Functional Blocks: Addition

- Binary addition occurs frequently in digital and computer systems.
- In this section, we:
  - Develop a Half-Adder (HA), a 2-input bitwise addition functional block,
  - Develop a Full-Adder (FA), a 3-input bitwise addition functional block,
  - Iterate full-adders using a ripple-carry to perform parallel binary addition, and
  - Develop a Carry-Look-Ahead Adder (CLA) to improve performance.
Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that produces the following values:
  
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>+Y</th>
<th>+1</th>
<th>+0</th>
<th>+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- The sum is expressed as a sum bit, S and a carry bit, C.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- A half adder adds two bits to produce a two-bit sum.

Logic Simplification: Half-Adder

- The K-Map for S, C is:

```
  0 1
0 1 2 3
X
```

- This is a pretty trivial map! By inspection:

\[
S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y
\]

\[
S = (X + Y) \cdot (\overline{X} + \overline{Y})
\]

and

\[
C = X \cdot Y
\]

\[
C = ((X \cdot Y))
\]

These equations lead to several implementations.
Five Implementations: Half-Adder

- We can derive following sets of equations for a half-adder:

\[
\begin{align*}
(a) \quad & S = X \cdot \overline{Y} + \overline{X} \cdot Y \\
& C = X \cdot Y \\
(b) \quad & S = (X + Y) \cdot (\overline{X} + \overline{Y}) \\
& C = X \cdot Y \\
(c) \quad & S = (C + X \cdot Y) \\
& C = X \cdot Y \\
(d) \quad & S = (X + Y) \cdot \overline{C} \\
& C = (X + \overline{Y}) \\
(e) \quad & S = X \oplus Y \\
& C = X \cdot Y
\end{align*}
\]

- (a), (b), and (e) are SOP, POS, and XOR implementations for \( S \).

- In (c), the \( C \) function is used as a term in the AND-NOR implementation of \( S \), and in (d), the \( \overline{C} \) function is used in a POS term for \( S \).

Implementations: Half-Adder

- The most common half adder implementation (e) is:

\[
S = X \oplus Y \\
C = X \cdot Y
\]

- A NAND only implementation (equivalent to equation d) is:

\[
S = (X + Y) \cdot C \\
C = ((X \cdot \overline{Y}))
\]
Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.
  - For a carry-in (Z) of zero, it is the same as the half-adder:
    \[
    \begin{array}{c|c|c|c|c}
    Z & X & Y & + & C & S \\
    \hline
    0 & 0 & 0 & +0 & 0 & 0 \\
    0 & 0 & 1 & +1 & 1 & 1 \\
    \end{array}
    \]
  - For a carry-in (Z) of one:
    \[
    \begin{array}{c|c|c|c|c|c|c|c|c}
    Z & X & Y & + & + & C & S \\
    \hline
    1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
    1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]

Design: Full-Adder

- Full-Adder Function Table:
  \[
  \begin{array}{c|c|c|c|c}
  X & Y & Z & C & S \\
  \hline
  0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 1 \\
  0 & 1 & 0 & 0 & 1 \\
  0 & 1 & 1 & 1 & 0 \\
  1 & 0 & 0 & 0 & 1 \\
  1 & 0 & 1 & 1 & 0 \\
  1 & 1 & 0 & 1 & 0 \\
  1 & 1 & 1 & 1 & 1 \\
  \end{array}
  \]
- Full-Adder K-Map:
**Design: Full-Adder**

- From the K-Map, we get:
  \[ S = X \overline{Y} \overline{Z} + \overline{X} Y \overline{Z} + \overline{X} Y Z + X Y Z \]
  \[ C = X Y + X Z + Y Z \]
- The S function is the three-bit XOR function (Odd Function):
  \[ S = X \oplus Y \oplus Z \]
- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:
  \[ C = X Y + (X \oplus Y) Z \]
- The term X·Y is **carry generate**.
- The term X·Y is **carry propagate**.

**Implementation: Full Adder**

- **Full Adder Schematic**

- Here X, Y, and Z, and C (from the previous pages) are A, B, C\(_i\) and C\(_o\) respectively. Also,
  - G = Generate and
  - P = Propagate.
- **Note:** This is really a combination of a 3-bit odd function (for S = sum) and Carry logic:
  \[ (G = \text{Generate}) \text{ OR } (P = \text{Propagate AND } C_i = \text{Carry In}) \]
  \[ C_o = G + P \cdot C_i \]
Parallel Binary Adders

- To add more than one bit, we "bundle" sets of logical signals together and build devices that operate on the whole set in parallel.

- Example: 4-bit binary adder:
  Adds an input vector "A(3..0)" to "B(3..0)"
to get a sum S(3..0) thus:

<table>
<thead>
<tr>
<th>Description</th>
<th>Subscript</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Carry</td>
<td>0 1 1 0</td>
<td>C_i</td>
</tr>
<tr>
<td>Augend</td>
<td>1 0 1 1</td>
<td>A_i</td>
</tr>
<tr>
<td>Addend</td>
<td>0 0 1 1</td>
<td>B_i</td>
</tr>
<tr>
<td>Sum</td>
<td>1 1 1 0</td>
<td>S_i</td>
</tr>
<tr>
<td>Output Carry</td>
<td>0 0 1 1</td>
<td>C_{i+1}</td>
</tr>
</tbody>
</table>

  Note: the carry out of Stage i becomes the carry in of Stage i+1.

4-bit Ripple-Carry Binary Adder

- A four-bit Full Adder made from four 1-bit Full Adders:

- Here FA is a Full-Adder from before:
Carry Propagation & Delay

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:

- Note: The "long path" is from A(0) or B(0) though the network to either C(4) or S(3).

Carry Look-Ahead

- Given Stage i from a Full Adder, we know that there will be a carry generated when \( A_i = B_i = "1" \), whether or not there is a carry-in.
- Alternately, there will be a carry propagated if the "Half-Sum" is "1" and a carry-in, \( C_i \) occurs.
- These two signal conditions are called Generate denoted as \( G_i \), and Propagate denoted as \( P_i \) respectively and are shown here:
Carry Look-Ahead (Continued)

- By defining the equations for the Full Adder in term of the \( P_i \) and \( G_i \), we have:
  \[
  P_i = A_i \oplus B_i \\
  G_i = A_i \cdot B_i
  \]
- And the output sum \( S(i) \) and carry \( C(i+1) \) is defined as:
  \[
  S_i = P_i \oplus C_i \\
  C_{i+1} = G_i + P_i \cdot C_i
  \]
- Starting the stage numbering at zero, we have:
  \[
  C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 \\
  + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0
  \]
where \( C_0 \) is a carry in to the least significant bit.

Carry Look-Ahead (Continued)

- Look at the following addition examples, all of which generate a carry of 1 out of the third stage:

<table>
<thead>
<tr>
<th>Example</th>
<th>Carry Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xxx +1xxx \rightarrow 1xxxx</td>
<td>Generate a carry in Stage 3.</td>
</tr>
<tr>
<td>11xx +01xx \rightarrow 10xxx</td>
<td>Generate a carry in Stage 2 and propagate it through stage 3.</td>
</tr>
<tr>
<td>111x +001x \rightarrow 100xx</td>
<td>Generate a carry in Stage 1 and propagate it through stage 2 &amp; stage 3.</td>
</tr>
<tr>
<td>1111 +0001 \rightarrow 1000x</td>
<td>Generate a carry in stage 0 and propagate it through Stage 1, stage 2, &amp; stage 3.</td>
</tr>
<tr>
<td>1111 +0000 \rightarrow 10000</td>
<td>Use a carry into stage 0 and propagate it through stage 0, Stage 1, stage 2 &amp; stage 3.</td>
</tr>
</tbody>
</table>
Group Carry Look-Ahead Logic

- Figure 3-28 in the text shows how to implement a carry look-ahead circuit for four bits. This could be extended to more than four bits. In practice, though, it becomes more difficult to implement this over more than a few bits. The concept can be extended another level by considering a Group Generate \((G_{0,3})\) and Group Propagate \((P_{0,3})\) logic condition:

\[
G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0
\]

\[
P_{0-3} = P_3 P_2 P_1 P_0
\]

- Using these two equations:

\[
C_4 = G_{0-3} + P_{0-3} C_0
\]

- Thus, it is possible to have four 4-bit adders use one of the same carry look-ahead circuits to add 16 bits!