Bitwise Operations

Shift left "<<". For integers, same as multiply by 2 for each bit shifted. Move all bits left by k positions, add k zeros to the right. Significant bits can be lost on the left size, still on same n bits. In the example below we represent an integer a=79 on $n = 32$ bits

$$\begin{aligned} 79 &= 0000\ 0000\ 0100\ 1111 \\ 79 << 1 &= 0000\ 0000\ 1001\ 1110 \\ &= 158 \\ 79 << 2 &= 0000\ 0001\ 0011\ 1100 \\ &= 316 \end{aligned}$$

Shift right ">>". For integers, same as division by 2 for each bit shifted. Move all bits right by k positions, add k zeros to the left. Non-significant bits will be lost on the right size, as result is still on same n bits.

$$\begin{aligned} 79 &= 0000\ 0000\ 0100\ 1111 \\ 79 >> 1 &= 0000\ 0000\ 0010\ 0111 \\ &= 39 \\ 79 >> 2 &= 0000\ 0000\ 0001\ 0011 \\ &= 19 \end{aligned}$$

Bitwise AND "&". Given an integer mask m on 32 bits, the operation $y = m\&x$ performs a bitwise AND: all 0 bits in m produce 0 bits in y, while all 1-bits in m simply leave the corresponding bit in x to pass to y. For

example x=78, $m = 5$ gives y as:

$$\begin{aligned} x = 78 &= 0000\ 0000\ 0100\ 1110 \\ m = 5 &= 0000\ 0000\ 0000\ 0101 \quad \& \\ \hline y &= 0000\ 0000\ 0000\ 0100 \\ &= 4 \end{aligned}$$

This is particularly useful when $m = 2^k$ (a power of two), in order to check if the k-bit in x is one or zero:
if $2^k \& x == 0$ then k-th bit in x is 0; otherwise the x k-th bit is one.

Bitwise OR "|". Given an integer mask m on 32 bits, the operation $y = m|x$ performs a bitwise OR: all 1-bits in m produce 1-bits in y, while all 0-bits in m simply leave the corresponding bit in x to pass to y. For example x=78, $m = 21$ gives y as:

$$\begin{aligned} x = 78 &= 0000\ 0000\ 0100\ 1110 \\ m = 21 &= 0000\ 0000\ 0001\ 0101 \quad | \\ \hline y &= 0000\ 0000\ 0101\ 1111 \\ &= 95 \end{aligned}$$

This is particularly useful when $m = 2^k$ (a power of two), in order to make the k-bit in x one:
if $y = 2^k|x$ makes the k-th bit in y one, but leaves all other bits as in x.

Exercise. Play with the attached C code "bitwise.cpp". You dont have to look into declarations of variables, but rather change the integer values and see what happens. Being C++ code, you will have to compile and run it; you can do so with the attached Makefile, on a UNIX-based system, by simple typing in the terminal window
`make FILE=bitwise`
which will both compile and run the code. Every edit of the source code have to be saved and followed by the same `make` command.

```
19:13>> make FILE=bitwise
g++ -Wall -pedantic -o bitwise bitwise.cpp
./bitwise
size_of_int=4
a11=150000 a12=150000 OVERFLOW (32 bits)?
a11*a12=1025163520

a14=79
a15=a14>>1=39
a16=a14<<1=316


a1=79
11110010 00000000 00000000 00000000
a1=39
11100100 00000000 00000000 00000000
a1=316
00111100 10000000 00000000 00000000


a2=316

d1=10234.9
00011101 01111000 10100001 11010111 10001110 10111111 11000011 00000010
result=00011101011110001010000111010111100011101011111111000011000000010
d2=10234.9
```