Lecture 23: Adders
Announcements

- Homework 9 posted
- 2 more weeks of lecture
- 2 more homework exercises
Outline

- “tricks with trees”
- Adder review, subtraction, carry-select
- Carry-lookahead
- Bit-serial addition, summary
Tricks with Trees
A log(n) lower (time) bound to compute any function of n variables

- Assume we can only use binary operations, each taking unit time
- After 1 time unit, an output can only depend on two inputs
- Use induction to show that after k time units, an output can only depend on $2^k$ inputs
  - After $\log_2 n$ time units, output depends on at most n inputs
- A binary tree performs such a computation
If each node (operator) is k-ary instead of binary, what is the delay?
Trees for optimization

What property of “+” are we exploiting?

Other associate operators? Boolean operations? Division? Min/Max?
Parallel Prefix, or “Scan”

If “+” is an associative operator, and \( x_0, \ldots, x_{p-1} \) are input data then parallel prefix operation computes:

\[
y_j = x_0 + x_1 + \ldots + x_j \quad \text{for } j=0,1,\ldots,p-1
\]

\[
x_0, \ x_0 + x_1, \ x_0 + x_1 + x_2, \ldots
\]
Adder review, subtraction, carry-select
4-bit Adder Example

- Motivate the adder circuit design by hand addition:

  \[
  \begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  + & b_3 & b_2 & b_1 & b_0 \\
  \hline
  c & r_3 & r_2 & r_1 & r_0
  \end{array}
  \]

- Add a0 and b0 as follows:

  \[
  \begin{array}{ccc|c|c}
  a & b & r & c \\
  \hline
  0 & 0 & 0 & 0 \\
  0 & 1 & 1 & 0 \\
  1 & 0 & 1 & 0 \\
  1 & 1 & 0 & 1 \\
  \end{array}
  \]

  \[r = a \oplus b = a \oplus b\]
  \[c = a \land b = ab\]

- Add a1 and b1 as follows:

  \[
  \begin{array}{rrrr|rr}
  a_3 & a_2 & a_1 & a_0 & r_3 & r_2 & r_1 & r_0 \\
  + & b_3 & b_2 & b_1 & b_0 \\
  \hline
  c & r_3 & r_2 & r_1 & r_0
  \end{array}
  \]

  \[
  \begin{array}{c|c|c|c}
  c_i & a & b & r & c_o \\
  \hline
  0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 1 & 0 \\
  0 & 1 & 0 & 1 & 0 \\
  0 & 1 & 1 & 0 & 1 \\
  1 & 0 & 0 & 1 & 0 \\
  1 & 0 & 1 & 0 & 1 \\
  1 & 1 & 0 & 0 & 1 \\
  1 & 1 & 1 & 1 & 1 \\
  \end{array}
  \]

  \[r = a \oplus b \oplus c_i\]
  \[c_o = ab + ac_i + bc_i\]
Carry-ripple Adder Revisited

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

- 4-bit adder:

- What about subtraction?
Subtractor/Adder

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

*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:

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

Ripple adder is inherently slow because, in worst case 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

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

\[ \text{COST} = 1.5 \times \text{COST}_{\text{ripple_adder}} + (n/2 + 1) \times \text{COST}_{\text{MUX}} \]
Carry Select Adder

- Extending Carry-select to multiple blocks

What is the optimal # of blocks and # of bits/block?
- If blocks too small delay dominated by total mux delay
- If blocks too large delay dominated by adder ripple delay

\[ T \propto \sqrt{N}, \quad \text{Cost} \approx 2 \times \text{ripple} + \text{muxes} \]
Compare to ripple adder delay:

\[ T_{\text{total}} = 2 \sqrt{N} \ T_{\text{FA}} - T_{\text{FA}}, \text{ assuming } T_{\text{FA}} = T_{\text{MUX}} \]

For ripple adder \( T_{\text{total}} = N \ T_{\text{FA}} \)

“cross-over” at \( N=3 \), Carry select faster for any value of \( N>3 \).

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 \([12 \ 11 \ 10 \ 9 \ 8 \ 7 \ 7]\), the exact answer depends on the relative delay of mux and FA

\( \text{note: one less block than } \sqrt{N} \text{ solution} \)
Carry-lookahead and Parallel Prefix
How do we arrange carry generation to be associative?

Reformulate basic adder stage:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c_i$</th>
<th>$c_{i+1}$</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 “kill”**
  - $k_i = a_i' b_i$
- **carry “propagate”**
  - $p_i = a_i \oplus b_i$
- **carry “generate”**
  - $g_i = a_i b_i$

$c_{i+1} = g_i + p_i c_i$

$s_i = p_i \oplus c_i$
Ripple adder using p and g signals:

\[ p_i = a_i \oplus b_i \]
\[ g_i = a_i b_i \]

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

- “Group” propagate and generate signals:

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

- \( P \) true if the group as a whole propagates a carry to \( c_{out} \)

- \( G \) true if the group as a whole generates a carry

- \[ c_{out} = G + P c_{in} \]

- Group \( P \) and \( G \) can be generated hierarchically.
Carry Look-ahead Adders

9-bit Example of hierarchically generated $P$ and $G$ signals:

$$P = P_a P_b P_c$$

$$G = G_c + P_c G_b + P_b P_c G_a$$

$$c_0$$

$$c_3 = G_a + P_a c_0$$

$$c_6 = G_b + P_b c_3$$

$$c_9 = G + P c_0$$
8-bit Carry Look-ahead Adder

\[ p = a \oplus b \]
\[ g = ab \]
\[ s = p \oplus c_i \]
\[ c_{i+1} = g + c_i \cdot p \]
8-bit Carry Look-ahead Adder with 2-input gates.

\[ \begin{align*}
  c_0 & \rightarrow p_0 \rightarrow g_0 \\
  c_1 & = g_0 + p_0 c_0 \\
  c_2 & = G_8 + P_8 c_0 \\
  c_3 & = g_2 + p_2 c_2 \\
  c_4 & = G_9 + P_9 c_2 \\
  c_5 & = g_4 + p_4 c_4 \\
  c_6 & = G_5 + P_5 c_4 \\
  c_7 & = g_6 + p_6 c_6 \\
  c_8 & = G_7 + P_7 c_6 \\
  P_8 &= p_0 p_1 \\
  P_9 &= p_2 p_3 \\
  P_a &= p_4 p_5 \\
  P_b &= p_6 p_7 \\
  P_c &= P_8 P_9 \\
  P_e &= P_c P_d \\
  P_d &= P_a P_b \\
  G_8 &= g_1 + p_1 g_0 \\
  G_9 &= g_3 + p_3 g_2 \\
  G_5 &= g_5 + p_5 g_4 \\
  G_7 &= g_7 + p_7 g_6 \\
  G_c &= G_9 + P_9 G_8 \\
  G_e &= G_d + P_d G_c \\
  G_d &= G_b + P_b G_a \\
  G_0 &= c_0 \\
  G_1 &= c_1 \\
  G_2 &= c_2 \\
  G_3 &= c_3 \\
  G_4 &= c_4 \\
  G_5 &= c_5 \\
  G_6 &= c_6 \\
  G_7 &= c_7 \\
  G_8 &= c_8 \\
  G_9 &= c_9
\end{align*} \]
Parallel-Prefix Adders

Can carry generation be made to be a kind of “reduction operation”?

Lowest delay for a reduction is a balanced tree.

- *But in this case all intermediate values are required.*
- *One way is to use “Parallel Prefix” to compute the carries.*

Parallel Prefix requires that the operation be associative, but simple carry generation is not!
Parallel-Prefix Carry Look-ahead Adders

- Ground truth specification of all carries directly (no grouping):
  \[
  c_0 = 0 \\
  c_1 = g_0 + p_0 c_0 = g_0 \\
  c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 \\
  c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_1 p_2 g_0 \\
  c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_4 p_3 p_2 g_0 \\
  \ldots \\
  \]

Binary \((G', P')\) associative operator

Can be used to form all carries!

Use binary \((G,P)\) operator to form parallel prefix tree
Parallel Prefix Adder Example

\[ G = g_3 + g_2 p_3 + (g_1 + g_0 p_1) p_3 p_2 \]
\[ P = p_3 p_2 \]

\[ G = g_2 + g_1 p_2 \]
\[ P = p_2 p_1 \]

\[ G = g_1 + g_0 p_1 \]
\[ P = p_1 p_0 \]

\[ G = g_3 + g_2 p_3 + g_1 p_3 p_2 + g_0 p_3 p_2 p_1 \]
\[ = c_3 \]

\[ s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i \]
Other Parallel Prefix Adder Architectures

**Kogge-Stone adder**: minimum logic depth, and full binary tree with minimum fan-out, resulting in a fast adder but with a large area.

**Ladner-Fischer adder**: minimum logic depth, large fan-out requirement up to n/2.

**Brent-Kung adder**: minimum area, but high logic depth.

**Han-Carlson adder**: hybrid design combining stages from the Brent-Kung and Kogge-Stone adder.
Carry look-ahead Wrap-up

- Adder delay $O(\log N)$.
- Cost?
- Can be applied with other techniques. Group P & G signals can be generated for sub-adders, but another carry propagation technique (for instance ripple) used within the group.
  - For instance on FPGA. Ripple carry up to 32 bits is fast, CLA used to extend to large adders. CLA tree quickly generates carry-in for upper blocks.
Bit-serial Addition, Adder summary
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 might not be needed.
Adders on FPGAs

- Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions.
- On Virtex-5
  - Cin to Cout (per bit) delay = 40ps, versus 900ps for F to X delay.
  - 64-bit add delay = 2.5ns.
Adder Final Words

- Dynamic energy per addition for all of these is $O(n)$.
- “$O$” notation hides the constants. Watch out for this!
- The “real” cost of the carry-select is at least 2X the “real” cost of the ripple. “Real” cost of the CLA is probably at least 2X the “real” cost of the carry-select.
- The actual multiplicative constants depend on the implementation details and technology.
- FPGA and ASIC synthesis tools will try to choose the best adder architecture automatically - assuming you specify addition using the “+” operator, as in “assign A = B + C”

<table>
<thead>
<tr>
<th>Type</th>
<th>Cost</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripple</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Carry-select</td>
<td>$O(N)$</td>
<td>$O(\sqrt{N})$</td>
</tr>
<tr>
<td>Carry-lookahead</td>
<td>$O(N)$</td>
<td>$O(\log(N))$</td>
</tr>
<tr>
<td>Bit-serial</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>