A bit of terminology

- “Binary” = “base two”.
- A bit is a single binary digit; it can be either 0 or 1.
- A byte is a sequence of 8 bits.
- A kilo-byte is a sequence of 1024 bytes ($1024 = 2^{10}$).

How a computer stores numbers

Integers are stored as one would expect: a series of binary digits (plus a bit for $\pm$). **Real** numbers are stored as three

parts: the mantissa, exponent, and sign. This is rather like standard form: in base ten, we would write (to 5 significant figures)

$$1234.567 \sim 1.2346 \times 10^3 \equiv +1.2346\mathrm{E}_{10}(+3)$$

We refer to the "1.2346" as the **mantissa** and the "$+03$" as the **exponent**. The computer does this, but instead does standard form in base two.

Floating point in binary

So how to write 1234.567 in binary standard form? We note that

$$
\begin{aligned}
1234.567 &\sim 1024 + 128 + 64 + 16 + 2 + \frac{1}{2} + \frac{1}{16} \\
&= 2^{10} + 2^{7} + 2^{6} + 2^{4} + 2^{1} + 2^{-1} + 2^{-4} \\
&= \left(1 + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-9} + 2^{-11} + 2^{-14}\right) \times 2^{10}
\end{aligned}
$$

where the stuff left out is less than 2^{-7}. So in binary

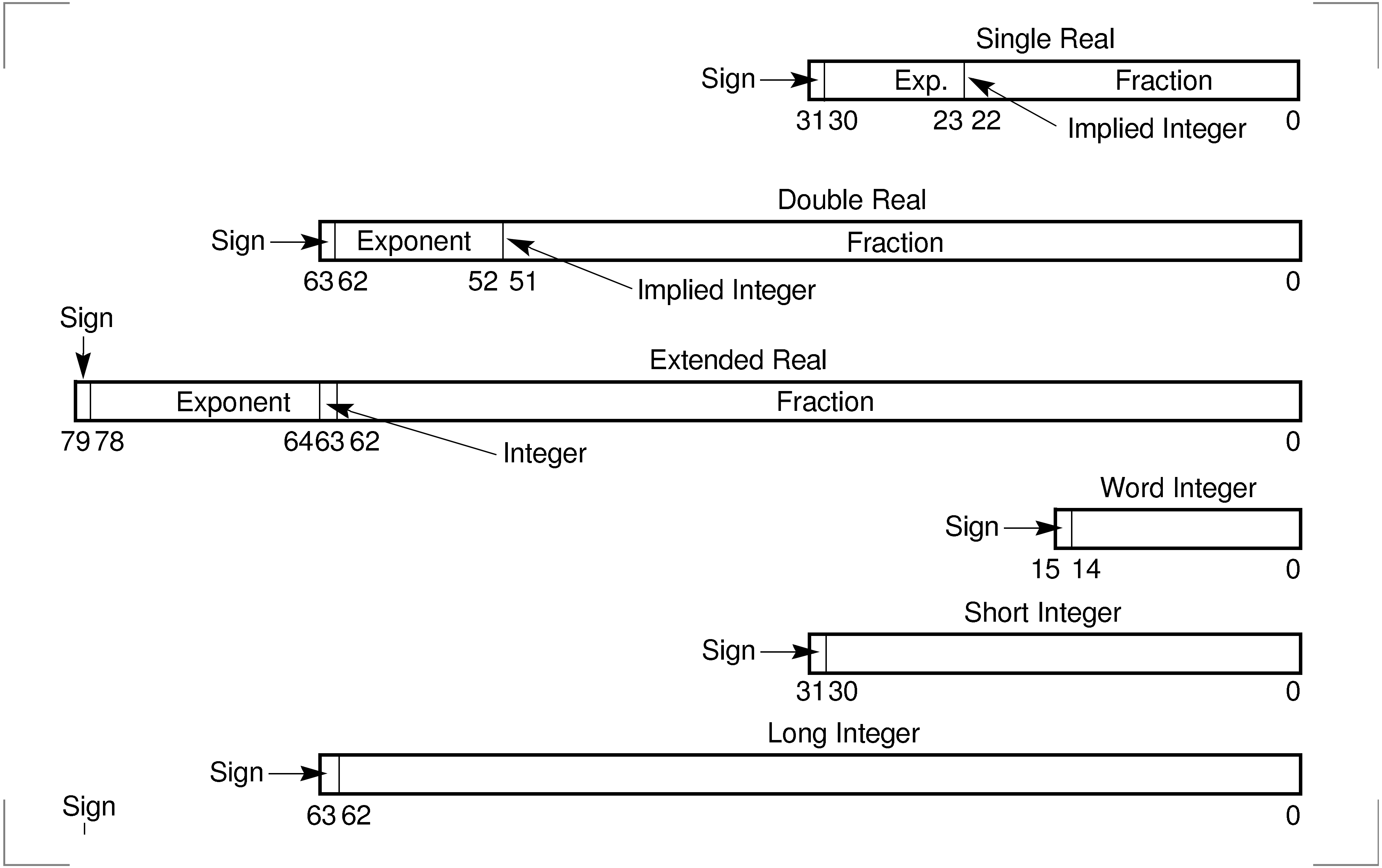
$$+1.234567\mathrm{E}_{10}(+3)(\mathrm{base}10) \sim +1.00110100101001\mathrm{0E}_{2}(+110)(\mathrm{base}2)$$

So the precision of a floating number involves the number of digits available in both the mantissa and the exponent.

Allowed representations

- Each particular processor architecture (e.g. Intel's P4, Sun UltraSPARC IV) has its own methods for storing integer and floating point numbers.
- Only certain choices of representation are permitted on a particular processor. E.g. Vol. 1 of the "Intel Architecture Software Developer's Manual" document no. 24319002 (fig. 7-17) (next slide):

Intel data types



Fortran implementation

- So far we have not specified the precision of variables that we have declared - we have left it to the compiler to choose the default for the system. Our declaration statements have looked like:

```
REAL    ::  x
```

- Fortran permits a further type specifier, called the `KIND` number (itself of type `INTEGER`). To declare a `REAL` variable with a `KIND` number of 4, we would program:

```
REAL(KIND=4)     ::  x
```

or the equivalent shorter form:

```
REAL(4)    ::  x
```

Fibonacci revisited

Recall we wrote a code to generate the series $\{x_n\}$ obeying the relation

$$x_n = x_{n-1} + x_{n-2} \quad n \geq 3 \tag{1}$$

with initial condition $x_1 = x_2 = 1$.
We also estimated the Golden Ratio ϕ from the relation

$$\phi \equiv \frac{\sqrt{5}+1}{2} = \lim_{n\to\infty} \frac{x_n}{x_{n-1}} \tag{2}$$

Recall: Fibonacci program 1

```
PROGRAM fibprog1    ! program name
  IMPLICIT NONE     ! assume nothing about variable names
  INTEGER :: x, y, z ! declare some variables
  REAL :: phi_est   ! approximate phi
  INTEGER :: n = 3  ! declare the iterator
  z = 1             ! initial condition
  y = 1             ! initial condition
2 x = y + z         ! iteration step, preceded by label '2'
  phi_est = x / (1.0 * y) ! approximate phi
  PRINT*, x, phi_est ! print out values
  z = y             ! shift variables
  y = x             ! shift variables
  n = n + 1         ! increment the counter
  IF (n < 50) GO TO 2
END PROGRAM fibprog1
```

Fibonacci program 2

```
PROGRAM fibprog2
  IMPLICIT NONE
  ! we want to use integers of at least 12 digits
  INTEGER, PARAMETER :: ki = SELECTED_INT_KIND(12)
  INTEGER(ki) :: x, y = 1, z = 1 ! declare some variables
  ! we want f.p. accurary of at least 16 digits
  INTEGER, PARAMETER :: kr = SELECTED_REAL_KIND(16)
  REAL(kr) :: phi_est                    ! our estimate for phi
  INTEGER :: n                           ! declare the iterator
  DO n = 3, 50                           ! begin loop
    x = y + z                            ! iteration step
    phi_est = x / (1.0_kr * y)           ! approximate phi
    PRINT*, x, phi_est                   ! print out values
    z = y                                ! shift variables
    y = x                                ! shift variables
  END DO
END PROGRAM fibprog2
```