The Exclusive OR (XOR)

The Exclusive Or (XOR) is a binary infix operator defined by the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>f^XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

That is, it is the same as the OR function with the exception that it is *not* TRUE when both A and B are TRUE. Thus, \(^{f^XOR}\) is TRUE whenever A is TRUE or B is TRUE, but not if both are TRUE. The XOR function can be expressed in terms of AND, OR and NOT as

\[
A \ XOR \ B = A'B + AB' \quad \text{[B8]}
\]

as indicated by the truth table.

From now on we will use the symbol ⊕ for the XOR operator.

Two other interpretations are of interest, and make this a very useful function:

- The result is 1 when the arguments are different; the result is 0 when the arguments are the same.
- The result is 1 when an odd number arguments are 1, and 0 when an even number of arguments (including none) are 1.

Like the OR, the XOR is associative and commutative, but it is *not* distributive.

\[
A \oplus B = B \oplus A \\
A \oplus (B \oplus C) = (A \oplus B) \oplus C
\]

Consider \(f = A \oplus B \oplus C\). The truth table for this function is

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
This demonstrates the second of the two bullets above, since a result of 1 appears only if there are one or three 1’s among the arguments. A useful consequence of this property of the XOR is that an XOR function can be complemented simply by complementing any odd number of variables.

\[
\text{if } f = A \oplus B \oplus C, \quad f' = A' \oplus B \oplus C = A \oplus B' \oplus C = A \oplus B \oplus C' = A' \oplus B' \oplus C' = (A \oplus B \oplus C)'
\]

Another useful consequence of this is that any Boolean expression can be complemented by XORing it with 1:

\[
f' = f \oplus 1
\]
Practice problems - Exclusive-Or

1. Identify each of the following truth tables is for an XOR function, a NOT XOR function, or neither. If the function is an XOR or NOT XOR function, write the corresponding boolean expression using only the XOR operator (⊕).

<table>
<thead>
<tr>
<th></th>
<th>ABC f</th>
<th></th>
<th>AB f</th>
<th></th>
<th>ABC f</th>
<th></th>
<th>AB f</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>001</td>
<td>1</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>010</td>
<td>1</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>011</td>
<td>0</td>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

2. Which of the following Boolean expressions represents an Exclusive-Or or a NOT XOR (can be written using only the XOR and NOT operators)? For those that do, rewrite the expression using only XOR and NOT operators

a. AB' + A'B
b. AB + A'B'
c. A'B + AB
d. ABC + AB'C' + A'BC' + A'B'C
e. A'BC + AB'C' + ABC' + A'B'C'
NAND and NOR

The NAND and NOR functions are the invert (complement) of the AND and OR functions. That is, they have the following truth tables, in which all the 1’s and 0’s in the output columns for the AND and OR operators have been complemented (inverted) to get the NAND and NOR functions, respectively.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A NAND B</th>
<th>A</th>
<th>B</th>
<th>A NOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Algebraically, 
\[ A \text{ NAND } B = \text{NOT} \, (A \text{ AND } B) = (AB)' \]
\[ A \text{ NOR } B = \text{NOT} \, (A \text{ OR } B) = (A+B)' \]

The importance of these two functions, or operators, lies in the fact that each of them is sufficient to implement any Boolean Expression! This is valuable in computer chip manufacturing as it means a chip can be built containing only one kind of circuit and personalized only through metalization.

We will use the symbol \( \oplus \) for the NOR function and \( \odot \) for the NAND. We now show that the basic Boolean operators, AND, OR, and NOT can all be expressed using only NANDs or NORs. From this the assertion in the previous paragraph follows.

- \( A' = (A \cdot 1)' = A \oplus 1 \)
- \( A' = (A+0)' = A \oplus 0 \)
- \( A' = (AA)' = A \oplus A \)
- \( A' = (A+A)' = A \oplus A \)

- \( AB = (((AB)'') = (A \oplus B)' = (A \oplus B) \oplus 1 \)
  \[ = (A' + B')' = ((A \oplus 0) + (B \oplus 0))' = (A \oplus 0) \oplus (B \oplus 0) \]

- \( A+B = (A'B)' = A' \oplus B' = (A \oplus 1) \oplus (B \oplus 1) \)
  \[ = ((A+B)'')' = (A \oplus B) \oplus 0 \]

- \( A \odot B = A'B + AB' = ((A \odot 1) \odot B) \oplus (A \odot (B \odot 1)) \)
  \[ = (AB + A'B)'' = AB \oplus A'B' = ((A \odot 0) \odot (B \odot 0)) \oplus (A \odot B) \]

Here are general algorithms for converting AND/OR/NOT expressions into NAND or NOR expressions:
• To convert to a NAND-only expression,
  1. Write the expression in sum of products form;
  2. Use DeMorgan’s Theorem to convert this to the negative of a product of sums.
  3. Rewrite using NANDs. The NAND operator is not associative.

• To convert to a NOR-only expression
  1. Write the expression in product of sums form;
  2. Use DeMorgan’s Theorem to convert this to the negative of a sum of products.
  3. Rewrite using NORs. The NOR operator is not associative.

Example: Write A+BC in NAND and NOR.

NAND: A+BC is already in Sum of Products form.
A+BC = ((A'(BC)')' = A'(B*C)


Practice Problems - NANDs and NORs

1. Assume the following symbolism: A nand B = A B, A nor B = . Evaluate the following Boolean expressions:
   a. A B
   b. B 0
   c. (A 0) 1
   d. B (0 1)

2. Rewrite the following Boolean expressions (written in the NAND/NOR symbolism above) into expressions using traditional AND, OR and NOT symbolism.
   a. (A B) C
   b. A( (A C) 0)
   c. (A B) C
   d. (B 1)

3. Convert the following functions into expressions using only NANDs and also into expressions using only NORs
   a. AB+C
   b. ABC
   c. (A+B)C
   d. A+B+C
Boolean Operator Application Notes

1. Bit manipulation (bit maps, word processing case conversion)
2. Error Checking (parity)
3. Chip design (Nands & Nors are ‘universal’ operators)

Practice Problems - Boolean Operator Applications

Logic Circuits

Logic circuits are electronic devices which implement Boolean expressions. Such circuits can be exceedingly complex, but they are built up from a few fundamental circuits, called logic gates, or just ‘gates’ (no relation.) As you might expect, there is an AND gate, an OR gate, and a NOT gate (which is commonly called an inverter, since it inverts the level of the incoming signal.) Since these gates are commonly used to implement more complex logic circuits, there is a standardized set of figures which are used to draw such circuits.
The function \( f = ABC + D' \) would be implemented in hardware as

**Practice Problems - Logic Circuits**

1. Draw the logic circuits corresponding to each of the following Boolean expressions. \textbf{Do not} simplify the expressions.

   a. \( AB + C(A' + B)' \)
   b. \( A + B + C(A' + C') \)
   c. \( (A + B')(BC + A) + D \)
   d. \( AB' + A'B \)

**Binary Addition Hardware**

Finally, to tie this topic and number systems together, let’s develop the Boolean expressions for hardware which performs Binary Addition. We will dispense with actually drawing the logic diagrams.

Consider an arbitrary 3-bit addition \( b_1 + b_2 = s \)

\[
\begin{align*}
&b_1 b_1 b_1_0 \\
+ &b_2 b_2 b_2_0 \\
&\underline{s_2 s_1 s_0}
\end{align*}
\]
If we take an arbitrary bit position, \( i \), then there may be a carry into that position from the results of adding the bits to the right of \( i \), and there may be a carry out of the \( i \)th position. Let’s call these carries \( c_{i-1} \) and \( c_i \), respectively. Then, for the \( i \)th position we can generate the truth table for this addition:

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_{i-1}, b_1, b_2 )</td>
<td>( s, c_i )</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

This is the truth table for a device called a **Full Adder**, which adds *two bits plus a carry and produces a sum and a carry*. The carry out becomes the carry in of the next Full Adder, and we can string together as many of these Full Adders as desired to produce additions of any number of bits.\(^6\)

We can now write the Boolean expressions which correspond to this truth table. In DNF form these are:

\[
\begin{align*}
  s_i &= c_{i-1}'b_1'b_2_i + c_{i-1}'b_1b_2_i' + c_{i-1}b_1'b_2_i' + c_{i-1}b_1b_2_i \\
  c_i &= c_{i-1}'b_1b_2_i + c_{i-1}b_1'b_2_i + c_{i-1}b_1b_2_i' + c_{i-1}b_1b_2_i
\end{align*}
\]

\(^*[B9, B10]\)

Let’s simplify these expressions using Karnaugh Maps.

---

\(^6\) In practice, large adders (more than 8 or 16 bits) are not built this way as the propagation delays become too large - that is, it takes too long for the signals from \( S_0 \) to propagate to \( c_n \). Engineers have developed parallel circuits to alleviate this problem.
For $s_i$, we have

\[
\begin{array}{cccc}
& b_1'b_2' & b_1'b_2 & b_1b_2 & b_1b_2' \\
\hline
c_{i-1}' & 0 & 1 & 0 & 1 \\
c_{i-1} & 1 & 0 & 1 & 0 \\
\end{array}
\]

Two points are of interest here:

4. There is no simplification possible since there are no 1's in adjacent cells which we can loop together.
5. A ‘checkerboard’ pattern of 0’s and 1’s such as this can always be represented as an exclusive-OR ‘tree’ (a circuit consisting solely of XOR gates).

From the second point above we can write

\[
s_i = c_{i-1} \oplus b_1 \oplus b_2
\]

[\text{B11}]

The Karnaugh Map for $c_i$ is

\[
\begin{array}{cccc}
& b_1'b_2' & b_1'b_2 & b_1b_2 & b_1b_2' \\
\hline
c_{i-1}' & 0 & 0 & 1 & 0 \\
c_{i-1} & 0 & 1 & 1 & 1 \\
\end{array}
\]

From this $c_i$ can be simplified to

\[
c_i = b_1b_2 + c_{i-1}b_2 + c_{i-1}b_1
\]

[\text{B12}]

As you can see, simplification is important from a hardware consideration since gates are a precious resource on the limited surface area of a chip. In this case the DNF functions [B9] and [B10] would require 5 gates each (assuming 3-input AND gates and 4-input OR gates were available) while [B11] only requires one 3-input XOR gate (if available) and [B12] needs only 3 2-input AND gates and 1 3-input OR gate. The net savings in circuits is 10 - 5 = 5, or 50% in silicon real estate.

Detecting overflows
Recall that an overflow is detected whenever the carry out of the high order bit and the carry into the high order bit are different. That is,

$$\text{Overflow} = c_{n-1} \cdot c_{n-2} + c_{n-1}' \cdot c_{n-1}$$

Looking back at [B8], this is the Exclusive-Or of $c_{n-1}$ and $c_{n-2}$,

$$\text{Overflow} = c_{n-1} \oplus c_{n-2}$$