Number Systems
and
Data Representation

Lecture Outline

- **Number Systems**
  - Binary, Octal, Hexadecimal
- **Representation of characters using codes**
- **Representation of Numbers**
  - Integer, Floating Point, Binary Coded Decimal
- **Program Language and Data Types**

Data Representation?

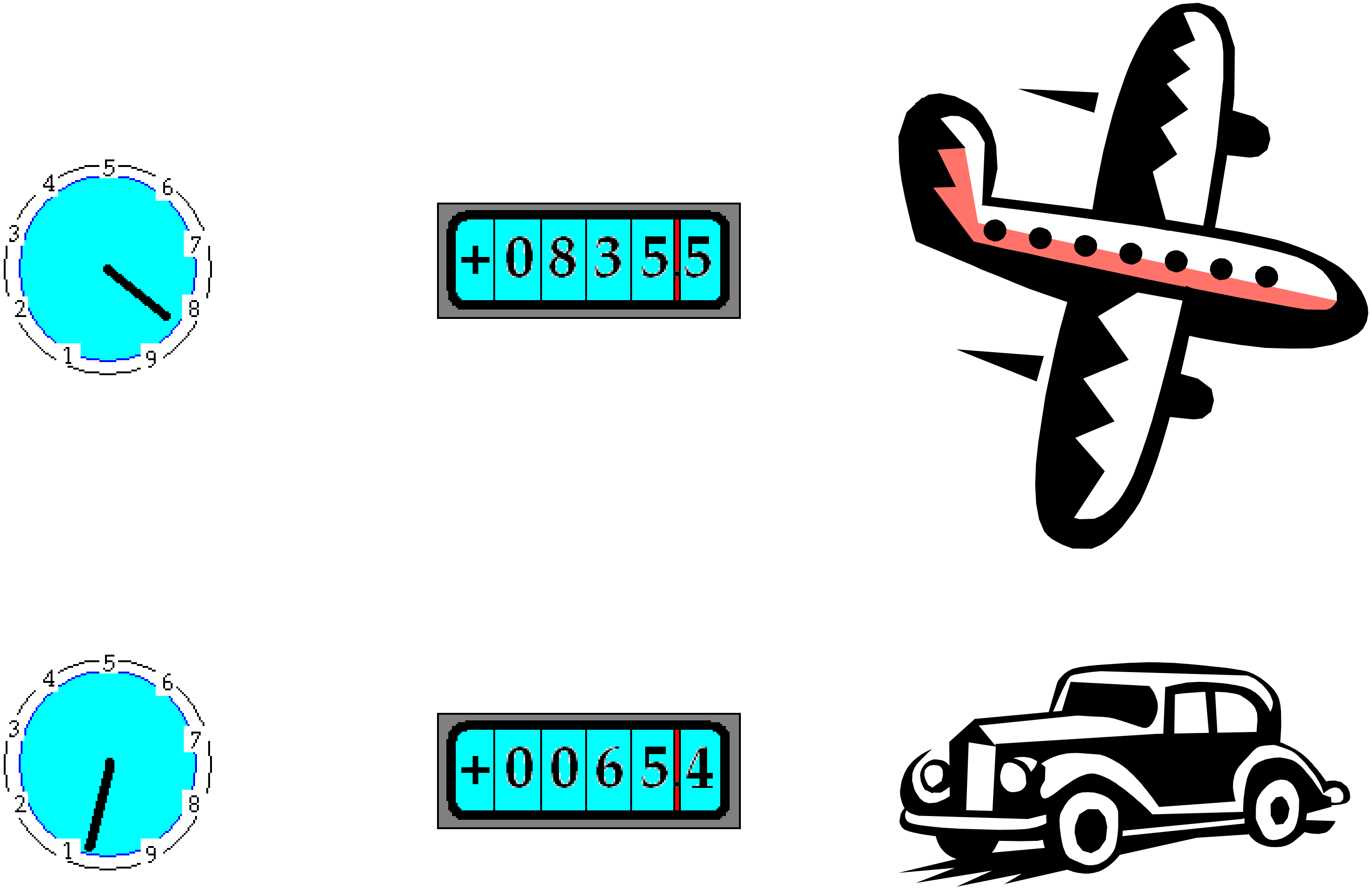
Representation = Measurement

- **Most things in the "Real World" actually exist as a single, continuously varying quantity** *Mass, Volume, Speed, Pressure, Temperature*
- **Easy to measure by "representing" it using a different thing that varies in the same way** *Eg.* <u>Pressure</u> as the **height** of column of mercury or as **voltage** produced by a pressure transducer
- **These are ANALOG measurements**

Digital Representation

- **Convert ANALOG to DIGITAL measurement by using a scale of units**
- **DIGITAL measurements**
  - **In units – a set of symbolic values -** digits
  - **Values larger than any symbol in the set use sequence of digits –** Units, Tens, Hundreds…
  - **Measured in discrete or whole units**
  - **Difficult to measure something that is not a multiple of units in size.** *Eg* Fractions

Analog vs. Digital representation



Data Representation

- Computers use digital representation
- Based on a binary system
  (uses on/off states to represent 2 digits).
- Many different types of data.
  - Examples?
- ALL data (no matter how complex) must be represented in memory as binary digits (bits).

Number systems and computers

- Computers store all data as binary digits, but we may need to convert this to a number system we are familiar with.
- Computer programs and data are often represented (outside the computer) using octal and hexadecimal number systems because they are a short hand way of representing binary numbers.

Number Systems - Decimal

- The decimal system is a base-10 system.
- There are 10 distinct digits (0 to 9) to represent any quantity.
- For an n-digit number, the value that each digit represents depends on its weight or position.
- The weights are based on powers of 10.

$$1024 = 1*10^3 + 0*10^2 + 2*10^1 + 4*10^0 = 1000 + 20 + 4$$

Number Systems - Binary

- The binary system is a base-2 system.
- There are 2 distinct digits (0 and 1) to represent any quantity.
- For an n-digit number, the value of a digit in each column depends on its position.
- The weights are based on powers of 2.

$1011_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 8+2+1 = 11_{10}$

Number Systems - Octal

- Octal and hexadecimal systems provide a shorthand way to deal with the long strings of 1's and 0's in binary.
- Octal is base-8 system using the digits 0 to 7.
- To convert to decimal, you can again use a column weighted system

$7512_8 = 7*8^3 + 5*8^2 + 1*8^1 + 2*8^0 = 3914_{10}$

- An octal number can easily be converted to binary by replacing each octal digit with the corresponding group of 3 binary digits

$7512_8 = 111101001010_2$

Number Systems - Hexadecimal

- Hexadecimal is a base-16 system.
- It contains the digits 0 to 9 and the letters A to F (16 digit values).
- The letters A to F represent the unit values 10 to 15.
- This system is often used in programming as a condensed form for binary numbers (0x00FF, 00FFh)
- To convert to decimal, use a weighted system with powers of 16.

Number Systems - Hexadecimal

- Conversion to binary is done the same way as octal to binary conversions.
- This time though the binary digits are organised into groups of 4.
- Conversion from binary to hexadecimal involves breaking the bits into groups of 4 and replacing them with the hexadecimal equivalent.

Example #1

Value of 2001 in Binary, Octal and Hexadecimal

Binary 1 1 1 1 1 0 1 0 0 0 1

$1 \times 2^{10} + 1 \times 2^{9} + 1 \times 2^{8} + 1 \times 2^{7} + 1 \times 2^{6} + 0 \times 2^{5} + 1 \times 2^{4} + 0 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0}$

$1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 1$

Octal 3 7 2 1

$3 \times 8^{3} + 7 \times 8^{2} + 2 \times 8^{1} + 1 \times 8^{0}$

$1536 + 448 + 16 + 1$

Decimal 2 0 0 1

$2 \times 10^{3} + 0 \times 10^{2} + 0 \times 10^{1} + 1 \times 10^{0}$

$2000 + 0 + 0 + 1$

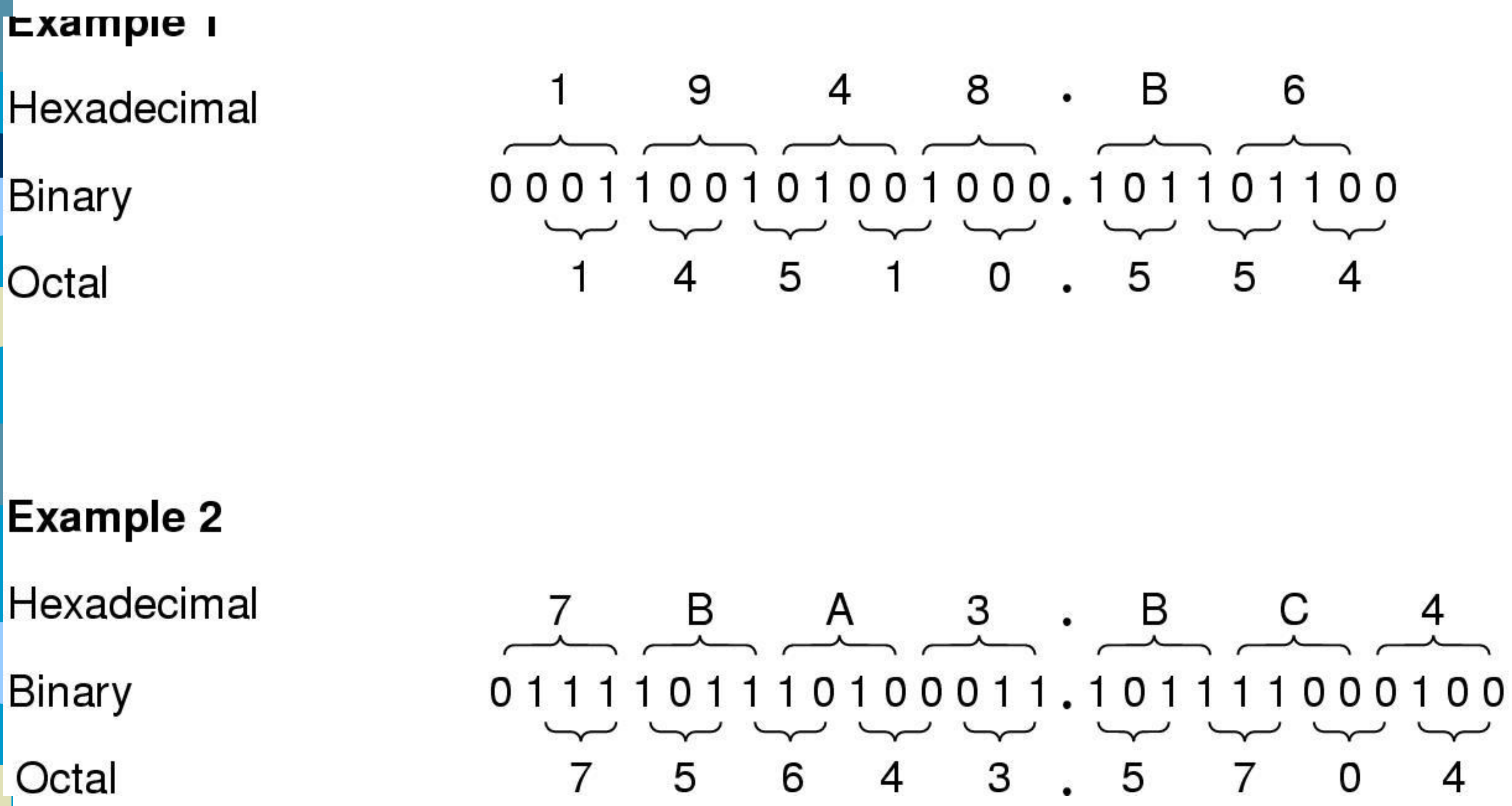
Hexadecimal 7 D 1

$7 \times 16^{2} + 13 \times 16^{1} + 1 \times 16^{0}$

$1792 + 208 + 1$

Example #2

Conversion: Binary ⇔ Octal ⇔ Hexadecimal



Example 1

Hexadecimal 1 9 4 8 . B 6

Binary 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 . 1 0 1 1 0 1 1 0 0

Octal 1 4 5 1 0 . 5 5 4

Example 2

Hexadecimal 7 B A 3 . B C 4

Binary 0 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 . 1 0 1 1 1 1 0 0 0 1 0 0

Octal 7 5 6 4 3 . 5 7 0 4

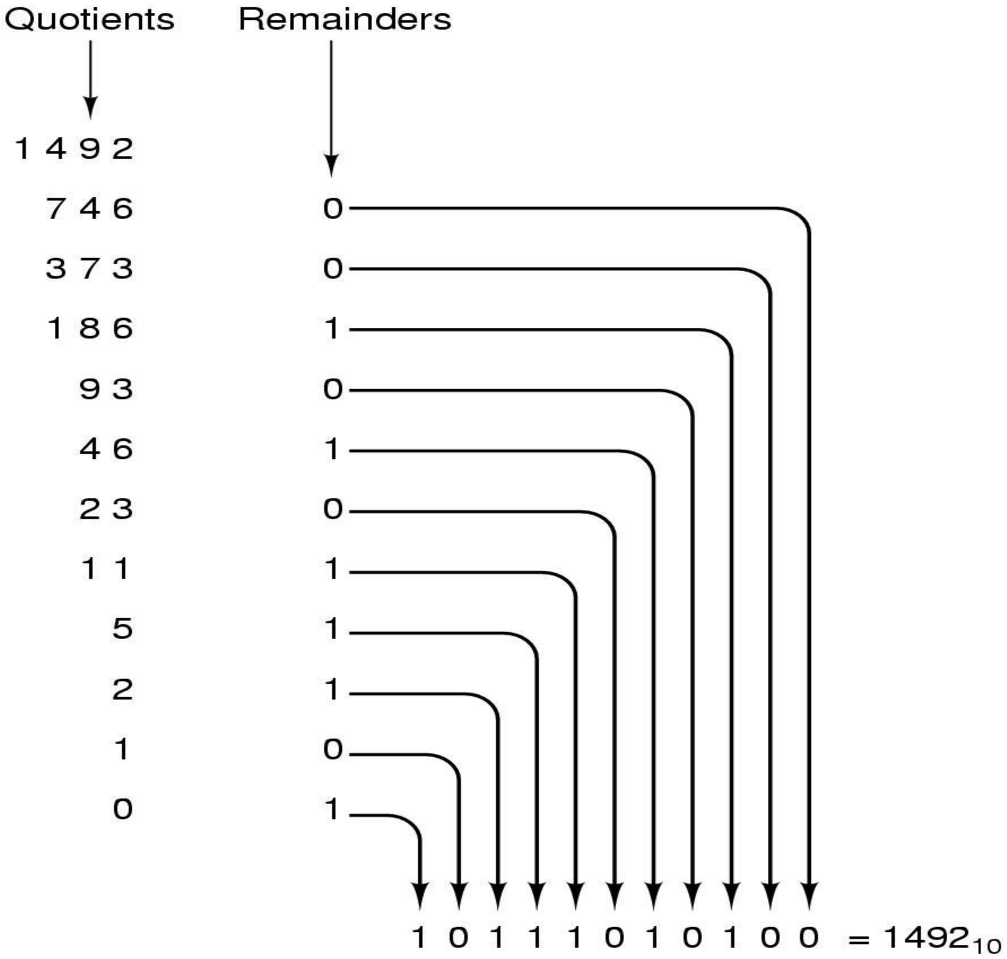
Decimal to Base$_N$ Conversions

- To convert from decimal to a different number base such as Octal, Binary or Hexadecimal involves repeated division by that number base
- Keep dividing until the quotient is zero
- Use the remainders in reverse order as the digits of the converted number

Example #3

Decimal to Binary 1492 (decimal) = ??? (binary)

Repeated Divide by 2

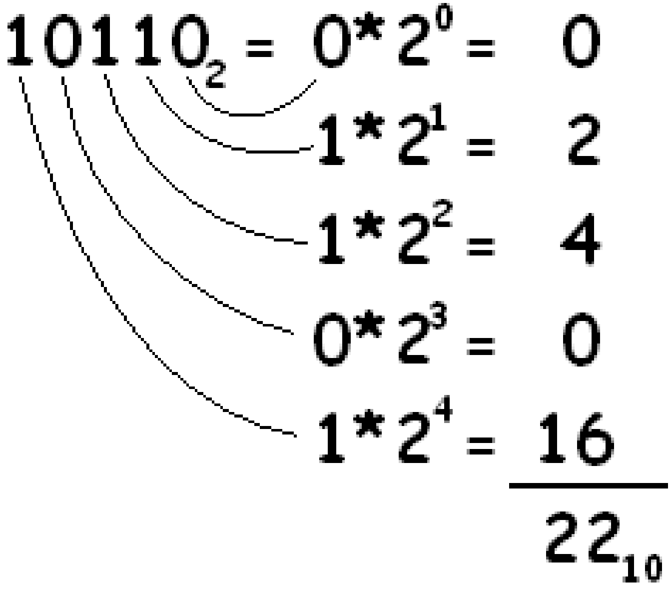


Quotients	Remainders
1492	
746	0
373	0
186	1
93	0
46	1
23	0
11	1
5	1
2	1
1	0
0	1

1 0 1 1 1 0 1 0 1 0 0 = 1492_{10}

Base$_N$ to Decimal Conversions

- Multiply each digit by increasing powers of the base value and add the terms
- Example: 10110_2 = ??? (decimal)



$10110_2 = 0*2^0 = 0$

$1*2^1 = 2$

$1*2^2 = 4$

$0*2^3 = 0$

$1*2^4 = 16$

22_{10}

Data Representation

- Computers store everything as binary digits. So, how can we encode numbers, images, sound, text ??
- We need standard encoding systems for each type of data.
- Some standards evolve from proprietary products which became very popular.
- Other standards are created by official industry bodies where none previously existed.
  - Some example encoding standards are ?

Alphanumeric Data

- Alphanumeric data such as names and addresses are represented by assigning a unique binary code or sequence of bits to represent each character.
- As each character is entered from a keyboard (or other input device) it is converted into a binary code.
- Character code sets contain two types of characters:
  - **Printable (normal characters)**
  - **Non-printable. Characters used as control codes.**
    - CTRL G (beep)
    - CTRL Z (end of file)

Alphanumeric Codes

- There are 3 main coding methods in use:
  - ASCII
  - EBCDIC
  - Unicode

ASCII

- 7-bit code (128 characters)
- has an extended 8-bit version
- used on PC's and non-IBM mainframes
- widely used to transfer data from one computer to another
- Examples:

EBCDIC

- An 8-bit code (256 characters)
- Different collating sequence to ASCII
- used on mainframe IBM machine
- Both ASCII and EBCDIC are 8 bit codes inadequate for representing all international characters
  - Some European characters
  - Most non-Alphabetic languages eg Mandarin, Kanji, Arabic, etc...

Unicode

- New 16 bit standard - can represent 65,536 characters
- Of which 49,000 have been defined
  - 6400 reserved for private use
  - 10,000 for future expansions
- Incorporates ASCII-7
- Example - Java code:

```
char letter = ‘A’;
char word[ ] = “YES”;
```

stores the values using Unicode characters

Java VM uses 2 bytes to store one unicode character.

0000 0000	0100 0001

0000 0000	0101 1001	0000 0000	0100 00101	0000 0000	0101 0011

Numeric Data

- Need to perform computations
- Need to represent only numbers
- Using ASCII coded digits is very inefficient
- Representation depends on nature of the data and processing requirements
  - Display purposes only (no computations): CHAR
    - PRINT 125.00
  - Computation involving integers: INT
    - COMPUTE 16 / 3 = 5
  - Computation involving fractions: FLOAT
    - COMPUTE 2.001001 * 3.012301 = 6.0276173133

Representing Numeric Data

- Stored within the computer using one of several different numeric representation systems
- Derived from the binary (base 2) number system.
- We can represent unsigned numbers from 0-255 just using 8 bits
- Or in general we can represent values from **0** to **2^N-1** using ***N*** bits.
- The maximum value is restricted by the number of bits available (called Truncation or Overflow)
- However, most programming languages support manipulation of signed and fractional numbers.
  - How can these be represented in binary form?

Representing Numeric Data

- Range of Values 0 to 2^N-1 in *N* bits

N	Range	*N*	Range
4	0 to 15	10	0 to 1023
5	0 to 31	16	0 to 65535
6	0 to 63	20	0 to 1048575
7	0 to 127	32	0 to 4294967295
8	0 to 255	64	0 to 1844674407370955165

Integer Representation

- UNSIGNED representing numbers from 0 upwards or SIGNED to allow for negatives.
- In the computer we only have binary digits, so to represent negative integers we need some sort of convention.
- Four conventions in use for representing negative integers are:
  - Sign Magnitude
  - 1's Complement
  - 2's Complement
  - Excess 128

Negative Integers – Sign Magnitude

- Simplest form of representation
- In an n-bit word, the rightmost n-1 bits hold the magnitude of the integer
- Example:
  - +6 in 8-bit representation is: 00000110
  - -6 in 8-bit representation is: 10000110
- Disadvantages
  - arithmetic is difficult
  - Two representations for zero
    - 00000000
    - 10000000

Binary Arithmetic

Addition Table

Digit	Digit	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Negative Integers – One's (1's) Complement

- Computers generally use a system called "complementary representation" to store negative integers.
- Two basic types - ones and twos complement, of which 2's complement is the most widely used.
- The number range is split into two halves, to represent the positive and negative numbers.
- **Negative numbers begin with 1, positive with 0.**

Negative Integers – One's (1's) Complement

- To perform 1's complement operation on a binary number, replace 1's with 0's and 0's with 1's (ie Complement it!)

+6 represented by: `00000110`

-6 represented by: `11111001`

- Advantages: arithmetic is easier (cheaper/faster electronics)
- Fairly straightforward addition
  - Add any carry from the Most Significant (left-most) Bit to Least Significant (right-most) Bit of the result
- For subtraction
  - form 1's complement of number to be subtracted and then add
- Disadvantages : still two representations for zero

`00000000` and `11111111` (in 8-bit representation)

Negative Integers – Two's (2's) Complement

- To perform the 2's complement operation on a binary number
  - replace 1's with 0's and 0's with 1's (i.e. the one's complement of the number)
  - add 1

+6 represented by: `00000110`

-6 represented by: `11111010`

- Advantages:
  - Arithmetic is very straightforward
  - End Around Carry is ignored
- only one representation for zero (`00000000`)

Negative Integers – Two's (2's) Complement

Two's Complement

- **To convert an integer to 2's complement**
  - **Take the binary form of the number**

`00000110` (6 as an 8-bit representation)

  - **Flip the bits:** (Find 1's Complement)

`11111001`

  - **Add 1**

```
 11111001
       +1
 11111010
```

(2's complement of 6)

- **Justification of representation: 6+(-6)=0?**

```
 00000110
+11111010
100000000
```

(6)

(2's complement of 6)

(0)

Negative Integers – Two's (2's) Complement

Properties of Two's Complement

–The 2's comp of a 2's comp is the original number

00000110 (6)

11111010 (2's comp of 6)

00000101

+1

00000110 (2's comp of 2's comp of 6)

–The sign of a number is given by its MSB

The bit patterns:

00000000 represents zero

0nnnnnnn represents positive numbers

1nnnnnnn represents negative numbers

Negative Integers – Two's (2's) Complement

- **Addition**
  - Addition is performed by adding corresponding bits

```
 00000111          ( 7)
+00000101          (+5)
 00001100          (12)
```

- **Subtraction**
  - Subtraction is performed by adding the 2's complement
  - Ignore End-Around-Carry

```
 00001100          (12)
+11111011          (-5)
100000111          ( 7)
```

Negative Integers – Two's (2's) Complement

- Interpretation of Negative Results

```
 00000101 (  5)
+11110100 (-12)
 11111001 ( __)
```

  - Result is negative

MSB of result is 1 so it is a negative number in 2's complement form

  - Negative what?

Take the 2's comp of the result to find out since the 2's comp of a 2's comp is the original number

  - Negative 7

the 2's complement of 11111001 is 00000111 or 7_{10}

excess 128 representation

- excess 128 for 8-bit signed numbers (or excess 2^{m-1} for *m*-bit numbers)
- Stored as the true value plus 128

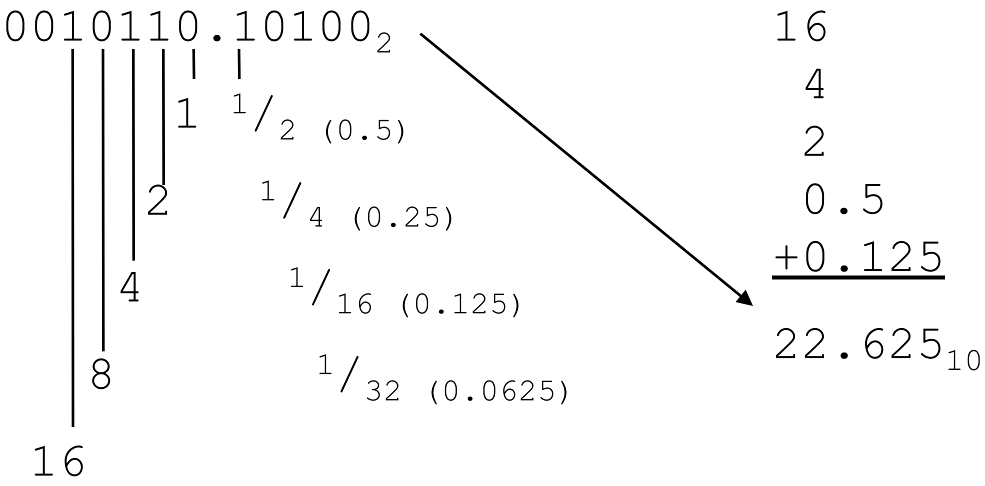
```
eg. -3 ⇒ -3+128=125 (01111101)
    26 ⇒ 26+128=154 (10011010)
```

- Number in range -128 to +127 map to bit values 0 to 255

same as 2's comp, but with sign bit reversed!!

Binary Fractions

- The Binary Point
  - Digits on the left ⇒ +ve powers of 2
  - Digits on the right ⇒ -ve powers of 2



0010110.10100_2

1 $^1/_2$ (0.5)

2 $^1/_4$ (0.25)

4 $^1/_{16}$ (0.125)

8 $^1/_{32}$ (0.0625)

16

16
4
2
0.5
+0.125

22.625_{10}

Integer Overflow

- **Problem: word size is fixed, but addition can produce a result that is too large to fit in the number of bits available.**
  This is called overflow.
- **If two numbers of the same sign are added, but the result has the opposite sign then overflow has occurred**
- **Overflow can occur whether or not there is a carry**
- Examples:

```
01000000 ( +64)   10000000 (-128)
   01000001 ( +65)   11000000 ( -64)
   10000001 (-127)   01000000 ( +64)
```

Floating Point Representation

- Fractional numbers, and very large or very small numbers can be represented with only a few digits by using scientific notation. For example:
  - 976,000,000,000,000 = $9.76 * 10^{14}$
  - 0.0000000000000976 = $9.76 * 10^{-14}$
- This same approach can be used for binary numbers. A number represented by

$\pm M * B^{\pm E}$

can be stored in a binary word with three fields:

  - Sign - plus or minus
  - Mantissa M (often called the significand)
  - Exponent E (includes exponent sign)
  - The base B is generally 2 and need not be stored.

Floating Point Representation

- Typical 32-bit Representation
  - The first bit contains the sign
  - The next 8 bits contain the exponent
  - The remaining 23 bits contain the mantissa
- The more bits we use for the exponent, the larger the range of numbers available, but at the expense of precision. We still only have a total of 2^{32} numbers that can be represented.
- A value from a calculation may have to be rounded to the nearest value that can be represented.
- Converting 5.75 to 32 bit IEEE format

5.75 (dec)= 101.11 (bin)

$= +1.0111 * 2^{+2}$

Floating Point Representation

- The only way to increase both range and precision is to use more bits.
- with 32 bits, 2^{32} numbers can be represented
- with 64 bits, 2^{64} numbers can be represented
- Most microcomputers offer at least single precision (32 bit) and double precision (64 bit) numbers.
- Mainframes will have several larger floating point formats available.

Floating Point Representation

- Standards
  - Several floating-point representations exist including:
    - IBM System/370
    - VAX
    - IEEE Standard 754
- **Overflow** refers to values whose magnitude is too large to be represented.
- **Underflow** refers to numbers whose fractional magnitude is too small to be represented - they are then usually approximated by zero.

Floating Point Arithmetic

- (Not Examinable)
- Multiplication and division involve adding or subtracting exponents, and multiplying the mantissas much like for integer arithmetic.
- Addition and subtraction are more complicated as the operands must have the same exponent - this may involve shifting the radix point on one of the operands.

Binary Coded Decimal

- Scheme whereby each decimal digit is represented by its 4-bit binary code

  7 = 0111

  246 = 001001000110

- Many CPUs provide arithmetic instructions for operating directly on BCD. However, calculations slower and more difficult.

Boolean Representation

- Boolean or logical data type is used to represent only two values:
  - TRUE
  - FALSE
- Although only one bit is needed, a single byte often used.
- It may be represented as:
  - 00_{16} = FALSE
  - FF_{16} or Non-Zero = TRUE
- This data type is used with logical operators such as comparisons = > < …

Programming languages and data types

- CPU will have instructions for dealing with limited set of data types (primitive data types). Usually these are:
  - Char
  - Boolean
  - Integer
  - Real
  - Memory addresses
- Recent processors include special instructions to deal with multimedia data eg MMX extension

Data Representation

- All languages allow programmer to specify data as belonging to particular data types.
- Programmers can also define special "user defined" variable data types such as days_of_ week
- Software can combine primitive data types to form data structures such as strings, arrays, records, etc…

Data Type Selection

- Consider the type of data and its use.
- Alphanumeric for text (eg. surname, subject name)
- Alphanumeric for numbers not used in calculations (eg. phone number, postcode)
- One of the numeric data types for numbers
- Binary integers for whole numbers
  - signed or unsigned as appropriate
- Floating point for large numbers, fractions, or approximations in measurement
- Boolean for flags

(end)