

## Review of Data Representation & Binary Operations

**Dhananjai M. Rao**

(raodm@muohio.edu)

**CSA Department**

**Miami University**

### **1. Introduction**

In digital computers all data including numbers, characters, and strings are ultimately represented, stored, and processed as a set of Binary Digits or bits. A bit stores either a binary value, namely a “0” or a “1”. Essentially a bit can be used to represent only 2 different values based on interpretation and context. For example, if bits are associated with colors, then “0” could be used to indicate the color red while “1” could be used to indicate the color “blue”. On the other hand, if bits are associated with characters, then “0” could be used to indicate the Greek letter  $\alpha$  while “1” indicates  $\beta$ .

In order to represent more than 2 different values, a group of bits need to be used. Given,  $n$  bits,  $2^n$  different values can be represented using different combinations of “1”s and “0”s. For example, given 3 bit positions,  $2^3$  or 8 different values can be represented. Recollect that the actual value that each combination represents depends on the context and interpretation. For example, assume that 3 bits are used to represent colors then one possible interpretation is shown in the table<sup>1</sup> below:

Bit pattern	Color	Bit pattern	Color
000	Black	100	Red
001	Blue	101	Magenta
010	Green	110	Tan
011	Cyan	111	Grey

Similar to colors, various combinations of bit values are also used to represent numbers and characters in computers. However, in order to ensure consistent interpretation of bits a number of conventions and standards have been developed to represent numbers and characters in modern computers.

---

<sup>1</sup> The color codes shown in the table are actually used in practice in today’s computers to display colored characters in text display modes.

## 2. Unsigned Integer Number Representation

Decimal numbers are represented using base-10 system using 10 distinct symbols namely “0” to “9” and increasing powers of 10. For example, the number 678 (six hundred and seventy eight) consists of 3 decimal digits, with 8 in the units ( $10^0$ ) place, 7 in the tens ( $10^1$ ) place, and 6 in the hundreds ( $10^2$ ) place. In other words,  $678 = (6*10^2) + (7*10^1) + (8*10^0)$ .

Similar to the decimal system of numbers represented using base-10, numbers are represented with bits (“0”s and “1”s) using a base-2 system or *binary*. For example, the binary number  $110_2$  corresponds to  $(1*2^2)+(1*2^1)+(0*2^0) = 6_{10}$  in the decimal system. Note the use of subscripts ( $X_2$  or  $Y_{10}$ ) indicating the base in which the numbers are expressed. It is important to remember that  $100_2 \neq 100_{10}$ .

### i. Binary (base-2) to Decimal (base-10) conversion:

As indicated in the example, binary numbers are converted to decimal numbers through successive multiplication by powers of 2 and adding them. Here are a few more examples to further illustrate the process:

$$1101_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 13_{10}$$

$$111_2 = 1*2^2 + 1*2^1 + 1*2^0 = 7_{10}$$

$$1_2 = 1*2^0 = 1_{10}$$


$$0_2 = 0*2^0 = 0_{10}$$

### Useful facts:

- Even numbers always have a “0” in the units place in a unsigned binary number.
- Odd numbers always have a “1” in the units place in an unsigned binary number.
- Given  $n$  bits, the range of unsigned numbers that can be represented in binary is  $0_{10}$  to  $2^n-1$ . For example if the number of bits  $n = 8$ , then the range of numbers that can be presented are  $0_{10}$  to  $2^8-1=255$ .

**ii. Decimal (base-10) to Binary (base-2) conversion:**

Converse to the earlier approach, decimal numbers are converted to binary through successive division by 2 and using the remainders as the binary digit. For example, the decimal number  $27_{10}$  is converted to binary through successive division (until the quotient is 1) and writing the remainders in reverse order, as shown below:

		<i>Remainder</i>	
2	26		
2	13	0	
2	6	1	
2	3	0	
2	1	1	
	1	1	

$27_{10} = 11010_2$

Here are a couple of more examples to further illustrate the process:

		<i>Remainder</i>
2	15	
2	7	1
2	3	1
	1	1

$15_{10} = 1111_2$

		<i>Remainder</i>
2	36	
2	18	0
2	9	0
2	4	1
2	2	0
	1	0

$36_{10} = 100100_2$

Useful facts:

- In order to represent an unsigned decimal number  $k$  in binary at least  $2\sqrt{k}$  bits are needed.

### 3. Signed Integer Representation (2's Complement)

Signed integers may be represented using a variety of different schemes. Amongst the various schemes the 2's Complement mechanism for representing signed integers is the most prevalent representation used in modern computers. 2's complement representation is used for representing both positive and negative numbers in the following manner:

Note: In 2's complement, the maximum number of bits that may be used to represent a number must be known and plays a crucial role to enable representation.

#### i. Decimal to 2's Complement Conversion:

a. **Positive number:** Positive numbers are represented using the same approach for unsigned numbers explained earlier in Section 2.ii. Zeros are padded to the beginning of the binary number to expand it to the needed bit size. For example the **8-bit** binary representation for  $+13_{10} = 00001101_2$  (note the 8 bits in the number with 4 leading zero bits)

b. **Negative number:** In the case of negative numbers, the following steps are performed:

1. The number is considered as positive and the zero padded binary representation is first obtained as illustrated in Section 2.ii. For example, if  $-14_{10}$  is to be represented in 2's complement, first the 8-bit binary representation of  $14_{10}$  is determined:  $14_{10} = 00001110_2$

2. Next all the bits are complemented, that is "1"s are made "0"s and "0"s are made "1"s. Continued from earlier step:

$$00001110_2 \Rightarrow \text{Complement} \Rightarrow 11110001_2$$

3. Next binary  $1_2$  is added to the number to obtain the final representation in of a negative number in 2's complement. Continuing with example from step 2:

$$\begin{array}{r} 11110001 \quad + \\ 00000001 \\ \hline 11110010 \end{array}$$

That is,  $-14_{10} = 11110010_2$  in 2's complement representation.

Here are a couple of more examples to illustrate the process of representing positive and negative numbers in 8-bit 2's complement:

$$\begin{aligned}27_{10} &= \underline{\mathbf{00011011}}_2 \\ -27_{10} &= (00011011_2 \text{ Complement } 11100100_2 + 1_2) = \underline{\mathbf{11100101}}_2 \\ -121_{10} &= (01111001_2 \text{ Complement } 10000110_2 + 1_2) = \underline{\mathbf{10000111}}_2\end{aligned}$$

**Note:** In the above 2's complement binary representation, the most significant bit is bold and underlined (that is: 0 or 1) while the least significant bit is just underlined.

Useful facts:

- ☑ Given  $n$  bits, the range of numbers that can be represented using 2's complement is  $-2^{n-1}$  to  $2^{n-1} - 1$ .
- ☑ The most significant bit (bit with highest power of 2) should always be 0 for positive numbers.
- ☑ The most significant bit (bit with highest power of 2) should always be 1 for negative numbers.
- ☑ In the case of negative numbers, the actual binary representation changes as the number of bits used for representation changes.

ii. **2's Complement to Decimal Conversion:**

Converting a binary number represented in 2's complement is performed in the following manner:

1. Inspect the most significant bit of the binary number. If the bit is a "0" then it is a positive number and perform step 2. If the most significant bit is a "1" then it is a negative number and perform step 3.
2. For positive numbers, simply convert the number using the standard conversion of multiplication by increasing powers of 2 and adding them together. See earlier section 2.i for details and examples.
3. For negative numbers, perform the following steps:
  - (a) First all the bits are complemented, that is "1"s are made "0"s and "0"s are made "1"s.
  - (b) Next binary  $1_2$  is added to the number to obtain the binary representation in unsigned form.
  - (c) Convert the binary number to decimal (using repeated multiplication by powers of 2 and addition as illustrated in Section 2.i) and place a negative sign before the number.

**iii. Examples of 2's Complement to Decimal Conversion:**

- Convert  $10000111_2$  to decimal:
  - i. To begin with, note that the most significant bit is a 1. Therefore, this value in 2's complement represents a negative number!
  - ii. First, complement all the bits (convert all "1"s to "0"s and "0"s to "1"s) to get:  $10000111_2$  *Complement*  $01111000_2$
  - iii. Next add binary  $1_2$  to the above value to get:  $01111000_2 + 1_2 = 01111001_2$
  - iv. Convert the binary to decimal (using repeated multiplication by powers of 2 and addition as illustrated in Section 2.i) to get: 121
  - v. Now place a negative sign before the number to get:  $-121_{10}$
  
- Convert  $00000111_2$  to decimal:
  - i. To begin with, note that the most significant bit is a 0. Therefore, this value in 2's complement represents a positive number!
  - ii. Simply convert the binary to decimal (using repeated multiplication by powers of 2 and addition as illustrated in Section 2.i) to get:  $7_{10}$
  
- Convert  $11111000_2$  to decimal:
  - vi. To begin with, note that the most significant bit is a 1. Therefore, this value in 2's complement represents a negative number!
  - vii. First, complement all the bits (convert all "1"s to "0"s and "0"s to "1"s) to get:  $11111000_2$  *Complement*  $00000111_2$
  - viii. Next add binary  $1_2$  to the above value to get:  $00000111_2 + 1_2 = 00001000_2$
  - ix. Convert the binary to decimal (using repeated multiplication by powers of 2 and addition as illustrated in Section 2.i) to get: 8
  - x. Now place a negative sign before the number to get:  $-8_{10}$

## 4. Hexadecimal Representation (Base-16)

Similar to decimal (base-10) or binary (base -2) system, Hexadecimal system uses Base-16. Note that the base (10, 2, or 16) indicates the number of unique symbols required to represent the different values. In other words, hexadecimal (hex for short) representation requires 16 unique symbols to represent values from 0 through 15. Accordingly, the first 10 values 0 through 9 are represented using the same decimal digits while values 10 through 15 are represented using the first 6 English alphabets namely A through F. The following table lists the decimal, binary, and hexadecimal values of the first 20 numbers:

<i>Decimal</i>	<i>Binary(5-bits)</i>	<i>Hexadecimal</i>	<i>Decimal</i>	<i>Binary(5-bits)</i>	<i>Hexadecimal</i>
0 <sub>10</sub>	00000 <sub>2</sub>	0 <sub>16</sub>	10 <sub>10</sub>	01010 <sub>2</sub>	A <sub>16</sub>
1 <sub>10</sub>	00001 <sub>2</sub>	1 <sub>16</sub>	11 <sub>10</sub>	01011 <sub>2</sub>	B <sub>16</sub>
2 <sub>10</sub>	00010 <sub>2</sub>	2 <sub>16</sub>	12 <sub>10</sub>	01100 <sub>2</sub>	C <sub>16</sub>
3 <sub>10</sub>	00011 <sub>2</sub>	3 <sub>16</sub>	13 <sub>10</sub>	01101 <sub>2</sub>	D <sub>16</sub>
4 <sub>10</sub>	00100 <sub>2</sub>	4 <sub>16</sub>	14 <sub>10</sub>	01110 <sub>2</sub>	E <sub>16</sub>
5 <sub>10</sub>	00101 <sub>2</sub>	5 <sub>16</sub>	15 <sub>10</sub>	01111 <sub>2</sub>	F <sub>16</sub>
6 <sub>10</sub>	00110 <sub>2</sub>	6 <sub>16</sub>	16 <sub>10</sub>	10000 <sub>2</sub>	10 <sub>16</sub>
7 <sub>10</sub>	00111 <sub>2</sub>	7 <sub>16</sub>	17 <sub>10</sub>	10001 <sub>2</sub>	11 <sub>16</sub>
8 <sub>10</sub>	01000 <sub>2</sub>	8 <sub>16</sub>	18 <sub>10</sub>	10010 <sub>2</sub>	12 <sub>16</sub>
9 <sub>10</sub>	01001 <sub>2</sub>	9 <sub>16</sub>	19 <sub>10</sub>	10011 <sub>2</sub>	13 <sub>16</sub>

### i. Binary to Hexadecimal Conversion:

Binary to hexadecimal conversion is pretty straightforward and is performed by grouping sets of 4 bits (or nibbles) and writing the corresponding hex value for it. The grouping is performed from the least significant bit to the most significant bit (or right to left). Here is a detailed example illustrating the conversion process:

- Convert 10110110<sub>2</sub> to hex
  - First group the bits in sets of 4 from left to right to get: 1011, 0110
  - For each nibble write out the hexadecimal equivalent to get: B, 6
  - Write the digits together to obtain the hex representation: B6<sub>16</sub>

Here are a few more examples:

- 101<sub>2</sub> = 5<sub>16</sub>
- 10111<sub>2</sub> = (1, 0111) = 17<sub>16</sub>
- 53<sub>10</sub> = 110101<sub>2</sub> = (11, 0101) = 35<sub>16</sub>

ii. **Hexadecimal to Binary conversion:**

Conversion from hexadecimal to binary is pretty straightforward and can be performed by simply writing out the binary nibbles for each digit. Here are a few examples:

- $27_{16} = 0010, 0111 = 100111_2$
- $5A_{16} = 0101, 1010 = 1011010_2$
- $CC_{16} = 1100, 1100 = 11001100_2$

iii. **Hexadecimal to Decimal conversion:**

Hexadecimal numbers can be converted to decimal values through successive multiplication by increasing powers of 16 and summing up the values. Here are a few examples to illustrate this technique:

- $27_{16} = (2 * 16^1) + (7 * 16^0) = 32 + 7 = 39_{10}$
- $DCC_{16} = (13 * 16^2) + (12 * 16^1) + (12 * 16^0) = 3328 + 192 + 12 = 3532_{10}$
- $10_{16} = (1 * 16^1) + (0 * 16^0) = 16 + 0 = 16_{10}$
- $9_{16} = (9 * 16^0) = 9 = 9_{10}$

## 5. Character Representation (ASCII)

Similar to numbers, different combinations of bits are used represent different characters in modern computers. The most prevalent standard for representing characters is ASCII. Default ASCII representation uses 7-bits to represents characters. Here are some standard representations obtained from the ASCII standard:

<i>ASCII Code</i>	<i>Character</i>
32	Space
42	*
48 – 57	0 – 9
65 – 90	A – Z
97 – 122	a – z

## 6. Bit-wise Operators

Bit-wise operations are important and powerful transformations to numbers. Consequently, almost all high level programming languages include bit-wise operators in them. In order to effectively use bit-wise operators, you must have a good understanding of their operations and work. All of the operations share the following characteristics:

- Each operator works only with 1 bit at a time. However, the overall result is always a group of bits and is considered as a single unit.
- Operands to the bit-wise operators must be exactly the same size.
- Bit-wise operations are performed without paying heed to the actual representation of the binary number. However, the results from the operations are interpreted using the appropriate number representation.
- Given a non-binary number, you have to first convert it to binary, perform the specified bit-wise operation, and then convert the result back to the original representation.

Note: The following bit order is assumed in the following discussions:							
bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
MOST SIGNIFICANT BIT				LEAST SIGNIFICANT BIT			

Some of the commonly use bit-wise operations are briefly described in the following subsections.

### Bit-wise NOT ( ~ ):

The bit-wise NOT operator ( $\sim$ ) is a unary operator and requires only 1 operand. It inverts all the bits in a given binary number. Here are a few examples of how the bit-wise NOT operator works:

1. Bit-wise NOT of  $5_{10}$  represented using 8-bit unsigned binary number representation is:
  - a. 8-bit unsigned representation of  $5_{10} = 00000101_2$ .
  - b. Bit-wise not of above number =  $11111010_2$ .
  - c. Interpreting above number as unsigned binary, result =  $250_{10}$ .
2. Bit-wise not of  $-5_{10}$  represented using 8-bit 2's complement binary number representation is:
  - a. 8-bit 2's complement representation of  $-5_{10} = 11111011_2$ .
  - b. Bit-wise not of above number =  $00000100_2$ .
  - c. Interpreting above number as unsigned binary, result =  $4_{10}$ .

## Bit-wise AND ( & )

Bit-wise AND operator (&) is a binary operator and requires 2 binary numbers for operation. The two operands must have the same number of bits. Each bit from each operand are AND-ed together according to the truth table shown to the right to obtain the final result.

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Here is an example of how this operator works:

1. Assume 8-bit 2's complement representation of numbers. The bit wise and of  $3_{10}$  and  $-2_{10} =$ 
  - a. The 8-bit 2's complement representation of  $3_{10} = 00000011_2$ .
  - b. The 8-bit 2's complement representation of  $2_{10} = 11111110_2$ .
  - c. Bit-wise AND of the above two binary numbers =  

```
00000011    &
11111110
-----
00000010
-----
```
  - d. Interpreting the above result ( $00000010_2$ ) as a 8-bit 2's complement number we get::  $2_{10}$
  - e. Consequently  $3_{10} \& -2_{10} = 2_{10}$ .

Bit-wise AND operation is often used to set bits in a number to 0 without affecting other bits. For this operation you need to create a suitable unsigned binary number, with appropriate bits set to 0 (all other bits set to 1) and bit-wise AND it with the given number. Here is an example to illustrate this concept.

- Using bit-wise AND, set bit-2 and bit-4 of  $75_{16}$  to  $0_2$ s without affecting other bits and show result of operation.
  - a. In order to set bit-2 and bit-4 to  $0_{16}$ , the value must be bit-wise AND-ed with:  $11101011_2$  (Note how only bit-2 and bit-4 are 0)
  - b.  $75_{16} = 01110101_2$
  - c. Result of bit wise AND:  $01110101_2 \& 11101011_2 = 01100001 = 61_{16}$ .

## Bit-wise OR ( | )

Bit-wise OR operator (|) is a binary operator and requires 2 binary numbers for operation. The two operands must have the same number of bits. Each bit from each operand are OR-ed together according to the truth table shown to the right to obtain the final result.

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Here is an example of how this operator works:

2. Assume 8-bit 2's complement representation of numbers. The bit wise OR of  $3_{10}$  and  $-2_{10} =$ 
  - a. The 8-bit 2's complement representation of  $3_{10} = 00000011_2$ .
  - b. The 8-bit 2's complement representation of  $-2_{10} = 11111110_2$ .
  - c. Bit-wise AND of the above two binary numbers =
 

```

00000011   |
11111110
-----
11111111
-----
          
```
  - d. Interpreting the above result ( $11111111_2$ ) as a 8-bit 2's complement number we get::  $-1_{10}$
  - e. Consequently  $3_{10} | -2_{10} = -1_{10}$ .

Bit-wise OR operation is often used to set bits in a number to 1 without affecting other bits. For this operation you need to create a suitable unsigned binary number, with appropriate bits set to 1 and bit-wise OR it with X. Here is an example to illustrate this concept.

- Using bit-wise OR, set bit-2 and bit-4 of  $61_{16}$  to  $1_2$ s without affecting other bits and indicate result of operation.
  - d. In order to set bit-2 and bit-4 to  $1_{16}$ , the value must be bit-wise OR-ed with:  $00010100_2$
  - e.  $61_{16} = 01100001_2$
  - f. Result of bit wise OR:  $01100001_2 | 00010100_2 = 01110101 = 75_{16}$ .

### Bit-wise XOR ( ^ )

Bit-wise XOR operator (^) is a binary operator and requires 2 binary numbers for operation. The two operands must have the same number of bits. Each bit from each operand are XOR-ed together according to the truth table shown to the right to obtain the final result.

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Here is an example of how this operator works:

3. Assume 8-bit 2's complement representation of numbers. The bit-wise XOR of  $3_{10}$  and  $-2_{10}$  =
  - a. The 8-bit 2's complement representation of  $3_{10} = 00000011_2$ .
  - b. The 8-bit 2's complement representation of  $-2_{10} = 11111110_2$ .
  - c. Bit-wise AND of the above two binary numbers =

```
00000011    ^
11111110
-----
11111101
-----
```
  - d. Interpreting the above result ( $11111101_2$ ) as a 8-bit 2's complement number we get::  $-3_{10}$
  - e. Consequently  $3_{10} \wedge -2_{10} = -3_{10}$ .

## 7. Shift Operators

Shift operations are one of the fastest operations supported by a microprocessor. They enable several mathematical operations and bit level manipulation of data. They are important and almost all high level programming languages include bit-wise operators in them. All of the shift operations share the following characteristics:

- Each operator works only with binary operand at a time. However, the shift operators require a second operand (a number) that indicates number of shifts to be performed.
- Shift operations are performed without paying heed to the actual representation of the binary number. However, the results from the operations are interpreted using the appropriate number representation.
- Shift operations do not change the size of the binary number.
- Given a non-binary number, you have to first convert it to binary (using suitable representation), perform the specified bit-wise operation, and then convert the result back to the original representation.

The two shift operations that are supported by almost all microprocessors are briefly described in the following subsections.

### Shift Left ( << )

As the name suggests, the shift left operator shifts bits to the left (assuming the most significant bit is written first). Left most bits are lost and 0s are padded to the right to ensure that the result is of the same size as the source operand. Example of the shift left operator is shown below:

1. Assuming 8-bit 2's complement representation, the result of shifting  $-2_{10}$  to the left 3 times (written as:  $-2_{10} \ll 3_{10}$ ) is:
  - a. The 8-bit 2's complement representation of  $-2_{10} = 11111110_2$ .
  - b. Shifting above binary number to left  $3_{10}$  times we get:  $11110000_2$ .
  - c. Interpreting the above number as 8-bit 2's complement we get:  $-16_{10}$ .

### Shift Right ( >> )

As the name suggests, the shift right operator ( $\gg$ ) shifts bits to the right (assuming the most significant bit is written first). Right most bits are lost and 0s are padded to the left to ensure that the result is of the same size as the source operand. Example of the shift right operator is shown below:

2. Assuming 8-bit 2's complement representation, the result of shifting  $-2_{10}$  to the left 3 times (written as:  $-2_{10} \ll 3_{10}$ ) is:
  - a. The 8-bit 2's complement representation of  $-2_{10} = 11111110_2$ .
  - b. Shifting above binary number to right  $3_{10}$  times we get:  $00011111_2$ .
  - c. Interpreting the above number as 8-bit 2's complement we get:  $31_{10}$ .