Adders

Full-adder cell (FA) revisited:

\[
\begin{array}{cccc}
 a & b & c_{in} & c_{out} & s \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 1 \\
 0 & 1 & 0 & 0 & 1 \\
 0 & 1 & 1 & 1 & 0 \\
 1 & 0 & 0 & 0 & 1 \\
 1 & 0 & 1 & 1 & 0 \\
 1 & 1 & 0 & 1 & 0 \\
 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Carry-ripple Adder

- Each cell:
  \[ r_i = a_i \text{ XOR } b_i \text{ XOR } c_{\text{in}} \]
  \[ c_{\text{out}} = a_i c_{\text{in}} + a_i b_i + b_i c_{\text{in}} = c_{\text{in}} (a_i + b_i) + a_i b_i \]

- 4-bit adder:

```
FA
```

“Full adder cell”

- What about subtraction?

Subtractors

\[ A - B = A + (-B) \]

How do we form \(-B\)?

1. complement \(B\)
2. add 1
Delay in Ripple Adders

- Ripple delay amount is a function of the data inputs:

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & & & & \\
1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & & & & \\
1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & & & & \\
1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & & & & \\
\end{array}
\]

- However, we usually only worry about the worst case delay on the critical path. There is always at least one set of input data that exposes the worst case delay.

Adders (cont.)

Ripple Adder

Ripple adder is inherently slow because, in general, \( s7 \) must wait for \( c7 \) which must wait for \( c6 \) …

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

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

\[ T = \frac{T_{\text{ripple_adder}}}{2} + T_{\text{MUX}} \]

\[ \text{COST} = 1.5 \times \text{COST}_{\text{ripple_adder}} + (n+1) \times \text{COST}_{\text{MUX}} \]

- 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

\[ \sqrt{N} \text{ stages of } \sqrt{N} \text{ bits} \]

\[ T \propto \sqrt{N}, \quad \text{Cost} \approx 2 \times \text{ripple} + \text{muxes} \]
Carry Select Adder

- \( T_{\text{total}} = \sqrt{N} T_{FA} \)
  - assuming \( T_{FA} = T_{MUX} \)
  - For ripple adder \( T_{\text{total}} = N T_{FA} \)

- Is \( \sqrt{N} \) really the optimum?
  - From right to left increase size of each block to better match delays
  - Ex: 64-bit adder, use block sizes \([13 12 11 10 9 8 7]\)
- How about recursively defined carry select?

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:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c_{i-1}</th>
<th>s_{i}</th>
<th>carry</th>
<th>k_i</th>
<th>p_i</th>
<th>g_i</th>
<th>c_{i+1}</th>
<th>s_{i}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>“kill”</td>
<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>“propagate”</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>“propagate”</td>
<td>1</td>
<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>
<td>“generate”</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>“propagate”</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>“generate”</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>“generate”</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>“generate”</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ c_{i+1} = g_i + p_i c_i \]
\[ s_i = p_i \oplus c_i \]
Carry Look-ahead Adders

- Ripple adder using p and g signals:

\[
\begin{align*}
  p_{0} & \oplus c_{0} = s_{0} \\
  g_{0} + p_{0}c_{0} & = c_{1} \\
  p_{1} & \oplus c_{1} = s_{1} \\
  g_{1} + p_{1}c_{1} & = c_{2} \\
  p_{2} & \oplus c_{2} = s_{2} \\
  g_{2} + p_{2}c_{2} & = c_{3} \\
  p_{3} & \oplus c_{3} = s_{3} \\
  g_{3} + p_{3}c_{3} & = c_{4}
\end{align*}
\]

- So far, no advantage over ripple adder: \(T \propto N\)

Carry Look-ahead Adders

- Expand carries:

\[
\begin{align*}
  c_{0} & = g_{0} + p_{0}c_{0} \\
  c_{1} & = g_{1} + p_{1}c_{1} = g_{1} + p_{1}g_{0} + p_{1}p_{0}c_{0} \\
  c_{2} & = g_{2} + p_{2}c_{2} = g_{2} + p_{2}g_{1} + p_{2}p_{1}g_{0} + p_{2}p_{1}p_{0}c_{0} \\
  c_{3} & = g_{3} + p_{3}c_{3} = g_{3} + p_{3}g_{2} + p_{3}p_{2}g_{1} + \ldots \\
  \vdots & \\
  \vdots
\end{align*}
\]

- 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:
  \[
  \begin{align*}
  P &= p_i p_{i+1} \ldots p_{i+k} \\
  G &= g_{i+k} + p_{i+k} g_{i+k-1} + \ldots + (p_{i+1} p_{i+2} \ldots p_{i+k}) g_i
  \end{align*}
  \]

- \( P \) 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

\[
C_{\text{out}} = G + PC_{\text{in}}
\]

- Group \( P \) and \( G \) can be generated hierarchically.

9-bit Example of hierarchically generated \( P \) and \( G \) signals:
8-bit Carry Look-ahead Adder

\[ p = a \oplus b \]
\[ g = ab \]
\[ s = p \oplus c_i \]
\[ c_{i+1} = g + c_i p \]

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

\[ P = P_a P_b \]
\[ G = G_a + G_b P_b \]
\[ C_{out} = G + c_{i+1} p \]
Bit-serial Adder

- A, B, and R held in shift-registers.
- Shift right once per clock cycle.
- Reset is asserted by controller.

Addition of 2 n-bit numbers:
- takes n clock cycles,
- uses 1 FF, 1 FA cell, plus registers
- the bit streams may come from or go to other circuits, therefore the registers are optional.

Adders on the Xilinx Virtex

- Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Virtex-E CLB supports two separate carry chains, one per Slice. The height of the carry chains is two bits per CLB.
- The arithmetic logic includes an XOR gate and AND gate that allows a 2-bit full adder to be implemented within a slice.
- Cin to Cout delay = 0.1ns, versus 0.4ns for F to X delay.