1. Introduction. The subroutines in this module are based upon the 32-bit versions in the `MMIX-ARITH` program supplied by professor Knuth. Since almost all systems today support 64-bit arithmetic, the author decided to include a 64-bit version of the original program. This program contains all the subroutines supplied by the original 32-bit program. The original *byte_diff* and *wyde_diff* are maintained although they are 32-bit routines. New 64-bit versions have been added that can be used instead. Subroutines like *oplus* are here for compatibility although they are supported directly by the C language.

⟨ Routines and includes 3 ⟩

2. ⟨ mmix-arith64.h 2 ⟩ ≡

```
#include <stdint.h>
  typedef uint64_t octa;
  typedef uint32_t tetra;
  typedef uint16_t wyde;
  typedef enum {
    false, true
  } bool;
  typedef enum {
    zro, num, inf, nan
  } ftype;
  octa oplus(octa, octa);
  octa ominus(octa, octa);
  octa incr(octa, int);
  octa shift_left(octa, int);
  octa shift_right(octa, int, bool);
  octa omult(octa, octa);
  octa signed_omult(octa, octa);
  octa odiv(octa, octa, octa);
  octa signed_odiv(octa, octa);
  octa oand(octa, octa);
  octa oandn(octa, octa);
  octa oxor(octa, octa);
  octa count_bits(octa);
  tetra byte_diff(tetra, tetra);
  tetra wyde_diff(tetra, tetra);
  octa octa_bdif(octa, octa);
  octa octa_wdif(octa, octa);
  octa octa_tdif(octa, octa);
  octa bool_mult(octa, octa, bool);
  octa fpack(octa, int, char, int);
  tetra sfpack(octa, int, char, int);
  octa load_sf(tetra);
  tetra store_sf(octa);
  ftype funpack(octa, octa *, int *, char *);
  ftype sfunpack(tetra, octa *, int *, char *);
  octa fmult(octa, octa);
  octa fdivide(octa, octa);
  octa fplus(octa, octa);
  int fepscomp(octa, octa, octa, int);
  int scan_const(char *);
  void print_float(octa);
  int fcomp(octa, octa);
  octa fintegerize(octa, int);
  octa fixit(octa, int);
  octa floatit(octa, int, int, int);
  octa froot(octa, int);
  octa fremstep(octa, octa, int);
  extern octa aux;
```

3. ⟨ Routines and includes 3 ⟩ ≡

```
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include "mmix-arith64.h"
```

⟨ Type definitions 53 ⟩
⟨ Global variables 4 ⟩
⟨ Subroutines 6 ⟩

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
octa zero_octa = 0;
octa neg_one = -1;
octa inf_octa = #7ff0000000000000;
octa standard_NaN = #7ff8000000000000;
octa aux;
octa overflow;
```

See also sections 29, 31, 70, and 77.

This code is used in section 3.

5. Arithmetic. We want to maintain the names of the routines of the (MMIXware) package although this is not strictly necessary.

6. ⟨ Subroutines 6 ⟩ ≡

```
octa oplus(octa y, octa z)
{
  return y + z;
}
octa ominus(octa y, octa z)
{
  return y − z;
}
```

See also sections 7, 8, 9, 10, 11, 22, 23, 24, 25, 26, 27, 28, 30, 33, 35, 36, 37, 38, 39, 42, 44, 48, 54, 55, 56, 57, 59, 69, 85, 86, 88, 89, 91, and 93.

This code is used in section 3.

7. In the following subroutine, *delta* is a signed number. We cannot use **tetra** as a type therefore.

⟨ Subroutines 6 ⟩ +≡

```
octa incr(octa y, int delta)
{
  return y + delta;
}
```

8. The shift operations are also easy to implement.

⟨ Subroutines 6 ⟩ +≡

```
octa shift_left(octa y, int s)
{
  return y ≪ s;
}
octa shift_right(octa y, int s, bool u)
{
  int64_t z = y;
  return (u ? z ≫ s : y ≫ s);
}
```

9. Multiplication. The advantage of having 64-bit arithmetic is that the multiplication is easier to implement than with 32-bit arithmetic. But to give a precise 128-bit result we still need some masking and shifting.

```
#define sign_bit ((octa)#8000000000000000)
⟨ Subroutines 6 ⟩ +≡
  octa omult(octa y, octa z)
  {
    octa yl, yh, zl, zh;
    octa x1, x2, x3, x4, acc;
    yl = y & #ffffffff;
    yh = y ≫ 32;
    zl = z & #ffffffff;
    zh = z ≫ 32;
    x1 = yl * zl;
    x2 = yl * zh;
    x3 = yh * zl;
    x4 = yh * zh;
    acc = x2 + x3 + (x1 ≫ 32);
    aux = x4 + (acc ≫ 32);
    acc = (x1 & #ffffffff) + (acc ≪ 32);
    return acc;
  }
```

10. 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 6 ⟩ +≡
  octa signed_omult(octa y, octa z)
  {
    octa acc;
    acc = omult(y, z);
    if (y & sign_bit) aux = aux − z;
    if (z & sign_bit) aux = aux − y;
    overflow = (aux ⊕ (aux ≫ 1) ⊕ (acc & sign_bit));
    return acc;
  }
```

11. Division. Long division of an unsigned 128-bit integer by an unsigned 64-bit integer is a challenging routine. Although many compilers offer 128-bit variables, we use prof. Knuth's method to make sure that the code is portable to all compilers. The quotient q is returned by the subroutine; the remainder r is stored in aux.

```
⟨ Subroutines 6 ⟩ +≡
  octa odiv(octa x, octa y, octa z) { int64_t k;
      register int i, j, n, d;
      octa u[4], v[2], q[2], mask, qhat, rhat, vh, vmh;
      octa t;
      octa acc = 0;
      ⟨ Check that x < z; otherwise give the trivial answer 12 ⟩
      ⟨ Unpack the dividend and divisor to u and v 13 ⟩
      ⟨ Determine the number of significant places n in the divisor v 14 ⟩
      ⟨ Normalise the divisor 15 ⟩
      for (j = 1; j ≥ 0; j−−) ⟨ Determine the quotient digit q[j] 18 ⟩⟨ Unnormalize the remainder 16 ⟩
      ⟨ Pack q and u to acc and aux 17 ⟩return acc; }
```

12.
```
⟨ Check that x < z; otherwise give the trivial answer 12 ⟩ ≡
  if (x ≥ z) {
    aux = y;
    return x;
  }
```
This code is used in section 11.

13. We need half the number of variables compared to the 32-bit version, but the amount of significant bits per variable is doubled.

```
⟨ Unpack the dividend and divisor to u and v 13 ⟩ ≡
  u[3] = x ≫ 32; u[2] = x & #ffffffff; u[1] = y ≫ 32; u[0] = y & #ffffffff;
  v[1] = z ≫ 32; v[0] = z & #ffffffff;
```
This code is used in section 11.

14.
```
⟨ Determine the number of significant places n in the divisor v 14 ⟩ ≡
  n = v[1] ? 2 : 1;
```
This code is used in section 11.

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

```
⟨ Normalise the divisor 15 ⟩ ≡
  vh = v[n − 1];
  for (d = 0; vh < #80000000; d++, vh ≪= 1) ;
  for (j = k = 0; j < n + 2; j++) {
    t = (u[j] ≪ d) + k;
    u[j] = t & #ffffffff, k = t ≫ 32;
  }
  for (j = k = 0; j < n; j++) {
    t = (v[j] ≪ d) + k;
    v[j] = t & #ffffffff, k = t ≫ 32;
  }
  vh = v[n − 1];
  vmh = n − 1;
```
This code is used in section 11.

16. ⟨ Unnormalize the remainder 16 ⟩ ≡
```
mask = (1 ≪ d) − 1;
for (j = 3; j ≥ n; j−−) u[j] = 0;
for (k = 0; j ≥ 0; j−−) {
  t = (k ≪ 32) + u[j];
  u[j] = t ≫ d, k = t & mask;
}
```
This code is used in section 11.

17. ⟨ Pack q and u to acc and aux 17 ⟩ ≡
```
acc = (q[1] ≪ 32) + q[0];
aux = (u[1] ≪ 32) + u[0];
```
This code is used in section 11.

18. ⟨ Determine the quotient digit $q[j]$ 18 ⟩ ≡
```
{
  ⟨ Find the trial quotient, q̂ 19 ⟩
  ⟨ Subtract b^j q̂v from u 20 ⟩
  ⟨ If the result was negative, decrease q̂ by 1 21 ⟩
  q[j] = qhat;
}
```
This code is used in section 11.

19. ⟨ Find the trial quotient, $\hat{q}$ 19 ⟩ ≡
```
t = (u[j + n] ≪ 32) + u[j + n − 1];
qhat = t/vh, rhat = t − vh ∗ qhat;
while (qhat ≡ #100000000 ∨ qhat ∗ vmh > (rhat ≪ 16) + u[j + n − 2]) {
  qhat−−, rhat += vh;
  if (rhat ≥ #100000000) break;
}
```
This code is used in section 18.

20. *Knuth:* "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 $b^j\hat{q}v$ from u 20 ⟩ ≡
```
for (i = k = 0; i < n; i++) {
  t = u[i + j] + #ffffffff00000000 − k − qhat ∗ v[i];
  u[i + j] = t & #ffffffff, k = #ffffffff − (t ≫ 32);
}
```
This code is used in section 18.

21. *Knuth:* "The correction here occurs only rarely, but it can be necessary—for example, when dividing the number #7fff800100000000 by #800080020005." This remains true in 64-bit logic, of course.

```
⟨ If the result was negative, decrease q̂ by 1 21 ⟩ ≡
  if (u[j + n] ≠ k) {
    qhat−−;
    for (i = k = 0; i < n; i++) {
      t = u[i + j] + v[i] + k;
      u[i + j] = t & #ffffffff, k = t ≫ 32;
    }
  }
```

This code is used in section 18.

22.

```
⟨ Subroutines 6 ⟩ +≡
  octa signed_odiv(octa y, octa z)
  {
    octa yy, zz, q;
    register int sy, sz;
    if (y & sign_bit) sy = 2, yy = −y;
    else sy = 0, yy = y;
    if (z & sign_bit) sz = 1, zz = −z;
    else sz = 0, zz = z;
    q = odiv(zero_octa, yy, zz);
    switch (sy + sz) {
    case 3: aux = zero_octa − aux;
      if (q & sign_bit) overflow = true;
    case 0: break;
    case 2:
      if (aux) aux = zz − aux, q = −1 − q;
      else q = −q;
      break;
    case 1:
      if (aux) aux = aux − zz, q = −1 − q;
      else q = −q;
      break;
    }
    return q;
  }
```

23. Bit fiddling. The bitwise operators are again easy to implement.

```
⟨ Subroutines 6 ⟩ +≡
  octa oand(octa y, octa z)
  {
    return y & z;
  }
  octa oandn(octa y, octa z)
  {
    return y & ∼z;
  }
  octa oxor(octa y, octa z)
  {
    return y ⊕ z;
  }
```

24. The method of counting the number of bits is also extended to 64-bit numbers. The same trick as in the MMIXware package is used.

Knuth: "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 6 ⟩ +≡
  octa count_bits(octa y)
  {
    register octa yy = y;
    yy = yy − ((yy ≫ 1) & #5555555555555555);
    yy = (yy & #3333333333333333) + ((yy ≫ 2) & #3333333333333333);
    yy = (yy + (yy ≫ 4)) & #0f0f0f0f0f0f0f0f;
    yy = (yy + (yy ≫ 8)) & #00ff00ff00ff00ff;
    yy = (yy + (yy ≫ 16)) & #0000ffff0000ffff;
    return (yy + (yy ≫ 32)) & #ffff;
  }
```

25. Computing of the nonnegative byte differences is not compatible with the MIXXware package since 64-bit arithmetic is used. This routine accepts and returns octas. For a 128-bit version the reader may want to use a widescreen monitor.

```
⟨ Subroutines 6 ⟩ +≡
  tetra byte_diff(tetra y, tetra 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);
  }
  octa octa_bdif(octa y, octa z)
  {
    register octa d = (y & #00ff00ff00ff00ff) + #0100010001000100 − (z & #00ff00ff00ff00ff);
    register octa m = d & #0100010001000100;
    register octa x = d & (m − (m ≫ 8));
    d = ((y ≫ 8) & #00ff00ff00ff00ff) + #0100010001000100 − ((z ≫ 8) & #00ff00ff00ff00ff);
    m = d & #0100010001000100;
    return x + ((d & (m − (m ≫ 8))) ≪ 8);
  }
```

26. Computing nonnegative wyde differences is not much different except for the width of the fields being compared.

```
⟨ Subroutines 6 ⟩ +≡
  tetra wyde_diff(tetra y, tetra 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))));
  }
  octa octa_wdif(octa y, octa z)
  {
    register octa d = (y & #0000ffff0000ffff) + #0001000000010000 − (z & #0000ffff0000ffff);
    register octa m = d & #0001000000010000;
    register octa x = d & (m − (m ≫ 16));
    d = ((y ≫ 16) & #0000ffff0000ffff) + #0001000000010000 − ((z ≫ 16) & #0000ffff0000ffff);
    m = d & #0001000000010000;
    return x + ((d & (m − (m ≫ 16))) ≪ 16);
  }
```

27. The same is true for tetra difference.

```
⟨ Subroutines 6 ⟩ +≡
  octa octa_tdif(octa y, octa z)
  {
    register octa d = (y & #ffffffff) + #100000000 − (z & #ffffffff);
    register octa m = d & #0001000000010000;
    register octa x = d & (m − (m ≫ 16));
    d = ((y ≫ 32) & #ffffffff) + #100000000 − ((z ≫ 16) & #ffffffff);
    m = d & #100000000;
    return x + ((d & (m − (m ≫ 32))) ≪ 32);
  }
```

28. The last bitwise subroutine requires a bit fiddling. This routine is used to implement the `MOR` and `MXOR` operations.

```
⟨ Subroutines 6 ⟩ +≡
  octa bool_mult(octa y, octa z, bool xor)
  {
    octa o, x;
    register octa a, c;
    register int k;
    for (k = 0, o = y, x = 0; o; k++, o = shift_right(o, 8, true)) {
      if (o & #ff) {
        a = ((z ≫ k) & #0101010101010101) ∗ #ff;
        c = (o & #ff) ∗ #0101010101010101;
        if (xor) x ⊕= a & c;
        else x |= a & c;
      }
    }
    return x;
  }
```

29. Floating point packing and unpacking. Standard IEEE floating binary numbers have sign, exponent and fraction encoded in a **tetra** or an **octa**. In this section we will find the routines to convert from packed format to unpacked format and vice versa.

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

⟨ Global variables 4 ⟩ +≡

```
  int cur_round;
```

30. An octabyte m, an exponent e and a sign s are packed into a floating binary number by the *fpack* routine. The result is a number $\pm 2^{e-1076} f$, using a given rounding mode. The value of f should satisfy $2^{54} \le f \le 2^{55}$. Careful readers of the `MMIXware` book will notice that the author has used the variable f instead of o and the variable m for mantisse instead of f.

Knuth: "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 */
#define zero_exponent  (−1000)
⟨ Subroutines 6 ⟩ +≡
  octa fpack(octa m, int e, char s, int r)
  {
    octa f;
    if (e > #7fd) e = #7ff, f = 0;
    else {
      if (e < 0) {
        if (e < −54) f = 1;
        else {
          octa ff;
          f = shift_right(m, −e, true);
          ff = shift_left(f, −e);
          if (ff ≠ m) {
            f |= 1;
          }
        }
        e = 0;
      }
      else f = m;
    }
    ⟨ Round and return the result 32 ⟩;
  }
```

31. ⟨ Global variables 4 ⟩ +≡

```
  int exceptions;
```

32. ⟨ Round and return the result 32 ⟩ ≡

```
if (f & 3) exceptions |= X_BIT;
switch (r) {
case ROUND_DOWN: if (s ≡ '-') f += 3; break;
case ROUND_UP: if (s ≠ '-') f += 3;
case ROUND_OFF: break;
case ROUND_NEAR: f += f & 4 ? 2 : 1; break;
}
f = shift_right(f, 2, 1);
f += (octa) e ≪ 52;
if (f ≥ #7ff0000000000000) exceptions |= O_BIT + X_BIT;
else if (f < #100000) exceptions |= U_BIT;
if (s ≡ '-') f |= #8000000000000000;
return f;
```

This code is used in section 30.

33. ⟨ Subroutines 6 ⟩ +≡

```
tetra sfpack(octa m, int e, char s, int r)
{
  register tetra f;
  if (e > #47d) e = #47f, f = 0;
  else {
    f = m ≫ 29;
    if (m & #1fffffff) f |= 1;
    if (e < #380) {
      if (e < #380 − 25) f = 1;
      else {
        register tetra f0, ff;
        f0 = f;
        f = f ≫ (#380 − e);
        ff = f ≪ (#380 − e);
        if (ff ≠ f0) f |= 1;
      }
      e = #380;
    }
  }
  ⟨ Round and return the short result 34 ⟩;
}
```

34. ⟨ Round and return the short result 34 ⟩ ≡

```
if (f & 3) exceptions |= X_BIT;
switch (r) {
case ROUND_DOWN: if (s ≡ '-') f += 3; break;
case ROUND_UP: if (s ≠ '-') f += 3;
case ROUND_OFF: break;
case ROUND_NEAR: f += f & 4 ? 2 : 1; break;
}
f ≫= 2;
f += (e − #380) ≪ 23;
if (f ≥ #7f800000) exceptions |= O_BIT + X_BIT;
else if (f < #100000) exceptions |= U_BIT;
if (s ≡ '-') f |= #80000000;
return f;
```

This code is used in section 33.

35. The funpack routine is the opposite of *fpack*. It splits a floating point number in the fraction f, the exponent e and the sign s. It returns the type found: *zro*, *num*, *inf* or *nan*.

⟨ Subroutines 6 ⟩ +≡

```
ftype funpack(octa x, octa *f, int *e, char *s)
{
  register int ee;
  *s = x & sign_bit ? '-' : '+';
  *f = (x ≪ 2) & #3fffffffffffff;
  ee = (x ≫ 52) & #7ff;
  if (ee) {
    *e = ee − 1;
    *f |= #40000000000000;
    return (ee < #7ff ? num : *f ≡ #40000000000000 ? inf : nan);
  }
  if (¬(x & #ffffffff) ∧ ¬*f) {
    *e = zero_exponent;
    return zro;
  }
  do {
    e−−;
    *f ≪= 1;
  } while (¬(*f & #40000000000000));
  *e = ee;
  return num;
}
```

36. ⟨ Subroutines 6 ⟩ +≡

```
ftype sfunpack(tetra x, octa *f, int *e, char *s)
{
  register int ee;
  *s = x & sign_bit ? '-' : '+';
  *f = (x >> 1) & #3fffff80000000;
  ee = (x >> 23) & #ff;
  if (ee) {
    *e = ee + #380 - 1;
    *f |= #40000000000000;
    return (ee < #ff ? num : (x & #7fffffff) == #7f800000 ? inf : nan);
  }
  if (¬(x & #7fffffff)) {
    *e = zero_exponent;
    return zro;
  }
  do {
    e--;
    *f <<= 1;
  } while (¬(*f & #40000000000000));
  *e = ee;
  return num;
}
```

37. ⟨ Subroutines 6 ⟩ +≡

```
octa load_sf(tetra z)
{
  octa f, x;
  int e;
  char s;
  ftype t;
  t = sfunpack(z, &f, &e, &s);
  switch (t) {
  case zro: x = 0;
    break;
  case num: return fpack(f, e, s, ROUND_OFF);
  case inf: x = inf_octa;
    break;
  case nan: x = shift_right(f, 2, true);
    x |= #7ff0000000000000;
    break;
  }
  if (s == '-') x |= sign_bit;
  return x;
}
```

38. ⟨ Subroutines 6 ⟩ +≡

```
tetra store_sf(octa x)
{
  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 & #200000)) {
      f |= #20000000000000;
      exceptions |= I_BIT;
    }
    z = #7f8 | (f ≫ 31);
    break;
  }
  if (s ≡ '-') z |= #80000000;
  return z;
}
```

39. Floating multiplication and division. The difference between the 32-bit version and the 64-bit version is not so great. Floating point multiplication mostly depends on calling other routines.

```
⟨ Subroutines 6 ⟩ +≡
  octa fmult(octa y, octa 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 - '+';
    switch (4 * yt + zt) {
      ⟨ The usual NaN cases 40 ⟩
    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 * num + num: ⟨ Multiply nonzero numbers and return 41 ⟩
    }
    if (xs ≡ '-') x |= sign_bit;
    return x;
  }
```

40. ⟨ The usual NaN cases 40 ⟩ ≡

```
case 4 * nan + nan:
  if (¬(y & #80000000000000)) exceptions |= I_BIT;
case 4 * zro + nan: case 4 * num + nan: case 4 * inf + nan:
  if (¬(z & #80000000000000)) exceptions |= I_BIT;
  y |= #80000000000000;
  return z;
case 4 * nan + zro: case 4 * nan + num: case 4 * nan + inf:
  if (¬(y & #80000000000000)) exceptions |= I_BIT;
  y |= #80000000000000;
  return y;
```

This code is used in sections 39, 42, 44, and 93.

41. ⟨ Multiply nonzero numbers and return 41 ⟩ ≡

```
  xe = ye + ze - #3fd;
  x = omult(yf, zf ≪ 9);
  if (aux ≥ #40000000000000) xf = aux;
  else xf = aux ≪ 1, xe--;
  if (x) xf |= 1;
  return fpack(xf, xe, xs, cur_round);
```

This code is used in section 39.

42. ⟨ Subroutines 6 ⟩ +≡

```
octa fdivide(octa y, octa 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 - '+';
  switch (4 * yt + zt) {
    ⟨ The usual NaN cases 40 ⟩
  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;
  case 4 * zro + zro: case 4 * inf + inf: x = standard_NaN;
    exceptions |= I_BIT;
    break;
  case 4 * num + num: ⟨ Divide nonzero numbers and return 43 ⟩
  }
  if (xs == '-') x |= sign_bit;
  return x;
}
```

43. ⟨ Divide nonzero numbers and return 43 ⟩ ≡

```
xe = ye - ze + #3fd;
xf = odiv(yf, zero_octa, zf ≪ 9);
if (xf ≥ #8000000000000000) {
  aux |= xf & 1;
  xf = shift_right(xf, 1, 1);
  xe++;
}
if (aux) xf |= 1;
return fpack(xf, xe, xs, cur_round);
```

This code is used in section 42.

44. Floating point addition is still not difficult, even with 64-bit arithmetic. All cases still have to be handled carefully.

```
⟨ Subroutines 6 ⟩ +≡
  octa fplus(octa y, octa 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);
    xs = ys + zs - '+';
    switch (4 * yt + zt) {
      ⟨ The usual NaN cases 40 ⟩
    case 4 * zro + num: return fpack(zf, ze, zs, ROUND_OFF);
      break;
    case 4 * num + zro: return fpack(yf, ye, ys, ROUND_OFF);
      break;
    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 ≠ (z ⊕ sign_bit)) ⟨ Add nonzero numbers and return 45 ⟩
    case 4 * zro + zro: x = zero_octa;
      xs = (ys ≡ zs ? ys : cur_round ≡ ROUND_DOWN ? '-' : '+');
      break;
    }
    if (xs ≡ '-') x |= sign_bit;
    return x;
  }
```

45. ⟨ Add nonzero numbers and return 45 ⟩ ≡

```
{
  octa f, ff;
  if (ye < ze ∨ (ye ≡ ze ∧ (yf < zf))) ⟨ Exchange y with z 46 ⟩
  d = ye − ze;
  xs = ys, xe = ye;
  if (d) ⟨ Adjust for difference in exponents 47 ⟩
  if (ys ≡ zs) {
    xf = yf + zf;
    if (xf ≥ #80000000000000) xe++, xf = shift_right(xf, 1, 1);
  }
  else {
    xf = yf − zf;
    if (xf ≥ #80000000000000) xe++, d = xf & 1, xf = shift_right(xf, 1, 1), xf |= d;
    else
      while (xf < #40000000000000) xe−−, xf = xf ≪ 1;
  }
  return fpack(xf, xe, xs, cur_round);
}
```

This code is used in section 44.

46. ⟨ Exchange y with z 46 ⟩ ≡

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

This code is used in sections 45 and 49.

47. *Knuth:* "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 47 ⟩ ≡

```
{
  if (d ≤ 2) zf = shift_right(zf, d, 1);
  else if (d > 54) zf = 1;
  else {
    if (ys ≠ zs) d−−, xe−−, yf = yf ≪ 1;
    f = zf;
    zf = shift_right(f, d, 1);
    ff = zf ≪ d;
    if (ff ≠ f) zf |= 1;
  }
}
```

This code is used in section 45.

48. *Knuth:* "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 6 ⟩ +≡
  int fepscomp(octa y, octa z, octa e, int s)
  {
    octa yf, zf, ef, f, ff;
    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 49 ⟩
  }
```

49. ⟨ Compare two numbers with respect to epsilon and **return** 49 ⟩ ≡

```
  ⟨ Undenormalize y and z, if they are denormal 50 ⟩
  if (ye < ze ∨ (ye ≡ ze ∧ (yf < zf))) ⟨ Exchange y with z 46 ⟩
  if (ze ≡ zero_exponent) ze = ye;
  d = ye − ze;
  if (¬s) ee −= d;
  if (ee ≥ 1023) return 1;
  ⟨ Compute the difference of fraction parts, f 51 ⟩
  if (f) return 1;
  if (e < 968) return 0;
  if (ee ≥ 1021) ef ≪= ee − 1021;
  else ef ≫= 1021 − ee;
  return (f < ef);
```

This code is used in section 48.

50. ⟨ Undenormalize y and z, if they are denormal 50 ⟩ ≡

```
if (ye < 0) yf = y ≪ 2, ye = 0;
if (ze < 0) zf = z ≪ 2, ze = 0;
```

This code is used in section 49.

51. ⟨ Compute the difference of fraction parts, f 51 ⟩ ≡

```
if (d > 54) f = 0, ff = zf;
else f = zf ≫ d, ff = f ≪ d;
if (ff ≠ zf) {
  if (ee < 1020) return 0;
  f += ys ≡ zs ? −1 : 1;
}
f = (ys ≡ zs ? yf − f : yf + f);
```

This code is used in section 49.

52. Supporting routines.

53. *Bignum* is converted to 64-bit integers also. We need just as many bits, but less integers.

```
⟨ Type definitions 53 ⟩ ≡
#define bignum_prec 79
  typedef struct {
    int a, b;
    octa dat[bignum_prec];
  } bignum;
```

This code is used in section 3.

54.
```
⟨ Subroutines 6 ⟩ +≡
  static void bignum_times_ten(bignum *f)
  {    /* radix = 2^56 */
    register octa *p, *q;
    register octa x, carry;
    for (p = &f→dat[f→b], q = &f→dat[f→a], carry = 0; p ≥ q; p−−) {
      x = *p * 10 + carry;
      *p = x & #ffffffffffffff;
      carry = x ≫ 56;
    }
    *p = carry;
    if (carry) f→a−−;
    if (f→dat[f→b] ≡ 0 ∧ f→b > f→a) f→b−−;
  }
```

55.
```
⟨ Subroutines 6 ⟩ +≡
  static void bignum_double(bignum *f)
  {    /* radix = 10^18 */
    register octa *p, *q;
    register octa x, carry;
    for (p = &f→dat[f→b], q = &f→dat[f→a], carry = 0; p ≥ q; p−−) {
      x = *p + *p + carry;
      if (x ≥ 1000000000000000000) carry = 1, *p = x − 1000000000000000000;
      else carry = 0, *p = x;
    }
    *p = carry;
    if (carry) f→a−−;
    if (f→dat[f→b] ≡ 0 ∧ f→b > f→a) f→b−−;
  }
```

56. ⟨ Subroutines 6 ⟩ +≡

```
static int bignum_compare(bignum *f, bignum *g)
{
  register octa *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;
}
```

57. ⟨ Subroutines 6 ⟩ +≡

```
static void bignum_dec(bignum *f, bignum *g, octa r)
{
  register octa *p, *q, *pp, *qq;
  register long long 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)−−;
    else *p = r;
  while (f→dat[f→a] ≡ 0) {
    if (f→a ≡ f→b) {
      f→a = f→b = bignum_prec − 1, f→dat[bignum_prec − 1] = 0;
      return;
    }
    f→a++;
  }
  while (f→dat[f→b] ≡ 0) f→b−−;
}
```

58. Floating point output conversion.

59. ⟨ Subroutines 6 ⟩ +≡

```
void print_float(octa x)
{
  ⟨ Local variables for print_float 61 ⟩
  if (x & sign_bit) printf("-");
  ⟨ Extract the exponent e and determine the fraction interval [f..g] or (f..g) 60 ⟩
  ⟨ Store f and g as multiprecise integers 63 ⟩
  ⟨ Compute the significant digits s and decimal exponent e 64 ⟩
  ⟨ Print the significant digis with proper context 67 ⟩
}
```

60. ⟨ Extract the exponent e and determine the fraction interval [f..g] or (f..g) 60 ⟩ ≡

```
f = x ≪ 1;
e = f ≫ 53;
f &= #1fffffffffffff;
if (¬f) ⟨ Handle the special case when the fraction part is zero 62 ⟩
else {
  g = f + 1;
  f −= 1;
  if (¬e) e = 1;
  else if (e ≡ #7ff) {
    printf("NaN");
    if ((g ≡ #10000000000000) ∨ (g ≡ 1)) return;
    e = #3ff;
  }
  else f |= #20000000000000, g |= #20000000000000;
}
```

This code is used in section 59.

61. ⟨ Local variables for print_float 61 ⟩ ≡

```
octa f, g;
register int e;
register int j, k;
```

See also section 66.

This code is used in section 59.

62. ⟨ Handle the special case when the fraction part is zero 62 ⟩ ≡

```
{
  if (¬e) {
    printf("0.");
    return;
  }
  if (e ≡ #7ff) {
    printf("Inf");
    return;
  }
  e−−;
  f = #3fffffffffffff;
  g = #40000000000002;
}
```

This code is used in section 60.

63.

```
#define magic_offset 2084
#define origin 18
```

⟨ Store f and g as multiprecise integers 63 ⟩ ≡

```
k = (magic_offset − e)/56;
ff.dat[k] = f ≫ (magic_offset − e − 56 ∗ k) & #fffffffffffff;
gg.dat[k] = g ≫ (magic_offset − e − 56 ∗ k) & #fffffffffffff;
ff.dat[k + 1] = f ≪ (56 + e + 56 ∗ k − magic_offset) & #fffffffffffff;
gg.dat[k + 1] = g ≪ (56 + e + 56 ∗ k − magic_offset) & #fffffffffffff;
ff.a = k;
ff.b = ff.dat[k + 1] ? k + 1 : k;
gg.a = k;
gg.b = gg.dat[k + 1] ? k + 1 : k;
```

This code is used in section 59.

64. ⟨ 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 (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';
}
∗p = '\0';
```

This code is used in section 59.

65. ⟨ Compute the significant digits in the large exponent case 65 ⟩ ≡

```
{
  register int open = x & 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, #100000000000000), bignum_dec(&gg, &tt, #100000000000000);
    if (bignum_compare(&gg, &tt) ≥ open) break;
    *p++ = j;
    if (ff.a ≡ bignum_prec − 1 ∧ ¬open) goto done;
  }
  for (k = j; bignum_compare(&gg, &tt) ≥ open; k++) bignum_dec(&gg, &tt, #100000000000000);
  *p++ = (j + 1 + k) ≫ 1;
done: ;
}
```

This code is used in section 64.

66. ⟨ Local variables for print_float 61 ⟩ +≡

```
bignum ff, gg;
bignum tt;
char s[18];
register char *p;
```

67. ⟨ Print the significant digis 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 (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 59.

68. Floating point input conversion.

69. ⟨ Subroutines 6 ⟩ +≡

```
int scan_const(char *s)
{
  ⟨ Local variables for scan_const 71 ⟩
  val = 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 75 ⟩
  else if (nan) ⟨ Set standard NaN 72 ⟩
  else if (strncmp(p, "Inf", 3) ≡ 0) ⟨ Set infinity 73 ⟩
  else {
    next_char = s;
    return −1;
  }
  ⟨ Pack and round the answer 74 ⟩
  return 1;
}
```

70. ⟨ Global variables 4 ⟩ +≡

```
octa val;
char *next_char;
```

71. ⟨ Local variables for scan_const 71 ⟩ ≡

```
register char *p, *q;
register bool nan;
int sign;
```

See also sections 78 and 82.

This code is used in section 69.

72. ⟨ Set standard NaN 72 ⟩ ≡

```
{
  next_char = p;
  val = #60000000000000;
  exp = #3fe;
}
```

This code is used in section 69.

73. ⟨ Set infinity 73 ⟩ ≡

```
{
  next_char = p + 3;
  exp = 99999;
}
```

This code is used in section 69.

74. ⟨ Pack and round the answer 74 ⟩ ≡

```
val = fpack(val, exp, sign, ROUND_NEAR);
if (nan) {
  if ((val & #7fffffff00000000) ≡ #4000000000000000) val |= #7fffffffffffffff;
  else if ((val & #7fffffffffffffff) ≡ #3ff0000000000000) val |= #4000000000000001;
  else val |= #4000000000000000;
}
```

This code is used in section 69.

75. ⟨ Scan a number 75 ⟩ ≡

```
{
  for (q = buf0, dec_pt = (char *) 0; isdigit(*p); p++) {
    val = val * 10 + *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 76 ⟩
  next_char = p;
  if (*p ≡ 'e' ∨ *p ≡ 'E') ⟨ Scan an exponent 79 ⟩
  else exp = 0;
  if (dec_pt) ⟨ Build a floating point constant 80 ⟩
  else {
    if (sign ≡ '-') val = −val;
    return 0;
  }
}
```

This code is used in section 69.

76. ⟨ Scan a fraction part 76 ⟩ ≡

```
{
  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 75.

77.

```
#define buf0 (buf + 17)
#define buf_max (buf + 786)
```

⟨ Global variables 4 ⟩ +≡

```
static char buf[794] = "00000000000000000";
```

78. ⟨ Local variables for scan_const 71 ⟩ +≡

```
register char *dec_pt;
register int exp;
register int zeros;
```

79. ⟨ Scan an exponent 79 ⟩ ≡

```
{
  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 < 1000) exp = exp * 10 + *p - '0';
    if (¬dec_pt) dec_pt = q, zeros = 0;
    if (exp_sign ≡ '-') exp = -exp;
    next_char = p;
  }
}
```

This code is used in section 75.

80. ⟨ Build a floating point constant 80 ⟩ ≡

```
{
  x = buf + 359 + zeros - dec_pt - exp;
  if (q ≡ buf0 ∨ x ≥ 1413) {
    exp = -99999;
  }
  else if (x < 0) {
    exp = 99999;
  }
  else {
    ff.a = x/18;
    for (p = q; p < q + 17; p++) *p = '0';
    q = q - 1 - (q + 359 + zeros - dec_pt - exp) % 18;
    for (p = buf0 - x % 18, k = ff.a; p ≤ q ∧ k ≤ 78; p += 18, k++)
      ⟨ Put the 18-digit number *p...*(p+17) into ff.dat[k] 81 ⟩
    ff.b = k - 1;
    for (x = 0; p ≤ q; p += 18)
      if (strncmp(p, "000000000000000000", 17) ≠ 0) x = 1;
    ff.dat[78] += x;
    while (ff.dat[ff.b] ≡ 0) ff.b--;
    ⟨ Determine the binary fraction and binary exponent 83 ⟩
  }
}
```

This code is used in section 75.

81. ⟨ Put the 18-digit number *p...*(p+17) into ff.dat[k] 81 ⟩ ≡

```
{
  for (x = *p - '0', pp = p + 1; pp < p + 18; pp++) x = x * 10 + *pp - '0';
  ff.dat[k] = x;
}
```

This code is used in section 80.

82. ⟨ Local variables for scan_const 71 ⟩ +≡
```
register long long int k, x;
register char *pp;
bignum ff, tt;
```

83. ⟨ Determine the binary fraction and binary exponent 83 ⟩ ≡
```
val = 0;
if (ff.a > 18) {
  for (exp = #3fe; ff.a > 18; exp--) bignum_double(&ff);
  for (k = 54; k; k--) {
    if (ff.dat[18]) {
      val |= (unsigned long long int) 1 ≪ k;
      ff.dat[36] = 0;
      if (ff.b ≡ 18) break;
    }
    bignum_double(&ff);
  }
}
else {
  tt.a = tt.b = 18; tt.dat[18] = 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) {
      val |= (unsigned long long int) 1 ≪ k;
      bignum_dec(&ff, &tt, 1000000000000000000);
      if (ff.a ≡ bignum_prec − 1) break;
    }
  }
}
if (k ≡ 0) val |= 1;
```
This code is used in section 80.

84. Floating point remainders.

85. ⟨ Subroutines 6 ⟩ +≡

```
int fcomp(octa y, octa 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;
  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 > z) x = 1;
    else if (y < z) x = −1;
    else return 0;
    break;
  }
  return ys ≡ '-' ? −x : x;
}
```

86. A float is truncated to an integer using *fintegerize*. The trick is to shift the mantissa down and up again so that the fractional part falls off.

⟨ Subroutines 6 ⟩ +≡

```
octa fintegerize(octa z, int r)
{
  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 & #80000000000000)) {
      exceptions |= I_BIT;
      z |= #80000000000000;
    }
  case inf: case zro: return z;
  case num: ⟨ Integerize and return 87 ⟩
  }
}
```

87. ⟨ Integerize and return 87 ⟩ ≡

```
if (ze ≥ 1074) return fpack(zf, ze, zs, ROUND_OFF);
if (ze ≤ 1020) xf = 1;
else {
  octa ff;
  xf = zf ≫ (1074 − ze);
  ff = xf ≪ (1074 − ze);
  if (ff ≠ zf) xf |= 1;
}
switch (r) {
case ROUND_DOWN:
  if (zs ≡ '-') xf += 3;
  break;
case ROUND_UP:
  if (zs ≠ '-') xf += 3;
  break;
case ROUND_OFF: break;
case ROUND_NEAR: xf += xf & 4 ? 2 : 1;
  break;
}
xf &= #fffffffffffffffc;
if (ze ≥ 1022) return fpack(xf ≪ (1074 − ze), ze, zs, ROUND_OFF);
if (xf & #ffffffff) xf = #3ff0000000000000;
if (zs ≡ '-') xf |= sign_bit;
return xf;
```

This code is used in section 86.

88. ⟨ Subroutines 6 ⟩ +≡

```
octa fixit(octa z, int r)
{
  ftype zt;
  int ze;
  char zs;
  octa zf, f;
  zt = funpack(z, &zf, &ze, &zs);
  if (¬r) r = cur_round;
  switch (zt) {
  case nan: case inf: exceptions |= I_BIT;
    return z;
  case zro: return zero_octa;
  case num:
    if (funpack(fintegerize(z, r), &zf, &ze, &zs) ≡ zro) return zero_octa;
    if (ze ≤ 1076) f = shift_right(zf, 1076 − ze, 1);
    else {
      if (ze > 1085 ∨ (ze ≡ 1085 ∧ (zf > #400000 ∨ ((zf & #40000000000000) ≡
          #40000000000000 ∧ (zf & #ffffffff ∨ zs ≠ '-'))))) exceptions = W_BIT;
      if (ze ≥ 1140) return zero_octa;
      f = zf ≪ (ze − 1076);
    }
    return (zs ≡ '-' ? −f : f);
  }
}
```

89. ⟨ Subroutines 6 ⟩ +≡

```
octa floatit(octa z, int r, int u, int p)
{
  int e;
  char s;
  register int t;
  exceptions = 0;
  if (¬z) return 0;
  if (¬r) r = cur_round;
  if (¬u ∧ (z & sign_bit)) s = '-', z = −z;
  else s = '+';
  e = 1076;
  while (z < #40000000000000) e−−, z = z ≪ 1;
  while (z ≥ #80000000000000) {
    e++;
    t = z & 1;
    z = shift_right(z, 1, 1);
    z |= 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. ⟨ Subroutines 6 ⟩ +≡

```
octa froot(octa z, int r)
{
  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 & #80000000000000)) exceptions |= I_BIT, z |= #80000000000000;
      return z;
    case inf: case zro: x = z;
      break;
    case num: ⟨ Take the square root and return 92 ⟩
    }
  if (zs ≡ '-') x |= sign_bit;
  return x;
}
```

92. ⟨ Take the square root and return 92 ⟩ ≡

```
xf = 2;
xe = (ze + #3fe) ≫ 1;
if (ze & 1) zf = zf ≪ 1;
rf = (zf ≫ 54) − 1;
for (k = 53; k; k−−) {
  rf = rf ≪ 2; xf = xf ≪ 1;
  if (k ≥ 27) rf += (zf ≫ (2 ∗ (k − 27))) & 3;
  if (rf > xf) {
    xf ++;
    rf −= xf;
    xf ++;
  }
}
if (rf) xf ++;
return fpack(xf, xe, '+', r);
```

This code is used in section 91.

93. ⟨ Subroutines 6 ⟩ +≡

```
octa fremstep(octa y, octa 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 40 ⟩
  case 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 ⟩
  zero_out: x = zero_octa;
  }
  if (ys ≡ '-') x |= sign_bit;
  return x;
}
```

94. ⟨ Remainderize nonzero numbers and **return** 94 ⟩ ≡

```
odd = 0;
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 ((ys ≡ zs) ∧ (ye < ze - 1)) return fpack(yf, ye, ys, ROUND_OFF);
yf ≫= 1;
try_complement: xf = zf - yf, xe = ze, xs = '+' + '-' - ys;
if (xf > yf ∨ ((xf ≡ yf) ∧ ¬odd)) xf = yf, xs = ys;
while (xf < #40000000000000) xe--, xf ≪= 1;
return fpack(xf, xe, xs, ROUND_OFF);
```

This code is used in section 93.

95. ⟨ Reduce (ye,yf) by a multiple of zf; **goto** *zero_out* if the remainder is zero, **goto** *try_complement* if appropriate 95 ⟩ ≡

```
{
  if (yf ≡ zf) goto zero_out;
  if (yf < zf) {
    if (ye ≡ ze) goto try_complement;
    ye−−, yf ≪= 1;
  }
  yf = yf − zf;
  if (ye ≡ ze) odd = 1;
  while (yf < #40000000000000) ye−−, yf ≪= 1;
}
```

This code is used in section 94.

MMIX-ARITH64