Data Representation

in
Computer Systems

Outline

- Data Organization
  - Bits, Nibbles, Bytes, Words, Double Words
- Numbering Systems
  - Unsigned Binary System
  - Signed and Magnitude System
  - 1's Complement System
  - 2's Complement System
  - Hexadecimal System
- Floating Point Representation
- BCD Representation
- Characters
  - ASCII Code
  - UNICODE
- Other Representations
  - Display colors
  - Audio

Data Organization

Computers use binary number system to store information as 0's and 1's

Bits

- A *bit* is the fundamental unit of computer storage
- A bit can be 0 (off) or 1 (on)
- Related bits are grouped to represent different types of information such as numbers, characters, pictures, sound, instructions

Nibbles

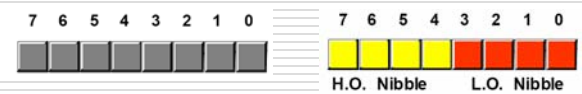
- Nibbles
  - A nibble is a group of 4 bits
  - A nibble is used to represent a digit in Hex (from 0-15) and BCD (Binary-Coded Decimal) (from 0-9) numbers

	BCD	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010		A
1011		B
1100		C
1101		D
1110		E
1111		F

Bytes

Bytes

- A *byte* is a group of 8 bits that is used to represent numbers and characters



- A standard code for representing numbers and characters is ASCII (American Standard Code for Information Interchange)

Byte Size

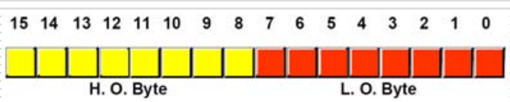
Bytes

- How many different combinations of 0's and 1's with 8 bits can form?
- In general, how many different combinations of 0's and 1's with N bits can form?
- How many different characters that a byte (8 bits) can represent?

Words

Words

- A *word* is a group of 16 bits or 2 bytes that is used to represent non-Roman characters in UNICODE



- An international standard code for representing non-Roman characters like Asian, Greek, and Russian characters is UNICODE

Double Words

Double Words

- A double *word* is a group of 32 bits or 4 bytes or 2 words

Related Bytes

- A *nibble* is a half-byte (4-bit) - hex representation
- A *word* is a 2-byte (16-bit) data item
- A *doubleword* is a 4-byte (32-bit) data item
- A *quadword* is an 8-byte (64-bit) data item
- A *paragraph* is a 16-byte (128-bit) area
- A *kilobyte* (KB) is 2^{10} = 1,024 bytes ≈ 1 K bytes)
- A *megabyte* (MB) is 2^{20} = 1,048,576 ≈ 1 MB
- A *Gigabyte* (GB) is 2^{30} = 1,073,741,824 ≈ 1 GB
- A *Terabyte* (TB) is 2^{40} = 1,099,511,627,776 ≈ 1 TB

Numbering Systems

- Unsigned number system
- Signed binary Systems
  - Signed and magnitude system
  - 1's complement system
  - 2's complement system
- Hexadecimal system

Binary Number System

- base 10 -- has ten digits: 0,1,2,3,4,5,6,7,8,9
  - positional notation

$$2401 = 2 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$$

- base 2 -- has two digits: 0 and 1
  - positional notation

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 8 + 4 + 0 + 1 = 13$$

Binary Positional Notation

If

$$N = b_{n-1} b_{n-2} \cdots b_1 b_0$$

then

$$N = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0 \times 2^0$$

Positional Notation – Convert base 2 or 16 to base 10

Position	...	4	3	2	1	0
Positional Value	...	10^4	10^3	10^2	10^1	10^0
	...	10000	1000	100	10	1
			2	4	0	1

$2\times1000 + 4\times100 + 0\times10 + 1\times1 = 2401_{10}$

Position	...	8	7	6	5	4	3	2	1	0
Positional Value	...	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	...	256	128	64	32	16	8	4	2	1

...	256	128	64	32	16	8	4	2	1
					1	0	1	1	0

16 + 0 + 4 + 2 + 0 = 22

BinHex Application --
http://cms.dt.uh.edu/faculty/ongards/links/links.php?id=1

Unsigned Binary Code

Use for representing integers without signed (natural numbers)

0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Number of Bits Required in Unsigned Binary Code

- What is the range of values that can be represented with *n* bits in the Unsigned Binary Code?

$$[0, 2^n-1]$$

- How many bits are required to represent a given number N in decimal?

$$\text{Min. Number of Bits} = \log_2(N+1)$$

Decimal to Binary Conversion

- **Suppose we want to convert the decimal number 190 to base 3.**
  - We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and $81 \times 2 = 162$.
  - Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ -\underline{162} \\ 28 \end{array} \quad = 3^4 \times 2$$

Decimal to Binary Conversion

- **Converting 190 to base 3...**
  - The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
  - The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

```
  190
- 162  = 3^4 × 2
   28
-  27  = 3^3 × 1
    1
-   0  = 3^2 × 0
    1
```


Decimal to Binary Conversion

- **Converting 190 to base 3...**
  - $3^1 = 3$ is again too large, so we assign a zero placeholder.
  - The last power of 3, $3^0 = 1$, is our last choice, and it gives us a difference of zero.
  - Our result, reading from top to bottom is:
    $190_{10} = 21001_3$

```
  190
- 162  = 3^4 × 2
   28
-  27  = 3^3 × 1
    1
-   0  = 3^2 × 0
    1
-   0  = 3^1 × 0
    1
-   1  = 3^0 × 1
    0
```


Decimal to Binary Conversion

- **Converting 190 to base 3...**
  - First we take the number that we wish to convert and divide it by the radix in which we want to express our result.
  - In this case, 3 divides 190 63 times, with a remainder of 1.
  - Record the quotient and the remainder.

```
3 | 190   1
     63
```


Decimal to Binary Conversion

- **Converting 190 to base 3...**
  - 63 is evenly divisible by 3.
  - Our remainder is zero, and the quotient is 21.

```
3 | 190   1
  3 | 63  0
      21
```

Decimal to Binary Conversion

- **Converting 190 to base 3...**
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

    $190_{10} = 21001_3$

Divisor	Dividend	Remainder
3	190	1
3	63	0
3	21	0
3	7	1
3	2	2
	0	

Successive division by 2

- What is representation of 79_{10} in binary?

Divisor	Dividend	Remainder
2	79	
2	39	1
2	19	1
2	9	1
2	4	1
2	2	0
2	1	0
	0	1

Therefore $79_{10} = 101111_2$

Reverse Positional Notation

Positional Value	512	256	128	64	32	16	8	4	2	1	
				1	0	0	1	1	1	1	
					+ 0	+ 0	+	+	+	+ 1	= 79

Since 64 < 79 5

15 - 8 Since 8 < 15 Since 4 < 7 Since 2 < 3 2 = 1

BinHex Application --
http://cms.dt.uh.edu/faculty/ongards/links/links.php?id=1

Decimal to Binary Conversion

- 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:

$$0.4710_{10} = 4 \times 10^{-1} + 7 \times 10^{-2} + 1 \times 10^{-3}$$

$$0.110_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= \tfrac{1}{2} + \tfrac{1}{4}$$

$$= 0.5 + 0.25 = 0.75$$

Decimal to Binary Conversion

- As with whole-number conversions, you can use either of two methods: a subtraction method and an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n - 1, where n is our radix, and work our way along using larger negative exponents.

Decimal to Binary Conversion

- **The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.**
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - Of course, this method works with any base, not just binary.

$$
\begin{array}{rl}
 & 0.8125 \\
- & 0.5000 = 2^{-1} \times 1 \\
\hline
 & 0.3125 \\
- & 0.2500 = 2^{-2} \times 1 \\
\hline
 & 0.0625 \\
- & 0 = 2^{-3} \times 0 \\
\hline
 & 0.0625 \\
- & 0.0625 = 2^{-4} \times 1 \\
\hline
 & 0
\end{array}
$$

Decimal to Binary Conversion

- **Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.**
  - The first product carries into the units place.

$$
\begin{array}{r}
.8125 \\
\times \quad 2 \\
\hline
1.6250
\end{array}
$$

Decimal to Binary Conversion

- **Converting 0.8125 to binary . . .**
  - Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

$$
\begin{array}{r}
.8125 \\
\times \quad 2 \\
\hline
1.6250
\end{array}
\qquad
\begin{array}{r}
.6250 \\
\times \quad 2 \\
\hline
1.2500
\end{array}
\qquad
\begin{array}{r}
.2500 \\
\times \quad 2 \\
\hline
0.5000
\end{array}
$$

Decimal to Binary Conversion

- **Converting 0.8125 to binary . . .**
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
  .8125
 ×    2
 1.6250

  .6250
 ×    2
 1.2500

  .2500
 ×    2
 0.5000

  .5000
 ×    2
 1.0000
```


Decimal to Binary Conversion

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers-- and even a modestly-sized decimal number becomes a very long binary number.
  - For example:

    $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

Unsigned Conversion

- Convert an unsigned binary number to decimal

  use positional notation (polynomial expansion)
- Convert a decimal number to unsigned Binary

  use successive division by 2

Examples

- Represent 26_{10} in unsigned Binary Code

  $26_{10} = 11010_2$
- Represent 26_{10} in unsigned Binary Code using 8 bits

  $26_{10} = 00011010_2$
- Represent $(26)_{10}$ in Unsigned Binary Code using 4 bits -- not possible

Signed Binary Codes

These are codes used to represent positive and negative numbers.

- Signed and Magnitude System
- 1's Complement System
- 2's Complement System

Signed and Magnitude

- The most significant (left most) bit represent the sign bit
  - 0 is positive
  - 1 is negative
- The remaining bits represent the magnitude

Examples of Signed & Magnitude

Decimal	5-bit Sign and Magnitude
+5	00101
-5	10101
+13	01101
-13	11101

Signed and Magnitude in 4 bits

0	0000	-0	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

Examples

Decimal	Signed	8-bit Signed
26_{10}	011010_2	00011010_2
-26_{10}	111010_2	10011010_2

1's Complement System

- Positive numbers:
  - same as in unsigned binary system
  - pad a 0 at the leftmost bit position
- Negative numbers:
  - convert the magnitude to unsigned binary system
  - pad a 0 at the leftmost bit position
  - complement every bit

Examples of 1's Complement

Decimal	5-bit 1's complement
5	00101
-5	11010
13	01101
-13	10010

1's Complement in 4 bits

0	0000	-0	1111
1	0001	-1	1110
2	0010	-2	1101
3	0011	-3	1100
4	0100	-4	1011
5	0101	-5	1010
6	0110	-6	1001
7	0111	-7	1000

Examples

Decimal	Signed	8-bit Signed
26_{10}	011010_2	00011010_2
-26_{10}	100101_2	11100101_2

CS 2401 Comp. Org. & Assembly — Data Representation in Computer Systems -- Chapter 2 — 41

2's Complement System

- Positive numbers:
  - same as in unsigned binary system
  - pad a 0 at the leftmost bit position
- Negative numbers:
  - convert the magnitude to unsigned binary system
  - pad a 0 at the leftmost bit position
  - complement every bit
  - add 1 to the complement number

CS 2401 Comp. Org. & Assembly — Data Representation in Computer Systems -- Chapter 2 — 42

Examples of 2's Complement

Decimal	5-bit 2's complement
5	00101
-5	11011
13	01101
-13	10011

CS 2401 Comp. Org. & Assembly — Data Representation in Computer Systems -- Chapter 2 — 43

2's Complement in 4 bits

0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

CS 2401 Comp. Org. & Assembly — Data Representation in Computer Systems -- Chapter 2 — 44

Examples

Decimal	Signed	8-bit Signed
26_{10}	011010_2	00011010_2
-26_{10}	100110_2	11100110_2

More Examples

- Represent 65 in 2's complement

  $65 = 0100\ 0001_2$

- Represent -65 in 2's complement

  $-65 = 1011\ 1111_2$

Convert 2's Complement to decimal

Positive 2's complement numbers

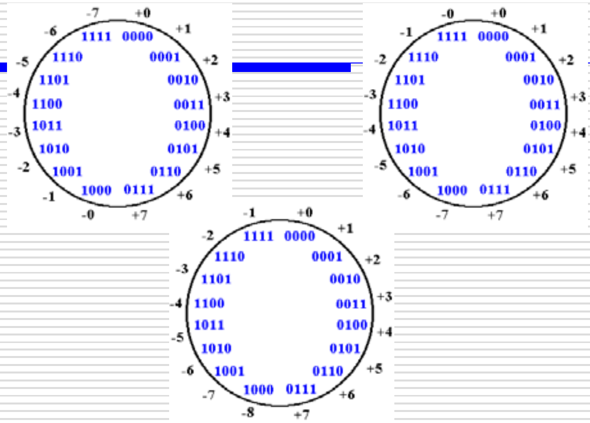
- convert the same as in unsigned binary

Negative 2's complement numbers

- complement the 2's complement number
- add 1 to the complemented number
- convert the same as in unsigned binary

Examples

2's complement	Decimal
00101	4 + 1 = 5
11011 → 00100 + 1	4 + 1 = 5 → -5
01101	8 + 4 + 1 = 13
10011 → 01100 + 1	8 + 4 + 1 = 13 → -13



Mathematical Formula

- Formula to convert a decimal number to a 1's complement --

$$N' = 2^n - N - 1$$

- Formula to convert a decimal number to a 2's complement --

$$N' = 2^n - N$$

where N is the binary number representing the decimal with n number of bits

Hexadecimal Notation

- base 16 -- has 16 digits:
  0 1 2 3 4 5 6 7 8 9 A B C D E F
- each Hex digit represents a group of 4 bits (i.e. half of a byte or a nibble) **0000** to **1111**
- use as a shorthand notation for convenient

Convert Binary ⟷ Hex

Binary	Hex
1111 0110b	**F6h**
1001 1101 0000 1010b	**9D0Ah**
1111 0110 1110 0111b	**F6E7h**
1011011b	**5Bh**

Examples

- ASCII value of character 'D' in Hex

  D = **0100 0100b**$_{ASCII}$ = **44h**$_{ASCII}$

- Represent **37d** in 2's complement using Hex.

  37d = 010 0101b$_{2's}$ = **0010 0101b**$_{2's}$

  = 25h$_{2's}$

- Represent -**37d** in 2's complement using Hex.

  -**37d** = 101 1011b$_{2's}$ = 1101 1011b$_{2's}$ = DBh$_{2's}$

Convert Hex ⟷ Decimal

- Convert Hex to decimal
  - use positional (polynomial expansion) notation

3BAh $= 3 \times 16^2 + B \times 16^1 + A \times 16^0$

$= 3 \times 256 + 11 \times 16 + 10 \times 1 = 954$d

- Convert decimal to Hex
  - Use successive divisions by 16

359/16 ⇨ 22 R 7,

22/16 ⇨ 1 R 6,

1/16 ⇨ 0 R 1

359d = 167h

Covert Large Binary to Decimal

Convert 1001 0011 0101 1100b to decimal

Method 1:

- Use polynomial expansion methods

Method 2:

- Convert number to hex, then convert it to decimal.

  1001 0011 0101 1100b = 935Ch

  935Ch = 37724d

Addition and Subtraction in Signed and Magnitude

	Decimal	Binary
(a)	5	0101
	+2	+0010
	7	0111
(b)	-5	1101
	-2	+1010
	-7	1111
(c)	5	0101
	-2	+1010
	3	0011
(d)	-5	1101
	+2	+0010
	-3	1011

Addition and Subtraction in 1's Complement

```
(a)     5      0101
       +2     +0010
       ---    -----
        7      0111

(b)    -5      1010
       -2     +1101
       ---    ------
       -7     1 0111
                   1
              ------
               1000

(c)     5      0101
       -2     +1101
       ---    ------
        3     1 0010
                   1
              ------
               0011

(d)    -5      1010
       +2     +0010
       ---    -----
       -3      1100
```


Addition and Subtraction in 2's Complement

```
(a)     5      0101
       +2     +0010
       ---    -----
        7      0111

(b)    -5      1011
       -2     +1110
       ---    ------
       -7     1 1001

(c)     5      0101
       -2     +1110
       ---    ------
        3     1 0011

(d)    -5      1011
       +2     +0010
       ---    -----
       -3      1101
```


Theoretical Facts

- Why is the carry out from adding 1's complements added to the sum?
  - $N_1' = 2^n - N_1 - 1$ and $N_2' = 2^n - N_2 - 1$
- Why is the carry out from adding 2's complements dropped?
  - $N_1' = 2^n - N_1$ and $N_2' = 2^n - N_2$

Overflow Conditions

Carry-in ≠ carry-out

```
        0111            1000
   5    0101      -5    1011
  +3   +0011      -4   +1100
  --   -----      --   -----
  -8    1000       7   10111
```

Carry-in = carry-out

```
        0000            1110
  +5    0101      -2    1110
  +2   +0010      -6   +1010
  --   -----      --   -----
   7    0111      -8   11000
```

Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

Expression	Result	Carry?	Overflow?	Correct result?
0100(+4)+0010(+2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(−6)	No	Yes	No
1100(−4)+1110(−2)	1010(−6)	Yes	No	Yes
1100(−4)+1010(−6)	0110(+6)	Yes	Yes	No

Signed Integer Representation

```
      0011 (3)              1011 (-5)
    × 0110 (6)            × 1100 (-4)
    + 0000                + 0000
   + 0011                + 0000
  + 0011                + 1011
 + 0000____            + 1011____
  00010010 (18)         10000100 (-124)
```


Signed Integer Representation

- Example:
  98765 × 1001 = 98765 ×(1000 + 1)
  = 98765 × 1000 + 98765
  98765 × 999 = 98765×(1000 - 1)
  = 98765 × 1000 – 98765
- Example:
  0011 × 0110 = 0011 ×(1000 - 0010)
  = 0011 × 1000 – 0011 × 0010

Signed Integer Representation

- Booth's Algorithm
  - Fast multiplication
  - Signed multiplication
- In Booth's algorithm, the first 1 in a string of 1s in the multiplier is replaced with a subtraction of the multiplicand.
- Shift the partial sums until the last 1 of the string is detected.
- Then add the multiplicand.

Signed Integer Representation

a_i	a_{i-1}	$a_{i-1} - a_i$	Operation
0	0	0	in middle of string 0. No operation.
0	1	1	end of string 1. Add multiplicand.
1	0	–1	beginning of string 1. Subtract multiplicand.
1	1	0	in middle of string 1. No operation.

```
     0011
 × 01100
+00000000
+01111101
+000000
+00011____
100010010
```

Ignore all bits over 2n.

Signed Integer Representation

```
    1101 (-3) 0011 (+3)
  × 1100 (-4)
+ 00000000
+ 0000000
+ 000001
+ 00000___
  00001100 (+12)
```


Signed Integer Representation

```
             00110101 (53)
×            01111110 (126)
+ 0000000000000000
+ 111111111001011
+ 00000000000000
+ 0000000000000
+ 000000000000
+ 00000000000
+ 0000000000
+ 000110101_______
 10001101000010110
```

–53

Ignore all bits over 2n.

Signed Integer Representation

```
    0101 (+5)
  × 1100 (-4)
+ 00000000
+ 0000000
+ 111011
+ 00000___
  11101100 (-20)
```

$a_{i-1} - a_i$	Action	Register		Carry
		0000	1100	0
00	rshf	0000	0110	0
00	rshf	0000	0011	0
10	sub rshf	+1011 1101	0011 1001	0 1
11	rshf	1110	1100	1

Signed Integer Representation

```
        0010110 (+22) 1101010(-22)
      × 1011110 (-34)
+ 00000000000000
+ 1111111101010
+ 000000000000
+ 00000000000
+ 0000000000
+ 000010110
+ 11101010
 111110100010100 (-748)
```


Signed Integer Representation

$a_{i-1} - a_i$	Action	Register		Carry
		0000000	1011110	0
00	rshf	0000000	0101111	0
10	sub rshf	+1101010 1110101	0101111 0010111	1
11	rshf	1111010	1001011	1
11	rshf	1111101	0100101	1
11	rshf	1111110	1010010	1
01	add rshf	+0010110 0010100 0001010	1010010 0101001	0
10	sub rshf	+1101010 1110100 1111010	0101001 0010100	1

Addition and Subtraction in Hexadecimal System

Addition

$(9F1B)_{16} + (4A36)_{16}$:

```
  1 1
  9F1B
+ 4A36
  ----
  E951
```

Subtraction

$(9F1B)_{16} - (4A36)_{16}$:

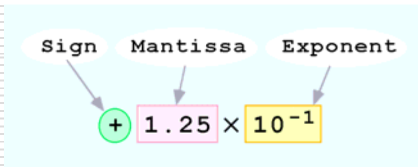
```
   16
  9F1B
- 4A36
  ----
  54E5
```


Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:
    - $0.125 = 1.25 \times 10^{-1}$
    - $5{,}000{,}000 = 5.0 \times 10^{6}$

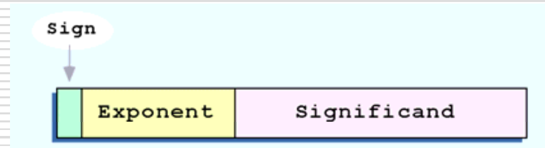
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



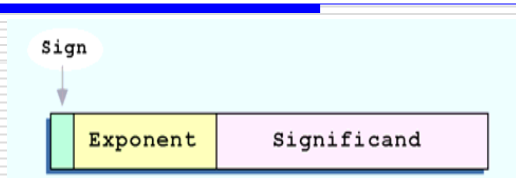
Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:



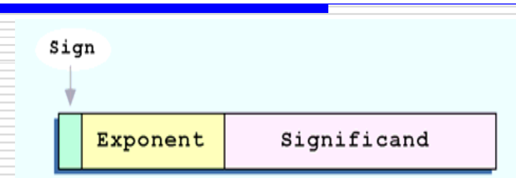
- This is the standard arrangement of these fields.

Floating-Point Representation



- The one-bit sign field is the sign of the stored value.
- The size of the exponent field, determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

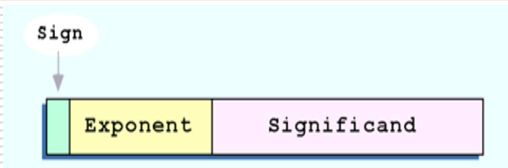
Floating-Point Representation



- The IEEE-754 single precision floating point standard uses an 8-bit exponent and a 23-bit significand.
- The IEEE-754 double precision standard uses an 11-bit exponent and a 52-bit significand.

For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

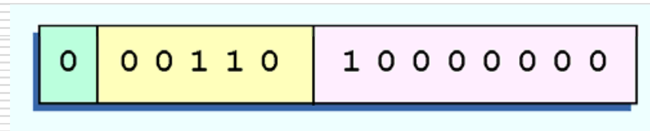
Floating-Point Representation



- The significand of a floating-point number is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 to which the significand is raised.

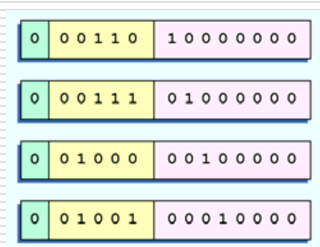
Floating-Point Representation

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 100000 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.

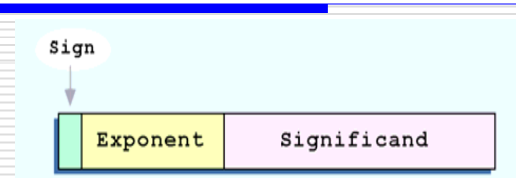


Floating-Point Representation

- The illustrations shown at the right are all equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 (=2^{-1})! (Notice that there is no sign in the exponent field!)

All of these problems can be fixed with no changes to our basic model.

Floating-Point Representation

- To resolve the problem of synonymous forms, we will establish a rule that the first digit of the significand must be 1. This results in a unique pattern for each floating-point number.
  - In the IEEE-754 standard, this 1 is implied meaning that a 1 is assumed after the binary point.
  - By using an implied 1, we increase the precision of the representation by a power of two. (Why?)

In our simple instructional model, we will use no implied bits.

Floating-Point Representation

- To provide for negative exponents, we will use a biased exponent.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.
  - In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called excess-16 representation.
- In our model, exponent values less than 16 are negative, representing fractional numbers.

Floating-Point Representation

- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that 32 = 1.0 x 2^5 = 0.1 x 2^6.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} (=10110_2).
- Graphically:

0	1 0 1 1 0	1 0 0 0 0 0 0 0

Floating-Point Representation

- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation 0.0625 = 1.0 x 2^{-4} = 0.1 x 2^{-3}.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} (=01101_2).

0	0 1 1 0 1	1 0 0 0 0 0 0 0

Floating-Point Representation

- Example:
  - Express -26.625$_{10}$ in the revised 14-bit floating-point model.
- We find 26.625$_{10}$ = 11010.101$_2$. Normalizing, we have: 26.625$_{10}$ = 0.11010101 x 2^5.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21$_{10}$ (=10101$_2$). We also need a 1 in the sign bit.

1	1 0 1 0 1	1 1 0 1 0 1 0 1

Floating-Point Representation

- The IEEE-754 single precision floating point standard uses bias of 127 over its 8-bit exponent.
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition.
- The double precision standard has a bias of 1023 over its 11-bit exponent.
  - The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

Floating-Point Representation

- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
  - Negative zero does not equal positive zero.

Floating-Point Representation

Floating Point Number	Single Precision Representation
1.0	0 01111111 00000000000000000000000
0.5 Is this correct?	0 10000000 00000000000000000000000
19.5	0 10000011 00111000000000000000000
-3.75	1 10000000 11100000000000000000000
Zero	0 00000000 00000000000000000000000
± Infinity	0/1 11111111 00000000000000000000000
NaN	0/1 11111111 any non-zero significand
Denormalized Number	0/1 00000000 any non-zero significand

Floating-Point Numbers

- A floating-point number is a representation for real numbers.
- IEEE standards set a format for representing floating-point numbers in binary.
- Example of an IEEE single-precision format (32 bits long):
  - $78.375_{10} \approx 429CC000_h$

Floating Point Structure

- The Sign Bit
  - 0 denotes a positive number; 1 denotes a negative number.
- The Exponent
  - represent both positive and negative exponents.
  - a *bias* is added to the actual exponent in order to get the stored exponent.
  - For IEEE single-precision floats, this value is 127 with 8 bits.
  - For double precision, the exponent field is 11 bits, and has a bias of 1023.

Floating Point Structure

- The Mantissa
  - The *mantissa*, also known as the *significand*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

	Sign	Exponent	Mantissa	Bias
Single	**1**	**8**	**23**	**127**
Double	**1**	**11**	**52**	**1023**

IEEE Single-Precision Format

Integral part: $78 \Rightarrow 1001110$

fractional part: $0.375 \Rightarrow 3/8 = 1/4 + 1/8$

$$= .01_2 + .001_2$$

$$= .011_2$$

$$78.375_{10} = 1001110.011_2$$

$$= 1.001110011 \times 2^6$$

IEEE Single-Precision Format

1.001110011×2^6

- Sign bit is 0
- Exponent including bias of 127 (127 +6 = 133) is 1000 0101 in 8 bits
- fraction is 00111001100000000000000 23 bits

0 1000 0101 00111001100000000000000

= 0100 0010 1001 1100 1100 0000 0000 0000

=42 9C C0 00

Conversion Procedure

- The leftmost bit is 0 for positive and 1 for negative.
- Convert the magnitude to decimal binary.
- Convert the binary decimal number to scientific notation
- Add a bias of 127_{10} to the exponent to form the next 8 bits. (to store exponent as a singed number).
- Fraction bits form the last 23 bits.

Example

45.5 $\Rightarrow$ 45 = 101101

0.5 = 1/2 = .1

45.5

= 101101.1 = 1.011011×2^5

= 0 1000 0100 01101100000000000000000

= 0100 0010 0011 0110 0000 0000 0000 0000

= 42 36 00 0 0

Example

-11.25 $\Rightarrow$ 1 = 1011

0.25 = 1/4 = .01

-11.25

= -1011.01 = -1.01101×2^3

= 1 1000 0010 01101000000000000000000

= 1100 0001 0011 0100 0000 0000 0000 0000

= C1 34 00 0 0

Example

$0.125 \Rightarrow 0 = 0$

$0.125 = 1/8 = .001$

0.125

$= 0.001 = 1.0 \times 2^{-3}$

= 0 0111 1100 00000000000000000000000

= 0011 1110 0000 0000 0000 0000 0000 0000

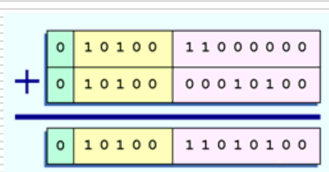
= 3E 00 00 00

Floating-Point Representation

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating-Point Representation

- Example:
  - Find the sum of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.
- Thus, our sum is 0.110101×2^4.

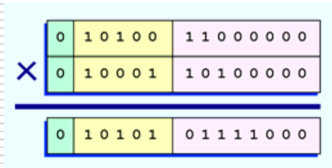


Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating-Point Representation

- Example:
  - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find 12_{10} = 0.1100 x 2^4. And 1.25_{10} = 0.101 x 2^1.
- Thus, our product is 0.0111100 x 2^5 = 0.1111 x 2^4.
- The normalized product requires an exponent of 22_{10} = 10110_2.



Floating-Point Representation

- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.
- At some point, every model breaks down, introducing errors into our calculations.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

Floating-Point Representation

- Our job becomes one of reducing error, or at least being aware of the possible magnitude of error in our calculations.
- We must also be aware that errors can compound through repetitive arithmetic operations.
- For example, our 14-bit model cannot exactly represent the decimal value 128.5 In binary, it is 9 bits wide:

  $10000000.1_2 = 128.5_{10}$

Floating-Point Representation

- When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

Floating-Point Representation

- Floating-point errors can be reduced when we use operands that are similar in magnitude.
- If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.
- In this example, the error was caused by loss of the low-order bit.
- Loss of the high-order bit is more problematic.

Floating-Point Representation

- Floating-point overflow and underflow can cause programs to crash.
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

Floating-Point Representation

- When discussing floating-point numbers, it is important to understand the terms range, precision, and accuracy.
- The range of a numeric integer format is the difference between the largest and smallest values that it can express.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value

Floating-Point Representation

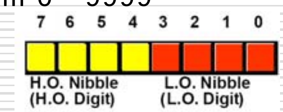
- Most of the time, greater precision leads to better accuracy, but this is not always true.
  - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

Floating-Point Representation

- This means that we cannot assume:
  - (a + b) + c = a + (b + c) or
  - a*(b + c) = ab + ac
- Moreover, to test a floating point value for equality to some other number, first figure out how close one number can be to be considered equal. Call this value epsilon and use the statement:
  - if (abs(x) < epsilon) then ...

BCD – Binary Coded Decimal

- A BCD digit is represented by 4 binary bits or a nibble.
- A BCD number is formed by a group of 4 binary bits or nibbles
- That means 8 bits can represent BCD from 0 – 99 and 16 bits can represent BCD from 0 - 9999



Character Representations

- ASCII
- UNICODE

ASCII Code

- ASCII: American Standard Code for Information Interchange.
- Used to represent characters and textual information
- Each character is represented with 1 byte
  - upper and lower case letters: a...z and A...Z
  - decimal digits -- 0,1,…,9
  - punctuation characters -- ; , . :
  - special characters --$ & @ / {
  - control characters -- carriage return (CR) , line feed (LF), beep

Examples of ASCII Code

Bit contents (S): `01010011`
Bit position: `76543210`

S ⟷ 83 (binary) , 53 (hex)

Bit contents (8): `00111000`
Bit position: `76543210`

8 ⟷ 56 (binary) , 38 (hex)

ASCII Code in Binary and Hex

Character	Binary	Hex
A	0100 0001	41
D	0100 0100	44
a	0110 0001	61
?	0011 1111	3F
2	0011 0010	32
DEL	0111 1111	7F

ASCII Groups

Bit 6	Bit 5	Group
0	0	Control Character
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39

UNICODE

- UNICODE uses a 16-bit word to represent a single character
- It can represent 65,536 different characters

Representing Colors on a Video Display

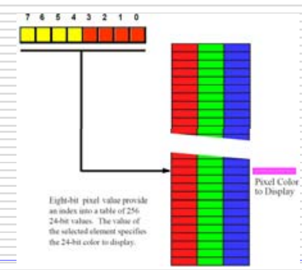
- An image is composed pixels (Picture elements)
- Different display modes use different data representations for each pixel
- A mixture of red, green, and blue form a specific color on the display
- *Color depth* describes amount of each red, green, and blue for a mixture on a pixel -- 8, 16, or 24 bits
- 24-bit display, each color has 256 different shades
- 16-bit display, each color has 5 or 6 bits of shades
- 8-bit display, each color has 2 or 3 bits of shades

Representing Colors on a Video Display

Bit-Depth	Number of Colors	Type
1	2	monochrome
2	4	CGA
4	16	EGA
8	256	VGA
16	65,536	High Color, XGA
24	16,777,216	True Color, SVGA
32	16,777,216	True Color + Alpha Channel

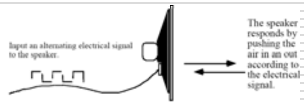
Representing Colors on a Video Display

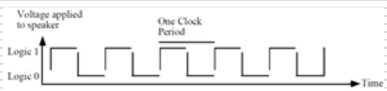
- A *hardware palette* allows an 8-bit display to display a specific color chosen from the colors of 24-bit display



Audio Information Representation

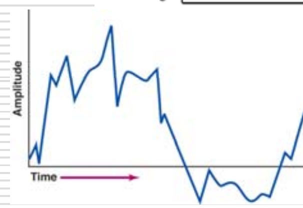
- Audible sounds are the result of vibrating air molecules quickly back and forth between 20 and 20,000 times per second (Hz)
- A computer is capable of generate a signal that repeatedly apply alternate logic 0 and 1 for a short period of time -- square wave
- Create a stream of bits fed to the speaker every 1/40,000 seconds with 1s and 0s, we get a 20 kHz sound
- It requires 5,000 bytes per second to generate 20 kHz sound

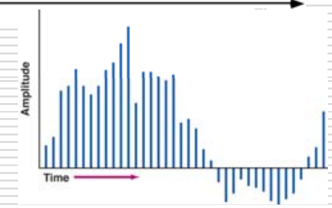




Audio Information Representation

- Analog audio signals are much more complex than square waves, that is only two different voltage levels are not enough for representation





Analog to Digital -- Sampling

- Sampling is done at regular intervals of time, often small fractions of a second.
- Like frequencies, sampling rates are measured in hertz.
- The precision in which a sample represents the actual amplitude of the waveform at the instant the sample is taken depends on the sample size or number of bits (also called bit depth) used in the binary representation of the amplitude value.
- An 8-bit sample can resolve 256 (=2^8) different amplitude or voltage values -- 40,000 bytes/second
- A 16-bit converter can resolve 65,536 (=2^{16}) values -- 80,000 bytes/second
- Sound recorded on audio CDs is stored as 16-bit samples.
- When a sample is taken, the actual value is rounded to the nearest value that can be represented by the number of bits in a sample.

Analog to Digital -- Sampling

- The minimum amount of storage (in bytes) required for a digitized signal is the product of
  - the sample rate (in samples/sec),
  - the sample size (in bytes; one byte equals 8 bits),
  - and the signal duration (seconds).
- The CD standard sampling rate of 44.1 kHz means that the waveform is sampled 44100 times per second.
- Thus, a 10-second signal sampled at 44.1 kHz with 16-bit (2-byte) precision requires 882,000 bytes (= 10 sec x 44,100 samples/sec x 2 bytes/sample), or about 861 Kbytes of storage (1 Kbyte = 1024 bytes).

Audio Formats

- MIDI
  - Musical Instrument Digital Interface is not technically an audio format, but it has recently become predominant as one of the main methods for delivering audio over the Internet. This is due to the fact that the file size are tiny compared to any other audio formats. The beauty behind MIDI files is the fact that it only save the data on what notes the instrument should play rather than the whole complex structure of sound waves.
- WAV
  - This format has become the standard audio format for sound files on the Internet. Almost every browser has built-in WAV playback support. The default Windows WAV format is PCM, which is basically uncompressed sound data, and these files tend to be rather large. However, many forms of compressed WAV files are available.
- MPEG (Layer 3)
  - This is latest of MPEG audio coding. It achieves high-fidelity sound quality, with a significant reduction in file size. It can shrink down CD audio by a factor of 12, without losing any clarity and quality. The encoded file are small enough to be transmitted at today's Internet speeds, this is one of the main reasons why mp3's are attracting so many users in the Internet community.