Decimal and Binary Representation Systems

Decimal and binary are positional representation systems.

Decimal represents numbers by their coefficients of powers of 10 while binary represents numbers by their coefficients of powers of 2.

For example 321 (in decimal):

- is $3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0 = 321_{10}$
- Decomposing 321 in powers of 2 yields 321$=$ 256 + 64 + 1, or
- $1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ or 101000001_2

More generally, a positive integer is represented in

- decimal by $\Sigma_{i=0}^{n} a_i \times 10^{n-i}$, where the a_i are between 0 and 9
- binary by $\Sigma_{i=0}^{m} b_i \times 2^{m-i}$, where the b_i are either 0 or 1

Binary and Hexadecimal Representation Systems

Writing binary numbers quickly becomes error-prone and unwieldy.

Instead use hexadecimal, the positional representation system with coefficients of powers of 16.

For example $321 = 1 \times 16^2 + 4 \times 16^1 + 1 \times 16^0 = 141_{16}$

To convert from binary to hexadecimal, group bits (abbreviation for binary digits) by groups of 4 starting from the right.

For example $101000001_2 = 1\ 0100\ 0001 = 141_{16}$

Since the coefficients are between 0 and 15, we need new symbols to represent 10-15. They are A (for 10) up to F (for 15).

For example $1\ 1100\ 1011_2 = 1CB_{16}$

Hexadecimal is very convenient to represent 16-bit (4 hex digits), 32-bit (8 hex), and 64-bit (16 hex) numbers.

Some useful powers of two and some notation

- $2^{10} = 1024_{10} \simeq 10^3 = 1\ K$
- $2^{20} \simeq 10^6 = 1\ M$
- $2^{30} \simeq 10^9 = 1\ G$

So, we'll often round-off and talk about, say 16 KB or 64 MB.

The least magnitude bit (or digit) is called lsb least significant bit (the one at the rightmost position of the representation)

The greatest magnitude bit is called msb most significant bit (the one at the leftmost position of the representation)

Representing positive and negative integers

In an n-bit register we can represent 2^n patterns. For example in a 32-bit register we can represent binary *unsigned integers* in the range $[0..2^{32} - 1]$

But we also need a representation (binary) system for *signed integers* with the following properties

1. Equal number of positive and negative numbers
2. Unique representation of zero (and easy test for 0)
3. Easy sign test
4. Easy rules for addition/subtraction

Historically there has been 3 numbering systems that have been used;

- Sign and magnitude (still used for floating point)
- Two's complement (now used in all machines for integer representation)
- One's complement

Two's complement

Positive numbers are represented as in unsigned binary except that the *msb* is always zero.

Zero is represented a a string of 0's.

To represent a negative number, take the representation of its absolute value, flip all 1 bits to 0 and all 0 bits to 1, and add 1 to the lsb using binary arithmetic.

Example: Assume a 4-bit register:

- What is the representation of 6_{10}?
- What is the representation of -6_{10}?
- What is the representation of 0?
- What is the range of representation of positive numbers?
- What is the range of representation of negative numbers?
- How do I recognize whether a number is positive or negative or zero?

Addition and subtraction in Two's complement

Addition

You do not have to check the signs of the operands. Just perform an ordinary binary addition and discard the carry out of the *msb*.

If you add two numbers of opposite signs, everything will always turn out right.

If you add two positive numbers and the result "*appears*" to be negative (i.e., the *msb* is 1) you have an *overflow*. This will generate an exception in your program (see end of quarter).

If you add two negative numbers and the result "*appears*" to be positive, you have an *underflow*.

Subtraction

Take the 2's complement of the subtrahend and add it to the other operand.