TSMC Details The Benefits of Its N3 Node

October 27, 2021, EETimes - TSMC, now chugging along with its N5 process node, said it will have its evolutionary N4 node ramped up to volume production this year. The N3 node, which will provide more of a technological leap than N4, is planned to go into volume production in the second half of 2022. N3 will indeed offer customers the kind of performance improvements they might hope for from a major node jump, though the speed improvement will be at the low-end of TSMC’s projected aspirations from last year; the company also just missed its target for density improvement.

<table>
<thead>
<tr>
<th>Metric</th>
<th>N7 to N5</th>
<th>N5 to N3 (2020 projection)</th>
<th>N5 to N3 (2021 actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic density improvement</td>
<td>1.87x</td>
<td>1.7x</td>
<td>1.6x</td>
</tr>
<tr>
<td>Speed improvement</td>
<td>15%</td>
<td>10-15%</td>
<td>11%</td>
</tr>
<tr>
<td>Power draw improvement</td>
<td>-20%</td>
<td>(n/a)</td>
<td>-27%</td>
</tr>
</tbody>
</table>

Source: TSMC, EE Times
Review

• Binary adders are a common building block of digital systems
• Carry is in the critical path
• Mirror adders cells are commonly found in libraries
• Ripple-carry adder is the least complex, lowest energy
Ripple-Carry Adders
Sizing the Mirror Adder

\[ g_{Ci} = 2 \]

- Carry is in the critical path
- Optimal effort is 4, logical effort is 2
- Drives one carry and one sum input
  - Conveniently split fanout
- All stages equally sized
The Mirror Adder

• The NMOS and PMOS chains are completely symmetrical.
  A maximum of two series transistors in the carry-generation stack.

• Only the transistors in the carry stage have to be optimized for optimal speed. All
  transistors in the sum stage can be smaller.

• The transistors connected to $C_i$ are placed closest to the output.

• Minimize the capacitance at node $C_o$. 
Transmission Gate Full Adder

Setup

Sum Generation

Carry Generation
Carry Bypass Adders
Carry-Bypass Adder

• Also called ‘carry skip’

Idea: If \( P_0 \) and \( P_1 \) and \( P_2 \) and \( P_3 = 1 \) then \( C_{o3} = C_0 \), else “kill” or “generate”.

Also called Carry-Skip
Carry-Bypass Adder (cont.)

\[ t_{adder} = t_{setup} + M \cdot t_{carry} + \left(\frac{N}{M} - 1\right) t_{bypass} + (M - 1) t_{carry} + t_{sum} \]
Carry Ripple versus Carry Bypass

• Depends on technology, design constraints
To Design a Faster Carry-Bypass Adder

a) Uniform groups of 4 are optimal
b) Uniform groups >4 are optimal

c) Uniform groups <4 are optimal
d) Increasing group size with higher bit position
e) Wider groups around mid bit positions are optimal

www.yellkey.com/ahead
Faster Carry-Bypass
Administrivia

- Homework 7 due this week
- Homework 8 due next week
  - In scope for midterm
- All labs need to be checked off by this week!
- Projects (ASIC and FPGA) started, first check point this week
- Midterm 2 is on November 4 at 7pm
  - Review session tonight at 7pm
Carry-Select Adders
Carry-Select Adder

Setup

"0" Carry Propagation

"1" Carry Propagation

Multiplexer

Sum Generation

\( C_{o,k-1} \) → "0" Carry Propagation

"0" Carry Propagation

"1" Carry Propagation

Multiplexer

Sum Generation

\( C_{o,k+3} \)
Carry Select Adder: Critical Path

Bit 0–3

Setup

0-Carry

1-Carry

Multiplexer

Sum Generation

$S_{0-3}$

Bit 4–7

Setup

0-Carry

1-Carry

Multiplexer

Sum Generation

$S_{4-7}$

Bit 8–11

Setup

0-Carry

1-Carry

Multiplexer

Sum Generation

$S_{8-11}$

Bit 12–15

Setup

0-Carry

1-Carry

Multiplexer

Sum Generation

$S_{12-15}$
Linear Carry Select

\[ t_{add} = t_{setup} + \left( \frac{N}{M} \right) t_{carry} + M t_{mux} + t_{sum} \]
Square Root Carry Select

$$t_{\text{add}} = t_{\text{setup}} + P \cdot t_{\text{carry}} + (\sqrt{2N} \cdot t_{\text{mux}} + t_{\text{sum}}$$
Adder Delays - Comparison

![Graph showing comparison of adder delays.]

- **Ripple adder**
- **Linear select**
- **Square root select**

Graph parameters:
- $t_p$ (in unit delays)
- $N$

Nikolić Fall 2021
Carry-Lookahead Adders
Lookahead - Basic Idea

\[ C_{o,k} = f(A_k, B_k, C_{o,k-1}) = G_k + P_k C_{o,k-1} \]
Lookahead: Topology

Expanding lookahead equations:

\[ C_{o,1} = G_1 + P_1 C_{i,1} = G_1 + P_1 G_0 + P_1 P_0 C_{i,0} \]

\[ C_{i,1} = G_0 + P_0 C_{i,0} \]

Carry at bit \( k \):

\[ C_{o,k} = G_k + P_k (G_{k-1} + P_{k-1} C_{o,k-2}) \]

Expanding at bit \( k \):

\[ C_{o,k} = G_k + P_k (G_{k-1} + P_{k-1} (\ldots + P_1 (G_0 + P_0 C_{i,0}) \ldots)) \]

Carry-lookahead gate grows at each bit position!

\[ C_o = A B + B C_i + A C_i = G + P C_i \]
Carry Lookahead Trees

Build the carry lookahead tree as a hierarchy of gates

\[ C_{o,0} = G_0 + P_0 C_{i,0} \]
\[ C_{o,1} = G_1 + P_1 C_{i,1} = G_1 + P_1 G_0 + P_1 P_0 C_{i,0} \]
\[ C_{o,2} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{i,0} \]
\[ = (G_2 + P_2 G_1) + (P_2 P_1)(G_0 + P_0 C_{i,0}) = G_{2:1} + P_{2:1} C_{O,0} \]

Can continue building the tree hierarchically.
Logarithmic (Tree) Adders – Idea

- “Look ahead” across groups of multiple bits to figure out the carry
  - Example with two bit groups:
    \[ P_{1:0} = P_1 \cdot P_0, \quad G_{1:0} = G_1 + P_1 \cdot G_0, \quad \Rightarrow \quad C_{out1} = G_{1:0} + P_{1:0} \cdot C_{0,in} \]

- Combine these groups in a tree structure:
  - Delay is now \(~\log_2(N)\)
Rest of the Tree

- Previous picture shows only half of the algorithm
  - Need to generate carries at individual bit positions too
Many Kinds of Tree Adders

- Many ways to construct these tree (or “carry lookahead”) adders
  - Many of these variations named after the people who created them
- Most of these vary three basic parameters:
  - Radix: how many bits are combined in each PG gate
    - Previous example was radix 2; often go up to radix 4
  - Tree depth: stages of logic to the final carry. Must be at least $\log_{\text{Radix}}(N)$
  - Sparseness
16-bit radix-2 Kogge-Stone tree
Tree Adders

16-bit radix-4 Kogge-Stone Tree
Sparse Trees

16-bit radix-2 sparse tree with sparseness of 2
Tree Adders

Brent-Kung Tree
Multipliers
Warmup

• Recall long multiplication of base-10 by hand:

\[
\begin{array}{c}
12 \\
\times 56 \\
\end{array}
\]

• In base-2 (binary), we do the same thing:

\[
\begin{array}{c}
011 \\
\times 101 \\
\end{array}
\]
Multiplication

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  \times & b_3 & b_2 & b_1 & b_0 \\
  \hline
  a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
  a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
  a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
  a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\end{array}
\]

\[
\cdots \quad a_1b_0 + a_0b_1 + a_0b_0 \quad \text{Product}
\]

Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (energy, 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 ← 0, A ← multiplicand, B ← 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) more times.
5. [P][B] has product.

• Performance: N cycles of N-bit additions
“Shift and Add” Multiplier

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
Convince yourself

• What’s \(-3 \times 5\)?

\[
\begin{array}{c}
1101 \\
\times 0101 \\
\end{array}
\]
Unsigned Parallel Multiplier
Parallel (Array) Multiplier

\[
a_3 \ a_2 \ a_1 \ a_0 \\
\times \ b_3 \ b_2 \ b_1 \ b_0 \\
\underline{-------------------}
\]

\[
a_{3b0} \ a_{2b0} \ a_{1b0} \ a_{0b0} \\
+ \ a_{3b1} \ b_{2a1} \ a_{1b1} \ a_{0b1} \\
+ \ a_{3b2} \ a_{2b2} \ a_{1b2} \ a_{0b2} \\
+ \ a_{3b3} \ a_{2b3} \ a_{1b3} \ a_{0b3} \\
\underline{-------------------}
\]

\[
p_7 \ p_6 \ p_5 \ p_4 \ p_3 \ p_2 \ p_1 \ p_0
\]

• Performance: What is the critical path?
Parallel (Array) Multiplier

Single cycle multiply: Generates all \( n \) partial products simultaneously.

Each row: \( n \)-bit adder with AND gates

What is the critical path?
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, $3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

\[
\begin{align*}
3_{10} & \quad 0011 \\
+ \quad 2_{10} & \quad 0010 \\
\quad c & \quad 0100 = 4_{10} \\
\quad s & \quad 0001 = 1_{10} \\
\end{align*}
\]

carry-save add

\[
\begin{align*}
3_{10} & \quad 0011 \\
+ \quad 3_{10} & \quad 0011 \\
\quad c & \quad 0010 = 2_{10} \\
\quad s & \quad 0110 = 6_{10} \\
1000 & \quad = 8_{10} \\
\end{align*}
\]

carry-propagate add

- In general, carry-save addition takes in 3 numbers and produces 2: "3:2 compressor":
- 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 inexpensive (full adders)
Array Multiplier Using Carry-Save Addition

- What is the critical path?
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
- It doesn’t matter where you add the carries and sums, as long as you eventually do add them
- This structure is the basis of the **Wallace Tree Multiplier**
- Partial products are summed with the CSA tree. Fast adder (ex: CLA) is used for final sum
- Multiplier delay \(\alpha \log_{3/2} N + \log_2 N\)
Wallace-Tree Multiplier

• Reduce the partial products in logic stages – 4 x 4 example

Partial products

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

First stage

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Bit position

Second stage

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Final adder

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
Wallace-Tree Multiplier

Note: Wallace tree is often slower than an array multiplier in FPGAs (which have optimized carry chains)
Increasing Throughput: Pipelining

• Multipliers have a long critical path: PP generation → reduction tree → final adder
  • Often pipelined before final adder (2x flip-flops for carry-save)
Summary

• Adders
  • Carry is in the adder critical path
  • Mirror adders cells are commonly found in libraries
  • Ripple-carry adder is the least complex, lowest energy
  • Carry-bypass, carry-select are usually faster than ripple-carry for bitwidths > 8

• Multipliers
  • Shift-and-add is the most compact
  • Parallel multipliers