Data Representation in Computer Systems

- A bit is the most basic unit of information in a computer.

  – It is a state of "on" or "off" in a digital circuit.

  – Sometimes these states are "high" or "low" voltage instead of "on" or "off".

- A byte is a group of eight bits.

  – A byte is the smallest possible addressable unit of computer storage.

  – The term, "addressable," means that a particular byte can be retrieved according to its location in memory.

- A word is a contiguous group of bytes.

  – Words can be any number of bits or bytes.

  – Word sizes of 16, 32, or 64 bits are most common.

  – In a word-addressable system, a word is the smallest addressable unit of storage.

- A group of four bits is called a nibble (or nybble).

  – Bytes, therefore, consist of two nibbles: a "high-order nibble," and a "low-order" nibble.

- Bytes store numbers when the position of each bit represents a power of 2.

  – The binary system is also called the base-2 system.

  – Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.

– Any integer quantity can be represented exactly using any base (or radix).

POSITIONAL NUMBERING SYSTEMS

- Three numbers are represented as powers of a radix.

$$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$$
$$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$$
$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

CONVERTING BETWEEN BASES

Decimal to binary conversion

- Every integer value can be represented exactly using any radix system.

Division remainder method.

EXAMPLE 2.3

Convert 538_{10} to base 8 using the division-remainder method.

8|538	2	8 divides 538 67 times with a remainder of 2.
8|67	3	8 divides 67 8 times with a remainder of 3.
8|8	0	8 divides 8 1 time with a remainder of 0.
8|1	1	8 divides 1 0 times with a remainder of 1.
0		

Reading the remainders from bottom to top, we have: $538_{10} = 1032_8$.

- This method works with any base.

Convert 147 to binary.

2⌊147	1	2 divides 147 73 times with a remainder of 1.
2⌊73	1	2 divides 73 36 times with a remainder of 1.
2⌊36	0	2 divides 36 18 times with a remainder of 0.
2⌊18	0	2 divides 18 9 times with a remainder of 0.
2⌊9	1	2 divides 9 4 times with a remainder of 1.
2⌊4	0	2 divides 4 2 times with a remainder of 0.
2⌊2	0	2 divides 2 1 time with a remainder of 0.
2⌊1	1	2 divides 1 0 times with a remainder of 1.
0		

Reading the remainders from bottom to top, we have: $147_{10} = 10010011_2$.

- A binary number with N bits can represent unsigned integers from 0 to 2^{N-1}.
- For example, 4 bits can represent the decimal values 0 through 15, whereas 8 bits can represent the values 0 through 255.
- Let's use the division remainder method to again convert 190 in decimal to base 3.

Converting Fractions

- Fractions in any base system can be approximated in any other base system using negative powers of a radix.
- Radix points separate the integer part of a number from its fractional part.

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$
$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= \frac{1}{2} + \frac{1}{4}$$
$$= 0.5 + 0.25 = 0.75$$

EXAMPLE 2.7

Convert 0.34375_{10} to binary with 4 bits to the right of the binary point.

.34375
× 2
0.68750 (Another placeholder)

.68750
× 2
1.37500

.37500
× 2
0.75000

.75000
× 2
1.50000 (This is our fourth bit. We will stop here.)

Reading from top to bottom, $0.34375_{10} = 0.0101_2$ to four binary places.

EXAMPLE

- Convert 3121 to base 3.
- First, convert to decimal:

$$3121_4 = 3 \times 4^3 + 1 \times 4^2 + 2 \times 4^1 + 1 \times 4^0$$
$$= 3 \times 64 + 1 \times 16 + 2 \times 4 + 1 = 217_{10}$$

Then convert to base 3:

3|217	1
3|72	0
3|24	0
3|8	2
3|2	2
0	We have $3121_4 = 22{,}001_3$.

Converting Between Power-of-Two Radices

- Binary numbers are often expressed in hexadecimal—and sometimes octal—to improve their readability.
- Because 16 = 2, a group of 4 bits (called a hextet) is easily recognized as a hexadecimal digit.
- Similarly, with 8 = 2, a group of 3 bits (called an octet) is expressible as one octal digit.
- Using these relationships, we can therefore convert a number from binary to octal or hexadecimal by doing little more than looking at it.

EXAMPLE 2.9

Convert 110010011101_2 to octal and hexadecimal.

110 010 011 101 6 2 3 5	Separate into groups of 3 bits for the octal conversion.
	$110010011101_2 = 6235_8$
1100 1001 1101 C 9 D	Separate into groups of 4 for the hexadecimal conversion.
	$110010011101_2 = C9D_{16}$

If there are too few bits, leading 0s can be added.

Signed integer representation

- There are three ways in which signed binary numbers may be expressed:

  – Signed magnitude,
  – One's complement and
  – Two's complement.
- In an 8-bit word, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit.
- For example, in 8-bit signed magnitude, positive 3 is: 00000011
- Negative 3 is: 10000011.
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  – Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.
- For sign magnitude, the largest integer an 8-bit word can represent is 2^7 - 1, or 127 (a 0 in the high-order bit, followed by 7 1s).

- The smallest integer is 8 1s, or -127.
- Therefore, N bits can represent $-(2^{(N-1)} - 1)$ to $2^{(N-1)} - 1$.
- Signed magnitude arithmetic is carried out using essentially the same methods that humans use with pencil and paper, but it can get confusing very quickly.
- As an example, consider the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude.
- The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one.
- If you consider these rules carefully, this is the method you use for signed arithmetic by hand.

EXAMPLE 2.10

Add 01001111_2 to 00100011_2 using signed-magnitude arithmetic.

```
          1  1  1  1      ⇐ carries
0    1  0  0  1  1  1  1      (79)
0  + 0  1  0  0  0  1  1    + (35)
0    1  1  1  0  0  1  0      (114)
```

- In signed magnitude, the sign bit is used only for the sign, so we can't "carry into" it.
- If there is a carry emitting from the seventh bit, our result will be truncated as the seventh bit overflows, giving an incorrect sum.

EXAMPLE 2.11

Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic.

Last carry overflows and is discarded.

1	←			1	1	1	1		⇐ carries
0		1	0	0	1	1	1	1	(79)
0	+	1	1	0	0	0	1	1	+ (99)
0		0	1	1	0	0	1	0	(50)

We obtain the erroneous result of 79 + 99 = 50.

- Mixed sign addition (or subtraction) is done the same way.
  - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

```
          0 2     0 2
0    0 1 0 1 1 1 0
1  + 0 0 1 1 0 0 1
0    0 0 1 0 1 0 1
```

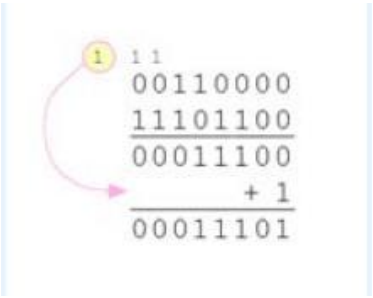
- The sign of the result gets the sign of the number that is larger.
  - Note the "borrows" from the second and sixth bits.

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ complement systems for numeric value representation.

Complement Systems

One′s Complement

- For example, in 8-bit one's complement,
- Positive 3 is: 00000011, negative 3 is: 11111100
  – In one's complement, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for special circuitry for subtraction.
- The difference of two values is found by adding the minuend to the complement of the subtrahend.
- With one's complement addition, the carry bit is "carried around" and added to the sum.
- – Example: Using one's complement binary arithmetic, find the sum of 48 and -19
- We note that 19 in one's complement is 00010011, so -19 in one's complement is: 11101100.



- Although the "end carry around" adds some complexity, one's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- **Two's complement solves this problem.**
- To express a value in two's complement:
- – If the number is positive, just convert it to binary and you're done.

- – If the number is negative, find the one's complement of the number and then add 1.
- Example:
  – In 8-bit one's complement, positive 3 is: 00000011
  – Negative 3 in one's complement is: 11111100
  – Adding 1 gives us -3 in two's complement form: 11111101.
- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
- Example: Using two's complement binary arithmetic, find the sum of 48 and -19.
- We note that 19 in one's complement is: 00010011, so -19 in one's complement is: 11101100, and -19 in two's complement is: 11101101.

```
  1 1
  00110000
+ 11101101
  00011101
```

EXAMPLE 2.18

Add 23_{10} to -9_{10} using one's complement arithmetic.

	1 ←	1	1	1		1	1		⇐ carries
		0	0	0	1	0	1	1	(23)
	+	1	1	1	1	0	1	1	0 + (−9)
The last		0	0	0	0	1	1	0	1
carry is added								+	1
to the sum.		0	0	0	0	1	1	1	0 14_{10}

EXAMPLE 2.19

Add 9_{10} to -23_{10} using one's complement arithmetic.

The last carry is 0 so we are done.

$$\begin{array}{rcccccccccc} 0 & \leftarrow & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & (9) \\ & + & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & + \ (-23) \\ \hline & & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & -14_{10} \end{array}$$

EXAMPLE 2.20

Express 23_{10}, -23_{10}, and -9_{10} in 8-bit binary, assuming a computer is using two's complement representation.

$$\begin{aligned} 23_{10} &= +(00010111_2) = 00010111_2 \\ -23_{10} &= -(00010111_2) = 11101000_2 + 1 = 11101001_2 \\ -9_{10} &= -(00001001_2) = 11110110_2 + 1 = 11110111_2 \end{aligned}$$

EXAMPLE 2.22

Add 9_{10} to -23_{10} using two's complement arithmetic.

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & (9) \\ + & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & + (-23) \\ \hline & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & -14_{10} \end{array}$$

EXAMPLE 2.23

- Find the sum of 23 and −9 in binary using two's complement arithmetic.

Discard carry

1 ←	1	1	1		1	1	1		⇐ carries
	0	0	0	1	0	1	1	1	(23)
+	1	1	1	1	0	1	1	1	+ (–9)
	0	0	0	0	1	1	1	0	14_{10}

EXAMPLE 2.24

Find the sum of 11101001_2 (−23) and 11110111_2 (−9) using two's complement addition.

Discard carry

1 ←	1	1	1	1	1	1	1		⇐ carries
	1	1	1	0	1	0	0	1	(–23)
+	1	1	1	1	0	1	1	1	+ (–9)
	1	1	1	0	0	0	0	0	(–32)