What’s the Deal?

- All we want to do is add up a couple numbers...
- Chapter one tells us that we can add signed numbers just by adding the mapped positive bit vectors

\[ Z = [X_R + Y_R] \mod C \]

\[ x + y + c_{in} = 2^r c_{out} + s \]

\[ c_{out} = \begin{cases} 
1 & \text{if } (x + y + c_{in}) \geq 2^r \\
0 & \text{otherwise} 
\end{cases} \]

\[ c_{out} = \left\lfloor \frac{(x + y + c_{in})}{2^r} \right\rfloor \]

Adder Bits

<table>
<thead>
<tr>
<th>Half Adder</th>
<th>Full Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S = A \times B )</td>
<td>( S = A \times B )</td>
</tr>
<tr>
<td>( C_{out} = A )</td>
<td>( C_{out} = A )</td>
</tr>
<tr>
<td>( C_{in} = B )</td>
<td>( C_{in} = B )</td>
</tr>
<tr>
<td>( C_{out} = A )</td>
<td>( C_{out} = A )</td>
</tr>
<tr>
<td>( A )</td>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
<td>( B )</td>
</tr>
<tr>
<td>( C_{in} )</td>
<td>( C_{in} )</td>
</tr>
<tr>
<td>( S )</td>
<td>( S )</td>
</tr>
</tbody>
</table>

- Carry generation!
  - Carry at \( i \) depends on \( j \times i \)
  - Non-trivial to do fast – lots of inputs...

Big Problem (for speed)

Fast Adders

- The primary objective is to speed up the generation of the carries!
  - Carry Propagate Adders –
    - Produce an answer in conventional fixed-radix NRS
  - Carry Save and Signed Digit adders –
    - Avoid carry propagation by producing sums in redundant notations
  - Hybrid Adders
    - Combine as many schemes as make sense...
### Carry-Propagate Adders
- Carry-Ripple
- Switched Carry-Ripple (Manchester Carry)
- Carry-Skip
- Carry-Lookahead
- Prefix Adders (Tree Adders)
- Carry-Select (Conditional Sum)
- Carry-Completion Sensing (self-timed)

### Redundant Adders
- Carry-Save
- Signed Digit

### Basic Carry-Ripple Adder (CRA)

![CRA Diagram]

\[
T_{CRA} = (n - 1)t_c + \max(t_c, t_s)
\]

### Adder Performance

- Most standard cell libraries have a Full Adder cell as a single cell
- Implements Full Adder function directly in nmos and pmos transistors
- Delays should be smaller…
Adder Bits (CMOS)

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

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

Brute Force circuit...

Mirror Adder

- Factor \( S \) in terms of \( C_{\text{out}} \)
  \[ S = ABC + (A + B + C)(\neg C_{\text{out}}) \]

- Critical path is usually \( C_{\text{in}} \) to \( C_{\text{out}} \) in ripple adder

Connect for carry-ripple adder

Inversions...

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

Mirror Adder

Build a faster circuit?

- **Truth Table**
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Sum</th>
<th>Carry</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>1</td>
<td>0</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- When \( A \oplus B = 0 \), \text{SUM} = C, and \text{Carry} = B.
- When \( A \oplus B = 1 \), \text{SUM} = \neg C, and \text{Carry} = C.
- Using the 6T XOR, this full adder uses 18T.
Build a faster circuit?

- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area

![CPL Diagram](image)

Build a faster circuit?

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

![Dual-rail Domino Diagram](image)

Build a faster carry chain?

- Manchester Carry Chain
  - Use transmission gates to make carry “wire”

![Manchester Carry Chain Diagram](image)

MCC

Switch-Logic:
- Implement propagate with pass gate
- Implement kill with a pull down transistor
- Implement generate with a pull up transistor

To reduce the logic needed, and the capacitance on the carry chain use precharge switch logic. Precharge the output high, and pull it low if needed. The inputs to the gate can be outputs from other domino gates (Carry is a monotonic function of P, C, K^1)

![MCC Diagram](image)

Manchester Carry control

![Manchester Carry Control Diagram](image)

Manchester Carry Chain

![Manchester Carry Chain Diagram](image)
The carry chain is only part of the adder. You need to generate the P, G signals that the adder needs and to generate the sum at the end. In addition to the carry chain, each bit cell needs the following gates:

- The gates that generate P, G, K can be precharge gates, since the inputs are usually stable signals. This means that P, G, K can be domino signals, and can drive the domino carry chain.
- The final EXOR must be a static gate since it is not a monotonic function of its inputs, and its inputs will be \( \bar{y} \) signals.

**Manchester Carry Delay**

\[ T_{SKC} = t_{sw} + (n-1)t_{p} + (n/m)t_{sw} + t_{s} \]

- \( t_{sw} \) is time to set all switches
- \( t_{p} \) is time to propagate through a switch
- \( t_{sw} \) is a buffer – need restoring buffer every m bits
- \( t_{s} \) computes the sum based on the carries

This works well if \( t_{p} \) is small.

**Timing of MCC**

The good news is there is not a gate between stages.
The bad news is that the number of series transistors increases with the number of stages, so the delay will grow like \( n^2 \).

- Capacitance per stage (assuming all 4-2 devices, no diff sharing): 3 nff + pff + Cq + inv + bit-width of wiv = 2DF + 4F + 4F + 8F + 8F (30) = 30FF
- Resistance per-stage is 6.5K, so the delay is approximately \( 1.2VF \times n^2 \times R(\text{Q}/2) \) where \( n \) is the number of stages directly fed together.

**Sizing MCC**

Critical path is through the pass chain. Try to reduce this delay:

- Make P and G transistors 4x larger, and share diffusion (1)

**Buffered Carry Chains**

What is the "right" number of stages?

Assume first transistor is 8x min, and final inverter is minimum delay. The inverter delay (Count rising) plus the delay of the chain including the resistance of the initial 8x transistor.
Timing MCC

<table>
<thead>
<tr>
<th>Stages</th>
<th>Total Delay per Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td>2</td>
<td>0.82</td>
</tr>
<tr>
<td>3</td>
<td>1.15</td>
</tr>
<tr>
<td>4</td>
<td>1.58</td>
</tr>
<tr>
<td>5</td>
<td>2.12</td>
</tr>
<tr>
<td>6</td>
<td>2.77</td>
</tr>
</tbody>
</table>

So for these sizes, the optimal number in a stage is around 4, and the average delay per bit is around 0.4 ns. This is not optimally sized (pMOS in final inverter should be larger) but it is probably close.

Layout of MCC

Layout of a Manchester adder is not too bad, even with groups:

Back to Adder Bits

- Revisit the full adder

\[
\text{FA} \\
\begin{array}{cccccc}
X & Y & C & C_{out} & S & \text{Comment} \\
0 & 0 & 0 & 0 & 0 & \text{Kill} \\
0 & 0 & 1 & 0 & 1 & \text{Kill} \\
0 & 1 & 0 & 0 & 1 & \text{Propagate} \\
0 & 1 & 1 & 1 & 0 & \text{Propagate} \\
1 & 0 & 1 & 0 & 1 & \text{Propagate} \\
1 & 1 & 0 & 1 & 0 & \text{Generate} \\
1 & 1 & 1 & 1 & 1 & \text{Generate} \\
\end{array}
\]

Carry Chains

- Two types
  - 1-carry chain and 0-carry chain
  - 1-carry always starts at \( g_i = 1 \) (or \( c_e = 1 \)), and propagates over consecutive positions \( p_j = 1 \)
  - 0-carry starts at \( k_i = 1 \) position (or \( c_e = 0 \))...

\[
\begin{array}{cccccccc}
& i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
k_i & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
g_i & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
p_j & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & k \\
c_{i+1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Group Carries

- Carry equation can be generalized to groups of bits

\[
c_{j+1} = g(j,i) \bullet p(j,i) c_i = g(j,i) \cdot a(j,i) c_i \\
c_{j+1} = g(j,0) \bullet p(j,0) c_0 = g(j,0) \cdot a(j,0) c_0 \\
\]

- Combine subranges recursively

\[
g(f,i) = g(f,i) \bullet p(f,i) g(f,-1,i) = g(f,i) \cdot a(f,i) g(f,-1,i) \\
a(f,i) = a(f,i) g(f,-1,i) \\
p(f,i) = p(f,i) p(f,-1,i) \\
\]
Example (2.1)

- find bit 13 of the following sum

\[ x = 0110|0010|1100|0011 \]
\[ y = 1011|1101|0001|1110 \]

- first compute pkg for each bit

\[ x = 0110|0010|1100|0011 \]
\[ y = 1011|1101|0001|1110 \]

\[ p_{12} = 1 \quad p_{(11-8)} = 1 \quad k_{(7-4)} = 1 \quad g_{(3-0)} = 1 \]
\[ p_{(12-8)} = 1 \quad k_{(7-0)} = k_{(7-4)} + p_{(7-4)}k_{(3-0)} = 1 \]

- now combine in groups

\[ x = 0110|0010|1100|0011 \]
\[ y = 1011|1101|0001|1110 \]

\[ p_{12} = 1 \quad p_{(11-8)} = 1 \quad k_{(7-4)} = 1 \quad g_{(3-0)} = 1 \]

- extend groups to whole range

\[ x = 0110|0010|1100|0011 \]
\[ y = 1011|1101|0001|1110 \]

\[ p_{12} = 1 \quad p_{(11-8)} = 1 \quad k_{(7-4)} = 1 \quad g_{(3-0)} = 1 \]
\[ p_{(12-8)} = 1 \quad k_{(7-0)} = k_{(7-4)} + p_{(7-4)}k_{(3-0)} = 1 \]
\[ k_{(12-0)} = k_{(12-8)} + p_{(12-8)}k_{(7-0)} = 1 \]
Example (2.1)

Now you can compute $c_{13}$

$x = 0110|0010|1100|0011$
$y = 1011|1101|0001|1110$

$p|pppp|ppkp|pppp$

$k_{(12-0)} = k_{(12-8)} + p_{(12-8)} k_{(7-0)} = 1$
$c_{13} = g_{(12-0)} + p_{(12-0)} c_{in} = 0$

Example (2.1)

With $c_{13}$ you can compute $s_{13}$

$x = 0110|0010|1100|0011$
$y = 1011|1101|0001|1110$

$p|pppp|ppkp|pppp$

$k_{(12-0)} = k_{(12-8)} + p_{(12-8)} k_{(7-0)} = 1$
$c_{13} = g_{(12-0)} + p_{(12-0)} c_{in} = 0$
$s_{13} = x_{13} \oplus y_{13} \oplus c_{13} = 1 \oplus 1 \oplus 0 = 0$

Carry Skip Adder

The idea is to reduce the number of cells the worst-case carry must propagate through

- Divide n-bit adder into groups of m-bits
- Determine group propagate for each m-bits
- If the entire group $p$ is true, skip around it

Carry Skip example

![Carry Skip example diagram]

Carry travels through at most two groups: the initiating group and the terminating group.

Carry Skip worst case

![Carry Skip worst case diagram]
Carry Skip delay

\[ T_{CSK} = m t_c + t_{max} + \left( \frac{m}{2} - 2 \right) t_{max} + (m - 1) t_c + t_s \]
\[ = (2m - 1) t_c + \left( \frac{m}{2} - 1 \right) t_{max} + t_s \]

- Worst case is when a carry is generated in the first bit of the adder
  - Then propagated through all bits up to but not including the high order bit
  - That is, skip all groups but the first and last

Problem with clearing carries

- Watch out – some books show an AND/OR version that doesn’t really work!
- Problem is that carries might be left over from previous addition and have to dribble out...

Group Size

- Previous delay analysis assumes all groups are the same size
  - This isn’t the best for speed…
  - Carries generated in the first group have to skip more groups!
  - For fixed size:

\[ m_{opt} = \left( \frac{t_{max} t_c}{2} \right)^{1/2} \] (minimum delay)
\[ T_{opt} \approx \left( \frac{1}{2} t_{max} t_c n \right)^{1/2} \]

Carry Skip with different m

- If you vary the group size, will the groups at the ends have to add carries to the middle, you can speed things up...

Carry Skip – Another View

Since we have divided the bits in the word into a number of groups:
- For each group check to see if all the P are true
- If so, then bypass the Cin to Cout of that group
- Otherwise, do the normal thing.
Carry Skip - Layout

Carry Lookahead

- General idea – find a way to compute all carries at the same time
- Generate logic for all carries in terms of just the X, Y and Cin bits

\[ x^{(i)} = \sum_{j=0} x_j \cdot 2^i \quad c_i = 1 \text{ if } (x^{(i-1)} + y^{(i-1)} + c_{i-1}) \neq 2^i \]
- This is a switching function of \( 2i+1 \) variables

Carry Lookahead equations

\[ C_i = G_i + P_i \cdot C_i \]
\[ C_1 = G_0 + P_0 \cdot C_0 \]
\[ C_2 = G_1 + P_1 \cdot C_1 \]
\[ C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 \]
\[ C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 \]

Or

\[ C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 \]

Carry Lookahead equations

- Remember: \( C_i = G_i + A_i \cdot C_i \)
- \( C_1 = G_0 + A_0 \cdot C_0 \)
- \( C_2 = G_1 + A_1 \cdot C_1 \)
  \[ = G_1 + A_1 \cdot (G_0 + A_0 \cdot C_0) \]
  \[ = G_1 + A_1 \cdot G_0 + A_1 \cdot A_0 \cdot C_0 \]
- \( C_3 = G_2 + A_2 \cdot G_1 + A_2 \cdot A_1 \cdot G_0 + A_2 \cdot A_1 \cdot A_0 \cdot C_0 \)
- \( C_4 = G_3 + A_3 \cdot G_2 + A_3 \cdot A_2 \cdot G_1 + A_3 \cdot A_2 \cdot A_1 \cdot G_0 \)
  \[ + A_3 \cdot A_2 \cdot A_1 \cdot A_0 \cdot C_0 \]
- Or \( C_4 = G_3 + A_3 \cdot G_2 + A_3 \cdot A_2 \cdot G_1 + A_3 \cdot A_2 \cdot A_1 \cdot G_0 \)

CLA-4 Module
$T_{CLG} = t_{ax} + \frac{n}{m} t_{px} + t_s$

**CLG-4 Module**

Motorola – 1u CMOS, 4.5ns for a 64-bit adder...

*Slide from Mark Horowitz, Stanford*

**Carries - Another View**

**Carry Ripple revisited**

$G_{0i} = G_i + PG_{i+1}$

**Carry Skip revisited**
Carry Skip revisited

Fixed group size (4,4,4,4)  
Variable group size (2,3,4,4,3)

Carry Lookahead revisited

- Carry-lookahead adder computes $G_{i0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs

Higher Valency Cell

Recall $C_3 = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0 C_0)))$

CLA/Manchester adder

Two-Level CLA

- For large $n$, lots of groups so CLA can be slow
  - Apply CLA principle among groups
  - Compute $G$ and $A$ for groups
  - $C^{(1)} = G_0 + A_0 C_0$
  - $C^{(2)} = G_1 + A_1 G_0 + A_1 C_0$
  - etc...
  - Once the carries from the groups are produced, they are used by the first-level CLAs to produce the bit carries and sums

Two-level CLA32 ($n=p=4$)
Three-level CLA

- Extend to three or more levels by having lookahead between sections
  - First compute $a_i$, $p_i$, $g_i$
  - $L-1$ level of CLA to compute $A$s and $G$s
  - $n/m^L$ CLGs connected in ripple to compute carries of bits
  - One level of XOR to compute the sum

$$T_{CLA} = T_{n-g} + (L-1)t_{n-g} + \frac{n}{m}t_{n-g} + (L-1)t_{n-g} + t_s$$

CLA Critical Path

- More general form of carry lookahead tree
  - Built using different organizations of the same set of basic PG cells (PA cells)
  - All based on the fact that $c_i$ corresponds to the generate signal spanning bit positions $(-1)$ to $i-1$
  - Prefix adder is an interconnection of cells that produce $g_{i+1}$ for all $i$
  - Cells connected to produce $g$ signals that span an increasing number of bits

Prefix Adders (Tree Adders)

Overlapping Ranges

- Starting with $g,a$ of each bit, first level generates $g,a$ for two bits, then four, etc.
  - If right input spans bits $[\text{right}_2,\text{right}_1]$, and left spans $[\text{left}_2,\text{left}_1]$, with $\text{right}_2 + 1 \geq \text{left}_1$
  - Then output spans $[\text{left}_2,\text{right}_1]$
  - For example, right$[5,2]$ and left$[8,4]$ means output spans bits $[8,2]$

PG (PA) cell
PG (PA) Cells

8-bit Prefix Adder

Fanout can be an issue...

8-bit Prefix Adder

Lower fanout, increase levels

8-bit prefix adder

Max fanout 2
Min levels

Another View of Prefix Adders

\[ T_{PA} = t_{ag} + \log_2(n) + t_{XOR} \]
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: $2^t$

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

- For each group:
  - Compute the sum assuming that $C_{in}$ is 0 and that $C_{in}$ is 1
  - When you find out the right answer, use a MUX to select the correct result

- Carry-select is 1-level select
- Conditional Sum is a general case – up to max levels
**Carry-Select Adder**

By using more parallelism, one can build even faster adders.

While waiting for the carry input, why not calculate both possible answers (answer if Cin is 0 and answer if Cin is 1)?

When Cin is known, it is only a Mux delay to get Cout and all the Sums for the group.

---

**Carry-Select – Another View**

---

**Carry-Select - Layout**

---

**Conditional Sum**

- Conditional principle is applied recursively
  - Each group is combined to double the number of bits at the next level

---

**16-bit Conditional Sum Adder**

---

**Example**

- Step 1: Compute all the bit results
- Step 2: Use the known results to select the next groups...
- \( T_{\text{COND-SUM}} = t_{\text{add}, m} + \left( \frac{n}{m} \right) t_{\text{mux}} \)
Pipelined Adders

Variable Time Adder

- Carry Completion Sensing Adder
  - Encode the carry in a form that lets you tell when it’s finished
  - When all carry chains have finished, the add is finished
  - One choice – dual-rail encoding

Two carry signals:

<table>
<thead>
<tr>
<th>$c_i^0$</th>
<th>$c_i^1$</th>
</tr>
</thead>
<tbody>
<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>

not determined (yet)
does not occur

Addition Time: proportional to $\log_2(n)$

For uniformly distributed numbers, length of longest carry chain is approx $\log_2(5n/4)$

Variable Time Adder

Addition Time: proportional to $\log_2(n)$

For uniformly distributed numbers, length of longest carry chain is approx $\log_2(5n/4)$
Aside – ALU Design

Once you have an adder, making an ALU is very simple.

Two approaches:

- Build a separate logic unit and mix together the outputs. This is probably the fastest solution, since you don’t slow down the add critical path, but it will take more area.
- Merge the two designs together by changing the definition of P and G. Since the output (Sum) is P XOR Cin, if G = 0, and Cin(to adder) = 0 then Sum will equal P. Can do logical operations by using a general function box for the P function.

The first is probably the preferable solution, but I will show the second, because it is a little more clever (and the programmable P function unit is a perfect LU for the first solution).

Aside – ALU +P function block

The block that generates the signal called P must be able to generate any Boolean function of two variables. This is easy — just use a mux. To reduce control lines, I will use a precharge mux.

Aside – ALU +G function block

This is similar to the P function block, but it does not need to be as complex. If we only wanted to do addition and logic functions, then it would only need to generate the functions (AND, 0). But we want to be able to do subtraction too.

- If A - B = A + B = 1, where B is the ones complement of B, which is just the complement of each bit.
- Since after the P and G function block, no other part of the adder uses A,B, we can get subtract by redefining P and G, an setting Cin to be 1.
- If we didn’t do this, we would need to add an explicit mux to invert one of the inputs to adder in the case of subtraction.

For addition:
\[ P = \overline{A} + B \overline{\overline{A}} \quad G = A \overline{B} \]

For subtraction:
\[ P = A \overline{B} + B \overline{A} \quad G = A \overline{B} \]

Rest of the ALU

Is basically the same as an adder:

- Need a fast carry chain
- Final static XOR gate
- Latch to hold the value (since the output of the ALU is \( y' \))
- Bus driver to drive the output of the latch on bus when the ALU result is needed

Redundant Digit Adders

- Use a redundant digit set
  - Operands might be in conventional or in redundant form
  - Main idea is to reduce the carry propagation
  - But, increases number of bits in the result
  - Useful for things like accumulation, multi-operand addition, multiplication, etc.

Carry Save Adder

- Add three binary vectors
  - Using an array of one-bit adders (i.e. full adders)
- But, don’t propagate the carries!
- Output is two vectors: carry and pseudo-sum (or sum)
  \[ x' + y' + z = v_x + v_z = v \]
- Several combinations of \( v_x \) and \( v_z \) represent the same result
Carry Save Adder

- If you want to convert back to conventional numbers, add vs and vc.
  - Because there are two bits for every conventional sum bit, you can think of the answer in Carry Save form to be digits in the set \( \{0, 1, 2\} \).
  - Carry Save produces a reduction from three binary vectors to two, so it's also called a 3-2 reduction.
  - Adder is a [3:2] adder.

Carry Save Example

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>VS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[(c_{out}, VC) = 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1\]

Digit value: 0 1 2 2 1 0 2 0 2

Carry Save [4:2]

- What if two operands are both carry-save?
  - Then each operand is in Xs Xc form.
  - Combine four vectors into two...
  - Still no carries!
  - Answer is still in redundant carry-save form.
Carry Save [4:2]

Note that even though it looks like carry is propagated, the Cout from each [4:2] cell is computed directly from the A and B inputs.

4:2 compressor cell

Nagamatsu, Toshiba

4:2 compressor cell

Navi and Etienne

[4:2] Compressor Adder

High Radix Carry Save

- Regular carry-save doubles the number of bits
- You can reduce the number of bits with high-radix carry-save
- If $r$ is the radix
  - $V_s$ is represented in radix $r$
  - $V_c$ has one bit per radix-$r$ digit
Radix-8 Carry Save

Signed Digit Adders

Case A: two SD operands, result SD
Step 1:
\[(i_{n+1}, w_n) = \begin{cases} (0, x_i + y_i) & \text{if } -n + 1 \leq x_i + y_i \leq n - 1 \\ (1, x_i + y_i - r) & \text{if } x_i + y_i \geq n \\ (-1, x_i + y_i + r) & \text{if } x_i + y_i \leq -n \end{cases}\]
- algorithm modified for \( r = 2 \)

Case B: two conventional operands, result SD
Case C: one conventional, one SD, result SD
Signed Digit Adder

- I’m not going to spend more time on this one…
  - My sense is that it’s not as important in terms of actual implementations as Carry Save
  - Reasonably complex stuff – multiple recodings

Summary

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Delay proportional to</th>
<th>Area proportional to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear structures:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry ripple</td>
<td>( t )</td>
<td>( n )</td>
</tr>
<tr>
<td>Carry lookahead (one level)</td>
<td>( t/n )</td>
<td>( k_{nl}(n)/n )</td>
</tr>
<tr>
<td>Carry select (one level)</td>
<td>( t/n )</td>
<td>( k_{ns}(n)/n )</td>
</tr>
<tr>
<td>Carry skip (one level)</td>
<td>( \sqrt{n} )</td>
<td>( n )</td>
</tr>
<tr>
<td>Logarithmic structures:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry lookahead (max. levels)</td>
<td>( 2 \log_2 n )</td>
<td>( k_{nl}(n)/n )</td>
</tr>
<tr>
<td>Prefix</td>
<td>( \log_2 n )</td>
<td>( (k_{nl}(n)/n \log_2 n) )</td>
</tr>
<tr>
<td>Conditional sum</td>
<td>( \log_2 (n/m) )</td>
<td>( k_{ns}(n)/n )</td>
</tr>
<tr>
<td>Completion signal (avg. delay)</td>
<td>( \log_2 (n/m) )</td>
<td>( k_{ns}(n)/n )</td>
</tr>
<tr>
<td>Redundant</td>
<td>const</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Case Study

- Dec Alpha 21064 64-bit adder
  - 5ns cycle time in a 0.75u CMOS process
  - Very high performance for the day!
  - A mix of multiple techniques!

Alpha 21064

- In 8-bit chunks – Manchester carry chain
  - Chain was also tapered to reduce the load caused by the remainder of the chain
  - Chain was pre-discharged at start of cycle
  - Three signals used: \( P \), \( G \), and \( K \)
  - Two Manchester chains:
    - One assuming \( C_{in}=0 \)
    - One assuming \( C_{in}=1 \)

Alpha 21064

- Carry Lookahead used on least significant 32 bits
  - Implemented as distributed differential circuits
  - Provide carry that controls most significant 32
- Conditional Sum used for most significant 32
  - Six 8-bit select switches used to implement conditional sum on the 8-bit level

Alpha 21064

- Finally, Carry Select used to produce the most significant 32 bits.
  - Final selection done using NMOS carry-select byte-wide muxes
  - Also apparently pipelined with a row of latches after the lookahead…
Adder from Imagine

- Part of Imagine
  - A high-performance media processor designed at Stanford
- 32-bit segmented integer adder
  - Two-level tree to compute global carries
  - Uses carry-select to compute final sums from global carries
- Static CMOS logic
  - Also pass gate logic
- Design constraints
  - Area
  - Design complexity (modularity)
  - Speed

Adder from Imagine

- It is balancing design/logic complexity and speed
  - It uses large groups which will ultimately limit performance
- It does use some tree structures
  - It does not ripple carries
  - But the group generation is a little slow
- Also uses large block sizes (8 bits)
  - Does not move the carry select input to lower significance
  - Need to worry about how outputs in block are generated

72-bit Pentium II Adder

- 72-bit adder (Jason Slinson)
- 0.35u process
- Domino
- Kogge-Stone
  - CLA+sumselect
- Combines terms in both domino and CMOS stages
Local PGK Logic (Imagine)

- Pre-computation necessary to do fast carry computation
  - P = a\oplus b
  - G = ab
  - K = -(a+b)

- Size gates to fan-out to four carry chains
- Note: To do A-B, use -B here

Group PKG (Imagine)

- Manchester Carry Chains.
  - Usually dynamic, but still works with static logic
  - Group PKG's:
    - GK = -(G+GP)
  - Use Carry chains
  - Example for GS (group generate):

Static Carry Chains

- Sizing is to reduce the parasitic delay
  - This delay dominates in large fanin structures, since it grows proportional to n^2
  - Using geometric sizing (reducing each transistor along the chain by 1) makes the parasitic delay linear
  - But still does not make them fast
  - Even though this chain is static logical
    - Drive the carry chain both up (K) and down (G)
    - Output is degraded, since it uses nMOS only pass devices
      - Using CMOS transmission gates is usually slower because of the added parasitic capacitors

Global Carry Chain

- Must fan out GCIN[3:1] to 8 muxes
  - Added load capacitance slows down the chain

Conditional Sums

- Use same carry chain
  - Do two of these (one for cin=0, one for cin=1):
  - Mux SUM[0:7] and SUM[1:7] with output of global carry chain
Arithmetic for Media Processing

- Used in Media processing
  - DSP’s, multimedia extensions to instruction set architectures (MMX, VIS)
- Consider three variations of conventional arithmetic:
  - Segmented Arithmetic
    - Break carry chain
  - Arithmetic operations similar to add/subtract
    - Example: 4 parallel 8-bit unsigned absolute differences
  - Saturation
    - Don’t wraparound on overflow

Segmented Add Operation

- Support 32-bit, dual half-word, or quad byte ops
- Example: 4 parallel byte additions
- Treat each byte as a separate 2’s complement number
- Don’t propagate the carries across byte boundaries

Modify for Segmentation

- Only modify carry propagation in global carry chain

Global Carry w/ Segment

- This method adds mux to critical path
  - Possible to move off critical path
  - By moving to start of adder
- If the op type is known early
  - For cells at start of segment
    - Change P0 definition
    - Change Cin to local carry chains

Absolute Difference

- Example: 4 parallel byte absolute differences
  - Important in MPEG encoding algorithm
- Algorithm:
  - Take two unsigned 8-bit numbers (between 0 and 255)
  - Compute |a-b|
  - Result is unsigned (between 0 and 255)

Absolute Difference

- How do we compute |a-b|?
  - We need to compute a-b and b-a and take the positive one
  - Remember that in 2’s complement, \(-x = \neg x + 1\)
  - The carry-select adder will compute a+a+1 and a+b
    - a+\neg b = a \cdot b - 1
    - Note that
      - \((b \cdot a) + 1 = a \cdot b\)
      - So
        - \((-b \cdot a) + 1 = a \cdot b - 1\)
        - or
          - \(b \cdot a = \neg (a \cdot b - 1)\)
  - So, to compute |a-b|, just choose between SUM1 or \neg SUM0 depending on the sign bit
### Sum of Absolute Differences

- Must do conditional sum for lower 8 bits
- Must further modify global carry chain to look at sign bits
  - If positive, choose `SUM1`; if negative, choose `SUMD`

![Slide from Mark Horowitz, Stanford](image1)

### Saturation

- Often in media and signal processors, saturating arithmetic is supported:
  - Don’t wrap around on overflow
  - Result should be largest (or smallest) value possible

- Examples:
  - 32-bit saturating integer add:
    - `IADD32D(0x7FFFFFFF, 0x00000001)=0x7FFFFFFF`
  - 8-bit saturating unsigned subtract:
    - `USUB8(0x02FE02FE, 0x03FE01FF)=0x00000010`

![Slide from Mark Horowitz, Stanford](image2)

### Hardware Support for Saturation

- Overflow detection
  - Example: signed addition
    - Can look at sign bits of inputs and outputs
    - Or can compute using `ovf = cinmbb & ccoutmbb`

- Overflow propagation
  - Similar to segmented, global carry chain, except for overflows

- Output muxing
  - Need a many-to-one mux for each byte to choose between:
    - `0xff, 0x00, 0x7f, 0x00` and the unsaturated value
  - Methods for speeding up saturation
    - Could probably do “carry-select saturation detection”

![Slide from Mark Horowitz, Stanford](image3)

### Simulated Performance

- Two implementations:
  - Custom circuits (using circuits from these slides)
    - 15.1 FO4 delays through integer adder
    - 9.3 FO4 delays through overflow detection and saturation
    - ~300x faster for adder only (excluding ovf det and saturation)
  - Standard cell implementation
    - ~23.5 FO4 delays through integer adder
    - ~10 FO4 delays through overflow detection and saturation
    - ~800x faster for adder only (excluding ovf det and saturation)

- Significant room for speed improvement through any of the following techniques:
  - Domino circuits
  - Faster carry-chain structures
    - e.g. carry-summed on upper half of carry chains within each group pgk

![Slide from Mark Horowitz, Stanford](image4)

### Related Results

- 8-bit Manchester carry chains are slow, no matter how you size them. If you are going to use 8-bit groups, you probably need look-ahead in that group

- Using dynamic gates is much faster than static gates but
  - Need to worry a lot more about clock skew and noise margin issues. You also need to think about power

- It is possible to move segment overhead off the critical path

- Saturation is a pain since you need to know overflow condition before you can select the correct sum
  - But you can calculate overflow early if you spend hardware

![Slide from Mark Horowitz, Stanford](image5)
Summary (from Harris/Weste)

- If they’re fast enough, use ripple-carry
  - Compact, simple
- Carry skip and carry select work well for small bit sizes (8-16)
  - Hybrids combining techniques are popular
- At 32, 64, and beyond, tree adders are much faster
  - Again, hybrids are common

Adder Summary

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Classification</th>
<th>Logic Levels</th>
<th>Max Fanout</th>
<th>Techs</th>
<th>Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry-ripple</td>
<td>N + 1</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Carry-Skip</td>
<td>N + 1</td>
<td>1</td>
<td>1</td>
<td>1.25N</td>
<td></td>
</tr>
<tr>
<td>Carry-Select</td>
<td>N + 2</td>
<td>4</td>
<td>1</td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>Carry-Lookahead</td>
<td>N + 1</td>
<td>1</td>
<td>1</td>
<td>1.5N</td>
<td></td>
</tr>
<tr>
<td>Skew</td>
<td>log₂(N+1)</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5N</td>
<td>N</td>
</tr>
<tr>
<td>Kogge-Breit</td>
<td>log₂(N+1)</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5N</td>
<td>N</td>
</tr>
<tr>
<td>Han-Carlson</td>
<td>log₂(N+1)</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5N</td>
<td>N</td>
</tr>
<tr>
<td>Leader-Fischer (N+1)</td>
<td>log₂(N+1)</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5N</td>
<td>N</td>
</tr>
<tr>
<td>Kwon et al.</td>
<td>log₂(N+1)</td>
<td>N/2 + 1</td>
<td>1</td>
<td>0.5N</td>
<td>N</td>
</tr>
</tbody>
</table>

Synthesized Adders (Harris/Weste)

- Similar to my experiment
  - But with 0.18u library, Synopsys DesignWare
  - Synopsys can map “+” to carry-ripple, carry-select, carry-lookahead, and some prefix adders
  - Fastest are tree adders with (prelayout) speeds of 7.0 and 8.5 FO4 delays for 32 and 64 bit adders

Area vs. Delay, Synthesized Adders