Division of Computing Science and Mathematics, University of Stirling

CSCU9V4 Systems
Tutorial 1b: Boolean algebra

Simplifying Boolean Expressions

This worksheet introduces some algebraic methods which allow the reduction of boolean expressions to simpler forms. The basic boolean laws seen in the lectures apply in this tutorial (such as $(\overline{A}+A)=1$) and you will also apply the distributive law frequently (e.g. A•(B + C) = A•B + A•C, and *vice versa*). In subsequent tutorials you will also use De Morgan's Theorem to help simplify expressions, but the methods exercised here are essential skills.

Note: these techniques can also be used when designing the logic of any computer program which uses statements such as 'IF-THEN', 'WHILE-DO', 'CASE-OF', 'DO-UNTIL' which require the use of boolean operators and conditional statements.

Part 1 – Basic Boolean Laws, a reminder!

Task: As a reminder of the technique, use a truth table to show the result of the following boolean laws:

(e.g. (i) $(A+\overline{A})$ draw a truth table:

A	$\overline{A}$	**+**
0	1	1
1	0	1

Therefore the result of **OR**'ing a boolean variable with its inverted form is always **1**, or **true**. Note that the variable 'A' can be any boolean variable or compound expression, e.g. $(B\bullet C)+\overline{(B\bullet C)}=1$ also works. (Remember this is NOT the same as $(B\bullet C)+(\overline{B}\bullet\overline{C})$ which is, in fact, a NXOR gate!).

It's useful to remember this method of using truth tables to back up your memory of the boolean laws; - in exams, for instance!

(ii) $A\bullet\overline{A}$ = ??	(iii) A + 1 = ??	(iv) A • 1 = ??
(v) A • A = ??	(vi) A + 0 = ??	(vii) A • 0 = ??

The other laws are in your lecture notes. You should also know three basic algebraic laws:

Associative Laws	**Distributive Law**	**Commutative Laws**
(A + B) + C = A + (B + C)	A• (B + C) = A • B + A • C	A + B = B + A
(A•B)• C = A •(B•C)	X•Y + X•Z = X• (Y + Z)	A • B = B • A

Part 2 – Applying the Basic Boolean Laws

The point of having all those boolean laws is that we can look at a complicated boolean expression and apply the laws to reduce it to a simpler form and thus make it cheaper and easier to build as hardware, or faster to execute if it's part of software. We'll start with some simple examples…

- In the expression $X = (A + B + \overline{B})$ we can see that the B + $\overline{B}$ part can be reduced to '1' by the first law seen in Part 1. This leaves us with $X = (A + 1) = (1)$ as law (iii) also applies to the (A + 1) part. So we end up with just X = 1 (that is, X is always true).

- $X = (A \bullet \overline{A} + B \bullet C)$ can be reduced by applying law (ii) from above to the $A \bullet \overline{A}$ part, giving us $X = (0 + B \bullet C)$ which reduces to $X = (B \bullet C)$ due to law (vi).

- $X = (A \bullet A + B \bullet \overline{B} \bullet C)$ can be reduced by seeing that A • A = A (using law (v)) and $B \bullet \overline{B} = 0$ leaving us with $X = (A + 0 \bullet C)$.
  As 0 • C = 0 we are left with X = (A + 0) which reduces to simply X = A (using law (vi)).

Task: Try simplifying the following expressions using the basic laws from your notes and Part 1 of this tutorial. Do this on a separate sheet of paper, so you have room.

(I) $X = \overline{A} + A + B$

(ii) $Q = A \bullet (B + \overline{A} + 1)$

(iii) $X = (A \bullet \overline{A} + B + \overline{B})$

(iv) $Z = B \bullet C \bullet (A + \overline{A})$

(v) $X = (A \bullet \overline{A} + B \bullet \overline{B})$

(vi) $Y = \overline{(A \bullet B)} + (A \bullet B) + (\overline{A} \bullet \overline{B})$

Do you recognise this part as a **NXOR**? If so, well done - BUT, it's a red-herring!

(vii) $X = (A \bullet A + B \bullet B \bullet C)$

(viii) $X = (A \bullet A + R \bullet A \bullet C)$

(ix) $M = G.E.O.R.G.E + \overline{G.E.O.R.G.E} + B.O.O.\overline{L.E} + B.O.O.L.E$

…OR…

Part 3 - Making Boolean Expressions Simpler

We can now turn our attention to the **reduction** (or, **simplification**) of boolean expressions. At first glance the reduction of boolean expressions appears to be an extremely tricky task only achievable by very clever people! In fact, reduction can be straightforward if you remember a basic two step process:

Step 1: manipulate the expression.
This involves either:

removing brackets e.g. (A + B)•B = A•B + B•B

or

adding brackets e.g. B + B•C = B•(1 + C)

using the **distributive** law. The aim of this step is *change* the form of the expression so that you can apply step 2.

Step 2: reduce the expression where possible.
This involves applying one or more of the boolean algebra laws to your manipulated expression,

e.g. B•B = B, (1 + C) = 1, and so on.

The aim of this step is *remove* or simplify terms in the expression.

Then rinse and repeat steps 1 and 2 until the expression cannot be reduced any further. With practice this process becomes easier.

Example: Show that **(A + B) • (A + C)** is equivalent to **A + B•C**

stage 1:	**manipulate** *by expanding the brackets of the original expression:*	
	(A + B)•(A + C)	= A•A + A•C + A•B + B•C
stage 2:	**reduce** *using A•A = A*	
	A•A + A•C + A•B + B•C	= A + A•C + A•B + B•C
stage 3:	**manipulate** *using A + A•C = A• (1 + C)*	
	A + A•C + A•B + B•C	= A•(1 + C) + A•B + B•C
stage 4:	**reduce** *using 1 + C = 1*	
	A•(1 + C) + A•B + B•C	= A•1 + A•B + B•C
stage 5:	**reduce** *using A•1 = A*	
	A•1 + A•B + B•C	= A + A•B + B•C
stage 6:	**manipulate** *using A + A•B = A• (1 + B)*	
	A + A•B + B	= A• (1+ B) + B•C
stage 7:	**reduce** *using 1 + B = 1*	
	A•(1+ B) + B•C	= A•1 + B•C
stage 8:	**reduce** *using A•1 = A*	
	A•1 + B•C	= A + B•C
therefore	(A + B)•(A + C) = ***A + B•C***	***QED***

A more complex example, $X = A \bullet B \bullet C + A \bullet B \bullet \overline{C} + B \bullet \overline{C}$ can be simplified in more than one way:

Spot that $A \bullet B$ is common to the first two 'product' terms and can therefore be taken 'outside the bracket' using the distributive law:

$$X = A \bullet B \bullet (C + \overline{C}) + B \bullet \overline{C}$$

Then reduce the $C + \overline{C}$ to 1 as before:

$$X = A \bullet B \bullet (1) + B \bullet \overline{C}$$

So:

$$X = A \bullet B + B \bullet \overline{C}$$

Again notice that B is common to both terms, so take 'outside the bracket':

$$X = B \bullet (A + \overline{C})$$

This is now as simple as we can get it.

Spot that $B \bullet \overline{C}$ is common to the last two 'product' terms and can therefore be taken 'outside the bracket' using the distributive law:

$$X = A \bullet B \bullet C + B \bullet \overline{C} \bullet (A + 1)$$

Notice how the '1' has appeared.
Now reduce the $A + 1$ to 1:

$$X = A \bullet B \bullet C + B \bullet \overline{C} \bullet (1)$$

So:

$$X = A \bullet B \bullet C + B \bullet \overline{C}$$

Notice that B is common to both terms, so take it 'outside the bracket', giving:

$$X = B \bullet (A \bullet C + \overline{C})$$

This is not as simple as the attempt made to the left, so what's gone wrong here?

Nothing! First of all there is no easy way to know which 'route' to take when simplifying expressions – at this stage all you can do is be methodical and try every possibility! With realistic designs this could quickly become tiresome, so this job is usually left to a computer and specialised software.

Secondly, there are other methods that can shortcut this process (e.g. **Karnaugh Maps**), and thirdly – we haven't finished simplifying that right-hand expression! We can apply the **negative absorption** law to the $(A \bullet C + \overline{C})$ (see lecture notes) to finish up with:

$$X = B \bullet (A \bullet \cancel{C} + \overline{C}) \qquad X = B \bullet (A + \overline{C})$$

which finally agrees with the left hand simplification. *Phew*!

Task: Using algebraic manipulation show that:

(i) $A + A \bullet B = A$

(ii) $A \bullet (A + B) = A$

(iii) $A \bullet (\overline{A} + B) = A \bullet B$

(iv) $\overline{A} \bullet (1 + A + B) + A \bullet B = \overline{A} + B$

(v) $(A + B) \bullet (\overline{A} + \overline{B}) = A \bullet \overline{B} + \overline{A} \bullet B$

***Question**: which logic gate does this equation represent?*

Task: identify the **XOR** or **NXOR** gates hidden in the following expressions (you may need to inspect each expression carefully and use some of the reduction methods used earlier) and then re-write the expression using the appropriate gate symbol ($\oplus$ or $\overline{\oplus}$) to replace those parts you've found.

e.g.(i) $Q = P.X.\overline{Y} + P.\overline{X}.Y$

$Q = P.(X.\overline{Y} + \overline{X}.Y)$

$Q = P.(X \oplus Y)$

(ii) $X = (A + B).(\overline{A} + \overline{B})$

(iii) $Q = P.\overline{X}.\overline{Y} + P.X.Y$

(iv) $X = (\overline{A} + B).(A + \overline{B})$

(v) $Z = P.\overline{Q}.\overline{R} + \overline{P}.\overline{Q}.R + P.Q.R$

(vi) $Y = (\overline{A}.B + A.\overline{B}).\overline{C} + (\overline{A}.\overline{B} + A.B).C$

(vii) $T = Q.\overline{R} + \overline{P}.\overline{Q}.R + P.\overline{Q}.R$

(viii) $Z = A.B.C + \overline{A}.\overline{B} + A.B.\overline{C} + \overline{A}.\overline{B}.C + A.\overline{B}.C$

George Boole, 1815 - 1864

Boole *approached logic in a new way reducing it to a simple algebra, incorporating logic into mathematics. He also worked on differential equations, the calculus of finite differences and general methods in probability.*

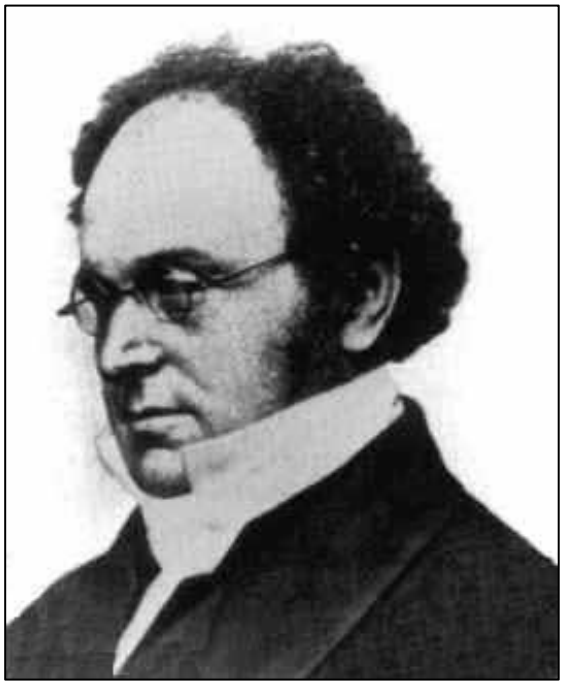
Born: 2 Nov 1815 in Lincoln, Lincolnshire, England
Died: 8 Dec 1864 in Ballintemple, County Cork, Ireland

http://www-groups.dcs.st-and.ac.uk/~history/Biographies/Boole.html

Augustus De Morgan, 1806 - 1871

http://www-history.mcs.st-andrews.ac.uk/history/Biographies/De_Morgan.html

De Morgan *was always interested in odd numerical facts and writing in 1864 he noted that he had the distinction of being x years old in the year x^2 (He was 43 in 1849). Anyone born in 1980 can claim the same distinction (when aged 45).*



Simplifying Boolean Expressions: De Morgan's Theorem

De Morgan's theorem allows us to swap an **OR** gate in a boolean expression for an **AND** gate, and vice versa. There are three simple steps that need to be applied to do this:

1. **invert** both sides (or 'inputs') to the gate (this may include more than one symbol on each side) - *this is typically where mistakes are made (see below)*,
2. **invert** the whole of the expression involved (i.e. a bar across both inputs and the operator),
3. **change** the gate (or operator) symbol from + to • or from • to +, as required.

These can be applied in any order, or simultaneously – the result will be the same.

Note: When inverting the two inputs to a gate it may help to visualise (or draw) the boolean expression as a circuit to correctly identify what the inputs are.
For example, what are the two inputs to the **OR** gate in this expression?:

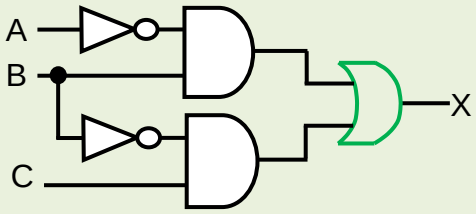
$$X = \overline{A} \bullet B + C \bullet \overline{B}$$

The precedence rules should tell you that it's ALL of the left hand side and ALL of the right hand side, thus (bracketed, in red):

$$X = (\overline{A} \bullet B) + (C \bullet \overline{B})$$

However, the mistake is often made of thinking that '**B**' and '**C**' alone are the two inputs!

As a circuit it would look like this:



This clearly shows the two inputs to the **OR** gate are the outputs from the two **AND**s, and *not* simply '**B**' and '**C**'.

So, inverting the two inputs to the **OR** gate would result in:

$$X = \overline{\overline{A} \bullet B} + \overline{C \bullet \overline{B}}$$

Using brackets around **AND** gates helps to avoid such problems.

Quick exercise: apply the remaining two steps of De Morgan's theorem to the expression above to convert the **OR** gate to an **AND** gate…

$$X = \overline{\overline{A} \bullet B} + \overline{C \bullet \overline{B}} \qquad X =$$

Where brackets already appear in an expression they dictate the inputs to a particular gate, e.g.:

$$X= \overline{A} \bullet B + C \bullet \overline{B} + B + C \bullet A$$

The two inputs to the middle **OR** gate are '$C \bullet \overline{B}$' and 'B' - but if brackets were in place, thus:

$$X= \overline{A} \bullet B + C \bullet \overline{B} + (B + C \bullet A)$$

the two inputs would now be '$C \bullet \overline{B}$' and '$(B+C \bullet A)$' and inverting them would result in:

$$X= \overline{A} \bullet B + \overline{C \bullet \overline{B}} + \overline{(B + C \bullet A)}$$

Applying the remaining two steps to this **OR** gate would give the final result (in green):

$$\mathbf{X= \overline{A} \bullet B + \overline{\overline{C \bullet \overline{B}} \bullet \overline{(B + C \bullet A)}}}$$

De Morgan's Theorem is useful to us for two main reasons:

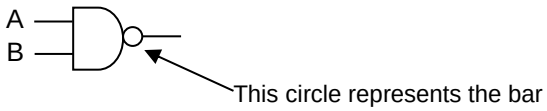
(1) we can convert any arbitrary expression to **NAND-only** form, and

(2) to show equivalences and/or help simplify expressions.

Using De Morgan's to convert expressions to NAND-only form

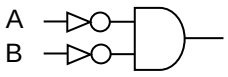
Remember that written algebraically a **NAND** gate looks like an **AND** gate but with a bar over the expression, e.g.:

$\overline{A \bullet B}$ the circuit looks like:

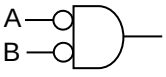


Remember that the following isn't a NAND gate:

$\overline{A} \bullet \overline{B}$ which looks like:



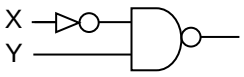
or



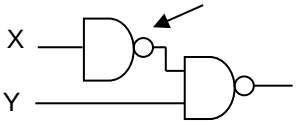
Can you remember how to build a NOT gate from a single NAND-gate?

Another example of a NAND gate:

$\overline{\overline{X} \bullet Y}$

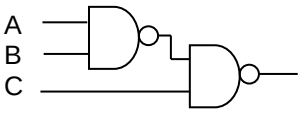


or, just using NAND gates:



Consider the circuit for the following NAND-only expression; notice that there are two NAND gates involved:

$\overline{\overline{(A \bullet B)} \bullet C}$



(It may be useful to refer to Logic Gates practical where you built circuits from NAND gates only.)

Task: For practice draw the circuits for the following expressions using **NAND-gates only**:

(i) $\overline{A}$

(ii) $\overline{\overline{X} \bullet \overline{Y}}$

(iii) $\overline{\overline{(A \bullet \overline{B})} \bullet \overline{C}}$

(iv) $A \bullet C$

hint: *think double inverters!*

(v) $\overline{\overline{(A \bullet C)} \bullet \overline{(D \bullet B)}}$

(vi) $\overline{P} \bullet Q \bullet \overline{R}$

Task: Now practice De Morgan's Theorem by changing the **AND** gate to an **OR** (or **NOR**) gate, and the **OR** gate to an **AND** or **NAND** gate in the following boolean expressions:

(i) $S = A \bullet B$

(ii) $Z = \overline{Q \bullet R}$

(iii) $A = M + \overline{T}$

Task: Now convert the following expressions to **NAND-only** form by applying De Morgan's Theorem to the **OR** gates (there is little point changing the AND gates to OR gates in this case as we're trying to get NANDs!).

(i) $S = A + \overline{B}$

(ii) $M = \overline{Q \bullet P} + R$

(iii) $Q = P \bullet Z + \overline{T}$

(iv) $D = \overline{A + C}$

(v) $\overline{G} = (\overline{A \bullet F}) + P$

(vi) $H = B + \overline{B} \bullet C$ *(can you simplify first?)*

(vii) $Z = A \bullet B + C + B \bullet \overline{C} \bullet D$

(viii) $W = P + \overline{Q \bullet R} + B \bullet D$

(ix) $F = A \bullet \overline{B} \bullet \overline{D} + \overline{A} \bullet B \bullet D$

(x) $M = Q \bullet \overline{R} + \overline{Q} \bullet R$ (*same as* $M = (Q \oplus R)$

Notice that in (vi) and (vii) the ***negative absorption rule*** *can be used to simplify the expression first.*

Using De Morgan's to show equivalences

An example of this process would be to prove the **negative absorption law**, e.g.:

$A + \overline{A} \bullet C$ we know this can be written as: $A + C$

To prove this we can apply De Morgan's theorem to the OR and the AND gates, simplify the expression and re-convert using De Morgan again. Here, each gate will be converted separately to illustrate the process.

First, convert the **OR** to an **AND** gate, using the three De Morgan steps:

$\overline{A} + \overline{(\overline{A} \bullet C)}$ step 1,identify & invert both inputs to the **OR** gate (note that <u>all</u> of the right-hand term is inverted, note use brackets around the AND to make this clear)

$\overline{\overline{A} + (\overline{A} \bullet C)}$ step 2, invert the whole gate (note that *none* of these invertors cancel!)

$\overline{A} \bullet (\overline{A} \bullet C)$ step 3, change the symbol from OR to AND.

Secondly, convert the **AND** gate in the bracketed part to an **OR** gate:

$\overline{A} \bullet (\overline{A} \bullet \overline{C})$ step 1 - invert the inputs

$\overline{A} \bullet (\overline{A} \bullet \overline{C})$ step 2 - invert the gate

$\overline{A} \bullet (\overline{A} + \overline{C})$ step 3 - change the operator from AND to OR

$\overline{A} \bullet (A + \overline{C})$ now cancel the double negatives (only those of the same 'length', or *scope*)

$\overline{\overline{A} \bullet A + \overline{A} \bullet \overline{C}}$ distribute $\overline{A}$ across the bracketed part (get rid of the brackets)

$\overline{\overline{A} \bullet \overline{C}}$ reduce using the law $\overline{A} \bullet A = 0$

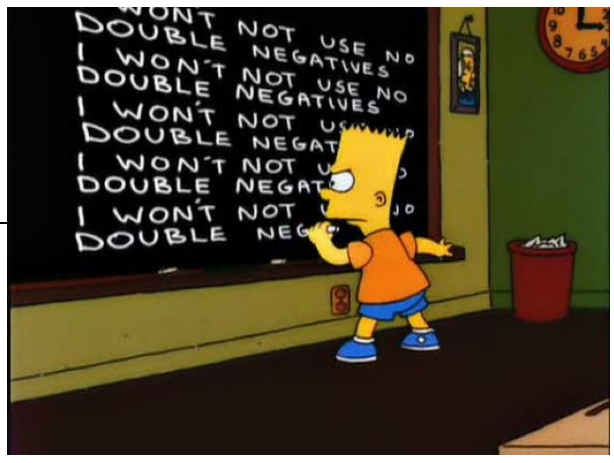
$\overline{\overline{A} \bullet \overline{C}}$ Now use De Morgan to convert the remaining **AND** gate, step 1

$\overline{\overline{A} \bullet \overline{C}}$ step 2

$\overline{\overline{A} + \overline{C}}$ step 3

$A + C$ now cancel the double negatives, leaving the desired form! ***QED***.

(**QED** = *quod erat demonstrantum* (latin), or more prosaically, "*job done!*")



Tasks: Using De Morgan's theorem, show the following:

(1) $\overline{A} + A \bullet C$ can be written as $\overline{A} + C$

(2) $X \bullet \overline{Y} + \overline{X} \bullet Y$ is equivalent to $\overline{\overline{X} \bullet \overline{Y} + X \bullet Y}$

(…in other words, prove that $X \oplus Y$ *is the same as* $\overline{\overline{X \oplus Y}}$ *! You should recognise these…)*

(3) $\overline{P} \bullet Q + P \bullet \overline{Q} + \overline{P \bullet Q}$ reduces to $\overline{P \bullet Q}$

(start by using De M on the $\overline{P \bullet Q}$ *then simplify…)*

Notice that De Morgan's Theorem can be applied to more than one gate at the same time, although it can get confusing if you try this with too complex a mix of AND's and OR's! Using brackets around the AND'ed terms helps prevent errors. Consider these examples:

$F = A + B + C = \overline{\overline{A} \bullet \overline{B} \bullet \overline{C}}$

$Q = A + (B \bullet C) = \overline{\overline{A} \bullet (\overline{B} + \overline{C})}$

$D = M \bullet \overline{O} \bullet R \bullet G \bullet \overline{A} \bullet \overline{N} = \overline{\overline{M} + O + \overline{R} + \overline{G} + A + N}$

$Z = A + \overline{(B \bullet C)} + (D \bullet E) = \overline{\overline{A} \bullet (B \bullet C) \bullet \overline{(D \bullet E)}}$

Example application of Boolean Algebra to a programming problem.

Imagine the need in a C++ program to check if three numbers are all even. This can be done by isolating and then checking the least significant bit (LSB, the right-hand most bit, if you wish) of the numbers. Here, this LSB has been isolated and placed in three variables, called **num1**, **num2** and **num3**. If they're all zeroes then the three numbers must all be even. If any one of them is a 1 then the test fails.
Writing this in C++, one might instinctively produce this **IF** statement to do the trick ('&&' is C++'s logical **AND** operator):

```
if ( num1 == 0 && num2 == 0 && num3 == 0)
{    allEven = TRUE;
}
```

This reads as "if num1 is zero AND num2 is zero AND num3 is zero then set boolean `allEven` to TRUE".

This construct reads easily and works, and may well be fast enough for the purpose intended.

Suppose now that you're looking for speedups in your code. It might surprise you to read that if you take the following steps this snippet of code can be made to run almost twice as fast!

Consider the three boolean comparisons above as boolean variables:

`num1 == 0`	write as	X (can be either TRUE or FALSE)
`num2 == 0`		Y
`num3 == 0`		Z

So we have: `allEven` = $X \bullet Y \bullet Z$

Read as: "`allEven` is TRUE IF X is TRUE and Y is TRUE and Z is TRUE"

Applying **de Morgan's** theorem to this gives:

`allEven` = $\overline{\overline{X} + \overline{Y} + \overline{Z}}$ = NOT($\overline{X} + \overline{Y} + \overline{Z}$)

Read as: "`allEven` is TRUE IF NOT (X is FALSE or Y is FALSE or Z is FALSE))

X is FALSE would occur when `num1` = 1, meaning `num1` is an odd (that is, *not even*) number. Same goes for the other two numbers.

Now, booleans are represented in a computer as simply 0 or 1. So, we can interpret the condition that X is FALSE (that is, when `num1` = 1) as being `num1` = TRUE. *Geddit*?

So X is FALSE is equivalent to saying "if `num1`". So we can express our revised IF condition as:

`allEven` = NOT (`num1` + `num2` + `num3`) where '+' means OR, as usual

- using NOT and brackets to replace the long inversion bar.

Writing this in C++ using the bitwise boolean operator for OR ('|') and '!' meaning NOT, we get the following less intuitive revised version:

```
if (!(num1 | num2 | num3 ))
{    allEven = TRUE;
}
```

(**Note**: bitwise operators such as '**&**' will operate on each corresponding individual bit of the two operands and return the result which could be any bit pattern, whereas logical operators such as '**&&**' return TRUE or FALSE only (1 or 0, in other words).

Using this revised code shows a speedup of *nearly 100%* on my PC…

The reasons for this are too complicated to explain here, but if you care to disassemble the C++ code into x86 and study it, the reasons *can* become clear! For those who are interested, it revolves around reducing the number of branch (or jump) instructions generated by the C++ compiler, and the effect of hardware *branch prediction*.

Note *Simply altering the initial logical operator '**&&**' in the original code to use the bitwise operator '**&**' gives an appreciable speedup too… this works because, again, we're dealing with purely 0's (FALSEs) and 1's (TRUEs).*
