Binary Adder [Wakerly 4th Ed., Sec. 6.10, p. 474]

- Binary addition is used frequently
- Addition Development:
  - Full-Adder (FA), a 3-input bit-wise addition functional block,
  - Ripple Carry Adder, an iterative array to perform binary addition, and
Functional Block: Half Adder

- A 2-input, 1-bit width binary adder that performs the following computations:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>+1</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>CO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HS</td>
<td>0</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.
- The low-order bit is named “half sum” (HS), and the high-order bit is named “carry out” (CO).
- The half adder can be specified as a truth table for HS and CO:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>CO</th>
<th>HS</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>

Half Adder

- Half-adder for 1-bit addends:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>CO</th>
<th>HS</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>

Half sum: \( HS = X \oplus Y \)

Carry out: \( CO = X \cdot Y \)
Full Adder [Recall Binary Addition from Lecture #2]

- Basic building block is “full adder”
  - 1-bit-wide adder, produces sum and carry outputs

- Truth table:

<table>
<thead>
<tr>
<th>Row</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Full Adder from Half Adders
Full Adder

\[ S = H \oplus C_{\text{in}} = X \oplus Y \oplus C_{\text{in}} = (X \cdot Y' + X' \cdot Y) \oplus C_{\text{in}} \]

\[ = X \cdot Y' \cdot C_{\text{in}}' + X' \cdot Y \cdot C_{\text{in}}' + X' \cdot Y' \cdot C_{\text{in}} + X \cdot Y \cdot C_{\text{in}} \]

- first term direct
- first term complement

\[ C_{\text{out}} = X \cdot Y + X \cdot C_{\text{in}} + Y \cdot C_{\text{in}} \]

Logic Optimization: Full Adder

- Full adder Karnaugh map

<table>
<thead>
<tr>
<th>Row</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S = X \cdot Y \cdot C_{\text{in}}' + X' \cdot Y' \cdot C_{\text{in}}' + X' \cdot Y \cdot C_{\text{in}} + X \cdot Y \cdot C_{\text{in}} \]

\[ C_{\text{out}} = X \cdot Y + X \cdot C_{\text{in}} + Y \cdot C_{\text{in}} \]
Subtraction

- Subtraction is the same as addition of the twos complement
- Recall Lecture #2:
  The two’s complement is the bit-by-bit complement plus 1
- Therefore, $X - Y = X + \bar{Y} + 1$
  - Complement Y inputs to adder, set first $C_{in}$ to 1
Subtractor Design Using Adders

- Ripple subtractor

\[ x_n \rightarrow y_n \rightarrow d_n \]

\[ COut \rightarrow CIn \]

\[ 74x04 \]

\[ \bar{Y}_i \]

\[ X_3 \rightarrow Y_3 \rightarrow COut \rightarrow CIn \]

\[ X_2 \rightarrow Y_2 \rightarrow COut \rightarrow CIn \]

\[ X_1 \rightarrow Y_1 \rightarrow COut \rightarrow CIn \]

\[ X_0 \rightarrow Y_0 \rightarrow COut \rightarrow CIn \]

\[ S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow S_0 \]

\[ \bar{S}_3 \rightarrow \bar{S}_2 \rightarrow \bar{S}_1 \rightarrow \bar{S}_0 \]

Subtractor Design Using Adders

- Ripple subtractor

\[ x_{n-1} \rightarrow y_{n-1} \rightarrow d_{n-1} \]

\[ b_{L_m} \rightarrow b_{L_{m-1}} \rightarrow b_{L_{m-2}} \rightarrow b_{L_1} \rightarrow b_{L_0} \rightarrow 1 \]

\[ OUT \rightarrow BIN \]

\[ d_{n-1} \rightarrow d_{n-2} \rightarrow \ldots \rightarrow d_1 \rightarrow d_0 \]
2's Complement Adder/Subtractor

- Subtraction can be done by addition of the 2's Complement.
  1. Complement each bit (1's Complement.)
  2. Add 1 to the result.
- The circuit shown computes both \( A + B \) and \( A - B \):
  - For \( S = 1 \), subtract, the 2’s complement of B is formed by using XORs to form the 1’s comp and adding the 1 applied to \( C_0 \).
  - For \( S = 0 \), add, B is passed through unchanged.

![Diagram of 2's Complement Adder/Subtractor](image.png)

How to Detect Overflow

- Rule was: Sign of the two operands identical and different from the sign of the result \([\text{recall Lecture #3}])
- Sign = most significant bit (MSB)

\[
OVR = X_{n-1} \cdot Y_{n-1} \cdot S'_{n-1} + X'_{n-1} \cdot Y'_{n-1} \cdot S_{n-1}
\]

or:

\[
OVR = C_{n-1} \oplus C_n \quad \text{carry-in different from carry-out}
\]

<table>
<thead>
<tr>
<th></th>
<th>000... 1</th>
<th>011... 1</th>
<th>( 2^{n-1} - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVR</td>
<td>( 0 \oplus 0 )</td>
<td>( 1 \oplus 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

OVR = \( 0 \oplus 0 = 1 \cdot 1 = 1 \) or
OVR = \( 1 \oplus 0 = 1 \)
Ripple Adder

- To add multiple operands, we “bundle” logical signals together into vectors and use functional blocks that operate on the vectors.

- Example:
  4-bit ripple carry adder:
  Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0).

- Note: carry-out of block $i$ becomes carry-in of block $i + 1$.

<table>
<thead>
<tr>
<th>Description</th>
<th>Bit index</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry in</td>
<td>0 1 1 0</td>
<td>Ci</td>
</tr>
<tr>
<td>Augend</td>
<td>1 0 1 1</td>
<td>Ai</td>
</tr>
<tr>
<td>Addend</td>
<td>0 0 1 1</td>
<td>Bi</td>
</tr>
<tr>
<td>Sum</td>
<td>1 1 1 0</td>
<td>Si</td>
</tr>
<tr>
<td>Carry out</td>
<td>0 0 1 1</td>
<td>Ci+1</td>
</tr>
</tbody>
</table>

Ripple Adder

- It is relatively slow: For $n$ bits, the worst case is:
  - All of the adder’s bits (and $c_0$) are present simultaneously.
  - $t_{ADD} = t_{XY\text{OUT}} + (n-2) \cdot t_{CIN\text{OUT}} + t_{\text{INS}}$

  $\text{LSB (out } C_1\text{)}$  $\text{MSB (in } C_{n-1}\text{)}$

- Carry look-ahead adders are the solution.
Carry Lookahead Adder

- Uses a different circuit to calculate the carry out (calculates it *ahead* of the addition), to speed up the overall addition

- Requires more complex circuits

- Trade-off: *speed vs. area* (complexity, cost)

---

Carry Look-Ahead Addition

- **Carry generate**: input bits combination \((x_i, y_i)\) that produces a carry-out of “1” \((c_{i+1} = 1)\) independent of lower-order bits \((x_0 \ldots x_{i-1}, y_0 \ldots y_{i-1})\) and \(c_0\).

- **Carry propagate**: input bits combination \((x_i, y_i)\) that produces a carry-out of “1” \((c_{i+1} = 1)\) when \(c_i = 1\).

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
X & Y & Cin & S & Cout \\
0 & 0 & 0 & 0 & 0 & \text{No carry} & & & & \\
0 & 0 & 1 & 0 & 0 & \text{No carry} & & & & \\
0 & 1 & 0 & 0 & 0 & \text{No carry} & & & & \\
0 & 1 & 1 & 0 & 1 & \text{Carry propagated} & & & & \\
1 & 0 & 0 & 1 & 0 & \text{No carry} & & & & \\
1 & 0 & 1 & 1 & 0 & \text{Carry propagated} & & & & \\
1 & 1 & 0 & 0 & 1 & \text{Carry generated} & & & & \\
1 & 1 & 1 & 1 & 1 & \text{Carry generated} & & & & \\
\end{array}
\]

\[
\begin{aligned}
g_i &= x_i \cdot y_i & \leftarrow& \text{carry-generate} \\
p_i &= x_i + y_i & \leftarrow& \text{carry-propagate} \\
\Rightarrow c_{i+1} &= g_i + p_i \cdot c_i \\
\end{aligned}
\]

Note that we could use half adder: 
\(p_i = x_i \oplus y_i\)

---
### 4-bit Carry Lookahead Adder

- **Conceptual diagram**

  \[ s_i = H S_i \oplus c_i = p_i \oplus c_i \]

  \[ c_{i+1} = g_i + p_i \cdot c_i \]

  Note that \( g_i = 1 \Rightarrow p_i = 1 \)
  (but not vice versa)

  \( g_i \Rightarrow p_i \cdot g_i \)

### Carry Lookahead Logic

- **Structure of one stage of a carry-lookahead adder:**

  \( g_i = x_i \cdot y_i \quad \text{carry-generate signal} \)

  \( p_i = x_i + y_i \quad \text{carry-propagate signal} \)
Carry equations for first 4 adder stages

\[ c_1 = p_0 \cdot (g_0 + c_0) \]
\[ c_2 = p_1 \cdot (g_1 + c_1) = p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0)) = p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0) \quad \text{distributivity theorem} \]
\[ c_3 = p_2 \cdot (g_2 + c_2) = p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)) = p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0) \]
\[ c_4 = p_3 \cdot (g_3 + c_3) = p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)) = p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \cdot (g_3 + g_2 + g_1 + g_0 + c_0) \]

74x283 4-bit Adder

- Uses carry lookahead (CLA) internally
- Differences from general CLA design:
  - Active-low versions of carry-generate \((g'_i)\) and carry-propagate \((p'_i)\) (b/c inverting gates are faster)
  - Algebraic manipulation of the half-sum:
    \[ h_s = x \cdot y' + x' \cdot y = (x + y) \cdot (x + y') \]
    \[ = p_i \cdot g_i \]
  - Creates the carry signals using INVERT-OR-AND (has \(\approx\) delay as a single inverting gate):
    \[ c_{i+1} = p_i \cdot g_i \]
    \[ = p_i \cdot (g_i + c_i) \]
- See Wakerly 4th ed., page 481, for carry equations
74x283 4-bit Adder (detail)

- \( p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0) \)
- \( p_0 \cdot (g_0 + c_0) \)

"generate"

"propagate"

"half sum"

carry-in from previous stage

16-bit Group-ripple Adder

- Ripple carry between groups
- Total propagation delay \( \approx 8 \) inverting gates