Number Systems, Base Conversions, and Computer Data Representation

Decimal and Binary Numbers

When we write decimal (base 10) numbers, we use a positional notation system. Each digit is multiplied by an appropriate power of 10 depending on its position in the number:

For example:

$$843 = 8 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$$
$$= 8 \times 100 + 4 \times 10 + 3 \times 1$$
$$= 800 + 40 + 3$$

For whole numbers, the rightmost digit position is the one's position ($10^0 = 1$). The numeral in that position indicates how many ones are present in the number. The next position to the left is ten's, then hundred's, thousand's, and so on. Each digit position has a weight that is ten times the weight of the position to its right.

In the decimal number system, there are ten possible values that can appear in each digit position, and so there are ten numerals required to represent the quantity in each digit position. The decimal numerals are the familiar zero through nine (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

In a positional notation system, the number base is called the radix. Thus, the base ten system that we normally use has a radix of 10. The term radix and base can be used interchangeably. When writing numbers in a radix other than ten, or where the radix isn't clear from the context, it is customary to specify the radix using a subscript. Thus, in a case where the radix isn't understood, decimal numbers would be written like this:

$$127_{10} \quad 11_{10} \quad 5673_{10}$$

Generally, the radix will be understood from the context and the radix specification is left off.

The binary number system is also a positional notation numbering system, but in this case, the base is not ten, but is instead two. Each digit position in a binary number represents a power of two. So, when we write a binary number, each binary digit is multiplied by an appropriate power of 2 based on the position in the number:

For example:

$$101101 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$
$$= 32 + 8 + 4 + 1$$

In the binary number system, there are only two possible values that can appear in each digit position rather than the ten that can appear in a decimal number. Only the numerals 0 and 1 are used in binary numbers. The term 'bit' is a contraction of the words 'binary' and 'digit', and when talking about binary numbers the terms bit and digit can be used interchangeably. When talking about binary numbers, it is often necessary to talk of the number of bits used to store or represent the number. This merely describes the number of binary digits that would be required to write the number. The number in the above example is a 6 bit number.

The following are some additional examples of binary numbers:

$$101101_2 \quad 11_2 \quad 10110_2$$

Conversion between Decimal and Binary

Converting a number from binary to decimal is quite easy. All that is required is to find the decimal value of each binary digit position containing a 1 and add them up.

For example: convert 10110_2 to decimal.

```
1 0 1 1 0
 \   \ \_____________1 x 2^1 =  2
  \   \_____________1 x 2^2 =  4
   \_________________1 x 2^4 = 16
                              22
```

Another example: convert 11011_2 to decimal

```
1 1 0 1 1
 \ \   \ \__________1 x 2^0 =  1
  \ \   \___________1 x 2^1 =  2
   \ \_____________1 x 2^3 =  8
    \_______________1 x 2^4 = 16
                              27
```

The method for converting a decimal number to binary is one that can be used to convert from decimal to any number base. It involves using successive division by the radix until the dividend reaches 0. At each division, the remainder provides a digit of the converted number, starting with the least significant digit.

An example of the process: convert 37_{10} to binary

```
37 / 2 = 18       remainder 1       (least significant digit)
18 / 2 = 9        remainder 0
9 / 2 = 4         remainder 1
4 / 2 = 2         remainder 0
2 / 2 = 1         remainder 0
1 / 2 = 0         remainder 1       (most significant digit)
```

The resulting binary number is: 100101

Another example: convert 93_{10} to binary

```
93 / 2 = 46       remainder 1       (least significant digit)
46 / 2 = 23       remainder 0
23 / 2 = 11       remainder 1
11 / 2 = 5        remainder 1
 5 / 2 = 2        remainder 1
 2 / 2 = 1        remainder 0
 1 / 2 = 0        remainder 1       (most significant digit)
```

The resulting binary number is: 1011101

Hexadecimal Numbers

In addition to binary, another number base that is commonly used in digital systems is base 16. This number system is called hexadecimal, and each digit position represents a power of 16. For any number base greater than ten, a problem occurs because there are more than ten symbols needed to represent the numerals for that number base. It is customary in these cases to use the

ten decimal numerals followed by the letters of the alphabet beginning with A to provide the needed numerals. Since the hexadecimal system is base 16, there are sixteen numerals required. The following are the hexadecimal numerals:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The following are some examples of hexadecimal numbers:

10_{16} 47_{16} $3FA_{16}$ $A03F_{16}$

The reason for the common use of hexadecimal numbers is the relationship between the numbers 2 and 16. Sixteen is a power of 2 ($16 = 2^4$). Because of this relationship, four digits in a binary number can be represented with a single hexadecimal digit. This makes conversion between binary and hexadecimal numbers very easy, and hexadecimal can be used to write large binary numbers with much fewer digits. When working with large digital systems, such as computers, it is common to find binary numbers with 8, 16 and even 32 digits. Writing a 16 or 32 bit binary number would be quite tedious and error prone. By using hexadecimal, the numbers can be written with fewer digits and much less likelihood of error.

To convert a binary number to hexadecimal, divide it into groups of four digits starting with the rightmost digit. If the number of digits isn't a multiple of 4, prefix the number with 0's so that each group contains 4 digits. For each four digit group, convert the 4 bit binary number into an equivalent hexadecimal digit. (See the Binary, BCD, and Hexadecimal Number Tables at the end of this document for the correspondence between 4 bit binary patterns and hexadecimal digits)

For example: Convert the binary number 10110101 to a hexadecimal number

Divide into groups for 4 digits	1011 0101
Convert each group to hex digit	B 5
	$B5_{16}$

Another example: Convert the binary number 0110101110001100 to hexadecimal

Divide into groups of 4 digits	0110 1011 1000 1100
Convert each group to hex digit	6 B 8 C
	$6B8C_{16}$

To convert a hexadecimal number to a binary number, convert each hexadecimal digit into a group of 4 binary digits.

Example: Convert the hex number 374F into binary

	3 7 4 F
Convert the hex digits to binary	0011 0111 0100 1111
	0011011101001111_2

There are several ways in common use to specify that a given number is in hexadecimal representation rather than some other radix. In cases where the context makes it absolutely clear that numbers are represented in hexadecimal, no indicator is used. In much written material where the context doesn't make it clear what the radix is, the numeric subscript 16 following the hexadecimal number is used. In most programming languages, this method isn't really feasible, so there are several conventions used depending on the language. In the C and C++ languages, hexadecimal constants are represented with a '0x' preceding the number, as in: 0x317F, or 0x1234, or 0xAF. In assembler programming languages that follow the Intel style, a hexadecimal constant begins with a numeric character (so that the assembler can distinguish it from a variable

name), a leading '0' being used if necessary. The letter 'h' is then suffixed onto the number to inform the assembler that it is a hexadecimal constant. In Intel style assembler format: 371Fh and 0FABCh are valid hexadecimal constants. Note that: A37h isn't a valid hexadecimal constant. It doesn't begin with a numeric character, and so will be taken by the assembler as a variable name. In assembler programming languages that follow the Motorola style, hexadecimal constants begin with a '$' character. So in this case: $371F or $FABC or $01 are valid hexadecimal constants.

Binary Coded Decimal Numbers

Another number system that is encountered occasionally is Binary Coded Decimal. In this system, numbers are represented in a decimal form, however each decimal digit is encoded using a four bit binary number.

For example: The decimal number 136 would be represented in BCD as follows:

```
136 = 0001 0011 0110
        1    3    6
```

Conversion of numbers between decimal and BCD is quite simple. To convert from decimal to BCD, simply write down the four bit binary pattern for each decimal digit. To convert from BCD to decimal, divide the number into groups of 4 bits and write down the corresponding decimal digit for each 4 bit group.

There are a couple of variations on the BCD representation, namely packed and unpacked. An unpacked BCD number has only a single decimal digit stored in each data byte. In this case, the decimal digit will be in the low four bits and the upper 4 bits of the byte will be 0. In the packed BCD representation, two decimal digits are placed in each byte. Generally, the high order bits of the data byte contain the more significant decimal digit.

An example: The following is a 16 bit number encoded in packed BCD format:

```
01010110 10010011
```

This is converted to a decimal number as follows:

```
0101 0110 1001 0011
  5    6    9    3
```

The value is 5693 decimal

Another example: The same number in unpacked BCD (requires 32 bits)

```
00000101 00000110 00001001 00000011
     5        6        9        3
```

The use of BCD to represent numbers isn't as common as binary in most computer systems, as it is not as space efficient. In packed BCD, only 10 of the 16 possible bit patterns in each 4 bit unit are used. In unpacked BCD, only 10 of the 256 possible bit patterns in each byte are used. A 16 bit quantity can represent the range 0-65535 in binary, 0-9999 in packed BCD and only 0-99 in unpacked BCD.

Fixed Precision and Overflow.

So far, in talking about binary numbers, we haven't considered the maximum size of the number. We have assumed that as many bits are available as needed to represent the number. In most computer systems, this isn't the case. Numbers in computers are typically represented using a fixed number of bits. These sizes are typically 8 bits, 16 bits, 32 bits, 64 bits and 80 bits. These sizes are generally a multiple of 8, as most computer memories are organized on an 8 bit byte basis. Numbers in which a specific number of bits are used to represent the value are called fixed precision numbers. When a specific number of bits are used to represent a number, that determines the range of possible values that can be represented. For example, there are 256

possible combinations of 8 bits, therefore an 8 bit number can represent 256 distinct numeric values and the range is typically considered to be 0-255. Any number larger than 255 can't be represented using 8 bits. Similarly, 16 bits allows a range of 0-65535.

When fixed precision numbers are used, (as they are in virtually all computer calculations) the concept of overflow must be considered. An overflow occurs when the result of a calculation can't be represented with the number of bits available. For example when adding the two eight bit quantities: 150 + 170, the result is 320. This is outside the range 0-255, and so the result can't be represented using 8 bits. The result has overflowed the available range. When overflow occurs, the low order bits of the result will remain valid, but the high order bits will be lost. This results in a value that is significantly smaller than the correct result.

When doing fixed precision arithmetic (which all computer arithmetic involves) it is necessary to be conscious of the possibility of overflow in the calculations.

Signed and Unsigned Numbers.

So far, we have only considered positive values for binary numbers. When a fixed precision binary number is used to hold only positive values, it is said to be *unsigned*. In this case, the range of positive values that can be represented is 0 -- 2^n-1, where n is the number of bits used. It is also possible to represent signed (negative as well as positive) numbers in binary. In this case, part of the total range of values is used to represent positive values, and the rest of the range is used to represent negative values.

There are several ways that signed numbers can be represented in binary, but the most common representation used today is called two's complement. The term two's complement is somewhat ambiguous, in that it is used in two different ways. First, as a representation, two's complement is a way of interpreting and assigning meaning to a bit pattern contained in a fixed precision binary quantity. Second, the term two's complement is also used to refer to an operation that can be performed on the bits of a binary quantity. As an operation, the two's complement of a number is formed by inverting all of the bits and adding 1. In a binary number being interpreted using the two's complement representation, the high order bit of the number indicates the sign. If the sign bit is 0, the number is positive, and if the sign bit is 1, the number is negative. For positive numbers, the rest of the bits hold the true magnitude of the number. For negative numbers, the lower order bits hold the complement (or bitwise inverse) of the magnitude of the number. It is important to note that two's complement representation can only be applied to fixed precision quantities, that is, quantities where there are a set number of bits.

Two's complement representation is used because it reduces the complexity of the hardware in the arithmetic-logic unit of a computer's CPU. Using a two's complement representation, all of the arithmetic operations can be performed by the same hardware whether the numbers are considered to be unsigned or signed. The bit operations performed are identical, the difference comes from the interpretation of the bits. The interpretation of the value will be different depending on whether the value is considered to be unsigned or signed.

For example: Find the 2's complement of the following 8 bit number

```
  00101001

  11010110        First, invert the bits
+ 00000001        Then, add 1
= 11010111
```

The 2's complement of 00101001 is 11010111

Another example: Find the 2's complement of the following 8 bit number

```
  10110101

  01001010        Invert the bits
+ 00000001        then add 1
= 01001011
```

The 2's complement of 10110101 is 01001011

The counting sequence for an eight bit binary value using 2's complement representation appears as follows:

01111111	7Fh	127	largest magnitude positive number
01111110	7Eh	126	
01111101	7Dh	125	
…			
00000011	03h		
00000010	02h		
00000001	01h		
00000000	00h		
11111111	0FFh	-1	
11111110	0FEh	-2	
11111101	0FDh	-3	
…			
10000010	82h	-126	
10000001	81h	-127	
10000000	80h	-128	largest magnitude negative number

Notice in the above sequence that counting up from 0, when 127 is reached, the next binary pattern in the sequence corresponds to -128. The values jump from the greatest positive number to the greatest negative number, but that the sequence is as expected after that. (i.e. adding 1 to –128 yields –127, and so on.). When the count has progressed to 0FFh (or the largest unsigned magnitude possible) the count wraps around to 0. (i.e. adding 1 to –1 yields 0).

ASCII Character Encoding

The name ASCII is an acronym for: American Standard Code for Information Interchange. It is a character encoding standard developed several decades ago to provide a standard way for digital machines to encode characters. The ASCII code provides a mechanism for encoding alphabetic characters, numeric digits, and punctuation marks for use in representing text and numbers written using the Roman alphabet. As originally designed, it was a seven bit code. The seven bits allow the representation of 128 unique characters. All of the alphabet, numeric digits and standard English punctuation marks are encoded. The ASCII standard was later extended to an eight bit code (which allows 256 unique code patterns) and various additional symbols were added, including characters with diacritical marks (such as accents) used in European languages, which don't appear in English. There are also numerous non-standard extensions to ASCII giving different encoding for the upper 128 character codes than the standard. For example, The character set encoded into the display card for the original IBM PC had a non-standard encoding for the upper character set. This is a non-standard extension that is in very wide spread use, and could be considered a standard in itself.

Some important things to note about ASCII code:

1) The numeric digits, 0-9, are encoded in sequence starting at 30h
2) The upper case alphabetic characters are sequential beginning at 41h
3) The lower case alphabetic characters are sequential beginning at 61h

4) The first 32 characters (codes 0-1Fh) and 7Fh are control characters. They do not have a standard symbol (glyph) associated with them. They are used for carriage control, and protocol purposes. They include 0Dh (CR or carriage return), 0Ah (LF or line feed), 0Ch (FF or form feed), 08h (BS or backspace).
5) Most keyboards generate the control characters by holding down a control key (CTRL) and simultaneously pressing an alphabetic character key. The control code will have the same value as the lower five bits of the alphabetic key pressed. So, for example, the control character 0Dh is carriage return. It can be generated by pressing CTRL-M. To get the full 32 control characters a few at the upper end of the range are generated by pressing CTRL and a punctuation key in combination. For example, the ESC (escape) character is generated by pressing CTRL-[(left square bracket).

Conversions Between Upper and Lower Case ASCII Letters.

Notice on the ASCII code chart that the uppercase letters start at 41h and that the lower case letters begin at 61h. In each case, the rest of the letters are consecutive and in alphabetic order. The difference between 41h and 61h is 20h. Therefore the conversion between upper and lower case involves either adding or subtracting 20h to the character code. To convert a lower case letter to upper case, subtract 20h, and conversely to convert upper case to lower case, add 20h. It is important to note that you need to first ensure that you do in fact have an alphabetic character before performing the addition or subtraction. Ordinarily, a check should be made that the character is in the range 41h–5Ah for upper case or 61h-7Ah for lower case.

Conversion Between ASCII and BCD.

Notice also on the ASCII code chart that the numeric characters are in the range 30h-39h. Conversion between an ASCII encoded digit and an unpacked BCD digit can be accomplished by adding or subtracting 30h. Subtract 30h from an ASCII digit to get BCD, or add 30h to a BCD digit to get ASCII. Again, as with upper and lower case conversion for alphabetic characters, it is necessary to ensure that the character is in fact a numeric digit before performing the subtraction. The digit characters are in the range 30h-39h.

Binary, BCD and Hexadecimal Number Tables

Powers of 2:

2^0 1
2^1 2
2^2 4
2^3 8
2^4 16
2^5 32
2^6 64
2^7 128
2^8 256
2^9 512
2^{10} 1024
2^{11} 2048
2^{12} 4096
2^{13} 8192
2^{14} 16384
2^{15} 32768
2^{16} 65536

Hexadecimal Digits

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

BCD Digits

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Equivalent Numbers in Decimal, Binary and Hexadecimal Notation:

Decimal	Binary	Hexadecimal
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F
16	00010000	10
17	00010001	11
31	00011111	1F
32	00100000	20
63	00111111	3F
64	01000000	40
65	01000001	41
127	01111111	7F
128	10000000	80
129	10000001	81
255	11111111	FF
256	0000000100000000	0100
32767	0111111111111111	7FFF
32768	1000000000000000	8000
65535	1111111111111111	FFFF

ASCII Character Set

Low Order Bits		High Order Bits 0000 **0**	0001 **1**	0010 **2**	0011 **3**	0100 **4**	0101 **5**	0110 **6**	0111 **7**
0000	**0**	NUL	DLE	Space	0	@	P	`	p
0001	**1**	SOH	DC1	!	1	A	Q	a	q
0010	**2**	STX	DC2	"	2	B	R	b	r
0011	**3**	ETX	DC3	#	3	C	S	c	s
0100	**4**	EOT	DC4	$	4	D	T	d	t
0101	**5**	ENQ	NAK	%	5	E	U	e	u
0110	**6**	ACK	SYN	&	6	F	V	f	v
0111	**7**	BEL	ETB	'	7	G	W	g	w
1000	**8**	BS	CAN	(	8	H	X	h	x
1001	**9**	HT	EM	)	9	I	Y	i	y
1010	**A**	LF	SUB	*	:	J	Z	j	z
1011	**B**	VT	ESC	+	;	K	[	k	{
1100	**C**	FF	FS	,	<	L	\	l	|
1101	**D**	CR	GS	-	=	M	]	m	}
1110	**E**	SO	RS	.	>	N	^	n	~
1111	**F**	SI	US	/	?	O	_	o	DEL