT;
switch (r) {
case ROUND_DOWN: if (s ≡ '-') o += 3; break;
case ROUND_UP: if (s ≠ '-') o += 3;
case ROUND_OFF: break;
case ROUND_NEAR: o += (o & 4 ? 2 : 1); break;
}
o = o ≫ 2;
o += (e − #380) ≪ 23;
if (o ≥ #7f800000) exceptions |= O_BIT + X_BIT;      /* overflow */
else if (o < #800000) exceptions |= U_BIT;      /* tininess */
if (s ≡ '-') o |= sign_bit;
return o;
```

This code is used in section 34.

36. The *funpack* routine is, roughly speaking, the opposite of *fpack*. It takes a given floating point number x and separates out its fraction part f, exponent e, and sign s. It clears *exceptions* to zero. It returns the type of value found: *zro*, *num*, *inf*, or *nan*. When it returns *num*, it will have set f, e, and s to the values from which *fpack* would produce the original number x without exceptions.

```
#define zero_exponent (−1000)      /* zero is assumed to have this exponent */
```

⟨ Other type definitions 36 ⟩ ≡

```
typedef enum {
  zro, num, inf, nan
} ftype;
```

See also section 59.

This code is used in section 1.

37. ⟨ Subroutines 5 ⟩ +≡

```
ftype funpack ARGS((octa, octa *, int *, char *));
ftype funpack(x, f, e, s)
    octa x;     /* the given floating point value */
    octa *f;    /* address where the fraction part should be stored */
    int *e;     /* address where the exponent part should be stored */
    char *s;    /* address where the sign should be stored */
{
  register int ee;
  exceptions = 0;
  *s = (x.h & sign_bit ? '-' : '+');
  *f = shift_left(x, 2);
  f→h &= #3fffff;
  ee = (x.h ≫ 20) & #7ff;
  if (ee) {
    *e = ee − 1;
    f→h |= #400000;
    return (ee < #7ff ? num : f→h ≡ #400000 ∧ ¬f→l ? inf : nan);
  }
  if (¬x.l ∧ ¬f→h) {
    *e = zero_exponent; return zro;
  }
  do { ee−−; *f = shift_left(*f, 1); } while (¬(f→h & #400000));
  *e = ee; return num;
}
```

38. ⟨ Subroutines 5 ⟩ +≡

```
ftype sfunpack ARGS((tetra, octa *, int *, char *));
ftype sfunpack(x, f, e, s)
    tetra x;    /* the given floating point value */
    octa *f;    /* address where the fraction part should be stored */
    int *e;     /* address where the exponent part should be stored */
    char *s;    /* address where the sign should be stored */
{
  register int ee;
  exceptions = 0;
  *s = (x & sign_bit ? '-' : '+');
  f→h = (x ≫ 1) & #3fffff, f→l = x ≪ 31;
  ee = (x ≫ 23) & #ff;
  if (ee) {
    *e = ee + #380 − 1;
    f→h |= #400000;
    return (ee < #ff ? num : (x & #7fffffff) ≡ #7f800000 ? inf : nan);
  }
  if (¬(x & #7fffffff)) {
    *e = zero_exponent; return zro;
  }
  do { ee−−; *f = shift_left(*f, 1); } while (¬(f→h & #400000));
  *e = ee + #380; return num;
}
```

39. Since MMIX downplays 32-bit operations, it uses *sfpack* and *sfunpack* only when loading and storing short floats, or when converting from fixed point to floating point.

```
⟨ Subroutines 5 ⟩ +≡
  octa load_sf ARGS((tetra));
  octa load_sf(z)
      tetra z;      /* 32 bits to be loaded into a 64-bit register */
  {
    octa f, x; int e; char s; ftype t;
    t = sfunpack(z, &f, &e, &s);
    switch (t) {
    case zro: x = zero_octa; break;
    case num: return fpack(f, e, s, ROUND_OFF);
    case inf: x = inf_octa; break;
    case nan: x = shift_right(f, 2, 1); x.h |= #7ff00000; break;
    }
    if (s ≡ '-') x.h |= sign_bit;
    return x;
  }
```

40.

```
⟨ Subroutines 5 ⟩ +≡
  tetra store_sf ARGS((octa));
  tetra store_sf(x)
      octa x;      /* 64 bits to be loaded into a 32-bit word */
  {
    octa f; tetra z; int e; char s; ftype t;
    t = funpack(x, &f, &e, &s);
    switch (t) {
    case zro: z = 0; break;
    case num: return sfpack(f, e, s, cur_round);
    case inf: z = #7f800000; break;
    case nan: if (¬(f.h & #200000)) {
        f.h |= #200000; exceptions |= I_BIT;      /* NaN was signaling */
      }
      z = #7f800000 | (f.h ≪ 1) | (f.l ≫ 31); break;
    }
    if (s ≡ '-') z |= sign_bit;
    return z;
  }
```

41. Floating multiplication and division. The hardest fixed point operations were multiplication and division; but these two operations are the *easiest* to implement in floating point arithmetic, once their fixed point counterparts are available.

```
⟨ Subroutines 5 ⟩ +≡
  octa fmult ARGS((octa, octa));
  octa fmult(y, z)
      octa y, z;
  {
    ftype yt, zt;
    int ye, ze;
    char ys, zs;
    octa x, xf, yf, zf;
    register int xe;
    register char xs;
    yt = funpack(y, &yf, &ye, &ys);
    zt = funpack(z, &zf, &ze, &zs);
    xs = ys + zs - '+';      /* will be '-' when the result is negative */
    switch (4 * yt + zt) {
    ⟨ The usual NaN cases 42 ⟩;
    default: case 4 * zro + zro: case 4 * zro + num: case 4 * num + zro: x = zero_octa; break;
    case 4 * num + inf: case 4 * inf + num: case 4 * inf + inf: x = inf_octa; break;
    case 4 * zro + inf: case 4 * inf + zro: x = standard_NaN;
      exceptions |= I_BIT; break;
    case 4 * num + num: ⟨ Multiply nonzero numbers and return 43 ⟩;
    }
    if (xs ≡ '-') x.h |= sign_bit;
    return x;
  }
```

42. ⟨ The usual NaN cases 42 ⟩ ≡

```
case 4 * nan + nan: if (¬(y.h & #80000)) exceptions |= I_BIT;      /* y is signaling */
case 4 * zro + nan: case 4 * num + nan: case 4 * inf + nan:
  if (¬(z.h & #80000)) exceptions |= I_BIT, z.h |= #80000;
  return z;
case 4 * nan + zro: case 4 * nan + num: case 4 * nan + inf:
  if (¬(y.h & #80000)) exceptions |= I_BIT, y.h |= #80000;
  return y;
```

This code is used in sections 41, 44, 46, and 93.

43. ⟨ Multiply nonzero numbers and return 43 ⟩ ≡

```
  xe = ye + ze - #3fd;      /* the raw exponent */
  x = omult(yf, shift_left(zf, 9));
  if (aux.h ≥ #400000) xf = aux;
  else xf = shift_left(aux, 1), xe--;
  if (x.h ∨ x.l) xf.l |= 1;      /* adjust the sticky bit */
  return fpack(xf, xe, xs, cur_round);
```

This code is used in section 41.

44. ⟨ Subroutines 5 ⟩ +≡

```
octa fdivide ARGS((octa,octa));
octa fdivide(y,z)
    octa y, z;
{
  ftype yt, zt;
  int ye, ze;
  char ys, zs;
  octa x, xf, yf, zf;
  register int xe;
  register char xs;
  yt = funpack(y,&yf,&ye,&ys);
  zt = funpack(z,&zf,&ze,&zs);
  xs = ys + zs - '+';      /* will be '-' when the result is negative */
  switch (4*yt + zt) {
  ⟨ The usual NaN cases 42 ⟩;
  case 4*zro + inf: case 4*zro + num: case 4*num + inf: x = zero_octa; break;
  case 4*num + zro: exceptions |= Z_BIT;
  case 4*inf + num: case 4*inf + zro: x = inf_octa; break;
  default: case 4*zro + zro: case 4*inf + inf: x = standard_NaN;
    exceptions |= I_BIT; break;
  case 4*num + num: ⟨ Divide nonzero numbers and return 45 ⟩;
  }
  if (xs == '-') x.h |= sign_bit;
  return x;
}
```

45. ⟨ Divide nonzero numbers and return 45 ⟩ ≡

```
xe = ye - ze + #3fd;      /* the raw exponent */
xf = odiv(yf, zero_octa, shift_left(zf, 9));
if (xf.h ≥ #800000) {
  aux.l |= xf.l & 1;
  xf = shift_right(xf, 1, 1);
  xe++;
}
if (aux.h ∨ aux.l) xf.l |= 1;      /* adjust the sticky bit */
return fpack(xf, xe, xs, cur_round);
```

This code is used in section 44.

46. Floating addition and subtraction. Now for the bread-and-butter operation, the sum of two floating point numbers. It is not terribly difficult, but many cases need to be handled carefully.

```
⟨ Subroutines 5 ⟩ +≡
  octa fplus ARGS((octa, octa));
  octa fplus(y, z)
      octa y, z;
  {
    ftype yt, zt;
    int ye, ze;
    char ys, zs;
    octa x, xf, yf, zf;
    register int xe, d;
    register char xs;
    yt = funpack(y, &yf, &ye, &ys);
    zt = funpack(z, &zf, &ze, &zs);
    switch (4 * yt + zt) {
    ⟨ The usual NaN cases 42 ⟩;
    case 4 * zro + num: return fpack(zf, ze, zs, ROUND_OFF); break;      /* may underflow */
    case 4 * num + zro: return fpack(yf, ye, ys, ROUND_OFF); break;      /* may underflow */
    case 4 * inf + inf: if (ys ≠ zs) {
        exceptions |= I_BIT; x = standard_NaN; xs = zs; break;
      }
    case 4 * num + inf: case 4 * zro + inf: x = inf_octa; xs = zs; break;
    case 4 * inf + num: case 4 * inf + zro: x = inf_octa; xs = ys; break;
    case 4 * num + num: if (y.h ≠ (z.h ⊕ #80000000) ∨ y.l ≠ z.l) ⟨ Add nonzero numbers and return 47 ⟩;
    default: case 4 * zro + zro: x = zero_octa;
      xs = (ys ≡ zs ? ys : cur_round ≡ ROUND_DOWN ? '-' : '+'); break;
    }
    if (xs ≡ '-') x.h |= sign_bit;
    return x;
  }
```

47. ⟨ Add nonzero numbers and **return** 47 ⟩ ≡

```
  { octa o, oo;
    if (ye < ze ∨ (ye ≡ ze ∧ (yf.h < zf.h ∨ (yf.h ≡ zf.h ∧ yf.l < zf.l)))) ⟨ Exchange y with z 48 ⟩;
    d = ye − ze;
    xs = ys, xe = ye;
    if (d) ⟨ Adjust for difference in exponents 49 ⟩;
    if (ys ≡ zs) {
      xf = oplus(yf, zf);
      if (xf.h ≥ #800000) xe++, d = xf.l & 1, xf = shift_right(xf, 1, 1), xf.l |= d;
    } else {
      xf = ominus(yf, zf);
      if (xf.h ≥ #800000) xe++, d = xf.l & 1, xf = shift_right(xf, 1, 1), xf.l |= d;
      else while (xf.h < #400000) xe−−, xf = shift_left(xf, 1);
    }
    return fpack(xf, xe, xs, cur_round);
  }
```

This code is used in section 46.

48. ⟨ Exchange y with z 48 ⟩ ≡

```
{
  o = yf, yf = zf, zf = o;
  d = ye, ye = ze, ze = d;
  d = ys, ys = zs, zs = d;
}
```

This code is used in sections 47 and 51.

49. Proper rounding requires two bits to the right of the fraction delivered to *fpack*. The first is the true next bit of the result; the other is a "sticky" bit, which is nonzero if any further bits of the true result are nonzero. Sticky rounding to an integer takes x into the number $\lfloor x/2 \rfloor + \lceil x/2 \rceil$.

Some subtleties need to be observed here, in order to prevent the sticky bit from being shifted left. If we did not shift yf left 1 before shifting zf to the right, an incorrect answer would be obtained in certain cases—for example, if $yf = 2^{54}$, $zf = 2^{54} + 2^{53} - 4$, $d = 52$.

⟨ Adjust for difference in exponents 49 ⟩ ≡

```
{
  if (d ≤ 2) zf = shift_right(zf, d, 1);      /* exact result */
  else if (d > 54) zf.h = 0, zf.l = 1;        /* tricky but OK */
  else {
    if (ys ≠ zs) d--, xe--, yf = shift_left(yf, 1);
    o = zf;
    zf = shift_right(o, d, 1);
    oo = shift_left(zf, d);
    if (oo.l ≠ o.l ∨ oo.h ≠ o.h) zf.l |= 1;
  }
}
```

This code is used in section 47.

50. The comparison of floating point numbers with respect to ϵ shares some of the characteristics of floating point addition/subtraction. In some ways it is simpler, and in other ways it is more difficult; we might as well deal with it now.

Subroutine *fepscomp*(y,z,e,s) returns 2 if y, z, or e is a NaN or e is negative. It returns 1 if $s=0$ and $y\approx z$ (e) or if $s\neq 0$ and $y\sim z$ (e), as defined in Section 4.2.2 of *Seminumerical Algorithms*; otherwise it returns 0.

```
⟨ Subroutines 5 ⟩ +≡
  int fepscomp ARGS((octa, octa, octa, int));
  int fepscomp(y, z, e, s)
      octa y, z, e;      /* the operands */
      int s;      /* test similarity? */
  {
    octa yf, zf, ef, o, oo;
    int ye, ze, ee;
    char ys, zs, es;
    register int yt, zt, et, d;
    et = funpack(e, &ef, &ee, &es);
    if (es ≡ '-') return 2;
    switch (et) {
    case nan: return 2;
    case inf: ee = 10000;
    case num: case zro: break;
    }
    yt = funpack(y, &yf, &ye, &ys);
    zt = funpack(z, &zf, &ze, &zs);
    switch (4 * yt + zt) {
    case 4 * nan + nan: case 4 * nan + inf: case 4 * nan + num: case 4 * nan + zro: case 4 * inf + nan:
      case 4 * num + nan: case 4 * zro + nan: return 2;
    case 4 * inf + inf: return (ys ≡ zs ∨ ee ≥ 1023);
    case 4 * inf + num: case 4 * inf + zro: case 4 * num + inf: case 4 * zro + inf: return (s ∧ ee ≥ 1022);
    case 4 * zro + zro: return 1;
    case 4 * zro + num: case 4 * num + zro: if (¬s) return 0;
    case 4 * num + num: break;
    }
    ⟨ Compare two numbers with respect to epsilon and return 51 ⟩;
  }
```

51. The relation $y \approx z\ (\epsilon)$ reduces to $y \sim z\ (\epsilon/2^d)$, if d is the difference between the larger and smaller exponents of y and z.

```
⟨ Compare two numbers with respect to epsilon and return 51 ⟩ ≡
  ⟨ Unsubnormalize y and z, if they are subnormal 52 ⟩;
  if (ye < ze ∨ (ye ≡ ze ∧ (yf.h < zf.h ∨ (yf.h ≡ zf.h ∧ yf.l < zf.l)))) ⟨ Exchange y with z 48 ⟩;
  if (ze ≡ zero_exponent) ze = ye;
  d = ye − ze;
  if (¬s) ee −= d;
  if (ee ≥ 1023) return 1;      /* if ε ≥ 2, z ∈ N_ε(y) */
  ⟨ Compute the difference of fraction parts, o 53 ⟩;
  if (¬o.h ∧ ¬o.l) return 1;
  if (ee < 968) return 0;      /* if y ≠ z and ε < 2^−54, y ≁ z */
  if (ee ≥ 1021) ef = shift_left(ef, ee − 1021);
  else ef = shift_right(ef, 1021 − ee, 1);
  return o.h < ef.h ∨ (o.h ≡ ef.h ∧ o.l ≤ ef.l);
```

This code is used in section 50.

52.

```
⟨ Unsubnormalize y and z, if they are subnormal 52 ⟩ ≡
  if (ye < 0 ∧ yt ≠ zro) yf = shift_left(y, 2), ye = 0;
  if (ze < 0 ∧ zt ≠ zro) zf = shift_left(z, 2), ze = 0;
```

This code is used in section 51.

53. At this point $y \sim z$ if and only if

$$yf + (-1)^{[ys=zs]} zf / 2^d \le 2^{ee-1021} ef = 2^{55}\epsilon.$$

We need to evaluate this relation without overstepping the bounds of our simulated 64-bit registers.

When $d > 2$, the difference of fraction parts might not fit exactly in an octabyte; in that case the numbers are not similar unless $\epsilon > 3/8$, and we replace the difference by the ceiling of the true result. When $\epsilon < 1/8$, our program essentially replaces $2^{55}\epsilon$ by $\lfloor 2^{55}\epsilon \rfloor$. These truncations are not needed simultaneously. Therefore the logic is justified by the facts that, if n is an integer, we have $x \le n$ if and only if $\lceil x \rceil \le n$; $n \le x$ if and only if $n \le \lfloor x \rfloor$. (Notice that the concept of "sticky bit" is *not* appropriate here.)

```
⟨ Compute the difference of fraction parts, o 53 ⟩ ≡
  if (d > 54) o = zero_octa, oo = zf;
  else o = shift_right(zf, d, 1), oo = shift_left(o, d);
  if (oo.h ≠ zf.h ∨ oo.l ≠ zf.l) {      /* truncated result, hence d > 2 */
    if (ee < 1020) return 0;      /* difference is too large for similarity */
    o = incr(o, ys ≡ zs ? 0 : 1);      /* adjust for ceiling */
  }
  o = (ys ≡ zs ? ominus(yf, o) : oplus(yf, o));
```

This code is used in section 51.

54. Floating point output conversion. The *print_float* routine converts an octabyte to a floating decimal representation that will be input as precisely the same value.

```
⟨ Subroutines 5 ⟩ +≡
  static void bignum_times_ten ARGS((bignum *));
  static void bignum_dec ARGS((bignum *, bignum *, tetra));
  static int bignum_compare ARGS((bignum *, bignum *));
  void print_float ARGS((octa));
  void print_float(x)
      octa x;
  {
    ⟨ Local variables for print_float 56 ⟩;
    if (x.h & sign_bit) printf("-");
    ⟨ Extract the exponent e and determine the fraction interval [f . . g] or (f . . g) 55 ⟩;
    ⟨ Store f and g as multiprecise integers 63 ⟩;
    ⟨ Compute the significant digits s and decimal exponent e 64 ⟩;
    ⟨ Print the significant digits with proper context 67 ⟩;
  }
```

55. One way to visualize the problem being solved here is to consider the vastly simpler case in which there are only 2-bit exponents and 2-bit fractions. Then the sixteen possible 4-bit combinations have the following interpretations:

0000	$[0 \,..\, 0.125]$
0001	$(0.125 \,..\, 0.375)$
0010	$[0.375 \,..\, 0.625]$
0011	$(0.625 \,..\, 0.875)$
0100	$[0.875 \,..\, 1.125]$
0101	$(1.125 \,..\, 1.375)$
0110	$[1.375 \,..\, 1.625]$
0111	$(1.625 \,..\, 1.875)$
1000	$[1.875 \,..\, 2.25]$
1001	$(2.25 \,..\, 2.75)$
1010	$[2.75 \,..\, 3.25]$
1011	$(3.25 \,..\, 3.75)$
1100	$[3.75 \,..\, \infty]$
1101	$\mathrm{NaN}(0 \,..\, 0.375)$
1110	$\mathrm{NaN}[0.375 \,..\, 0.625]$
1111	$\mathrm{NaN}(0.625 \,..\, 1)$

Notice that the interval is closed, $[f \,..\, g]$, when the fraction part is even; it is open, $(f \,..\, g)$, when the fraction part is odd. The printed outputs for these sixteen values, if we actually were dealing with such short exponents and fractions, would be `0.`, `.2`, `.5`, `.7`, `1.`, `1.2`, `1.5`, `1.7`, `2.`, `2.5`, `3.`, `3.5`, `Inf`, `NaN.2`, `NaN`, `NaN.8`, respectively.

```
⟨ Extract the exponent e and determine the fraction interval [f..g] or (f..g) 55 ⟩ ≡
  f = shift_left(x, 1);
  e = f.h ≫ 21;
  f.h &= #1fffff;
  if (¬f.h ∧ ¬f.l) ⟨ Handle the special case when the fraction part is zero 57 ⟩
  else {
    g = incr(f, 1);
    f = incr(f, −1);
    if (¬e) e = 1;      /* subnormal */
    else if (e ≡ #7ff) {
      printf("NaN");
      if (g.h ≡ #100000 ∧ g.l ≡ 1) return;      /* the "standard" NaN */
      e = #3ff;      /* extreme NaNs come out OK even without adjusting f or g */
    } else f.h |= #200000, g.h |= #200000;
  }
```

This code is used in section 54.

56.

```
⟨ Local variables for print_float 56 ⟩ ≡
  octa f, g;      /* lower and upper bounds on the fraction part */
  register int e;      /* exponent part */
  register int j, k;      /* all purpose indices */
```

See also section 66.

This code is used in section 54.

57. The transition points between exponents correspond to powers of 2. At such points the interval extends only half as far to the left of that power of 2 as it does to the right. For example, in the 4-bit minifloat numbers considered above, case 1000 corresponds to the interval [1.875 .. 2.25].

⟨ Handle the special case when the fraction part is zero 57 ⟩ ≡

```
{
  if (¬e) {
    printf("0."); return;
  }
  if (e ≡ #7ff) {
    printf("Inf"); return;
  }
  e−−;
  f.h = #3fffff, f.l = #ffffffff;
  g.h = #400000, g.l = 2;
}
```

This code is used in section 55.

58. We want to find the "simplest" value in the interval corresponding to the given number, in the sense that it has fewest significant digits when expressed in decimal notation. Thus, for example, if the floating point number can be described by a relatively short string such as '`.1`' or '`37e100`', we want to discover that representation.

The basic idea is to generate the decimal representations of the two endpoints of the interval, outputting the leading digits where both endpoints agree, then making a final decision at the first place where they disagree.

The "simplest" value is not always unique. For example, in the case of 4-bit minifloat numbers we could represent the bit pattern 0001 as either `.2` or `.3`, and we could represent 1001 in five equally short ways: `2.3` or `2.4` or `2.5` or `2.6` or `2.7`. The algorithm below tries to choose the middle possibility in such cases.

[A solution to the analogous problem for fixed-point representations, without the additional complication of round-to-even, was used by the author in the program for TEX; see *Beauty is Our Business* (Springer, 1990), 233–242.]

Suppose we are given two fractions f and g, where $0 \le f < g < 1$, and we want to compute the shortest decimal in the closed interval $[f\,..\,g]$. If $f = 0$, we are done. Otherwise let $10f = d + f'$ and $10g = e + g'$, where $0 \le f' < 1$ and $0 \le g' < 1$. If $d < e$, we can terminate by outputting any of the digits $d+1, \ldots, e$; otherwise we output the common digit $d = e$, and repeat the process on the fractions $0 \le f' < g' < 1$. A similar procedure works with respect to the open interval $(f\,..\,g)$.

59. The program below carries out the stated algorithm by using multiprecision arithmetic on 77-place integers with 28 bits each. This choice facilitates multiplication by 10, and allows us to deal with the whole range of floating binary numbers using fixed point arithmetic. We keep track of the leading and trailing digit positions so that trivial operations on zeros are avoided.

If f points to a **bignum**, its radix-2^{28} digits are $f\to dat[0]$ through $f\to dat[76]$, from most significant to least significant. We assume that all digit positions are zero unless they lie in the subarray between indices $f\to a$ and $f\to b$, inclusive. Furthermore, both $f\to dat[f\to a]$ and $f\to dat[f\to b]$ are nonzero, unless $f\to a = f\to b = bignum\_prec-1$.

The **bignum** data type can be used with any radix less than 2^{32}; we will use it later with radix 10^9. The *dat* array is made large enough to accommodate both applications.

```
#define bignum_prec 157 /* would be 77 if we cared only about print_float */
⟨ Other type definitions 36 ⟩ +≡
  typedef struct {
    int a; /* index of the most significant digit */
    int b; /* index of the least significant digit; must be ≥ a */
    tetra dat[bignum_prec]; /* the digits; undefined except between a and b */
  } bignum;
```

60. Here, for example, is how we go from f to $10f$, assuming that overflow will not occur and that the radix is 2^{28}:

```
⟨ Subroutines 5 ⟩ +≡
  static void bignum_times_ten(f)
      bignum *f;
  {
    register tetra *p, *q;
    register tetra x, carry;
    for (p = &f→dat[f→b], q = &f→dat[f→a], carry = 0; p ≥ q; p--) {
      x = *p * 10 + carry;
      *p = x & #fffffff;
      carry = x ≫ 28;
    }
    *p = carry;
    if (carry) f→a--;
    if (f→dat[f→b] ≡ 0 ∧ f→b > f→a) f→b--;
  }
```

61. And here is how we test whether $f < g$, $f = g$, or $f > g$, using any radix whatever:

```
⟨ Subroutines 5 ⟩ +≡
  static int bignum_compare(f, g)
      bignum *f, *g;
  {
    register tetra *p, *pp, *q, *qq;
    if (f→a ≠ g→a) return f→a > g→a ? -1 : 1;
    pp = &f→dat[f→b], qq = &g→dat[g→b];
    for (p = &f→dat[f→a], q = &g→dat[g→a]; p ≤ pp; p++, q++) {
      if (*p ≠ *q) return *p < *q ? -1 : 1;
      if (q ≡ qq) return p < pp;
    }
    return -1;
  }
```

62. The following subroutine subtracts g from f, assuming that $f \geq g > 0$ and using a given radix.

```
⟨ Subroutines 5 ⟩ +≡
  static void bignum_dec(f, g, r)
      bignum *f, *g;
      tetra r;      /* the radix */
  {
    register tetra *p, *q, *qq;
    register int x, borrow;
    while (g→b > f→b) f→dat[++f→b] = 0;
    qq = &g→dat[g→a];
    for (p = &f→dat[g→b], q = &g→dat[g→b], borrow = 0; q ≥ qq; p--, q--) {
      x = *p - *q - borrow;
      if (x ≥ 0) borrow = 0, *p = x;
      else borrow = 1, *p = x + r;
    }
    for ( ; borrow; p--)
      if (*p) borrow = 0, *p = *p - 1;
      else *p = r - 1;
    while (f→dat[f→a] ≡ 0) {
      if (f→a ≡ f→b) {      /* the result is zero */
        f→a = f→b = bignum_prec - 1, f→dat[bignum_prec - 1] = 0;
        return;
      }
      f→a++;
    }
    while (f→dat[f→b] ≡ 0) f→b--;
  }
```

63. Armed with these subroutines, we are ready to solve the problem. The first task is to put the numbers into **bignum** form. If the exponent is e, the number destined for digit $dat[k]$ will consist of the rightmost 28 bits of the given fraction after it has been shifted right $c - e - 28k$ bits, for some constant c. We choose c so that, when e has its maximum value `#7ff`, the leading digit will go into position $dat[1]$, and so that when the number to be printed is exactly 1 the integer part of g will also be exactly 1.

```
#define magic_offset 2112      /* the constant c that makes it work */
#define origin 37      /* the radix point follows dat[37] */
⟨ Store f and g as multiprecise integers 63 ⟩ ≡
  k = (magic_offset - e)/28;
  ff.dat[k - 1] = shift_right(f, magic_offset + 28 - e - 28 * k, 1).l & #fffffff;
  gg.dat[k - 1] = shift_right(g, magic_offset + 28 - e - 28 * k, 1).l & #fffffff;
  ff.dat[k] = shift_right(f, magic_offset - e - 28 * k, 1).l & #fffffff;
  gg.dat[k] = shift_right(g, magic_offset - e - 28 * k, 1).l & #fffffff;
  ff.dat[k + 1] = shift_left(f, e + 28 * k - (magic_offset - 28)).l & #fffffff;
  gg.dat[k + 1] = shift_left(g, e + 28 * k - (magic_offset - 28)).l & #fffffff;
  ff.a = (ff.dat[k - 1] ? k - 1 : k);
  ff.b = (ff.dat[k + 1] ? k + 1 : k);
  gg.a = (gg.dat[k - 1] ? k - 1 : k);
  gg.b = (gg.dat[k + 1] ? k + 1 : k);
```

This code is used in section 54.

64. If e is sufficiently small, the fractions f and g will be less than 1, and we can use the stated algorithm directly. Of course, if e is extremely small, a lot of leading zeros need to be lopped off; in the worst case, we may have to multiply f and g by 10 more than 300 times. But hey, we don't need to do that extremely often, and computers are pretty fast nowadays.

In the small-exponent case, the computation always terminates before f becomes zero, because the interval endpoints are fractions with denominator 2^t for some $t > 50$.

The invariant relations $ff.dat[ff.a] \neq 0$ and $gg.dat[gg.a] \neq 0$ are not maintained by the computation here, when $ff.a = origin$ or $gg.a = origin$. But no harm is done, because *bignum_compare* is not used.

```
⟨ Compute the significant digits s and decimal exponent e 64 ⟩ ≡
  if (e > #401) ⟨ Compute the significant digits in the large-exponent case 65 ⟩
  else {      /* if e ≤ #401 we have gg.a ≥ origin and gg.dat[origin] ≤ 8 */
    if (ff.a > origin) ff.dat[origin] = 0;
    for (e = 1, p = s; gg.a > origin ∨ ff.dat[origin] ≡ gg.dat[origin]; ) {
      if (gg.a > origin) e--;
      else *p++ = ff.dat[origin] + '0', ff.dat[origin] = 0, gg.dat[origin] = 0;
      bignum_times_ten(&ff);
      bignum_times_ten(&gg);
    }
    *p++ = ((ff.dat[origin] + 1 + gg.dat[origin]) ≫ 1) + '0';      /* the middle digit */
  }
  *p = '\0';      /* terminate the string s */
```

This code is used in section 54.

65. When e is large, we use the stated algorithm by considering f and g to be fractions whose denominator is a power of 10.

An interesting case arises when the number to be converted is `#44ada56a4b0835bf`, since the interval turns out to be

$$(69999999999999991611392 \quad .. \quad 70000000000000000000000).$$

If this were a closed interval, we could simply give the answer `7e22`; but the number `7e22` actually corresponds to `#44ada56a4b0835c0` because of the round-to-even rule. Therefore the correct answer is, say, `6.9999999999999995e22`. This example shows that we need a slightly different strategy in the case of open intervals; we cannot simply look at the first position in which the endpoints have different decimal digits. Therefore we change the invariant relation to $0 \le f < g \le 1$, when open intervals are involved, and we do not terminate the process when $f = 0$ or $g = 1$.

```
⟨ Compute the significant digits in the large-exponent case 65 ⟩ ≡
  { register int open = x.l & 1;
    tt.dat[origin] = 10;
    tt.a = tt.b = origin;
    for (e = 1; bignum_compare(&gg, &tt) ≥ open; e++) bignum_times_ten(&tt);
    p = s;
    while (1) {
      bignum_times_ten(&ff);
      bignum_times_ten(&gg);
      for (j = '0'; bignum_compare(&ff, &tt) ≥ 0; j++)
        bignum_dec(&ff, &tt, #10000000), bignum_dec(&gg, &tt, #10000000);
      if (bignum_compare(&gg, &tt) ≥ open) break;
      *p++ = j;
      if (ff.a ≡ bignum_prec − 1 ∧ ¬open) goto done;      /* f = 0 in a closed interval */
    }
    for (k = j; bignum_compare(&gg, &tt) ≥ open; k++) bignum_dec(&gg, &tt, #10000000);
    *p++ = (j + 1 + k) ≫ 1;      /* the middle digit */
  done: ;
  }
```

This code is used in section 64.

66. The length of string s will be at most 17. For if f and g agree to 17 places, we have $g/f < 1 + 10^{-16}$; but the ratio g/f is always $\ge (1 + 2^{-52} + 2^{-53})/(1 + 2^{-52} - 2^{-53}) > 1 + 2 \times 10^{-16}$.

```
⟨ Local variables for print_float 56 ⟩ +≡
  bignum ff, gg;      /* fractions or numerators of fractions */
  bignum tt;      /* power of ten (used as the denominator) */
  char s[18];
  register char *p;
```

67. At this point the significant digits are in string s, and $s[0] \neq$ `'0'`. If we put a decimal point at the left of s, the result should be multiplied by 10^e.

We prefer the output '`300.`' to the form '`3e2`', and we prefer '`.03`' to '`3e-2`'. In general, the output will use an explicit exponent only if the alternative would take more than 18 characters.

```
⟨ Print the significant digits with proper context 67 ⟩ ≡
  if (e > 17 ∨ e < (int) strlen(s) − 17) printf("%c%s%se%d", s[0], (s[1] ? "." : ""), s + 1, e − 1);
  else if (e < 0) printf(".%0*d%s", −e, 0, s);
  else if ((int) strlen(s) ≥ e) printf("%.*s.%s", e, s, s + e);
  else printf("%s%0*d.", s, e − (int) strlen(s), 0);
```

This code is used in section 54.

68. Floating point input conversion. Going the other way, we want to be able to convert a given decimal number into its floating binary equivalent. The following syntax is supported:

⟨ digit ⟩ −→ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨ digit string ⟩ −→ ⟨ digit ⟩ | ⟨ digit string ⟩⟨ digit ⟩
⟨ decimal string ⟩ −→ ⟨ digit string ⟩`.` | `.`⟨ digit string ⟩ | ⟨ digit string ⟩`.`⟨ digit string ⟩
⟨ optional sign ⟩ −→ ⟨ empty ⟩ | `+` | `-`
⟨ exponent ⟩ −→ `e`⟨ optional sign ⟩⟨ digit string ⟩
⟨ optional exponent ⟩ −→ ⟨ empty ⟩ | ⟨ exponent ⟩
⟨ floating magnitude ⟩ −→ ⟨ digit string ⟩⟨ exponent ⟩ | ⟨ decimal string ⟩⟨ optional exponent ⟩ |
`Inf` | `NaN` | `NaN.`⟨ digit string ⟩
⟨ floating constant ⟩ −→ ⟨ optional sign ⟩⟨ floating magnitude ⟩
⟨ decimal constant ⟩ −→ ⟨ optional sign ⟩⟨ digit string ⟩

For example, '`-3.`' is the floating constant `#c008000000000000`; '`1e3`' and '`1000`' are both equivalent to `#408f400000000000`; '`NaN`' and '`+NaN.5`' are both equivalent to `#7ff8000000000000`.

The *scan_const* routine looks at a given string and finds the longest initial substring that matches the syntax of either ⟨ decimal constant ⟩ or ⟨ floating constant ⟩. It puts the corresponding value into the global octabyte variable *val*; it also puts the position of the first unscanned character in the global pointer variable *next_char*. It returns 1 if a floating constant was found, 0 if a decimal constant was found, −1 if nothing was found. A decimal constant that doesn't fit in an octabyte is computed modulo 2^{64}.

The value of *exceptions* set by *scan_const* is not necessarily correct.

```
⟨ Subroutines 5 ⟩ +≡
  static void bignum_double ARGS((bignum *));
  int scan_const ARGS((char *));
  int scan_const(s)
      char *s;
  {
    ⟨ Local variables for scan_const 70 ⟩;
    val.h = val.l = 0;
    p = s;
    if (*p ≡ '+' ∨ *p ≡ '-') sign = *p++; else sign = '+';
    if (strncmp(p, "NaN", 3) ≡ 0) NaN = true, p += 3;
    else NaN = false;
    if ((isdigit(*p) ∧ ¬NaN) ∨ (*p ≡ '.' ∧ isdigit(*(p + 1)))) ⟨ Scan a number and return 73 ⟩;
    if (NaN) ⟨ Return the standard NaN 71 ⟩;
    if (strncmp(p, "Inf", 3) ≡ 0) ⟨ Return infinity 72 ⟩;
    next_char = s; return −1;
  }
```

69.
```
⟨ Global variables 4 ⟩ +≡
  octa val;      /* value returned by scan_const */
  char *next_char;      /* pointer returned by scan_const */
```

70.
```
⟨ Local variables for scan_const 70 ⟩ ≡
  register char *p, *q;      /* for string manipulations */
  register bool NaN;      /* are we processing a NaN? */
  int sign;      /* '+' or '-' */
```

See also sections 76 and 81.

This code is used in section 68.

71. ⟨ Return the standard NaN 71 ⟩ ≡

```
{
  next_char = p;
  val.h = #600000, exp = #3fe;
  goto packit;
}
```

This code is used in section 68.

72. ⟨ Return infinity 72 ⟩ ≡

```
{
  next_char = p + 3;
  goto make_it_infinite;
}
```

This code is used in section 68.

73. We saw above that a string of at most 17 digits is enough to characterize a floating point number, for purposes of output. But a much longer buffer for digits is needed when we're doing input. For example, consider the borderline quantity $(1 + 2^{-53})/2^{1022}$; its decimal expansion, when written out exactly, is a number with more than 750 significant digits: `2.2250738585...8125e-308`. If *any one* of those digits is increased, or if additional nonzero digits are added as in `2.2250738585...81250000001e-308`, the rounded value is supposed to change from `#0010000000000000` to `#0010000000000001`.

We assume here that the user prefers a perfectly correct answer to a speedy almost-correct one, so we implement the most general case.

⟨ Scan a number and **return** 73 ⟩ ≡

```
{
  for (q = buf0, dec_pt = (char *) 0; isdigit(*p); p++) {
    val = oplus(val, shift_left(val, 2));      /* multiply by 5 */
    val = incr(shift_left(val, 1), *p - '0');
    if (q > buf0 ∨ *p ≠ '0') {
      if (q < buf_max) *q++ = *p;
      else if (*(q - 1) ≡ '0') *(q - 1) = *p; }
  }
  if (NaN) *q++ = '1';
  if (*p ≡ '.') ⟨ Scan a fraction part 74 ⟩;
  next_char = p;
  exp = 0;
  if (*p ≡ 'e' ∧ ¬NaN) ⟨ Scan an exponent 77 ⟩;
  if (dec_pt) ⟨ Return a floating point constant 78 ⟩;
  if (sign ≡ '-') val = ominus(zero_octa, val);
  return 0;
}
```

This code is used in section 68.

74. ⟨Scan a fraction part 74⟩ ≡

```
{
  dec_pt = q;
  p++;
  for (zeros = 0; isdigit(*p); p++)
    if (*p ≡ '0' ∧ q ≡ buf0) zeros++;
    else if (q < buf_max) *q++ = *p;
    else if (*(q − 1) ≡ '0') *(q − 1) = *p;
}
```

This code is used in section 73.

75. The buffer needs room for eight digits of padding at the left, followed by up to 1022+53−307 significant digits, followed by a "sticky" digit at position *buf_max* − 1, and eight more digits of padding.

```
#define buf0 (buf + 8)
#define buf_max (buf + 777)
```

⟨Global variables 4⟩ +≡

```
static char buf[785] = "00000000";    /* where we put significant input digits */
```

76. ⟨Local variables for *scan_const* 70⟩ +≡

```
register char *dec_pt;    /* position of decimal point in buf */
register int exp;    /* scanned exponent; later used for raw binary exponent */
register int zeros = 0;    /* leading zeros removed after decimal point */
```

77. Here we don't advance *next_char* and force a decimal point until we know that a syntactically correct exponent exists.

The code here will convert extra-large inputs like '`9e+9999999999999999`' into ∞ and extra-small inputs into zero. Strange inputs like '`-00.0e9999999`' must also be accommodated. (But we *don't* try to deliver precise answers when there are a billion or more leading zeros.)

⟨Scan an exponent 77⟩ ≡

```
{ register char exp_sign;
  p++;
  if (*p ≡ '+' ∨ *p ≡ '-') exp_sign = *p++; else exp_sign = '+';
  if (isdigit(*p)) {
    for (exp = *p++ − '0'; isdigit(*p); p++)
      if (exp < 100000000) exp = 10 * exp + *p − '0';
    if (¬dec_pt) dec_pt = q, zeros = 0;
    if (exp_sign ≡ '-') exp = −exp;
    next_char = p;
  }
}
```

This code is used in section 73.

78. ⟨Return a floating point constant 78⟩ ≡

```
{
  ⟨Move the digits from buf to ff 79⟩;
  ⟨Determine the binary fraction and binary exponent 83⟩;
packit: ⟨Pack and round the answer 84⟩;
  return 1;
}
```

This code is used in section 73.

79. Now we get ready to compute the binary fraction bits, by putting the scanned input digits into a multiprecision fixed-point accumulator *ff* that spans the full necessary range. After this step, the number that we want to convert to floating binary will appear in $ff.dat[ff.a], ff.dat[ff.a+1], \ldots, ff.dat[ff.b]$. The radix-$10^9$ digit in $ff[36-k]$ is understood to be multiplied by 10^{9k}, for $36 \ge k \ge -120$.

```
⟨ Move the digits from buf to ff 79 ⟩ ≡
  x = buf + 341 + zeros − dec_pt − exp;
  if (q ≡ buf0 ∨ x ≥ 1413) {
    exp = −99999; goto packit;
  }
  if (x < 0) {
  make_it_infinite: exp = 99999; goto packit;
  }
  ff.a = x/9;
  for (p = q; p < q + 8; p++) *p = '0';      /* pad with trailing zeros */
  q = q − 1 − (q + 341 + zeros − dec_pt − exp) % 9;      /* compute stopping place in buf */
  for (p = buf0 − x % 9, k = ff.a; p ≤ q ∧ k ≤ 156; p += 9, k++)
    ⟨ Put the 9-digit number *p ... *(p + 8) into ff.dat[k] 80 ⟩;
  ff.b = k − 1;
  for (x = 0; p ≤ q; p += 9)
    if (strncmp(p, "000000000", 9) ≠ 0) x = 1;
  ff.dat[156] += x;      /* nonzero digits that fall off the right are sticky */
  while (ff.dat[ff.b] ≡ 0) ff.b−−;
```

This code is used in section 78.

80.

```
⟨ Put the 9-digit number *p ... *(p + 8) into ff.dat[k] 80 ⟩ ≡
  {
    for (x = *p − '0', pp = p + 1; pp < p + 9; pp++) x = 10 * x + *pp − '0';
    ff.dat[k] = x;
  }
```

This code is used in section 79.

81.

```
⟨ Local variables for scan_const 70 ⟩ +≡
  register int k, x;
  register char *pp;
  bignum ff, tt;
```

82. Here's a subroutine that is dual to *bignum_times_ten*. It changes f to $2f$, assuming that overflow will not occur and that the radix is 10^9.

```
⟨ Subroutines 5 ⟩ +≡
  static void bignum_double(f)
      bignum *f;
  {
    register tetra *p, *q;
    register int x, carry;
    for (p = &f→dat[f→b], q = &f→dat[f→a], carry = 0; p ≥ q; p−−) {
      x = *p + *p + carry;
      if (x ≥ 1000000000) carry = 1, *p = x − 1000000000;
      else carry = 0, *p = x;
    }
    *p = carry;
    if (carry) f→a−−;
    if (f→dat[f→b] ≡ 0 ∧ f→b > f→a) f→b−−;
  }
```

83.
```
⟨ Determine the binary fraction and binary exponent 83 ⟩ ≡
  val = zero_octa;
  if (ff.a > 36) {
    for (exp = #3fe; ff.a > 36; exp−−) bignum_double(&ff);
    for (k = 54; k; k−−) {
      if (ff.dat[36]) {
        if (k ≥ 32) val.h |= 1 ≪ (k − 32); else val.l |= 1 ≪ k;
        ff.dat[36] = 0;
        if (ff.b ≡ 36) break;      /* break if ff now zero */
      }
      bignum_double(&ff);
    }
  } else {
    tt.a = tt.b = 36, tt.dat[36] = 2;
    for (exp = #3fe; bignum_compare(&ff, &tt) ≥ 0; exp++) bignum_double(&tt);
    for (k = 54; k; k−−) {
      bignum_double(&ff);
      if (bignum_compare(&ff, &tt) ≥ 0) {
        if (k ≥ 32) val.h |= 1 ≪ (k − 32); else val.l |= 1 ≪ k;
        bignum_dec(&ff, &tt, 1000000000);
        if (ff.a ≡ bignum_prec − 1) break;      /* break if ff now zero */
      }
    }
  }
  if (k ≡ 0) val.l |= 1;      /* add sticky bit if ff nonzero */
```

This code is used in section 78.

84. We need to be careful that the input '`NaN.99999999999999999`' doesn't get rounded up; it is supposed to yield `#7fffffffffffffff`.

Although the input '`NaN.0`' is illegal, strictly speaking, we silently convert it to `#7ff0000000000001`—a number that would be output as '`NaN.0000000000000002`'.

⟨ Pack and round the answer 84 ⟩ ≡

```
val = fpack(val, exp, sign, ROUND_NEAR);
if (NaN) {
  if ((val.h & #7fffffff) ≡ #40000000) val.h |= #7fffffff, val.l = #ffffffff;
  else if ((val.h & #7fffffff) ≡ #3ff00000 ∧ ¬val.l) val.h |= #40000000, val.l = 1;
  else val.h |= #40000000;
}
```

This code is used in section 78.

85. Floating point remainders. In this section we implement the remainder of the floating point operations—one of which happens to be the operation of taking the remainder.

The easiest task remaining is to compare two floating point quantities. Routine *fcomp* returns -1 if $y < z$, 0 if $y = z$, $+1$ if $y > z$, and $+2$ if y and z are unordered.

```
⟨ Subroutines 5 ⟩ +≡
  int fcomp ARGS((octa, octa));
  int fcomp(y, z)
      octa y, z;
  {
    ftype yt, zt;
    int ye, ze;
    char ys, zs;
    octa yf, zf;
    register int x;
    yt = funpack(y, &yf, &ye, &ys);
    zt = funpack(z, &zf, &ze, &zs);
    switch (4 * yt + zt) {
    case 4 * nan + nan: case 4 * zro + nan: case 4 * num + nan: case 4 * inf + nan: case 4 * nan + zro:
      case 4 * nan + num: case 4 * nan + inf: return 2;
    default: case 4 * zro + zro: return 0;
    case 4 * zro + num: case 4 * num + zro: case 4 * zro + inf: case 4 * inf + zro: case 4 * num + num:
      case 4 * num + inf: case 4 * inf + num: case 4 * inf + inf:
      if (ys ≠ zs) x = 1;
      else if (y.h > z.h) x = 1;
      else if (y.h < z.h) x = -1;
      else if (y.l > z.l) x = 1;
      else if (y.l < z.l) x = -1;
      else return 0;
      break;
    }
    return (ys ≡ '-' ? -x : x);
  }
```

86. Several `MMIX` operations act on a single floating point number and accept an arbitrary rounding mode. For example, consider the operation of rounding to the nearest floating point integer:

```
⟨ Subroutines 5 ⟩ +≡
  octa fintegerize ARGS((octa, int));
  octa fintegerize(z, r)
      octa z;      /* the operand */
      int r;      /* the rounding mode */
  {
    ftype zt;
    int ze;
    char zs;
    octa xf, zf;
    zt = funpack(z, &zf, &ze, &zs);
    if (¬r) r = cur_round;
    switch (zt) {
    case nan: if (¬(z.h & #80000)) { exceptions |= I_BIT; z.h |= #80000; }
    case inf: case zro: default: return z;
    case num: ⟨ Integerize and return 87 ⟩;
    }
  }
```

87. ⟨ Integerize and **return** 87 ⟩ ≡

```
  if (ze ≥ 1074) return fpack(zf, ze, zs, ROUND_OFF);      /* already an integer */
  if (ze ≤ 1020) xf.h = 0, xf.l = 1;
  else { octa oo;
    xf = shift_right(zf, 1074 − ze, 1);
    oo = shift_left(xf, 1074 − ze);
    if (oo.l ≠ zf.l ∨ oo.h ≠ zf.h) xf.l |= 1;      /* sticky bit */
  }
  switch (r) {
  case ROUND_DOWN: if (zs ≡ '-') xf = incr(xf, 3); break;
  case ROUND_UP: if (zs ≠ '-') xf = incr(xf, 3);
  case ROUND_OFF: break;
  case ROUND_NEAR: xf = incr(xf, xf.l & 4 ? 2 : 1); break;
  }
  xf.l &= #fffffffc;
  if (ze ≥ 1022) return fpack(shift_left(xf, 1074 − ze), ze, zs, ROUND_OFF);
  if (xf.l) xf.h = #3ff00000, xf.l = 0;
  if (zs ≡ '-') xf.h |= sign_bit;
  return xf;
```

This code is used in section 86.

88. To convert floating point to fixed point, we use *fixit*.

```
⟨ Subroutines 5 ⟩ +≡
  octa fixit ARGS((octa, int));
  octa fixit(z, r)
      octa z;      /* the operand */
      int r;      /* the rounding mode */
  {
    ftype zt;
    int ze;
    char zs;
    octa zf, o;
    zt = funpack(z, &zf, &ze, &zs);
    if (¬r) r = cur_round;
    switch (zt) {
    case nan: case inf: exceptions |= I_BIT; return z;
    case zro: default: return zero_octa;
    case num: if (funpack(fintegerize(z, r), &zf, &ze, &zs) ≡ zro) return zero_octa;
      if (ze ≤ 1076) o = shift_right(zf, 1076 − ze, 1);
      else {
        if (ze > 1085 ∨ (ze ≡ 1085 ∧ (zf.h > #400000 ∨
              (zf.h ≡ #400000 ∧ (zf.l ∨ zs ≠ '-'))))) exceptions |= W_BIT;
        if (ze ≥ 1140) return zero_octa;
        o = shift_left(zf, ze − 1076);
      }
      return (zs ≡ '-' ? ominus(zero_octa, o) : o);
    }
  }
```

89. Going the other way, we can specify not only a rounding mode but whether the given fixed point octabyte is signed or unsigned, and whether the result should be rounded to short precision.

```
⟨ Subroutines 5 ⟩ +≡
  octa floatit ARGS((octa, int, int, int));
  octa floatit(z, r, u, p)
      octa z;      /* octabyte to float */
      int r;       /* rounding mode */
      int u;       /* unsigned? */
      int p;       /* short precision? */
  {
    int e; char s;
    register int t;
    exceptions = 0;
    if (¬z.h ∧ ¬z.l) return zero_octa;
    if (¬r) r = cur_round;
    if (¬u ∧ (z.h & sign_bit)) s = '-', z = ominus(zero_octa, z); else s = '+';
    e = 1076;
    while (z.h < #400000) e−−, z = shift_left(z, 1);
    while (z.h ≥ #800000) {
      e++;
      t = z.l & 1;
      z = shift_right(z, 1, 1);
      z.l |= t;
    }
    if (p) ⟨ Convert to short float 90 ⟩;
    return fpack(z, e, s, r);
  }
```

90. ⟨ Convert to short float 90 ⟩ ≡

```
  {
    register int ex; register tetra t;
    t = sfpack(z, e, s, r);
    ex = exceptions;
    sfunpack(t, &z, &e, &s);
    exceptions = ex;
  }
```

This code is used in section 89.

91. The square root operation is more interesting.

```
⟨ Subroutines 5 ⟩ +≡
  octa froot ARGS((octa, int));
  octa froot(z, r)
      octa z;      /* the operand */
      int r;      /* the rounding mode */
  {
    ftype zt;
    int ze;
    char zs;
    octa x, xf, rf, zf;
    register int xe, k;
    if (¬r) r = cur_round;
    zt = funpack(z, &zf, &ze, &zs);
    if (zs ≡ '-' ∧ zt ≠ zro) exceptions |= I_BIT, x = standard_NaN;
    else switch (zt) {
      case nan: if (¬(z.h & #80000)) exceptions |= I_BIT, z.h |= #80000;
        return z;
      default: case inf: case zro: x = z; break;
      case num: ⟨ Take the square root and return 92 ⟩;
      }
    if (zs ≡ '-') x.h |= sign_bit;
    return x;
  }
```

92. The square root can be found by an adaptation of the old pencil-and-paper method. If $n = \lfloor\sqrt{s}\rfloor$, where s is an integer, we have $s = n^2 + r$ where $0 \le r \le 2n$; this invariant can be maintained if we replace s by $4s + (0, 1, 2, 3)$ and n by $2n + (0, 1)$. The following code implements this idea with $2n$ in xf and r in rf. (It could easily be made to run about twice as fast.)

```
⟨ Take the square root and return 92 ⟩ ≡
  xf.h = 0, xf.l = 2;
  xe = (ze + #3fe) ≫ 1;
  if (ze & 1) zf = shift_left(zf, 1);
  rf.h = 0, rf.l = (zf.h ≫ 22) − 1;
  for (k = 53; k; k−−) {
    rf = shift_left(rf, 2); xf = shift_left(xf, 1);
    if (k ≥ 43) rf = incr(rf, (zf.h ≫ (2 ∗ (k − 43))) & 3);
    else if (k ≥ 27) rf = incr(rf, (zf.l ≫ (2 ∗ (k − 27))) & 3);
    if ((rf.l > xf.l ∧ rf.h ≥ xf.h) ∨ rf.h > xf.h) {
      xf.l++; rf = ominus(rf, xf); xf.l++;
    }
  }
  if (rf.h ∨ rf.l) xf.l++;      /* sticky bit */
  return fpack(xf, xe, '+', r);
```

This code is used in section 91.

93. And finally, the genuine floating point remainder. Subroutine *fremstep* either calculates $y \bmod z$ or reduces y to a smaller number having the same remainder with respect to z. In the latter case the `E_BIT` is set in *exceptions*. A third parameter, *delta*, gives a decrease in exponent that is acceptable for incomplete results; if *delta* is sufficiently large, say 2500, the correct result will always be obtained in one step of *fremstep*.

```
⟨ Subroutines 5 ⟩ +≡
  octa fremstep ARGS((octa, octa, int));
  octa fremstep(y, z, delta)
      octa y, z;
      int delta;
  {
    ftype yt, zt;
    int ye, ze;
    char xs, ys, zs;
    octa x, xf, yf, zf;
    register int xe, thresh, odd;
    yt = funpack(y, &yf, &ye, &ys);
    zt = funpack(z, &zf, &ze, &zs);
    switch (4 * yt + zt) {
    ⟨ The usual NaN cases 42 ⟩;
    case 4 * zro + zro: case 4 * num + zro: case 4 * inf + zro: case 4 * inf + num: case 4 * inf + inf:
      x = standard_NaN;
      exceptions |= I_BIT; break;
    case 4 * zro + num: case 4 * zro + inf: case 4 * num + inf: return y;
    case 4 * num + num: ⟨ Remainderize nonzero numbers and return 94 ⟩;
    default: zero_out: x = zero_octa;
    }
    if (ys ≡ '-') x.h |= sign_bit;
    return x;
  }
```

94. If there's a huge difference in exponents and the remainder is nonzero, this computation will take a long time. One could compute $(2^n y) \bmod z$ much more quickly for large n by using $O(\log n)$ multiplications modulo z, but the floating remainder operation isn't important enough to justify such expensive hardware.

Results of floating remainder are always exact, so the rounding mode is immaterial.

```
⟨ Remainderize nonzero numbers and return 94 ⟩ ≡
  odd = 0;      /* will be 1 if we've subtracted an odd multiple of z from y */
  thresh = ye − delta;
  if (thresh < ze) thresh = ze;
  while (ye ≥ thresh) ⟨ Reduce (ye, yf) by a multiple of zf; goto zero_out if the remainder is zero, goto
        try_complement if appropriate 95 ⟩;
  if (ye ≥ ze) {
    exceptions |= E_BIT; return fpack(yf, ye, ys, ROUND_OFF);
  }
  if (ye < ze − 1) return fpack(yf, ye, ys, ROUND_OFF);
  yf = shift_right(yf, 1, 1);
try_complement: xf = ominus(zf, yf), xe = ze, xs = '+' + '-' − ys;
  if (xf.h > yf.h ∨ (xf.h ≡ yf.h ∧ (xf.l > yf.l ∨ (xf.l ≡ yf.l ∧ ¬odd)))) xf = yf, xs = ys;
  while (xf.h < #400000) xe−−, xf = shift_left(xf, 1);
  return fpack(xf, xe, xs, ROUND_OFF);
```

This code is used in section 93.

95. Here we are careful not to change the sign of y, because a remainder of 0 is supposed to inherit the original sign of y.

```
⟨ Reduce (ye, yf) by a multiple of zf; goto zero_out if the remainder is zero, goto try_complement if
      appropriate 95 ⟩ ≡
  {
    if (yf.h ≡ zf.h ∧ yf.l ≡ zf.l) goto zero_out;
    if (yf.h < zf.h ∨ (yf.h ≡ zf.h ∧ yf.l < zf.l)) {
      if (ye ≡ ze) goto try_complement;
      ye−−, yf = shift_left(yf, 1);
    }
    yf = ominus(yf, zf);
    if (ye ≡ ze) odd = 1;
    while (yf.h < #400000) ye−−, yf = shift_left(yf, 1);
  }
```

This code is used in section 94.

96. Index.

⟨ Unsubnormalize y and z, if they are subnormal 52 ⟩ Used in section 51.

May 1, 2025 at 02:23

MMIX-ARITH