

## Number Systems and Bit Operations

### Integer Binary Representation

You are familiar with decimal numbers, which use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Each digit in a place value system uses the place values of 10 to 0 = 1, 10 to 1 = 10, 10 to 2 = 100, 10 to 3 = 1000, and so on.

For example, we represent the value four-thousand thirty-five as

$$4035 = 4 * 1000 + 0 * 100 + 3 * 10 + 5 * 1$$

We represent numbers in the computer in analogous fashion, using just the digits 0 and 1. The place values are powers of 2 instead of powers of 10.

Following the same pattern we used above, this gives us binary place values of 2 to 0 = 1, 2 to 1 = 2, 2 to 2 = 4, 2 to 3 = 8, 2 to 4 = 16, 2 to 5 = 32, 2 to 6 = 64, 2 to 7 = 128, and so on.

So we can easily compute the value of a binary number:

$$\begin{aligned} 10111001 &= 1 * 128 + 0 * 64 + 1 * 32 + 1 * 16 + \\ &\quad 1 * 8 + 0 * 4 + 0 * 2 + 1 * 1 \\ &= 128 + 32 + 16 + 8 + 1 \\ &= 185 \end{aligned}$$

It is important to remember that every number we store in a variable in a computer program is, in fact, represented in its binary form. It is only when we express that variable within a program by outputting a value to the console that the computer

converts to the equivalent decimal value for display.

Since all numbers (and for that matter the other primitive types, including char and byte) are represented in binary, we can perform binary-related operations on any number.

## Shift Operations

A shift operation is the "shifting", or moving, of a binary number by a specified number of bit positions. We can either shift a binary number to the right, or to the left. Recall that an int is a 32 bit data element and a byte is an 8 bit data element. These binary operations operate on int (32 bit) data values. In the examples that follow, I will use 8 bit data values to make the examples simpler, but know that the operations are intended to operate on 32 bit int data.

Shift left:

The `<<` operator is used to shift left.

`op1 << op2` shifts bits of `op1` left by distance `op2` ; fills with zero bits on the right-hand side. Bits on the left hand side (from the high order 32nd bit of the data word) are shifted off the word and lost.

Say that I have binary value

`0 0 0 0 1 1 0 1` in the `int` variable `i`. This is decimal value 13.

The `int` expression `i << 3` shifts the bits of `i` by 3 bit positions, resulting in binary value `0 1 1 0 1 0 0 0`, which has decimal value 104. Try this in Dr. Java. Note that this shift operation is equivalent to multiplying the first operand by 2 to `op2` power.

Shift right:

There are two operators used to shift right, `>>` and `>>>`.

`op1 >> op2` shifts bits of `op1` right by distance `op2` ; fills with highest (sign) bit on the left-hand side. Bits on the right hand

side are shifted off the word and lost.

`op1 >>> op2` shifts bits of `op1` right by distance `op2` ; fills with zero bits on the left-hand side. Bits on the right hand side are shifted off the word and lost.

We will focus on the second right shift operator, as we will not be dealing with the sign bit in what we need in this class.

Say that I have the binary value

`0 0 1 0 1 1 0 1` in the `int` variable `j`. This is decimal value 45.

The `int` expression `j >>> 2` shifts the bits of `j` by 2 bit positions, resulting in binary value `0 0 0 0 1 0 1 1`, which has decimal value 11. Try this in Dr. Java. Note that this shift operation is equivalent to dividing the first operand by 2 to `op2` power.

## Bitwise Operators

We often need to perform bitwise operations between two integer operands. When we say we are performing bitwise operations between two integer operands `op1` and `op2`, this means that the operator is performed pairwise between each bit place-value position in the operands, so an operation is performed between the 1's bit of `op1` and `op2`, and between the 2's bit of `op1` and `op2`, and between the 4's bit of `op1` and `op2`, and so on.

Bitwise AND, operator `&`

Each result bit is 1 iff the corresponding bit of both operands is 1

Bitwise OR (inclusive or), operator `|`

Each result bit is 1 iff either or both of the corresponding bit of the operands is 1

Bitwise XOR (exclusive or), operator `^`

Each result bit is 1 iff either (but not both) of the corresponding bit of the operands is 1

Bitwise complement, operator `~`

Each result bit is the opposite of the corresponding bit of the

single operand.

Say that I have i and j with value 35 (100011) and 26 (011010)

$$\begin{array}{r} 100011 \\ \& \underline{011010} \\ 000010 \end{array}$$

Bitwise AND example

$$\begin{array}{r} 100011 \\ | \underline{011010} \\ 111011 \end{array}$$

Bitwise inclusive OR example

$$\begin{array}{r} 100011 \\ \wedge \underline{011010} \\ 111001 \end{array}$$

Bitwise exclusive XOR example

$$\begin{array}{r} \sim \underline{100011} \\ 011100 \end{array}$$

Bitwise complement