Number Systems

Introduction

The goal of this handout is to make you comfortable with the binary number system. We will correlate your previous knowledge of the decimal number system to the binary number system. That will lay the foundations on which our discussion of various representation schemes for numbers (both integer and real numbers) will be based.

Decimal Number System

Counting as we have been taught since kindergarten is based on the decimal number system. *Decimal* means base 10 (the prefix *dec*). In any number system, given the base (often referred to as *radix*), the number of digits that can be used to count is fixed. For example in the base 10 number system, the digits that can be used to count are 0,1,2,3,4,5,6,7,8,9.

Generalizing that for any base b, the first b digits (starting with 0) represent the digits that are used to count. When a number $\geq$ b has to be represented, the *place values* are used.

Example 1. Consider the number 1234. It can be represented as

$$1*10^3 + 2*10^2 + 3*10^1 + 4*10^0 \qquad (1)$$

Where:
- 1 is in the thousand's place
- 2 is in the hundred's place
- 3 is in the ten's place
- 4 is in the one's place.

The equation (1) is an expanded representation of 1234. The expanded representation has the advantage of making the base of the number system explicit.

Example 2. Consider the number 1234.567. It is represented as

$$1*10^3 + 2*10^2 + 3*10^1 + 4*10^0 + 5*10^{-1} + 6*10^{-2} + 7*10^{-3} \qquad (2)$$

Where:
- 5 is in the tenth's place
- 6 is in the hundredth's place
- 7 is in the thousandth's place

In equation (2), the representation includes digits both to the left and to the right of the decimal point.

Binary Number System

Binary means base 2 (the prefix *bi*). Based on our earlier discussion of the decimal number system, the digits that can be used to count in this number system are 0 and 1. The 0,1 used in the binary system are called *binary digits* (*bits*)

The *bit* is the smallest piece of information that can be stored in a computer. It can have one of two values 0 or 1. Think of a bit as a switch that can be either on or off. For example,

Bit	Value
0	OFF / FALSE
1	ON/ TRUE

Table 1. Interpreting Bit Values

From the hardware perspective, ON and OFF can be represented as voltage levels (typically 0V for logic 0 and +3.3 to +5V for logic 1). Since only two values can be stored in a bit, we combine a series of bits to represent more information. Again the concept of place values is applicable here as well.

Example 3. Consider the binary number 1101. It can be represented as

$$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \qquad (3)$$

This expanded notation also gives you the means of converting binary numbers directly into the equivalent decimal number.

$$8 + 4 + 0 + 1 = 13$$

Example 4. Consider the binary number 1101.101. It can be represented as:

$$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} \qquad (4)$$

The same notation is applicable to real numbers represented in binary notation. The equivalent decimal number is

$$13 + 0.5 + 0 + 0.125 = 13.625$$

To represent larger numbers, we have to group series of bits. Two of these groupings are of importance:
- **Nibble**   A nibble is a group of four bits
- **Byte**   A byte is a group of eight bits

The byte is the smallest addressable unit in most computers. The key components of a byte are shown below in figure 1.
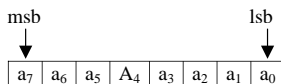
msb   lsb

| a₇ | a₆ | a₅ | A₄ | a₃ | a₂ | a₁ | a₀ |

Figure 1. Byte

The **m**ost **s**ignificant **b**it (msb) is the bit with the highest place value, while the **l**east **s**ignificant **b**it (lsb) denotes the bit position that has the lowest place value. To convert the byte to the equivalent integer number, the formula in (5) is used.

$$a_7 * 2^7 + a_6 * 2^6 + a_5 * 2^5 + a_4 * 2^4 + a_3 * 2^3 + a_2 * 2^2 + a_1 * 2^1 + a_0 * 2^0 \quad (5)$$

Self Exercise: What is the value of the bit pattern 11111111 ?

Answer: 255

Self- Exercise: What is the range of values represented by an 8-bit binary number?

Answer: 0 to 255.

Note: When we look at negative number representation using the same 8-bit number, the range will change.

When we want to represent a value larger than 255, we have to group bytes together to form *words*. The size of words is dependent on the underlying processor, but is usually an even number of bytes (typically 4 bytes).

In the *big-endian* system, the byte with the largest significance is stored first (big-end-first). Conversely, in the *little-endian* system, the byte with the least significance is stored first (little-end-first). The number 1025 in binary is 00000100 00000001.

The sequencing of bytes to form larger numbers leads to the issue of which is the first byte in the sequence. Many mainframe computers, particularly IBM mainframes, use a big-endian architecture. Most modern computers, including PCs, use the little-endian system. The PowerPC system is *bi-endian* because it can understand both systems

The term *endian* comes from Swift's "Gulliver's Travels" via the famous paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, 1980-04-01. The Lilliputians, being very small, had correspondingly small political problems. The *Big-Endian* and *Little-Endian* parties debated over whether soft-boiled eggs should be opened at the big end or the little end.

| Byte	Big-Endian	Little-Endian
0	00000100	00000001
1	00000001	00000100

Table 2. Big-Endian versus Little-Endian Representation of 00000100 00000001

The endian system may sometimes be used to represent the bit order within a byte. In this context, equation (5) refers to the *little-endian ordering* of bits within the byte.

Operations on Numbers

For any given number system, the operations of addition and subtraction are fundamental. Operations such as multiplication and division can be implemented using addition and subtraction. Again, we will correlate the addition and subtraction operations in the decimal number system to the binary number system.

Example 5. Consider the operation 145 + 256

Carry 1   Carry 1

```
    145
+   256
    401
```
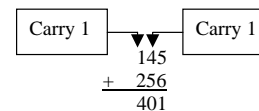
The algorithm works by starting at the least significant digit and working from right to left. At each place position, the digits are added and if the resulting number is a single digit, it is entered in the same place position in the sum. If on the other hand, the summation operation results in 2 digits, the digit of lower significance is entered into the place position of the sum and the digit of higher significance is added along with the other digits in the next most significant place (or is *carried over* to the next most significant place).

The same principle applies to binary addition.

Rule	Step	Result	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

Table 3.Rules for Binary Addition

Example 6. Consider the operation:

$$\begin{array}{r} \text{Carry 1} \\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0 \\ +\quad 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \\ \hline 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1 \end{array}$$

Note that at the 2nd bit position, there is a carry of 1 into the 3rd bit position (counting from the right with 0 being the first position).

Self-Exercise: Convert the numbers above into decimal to verify that the answer is correct.

Answer: 1 1 1 0 0 1 0 0 = 228, 0 0 0 0 0 1 0 1 = 5, 1 1 1 0 1 0 0 1 = 233 = 228+ 5

Decimal subtraction works very similar to decimal addition, the numbers are aligned to the same place values and the algorithm proceeds from right to left. The bottom digit is subtracted from the top digit, and the result written in the place value position in the result. If the top digit is less than the bottom digit, then we must 'borrow' from the next place value position. That means decrementing the top digit in the next significant position and adding the base to the top digit of this position before performing the subtraction. This operation gets even more complicated when there is a '0' in the next significant position.

Example 7. Consider the decimal operation 445 - 256:

$$\begin{array}{r} 445 \\ -\quad 256 \\ \hline 189 \end{array}$$

The decimal subtraction algorithm can get very complicated and the time taken to perform the subtraction can vary greatly. Since the binary addition algorithm is already understood and already implemented in hardware, it can be reused to also perform subtraction.

The operation x – y can be re-written as x+ (-y). That brings up the question – How are negative numbers represented in binary notation?

Negative Numbers

Negative binary numbers are represented by the '-' sign followed by the magnitude of the number. The computer however does not have a means of representing signs. The sign has to be captured in the bit pattern itself.

Signed Magnitude Representation

The signed magnitude representation uses the most significant bit to determine if the number is positive or negative. The advantage of this notation is that by examining the msb alone, it is possible to determine if the number is positive or negative. The disadvantage however is that one bit pattern is wasted (there are two possible representations for zero) and subtraction cannot be performed using addition alone. Table 4. shows the signed magnitude representation of numbers using 4 bits.

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

Table 4. Numbers using 4-bit signed magnitude representation

Note: +0 and –0 have different bit patterns.

Example 8. Consider the following operation 7 – 2. Substituting the bit patterns from the table:

$$\begin{array}{r} 0111 \\ +\quad 1010 \\ \hline \mathbf{1}\ 0001 \end{array}$$

The bit pattern 0001 is 1 but the result should by 5 0101.

One's Complement

The one's complement notation represents a negative number by inverting the bits in each place. The one's complement notation for a 4-bit number is shown in Table 5. Again the limitations of the sign magnitude representation are not overcome (there are two bit patterns used to represent 0 and the addition operation cannot be used to perform subtraction). The one's complement is important because it is very easy to perform the inversion operation in hardware and it forms the basis of computing the two's complement.

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

Table 5. Numbers using 1's complement representation

Example 9. Consider the following operation 7 – 2. Substituting the bit patterns from the table:

$$\begin{array}{r} 0111 \\ +\quad 1101 \\ \hline \mathbf{1}\,0100 \end{array}$$

The bit pattern 0100 is 4 but the result should by 5 0101.

Two's Complement

The two's complement notation builds on the one's complement notation. The algorithm goes as follows:

1. Compute the 1's complement
2. Add 1 to the result to get the 2's complement.

The two's complement notation has the advantages that the sign of the number can be computed by looking at the msb. The addition operation can be used to perform subtraction. Also, there is only one bit-pattern to represent '0' so an extra number can be represented. Table 6 summarizes the 2's complement notation for a 4-bit number.

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

Table 6. Numbers using 2's complement representation

Example 10. Consider the following operation 7 – 2. Substituting the bit patterns from the table:

$$\begin{array}{r} 0111 \\ +\quad 1110 \\ \hline \mathbf{1}\,0101 \end{array}$$

The bit pattern 0101 is 5, which is the expected result.

The limitation with the 2's complement notation is that the bit patterns are not in order i.e. comparing the bit patterns alone does not provide any information as to which number is larger.

Excess Notation

The excess notation is a means of representing both negative and positive numbers in a manner in which the order of the bit patterns is maintained. The algorithm for computing the excess notation bit pattern is as follows:

1. Add the excess value (2^{N-1}, where N is the number of bits used to represent the number) to the number.

2. Convert the resulting number into binary format.

The 2^{N-1} is often referred to as the *Magic Number* for computing the excess representation of the number (except that there is no magic in it). Table 7 presents all the numbers that can be represented using the excess-8 notation.

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

Table 7. Numbers using the Excess-8 representation

The number of bits used to represent a code in excess-8 is 4 bits ($2^{4-1} = 8$). Also, the bit patterns are in sequence (the largest number that can be represented has the bit pattern 1111).

Example 11. Consider the following operation 7 – 2. Substituting the bit patterns from the table:

$$\begin{array}{r} 1111 \\ +\quad 0110 \\ \hline \mathbf{1}\,0101 \end{array}$$

The result of the addition operation is the bit-pattern used for 5 in binary. The excess notation representation however takes longer to compute than the 2's complement notation. The excess notation will however play an important part in computing floating-point representations.

Self-Exercise: What is the range of numbers that can be represented using the Excess-2^{N-1} notation?

Answer: $2^{N-1}- 1$ to -2^{N-1}

Bias Notation

The excess notation is a special case of the biased notation. For instance, excess-8 is biased around 8 (i.e.0 has the bit pattern associated with decimal 8). Instead of using the magic number, any number (bias) can be used.

Note: This concept becomes important when we address the IEEE Single Precision Floating-Point standard.

Floating-Point Notation

The floating-point notation is used:
   a. To represent integers that are larger than the maximum value that can be held by a bit-pattern (the maximum value that can be held by 8 bits is 255).
   b. To represent real numbers.

Large Integers

Consider a really large number 1,234,567. The number requires seven places to represent the value. If the number of places available to represent the number is limited to say four places, certain digits have to be dropped. The selection of digits to be dropped is based on the value associated with the digit. In this case, we will drop the last three digits '567'. The resulting number is:

1,234,000

The loss of '567' is a loss of *precision* but if the most significant digits were to be eliminated, say '123', then the resulting number is **4,567,** which presents an even greater loss of precision.

Rules for determining significance (integers):
   1. A nonzero digit is always significant
   2. The digit '0' is significant if it lies between other significant digits
   3. The digit '0' is *never* significant if it precedes all the nonzero digits

Self-Exercise: What are the significant digits in 0012340?

Answer: 00<u>12340</u>

1,234,000 can be represented using only four digits as $1234 * 10^3$. This representation is called the ***exponential form***.

The exponential form consists of two parts:
- ***mantissa*** – (the significand) the significant digits (1234)
- ***exponent*** – the power to which the base is raised before being multiplied by the mantissa. (3).

Two special forms of representation are:

- $1.234 * 10^6$ – the ***scientific notation***, in which the decimal point is to the right of the most significant digit.
- $0.1234 * 10^7$ – the ***normalized notation***, in which the decimal point is to the left of the most significant digit.

Real Numbers

Mathematically, real numbers are set of rational and irrational numbers. A rational number is any number that can be represented as a ratio of two integers (a/b, b ≠ 0). Irrational numbers are real numbers that are not rational i.e. they cannot be represented as a ratio of two integers (typically numbers whose decimal expansion never ends and never enters a periodic pattern).

There are a number of ways of representing a real number in a computer.

1. One notation is the fixed point, wherein the decimal point (radix) is fixed at some position between the digits. The digits to the left of the radix are integer values and those to the right of the radix are fractions of some fixed unit. For example: $10.82 = 1 \times 10^1 + 0 \times 10^0 + 8 \times 10^{-1} + 2 \times 10^{-2}$, the radix is between 0 and 8. This notation is limited both in the range of numbers that can be represented as well as in the precision of the numbers that can be represented.

2. Another notation is to use rational notation (represent the real number as a ratio of two integers). This representation is not always possible because all real numbers are not necessarily rational!

3. The floating-point notation is the most common of the representation schemes. It is based on either the scientific or normalized representation of the number.

Binary Fixed-Point Notation

In the binary fixed-point notation, the radix position is fixed at a certain point within the bit pattern. To illustrate the notation, we will consider an 8-bit bit-pattern (byte) as shown below.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

We can define the fixed-point notation to be:

- a_7, a_6, a_5, a_4 – contain the integer part in two's complement form
- a_3, a_2, a_1, a_0 – contain the fractional part in normal binary form

$a_7a_6a_5a_4$	Value
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 1 1 1	-1
1 1 1 0	-2
1 1 0 1	-3
1 1 0 0	-4
1 0 1 1	-5
1 0 1 0	-6
1 0 0 1	-7
1 0 0 0	-8

$a_3a_2a_1a_0$	Value
0 0 0 0	0
0 0 0 1	0.0625
0 0 1 0	0.125
0 0 1 1	0.1875
0 1 0 0	0.250
0 1 0 1	0.3125
0 1 1 0	0.375
0 1 1 1	0.4375
1 0 0 0	0.5
1 0 0 1	0.5625
1 0 1 0	0.625
1 0 1 1	0.6875
1 1 0 0	0.75
1 1 0 1	0.8125
1 1 1 0	0.875
1 1 1 1	0.9375

Table 8. Integer part values   **Table 9**. Fractional Part Values

Tables 8,9 show the possible values that the different segments of the fixed-point notation can take.

Self-Exercise: What is the range of values that the fixed-point notation can take?

Answer: -8.9375 to 7.93725

Self-Exercise: What is the precision of the notation?

Answer: The precision is 0.0625.

The notation makes an assumption about the representation (primarily the location of the radix and the format used for the integer part). For two computers (or two programs for

that matter), they must understand the representation that is being used. This is captured by means of standards. At the end of this handout, we will discuss the IEEE standard used for representing floating point numbers.

Example12. What is the value represented by the bit pattern 11011110? Assume the radix point is between bit positions 3 and 4.

The bit pattern can be split into the integer part 1101 and the fractional part 1110.

$$
\begin{array}{rcl}
1101 & = & -3 \\
1110 & = & 0.875 \\
\\
11011110 & = & -3.875
\end{array}
$$

Modified Significance Rules

Rules for determining significance:

1. A '1' bit is always significant

2. The bit '0' is significant if it lies between other significant bits

3. The bit '0' is *never* significant if it precedes all 0's, even if it follows an embedded radix (in the example above, the radix is embedded between bit positions 3 and 4).

4. The bit '0' is significant if it follows an embedded radix point and other significant bits

Self-Exercise: What are the significant bits in the bit pattern00.010000?

Answer: 00.0<u>10000</u>

8-bit Floating Point Notation

The 8-bit floating point notation can be describe based on the byte shown below:
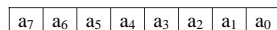
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Where
- a_7 – sign bit
- $a_6\, a_5\, a_4$ – exponent in excess-4 notation
- $a_3\, a_2\, a_1\, a_0$ – fractional part in normal binary

The algorithm for converting from **8-bit floating point to decimal** is detailed below:

1. Determine the value of the exponent
2. If the exponent x is negative,
   a. Add x leading 0's to the fractional part
   b. Insert a radix point before the 0's
3. If the exponent is positive,
   a. Move the radix point to the right x places.
4. Convert the binary number into decimal
5. Add the sign based on the a_7 bit.

Self-Exercise: Why does step 3 in the algorithm above not address adding zeroes?

Answer: From the excess table above, it is clear that the maximum positive exponent is 3. Moving the radix point right does not move to the right of the fraction part, hence, padding (adding leading 0's or trailing 0's) to the right is not needed.

Excess Number	Actual Value	Bit-Pattern
7	3	111
6	2	110
5	1	101
4	0	100
3	-1	011
2	-2	010
1	-3	001
0	-4	000

Table 10. Excess-4 Conversion Table

Example 13. What is the decimal value associated with 01111001(assume that the bit pattern is in 8-bit floating point format)?

Answer: Split the byte into the respective components,

<u>0</u>   <u>111</u>   <u>1001</u>

Step 1. Convert the excess-four exponent into its decimal equivalent.

111 = 3, hence the exponent = 3.

Step 2. Since the exponent is positive, move the radix point three place to the right.

$$
\begin{array}{rl}
\text{The binary number} & = 0.1001 * 2^3 \\
& = 100.1
\end{array}
$$

Step 3. Convert the binary number to decimal, 100.1 = 4.5

Step 4. Add the sign, hence 01111001 = 4.5

The algorithm for converting from **decimal to 8-bit floating point** is detailed below:

1. Set the sign bit (a_7) to 1 if the number is negative, 0 otherwise
2. Convert the number into binary representation.
3. Normalize the binary number (move the radix point to the left of the most significant bit).
4. Convert the exponent into excess-4 notation
5. Set a_6 a_5 a_4 to the exponent value.
6. Select the 4 most significant bits and enter them into the fraction part (a_3 a_2 a_1 a_0).
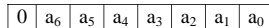
There are some limitations of the 8-bit floating-point notation. Some of them are listed below:
- The maximum positive exponent is 3
- The lowest negative exponent is –4
- The precision is determined by the exponent

Example 14: Convert 11/4 into 8-bit floating-point notation.
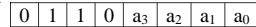
Answer:

Step 1.

0	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Step 2. Convert the number into binary form,11/4 = 2 ¾ = 0010.11

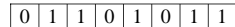Step 3. Normalize 0010.11 = 0.1011 * 2^2

Step 4. Convert the exponent to excess-4, 2 = 110

Step 5. Fill in the exponent

0	1	1	0	a_3	a_2	a_1	a_0

Step 6. Fill in the fraction

0	1	1	0	1	0	1	1

Hence 2 ¾ = 01101011

IEEE 754 Single Precision Floating Point Notation

Given that the normalization procedure always has a 1 in the first significant bit position, it is possible to use scientific notation instead and always implicitly assume that the first bit is 1.This allows us to gain an additional bit of precision in the representation.
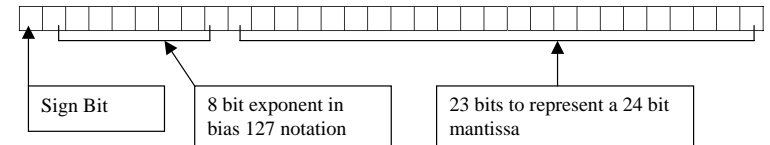

| Sign Bit | 8 bit exponent in bias 127 notation | 23 bits to represent a 24 bit mantissa |

Figure 2. IEEE 32 bit floating point notation

The *single precision* representation uses 32 bits as shown in Figure 2 above. The standard also defines a *double precision* standard that uses 64 bits. We will only be discussing the single precision standard.

The method for converting decimal numbers into the 754 standard representation is as follows:

1. If the number is negative
   a. set the sign bit to '1'
2. If the number is positive
   a. Set the sign bit to '0'
3. Convert the decimal number into binary form.
4. Convert the binary number into scientific notation (move the radix immediately right of the most significant bit).
5. Convert the exponent into bias-127 notation
6. Fill in the 8 bits demarcated for the exponent
7. Fill in the bits (except the most significant bit) into the mantissa from left to right. Pad any remaining spaces with 0's.

The exponent is computed as bias-127. The algorithm for computing the biased exponent is as follows:

1. Add 127 to the decimal value of the exponent (The exponent value is negative if the radix is moved to the left).
2. Convert the decimal value into binary

Example 15. Convert 194.375 into IEEE-754 single precision representation.

Answer:

Step1. Fill in the sign bit

0																															

Step 2. Convert the decimal number into binary.

$$194 \quad = \quad 11000010$$
$$0.375 \quad = \quad 0.011$$

$$194.375 \quad = \quad 11000010.011$$

Step 3. Convert to scientific notation

$$1.1000010011 * 2^7$$

Step 4. Convert the exponent into bias-127 notation

$$7 + 127 = 134 = 10000110$$

Step 5. Fill in the exponent

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

Step6. Fill in the mantissa without the most significant bit.

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

Hence 194.375 = 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0

Endnote