CSCI 230 Class Notes
Binary Number Representations and Arithmetic

Mihran Tuceryan

with some modifications by
Snehasis Mukhopadhyay
Jan 22, 1999

1 Decimal Notation

What does it mean when we write 495? How do we figure out what the value of the number is. In this example, the digit 5 is called the units digit, 9 is called the tens, and 4 is called the hundreds. To figure out what the value is we add

$$4 \times 100 + 9 \times 10 + 5 \times 1 = 495$$

This is what it means to write a number in decimal notation. Another way of saying it is there are 4 100 units bunches, 9 ten unit bunches and 5 units. It is hard to see what's happening because the way we represent the numbers and the way we say the value of the number are embedded in the language. That is, in the English language, the way numbers are said (not all but most) is based on the decimal system. Another way to look at the above equation is:

$$4 \times 10^2 + 9 \times 10^1 + 5 \times 10^0 = 495$$

A number of points to note here:

- 10 is the base unit
- How much each digit contributes to the total value depends on its *position* in the notation. **Definition:** The idea of giving symbols different values that depend on where they are written and using 0's to fill empty positions is called the *positional-number system*
- For base 10 (decimal) system, there are 10 symbols (digits) used: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Why?

2 Binary System

- $Base = 2$. Instead of using 10 as the base, we use 2.
- Symbols (digits) used: 0 and 1. The coefficients for each power of 2 in the positional representation is 0 or 1.

Example:

$$(1101)_2 = ?$$

To get the answer for this, we plug in the powers of 2 at each position and multiply by the coefficient. So,

$$\begin{aligned}(1101)_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= (13)_{10}\end{aligned}$$

This is **conversion** from binary to decimal system.

2.1 Conversion from decimal to binary

Given a decimal value v, how do we find its binary representation? We start with the binary expansion of v as follows:

$$(v)_{10} = a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_1 2^1 + a_0 2^0$$

where v is the decimal value of the binary number represented by $a_n a_{n-1} \cdots a_1 a_0$. The problem of converting from decimal to binary is to find the coefficients a_i for $i = 0, \cdots, n$, given the decimal value v. How do we do this? Looking at the expanded version of the binary number, we see that if we divide it by 2 and take the remainder, we would obtain a_0. That is,

$$(v)_{10}/2 = \begin{cases} \text{quotient} & a_n 2^{n-1} + a_{n-1} 2^{n-2} + \cdots a_1 2^0 \\ \text{remainder} & a_0 \end{cases}$$

Now we repeat this division by 2 on the quotient after the first division and we get a remainder of a_1, and so on. As we repeat this process, we get each one of the **bits** (digits of the binary number) in sequence from **least significant bit** (coefficient of 2^0) to the **most significant bit** (coefficient of 2^n).
Example:

$$(13)_{10} = \text{what in binary?}$$

We use repeated division by 2 and take the remainders:

	quotient	remainder	which position
13 / 2	6	1	2^0
6 / 2	3	0	2^1
3/2	1	1	2^2
1/2	0	1	2^3

So, putting the remainder bits in the correct positions, we obtain

$$(13)_{10} = (1101)_2.$$

Another example:

$$(6786)_{10} = \text{what in binary?}$$

We do the repeated division:

	quotient	remainder	bit position
6786 / 2	3393	0	2^0
3393 / 2	1696	1	2^1
1696 / 2	848	0	2^2
848 / 2	424	0	2^3
424 / 2	212	0	2^4
212 / 2	106	0	2^5
106 / 2	53	0	2^6
53 / 2	26	1	2^7
26 / 2	13	0	2^8
13 / 2	6	1	2^9
6 / 2	3	0	2^{10}
3 / 2	1	1	2^{11}
1 / 2	0	1	2^{12}

So, putting the bits together in the correct positions, we get

$$(6786)_{10} = (1101010000010)_2$$

2.2 Reasons for using binary numbers in a computer based system

A computer based system is made up of electronic circuits. The components of these circuits (*e.g*, a transistor) carry two states, a low and a high charge. Manipulating the charges so that they can do something of use is what the computer does. The binary number system is well suited for this manipulation because it is a two state number system. It can be use as a model of the computer's circuit's with a '0' representing a low charge and a '1' as a high charge. By manipulating the circuits of the computer system to behave like the binary number system, the computer is able to perform calculations, the bases of all it's operation. Other number systems can be used, but the difficulty of creating more than two states makes them impractical.

3 Binary Arithmetic on Integers

We define the basic operations:
Addition: $0+0=0$; $0+1=1$; $1+0=1$; and $1+1=10$
We do the addition of binary numbers just like decimal numbers: add digit by digit, taking into account the carries.
Example: $(13)_{10}+(5)_{10}$

carries		$+1$		$+1$			decimal number
first number		1	1	0	1	=	13
second number		0	1	0	1	=	5
sum	1	0	0	1	0	=	18

Multiplication: Similar to long multiplication in decimal system.
Example: $(33)_{10}\times(5)_{10}$

First number				1	0	0	0	0	1	=	$(33)_{10}$
Second number	$\times$						1	0	1	=	$(5)_{10}$
First partial result				1	0	0	0	0	1		
Second partial result			0	0	0	0	0	0			
Third partial result	$+$	1	0	0	0	0	1				
Final sum		1	0	1	0	0	1	0	1	=	$(165)_{10}$

Signed Number Representations

For subtraction, we need a way to represent negative numbers (or signed numbers). There a a number of different representations. Below we give four of them. In all of the following, we assume that we have fixed the number of bits to represent numbers. This is because in most representations the sign bit tends to be placed in the leftmost bit location, and one has to know where the leftmost bit location is.

1. **Signed magnitude representation:** Simplest representation. The leftmost bit is the sign bit and the remaining bits are the magnitude represented as above.

1	2	3	4	5	6	7	8

   Here bit 1 is the sign bit and bits 2–8 are the magnitude bits. Sign bit $=0$ means a positive number and sign bit $=1$ means a negative number.

   Example:

   $$\begin{aligned}(+5)_{10} &= 0\,0000101\\ (-5)_{10} &= 1\,0000101\end{aligned}$$

Problems are, treating the sign bit separately makes the hardware more complicated in order to take care of special conditions. Also in this representation we get two 0's

$$\begin{aligned}(+0)_{10} &= 0\ 0000000\\ (-0)_{10} &= 1\ 0000000\end{aligned}$$

Non-unique representation for 0's also complicates certain operations in hardware (e.g., comparison with 0) more complex. So this propery is not desirable.

2. **Excess representation:** For a given fixed number of bits, we remap the range such that roughly half the numbers are negative and half are positive. Here is an example of excess-8 notation for 4-bit numbers:

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

So, here is where the term excess-8 notation comes from: binary value $= 8+$"excess-8 value". The leftmost bit in this case can also be used as the sign bit: sign bit $= 1$ means positive number and sign bit $= 0$ means negative number.

In a similar manner, excess-64 notation will use 7 bits and represent numbers between -64 to 63 both inclusive: excess-64 value $=$ binary value $- 64$.

3. **One's complement representation:** Negative numbers are represented by complementing all the bits in the binary representation of the magnitude. Complementing in binary means

changing all the 1's to 0's and all the 0's to ones. It is represented by the tilde ($\sim$) character. So $\sim 1 = 0$ and $\sim 0 = 1$. Let's assume 8 bit long numbers.

Example:

$$\begin{aligned}(+5)_{10} &= 0000\ 0101 \\ (-5)_{10} &= \sim (0000\ 0101) = 1111\ 1010\end{aligned}$$

Once more the leftmost bit can be regarded as the sign bit. Sign bit $= 0$ means positive numbers and sign bit $= 1$ means negative numbers.

Disadvantage once more is the non-unique representation for 0.

4. **Two's complement representation:** This is represented by taking the one's complement representation and adding 1.

   Examples:

$$\begin{aligned}(+5)_{10} &= 0000\ 0101 \\ (-5)_{10} &= \sim (0000\ 0101) + 1 = 1111\ 1010 + 1 = 1111\ 1011\end{aligned}$$

   Looking at 0 representation:

$$\begin{aligned}(+0)_{10} &= 0000\ 0000 \\ (-0)_{10} &= \sim (0000\ 0000) + 1 = 1111\ 1111 + 1 = 1\ 0000\ 0000\end{aligned}$$

   The leftmost bit in this case is dropped because it is the ninth bit. So we get

$$-0 = 0000\ 0000$$

   Thus, we now have a unique representation for 0 in two's complement.

Subtraction: The subtraction is accomplished by adding the negative of the subtrahend. That is, to perform the subtraction $a - b$, we perform $a + (-b)$. This is easy to perform with two's complement notation.

$$a - b = a + (b)_{\text{2's complement}}$$

Example:
Compute $7 - 3$.

$$-3 = (3)_{\text{2's complement}} = \sim (0000\ 0011) + 1 = (1111\ 1100) + 1 = 1111\ 1101$$

$$7 - 3 = 7 + (3)_{\text{2's complement}}$$

7		0000 0111
-3	$+$	1111 1101
sum	1	0000 0100
	overflow bit discarded	

So, $(7 - 3)_{10} = (0000\ 0100)_{\text{2's complement}} = 4_{10}$.

4 Octal Representations

Octal representations are base-8 representations which are sometimes used in computer science to present long binary (base-2) codes in a more compact (or, shorter) fashion. Basically, each octal digit (symbol) encodes three binary digits (bits). So conversion between octal and binary representations are quite straightforward. An n-digit octal number, $a_{n-1}a_{n-2}\cdots a_1a_0$ is represented by the sum

$$a_{n-1} \times 8^{n-1} + a_{n-2} \times 8^{n-2} + \cdots a_1 \times 8^1 + a_0 \times 8^0$$

We only need 8 digits for octal numbers: $\{0, 1, 2, 3, 4, 5, 6, 7\}$. The conversions back and forth between octal and decimal are done in exactly the same manner as between decimal and binary. To convert from octal to decimal, we evaluate the above sum. To convert from decimal to octal, we perform repeated divisions *by 8*, and take the remainders as the digits of the octal number.
Example: Convert the decimal number 94 to octal. To do this we perform repeated divisions by 8:

number	operation	quotient	remainder	octal digit
94	$94 \div 8$	11	6	$a_0 = 6$
11	$11 \div 8$	1	3	$a_1 = 3$
1	$1 \div 8$	0	1	$a_2 = 1$
0	stop	stop	stop	stop

Therefore, $(94)_{10} = (136)_8$. We can check this by plugging the octal digits into the sum:

$$1 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 = 64 + 24 + 6 = 94.$$

One special property of number representations whose base is a power of 2 (such as octal) is that the conversion between this and base 2 (i.e., binary is as simple as writing the binary representations of

each digit is and just inserting them in place of the digit; or grouping bits in the binary representation and replacing them with the equivalent digit. So, for octal numbers, since $8 = 2^3$, we work in groups of 3 bits. We can write the binary representation for each octal digit:

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

So to convert octal 136 to binary, I substitute for each octal digit its equivalent in binary given in the table above. This gives me

$$(136)_8 = (001\ 011\ 110)_2$$

And if we evaluate this binary number we get:

$$1 \times 2^6 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 = 64 + 16 + 8 + 4 + 2 = (94)_{10}$$

5 Hexadecimal Numbers

Hexadecimal system is base-16 number representation system. An n-bit hexadecimal number, $a_{n-1}a_{n-2}\cdots a_1a_0$ is represented by the sum

$$a_{n-1} \times 16^{n-1} + a_{n-2} \times 16^{n-2} + \cdots a_1 \times 16^1 + a_0 \times 16^0$$

We need 16 digits to represent hexadecimal numbers: $\{$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F $\}$. The conversions back and forth between octal and decimal are done in exactly the same manner as between decimal and binary or octal. To convert from hexadecimal to binary, we evaluate the above sum. To convert from decimal to hexadecimal, we perform repeated divisions *by 16,* and take the remainders as the digits of the hexadecimal number.

Example: Convert decimal 94 to hexadecimal representation. To do this we perform repeated divisions by 16:

number	operation	quotient	remainder	hex digit
94	$94 \div 16$	5	14	$a_0 = E$
5	$5 \div 16$	0	5	$a_1 = 5$
0	stop	stop	stop	stop

Therefore, $(94)_{10} = (5E)_{16}$. We can check this by plugging the hexadecimal digits into the sum:

$$5 \times 16^1 + E \times 16^0 = 80 + 14 = 94.$$

Base 16 is a power of 2 just like the octal system is. Do the conversion between hex and binary by grouping bits (of four in this case) applies equally. We write the binary equivalents for each of the hex digits:

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

So, to convert hexadecimal 5E to binary, we substitute the binary codes for each of the hex digits:

$$(5E)_{16} = (0101\ 1110)_2$$

Which when evaluated will yield

$$1 \times 2^6 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 = 64 + 16 + 8 + 4 + 2 = (94)_{10}$$

6 Fractional Representations in Binary

The fractional numbers in binary are represented just like in the decimal system that we are used to: with a fractional point indicating where the base to the power of 0 ends. So in decimal system when we write the number 456.5, we mean

$$4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 5 \times 10^{-1}$$

The fractional point in 456.5 separates the 10^0 from the negative powers of 10. The binary fractional representation works in the same way, except we have a base of 2 instead of 10. So, when we write the binary fraction 101.11, we mean

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

6.1 Conversions

Going from binary to decimal is similar to evaluating the integer representations. For a binary number with *n* bits for the integer part and *m* bits for the fractional part, we evaluate the sum:

$$a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_0 \times 2^0 + a_{-1} \times 2^{-1} + \cdots + a_{-m} \times 2^{-m}$$

Converting from decimal to binary works as follows:

1. The integer part is converted just like before: with repeated division by 2 and taking the remainders in the order from LSB to MSB.

2. The fractional part is converted after a reasoning similar to how we derived the conversion of the integer part. A fractional part with *m* bits, $0.a_{-1}a_{-2}\cdots a_{-m}$, is represented by the following sum:

   $$a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \cdots + a_{-m} \times 2^{-m}$$

   Now, if we multiply this by 2, we get the sum:

   $$a_{-1} + a_{-2} \times 2^{-1} + \cdots + a_{-m} \times 2^{-m+1}$$

   which means that the a_{-1} is the coefficient of 2^0 we get after the multiplication. Now, we can multiply the fractional part of the result by two and get a_{-2}, and so on, until we get all the bits.

   Example: Convert the decimal number 5.625 to binary fractional representation. Here's how it works. We first convert the integer part, 5, to binary. I will not go over this in detail, but we perform repeated divisions by 2 on 5 and get the string 101 for its binary representation from the remainders of each division. We are now left with the fractional part to convert. We start by multiplying by two and get the resulting whole part of each multiplication result. Here's how it works.

fraction	operation	result	binary bit
0.625	2×0.625	1.25	$a_{-1} = 1$
0.25	2×0.25	0.5	$a_{-2} = 0$
0.5	2×0.5	1.0	$a_{-3} = 1$
0.0	stop	stop	stop

   So, the resulting binary representation of $(0.625)_{10}$ is $(0.101)_2$. Putting this together with the integer part we get, $(5.625)_{10} = (101.101)_2$.

7 Machine representation of floating point numbers

We saw above how to represent integers and binary fractions in the machine. The floating point representation is another way to represent real numbers in the hardware in a fixed number of bits.

Since we have a fixed number of bits (usually 32 bits for single precision and 64 bits for double precision, although this can vary from machine to machine), we would like to have a presentation which can represent a wide range of the real numbers. Once more, notice that for any fixed precision representation, we will only have a fixed number of possible bit combinations. For example, for 32 bits, we have 2^{32} possible combinations of 0's and 1's. So the question becomes what we do with these 2^{32} possibilities. Basically, we will regard each of these possible combinations of binary bits as samples from the real line.

There are a number of different floating point formats in the industry. Different vendors may have different formats for representing floating point numbers. For the purposes of this discussion, we will use the IBM System/370 floating point format. The IEEE standard floating point format is similar to this with some differences which we will mention later.

The 32 bit single precision floating point format for IBM 370 is as follows:

sign	exponent	mantissa
1 bit	7 bits	24 bits

We need to pick a radix with which we can interpret the bit pattern given above as a real number. The IBM format uses radix 16. Here the exponent is excess-64 notation. The number in the exponent field will represent the radix (16 in this case) raised to this number. The sign bit of the number (leftmost bit) is 0 if the number is positive and 1 if it is negative. The mantissa is a normalized fractional representation using the radix as the base. Normalized mantissa means that the digit after the fractional point is non-zero and if needed the mantissa should be shifted appropriately to make this digit non-zero and the exponent adjusted appropriately.

Example:

What is the value of the following floating point number?

1	1000010	1001 0011 1101 0111 1100 0010

First, this number is negative because its sign bit is 1. The exponent (1000010) has a binary value of 66. But since the exponent is in excess-64 notation, it is representing the value 2. Now, we compute the value of the mantissa. The mantissa is 1001 0011 1101 0111 1100 0010 which in hexadecimal is 93D7C2. This is the normalized fractional representation and the implicit fractional point is right before the digit 9. Putting all of this together, the value of the above floating point number is:

$$\begin{aligned} & -(9 \times 16^{-1} + 3 \times 16^{-2} + D \times 16^{-3} + 7 \times 16^{-4} + C \times 16^{-5} + 2 \times 16^{-6}) \times 16^2 \\ = \; & -(9 \times 16^{-1} + 3 \times 16^{-2} + 13 \times 16^{-3} + 7 \times 16^{-4} + 12 \times 16^{-5} + 2 \times 16^{-6}) \times 16^2 \\ = \; & -(9 \times 16^{1} + 3 \times 16^{0} + 13 \times 16^{-1} + 7 \times 16^{-2} + 12 \times 16^{-3} + 2 \times 16^{-4}) \\ = \; & -(144 + 3 + 0.8125 + 0.02734375 + 0.0029296875 + 0.000030517578125) \\ = \; & -147.842803955078125 \end{aligned}$$

Example:

What is the value of the following floating point number?

1	0111110	1100 1010 0100 1101 1010 1011

Once again, the sign bit is 1, so the number is negative. The exponent (0111110) has a binary value of 62. In excess-64 notation this corresponds to an integer value of -2. So the exponent is -2. Now we compute the mantissa just like the first example above. The mantissa in hex notation is CA4DAB, with the implicit fractional point being at the left before the C. Putting all this together, we get

$$\begin{aligned}
& -(C \times 16^{-1} + A \times 16^{-2} + 4 \times 16^{-3} + D \times 16^{-4} + A \times 16^{-5} + B \times 16^{-6}) \times 16^{-2} \\
= \; & -(12 \times 16^{-1} + 10 \times 16^{-2} + 4 \times 16^{-3} + 13 \times 16^{-4} + 10 \times 16^{-5} + 11 \times 16^{-6}) \times 16^{-2} \\
= \; & -(12 \times 16^{5} + 10 \times 16^{4} + 4 \times 16^{3} + 13 \times 16^{2} + 10 \times 16^{1} + 11 \times 16^{0}) \times 16^{-8} \\
= \; & -(12 \times 1048576 + 10 \times 65536 + 4 \times 4096 + 13 \times 256 + 10 \times 16 + 11) \times 16^{-8} \\
= \; & -(12582912 + 655360 + 16384 + 3328 + 160 + 11) \times 16^{-8} \\
= \; & -13258155 \times 16^{-8} \\
= \; & 0.00308690476231
\end{aligned}$$

Definition: The number of place values used to represent a number is called the *precision*.
Most *single precision* floating point numbers are represented by 32 bits although this may vary from specific machine to machine. *Double precision* floating point numbers use 64 bits to store the value. The organization is given below. For 32 bit numbers we have:

sign	exponent	mantissa
1 bit	7 bits	24 bits

And for 64 bit double precision numbers we have:

sign	exponent	mantissa
1 bit	7 bits	56 bits

The special string of all 0's represents the value of 0 in the IBM 370 format.
Now we can answer the following questions:

- What is the largest single precision number we can represent?

  This can be derived from the bit pattern in the 32 bit representation. The exponent is allocated 7 bits and for the largest value it has to have all the bits set to 1. The mantissa also has to have all the hexadecimal digits set to F, and the sign field has to be 0. So, we get the following bit pattern:

0	1111111	1111 1111 1111 1111 1111 1111

  The exponent in excess-64 representation represents the value of +63. The value of the total number then is:

$$\begin{aligned}
& (F \times 16^{-1} + F \times 16^{-2} + F \times 16^{-3} + F \times 16^{-4} + F \times 16^{-5} + F \times 16^{-6}) \times 16^{+63} \\
\approx \; & 1 \times 16^{+63} \approx 10^{+76}
\end{aligned}$$

- What is the most negative single precision number we can represent?

  The bit pattern for the mantissa and exponent would be similar to the largest positive single precision number given above (i.e., the absolute values should be the maximum in both cases). The sign bit would be negative in this case. So, the bit pattern would be

1	1111111	1111 1111 1111 1111 1111 1111

  Which going through a similar analysis as above would result in this number being approximately -10^{+76}.

- What is the smallest positive single precision number we can represent?

  The bit pattern in this case would be

0	0000000	0001 0000 0000 0000 0000 0000

  The first hexadecimal digit in the mantissa in this case is non-zero because of normalization and the smallest non-zero hex digit is 1. The exponent is the most negative it can be (-64 in excess-64 notation in this case). So putting all these together we get the value of the number to be

  $$+(1 \times 16^{-1}) \times 16^{-64} = 16^{-65} \approx 10^{-78}$$

- What is the least negative single precision number?

  This is similar to the smallest positive number, only the sign is negative. So the bit pattern in this case looks like:

1	0000000	0001 0000 0000 0000 0000 0000

  and its value is approximately -10^{-78}.

The IEEE floating point standard 754 is similar to the IBM 370 format. Here is the IEEE format for 32 bit single precision numbers:

sign	exponent	mantissa
1 bit	8 bits	23 bits

And for 64 bit double precision numbers we have:

sign	exponent	mantissa
1 bit	11 bits	52 bits

The radix in IEEE format is 2. Let's look at the single precision, 32-bit case. The IEEE standard defines special values for some of the special strings of bits which are given in the following table (s is the sign bit):

Exponent, e	**Mantissa**, f	**Value**
255	$\neq 0$	NaN (not-a-number)
255	0	$(-1)^s \infty$
$0 < e < 255$	—	$(-1)^s 2^{e-127}(1.f)$
0	$\neq 0$	$(-1)^s 2^{-126}(0.f)$
0	0	$(-1)^s 0$

For the values of the exponent in the range 1 to 254 (the values 0 and 255 are treated as special cases as is seen in the table above), the exponent is in excess notation with a range of -126 through 127. In this case, the mantissa is interpreted as a normalized number in which the bit immediately to the left of the fractional point is implied (i.e., it is not explicitely represented). Therefore, the mantissa is effectively 24 bits even though physically there are 23 bits allocated.