Outline

- Single-bit Addition
- Carry-Ripple Adder
- Carry-Skip Adder
- Carry-Lookahead Adder
- Carry-Select Adder
- Carry-Increment Adder
- Tree Adder
Single-Bit Addition

Half Adder

\[ S = A \oplus B \]
\[ C_{out} = A \cdot B \]

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & C_{out} & S \\
\hline
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\hline
\end{array}
\]

Full Adder

\[ S = A \oplus B \oplus C \]
\[ C_{out} = MAJ(A, B, C) \]

\[
\begin{array}{|c|c|c|c|c|}
\hline
A & B & C & C_{out} & S \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]
Full Adder Design I

- Brute force implementation from eqns

\[ S = A \oplus B \oplus C \]

\[ C_{out} = MAJ(A, B, C) \]
Factor $S$ in terms of $C_{out}$

$$S = ABC + (A + B + C)(\sim C_{out})$$

Critical path is usually $C$ to $C_{out}$ in ripple adder
Clever layout circumvents usual line of diffusion

- Use wide transistors on critical path
- Eliminate output inverters
Full Adder Design III

- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area
Full Adder Design IV

- Dual-rail domino
  - Very fast, but large and power hungry
  - Used in very fast multipliers
Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from $C_{in}$ to $C_{out}$
  - Design full adder to have fast carry delay

```
+-----------------+-----------------+-----------------+-----------------+
|     A_4         |     B_4         |     A_3         |     B_3         |
|     +           |     +           |     +           |     +           |
|     C_{out}     |     C_3         |     C_2         |     C_1         |
|   S_4           |   S_3           |   S_2           |   S_1           |
| +-----------------+-----------------+-----------------+-----------------+
|     A_2         |     B_2         |     A_1         |     B_1         |
|     +           |     +           |     +           |     +           |
|     C_{in}      |                   |                   |                   |
```
Inversions

- Critical path passes through majority gate
  - Built from minority + inverter
  - Eliminate inverter and use inverting full adder
Carry Propagate Adders

- N-bit adder called CPA
  - Each sum bit depends on all previous carries
  - How do we compute all these carries quickly?

\[
\begin{array}{c}
A_{N-1} & B_{N-1} \\
C_{\text{in}} & C_{\text{out}} \\
S_{N-1} & \\
\end{array}
\quad
\begin{array}{c}
0000 \\
1111 \\
+0000 \\
1111 \\
\end{array}
\quad
\begin{array}{c}
1111 \\
1111 \\
+0000 \\
0000 \\
\end{array}
\]

\text{carries}
A_{4-1} \\
B_{4-1} \\
S_{4-1}
For a full adder, define what happens to carries (in terms of A and B)

- **Generate:** $C_{out} = 1$
  independent of C
  - $G = A \cdot B$
- **Propagate:** $C_{out} = C$
  - $P = A \oplus B$
- **Kill:** $C_{out} = 0$
  independent of C
  - $K = \sim A \cdot \sim B$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$C_{out}$</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>
Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning i:j

\[
G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \quad G_{0:0} = C_{in}
\]
\[
P_{i:j} = P_{i:k} \cdot P_{k-1:j} \quad P_{0:0} = 0
\]

- Base case

\[
G_{i:i} \equiv G_i = A_i \cdot B_i \quad G_{0:0} \equiv G_0 = C_{in}
\]
\[
P_{i:i} \equiv P_i = A_i \oplus B_i \quad P_{0:0} \equiv P_0 = 0
\]

- Sum:

\[
S_i = P_i \oplus G_{i-1:0} = A_i \text{ xor } B_i \text{ xor } C_{in,i}
\]
PG Logic

1: Bitwise PG logic

2: Group PG logic

3: Sum logic
PG Diagram Notation

Black cell

\[ \begin{align*}
  & i:k \\
  & k-1:j \\
  & i:j
\end{align*} \]

Gray cell

\[ \begin{align*}
  & i:k \\
  & k-1:j \\
  & i:j
\end{align*} \]

Buffer

\[ \begin{align*}
  & i:j
\end{align*} \]

Black cell accepts lower order \( P_{k-1:j} \) input and provides aggregated \( P_{i:j} \) output
Carry-Ripple Revisited

\[ G_{i:0} = G_i + P_i \cdot G_{i-1:0} \]

\[ G_{i:0} = C_{o,i} = A_i \cdot B_i + C_i (A_i \oplus B_i) \]

\[ S_i = A_i \oplus B_i \oplus C_i \]

\[ = P_i \oplus G_{i:0} \]
Carry-Ripple PG Diagram

\[ t_{\text{ripple}} = t_{pg} + (N - 1)t_{AO} + t_{\text{xor}} \]

- Single-bit P/G logic embedded in input block
- Sum XOR embedded in output block
Carry-Skip Adder

- Carry-ripple is slow through all N stages
- Carry-skip allows carry to skip over groups of n bits
  - Decision based on n-bit propagate signal
- Exploits fact that the ripple-through is only needed when P=1

n-bit full adder
For k n-bit groups (N = nk)
\[ t_{\text{skip}} = t_{\text{pg}} + 2(n-1)t_{AO} + (k-1)t_{\text{Mux}} + t_{\text{xor}} \]
Compare to ripple:
\[ t_{\text{ripple}} = t_{\text{pg}} + (N-1)t_{AO} + t_{\text{xor}} \]
- Delay depends on length of 1\textsuperscript{st} and last groups + # of middle groups
- Delay grows as $O(\sqrt{N})$
Carry-Lookahead Adder

- Carry-lookahead adder computes $G_{i:k}$ in addition to $P_{i:k}$ for each block.
- Uses higher-valency cells with more than two inputs.

For $k$ groups of $n$ bits

$$t_{CLA} = t_{pg} + t_{pg(n)} + [(n-1) + (k-1)]t_{AO} + t_{XOR}$$
Higher-Valency Cells

Combines $P,G$ from >2 slices

High-valence gray cell similar, only provides $G_{\text{block}}$
CLA PG Diagram

\[ t_{CLA} = t_{pg} + t_{pg(n)} + [(n-1) + (k-1)] t_{AO} + t_{XOR} \]
CLA PG Diagram

\[ t_{CLA} = t_{pg} + t_{pg(n)} + [(n-1) + (k-1)]t_{AO} + t_{XOR} \]
Carry-Select Adder

- Trick for critical paths dependent on late input $X$
  - Precompute two possible outputs for $X = 0, 1$
  - Select proper output when $X$ arrives

- Carry-select adder precomputes $n$-bit sums
  - For both possible carries into $n$-bit group

\[ t = t_{pg} + [n + (k-2)]t_{AO} + t_{Mux} \]
Carry-Increment Adder

- Factor initial PG and final XOR out of carry-select
- Implement MUX with gray cell

\[ t_{\text{increment}} = t_{\text{pg}} + \left[ (n-1) + (k-1) \right] t_{\text{AO}} + t_{\text{xor}} \]
Variable Group Size

- Also buffer noncritical signals

\[ t = t_{pg} + \sqrt{(2N)t_{AO}} + t_{Mux} \]
Tree Adder

- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives $O(\log N)$ delay

- Many variations on tree adders

- **Terminology Note**: P-G logic is an example of prefix logic. Accepts inputs related to bit-wise information ($P_0:P_N; G_0:G_N$) and computes the prefix: $P_{0:N}; G_{0:N}$.
  - Also known as delta operator, fundamental carry operator
  - Techniques will apply to other problems, e.g. priority encoding.
Brent-Kung

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

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

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

- Finds prefixes for 2-bit groups, then 4b, on up to N/2, to quickly get Cin to MSB
- Then carry fans back out to intermediate bits
- Delay $\approx 2(\log_2(N) - 1) t_{AO}$
Sklansky

- Computes intermediate prefixes in parallel
- High fanout ([8,4,2,1] here)
  - Buffering can help
- Delay $\sim \log_2(N) \cdot t_{AO}$
- Kogge-Stone

- Uses overlapping blocks to get fanout of 2/stage \textit{and} delay $\sim \log_2(N) t_{AO}$

- Many gates & wiring tracks (high power and area)
  - If wiring drives area/delay, performance will suffer
Tree Adder Taxonomy

- Ideal N-bit tree adder would have
  - $L = \log N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels

- Describe adder with 3-D taxonomy ($l, f, t$)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: $2^t$
  - Ideal would be (0,1,0)

- Known tree adders sit on plane defined by
  $$l + f + t = L - 1$$
Tree Adder Taxonomy

(L) And vertices for a 16b adder
Knowles [2, 1, 1, 1]

[2,1,1,1] is excess fanout at each stage
Taxonomy Revisited

(a) Brent-Kung

(b) Sklansky

(c) Kogge-Stone

(d) Han-Carlson

(e) Knowles [2,1,1,1]

(f) Ladner-Fischer

Kogge-Stone

Sklansky

Brent-Kung

Han-Carlson

Ladner-Fischer
Other Notes

- Can use higher-valence cells (gray/black cells with >2 inputs) to decrease stages
  - Increase stage delay.
  - Typically poor tradeoff for static CMOS, but often beneficial in faster logic families (e.g. domino)
  - 3-D plane constraint still applies with modified equation (see text)

- Internal pipelining can substantially improve throughput
  - Costs latency, area, power

- Careful attention to buffering. De-load critical path, but don’t create a new critical path with buffer delay
## Summary

Adder architectures offer area / power / delay tradeoffs. Choose the best one for your application.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Classification</th>
<th>Logic Levels</th>
<th>Max Fanout</th>
<th>Tracks</th>
<th>Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry-Ripple</td>
<td></td>
<td>N-1</td>
<td>1</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>Carry-Skip n=4</td>
<td></td>
<td>N/4 + 5</td>
<td>2</td>
<td>1</td>
<td>1.25N</td>
</tr>
<tr>
<td>Carry-Inc. n=4</td>
<td></td>
<td>N/4 + 2</td>
<td>4</td>
<td>1</td>
<td>2N</td>
</tr>
<tr>
<td>Brent-Kung</td>
<td>(L-1, 0, 0)</td>
<td>2log₂N − 1</td>
<td>2</td>
<td>1</td>
<td>2N</td>
</tr>
<tr>
<td>Sklansky</td>
<td>(0, L-1, 0)</td>
<td>log₂N</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5 Nlog₂N</td>
</tr>
<tr>
<td>Kogge-Stone</td>
<td>(0, 0, L-1)</td>
<td>log₂N</td>
<td>2</td>
<td>N/2</td>
<td>Nlog₂N</td>
</tr>
</tbody>
</table>
Adder Selection

- Ripple adders are good if fast enough
- Carry-skip/increment good compromise w/ $N \in [8:32)$
- For high bit-widths ($\geq 32$), tree adders offer improved performance
- Kogge-Stone may be fastest, but power consumption is high
- Valency $\sim 4$ with dynamic logic
- Delay of 64b domino adder $\sim 7-9$ $t_{FO4}$ and area of 4-12 $M\lambda^2$
- Sklansky offers decent high-performance compromise (avoids high F and wiring area)
In synthesis, timing constraints will affect automatic adder selection. Can be a steep price for reduced delay, so don’t blindly overspec.
Adder Variations

- Ex: Addition with conditional +1 needed for crypto, ECC encoding, others.
- Single extra layer of gray cells rather than conditional extra operation or parallel adders
- When performance matters, know what your synthesizer is doing