Number Systems and Binary Arithmetic

Introduction to Numbering Systems

- We are all familiar with the decimal number system (Base 10). Some other number systems that we will work with are:
  - **Binary → Base 2**
  - **Octal → Base 8**
  - **Hexadecimal → Base 16**

Characteristics of Numbering Systems

1) The number of digits is equal to the size of the base.
2) Zero is always the first digit.
3) The base number is never a digit.
4) When 1 is added to the largest digit, a sum of zero and a carry of one results.
5) Numeric values determined by the have implicit positional values of the digits.

Significant Digits

Binary: 11101101

Most significant digit | *Least significant digit*

Hexadecimal: 1D63A7A

Most significant digit | *Least significant digit*

Binary Number System

- Also called the **"Base 2 system"**
- The binary number system is used to model the series of electrical signals computers use to represent information
- 0 represents the no voltage or an **off state**
- 1 represents the presence of voltage or an **on state**

Binary Numbering Scale

Base 2 Number	Base 10 Equivalent	Power	Positional Value
000	0	2^0	1
001	1	2^1	2
010	2	2^2	4
011	3	2^3	8
100	4	2^4	16
101	5	2^5	32
110	6	2^6	64
111	7	2^7	128

Binary Addition

4 Possible Binary Addition Combinations:

(1)
$$\begin{array}{r} 0 \\ +0 \\ \hline 00 \end{array}$$

Carry → 00 ← Sum

(2)
$$\begin{array}{r} 0 \\ +1 \\ \hline 01 \end{array}$$

Note that leading zeroes are frequently dropped.

(3)
$$\begin{array}{r} 1 \\ +0 \\ \hline 01 \end{array}$$

(4)
$$\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

Decimal to Binary Conversion

- The easiest way to convert a decimal number to its binary equivalent is to use the ***Division Algorithm***
- This method repeatedly divides a decimal number by 2 and records the quotient and remainder
  - ***The remainder digits (a sequence of zeros and ones) form the binary equivalent in least significant to most significant digit sequence***

Division Algorithm

Convert 67 to its binary equivalent:

$67_{10} = x_2$

Step 1: 67 / 2 = 33 R 1 — ***Divide 67 by 2. Record quotient in next row***

Step 2: 33 / 2 = 16 R 1 — ***Again divide by 2; record quotient in next row***

Step 3: 16 / 2 = 8 R 0 — ***Repeat again***

Step 4: 8 / 2 = 4 R 0 — ***Repeat again***

Step 5: 4 / 2 = 2 R 0 — ***Repeat again***

Step 6: 2 / 2 = 1 R 0 — ***Repeat again***

Step 7: 1 / 2 = 0 R 1 — **STOP *when quotient equals 0***

$$\boxed{1\,0\,0\,0\,0\,1\,1_2}$$

Binary to Decimal Conversion

- The easiest method for converting a binary number to its decimal equivalent is to use the ***Multiplication Algorithm***
- Multiply the binary digits by increasing powers of two, starting from the right
- Then, to find the decimal number equivalent, sum those products

Multiplication Algorithm

Convert $(10101101)_2$ to its decimal equivalent:

Binary →	1	0	1	0	1	1	0	1
	x	x	x	x	x	x	x	x
Positional Values →	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Products → 128 + 32 + 8 + 4 + 1

$$\boxed{173_{10}}$$

Octal Number System

- Also known as the Base 8 System
- Uses digits 0 - 7
- Readily converts to binary
- Groups of three (binary) digits can be used to represent each octal digit
- Also uses multiplication and division algorithms for conversion to and from base 10

Decimal to Octal Conversion

Convert 427_{10} to its octal equivalent:

427 / 8 = 53 R3	Divide by 8; R is LSD
53 / 8 = 6 R5	Divide Q by 8; R is next digit
6 / 8 = 0 R6	Repeat until Q = 0

$\boxed{653_8}$

Octal to Decimal Conversion

Convert 653_8 to its decimal equivalent:

Octal Digits	6	5	3
	x	x	x
Positional Values	8^2	8^1	8^0
Products	384 +	40 +	3

$\boxed{427_{10}}$

Octal to Binary Conversion

Each octal number converts to 3 binary digits

Code
0 - 000
1 - 001
2 - 010
3 - 011
4 - 100
5 - 101
6 - 110
7 - 111

To convert 653_8 to binary, just substitute code:

6	5	3
↓	↓	↓
110	101	011

Hexadecimal Number System

- Base 16 system
- Uses digits 0-9 & letters A,B,C,D,E,F
- Groups of four bits represent each base 16 digit

Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Hexadecimal Number System

- Each hexadecimal digit represents four binary digits, and the primary use of hexadecimal notation is as a human-friendly representation of binary coded values in computing and digital electronics.

Decimal to Hexadecimal Conversion

Convert 830_{10} to its hexadecimal equivalent:

830 / 16 = 51 R14 ← = E in Hex

51 / 16 = 3 R3

3 / 16 = 0 R3

$\boxed{33E_{16}}$

Hexadecimal to Decimal Conversion

Convert 3B4F16 to its decimal equivalent:

Hex Digits	3	B	4	F
	x	x	x	x
Positional Values	16^3	16^2	16^1	16^0
Products	12288 +	2816 +	64 +	15

$$\boxed{15{,}183_{10}}$$

Binary to Hexadecimal Conversion

- The easiest method for converting binary to hexadecimal is to use a substitution code
- Each hex number converts to 4 binary digits

Substitution Code

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

Substitution Code

Convert 0101011010101110011010_2 to hex using the 4-bit substitution code :

$$\underbrace{0101}_{5}\ \underbrace{0110}_{6}\ \underbrace{1010}_{A}\ \underbrace{1110}_{E}\ \underbrace{0110}_{6}\ \underbrace{1010}_{A}$$

$$\boxed{56AE6A_{16}}$$

Substitution Code

Substitution code can also be used to convert binary to octal by using 3-bit groupings:

2	5	5	2	7	1	5	2
010	101	101	010	111	001	101	010

$\boxed{25527152_8}$

Complementary Arithmetic

- 1's complement
  - Switch all 0's to 1's and 1's to 0's

Binary #	→	10110011
1's complement	→	01001100

Complementary Arithmetic

- 2's complement
  - Step 1: Find 1's complement of the number

    Binary # → 11000110

    1's complement → 00111001
  - Step 2: Add 1 to the 1's complement

$$\begin{array}{r} 00111001 \\ +\ 00000001 \\ \hline 00111010 \end{array}$$

Binary Addition

- 0 + 0 = 0
  0 + 1 = 1
  1 + 0 = 1
  1 + 1 = 10
- E.g.:

$$\begin{array}{r} 101 \\ \underline{+101} \\ 1010 \end{array}$$

Binary Subtraction

- 0 - 0 = 0
- 0 - 1 = 1, and borrow 1 from the next more significant bit
- 1 - 0 = 1
- 1 - 1 = 0

Binary Subtraction & Addition

- E.g.:

- $$\begin{array}{r} 111 \\ \underline{-\ 10} \\ 101 \end{array}$$

Binary Subtraction & Addition

- E.g.:

- $$\begin{array}{r} 111 \\ \underline{-\ 10} \\ 101 \end{array}$$

Binary Subtraction & Addition

- E.g.:

- $$\begin{array}{r} 111 \\ \underline{-\ 10} \\ 101 \end{array}$$

Signed Binary

<u>Example:</u> **11101011 - 01100110** (235_{10} - 102_{10})

- First we apply two's complement to 01100110 which gives us **10011010**.

```
Step 1   01100110  ——— (reverse zeroes and ones)
         10011001

         10011001
Step 2        + 1  ——— (take result and add 1)
         --------
         10011010
```

- Now we need to add **11101011 + 10011010**, however when you do the addition you always disregard the last carry, so our example would be:

```
                 11101011
               + 10011010
                 --------
                 10000101
                 --------
                11111  1
ignore the  ↗
last carry
```

- which gives us **10000101**, now we can convert this value into decimal, which gives **133_{10}**

So the full calculation in decimal is 235_{10} - 102_{10} = 133_{10} (correct !!)

Signed Binary

<u>Example:</u> **11101011 - 01100110** (235_{10} - 102_{10})

- First we apply two's complement to 01100110 which gives us **10011010**.

```
Step 1  01100110  —— (reverse zeroes and ones)
        10011001

        10011001
Step 2       + 1  —— (take result and add 1)
        --------
        10011010
```

- Now we need to add **11101011 + 10011010**, however when you do the addition you always disregard the last carry, so our example would be:

```
                   11101011
                 + 10011010
                  ---------
                   10000101
                  ---------
                  11111 1
ignore the  ↗
last carry
```

- which gives us **10000101**, now we can convert this value into decimal, which gives **133_{10}**

  So the full calculation in decimal is 235_{10} - 102_{10} = 133_{10} (correct !!)

Example: **01100100 - 01111000** (100_{10} - 120_{10})

- 1's complement and 2's complement the bigger number which is 120

```
 01111000 (120)
 10000111   1's complement
+       1   2's complement
 --------
 10001000
```

- Then add the result of the 2's complement to the smaller number (100).

```
 10001000
+01100100 (100)
 --------
 11101100
```

- 1's complement and 2's complement the result of the addition leaving the most significant bit (MSB) which is the sign bit as it is which is 1.

The sign bit stays the same. It is not 1's complement and 2's complement

```
 1 1101100
 1 0010011
+        1
 ---------
 1 0010100 (-20)
```

Remember, 1 bit indicates a **NEGATIVE** sign, whereas 0 bit indicates positive.

- Now we can convert this value into a negative decimal, which gives **-20**$_{10}$
So, the full calculation in decimal is 100_{10} - 120_{10} = -20_{10} (correct !!)

Overflow

What is overflow?

- Overflow or arithmetic overflow is a condition that occurs when a calculation produces a result that is greater in magnitude than what a given data type can store or represent.

Range of whole numbers

- We can check the range of whole numbers of a computer using the following formula:

$$-2^{n-1} \text{ to } +2^{n-1} - 1$$

- Example, for an 8-bit, the range is as follows:

$$-2^{8-1} \text{ to } +2^{8-1} - 1$$

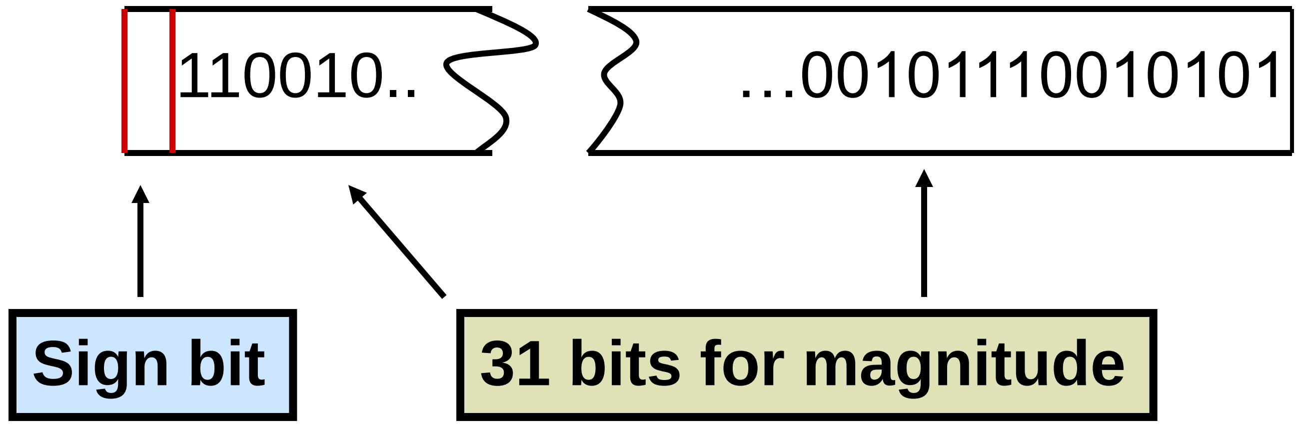
$$= -2^{7} \text{ to } +2^{7} - 1$$

$$= -128 \text{ to } +127$$

How to identify the condition of occurrence of overflow?

- Assuming we’re dealing with an 8-bit computer, let’s look at the situations below:
- 65 + 65 = +130 (An occurrence of overflow!)
- +128 – 5 = +123 (An occurrence of overflow!)
- Hence, an overflow occurs when the input or output exceeds the range of the whole number of the bits contained.

Signed Magnitude Numbers



0 = positive

1 = negative

This is your basic Integer format

Floating Point Numbers

- Real numbers must be normalized using scientific notation:

  $0.1\ldots \times 2^{n}$ where n is an integer

- Note that the whole number part is always 0 and the most significant digit of the fraction is a 1 – ALWAYS!

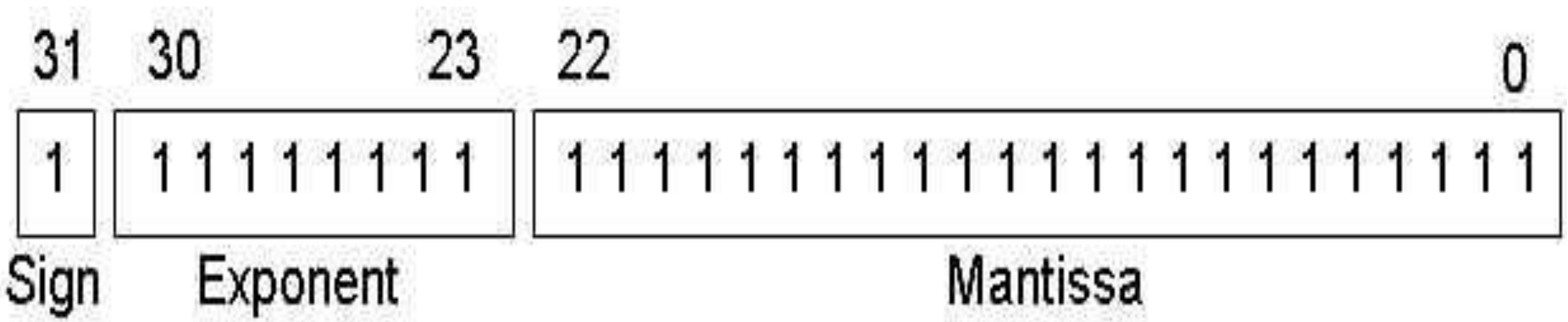
Floating Point Numbers

Single and Double Precision

- There are two most common floating point storage format:

IEEE Short Real: 32 bits	Sign: 1 bit Exponent: 8 bits Mantissa: 23 bits Also called *single precision*
IEEE Long Real: 64 bits	Sign: 1 bit Exponent: 11 bits Mantissa: 52 bits Also called *double precision*

- How do floating-numbers stored?



The Sign

- The sign of a binary floating-point number is represented by a single bit.
- A 1 bit indicates a negative number, and a 0 bit indicates a positive number.

The Mantissa

- It is useful to consider the way decimal floating-point numbers represent their mantissa. Using -3.154×10^5 as an example, the **sign** is negative, the **mantissa** is 3.154, and the **exponent** is 5. The fractional portion of the mantissa is the sum of each digit divided by a power of 10:

$$.154 = 1/10 + 5/100 + 4/1000$$

- A binary floating-point number is similar. For example, in the number $+11.1011 \times 2^3$**,** the sign is positive, the mantissa is 11.1011, and the exponent is 3. The fractional portion of the mantissa is the sum of successive powers of 2. In our example, it is expressed as:

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16$$

- Or, you can calculate this value as 1011 divided by 2^4. In decimal terms, this is eleven divided by sixteen, or 0.6875. Combined with the left-hand side of 11.1011, the decimal value of the number is 3.6875.

The Exponent

- IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127. Let's use the number 1.101 x 2^5 as an example. The exponent (5) is added to 127 and the sum (132) is binary 10100010. Here are some examples of exponents, first shown in decimal, then adjusted, and finally in unsigned binary:

Exponent (E)	Adjusted (E+127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000
-1	126	01111110

- The binary exponent is unsigned, and therefore cannot be negative. The largest possible exponent is 128-- when added to 127, it produces 255, the largest unsigned value represented by 8 bits. The approximate range is from 1.0 x 2^{-127} to 1.0 x 2^{+128}.

- While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it.

- The exponent bias for single precision is 127 and for double precision is 1023.

- The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field. For the eight-bit format, $k = 3$, so the bias is 23−1 − 1 = 3. For IEEE 32-bit, $k = 8$, so the bias is 28−1 − 1 = 127.

The binary value stored in the IEEE floating point number

- To calculate the floating point numbers for known width of mantissa and exponent, we use this formula:

$$\boxed{\text{Value} = (-1)^{\text{sign}} \times (1.\text{mantissa}) \times 2^{\text{exponent-bias}}}$$

- E.g.: 1 00001100 10001110000000000000000

  s= -1

  e= 00001100 = 12

  m= 1000111

  $\text{Value} = (-1)^{s} \times (1.m) \times 2^{e-127}$

  $= (-1)^{-1} \times (1.1000111) \times 2^{12-127}$

  $= \underline{\mathbf{-1.1000111 \times 2^{115}}}$