

Objectives:

1. To review binary representation for unsigned integers
2. To introduce octal and hexadecimal shorthands
3. To review binary representation for characters
4. To introduce binary representations for signed integers
5. To introduce the IEEE 754 representation floating point representation
6. To discuss the basic process of doing floating point arithmetic
7. To review/introduce binary representations for sounds, graphics, movies

Materials:

1. Dr. Java
2. RGB Demo Applet
3. Floating point examples handout

I. Introduction

A. One of the key suggestions in Von Neumann's landmark paper was that computer systems should be based on the binary system, using just the two bit values 0 and 1 to encode information.

1. This is consistent with the nature of the physical devices used to construct computer systems, and has contributed to the high reliability we associate with digital systems.
2. Although Von Neumann was thinking just in terms of representing integers in binary, we have since learned to represent many other kinds of information this way, including text, sounds, and graphics.
3. To this day - over 6 decades later, and through several changes in fundamental technology - this idea still dominates the field of computing.

B. At the outset, we should note that there is a crucial distinction between a NUMBER (an abstraction) and a REPRESENTATION of a number (a symbol).

1. For example, the number we call four can be represented by the following symbols:

FOUR QUATUOR 4 IV IIII 11_3 100_2 etc.

2. On the other hand, the symbol 100 could represent 4 (as above) or one hundred (decimal system) or 9 (trinary system) or for any one of an infinite variety of possible values.

C. Most often, we represent numbers by a symbol based on a positional notation system. The development of positional notation was a crucial advance in the development of modern mathematics. In a positional system, we have a base or radix (such as 2 for binary, 3 for trinary, or ten for decimal). We make use of a limited set of symbols for representing values $0 \dots \text{radix} - 1$.

1. For example, in binary, we use two symbols (0,1); in octal (base 8) we use 8 (0,1,2,3,4,5,6,7); in hexadecimal (base 16) we use 16 symbols (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) etc.

2. When there is any possibility of confusion, we denote the radix of a numerical symbol by a subscript (written using decimal notation!) - eg:
 9010_{16}

D. Further, in a positional system the value of a symbol depends on where it is written. The rightmost symbol (digit) has the value (symbol_value)*1; the one next to it has value (symbol_value) * radix etc. Thus, the radix-2 (binary) number 01000001 is equivalent to the decimal number:

$$0*128 + 1*64 + 0*32 + 0*16 + 0*8 + 0*4 + 0*2 + 1*1 = 65_{10}$$

E. In this and subsequent lectures, we will consider how various kinds of information can be represented in binary. First, we limit ourselves to unsigned integers.

II. Review of Internal Encoding for Unsigned Binary Integers

A. The normal way of encoding unsigned binary integers is to use a straightforward place-value system, in which the bits are assigned weights $2^0, 2^1, 2^2$ etc. Thus, for example, the binary integer 1011 would be interpreted as

$$\begin{array}{r} 1 * 2^0 = \quad 1 \\ + 1 * 2^1 = \quad 2 \\ + 0 * 2^2 = \quad 0 \\ + 1 * 2^3 = \quad 8 \\ \text{---} \\ 11 \end{array}$$

B. Note that with n bits we can store values in the range $0 .. (2^n) - 1$

C. Conversion between decimal and binary

1. To go from binary to decimal, we use the basic approach outlined above: multiply the rightmost bit by 2^0 , the next bit by 2^1 , the next bit by 2^2 etc. Add the products. (It helps if you memorize the powers of 2)

Example: convert 10011101 to decimal (157)

Exercise: 10101010 (170)

2. To go from decimal to binary, we can use successive division: Divide the decimal number by 2. The remainder is the rightmost bit of the binary equivalent. Divide the quotient by 2. The new remainder is the second rightmost bit. Divide the new quotient by 2. The new remainder is third rightmost bit ... Continue until the quotient is 0.

Example: convert 238 to binary

Example: assume a certain machine uses 8 bits for integers. Consider the following problem:

$$\begin{array}{r}
 11001000 \\
 + \underline{11001000} \\
 \hline
 10010000
 \end{array}
 \qquad
 \begin{array}{r}
 200 \\
 + \underline{200} \\
 \hline
 144 \text{ !!}
 \end{array}$$

The error arises because 8 bits cannot represent the correct sum. (Here, we detect overflow by the fact that there was carry out of the most significant bit position.)

F. Other arithmetic operations can be done on binary numbers in a manner analogous to the decimal operations, but using the binary tables - though other methods are actually used in practice, as we shall see.

III. Octal and Hexadecimal Shorthands

A. By now you are probably tired of writing 1's and 0's all the time. Writing numbers in binary is tiring, and it is very easy to make mistakes. On the other hand, converting numbers between decimal and binary is complex, so at the hardware level we like to work with the binary form.

B. Consider, for a moment, the radix-8 (octal) number system. Since there are 8 different symbols in this system, octal numbers are about as easy to write as decimal numbers. Further, because 8 is a power of 2, it is very easy to convert between binary and octal notations.

1. Binary to octal: group binary number into groups of three bits, starting from the right. Each group will now represent a value in the range 0 .. 7 - i.e. an octal digit.

Example: 11 000 111 --> 307₈

Exercise: 10 101 100 (254)

2. Octal to binary: convert each digit to three bits:

Example: 146 --> 001 100 110

Exercise: 321 (011 010 001)

3. In a sense, then octal becomes a form of shorthand for binary. Any given bit can be quickly recovered from the octal representation.

Example: What is bit 4 of the number represented by 246 ?

Observe: octal digit 0 is bits 0..2; digit 1 is bits 3..5. So bit 4 is the middle bit of middle digit - i.e. middle bit of 4 - i.e. middle bit of 100 - i.e. 0

Exercise: Bit 5 of 172? (1)

- C. Another system often used as a shorthand for binary is hexadecimal - base 16. The hex digits are written 0 .. 9, A, B, C, D, E, F

Example: A13E --> 1010 0001 0011 1110

Exercise: 4FB7 (0100 1111 1011 0111)

Exercise: 1100 0101 1111 0100 (C5F4)

- D. Historically, the instruction set architecture of the DEC PDP-11 (the system on which C was developed) was such that using the octal shorthand was the more natural notation. (Octal digit boundaries fell at field boundaries within the instruction.)

1. As a result, octal notation found its way into the C programming language and its descendants. (A numeric constant written with a leading 0 is interpreted as octal - e.g. 010 is 8!)
2. However, for most modern architectures (including MIPS) the documentation convention is to use hexadecimal, rather than octal, as a shorthand for binary. Therefore, hexadecimal is the main shorthand we will use in this course.

- E. Always remember, though, that octal and hexadecimal are simply shorthands for making the writing and reading of binary numbers easier. The internal representation is always binary.

IV. Representations for Signed Integers

- A. The method we have developed thus far for representing integers in binary only allows us to represent integers ≥ 0 . Obviously, we also need a way to represent negative numbers.
- B. In decimal, we represent negative numbers by representing their absolute value and preceding it by a negative sign. This, in effect, means that we have added one more symbol to our set of numerical symbols (a 10% increase in the size of our set of symbols). On a computer using binary internally, this is highly undesirable, since we would then have to have three different symbols to represent numbers with, instead of two (a 50% increase in the size of our set of symbols.)
- C. Instead, we take advantage of the fact that the number of bits in a binary number is fixed by the hardware design (the machine's word length).
1. If we wish to represent signed numbers, then, we can reserve the leftmost bit as a sign bit, and interpret it as follows:

0 --> the number is ≥ 0 1 --> the number is negative
 2. A consequence of reserving one bit for the sign is that the largest positive number we can represent is about half as big as what we could have represented with the same number of bits using unsigned notation (we've given up one bit for the sign.) (This is why some languages (e.g. C++) allow integers to explicitly be declared as unsigned - to allow all bits to be used for representing magnitude.)
 3. An important ramification of a choice like this is that we have to be careful about OVERFLOW.
 - a) We have already seen that, with a fixed number of bits allocated for representing an integer, an addition can produce a result that is too big to be represented in the specified number of bits.

b) But now, this problem can manifest itself without any carry, but instead as a result of the wrong sign.

Example: represent 100 (decimal) as an 8 bit signed binary number:

01100100 (rightmost bit is sign - 0)

Now take the sum $100 + 100 = 200$. This results in overflow, because we only have seven bits available for representing the magnitude of the number (since one bit is reserved for the sign)

In fact, the result from performing addition in the usual way is 11001000 which looks like -56, since the rightmost bit is 1 !

D. With this basic encoding (using the leftmost bit for the sign), there are actually three possible schemes for representing signed numbers. All agree on using the leftmost bit in the number to encode the sign - with 0 encoding + and 1 encoding - but they differ in what they do with the remaining bits.

1. Sign magnitude
2. One's complement
3. Two's complement

We will only discuss the first and last schemes, since one's complement is rarely used.

E. For the following examples, we will assume the use of 8 bits to represent an integer - 1 for the sign and 7 for the value. This is just so we don't have to write lots of 0's or 1's on the board - most typically we use 32 bits (or even 64 bits) to represent an integer.

V. Sign-Magnitude Representation

A. The simplest way to represent signed numbers is called sign-magnitude. It is based on the method we use with decimal numbers:

1. To represent a positive number in (say) 8 bits, represent its magnitude as seven bits and prefix a 0.
2. To represent a negative number in (say) 8 bits, represent its magnitude as seven bits and prefix a 1.

Example: +65 --> 1000001 (7 bits) --> 01000001
-65 --> 1000001 (7 bits) --> 11000001

Exercise: + 72, -72 (01001000, 11001000)

3. To change the sign of a number, invert the sign bit.

B. Some Ramifications of the Sign-Magnitude Representation

1. There are two representations for zero: 00000000, 10000000

(The latter can be used as an "undefined value" to be stored in uninitialized variables. If the hardware ever reads it from memory, a trap can be generated to indicate that the programmer has used an undefined variable.)

2. Range of values representable with n bits: $-(2^{(n-1)} - 1) .. 2^{(n-1)} - 1$

3. Unfortunately, while simple for us, sign-magnitude is awkward in hardware. For example, the algorithm to add two sign magnitude numbers looks like this:

If signs agree: add magnitudes, retaining signs. Overflow occurs if the sum magnitude exceeds $2^{(n-1)} - 1$ - i.e. if there is carry out of the most significant bit of the magnitude sum.

If signs differ: find number with greater magnitude and retain its sign.
 Form magnitude of result by subtracting smaller magnitude from larger.
 (Overflow cannot occur.)

Examples: $0\ 0000001\ 1$
 $+ \underline{0\ 0000001} + \underline{1}$
 $0\ 0000010\ 2$
 $1\ 1000000\ -64$
 $+ 0\ 0000001\ 1$

becomes: $1\ 1000000\ -64$
 $- \underline{1\ 0000001} - \underline{-1}$
 $1\ 0111111\ -63$

$1\ 0000001\ -1$
 $+ \underline{1\ 1111111} + \underline{-127}$
 $1\ 0000000$ OVERFLOW - carry out of most
 significant bit of magnitude sum
 (Result looks like zero!)

Exercise: 00100000
 $+ \underline{10000010}$
 (00011110)

4. Actually, multiplication and division are more straightforward: multiply/
 divide magnitudes and set sign = xor of original signs.

C. Sign-magnitude is rarely used for integers, but is used as part of the
 representation for reals as we shall see later.

VI. Two's Complement Representation

A. The most commonly used scheme for representing signed numbers is called two's complement.

1. The basic idea is this: when we write a positive number, we are really writing an abbreviation for the representation, in the sense that the number could be thought of as having infinitely many 0's to the left -

e.g. The representation for 42 (using 8 bits) is 00101010 - but we could think of this as standing for

... 0000000000000000101010

infinitely many 0's

2. To form the representation for a negative number, think of what would happen if we subtracted its absolute value from 0 ($-X = 0 - X$). E.g. if we subtracted the representation for 42 from 0, we would get

... 000000000000000000000000
- ... 0000000000000000101010
... 1111111111111111010110

infinitely many 1's

a) If we abbreviate down to 8 bits, we get 11010110

b) Actually, we don't have to work with infinitely many bits - to negate an n bit number, it suffices to subtract it from the (n+1)-bit number

2^n and then truncate to n bits - e.g.

10000000 Representation for 2^8

- 00101010 42

011010110 = 11010110 in 8 bits

^---- Discard this bit

3. This is called the TWO'S-COMPLEMENT representation - i.e. we say that 11010110 is the two's complement of 00101010.

- a) If we apply this process again to 11010110, we end up with 00101010 again - which is what we would expect, given that $0-(0-X)=X$ necessarily holds.) That is, we can say that 11010110 is the two's complement of 00101010 and vice versa.
- b) Observe that if we add the n-bit representation of a number and its two's complement, without treating the sign bit specially, the result is 2^n . However, if we truncate the result to n bits, it becomes 0 - which is surely what we want since $X + (-X) = 0$

$$\begin{array}{r} 00101010 \\ +11010110 \\ \hline 100000000 = 00000000 \text{ in 8 bits} \end{array}$$

B. An Easy Way to Represent Numbers in Two's complement

1. To represent a positive number in n bits, represent it as an unsigned number using n-1 bits and prefix a 0. (If this cannot be done, then the number is too big to be represented as an n-bit two's complement number.)

Example: Represent +65 as an 8-bit two's complement number.

Since it is positive, represent it as a 7 bit number: 1000001 and prefix a 0: 01000001

2. To represent a negative number in n bits, represent its absolute value (a positive number) in n bits as above, then invert all the bits (including the sign bit) and add 1.

Example: Represent -65 as an 8-bit two's complement number.

Since it is negative, represent its absolute value in 8 bits as above: 01000001.

$$\begin{array}{r} \text{Flip all the bits: } 10111110 \\ \text{Add one: } \quad \quad +00000001 \\ \hline 10111111 \end{array}$$

Exercise: + 72, -72 (01001000, 10111000)

3. To change the sign of a number, simply flip all the bits, then add 1.

Example: Negate -65

Flip all the bits: 01000000
Add one: +0000001
01000001 (+65, as expected)

4. Observe: if we are using n bits, then X , if negative, looks like the unsigned number

$$2^n + X$$

Example: the representation for -65: 10111111 looks like 191 if viewed as an unsigned number, which is $2^8 (256) + -65!$

5. This last observation raises an important point: to interpret a pattern of bits, we need to know what representation is being used.

That is, if you were to see the bit pattern 10111111:

- if you thought it was using the unsigned number representation, you would interpret it as 191
- if you thought it was using the two's complement representation, you would interpret it as -65
- if you thought it was using the sign-magnitude representation, what would you interpret it as?

ASK

-63!

C. Some Ramifications of the Two's-Complement Representation

1. There is one representation for 0: all bits zero (e.g. 00000000 using 8 bits)

To see this, consider what happens if we try to form -0 by flipping all the bits of 0 and adding 1.

Flip all the bits: ...11111111
 Add one: +00000001
 ...00000000

2. Range of values representable with n bits: $-2^{(n-1)} \dots 2^{(n-1)} - 1$

Note that this is asymmetric: there is one more negative value than positive value. For example, using 8 bits -128 is representable as 10000000, though +128 is not representable.

Hence, trying to negate the most negative possible number gives overflow)

Example: The most negative possible number in 8 bits is -128

Flip all the bits: 01111111
 Add one: +00000001
 10000000

So, in 8 bits, $-(-128) = -128$ - which is overflow.

3. To add two 2's complement numbers: add them as if they were unsigned (ie treat the sign as if it were the most-significant bit of an unsigned number.) Discard (and ignore) any carry out of the sign position.

a) To see why this works (in the absence of overflow), observe the following:

(1) If we add two non-negative numbers X and Y, the operation is the same as unsigned addition in any case.

(2) If we add two negative numbers X and Y, X looks like the unsigned number $X + 2^n$ and Y looks like the unsigned number $Y + 2^n$. The sum therefore looks like $X + Y + 2^n + 2^n$.

However, since there is (of necessity) a carry out of the most significant position, one of the 2^n 's is cancelled

and the sum looks like $X + Y + 2^n$, which is correct since the sum of two negative numbers is itself a negative number.

(3) If we add negative number (X) to a non-negative number (Y), X looks like the unsigned number $X + 2^n$ and the sum looks like the unsigned number $X + Y + 2^n$.

(a) If the final sum is negative, this is the correct two's complement representation.

(b) If the final sum is positive, there is (of necessity) a carry out of the most significant position which cancels the 2^n term, so the result looks like $X + Y$ as desired.

b) The above analysis assumed no overflow. There is overflow if carry out of sign position (leftmost bit) \neq carry in to sign position (leftmost bit)

Examples:	01100100	100 ₁₀
	+01100100	100 ₁₀
	11001000	-56 ₁₀

Carry out of sign position = 0; carry into sign position = 1 -
OVERFLOW

	00001000	8
	+11111111	-1
	00000111	7

Carry out of sign position = carry into sign position = 1 - no overflow

As the above examples illustrate, it is possible to have

- No carry out of sign position, but overflow nonetheless
- Carry out of sign position without overflow

(Of course, it is also possible to have neither carry out of the sign position nor overflow, or to have both. All four combinations of carry out of the sign and overflow are possible - they are distinct concepts!)

Exercise: 11000001 -63
 +01000000 64
 00000001 1

c) Proof of overflow rule:

Carry in to sign \neq carry out of sign \Leftrightarrow overflow

- (1) If we add two positive numbers, there will never be carry out of the sign (since both signs are zero). The sign of the result will be 1 (which must be an overflow) iff there is carry in to the sign - that is, there is overflow iff the two carries differ.
- (2) If we add two negative numbers, there will always be carry out of the sign (since both signs are one). The sign of the result will be 0 (which must be an overflow) iff there is no carry in to the sign - that is, there is overflow iff the two carries differ.
- (3) If we add two numbers of unlike sign, there can never be overflow (since the result must lie somewhere between these two values, so since they are representable the result must be as well). Further, there will be carry out of the sign iff there is carry in to the sign (since just one sign is one) - i.e. the two carries will always be the same.

4. To subtract: Negate (flip all the bits, add 1) the subtrahend and add
5. Multiplication and division are more complex. One approach is to complement negative values so as to do the whole operation with positive values, then if the two original operands were of opposite sign then

complement the result. Or, the whole operation can be done in 2's complement - we'll discuss this later.

D. Two's complement is the preferred scheme for integer arithmetic on most machines, though many (including MIPS) use Sign-Magnitude for floating point. (Note that, since the arithmetic algorithms are wired into the hardware, the choice is made by the hardware designer and cannot easily be altered in software.)

VII. Internal encoding for reals (floating point numbers)

A. Thus far we have confined our discussion to integers. How can we represent real numbers? It turns out that this is an area where architectures diverge widely. We will discuss general principles, and then a particular approach that has become widely accepted.

B. A real number is stored internally as a mantissa times a power of some radix - i.e.

$$m * r^e$$

This is familiar to us - "scientific notation" uses $r = 10$ - example:

$$6.02 * 10^{23}$$

1. CPU architects have several basic options regarding the format of the mantissa, m :
 - a) It may be a pure fraction. That is, there is an assumed binary point just to the LEFT of the first bit of the mantissa.
 - b) It may be a number between 1 and 2, with an assumed binary point between the first and second bits. (This is analogous to the convention in scientific notation, which uses a mantissa between 1 and 10)
 - c) It may be an integer, with an assumed binary point just to the RIGHT of its last bit.

2. The exponent e specifies a power of some radix, r , by which the mantissa is to be multiplied.

a) The radix is often 2, but in some machines is 4, 8, or 16

Example: If a machine uses 2 as the radix of its exponent, then a number with binary mantissa 1.1 and exponent 2 is interpreted as

$$1.1_2 * 2^2 = 6$$

However, if the exponent radix is 16 (as on the IBM mainframe architecture, then the interpretation is

$$1.1_2 * 16^2 = 384_{10}$$

b) Of course, the choice of what radix to use for the exponent is an architectural choice, involving tradeoffs between RANGE and PRECISION. Using a larger radix means a larger range of values can be represented, at the cost of some precision.

C. In an attempt to deal with data exchange problems arising from a diversity of floating point representations used by different manufacturers, the IEEE has developed a floating point standard (Standard 754) that is used by most systems

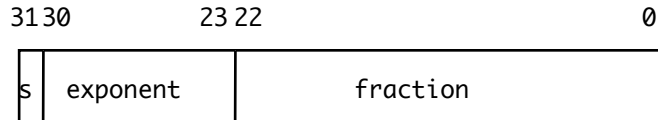
1. Older architectures developed prior to this standard may use a different format (in fact, the proliferation of such formats was what led to the development of IEEE 754).

a) One important architectures that is still in use that utilize a pre-IEEE 754 floating point format is the IBM mainframe architecture.

b) All current microprocessor architectures in wide use utilize the IEEE 754 format, including MIPS, IA32, and PowerPC

c) The Java Virtual Machine utilizes the IEEE 754 format.

2. The standard provides two different formats for single and double precision numbers. We will consider the single precision representation in this system in some detail.



a) s = sign of mantissa: 0 = +, 1 = -

b) exponent = exponent as power of 2, stored as an 8-bit unsigned number in excess 127 form - i.e. value in this field stored is 127 + true value of exponent.

Examples: true exponent = 0; stored exponent = 127
 true exponent = 127; stored exponent = 254
 true exponent = -126; stored exponent = 1

The extreme values of the stored exponent (stored exponent = 0 and 255) are reserved for special purposes to be described below, so the largest possible true exponent is 127 and the smallest is -126.

c) The significand (the magnitude of mantissa) is normalized to lie in the range $1.0 \leq |m| < 2.0$. This implies that the leftmost bit is a 1. It is not actually stored - the 23 bits allocated are used to store the bits to the right of the binary point, and a 1 is inserted to the left of the binary point by the hardware when doing arithmetic. (This is called hidden-bit normalization, and is why this field is labelled "fraction".) In effect, we get 24 bits of precision by storing 23 bits.

Example: stored fraction = 00000000000000000000000
 true significand = 1.0
 stored fraction = 11111111111111111111111
 true significand = 1.11111111111111111111111

d) As noted above, certain exponent values are reserved for special purposes, and when they appear the interpretation of the significand changes:

(1) Stored exponent = 0

(a) If the significand is zero, then the number represented is 0.0 (Since an all-zero stored significand represents 1.0, a special case is needed to represent zero.)

(b) If the significand is non-zero, then we have a denormalized number; no hidden bit is inserted and the true exponent is taken as -126 (This allows representing very small numbers with the loss of some precision)

(2) Stored exponent = 255

(a) If the significand is non-zero, then the representation is not a number (NaN). Any use of NaN in an arithmetic operation always produces NaN as the result.

(b) If the significand is zero, then the number represented is +/- infinity (depending on the sign bit.) The hardware correctly propagates infinity in arithmetic - e.g. infinity + anything is infinity; infinity - any finite value is infinity, but infinity - infinity is not a number (NaN).

DEMO (using Dr. Java)

```
float f = 5.0e38f + 5.0e38f;  
f          (be sure to omit ;)  
f+f  
f-f  
f*f  
f/f
```

e) Examples: HANDOUT 1.0, -0.375, binary -> decimal

0.0: 0 00000000 000000000000000000000000

1.0: 0 01111111 000000000000000000000000 (1.0 * 2⁰)

0.5₁₀: 0 01111110 000000000000000000000000

(1.0 * 2⁻¹) - note that 0.5₁₀ = 0.1₂

0.75: 0 01111110 100000000000000000000000

(1.1 * 2⁻¹) - note that 0.75₁₀ = 0.11₂

3.0: 0 10000000 100000000000000000000000 (1.1 * 2¹)

-0.375 1 01111101 100000000000000000000000

(-1.1 * 2⁻²) - note that -0.375₁₀ = -0.011₂

1 10000011 110000000000000000000000 =

- 1.11₂ * 2⁴ = -11100₂ = -28.0₁₀

3. Range of values for single precision:

-largest finite positive: 0 11111110 111111111111111111111111 =

1.111111111111111111111111 * 2¹²⁷ or ~2¹²⁸ (about 3 * 10³⁸)

-smallest normalized positive: 0 00000001 000000000000000000000000 =

1.000000000000000000000000 * 2⁻¹²⁶ (about 1 * 10⁻³⁸)

(The precision of both of the above is 24 bits = ~ 7 decimal places)

- smallest positive: 0 00000000 000000000000000000000001 =

.000000000000000000000001 * 2⁻¹²⁶ or 2⁻¹⁴⁹ (about 2 * 10⁻⁴⁵)

(But precision is only one bit!)

4. IEEE 754 also defines a double precision floating point standard, which represents a number using 64 bits: 1 for the sign, 11 for the exponent (excess 1023), and 52 for the fraction.

VIII. Floating point Arithmetic

A. Arithmetic on floating point numbers is, of course, much more complex than integer (or fixed-point) arithmetic.

1. It is not necessary to have hardware provisions for doing floating point arithmetic - it is possible to code subroutines to perform basic floating point operations using a combination of integer arithmetic and shift operations.
 - a) Historically, when integrated circuit technology was more limited that was often the case.
 - b) It still is the case for low-end microprocessors used in embedded systems. (In fact, such systems often do not need to do floating point arithmetic!)
2. Historically, many CPU's relegated floating point arithmetic to a separate processor, often called a "floating point coprocessor". On older systems, this was often a separate chip, which may or may not be installed in a given computer. (If not, floating point arithmetic would be done in software.) Today, the "coprocessor" is actually often part of the main CPU chip. The term coprocessor remains in use for historical reasons, and because floating point operations often use their own register set.
3. What we have to say in this section is applicable regardless of how floating point operations are physically implemented. We will briefly consider the basic task facing floating point processors, but will not look at the algorithms in detail.

B. Recall that a floating point number is actually represented internally by two fixed-point numbers: a mantissa and an exponent. That is, it is of the form:

$$m * r^e$$

We will assume use of the IEEE standard - i.e. $1 \leq m < 2$, with only the fraction part stored; $r = 2$.

C. Floating point addition entails the following steps:

1. Reinsertion of the hidden bit. Though normalized floating point numbers are STORED without the 1 to the left of the binary point, the arithmetic unit can work on an internal form of the number with the hidden bit explicitly present.

(Of course, if an operand is zero or a denormalized number, a 0 is inserted in the hidden bit position.)

2. Denormalization: if the exponents of the two operands differ, then the operand with the smaller exponent must be shifted right to line up the implied binary points. The larger exponent will then be the exponent of the result.

Example: $1.00 * 2^0 + 1.00 * 2^{-1}$ must be converted to:

$1.00 * 2^0 + 0.10 * 2^0$ before adding

3. The addition proper. Note that, since the mantissa is represented using sign-magnitude, this may actually entail performing a subtraction!
4. Renormalization: There are three possibilities for the result of the addition/subtraction step

a) The result could be correct and normalized as it stands

b) There could be carry out from the leftmost mantissa bit:

Example: $1.10 * 2^0 + 1.00 * 2^0$ yields $0.10 * 2$ plus a carry out

In this case, the mantissa is shifted right (bringing the carry bit in), and the exponent is increased.

Example: The final result of the above is $1.01 * 2^1$

c) There could be no carry out, and the leftmost bit of the mantissa could be zero - i.e. the result could be unnormalized. (This only occurs when adding numbers of unlike signs)

Example: $1.10 * 2^0 + -1.01 * 2^0$ yields $0.01 * 2^0$ (with no carry out)

In this case, the mantissa must be shifted left one or more places (until its leftmost bit is 1) to renormalize it, and the exponent must be decreased for each shift.

Example: The final result of the above is $1.00 * 2^{-2}$

Note: To reduce the loss of precision in cases like this, the hardware often includes one or two GUARD BITS to the right of the mantissa which "catch" bits shifted out during denormalization and make them available for renormalization

Note: If the exponent would be reduced below the smallest permissible value, the result is left in denormalized form.

5. Preparation for storage.

- a) If the number has been shifted right during renormalization, then a bit will have been shifted out, and will be caught by the guard bits. Moreover, the guard bits may contain bits that were shifted out during initial denormalization which are properly part of the infinite-precision result.
- b) IEEE 754 defines various ROUNDING MODES that control how to handle the guard bits:
 - (1) Round toward zero: the guard bits are discarded. (Also called truncation.)
 - (2) Round to nearest: round the result to the nearest representable value - e.g. if the guard bits are 11, then add 1 to the least significant bit of the result. Ties are broken by rounding toward zero.
 - (3) Round toward plus infinity: if the result is positive, and the guard bits are non-zero, add one to the least significant bit of the result - else discard the guard bits.

(4) Round toward minus infinity: same, but round if result is negative.

c) In any case, the the hidden bit is removed prior to storing the result.

D. Floating point subtraction can be converted to an addition problem by simply negating the subtrahend and then proceeding as above.

E. Floating point division and multiplication are - relatively speaking - simpler than addition and subtraction.

1. The basic rule for multiplication is

a) Reinsert the hidden bit.

b) Multiply the mantissas

c) Add the exponents

d) If necessary, normalize the product by shifting right and increase the exponent by 1.

(Note that if the mantissas are normalized to begin with, they will lie in the range: $1 \leq m < 2$

Therefore, the product of the mantissas will lie in the range:
 $1 \leq m < 4$

So at most one right shift is needed.)

e) Store the result less the hidden bit after appropriate rounding.

f) Additional complexity arises when one considers cases like an exponent too large/too small, or when one or both of the mantissas are denormalized to begin with. We will not discuss these, though a hardware designer must consider them.

2. The basic rule for division is

- a) Reinsert the hidden bit.
- b) Divide the mantissas
- c) Subtract the exponents
- d) If necessary, normalize the quotient by shifting left and decrease the exponent by 1.

(Note that if the mantissas are normalized to begin with, they will lie in the range: $1 \leq m < 2$

Therefore, the quotient of the mantissas will lie in the range

$$0.1_2 < m < 2.0_2$$

So at most one left shift is needed.

- e) Store the result less the hidden bit after appropriate rounding.
- f) As with multiplication, additional complexity arises when one considers cases like an exponent too large/too small, or when one or both of the mantissas are denormalized to begin with. We will not discuss these, though a hardware designer must consider them.

F. As can be seen, a floating point arithmetic unit needs to be able to add and subtract exponents, and to shift, add, and subtract mantissas. The latter can be done by using the same hardware as for the integer multiply/divide operations, or special, dedicated hardware.

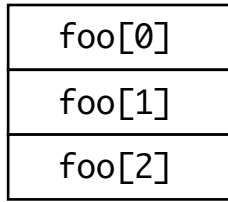
IX. Characters

- A. In principle, any information that can be represented as integers (unsigned or signed) can be represented in binary by converting the integer representation into binary.
- B. We have previously seen how textual information can be represented by assigning integer codes to individual characters - either:
1. ASCII: each character is assigned an 8 bit code in the range 1 .. 127

(With one bit unused - sometimes the additional codes (128 .. 255) are used for various extensions to the character set.)
 2. Unicode: each character is assigned a 16 bit code in the range 1..65535.

X. Aggregates

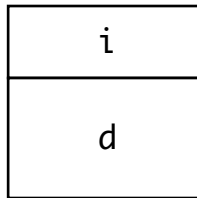
- A. Most programming languages support two basic kinds of aggregate data: arrays and structures/records.
- B. As you know, languages in the C family support declarations of the form `type variable [size]` (C/C++) or `type [] variable` (Java), and languages in the Pascal family support declarations of the form `type variable: array [range] of type`. (Similar forms with other syntax are found in other languages as well.)
1. Such variables are represented by a series of successive memory locations, each holding one element of the array.
 2. For example, the C declaration `int foo [3];` would be represented in memory as.



where each element is represented in the normal way for an integer (typically 32 bits)

C. As you know, languages in the C family support declarations of the form `struct ... { type1 field1; type2 field2 ... }` (C/C++) or `class ... { type1 field1; type2 field2 ... }` (C++/Java).

1. Variables of such type are represented by a series of successive memory locations, each holding one field.
2. For example, the C declaration `struct { int i; double d }` would be represented in memory as.



where `i` is represented in the normal way for an integer (typically 32 bits) and `d` is represented in the normal way for a double (typically 64 bits).

XI. Representing Sounds, Images and Movies

- A. Thus far, we have looked at how numeric and textual information can be represented in binary. Although our senses are involved in our acquiring information of these kinds (e.g. through reading or listening to text), we work with these basically as mental abstractions.
- B. Of course, it is also possible to use binary to represent information perceived by our senses more directly.

1. There exist standard binary representations for sounds, images, and combinations of the two (e.g. movies).

The standards are “tuned” to the characteristics of the human ear and eye - e.g. sound representations are designed around the range of frequencies the human ear can hear, and image representations around the range of colors the human eye can see.

2. There do not, at present, exist standard binary representations for smells, tastes, or information perceived by touch. (Though systems exist that work with all of these, each uses its own system of representation.)

C. Sounds

1. Computers can store and reproduce sounds by storing digitized SAMPLES of the sound signal intensity. A sample represents the intensity of the audio signal at a particular instant of time, which corresponds to the position of the sound cone in a loudspeaker playing the sound or the acoustic pressure on the eardrum of a listener at that instant.

- a) For high quality, these samples must be taken tens of thousands of times per second. There is an important theorem, called the sampling theorem, that says that any sound can be accurately reproduced given samples taken at twice the highest frequency present in it.

-The human ear can hear pitches up to about 20 KHz. CD quality audio is based on 44,000 samples per second.

- To conserve storage, computer systems may use 22,000 or 11,000 or 8,000 samples per second. This loses the upper-end of the frequency spectrum, but is adequate for many purposes

- b) The precision with which the samples are stored is also important.

- High quality systems (e.g. CD's) typically use 16 bit samples, which gives a precision of one part in 32768, (given that the signal oscillates above/below 0) or about .003 %. This level of precision is such that the human ear cannot perceive the difference between

the recorded sound and the original.

- Lower quality systems have used 8 bit samples, which gives a precision of one part in 128, or about 0.8%, which is audible to a trained ear. With increasing capacity/decreasing cost of storage, the rationale for using smaller samples like this has declined.

2. The storage requirements for sound can be reduced by various compression techniques we won't discuss here. One key concept is the distinction between lossless and lossy compression.

a) If sound data is compressed using a lossless strategy, and then decompressed, the result is identical to the original - i.e. nothing is lost. Lossless strategies, of necessity, can only achieve minimal compression - perhaps as good as 2:1.

b) If sound data is compressed using a lossy strategy, and then decompressed, the result will be slightly different from the original. The goal of the designers of the compression scheme is, of course, for the differences to be in an area of the audio spectrum that cannot be noticed by the human ear.

Some compression schemes (e.g. mp3) allow the user to select a tradeoff between degree of compression and overall quality - e.g. more compression can be achieved at the expense of more noticeable compression artifacts.

D. Images

1. As we have seen in earlier courses, pictorial information is displayed by breaking the screen into individual dots, known as PIXELS. The quality of the image is in part determined by the number of pixels per inch (often abbreviated dpi = dots per inch.) This is called the RESOLUTION of the image.

a) Computer monitors typically use a resolution of around 72 dpi.

b) Typical laser printers use 300-600 dpi; some publication-quality printers go to 600 or 1200 dpi or more.

2. For black and white graphics, each pixel can be represented by a single bit in memory.
3. For gray scale graphics, each pixel can be represented by a small integer (often a single byte) representing a degree of lightness or darkness. For example, using one byte per pixel:
0 = black 255 = white 128 = medium gray
4. For color graphics, each pixel is represented by three small integers, representing the intensity of each of the three primary colors (red, green, and blue) at that point.

a) The most sophisticated systems currently available store 24 bits for each pixel - 8 for each color. This allows for over 16 million colors, ranging from black (all 0's - all colors totally off) to white (all 1's - all colors totally on.)

Examples

R	G	B	
11111111	00000000	00000000	Pure red
11111111	11111111	00000000	Pure yellow
10000000	10000000	11111111	Light blue

DEMO: RGB Applet on course page

b) Where the graphics system represents an image on top of a background or another image, an additional byte for each pixel (32 bits in all) can be used to hold a value called alpha - representing the opacity / transparency of the pixel. A pixel with alpha 255 is totally opaque - nothing of the color beneath it shows through; a pixel with alpha 0 is totally transparent - i.e. the underlying color is all that shows - and when alpha is between 0 and 255 the color that

is actually displayed is a blend of the color of the pixel and the color of its background.

c) To conserve storage, some systems store only 8 bits per pixel, where each possible value selects one of 256 predefined colors.

d) Other systems store 16 bits per pixel, allowing a choice of one of 65536 predefined colors.

5. Images are often compressed using various strategies we won't discuss in detail here.

a) The key concept in such strategies is to take advantage of the fact that an image often contains regions of the same or similar color (e.g. a blue sky) which can be stored in a compressed form by storing the basic color and information about which pixels contain it, rather than storing the full color information for each pixel.

b) As with audio compression, image compression schemes can be either lossless or lossy - for example gif and png are lossless, while jpeg is lossy. As was the case with the audio schemes, there is a tradeoff between amount of compression and quality of the resulting image - though, in the case of images, even lossless schemes do manage to achieve significant compression.

E. Movies

1. Current television technology is based on redrawing the screen 30 times per second (in the US), or 25 times per second (in Europe and Asia). Celluloid movies (as displayed in a theatre) typically consist of 24 images per second.

a) Each such image is called a FRAME.

b) The frame rates used by television or movies are high enough to ensure that the human eye does not perceive the individual frames, but gets the perception of overall smooth motion.

2. Thus, movies can be represented by a series of individual frames (using image representation) - plus an associated sound track.

The frame rate (number of frames per second) may be limited by the hardware or by storage capabilities; but a frame rate that corresponds to that used for TV or films is desirable since it avoids the visual appearance of flicker.

3. As you can imagine, the storage requirements for video information can be huge. Instead of using a lower frame rate (which results in perceptible flicker or worse), the storage (and transmission time) requirements can be significantly reduced by various compression techniques we will not discuss in detail here, beyond noting two typical targets for compression.
 - a) Normally, the difference between successive frames of a video (representing a time interval of $1/30$ or $1/24$ of a second) are quite small; thus, one standard compression strategy is to store a subset of the frames (called the KEY FRAMES) en toto, and then to store the remaining frames in terms of the DIFFERENCE relative to the most recent key frame.
 - b) Since any given frame is an image, it can be compressed using an image compression scheme as well.