CS61B, Spring 2003 **Optional** **Amir Kamil**
UC Berkeley **4/5/03**

Topics: Integers

1 Integers

1.1 Number Representation

1.1.1 Base Conversion

A number just consists of a sequence of digits and a rule for converting those digits to a value. The value of decimal numbers can be computed from the sequence of digits as follows:

$$value((a_n, a_{n-1}, ..., a_0)) = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + ... + a_0 \cdot 10^0$$

This formula can be generalized to any base:

$$value((a_n, a_{n-1}, ..., a_0)_k) = a_n \cdot k^n + a_{n-1} \cdot k^{n-1} + ... + a_0 \cdot k^0$$, where k is the base

Converting from the value to a specific base is a little more complicated. Given a value v and a base k, let m be the largest integer such that $k^m < v$. Then the following is an algorithm to get the digits of v in base k:

```
digits(v, k, m, darray): // darray is the array of digits
  for i = m to 0 do:
    darray[m] = v / pow(k, m); // result is truncated if not an integer
    v = v - darray[m] * pow(k, m);
    m = m - 1;
  od;
end;
```

1.1.2 Machine Representation

All modern digital systems represent numbers in binary (base two). A digit in a binary number is called a *bit* and can have the value `0` or `1`. So if this is the case, how come you use decimal numbers when you write programs? The compiler automatically converts the decimal numbers to binary numbers.

If you want to use the binary representation of a number (e.g. with the bitwise operators below), you don't have to do anything to convert a number to binary; they already are in binary. So a statement such as

```
x = 123 & 8;
```

would operate on `123` and `8` in their binary representations.

Since machines have limited memory, they can't store numbers of arbitrary length. So they provide integers of different finite lengths. Table 1 lists the integer types in Java. Java allows conversion between these types. A conversion from type `A` to type `B` requires a cast if `A` can hold values not in the range of `B`. For example, converting a `short` to a `char` requires a cast because negative numbers that are in the range of `short` are not in the range of `char`. If this is not the case, such as a conversion from a `char` to an `int`, then no cast is required.

All integer types except `char` are signed. But if you recall our above representation of numbers in binary, you'll notice that we made no provision for negative numbers. Signed numbers are actually stored in a form called *two's complement*:

- Positive numbers are stored normally using binary digits as above, but have a zero in the most significant bit (the $(k-1)$th bit in a k bit number, counting from `0`). This limits the range of a positive number to $2^{k-1} - 1$ in a k bit type.

Type	# of Bits	Range	Signed?
`byte`	8	-128 to 127	**X**
`short`	16	-32768 to 32769	**X**
`char`	16	0 to 65535	
`int`	32	-2^{31} to $2^{31}-1$	**X**
`long`	64	-2^{63} to $2^{63}-1$	**X**

Table 1: The Java integer types.

- Arithmetic negation is done as follows:

  1. flip all bits in the number
  2. add one to the resulting number

  For example, `-93` is represented as follows in a `byte`:

  $$\begin{aligned} 93 &= 01011101_2 \\ -93 &= 10100010_2 + 1 \\ -93 &= 10100011_2 \end{aligned}$$

  Note that this conversion can be applied from negative to positive as well (try it on `-93`). This representation limits the range of negative numbers to -2^{k-1}.

What happens you try storing an out-of-range number in an integer type? This means you are trying to store a number that requires more bits than the type has and is called *overflow*. This can happen in a cast or when you are adding two numbers close to the boundary of a type's range. In Java, the most significant bits of the number are discarded. So if you try storing `349` in a `byte`, since the representation of `349` is the nine-bit 101011101_2, the extra bit is discarded, resulting in 01011101_2 or `93`. An interesting case is when a positive number wraps around to a negative result. Consider the representation of `163` in a `byte`. The binary representation is 010100011_2. The `0` is discarded, resulting in 10100011_2 or `-93`.

1.2 Bitwise Operations

Many languages provide facilities for operating on the bit representation of a number. Java provides the three binary operators (by *binary operators* I mean operators that take two arguments) `and`, `or`, and `xor`, the unary operator `not`, and the three shift operators. As noted above, you don't have to do anything special to "convert" numbers to binary, since they are internally represented as binary. The examples given in this section all use `byte`s as the type for simplicity, but the ideas can be extended to the other types.

1.2.1 Binary Operators

The Java binary operators are extensions of the corresponding logical operators. The logical operators take in two inputs, either of which can be `0` or `1`, and are defined as follows on the inputs A and B:

- The result of A `and` B is `1` only if $A = B = 1$.
- The result of A `or` B is `1` if at least one of A and B is `1`.
- The result of A `xor` B is `1` only if $A \neq B$.

These definitions can be concisely enumerated using *truth tables*. In a truth table, rows represent possible values of one input and columns possible values of the other. An entry in the table is the result of applying the operator to the corresponding two values. The truth tables for the three logical operators are given in table 2.

Java has operators that correspond the above logical operators: the `and` operator is `&`, the `or` operator is `|`, and the `xor` operator is `^`. These operators can be applied to entire numbers rather than to single bits.

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

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

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

Table 2: Truth tables for the binary operators `and`, `or`, and `xor`.

The result is a number in which each bit is the result of applying the logical operator on the corresponding bits in the two inputs. For example:

93 `&` 112
$= 01011101_2$ `&` 01110000_2
$= 01010000_2$
$= 80$

As you can see, the least significant bit in the result is the `and` of the least significant bits in the operands, the next significant bit in the result is the `and` of the next significant bits in the operands, and so on.

1.2.2 The Unary not

The unary `not` operator flips a bit from `0` to `1` or from `1` to `0`. The Java version of the `not` operator, `~`, flips all the bits in a number.

1.2.3 Shift Operators

The shift operators take in a number and a shift amount as inputs and shift the bits in the number right or left the amount specified. The syntax is *num* `<op>` *amt*, and the shifts are defined as follows:

- The left shift `<<`: shifts the bits in *num* to the left *amt* places, discarding bits that fall of the ends and filling in new bits with zeroes. For example:

  93 `<<` 4
  $= 01011101_2$ `<<` 4
  $= 11010000_2$

- The arithmetic right shift `>>`: shifts the bits in *num* to the right *amt* places, discarding bits that fall of the ends and filling in new bits with the sign bits of the original number (`0` for a positive number, `1` for a negative number). For example:

  -93 `>>` 4
  $= 10100011_2$ `>>` 4
  $= 11111010_2$

- The logical right shift `>>>`: shifts the bits in *num* to the right *amt* places, discarding bits that fall of the ends and filling in new bits with zeroes. For example:

  -93 `>>>` 4
  $= 10100011_2$ `>>>` 4
  $= 00001010_2$