Recap

- clocks and timing in Xilinx

- Carry Select Adder
Outline

• Carry Look-ahead Adder- Can we speed up addition by being clever with carry?

• How can we multiply quickly?

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 delay

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

\[ T \sim \text{sqrt}(N), \quad \text{Cost} \equiv 2 \text{ripple + muxes} \]
Carry Select Adder

- 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]

Carry Look-ahead Adders

- In general, for n-bit addition best we can achieve is delay \( \sim \log(n) \)
- How do we arrange this? (think trees)
- First, reformulate basic adder stage:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c_i</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>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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>1</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” \( p_i = a_i \ XOR \ b_i \)
  - carry “generate” \( g_i = a_i \ b_i' \)

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

- Ripple adder using p and g signals:

  \[ p_i = a_i \oplus b_i \]

  \[ g_i = a_i \cdot b_i \]

  \[ c_0 = g_0 = p_0 \]

  \[ s_0 = p_0 \oplus c_0 \]

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

Carry Look-ahead Adders

- Expand carries:

  \[ c_0 = g_0 + p_0 \]

  \[ c_1 = g_1 + p_1 c_0 \]

  \[ c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_0 + p_2 p_0 c_0 \]

  \[ c_3 = g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_0 + p_3 p_2 p_0 c_0 \]

  \[ c_4 = g_4 + p_4 c_3 = g_4 + p_4 g_3 + p_4 g_3 g_2 + p_4 g_3 p_2 g_0 + p_4 g_3 p_2 p_0 c_0 \]

  \[ \ldots \]

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

  \[ s_i = p_i \oplus c_i \]

- 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:

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

- \( P \) 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 \( c_{\text{out}} = G + P_{\text{in}} \)
- Group \( P \) and \( G \) can be generated hierarchically.

9-bit Example of hierarchically generated \( P \) and \( G \) signals:

\[
\begin{align*}
P &= P_a P_b P_c \\
G &= G_c + P_c G_b + P_b P_c G_a + P_c G_c
\end{align*}
\]
\[
p = a \text{ XOR } b \\
g = ab \\
s = p \text{ XOR } c_i \\
c_{i+1} = g + c_i p
\]
Carry look-ahead Wrap-up

- Adder delay $O(\log N)$ (up then down the tree).
- 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 (1.25ns), CLA used to extend to large adders. CLA tree quickly generates carry-in for upper blocks.
- Other more complex techniques exist that can bring the delay down below $O(\log N)$, but are only efficient for very wide adders.

Adders on the Xilinx Virtex-5

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

We can map ripple-carry addition onto carry-chain block.

\[ p_i = a_i \text{ XOR } b_i \]

\[ g_i = a_i \text{ b}_i \]

The carry-chain block also useful for speeding up other adder structures and counters.

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

\[ s_i = p_i \text{ XOR } c_i \]

**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 might not be needed.
Adder Final Words

<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>
</tbody>
</table>

- 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"

### Multiplication

```
  a_3 a_2 a_1 a_0 ← Multiplicand
  b_3 b_2 b_1 b_0 ← Multiplier

  X
  a_3b_0 a_2b_0 a_1b_0 a_0b_0
  a_3b_1 a_2b_1 a_1b_1 a_0b_1
  a_3b_2 a_2b_2 a_1b_2 a_0b_2
  a_3b_3 a_2b_3 a_1b_3 a_0b_3

  . . .

  ... a_1b_0 + a_0b_1 a_0b_0 ← Product
```

Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (cost).
**“Shift and Add” Multiplier**

- Sums each partial product, one at a time.
- In binary, each partial product is shifted versions of A or 0.

**Control Algorithm:**
1. \( P \leftarrow 0, A \leftarrow \) multiplicand, \( B \leftarrow \) multiplier
2. If LSB of B==1 then add A to P else add 0
3. Shift [P][B] right 1
4. Repeat steps 2 and 3 \( n-1 \) times.
5. [P][B] has product.

- Cost \( \sim n, T = n \) clock cycles.
- What is the critical path for determining the min clock period?

**Signed Multiplication:**

*Remember for 2’s complement numbers MSB has negative weight:*

\[
X = \sum_{i=0}^{N-2} x_i 2^i - x_{n-1} 2^{n-1}
\]

ex: \(-6 = 11010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4\)

\[
= 0 + 2 + 0 + 8 - 16 = -6
\]

- Therefore for multiplication:
  a) subtract final partial product
  b) sign-extend partial products
- Modifications to shift & add circuit:
  a) adder/subtractor
  b) sign-extender on P shifter register
Bit-serial Multiplier

- Bit-serial multiplier ($n^2$ cycles, one bit of result per $n$ cycles):

  \[ \text{Control Algorithm:} \]

  \[
  \text{repeat n cycles} \{ \quad \text{// outer (i) loop} \\
  \text{\hspace{1em} repeat n cycles} \{ \quad \text{// inner (j) loop} \\
  \text{\hspace{2em} shiftA, selectSum, shiftHI} \\
  \text{\hspace{2em} } \}
  \]

  \[
  \text{\hspace{1em} shiftB, shiftHI, shiftLOW, reset} \\
  \text{\hspace{1em} } \}
  \]

  \[ \text{Note: The occurrence of a control signal x means x=1. The absence of x means x=0.} \]

Array Multiplier

Single cycle multiply: Generates all $n$ partial products simultaneously.
Carry-Save Addition

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- “Carry-save” addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

Example: sum three numbers, $3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

\[
\begin{array}{c}
\text{carry in} \\
0000 \\
0111 \\
0010 \\
\hline
\text{carry-save add} \\
0010 \\
0001 \\
0011 \\
\hline
\text{carry-propagate add} \\
0010 \\
0110 \\
1000 \\
\hline
4_{10} = 0100 \\
1_{10} = 0001 \\
8_{10} = 1000
\end{array}
\]

- In general, carry-save addition takes in 3 numbers and produces 2.
- Whereas, carry-propagate takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition.

Carry-save Circuits

- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and cheap (same cost as ripple adder)
Array Multiplier using Carry-save Addition

CSA is associative and commutative. For example:

\[((X_0 + X_1) + X_2) + X_3\) = \((X_0 + X_1) + (X_2 + X_3)\)

- A balanced tree can be used to reduce the logic delay.
- This structure is the basis of the **Wallace Tree Multiplier**.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.
- Multiplier delay \(< \log_{3/2}N + \log_2N\)
Constant Multiplication

- Our discussion so far has assumed both the multiplicand (A) and the multiplier (B) can vary at runtime.
- What if one of the two is a constant?
  \[ Y = C \times X \]
- “Constant Coefficient” multiplication comes up often in signal processing and other hardware. Ex:
  \[ y_i = \alpha y_{i-1} + x_i \]
  where \( \alpha \) is an application dependent constant that is hard-wired into the circuit.

- How do we build and array style (combinational) multiplier that takes advantage of the constancy of one of the operands?

Multiplication by a Constant

- If the constant C in C*X is a power of 2, then the multiplication is simply a shift of X.
  - Ex: 4*X

- What about division?
- What about multiplication by non-powers of 2?
Multiplication by a Constant

• In general, a combination of fixed shifts and addition:
  – Ex: $6 \times X = 0110 \times X = (2^2 + 2^1) \times X$

- Details:

+ Another example: $C = 23_{10} = 010111$

- In general, the number of additions equals the number of 1’s in the constant minus one.

- Using carry-save adders (for all but one of these) helps reduce the delay and cost, but the number of adders is still the number of 1’s in C minus 2.

- Is there a way to further reduce the number of adders (and thus the cost and delay)?
Multiplication using Subtraction

- *Subtraction is ~ the same cost and delay as addition.*
- Consider C*X where C is the constant value $15_{10} = 01111$. C*X requires 3 additions.
- We can “recode” 15
  
  From $01111 = (2^3 + 2^2 + 2^1 + 2^0)$
  
  to $1000\bar{1} = (2^4 - 2^0)$

  where $\bar{1}$ means negative weight.
- Therefore, 15*X can be implemented with only one subtractor.

Canonic Signed Digit Representation

- CSD represents numbers using 1, $\bar{1}$, & 0 with the least possible number of non-zero digits.
  - Strings of 2 or more non-zero digits are replaced.
  - Leads to a unique representation.
- To form CSD representation might take 2 passes:
  - First pass: replace all occurrences of 2 or more 1’s:
    
    $01..10$ by $10..10$
  - Second pass: same as a above, plus replace $0110$ by $0010$
- Examples:
  
  $011101 = 29$
  $100\bar{1}01 = 32 - 4 + 1$
  $0010111 = 23$
  $0011001\bar{1} = 10\bar{1}0\bar{1}0$
  $0110110 = 54$
  $1011010 = 64 - 8 - 2$
  $0101001\bar{1} = 32 - 8 - 1$
  $100\bar{1}0\bar{1}0\bar{1} = 64 - 8 - 2$

- Can we further simplify the multiplier circuits?
“Constant Coefficient Multiplication” (KCM)

Binary multiplier: \( Y = 231 \times X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0) \times X \)

- CSD helps, but the multipliers are limited to shifts followed by adds.
  - CSD multiplier: \( Y = 231 \times X = (2^8 - 2^5 + 2^3 - 2^0) \times X \)

- How about shift/add/shift/add …?
  - KCM multiplier: \( Y = 231 \times X = 7 \times 33 \times X = (2^3 - 2^0)(2^5 + 2^0) \times X \)

- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.

Summary

- Carry Look-ahead Adder

  ![Carry Look-ahead Adder Diagram]

  constant coefficient multiplication

  ![Constant Coefficient Multiplication Diagram]

- Carry save adder

  ![Carry save adder Diagram]

  \[ \log_{3/2} N \]

  \[ \log_2 N \]