CPE 323 REVIEW
DATA TYPES AND NUMBER REPRESENTATIONS IN MODERN COMPUTERS

Aleksandar Milenković

The LaCASA Laboratory, ECE Department, The University of Alabama in Huntsville

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Objective

Review numeral systems, storing and interpreting numbers in modern computers, including unsigned and signed integers and arithmetic operations on them, BCD representation, and representation of fractional numbers (fixed-point and floating-point).

Contents

1 Numeral Systems: Decimal, binary, hexadecimal, and octal

We ordinarily represent numbers using positional decimal number system that has 10 as its base (also known as base-10 or denary system). A positional numeral system is a system for representation of numbers by an ordered set of numeral symbols (called digits) in which the value of a numeral symbol depends on its position - for example, the "ones place", "tens place", "hundreds place," and so on. Note: other notations such as Roman are not positional. The decimal numeral system uses 10 symbols (digits): 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. It is the most widely used numeral system, perhaps because humans have ten fingers over both hands.

A decimal number 456 could also be represented as four hundreds (10^2), 5 tens (10^1), and 6 ones (10^0) and can be written as follows:

$$456_{10} = 4 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0 = 400 + 50 + 6$$

position	2	1	0
weight	10^2	10^1	10^0
digit	4	5	6

The binary positional numeral system, or the base-2 (radix-2) number system, represents numeric values using only two symbols, 0 and 1. These symbols can be directly implemented using logic gates and that is why all modern computers internally use the binary system to store information.

For example, 10111_2 represents a binary number with 5 binary digits. The left-most bit is known as the most significant bit (msb) – in our case it has bit position 4. The rightmost bit is known as the least-significant bit (lsb) or bit at position 0. This binary number 10111_2 can be converted to a corresponding decimal number as follows:

$$10111_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 2 + 1 = 23_{10}$$

position	4	3	2	1	0
weight	2^4	2^3	2^2	2^1	2^0
digit	1	0	1	1	1

Hexadecimal (base-16, hexa, or hex) is a positional numeral system with the base 16. It uses sixteen distinct symbols, the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F (or *a* through *f*) to represent values ten to fifteen, respectively.

For example, a hex number $1A3_{16}$ corresponds to:

$$1A3_{16} = 1 \cdot 16^2 + 10 \cdot 16^1 + 3 \cdot 16^0 = 256 + 160 + 3 = 419_{10}$$

To convert the hex number into its binary format, simply replace each hex symbol with its 4-bit binary counterpart (0 – 0000_2, 1 – 0001_2, . . ., 9 – 1001_2, A – 1010_2, B – 1011_2, C – 1100_2, D – 1101_2, E – 1110_2, F – 1111_2).

$1A3_{16} = 1\_1010\_0011_2$ (the '_' character is here just for easier visualization)

Hexademical numbers are used as a human friendly representation of binary coded values, so they are often used in digital electronics and computer engineering. Since each hexadecimal

digit represents four binary digits (bits), it is a compact and easily translated shorthand to express values in base two.

Octal (also base-8) is a number system with the base 8 that uses 8 distinct symbols 0, 1, 2, 3, 4, 5, 6, and 7. An octal number 372_8 corresponds to:

$$372_8 = 3 \cdot 8^2 + 7 \cdot 8^1 + 2 \cdot 8^0 = 192 + 56 + 2 = 250_{10}$$

2 Conversion to binary from other numeral systems

To convert a decimal positive number to its binary equivalent, the number is divided by two, and the remainder is the least-significant bit at position 0. The result is again divided by two and the remainder is the next bit at position 1. This process repeats until the result of division becomes zero.

For example, to convert 23_{10} to binary, the following steps are performed:

```
Operation       ⌊ Remainder     (Position)
23 ÷ 2 = 11     ⌊ 1             (bit 0)
11 ÷ 2 = 5      ⌊ 1             (bit 1)
5 ÷ 2 = 2       ⌊ 1             (bit 2)
2 ÷ 2 = 1       ⌊ 0             (bit 3)
1 ÷ 2 = 0       ⌊ 1             (bit 4)
```

Reading the sequence of remainders from the bottom up gives the binary numeral 10111_2.

This method works for conversion from any base, but there are better methods for bases which are powers of two, such as octal and hexadecimal given below.

As described above, to convert a binary number to a decimal number use the following approach:

$$10111_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 2 + 1 = 23_{10}$$

Octal and hexadecimal numbers are easily converted to binary numbers and vice versa. For example, an octal number 372_8 is converted into binary by simply replacing each octal digit with its binary counterpart, $011\_111\_010_2$. To get a corresponding hex representation, create groups of 4 bits starting from the least significant bit. By replacing each group with a corresponding hex digit, a hex representation is created (see below).

$372_8 = 011\_111\_010_2 = 0FA_{16}$

$1011\_0101_2 = 265_8 = B5_{16}$.

3 Storing and interpreting information in modern computers

An *n*-bit binary number can differentiate between 2^n things. In digital computers information and space are organized in bytes. A byte has 8 bits and can represent up to $2^8 = 256$ things.

Definitions:

- 1 byte = 8 binary digits = 8 bits (e.g. 1001_0011_2)
- ½ byte = 1 nibble = 4 bits
- In 16-bit computers 1 word = 2 bytes (16 bits).
- In 32-bit computers 1 word = 4 bytes (32 bits), a half-word is 2 bytes (16-bits), and a double word is 8 bytes (64 bits).

Meaning of bits and bytes is assigned by the convention. Some examples of common encoding formats are as follows:

- 1 byte ASCII = one of 256 alphanumeric or special-purpose text characters (definitions are in the ASCII table, see http://en.wikipedia.org/wiki/ASCII)
- 1 byte = one short unsigned integer (0 – 255)
- 1 byte = one short signed integer (-128 – 127)
- 1 byte = two Binary Coded Decimal (BCD) digits (i.e. one per nibble)
- 1 byte = two hexadecimal (hex) digits (one per nibble)
- 1 byte = eight individual bits (e.g. 8 status flags)
- 2 bytes = one unsigned integer (0 – 65,535)
- 2 bytes = one signed integer (-32,768 – 32,767)
- 4 bytes = one unsigned long integer (0 – 4,294,967,295)

For example, the IAR Software Development Environment for MSP430 assumes the following conventions for standard C data types (see Table 1).

Table 1. C data types in IAR, sizes, ranges, and alignments in memory

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
signed int	16 bits	-32768 to 32767	2
unsigned int	16 bits	0 to 65535	2
signed long	32 bits	-2^{31} to $2^{31}-1$	2
unsigned long	32 bits	0 to $2^{32}-1$	2
signed long long	64 bits	-2^{63} to $2^{63}-1$	2
unsigned long long	64 bits	0 to $2^{64}-1$	2
float	32 bits		2
double	64 bits		2

4 Integers

The term integer is used in computer engineering/science to refer to a data type which represents some finite subset of the mathematical integers. Integers may be unsigned or signed.

An *n*-bit unsigned integer is defined as follows:

$$A = A_{n-1}A_{n-2} \dots A_1A_0 = A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \cdots + A_1 \cdot 2^1 + A_0 \cdot 2^0$$

An *n*-bit unsigned integer can encode 2^n numbers that represent the non-negative values from 0 ($A_{n-1}A_{n-2} \dots A_1A_0 = 00 \dots 00$) through 2^n-1 ($A_{n-1}A_{n-2} \dots A_1A_0 = 11 \dots 11$).

There are three different ways to represent negative numbers in a binary numeral system. The most common is **two's complement**, which allows a signed integer type with *n* bits to represent numbers from $-2^{(n-1)}$ through $2^{(n-1)}-1$. Two's complement arithmetic is convenient because there is a perfect one-to-one correspondence between representations and values, and because addition, subtraction and multiplication do not need to distinguish between signed and unsigned integer types (thus same hardware resources can be used to perform these operations).

The other possibilities for representing signed numbers are *sign-and-magnitude* and *ones' complement*. *Sign and Magnitude* representation uses *n-1* bits to convey the magnitude with the most significant bit (MSB) used for sign (0 for +, 1 for -). The problem with this approach is that there exists two representations for value 0 ($00...00_2$ for a positive 0 and $10...00_2$ for a negative 0). With *one's complement* representation a negative binary number is created by applying the bitwise NOT to its positive counterpart. Like *sign-and-magnitude* representation, ones' complement has two representations of 0: 00000000_2 (+0) and 11111111_2 (−0). As an example, the ones' complement form of 00101011_2 (43) becomes 11010100_2 (−43). The range of signed numbers using ones' complement in a conventional eight-bit byte is -127_{10} to $+127_{10}$.

We focus on two's complement because it is the dominant way to represent signed integers today.

4.1 Two's Complement

In the two's complement notation, a positive number is represented by its ordinary binary representation, using enough bits so that the MSB bit, also known as the sign bit, is 0. An *n*-bit integer in two's complement is defined as follows:

$$A = A_{n-1}A_{n-2} \dots A_1A_0 = -A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \cdots + A_1 \cdot 2^1 + A_0 \cdot 2^0$$

When the MSB bit is zero, $A_{n-1} = 0$, a positive number in range from 0 ($A_{n-1}A_{n-2} \dots A_1A_0 = 00 \dots 00$) to ($2^{n-1} - 1$) ($A_{n-1}A_{n-2} \dots A_1A_0 = 01 \dots 11$) is represented. When the MSB bit is one, $A_{n-1} = 1$, a negative number from -2^{n-1} ($A_{n-1}A_{n-2} \dots A_1A_0 = 10 \dots 00$) to -1 ($A_{n-1}A_{n-2} \dots A_1A_0 = 11 \dots 11$) is represented. So, the range of signed numbers that can be represented with *n* bits in 2's complement is from -2^{n-1} to $(2^{n-1} - 1)$. There is only one representation of 0 ($00....00_2$).

Table 2 shows ranges of signed and unsigned integers depending on the number of bits used for representation.

Table 2. Common integer data types and ranges.

Bits	Name	Range
n	*n*-bit integer (general case)	*Signed:* (-2^{n-1}) to $(2^{n-1}-1)$
		Unsigned: 0 to (2^n-1)
8	byte, octet	*Signed:* −128 to +127
		Unsigned: 0 to +255
16	halfword, word	*Signed:* −32,768 to +32,767
		Unsigned: 0 to +65,535
32	word, doubleword, longword	*Signed:* −2,147,483,648 to +2,147,483,647
		Unsigned: 0 to +4,294,967,295
64	doubleword, longword, long long, quad, quadword	*Signed:* −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
		Unsigned: 0 to +18,446,744,073,709,551,615
128	octaword	*Signed:* −170,141,183,460,469,231,731,687,303,715,884,105,728 to +170,141,183,460,469,231,731,687,303,715,884,105,727
		Unsigned: 0 to +340,282,366,920,938,463,463,374,607,431,768,211,455

4.2 Calculating two's complement

The two's complement operation is the negation operation. In finding the two's complement of a binary number, the bits are first inverted, or "flipped", by using the bitwise NOT operation and then the constant 1 is added to that value.

Let's assume 8-bit signed binary numbers. Consider A, A = 23_{10} = 17_{16} = 00010111_2. We would like to find the two's complement for A or (-A). The steps are shown below.

```
A:      0001 0111      (23)
-------------------------------------
Ā:      1110 1000      (first complement of A)
+              1       (1)
=====================================
-A:     1110 1001      (-23)
```

So the two's complement representation of -23_{10} = 11101001_2 = $E9_{16}$.

You may shorten this process by using hexadecimal representation as follows:

```
A                       :     17
one's complement of A :     E8
                        +      1
                  ------------------
                              E9
```

A shortcut to manually convert a binary number into its two's complement is to start at the least significant bit (LSB), and copy all the zeros (working from the LSB toward the MSB) until the first 1 is reached; then copy that 1, and flip all the remaining bits. This shortcut allows you to convert a number to its two's complement without first forming its ones' complement. For example: the two's complement of "00010111" is "11101001", where the underlined digits are unchanged by the copying operation.

4.3 Arithmetic Operations: Addition

Adding two's-complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding A=15 and B=-5 if *n*=8 (8-bit numbers):

```
       1111 1111 (carry bits=C8C7...C1)
-------------------------------
A:      0000 1111      (15)
B:    + 1111 1011      (-5)
===============================
S:      0000 1010      (10)
```

This process depends upon restricting arithmetic operations to 8 bits of precision; a carry to the 8th (nonexistent) bit position, C_8, is ignored producing arithmetically correct result in two's complement. The Carry (C) flag is set to the most significant carry bit, $C=C_8$. Please note that the Carry flag does not have any particular meaning for signed numbers in 2's complement. However, if 8-bit values represent unsigned numbers (15 and 251 in our example), the Carry flag indicates that the result of addition is outside the range of numbers that can be represented (from 0 to 255). And indeed, 251+15=266, but our 8-bit result contains 266 % 256 = 10. Using the carry bit as the most significant field of the 9-bit result would give a correct result of the unsigned addition.

The last two bits of the carry row (C_8 and C_7) contain vital information for 2's complement arithmetic: whether the calculation resulted in an arithmetic overflow (V flag), which means that the result is outside the range of the 2's complement system. An overflow condition exists when a carry (an extra 1) is generated into but not out of the far left sign bit, or out of but not into the sign bit. As mentioned above, the sign bit is the leftmost bit of the result.

If the last two carry bits are both 1's or both 0's, the result is valid; if the last two carry bits are "1 0" or "0 1", a sign overflow has occurred. Conveniently, an XOR operation on these two bits can quickly determine if an overflow condition exists. As an example, consider the 4-bit addition of A=7 and B=3:

```
       0111 (carry bits)
-------------------------------------------
A:      0111     (7)
B:    + 0011     (3)
===========================================
S:      1010     (-6) Invalid in 2's complement
```

In this case, the far left two carry bits are "01", which means there was a two's-complement addition overflow. That is, ten is outside the permitted range for 4-bit numbers in 2's complement, which ranges from −8 to 7.

An alternative way to calculate Overflow (V) flag is to consider the sign bits of the operands and the result. Let us assume we add two 8-bit integers in two's complement, A and B, and the result is an 8-bit S. The MSB bits of the operands and the result can be used to calculate the V flag. An overflow in case of addition occurs if one of the following two conditions are met: (a) both operands A and B are positive numbers (A_7=0 and B_7=0), and the result S is a negative number (S_7=1), and (b) both operands A and B are negative numbers (A_7=1 and B_7=1), and the result S is a positive number (S_7=0). Consequently, overflow flag in case of addition can be calculated as follows:

$$V_{Add} = \overline{A_7} \cdot \overline{B_7} \cdot S_7 + A_7 \cdot B_7 \cdot \overline{S_7}$$

In addition to the Carry flag and the Overflow flag, two more flags are commonly used in digital computers, Zero (Z) and Negative (N). The Z flag is set when the result of the current arithmetic or logic operation is equal to 0 (S=0) and reset otherwise. The negative flag keeps information about the sign bit of the result S (N=S_7).

4.4 Arithmetic Operations: Subtraction

Computers usually use the method of complements to implement subtraction for numbers in two's complement. But although using complements for subtraction is related to using complements for representing signed numbers, they are independent. Direct subtraction works with two's-complement numbers as well. Like addition, the advantage of using two's complement is the elimination of examining the signs of the operands to determine if addition or subtraction is needed. For example, subtracting -5 from 15 is really adding 5 to 15, but this is hidden by the two's-complement representation:

```
R:     11110 000 (borrow bits)
-------------------------------
A:      0000 1111      (15)
B:    - 1111 1011      (-5)
===============================
S:      0001 0100      (20)
```

Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrow row (R); overflow occurs if they are different. Alternatively, the overflow flag can be calculated as follows:

$$V_{Sub} = \overline{A_7} \cdot B_7 \cdot S_7 + A_7 \cdot \overline{B_7} \cdot \overline{S_7}$$

An easy way to understand this equation is to ask yourself the following: when does an overflow occurs in operation S = A – B. If both numbers A and B are positive (A_7=0 and B_7=0) or negative (A_7=1 and B_7=1), the overflow cannot occur because the result will be within the range we can represent. The overflow occurs if A is a positive and B is a negative and the result S turns out to be negative – since A is a positive, subtracting a negative B would result in an even greater positive result – however, if the result exceeds the range of positive numbers that can be represented, the sign bit will be set and that is a negative number in two's complement. This condition is captured by the first term in the equation above. Another scenario when overflow occurs is if A is a negative number, B is a positive number, and the results is a positive number. Again, expected result is a negative number, but if it exceeds the range of negative numbers that can be represented, the result will turn out to be a positive number. The second term in the equation above captures that scenario. If either term is computed to a logic one, the overflow bit is set.

Another example is a subtraction operation where the result is negative: 15 − 35 = −20:

```
R:     11100 000 (borrow bits)
-------------------------------
A:      0000 1111      (15)
B:    - 0010 0011      (35)
===============================
S:      1110 1100      (-20)
```

Arithmetic, logic, and shift operations on operands are performed in Arithmetic Logic Units or ALUs. Typically an adder is used for both addition and subtraction operations as mentioned above. To perform a subtract operation, S = A – B, an equivalent operation in two's complement can be carried out as follows: S = A + ($\overline{B} + 1$). Remember, we already showed that –B = $\overline{B} + 1$. The first-complement of B is computed by inverting the input operand B, and adding a constant one is achieved by setting the input Carry bit to 1 (C_0=1). This way, subtract operations are implemented using adder logic. Often that is the case and the flags are set following the rules for additions.

Using the previous example A=15 and B=35:

```
       00011 111 (carry bits)
-------------------------------
A:      0000 1111
B̄:    + 1101 1100
      +         1
===============================
S:      1110 1100      (-20)
```

The result is identical, S=1110_1100$_2$ (-20). The flag bits are set as follows:
C=C_8=0; Z=0; V=C_8 xor C_7=0 xor 0=0; N=S_7=1.

A careful reader would notice that the most significant bit of the borrow row R_8=1 is different from the most significant bit of the carry row (C_8=0) in the examples above. Processor

manufacturers may decide on the meaning of the Carry bit for subtraction. If the carry flag, C, should keep the value that is equivalent to R_8 and the adders are used for subtractions, then simply the bit C_8 is inverted and assigned to the carry bit ($C=\overline{C_8}$) when a subtraction is performed. In other architectures the carry bit simply takes the value of C_8 as is. In this course we will always assume the latter approach.

4.5 Arithmetic Operations: Multiplication

The product of two *n*-bit numbers can potentially require 2*n* bits to encode the result. If the precision of the two two's-complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take 6 × −5 = −30 and *n*=4. First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```
          0000 0110  (6)     Multiplicand
        × 1111 1011  (-5)    Multiplier (M7M6M5M4M3M2M1M0)
=========================
               110           Multiplicand*M0*2^0
+             110            Multiplicand*M1*2^1
+              0             Multiplicand*M2*2^2
+            110             Multiplicand*M3*2^3
+           110              Multiplicand*M4*2^4
+          110               Multiplicand*M5*2^5
+         x10                Multiplicand*M6*2^6
+        xx0                 Multiplicand*M7*2^7
 ========================
P:      x11110010     (-30)
```

4.6 Binary Coded Decimal Numbers

Binary-coded decimal (BCD) is an encoding for decimal numbers in which each decimal digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of logic circuits needed to implement mathematical operations and a relatively inefficient encoding—it occupies more space than a pure binary representation. For example, a byte can represent positive integers in range from 0 to 99 if using packed BCD representation, whereas regular binary represents positive integers in range from 0 to 255. Though BCD is not as widely used as it once was, decimal fixed-point and floating-point are still important and still used in financial, commercial, and industrial computing.

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

```
Decimal:    0     1     2     3     4     5     6     7     8     9
BCD:     0000  0001  0010  0011  0100  0101  0110  0111  1000  1001
```

Thus, the BCD encoding for the number 127 would be: 0001_0010_0111

Since most computers store data in eight-bit bytes, there are two common ways of storing four-bit BCD digits in those bytes:

- each digit is stored in one nibble of a byte, with the other nibble being set to all zeros, all ones (as in the EBCDIC code), or to 0011 (as in the ASCII code)
- two digits are stored in each byte (so called packed BCD encoding).

Unlike binary encoded numbers, BCD encoded numbers can easily be displayed by mapping each of the nibbles to a different character. Converting a binary encoded number to decimal for display is much harder involving integer multiplication or divide operations.

5 Fraction Numbers

To represent real numbers in computers two major approaches are used: fixed-point and floating-point representation.

5.1 Fixed-point

For fixed-point representation, the binary radix point is assigned to a fixed location in a byte (or word). Fixed-point numbers are useful for representing fractional values when the executing processor has no floating-point unit (FPU) and when fixed-point representation provides improved performance and sufficient accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU.

In binary fixed-point numbers, each magnitude bit represents a power of two, while each fractional bit represents an inverse power of two. Thus the first fractional bit is ½, the second is ¼, the third is ⅛ and so on. Signed fixed-point numbers in two's complement format are defined as follows, where *m* and *f* are the number of bits in the integer portion (M) and the fraction portion (F), respectively:

$$A = A_{m-1}A_{m-2} \dots A_0.A_{-1}A_{-2} \dots A_{-\mathrm{f}} =$$
$$-A_{m-1} \cdot 2^{m-1} + A_{m-2} \cdot 2^{m-2} + \cdots + A_0 \cdot 2^0$$
$$+A_{-1} \cdot 2^{-1} + \cdots A_{-\mathrm{f}} \cdot 2^{-\mathrm{f}}$$

The upper bound for this representation is $2^{(m-1)}-2^{(-f)}$ and the lower bound is given by $-2^{(m-1)}$.

Unsigned fixed-point numbers are defined as follows (the range is from 0 to 2^m-2^{-f}):

$$A = A_{m-1}A_{m-2} \dots A_0.A_{-1}A_{-2} \dots A_{-\mathrm{f}} =$$
$$A_{m-1} \cdot 2^{m-1} + A_{m-2} \cdot 2^{m-2} + \cdots + A_0 \cdot 2^0$$
$$+A_{-1} \cdot 2^{-1} + \cdots A_{-\mathrm{f}} \cdot 2^{-\mathrm{f}}$$

Let's consider the following unsigned number with 5-bit integer portion and 3-bit fraction:

$00000.101_2 = 0.625_{10}$

Granularity of precision is a function of the number of fractional bits assigned. Thus, if f=3, the smallest fraction is $2^{-3} = 0.125$. The maximum number that can be represented in this notation (8 bits, 5 for integer portion, 3 for fraction) is 63.875 (encoded as 11111.111) and the minimum is 0.0 (encoded as 00000.000).

5.2 Floating-point

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU (Central Processing Unit) and FPU (Floating-Point Unit) implementations. The standard defines formats for representing floating-point numbers (including negative zero and denormal numbers) and special values (infinities and NaNs), together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions (including when the exceptions occur, and what happens when they do occur).

Floating-point numbers are typically packed into a computer datum as the sign bit, the exponent field, and the significand (mantissa), from left to right. For the IEEE 754 binary formats they are apportioned as shown in Table 3.

Table 3. Floating-point types as defined by the IEEE 754 standard.

Type	Sign	Exponent	Exponent bias	Significand	Total
Half (IEEE 754-2008)	1	5	15	10	16
Single	1	8	127	23	32
Double	1	11	1023	52	64
Quad	1	15	16383	112	128

Let's take a look at a single precision floating-point format that requires 32 bits as follows: S_EEEEEEEE_FFFFFFFFFFFFFFFFFFFFFFF, where

S: sign bit, E: exponent bits; F – mantissa (significand) bits

position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
field	S	E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0	F_{-1}	F_{-2}	F_{-3}	F_{-4}	F_{-5}	F_{-6}	F_{-7}	F_{-8}	F_{-9}	F_{-10}	F_{-11}	F_{-12}	F_{-13}	F_{-14}	F_{-15}	F_{-16}	F_{-17}	F_{-18}	F_{-19}	F_{-20}	F_{-21}	F_{-22}	F_{-23}

The 32-bit representation consists of three parts. The first bit is used to indicate if the number is positive or negative. The next 8 bits are used to indicate the exponent of the number, and the last 23 bits are used for the fraction.

The value of the single-precision number is determined as follows:

Value = $(-1)^S\ 2^{(E-127)}*(1.F)$

Converting decimal digits to IEEE binary floating point is a little tricky. There are 3 steps.

- The first step in the conversion is the simplest. This is determining the sign bit. If the number is positive, then the sign bit is 0, S=0, otherwise it is 1, S=1.
- The next eight digits are used to express the exponent, which we will figure out last.
- The final 23 digits are used to express the fraction.

Let us first illustrate the method for the conversion using a simple example. We will use -210.25 to walk through the conversion. Determining sign bit is easy – this is a negative number, so S=1.

The next step is to convert the absolute value of your decimal number to binary (210.25). It is easiest to focus on the integer portion of the number first, then the fraction. 210 is 11010010 or 128+64+16+2, otherwise expressed as:

$$210_{10} = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 11010010_2$$

Next, we need to convert the decimal part. To do this we have to convert the number into a binary sequence that takes the following form:

$A_{-1} \cdot 2^{-1} + A_{-2} \cdot 2^{-2} + A_{-3} \cdot 2^{-3} + \ldots$

Luckily .25 is 1/4 and so this is easy to convert it to binary:

$0.25_{10} = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = .01_2$

Thus, the decimal 210.25 is represented as 11010010.01 in binary.

$210.25_{10} = 11010010.01_2 = 1.101001001 \cdot 2^7$

The next step is to normalize this number so that only one non-zero decimal place is in the non-fraction part of the number. To do this we must shift the decimal place 7 positions to the left (see above). The number 7 becomes important so we note it. This process leaves us with the number 1.101001001, which is the normalized form 1.F. The fraction F is represented in the last 23 bit places in the 32 bit binary, F=101001001. The fraction is padded with 0's to fill in the full 23 bits - leaving us with F=10100100100000000000000 for the mantissa or significand.

So we now have the first bit S, S=1, and the last 23 bits of the 32 bit sequence. We must now derive the middle 8 bits. To do this we take our exponent (7) and add 127 (the exponent bias for single-precision numbers) to get E=134. We then express this number as an 8 bit binary. This is 10000110 (or 128+4+2). Now we have the middle bits and can stitch together all bits to get a complete single-precision binary representation:

$1\_10000110\_10100100100000000000000_2 = C3524000_{16}$

We can convert this bit sequence back into a number by reversing the process. First we can note by the leading 1 that the number is negative. Next we can determine the exponent. 10000110 is binary for 2+4+128 or 134. 134-127 is 7, so the exponent is 7. Finally we take the last 23 digits, convert them back into the original fraction (adding the preceding 1.) to get:

1.101001001

Moving the decimal place to the right by 7 (corresponding to the exponent) we get:

11010010.01

This binary is equal to 128+64+16+2 + 1/4 or 210.25. Once we apply the negative sign (indicated by the leading bit set to 1) we get our original number:

-210.25

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we need to specify a true zero mantissa to yield a value of zero).

Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized Numbers

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.F \times 2^{-126}$, where S is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.F \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in the IEEE floating-point.

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaNs denote indeterminate operations, while SNaNs denote invalid operations.

Summary

Table 4 summarizes IEEE 754 floating-point representation for fractional numbers.

Table 4. Floating-point representation (b is bias).

Sign (s)	Exponent (e)	Fraction (f)	Value
0	00 ... 00	000....0	+0
0	00 ... 00	00 ... 01 11 ... 11	Positive denormalized real $0.f \times 2^{(-b+1)}$
0	00 ... 01 11 ... 10	xx ... xx	Positive normalized real $1.f \times 2^{(e-b)}$
0	11 ... 11	00 ... 00	+Infinity
0	11 ... 11	00 ... 01 01 ... 11	SNaN
0	11 ... 11	10 ... 00 11 ... 11	QNaN
1	00 ... 00	000....0	-0
1	00 ... 00	00 ... 01 11 ... 11	Negative denormalized real $-0.f \times 2^{(-b+1)}$
1	00 ... 01 11 ... 10	xx ... xx	Negative normalized real $-1.f \times 2^{(e-b)}$
1	11 ... 11	00 ... 00	-Infinity
1	11 ... 11	00 ... 01 01 ... 11	SNaN
1	11 ... 11	10 ... 00 11 ... 11	QNaN

6 References

http://en.wikipedia.org/wiki/Binary_number

http://en.wikipedia.org/wiki/Two's_complement

http://en.wikipedia.org/wiki/IEEE_754

7 Exercises

Problem #1.

Fill in the following table. We consider a 16-bit memory location and you should provide binary, hexademical, decimal (as positive integer), and binary coded decimal representations. Show your work as illustrated for (i). Note: for 4-bit BCD digits if the value is outside the range 0-9 use '?'.

	16-bit binary	Hexadecimal (4 hex digits)	Decimal (unsigned int)	2-byte packed BCD (4-BCD digits)
(i)	**0011.0000.0011.1100**	**303C**	**12,348**	**303?**
(a)		cbf3		
(b)			9,080	
(c)	1001.0100.0010.1100			
(d)		1943		

(i)

Here 12,348 decimal representation is provided. We convert this number into a hexadecimal number by dividing by 16 as follows.

```
12348/16 =      771  ⌊12
771/16 =        48   ⌊3
48/16 =         3    ⌊0
3/16 =          0    ⌊3
```

From hexadecimal representation we can get directly binary and BCD representations.

$12348_{10} = 303C_{16} = 0011\_0000\_0011\_1100_2 = 303?$ ("?" marks an illegal BCD digit).

Problem #2.

What is the range of unsigned and signed integers that can be represented using 16 bits?

Range of 16-bit unsigned integers:_____

Range of 16-bit signed integers (in 2's complement):_____

Consider the following 16-bit hexadecimal numbers given in the second column of the table below. Each of these values can be interpreted as an unsigned 16-bit integer or a signed 16-bit integer represented in 2's complement.

Provide the decimal value for each number and interpretation. Show your work as illustrated in (i).

	16-bit hex	Unsigned int	Signed int (2's complement)
(i)	**A223**	**41507**	**-24029**
(a)	81C2		
(b)	B607		
(c)	39CD		
(d)	0012		

(i) unsigned: $A223_{16} = 10*16^3 + 2*16^2 + 2*16^1 + 3*16^0 = 41{,}507_{10}$

signed: $A223_{16} = 1010.0010.0010.0011_2$ => this is a negative number;

two's complement is: $0101.1101.1101.1101 = 5DDD_{16} = 24{,}029_{10}$ => $A223_{16} = -24{,}029$

Problem #3.

Consider the following arithmetic operations. Find the results and set the flags C, V, N, and Z accordingly. Show your work using hexadecimal or binary representation.

(a) 8-bit, two's complement

$24_{10} + 105_{10}$

(b) 8-bit, two's complement

$(-55)_{10} - 68_{10}$

(c) 16-bit, two's complement

$(-45)_{10} + 88_{10}$

Problem #4.

(a) Convert the following number from decimal to a single-precision IEEE-754 floating point number.

-32.125_{10}

(b) Convert the following IEEE-754 32-bit floating point number from hex/binary to decimal.

$60E3AB00_{16}$

(c) What data type would you need to use to represent the following number, 1,048,576.0625, without losing any significant bits of the mantissa? Explain your answer.

(d) What is the smallest negative normalized half-precision floating-point number? Give its decimal representation. Show your work.

(e) What is the largest positive normalized half-precision floating-point number? Give its decimal representation. Show your work.