CLARION TECHNICAL BULLETIN

Bulletin #119

Using Binary Numbers

Overview

This technical bulletin discusses binary numbers. It explains how to convert decimal numbers to binary numbers, how to shift binary numbers, and how to use ANDs, ORs, and XORs with binary numbers.

Using Binary Numbers

A binary number is a way to represent numbers. The numbers we use everyday are decimal numbers. This means that the set of numbers belongs to the base 10. Base 10 (decimal) numbers consist of any combination of the numbers 0 - 9. The decimal number 10 itself is a combination of a 1 and 0.

Zero is the smallest positive decimal number. Zero can be thought of as 0, 00, 000, etc. Leading zeroes do not change the value of the number. (This information is obvious, but keep it in mind.) The numbers 1, 10, 100, etc. are not the same as 1, 01, 001, etc. Both series of numbers are a combination of a single 1 and some number of zeroes; so why aren't they the same? The answer, of course, is that the placement of the digits is important.

What does it mean when we think of the decimal number 10? The rightmost digit represents the number of ones we have. The next rightmost digit represents the number of tens we have. And the next represents the number of hundreds we have, and so on.

Decimal number 10 is: $(1 * 10^{1}) + (0 * 10^{0})$.

Decimal number 46 is: $(4 * 10^{1}) + (6 * 10^{0})$.

Decimal number 281 is: $(2 * 100^{2}) + (8 * 10^{1}) + (1 * 1^{0})$

Binary numbers are the set of numbers belonging to the base 2. Base 2 (binary) numbers consist of any combination of the numbers 0 - 1. The binary number 10 itself is a combination of a 1 and 0 but does not have the same value as the decimal number 10.

What does it really mean when we think of the binary number 10? The rightmost digit represents the number of ones we have. The next rightmost digit represents the number of twos we have. And the next rightmost digit represents the number of fours we have. The next digit represents the number of eights we have, and so on.

Binary number 10 is: $(1 * 2^{1}) + (0 * 1^{0})$

Binary number 101 is: $(1 * 4^{1}) + (0 * 2^{0}) + (1 * 1^{0})$

The binary number 10 is the same as the decimal number 2. The binary number 101 is the same as the decimal number 5. Any decimal number can be represented as a binary number.

Consider the following chart:

POWER / PLACEMENT

BASE	7	6	5	4	3	2	1	0
10	10,000,000	1,000,000	100,000	10,000	1,000	100	10	1
2	128	64	32	16	8	4	2	1

Converting Decimal to Binary

How do we convert decimal numbers to binary numbers? The best way to explain it is to actually do some examples. Let's convert the decimal number 26 to binary.

	BASE 2 Positions	128	64	32	16	8	4	2	1

	Is 26 >= 128	NO	0.....B
	Is 26 >= 64	NO	00.....B
	Is 26 >= 32	NO	000.....B
	Is 26 >= 16	YES	0001....B
(26-16)	Is 10 >= 8	YES	00011...B
(10-8)	Is 2 >= 4	NO	000110..B
	Is 2 >= 2	YES	0001101.B
(2-2)	Is 0 >= 1	NO	00011010B

As you can see, the decimal number 26 is the same as the binary number 11010. Any decimal number can be converted to binary in a similiar fashion. The leading zeroes can be ignored, just like in the decimal number 0026. For practice, try converting the numbers 25 and 50 (you should get 11001 and 110010).

Likewise, we can easily convert a binary number to a decimal.

	BASE 2 Positions	128	64	32	16	8	4	2	1
	Our Number	0	0	0	1	1	0	1	0

DECIMAL = (0 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (0 * 4) + (1 * 2) + (0 * 1)

= 16 + 8 + 2

= 26

ANDs / ORs / XORs with Binary Numbers

Before we begin to explain ANDs and ORs, let's begin with the concept of truth tables. If someone said "I am a boy AND I am a girl," this would be a FALSE statement. It is a FALSE statement regardless of which condition is TRUE and which is FALSE. This is FALSE because the AND condition requires BOTH conditions to be TRUE in order for the statement to be TRUE. Obviously, one condition is TRUE and the other condition is FALSE. If someone said "I am dead AND I am invisible," this would also be a FALSE statement. Both conditions are FALSE and so the statement is FALSE. You have just learned the truth table for ANDs. If binary 1s represent TRUE conditions and binary 0s represent FALSE conditions, we get the following truth tables.

Cond1	AND	Cond2		Result
TRUE	AND	TRUE	=	TRUE
TRUE	AND	FALSE	=	FALSE
FALSE	AND	TRUE	=	FALSE
FALSE	AND	FALSE	=	FALSE

Cond1	AND	Cond2		Result
1	AND	1	=	1
1	AND	0	=	0
0	AND	1	=	0
0	AND	0	=	0

Now let's try to AND two binary numbers together. In Clarion, BAND(6,12) returns 4. Let's see why. Decimal 6 is also 00000110 binary. Decimal 12 is 00001100 binary. Let's place these two numbers into our truth table:

(6) Cond1	AND	(12) Cond2	=	(4) Result
0	AND	0	=	0
0	AND	0	=	0
0	AND	0	=	0
0	AND	0	=	0
0	AND	1	=	0
1	AND	1	=	1
1	AND	0	=	0
0	AND	0	=	0

All this is good to know, but what use is it? Since binary numbers consist of the numbers 0 and 1, many programmers use these Binary digITS (BITS) to represent off(0) and on(1) states. Let's say we have 8 machines, any of which may be running at any time.

If machine 0 is on, then the 0th power (BIT 0) of two digit is on(1).
If machine 1 is on, then the 1th power (BIT 1) of two digit is on(1).

. .
. .

If machine 7 is on, then the 7th power (BIT 7) of two digit is on(1), etc.

Let's say that the program calls this function and it returns a value of 135 (decimal). Which machines are on? By treating the decimal value as a binary number (10000111) we realize that machines 7, 2, 1 and 0 are on. Let's assume that this program only cares if machine 5 is on. Thus, we must check to see that BIT 5 is on. To do this, we take 135 ANDed with 32 (00100000). In Clarion, this would be BAND(135,32).

	(135) Cond1		(32) Cond2		(0) Result
	1	AND	0	=	0
	0	AND	0	=	0
BIT 5->	0	AND	1	=	0
	0	AND	0	=	0
	0	AND	0	=	0
	1	AND	0	=	0
	1	AND	0	=	0
	1	AND	0	=	0

You'll notice that the result of the AND is zero. This is because machine 5 is off (not on).

Let's build a truth table for ORing conditions. If someone said "I am a girl OR I am a boy," this would be a TRUE statement. Obviously, one condition is TRUE and the other condition is FALSE. It is a TRUE statement regardless of which condition is TRUE and which is FALSE. This is TRUE because the OR condition requires at least ONE condition to be TRUE in order for the statement to be TRUE. If someone said "I am dead OR I am invisible," this would be a FALSE statement. Both conditions are FALSE and so the statement is FALSE. You have just learned the truth table for ORs. If binary 1s represent TRUE conditions and binary 0s represent FALSE conditions we get the following truth tables.

Cond1	OR	Cond2		Result
TRUE	OR	TRUE	=	TRUE
TRUE	OR	FALSE	=	TRUE
FALSE	OR	TRUE	=	TRUE
FALSE	OR	FALSE	=	FALSE

Cond1	OR	Cond2		Result
1	OR	1	=	1
1	OR	0	=	1
0	OR	1	=	1
0	OR	0	=	0

Recalling the function before which returned the state of 8 machines, let's say that we wish to turn on machine 5 and leave the states of the other machines the same. To do this, we take 135 ORed with 32 (00100000). The code in Clarion is BOR(135,32).

	(135) Cond1		(32) Cond2		(167) Result
	1	OR	0	=	1
	0	OR	0	=	0
BIT 5->	0	OR	1	=	1
	0	OR	0	=	0
	0	OR	0	=	0
	1	OR	0	=	1
	1	OR	0	=	1
	1	OR	0	=	1

As you can see, by ORing BIT 5 with a 1 (TRUE), we get a 1 in the result. What would happen if BIT 5 was already on? Since 1 OR 1 is TRUE, we would get the identical result.

Now for some XORing. XOR can be very useful for such things as toggling bits on and off. Let's first look at a truth table, since it is more difficult to express how XOR works with simple sentences.

Cond1	XOR	Cond2		Result
TRUE	XOR	TRUE	=	FALSE
TRUE	XOR	FALSE	=	TRUE
FALSE	XOR	TRUE	=	TRUE
FALSE	XOR	FALSE	=	FALSE

Cond1	XOR	Cond2		Result
1	XOR	1	=	0
1	XOR	0	=	1
0	XOR	1	=	1
0	XOR	0	=	0

It is quite obvious how XOR works: any two statements that are both equal results in FALSE, any two that are unequal results in TRUE. Let's XOR two numbers, 134 (10000110) and 27 (00011011):

	(134) Cond1	XOR	(27) Cond2	=	(157) Result
	1	XOR	0	=	1
BIT 6->	0	XOR	0	=	0
BIT 5->	0	XOR	0	=	0
	0	XOR	1	=	1
	0	XOR	1	=	1
	1	XOR	0	=	1
BIT 1->	1	XOR	1	=	0
	0	XOR	1	=	1

This would be represented by BXOR(134,27) in Clarion. Notice that in each case where both numbers are the same, the outcome is FALSE (BITS 1, 5 and 6).

Have you ever wondered how certain strings of Christmas lights seem to "move?" Well, one possible way of doing it would be to use the XOR function. Pretend that each BIT represents a light, 1 for on and 0 for off. Here's our string of lights: 10101010. All we need to do is simply XOR this value with 11111111 over and over again.

Cond1	XOR	Cond2		Result
1	XOR	1	=	0
0	XOR	1	=	1
1	XOR	1	=	0
0	XOR	1	=	1
1	XOR	1	=	0
0	XOR	1	=	1
1	XOR	1	=	0
0	XOR	1	=	1

01010101 XOR 11111111 = 10101010
10101010 XOR 11111111 = 01010101
01010101 XOR 11111111 = 10101010
10101010 XOR 11111111 = 01010101

.
.
.

Voila! "Moving" lights. By toggling each light (BIT) on and off, this produces an effect making the lights move. Follow the 1's in the example above to better understand how it works.

Many encryption methods use XOR to "disguise" data. For example, take the sentence "Hello World." This sentence is represented in memory by the decimal ASCII values (H)72, (e)101, (l)108, (l)108, (o)111, ( )32, (W)87, (o)101, (r)114, (l)108, and (d)100. Now we need a "key," and any number will do - we'll use 52. A simple encryption would XOR each value in the data with the key like this:

ENCRYPT

ASCII	XOR	KEY		Result	
72	XOR	52	=	124	¦
101	XOR	52	=	81	Q
108	XOR	52	=	88	X
108	XOR	52	=	88	X
111	XOR	52	=	91	[
32	XOR	52	=	20	^T
87	XOR	52	=	99	c
111	XOR	52	=	91	Q
114	XOR	52	=	70	F
108	XOR	52	=	88	X
100	XOR	52	=	80	P

DECRYPT

ASCII	XOR	KEY		Result	
124	XOR	52	=	72	H
81	XOR	52	=	101	e
88	XOR	52	=	108	l
88	XOR	52	=	108	l
91	XOR	52	=	111	o
20	XOR	52	=	32	
99	XOR	52	=	87	W
91	XOR	52	=	111	o
70	XOR	52	=	114	r
88	XOR	52	=	108	l
80	XOR	52	=	100	d

Our resulting encrypted data looks like this: "¦QXX[^TcQFXP." Try to read that! Now, to get our original sentence back again, simply take each of the resulting numbers and re-XOR them with 52. Our original "Hello World" is back. Of course, this is a very simple encryption routine, but by using XOR in other different ways, we can create much more difficult patterns.

SHIFTing Binary Numbers

Shifting binary numbers is a method of multiplying or dividing binary numbers by some power of 2. Let's first think about shifting decimal numbers.

First, I'll explain the concept of shifting to the right. If I took the number 234 and shifted all the numbers to the right 1 place, I would get the number 023. If I shifted right 1 more place, I would get the number 002. Notice that each shift to the right 1 place is equivalent to dividing the decimal number by 10. Zeroes are added in the leftmost position to replace the vacated position. The same concept applies for binary numbers. Let's start with the number 234 (11101010 binary). If I shift right all the binary digits (bits) 1 place, I would get the binary number 01110101 (117 decimal). Shifting to the right 1 more place would create the number 00111010 (58 decimal). Notice that each shift to the right 1 place is equivalent to dividing by 2.

Shifting to the left is similiar to shifting to the right. Zeroes are added in the rightmost position to replace the vacated position. Each shift to the left 1 place is equivalent to multiplying by 2.

In Clarion, BSHIFT(value,count) is how you shift binary numbers. Say, for example, that machines 0 - 3 are on, and machines 4 - 7 are off. One way of turning off the lower four machines and turning on the upper four machines would be to shift the BITS to the left 4 places. To do this we would use BSHIFT(15,4):

Original Setup	00001111
1st Shift	00011110
2nd Shift	00111100
3rd Shift	01111000
4th Shift	11110000

Since each shift to the left is equivilant to multiplying by two, we can assume BSHIFT(15,4) = 15 * 2 * 2 * 2 * 2 = 11110000B. To shift to the RIGHT, we would use BSHIFT(15,-4).

Realize that if we shift 11110000 to the left one more time, we lose the high BIT, ending up with 11100000. Likewise, if we shift 00001111 to the right one we lose the low BIT, resulting in 00000111.