Dr Milunka Damnjanović, red.prof,

Projektovanje VLSI

Sabirači
Addition / Subtraction

Review addition schemes and various speedup methods
- Addition is a key op (in itself, and as a building block)
- Subtraction = negation + addition
- Carry propagation speedup: lookahead, skip, select, …
- Two-operand versus multioperand addition

Topics in This Part

| Basic Addition and Counting
| Carry-Lookahead Adders
| Variations in Fast Adder
| Multioperand Addition |
Chapter Goals

Study the design of ripple-carry adders, discuss why their latency is unacceptable, and set the foundation for faster adders.

Chapter Highlights

Full adders are versatile building blocks
Longest carry chain on average: $\log_2 k$ bits
Fast asynchronous adders are simple
Counting is relatively easy to speed up
### Basic Addition and Counting: Topics

#### Topics in This Chapter

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bit-Serial and Ripple-Carry Adders</td>
</tr>
<tr>
<td>2</td>
<td>Conditions and Exceptions</td>
</tr>
<tr>
<td>3</td>
<td>Analysis of Carry Propagation</td>
</tr>
<tr>
<td>4</td>
<td>Carry Completion Detection</td>
</tr>
<tr>
<td>5</td>
<td>Addition of a Constant</td>
</tr>
<tr>
<td>6</td>
<td>Manchester Carry Chains and Adders</td>
</tr>
</tbody>
</table>
1 Bit-Serial and Ripple-Carry Adders

Half-adder (HA): Truth table and block diagram

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$y$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Full-adder (FA): Truth table and block diagram

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$y$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Half-Adder Implementations

(a) AND/XOR half-adder.

(b) NOR-gate half-adder.

(c) NAND-gate half-adder with complemented carry.
Possible designs for a full-adder in terms of half-adders, logic gates, and CMOS transmission gates.
(alternate version) Possible designs for a full-adder in terms of half-adders, logic gates, and CMOS transmission gates.
Some Full-Adder Details

Logic equations for a full-adder:
\[
\begin{align*}
    s &= x \oplus y \oplus c_{in} \\
    &= xy c_{in} \lor x'y'c_{in} \lor x'y c_{in}' \lor xy'c_{in}' \\
    c_{out} &= xy \lor x c_{in} \lor y c_{in}
\end{align*}
\]

(a) CMOS transmission gate: circuit and symbol
(b) Two-input mux built of two transmission gates

CMOS transmission gate and its use in a 2-to-1 mux.
Simple Adders Built of Full-Adders

Using full-adders in building bit-serial and ripple-carry adders.

(a) Bit-serial adder.

(b) Ripple-carry adder.
The layout of a 4-bit ripple-carry adder in CMOS implementation [Puck94].
Critical Path Through a Ripple-Carry Adder

\[ T_{\text{ripple-add}} = T_{\text{FA}}(x, y \rightarrow c_{\text{out}}) + (k - 2) \times T_{\text{FA}}(c_{\text{in}} \rightarrow c_{\text{out}}) + T_{\text{FA}}(c_{\text{in}} \rightarrow s) \]

Critical path in a k-bit ripple-carry adder.
Binary Adders as Versatile Building Blocks

Set one input to 0: \( c_{\text{out}} = \text{AND of other inputs} \)
Set one input to 1: \( c_{\text{out}} = \text{OR of other inputs} \)
Set one input to 0 and another to 1: \( s = \text{NOT of third input} \)

Four-bit binary adder used to realize the logic function \( f = w + xyz \) and its complement.
Two's-complement adder with provisions for detecting conditions and exceptions.

\[
\text{overflow}_{2's\text{-compl}} = x_{k-1} y_{k-1} s_{k-1}' \lor x_{k-1}' y_{k-1}' s_{k-1}
\]

\[
\text{overflow}_{2's\text{-compl}} = c_k \oplus c_{k-1} = c_k c_{k-1}' \lor c_k' c_{k-1}
\]
Saturating Adders

Saturating (saturation) arithmetic:

When a result’s magnitude is too large, do not wrap around; rather, provide the most positive or the most negative value that is representable in the number format.

**Example** – In 8-bit 2’s-complement format, we have:
120 + 26 → 18 (wraparound); 120 + \text{sat} 26 → 127 (saturating)

Saturating arithmetic in desirable in many DSP applications

Designing saturating adders

Unsigned (quite easy)

Signed (only slightly harder)
3 Analysis of Carry Propagation

Bit positions

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- -----
Using Probability to Analyze Carry Propagation

Given binary numbers with random bits, for each position $i$ we have

- Probability of carry generation = $\frac{1}{4}$ (both 1s)
- Probability of carry annihilation = $\frac{1}{4}$ (both 0s)
- Probability of carry propagation = $\frac{1}{2}$ (different)

Probability that carry generated at position $i$ propagates through position $j - 1$ and stops at position $j$ ($j > i$)

$$2^{-(j-1-i)} \times \frac{1}{2} = 2^{-(j-i)}$$

Expected length of the carry chain that starts at position $i$

$$2 - 2^{-(k-i-1)}$$

Average length of the longest carry chain in $k$-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

**Analogy:** Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows
4 Carry Completion Detection

$x_i \ y_i = x_i + y_i$

$b_k \ldots b_{i+1}$
$c_k = c_{out} \ldots c_{i+1}$

$x_i \ y_i$
$x_i + y_i$

$b_i \ c_i$

0 0 Carry not yet known
0 1 Carry known to be 1
1 0 Carry known to be 0

From other bit positions

alldone

Example addition and its carry propagation chains.
5 Addition of a Constant: Counters

An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.
Implementing a Simple Up Counter

(Fm arch text) Ripple-carry incrementer for use in an up counter.

Four-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.
Faster and Constant-Time Counters

Any fast adder design can be specialized and optimized to yield a fast counter (carry-lookahead, carry-skip, etc.)

One can use redundant representation to build a constant-time counter, but a conversion penalty must be paid during read-out

Fast (constant-time) three-stage up counter.
6 Manchester Carry Chains and Adders

Sum digit in radix $r$

$$s_i = (x_i + y_i + c_i) \mod r$$

Special case of radix 2

$$s_i = x_i \oplus y_i \oplus c_i$$

Computing the carries $c_i$ is thus our central problem. For this, the actual operand digits are not important. What matters is whether in a given position a carry is generated, propagated, or annihilated (absorbed).

For binary addition:

$$g_i = x_i \cdot y_i$$
$$p_i = x_i \oplus y_i$$
$$a_i = x'_i \cdot y'_i = (x_i \lor y_i)'$$

It is also helpful to define a transfer signal:

$$t_i = g_i \lor p_i = a'_i = x_i \lor y_i$$

Using these signals, the carry recurrence is written as

$$c_{i+1} = g_i \lor c_i \cdot p_i = g_i \lor c_i \cdot g_i \lor c_i \cdot p_i = g_i \lor c_i \cdot t_i$$
Manchester Carry Network

The worst-case delay of a Manchester carry chain has three components:
1. Latency of forming the switch control signals
2. Set-up time for switches
3. Signal propagation delay through $k$ switches

(a) Conceptual representation

(b) Possible CMOS realization.

Fig. 5.13 One stage in a Manchester carry chain.
The main part of an adder is the carry network. The rest is just a set of gates to produce the $g$ and $p$ signals and the sum bits.
Chapter Goals

Understand the carry-lookahead method and its many variations used in the design of fast adders

Chapter Highlights

Single- and multilevel carry lookahead
Various designs for log-time adders
Relating the carry determination problem to parallel prefix computation
Implementing fast adders in VLSI
## Carry-Lookahead Adders: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Unrolling the Carry Recurrence</td>
</tr>
<tr>
<td>2. Carry-Lookahead Adder Design</td>
</tr>
<tr>
<td>3. Ling Adder and Related Designs</td>
</tr>
<tr>
<td>4. Carry Determination as Prefix Computation</td>
</tr>
<tr>
<td>5. Alternative Parallel Prefix Networks</td>
</tr>
<tr>
<td>6. VLSI Implementation Aspects</td>
</tr>
</tbody>
</table>
1 Unrolling the Carry Recurrence

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Radix $r$ [\text{is 1 iff } x_i + y_i \geq r]</th>
<th>Binary [x_i y_i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_i$</td>
<td>$g_{i-1} \lor c_{i-1} p_{i-1}$</td>
<td></td>
</tr>
<tr>
<td>$p_i$</td>
<td>$g_{i-1} \lor (g_{i-2} \lor c_{i-2} p_{i-2}) p_{i-1}$</td>
<td>$x_i \oplus y_i$</td>
</tr>
<tr>
<td>$a_i$</td>
<td>$g_{i-1} \lor g_{i-2} p_{i-1} \lor c_{i-2} p_{i-2}$</td>
<td>$x_i'y_i' = (x_i \lor y_i)'$</td>
</tr>
<tr>
<td>$t_i$</td>
<td>$g_{i-1} \lor g_{i-2} p_{i-1} \lor g_{i-3} p_{i-2} p_{i-1} \lor c_{i-3} p_{i-3} p_{i-2} p_{i-1}$</td>
<td>$x_i \lor y_i$</td>
</tr>
<tr>
<td>$s_i$</td>
<td>$(x_i + y_i + c_i) \mod r$ $x_i \oplus y_i \oplus c_i$</td>
<td></td>
</tr>
</tbody>
</table>

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation.

\[
c_i = g_{i-1} \lor c_{i-1} p_{i-1} = g_{i-1} \lor (g_{i-2} \lor c_{i-2} p_{i-2}) p_{i-1} = g_{i-1} \lor g_{i-2} p_{i-1} \lor c_{i-2} p_{i-2} = g_{i-1} \lor g_{i-2} p_{i-1} \lor g_{i-3} p_{i-2} p_{i-1} \lor c_{i-3} p_{i-3} p_{i-2} p_{i-1} = g_{i-1} \lor g_{i-2} p_{i-1} \lor g_{i-3} p_{i-2} p_{i-1} \lor g_{i-4} p_{i-3} p_{i-2} p_{i-1} \lor c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} = \ldots
\]
Theoretically, it is possible to derive each sum digit directly from the inputs that affect it.

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits.
Four-Bit Carry-Lookahead Adder

Complexity reduced by deriving the carry-out indirectly

Full carry lookahead is quite practical for a 4-bit adder

\[
\begin{align*}
c_1 &= g_0 \lor c_0 p_0 \\
c_2 &= g_1 \lor g_0 p_1 \lor c_0 p_0 p_1 \\
c_3 &= g_2 \lor g_1 p_2 \lor g_0 p_1 p_2 \lor c_0 p_0 p_1 p_2 \\
c_4 &= g_3 \lor g_2 p_3 \lor g_1 p_2 p_3 \lor g_0 p_1 p_2 p_3 \lor c_0 p_0 p_1 p_2 p_3
\end{align*}
\]

Four-bit carry network with full lookahead.
Consider a 32-bit adder

\[ c_1 = g_0 \lor c_0 p_0 \]
\[ c_2 = g_1 \lor g_0 p_1 \lor c_0 p_0 p_1 \]
\[ c_3 = g_2 \lor g_1 p_2 \lor g_0 p_1 p_2 \lor c_0 p_0 p_1 p_2 \]
\[ \ldots \]
\[ c_{31} = g_{30} \lor g_{29} p_{30} \lor g_{28} p_{29} p_{30} \lor g_{27} p_{28} p_{29} p_{30} \lor \ldots \lor c_0 p_0 p_1 p_2 p_3 \ldots p_{29} p_{30} \]

High fan-ins necessitate tree-structured circuits
Two Solutions to the Fan-in Problem

High-radix addition (i.e., radix $2^h$)

- Increases the latency for generating $g$ and $p$ signals and sum digits, but simplifies the carry network (optimal radix?)

Multilevel lookahead

Example: 16-bit addition

- Radix-16 (four digits)
- Two-level carry lookahead (four 4-bit blocks)

Either way, the carries $c_4$, $c_8$, and $c_{12}$ are determined first

\[
\begin{array}{cccccccccccccccc}
C_{16} & C_{15} & C_{14} & C_{13} & C_{12} & C_{11} & C_{10} & C_9 & C_8 & C_7 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 \\
\end{array}
\]
2 Carry-Lookahead Adder Design

Block *generate* and *propagate* signals

\[
g_{[i,i+3]} = g_{i+3} \lor g_{i+2}p_{i+3} \lor g_{i+1}p_{i+2}p_{i+3} \lor g_i p_{i+1}p_{i+2}p_{i+3}
\]

\[
p_{[i,i+3]} = p_i p_{i+1}p_{i+2}p_{i+3}
\]

Schematic diagram of a 4-bit lookahead carry generator.

Schematic diagram of a 4-bit lookahead carry generator.
A Building Block for Carry-Lookahead Addition

Four-bit lookahead carry generator.

Four-bit adder

Block Signal Generation
Intermediate Carries
Combining Block $g$ and $p$ Signals

Block *generate* and *propagate* signals can be combined in the same way as bit $g$ and $p$ signals to form $g$ and $p$ signals for wider blocks.

Combining of $g$ and $p$ signals of four (contiguous or overlapping) blocks of arbitrary widths into the $g$ and $p$ signals for the overall block $[i_0, j_3]$. 
A Two-Level Carry-Lookahead Adder

Building a 64-bit carry-lookahead adder from 16 4-bit adders and 5 lookahead carry generators.

Carry-out: \[ c_{\text{out}} = g_{[0,k-1]} \lor c_0 p_{[0,k-1]} = x_{k-1} y_{k-1} \lor s_{k-1}' (x_{k-1} \lor y_{k-1}) \]
Latency of a Multilevel Carry-Lookahead Adder

Latency through the 16-bit CLA adder consists of finding:

- \( g \) and \( p \) for individual bit positions: 1 gate level
- \( g \) and \( p \) signals for 4-bit blocks: 2 gate levels
- Block carry-in signals \( c_4, c_8, \) and \( c_{12} \): 2 gate levels
- Internal carries within 4-bit blocks: 2 gate levels
- Sum bits: 2 gate levels

Total latency for the 16-bit adder: 9 gate levels

(compare to 32 gate levels for a 16-bit ripple-carry adder)

Each additional lookahead level adds 4 gate levels of latency

Latency for \( k \)-bit CLA adder: \( T_{\text{lookahead-add}} = 4 \log_4 k + 1 \) gate levels
3 Ling Adder and Related Designs

Consider the carry recurrence and its unrolling by 4 steps:
\[
c_i = g_{i-1} \lor c_{i-1} t_{i-1}
\]
\[
= g_{i-1} \lor g_{i-2} t_{i-1} \lor g_{i-3} t_{i-2} t_{i-1} \lor g_{i-4} t_{i-3} t_{i-2} t_{i-1} \lor c_{i-4} t_{i-3} t_{i-2} t_{i-1}
\]

Ling’s modification: Propagate \( h_i = c_i + c_{i-1} \) instead of \( c_i \)
\[
h_i = g_{i-1} \lor h_{i-1} t_{i-2}
\]
\[
= g_{i-1} \lor g_{i-2} \lor g_{i-3} t_{i-2} \lor g_{i-4} t_{i-3} t_{i-2} \lor h_{i-4} t_{i-3} t_{i-2}
\]

CLA: 5 gates max 5 inputs 19 gate inputs
Ling: 4 gates max 5 inputs 14 gate inputs

The advantage of \( h_i \) over \( c_i \) is even greater with wired-OR:
CLA: 4 gates max 5 inputs 14 gate inputs
Ling: 3 gates max 4 inputs 9 gate inputs

Once \( h_i \) is known, however, the sum is obtained by a slightly more complex expression compared to \( s_i = p_i \oplus c_i \)
\[
s_i = (t_i \oplus h_{i+1}) \lor h_i g_i t_{i-1}
\]
Combining of $g$ and $p$ signals of two (contiguous or overlapping) blocks $B'$ and $B''$ of arbitrary widths into the $g$ and $p$ signals for block $B$. 
Formulating the Prefix Computation Problem

The problem of carry determination can be formulated as:

Given \((g_0, p_0), (g_1, p_1), \ldots, (g_{k-2}, p_{k-2}), (g_{k-1}, p_{k-1})\)
Find \((g_{[0,0]}, p_{[0,0]}), (g_{[0,1]}, p_{[0,1]}), \ldots, (g_{[0,k-2]}, p_{[0,k-2]}), (g_{[0,k-1]}, p_{[0,k-1]})\)

\[c_1 c_2 \ldots c_{k-1} c_k\]

Carry-in can be viewed as an extra \((-1)\) position: \((g_{-1}, p_{-1}) = (c_{\text{in}}, 0)\)

The desired pairs are found by evaluating all prefixes of
\[
(g_0, p_0) \triangleleft (g_1, p_1) \triangleleft \ldots \triangleleft (g_{k-2}, p_{k-2}) \triangleleft (g_{k-1}, p_{k-1})
\]

The carry operator \(\triangleleft\) is associative, but not commutative
\[
[(g_1, p_1) \triangleleft (g_2, p_2)] \triangleleft (g_3, p_3) = (g_1, p_1) \triangleleft [(g_2, p_2) \triangleleft (g_3, p_3)]
\]

Prefix sums analogy:

Given \(x_0, x_1, x_2, \ldots, x_{k-1}\)
Find \(x_0, x_0+x_1, x_0+x_1+x_2, \ldots, x_0+x_1+\ldots+x_{k-1}\)
Example Prefix-Based Carry Network

Four-input prefix sums network

Scan order

Four-bit Carry lookahead network

\[ g_0, p_0 = (c_1, --) \]
\[ g_1, p_1 = (c_2, --) \]
\[ g_2, p_2 = (c_3, --) \]
\[ g_3, p_3 = (c_4, --) \]
5 Alternative Parallel Prefix Networks

Parallel prefix sums network built of two $k/2$-input networks and $k/2$ adders. (Ladner-Fischer)

Delay recurrence \( D(k) = D(k/2) + 1 = \log_2 k \)

Cost recurrence \( C(k) = 2C(k/2) + k/2 = (k/2) \log_2 k \)
The Brent-Kung Recursive Construction

\[ D(k) = D(k/2) + 2 = 2 \log_2 k - 1 \quad (\text{–2 really}) \]

\[ C(k) = C(k/2) + k - 1 = 2k - 2 - \log_2 k \]

Parallel prefix sums network built of one \( k/2 \)-input network and \( k - 1 \) adders.
Brent-Kung Carry Network (8-Bit Adder)
Brent-Kung Carry Network (16-Bit Adder)

Reason for latency being $2 \log_2 k - 1$

Brent-Kung parallel prefix graph for 16 inputs.
Kogge-Stone Carry Network (16-Bit Adder)

Cost formula
\[ C(k) = (k - 1) + (k - 2) + (k - 4) + \ldots + (k - k/2) = k \log_2 k - k + 1 \]

\[ \log_2 k \] levels (minimum possible)

Kogge-Stone parallel prefix graph for 16 inputs.
## Speed-Cost Tradeoffs in Carry Networks

<table>
<thead>
<tr>
<th>Method</th>
<th>Delay</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ladner-Fischer</td>
<td>$\log_2 k$</td>
<td>$(k/2) \log_2 k$</td>
</tr>
<tr>
<td>Kogge-Stone</td>
<td>$\log_2 k$</td>
<td>$k \log_2 k – k + 1$</td>
</tr>
<tr>
<td>Brent-Kung</td>
<td>$2 \log_2 k – 2$</td>
<td>$2k – 2 – \log_2 k$</td>
</tr>
</tbody>
</table>

Improving the Ladner/Fischer design

These outputs can be produced one time unit later without increasing the overall latency.

This strategy saves enough to make the overall cost linear (best possible).
Hybrid B-K/K-S Carry Network (16-Bit Adder)

A Hybrid
Brent-Kung/
Kogge-Stone
parallel prefix
graph for
16 inputs.
Example: Radix-256 addition of 56-bit numbers as implemented in the AMD Am29050 CMOS micro

Our description is based on the 64-bit version of the adder

In radix-256, 64-bit addition, only these carries are needed:

\[ c_{56} \quad c_{48} \quad c_{40} \quad c_{32} \quad c_{24} \quad c_{16} \quad c_{8} \]

First, 4-bit Manchester carry chains (MCCs) of Fig. 6.12a are used to derive \( g \) and \( p \) signals for 4-bit blocks

Next, the \( g \) and \( p \) signals for 4-bit blocks are combined to form the desired carries, using the MCCs in Fig. 6.12b
Four-Bit Manchester Carry Chains

Example four-bit Manchester carry chain designs in CMOS technology [Lync92].
Spanning-tree carry-lookahead network [Lync92]. The 16 MCCs at level 1, that produce generate and propagate signals for 4-bit blocks, are not shown.
Chapter Goals
Study alternatives to the carry-lookahead method for designing fast adders

Chapter Highlights
Many methods besides CLA are available (both competing and complementary)
Best design is technology-dependent (often hybrid rather than pure)
Knowledge of timing allows optimizations
Variations in Fast Adders: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simple Carry-Skip Adders</td>
</tr>
<tr>
<td>2. Multilevel Carry-Skip Adders</td>
</tr>
<tr>
<td>3. Carry-Select Adders</td>
</tr>
<tr>
<td>4. Conditional-Sum Adder</td>
</tr>
<tr>
<td>5. Hybrid Adder Designs</td>
</tr>
<tr>
<td>6. Optimizations in Fast Adders</td>
</tr>
</tbody>
</table>
1 Simple Carry-Skip Adders

Converting a 16-bit ripple-carry adder into a simple carry-skip adder with 4-bit skip blocks.

(a) Ripple-carry adder.

(b) Simple carry-skip adder.
Another View of Carry-Skip Addition

Street/freeway analogy for carry-skip adder.
Carry-Skip Adder with Fixed Block Size

Block width $b$; $k/b$ blocks to form a $k$-bit adder (assume $b$ divides $k$)

$$T_{\text{fixed-skip-add}} = (b - 1) + 0.5 + \frac{k}{b} - 2 + (b - 1)$$

in block 0 OR gate skips in last block

$$\approx 2b + k/b - 3.5 \text{ stages}$$

$$dT/db = 2 - k/b^2 = 0 \quad \Rightarrow \quad b^{opt} = \sqrt{k/2}$$

$$T^{opt} = 2\sqrt{2k} - 3.5$$

Example: $k = 32$, $b^{opt} = 4$, $T^{opt} = 12.5$ stages
(contrast with 32 stages for a ripple-carry adder)
Carry-Skip Adder with Variable-Width Blocks

The total number of bits in the $t$ blocks is $k$:

$$2[b + (b + 1) + \ldots + (b + t/2 - 1)] = t(b + t/4 - 1/2) = k$$

$$b = \frac{k}{t} - \frac{t}{4} + 1/2$$

$$T_{\text{var-skip-add}} = 2(b - 1) + 0.5 + t - 2 = 2k/t + t/2 - 2.5$$

$$\frac{dT}{db} = -2k/t^2 + 1/2 = 0 \quad \Rightarrow \quad t^{\text{opt}} = 2\sqrt{k}$$

$$T^{\text{opt}} = 2\sqrt{k} - 2.5 \quad \text{(a factor of } \sqrt{2} \text{ smaller than for fixed-block)}$$
2 Multilevel Carry-Skip Adders

Schematic diagram of a one-level carry-skip adder.

Example of a two-level carry-skip adder.

Two-level carry-skip adder optimized by removing the short-block skip circuits.
Designing a Single-Level Carry-Skip Adder

Example 1

Each of the following takes one unit of time: generation of $g_i$ and $p_i$, generation of level-$i$ skip signal from level-$(i-1)$ skip signals, ripple, skip, and formation of sum bit once the incoming carry is known.

Build the widest possible one-level carry-skip adder with total delay of 8 units.

Fig. 7.6 Timing constraints of a single-level carry-skip adder with a delay of 8 units.

Max adder width = 18

$(1 + 2 + 3 + 4 + 4 + 3 + 1)$

Generalization of Example 7.1 for total time $T$ (even or odd)

1 2 3 . . . $T/2$ $T/2$ . . . 4 3 1
1 2 3 . . . $(T + 1)/2$ . . . 4 3 1

Thus, for any $T$, the total width is $\lceil(T + 1)2/4\rceil - 2$
Designing a Two-Level Carry-Skip Adder

Example 2

Each of the following takes one unit of time: generation of \( g_i \) and \( p_i \), generation of level-\( i \) skip signal from level-(\( i-1 \)) skip signals, ripple, skip, and formation of sum bit once the incoming carry is known.

Build the widest possible two-level carry-skip adder with total delay of 8

Max adder width = 30

\[(4 + 8 + 8 + 6 + 3 + 1)\]

Two-level carry-skip adder with a delay of 8 units: (a) Initial timing constraints, (b) Final design.
Elaboration on Two-Level Carry-Skip Adder

Example 2

Given the delay pair \( \{ \beta, \alpha \} \) for a level-2 block in Fig. 7.7a, the number of level-1 blocks that can be accommodated is \( \gamma = \min(\beta - 1, \alpha) \)

Single-level carry-skip adder with \( T_{\text{assimilate}} = \alpha \)

Single-level carry-skip adder with \( T_{\text{produce}} = \beta \)

Width of the \( i \)th level-1 block in the level-2 block characterized by \( \{ \beta, \alpha \} \) is \( b_i = \min(\beta - \gamma + i + 1, \alpha - i) \); the total block width is then \( \sum_{i=0}^{\gamma-1} b_i \)
Carry-Skip Adder Optimization Scheme

Block of $b$ full-adder units

Generalized delay model for carry-skip adders.
3 Carry-Select Adders

Carry-select adder for $k$-bit numbers built from three $k/2$-bit adders.

\[
C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1
\]

\[
T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1
\]
Multilevel Carry-Select Adders

Two-level carry-select adder built of $k/4$-bit adders.
### 4 Conditional-Sum Adder

Multilevel carry-select idea carried out to the extreme (to 1-bit blocks.

\[
C(k) \equiv 2C(k/2) + k + 2 \equiv k \log_2 k + 2 + k C(1)
\]

\[
T(k) = T(k/2) + 1 = \log_2 k + T(1)
\]

where \( C(1) \) and \( T(1) \) are the cost and delay of the circuit of Fig. 7.11 for deriving the sum and carry bits with a carry-in of 0 and 1.

\[ k + 2 \] is an upper bound on number of single-bit 2-to-1 multiplexers needed for combining two \( k/2 \)-bit adders into a \( k \)-bit adder.

**Fig. 7.11** Top-level block for one bit position of a conditional-sum adder.
Conditional-Sum Addition Example

Conditional-sum addition of two 16-bit numbers. The width of the block for which the sum and carry bits are known doubles with each additional level, leading to an addition time that grows as the logarithm of the word width \( k \).

<table>
<thead>
<tr>
<th>Block width</th>
<th>Block carry-in</th>
<th>Block sum and block carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>( x: 00100110110110101010 )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( y: 01001001101101010000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( s: 011011011011010000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c: 0000001000100000 )</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>( x: 011011011011010000 )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( y: 0000001000100000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( s: 0111010010010000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c: 0000001000100000 )</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>( x: 011000011010101010 )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( y: 0000001000100000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( s: 01110010010000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c: 0000001000100000 )</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>( x: 011100001101010100 )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( y: 0000001000100000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( s: 011100001101010100 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c: 0000001000100000 )</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>( x: 01110010010000 )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( y: 0000001000100000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( s: 01110010010000 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c: 0000001000100000 )</td>
</tr>
</tbody>
</table>
5 Hybrid Adder Designs

The most popular hybrid addition scheme:

A hybrid carry-lookahead/carry-select adder.
Any Two Addition Schemes Can Be Combined

Example 48-bit adder with hybrid ripple-carry/carry-lookahead design.

Other possibilities: hybrid carry-select/ripple-carry
hybrid ripple-carry/carry-select
...
6 Optimizations in Fast Adders

What looks best at the block diagram or gate level may not be best when a circuit-level design is generated (effects of wire length, signal loading,...)

Modern practice: Optimization at the transistor level

Variable-block carry-lookahead adder

Optimizations for average or peak power consumption

Timing-based optimizations (next slide)
Optimizations Based on Signal Timing

So far, we have assumed that all input bits are presented at the same time and all output bits are also needed simultaneously.

Example arrival times for operand bits in the final fast adder of a tree multiplier [Oklo96].
Chapter Goals

Learn methods for speeding up the addition of several numbers (needed for multiplication or inner-product)

Chapter Highlights

Running total kept in redundant form
Current total + Next number → New total
Deferred carry assimilation
Wallace/Dadda trees and parallel counters
Multioperand Addition: Topics

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Using Two-Operand Adders</td>
</tr>
<tr>
<td>2. Carry-Save Adders</td>
</tr>
<tr>
<td>3. Wallace and Dadda Trees</td>
</tr>
<tr>
<td>4. Parallel Counters</td>
</tr>
<tr>
<td>5. Generalized Parallel Counters</td>
</tr>
<tr>
<td>6. Adding Multiple Signed Numbers</td>
</tr>
</tbody>
</table>
1 Using Two-Operand Adders

Some applications of multioperand addition

\[
\begin{array}{c}
\times \\
\hline
x_0 \ a_2^0 \\
x_1 \ a_2^1 \\
x_2 \ a_2^2 \\
x_3 \ a_2^3 \\
\hline
p \\
\end{array}
\begin{array}{c}
a \\
x \\
\hline
\end{array}
\]

\[
\begin{array}{c}
p^{(0)} \\
p^{(1)} \\
p^{(2)} \\
p^{(3)} \\
p^{(4)} \\
p^{(5)} \\
p^{(6)} \\
\hline
s \\
\end{array}
\]

Multioperand addition problems for multiplication or inner-product computation in dot notation.
Serial Implementation with One Adder

Serial implementation of multi-operand addition with a single 2-operand adder.

\[ T_{\text{serial-multi-add}} = O(n \log(k + \log n)) \]
\[ = O(n \log k + n \log \log n) \]

Therefore, addition time grows superlinearly with \( n \) when \( k \) is fixed and logarithmically with \( k \) for a given \( n \).
Pipelined Implementation for Higher Throughput

Problem to think about: Ignoring start-up and other overheads, this scheme achieves a speedup of 4 with 3 adders. How is this possible?

Fig. 8.1  Serial multi-operand addition when each adder is a 4-stage pipeline.
Parallel Implementation as Tree of Adders

Adding 7 numbers in a binary tree of adders.

\[ T_{\text{tree-fast-multi-add}} = O(\log k + \log(k + 1) + \ldots + \log(k + \left\lceil \log_2 n \right\rceil - 1)) \]

\[ = O(\log n \log k + \log n \log \log n) \]

\[ T_{\text{tree-ripple-multi-add}} = O(k + \log n) \quad \text{[Justified on the next slide]} \]
Elaboration on Tree of Ripple-Carry Adders

The absolute best latency that we can hope for is $O(\log k + \log n)$

There are $kn$ data bits to process and using any set of computation elements with constant fan-in, this requires $O(\log(kn))$ time

We will see shortly that carry-save adders achieve this optimum time
A ripple-carry adder turns into a carry-save adder if the carries are saved (stored) rather than propagated.

Carry-propagate adder (CPA) and carry-save adder (CSA) functions in dot notation.

Specifying full- and half-adder blocks, with their inputs and outputs, in dot notation.
Multioperand Addition Using Carry-Save Adders

\[ T_{\text{carry-save-multi-add}} = O(\text{tree height} + T_{\text{CPA}}) \]
\[ = O(\log n + \log k) \]

\[ C_{\text{carry-save-multi-add}} = (n - 2)C_{\text{CSA}} + C_{\text{CPA}} \]

Tree of carry-save adders reducing seven numbers to two.

Serial carry-save addition using a single CSA.
Example Reduction by a CSA Tree

Addition of seven 6-bit numbers in dot notation.

<table>
<thead>
<tr>
<th>Bit position</th>
<th>8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x2</td>
<td>7 7 7 7 7 7 6</td>
</tr>
<tr>
<td>6 FAs</td>
<td>2 5 5 5 5 5 3</td>
</tr>
<tr>
<td>6 FAs</td>
<td>3 4 4 4 4 4 1</td>
</tr>
<tr>
<td>4 FAs + 1 HA</td>
<td>1 2 3 3 3 3 2 1</td>
</tr>
<tr>
<td>7-bit adder</td>
<td>2 2 2 2 2 1 2 1</td>
</tr>
</tbody>
</table>

---Carry-propagate adder---

1 1 1 1 1 1 1 1

Total cost = 7-bit adder + 28 FAs + 1 HA

Representing a seven-operand addition in tabular form.

A full-adder compacts 3 dots into 2 (compression ratio of 1.5)

A half-adder rearranges 2 dots (no compression, but still useful)
Adding seven $k$-bit numbers and the CSA/CPA widths required. Due to the gradual retirement (dropping out) of some of the result bits, CSA widths do not vary much as we go down the tree levels.
3 Wallace and Dadda Trees

The maximum number \( n(h) \) of inputs for an \( h \)-level CSA tree

<table>
<thead>
<tr>
<th>( h )</th>
<th>( n(h) )</th>
<th>( h )</th>
<th>( n(h) )</th>
<th>( h )</th>
<th>( n(h) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>7</td>
<td>28</td>
<td>14</td>
<td>474</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>42</td>
<td>15</td>
<td>711</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>63</td>
<td>16</td>
<td>1066</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>10</td>
<td>94</td>
<td>17</td>
<td>1599</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>11</td>
<td>141</td>
<td>18</td>
<td>2398</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>12</td>
<td>211</td>
<td>19</td>
<td>3597</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>13</td>
<td>316</td>
<td>20</td>
<td>5395</td>
</tr>
</tbody>
</table>

\( n(h) \): Maximum number of inputs for \( h \) levels

\( h(n) = 1 + h(\lceil \frac{2n}{3} \rceil) \)

\( n(h) = \lfloor \frac{3n(h-1)}{2} \rfloor \)

\( 2 \times 1.5^{h-1} < n(h) \leq 2 \times 1.5^h \)
Example Wallace and Dadda Reduction Trees

Wallace tree:
Reduce the number of operands at the earliest possible opportunity

Dadda tree:
Postpone the reduction to the extent possible without causing added delay

Addition of seven 6-bit numbers in dot notation.

Adding seven 6-bit numbers using Dadda’s strategy.
A Small Optimization in Reduction Trees

Adding seven 6-bit numbers by taking advantage of the final adder’s carry-in.

Adding seven 6-bit numbers using Dadda’s strategy.

Total cost = 7-bit adder + 28 FAs + 1 HA

Total cost = 7-bit adder + 26 FAs + 3 HA
4 Parallel Counters

1-bit full-adder = (3; 2)-counter

Circuit reducing 7 bits to their 3-bit sum = (7; 3)-counter

Circuit reducing $n$ bits to their $\lceil \log_2(n + 1) \rceil$-bit sum
= $(n; \lceil \log_2(n + 1) \rceil)$-counter

A 10-input parallel counter also known as a (10; 4)-counter.
5 Generalized Parallel Counters

Multicolumn reduction

(5, 5; 4)-counter

Unequal columns

(2, 3; 3)-counter

Dot notation for a (5, 5; 4)-counter and the use of such counters for reducing five numbers to two numbers.

Gen. parallel counter = Parallel compressor
A General Strategy for Column Compression

(n; 2)-counters

To $i + 1$ 
To $i + 2$ 
To $i + 3$

One circuit slice

$n + \psi_1 + \psi_2 + \psi_3 + \ldots \leq 3 + 2\psi_1 + 4\psi_2 + 8\psi_3 + \ldots$

$n - 3 \leq \psi_1 + 3\psi_2 + 7\psi_3 + \ldots$

Example: Design a bit-slice of an (11; 2)-counter

Solution: Let’s limit transfers to two stages. Then, $8 \leq \psi_1 + 3\psi_2$

Possible choices include $\psi_1 = 5$, $\psi_2 = 1$ or $\psi_1 = \psi_2 = 2$
### Adding Multiple Signed Numbers

<table>
<thead>
<tr>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-1})</th>
<th>(x_{k-2})</th>
<th>(x_{k-3})</th>
<th>(x_{k-4})</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-1})</td>
<td>(y_{k-2})</td>
<td>(y_{k-3})</td>
<td>(y_{k-4})</td>
<td>\ldots</td>
</tr>
<tr>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-1})</td>
<td>(z_{k-2})</td>
<td>(z_{k-3})</td>
<td>(z_{k-4})</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

(a) Using sign extension

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

\(-b = (1 - b) + 1 - 2\)

(b) Using negatively weighted bits

Adding three 2's-complement numbers.
Comparisons
Adders:

• Ripple carry adder
• Carry Look ahead adder
• Carry Select adder
• Carry Skip adder
• Carry Save adder
• Manchester Carry Chains
• Variable Skip adder
• Brent-Kung adder
• Kogge-Stone adder
• Sklansky adder
• ELM adder and many more…
Adder Delay Comparisons

- CSkip-A
- VSkip-A
- CCSel-A
- VCSel-A
- 2CLA 4bCLA
- 2b B&K
Adder Area Comparisons

16 bits  32 bits  64 bits

- RCA
- Manchester
- CCSkip-A
- VCSkip-A
- CCSi1-A
- CLA
- B&K
- ELM-(prefix)
Adder Average Power Comparisons

- RCA
- MCC
- CCSkA
- VCSkA
- CCSkA
- CLA
- B&K
- ELM

[Graph showing power comparisons for different adder implementations across 16, 32, and 64 bits.]
How to do fast Arithmetic?

PDP of Different Adders

PDP: power delay product

How to do fast Arithmetic?