Topic 4.2 (AHL) – Boolean Logic

This information supports the following sections of the IB Computer Science Syllabus, which came into force for first examinations in 2000:

Additional Higher Level Material: Topic 4: Computer Mathematics and Logic.
Subtopic 4.2: Boolean Logic.

4.2.1 Define the Boolean operators: **AND**, **OR**, **NOT**, **NAND**, **NOR** and **XOR**, by drawing the appropriate truth table.

Most Boolean operators are binary operators (i.e. they demand two operands to give a single result). The exception is **NOT**, which is unary.

All Boolean operands (inputs) and results (outputs) can be in one of two possible states or conditions:

0 – False, also described (in digital electronics) as low, logic 0, off, no charge present.
1 – True, also described (in digital electronics) as high, logic 1, on, charge present.

It's a fundamental rule that a Boolean operand (or result) can never be in both conditions at the same time!

A truth table is a convenient way of listing all the possible input combinations for a Boolean expression, along with the corresponding outputs for each one. The inputs are usually labelled A, B, C, D…. etc. There is somewhat more debate about the labelling of the output. Some people (and publications) use Y, others Z. Personally I have always used Q, but then I have primarily approached logic in the past from the point of view of digital electronics, where Q is pretty much the norm. The IB appear (from the evidence of the last examination papers) to favour Z, so in deference to them and to consistency, I will stick to Z.

The basic truth tables for the operators or functions listed above are shown here, along with their symbols and a commentary. Notice that although the **symbols** used may be familiar to you, they have a **different meaning assigned to them** when they are used in Boolean expressions:

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

AND

AND

"The output is true when A is true and B is true"

Z = A . B also written as **Z = AB**

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NAND

NAND

"The output is true when A is false and B is false"
"false" is logically equivalent to "not true" and is indicated by placing an "overbar" above the operand, signifying "negation".

$\mathbf{Z = \overline{A} . \overline{B}}$ also written as $\mathbf{Z = \overline{A .B}}$ or $\mathbf{Z = \overline{AB}}$

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

OR

OR
"The output is true when A is true or B is true, or both are true"

$\mathbf{Z = A + B}$

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

NOR

NOR
"The output is true when A is false or B is false, or both are false"

$\mathbf{Z = \overline{A} + \overline{B}}$ also written as $\mathbf{Z = \overline{A + B}}$

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

XOR

XOR (e**X**clusive **OR**)
"The output is true when A is true or B is true."
(notice that the "or both" condition does not apply)

$\mathbf{Z = A \oplus B}$

A	Z
0	1
1	0

NOT

NOT (a unary operator)
"The output is true when the input is false"

$\mathbf{Z = \overline{A}}$

4.2.2 Construct Boolean expressions using the operators in 4.2.1:

The symbols detailed above can be used as a compact and efficient way of representing Boolean expressions. Remember, a Boolean expression is one that can either have the value of true or false.

When faced with Boolean expressions like this one: $Z = (A\overline{B}) + (\overline{A}B)$, its helpful to think of it as **illustrating the input conditions needed for the output to be true**. So in words it would run something like: Z will be 1 when A is 1 and B is 0 *or* when A is 0 and B is 1.

If you compare this to the truth tables above, you should easily be able to spot that this expression is another way of representing the **XOR** function. It's useful to be aware of this alternative. Although the IB uses the $\oplus$ symbol for XOR, some of the methods you will be using to manipulate and simplify Boolean expressions don't.

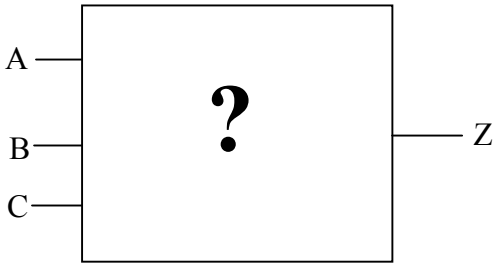
For your IB examinations, you will probably be faced with expressions that use three operands or inputs. There is no point in printing out a complete listing of 3 input basic truth tables here. They follow exactly the same logic as before. Thus the **AND** function for 3 inputs becomes: "The output will be true when A **AND** B **AND** C are true."

4.2.3 / 4.2.4 Calculate the values of a Boolean expression using truth tables and convert these Boolean Expressions into simpler forms:

The best way to explain and illustrate this is by working through an example:

Example:

A 3 input function (of unknown composition) is shown below, along with the truth table that has been derived from it. If we knew what the function contained (e.g. If logic **gates** were shown), we could construct a *trace table* to arrive at the same starting point. Trace tables will be dealt with separately.



A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Now, remember that a Boolean expression illustrates the input conditions needed for the output to be true. By convention, if an input is shown in its true or positive state (e.g. A), then this represents a 1. In its negative or negated state (e.g. $\overline{A}$), a 0 is recorded.

Look for the lines where the output is true (1). Line 2 is the first instance. You could write this out in words as: Z is true when A is false **AND** B is false **AND** C is true.

In Boolean notation: $Z = \overline{A}\ \overline{B}\ C$

Of course, this is not the only line where Z is true, so work through the table, noting down all the instances. Logically (duh) the expression for each line should be separated with an **OR** (+) symbol.

The resulting complete expression should look like this:

$Z = \overline{A}\ \overline{B}\ C + \overline{A}\ B\ C + A\ \overline{B}\ \overline{C} + A\ \overline{B}\ C$

All you need to do now is to simplify this expression, canceling out as many of the expressions or "minterms" as possible.

There are two ways of simplifying a Boolean expression (You could also stare at it for an indeterminate length of time until inspiration strikes [not recommended]). Failing that you could use:

a) **The laws and theorems of Boolean algebra.**

-or-

b) **Karnaugh Maps.**

a) **Boolean Algebra.**

There are ten basic Boolean laws/theorems that can be applied to simplify an expression. Each theorem is described by two parts that are duals of each other. The principles of duality state that you can interchange the **or** and **and** operators of the expression as well as interchanging the 0 and 1 elements. The form of the variables does not change.

Thus:

T (theorem) **1**: **The Commutative Law**, states that…

a) $A + B = B + A$
b) $A\ B = B\ A$

… are duals of each other.

The other nine are as follows:

T2: The Associative Law
a) $(A + B) + C = A + (B + C)$
b) $(A\ B)\ C = A\ (B\ C)$

T3: The Distributive Law
a) $A\ (B + C) = A\ B + A\ C$
b) $A + (B\ C) = (A + B)\ (A + C)$

T4: The Identity Law
(a) $A + A = A$
(b) $A\ A = A$

T5: The Negation Law
a) $(\overline{A}) = A$
b) $(\overline{A}) = A$

T6: The Redundancy Law
(a) $A + A\ B = A$
(b) $A\ (A + B) = A$

T7:
(a) $0 + A = A$
(b) $1\ A = A$
(c) $1 + A = 1$
(d) $0\ A = 0$

T8:
a) $\overline{A} + A = 1$
b) $\overline{A}\ A = 0$

T9:

a) $A + \overline{A}\,B = A + B$

b) $\overline{A}\,A = 0$

T10: De Morgan's Theorem

a) $\overline{(A + B)} = \overline{A}\,\overline{B}$

b) $\overline{(A\,B)} = \overline{A} + \overline{B}$

Phew!!! You can then apply these laws and/or theorems to the expression in order to simplify it.

You can - and good luck to you. I don't. I have nothing against algebraic simplification, except that I can't see the point, at this level at least, in remembering ten theorems and then trying to apply them. There again, I will admit that I have never really been that comfortable with algebra in general. If you fall into the same camp, then congratulations, you could be a fan of the second method of simplification:

b) **Karnaugh Maps**

Karnaugh maps are a simple, graphical and elegant way of simplifying any Boolean expression. Well, almost any. They work fine for expressions of up to four inputs, they are possible for up to six inputs. Beyond that they would become so esoteric that you might as well just admit defeat and learn to do it algebraically. Fortunately for us the IB limits itself to a maximum of three inputs.

This is the map for three inputs, in its two popular forms of representation:

Either……

	$\overline{A}$		A	
$\overline{C}$				
C				
	$\overline{B}$	B		$\overline{B}$

Or…

C \ AB	00	01	11	10
0				
1				

Personally I find the first representation slightly easier to fill and use, but the second one is slightly easier to draw and remember. Either way, they are exactly the same, the difference being aesthetic rather than functional! Notice that each adjacent cell varies by only one bit. This sort of coding is known as grey code. So how *does* it work?

How to Use a Karnaugh Map:

Remember the expression we derived from the truth table on page 3 of this document. Here it is again:

$$\mathbf{Z = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C}$$

1. Take each minterm and place a 1 in the cell that matches the input conditions to make the output true. In other words, each cell corresponds to one row in the original truth table. For example, $\overline{A}\,\overline{B}\,C$ gets a 1 in the cell shown (remembering that negated inputs (overbar) correspond to a 0).

C \ AB	00	01	11	10
0				
1	1			

Here is the map filled with the rest of the four minterms in the example.

C \ AB	00	01	11	10
0				1
1	1	1		1

Now group the 1's together, following these rules:

- Groups must contain only 1's.
- They can be horizontal or vertical, but never diagonal.
- The number of cells in a group must be a power of 2.
- All of the 1's in the map must be in at least one group.
- The overlapping of groups is allowed.
- Groups should be as large as possible, and there should be as few of them as possible.

C \ AB	00	01	11	10
0				1
1	1	1		1

You then apply the original, fundamental rule of all Boolean expressions: An operand cannot be in both conditions (0 and 1) at the same time:

C \ AB	00	01	11	10
0				1
	1	1		1

Therefore in this group, B is eliminated, as it breaks the rule, so you are left with $\overline{A}\,C$…

…and in this group, C is eliminated, so you are left with $A\,\overline{B}$

Therefore the expression is true when $\overline{A}\,C$ is true **or** $A\,\overline{B}$ is true. Put another way….

$$\mathbf{Z = \overline{A}\,C + A\,\overline{B}}$$

Final Words about Karnaugh Maps:

There is one other thing that needs to be said about grouping. In a Karnaugh Map, the cells are considered to be "wrapped around", so the cells at the left hand end of the map are treated as being adjacent (next) to the cells at the right hand end.

Therefore, in this map:

AB / C	00	01	11	10
0	1			1
1	1			1

The grouping would be:

AB / C	00	01	11	10
0	1			1
1	1			1

The only surviving operand once you have applied the rule of simplification is $\overline{B}$. This square arrangement is also called a "cube".

Incidentally, the top and bottom rows also wrap. In a three input map this is not really helpful, as the top and bottom rows are already adjacent! However, if you go on to use maps with four inputs or more, this rule becomes very important.

From the information so far, you should be able to see how a Karnaugh map can handle the functions **AND** (A B), **OR** (A + B), **NOR** ($\overline{A + B}$), **NOT** ($\overline{A}$) & **NAND** ($\overline{A\ B}$).

This leaves us with the **XOR** function ($A \oplus B$)

Consider this map:

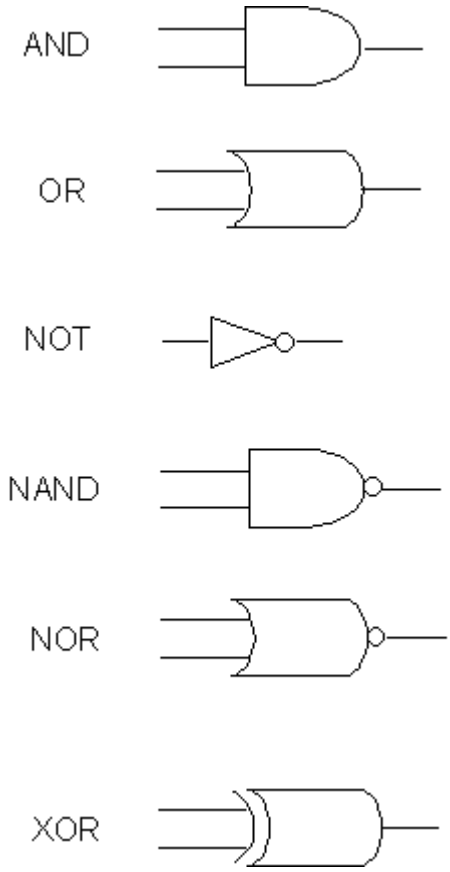
AB / C	00	01		
0		1		1
1		1		1

In both groups, C is eliminated, leaving the expression $Z = A\ \overline{B} + \overline{A}\ B$. This is the "missing" XOR function discussed on page 2. The expression could be re-written $A \oplus B$.

4.2.5 Construct a logic circuit that corresponds to a specific Boolean expression by using standard logic gates:

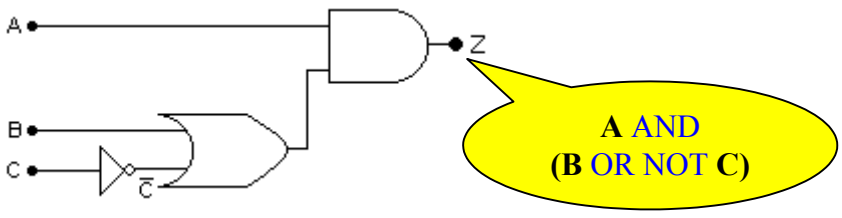
Circuits up to half-adder and full-adder must be constructed.

Logic circuits are constructed from logic gates. Each type of gate performs a specific logic function and has a standard symbol. Below is a list symbols for the gates covered in the syllabus. The function that each performs is pretty obvious (An AND gate performs the AND function etc. etc…!)

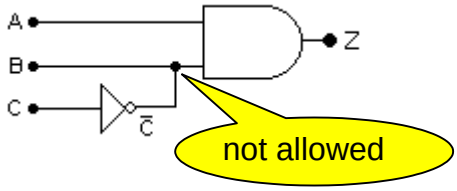


By combining two or more of these gates, logic *circuits* can be constructed to represent more complex Boolean expressions. Take the expression $Z = AB + A\overline{C}$ as an example. The easiest way to handle this is to firstly examine each minterm to see if there is a common condition

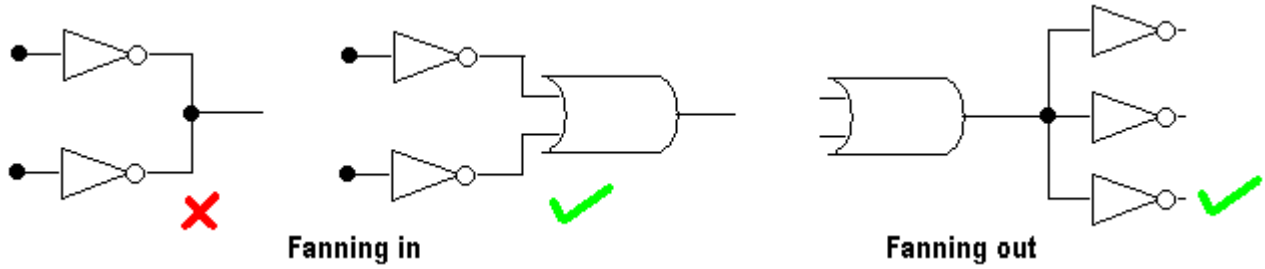
Here **A** exists in both minterms, so the expression could be written as $Z = A\ (B + \overline{C})$. Therefore a bit of thought should lead you to the conclusion that the same AND gate could be pressed into service to serve both minterms. Algebraic fans will recognize this re-write as corresponding to T3, The Distributive Law. Everybody else can just use his or her common sense.



The circuit shown below is logically equivalent to the above example (i.e. the truth table comes out the same) and *appears* to be an even more efficient simplification. However, if we were to construct this circuit using real gates, we would probably end up by overloading the NOT gate.



In practical digital logic circuits, you should never join two outputs together directly (yes, I know that B is an input, but it must also an output from *something*, if only a switch). Combining outputs (called "fanning in" in electronics) should always be done properly via an OR gate or another appropriate gate. On the other hand, "fanning out" is perfectly acceptable to do directly:



The same etiquette applies to logic circuits drawn in Computer Science!

Half-adders and full-adders:

The syllabus requires you to be able to construct circuits up to a half-adder and a full-adder. You could just learn these off by heart – but that wouldn't really teach you how to construct them, or help you to understand what they do and why they are important. Therefore, lets take it from first principles:

One of the most common operations that the CPU has to perform is adding together two binary numbers. The actual adding operation is carried out by arrangements of logic gates in the Arithmetic Logic Unit or ALU. There are two such arrangements, the half-adder and the full-adder.

The half-adder:

From your work with binary mathematics, you should be fairly comfortable with the idea of adding bits. If we want to add just two bits together, there are these four possible outcomes:

0	0	1	1	A
0	1	0	1	B
0	1	1	0	Sum
			carry1	

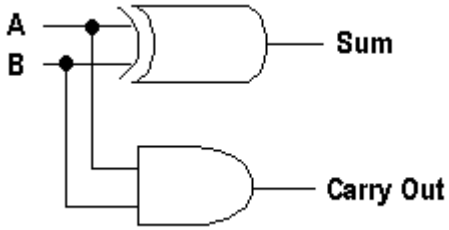
If you compare these with the truth table for the XOR function, you will notice that they are logically equivalent (leaving aside the carry-out, which we'll return to shortly): So to add two numbers together, we can use an XOR gate.

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

XOR

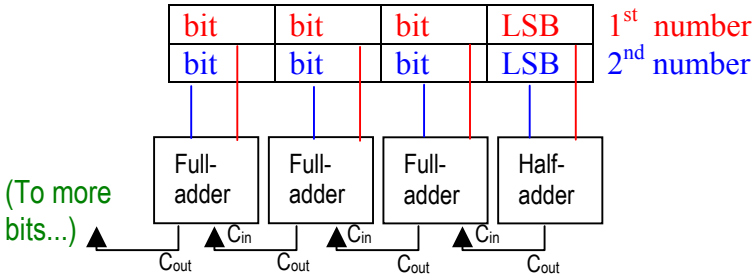
What about that carry-out?

There is only ever a bit to carry out when A AND B are both true (1). So if we extend the circuit to include an AND gate to handle this, we've got what's called a half-adder:



A half adder is used to add together the LSB's (Least Significant Bits) of the two binary numbers we wish to add. All the other bits will need to be added using a circuit that cannot only *generate* carry-outs (C_{out}), but can also correctly *respond to* carry-ins (C_{in}). This circuit is called a full-adder.

Here's the general arrangement:



The full-adder:

The full-adder is constructed from two half-adders (now, there's a surprise…).

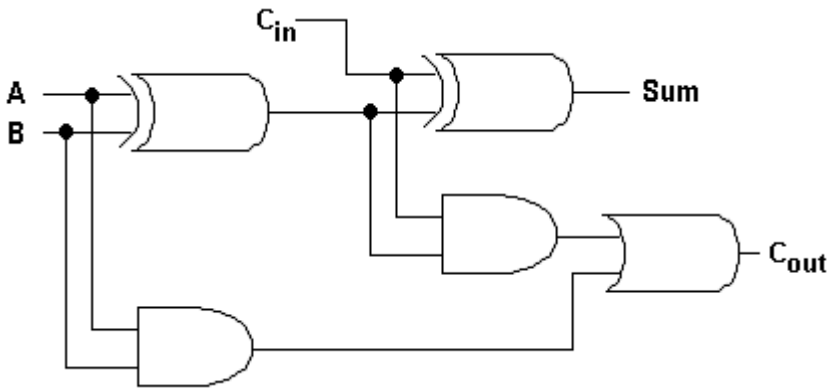
Here's a "bit" more binary maths… The table below shows all the possible combinations for the Sum and Carry-out when you are adding two bits (A and B) along with a Carry-in (C_{in}).

A	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
C_{in}	0	1	0	1	0	1	0	1
Sum	0	1	1	0	1	0	0	1
C_{out}	0	0	0	1	0	1	1	1

This is a little more chewy than the half-adder, but a couple of relationships become evident with some careful observation (plus appropriate use of algebra/maps):

1. The Sum is 1 when $(A \oplus B) \oplus C_{in}$
2. The carry out is 1 when BC + AC + AB

The full-adder circuit below satisfies both of these conditions (of course). Compare this with the half-adder, just to reinforce the relationship between the two!



Obviously, the truth table for a full adder corresponds exactly with the binary additions shown above, however here it is in classic truth table format, just for good measure:

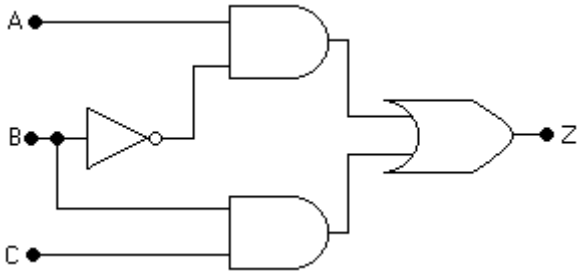
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

4.2.6 Construct a Boolean expression that corresponds to a specific logic circuit:

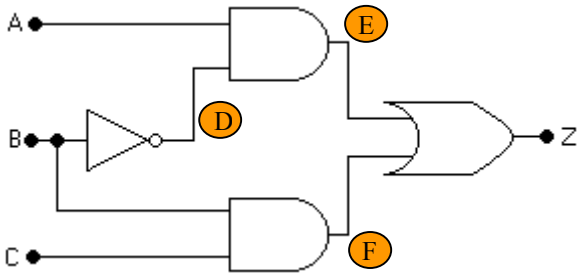
Converting Boolean expressions into their simplest form, as detailed in 4.2.4 above, relies on you having a truth table available from which to extract all the conditions that make the output (Z) true.

For single gates, deriving a truth table and an expression doesn't present us with much of a problem. Since more complex logic circuits are constructed by joining two or more single gates together, we can derive the final truth table (and from that, the Boolean expression it represents) by tracing through the circuit, a single gate at a time. The output from one gate will become one of the inputs to the next gate in the circuit. The easiest way to keep track of all this is to construct a *trace table* with the inputs at one end, the output at the other and whatever interim inputs/outputs you might need in between.

Example: What Boolean expression is this circuit representing?



The first step is to sketch in some interim points on the output of each gate:



Then we can construct a trace table:

A	B	C	D (not B)	E (A.D)	F (B.C)	Z (E+F)
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

From this we can derive an expression for Z (remember that D, E & F are just our temporary labels, they **must not** appear in the final expression!):

$$Z = \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + ABC$$

Simplify according to taste and we arrive at: $Z = A\bar{B} + BC$ or Z = (A not B) OR (BC) - which amounts to the same thing.

I must admit that in this case you could probably have come to that conclusion in a fraction of the time, just by examining the circuit and applying a few moments conscious thought. However, in an exam situation you will almost certainly be awarded quite a few marks for the construction of the trace/truth table and yet more marks for a proper algebraic or Karnaugh map simplification (which I didn't detail here) – So don't take short cuts, even the answer seems obvious!

Worked Past Paper Question:

November 2001, Paper 1, Question 14:

An elevator (lift) operates only between floors 2 and 5 of a building. The current floor number of the lift is stored in a three-bit register in binary code. If the value in the register corresponds to one of the floors that the lift serves, a circuit turns the lights in the elevator on, otherwise it turns them off.

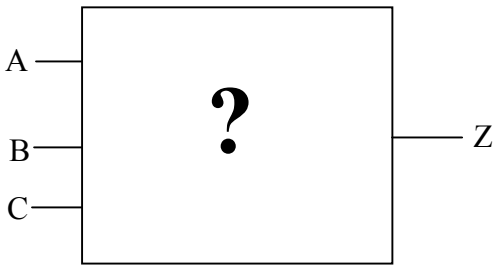
(a) Construct the truth table for the operation of the lights (where an output signal of 1 will activate the lights) for all possible contents of the register. *[4 marks]*

(b) State the Boolean expression for the truth table in (a) using only operators from AND,OR and NOT. *[2 marks]*

(c) Simplify the Boolean expression given in (b). Your final answer can use any valid Boolean operators. *[4 marks]*

The Answer:

(a): We can construct the truth table quite easily from the information given in the question. When the code held in the 3 bit register indicates a floor between 2 and 5 inclusive (corresponding to the three inputs A, B & C), the lights will be on (output Z will be true or 1).



floor	A	B	C	Z
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

(b): From this we can derive the Boolean expression Z = $\overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C$ (This only uses the operators AND, OR and NOT).

(c): Simplify by canceling out as many minterms as possible. It will come as no surprise to you that I am going to opt for a Karnaugh Map:

AB / C	00	01	11	10
0		1		1
1		1		1

C is eliminated, leaving the expression Z = A $\overline{B}$ + $\overline{A}$ B. (*Hmm*… This is beginning to look familiar)

Both you and I know that this expression is an Exclusive OR (XOR) function, so to ensure maximum marks, we should write it as Z = A⊕B or A XOR B.

Et Voila! , Ten marks in the bag.

A bit of advice: **Do** learn to spot Z = A $\overline{B}$ + $\overline{A}$ B as the XOR function. A quick check in the mark scheme revealed that had we left the answer in its "raw" form, a mark would have been deducted!

Recommended Further resources:

Software:

There are a number of logic design and simulation packages available. The Crocodile Clips range is a long-standing favourite and the apps. allow you to simulate far more than just logic circuits. They're commercial packages, so you'll eventually have to invest some money! (Demo's and tryouts are available).
http://www.crocodile-clips.com/index.htm has all the details.

If you want something intuitive, reliable and best of all **free** (Please consider a donation to Multiple Sclerosis Research), try Multimedia Logic from Softronix: http://www.softronix.com/logic.html

Books:

Clive (Max) Maxfield has produced some great books on electronics, logic design and computers – clearly written, easy to understand and humorous. I can personally recommend *Bebop to the Boolean Boogie* from LLH Technological Publishing (ISBN 1-878707-22-1). Max has also co-written *Bebop Bytes Back - An Unconventional Guide to Computers* (available on CD only) and *Designus Maximus Unleashed* – neither of which I have read (yet), but if they are anything like "Boolean Boogie" they'll be worth the effort. See http://www.maxmon.com/default.htm

Cheers, DC