## Using The Conversion Procedure

- Convert 2.625 to our 8-bit floating point format.
  - A. The integral part is easy, $2_{10} = 10_2$. For the fractional part:
    - $0.625 \times 2 = 1.25$ [1] *Generate 1 and continue with the rest.*
    - $0.25 \times 2 = 0.5$ [0] *Generate 0 and continue.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $0.625_{10} = 0.101_2$, and $2.625_{10} = 10.101_2$.
  - B. Add an exponent part: $10.101_2 = 10.101_2 \times 2^0$.
  - C. Normalize: $10.101_2 \times 2^0 = 1.0101_2 \times 2^1$.
  - D. Mantissa: 0101
  - E. Exponent: $1 + 3 = 4 = 100_2$.
  - F. Sign bit is 0.

  The result is [0|100|0101]. Represented as hex, that is $45_{16}$.
- Convert -4.75 to our 8-bit floating point format.
  - a. The integral part is $4_{10} = 100_2$. The fractional:
    - $0.75 \times 2 = 1.5$ [1] *Generate 1 and continue with the rest.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $4.75_{10} = 100.11_2$.
  - b. Normalize: $100.11_2 = 1.0011_2 \times 2^2$.
  - c. Mantissa is 0011, exponent is $2 + 3 = 5 = 101_2$, sign bit is 1.

  So -4.75 is [1|101|0011] = $d3_{16}$
- Convert 0.40625 to our 8-bit floating point format.
  - a. Converting:
    - $0.40625 \times 2 = 0.8125$ [0] *Generate 0 and continue.*
    - $0.8125 \times 2 = 1.625$ [1] *Generate 1 and continue with the rest.*
    - $0.625 \times 2 = 1.25$ [1] *Generate 1 and continue with the rest.*
    - $0.25 \times 2 = 0.5$ [0] *Generate 0 and continue.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $0.40625_{10} = 0.01101_2$.
  - b. Normalize: $0.01101_2 = 1.101_2 \times 2^{-2}$.
  - c. Mantissa is 1010, exponent is $-2 + 3 = 1 = 001_2$, sign bit is 0.

  So 0.40625 is [0|001|1010] = $1a_{16}$

The result is [0|011|1011] = $3b_{16}$. This is not exact, of course. If you convert it back to decimal, you get

- Convert -1313.3125 to IEEE 32-bit floating point format.
  - a. The integral part is $1313_{10} = 10100100001_2$. The fractional:
    - $0.3125 \times 2 = 0.625$ [0] *Generate 0 and continue.*
    - $0.625 \times 2 = 1.25$ [1] *Generate 1 and continue with the rest.*
    - $0.25 \times 2 = 0.5$ [0] *Generate 0 and continue.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $1313.3125_{10} = 10100100001.0101_2$.
  - b. Normalize: $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$.
  - c. Mantissa is 01001000010101000000000, exponent is $10 + 127 = 137 = 10001001_2$, sign bit is 1.

  So -1313.3125 is [1|10001001|01001000010101000000000] = $c4a42a00_{16}$
- Convert 0.1015625 to IEEE 32-bit floating point format.
  - a. Converting:
    - $0.1015625 \times 2 = 0.203125$ [0] *Generate 0 and continue.*
    - $0.203125 \times 2 = 0.40625$ [0] *Generate 0 and continue.*
    - $0.40625 \times 2 = 0.8125$ [0] *Generate 0 and continue.*
    - $0.8125 \times 2 = 1.625$ [1] *Generate 1 and continue with the rest.*
    - $0.625 \times 2 = 1.25$ [1] *Generate 1 and continue with the rest.*
    - $0.25 \times 2 = 0.5$ [0] *Generate 0 and continue.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $0.1015625_{10} = 0.0001101_2$.
  - b. Normalize: $0.0001101_2 = 1.101_2 \times 2^{-4}$.
  - c. Mantissa is 10100000000000000000000, exponent is $-4 + 127 = 123 = 01111011_2$, sign bit is 0.

  So 0.1015625 is [0|01111011|10100000000000000000000] = $3dd00000_{16}$
- Convert 39887.5625 to IEEE 32-bit floating point format.
  - a. The integral part is $39887_{10} = 1001101111001111_2$. The fractional:
    - $0.5625 \times 2 = 1.125$ [1] *Generate 1 and continue with the rest.*
    - $0.125 \times 2 = 0.25$ [0] *Generate 0 and continue.*
    - $0.25 \times 2 = 0.5$ [0] *Generate 0 and continue.*
    - $0.5 \times 2 = 1.0$ [1] *Generate 1 and nothing remains.*

    So $39887.5625_{10} = 1001101111001111.1001_2$.
  - b. Normalize: $1001101111001111.1001_2 = 1.00110111100111111001_2 \times 2^{15}$.

n particular:

```
0 00000000 00000000000000000000000 =  0
0 00000000 00000000000000000000001 = +1 * 2^(   -126) *
                                       0.00000000000000000000001_2 =
                                       2^(-149)  (Smallest positive value)
0 00000000 10000000000000000000000 = +1 * 2^(   -126) * 0.1_2 = 2**(-127)

0 00000001 00000000000000000000000 = +1 * 2^(  1-127) * 1.0_2 = 2**(-126)
0 10000000 00000000000000000000000 = +1 * 2^(128-127) * 1.0_2   = 2
          1 0  0
          1 .5  .5     2^1
0 10000001 10100000000000000000000 = +1 * 2^(129-127) * 1.101_2 = 6.5

           1.10
           1  1  0   1
           1 .5 .25  .125   1.624*2^2= 6.5

0 11111110 11111111111111111111111 = +1 * 2^(254-127) *
                                     1.11111111111111111111111_2
                                     (Most positive finite value)
```

```
0 11111111 00000000000000000000000 =  Infinity
0 11111111 00000100000000000000000 =  NaN
1 00000000 00000000000000000000000 = -0
1 10000000 00000000000000000000000 = -1 * 2^(128-127) * 1.0_2   = -2
1 10000001 10100000000000000000000 = -1 * 2^(129-127) * 1.101_2 = -6.5
1 11111110 11111111111111111111111 = -1 * 2^(254-127) *
                                     1.11111111111111111111111_2
                                     (Most negative finite value)
1 11111111 00000000000000000000000 = -Infinity
1 11111111 00100010001001010101010 = NaN
```



# Floating Point

## Introduction

Int and unsigned int are approximations to the set of integers and the set of natural numbers. Unlike int and unsigned, the set of integers and the set of natural numbers is infinite. Because the set of int is finite, there is a maximum int and a minimum int.
Ints are also contiguous. That is, between the minimum and maximum int, there are no missing values.
To summarize:

- The set of valid ints is finite.
- Therefore, there is a minimum and maximum int.
- ints are also contiguous. That is, there are no missing integer values between the minimum and maximum int.

There's also another key feature of ints that doesn't appear in the set of integers. ints have an underlying representation. The representation is binary numbers. The set of integers is often represented as base 10 numerals, but is often thought of more abstractly. That is, the set is independent of its representation (i.e., we can represent the set of integers in any way we want).
What issues comes up when trying to devise a data representation for floating point numbers? It turns out these issues are more complicated than representing integers. While most people agree that UB and 2C are the ways to represented unsigned and signed integers. Representing real numbers has traditionally been more problematic.
In particular, depending on which company manufactured the hardware, there were different ways to represent real numbers, and to manipulate it. There's no particularly obvious choice of how to represent real numbers. In the mid 1980's, the need for uniform treatment of real numbers (called floating point numbers) lead to the IEEE 754 standard.
Standards are often developed to give consistent behavior. For example, when C was first being developed, what was considered a valid C program very much depended on the compiler. A program that compiled on one C compiler might not compile on another. Effectively, many different flavors of C were being created, and the need to have a standard definition of a language was seen as important.
Similar, for purposes of agreeing on results performed on floating point numbers, there was a desire to standardize the way floating point numbers were represented.
Before we get to such issues, let's think about what restrictions will have to be imposed on floating point numbers.

- Since the number of bits used to represent a floating point number is finite, there must be a maximum float and a minimum float.
- However, since real numbers are dense (i.e., between any two distinct real numbers, there's another real number), there's no way to make any representation of real numbers contiguous. Integers do not have this denseness property.

This means we need to decide which real numbers to keep and which ones to get rid of. Clearly, any number that has repeated decimals or never repeats is not something that can be represented as a floating point number.

## Scientific Notation

Why do we need to represent real numbers? Of course, it's important in math. However, real numbers are important for measurements in science.
**Precision vs. Accuracy**
Let's define these two terms:

**Definition** *Precision* refers to the number of significant digits it takes to represent a number. Roughly speaking, it determines how fine you can distinguish between two measurements.
**Definition** *Accuracy* is how close a measurement is to its correct value.

A number can be precise, without being accurate. For example, if you say someone's height is 2.0002 meters, that is precise (because it is precise to about 1/1000 meters). However, it may be inaccurate, because a person's height may be significantly different.
In science, precision is usually defined by the number of significant digits. This is a different kind of precision than you're probably used to. For example, if you have a scale, you might be lucky to have precision to one pound. That is, the error is +/- 1/2 pound. Most people think of precision as the smallest measurement you can make.
In science, it's different. It's about the number of significant digits. For example, $1.23 * 10^{10}$ has the same precision as $1.23 * 10^{-10}$, even though the second quantity is much, much smaller than the first. It may be unusual, but that's how we'll define precision.
When we choose to represent a number, it's easier to handle precision than accuracy. I define accuracy to mean the result of a measured value to its actual value. There's not much a computer can do directly to determine accuracy (I suppose, with sufficient data, use statistical methods to determine accuracy).
**Accuracy of Computations**

There are two distinct concepts: the accuracy of a value that's recorded or measured, and the accuracy of performing operations with numbers.

We can't do much about the accuracy of the recorded value (without additional information). However, the hardware does perform mathematical operations reasonably accurately. The reason the computations aren't perfectly accurate is because one needs infinitely precice math, and that requires an infinite number of bits, which doesn't exist on a computer.
Since floating point numbers can not be infinite precise, there's always a possibility of error when performing computations. Floating point numbers often approximate the actual numbers. This is precisely because floating point numbers can not be infinitely precise.
In the field on computer science, numerical analysis is concerned with ways of performing scientific computations accurately on a computer. In particular, there are ways to minimize the effect of "round-off errors", errors that are due to the approximate nature of floating point representation.

**Canonical Representation**

When representing numbers in scientific notation, it has the following form:

$$+/- \ S \ X \ 10^{exp}$$

where **S** is the significand or the mantissa, and **exp** is the exponent. "10" is the base.

In scientific notation, there can be more than one way of writing the same value. For example, **6.02 X $10^{23}$** is the same as **60.2 X $10^{22}$** is the same as **602 X $10^{21}$**.
For any number represented in this way, there are an infinite number of other ways to represent this (by moving the decimal point, and adjust the exponent).
It might be nice to have a single, consistent way of doing this, i.e. a *canonical* or standard way of doing this. And, so there is such a way.

$$+/- \ D.FFFF \ X \ 10^{exp}$$

You can write the significand as **D.FFF...** where **1 <= D <= 9**, and **FFF...** represents the digits after the decimal point. If you follow this restriction on **D**, then there's only one way to write a number in scientific notation. The obvious exception to this rule is representing 0, which is a special case.
You can generalize this formula for other bases than base 10. For base K (where **K > 1**), you write the canonical scientific notation form as:

$$+/- \ D.FFFF \ X \ K^{exp}$$

where **1 <= D <= K - 1**. The F's that appear in the fraction must follow the rule: **0 <= D <= K - 1**.
The *number of significant digits*, which is also the *precision*, is the number of digits after the radix point. We call it the *radix point* rather than the *decimal point* because decimal implies base 10, and we could be talking about any other base.

## Binary Scientific Notation

As with any representation on a computer, we need to represent numbers in binary. So, this means we specialize the formula to look like:

$$+/- \ D.FFFF \ X \ 2^{exp}$$

This creates an interesting constraint on **D**. In particular, **1 <= D <= 1**, which means **D** is forced to be 1. We'll use this fact later on.

## IEEE 754 Single Precision

IEEE 754 floating point numbers was a standard developed in the 1980s, to deal with the problem of non-standard floating point representation.
There is a standard for single precision (32 bits) and double precision (64 bits). We'll primarily focus on single precision.

## Chart

| Sign | Exponent | Fraction |
|---|---|---|
| 1 | 8 bits | 23 bits |
| $b_{31}$ | $b_{30-23}$ | $b_{22-0}$ |
| X | XXXX XXXX | XXXX XXXX XXXX XXXX XXXX XXX |

An IEEE 754 single precision number is divided into three parts. The three parts match the three parts of a number written in canonical binary scientific notation.

- **sign bit** This is $b_{31}$. If this value is 1, the number is negative. Otherwise, it's non-negative.
- **exponent** The exponent is excess/bias 127. Normally, one expect the excess/bias to be half the number of representations. In this case, the number of representations is 256, and half of that is 128. Nevertheless, the excess is 127. Thus, the range of possible exponents is **-127 <= exp <= 128**.
- **fraction** Normally, you would represent the significant (also called the *mantissa*. This would mean representing **D.FFFF...**. However, recall that for base 2, **D = 1**. Since **D** is always 1, there's no need to represent the 1. You only need to represent the bits after the radix point.
  Thus, the "1" left of the radix point is NOT explicitly represented. We call this the *hidden one*.

IEEE 754 single precision has 24 bits of precision. 23 of the bits are explicitly represented, and the additional "hidden 1" is the 24th bit.
There is something of a fallacy when we say that a IEEE single precision has 24 bits of precision. In particular, it's very much like saying that a calculator with 15 digits has 15 digits of precision. It's true that it may represent all numbers with 15 digits, but the question is whether the value is really that precise.
For example, suppose a measured number has only 3 digits of precision. There's no way to indicate this on a calculator. The calculator is prepared to have 15 digits of precision, even though that's more accurate than the number.
The same can be said about representing floating point numbers. It has 24 bits of precision, but that may not accurately represent the true number of significant bits. Unfortunately, that's the best computers can do. One could store additional information to determine exactly how many bits really are significant, but this is not usually done.

**Categories**

Unlike UB or 2C, floating point numbers in IEEE 754 do not all fall in the same category. IEEE 754 identifies 5 different categories of floating point numbers.

You might wonder why they do this. Here's one reason. Given the representation, as is, there would be no way to represent 0. If all the bits were 0, this would be the number **1.0 X $2^{-127}$**. Although this is a small number, it's not 0.
Thus, we designate the bitstring containing all 0's to be zero.
The following is a list of categories of floating point numbers in IEEE 754.

- **zero** Because there is a sign bit, there is a positive and negative representation of 0.
- **infinity** There is also a positive and negative infinity. Infinity occurs when you divide a non-zero number by zero. For example, 1.0/0.0 produces infinity.
- **NaN** This stands for "not a number". NaN usually occurs when you do an ill-defined operation. The canonical example is dividing 0.0/0.0, which does not have a defined value.
- **denormalized numbers** These are numbers which have fewer bits of precision, and are smaller (in magnitude) than normalized numbers. We'll discuss denormalized numbers in detail momentarily.
- **normalized numbers** These are standard floating point numbers. Most bitstring patterns in IEEE 754 are normalized numbers.

**How to Tell Which Category a Float is**

It would be useful to know which category a given a bitstring falls in. Here's the chart.

| Category | Sign Bit | Exponent | Fraction |
|---|---|---|---|
| Zero | Anything | $\mathbf{0^8}$ | $\mathbf{0^{23}}$ |
| Infinity | Anything | $\mathbf{1^8}$ | $\mathbf{0^{23}}$ |
| NaN | Anything | $\mathbf{1^8}$ | Not $\mathbf{0^{23}}$ |
| Denormalized numbers | Anything | $\mathbf{0^8}$ | Not $\mathbf{0^{23}}$ |
| Normalized numbers | Anything | Not $\mathbf{0^8}$, nor $\mathbf{1^8}$ | Anything |

Again, we write $\mathbf{0^8}$ to mean 0, repeated 8 times, i.e. 0000 0000.

Notice that there is a positive and negative 0.

### Denormalized Numbers

Suppose that we allowed all 0's to be a normalized number (it's not though, it's really designated as zero).
A bitstring with 32 zeros would be $\mathbf{1.0 \text{ X } 2^{-127}}$. That's a pretty small value. However, we could represent numbers to get even smaller, if we do the following when the exponent is $\mathbf{0^8}$.

- Do not have a hidden 1. $\mathbf{b_{23\text{-}0}}$ would be the bits appearing after a radix point.
- Fix the exponent to -126.

Recall that the exponent is written with a bias of 127. So, you would expect that if the exponent is $\mathbf{0^8}$, this bitstring would represent the exponent -127, and not -126. However, there's a good reason why it's -126. We'll explain why in a moment.
For now, let's accept the fact that the exponent is -126 whenever the exponent bitstring is $\mathbf{0^8}$.

**Largest Positive Denormalized Number**

What's the largest positive denormalized number? This is when the fraction is $\mathbf{1^{23}}$. It looks like:

```
S    Exp              Fraction
- --------- -----------------------------
0 0000 0000 1111 1111 1111 1111 1111 111
```

This bitstring maps to the number $\mathbf{0.(1^{23}) \text{ x } 2^{-126}}$. This number has 23 bits of precision, since there are 23 1's after the radix point.
**Smallest Positive Denormalized Number**
What's the smallest positive denormalizd number?

- Exponent bitstring $\mathbf{0^8}$. (All denormalized numbers have this bitstring). It's value is -126.
- The fraction is $\mathbf{(0^{22})1}$, i.e., 22 zeroes followed by a single 1.

This looks like:

```
S    Exp              Fraction
- --------- -----------------------------
0 0000 0000 0000 0000 0000 0000 0000 001
```

This bitstring pattern maps to the number $\mathbf{0.0^{22}1 \text{ x } 2^{-126}}$, which is $\mathbf{1.0 \text{ x } 2^{-149}}$. This number has 1 bit of precision. The 22 zeroes are merely place holders and do not affect the number of bits of precision.
You may not *believe* that this number has only 1 bit of precision, but it does. Consider the decimal number 123. This number has 3 digits of precision. Consider 00123. This also has 3 digits of precision. The leading 0's do not affect the number of digits of precision. Similary, if you have 0.000123, the zeroes are merely to place the 123 correctly, but are not significant digits. However, 0.01230 has 4 significant digits, because the rightmost 0 actually adds to the precision.
Thus, for our example, we have 22 zeroes followed by a 1 after the radix point, and the 22 zeroes have nothing to do with the number of significant bits.

By using denormalized numbers, we were able to make the smallest positive float to be $\mathbf{1.0 \text{ X } 2^{-149}}$, instead of $\mathbf{1.0 \text{ X } 2^{-127}}$, which we would have had if the number had been normalized.
Thus, we were able to go 22 orders of magnitude smaller, by sacrificing bits of precision.

### Why -126 and not -127?

When the exponent bitstring is $\mathbf{0^8}$, this is mapped to exponent -126. Yet, for normalized IEEE 754 single precision floating point numbers, the bias on the exponent is -127. Why is it -126 instead of -127.

To answer this question, we need to look at the smallest positive normalized number. This occurs with the following bitstring pattern

```
S    Exp            Fraction
- --------- ----------------------------
0 0000 0001 0000 0000 0000 0000 0000 000
```

This bitstring maps to **1.0 x $2^{-126}$**.
Let's look at the two choices for the largest positive denormalized numbers.

- **0.($1^{23}$) x $2^{-127}$** (exponent is 127)
- **0.($1^{23}$) x $2^{-126}$** (exponent is 126---this is what's really used in IEEE 754 single precision)

Both choices are smaller than **1.0 x $2^{-126}$**, the smallest *normalized* (In particular, notice that the number with the exponent of -126 is smaller). That's *good* because we want to avoid overlap between normalized and denormalized numbers. Also notice that the number with the -126 as exponent is larger than the number that has -127 as exponent (both have the same mantissa/significand, and -126 is larger than -127).
Thus, by picking -126 instead of -127, the gap between the largest denormalized number and the smallest normalized number is smaller. Is this a necessary feature? Is it really necessary to make that gap small? Maybe not, but at least there's some rationale behind the decision.

### Converting Normalized from Base 10 to IEEE 754

Let's convert 10.25 from base 10 to IEEE 754 single precision. Here's the steps:

- **Convert the number left of the radix point to base 2** Thus, $10_{10}$ is $1010_2$.
- **Convert the number right of the radix point to base 2.** Thus, $.25_{10}$ is $.01_2$.
- **Add the two.** This results in 1010 + 0.01, which is 1010.01.
- **Write this in binary scientific notation.** This is **1010.01 X $2^0$**, which is **1.01001 X $2^3$**.
- **Write this in IEEE 754 single precision.** This is **1010.01 X $2^0$**, which is **1.01001 X $2^3$**.
- Convert 3 to the correct bias. Since the bias is 127, add 127 to 3 to get 130 and convert to binary. This turns out to be 1000 0010.
- Write out the number in the correct representation `S    Exp            Fraction`
- `- --------- ----------------------------`
- `0 1000 0010 0100 1000 0000 0000 0000 000`
-

  Notice that the hidden "1" is *not* represented in the fraction.

### An Algorithm for Writing Positive Exponent in Excess 127

Converting 130 to binary seems a bit painful. It seems to require many steps. However, there's a fairly easy way to convert positive exponents to binary.
First, we take advantage of the following fact: **1000 0000** maps to exponent +1 in excess 127. If this were excess 128, it would map to 0. It would be nice, in fact, if it were excess 128, because then we would write out the positive number in unsigned binary, then flip the most significant bit from 0 to 1, and we'd be done. (Verify this for yourself with an example or two).
However, excess 127 and excess 128 are only off-by-one, so it's not too hard to adjust the algorithm appropriately. Here's what you do to convert positive exponents to excess 127.

- Subtract 1 from the positive exponent.
- Convert the number to unsigned binary, using 8 bits.
- Flip the MSb to 1

For example, we had an exponent of 3 in the previous example. Subtract 1 to get 2, convert to UB to get 0000 0010. Flip the MSb to get 1000 0010. That's the answer from the previous section.
Before you memorize this algorithm, you should really try to understand where it comes from.
This is where it comes from. Consider a positive exponent, **x**, represented in base 10. To convert it to excess 127, we add 127. Thus, we have **x + 127**. We can rewrite this as: **(x - 1) + 128**. This is simple algebra.
128 is 1000 0000 in binary. And we have **x - 1**, which is where the subtraction of 1 occurs. As long as **x - 1** is smaller than 128 (and it will be, since the maximum value of **x** is 128), then it's easy to add this binary number to 1000 0000.
Remember, memorization is a poor second to understanding. It's better to understand why something works than to memorize an answer. However, it's even better to understand why something works and remember it too.

### Converting Denormalized from Base 10 to IEEE 754

Suppose you're asked to convert **1.1 x $2^{-128}$** to IEEE 754 single precision. How would you do this? If you're not careful, you might think the number is normalized, and you might convert this to a normalized number using the procedure from before.
You'd get stuck trying to convert the exponent, because you'd discover the number is negative, and the number has to be non-negative when convert from base 10 (after adding the bias) to UB.
You can save yourself this hassle if you recall that the smallest, positive normalized number has an exponent of -126, and that the exponent we have is -128, which is less than -126. If you've written the number in binary scientific notation (in canonical form), and the exponent is less than -126, then you have a denormalized number.
Since -128 < -126, the number we're trying to represent is a denormalized number.
The rules for representing denormalized numbers is different from representing normalized numbers.
To represent a denormalized number, you need to shift the radix point so that the exponent is -126. In this case, the exponent must be increased by 2 from -128 to -126, so the radix point must shift left by 2.
This results in: **0.011 x $2^{-126}$**
At this point, it's easy to convert. The exponent bitstring is $0^8$. You copy the bits after the radix point into the fraction. The sign bit is 0.

```
S    Exp            Fraction
- --------- --------------------------
0 0000 0000 0110 0000 0000 0000 0000 000
```

**No Unsigned Float**

Unlike ints, there isn't an unsigned float. One reason for this may be the complicated nature of representing floating point numbers. If we get rid of the sign bit, how would we use it? Would we add one more bit to the exponent? That would make the most sense, since it sits adjacent to the exponent, but the bias would have to be changed.
We could add one more bit to the fraction. At least, that would cause the least amount of disruption. Would that one additonal bit help us in any meaninful way? On the one hand, it allows us to represent twice as many floating point numbers. On the other, it does so by adding a single bit of precision.
Perhaps through this kind of reasoning, the developers of the IEEE 754 standard felt that having an unsigned float did not make sense, and thus there is no unsigned float in IEEE 754 floating point.

**Why Sign Bit, Exponent, then Fraction?**

If you look at the representation for IEEE 754, you'll notice that it's sign bit, then exponent, then fraction.
Why do it in that order?
Here's a plausible explanation. Suppose you want to compare two dates. The date includes month, day, and year. You use two digits for the month, two for the day, and four for the year. Suppose you want to store the date as a string, and want to use string comparison to compare dates.
Which order should you pick?
You should pick the year, the month, and the day. Why? Because when you are doing string comparison, you compare left to right, and you want the most significant quantity to the left. That's the year.
When you look at a floating point number, the exponent is the most important, so it's to the left of the fraction.
You can also do comparisons because the exponent is written in bias notation (you *could* use two's complement, as well, although it would make the comparison only a little more complicated).
So why is the sign bit to the far left? Perhaps the answer is because that's where it appears in signed int representation. It may be unusual to have the sign in any other position.

**Summary**

After reading and practicing, you should be able to do the following:

- Give the names of each of the five categories of floating point numbers in IEEE 754 single precision.
- Given a 32 bitstring, determine which category the bitstring falls in.
- Given a normalized or denormalized number, write the number in canonical binary scientific notation (you can leave the exponent written in base 10).
- Given a number in base 10 or canonical binary scientific notation, convert it to an IEEE 754 single precision floating point number.
- Know what bias is used for normalized numbers.
- Know what exponent is used for denormalized numbers.
- Know what the hidden 1 is.

# The IEEE standard for floating point arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

Because many of our users may have occasion to transfer unformatted or "binary" data between an IEEE machine and the Cray or the VAX/VMS, it is worth noting the details of this format for comparison with the Cray and VAX representations. The differences in the formats also affect the accuracy of floating point computations.

## Summary:

### Single Precision

The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F':

```
 S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
 0 1       8 9                     31
```

The value V represented by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If 0<E<255 then V=(-1)**S * 2 ** (E-127) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then V=(-1)**S * 2 ** (-126) * (0.F) These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

In particular,

```
 0 00000000 00000000000000000000000 = 0
```

```
 1 00000000 00000000000000000000000 = -0

 0 11111111 00000000000000000000000 = Infinity
 1 11111111 00000000000000000000000 = -Infinity

 0 11111111 00000100000000000000000 = NaN
 1 11111111 00100010001001010101010 = NaN

 0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
 0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5
 1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5

 0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
 0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
 0 00000000 00000000000000000000001 = +1 * 2**(-126) *
                                      0.00000000000000000000001 =
                                      2**(-149)  (Smallest positive value)
```

### Double Precision

The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F':

```
 S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 0 1         11 12                                                63
```

The value V represented by the word may be determined as follows:

- If E=2047 and F is nonzero, then V=NaN ("Not a number")
- If E=2047 and F is zero and S is 1, then V=-Infinity
- If E=2047 and F is zero and S is 0, then V=Infinity
- If 0<E<2047 then V=(-1)**S * 2 ** (E-1023) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then V=(-1)**S * 2 ** (-1022) * (0.F) These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

## Reference:

*ANSI/IEEE Standard 754-1985,*
*Standard for Binary Floating Point Arithmetic*

## See also:

- Other software installed at PSC.
- Other installed at PSC.
- Other installed at PSC.


300 S. Craig Street, Pittsburgh, PA 15213 Phone: 412.268.4960 Fax: 412.268.5832
**This page last updated**: February 01, 2010

# Survey of Floating-Point Formats

This page gives a very brief summary of floating-point formats that have been used over the years. Most have been implemented in hardware and/or software and used for real work; a few (notably the small ones at the beginning) just for lecture and homework examples. They are listed in order of increasing *range* (a function of exponent size) rather than by precision or chronologically.

| *range* (overflow value) | *precision* | *bits* | *B* | $W_e$ | $W_m$ | *what* |
|---|---|---|---|---|---|---|
| 14 | 0.6 | 6 | 2 | 3 | 2 | Used in university courses[21,22] |
| 240 | 0.9 | 8 | 2 | 4 | 3 | Used in university courses[21,22] |

| | | | | | | |
|---|---|---|---|---|---|---|
| $65504 = 2^{15}\times(2-2^{-10})$ | 3.3 | 16 | 2 | 5 | 10 | IEEE 754-2008 binary16, also called "half", "s10e5", "fp16". 2-byte, excess-15 exponent used in nVidia NV3x and subsequent GPUs[24,25,27,33]; largest minifloat. Can approximate any 16-bit unsigned integer or its reciprocal to 3 decimal places. |
| $9.9999\times10^{8}$ (see note[32]) | 4.8 | 24 | 2 | 7 | 16 | Zuse Z1, the first-ever implementation of binary floating-point[31] |
| $9.999\times10^{9}$ (see note[32]) | 4.8 | 24 | 2 | 7 | 16 | Zuse Z3 |
| $2.81\times10^{14}$ | 3.6 | 18 | 8 | 5 | 12 | excess-15 octal, 4-digit mantissa. A fairly decent radix-8 format in an 18 bit PDP-10 halfword |
| $9.22\times10^{18} = 2^{2^6-1}$ | 4.8 | 24 | 2 | 7 | 16 | 3-byte excess-63 ATI R3x0 and Rv350 GPUs[13]. Also called "s16e7", "fp24". |
| $1.84\times10^{19} = 2^{2^6}$ | 6.9 | 30 | 2 | 7 | 23 | AMD 9511 (1979)[5] |
| $9.90\times10^{27} = 8^{8^2/2-1}$ | 5.1 | 24 | 8 | 6 | 17 | Octal excess-32[12] |
| $1.70\times10^{28} = 2^{2^7-1}$ | 8.1 | 36 | 2 | 8 | 27 | Digital PDP-10[1,18], VAX (F and D formats)[1]; Honeywell 600, 6000[1,16]; Univac 110x single[1]; IBM 709x, 704x[1] |
| $1.70\times10^{38} = 2^{2^7-1}$ | 9.6 | 40 | 2 | 8 | 1+31 | Apple II, Sinclair ZX Spectrum [30], perhaps others |
| $1.70\times10^{38} = 2^{2^7-1}$ | 7.2 | 32 | 2 | 8 | 1+23 | TRS-80 single-precision [35] |
| $1.70\times10^{38} = 2^{2^7-1}$ | 16.8 | 64 | 2 | 8 | 1+55 | TRS-80 double-precision [35] |
| $3.40\times10^{38} = 2^{2^7}$ | 7.2 | 32 | 2 | 8 | 1+23 | IEEE 754 single-precision (or IEE 754-2008 "binary32") (ubiquitous) |
| $3.40\times10^{38} = 2^{2^7}$ | 7.2 | 32 | 2 | 8 | 1+23 | Digital PDP-11[19], PDP 16[6], VAX |
| $9.99\times10^{49} = 10^{10^2/2}$ | 8.0 | 44 | 10 | 2d | 8d | Burroughs B220[7] |
| $4.31\times10^{68} = 8^{76}$ | 11.7 | 47 | 8 | 7 | 39 | Burroughs 5700, 6700, 7700 single[1,14,16,17] |
| $7.24\times10^{75} = 16^{63}$ | 7.2 | 32 | 16 | 7 | 24 | IBM 360, 370[6]; Amdahl 1; DG Eclipse M/600[1] |
| $7.24\times10^{75} = 16^{63}$ | 16.8 | 64 | 16 | 7 | 56 | IBM 360 double[15] |
| $5.79\times10^{76} = 2^{255}$ | 7.2 | ? | 2 | 9 | 24 | Burroughs 1700 single[16] |
| $1.16\times10^{77} = 16^{64}$ | 7.2 | 32 | 16 | 7 | 24 | HP 3000[1] |
| $9.99\times10^{96} = 10^{3\times2^5+1}$ | 7.0 | 32 | 10 | 8- | 7d | IEEE 754r decimal32[3,4] |
| $9.99\times10^{99} = 10^{10^2}$ | 10.0 | ? | 10 | 2d | 10d | Most scientific calculators |
| $4.9\times10^{114} = 8^{127}$ | 12.0 | 48 | 8 | 8 | 40 | Burroughs 7700[6] |
| $9.99\times10^{127} = 100^{64}$ | ~13 | 64 | 100 | 1 | 7 | TI-99/4A computer[36] |
| $8.9\times10^{307} = 2^{2^{10}-1}$ | 14.7 | 60 | 2 | 11 | 1+48 | CDC 6000, 6600[6], 7000 CYBER |
| $8.9\times10^{307} = 2^{2^{10}-1}$ | ? | ? | ? | ? | ? | DEC Vax G format; UNIVAC; 110x double[1] |
| $1.8\times10^{308} = 2^{2^{10}}$ | 15.9 | 64 | 2 | 11 | 1+52 | IEEE 754 double-precision (or IEE 754-2008 "binary64") (nearly ubiquitous) |
| $1.27\times10^{322} = 2^{1070}$ | ? | ? | ? | ? | ? | CDC 6x00, 7x00, Cyber[1] |
| $9.99\times10^{384} = 10^{3\times2^7+1}$ | 16.0 | 64 | 10 | 10- | 16d | IEEE 754r decimal64[3,4] |
| $9.99\times10^{499} = 10^{10^3/2}$ | 12.0 | ? | 10 | 3d | 12d | HP 71B[13], 85[1] calculators |
| $9.99\times10^{999} = 10^{10^3}$ | 12.0 | ? | 10 | 3d | 12d | Texas Instruments 85, 92 calculators |

| | | | | | | |
|---|---|---|---|---|---|---|
| $9.99\times10^{999} = 10^{10_3}$ | 14.0 | ? | 10 | 3d | 14d | Texas Instruments 89 calculator[13] |
| $9.99\times10^{999} = 10^{10_3}$ | 17.0 | 82 | 10 | 3d | 17d | 68881 Packed Decimal Real (3 BCD digits for exponent, 17 for mantissa, and two sign bits) |
| $1.4\times10^{2465} = 2^{2_{13}-3}$ | 7.2 | 38? | 2 | 14 | 24 | Cray C90 half[8] |
| $1.4\times10^{2465} = 2^{2_{13}-3}$ | 14.1 | 61? | 2 | 14 | 47 | Cray C90 single[8] |
| $1.4\times10^{2465} = 2^{2_{13}-3}$ | 28.8 | 110? | 2 | 14 | 96 | Cray C90 double[8] |
| $1.1\times10^{2466} = 2^{2_{13}}$ | ? | ? | ? | ? | ? | Cray I[1] |
| $5.9\times10^{4931} = 2^{2_{14}-1}$ | ? | ? | ? | ? | ? | DEC VAX H format[1] |
| $1.2\times10^{4932} = 2^{2_{14}}$ | 19.2 | 80 | 2 | 15 | 64 | The minimum IEEE 754-1985 double extended size (Pentium; HP/Intel Itanium; Motorola 68040, 68881, 88110) |
| $1.2\times10^{4932} = 2^{2_{14}}$ | 34.0 | 128 | 2 | 15 | 1+112 | IEEE 754-2008 "binary128" aka "quad" or "quadruple"[2,3,28,29,37] (DEC Alpha[9]; IBM S/390 G5[10]) |
| $9.99\times10^{6144} = 10^{3\times2_{11}+1}$ | 34.0 | 128 | 10 | 14- | 34d | IEEE 754-2008 decimal128[3,4,28,29,37,38] |
| $5.2\times10^{9824} = 2^{2_{15}-131}$ | 16.0 | ? | 2 | 16 | 47 | PRIME 50[16] |
| $1.9\times10^{29603} = 8^{2_{15}+12}$ | ? | ? | 8 | 16 | ? | Burroughs 6700, 7700 double[1,16] |
| $4.3\times10^{2525222} = 2^{2_{23}}$ | ? | ? | 2 | 24 | ? | PARI |
| $1.4\times10^{323228010} = 2^{2_{30}-1616}$ | ? | ? | 2 | 31 | ? | *Mathematica®* |
| $\approx10^{2147483646} = 10^{2_{31}-2}$ | ? | ? | ? | ? | ? | *Maple®* |

Legend:

**B** : Base of exponent. This is the amount by which your floating-point number increases if you raise its exponent by 1. Modern formats like IEEE 754 all use base 2, so B is 2, and increasing the exponent field by 1 amounts to multiplying the number by 2. Older formats used base 8, 10 or 16.

$\mathbf{W_e}$ : Width of exponent. If B is 2, 8 or 16, this is the number of bits (binary digits) in the exponent field. For the specific case of B=2, $W_e$ is equal to K+1 in the equation $1-2^K<e<2^K$ specifying the bounds of the excess-2$n$ exponent in an IEEE 754 representation (see below). When B is 10, there are two cases: "6d" indicates an exponent stored as base-10 *digits* and the letter **d** is included to make this clear; "8-" indicates an IEEE binary decimal format, using 2 bits in the combination field and 6 bits in the following exponent field, which together can hold only 3/4 of the values such a width would imply (because the high 2 bits cannot both be 1), thus the legal values are $e$ such that $0\le e<3\times2^6$.

$\mathbf{W_m}$ : Width of mantissa. For binary formats with "hidden" or "implied" leading mantissa bits, this is given as "1+N", such as "1+23", the "1+" refers to the leading 1 bit; this plus 23 actual bits gives a total of 24 bits of precision. For decimal formats the letter "d" is shown to make it clear the precision is in decimal digits.

## IEEE 754 Single Representation

This is worth describing in a bit more detail because it is so prevalent in the hardware used today, and it is probably what you'll be looking at when you try to decipher a floating-point value from its "raw binary".

First a warning: Although the "normal" values are what you see when your program is working with real data, proper handling of the rest of the values (denorms, NANs, etc.) is vitally important; otherwise you'll get all sorts of horrible results that are difficult to understand, and usually impossible to fix.

So, for the normal values (which in this case means, not including the zeros, denorms, NANs, and infinities) the value being represented can be expressed in the following form:

$$\text{value} = s\ 2^{k+1-N}\ n$$

where the sign $s$ is -1 or 1, and $k$ and $n$ are integers that fall within the ranges given by:

$$2-2^K < k < 2^K-1 \quad \text{and} \quad 2^{N-1}-1 < n < 2^N$$

for two integers $K$ and $N$. If you look at the range of $k$ and $n$ you can see that $k$ can have exactly $2^{K+1}-2$ values and $n$ can have exactly $2^{N-1}$ values, and therefore exactly $K+1$ bits can be used to store the exponent (including two unused values discussed below) and $N-1$ bits to store the mantissa. To give a specific example, for IEEE 754 single precision, as the above table shows there are $W_e$=8 bits for the exponent and $W_m$=23 bits for the mantissa, so K is 7 and N is 24.

The exponent is stored in "excess $2^K$ format", which means the binary value you see is $2^K$ bigger than the actual value of $k$ being represented. For example, when $K$ is 7, is the value 254 is seen, $k$ is 126, and the value being represented is $s\ 2^{127-N}\ n$. This is *only* true for the normal values just described, not for denorms.

The next set of values to understand are the *denormalized values* (or "denorms"), very small values for which

$$k = 2-2^K \quad \text{and} \quad 0 < n < 2^{N-1}$$

using the same definitions as above. These values use one of the "unused" exponent values, namely the one that is all 0 bits. They are very important because they make overflow work better: instead of jumping suddenly to 0, you lose precision *gradually* as you go towards 0.

In addition to making the underflow case a little less severe by losing precision gradually instead of suddenly, denormalized values eliminate a lot of strange bugs that would otherwise occur. For example, the tests "`if x>y`" and "`if x-y>0`" can yield different results, unless you use denorms.

All of the various values are arranged in such a way that hardware or software can perform comparisons treating the data as signed-magnitude integers, and as long as neither argument is a NAN the proper answer will result. Such comparisons even properly handle the infinities and negative zero. (A signed-magnitude integer is a sign bit followed by an unsigned expression of its magnitude — this is *not* the normal signed integer format which is called "2's complement signed integer". As with floats, there are ways to express 0 as a signed-magnitude integer.)
Here are some sample values with their binary representation. The binary digits are broken into groups of 4 to help with interpreting a value in hexadecimal. They are shown in order from largest to smallest, with the non-numbers in the places they would fall if they were sorted by their bit patterns.

| s | exponent | mantissa | value(s) |
|---|---|---|---|
| 0 | 111.1111.1 | 111.1111.1111.1111.1111.1111 | Quiet NANs |
| 0 | 111.1111.1 | 100.0000.0000.0000.0000.0000 | Indeterminate |
| 0 | 111.1111.1 | 0xx.xxxx.xxxx.xxxx.xxxx.xxxx | Signaling NANs |
| 0 | 111.1111.1 | 000.0000.0000.0000.0000.0000 | Infinity |
| 0 | 111.1111.0 | 111.1111.1111.1111.1111.1111 | $3.402\times10^{38}$ |
| 0 | 100.0000.1 | 000.0000.0000.0000.0000.0000 | 4.0 |
| 0 | 100.0000.0 | 100.0000.0000.0000.0000.0000 | 3.0 |
| 0 | 100.0000.0 | 000.0000.0000.0000.0000.0000 | 2.0 |
| 0 | 011.1111.1 | 000.0000.0000.0000.0000.0000 | 1.0 |
| 0 | 011.1111.0 | 000.0000.0000.0000.0000.0000 | 0.5 |
| 0 | 000.0000.1 | 000.0000.0000.0000.0000.0000 | $1.175\times10^{-38}$ (Smallest normalized value) |
| 0 | 000.0000.0 | 111.1111.1111.1111.1111.1111 | $1.175\times10^{-38}$ (Largest denormalized value) |
| 0 | 000.0000.0 | 000.0000.0000.0000.0000.0001 | $1.401\times10^{-45}$ (Smallest denormalized value) |
| 0 | 000.0000.0 | 000.0000.0000.0000.0000.0000 | 0 |
| 1 | 000.0000.0 | 000.0000.0000.0000.0000.0000 | -0 |
| 1 | 000.0000.0 | 000.0000.0000.0000.0000.0001 | $-1.401\times10^{-45}$ (Smallest denormalized value) |
| 1 | 000.0000.0 | 111.1111.1111.1111.1111.1111 | $-1.175\times10^{-38}$ (Largest denormalized value) |
| 1 | 000.0000.1 | 000.0000.0000.0000.0000.0000 | $-1.175\times10^{-38}$ (Smallest normalized value) |
| 1 | 011.1111.0 | 000.0000.0000.0000.0000.0000 | -0.5 |
| 1 | 011.1111.1 | 000.0000.0000.0000.0000.0000 | -1.0 |
| 1 | 100.0000.0 | 000.0000.0000.0000.0000.0000 | -2.0 |
| 1 | 100.0000.0 | 100.0000.0000.0000.0000.0000 | -3.0 |
| 1 | 100.0000.1 | 000.0000.0000.0000.0000.0000 | -4.0 |
| 1 | 000.0000.1 | 000.0000.0000.0000.0000.0000 | $-1.175\times10^{-38}$ |
| 1 | 111.1111.0 | 111.1111.1111.1111.1111.1111 | $-3.402\times10^{38}$ |
| 1 | 111.1111.1 | 000.0000.0000.0000.0000.0000 | Negative infinity |
| 1 | 111.1111.1 | 0xx.xxxx.xxxx.xxxx.xxxx.xxxx | Signaling NANs |
| 1 | 111.1111.1 | 100.0000.0000.0000.0000.0000 | Indeterminate |
| 1 | 111.1111.1 | 111.1111.1111.1111.1111.1111 | Quiet NANs |

## IEEE 754d Decimal Formats

The decimal32, decimal64 and decimal128 formats defined in the IEEE 754-2008 standard are interesting largely because of their innovative packing of 3 decimal digits into 10 binary digits. Decimal formats are still useful because they can store decimal fractions (like 0.01) precisely. Normal BCD (binary-coded decimal) uses 4 binary digits for each decimal digit, requiring a waste of about 17% of the information capacity of the bits. The 1000 combinations of 3 decimal digits fit nearly perfectly into the 1024 combinations of 10 binary digits. In addition to the space efficiency, groups of 3 work well for formatting and printing which typically use a thousands separator (such as "," or a blank space) between groups of 3 digits. However, prospects for easy encoding and decoding seem bleak. In 1975 Chen and Ho published the first such system, but it had some drawbacks. The Cowlishaw encoding[4], used in IEEE 754-2008, is remarkable because it manages to achieve all of the following desirable goals:

- The encoding of 000 is all 0's; if the 3 digits are 000-009, the high 6 bits of the encoded result are 0; and if the digits are 010-099 the high 3 bits are 0. Thus you can store 1 digit in 4 bits or 2 digits in 7 bits, making it easy to store any number of decimal digits, not just a multiple of 3; and you can expand any field into a larger field by adding 0's on the left.
- All combinations from 000-079 encode into the same bit pattern as normal BCD.
- You can easily discover if any decimal digit is odd or even by testing a single bit in the binary encoding: test bit 0 (the lowest bit) to see if the low digit is odd; test bit 4 to see if the middle digit is odd and test bit 7 to see if the high digit is odd. These tests always work regardless of the values of the other digits. (As a consequence of this, the hardware implementations for encoding and decoding require no gates for these 3 bits)
- The hardware implementation for encoding 3 decimal digits into 10 binary requires only a total of 33 NAND gates, and decoding back to decimal requires only 54 NAND gates, with a 3-gate-delay in both directions (not including fanout drivers).

- The 24 unused bit patterns are easily characterized as [ddx11x111x] with [dd] equal to 01, 10 or 11.

**Minifloats and Microfloats: Excessively Small Floating-Point Formats**

Although they do not have much practical value as a universal format for computation, very small floating-point formats are of interest for other reasons.

One can refer to a format using 16 bits or less as a ***minifloat***. Of these, the most popular by far is 1.5.10 (or *s10e5* or *binary16*), the 16-bit format invented at nVidia and ILM and now a part of IEEE 754-2008. This format uses 1 sign bit, a 5-bit excess-15 exponent, 10 mantissa bits (with an implied 1 bit) and all the standard IEEE rules including denormals, infinities and NaNs. The minimum and maximum representable values are $2.98\times10^{-8}$ and 65504 respectively.

| *s* | *expon.* | *mantissa* | *value(s)* |
|---|---|---|---|
| 0 | 111.11 | xx.xxxx.xxxx | various NANs |
| 0 | 111.11 | 00.0000.0000 | Infinity |
| 0 | 111.10 | 11.1111.1111 | 65504 (Largest finite value) |
| 0 | 100.11 | 10.1100.0000 | 27.0 |
| 0 | 100.01 | 11.0000.0000 | 7.0 |
| 0 | 100.00 | 10.0000.0000 | 3.0 |
| 0 | 011.11 | 00.0000.0000 | 1.0 |
| 0 | 011.10 | 00.0000.0000 | 0.5 |
| 0 | 000.01 | 00.0000.0000 | $6.104\times10^{-5}$ (Smallest normalized value) |
| 0 | 000.00 | 11.1111.1111 | $6.098\times10^{-5}$ (Largest denormalized value) |
| 0 | 000.00 | 00.0000.0001 | $6\times10^{-8}$ (Smallest denormalized value) |
| 0 | 000.00 | 00.0000.0000 | 0 |
| 1 | 011.11 | 00.0000.0000 | -1.0 (other negative values are analogous) |

This format is supported in hardware by many nVidia graphics cards including GeForce FX and Quadro FX 3D (they call it **fp16** or **s10e5**), and is used by Industrial Light and Magic (as part of their OpenEXR standard) and Pixar as the native format for raw output rendered frames (prior to conversion to a compressed format like DVD, HDTV, or imaging on photographic film for exhibition in a theater). s10e5 is more than sufficient to represent light levels in a rendered image, and compared to 32-bit floating-point, it presents quite a few advantages: it requires half as much memory space (and bus bandwidth); an operation (such as addition or multiplication) takes less than half the time (as measured in gate delay) and about 1/4 as many transistors. All of these advantages are very important when you are expected to perform trillions of operations to render a frame.

To give a concrete example: at the time of the 3-GHz Pentium, which was capable of 12 billion floating-point operations per second (12 GFLOPs), nVidia graphics cards for consumers could manage around 40 billion operations per second. Soon after that, ATI (which uses 24-bit 1.7.16 or *s16e7* format) surpassed that, and the two companies repeatedly leapfrogged each other. In subsequent years, the graphics cards continued to widen their lead over CPUs, and even when 32-bit floating-point became common on graphics cards, 16-bit is still very commonly used typically because it presents a lesser load on membery bandwidth.

The computer-graphics industry has long recognized the value of floating-point to represent pixels, because a pixel expresses (essentially) a light level. Light levels can vary over a very wide range — for example, the ratio between broad daylight and a clear night under a full moon, is 14 "magnitudes" on the scale used by astronomers. That's $2.512^{14} \approx 400{,}000$. The ratio of brightnesses in nighttime environments with bright lights (such as when driving at night, or in a candlelit room) are similar. Such scenes have "high-contrast" lighting. The human eye can handle this range easily. A standard 8-bit format for pixel values (typically 8 bits for each of the three components red, green and blue) doesn't even come close. Doubling the pixel width to 16 bits produces the 48-bit format (common in the industry) but does little to improve the situation for high-contrast lighting — for pixel values near the bottom of the range, roundoff error is terrible. But using 1.5.10 float format increases the range to over $10^9$ (values as small as $6.1\times10^{-5}$ and as large as 65504), with the equivalent of 3 decimal digits of precision over the entire range. It can also represent any integer from -2048 to 2048, so is even useful in some situations like pixel addresses within a texture.

**Microfloats**

A floating-point format using 8 bits or less fits in a byte; I call this a ***microfloat***. These are the best for learning, particularly when you have to convert to/from floating-point using pencil and paper. I am not alone in thinking they are useful as an educational tool for learning about and practising the implementation of floating-point algorithms — I have found courses at no fewer than 11 colleges and universities that use them in lectures.[21],[22],[23]

But surprisingly, such small representations even have use in the real world — sort of. Some encodings used for waveforms and other time-variable analog data are very close to being a floating-point encoding with a small number of exponent and mantissa bits. An example is "mu-law" coding used for audio. Such codes usually store the logarithm of a value plus a sign, and have a special value for zero. This is not the same as a true fbating-point format, but it has a similar range and precision.

The smallest format that has all three fields would be 1.1.1 format — using 3 bits with one bit each for sign, exponent and mantissa. 1.1.1 format encodes the values {-3, -2, -1, -0, 0, 1, 2, 3} or an equivalent set multiplied by a scaling factor. But this isn't very "useful" because you can do a little better just by treating the 3 bits as a signed integer (which gives you the integers -4 through 3).

The smallest formats that are "useful" in the sense of covering a broader range than the same number of bits as a signed integer have at least a 2-bit exponent field. There is always at least 1 mantissa bit anyway (the hidden leading 1, or leading 0 for the denormalized values when the exponent field is 0). The smallest of these is 1.2.0 format — three bits, encoding the values {-4, -2, -1, -0, 0, 1, 2, 4}.

Adding one mantissa bit to get the 4-bit format 1.2.1 gives us a lot more — it encodes the set {-12, -8, -6, -4, -3, -2, -1, -0, 0, 1, 2, 3, 4, 6, 8, 12} giving quite a bit more than the range of the 4-bit signed integer {-8 ... 7}.

5 bits are best used in a 1.2.2 format, using 1 sign bit, 2 exponent bits and 2 mantissa bits (plus an implied leading 1 bit for a mantissa precision of 3 bits). If the exponent is treated as excess -2 (that's "excess minus-two"), all representable values are integers and the range is {-28 .. 28} (or {-24 .. 24} if the highest exponent value is used for infinities). 5 bits as a normal two's complement integer has a range of {-16 .. 15}.

Reader George Spelvin suggests unsigned, all-integer formats, with no infities or NANs. The exponent excess is taken to be whatever value causes the denorms to use the same storage format as the corresponding integer.

Using 0.5.3 format as an example: There is no sign bit, so all values are positive. When the exponent field is 0, the mantissa is denormalized. So the values (in binary) 00000.000 through 00000.111 express the integers 0 through 7. The next exponent value is 00001 in binary, its 1 bit happens to correspond with the implied leading 1 of the (now normalized) mantissa, so values 00001.000 through 00001.111 express integers 8 through 15. Notice how all of these values for the integers 0 through 15 are the same as the normal 8-bit integer representation.

After that, values scale in the normal way: 00010.000 through 00010.111 expresses the even integers 16 through 30 (note that only the first of these corresponds to the integer representation); 00011.000 through 00011.111 are the integer values from 32 through 60; and so on. The highest value is 11111.111 which is $15\times2^{30} = 2^{25-2}\times(2^{3+1}-1) = 16106127360$. Another similar format is 0.4.4, excess -4, which expresses integers from 0 up to $2^{24-2}\times(2^{4+1}-1) = 507904$.

In general, using $E$ exponent bits and $M$ mantissa bits, you can express all integers from 0 to $2^{M+1}$, and various higher values up to $2^{2^E-2}\times(2^{M+1}-1)$.

Here is a table presenting most of the smaller entries from the main table in a somewhat different format, along with the integer-only formats that bias the exponent so that the smallest denorm is 1.

| *s.e.m* | *excess* | *range* | *comments* |
|---|---|---|---|
| 1.1.1 | 0 | 1 to 3 | Less range than signed-magnitude integer |
| 1.2.0 | 0 | 1 to 4 | The smallest format whose range exceeds that of the same number of bits interpreted as a signed-magnitude integer |
| 1.2.1 | 0 | 1 to 12 | Best use of 4 bits |
| 1.2.2 | -2 | 1 to 28 | Using no infinity values; range is 1 to 24 if the biggest values are used for infinities |
| 1.3.2 | 3 | 0.0625 to 14 | Used in university courses[21,22] |
| 0.4.4 | -4 | 1 to 507904 | George Spelvin[34] |
| 0.5.3 | -8 | 1 to 16106127360 | George Spelvin[34] |
| 1.4.3 | -3 | 1 to 229376 | About the best compromise for a 1-byte format |
| 1.4.3 | 7 | 0.002 to 240 | Used in university courses[21,22] |
| 1.4.7 | -7 | 1 to 4161536 | One option for 12 bits |
| 1.5.6 | -6 | 1 to $1.35\times10^{11}$ | Another option for 12 bits |
| 1.5.10 | -10 | 1 to $2.20\times10^{12}$ | Largest unbalanced format; range exceeds 32-bit unsigned |
| 1.5.10 | 15 | 0.000061 to 65504 | IEEE 754-2008 binary16, also called "half", "s10e5", "fp16". 2-byte, excess-15 exponent[24,25,27,33]. Can approximate any 16-bit unsigned integer or its reciprocal to 3 decimal places. |
| 1.5.12 | 15 | 1/M to $2.81\times10^{14}$ | A fairly decent radix-8 format in an 18 bit PDP-10 word |
| 1.7.16 | 63 | 1/M to $9.22\times10^{18}$ | 3-byte excess-63[17], aka "fp24", "s16e7" |

**Footnotes**

**1 :** http://http.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf W. Kahan, "Why do we need a floating-point arithmetic standard?", 1981

**2 :** http://http.cs.berkeley.edu/~wkahan/ieee754status/Names.pdf W. Kahan, "Names for Standardized Floating-Point Formats", 2002 (work in progress)

**3 :** http://754r.ucbtest.org/ "Some Proposals for Revising ANSI/IEEE Std 754-1985"

**4 :** http://www2.hursley.ibm.com/decimal/DPDecimal.html "A Summary of Densely Packed Decimal encoding" (web page)

**5 :** http://www3.sk.sympatico.ca/jbayko/cpu1.html

**6 :** http://twins.pmf.ukim.edu.mk/predava/DSM/procesor/float.htm

**7 :** http://www.cc.gatech.edu/gvu/people/randy.carpenter/folklore/v5n2.html

**8 :** http://www.usm.uni-muenchen.de/people/puls/f77to90/cray.html

**9 :** http://www.usm.uni-muenchen.de/people/puls/f77to90/alpha.html

**10 :** http://www.research.ibm.com/journal/rd/435/schwarz.html and http://www.research.ibm.com/journal/rd/435/schwa1.gif

**11 :** http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html

**12 :** One source gave $8^{31}$ as the range for the Burroughs B5500. (I forgot to save my source for this. I have sources for other Burroughs systems, giving $8^{76}$ as the highest value (and $8^{-50}$ as the lowest, for a field width of 7 bits). I might have inferred it from http://www.cs.science.cmu.ac.th/panutson/433.htm which only gives a field width of 6 bits, and no bias. The Burroughs 5000 manual says the mantissa is 39 bits, but does not talk about exponent range. Did some models have a 6-bit exponent field? Since these are the folks who simplified things by storing all integers as floating-point numbers with an exponent of 0 [17], I suspect anything is possible.

**13 :** http://www2.hursley.ibm.com/decimal/IEEE-cowlishaw-arith16.pdf Michael F. Cowlishaw, "Decimal Floating-Point: Algorism for Computers", Proceedings of the 16th IEEE Symposium on Computer Arithmetic, 2003; ISSN 1063-6889/03

**14 :** http://research.microsoft.com/users/GBell/Computer_Structures_Principles_and_Examples/csp0146.htm D. Siewiorek, C. Gordon Bell and Allen Newell, "Computer Structures: Principles and Examples", 1982, p. 130

**15 :** http://research.microsoft.com/~gbell/Computer_Structures__Readings_and_Examples/00000612.htm C. Gordon Bell and Allen Newell, "Computer Structures: Readings and Examples", 1971, p. 592.

**16 :** http://www.netlib.org/slap/slapqc.tgz FORTRAN-90 implementation of a linear algebra package, including a file (`mach.f`) which curiously begins with a table of machine floating-point register parameters for lots of old mainframes. See also the NETLIB D1MACH fiunction, which gives similar values for many systems. (formerly at `http://www.csit.fsu.edu/~burkardt/f_src/slap/slap.f90` and `http://interval.louisiana.edu/pub/interval_math/Fortran_90_software/d1i1mach.for` respectively)

**17 :** http://grouper.ieee.org/groups/754/meeting-minutes/02-04-18.html Includes this brief description of the key design feature of the Burroughs B5500: "ints and floats with the same value have the same strings in registers and memory. The octal point at the right, zero exponent." This shows why the exponent range is quoted as $8^{-50}$ (or $8^{-51}$) to $8^{76}$: The exponent ranged from $8^{-63}$ to $8^{63}$, and the (for floating-point, always normalized) 13-digit mantissa held any value from $8^{12}$ up to nearly $8^{13}$, shifting both ends of the range up by that amount.

**18 :** http://www.inwap.com/pdp10/hbaker/pdp-10/Floating-Point.html

**19 :** http://nssdc.gsfc.nasa.gov/nssdc/formats/PDP-11.htm

**20 :** This format would be easy to implement on an 8-bit microprocessor. It has the sign and exponent in one byte, and a 16-bit mantissa and an explicit leading 1 bit (if the leading 1 is hidden/implied, we get twice the range). With only 4-5 decimal digits it isn't too useful, but it's what you could expect to see on a really small early home computer.

**21 :** http://turing.cs.plymouth.edu/~wjt/Architecture/CS-APP/L05-FloatingPoint.pdf This lecture presentation (or a variation of it) appears at clarkson.edu, plymouth.edu, sc.edu, ucar.edu, umd.edu, umn.edu, utah.edu, utexas.edu and vancouver.wsu.edu. Good discussion of floating-point representations, subnormals, rounding modes and various other issues. Pages 14-16 use the 1.4.3 microfloat format as an example to illustrate in a very concrete way how the subnormals, normals and NANs are related; pages 17-18 use the even smaller 1.3.2 format to show the range of representable values on a number line. Make sure to see page 30 — this alone is worth the effort of downloading and viewing the document!

**22 :** http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f98/H3/H3.pdf Homework assignment that uses the microfloat formats 1.4.3 and 1.3.2. Another similar one is here.

**23 :** http://www.arl.wustl.edu/~lockwood/class/cse306-s04/lecture/l11.html Lecture notes that use the 1.3.5 minifloat format for in-class examples.

**24 :** http://developer.nvidia.com/docs/IO/8230/D3DTutorial1_Shaders.ppt nVidia presentation describing their fp16 format (starting on slide 75).
**25 :** http://developer.nvidia.com/attach/6655 nVidia language specification including definition of fp16 format (page 175).
**26 :** http://www.cs.unc.edu/Events/Conferences/GP2/slides/hanrahan.pdf describes the nVidia GeForce 6800 and ATI Radeon 9800 graphics cards as general-purpose pipelined vector floating-point processors, and shows a rough design for a supercomputer employing 16384 of the GPU chips to achieve a theoretical throughput of 2 petaflops ($2\times10^{15}$ floating-point operations per second). The rackmount pictured is described here.
**27 :** http://www.digit-life.com/articles2/ps-precision/ This is the only source I have found that describes all of the current hardware standard formats, from IEEE binary128 all the way down to nVIDIA s10e5.
**28 :** Wikipedia, IEEE 754 revision. Good summary of the development of IEEE 754-2008.
**29 :** http://grouper.ieee.org/groups/754/revision.html Official status of the IEEE working group responsible for 754r.
**30 :** Steven Vickers, editied by Robin Bradbeer "ZX Spectrum Basic Programming" 2nd edition 1983, Sinclair Research, pp 169-170. (via Lennart Benschop)
**31 :** http://en.wikipedia.org/wiki/Floating_point Wikipedia, *Floating point* (encyclopedia article). While it's possible the idea of floating-point might have been devised for use in mechanical calculators, Konrad Zuse had formulated the ideas behind his model Z3 before building the Z1, and the Z3 is generally regarded as the first generally-programmable computer (more on that topic here),
**32 :** http://www.epemag.com/zuse/part3c.htm Horst Zuse, *The Life and Work of Konrad Zuse*. The Zuse Z1 took numeric input from the operator in *decimal* form, and then converted it to binary. For output, binary was converted back to decimal. The input and output devides both used 5 decimal digits and an exponent ranging from $10^{-8}$ to $10^{8}$. However, the internal representation had 7 binary digits of exponent, so the range for intermediate calculations was somewhat larger — perhaps $2^{63}$ or $2^{64}$. Zuse Z3 was similar, but had 4 or 5 digits and exponent ranges of -9 to 9 (for input) and -13 to +12 (for output).
**33 :** http://www.gpgpu.org/s2004/slides/buck.StrategiesAndTricks.ppt Ian Buck, *GPU Computation Strategies & Tricks*, PowerPoint slides. Slide 4 describes the ATI and nVidia floating-point formats at the time (2004).
**34 :** George Spelvin, email correspondence.
**35 :** http://www.trs-80.com/trs80-zaps-internals.htm The TRS-80 passes 4-byte single-precision (and with Level II BASIC, 8-byte double-precision) values into and out of its ROM routines, and it is clear that one byte is an exponent. The exponent is often described as being in excess-128 (or "XS128") format. However, as reported by reader Ulrich Müller, emulators show that the range is $2^{127}$, and that the internal representation actually uses excess-129.
**36 :** Joe Zbiciak, email correspondence. The TI 99/4A uses radix 100 in an 8-byte storage format. 7 bytes are base-100 mantissa "digits" (equivalent to 14 decimal digits), and the exponent (a value from -64 to 63) is stored in the 8th byte along with a sign bit. The exponent is treated as a power of 100. The largest-magnitude values are $\pm99.999999999999\times100^{63}$, and the smallest-magnitude values (apart from 0) are $\pm1\times100^{-64}$. Precision varies from just over 12 decimal digits to just under 14: for example, $\pi/3$ is $01.047197551197\times100^{0}$ and $3/\pi$ is $0.95492965855137$ (represented as $95.492965855137\times100^{-1}$).
**37 :** Wikipedia, IEEE 754-2008.
**38 :** Wikipedia, Decimal128 floating-point format.

**Other Sources**

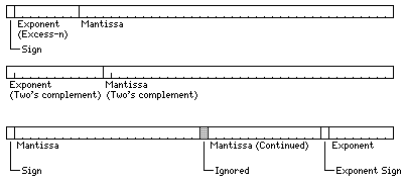
Lennart Benschop
datapeak.net Computer History (a long timeline of events in reverse order, with many pictures)

See also:
Alternative Number Formats
F107 and F161



# Floating-Point Formats

The examples of floating-point numbers shown on the previous page illustrated the most common general type of floating-point format, the one shown in the first line of the diagram below:




The format shown in the first line begins with a single sign bit, which is 0 if the number is positive, and 1 if the number is negative. Next is the exponent. If the exponent is eight bits long, as shown in the diagram, it is in excess-128 notation, so that the smallest exponent value, 00000000, stands for -128, and the largest exponent value, 11111111, stands for 127. Finally, we find the mantissa, which is an unsigned binary fraction.

If the mantissa is normalized, non-negative floating-point numbers can be compared by the same instructions as are used to compare integers.

This format is particularly popular on computers that have hardware support for floating-point numbers. A number of variations on this format are used.

Of course, the length of the exponent field shown in the diagram is only one possibility.

The second line in the diagram illustrates the kind of floating-point format used on computers such as the PDP-8 and the RECOMP II. Here, a floating-point number is simply represented by two signed binary numbers, the first, being the exponent, treated as an integer, and the second, being the mantissa, treated as a fraction, both represented in the ordinary format for signed fixed-point numbers used on the computer.

The third line of the diagram illustrates a kind of format which, with a number of variations, was found on most computers with a 24-bit word length. Computers with a 48-bit word length, on the other hand, typically had hardware floating-point, and used a floating-point format of the type given in the first line.

Why did these computers use such an unusual floating-point format?

Typically, although these computers did not have hardware floating-point support, the way bigger computers with a 32-bit, 36-bit, 48-bit, or 64-bit word length did, they did come standard with hardware integer multiplication, unlike smaller

computers with an 18-bit, 16-bit, or 12-bit word length.

In order to support floating-point arithmetic, the format of double-precision fixed-point numbers on most of these computers omitted the first bit of the second word of the number from the number itself, sometimes treating it as a second copy of the sign, so that fixed-point numbers could be treated as having the binary point on the right, making them integers, or on the left, after the sign, making them fractions on the interval [0,1), without having to adjust them by shifting them one place to the left after a multiplication.

A number of variations of each type of floating-point format exist, of course. When a floating-point format consists of a mantissa field followed by an exponent field, with no omitted bit, the choice of whether to consider the floating-point format as belonging to Group II or Group III is arbitrary.

This page, as currently constructed, has placed all the formats with no omitted mantissa bits in Group II. My current thinking on this issue is now favoring moving those formats where the exponent is part of a partial word within a multi-word format (that is, among those formats on this page, that of the Hewlett-Packard 2114/2115/2116, and that of the SDS 920/930/940/9300 and the SCC 600) in with Group III.

A Group I format, as noted, tends to be used on an architecture for which the initial implementations supported floating-point in hardware. A Group III format tends to be used where the word length is too long to use a full word for the exponent, and hardware multiplication features make it useful to align the beginning of the mantissa with the beginning of a word. The Group II format is most popular with machines with small word lengths and limited hardware arithmetic, but it was also used, as in the case of the Maniac II and the Philco 2000, with architectures that started out with hardware floating-point as well.

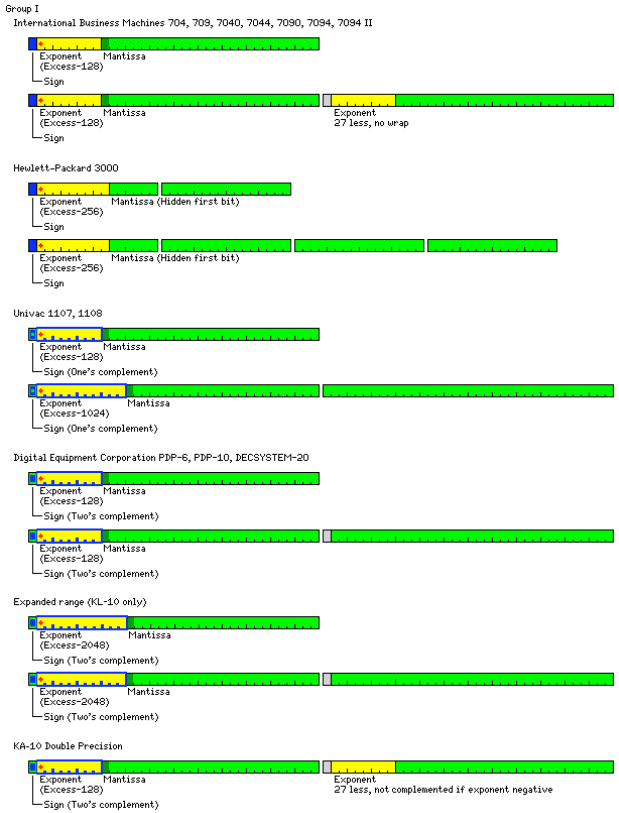
Also, in some cases, the floating point formats of different sizes for some machines belonged to different groups. When this happened, all the formats for any one architecture were placed within the discussion of one of the groups of formats included. Particularly unsusual formats discussed below include the single precision floating-point format for the PDP-4, 7, 9 and 15, which can be thought of as a rearranged Group II or Group III format, and the double precision floating-point format of the ICL 1900, which applies the principle of making a double-precision float out of two single-precision floats, usually used with hardware Group I formats, to a base floating-point format which belongs to Group III; this appears to be the result of either successive implementations of the architecture evolving from hardware multiply to full hardware floating-point or the availability of hardware floating-point as an option.

Another case where the same machine had floating point formats belonging to different groups is the Harris 800, which added a Group II quad-precision floating-point format to an existing architecture whose single and double precision floating-point formats belonged to Group III: here, the hardware level seems to have remained constant, and what happened was that the larger size of the quad-precision format made it reasonable to use a full word instead of a partial word for the exponent, and once that was done, a Group II format appeared more reasonable than a Group III format.

The Univac 418 single-precision floating-point format belongs to Group I, and its double-precision format to Group II. Hardware floating-point was only introduced to this architecture with the Univac 418-III, and so this combination is not too surprising.

### Group I Floating-Point Formats

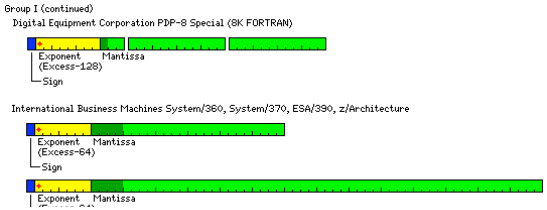
The diagram below shows several examples of floating-point formats of the first type, but they are only a very small sampling of the number of formats of this type that have been used.



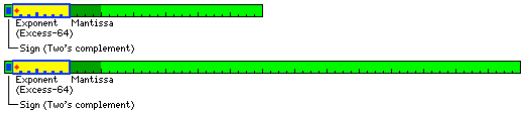

The IBM System/360 computer used an exponent that was seven bits long, in excess-64 notation, that represented a power of 16 instead of a power of two. Thus, a mantissa was normalized when any of the first four bits of the mantissa was not zero. In addition to the computers that were built to be compatible with the System/360 (including the RCA Spectra/70, which was compatible only for user programs), many other computers followed the same floating point format as the System/360, such as the Texas Instruments 990/12, the Data General Nova and Eclipse computers, and the SEL 32.

Another similar computer that I might have expected to use the System/360 format instead originally used a format very similar to that of the PDP-11, except for allocating one more bit to the exponent, and one less bit to the mantissa. During the lifetime of this particular system, a change was made to the IEEE-754 standard format. The system of which I speak is the Hewlett-Packard 3000 series of computers.

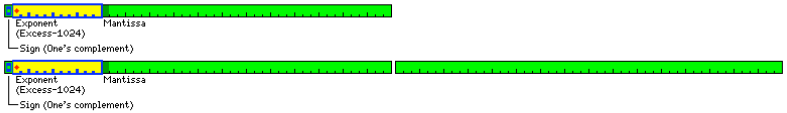
The PDP-10 (and its compatible relatives, the PDP-6 and the DECSYSTEM-20) and the Xerox Sigma computers (which, like the System/360, also used a hexadecimal exponent), which both used two's complement notation for integers, performed a two's complement on the combined exponent and mantissa fields of a floating-point number when it was negative. This meant that all normalized floating-point numbers, whether they were positive or negative, could be compared by integer compare instructions, producing correct results.




Xerox Data Systems Sigma

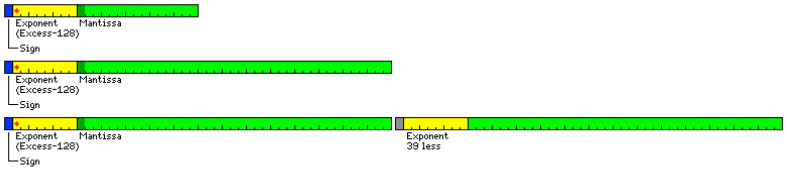



Control Data Corporation 1604, 3600

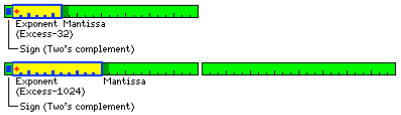



Control Data Corporation 6600


English Electric KDF9



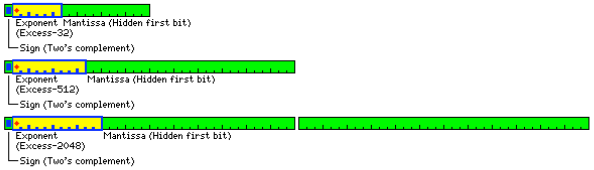

Foxboro FOX-1



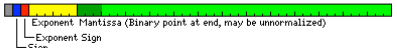

Packard-Bell PB440


Univac 418


Stanford S-1




Burroughs B5500, B6700



The Control Data 1604 computer used an exponent field that was 11 bits long; also, it used one's complement notation for integers, and the mantissa (called the *coefficient* in that computer's manuals) of floating-point numbers was also complemented for negative numbers. Double-precision floating-point numbers, processed in hardware on the later 3400 and 3600, simply added another full 48 bits to the mantissa. This same representation of negative numbers was used on the Control Data 6600, but for a 60-bit floating-point number, again with an 11-bit exponent.

The diagram depicts the exponent as being in excess-1024 notation; actually, that is not quite accurate. Because of its use of one's complement notation for integers, to use the same type of circuitry for arithmetic on exponents, zero and positive exponents were represented in excess-1024 notation, but negative exponents were represented in excess-1023 notation. Thus, on the Control Data 1604, the exponent value of octal 1777 was not used. On the Control Data 6600, the exponent value of 3777 octal represented an overflow, the exponent value of 0000 octal represented an underflow, and the exponent value of 1777 octal represented an indeterminate quantity. The exponent here is shown as in excess-976 notation, with the binary point located at the beginning of the mantissa field as with the other formats shown here, since it was considered to be in excess-1024 (or excess-1023) notation, but with the binary point at the end of the mantissa, which was considered to be an integer.

Of course, this same floating-point format was used on the compatible successors of the Control Data 1604, such as the Control Data 3600. It was also used on the 24-bit members of that line of computers, such as the Control Data 3300. While I have no definite source for the floating-point format used in software for the Control Data 924, it is very likely to have used the same format, since not only is its instruction format the same as that of the Control Data 1604, but corresponding instructions have the same opcode. Despite the Control Data 3300 and similar machines having indirect addressing added, and fewer index registers, apparently it was possible to write lowest-common-denominator software which could be used by all three types of machine.

The AN/FSQ-32 computer, built by IBM, had a 48-bit floating-point format which could be represented by the same diagram as that used for the floating-point format of the Control Data 1604, but it lacked the above-described peculiarities of that computer's floating-point format, and simply used excess-1024 notation consistently for the exponent.

The Cray-1, on the other hand, had a sign bit, 15 bits of excess-16,384 exponent, and 48 bits of mantissa using the more common sign-magnitude format for floating-point numbers.

The AN/FSQ-31 and 32, with a 48-bit word, used 11 bits for the exponent and 37 bits for the mantissa.

The Burroughs 5500, 6700, and related computers used an exponent which was a power of eight. The internal format of a single-precision floating-point number consisted of one unused bit, followed by the sign of the number, then the sign of the exponent, then a six-bit exponent, then 39-bit mantissa. The bias of the exponent was such that it could be considered to be in excess-32 notation as long as the mantissa was considered to be a binary integer instead of a binary fraction. This allowed integers to also be interpreted as unnormalized floating-point numbers.

A double-precision floating-point number had a somewhat complicated format. The first word had the same format as a single-precision floating-point number; the second word consisted of nine additional exponent bits, followed by 39 additional mantissa bits; in both cases, these were appended to the bits in the first word as being the most significant bits of the number.

The BRLESC computer, with a 68-bit word length, used a base-16 exponent; it remainined within the bounds of convention, as the word included a three-bit tag field, followed by a one-bit sign; then, for a floating-point number, 56 bits of mantissa followed by 8 bits of exponent. Thus, the 68-bit word contained 65 data bits and three tag bits, while the whole 68-bit word was used for an instruction. (In addition, four parity bits accompanying each word were usually mentioned.)

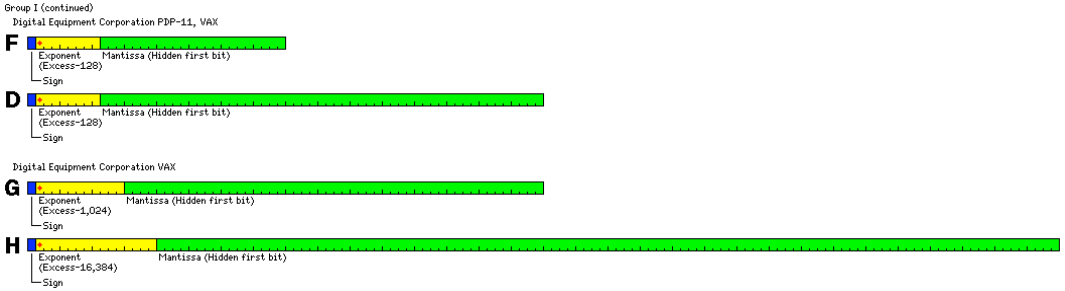
The historic English Electric KDF9 computer used a floating-point format very similar to that of the IBM 7090 computer, except for being adapted to its 48-bit word length. Mentioned in the advertising brochure for the machine, but apparently unknown to its users, it included a 24-bit floating-point format among its data types.

The Foxboro FOX-1 computer, with a 24-bit word length, used a floating-point format belonging to Group I; single-precision floating-point numbers occupied only 24 bits, and consisted of the sign followed by a six-bit excess-32 exponent and 17 bits of mantissa. A negative number was the two's complement of the corresponding positive number, as with the PDP-10 and the Sigma. Double-precision floating-point numbers occupied 48 bits, and consisted of the sign followed by a twelve-bit excess-2,048 exponent and 35 bits of mantissa. Also, the CDC 924 computer, which also did not have hardware floating-point support, when it multiplied two 24-bit integers, it produced a conventional 48-bit integer result with no omitted bits. It used one's complement notation for 24-bit integers, but 15-bit values used in the index registers were in two's complement form. It is likely, therefore, that its sofware floating-point format was compatible with the conventional floating-point format used by the CDC 1604.

The Packard-Bell 440 computer was microprogrammable, but its design was optimized for the floating-point format shown here, which belongs to Group I, although, as is the case for some other Group I formats, it includes an omitted sign bit in the second word.

The Univac 418 computer used a Group I floating-point format for single-precision numbers, but its format for double-precision numbers belonged to Group II. Note how some bits of the first word were ignored because 15 bits was felt to be more than adequate for an exponent field, and that the exponent was complemented for negative numbers only for the single-precision Group I format.

The S-1 computer, built at Stanford University, is noteworthy for having a floating-point format even shorter than that of the Foxboro FOX-1 computer. While 16-bit floating-point formats are used today with graphics cards, floating-point formats shorter than 32 bits were seldom used with general-purpose computers. Note, though, that it used a floating-point format with a hidden first bit, like that of the PDP-11; the extra bit of precision this provided would have been particularly beneficial with this format.



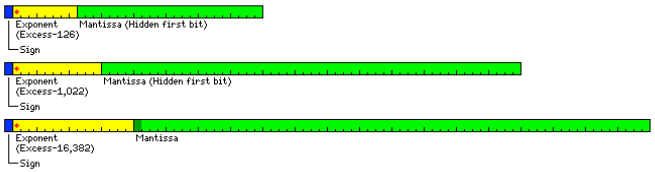

Other computers, such as the PDP-11, and its successor, the VAX, dealt with the wastefulness of having the first bit of the mantissa (almost) always one by omitting that bit from the representation of a number. The number zero was still represented by a zero mantissa combined with the lowest possible exponent value; thus, this exponent value had to be given the additional meaning that the hidden one bit was not present. The diagram above shows only one of the formats used with the PDP-11, although in single precision it was called the F format, and in double precision, the D format. An early sofware format, belonging to Group II, also existed; it involved a 16-bit two's complement exponent followed by a 32-bit two's complement mantissa; unlike H format, it did not have a hidden first bit, and the sign of the mantissa was within the mantissa.

On the VAX and on the Alpha; other formats were used, including the G format, which had an exponent field that was 11 bits in length, used in a 64-bit floating-point number, and which led to the expanded range format for the PDP-10 which is shown above, and the H format, which had an exponent field 15 bits in length, and which was used in a 128-bit floating-point number; these formats are also shown above. Note that the hidden first bit was retained even in the 128-bit format, unlike the case for IEEE-754.

Of course, the Alpha now also supports the standard IEEE-754 floating-point format, which is described here as the "Standard" floating-point format. In its documentation, the 32-bit format is referred to as S format, and the 64-bit format is referred to as T format. In other documentation, the term X format is applied to the new 128-bit format added to IEEE 754. Of the old VAX formats, Alpha chips support arithmetic in F format and G format, and can convert to and from the D format.

The current standard floating-point representation used in today's microcomputers, as specified by the IEEE 754 standard, is based on that of the PDP-11, but in addition also allows gradual underflow as well. This is achieved by making the lowest possible exponent value special in two ways: it indicates no hidden one bit is present, and in addition the value represented by the floating-point number is formed by multiplying the mantissa by the power of two that the next lower exponent value also indicates. It is therefore considerably more complicated than the way in which floating-point numbers were typically represented on older computers.

Of course, the current official floating-point standard, IEEE-754, involves a format that also belongs to Group I:




As it was already depicted on a page in my discussion of an imaginary computer architecture, I had been hesitant to depict it here.

The single and double precision formats both have the suppressed first bit introduced on the PDP-11 and also taken up by the HP 3000. The lengths of the exponent fields match those of the Univac 1107 and its successors.

Essentially, one can trace the lineage of the IEEE 754 floating-point format back to the IBM 704. That floating-point format was continued with the 7090, and both the Univac 1107 mainframe and the DEC PDP-10 computer chose to be largely, but not completely, compatible with it.

The Univac 1107 added a longer exponent part for double-precision numbers, which went directly to the IEEE 754 format.

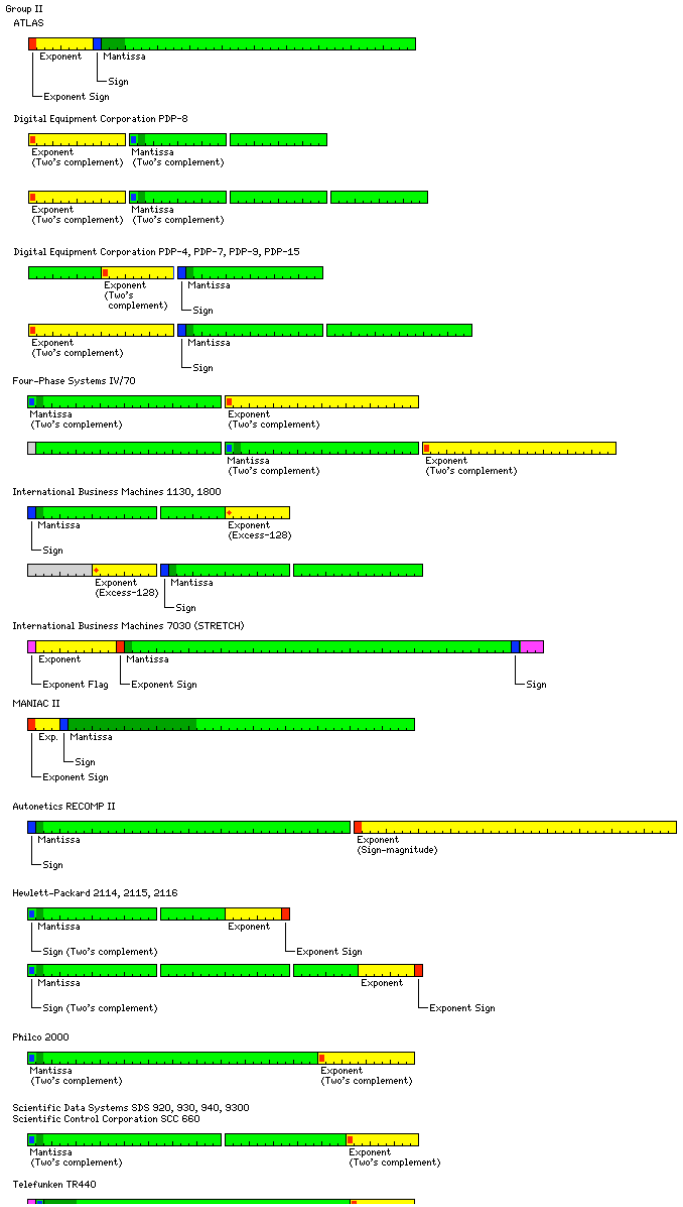
The PDP-11 computer, with a 16-bit word, was given a floating-point format with the same size of exponent field as the PDP-10 for compatibility. Of course, with a 16-bit word, the lengths of floating-point numbers were now 32 and 64 bits, not 36 and 72 bits. This loss of precision was perhaps what encouraged the designers of PDP-11 floating-point to come up with the hidden first bit, also incorporated into the IEEE 754 standard, and one of its features considered unusual at the time.

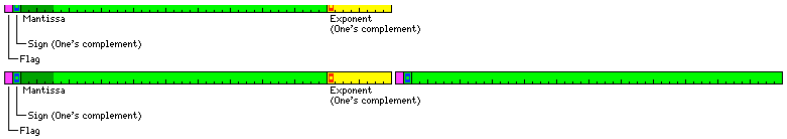
Note that in the 80-bit "temporary real" format, the hidden first bit is not used; this is not simply because the extra precision is not needed in a longer format, but also because this format is intended for *internal* use in implementations of the standard, and thus every efficiency is needed.

Not shown is the new 128-bit format, similar to temporary real, except having additional precision to occupy a full 128-bit storage area.

### Group II Floating-Point Formats

Some of the formats of the type given in the second line of the diagram at the top of the page are illustrated below:






The Manchester ATLAS computer, notable for introducing virtual memory, used an 8-bit sign-magnitude exponent followed by a 40-bit sign-magnitude mantissa. The exponent was a power of eight. A power-of-eight exponent was also used on the Burroughs 5500; thus, a claim I once read that a power-of-eight exponent did not, in practice, lead to the type of problems encountered with the power-of-sixteen exponent on the IBM System/360 could have had practical experience behind it.

The floating-point hardware optionally available for the PDP-8, called the Floating Point Processor-12, as it was originally introduced as an option for the PDP-12 (an updated version of the LINC-8), and a set of floating-point routines for the PDP-8 available as a separate product, both used a single 12-bit word for the exponent, and multiple 12-bit words to represent the mantissa. Other floating-point representations also were used in software on the PDP-8, however; for example, 8K FORTRAN used a format which began with one bit for the sign of the number, followed by an eight-bit signed exponent, with the first three bits of the mantissa completing the first word; this format belonged to the class illustrated by the first line of the diagram above, and was used in order to provide compatibility with the PDP-10 and/or the IBM 7090. However, 4K FORTRAN for the PDP-8 used the same floating-point format as the Floating Point Processor and the Floating Point Package.

Double-precision floating-point numbers on the PDP-4, 7, 9 and 15 were represented by one 18-bit word for the exponent, and two 18-bit words containing the mantissa; the format of single-precision floating-point numbers on those machines was more complicated, and therefore of a form which does not fully belong to any of the three groups examined here, but which allowed quick conversion to the double-precision floating-point format by first appending a copy of the first word as the third word, and then performing masking and, in the case of the exponent, sign extension.

The Four-Phase Systems IV/70 computer was marketed as a system for database applications, and thus floating-point capabilities were a secondary consideration. Thus, a full 24-bit word is used for the exponent, In addition, not only were single-precision floating-point numbers required to be aligned on 48-bit boundaries, but the last 48-bits of a double-precision floating-point number, which had the form of a single-precision float, to which a 24-bit word containing an additional 23 mantissa bits were prepended, also had to be so aligned. This meant that an array of double-precision floating-point numbers would have to contain at least one additional word of storage between successive array elements. This unusual state of affairs can be explained, however, by the fact that only single-precision floating-point aritmetic was directly implemented in hardware on the system.

The IBM 1130 computer did not have hardware floating-point, but its FORTRAN compiler used the floating-point formats shown here. The compiler, for Basic Fortran IV, did not support double-precision; instead, when compiling a FORTRAN program, you could simply request extended precision, so that instead of the default two-word format being used, the second, three-word format would be used instead for all the floating-point numbers in the program. While the IBM 1130 was a 16-bit computer, it did have instructions for 32-bit arithmetic, and so, in order to avoid making floating-point operations unreasonably slow for little benefit, eight bits were left unused instead of extending the precision to 39 bits for the three-word format.

In the IBM 7030 or STRETCH computer, an exponent flag bit was followed by a ten bit exponent, the exponent sign, a 48-bit mantissa, the mantissa sign, and three additional flag bits. The exponent flag was used to indicate that an overflow or underflow had occurred; the other flag bits could simply be set by the programmer to label numbers.

The SDS 930, 940, and 9300 computers used two's complement representation for integers. When they performed a fixed-point multiply, the product was a 48-bit fraction; the most significant bit of the second word was not skipped. The SDS 9300 had a hardware floating-point unit, and, thus, its manual described a hardware floating-point format consisting of a 39-bit two's complement mantissa followed by a 9-bit two's complement exponent. Therefore, although the floating-point format did belong to this general class, by virtue of having the exponent at the end rather than the beginning, it did not include a skipped bit in floating-point numbers. This is also the floating-point format supported by software for the 930 and 940 as well, as noted in Bell and Newell.

The Scientific Control Corporation 660 computer used two's complement notation for integers, and also produced a 48-bit fraction when it multiplied two 24-bit fixed-point numbers, and its floating-point format also consisted of a 39-bit two's complement mantissa followed by a 9-bit two's complement exponent.

It may also be noted that the MANIAC II computer used a floating-point format where the exponent was a power of 65,536. This reduced the number of shifts required, which was very important on a very early vacuum-tube computer, although the maximum possible loss of precision was rather drastic on a machine with a 48-bit word length. But the machine performed floating-point arithmetic only, and it used only a four-bit field for the exponent and its sign; thus, the intent behind its floating-point format can be considered to be one of using a format that is halfway between conventional floating-point format and integer format, so as to obtain the extended range of the former with the speed of the latter.

The RECOMP II, a drum-based computer with a 40-bit word length, simply used one 40-bit word for the exponent, and one 40-bit word for the mantissa. (Incidentally, it used sign-magnitude notation for numbers, not two's complement.) While this was obviously done merely to simplify the design of the computer, advertisements (appearing, for example, in *Scientific American* during the late 1950s) extolled the ability of this computer to handle numbers which, if written down, would girdle the entire globe.

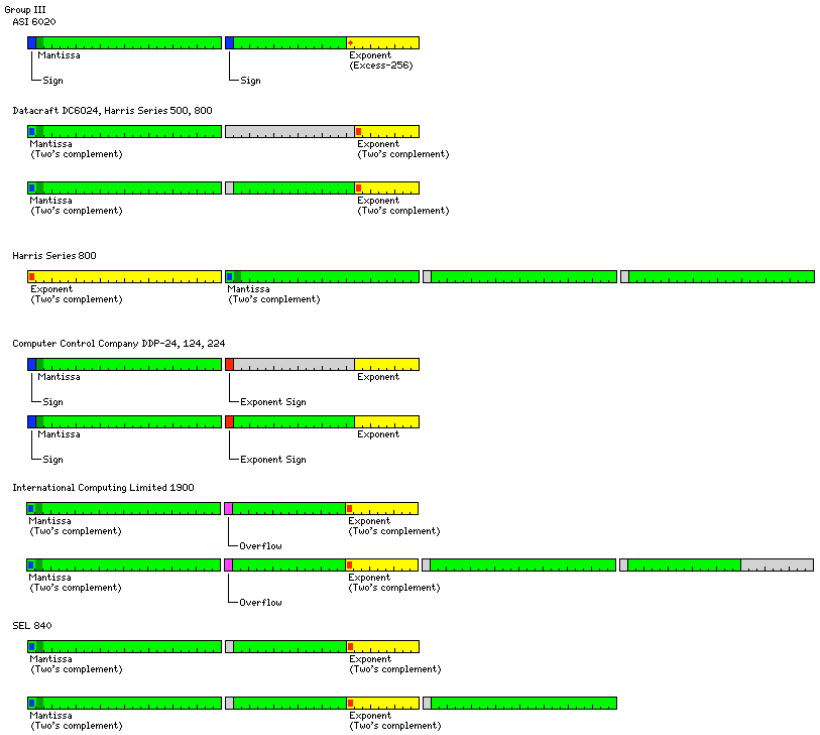
This diagram does not show all the formats of this type that were in use; the Paper Tape Software Package for the PDP-11 included a Math Package with floating-point routines that worked on a format consisting of a 16-bit two's complement exponent followed by a 32-bit two's complement mantissa.

Another floating-point format that could be considered as belonging to this class (although I now tend to incline to placing it in Group III) was used with the Hewlett-Packard 2114/5/6 computers. A floating-point number began with a two's complement mantissa, and then ended with seven bits of exponent, *followed* by the sign of the exponent, neither of which was complemented when the number was negative. Floating-point numbers could occupy either two or three 16-bit words, depending on whether they were single or double precision.

The floating-point format of the Telefunken TR440, a machine with a 48-bit word length, is also shown. Words containing numbers included a flag bit; in memory, this was a second copy of the sign, but internally in arithmetic registers the bit was used for detecting overflows. Both the mantissa and exponent parts of a floating point number were in one's complement format just as integers were on this machine. Like the IBM 360, the exponent was a power of 16. But the mantissa field in single-precision was 38 bits long, so, as in the MANIAC II, the length of the mantissa field was not a multiple of the digit size.

### Group III Floating-Point Formats

The floating-point formats of many 24-bit computers followed the model shown in the third line of the diagram at the top of the page, but they varied in minor ways from it, and are illustrated below.




The ASI 6020 computer used a 39-bit mantissa in sign-magnitude format, followed by a nine-bit exponent in excess-256 notation, a rare instance of excess-n format being used in a Group III representation of floating-point numbers.

The Datacraft 6024 computer, and its successors from Harris, used two's complement form to represent integers. The exponent field, including sign, was eight bits long. The basic format shown above was used for double-precision floating-point; in single-precision floating point, numbers still occupied two 24-bit words in memory, but the portion of the mantissa in the second word was not used.

The Harris 800 computer added a quad-precision floating-point format that used a full 24-bit word for the exponent. In this format, the exponent came first instead of last, so it belongs to Group II. The sign bits of the second and third words of the mantissa were unused, but this is even found in some of the double-precision formats in Group I.

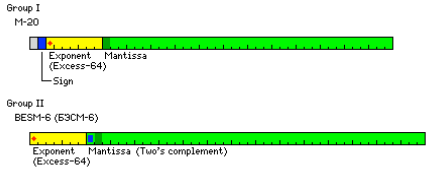
The DDP-24 computer, from 3C and then Honeywell, used sign-magnitude representation for integers, and a multiply instruction ensured both words of the product contained the same sign. It also left the mantissa portion of the second word unused for single-precision numbers. The eight least significant bits of the second word contained the value of the exponent; the sign of the exponent was contained in the sign bit of the second word, instead of that bit being unused.

The ICL 1900 computer used two's complement notation for integers. In double-length fixed-point numbers, the first bit of the second word was always zero. The exponent field of a floating-point number consisted of nine bits in excess-256 notation; the first bit of the second word was a flag which, if one, indicated that a floating-point overflow had taken place. Single-precision numbers were 48 bits long. A double-precision number was 96 bits long; in the second half of the number, which contained the least significant 35 bits of the mantissa, the first bit of each word, and the area corresponding to the exponent field in the first half, were ignored.

The SEL 820 computer, which I believe to be the last of the major members of the "classic" group of 24-bit computers to be described on these pages (here, I am thinking primarily of the Datacraft DC 6024, the Computer Controls Corporation DDP 224, the ASI 6020, the SDS 920 and the SDS 9300, as well as the other computers compatible with these as belonging to this group) has its floating-point format illustrated here as well. Despite the fact that the exponent is at the end of the number and not the beginning, the double-precision format simply appends a word of mantissa to an unaltered single-precision floating-point number.

### And a Few More...

Here are two floating-point formats




from computers in the former Soviet Union; that of the M-20 is a Group I format, and that of the BESM-6 is a Group II format.

### A Note on Field Designations

In the discussion above, I refer to the two major fields in a floating-point number as the "exponent" and the "mantissa". It should be noted that these designations are not without controversy.

The base-10 or common logarithms of numbers, when used for facilitating arithmetic calculations, are divided into an integer portion, called the characteristic, and a fractional part, called the mantissa. Furthermore, when dealing with numbers less than one, whose logarithms are negative, instead of noting the logarithm as a conventional negative number, the integer part is decremented by one (increased by one in magnitude) and noted with a bar over it, instead of a negative sign in front, to indicate that the property of being negative applies to it only, so that the fractional part can be left positive. This further facilitates arithmetic with logarithms, and is illustrated below:

| Number | Base-10 Logarithm | Conventional Common Logarithm<br>Characteristic Mantissa |
|---|---|---|
| 1250 | 3.096910013 | 3.096910013 |
| 1.25 | 0.096910013 | 0.096910013 |
| 0.00125 | -2.903089987 | $\bar{3}$.096910013 |

When numbers are written in scientific notation,

$$1.25 * 10^{3}$$

$$1.25 * 10^{0}$$

$$1,25 * 10^{-3}$$

clearly the exponent to which 10 is raised conveys the same information as the characteristic of the logarithm, and the number by which the power of 10 is multiplied conveys the same information as the mantissa of the logarithm.

The exponent and the characteristic are the same integer, while the mantissa is the logarithm of the number by which the power of 10 is multiplied, rather than that number itself.

Despite this distinction, the corresponding two fields of a floating-point number were referred to as the "characteristic" and the "mantissa" in the documentation for the Univac Scientific 1103A computer, and these terms continued to be used with the successor 36-bit architecture used in the Univac 1107 computer and with other Univac computers, like the Univac 418 and 494 real-time systems.

IBM, on the other hand, was even at an early date dissatisfied with the apparent misuse of the word mantissa, and referred to the two fields of a floating-point number as the "characteristic" and the "fraction" across architectures ranging from the IBM 704 to the System/360. These terms were also used by Xerox in association with its Sigma series of computers, designed to closely resemble the System/360 in some aspects without being compatible. They were also used with the English Electric KDF9 computer, an unrelated machine, and the Foxboro FOX-1.

The SDS 9300 computer, made by the same company that later produced the Sigma computers and was then acquired by Xerox, came with documentation that referred to those fields as the "exponent" and "fraction". The same designations were used in the documentation for the Digital Equipment Corporation PDP-6 and its successors, the PDP-10 and the DECsystem-20, as well as the VAX, the Interdata-8/16 and 7/32 computers, the RC4000, and the Scientific Controls Corporation 660 computer. IBM also slipped, and used these designations in its manual for the STRETCH.

The documentation for other DEC computers, such as the PDP-8, PDP-11, and PDP-15, used the terms "exponent" and "mantissa". These terms were also used in documentation for the Atlas, the Burroughs B5500, the Data General Eclipse MV/4000, the Datacraft DC6024 and its successors from Harris, the DDP 24, the General Electric 235, the Hewlett-Packard 2116, the Honeywell series 6000, the Philco 212, the SEL 840A, among other computers, and with Floating Point Systems' add-on array processing unit for minicomputers as well. Even the RCA Spectra 70 computer, largely compatible with the IBM System/360, used these terms in its documentation in preference to IBM's.

When Seymour Cray left Univac to form Control Data, this new company used the terms "exponent" and "coefficient" in the documentation for the Control Data 1604 and the 6600, and these same terms were used in the manual for the Cray I.

As for the ICL 1900 computer, its documentation used the terms "exponent" and "argument".

And, in the description of the IEEE 754 standard for floating-point numbers, which was first implemented in the 8087 math coprocessor for the Intel 8086 and 8088 microprocessors, the fields were referred to as the "exponent" and "significand". I had wondered if the term "significand" had been coined during the standardization process, by analogy with the term "multiplicand", but I have since discovered that it was in existence long before that had begun.. In the book *Floating-Point Computation*, its author, Pat. H. Sterbenz, cites G. E. Forsythe and C. B. Moler as having used this term in 1967 in *Computer Solution of Linear Algebraic Systems*.

In this connection, it may be noted that the manual for the RECOMP II computer tried very hard to please everyone, by noting that in its two-word floating-point format, "One word contains the mantissa (fraction) and the other word contains the characteristic (exponent)."

To summarize the information listed in the preceding few paragraphs, concerning the designations for the fields of a floating-point number as used by different computer manufacturers, here is a convenient table:

| | | |
|---|---|---|
| characteristic | mantissa | Univac (Univac Scientific 1103A, Univac 1107, Univac 418, Univac 494) |
| characteristic | fraction | IBM (IBM 704, IBM 7090, IBM System/360); XDS Sigma; English Electric KDF9; Foxboro FOX-1 |
| exponent | fraction | SDS 9300; DEC PDP-6, PDP-10; IBM 7030 (STRETCH); Interdata-8/16, 7/32; Regencentralen RC4000; SCC 660 |
| exponent | mantissa | DEC (PDP-8, PDP-15, PDP-11); Atlas; Burroughs B5500; Data General Eclipse MV/4000; Datacraft DC6024; DDP 24; GE 235; HP 2116; Honeywell 6000; Philco 212; SEL 840A; RCA Specra 70; FPS AP-120B |
| exponent | coefficient | Control Data (CDC 1604, CDC 6600), Cray I |
| exponent | argument | ICL 1900 |
| exponent | significand | IEEE 754 standard |

The terms "exponent" and "mantissa" appear to be the ones most commonly found in general use for these two portions of a floating-point number. I've seen it used, for example, in textbooks for users of the IBM System/360 computer, despite the conflict with the terms actually used in manufacturer documentation. I have used them here, as it is my opinion that the primary purpose of language is to convey information easily and rapidly, and using less common if more correct terms would hinder this.

This is despite the fact that one would *never* use the term mantissa for the corresponding portion of a number written in scientific notation. It may well be properly called the significand; since a numerical term multiplying a variable in algebra, such as the 3 in 3x+5, is called a coefficient, and an input to a formula is called an argument, it is easy to see how these terms could have been used as well.

In the early days of using floating-point arithmetic with computers, it was natural to take terms that were familiar and which quickly indicated what they referred to. The two fields of a floating-point number contained the same information as the characteristic and the mantissa of the logarithm of the number it represented, and so those names were used initially. Since the word "characteristic" is commonly used with many other meanings, it gradually gave way to the term "exponent", which was more immediately understandable.

There was, on the other hand, no readily available substitute for "mantissa"; while "coefficient" and "fraction" were also used, they did not indicate the function of this part of a floating-point number as clearly. Thus, despite the fact that the term "mantissa" could be said to be simply incorrect, since it is the number itself, normalized to a small range of magnitudes, rather than a part of the number's logarithm, it continued to be used. At least, the term "significand" also clearly indicates the function of this part of a floating-point number.

Still, "exponent" is less specific than "characteristic" in much the same way as "coefficient" is less specific than "mantissa". But, in the former case, "characteristic" has, as noted, other commonly-used meanings in English, while "mantissa" is only used with one other meaning, and that one closely related to the intended one. Thus, the evolution of this terminology is an illustration of how languages develop, by people settling on those terms that are the easiest for them to remember, and which cause them the least confusion.

To indicate the depth of feeling that this issue has aroused, in *Seminumerical Algorithms*, Volume 2 of *The Art of Computer Programming*, Donald W. Knuth correctly notes that the use of the term "mantissa" for this field of a floating-point number constitutes an "abuse of terminology", and goes on to note that "mantissa" has the meaning of "a worthless addition" in English.

In the case of the ICL 1900, one wonders if perhaps during the design of its architecture, or the preparation of its documentation (or perhaps those of its Canadian predecessor, the Ferranti-Packard 6000) someone might have decided to choose the term "argument" for this field because it seems to be good at *starting* one.

Of course, floating-point does not strictly correspond to scientific notation. Even when decimal rather than binary arithmetic is involved, the form

```
+ - 02 12500000
```

would refer to

```
          -2
0.125 * 10
```

which, of course, is equal to

```
         -3
1.25 * 10
```

and, thus, with the radix point preceding the leading edge of the field, it does contain a fraction, explaining IBM's chosen terminology.

I had been interested in finding out if the components of a number expressed in scientific notation had ever been given conventional names, and had thought that perhaps this would have been done around the time of its origin. Another place to look would be in textbooks of arithmetic of a traditional bent which dealt with scientific notation.

Archimedes, in *The Sand Reckoner*, illustrated large numbers by means of exponentiation, and René Descartes is responsible for our modern notation for exponentiation, and thus I have come across web sites crediting each of them with its invention. However, the use of scientific notation in its modern form appears to date from the late 19th century; this page gives as its earliest example of the use of modern scientific notation one from a paper by Johann Jakob Balmer in 1885, and on the same web site, this page notes a use of the name "scientific notation" from a book published in 1921, which beats the Oxford English Dictionary, which had cited another dictionary as its first example of the use of the phrase.

Having failed, however, to find a traditional name for this portion of a number in scientific notation within the mists of time, it can still be noted that that there are other possibilities, as yet unused, for names for the part of a floating-point number which is vulgarly called the mantissa.

The other part, usually called the exponent, really does correspond exactly to the characteristic. So, this part does at least correspond to the mantissa; it is the **antilogarithm of the mantissa**. As it happens, old tables of the logarithms of sines, cosines, and tangents were called tables of *logarithmic* sines, cosines, and tangents, so I suppose that one could shorten the name, and call it the **antilogarithmic mantissa** of a floating-point number. Of course, if it is a special *kind* of mantissa, that justifies using the common term mantissa after all.

Since the other part is usually called the exponent, which applies when raising anything to a power, rather than more specifically the characteristic, another possibility is to use a nonspecific term for it. The existing term "coefficient" already

answers fairly well to that, but another alternative is **factor**; if the original meaning of *mantissa* is "a small addition", perhaps this comes as close as anything to a term meaning "a small multiplication".

The term *significand* is, I must admit, a particularly felicitous coinage, even if it is of recent origin. Just as a multiplicand is that which gets multiplied by the multiplier, the significand is that which derives its significance from the exponent part of the floating-point number. Another way of expressing this might be to call this part of a floating-point number the **unscaled number**: the number before it is given a scale by the scale factor of the radix to the power of the exponent.

Since this field contains the significant digits of the floating-point number, perhaps we could get even simpler, and just call it the **digits** field!

Of course, where there is no good word in English for an object, the English language has often had recourse to absconding with a word from some foreign language; as perhaps in some distant land, an obvious and natural name for this has rescued the people there from the perplexity that so besets us.

However, as one final argument for retaining the traditional term "mantissa", it might be noted that while mathematicians working with calculus and related disciplines will continue using the logarithm function for a long time to come, generally the logarithms they will be using will be natural logarithms; that is, logarithms to the base e (2.71828...); and, even in those cases when they use logarithms to integer bases, it will not usually be for the purpose of simplifying the task of performing a multiplication by hand.

Thus, for such purposes, a logarithm will simply be a number, which may be positive or negative, and with an integer part and a fractional part. It is only when using log tables as tools in performing multiplication that it is useful to subtract one from the integer part of a negative logarithm so that the fractional part can always be positive - and it is the fractional part of a logarithm, so modified, that is called the mantissa.

In decimal floating point, 3 * 10-3 might become + -2 3000000, but it isn't going to become + -2 3333333 - that is, for numbers less than one in magnitude, the reciprocal of the number isn't going to be what is represented. This is what floating-point numbers have in common with logarithms in the computational characteristic and mantissa representation.

Because performing multiplication by the aid of logarithms is falling into disuse, and because the term mantissa belongs not to mathematics in the sense of analysis or algebra, but to practical arithmetic, its meaning in connection with floating-point numbers is not, in fact, a threat to the integrity of the language of mathematics.

### A Final Comment

Note also that the three lines in the first diagram which illustrated the three possible general types of format, as well as the illustrations of floating-point formats in the other diagrams, assume that the component parts of the floating-point number, whether they are 24-bit words or 8-bit bytes, are lined up so as to be in the normal left to right direction from most significant to least significant for representing integers. Thus, on a *little-endian* machine, the component of the number on the left would be at a location with a higher address instead of a lower one. Note, however, that on at least some machines, while integers were represented in little-endian form, floating-point numbers were represented in big-endian form.

On the PDP-11, a particularly unfortunate variation of this took place. As it was the first computer to attempt to achieve the consistent use of a little-endian representation for data (previous computers were always big-endian when packing characters into words, but sometimes were little-endian when using two words to represent a long integer) I had thought that the likeliest cause of this was a failure of communication concerning the design of the PDP-11 with the engineers designing the FP-11, the hardware floating-point unit for the PDP-11 that first embodied that format. However, a memo clearly showing that the word containing the exponent and the most significant part of the mantissa would be at the lowest address in that format was in fact discussed at the highest levels within DEC, so explaining this as an accident is not possible. It may instead have been that putting the first part of the number in the lowest address would simplify software floating-point, and the fact that there was a clash between that and the ordering of bytes within a word may simply have been thought of as completely irrelevant.

The PDP-11 had a 16-bit word, but could also address and manipulate 8-bit bytes directly. As with many other computers, such as the Honeywell 316, a 32-bit integer was stored with its least significant 16-bit word first, in the lower memory address, so that addition could begin while the more significant words of the operands were being fetched. However, unlike other computers in existence at the time, for consistency, the PDP-11 was designed so that the least significant 8 bits of a 16-bit word had the lower byte address, and the more significant 8 bits of a 16-bit word had the higher byte address.

Because the FP-11 was designed as though the PDP-11 were a big-endian computer instead of a little-endian computer, it placed the most significant 16 bits of the values on which it acted in the 16-bit word at the lowest memory address, the next less significant 16 bits in the 16-bit word at the next higher memory address, and so on.

In addition to floating-point numbers, this included 32-bit integers as well, but as the PDP-11 already posessed instructions to assist in handling 32-bit integers in little-endian format, this flaw was corrected in subsequent extensions to the PDP-11 architecture. The floating-point format, however, remained unaltered.

The byte addressing within a word was a property of the base PDP-11 architecture, and was not altered by the design of the FP-11 as though it were for a big-endian machine. The most significant bit within a 16-bit word inside the FP-11 was still transmitted to the most significant bit of a 16-bit word inside the PDP-11. Hence, in the illustrations of the floating-point format for the PDP-11 shown above, the successive bytes in a floating-point number have addresses in the order:

```
 1  0  3  2  5  4  7  6
```

instead of

```
 7  6  5  4  3  2  1  0
```

as is the case on a consistently little-endian machine, as it had been intended to make the PDP-11, or

```
 0  1  2  3  4  5  6  7
```

as they would be on a consistently big-endian machine, like the IBM System/360 or many other computers.

This aspect of the PDP-11 floating-point format was preserved on the VAX computer, because it included a PDP-11 compatibility mode; this was true not only for the F and D formats, but also for the new G and H formats, as its documentation noted. On the other hand, IEEE 754 floating-point numbers appear to be stored in consistent little-endian order, although the document in which I saw that may have been referring only to Itanium systems and not Alpha systems.

[Next] [Up] [Previous]

# Computer Arithmetic

Because electronic logic deals with currents that are on or off, it has been found convenient to represent quantities in binary form to perform arithmetic on a computer. Thus, instead of having ten different digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, in binary arithmetic, there are only two different digits, 0 and 1, and when moving to the next column, instead of the digit representing a quantity that is ten times as large, it only represents a quantity that is two times as large. Thus, the first few

numbers are written in binary as follows:

| | Decimal | Binary |
|---|---|---|
| Zero | 0 | 0 |
| One | 1 | 1 |
| Two | 2 | 10 |
| Three | 3 | 11 |
| Four | 4 | 100 |
| Five | 5 | 101 |
| Six | 6 | 110 |
| Seven | 7 | 111 |
| Eight | 8 | 1000 |
| Nine | 9 | 1001 |
| Ten | 10 | 1010 |
| Eleven | 11 | 1011 |
| Twelve | 12 | 1100 |

The addition and multiplication tables for binary arithmetic are very small, and this makes it possible to use logic circuits to build binary adders.

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

| * | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Thus, from the table above, when two binary digits, A and B are added, the carry bit is simply (A AND B), while the last digit of the sum is more complicated; ((A AND NOT B) OR ((NOT A) AND B)) is one way to express it.

And multiplication becomes a series of shifts, accompanied by successive decisons of whether to add or not to add.

When a number like 127 is written in the decimal place-value system of notation, one can think of it as indicating that one has:

seven marbles,

two boxes that have ten marbles in each box,

one box that has ten boxes each with ten marbles in that box.

The binary system is another place-value system, based on the same principle; however, in that system, the smallest boxes only have two marbles in them, and any larger box contains only two of the next smaller size of box.

We have seen how we can use two digits, 0 and 1, to do the same thing that we can do with the digits 0 and 9 only; write integers equal to or greater than zero. In writing, it is easy enough to add a minus sign to the front of a number, or insert a decimal point. When a number is represented only as a string of bits, with no other symbols, special conventions must be adopted to represent a number that is negative, or one with a radix point indicating a binary fraction.

Historically, computers have represented negative numbers in several different ways.

One method is sign-magnitude representation, in which the first bit of the integer, if a one, indicates that the number is negative.

Another is one's complement notation, in which, when a number is negative, in addition to this being indicated by setting the first bit of the number to a one, the other bits of the number are all inverted.

By far the most common way to represent negative integers on modern computers, however, is two's complement notation, because in this notation, addition of signed quantities, except for the possible presence of a carry out when an overflow is not actually taking place, is identical to the normal addition of positive quantities. Here, to replace a number with its negative equivalent, one inverts all the bits, and then adds one.

Another method of representing negative numbers is simply to add a constant, equal to half the range of the integer type, to the number to obtain its binary representation. This is usually used for the exponent portion of floating-point numbers, as we will see later.

| Number | Binary representations | | | | Decimal representations | | | |
|---|---|---|---|---|---|---|---|---|
| | Sign-magnitude | One's complement | Two's complement | Excess-512 | Sign-magnitude | Nine's complement | Ten's complement | Excess-500 |
| +511 | 0111111111 | 0111111111 | 0111111111 | 1111111111 | | | | |
| +510 | 0111111110 | 0111111110 | 0111111110 | 1111111110 | | | | |
| ... | | | | | | | | |
| +500 | 0111110100 | 0111110100 | 0111110100 | 1111110100 | | | | |
| +499 | 0111110011 | 0111110011 | 0111110011 | 1111110011 | 499 | 499 | 499 | 999 |
| +498 | 0111110010 | 0111110010 | 0111110010 | 1111110010 | 498 | 498 | 498 | 998 |
| ... | | | | | | | | |
| +2 | 0000000010 | 0000000010 | 0000000010 | 1000000010 | 002 | 002 | 002 | 502 |
| +1 | 0000000001 | 0000000001 | 0000000001 | 1000000001 | 001 | 001 | 001 | 501 |
| 0 | 0000000000 | 0000000000 | 0000000000 | 1000000000 | 000 | 000 | 000 | 500 |
| -1 | 1000000001 | 1111111110 | 1111111111 | 0111111111 | 501 | 998 | 999 | 499 |
| -2 | 1000000010 | 1111111101 | 1111111110 | 0111111110 | 502 | 997 | 998 | 498 |
| ... | | | | | | | | |
| -498 | 1111110010 | 1000001101 | 1000001110 | 0000001110 | 998 | 501 | 502 | 002 |
| -499 | 1111110011 | 1000001100 | 1000001101 | 0000001101 | 999 | 500 | 501 | 001 |
| -500 | 1111110100 | 1000001011 | 1000001100 | 0000001100 | | | 500 | 000 |
| ... | | | | | | | | |
| -510 | 1111111110 | 1000000001 | 1000000010 | 0000000010 | | | | |

```
-511    1111111111 1000000000 1000000001 0000000001
-512                          1000000000 0000000000
```

Floating-point numbers are used in scientific calculations. In these calculations, instead of dealing in numbers which must always be exact in terms of a whole unit, whether that unit is a dollar or a cent, a certain number of digits of precision is sought for a quantity that might be very small or very large.

Thus, floating-point numbers are represented internally in a computer in something resembling scientific notation.

In scientific notation,

$$1{,}230{,}000 \text{ becomes } 1.23 \times 10^{6}$$

and the common logarithm of 1,230,000 is 6.0899; the integer and fractional parts of a common logarithm are called, respectively, the *characteristic* and the *mantissa* when using common logarithms to perform calculations.

To keep the fields in a floating-point number distinct, and to make floating-point arithmetic simple, a common way to represent floating point numbers is like this:

First, the sign of the number; 0 if positive, 1 if negative.

Then the exponent. If the exponent is seven bits long, it is represented in excess-64 notation; if it is eleven bits long, as in this example, it is represented in excess-1,024 notation. That is, the bits used for the exponent are the binary representation of the exponent value plus 1,024; this is the simplest way to represent exponents, since everything is now positive.

Finally, the leading digits of the actual binary number. As these contain the same information as the fractional part of the base-2 logarithm (or the base-16 logarithm, had the exponent been a power of 16, as is the case on some other architectures instead of the one examined here) of the number, this part of a floating-point number is also called the mantissa, even if this term is much criticized as a misnomer.

The exponent is in excess 1,024 notation when the binary point of the mantissa is considered to precede its leading digit.

As the digits of the mantissa are not altered when the number is negative, sign-magnitude notation may be said to be what is used for that part of the number.

Some examples of floating-point numbers are shown below:

```
1,024          0 10000001011 10000
  512          0 10000001010 10000
  256          0 10000001001 10000
  128          0 10000001000 10000
   64          0 10000000111 10000
   32          0 10000000110 10000
   16          0 10000000101 10000

   14          0 10000000100 11100
   12          0 10000000100 11000
   10          0 10000000100 10100

    8          0 10000000100 10000

    7          0 10000000011 11100
    6          0 10000000011 11000
    5          0 10000000011 10100

    4          0 10000000011 10000

    3.5        0 10000000010 11100
    3          0 10000000010 11000
    2.5        0 10000000010 10100

    2          0 10000000010 10000

    1.75       0 10000000001 11100
    1.5        0 10000000001 11000
    1.25       0 10000000001 10100

    1          0 10000000001 10000

    0.875      0 10000000000 11100
    0.75       0 10000000000 11000
    0.625      0 10000000000 10100

    0.5        0 10000000000 10000
    0.25       0 01111111111 10000
    0.125      0 01111111110 10000
    0.0625     0 01111111101 10000
    0.03125    0 01111111100 10000

    0          0 00000000000 00000

   -1          1 10000000001 10000
```

In each of the entries above, the entire 11-bit exponent field is shown, but only the first five bits of the mantissa field are shown, the rest being zero.

Note that for all the numbers except zero, the first bit of the mantissa is a one. This is somewhat wasteful; if the exponent is a power of 4, 8, or 16 instead of a power of two, then the restriction seen will only be that the first 2, 3, or 4 bits, respectively, of the mantissa will not all be zero.

A floating-point number whose first digit, where the size of a digit is determined by the base of the exponent, is not zero is said to be *normalized*. In general, when a floating-point number is normalized, it retains the maximum possible precision, and normalizing the result is an intrinsic part of any floating-point operation.

In the illustration above,

```
0 00000000000 10000
```

would represent the smallest number that can possibly be normalized. Some computers permit gradual underflow, where quantities such as

```
0 00000000000 01000
```

are also allowed, since they are as normalized as is possible, and their meaning is unambiguous.

### The Early Days of Hexadecimal

I don't know where else on my site to put this amusing bit of trivia.

Most computers, internally, use binary numbers for arithmetic, as circuits for binary arithmetic are the simplest to implement. Some computers will perform base-10 arithmetic instead, so as to directly calculate on numbers as they are used by people in writing. This is usually done by representing each digit in binary form, although a number of different codings for decimal digits have been used.

When a computer uses binary arithmetic, it is desirable to have a short way to represent binary numbers. One way to do this is to use octal notation, where the digits from 0 to 7 represent three consecutive bits; thus, integers in base-8 representation can be quickly translated to integers in base-2 representation. Another way, more suited to computers with word lengths that are multiples of four digits, is hexadecimal notation.

Today, programmers are familiar with the use of the letters A through F to represent the values 10 through 15 as single hexadecimal digits. Before the IBM System/360 made this the universal standard, some early computers used alternate notations for hexadecimal:

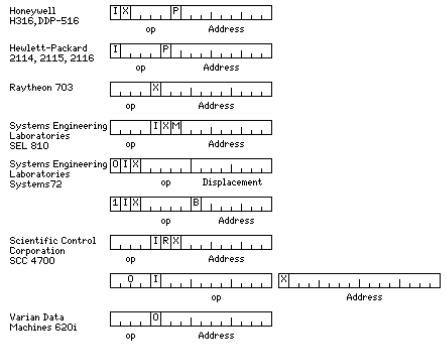
| | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|
| System/360 | A | B | C | D | E | F |
| C | a | b | c | d | e | f |
| SWAC | u | v | w | x | y | z |
| Monrobot XI | S | T | U | V | W | X |
| Datamatic D-1000 | b | c | d | e | f | g |
| LGP-30 | f | g | j | k | q | w |
| ILLIAC | k | s | n | j | f | l |

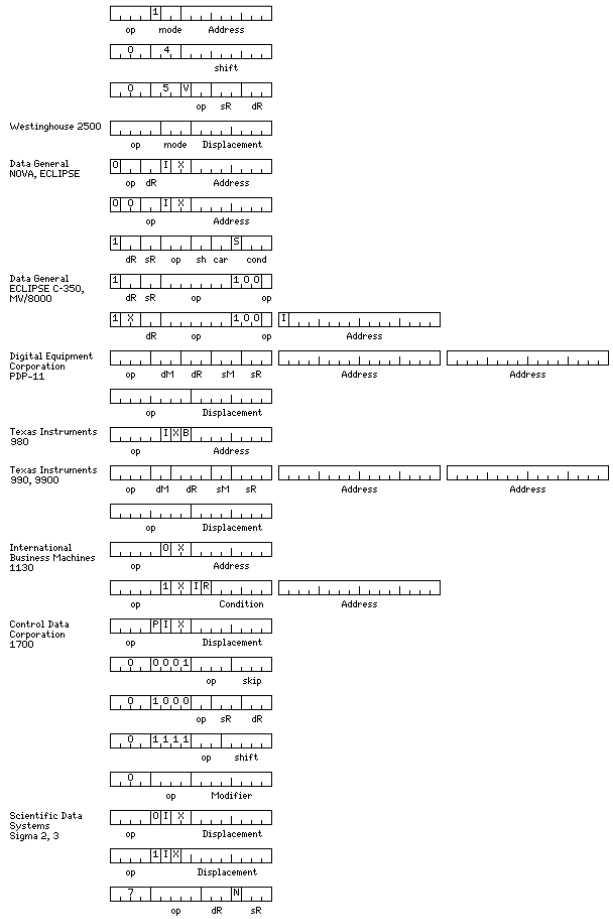

- Floating-Point Formats
- Advanced Arithmetic Techniques
- Decimal Representations
  - Chen-Ho Encoding and Densely Packed Decimal
  - The Proposed Decimal Floating Point Standard
  - Quasilogarithmic Floating Point

[Next] [Up] [Previous]


# Real Machines with 16, 32, and 30-bit words

The following diagram






illustrating the instruction formats of some real computers with a 16-bit word length, shows some of the variety that is available.

The Honeywell 316 and the Hewlett-Packard 211x computers illustrate the type of simple computer which uses a bit, shown here as P, to indicate if the instruction refers to an address on the current page, or one on page zero. The I bit at the beginning of the instruction indicates indirect addressing. (This classic architecture is also exemplified by the very popular 12-bit PDP-8.) And the Honeywell 316 computer also provides for indexed addressing with one index register.

The Hewlett-Packard 2116A computer was brought out by that company shortly after Digital Equipment Corporation started the minicomputer revolution with the original PDP-8 computer in 1965, and is considered to be the first 16-bit minicomputer. Just as the PDP-8 architecture was inherited from the PDP-5, not generally considered to be a minicomputer, and which came out in 1963, the Honeywell 316 shares its architecture with earlier machines as well; the DDP-116, originally brought out by Computer Controls Corporation before its purchase by Honeywell, dates from 1964.

If one thinks of a minicomputer as something that comes, or at least can come, in a small box, like a PDP-8/S or, at most, a Hewlett-Packard 2115A or a Honeywell 316, then one could claim that the PDP-8/S, from 1966, might be the first minicomputer. The chip used internally in such HP calculators as the 9825A implemented a variation of the HP2100 instruction set, and was the world's first 16-bit microprocessor, although not made generally available.

The Raytheon 703 normally always addressed the current page the instruction was on, but it had an instruction that allowed a different page to be selected for the following memory-reference instruction.

The SCC 4700 used a bit to indicate relative addressing with a signed displacement instead of a bit for current page addressing. It also had a number of two-word instructions with a full address in the second word, including optional hardware

floating-point instructions.

The Varian 620 had a direct addressing mode which provided access to the first 2048 words of memory, shift instructions that shifted the accumulator an arbitrary number of places, and relative and indirect addressing, for a powerful and advanced instruction set.

The Westinghouse 2500 allocated a generous five-bit field for the opcode, and a three-bit field for the addressing mode, leaving only eight bits for the address portion of the instruction. This allowed for indirect addressing, and addresses could be absolute, relative, or indexed by one of two index registers.

The Data General NOVA computer used a very tightly packed instruction format to allow it to provide four general registers, the last three of which could serve as index registers. Memory is referenced by load, store, and jump instructions; arithmetic operations are performed only between registers.

The bit marked S in the operate instruction format, if equal to 1, indicated that the result of the operation would not be loaded into the destination register, but instead would only be tested for the conditional skip condition in the instruction.

The instruction set was extended, first in the Eclipse C/350 to provide for decimal arithmetic and related operations, and then in the MV/8000 to permit 32-bit operation, by making use of the opcodes where the S bit is set, and the skip condition was either to always skip or to never skip, since then the operation performed would be irrelevant, and only one opcode in each case would need to be saved for a no-operation instruction or a skip instruction.

The Digital Equipment Corporation PDP-11 and the Texas Instruments 9900 microprocessor illustrate a more modern and symmetrical architecture. Both source and destination operands have the same form, and may either be registers or memory locations. If the operands are memory locations, the register field indicates an index register. The 9900 had sixteen general registers which were in a workspace in memory; the PDP-11 had six general registers, and could also use a stack pointer or the program counter as an index, which were denoted by placing 6 or 7 in the index register field respectively.

The addressing modes offered with the TI 9900 were:

```
00 Register operand
01 Register Indirect
10 Memory operand (indexed if register field not zero)
11 Register Indirect with Autoincrement
```

and those offered with the PDP-11 were:

```
000 Register operand
001 Register Indirect
010 Register Indirect with post-autoincrement
011 Indirect Register Indirect with post-autoincrement
100 Register Indirect with pre-autodecrement
101 Indirect Register Indirect with pre-autodecrement
110 Memory operand (indexed if register field not zero)
111 Indirect memory operand (indexed if register field not zero)
```

The Texas Instruments 980 computer, in addition to having an indirect bit, and an index bit that, as in other small computer architectures, controlled the use of one index register, had a B bit in the instruction, which determined if an address was relative to the address of the instruction in which it was found, or to the area to which the base register pointed. This was easier for compilers to handle than the type of paged addressing found on the Honeywell H-316 or the PDP-8.

The instruction formats for the Control Data 1700 computer are shown in some detail here. The three bit destination register field in the register to register instructions controls three possible destination registers, so it is possible to broadcast a result to multiple registers:

```
001  M register
010  Q register (multiplier-quotient, or index register 1)
100  A register (accumulator)
```

or, if the field contains a zero, it refers to index register 2, which is the location at the high end of memory. When multiple source registers are specified, the OR of their contents is taken as the source value.

The IBM 1130 was able to reserve seven bits in two-word instructions for functions that were only used with a limited number of instructions; the condition bits for conditional branch instructions, and the R bit used for returning from an interrupt service routine. In the single-word format, when one of the three index registers is specified, it serves as a base register in effect; when 0 is in the index field, the address field is interpreted as a displacement relative to the current location.

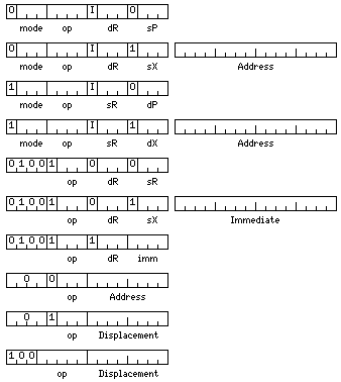
The Control Data 1700 had an interesting feature not shown in the diagram; every 16-bit word in the memory also had a program protect bit associated with it. It could have up to 32K of memory, and was intended for real-time control applications.

These different instruction formats show different ways in which small computers could cope with the fact that their word size was too small to contain both an operation code and the complete address of a location in memory. They could use various ways to make use of a shortened address to indicate a memory location, or make instructions that did not refer to memory do more of the work, or both.

While the larger computers in the Scientific Data Systems Sigma series of computers had instructions that were 32 bits long, the smaller ones had 16-bit instructions. Program-relative instructions had a 9-bit displacement field, while 8 bits were used for the displacement with other addressing modes. One opcode was instead used to indicate register to register instructions; the bit marked N usually indicated that the source operand was to be inverted before use.

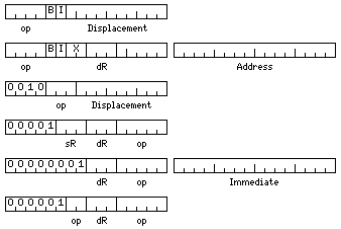
The illustration above just briefly shows the main instruction formats for the computers illustrated; a later section will show, just for one very famous computer, the popular Digital Equipment Corporation PDP-8, a more complete illustration of its instruction formats.

Some later minicomputers had considerably more elaborate repertoires of addressing modes, once the PDP-11 led the way. For example, here is one computer clearly named so as to endow it with a fighting spirit:




The diagram above illustrates the addressing modes of the Lockheed SUE computer. Only a subset of them is shown. Instructions can be from memory to register or register to memory. The return from interrupt instruction can use either page zero addressing or program-relative addressing, thus making a slight nod to the addressing modes of earlier minicomputers. Branches are program-relative, but there is also a jump instruction similar in format to a memory-reference instruction (but sharing its first four bits with the register-to-register instructions). Note also that there is both a long and a short format for immediate operands.

The SPC-16 from General Automation was an interesting design. Its instruction formats are shown below:



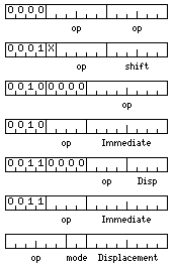

One of its most interesting features is that, like the PDP-11, it had eight registers, some of which served special purposes. Its registers were:

```
000 A    100 B
001 X    101 C
010 Y    110 D
011 Z    111 E
```

If an instruction indicated indexing, one of registers X, Y, and Z would be the index register. If base-relative addressing, which could also be considered pre-indirect indexing, was indicated, register D would supply the base address. Register E was used to save the return address of subroutines. The B and C registers could serve as extensions of register A, the accumulator, for multiplication and division.

To allow rapid handling of interrupts, the machine had a second set of registers for interrupt service routines.

One computer with a particularly complex set of instruction formats, of which the diagram below represents but a sketchy summary,



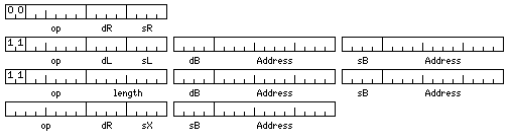

is the HP 3000 computer. The first format depicted is one in which the 16-bit word contains two 6-bit stack operations. The one with a four-bit displacement is used for input-output instructions, to indicate where a device address is on the stack.

Note that this reflects the architecture of the original HP 3000 computers; later systems in that product line used the PA-RISC architecture instead, the same one as used with the HP 9000 computers (which themselves migrated to the Itanium).

Even large computers sometimes made the effort to make instructions shorter than they would be if they included full memory addresses.

The IBM 360 computer allowed general registers 1 through 15 to be used as index registers; general register 0 could not be so used, so that zeroes in the index field of an instruction would indicate that indexing was not used. The base field acted the same way; since the contents of both registers were applied to the address in the same way, by addition, it was only a convention that one field was used to indicate the index register, and the other field the base register.

Thus, the formats of its instructions were:

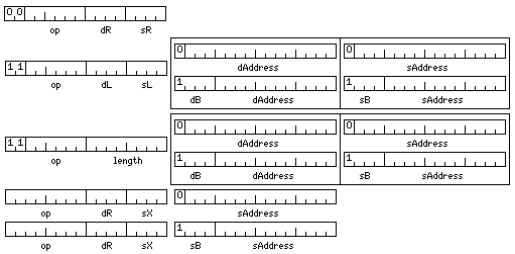



Initially, on April 7, 1964, when IBM announced System/360, they announced models 30, 40, and 50, as well as what later became models 65 and 75. Later, they introduced other models in the series. The System/360 Model 85 was notable for introducing high-speed cache memory in the form we understand it today, and it also added the ability to use two floating-point registers together for a 128-bit extended-precision quantity. Like double-precision floating-point on the 7090 and related machines, the second half of the number had an offset exponent field.

The System/360, as can be seen from the instruction formats above, was a general-register machine, with 16 registers numbered from 0 to 15. These registers could be used as accumulators, as index registers, or as base registers. Since the floating-point registers were used only as floating-point accumulators, only four of them were provided initially in the System/360 architecture. The floating-point registers were numbered 0, 2, 4, and 6. It was presumably decided to number them in this way because they were twice as long (64 bits) as the fixed-point general registers. Only much later, with Enterprise Systems Architecture/390, was a full complement of 16 floating-point registers included; for compatibility, the eight possible register pairs that can contain a 128-bit extended-precision floating-point number are (0,2), (1,3), (4,6), (5,7), (8,10), (9,11), (12,14), (13,15). It was also at this time that the ability to handle "binary floating-point", that is, floating-point numbers following the IEEE 754 standard, was added to the architecture, and the term "hexadecimal floating-point" was applied to the original floating-point format of the IBM System/360.

One low-end model, model 20, could only have a maximum memory size of 16 K bytes, and it was incompatible in that not only the optional floating-point feature was not available for it, but also 32-bit integer arithmetic, part of the standard instruction set, was omitted.

But it also had an even greater divergence from the standard 360 architecture, a modification to its method of addressing memory:




Only general registers 8 through 15 could be used as base registers. When the first bit of a 16-bit address field in an instruction was a zero, then the remaining 15 bits were used as an address, without the need to add the contents of a base register to it. Where it is shown for the memory to register instruction format, that format is simply divided into two formats, one for each case; it also applied to both address fields in the decimal and string instructions as well, and there boxes

have been drawn around the address fields to show that either field could take either format. Incidentally, the Univac 9200 and 9300 also adhered to this convention.

There was actually a good reason why only general registers 8 through 15 could be used as base registers on the System/360 model 20; these general registers, 16 bits long instead of 32 bits long on this model, were the only general registers it had.

This accorded well with the regular conventions on register use on other models of System/360, where the lower-numbered general registers were used for data, and the higher-numbered ones were used as base registers or for memory pointers associated with subroutine calls.

Note that this necessitated a modification to the relocating linking loader, and to how programs would prepare object modules for input to the loader; for a normal System/360, only address constants, which were word-aligned 32-bit values, would need to be relocated, but here, when base registers were not used, the address fields in instructions would need to be relocated as well. At the least, the loader would need to make provision for 16-bit address constants.

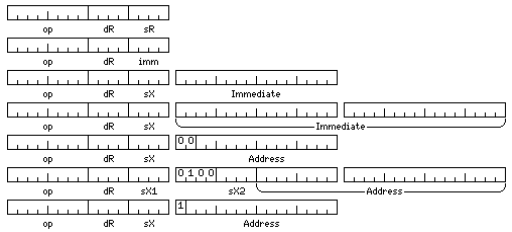
One other member of the IBM System/360 family was slightly incompatible with the standard. The System/360 Model 44 computer had a knob on the front, which could be set at 8, 10, 12, or 14. This referred to the number of hexadecimal digits in the mantissa of a double-precision floating-point number. Single-precision floating-point operations always took place on a 32-bit operand, occupying four bytes, but double-precision operations, while they could take place on 64-bit operands, occupying eight bytes, if the knob was set at 14, could also take place on 5, 6, or 7 byte-long operands for the settings of 8, 10, and 12; this allowed the highest possible floating-point speed for the precision actually required. Apparently, to maintain compatibility, regardless of the selected precision, double-precision floating-point numbers still occupied eight bytes of memory. Also, the character and decimal instructions were not available for it.

RCA, in 1966, announced the Spectra/70 series of computers, made from monolithic integrated circuits, which had the same user mode (but not privileged mode) instructions as the System/360. This line of computers was taken over by Univac in September, 1971. Before this happened, Univac first made the 9200 and 9300 computers, which were IBM compatible and which, having a line printer built in, or being built into a line printer, were smaller-scale systems, but then introduced, in 1968, the 9400, which was more clearly a large mainframe. Planned later models in that series were modified to integrate them with Series 90, which is what the Spectra 70 computers became when offered by Univac.

The two lower-end members of the Spectra/70 family, the 70/15 and 70/25, were also not fully compatible with System/360. Unlike the System/360 model 20, or the Univac 9200 and 9300, they did not change the form of addresses.

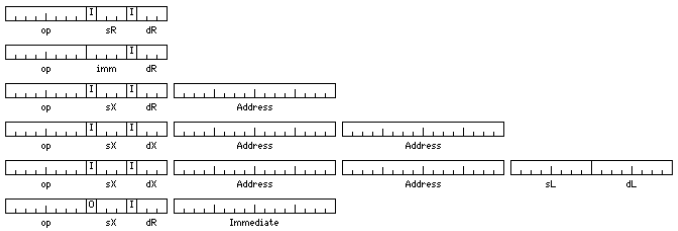
However, while they still had general registers used as base and index registers, these were accessed only with the load multiple and store multiple instructions. In addition to the instructions for packed decimal arithmetic, these machines had a set of instructions with a similar format for binary arithmetic, with a four-bit length field for each operand. The System/360 architecture already included instructions for AND, OR, and XOR that had the format of the MVC instruction with a single 8-bit length, and these Spectra/70 models made use of them as well.

Interdata constructed minicomputers which were originally a compatible subset of the IBM 360, but when they came out with successor systems with expanded capabilities, they extended them in a different manner:




They added a short form instruction with a four-bit immediate operand, as well as instructions with immediate operands that could be either 16 or 32 bits long, and which could have the contents of an index register added to them (as would be the case in a load effective address instruction). Normal memory-reference instructions could either have a plain 14-bit address, for compatibility with earlier Interdata machines, a 24-bit address which could be indexed by two index registers at once if desired, or a 15-bit program-relative address.

Another series of computers inspired by the IBM System/360, this time one which directly competed against some of the lower-end IBM mainframes, was made by Memorex:




In this architecture, only eight general registers were used, but a fourth bit was used with each field specifying a register to indicate indirect addressing. Thus, register indirect addressing was included in the architecture. As well, there were two forms of immediate addressing, one with a four-bit value in the first 16 bits of the instruction, and one where indexing could be specified to cause contents of a general register other than register 0 to be added to the immediate operand before use. This was termed direct addressing, thus the add instruction of that form was termed ADDD, Add Direct, instead of, say, Add Effective Address.

Another computer inspired by the System/360, but with very different instruction formats nevertheless, is the Sigma; this computer had a 32-bit instruction word, and will be discussed below along with other computers with that type of instruction. Not only its data formats, but the number of registers, the amount of addressable memory, and the instruction repertoire were chosen to make a machine equal in power to the System/360, but, due to a more traditional design, less expensive to produce, despite being made from discrete transistors.

Numerous machines were made that had the same instruction set as the System/360. IBM itself used a different designation, 4 Pi, for computers sold to the military that had the 360 instruction set.

The Union of Soviet Socialist Republics redirected its computer efforts to the Unified System of computers. Such machines as the Ryad EC-1040 were compatible with the System/360, and were believed to have been partially designed through information obtained by espionage activities.

Later, a plug-compatible computer industry arose in the United States, leading to such computers as the Amdahl 470/V6, the Itel AS/5, and the Magnuson M80.

Gene Amdahl was, of course, the designer of the IBM 360 architecture; he left IBM out of frustration with the company not pursuing an opportunity to make larger systems in that line. Itel was a company that purchased 360 mainframes and then leased them to others which then went on to include memory made for it by National Semiconductor for them. Fujitsu was a partner of Amdahl, and continues to make plug-compatible machines; first National Semiconductor, and then Hitachi, which had been the actual maker of the CPUs, acquired Itel's plug-compatible business.

Also, there was the Two-Pi computer, a computer made with bit-slice technology that was an attempt to offer mainframe power in the size of a minicomputer.

Incidentally, Magnuson Computers was founded by Carl Amdahl, the son of Gene Amdahl, and his partner in Trilogy Computers. As I recall, this was all amicable; Amdahl computers made the larger computers, and Magnuson the smaller ones.

A while back, I had heard of a company called Standard Computer, that made perhaps the only IBM 7090 clone ever attempted. I had not known of their IC 9000 computer, a microprogrammable computer clearly designed to facilitate imitation of the IBM 360 instruction set; there is a preliminary manual for it on Al Kossow's site, and I have heard the claim that this computer was actually constructed and sold. It may have been sold to OEMs that later resold it as their own plug-compatible machine rather than directly to end-users, which would explain why I hadn't heard of it before.

A circle spans either 360 degrees or 2*pi radians, but a sphere comprises 4*pi steradians; IBM's military version of the 360 was largely used in avionics, the air being a realm of freedom of three-dimensional movement (thus, it was not that IBM could not count that it used the name 4 Pi and left the 2 Pi trademark lying around), and one of the later versions of it, the System/4Pi model A-101S, serves still on the Space Shuttle.

IBM's later System/3 computer had a very simple and regular instruction format, which it shared with the System/32:

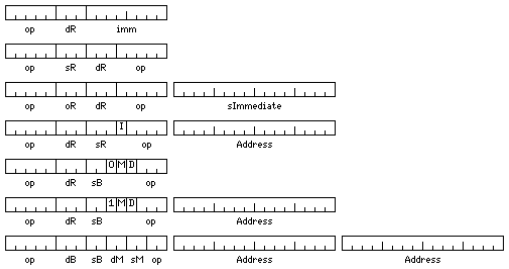

The mode bits had the following values:

```
00 16-bit absolute address
01 8-bit base-relative address, base register 1
10 8-bit base-relative address, base register 2
11 no operand
```

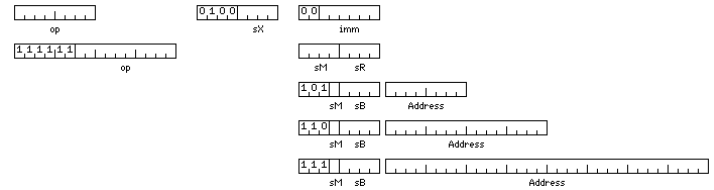
Although the opcode field was only four bits long, the machine could have up to 64 instructions, since whether an instruction had two operands, no operands, or only a source operand, or only a destination operand, also determined what instruction it was.

The byte shown as the length field was actually called the Q byte, and was used for various purposes in different instructions, but a length field was one of its most common uses.

The IBM Series/1 computer, on the other hand, had a more complicated and thus more compact and flexible set of instruction formats, some of which are shown in the diagram below:

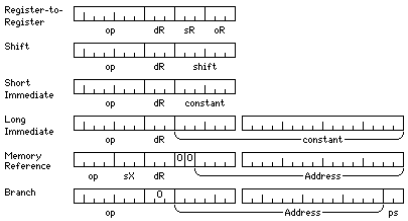



The diagram below




attempts to illustrate the native instruction formats of the Digital Equipment Corporation's famous VAX computers, which also, at least in their early models, included PDP-11 emulation. An instruction consists of an opcode, which may be one or two bytes long, followed by as many operands as needed. The diagram shows the two opcode formats, followed by the operand formats. One possible value for the bits indicating the operand mode instead indicates a prefix byte for the operand which indicates an index register.

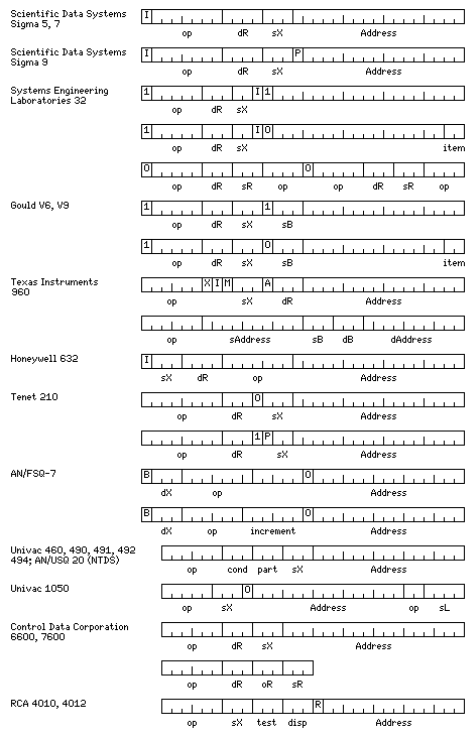
This diagram shows the instruction formats for the justly famed Cray-1 computer:




Addresses point to 64-bit words, and thus two additional packet select bits are required in branch instructions, where a unit of 16 bits is referred to as a packet.

Memory-reference instructions have only a four-bit opcode field because only loads and stores are done on main memory, following the same practice as with RISC computers.

And now, here are seven computers with 32-bit instructions, and four others with 30-bit instructions:




The first instruction format shown is the instruction format for the SDS Sigma series of computers, for which all instructions were 32 bits long.

This computer was an attempt to design a computer that might be arguably just as good as a System/360, only simpler in design, and therefore cheaper to make. It used EBCDIC internally for characters, and it used the same floating-point format as the System/360 for positive numbers, but negative numbers were indicated by two's complementing the corresponding positive number instead of by merely setting the sign bit.

It had sixteen general registers, each 32 bits in length. Only registers 1 through 7 could be used as index registers, allowing a three-bit index field in instructions.

The address space covered 524,288 bytes; thus, instructions contained a 17-bit word address, and did not require a base register. If an instruction was indexed, the index was taken to be in terms of the operand length, thus allowing the use of bytes and halfwords in positions other than the leftmost part of a word through indexing.

Registers 12 through 15 were a packed decimal accumulator, so in decimal instructions, the destination register field instead served as a length field.

Character string instructions omitted the index field, and lengthened the address field by three bits to allow a byte address. The destination register field (usually) indicated a pair of registers, containing the other information needed to specify the instruction.

Register-to-register instructions were simply memory-reference instructions referring to the first 16 words of memory; but the registers were kept in fast semiconductor memory instead of core. The machine also had memory mapping and protection; the memory was divided into 256 equal parts of 2,048 bytes, and each one had an associated two-bit storage key.

The general registers were also used as the floating-point registers.

The Sigma 9 computer allowed the use of a 22-bit physical address by using a page select register; the first bit of the address field of the instruction could be used to choose between page zero of memory, if it was zero, and the 65,536-word page of memory indicated by the six bits of the page select register if it was one. This was used in Real Extended Addressing, so that programs could use the entire memory of the computer, and is somewhat reminiscent of the addressing methods used with such computers as the PDP-8 or the Honeywell 316. The Sigma 9 also provided, in another addressing mode, virtual memory for applications remaining within the original 131,072-word address space of the architecture.

Later, Xerox made the Xerox 550 and Xerox 560 computers which were compatible successors of this series; there was also a computer called the Xerox 530, but this was a computer with a 16-bit word and a 48-bit floating-point type, and so it was not compatible.

The second instruction format shown in this illustration is that of the SEL 32 computer from Systems Engineering Laboratories. A 32-bit word could contain either one 32-bit instruction or two 16-bit instructions, but a 32-bit instruction had to be aligned on a 32-bit boundary, which is similar to how instructions in 48-bit or 60-bit words were handled on many Control Data computers.

This computer is particularly interesting for how it avoided wasting space in the addresses of aligned operands. The first of the two instruction formats shown is that for byte operands, which have a byte address. In the second format, a word address is given, and so the last two bits are used to indicate whether the operand is a 32-bit word, the left or right 16-bit halfword of the target word, or a 64-bit doubleword. Before that, an extended addressing mode was available that allowed addresses to be extended by one bit, from 19 bits to 20 bits, by allowing values in the index register to have a larger range. As well, memory mapping was included in the SEL 32/70, which allowed a 24-bit physical address.

A later version of the computer made after SEL was acquired by Gould had an alternate base register mode, which allowed 24-bit virtual addresses.

The Texas Instruments 960 computer, in its single-address instruction format, included a bit, shown here labelled by the letter M, to indicate an immediate operand, and an A bit to indicate the instruction would use, of two banks of registers included in the architecture to provide for fast context switching, the one opposite to that currently in use, in addition to bits for indirection and indexing. The index register was indicated by a three-bit field in the instruction.

The fourth instruction format shown is that of the Honeywell 632 computer. This computer was a minicomputer with a 32-bit word, unlike the Sigma computers, which were mainframes with a 32-bit word, but the basic memory reference instruction format for this computer included the same fields, with the same lengths, but in a different order, as that of the Sigma.

The fifth set of instruction formats shown are the memory-reference instruction formats of the TENET 210 computer. This computer had the interesting feature that if the indirect bit of the instruction was set, only registers 1, 2, and 3 instead of registers 1 through 7 could be used as index registers, so that a bit was available to distinguish between pre-indexing and post-indexing.

The sixth set of instruction formats shown are those of the legendary AN/FSQ-7 computer, made for IBM for the SAGE (Semi-Automatic Ground Environment) defense project. This is the computer whose front panel, with a record number of blinking lights, was prominently featured on the television show *The Time Tunnel*, as well as appearing in many other movies and television shows. This computer had the unusual characteristic that its arithmetic instructions dealt with vectors of two 16-bit numbers.

The seventh instruction format shown is that of the Univac 460, 490, 491, 492, and 494 computers, as well as the NTDS (Navy Tactical Data System) computer, also by Univac, which was essentially identical to the Univac 460. This architecture also served as the basis for the AN/UYK-7 computer (which, however, had its word size expanded to 32 bits).

One interesting feature of this architecture is that most instructions contained a three-bit field indicating when the next instruction should be skipped. This field was also used in a conditional branch instruction, however, so it would seem it was primarily useful in actually allowing single instructions performing a calculation to be skipped, instead of facilitating conditional branches.

The eighth instruction format is that of the Univac 1050. It competed against the IBM 1401. However, this computer did not have a field bit in the six-bit characters it stored; instead, there was a length field in the instruction, as later used on the IBM System/360. All instructions were 30 bits in length; the machine's internal registers, stored in low memory, were called tetrads, and were 24 bits in length. Also unlike the 1401, but like the 360, it performed binary arithmetic as well as decimal arithmetic. Interestingly enough, when Univac replaced this computer with a newer one aimed at the same markets and applications, often used as an auxilliary computer to larger systems, it replaced it with the 9200 and 9300 computers, based on the IBM System/360 Model 20, described above.

The ninth set of instruction formats shown is that of the Control Data 6600 computer and its successors. This computer had a 60-bit word; 15-bit and 30-bit instructions could have any positions within the word, but a 30-bit instruction could not cross a word boundary; this restriction simplified the circuitry for instruction fetch and decoding, thus speeding the computer.

The tenth instruction format is that of the RCA 4100 series of computers. In addition to a six bit opcode, a three-bit index field, and a 14-bit address displacement, a three bit field indicated the condition to be tested after the instruction, with another three bit field indicating how many words to skip if the condition was met. And one bit was used to indicate returning from an interrupt, and apparently also for cooperative multitasking.

[Next] [Up] [Previous]
[Next] [Up] [Previous]
[Next] [Up/Previous]