Chapter 2

Data Representation

LO-1:

- **Represent data** in various **formats**, and **convert** between decimal, binary, octal, hexadecimal, sign-magnitude, and ones and twos-complement.
- Perform some **basic** binary **arithmetic**, multiplication and division.

>>> Quiz-1 and **Test-1**

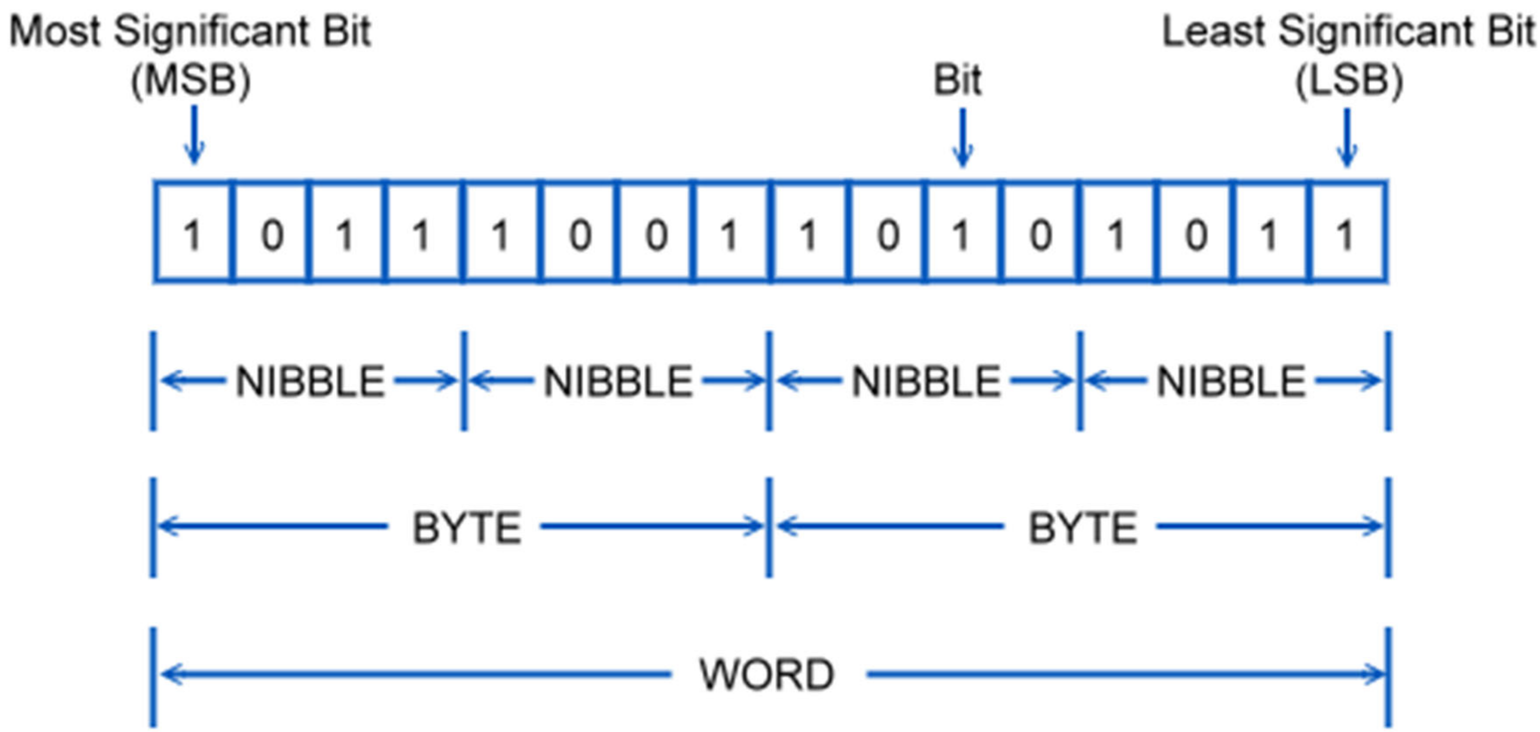
Data
1- numeric
  i- integer (unsigned, signed)
  ii- fraction
2- non-numeric (text, symbols, etc.)

Outline

- Converting between different numeric-radix systems
- Binary addition and subtraction
- Two's complement representation
- Floating-point representation
- Characters in computer

Human/natural language	Ch-2 →	Machine language
(Decimal numbers, letters & special characters)	binary 2's complement Floating-point ASCII, Unicode	(Binary/digital form)

2.1 Introduction



Byte or Word Addressable

- A computer allows either a byte or a word to be *addressable*
  - *Addressable*: a particular unit of storage can be retrieved by CPU, according to its location in memory.
  - A byte is the *smallest* possible *addressable* unit of storage in a *byte-addressable* computer
  - A word is the smallest addressable unit of storage in a *word-addressable* computer

2.2 Positional Numbering (decimal) System

- Let's first look at numbers in base-10 number system
- The decimal number 947_{10} (**base-10**) is:

$$947_{10} = 9 \times 10^{2} + 4 \times 10^{1} + 7 \times 10^{0}$$

- The decimal number 5836.47_{10} (**base-10**) is:

$$5836.47_{10} = 5 \times 10^{3} + 8 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 4 \times 10^{-1} + 7 \times 10^{-2}$$

2.2 Positional Numbering Systems (binary to decimal)

- Then, look at numbers in base-2 number system
- The binary number 11001_2 (base-2) is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1 = 25_{10}$$

- $11001_2 = 25_{10}$

Practice: any base to decimal

- $(01111101)_2 = ?$

- $(123)_8 = ?$

- $(123)_3 = ?$

A digit in a numeral that is greater than or equal to the base of the number is not allowed.

Any problem?

Practice

- $(01111101)_2 = 64+32+16+8+4+1=125$
- $(123)_8 = 1x8^2+2x8+3x8^0=83$
- $(123)_3 \rightarrow 130_3 \rightarrow 200_3 = 2x3^2 = 18$
  - $123_3 = 1x3^2+2x3^1+3x3^0=9+6+3=18$

2.3 Converting Between Bases (decimal to any base:integers)

- How can any integer (base-10 number) be converted into any radix system?
- There are two methods of conversion:
  - The ***Subtraction*** ***(-)*** method, and
  - The ***Division*** ***(/)*** **remainder** method.
- Let's use the subtraction method to convert $\mathbf{190_{10}}$ to $\mathbf{(x)_3}$.

2.3 Converting Between Bases

- **Converting 190_{10} to base 3...**
  - Continue in this way until the quotient is 0.
  - 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:

$$190_{10} = 21001_3$$

3	190	1
3	63	0
3	21	0
3	7	1
3	2	2
	0	

It is algorithmic and easier!

Exercise: decimal to any base:integers

- $458_{10}=\underline{\hspace{6cm}}_2$
- $652_{10}=\underline{\hspace{6cm}}_2$
- $458_{10}=\underline{\hspace{6cm}}_3$
- $652_{10}=\underline{\hspace{6cm}}_5$

- Once you get the result, please verify your result by converting back!
- Don't use calculator!

Exercise

- $458_{10} = 1\ 1100\ 1010_2$
- $652_{10} = 10\ 1000\ 1100_2$
- $458_{10} = 121222_3$
- $652_{10} = 10102_5$

- Now, please verify your result by converting back! How? Hint: use order-based multipliers for the radix.
- Don't use calculator!

2.3 Converting Fractional Numbers

- Fractional decimal numbers have non-zero digits on the right of the decimal point.
  - Fractional values of other radix systems have nonzero digits on the right of the *radix point*.
- Numerals on the right of a radix point represent negative powers of the radix. For example

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned} 0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= \tfrac{1}{2} + \tfrac{1}{4} \\ &= 0.5 + 0.25 \\ &= 0.75_{10} \end{aligned}$$

2.3 Converting Fractional Numbers

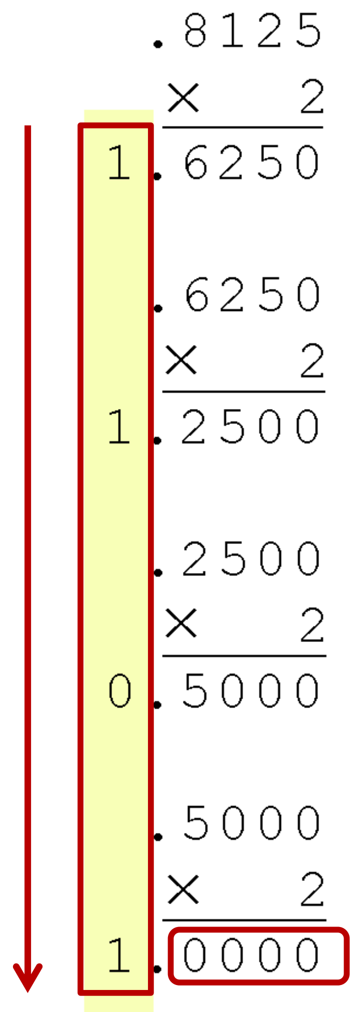
- Like the integer conversions, you can use either of the following two methods:
  - The ***Subtraction*** ***(-)*** method, or
  - The ***multiplication*** **(*x*)** method.
- The subtraction method for fractions is same as the method for integers
  - Subtract *negative powers of the radix*.
- Always start with the *largest value* --- first, n^{-1}, where n is the radix.

2.3 Converting Fractional Numbers (decimal to any base)

- **Converting 0.8125_{10} to X_2 . .**
  - You are finished when the product is 0, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $$\mathbf{0.8125_{10} = 0.1101_2}$$

  - Multiplication stops when the fractional part becomes 0
  - This method also works with any base. Just use *the target radix* as the multiplier.



2.3 Binary and Hexadecimal Number

- Binary numbering (base 2) system is the most important radix system in computers.
- But, it is difficult to read long binary strings
  - For example: $11010100011011_2 = 13595_{10}$
- For **compactness**, binary numbers are usually expressed as ***hexadecimal*** (***base-16***) numbers.

2.3 Converting Between Bases

- The ***hexadecimal*** numbering system uses the numerals 0,.. ,9, A,…,F
  - $12_{10} = C_{16}$
  - $26_{10} = 1A_{16}$
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal,
  - Group the binary digits into groups of ***4 bits*** --- a **nibble**.

Binary	Hex	Decimal
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	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

2.3 Converting Between Bases (binary-derived systems)

- Using groups of hextets, the binary number 13595_{10} (= 11010100011011_2) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros!

$\mathbf{351B_{16}}$

- Octal (base 8) values are derived from binary by using groups of three bits "octets" ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

$\mathbf{32433_8}$

Octal was useful when a computer used six-bit words.

Conversion between bases 2^m and 2^n

- Convert from base 16 to base 8
- You can use a intermediate radix number
- For example
  - Base 16 to Base 2 (binary)
  - Base 2 (binary) to Base 8

$$
\begin{aligned}
A9DB3_{16} &= 1010\ 1001\ 1101\ \ 1011\ \ 0011_2 \\
&= 10\ 101\ 001\ 110\ 110\ 110\ 011_2 \\
&= 2516663_8
\end{aligned}
$$

Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16:

  Decimal $= (d_3 \times 16^3) + (d_2 \times 16^2) + (d_1 \times 16^1) + (d_0 \times 16^0)$

  d_i = hexadecimal digit at the ith position

- Examples:
  - $1234_{16} = (1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = 4660_{10}$
  - $3BA4_{16} = (3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0) = 15268_{10}$

Exercise

- $58_{16}=$ ______________ $_{10}$
- $152_{8}=$ ______________ $_{10}$
- $56_{7}=$ ______________ $_{10}$
- $52_{11}=$ ______________ $_{10}$

- Once you get the result, please verify your result by converting back!
- Don’t use calculator!

Exercise

- $58_{16} = 88_{10}$
- $152_{8} = 106_{10}$
- $56_{7} = 41_{10}$
- $52_{11} = 57_{10}$

- Now, please verify your result by converting back (decimal to non-decimal bases)! But how? Hint: use successive division!
- Don’t use calculator!

EXERCISES: any bases to any other

- 176_{10} = ____________$_{16}$
- 55801_{10} = ____________$_{8}$
- $A6_{16}$ = ____________$_{13}$
- 55_{8} = ____________$_{16}$

EXERCISES

- $176_{10} = B0_{16}$
- $55801_{10} = 154771_{8}$
- $A6_{16} = 166_{10} = CA_{13}$
- $55_{8} = 2D_{16}$

Any to Decimal (A2D): ordered terms' addition using given radix-based multipliers.

(integer) Decimal to Any (D2Ai): successive division w/ given radix as divisor, collect remainders bottom up.

(fraction) Decimal to Any (D2Af): successive multiplication with given radix as multiplicand, collect integral carries top down.

Outline

- Converting between different numeric-radix systems ==> unsigned integer representation
- **Binary addition and subtraction**
- Two's complement representation ==> signed integer represen-tation
- Floating-point representation
- Characters in computer

Binary arithmetic: addition

- When the sum exceeds 1, carry a 1 over to the next-more-significant column (addition rules)
  - 0 + 0 = 0 carry 0
  - 0 + 1 = 1 carry 0
  - 1 + 0 = 1 carry 0
  - **1 + 1 = 0 carry 1**

Binary arithmetic: subtraction

- Subtraction rules
  - 0 - 0 = 0 borrow 0
  - **0 - 1 = 1 borrow 1**
  - 1 - 0 = 1 borrow 0
  - 1 - 1 = 0 borrow 0

$$\begin{array}{r} {}^{1}\mathbf{0} \\ \mathbf{-\ 1} \\ \\ \mathbf{=1} \end{array}$$

Unsigned number: Addition and subtraction

- Exercise: Use unsigned binary to compute
  - $100_{10}+10_{10}$
  - $100_{10}-10_{10}$

```
  0110 0100
+ 0000 1010
---------------
  0110 1110
```

```
  0110 0100
-  0000 1010
---------------
  0101 1010
```

- Use 8-bit unsigned numbers to calculate 100_{10} + 100_{10} + 100_{10} using binary addition =$(300)_{10}$ ==> will need 9-bit USigned system!

==> Overflow; a 2-byte number!

Unsigned number: Overflow

- Possible solution:
  - If data is stored in register, you should **use longer register,** which can hold more bits
  - In this case, you need a register, which has at least two bytes to hold the result

Unsigned number: Addition & Subtraction

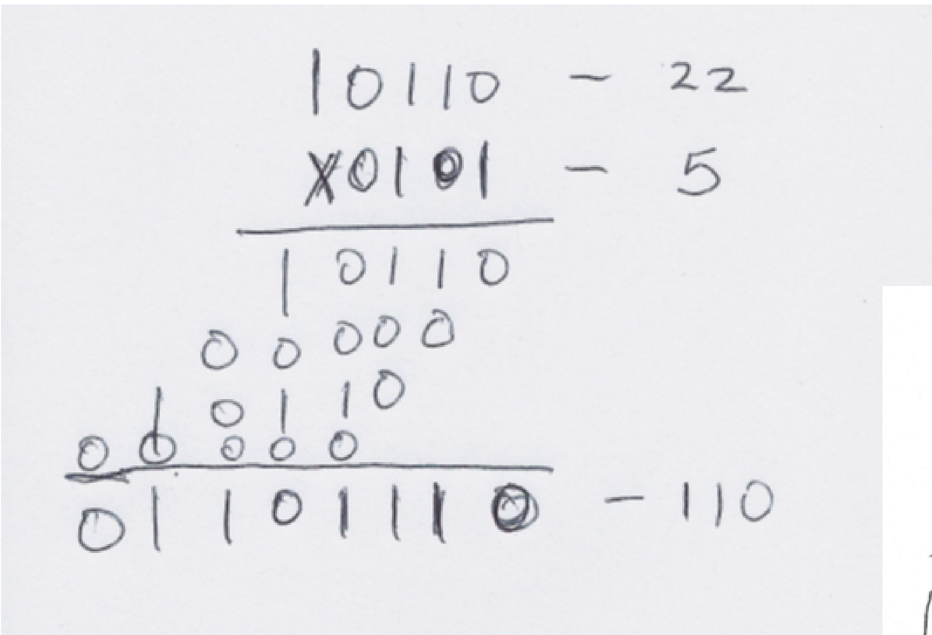
Carry in addition

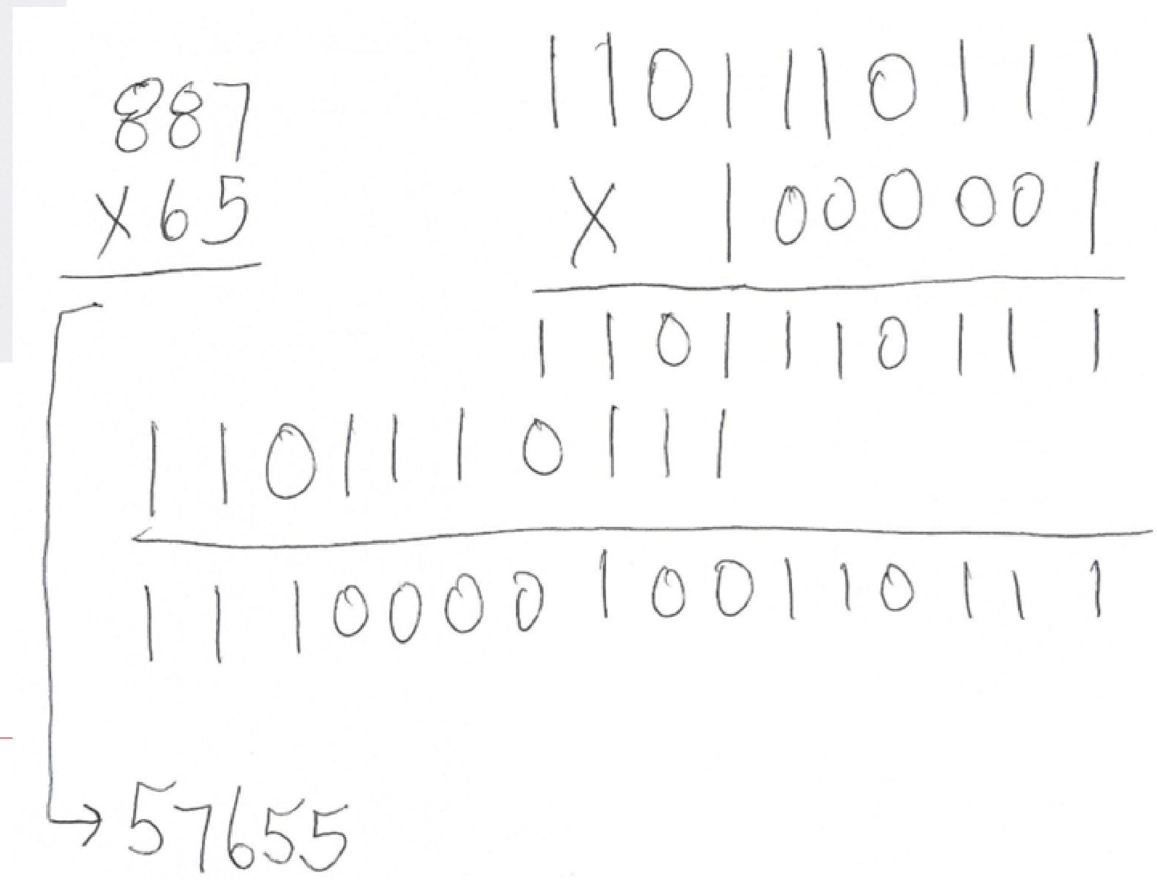
```
   1 1                 1
   146              10010010
+   89            + 01011001
-------           ---------------
   235              11101011
-------           ---------------
```

Borrow in subtraction

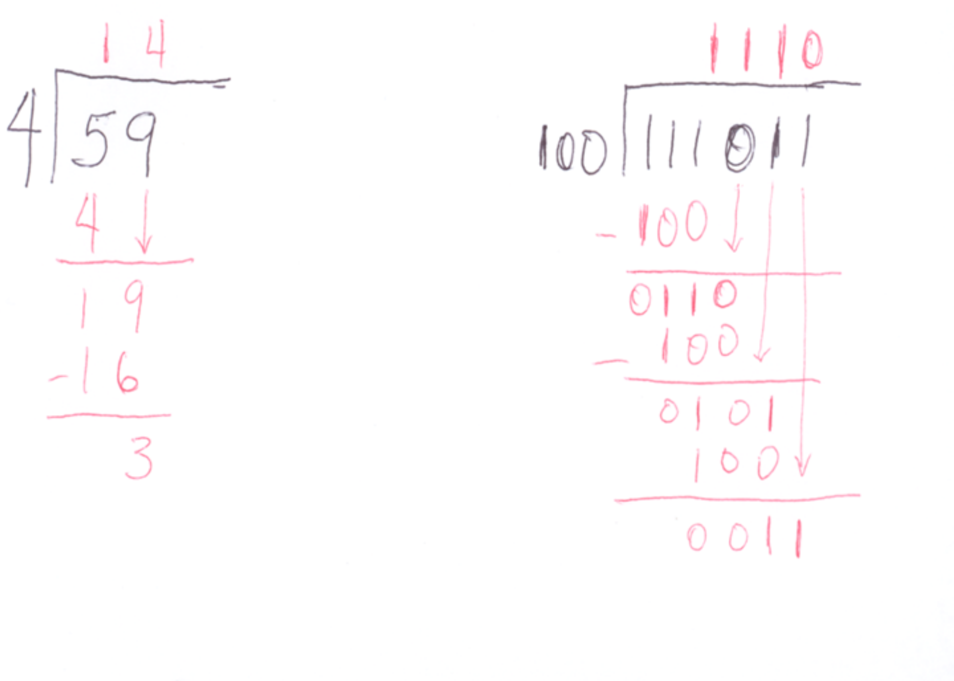
```
  1 3 1             1  1   1 1    1
   146              10010010
-   89            - 01011001
-------           ---------------
    57              00111001
-------           ---------------
```

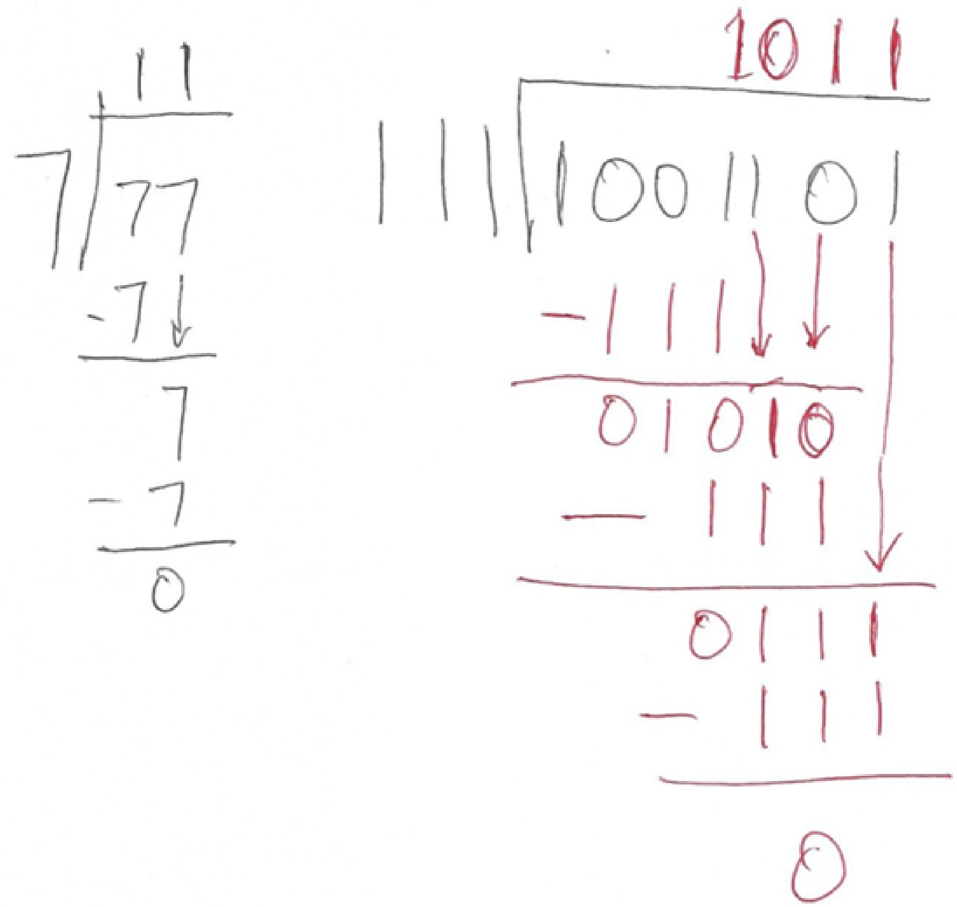
Unsigned number: Multiplication





Unsigned number: Division





Outline

- Converting between different numeric-radix systems
- Binary addition and subtraction
- **Two's complement representation**
- Floating-point representation
- Characters in computer

2.4 Signed Integer Representation

- In a byte, ***signed integer*** representation
  - 7 bits to represent *the value* of the number
  - 1 sign bit.
- There are three ways, where signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - **Two's complement**

==> signed binary rep.
==> 1111 1111 = -127

==> 1111 1111 = $(-1)_{10}$
how: MSb=1 ==> -ve number
flip all of the bits & increment:
0000 0000 + 1 = 0000 0001 = $(1)_{10}$

Two's Complement Representation

- Positive numbers
  - Signed value = Unsigned value
- Negative numbers
  - Signed value = Unsigned value - 2^n
  - n = number of bits

3-bit Bin Repr.	Unsigned Value	Sign-Mag Value	1C Value	2C Value
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

3 ways: Signed Value Representations

N=3 bit system

Number of combinations = 2^N = 2^3 = 8

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
. . .	. . .	. . .
11111110	254	-2
11111111	255	-1

Negative Integer Representation

- 2's compliment for a negative number $-x$
  1. Represent the positive number x in binary
  2. *Negate* all bits
  3. *Add 1* to the result

Let n = 6 bits

Represent magnitude	$+14_{10}$ = 001110
Complement each bit	110001
Add 1	+ 1
	110010

Result $-14_{10} = 110010_2$

Check by negating the result

Start with result	-14_{10} = 110010
Complement each bit	001101
Add 1	+ 1
	001110

As expected, we get $+14_{10} = 001110_2$

Another Example

- Represent -36 in 2’s complement format

starting value	`00100100 = +36`
step1: reverse the bits (1's complement)	`11011011`
step 2: add 1 to the value from step 1	`+        1`
sum = 2's complement representation	`11011100 = -36`

- Verification:

Sum of an integer and its 2's complement **must be zero**:

00100100 + 11011100 = 00000000 (8-bit sum) $\Rightarrow$ **Ignore Carry**

Addition and subtraction

- Addition of two's complement numbers
  - *Add all n bits* using binary arithmetic
  - ***Throw away any carry*** from the leftmost bit position
  - Do this **for any sign** (whether the same or different) See Overflow rules A and C upcoming!
- For example: X-Y
  - First, negate Y. Then, add to X
  - Thus, X-Y= X + (-Y)

Examples of addition

+14 = 00 1110
-14 = 11 0010

+9 = 00 1001
-9 = 11 0111

Example-1

Let n = 6 bits
Add 5 and 6 to obtain 11

$+5_{10}$ = 000101
$\underline{+6_{10}}$ = $\underline{000110}$
$+11_{10}$ = 001011

Example-2

Let n = 6 bits
-14 + 9 = –5

-14_{10} = 110010
$\underline{+9_{10}}$ = $\underline{001001}$
-5_{10} = 111011

<u>Check magnitude of -5_{10}</u>
Negate -5_{10} = 111011
Complement 000100
Add 1 $\underline{+\quad 1}$
Magnitude: +5 = 000101
OK

Example-3

Let n = 6 bits
-14 - 9 = -23

-14_{10} = 110010
$\underline{-9_{10}}$ = $\underline{110111}$
-23_{10} = 101001

<u>Check magnitude of -23_{10}</u>
Negate -23_{10} = 101001
Complement 010110
Add 1 $\underline{+\quad 1}$
Magnitude: +23 = 010111
OK

Verification

Background for EFLAGS

Overflow detection

- *X*, *Y* and *Z* are *N*-bit 2's-complement numbers and $Z_{2c}=X_{2c}+Y_{2c}$
- Overflow occurs if $X_{2c}+Y_{2c}$ exceeds the

A maximum value represented by N-bits.

- **If the signs of *X* and *Y* are different,**

B **don't detect overflow for $Z_{2c}=X_{2c}+Y_{2c}$.**

- **In case the signs of *X* and *Y* are the same, if the sign of $X_{2c}+Y_{2c}$ is opposite, overflow detected.** OF=C_{in} XOR C_{out} (of MSb)
  - Case 1: X, Y positive, Z sign bit ='1'
  - Case 2: X, Y negative, Z sign bit ='0'

C If X, Y and Z have same, dont detect overflow.

- Case 1: X, Y positive, Z sign bit ='1'

Example

Overflow detected!

Case-B-1

- $X_{2c}=(01111010)_{2c}$, $Y_{2c}=(00001010)_{2c}$,

  $X_{2c}+Y_{2c}=(10000100)_{2c}$ Overflow detected

OF=C_{in} XOR C_{out} (of MSb)

Signed value = Unsigned value - 2^n
Signed value = Unsigned value - 256

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

122
+
10
=
132

A negative sum of positive operands (or vice versa) is an overflow.
Ignore the sign bit and depend on the overflow behavior

Sign=1 negative

BUT actual answer is -124 ==> problem.

No carry-out of MSb but there is carry-in ==>O.F. Hence, dont ignore it!

Complement it to fix this, once this OF is detected! i.e. 2^8+(-124)=132

■ Case 2: X, Y negative, Z sign bit ='0'

Example

Overflow detected!

Case-B-2

- X_{2c}=(10011010)$_{2c}$, Y_{2c}=(10001010)$_{2c}$, $X_{2c}+Y_{2c}$=(00100100)$_{2c}$ **Overflow detected**

OF=C_{in} XOR C_{out} (of MSb)

Signed value = Unsigned value - 2^n
Signed value = Unsigned value - 256

	1	0	0	1	1	0	1	0	-102
									+
	1	0	0	0	1	0	1	0	-118
									=
1	0	0	1	0	0	1	0	0	-220

A negative sum of positive operands (or vice versa) is an overflow.
Ignore the sign bit and depend on the overflow behavior

Sign =0

BUT actual answer is 36 ==> problem.

carry-out=1, carry-in=0
==> O.F.
dont ignore it!

Solution:
Once this OF is detected, Do this:
36-2^8 = -220

C Programming Example

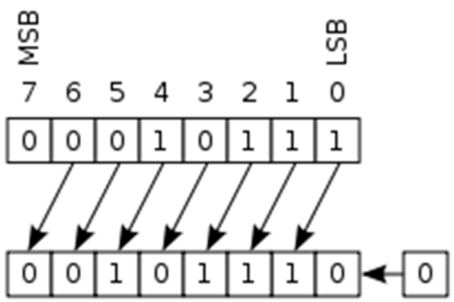
```
#include <stdio.h>

int main()
{
    int a = 32767;
    short b;

    printf ("size of int = %ld, size of short = %ld\n", sizeof(int), sizeof(short));

    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);

    a ++;
    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);

    return 0;
}
```

size of int = 4, size of short = 2

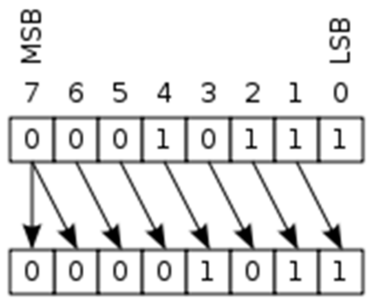
a = 32767, b = 32767

a = 32768, b = -32768

Bit Shifting (Arithmetic & Logical shift)



Left arithmetic shift



Right arithmetic shift

```
00010111 (decimal +23) LEFT-SHIFT
= 00101110 (decimal +46)
```

```
10010111 (decimal −105) RIGHT-SHIFT
= 11001011 (decimal −53)
```

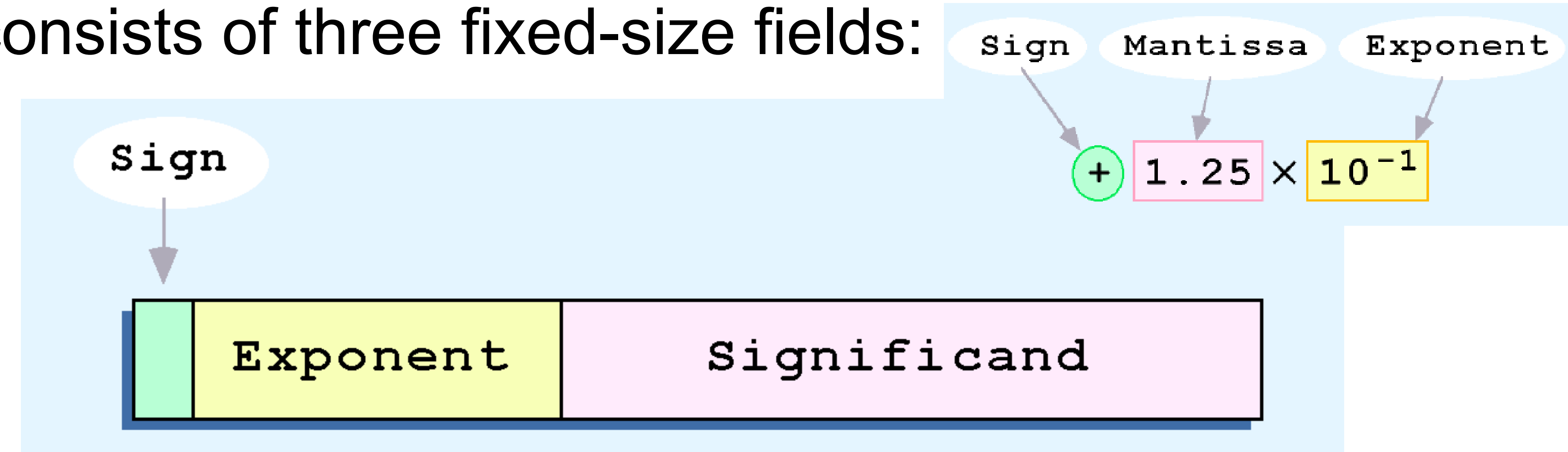
- To multiply 23 by 4, simply left-shift *twice*
- To divide 105 by 4, simply right-shift *twice*

Outline

- Converting between different numeric-radix systems
- Binary addition and subtraction
- Two’s complement representation
- **Floating-point representation**
- Characters in computer

2.5 Floating-Point Representation

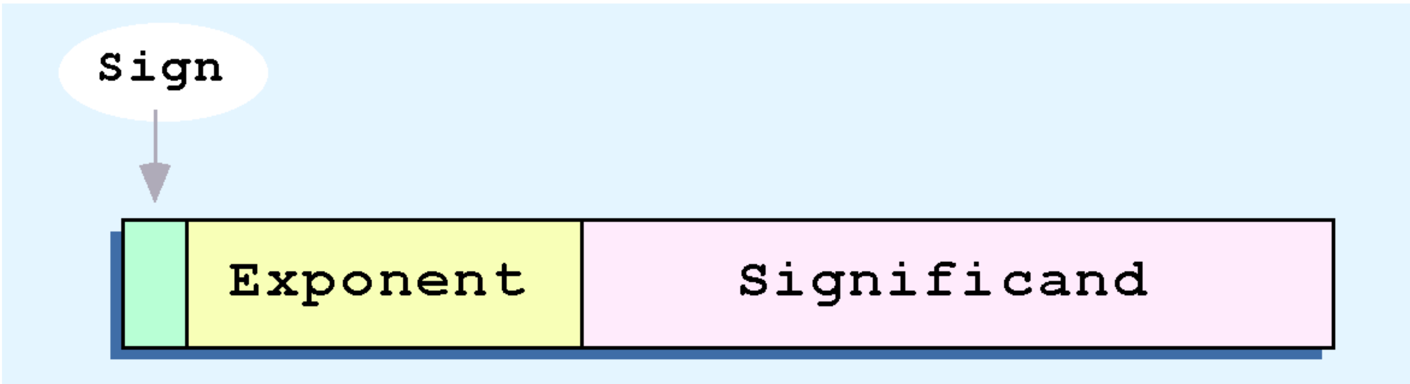
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation



- We introduce a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits

2.5 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 = 1.0×2^5 = $\mathbf{0.1 \times 2^6}$
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.

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

2.5 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**.

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

2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule **for the significand** that the **first '1' will appear after the radix point**.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 =$ $\mathbf{0.1001 \times 2^3}$. The last expression is correctly normalized.

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

2.5 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.

2.5 Floating-Point Representation

- To provide for negative exponents, we **will use a** ***biased*** ***exponent*.** = 2^{exp-1} - 1 where 'exp' is the number of bits of the exponent field
  - In our case, we have a 5-bit exponent.
  - $2^{5-1} - 1 = 2^4 - 1 = 15$
  - Thus will use **15 for our bias**: our exponent will use ***excess 15*** **representation**.
- In our model, exponent **values less than 15 are negative, representing fractional** numbers.

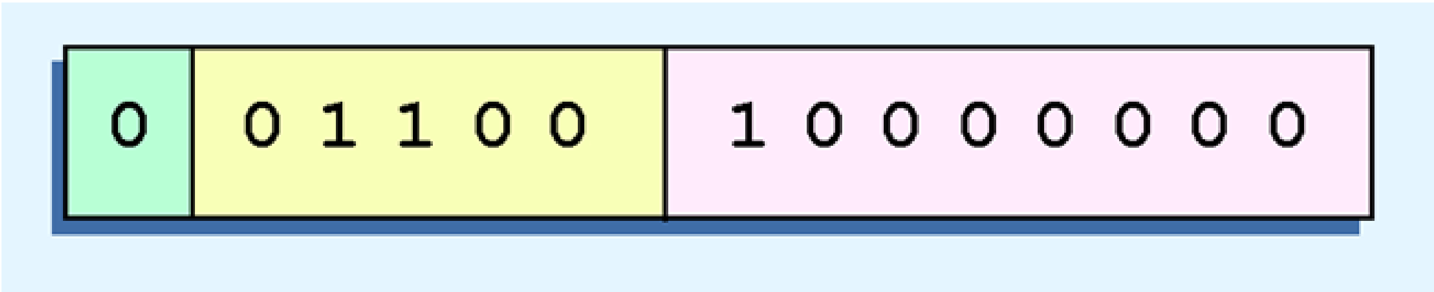
2.5 Floating-Point Representation

- Example:
  - Express 32_{10} in the revised 14-bit floating-point Simple Model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 15 biased exponent, we add 15 to 6, giving 21_{10} ($=10101_2$).
- So we have:

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

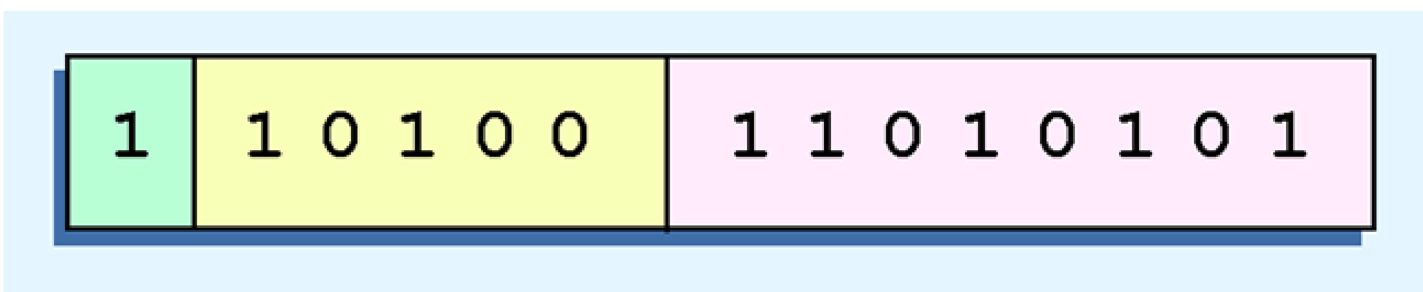
2.5 Floating-Point Representation

- 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 $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 15 biased exponent, we add 15 to -3, giving 12_{10} ($=01100_2$).



2.5 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: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 15 biased exponent, we add 15 to 5, giving 20_{10} ($=10100_2$). We also need a 1 in the sign bit.



Floating-Point Simple model to decimal number

Example-2

0 | 0 1 1 1 0 | 1 0 0 0 0 0 0 0
FPS

= + 0.1 0 0 0 0 0 0 0 x 2^-1

= (0.01)2

= 1x2^-2

= 0.25

Example-1

0 | 1 0 1 1 0 | 1 1 0 0 1 0 0 0

FPS

+ 0.1 1 0 0 1 0 0 x 2^7

= (1 1 0 0 1 0 0)2

= 100

2.5 Floating-Point Representation

- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

IEEE: Institute of Electrical and Electronics Engineers

2.5 Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significand has an implied 1 to the LEFT of the radix point.
  - The format for a significand using the IEEE format is: 1.xxx…
  - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$. The 1 is **implied, which means it does not need to be listed in the significand** (the significand would include only 001).

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - $-3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias is 127, so we add 127 + 1 = 128 (this is our exponent)
  - The first 1 in the significand is implied, so we have:

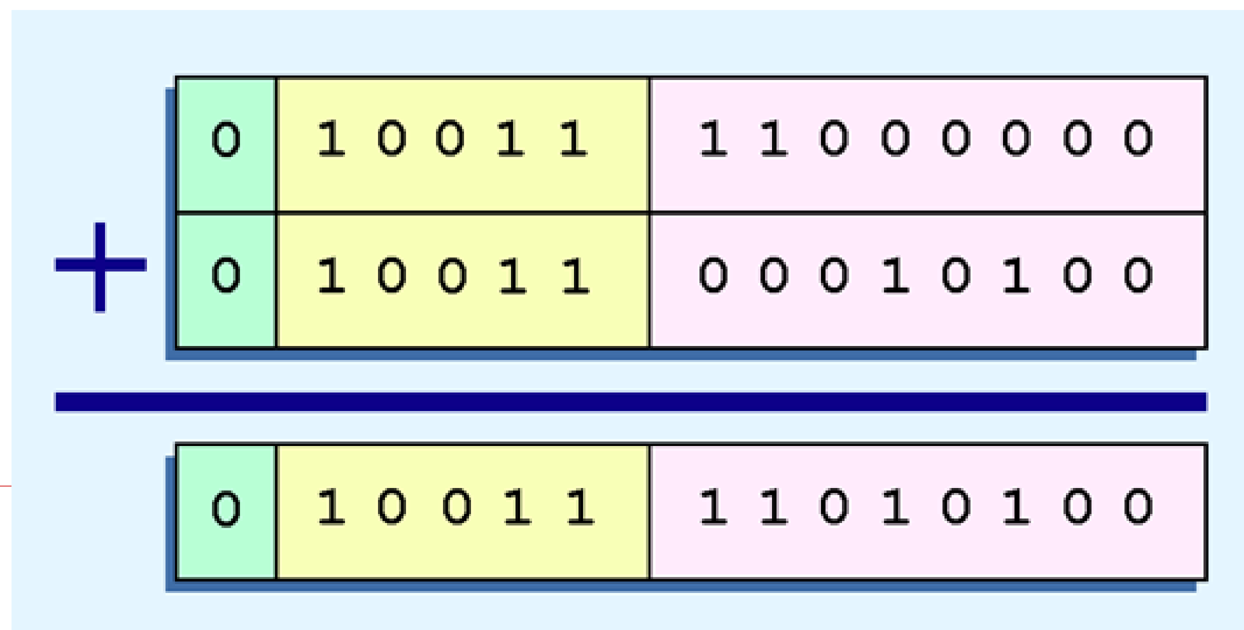
1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(implied)

  - Since we have an implied 1 in the significand, this equates to $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75$.

2.5 Floating-Point Representation

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



Outline

- Converting between different numeric-radix systems
- Binary addition and subtraction
- Two’s complement representation
- Floating-point representation
- **Characters in computer**

ASCII Code

- It encodes 128 specified characters into 7-bit binary integers as shown by the ASCII chart.
  - The characters encoded are numbers 0 to 9, lowercase letters a to z, uppercase letters A to Z, basic punctuation symbols, control codes that originated with Teletype machines, and a space.
  - For the full ASCII table, see next page

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

- For example, lowercase j would become binary 1101010 (decimal 106) in ASCII.

ASCII: 7-bit, 128 char --> C/C++ uses it for it's primitive data type 'char' representation
----- Unicode, Universal Coded Character Set, or UCS
UCS-2: 16-bit code units, 65,536 char, insufficient, fixed-width > Java uses it for its primitive data type 'char'
-------------- Unicode Transformation Format
UTF-8: four 8-bit code units, extends ASCII, variable-width via code-points, WWW
UTF-16: three 16-bit code units, extends UCS-2, variable-width, no ASCII, MS+Java

Unicode

- **Unicode** is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
- The latest version of Unicode contains a repertoire of more than 110,000 characters covering 100 scripts and multiple symbol sets.
  - As of June 2014, the most recent version is *Unicode 7.0*. The standard is maintained by the Unicode Consortium.
- The most commonly used Unicode encodings are UTF-8, UTF-16 and the now-obsolete UCS-2.
  - UTF-8 uses one byte for any ASCII character, all of which have the same code values in both UTF-8 and ASCII encoding, and up to four bytes for other characters.
  - UTF-16 extends UCS-2, using one 16-bit unit for the characters that were representable in UCS-2 and two 16-bit units (4 × 8 bit) to handle each of the additional characters.
  - UCS-2 uses a 16-bit code unit (two 8-bit bytes) for each character, but cannot encode every character in the current Unicode standard.

http://www.w3schools.com/charsets/ref_utf_misc_symbols.asp

Chapter 2 Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.
- Hexadecimal numbers are formed using four-bit groups called nibbles.
- Signed integers can be stored in one's complement, two's complement, or signed magnitude representation.
- Floating-point numbers are usually coded using the IEEE 754 floating-point standard.
- Floating-point operations are not necessarily commutative or distributive.
- Character data is stored using ASCII, EBCDIC, or Unicode.

End of Chapter 2