EECS150 - Digital Design
Lecture 10 - Combinational Logic Circuits
Part 1
February 26, 2002
John Wawrzynek

Combinational Logic (CL) Defined

\[ y = f(x_0, \ldots, x_{n-1}) \], where \( x, y \in \{0,1\} \).
Y is a function of only X.

- If we change X, Y will change immediately (well almost!).
- There is an implementation dependent delay from X to Y.

Adders

Full-adder cell (FA) revisited:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>s</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</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>

Carry-ripple Adder

- Each cell:
  \[ r_i = a_i \oplus b_i \oplus c_{in} \]
  \[ c_{out} = c_i + a_i b_i = c_{in} (a_i + b_i) + a_i \cdot b_i \]

- 4-bit adder: “Full adder cell”

- What about subtraction?

Subtractors

A - B = A + (-B)

How do we form -B?
- 1. complement B
- 2. add 1

Adders (cont.)

Ripple Adder

Ripple adder is inherently slow because, in general, \( s_7 \) must wait for \( c_7 \) which must wait for \( c_6 \) …

\[ T \propto n, \quad \text{Cost} \propto n \]

How do we make it faster, perhaps with more cost?
Carry Select Adder

- Extending Carry-select to multiple blocks
  - What is the optimal # of blocks and # of bits/block?
    - If # blocks too large delay dominated by total mux delay
    - If # blocks too small delay dominated by adder delay

\[ T = \text{ripple adder delay} / 2 + T_{\text{mux}} \]

\[ \text{COST} = 1.5 \times \text{COST}_{\text{ripple adder}} + (n+1) \times \text{COST}_{\text{mux}} \]

Carry Look-ahead Adders

- In general, for n-bit addition best we can achieve is delay \( \alpha \log(n) \)
- How do we arrange this? (think trees)
  - First, reformulate basic adder stage:
    \[
    \begin{array}{c|c|c|c|c}
    a & b & c_{i-1} & s & c_i \\
    \hline
    0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 1 & 1 \\
    1 & 0 & 0 & 1 & 1 \\
    1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]
    - carry "kill" (\( c_{i-1} = b_i \))
    - carry "propagate" (\( b_i = a_i \iff b_i \))
    - carry "generate" (\( c_i = a_i \iff b_i \))

Carry Look-ahead Adders

- Ripple adder using \( p \) and \( g \) signals:
  - \( s_{i+1} = g_i + p_i c_i \)
  - \( s_i = p_i \iff c_i \)
  - So far, no advantage over ripple adder: \( T \propto N \)

Carry Look-ahead Adders

- Expand carries:
  - \( c_i = c_{i-1} + b_i c_i \)
  - \( c = c_{i-1} + b_i c_i + b_i p_i c_i \)
  - Why not implement these equations directly to avoid ripple delay?
    - Lots of gates. Redundancies (full tree for each).
    - Gate with high # of inputs.
  - Let's reorganize the equations.
Carry Look-ahead Adders

- “Group” propagate and generate signals:
  - \( P \) true if the group as a whole propagates a carry to \( c_{\text{out}} \)
  - \( G \) true if the group as a whole generates a carry
  - Group \( P \) and \( G \) can be generated hierarchically.

\[
P = p_i p_{i+1} \ldots p_{i+k}
\]

\[
G = g_i + p_{i+k} g_{i+k-1} + \ldots + (p_{i+1} p_{i+2} \ldots p_{i+k}) g_i
\]

\[
c_{\text{out}} = G + PC
\]

8-bit Carry Look-ahead Adder with 2-input gates.

Adders in FPGAs