Bitwise Operations Primer

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

OzoBlockly Mode 5 (Master Level) contains three bitwise math blocks shown in Figure 1. If you really want to consider yourself a master of Level 5, then you should know how to work with bitwise operations. Why? There are several reasons, one of which is that these operations exist in most modern programming languages. Another reason is that bitwise operations are essential in low-level programming of device drivers, graphics, assembly of packets in communications protocols, compression, and encryption. A third reason is that bitwise operations are extensively used in embedded systems, hardware register operations, control systems, and PLCs (Programmable Logic Controllers), just to mention a few.

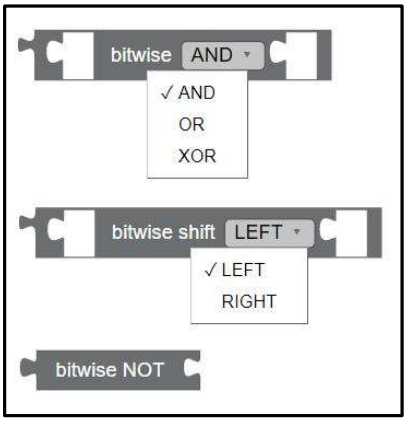

Figure 1

As can be seen in Figure 1, OzoBlockly has the three bitwise operations AND, OR, and XOR, which act on the binary representations of two integers. AND returns 1 only if ***both*** compared bits are one, otherwise 0 is returned. OR returns 1 if ***either*** compared bit is 1, otherwise 0 is returned. XOR (exclusive or) returns 1 if ***exactly one*** of the compared digits is 1, otherwise 0 is returned.

The bitwise shift operations perform a logical shift on the binary form of the input integer. The leftmost argument is the input integer, and the rightmost argument is the number of bits to be shifted either right or left. The shift is a ***logical shift***, not an arithmetic shift. This means that 0 is introduced on the left or right, dependent on the programmer's choice in the block.

The bitwise NOT block is used on a single integer number block or variable block. This block changes all 0's to 1's and all 1's to 0's in the binary representation of the number. We will not be studying this block in this lesson.

In order to keep things from getting complicated, we will just be working with positive numbers in this lesson. OzoBlockly uses 8-bit signed numbers with the leftmost bit 0 for positive numbers and 1 for negative numbers. Since we will be using only positive numbers and know that the leftmost bit is therefore 0, then we need only be concerned with the remaining 7 bits in our discussion. These seven bits will be referred to as a *word*.

Bit Numbering

In a decimal number such as 3759, the digit 9 is the least significant digit, and the digit 3 is the most significant digit. The digit 9 is in the 1's place, 5 in the 10's place, 7 in the 100's place, and 3 in the 1000's place. Similarly, in a seven-bit binary number such as 1010110, the least significant bit (LSB) is in the 1's place on the far right, and the most significant bit (MSB) is on the far left. In a seven-bit binary number, it is common to number the bits from right to left with bit numbers 0 through 6, respectively, as shown in Figure 2. We note that the place value is $2^{\text{bit number}}$. This is sometimes known as *LSB 0-bit numbering*. We will use this bit numbering scheme throughout this lesson.

Place Value	2^6 = 64	2^5 = 32	2^4 = 16	2^3 = 8	2^2 = 4	2^1 = 2	2^0 = 1
Bit number	6	5	4	3	2	1	0
Data word	1	0	1	0	1	1	0

Figure 2

Packing Data into Bits

Bitwise operations are frequently used to save memory by *packing data into bits* in a single word instead of placing each data item into a separate word. This is particularly useful when you have huge amounts of data but limited memory. As an example, suppose that you want to keep track of properties of a piece of text in a specific word processing document. Is the text underlined, is it bold, is it crossed out, etc.? These are all questions with yes/no (or 1/0) answers. We could keep all of this information in a single word as shown in Figure 3.

Text Property	**Bold**	*Italic*	Underlined	$x^{\text{Subscript}}$	$X_{\text{Subscript}}$	~~Crossed Out~~	Future Use
Place Value	64	32	16	8	4	2	1
Bit number	6	5	4	3	2	1	0
Data word	1	0	1	0	1	1	0

Figure 3

We see by examining the 1's in the data word of Figure 3 that our text is bold, underlined, subscripted, and cross out. And—we have kept all of this information in a single word! We will now make use of this example while looking at several uses of bitwise operations.

Turning Bits Off

Let's suppose that we want to turn off subscripting in Figure 3, leaving everything else unchanged in our data word. We see that subscripting is controlled by bit number 2.

Bitwise operations often make use of what is known as a ***mask*** word. The mask for turning off bit number 2 is 1111011. The ***AND*** bitwise operation is then applied to the mask 1111011 and our data 1010110, as shown in Figure 4. 1 AND X = X. 0 AND X = 0. Therefore, data bits ANDed with 1 are unchanged. Data bits ANDed with 0 are changed to 0, as happens to bit number 2. We have turned off subscripting in our example.

Note in Figure 3 that the AND operator for many languages is the & character.

	6	5	4	3	2	1	0	Bit Number
	1	0	1	0	1	1	0	Data
AND (&)	1	1	1	1	0	1	1	Mask
=	1	0	1	0	0	1	0	Result
Bit Number 2 is now OFF ↑								

Figure 4

Now let's see how the provided OzoBlockly program can perform the bitwise operation to turn bit 2 off, i.e., turn off subscripting in our example. Open the program *bitwise.ozocode* in OzoBlockly. It is a fairly large program, but all that you need to deal with is the blocks when scrolling to the far left. These are shown in Figure 5.

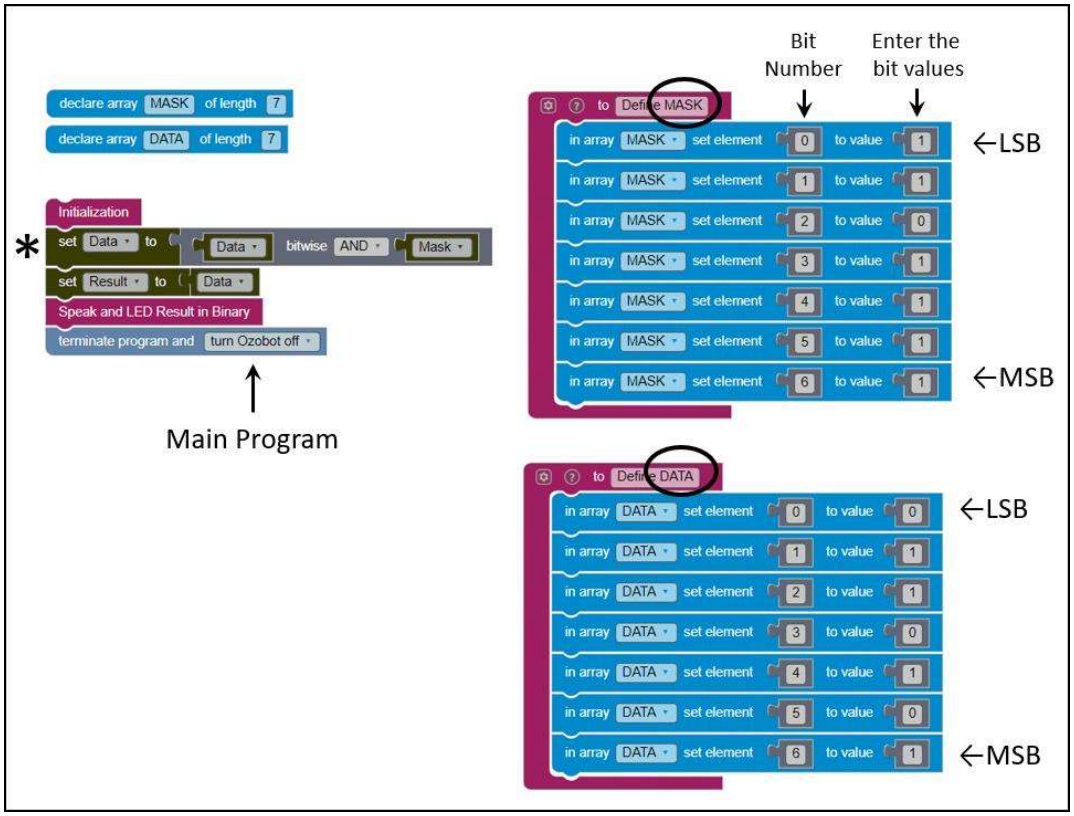

Figure 5

All you need to do is enter the bit values for the mask and the data in the column shown in Figure 5. Note that we are using LSB-0 bit numbering, so you need to pay attention to the location of the LSB and MSB as shown.

The block in the main program with an asterisk (*) performs the AND bitwise operation involving the data and mask words, storing the result back in the data word. This has, in effect, turned off subscripting in our example.

To run the *bitwise.ozocode* OzoBlockly program after entering the bit values for the data and mask and after loading the program into Evo, follow these steps:

1. Press Evo's button once to turn Evo on.
2. When the front lights turn blue, quickly double-press the button.
3. The program will start running, and the top light will turn white for 3 seconds.
4. Evo will then indicate the result beginning with its LSB and proceeding through the MSB. It will speak either 0 or 1, dependent on the value of the bit in the result. As an aid to students with hearing impairment, Evo will blink red for 0's and green for 1's.

Exercise 1

Load the bitwise.ozocode program into Evo. It already contains the data and mask bits required to turn off bit 2, the subsripting bit. Run the program on Evo. Evo should indicate the result by speaking the bit values and by its top LED showing red for 0 and green for 1.

Turning Bits On

Let's start with the original data word of Figure 3. The "italic" bit, bit number 5, is currently off since its value is 0. Suppose that we want to turn "italic" on, leaving everything else unchanged. The mask for turning on bit number 5 is 0100000. To accomplish this task, the OR bitwise operation is then applied to the mask 0100000 and data 1010110, as shown in Figure 6. 0 OR X = X. 1 OR X = 1. Therefore, data bits ORed with 0 are unchanged. Data bits ORed with 1 are changed to 1. We have turned "italic" on for our example. Note in Figure 6 that the OR operator for many languages is the | character.

	6	5	4	3	2	1	0	Bit Number
	1	0	1	0	1	1	0	Data
OR (|)	0	1	0	0	0	0	0	Mask
=	1	1	1	0	1	1	0	Result
↑ Bit Number 5 is now ON								

Figure 6

Exercise 2

*Modify the mask in the OzoBlockly program bitwise.ozocode so that it will turn off bit 5, the "italic" bit in our example. Change the operation in the * block of Figure 5 so that it will perform a bitwise OR instead of AND. Load and run the program on Evo. Evo should indicate the result by speaking the bit values and by its top LED showing red for 0 and green for 1.*

Selecting a Bit—Determining if It Is 0n or Off

There are situations in bitwise operations when you simply want to know if a particular bit is on (1) or off (0). You don't want to change any bits in the data word. For example, let's imagine in our word processing application of Figure 3 that we want to know if the "Underlined" bit (bit number 4) is on or off. The mask for doing this would be 0010000. The ***AND*** bitwise operation is then applied to the mask 0010000 and our data 1010110, as shown in Figure 7. 1 AND X = X. 0 AND X = 0. Therefore, data bits ANDed with 1 are unchanged. Data bits ANDed with 0 remain 0 or are changed to 0. The result is a positive number, with a place value of 2^4 = 16. But we don't have to know the place value. We only need to know that the result is positive.

	6	5	4	3	2	1	0	Bit Number
	1	0	1	0	1	1	0	Data
AND (&)	0	0	1	0	0	0	0	Mask
=	0	0	1	0	0	0	0	Result
↑ Positive result means bit 4 is on (1).								

Figure 7

Further, let's imagine in our word processing application of Figure 3 that we want to know if the "Italic" bit (bit number 5) is on or off. The mask for doing this would be 0100000. The ***AND*** bitwise operation is then applied to the mask 0100000 and our data 1010110, as shown in Figure 8. Since all the bits in the result are 0, then the result is 0.

	6	5	4	3	2	1	0	Bit Number
	1	0	1	0	1	1	0	Data
AND (&)	0	1	0	0	0	0	0	Mask
=	0	0	0	0	0	0	0	Result
↑ Zero result means bit 5 is off (0).								

Figure 8

In conclusion, this technique tells us that if the result is positive, then the selected bit is on (1). If the result is zero, then the selected bit is off (0).

In order to select a bit and determine if it is on or off, a modified main program is needed in our *bitwise.ozocode* program. To make use of this modified main program, open the provided program *BitwiseOnOrOff.ozocode* in OzoBlockly. You will notice the modified main program appears as shown in Figure 9. The *if…do…else* block handles the logic: "*if the result is positive, then the selected bit is on (1). If the result is zero, then the selected bit is off (0)*". You will also note that the mask has been set to that of Figure 7, where we were interested in determining if the "Underlined" bit (bit number 4) is on or off.

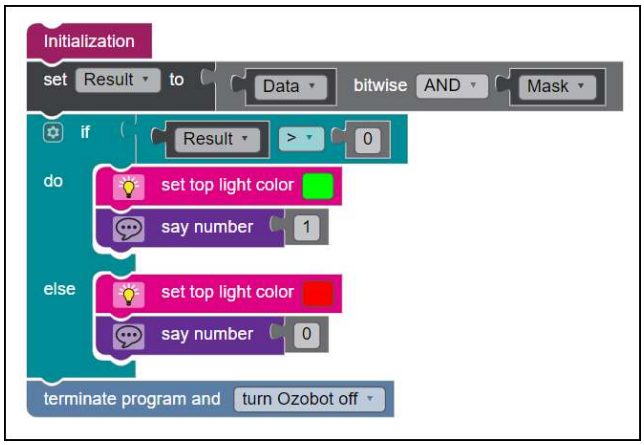

Figure 9

Exercise 3

a. Load the program BitwiseOnOrOff.ozocode into Evo and then run the program. It should indicate that the "Underlined" bit (bit number 4) is on (1). Evo will speak "1" and the top light should be green.

b. Adjust the mask so that it agrees with that of Figure 8, in which we were interested in determining if the "Italic" bit (bit number 5) is on or off. Reload and run this program on Evo. Evo will speak "0" and the top light should be red.

Inverting Bits

There are circumstances under which it might be desirable to invert a bit in a data word. To invert means to change 1 to 0 or 0 to 1. For example, suppose that you want to invert every bit in a data word. This can be done in a single bitwise operation by making use of the ***XOR*** (e***X***clusive ***OR***) operation. The XOR operation outputs 1 only when the inputs differ. For clarification, Figure 10 shows the difference between OR and XOR.

A	B	A OR B	A XOR B
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Figure 10

Unlike OR, which outputs a 1 when both inputs are 1, XOR ***excludes*** this case by outputting a 0—thus the term "exclusive or". This is necessary as the word "or" is ambiguous when both operands are 1.

Figure 11 shows how we can invert all of the bits in a data word. The mask consists of all 1's. (If you just wanted to invert bit numbers 2, 3, and 4, then the mask would be 0011100.) Note the use of the caret (^) symbol to represent the EOR bitwise operation.

	6	5	4	3	2	1	0	Bit Number
	1	0	1	0	1	1	0	Data
EOR (^)	1	1	1	1	1	1	1	Mask
=	0	1	0	1	0	0	1	Result
All of the data bits have been inverted.								

Figure 11

Exercise 4
Load the bitwise.ozocode program into the OzoBlockly workspace. Adjust the block with the asterisk (*) so that it uses the EOR bitwise operation. Set the mask bits so that every bit in the data word will be inverted. Then load and run the program on Evo. Verify that all of the data bits of the word have been inverted.

In addition to inverting, there are numerous other applications of XOR. Although we will not detail them in this lesson, they include implementation of binary addition and subtraction in computers, comparators, parity checking, and combinational logic circuit minimization.

Bitwise Shift Operations

OzoBlockly has two bitwise shift operations—a logical shift left and a logical shift right. These shift operations work on the binary form of an integer. A ***logical shift left*** by one bit moves each bit to the left by one place. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded. A ***logical shift right*** by one bit moves each bit to the right by one. The vacant MSB is filled with zero and the least significant bit is discarded. A commonly used symbol for logical shift left is << (*two* greater than signs). For a logical shift right, the symbol >>> (*three* greater than signs) is quite common. OzoBlockly bitwise shift blocks provide a means of specifying how many bits to shift left or right.

Note that there is an ***arithmetic*** right shift, that preserves the sign bit when the shift is performed. 0 is shifted in for positive numbers, and 1 is shifted in for negative numbers. The arithmetic right shift, however, is not available in OzoBlockly. OzoBlockly contains only the ***logical*** bitwise shifts.

As shown in Figure 12, each bit shifted *left/right* effectively *multiplies/divides* a number by 2. Therefore, shifting left/right by *n* bits effectively multiplies/divides a number by 2^n.

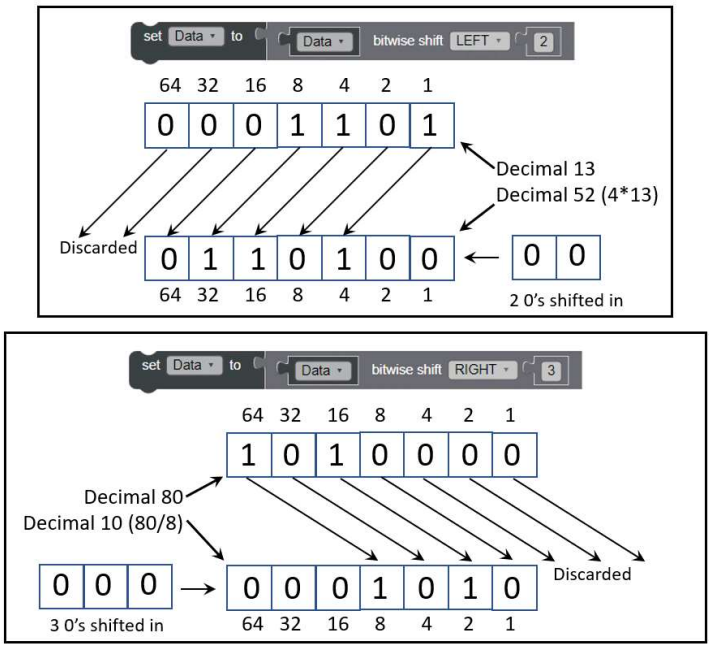

Figure 12

The top half of the figure shows a bitwise shift of 2 bits to the left. The OzoBlockly block to accomplish this task is also shown. The arrows show how the bits are shifted, with the two bits on the far left being discarded and two 0's shifted in on the right. We started with decimal 13, as can be determined by adding place values for bits that are 1. After the left shift, we see that two 0's have been shifted in on the right, giving a decimal value of 52. This is $2^2 = 4$ times the original value of 13.

The bottom half of the figure shows a bitwise shift of 3 bits to the right. The OzoBlockly block to accomplish this task is also shown. The arrows show how the bits are shifted, with the three bits on the far right being discarded and three 0's shifted in on the left. We started with decimal 80, as can be determined by adding place values for bits that are 1. After the left shift, we see that three 0's have been shifted in on the left, giving a decimal value of 10. This is $2^{-3} = 1/8$ of the original value of 80.

Exercise 5

a. Open the bitwise.ozocode program in OzoBlockly. Replace the "set Data to" block with the bitwise shift left block shown at the top of Figure 12. You will find the bitwise shift left block in the OzoBlockly Mode 5 group of Math blocks. Set the data to the word 0001101. (Remember that the LEFT bit is the MSB and the right bit is the LSB.) Note that the Mask is not used in this calculation, so the bits for this block can be left at their default values. Load the program into Evo and then run the program. The result should agree with that of the top half of Figure 12.

b. Now do the same thing for the bitwise right shift example in the bottom half of Figure 12.

c. Does left shifting by n bits always produce a number 2^n times greater than the original when using OzoBlockly and Evo? Try a logical bitwise left shift of 6 bits on the data 0001101.