5 Number representation

IEEE 754 encoding of the four *binary* floating-point types stored in 32, 64, 80, and 128 bits uses three fields: a 1-bit sign, an e-bit exponent ($e = 8, 11, 15, 15$), and an s-bit significand ($s = 24, 53, 64, 113$). The four significand sizes can hold 7, 15, 19, and 34 decimal digits, respectively. Unlike some historical floating-point designs, all stored bits are used in the IEEE 754 formats. The figures on page 6 show the storage layout.

Ignoring the 80-bit *temporary real* format, the exponent and significand sizes are, by design, large enough that double-length products can be computed *exactly*, and *without overflow*, in the next longer format, if available.

Except for zero and subnormals, the significand of a finite number is normalized so that the high-order bit is always 1, and that bit is *hidden* (not stored) in the 32-, 64-, and 128-bit formats. The binary point of normal numbers *follows* the leading bit, so the significand always lies in $[1, 2)$.[1]

The stored exponent field must be reduced by a bias of 127, 1024, 16 383, and 16 383 to obtain the true power-of-two. The smallest stored exponent, 0, indicates a subnormal number, which has no hidden bit, and may have leading zeros, reducing its precision until the smallest subnormal, with only a single bit of precision, is reached. Coonen [6] describes the mathematical reasons for requiring subnormals, including *gradual*, rather than *abrupt*, underflow to zero, but some deficient floating-point hardware does not support subnormals.

Evaluation of subnormals requires special handling, because the actual exponent of subnormals must be adjusted upward by one, and because no hidden bit must be supplied. Here is an experiment in **hoc32**, using the `ftoh()` function (floating-point to host representation) to reveal what is actually stored in memory:

```
hoc32> for (k = -1; k < 25; ++k) \
       {
          x = MINSUBNORMAL * 2**k
          printf("%3d %d %a %s\n", k, issubnormal(x), \
                                   hexfp(x), ftoh(x))
       }
 -1 0 0x0.000000p+0   00000000
  0 1 0x1.000000p-149 00000001
  1 1 0x1.000000p-148 00000002
  2 1 0x1.000000p-147 00000004
  3 1 0x1.000000p-146 00000008
  4 1 0x1.000000p-145 00000010
  ...
 20 1 0x1.000000p-129 00100000
 21 1 0x1.000000p-128 00200000
 22 1 0x1.000000p-127 00400000
```

[1] In this document, we use the common notation of *closed* (bracketed) and *open* (parenthesized) *intervals*: $[a, b]$ means any value x such that $a \le x \le b$; $[a, b)$ means $a \le x < b$; $(a, b]$ means $a < x \le b$; and (a, b) means $a < x < b$. In mathematics, Infinity (∞) would normally not occur in a closed interval, but on computers with IEEE 754 arithmetic, it can.

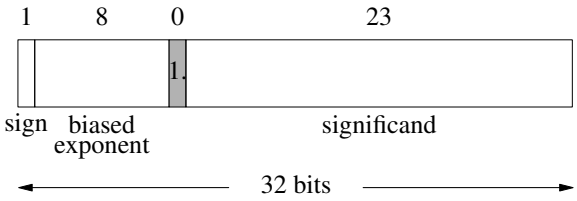

Figure 1: **IEEE 754 32-bit binary format.** The shaded box indicates the hidden (not stored) bit that is 1 for normal numbers, and 0 for subnormals. The binary point follows the hidden bit.

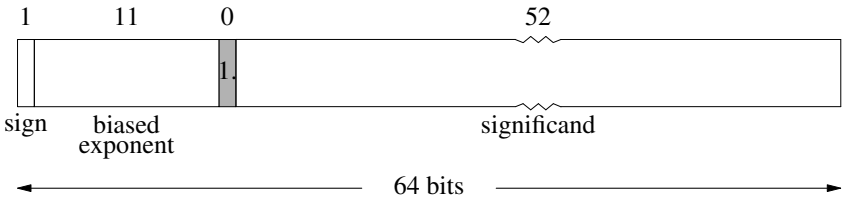

Figure 2: **IEEE 754 64-bit binary format.** The shaded box indicates the hidden (not stored) bit that is 1 for normal numbers, and 0 for subnormals. The binary point follows the hidden bit. The significand is not to scale.

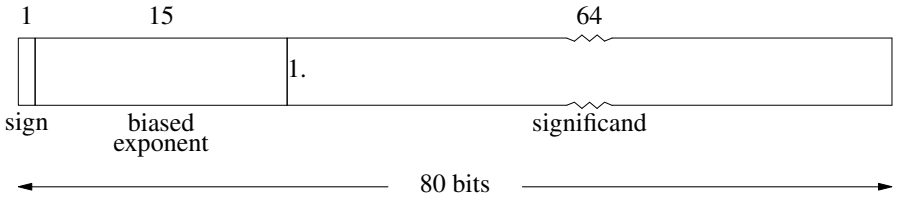

Figure 3: **IEEE 754 80-bit binary format.** There is no hidden bit. The binary point follows the first stored significand bit, which is 1 for normal numbers, and 0 for subnormals. The significand is not to scale. Depending on the CPU architecture, the *actual* storage format may be 10, 12, or 16 bytes (80, 96, or 128 bits).

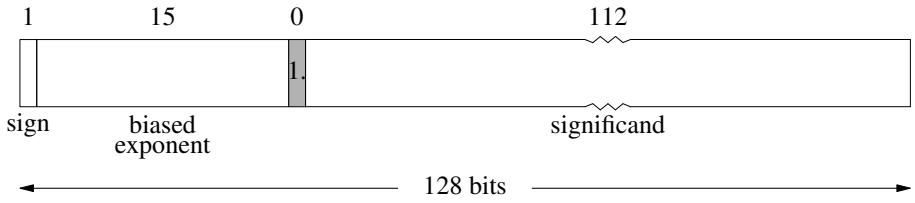

Figure 4: **IEEE 754 128-bit binary format.** The shaded box indicates the hidden (not stored) bit that is 1 for normal numbers, and 0 for subnormals. The binary point follows the hidden bit. The significand is not to scale.

```
23 0 0x1.000000p-126 00800000
24 0 0x1.000000p-125 01000000
```

The first output line shows an underflow to an exact zero, which is stored as all-bits-zero in the IEEE 754 binary formats. Results from $k = 0$ to $k = 22$ are subnormals, and the last column shows the hexadecimal representation of the four stored bytes. We can easily decode the first nine bits (sign + 8-bit exponent) as all zeros, but at $k = 23$, where we have the smallest normal number, the stored biased exponent is 1. *Thus, to construct a correct value from the product of the subnormal significand and its power-of-two, we need to treat its stored biased exponent,* 0, *as* 1.

The largest stored exponent field is reserved for Infinity and quiet and signaling NaNs. The stored significand is zero for Infinity, and nonzero for either type of NaN. Quiet and signaling NaNs are further distinguished by a specific significand bit (all vendors have chosen the highest stored *fraction* bit), but some CPU designs, notably the Intel x86, support only a single kind of NaN, treated as a quiet NaN. However, CPU designs differ in whether a 1-bit in the distinguishing position means quiet or signaling. The remaining significand bits can be used to store a user-defined *payload*, but until IEEE 754-2019, whether payloads are preserved in numerical operations was not specified, and designs differed in behavior.

In general, NaNs propagate in numerical operations, and with rare exceptions, software for computing a function $f(x)$ should generate a *run-time* NaN when x is a NaN, in order to set sticky exception flags, and return the *input* NaN, preserving the payload. When a numerical operator takes two operands, and both are NaNs, either of them may be returned.

We can see the implementation-dependent propagation of payloads in this experiment with **hoc64**:

```
hoc64> __INDENT__ = "\t"
hoc64> q = qnan("0xbeadface"); s = snan("0xcafefeed"); n = 0/0
hoc64> ftoh(q); ftoh(s); ftoh(n)
        7ff80000_beadface
        7ff00000_cafefeed
        fff80000_00000000
hoc64> ftoh(q + s); ftoh(q - s); ftoh(q * s); ftoh(q / s)
        7ff80000_cafefeed
        7ff80000_beadface
        7ff80000_cafefeed
        7ff80000_beadface
hoc64> ftoh(q + n); ftoh(q - n); ftoh(q * n); ftoh(q / n)
        fff80000_00000000
        7ff80000_beadface
        fff80000_00000000
        7ff80000_beadface
```

The sign of a NaN is *not significant*, and CPU designs again differ on whether a generated NaN has a 0 sign bit (positive) or a 1 sign bit (negative). In **hoc**, NaNs returned from `qnan()` and `snan()` always have a 0 sign bit. Those that come from

numerical operations have a hardware-dependent sign bit. Our examples are from an AMD64 (x86_64) CPU that always sets the sign bit of NaNs to 1. On that system, a floating-point stored value with all-bits-one means a quiet NaN with a negative sign.

NaNs have one other important property that was intended by the IEEE 754 designers to produce a fast and simple test for them: a NaN is unequal to anything, *including itself*. Thus, software in any programming language should be able use the equivalent of this pseudocode to report a NaN:

```
if (x != x)
    print "x is a NaN"
```

Unfortunately, overzealous optimizations by compiler writers insufficiently skilled in IEEE 754 floating-point arithmetic mean that such tests can fail. That is why math libraries should supply a function to make such tests immune to compiler vagaries:

```
if (isnan(x))
    print "x is a NaN"
```

The encoding for Infinity and NaNs in IEEE 754 binary floating-point formats means that all bits must be examined to distinguish between them. By contrast, in the IEEE 754 decimal format, the encoding was changed so that a particular single bit indicates which of the two is meant.

For a library that supports fixed-point arithmetic, we need to store a large array of coefficients, so there is little extra overhead in encoding the sign, Infinity, and NaN types in separate byte-sized fields. Importantly, that makes tests for them fast, because the coefficients can be completely ignored, and such tests are needed on *every* numerical operation.

6 Arithmetic exceptions

The IEEE 754 Standards require separate sets of exception flags for binary and decimal floating-point arithmetic, and for compatibility, a fixed-point arithmetic library must also supply a similar set of flags.

IEEE 754 exception flags are implemented as *sticky bits* inside a special CPU register (*not stored in DRAM* (dynamic random-access memory)): they are all zero when a program begins execution, are set by floating-point operations during execution, and never cleared, except by user request. Thus, a program can interrogate and report the flag status at the end of a computation, or program execution, as a retrospective on the run.

The sticky exception flags record minimal information. There is no provenance of where, or when, or why, the exceptions happened. There is only a single set of flags for the CPU and all of its cores and threads, so it is impossible to tell which of many threads in the current job got an exception. Because CPU registers, including exception flags, are saved and restored by the operating system kernel across context switches, other processes *cannot* affect your job's flags.

Add exception flags to `fx_t`?

Trapping instruction exceptions was a traditional way of handling run-time errors on computer architectures designed before the late 1980s. However, most modern

CPUs have up to several hundred instructions in an instruction pipeline, so even a single-threaded process cannot reliably identify the particular instruction that caused an exception. Thus, run-time exception handling through traps is impractical, and sticky flags provide a reasonable alternative.

At the hardware level, the flag register in the CPU is a single resource that must be contended for by all instruction units and all threads, and thus, can limit performance. In addition, it can sometimes be a source of the infamous TOCTOU (*time-of-check to time-of-use*) bug in parallel programming. Checking a bit flag and then taking one of two actions based on it may be incorrect if the flag changes between the check and the branch.

The ISO C Standards define five exception flags for binary floating-point arithmetic:

`FE_DIVBYZERO` division by zero was attempted;

`FE_INEXACT` a computed result required rounding;

`FE_INVALID` a NaN was input to, or produced by, one of the five basic arithmetic operations;

`FE_OVERFLOW` an Infinity (such as 1/0), or a value too large to represent as a finite machine number, was produced and replaced by an Infinity of the same sign;

`FE_UNDERFLOW` a computed value too small to represent as a normal number was returned as a subnormal number, or zero, and *in addition*, the result is *inexact*.

The C math libraries on some systems provide access to an additional exception flag with nonstandard names that the author's MathCW library [1] maps to the common name `FE_SUBNORMAL`, meaning that an operand of a floating-point instruction is in the subnormal region of nonzero values, and thus, has lower than normal precision. Sometimes, calculations that wander too frequently into the subnormals risk returning results with fewer significant digits than could be obtained by rescaling numbers to stay in the range of normal numbers.

A composite flag, `FE_ALL_EXCEPT`, stands for the logical *OR* of the five standard flags. Its primary use is for clearing *all* sticky flags, and testing whether *any* sticky flag is set.

The flags are set and tested with the functions `fegetenv()`, `fegetexceptflag()`, `feholdexcept()`, `feraiseexcept()`, `fesetenv()`, `fesetexceptflag()`, `fetestexcept()`, and `feupdateenv()`. The `get/set` or `clear/test` exception-flag pairs suffice for most applications.

A fixed-point arithmetic library should have counterparts of all of the five standard flags, except that the *inexact* flag would only be set when a number outside the fixed-point range is generated. There is no equivalent of `FE_SUBNORMAL` in fixed-point arithmetic, because all such numbers have the same storage format and potential precision. However, conversion of a floating-point subnormal to a fixed-point value sets the *subnormal* flag.

The specific values of the `FE_*` macros are CPU and implementation dependent, so user code should never use their particular numeric values, but only their symbolic names.

Our library uses similar names for the exception flags, but the prefix FE is changed to FX, and only three functions are provided for access to the fixed-point sticky flags hidden inside the library. Their prototypes look like this:

```
int fxclearexcept (int excepts);
                           /* clear specified exception flags */
int fxraiseexcept (int excepts);
                           /* raise specified exception flags */
int fxtestexcept  (int excepts);
                           /* test specified exception flags  */
```

Their integer argument is the bitwise-OR of one or more of the FX_* exception flags defined in fx.h.

All three return a negative value on failure. The first two return zero on success, and the third returns the bitwise-OR of the selected flags on success.

Two additional exception flags are defined. FX_FORMATUNKNOWN is set by fxfmt() if it encounters an unrecognized format descriptor. FX_STRINGOVERFLOW is set by conversion functions described starting in Section 16 on page 32 to indicate that a user-supplied output string was too small.

The four conversion functions from floating-point to fixed-point arithmetic all set the FX_SUBNORMAL flag if they receive a subnormal floating-point value.

The special value FX_ALL_EXCEPT is the logical *OR* of all defined fixed-point exception flags.

For convenience, you can also get a character string with a list of currently set exception flags:

```
char * fxshowflags (char s[], size_t maxs);
```

7 Investigations of underflow

The conditions under which the underflow exception is raised are subtle, and astonishingly hard to find clearly documented in CPU instruction set manuals, so we write code for a numerical experiment in **hoc32**:

```
printf("u i s    e         x             y              x*y\n")

for (e = 1/2; (1 + e) != 1; e /= 2) \
{
    x = 1 + e
    y = (1 - e) * MINNORMAL
    feclearexcept(FE_UNDERFLOW | FE_INEXACT)
    z = x * y
    u = (fetestexcept(FE_UNDERFLOW) != 0)
    i = (fetestexcept(FE_INEXACT)   != 0)
    s = issubnormal(z)
    printf("%d %d %d %.1a %.6a %.6a %a\n", u, i, s, e, x, y, z)
}
```

The exact product is $xy = (1-e)(1+e)$`MINNORMAL`, and the two parenthesized terms reduce to $1-e^2$. Thus, in *infinite precision*, the product xy is always too small to be a normal number.

Here is the program output on an AMD64 (x86_64) CPU, with empty lines inserted around the line where behavior changes.

```
u i s    e         x             y              x*y
0 0 1 0x1.0p-01 0x1.800000p+0 0x1.000000p-127 0x1.800000p-127
0 0 1 0x1.0p-02 0x1.400000p+0 0x1.800000p-127 0x1.e00000p-127
0 0 1 0x1.0p-03 0x1.200000p+0 0x1.c00000p-127 0x1.f80000p-127
0 0 1 0x1.0p-04 0x1.100000p+0 0x1.e00000p-127 0x1.fe0000p-127
0 0 1 0x1.0p-05 0x1.080000p+0 0x1.f00000p-127 0x1.ff8000p-127
0 0 1 0x1.0p-06 0x1.040000p+0 0x1.f80000p-127 0x1.ffe000p-127
0 0 1 0x1.0p-07 0x1.020000p+0 0x1.fc0000p-127 0x1.fff800p-127
0 0 1 0x1.0p-08 0x1.010000p+0 0x1.fe0000p-127 0x1.fffe00p-127
0 0 1 0x1.0p-09 0x1.008000p+0 0x1.ff0000p-127 0x1.ffff80p-127
0 0 1 0x1.0p-10 0x1.004000p+0 0x1.ff8000p-127 0x1.ffffe0p-127
0 0 1 0x1.0p-11 0x1.002000p+0 0x1.ffc000p-127 0x1.fffff8p-127

1 1 0 0x1.0p-12 0x1.001000p+0 0x1.ffe000p-127 0x1.000000p-126

0 1 0 0x1.0p-13 0x1.000800p+0 0x1.fff000p-127 0x1.000000p-126
0 1 0 0x1.0p-14 0x1.000400p+0 0x1.fff800p-127 0x1.000000p-126
0 1 0 0x1.0p-15 0x1.000200p+0 0x1.fffc00p-127 0x1.000000p-126
0 1 0 0x1.0p-16 0x1.000100p+0 0x1.fffe00p-127 0x1.000000p-126
0 1 0 0x1.0p-17 0x1.000080p+0 0x1.ffff00p-127 0x1.000000p-126
0 1 0 0x1.0p-18 0x1.000040p+0 0x1.ffff80p-127 0x1.000000p-126
0 1 0 0x1.0p-19 0x1.000020p+0 0x1.ffffc0p-127 0x1.000000p-126
0 1 0 0x1.0p-20 0x1.000010p+0 0x1.ffffe0p-127 0x1.000000p-126
0 1 0 0x1.0p-21 0x1.000008p+0 0x1.fffff0p-127 0x1.000000p-126
0 1 0 0x1.0p-22 0x1.000004p+0 0x1.fffff8p-127 0x1.000000p-126
0 1 0 0x1.0p-23 0x1.000002p+0 0x1.fffffcp-127 0x1.000000p-126
```

The first three output columns show the *underflow* and *inexact* flags, and whether the rounded product is *subnormal*.

In the first 11 lines, xy rounds to a subnormal value, but the sum of the nonzero leading bits in the factors does not exceed 23, so the product fits exactly in the 24 available bits. Thus, the underflow flag is *not* set.

In line 12, the factors have 13 and 12 bits, respectively, and the product has 25 bits, so 1 bit is lost in the rounding operation, producing an *inexact* result. The *underflow* flag is set because both conditions — tiny and inexact — hold.

In lines 13 to 23, the exact products need 27 to 47 bits, so the *inexact* exception is set. However, this processor detects underflow *after* rounding, and we investigate shortly why the *underflow* flag is not set.

Output of the same program on a SPARC64 CPU differs in that the underflow column is 1 in lines 12 to 23: that CPU detects underflow *before* rounding.

Table 1: **Results causing tininess on HPPA**. The value m is the minimum normal number, p is number of bits in the significand, and the computed value v is the higher-precision result *before* rounding.

Rounding mode	Range
FE_TONEAREST	$-(1-2^{-(p+1)})m < v < +(1-2^{-(p+1)})m$
FE_TOWARDZERO	$-m < v < +m$
FE_UPWARD	$-m < v \le +(1-2^{-p})m$
FE_DOWNWARD	$-(1-2^{-p})m \le v < +m$

Both underflow-detection behaviors are permitted by the IEEE 754 Standards for binary arithmetic. However, for decimal arithmetic, the Standards require underflow detection *before* rounding.

The same code in both **hoc32** and C was run on systems with other CPU architectures. They demonstrated that AMD64, HPPA, IA-64, MIPS32, MIPS64, RISC-V, and x86 all detect tininess *after* rounding. The Alpha, ARM32, ARM64, PowerPC, S390x, and SPARC64 CPUs all detect tininess *before* rounding. Because Intel and ARM processors are used in most of the world's desktop computers and mobile devices, we must conclude that both rounding behaviors are in wide use.

The unset underflow flag in lines 13 to 23 is inexplicable from the descriptions of underflow handling in most CPU architecture manuals, but I eventually found some crucial details in the *HP Precision Architecture and Instruction Set Reference Manual* (April 1989 edition). Table 6-18 on page 6-24 of that book is reproduced here in Table 1, with slight changes in notation. Figure 5 shows the narrow region $[b, m)$ where an exact subnormal result fails to set the underflow flag on some CPU designs, *even when the rounded result is inexact*. By contrast, processors that detect tininess before rounding show no such anomalies: the underflow flag is set *only* when both conditions required by IEEE 754 are met.

Another anomaly detected by our underflow tests is on the QEMU emulation of

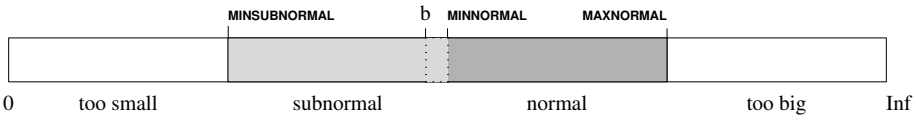

Figure 5: **Anomalous rounding behavior**. The white regions indicate numbers outside the floating-point range. The dotted region $[b, m)$, where $b = (1-2^{-(p+1)})m$, and m is the minimum normal number, is closer to the smallest normal number than the separation between representable numbers, yet is accessible with higher internal precision. In some CPU designs, results falling in the dotted region do not cause the underflow flag to be set, even though the stored result is inexact, and is either the largest subnormal, or the smallest normal.

M68K: the factors and products agree with the results from other CPU architectures, but the first two exception bits in the output are always zero. To further check that surprise, I wrote a small assembly language function to retrieve the floating-point status register where the IEEE 754 sticky flag bits are recorded: the returned register value is *always zero*. This could indicate a flaw in the instruction emulation, but tests on old, but still-running, machines with real M68K CPUs are needed.

8 Choice of base

Sensible representations of numbers for numerical computation are essentially polynomials in powers of a particular base. Thus, when we write $\pi \approx$ `3.141592...`, we mean that in base 10

$$\pi \approx 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3} + 5 \times 10^{-4} + 9 \times 10^{-5} + 2 \times 10^{-6} + \cdots .$$

In base $2^4 = 16$, we could equally write $\pi \approx$ `0x3.243f6ap0`, requiring 4-bit coefficients in the sum

$$\pi \approx 3 \times 16^0 + 2 \times 16^{-1} + 4 \times 16^{-2} + 3 \times 16^{-3} + 15 \times 16^{-4} + 6 \times 16^{-5} + 10 \times 16^{-6} + \cdots .$$

In base $2^8 = 256$, we need 8 bits for each coefficient, and we have

$$\pi \approx 3 \times 256^0 + 36 \times 256^{-1} + 63 \times 256^{-2} + 106 \times 256^{-3} + \cdots .$$

In base $2^{16} = 65\,536$, we have 16-bit coefficients and the expansion

$$\pi \approx 3 \times 65\,536^0 + 9279 \times 65\,536^{-1} + 27272 \times 65\,536^{-2} + \cdots .$$

By contrast, in base $2^1 = 2$, each coefficient requires only a single bit, but we need many more terms:

$$\begin{aligned}
\pi \approx\ & 1 \times 2^1 + 1 \times 2^0 + \\
& 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + \\
& 0 \times 2^{-5} + 1 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8} + \\
& 0 \times 2^{-9} + 0 \times 2^{-10} + 1 \times 2^{-11} + 1 \times 2^{-12} + \\
& 1 \times 2^{-13} + 1 \times 2^{-14} + 1 \times 2^{-15} + 1 \times 2^{-16} + \\
& 0 \times 2^{-17} + 1 \times 2^{-18} + 1 \times 2^{-19} + 0 \times 2^{-20} + \\
& 1 \times 2^{-21} + 0 \times 2^{-22} + 1 \times 2^{-23} + 0 \times 2^{-24} + \cdots .
\end{aligned}$$

The run time of code that implements addition and subtraction in fixed-point arithmetic is *linear* in the number of coefficients, but that for division, multiplication, and square root is proportional to the *product* of the numbers of coefficients in the two operands. Thus, fast execution calls for a large base, but other implementation details determine the practical limit on the size of the base.

We choose an integer type for the coefficients, because the 1999 ISO Standard for C requires support for arithmetic on signed and unsigned 64-bit integer types, and most CPU designs since then provide them in hardware. Recall that the significand sizes on the 32-, 64-, 80-, and 128-bit IEEE 754 binary floating-point types are 24, 53, 64, and 113 bits, respectively.