Sum of Products Algorithm

- Automates construction of circuit from truth table
- Identify each row of the output that has a 1.
- For each such row
  - Make a product of all the input variables.
  - Put bar over each variable with a 0 in this row.
  - Make a sum of all of these product terms.

Sum of Products Example

- Logical implication: e.g., “If it's raining, it's also cloudy”

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A → B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum of Products Example

- Identify rows with a 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A → B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Sum of Products Example**

- Make a product of the input variables

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(A \cdot B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 (\overline{A}\overline{B})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 (AB)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (AB)</td>
</tr>
</tbody>
</table>

**Sum of Products Example**

- Put bar over variable that is zero in its row:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(A \cdot B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 (\overline{A}\overline{B})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 (\overline{AB})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (AB)</td>
</tr>
</tbody>
</table>

**Sum of Products Example**

- Make a sum of all these product terms:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(A \cdot B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 (\overline{A}\overline{B})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 (\overline{AB})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (AB)</td>
</tr>
</tbody>
</table>

\(\overline{A}\overline{B} + \overline{AB} + AB\)

**Sum of Products Example**

- Now simplify:

\(\overline{A}\overline{B} + \overline{AB} + AB\)

\(\overline{A}(\overline{B} + B) + AB\)

\(\overline{A}(1) + AB\)

\(\overline{A} + AB\)
One-bit Compare for Equality

- We want a circuit with
  - Two input lines (1 bit each)
  - One output line
  - Output is 1 if the two inputs are equal
  - Output is 0 if the inputs are not equal
- Circuit will be referred to as 1-CE for 1 bit compare for equality.

1-CE Design

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A(\neq)B</th>
<th>The Boolean expression is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>AB + AB</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

3-CE

- We want a circuit with
  - Two 3-bit numbers as input (6 input lines)
  - One output
  - Output is 1 only if the two input numbers are equal
- Using the sum of products would require a table with 64 rows!

Use Abstraction

- Once we've built a circuit, we can refer to it without having to show the whole thing.
- Represent the circuit as a labeled box, called a module.
- (Re)use the module to build bigger, more complicated circuits.
- A fundamental design strategy in CS – like Java methods (re-use code)
- *How modular is Nature?*
3-CE: Use abstraction
A1 A2 A3 = B1 B2 B3 only if A1=B1 AND A2=B2 AND A3=B3

Digression: Binary Numbers (Section 4.2)
- Base-10 (decimal) numbers are more familiar:
  \[ 943_{10} = 9 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 \]
- Base-2 (binary) numbers work the same way, but with 2 instead of 10 as the base:
  \[ 101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10} \]

Digression: Binary Numbers (Section 4.2)
- To add numbers in Base-10, we compute the sum of each column, and carry a 1 if the sum is greater than 9:

\[
\begin{array}{c}
  9 \ 1 \ 5 \\
+ \ 1 \ 2 \ 8 \\
\hline
  \ 3
\end{array}
\]

\[
\begin{array}{c}
  1 \\
9 \ 1 \ 5 \\
+ \ 1 \ 2 \ 8 \\
\hline
  \ 3
\end{array}
\]
Digression: Binary Numbers  
(Section 4.2)
• To add numbers in Base-10, we compute the sum of each column, and carry a 1 if the sum is greater than 9:

\[
\begin{array}{c}
0 \\
915 \\
+ 128 \\
\hline
43
\end{array}
\]
Digression: Binary Numbers  
(Section 4.2)

- To add numbers in Base-10, we compute the sum of each column, and carry a 1 if the sum is greater than 9:

\[
\begin{array}{c}
1 \\
915 \\
+ 128 \\
\hline
1043
\end{array}
\]

Digression: Binary Numbers  
(Section 4.2)

- To add numbers in Base-2, we do the same thing, except that we carry when the sum in a column exceeds 1:

\[
\begin{array}{c}
1 \\
011 \\
+ 101 \\
\hline
00
\end{array}
\]
Digression: Binary Numbers
(Section 4.2)
• To add numbers in Base-2, we do the same thing, except that we carry when the sum in a column exceeds 1:

\[
\begin{array}{c}
0 \\
0 1 1 \\
+ 1 0 1 \\
0 0
\end{array}
\]

Half Adder Circuit
• For right hand column of binary addition.
• Two 1-bit inputs.
• Output for sum bit (recorded at bottom).
• Output for carry bit.
  1 (carry bit)
  101
  101
  0 (sum bit)

Digression: Binary Numbers
(Section 4.2)
• To add numbers in Base-2, we do the same thing, except that we carry when the sum in a column exceeds 1:

\[
\begin{array}{c}
0 \\
0 1 1 \\
+ 1 0 1 \\
1 0 0
\end{array}
\]

Half Adder - Continued
\[
\begin{array}{cccc}
A & B & \text{Sum} & \text{Carry} \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]
• Sum: \(AB + \overline{AB}\)
• Carry: \(AB\)
**Full Adder Circuit**

- Columns other than right hand column.
- Three 1-bit inputs.
- Output for sum bit (recorded at bottom).
- Output for carry bit.
  - 11 (carry bit)
  - 111
  - 10 (sum bit)

**Full Adder - continued**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$S$</th>
<th>$C'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum: __ __ __ __

ABC+ABC+ABC

Carry: __ __ __

ABC+ABC+ABC

+ABC

**Full Adder - The Circuit**

**Check A=1, B=1, C=0**
Check A=1, B=1, C=1

3-bit adder: Use abstraction

- Circuit to add two 3-bit numbers A1 A2 A3 and B1 B2 B3