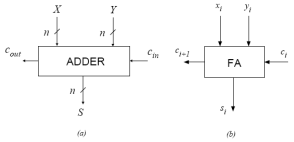
TWO-OPERAND ADDITION

Chapter two…

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 ($Z_R=(X_R+Y_R) \bmod C$)



$$x + y + c_{in} = 2^n c_{out} + s$$

$$s = (x + y + c_{in}) \bmod 2^n$$

$$c_{out} = \begin{cases} 1 \text{ if } (x + y + c_{in}) \geq 2^n \\ 0 \text{ otherwise} \end{cases}$$

$$= \lfloor (x + y + c_{in})/2^n \rfloor$$

Adder Bits

Half Adder

$S =$

$C_{out} =$

A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

Full Adder

$S =$

$C_{out} =$

A	B	C_{in}	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Adder Bits

Half Adder

$S = A \oplus B$

$C_{out} = A \wedge B$

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder

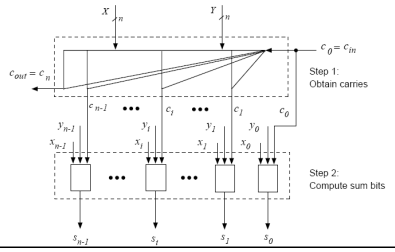
$S = A \oplus B \oplus C_{in}$

$C_{out} = MAJ(A, B, C_{in})$

A	B	C_{in}	C_{out}	S
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

Big Problem (for speed)

- Carry generation!
  - Carry at *i* depends on $j \leq i$
  - Non-trivial to do fast – lots of inputs…



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)

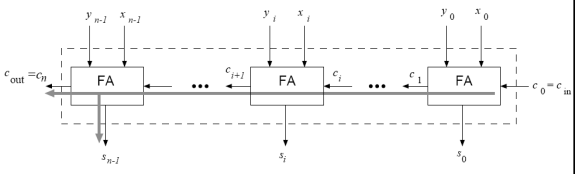
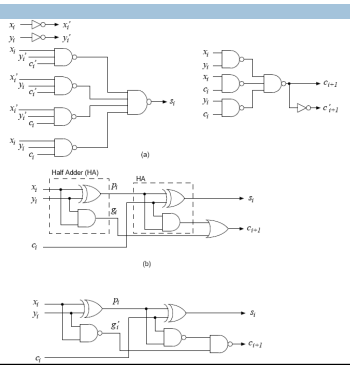


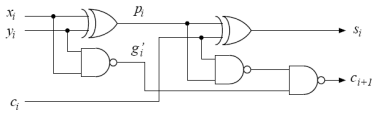
Figure 2.4: Carry-ripple adder.

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

Adder Bits (gates)



Adder Performance



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

$$T_{CRA} = t_{xor} + 2_{(n-1)}t_{NAND} + \max(2t_{NAND}, t_{XOR})$$

Adder Performance

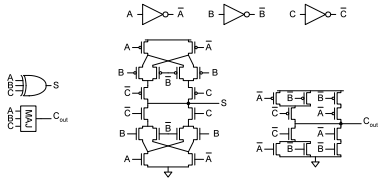
- Most standard cell libraries have a Full Adder cell as a single cell
  - Implements Full Adder function directly in nmos and pmos transistors
  - Delays should be smaller…

Adder Bits (CMOS)

$S = A \oplus B \oplus C$

$C_{out} = MAJ(A, B, C)$

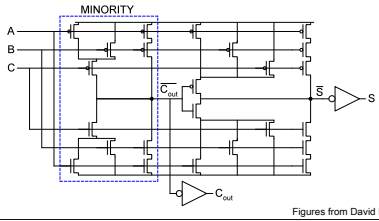
Brute Force circuit…



Figures from David Harris…

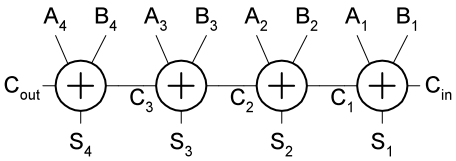
Mirror Adder

- Factor S in terms of C_{out}
  $S = ABC + (A + B + C)(\sim C_{out})$
- Critical path is usually C_{in} to C_{out} in ripple adder



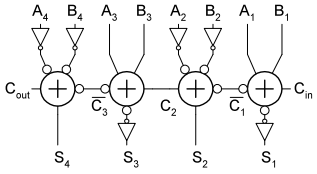
Figures from David Harris…

Connect for carry-ripple adder

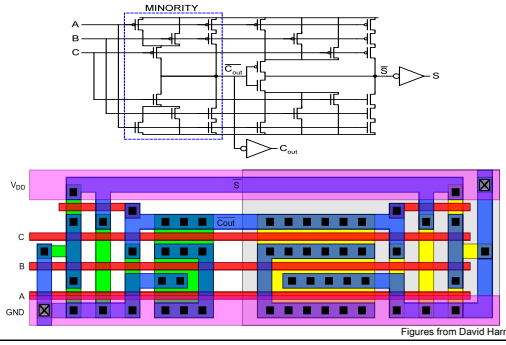


Inversions…

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



Mirror Adder

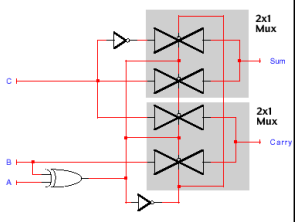


Figures from David Harris…

Build a faster circuit?

- **Truth Table**

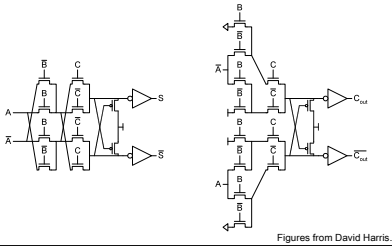
A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



- **When $A \oplus B = 0$, SUM = C, and Carry = B.**
- **When $A \oplus B = 1$, SUM = $\overline{C}$, and Carry = C.**
- **Using the 6T XOR, this full adder uses 18T.**

Build a faster circuit?

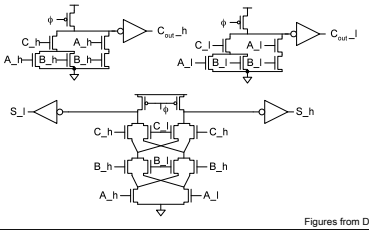
- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area



Figures from David Harris…

Build a faster circuit?

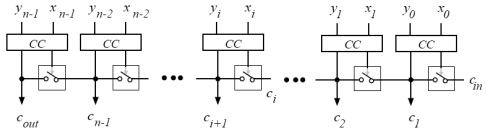
- Dual-rail domino
  - Very fast, but large and power hungry
  - Used in very fast multipliers



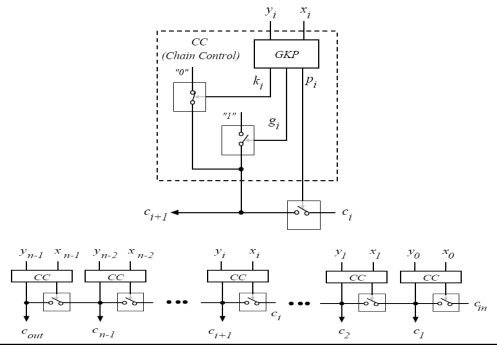
Figures from David Harris…

Build a faster carry chain?

- Manchester Carry Chain
  - Use transmission gates to make carry "wire"



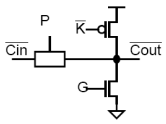
Manchester Carry control



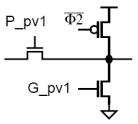
MCC

Switch-Logic:

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

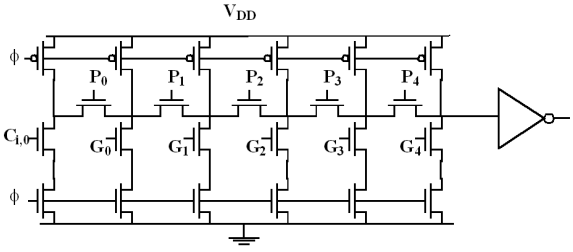


To reduce the logic needed, and the capacitance on the carry chain use precharge switch logic. Precharge the output high, and pull it low if needed. The inputs to the gate can be outputs from other domino gates (Carry is a monotonic function of P, C, K[1]



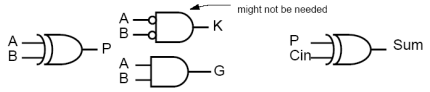
1. Need to be careful, since we will use a inverter to buffer the output before we use it. That is the reason that the switch logic is generating C_b and not C. Switching G and K will generate C directly.

Manchester Carry Chain



MCC

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 _v 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 _v signals.

Manchester Carry Delay

$$T_{SRA} = t_{sw} + (n-1)t_p + (n/m)t_{buf} + t_s$$

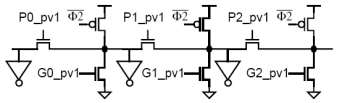
- t_{sw} is time to set all switches
- t_p is time to propagate through a switch
- t_{buf} is a buffer – need restoring buffer every m bits
- t_s computes the sum based on the carries

- 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



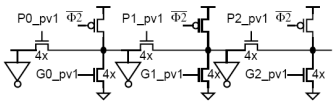
- Capacitance per stage (assuming all 4:2 devices, no diff sharing)
  3 ndiff + pdiff + Cg + inv + bit-width of wire = 12fF + 4fF + 4fF + 8fF + 8fF (30) = 36fF
- Resistance per-stage is 6.5K, so the delay is approximately .12ns * n^2, ($RCn^2/2$) where n is the number of stages directly tied together.

Slide from Mark Horowitz, Stanford

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]



- Capacitance per stage:
  2ndiff (16λ) + pdiff + Cg + inv + bit-width of wire = 32fF + 4fF + 16fF + 8fF + 8fF (30) = 68fF
- Resistance per-stage is 1.6K, delay is 0.054ns * n^2.

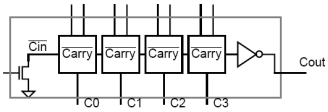
1. Make G larger since it does not hurt (diffusion is shared, and since it will be important in faster adders)

Slide from Mark Horowitz, Stanford

Sizing MCC

To limit the effect of the n^2 term, break carry chain into sections.

- Each section is about 4 stages long (3 stages might be better)
- Between sections the carry is buffered.

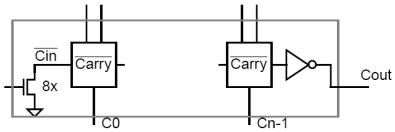


- The buffering makes the delay linear with the number of bits
- But the carry stills needs to flow through all the carry chains.

Slide from Mark Horowitz, Stanford

Buffered Carry Chains

What is the 'right' number of stages?



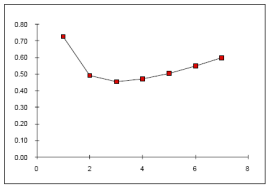
Assume first transistor is 8x min, and final inverter is minimum

Delay is the inverter delay (Cout rising) plus the delay of the chain including the resistance of the initial 8x transistor.

Slide from Mark Horowitz, Stanford

Timing MCC

Stages	Total Delay	Delay per bit
1	0.61	0.61
2	0.82	0.42
3	1.15	0.38
4	1.58	0.39
5	2.12	0.42
6	2.77	0.46



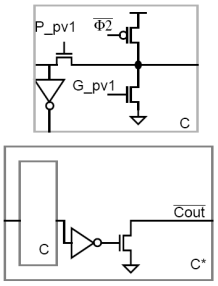
So for these sizing, the optimal number in a stage is around 4, and the average delay per bit is around 0.4 ns. This is not optimally sized (pMOS in final inverter should be larger) but it is probably close.

Slide from Mark Horowitz, Stanford

Layout of MCC

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

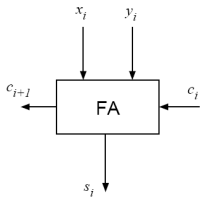
P,G gen ↑	C ↑	XOR↑
P,G gen ↓	C ↑	XOR↓
P,G gen ↑	C ↑	XOR↑
P,G gen ↓	C* ↑	XOR↓
P,G gen ↑	C ↑	XOR↑
P,G gen ↓	C ↑	XOR↓
P,G gen ↑	C ↑	XOR↑
P,G gen ↓	C* ↑	XOR↓



Slide from Mark Horowitz, Stanford

Back to Adder Bits

- Revisit the full adder:



Xi	Y_i	C_i	C_{i+1}	S_i	Comment
0	0	0	0	0	Kill
0	0	1	0	1	Kill
0	1	0	0	1	Propagate
0	1	1	1	0	Propagate
1	0	0	0	1	Propagate
1	0	1	1	0	Propagate
1	1	0	1	0	Generate
1	1	1	1	1	Generate

Back to Adder Bits

- Revisit the full adder:

Case 1 (Kill): $k_i = x_i' y_i' = (x_i + y_i)'$
Case 2 (Propagate): $p_i = x_i \oplus y_i$
Case 3 (Generate): $g_i = x_i y_i$

$C_{i+1} = g_i + p_i c_i = x_i y_i + (x_i \oplus y_i) c_i$

Consider $a_i = p_i + g_i$

[note $g_i + p_i c_i = g_i + (g_i + p_i) c_i$]

then $c_{i+1} = g_i + a_i c_i$

Xi	Y_i	C_i	C_{i+1}	S_i	Comment
0	0	0	0	0	Kill
0	0	1	0	1	Kill
0	1	0	0	1	Propagate
0	1	1	1	0	Propagate
1	0	0	0	1	Propagate
1	0	1	1	0	Propagate
1	1	0	1	0	Generate
1	1	1	1	1	Generate

Carry Chains

- Two types
  - 1-carry chain and 0-carry chain
  - 1-carry always starts at g_i=1 (or c_{in} = 1), and propagates over consecutive positions p_i=1
  - 0-carries start at k_i=1 position (or c_{in} = 0)…

i	9	8	7	6	5	4	3	2	1	0
x_i	1	0	1	0	1	1	1	1	0	0
y_i	0	0	0	1	0	1	0	0	1	0
	p	k	p	p	p	g	p	p	p	k
	a		a	a	a	a	a	a	a	
c_{i+1}	0 ←	0	1 ←	1 ←	1 ←	1 ←	1	0 ←	0 ←	0 ←	0

Group Carries

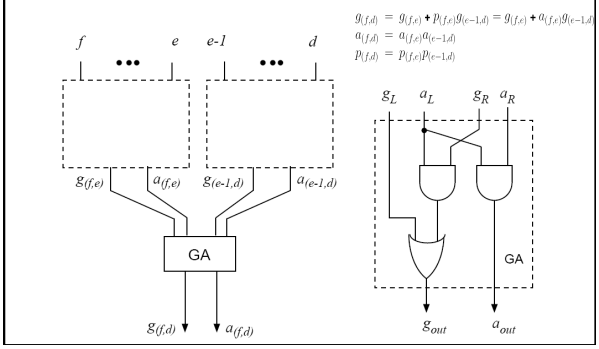
- Carry equation can be generalized to groups of bits

$$c_{j+1} = g_{(j,i)} + p_{(j,i)} c_i = g_{(j,i)} + a_{(j,i)} c_i$$
$$c_{j+1} = g_{(j,0)} + p_{(j,0)} c_0 = g_{(j,0)} + a_{(j,0)} c_0$$

  - Combine subranges recursively

$$g_{(f,d)} = g_{(f,e)} + p_{(f,e)} g_{(e-1,d)} = g_{(f,e)} + a_{(f,e)} g_{(e-1,d)}$$
$$a_{(f,d)} = a_{(f,e)} a_{(e-1,d)}$$
$$p_{(f,d)} = p_{(f,e)} p_{(e-1,d)}$$

Group Carries



Example (2.1)

- find bit 13 of the following sum

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

Example (2.1)

- first compute pkg for each bit

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

Example (2.1)

- now combine in groups

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

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

Example (2.1)

- extend groups

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

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

Example (2.1)

- extend groups to whole range

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

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

Example (2.1)

- Now you can compute c_{13}

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

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

Example (2.1)

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

x = 0110|0010|1100|0011
y = 1011|1101|0001|1110
p|pppp|ppkp|ppgp

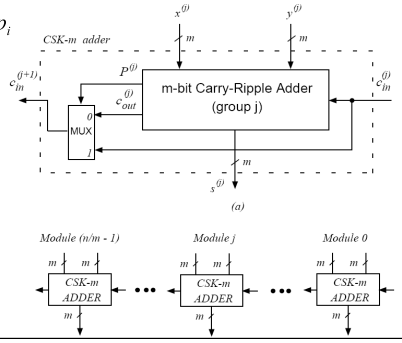
$$k_{(12-0)} = k_{(12-8)} + p_{(12-8)}k_{(7-0)} = 1$$
$$c_{13} = g_{(12-0)} + p_{(12-0)}c_{in} = 0$$
$$s_{13} = x_{13} \oplus y_{13} \oplus c_{13} = 1 \oplus 1 \oplus 0 = 0$$

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 Adder

$$p^{(j)} = \overset{m-1}{\underset{i=0}{AND}} p_i$$



Carry Skip example

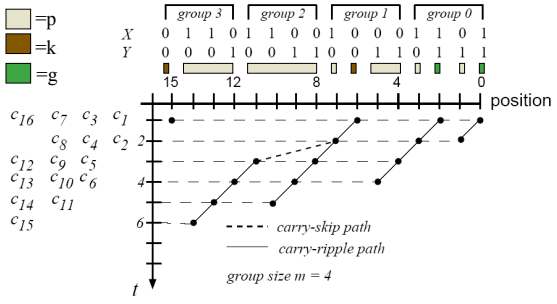
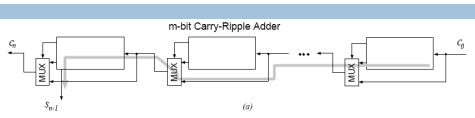
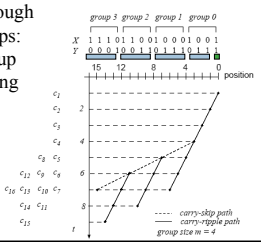


Figure 2.8: Carry chains in carry-skip adder: A case with several carry chains.

Carry Skip worst case



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



Carry Skip delay

$$T_{CSK} = mt_c + t_{mux} + (\frac{n}{m} - 2)t_{mux} + (m-1)t_c + t_s$$
$$= (2m-1)t_c + (\frac{n}{m} - 1)t_{mux} + t_s$$

- 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…

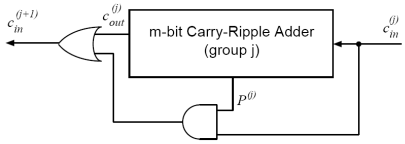


Figure 2.10: Carry-skip adder using AND-OR for bypass

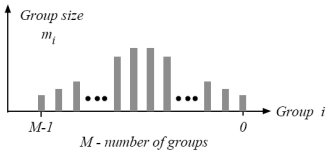
Group Size

- 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:

$$m_{opt} = (\frac{t_{mux}}{2t_c}n)^{1/2} \quad \text{(minimum delay)}$$
$$T_{opt} \approx (8t_{mux}t_c n)^{1/2}$$

Carry Skip with different m

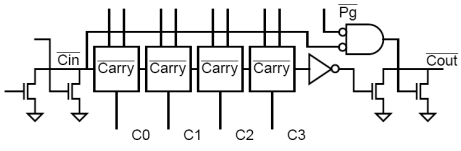
- If you vary the group size with the groups at the ends shorter than the groups in the middle, you can speed things up
  - N=60, t_c=t_{mux}=t_s=1δ
  - M=6, T_{csk}=21 δ
  - M=4,5,8,7,8,8,7,5,5,4, T_{csk}=17 δ



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.



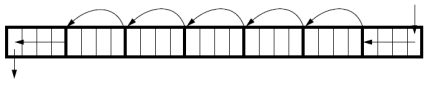
Slide from Mark Horowitz, Stanford

Carry Skip – Another View

All groups can calculate Pg at the same time (in parallel)

Worse-case is when carry needs to propagate through all bits

- Since we precomputed Pg, that path is now much shorter
  Hop around groups, rather than through them
- Critical path is now through one local carry chain, then through a number of bypass gates, then back into a final local carry chain.

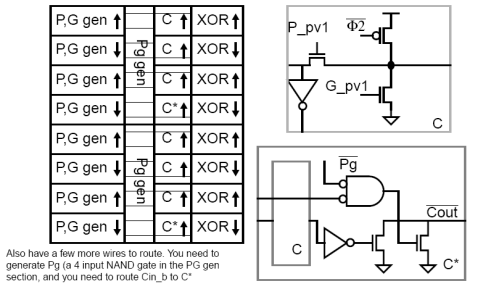


- This improvement did not cost much hardware.

Slide from Mark Horowitz, Stanford

Carry Skip - Layout

Layout of a bypass adder is almost the same, C* gets a more stuff:



Also have a few more wires to route. You need to generate Pg (a 4 input NAND gate in the PG gen section, and you need to route Cin_b to C*

Slide from Mark Horowitz, Stanford

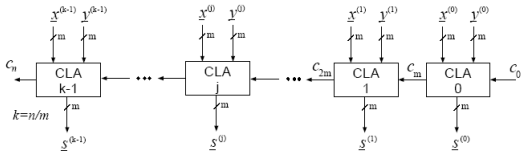
Carry Lookahead

- 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

$$x^{(i)} = \sum_{v=0}^{i} x_v 2^v \qquad c_i = 1 \text{ if } (x^{(i-1)} + y^{(i-1)} + c_0) \geq 2^i$$

  - This is a switching function of $2i+1$ variables

Carry Lookahead



$$T_{1-CLA} = \frac{n}{m} t_{group} + t_s$$

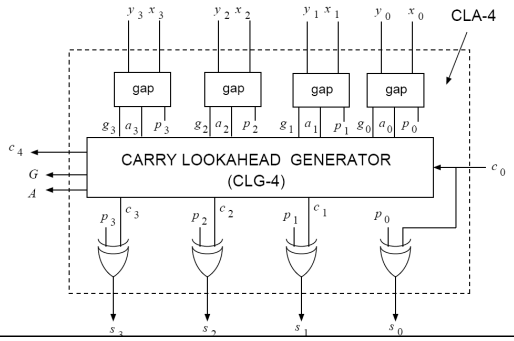
Carry Lookahead equations

- Remember: $C_i = G_i + P_i C_i$
- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1 C_1$
  $= G_1 + P_1(G_0 + P_0 C_0)$
  $= G_1 + P_1 G_0 + P_1 P_0 C_0$
- $C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$
  $+ P_3P_2P_1P_0C_0$
- Or $C_4 = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0 C_0)))$

Carry Lookahead equations

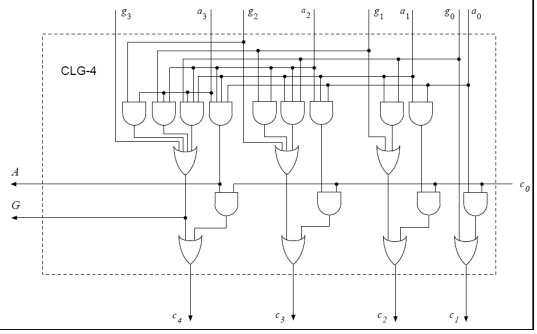
- Remember: $C_i = G_i + A_i C_i$
- $C_1 = G_0 + A_0 C_0$
- $C_2 = G_1 + A_1 C_1$
  $= G_1 + A_1(G_0 + A_0 C_0)$
  $= G_1 + A_1 G_0 + A_1 A_0 C_0$
- $C_3 = G_2 + A_2G_1 + A_2A_1G_0 + A_2A_1A_0C_0$
- $C_4 = G_3 + A_3G_2 + A_3A_2G_1 + A_3A_2A_1G_0$
  $+ A_3A_2A_1A_0C_0$
- Or $C_4 = G_3 + A_3(G_2 + A_2(G_1 + A_1(G_0 + A_0 C_0)))$

CLA-4 Module

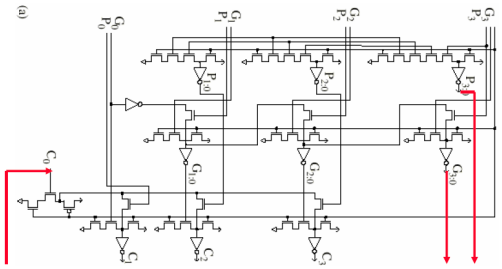


CLG-4 Module

$$T_{1-CLA} = t_{a,g} + \frac{n}{m} t_{c\lg} + t_s$$



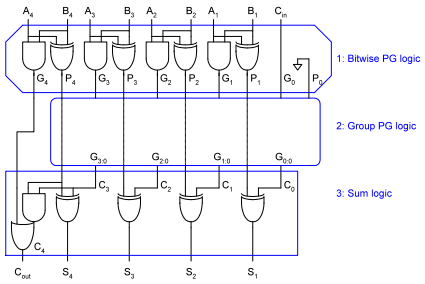
Dynamic Logic for 4-bit CLG



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

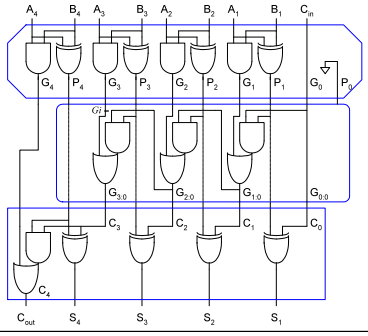
Slide from Mark Horowitz, Stanford

Carries - Another View



Carry Ripple revisited

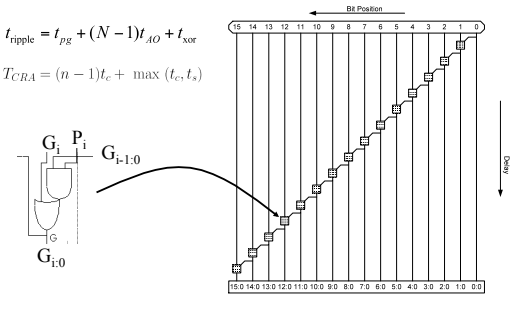
$$G_{i:0} = G_i + P_i G_{i-1:0}$$



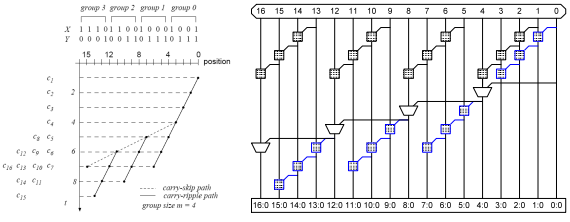
Carry Ripple revisited

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

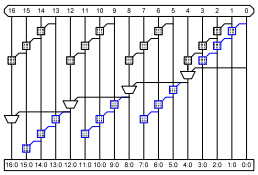
$$T_{CRA} = (n-1)t_c + \max(t_c, t_s)$$



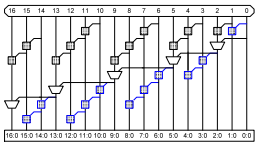
Carry Skip revisited



Carry Skip revisited



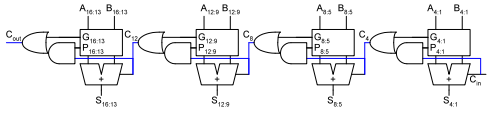
Fixed group size
(4,4,4,4)



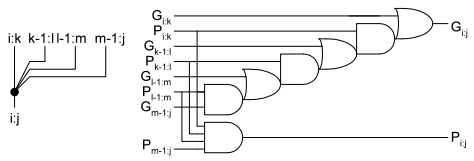
Variable group size
(2,3,4,4,3)

Carry Lookahead revisited

- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel.
- Uses higher-valency cells with more than two inputs

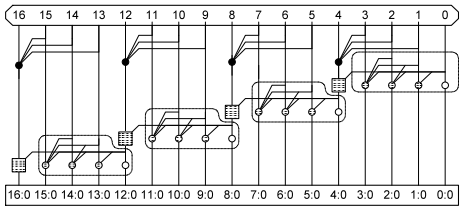


Higher Valency Cell



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

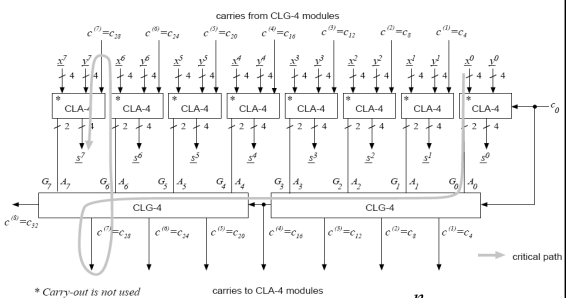
CLA/Manchester adder



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
  - $C^{(1)} = G_0+A_0C_0$
    $C^{(2)} = G_1+A_1G_0+A_1A_1C_0$
    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)



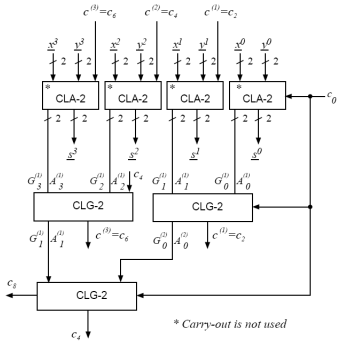
$$T_{2-CLA} = t_{a,g} + t_{A,G} + \frac{n}{pm} t_{c\lg} + t_{c\lg} + t_s$$

Three-level CLA

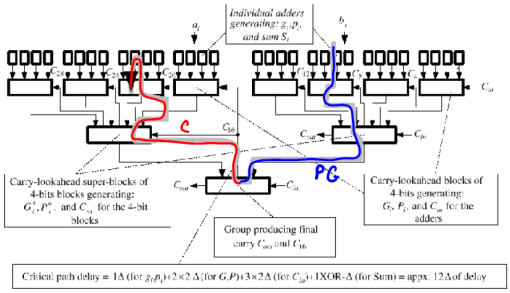
- Extend to three or more levels by having lookahead between sections
  - First compute a_i, p_i, g_i
  - L-1 level of CLA to compute As and Gs
  - n/m^L CLGs connected in ripple to compute carries of bits
  - One level of XOR to compute the sum

$$T_{L-CLA} = t_{a,g} + (L-1)t_{A,G} + \frac{n}{m^L}t_{clg} + (L-1)t_{clg} + t_s$$

Three-level CLA (n=8, m=2)



CLA Critical Path



Prefix Adders (Tree Adders)

- 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 $g_{(i-1,-1)}$ for all i
  - Cells connected to produce g signals that span an increasing number of bits

PG (PA) cell

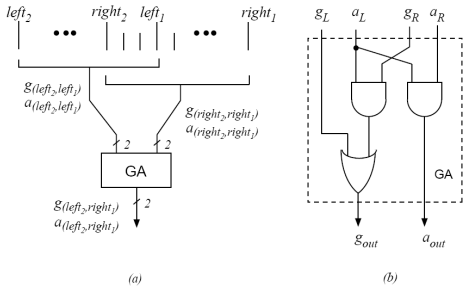
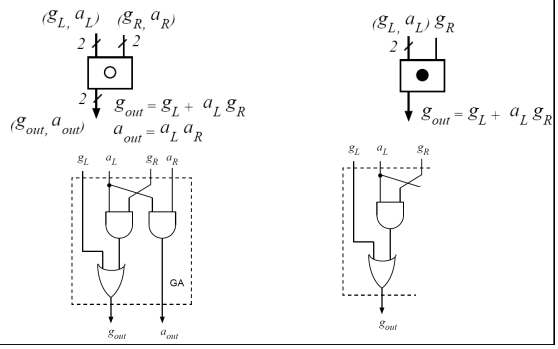


Figure 2.17: Composition of spans in computing (g, a) signals.

Overlapping Ranges

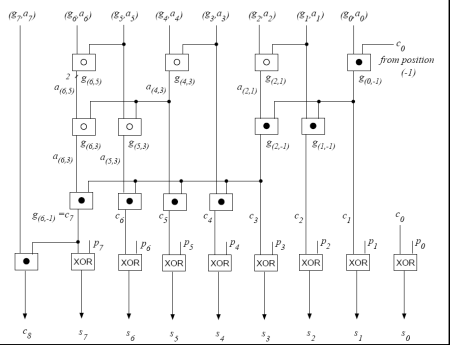
- Starting with g,a of each bit, first level generates g,a for two bits, then four, etc.
  - If right input spans bits [right2,right1] , and left spans [left2,left1], with right2+1 >= left1
  - Then output spans bits [left2,right1]
  - For example right[5,2] and left[8,4] means output spans bits [8,2]

PG (PA) Cells



8-bit Prefix Adder

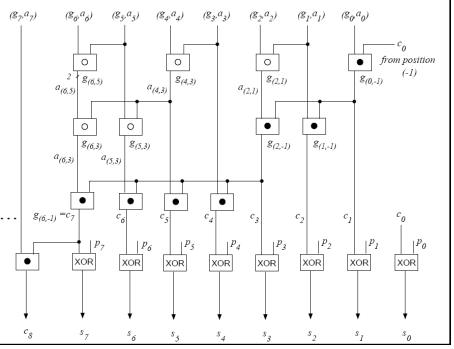
$$T_{PA} = t_{a,g} + \log_2(n)t_{cell} + t_{XOR}$$



8-bit Prefix Adder

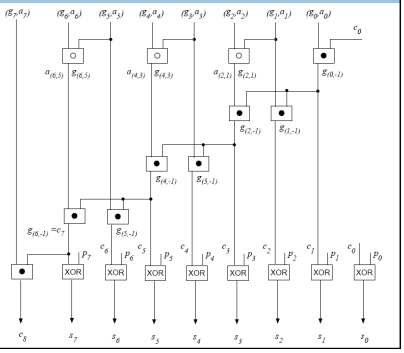
$$T_{PA} = t_{a,g} + \log_2(n)t_{cell} + t_{XOR}$$

Fanout can
Be an issue…



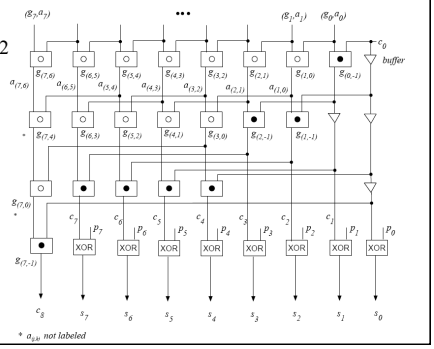
8-bit Prefix Adder

Lower fanout
Increase levels

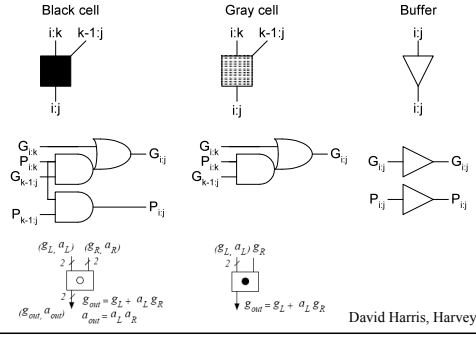


8-bit prefix adder

Max fanout 2
Min levels

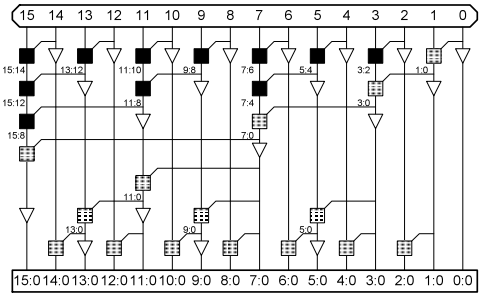


Another View of Prefix Adders

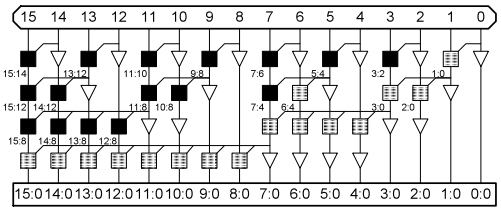


David Harris, Harvey Mudd

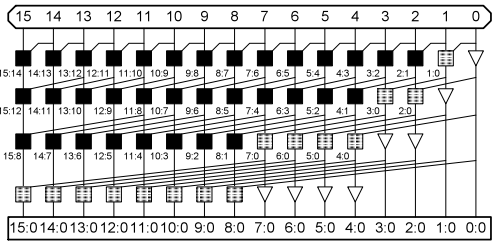
Brent-Kung



Sklansky Adder



Kogge-Stone Adder

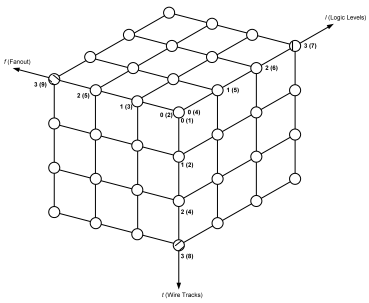


Tree Adder Taxonomy

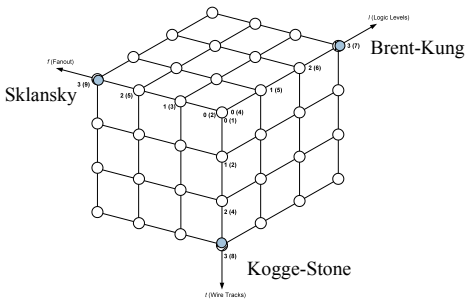
- Ideal N-bit tree adder would have
  - $L = \log N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (l, f, t)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- Known tree adders sit on plane defined by

$$l + f + t = L\text{-}1$$

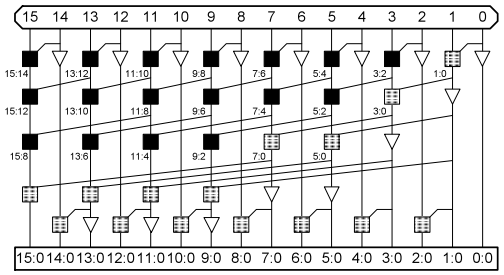
Tree Adder Taxonomy



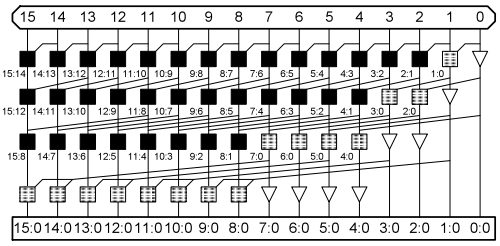
Tree Adder Taxonomy



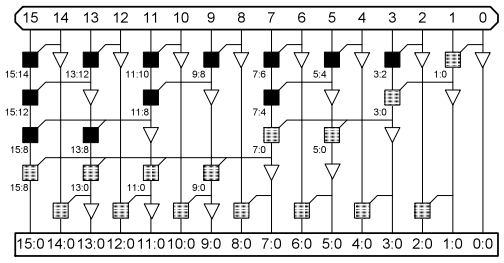
Han-Carlson



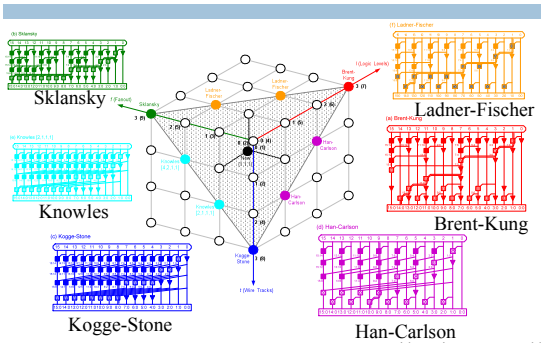
Knowles [2, 1, 1, 1]



Ladner-Fischer



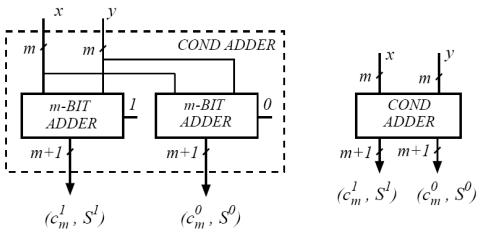
Taxonomy Revisitied



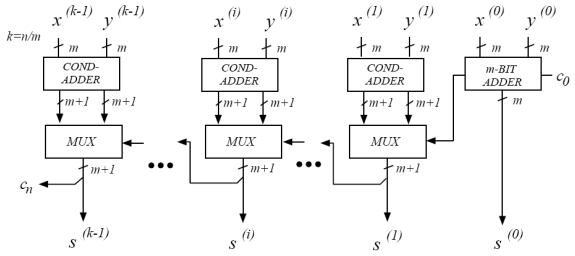
Conditional Sum Adder

- 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

Conditional Sum Adder



Carry-Select Adder



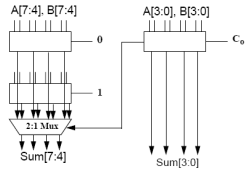
$$T_{CSEL} = t_{add,m} + \left(\frac{n}{m} - 1\right) t_{mux}$$

Carry-Select – Another View

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)

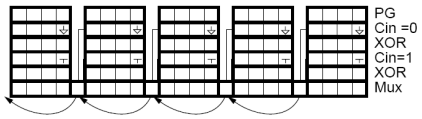
When Cin is known, it is only a Mux delay to get Cout and all the Sums for the group.



Slide from Mark Horowitz, Stanford

Carry-Select - Layout

A larger adder would look something like this:



Notice that the PG logic can be shared with both carry chains

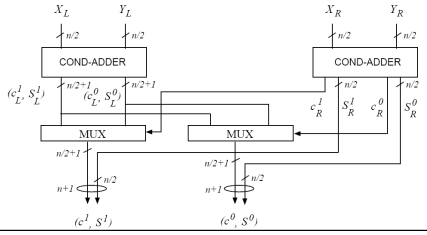
Critical path is first carry chain and then n mux delay

What is the optimal block size for a carry select adder? (Hint they are not all the same)

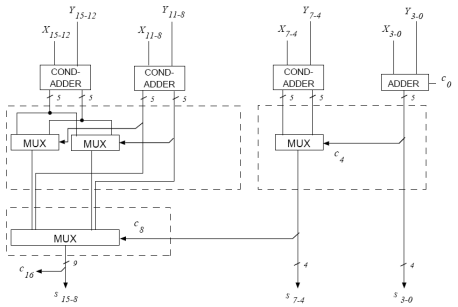
Slide from Mark Horowitz, Stanford

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

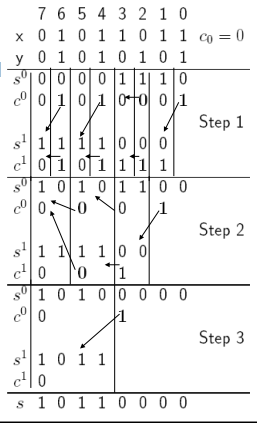


Example

Step 1: Compute all the bit results

Step 2: Use the known results to select the next groups…

$$T_{COND-SUM} = t_{add-m} + (\log_2\left(\frac{n}{m}\right))t_{mux}$$



Pipelined Adders

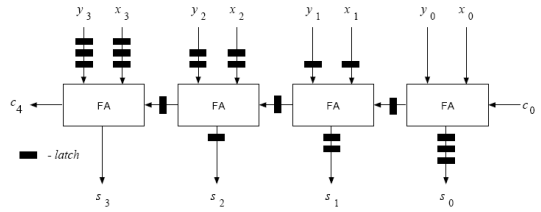


Figure 2.26: Pipelined carry-ripple adder (for group size of 1 and $n = 4$)

Variable Time Adder

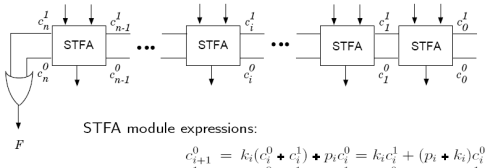
- Carry Completion Sensing Adder
  - Encode the carry in a form that lets you tell when it's finished
  - When all carry chains have finished, the add is finished
  - One choice – dual-rail encoding

Two carry signals:

c_i^0 *zero carry* c_i^1 *one carry*

with coding:

c_i^0	c_i^1	c_i
0	0	not determined (yet)
0	1	1
1	0	0
1	1	does not occur



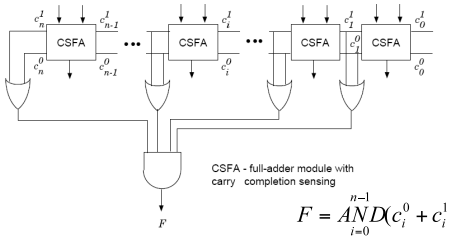
STFA module expressions:

$$c_{i+1}^0 = k_i(c_i^0 + c_i^1) + p_i c_i^0 = k_i c_i^1 + (p_i + k_i)c_i^0$$
$$c_{i+1}^1 = g_i(c_i^0 + c_i^1) + p_i c_i^1 = g_i c_i^0 + (p_i + g_i)c_i^1$$
$$s_i = p_i \oplus c_i^1$$
$$k_i = x_i' y_i', \quad g_i = x_i y_i, \quad p_i = x_i \oplus y_i$$

Addition time: based on *actual* delays, not worst-case

$$T_{var-1} = \sum_{i=0}^{n-1} t_{c,i}$$

Variable Time Adder



CSFA - full-adder module with carry completion sensing

$$F = \overline{AND}_{i=0}^{n-1}(c_i^0 + c_i^1)$$

Figure 2.28: Variable-time adder: Type 2.

Carry chains initiated simultaneously

CSFA module expressions:

$$c_{i+1}^0 = k_i + p_i c_i^0, \quad c_{i+1}^1 = g_i + p_i c_i^1$$

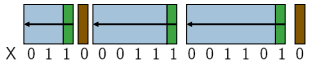
Variable Time Adder

X 0 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0
Y 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0
\+ a a a b c c c c c d d d d d d e Prop.chains

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



X 0 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0
Y 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0
\+ a a a b c c c c c d d d d d d e Prop.chains

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:

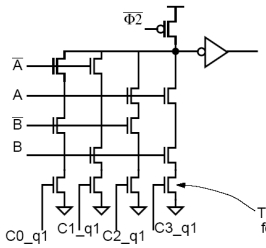
Build a separate logic unit and mux together the outputs. This is probably the fastest solution, since you don't slow down the add critical path, but it will take more area.

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 variable. This is easy -- just use a mux. To reduce control lines, I will use a precharge mux.



By setting the right values on the control lines, this block can generate any logic function

Exor = 0 1 1 0

And = 0 0 0 1

Or = 0 1 1 1

These transistors can be shared for all the bit slices.

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.

- A - B = A + $\overline{B}$ + 1, where $\overline{B}$ 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 = A $\overline{B}$ + B $\overline{A}$; G = A B
- For substraction:

  P = A B + $\overline{B}$ $\overline{A}$; 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 _v1)
- 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)

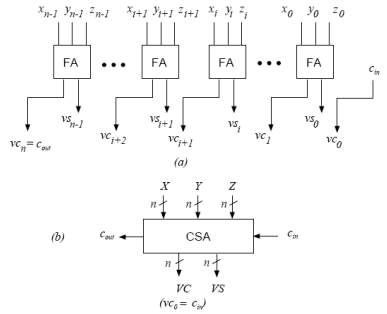
$$x + y + z = vc + vz = v$$

  - Several combinations of vc and vs represent the same result

Carry Save Adder

- If you want to convert back to conventional numbers, add *vs* and *vc*
  - Because there 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 Adder



Carry Save Example

X	0	1	1	1	0	1	0	0
Y	0	0	1	1	1	0	1	1
Z	1	0	1	0	1	0	1	0
VS	1	1	1	0	0	1	0	1
(c_{out}, VC)	0	0	1	1	1	0	1	0	1
digit value	0	1	2	2	1	0	2	0	2

Carry Save Example

	X	0	1	1	1	0	1	0	0	116
	Y	0	0	1	1	1	0	1	1	59
	Z	1	0	1	0	1	0	1	0	170
229	VS	1	1	1	0	0	1	0	1	=345
117	(c_{out}, VC)	0	0	1	1	1	0	1	0	1	Cin
	digit value	0	1	2	2	1	0	2	0	2	

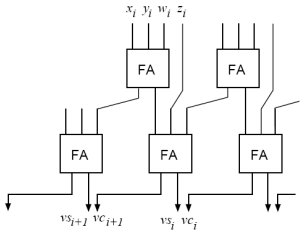
256 128 64 32 16 8 4 2 1

128 + 128 + 64 +16 + 0 + 8 + 2 = 346

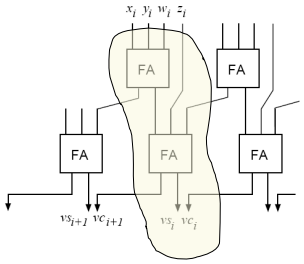
Carry Save

- What if two operands are both carry-save?
  - Then each operand is in Xs Xc form
  - So, you need a [4:2] adder instead of a [3:2]
  - Combine four vectors into two…
  - Still no carries!
  - Answer is still in redundant carry-save form

Carry Save [4:2]



Carry Save [4:2]



[4:2] Compressor Adder

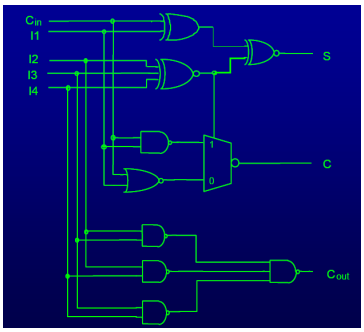
Note that even though it looks like carry is propagated, the Cout from each [4:2] cell is computed directly from the A and B inputs…

4:2 compressor cell

Inputs				Cin=0		Cin=1		Cout
A	B	C	D	C	S	C	S	
0	0	0	0	0	0	0	1	0
0	0	0	1					
0	0	1	0	0	1	1	0	0
0	1	0	0					
1	0	0	0					
0	0	1	1					
0	1	1	0					
1	1	0	0	0	0	0	1	1
0	1	0	1					
1	0	1	0					
1	0	0	1					
1	1	1	0					
1	1	0	1					
1	0	1	1	0	1	1	0	1
0	1	1	1					
1	1	1	1	1	0	1	1	1

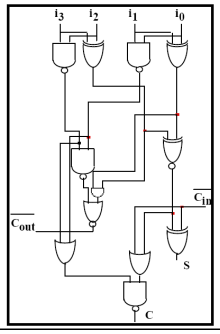
4:2 compressor cell

Nagamatsu, Toshiba



[4:2] Compressor Cell

Navi and Etiemble



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
    - Vs is represented in radix r
    - Vc has one bit per radix-r digit

Radix-8 Carry Save

XS	1	0	1	1	0	1	1	0	0
XC			1			1			0
Y	0	1	0	0	0	1	1	1	1
VS	0	0	0	1	1	1	0	1	1
(c_{out},VC) 1			0			1			0

Radix-8 Carry Save

	512			64			8			1				
XS	1	0	1	1	0	1	1	0	0	4_8				
XC			1			1			0	0_2	436			
Y	0	1	0	0	0	1	1	1	1	7_8	217			
VS	0	0	0	1	1	1	0	1	1	13_8	↓			
(c_{out},VC) 1			0			1			0					
	1*512			0			(7+1)*8			(3+0)*1	=	579		

Radix-8 Carry Save

XS	1	0	1	1	0	1	1	0	0
XC			1			1			0
Y	0	1	0	0	0	1	1	1	1
VS	0	0	0	1	1	1	0	1	1
(c_{out},VC) 1			0			1			0

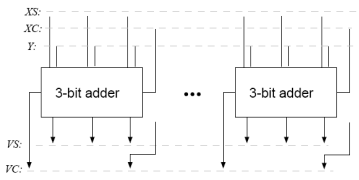


Figure 2.33: Radix-8 carry-save adder.

Signed Digit Adders

- Another form of redundant digit representation
  - Uses signed-digit representation (redundant)

$$x = \sum_{0}^{n-1} x_i r^i$$

with digit set

$$D = \{-a, \ldots, -1, 0, 1, \ldots, a\}$$

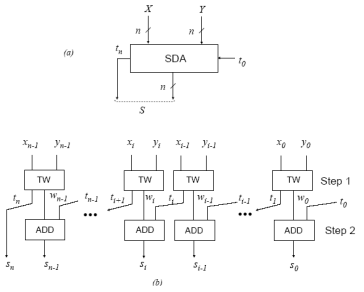
  - Limits carry propagation to next position
  - Addition algorithm:

Step 1: $x + y = w + t$
$x_i + y_i = w_i + r t_{i+1}$

Step 2: $s = w + t$
$s_i = w_i + t_i$

  - No carry produced in Step 2

Signed Digit Adder



Signed Digit Adder

Case A : two SD operands; result SD

Step 1:

$$(t_{i+1},\ w_i) = \begin{cases} (0,\ x_i + y_i) & \text{if } -a+1 \le x_i + y_i \le a-1 \\ (1,\ x_i + y_i - r) & \text{if } x_i + y_i \ge a \\ (-1,\ x_i + y_i + r) & \text{if } x_i + y_i \le -a \end{cases}$$

- 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

Scheme	Delay proportional to	Area proportional to
Linear structures:		
Carry ripple	n	n
Carry lookahead (one level)	n/m	$(k_m m)(n/m) = k_m n$
Carry select (one level)	n/m	$(k_m m)(n/m) = k_m n$
Carry skip (one level)	$\sqrt{n}$	n
Logarithmic structures:		
Carry lookahead (max. levels)	$2\log_m n$	$(k_m m)(n/m) = k_m n$
Prefix	$\log_m n$	$((k_m m)\log_m n)n$
Conditional sum	$\log_2(n/m)$	$(k_m + \log_2(n/m))n$
Completion signal (avg. delay)	$(\log_2 n)/m$	$k_m m(n/m) = k_m n$
Redundant	$const.$	n

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 Cin=0
    - One assuming Cin=1

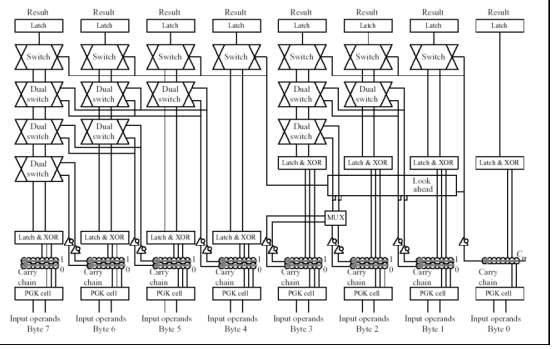
Alpha 21064

- 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

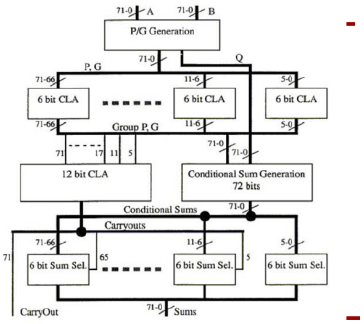
- Finally, Carry Select used to produce the most significant 32 bits.
  - Final selection done using NMOS carry-select byte-wide muxes
- Also apparently pipelined with a row of latches after the lookahead…

Alpha 21064



72-bit Pentium II Adder

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



MAH

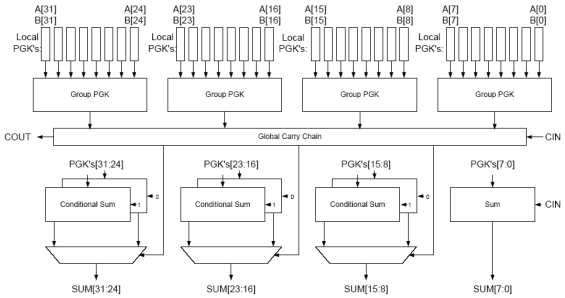
Slide from Mark Horowitz, Stanford

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

Slide from Mark Horowitz, Stanford

Adder from Imagine



Slide from Mark Horowitz, Stanford

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

Slide from Mark Horowitz, Stanford

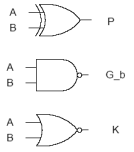
Adder from Imagine

- Local PGK's:
  - Convert input operands into Propagate (P), Generate (G), Kill(K)
- Group PGK's:
  - Determine P,G,K for groups of 8 bits
- Global carry chain:
  - Compute cin[8], cin[16], cin[24], cout(cin[32]) from group PGK's and cin
- Conditional sums:
  - Compute 8-bit sum for cin=0 and 8-bit sum for cin=1 as soon as PGK's are known
- Final Mux:
  - Use cin's from global carry chain to select conditional sums

Slide from Mark Horowitz, Stanford

Local PGK Logic (Imagine)

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



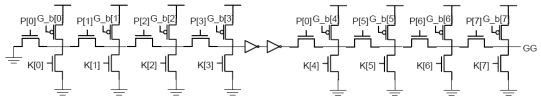
- Size gates to fan-out to four carry chains
- Note: To do A-B, use ~B here

Slide from Mark Horowitz, Stanford

Group PKG (Imagine)

Manchester Carry Chains.

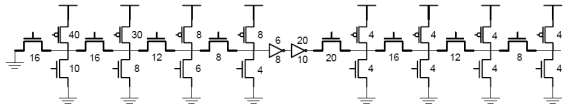
- Usually dynamic, but still works with static logic
- Group PGK's:
  - GP = P[7].P[6].P[5].P[4].P[3].P[2].P[1].P[0]
  - GG = G[7] + G[6].P[7] + G[5].P[6].P[7]+…
  - GK = ~(GG+GP)
- Use Carry chains
- Example for GG (group generate):



Slide from Mark Horowitz, Stanford

Carry Chain Sizing

- Minimize size of transistors not on critical path
- Taper sizes along carry chain
  - Reduces diffusion capacitance



Slide from Mark Horowitz, Stanford

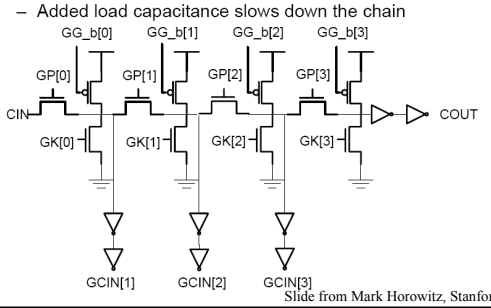
Static Carry Chains

- 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 α) makes the parasitic delay linear
  - But still does not make them fast
- Even though this chain is static logic
  - 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

Slide from Mark Horowitz, Stanford

Global Carry Chain

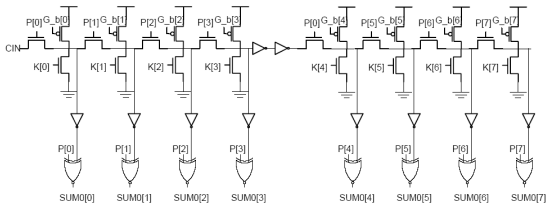
- Must fan out GCIN[3:1] to 8 muxes
  - Added load capacitance slows down the chain



Slide from Mark Horowitz, Stanford

Conditional Sums

- Use same carry chain
- Do two of these (one for cin=0, one for cin=1):



- Mux SUM0[7:0] and SUM1[7:0] with output of global carry chain

Slide from Mark Horowitz, Stanford

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

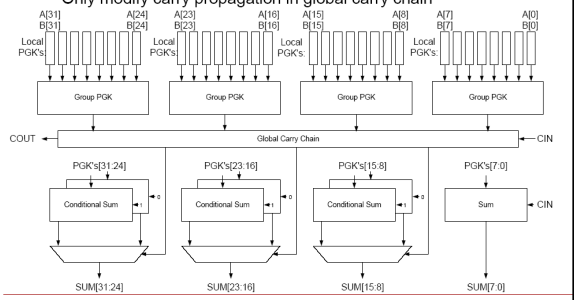
- Support 32-bit, dual half-word, or quad byte ops
- Example: 4 parallel byte additions
- Treat each byte as a separate 2's complement number
- Don't propagate the carries across byte boundaries

	1 4	0 2	F F	F F
+	F E	0 2	0 1	0 2
	1 2	0 4	0 0	0 1

Slide from Mark Horowitz, Stanford

Modify for Segmentation

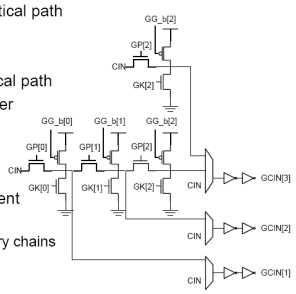
- Only modify carry propagation in global carry chain



Slide from Mark Horowitz, Stanford

Global Carry w/ Segment

- This method adds mux to critical path
- We can improve on this
  - 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 P, G definition
    - Change Cin to local carry chains



Slide from Mark Horowitz, Stanford

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)

Slide from Mark Horowitz, Stanford

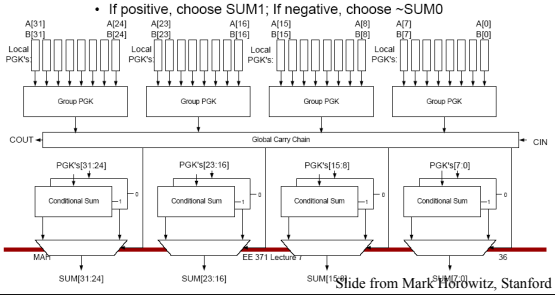
Absolute Difference

- 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 = ~x + 1
  - The carry-select adder will compute a+~b+1 and a+~b
    a+~b+1 = a-b
    a+~b = a - b - 1
  - Note that
    ~(b - a) + 1 = a - b
    so
    ~(b - a) = a - b - 1
    or
    b - a = ~(a - b - 1)
- So, to compute |a-b|, just choose between SUM1 or ~SUM0 depending on the sign bit

Slide from Mark Horowitz, Stanford

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 ~SUM0



Slide from Mark Horowitz, Stanford

Saturation

- Often in media and signal processors, saturating arithmetic is supported:
  - Don't wraparound on overflow
  - Result should be largest (or smallest) value possible
- Examples:
  - 32-bit saturating integer add:
    - IADDS32(0x7FFFFFFF,0X00000001)=0x7FFFFFFF
  - 8-bit saturating unsigned subtract:
    - USUBS8(0x02FE02FE,0x03FE01FF)=0x00000100

Slide from Mark Horowitz, Stanford

Hardware Support for Saturation

- Overflow detection
  - Example: signed addition
    - Can look at sign bits of inputs and outputs
    - Or can compute using ovf = cinmsb⊕coutmsb
- 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,0x80 and the unsaturated value
- Methods for speeding up saturation
  - Could probably do "carry-select saturation detection"

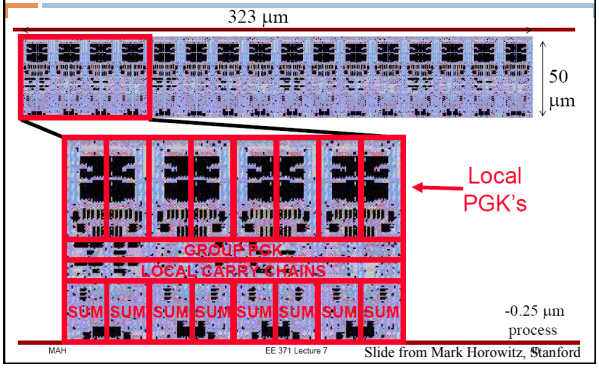
Slide from Mark Horowitz, Stanford

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
    - ~3000λ x 500λ 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
    - ~8000λ x 800λ 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-select on upper half of carry chains within each group pgk

Slide from Mark Horowitz, Stanford

Adder Layout



Slide from Mark Horowitz, Stanford

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

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 10.3 Comparison of adder architectures

Architecture	Classification	Logic Levels	Max Fanout	Tracks	Cells
Carry-Ripple		$N-1$	1	1	N
Carry-Skip ($n = 4$)		$N/4 + 5$	2	1	$1.25N$
Carry-Increment ($n = 4$)		$N/4 + 2$	4	1	$2N$
Carry-Increment (variable group)		$\sqrt{2N}$	$\sqrt{2N}$	1	$2N$
Brent-Kung	$(L-1, 0, 0)$	$2\log_2 N - 1$	2	1	$2N$
Sklansky	$(0, L-1, 0)$	$\log_2 N$	$N/2 + 1$	1	$0.5\,N\log_2 N$
Kogge-Stone	$(0, 0, L-1)$	$\log_2 N$	2	$N/2$	$N\log_2 N$
Han-Carlson	$(1, 0, L-2)$	$\log_2 N + 1$	2	$N/4$	$0.5\,N\log_2 N$
Ladner Fischer ($l = 1$)	$(1, L-2, 0)$	$\log_2 N + 1$	$N/4 + 1$	1	$0.25\,N\log_2 N$
Knowles [2,1,…,1]	$(0, 1, L-2)$	$\log_2 N$	3	$N/4$	$N\log_2 N$

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

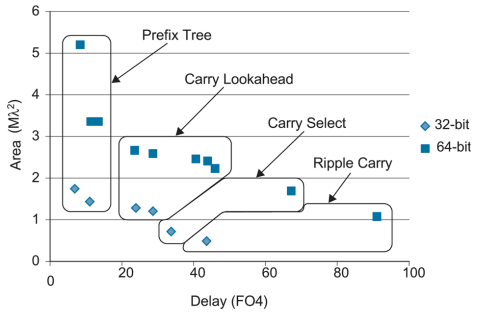


FIG 10.47 Area vs. delay of synthesized adders