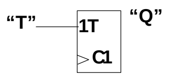
Logarithmic Circuits

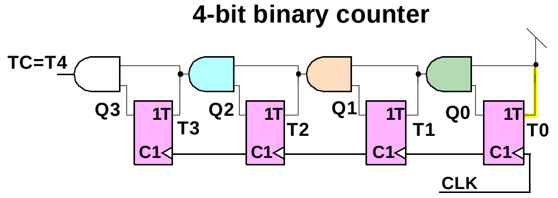
Binary Up Counter



Toggle Flip Flop

@(posedge clk)
- If T=1, toggle output Q
- If T=0, hold old Q

4-bit binary counter

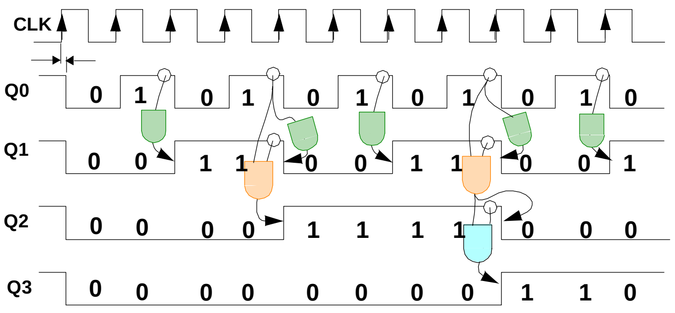


Simplest Counter Circuit Uses T Flip-Flops

The circles show how the toggle signals are defined by ANDs:

$T1 = Q_0$
$T2 = Q_1Q_0$
$T3 = Q_2Q_1Q_0$
$TC = Q_3Q_2Q_1Q_0$

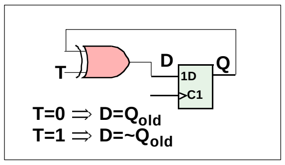
TC = Teminal Count



Propagation Delay

Notice the ripple carry. This limits the speed of the counter.

Binary Counters



The simplest counters use toggle flip-flops. These are made from D flip-flops as shown.

This is why many circuits of counters show an XOR gate and an AND gate for each flip-flop.

The AND gates, drawn on top of the waveforms, show how the flip-flops toggle whenever all the preceding flip-flops are one.

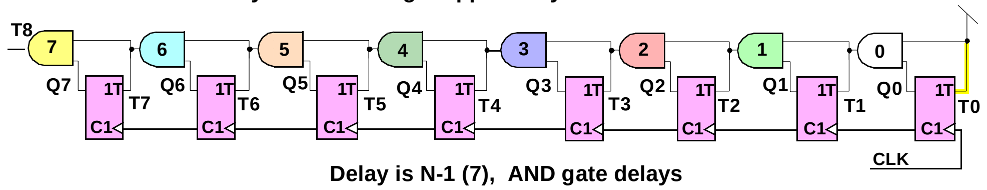
Counter Speed

Note that for long counters the chain of AND gates will get very long and will limit the speed of the counter. For a 32 bit counter, the clock speed must allow propagation through 31 AND gates plus the clock-to-output and setup times of a flip-flop.

Logarithmic Carry in a Binary Counter

Ripple Carry Circuit Using T Flip-Flops

8-bit binary counter using a ripple carry

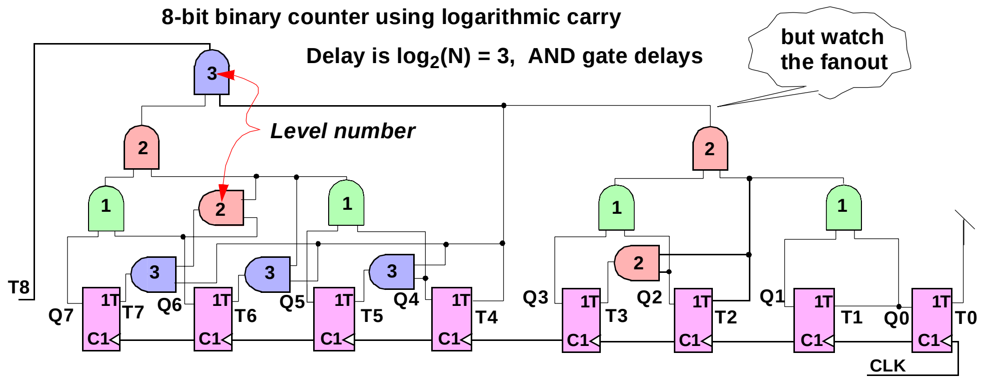


Delay is N-1 (7), AND gate delays

Logarithmic Carry Circuit Using T Flip-Flops

8-bit binary counter using logarithmic carry

Delay is $\log_2(N) = 3$, AND gate delays



Logarithmic Ripple-Carry Counter

By placing a buffer as shown, the end-to-end carry need only pass through $2\log_2(N)$ gates.

Counter Speed

In the ripple-carry circuit.

The time between a clock edge and a stable outbut from TC is:

T_{CHQV} for Q_0 + (T_{pd} for 7 AND gates).

Ignoring the flip-flop delay, this is a delay proportional to **N-1,** for N flip-flops.

The fanout also effects the delay. Here all fanouts are 2.

In the logarithmic carry circuit.

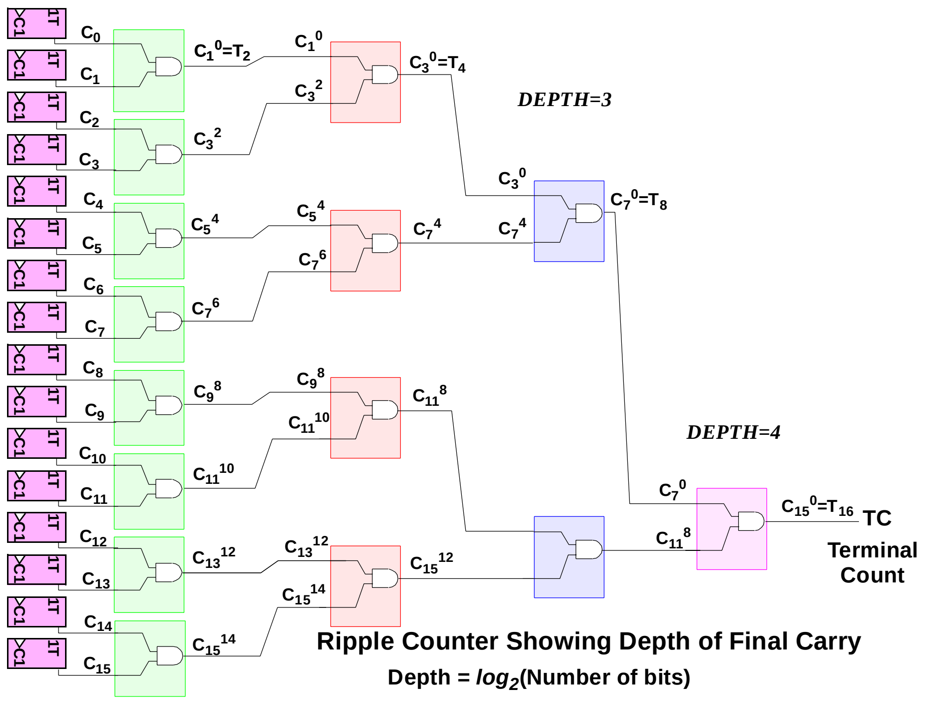
The slowest carry, TC, must go from:

a. The clock input to Q0
b. Q0 through a 1st level AND (green with a “1” in side).
c. through a 2nd level AND gate (red with a “2” inside).
d. through a 3rd level AND gate (blue with a “3” inside).

For N flip-flops this delay is proportional to ***log*$_2$(N).**

However the fanouts increase the delay. The largest fanout is N/2 + 1, (five for N=8).

Logarithmic Carry for TC for a 16-Bit Counter



Ripple Counter Showing Depth of Final Carry

Depth = *log*$_2$(Number of bits)

More Organized Picture of The Logarithmic Carry

This shows how the earlier ANDs are done in parallel so any signal used in calculating TC only has to pass through 4 AND gates.

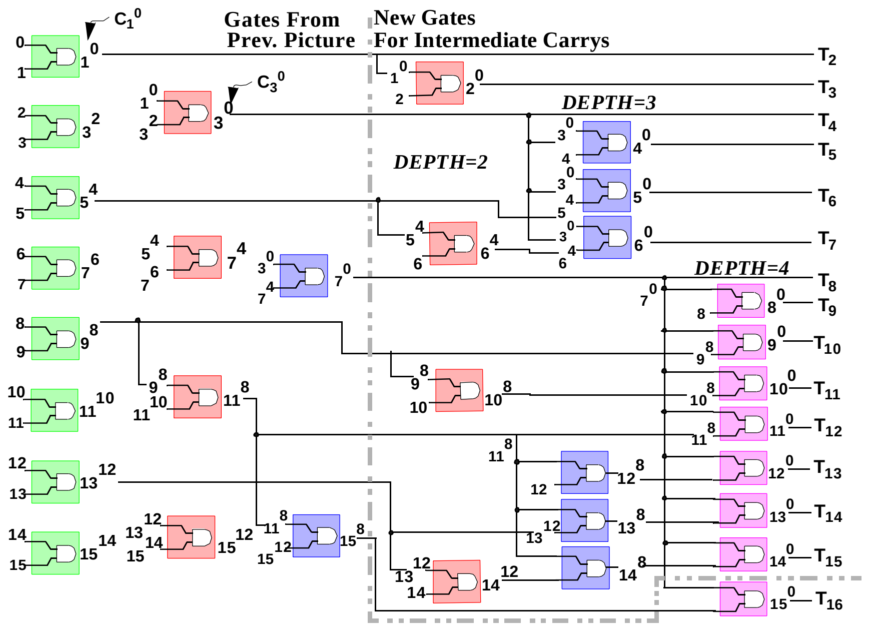
The symbol C_n^m is used to show what signals are ANDed together to make up a component of the carry. Thus the symbol C_7^4 means $Q_7 \cdot Q_6 \cdot Q_5 \cdot Q_4$.

This only shows the gates needed to calculate TC. The next page shows the gates needed to calculate the intermediate carries.

The next page also gives a rough approximation of the delay. It only hints at delay optimization from transistor sizing and adding buffers. Both of these will reduce delay at the expense of power and area.

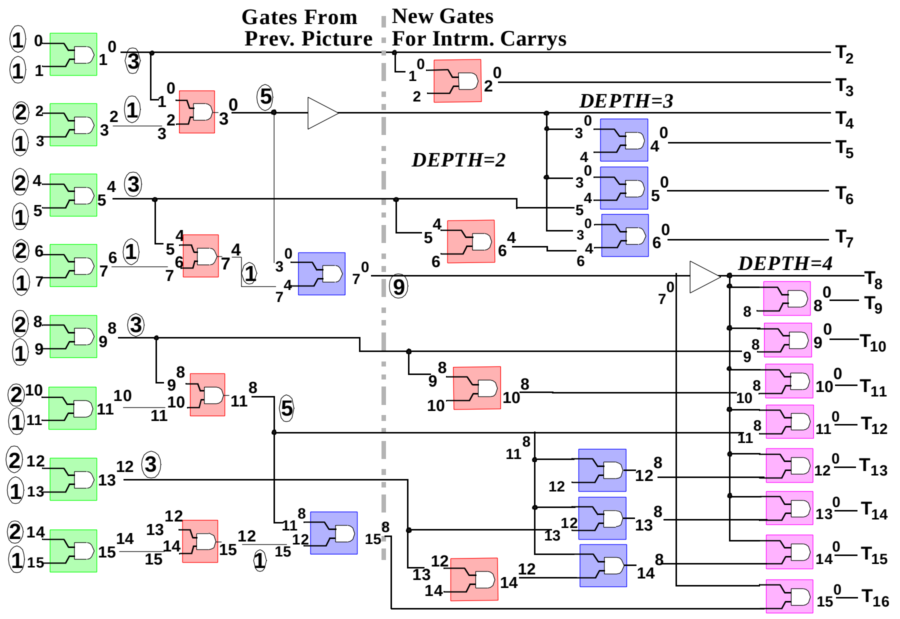
Logarithmic Circuits

Showing extra hardware for intermediate carrys
The depth is still 4 or less. The amount of hardware has doubled (16 gates).



Approximate summary: delay under half, area more than double.

If the delay of a two-input AND equals its fanout, then the delays are shown in ovals.



Without the buffers shown, the total delay, Q_0 to T_{16}, is 1+3+5+9+1 = 19. In general N-1 + $\log_2$(N).
The buffer requires a more complex calculation but it will decrease the 9 substantially.

Compare with the serial carry which is 1 + 15·2 = 31. In general 2·N -1

Addition

The 1-Bit Full Adder

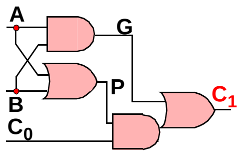
CARRY

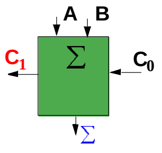
- **Generate** C_1 from inputs A,B
  C_1 =1 if A=1,B=1 (AB)
- **Propagate** previous $C_0 \rightarrow C_1$
  if A=1 or B=1 (A + B)C_0

$$C_1 = \underbrace{AB}_{G} + \underbrace{(A+B)}_{P}C_0$$

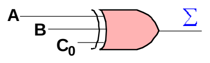
$$= G + PC_0$$

Logical C_1 using G and P





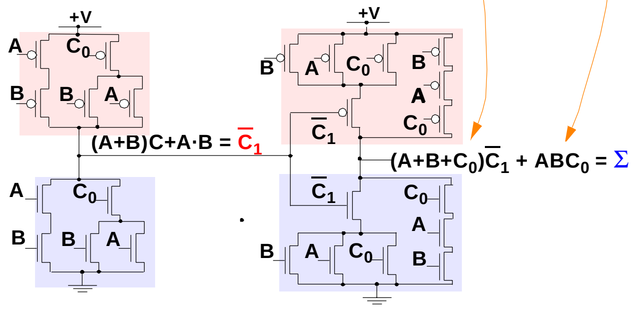
SUM



Alternate Definition

- Σ=1 if all 3 inputs are 1, (ABC_0)
- Or if only one of A, B or C_0 is 1
  ie any inputs =1, but not two
  ie any inputs =1, but no carry out
  $(A+B+C_0)\overline{C_1}$

Compact Circuit But No Separate P and G Signals



The 1-Bit Full Adder

It adds 3 bits, A + B +C_0

The Karnaugh maps for the full adder.

Maps showing $\Sigma = C_0 \oplus (A \oplus B)$

Diagonal 1s on map indicate xor

C_0 \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Σ

The map for Σ

B \ A	0	1
0	0	1
1	1	0

$x = A\oplus B$

C_0 \ x	0	1
0	0	1
1	1	0

Map of $C_0 \oplus x$

C_0 \ x	0	0	1	1
0	0	0	1	1
1	1	1	0	0
AB	00	11	01	10

$A\oplus B$ $\overline{A\oplus B}$

Map of $C_0 \oplus x$ with x expanded so one can see A, B

C_0 \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$\overline{A\oplus B}$ $A\oplus B$

Same map with columns arranged in K-map order, showing $C_0 \oplus x = C_0 \oplus (A \oplus B)$

Maps showing carry out, $C_1 = A\cdot B + C_0(B + A)$

C_0 \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

C_1

Map for C_1.

C_0 \ AB	00	01	11	10
0	0	0	AB	0
1	0	BC		AC

Circle map.

$$C_1 = A\cdot B + B\cdot C + C_0\cdot A$$
$$= A\cdot B + C_0(B + A)$$
$$= (A \oplus B) \oplus C_0$$
$$\Sigma = (A \oplus B)\overline{C_I} + \overline{(A \oplus B)}C_0$$

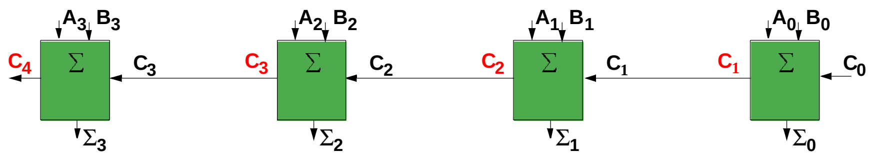
The CMOS carry circuit

The PMOS part of the circuit does not implement $\overline{C}_1 = (\overline{A}+\overline{B})\cdot(\overline{B}+\overline{C}_0)\cdot(\overline{A}+\overline{C}_0)$ as good CMOS is supposed(?) to do.

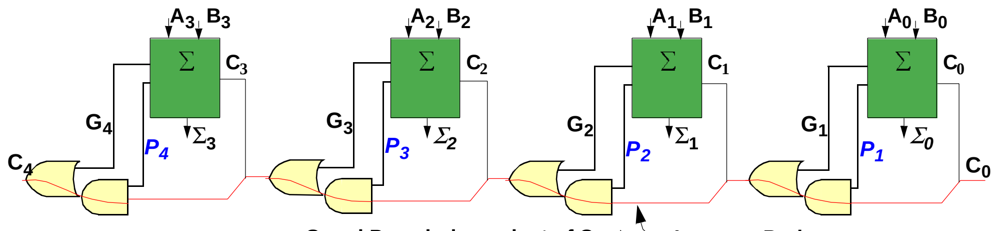
1. PROBLEM
   a. Find the expression it does implement.
   b. Show that it really is $\overline{C}_1$.

The 4-Bit Adder

Ripple-Carry Adder



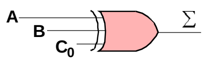
Ripple Carry Using Propagate and Generate



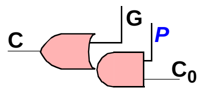
G and P are independent of C ⇒ Longest Path

Basic Adder Blocks

Σ = (A ⊕ B) ⊕ C_0

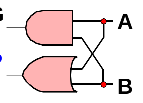


C = G + PC_0



Generate carry; G

Propagate carry; *P*



2. PROBLEM
Show that the PMOS part of the full adder on Slide 54, implements the complement of the NMOS part.
Hint: plot the PMOS function on a Karnaugh map. Then invert each square on the map to get the complement of The PMOS function, and check that it matches the NMOS map.

3.• PROBLEM
Some books define P=A⊕B. Show what this does to the C_1 and ∑ functions.
What advantage does it have, i.e. smaller, faster, less power.

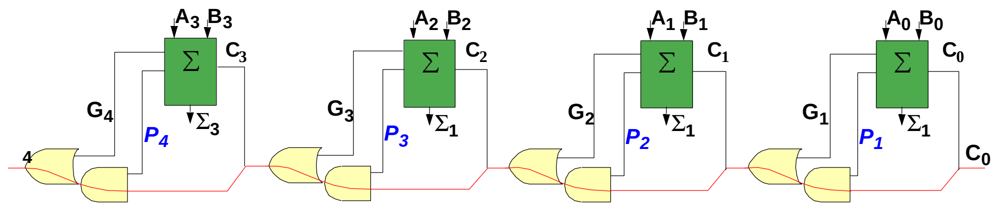
Solution:

The alternate P is generated for free because it is needed in the sum, thus saving an OR gate in each block. However the XOR takes more time to calculate, so the Ps will be delayed.
Some circuits,. like the carry-bypass adder, demand that P=XOR.

▪ Addition ▪

Derivation of Carry Look-Ahead Blocks

Serial Carry Using P and G (repeated)



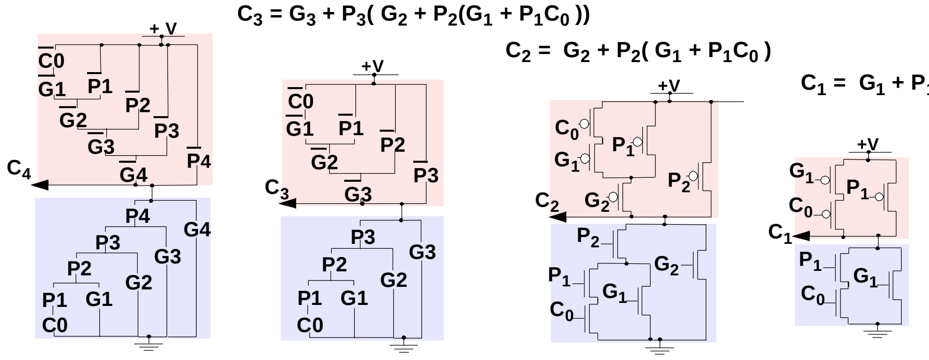
Convert Serial Carry to Parallel Circuits

$C_4 = G_4 + P_4(G_3 + P_3(G_2 + P_2(G_1 + P_1C_0)))$

$C_3 = G_3 + P_3(G_2 + P_2(G_1 + P_1C_0))$

$C_2 = G_2 + P_2(G_1 + P_1C_0)$

$C_1 = G_1 + P_1C_0$



Carry Look-Ahead Adder

Eliminating The Long Path-Delay For The Carry

Deep Gates Instead of Long Paths?

The long carry chain makes the add slow. One can factor the carry propagation equation to make each C_i one complex gate. However for C_4, this complex gate will have 5 series transistors (one would not go to C5). To compensate for the five channel resistances, the transistors in the C_4 gate would be made 2.5 times wider than those in the C_1 NAND-OR gate. This greatly increases the adder area and power consumption.

The large capacitance seen by $C_{0,}$ especially at the wide gates of the left most circuits, causes some delay. However, the propagation time from C_0 to C_4 is still about half that of the serial carry using P and G.

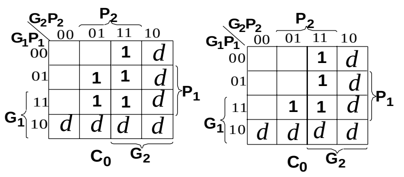
PMOS and NMOS are Almost Symmetric

The PMOS logic here is not that found by applying DeMorgan directly to the NMOS logic. Neither is it derived directly from the corresponding NMOS equation. For example:

$$C_3 = G_3 + P_3 (G_2 + P_2(G_1 + P_1C_0))$$

using the methods in “Using the Sum of Products (Σ of Π) for the PMOS function” on page 28

Remember that $G_{i+1} = A_iB_i$, and $P_{i+1}= A_i + B_i$, hence one will never get the case $G_{i+1},P_{i+1} =1,0$. The outputs for these inputs become don’t cares on the map, for the C_3 equation and the equation for the PMOS circuit can be derived.

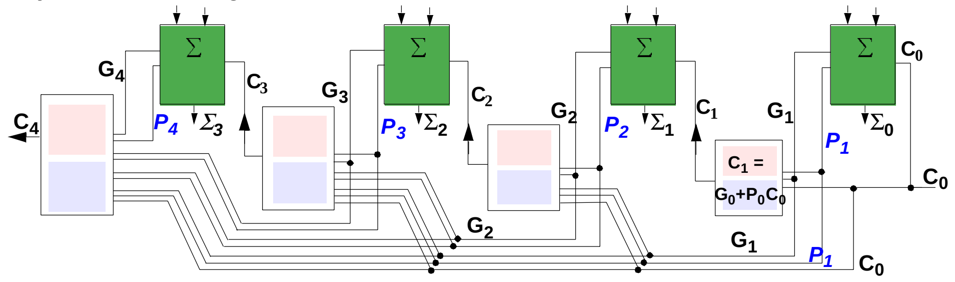


4. PROBLEM

Derive the equation for the PMOS circuits for C_2.
C_3 will have a similar but longer derivation.

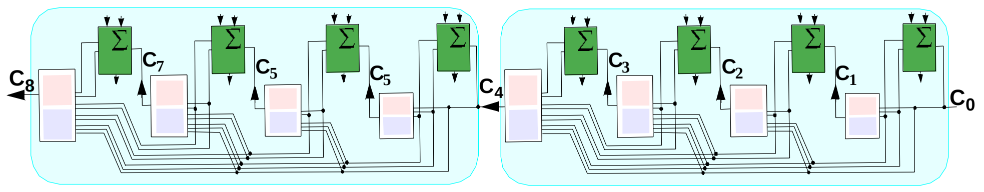
The 4-Bit Carry Look-Ahead Adder

Carry Lookahead Using P and G



Note no carry signal propagates through the Σ blocks. C_i is not connected to P_{i+1} or G_{i+1}

Look-Ahead Carry Blocks



Blocks Connected By Ripple Carry

Look-Ahead

Carry look-ahead depends on single gates, albeit with large fan-in, being faster than a chain of gates. This is true up to 3 or 4 full-adders (4 or 5 series transistors in the final carry block).

The area used increases significantly with *carry look-ahead* because the gates are larger, and to maintain speed, the transistors in the series chains of large gates must be larger.

Grouping Blocks

Since only four adders can be put in a block, larger adders must have chains of blocks. The longest delay is the delay for C_0 to reach C_8 above, or C_{4n} for n blocks.
If all gates had the same delay τ, the adder delay would be $4\tau n$ for the ripple-carry and τn for the carry look-ahead.

Unfortunately the large gates provide a large capacitive load to their source gates and slow down the look-ahead carry signal, $C_{4n,}$ to roughly $2\tau n$. This calculation is fairly complex and involves placing buffer inverters after C_4, C_8 ...

5. PROBLEM

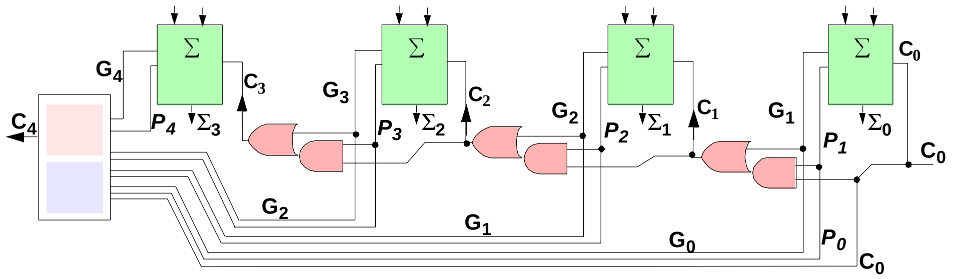
Take the NMOS part of the carry circuit for C_1 on Slide 56. If the G_0 transistor has a width of 1 unit in order to pull down at a certain speed, then the P_1 and C_0 transistors must have a width of 2 units to pull that path down at the same speed. This assumes channel resistance is proportional; to width which is close to true. Show that in the NMOS circuit for $C_{4,}$ the G_4 transistor need only have a width of 1 to maintain speed but some other transistors need a width of 5 units.

6. PROBLEM

There are five outputs in the *compromise adder* on Slide 58, Σ_1, Σ_2. Σ_3, Σ_4 and C_4.
Rate each output as being slower, the same, or faster than the *4-bit carry look-ahead adder.*

S_1 is about the same, although the load on C_0 will slow its rise time. S_2, S_3 are slower because their signal goes through the same number of gates, but the P and G inputs are more highly loaded. S_4 is faster because of fast path to the complex gate.

Compromise Carry Look-Ahead Adder.



Properties	Delay C_0->C_4	Area
Ripple P&G	2	1
Full parallel	1.1	3.5
Compromise	1	2.5

Description

By not using parallel look-ahead on C_3, C_2, and C_1

C_3 will have 50% more delay than C_4,
but C_3 is not passed on to later stages.

C_0->C_4 is slightly faster because of reduced loading on C_0.

Compromise Carry Look-Ahead Adder

For more than 4-bits, this is the fastest adder shown so far.

In a multi-block adder, one might want to make the final block or the last two blocks fully carry look-ahead so that the sum bits would not be delayed more than the final carry.

Later we will show how gradually increasing the length of the blocks will reduce the number of blocks. For long adders, the delay will increase with *sqrt*(n) rather than linearly with n (2τn) as it does here.

The next carry look-ahead adder will have a delay which increases with log_2(n).

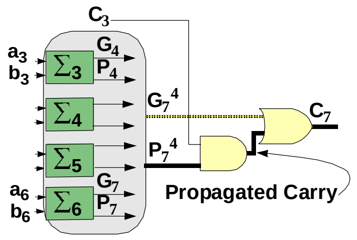
The Brent-Kung Carry-Lookahead Adder

Generalized *propagate* and *generate*

Generalized Propagate P_m^k

A *propagate* through a block of adders

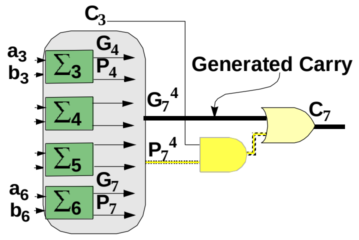
If P_7^4=1 then:
C_3 can propagate thru adders Σ_3 to Σ_6
and come out as C_7



Generalized Generate G_m^k

A *generate* within a block of adders
that can also propagate thru the block internaly

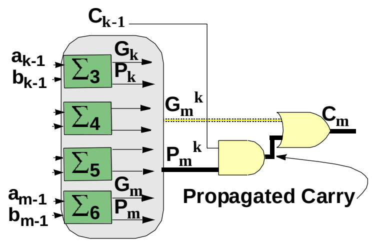
If G_7^4=1 then:
A carry was generated on G_4, G_5, G_6, and/or G_7
it was able to propagate to the output G_7^4.
And come out on C_7



Generalized *generate* and *propagate*.

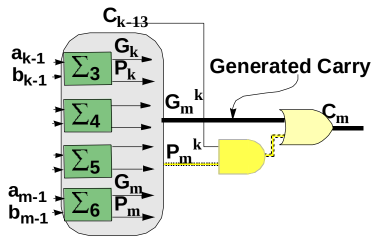
P_m^k is a propagate between adder m and adder k

If P_m^k=1 then:
C_{k-1} can propagate thru the adders
that produce $G_k, G_{k+1} \ldots G_m$ and $P_k, P_{k+1} \ldots P_m$
(adders $\Sigma_{k-1}, \ldots, \Sigma_{m-1}$)
It comes out on C_m



G_m^k is a generate for adders m thru k that reaches m.

If G_m^k=1 then:
a carry was generated inside the block of adders
that produce $G_k, G_{k+1} \ldots G_m$ and $P_k, P_{k+1} \ldots P_m$
and propagated inside the block to the output.



Generalized *generate* and *propagate*

P_m^k ***composite propagate.***

For the adder's inputs a_{m-1} and b_{m-1}

$P_m^m = P_m = a_{m-1}+b_{m-1},$ $\quad P_0 = 1$

From other G and P signals

$P_m^k = P_m P_{m-1}^k = P_m^j P_{j-1}^k$

Examples:

$P_5^2 = P_5P_4P_3P_2$ $\qquad P_m^0 = P_mP_{m-1}P_{m-2}....P_0$

$P_5^3 = P_5^5P_4^3$ $\qquad P_5^3 = P_5^4P_3^3$

G_m^k ***composite generate***

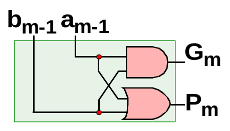
A carry is generated between m and k, and reaches m.

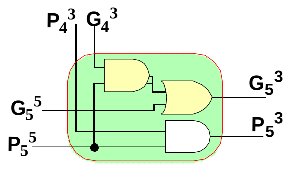
For the adder inputs a_{m-1} and b_{m-1}

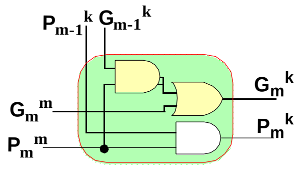
$G_m^m = G_m = a_{m-1}b_{m-1},$ $\quad G_0 = C_0$ which is often 0

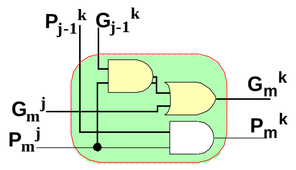
From other G and P signals

$G_m^k = G_m + P_mG_{m-1}^k = G_m^j + P_m^jG_{j-1}^k$









The Brent-Kung Carry-Lookahead Adder

Brent and Kung introduced a generalized *generate* and *propagate*. Thus P_m^k represents a *composite propagate* from adder m down to adder k.

For the adder's initial inputs a_m and b_m:-

$P_m^m = P_m = a_{m-1}+b_{m-1},$ $\quad P_0 = 1$

After the initial inputs:-

$P_m^k = P_mP_{m-1}^k = P_m^jP_{j-1}^k$

Examples:

$P_5^2 = P_5P_4P_3P_2$ $\qquad P_m^0 = P_mP_{m-1}P_{m-2}....P_0$

$P_5^3 = P_5^5P_4^3$, $\qquad P_5^3 = P_5^4P_3^3$ $\qquad$ (c)

Also G_m^k represents a *composite generate* between adders m thru k.
For the initial adder inputs a_m and b_m:-

$G_m^m = G_m = a_{m-1}b_{m-1},$ $\quad G_0 = C_0$ which is often 0

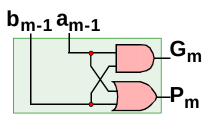
After the initial inputs:-

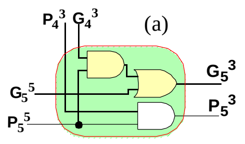
$G_m^k = G_m + P_mG_{m-1}^k = G_m^j + P_m^jG_{j-1}^k$

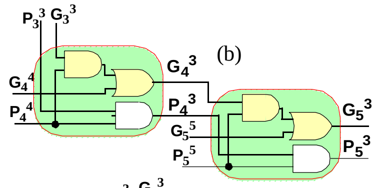
Examples:

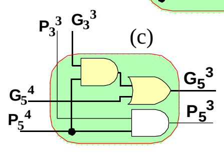
$G_m^0 = C_m$

$$\begin{aligned} G_5^3 &= G_5 + P_5G_4^3 && \text{(a)}\\ &= G_5 + P_5(G_4 + P_4G_3^3) && \text{(b)}\\ &= (G_5 + P_5G_4) + P_5P_4G_3 &&\\ &= G_5^4 + P_5^4G_3 && \text{(c)} \end{aligned}$$









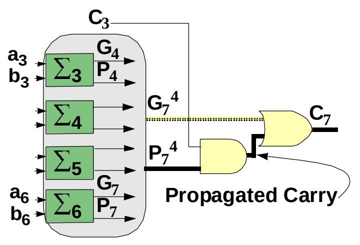
Generalized ***generate*** and ***propagate***.

<ins>$P_m{}^k$ is a *propagate* between adder m and adder k</ins>

If $P_m{}^k$=1 then:
C_{k-1} can propagate thru the adders
that produce G_k, G_{k+1} ... G_m and P_k, P_{k+1} ... P_m
(adders Σ_{k-1}, ..., Σ_{m-1})
It comes out on C_m

<ins>Example</ins>

If $P_7{}^4$=1 then:
C_3 can propagate thru adders Σ_3 to Σ_6
and come out as C_7

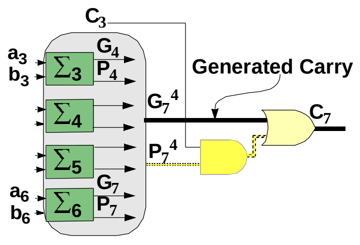


<ins>$G_m{}^k$ is a *generate* for adders m thru k that reaches m</ins>.

If $G_m{}^k$=1 then:
a carry was generated inside the block of adders
that produce G_k, G_{k+1} ... G_m and P_k, P_{k+1} ... P_m
and propagated inside the block to the output.

<ins>Example</ins>

If $G_7{}^4$=1 then:
a carry was generated on G_4, G_5, G_6, and/or G_7
it was able to propagate to the output $G_7{}^4$.
and come out on C_7



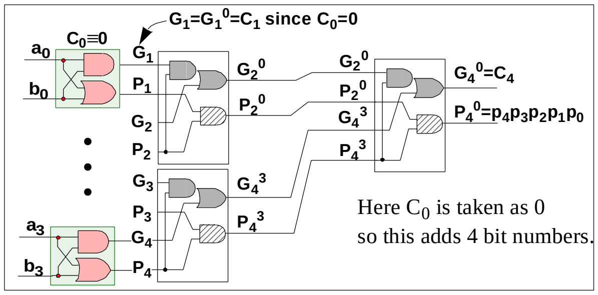
<ins>Add without an initial carry in C_0</ins>

Larger Example

$G_1{}^0= C_1= G_1+ P_1C_0 =G_1 + 0$
$G_2{}^0= C_2=G_2 + P_2C_1 =G_2 + P_2G_1{}^0$

$G_4{}^3=G_4+P_4G_3$
$G_4{}^0= \quad G_4{}^3 \quad + \quad P_4{}^3G_2{}^0$
$=(G_4+P_4G_3) + (P_4P_3)(G_2 + P_2G_1{}^0)$

Since $C_0 \equiv 0$, we do not have to “waste” an input, and the block can add 4-bit numbers.

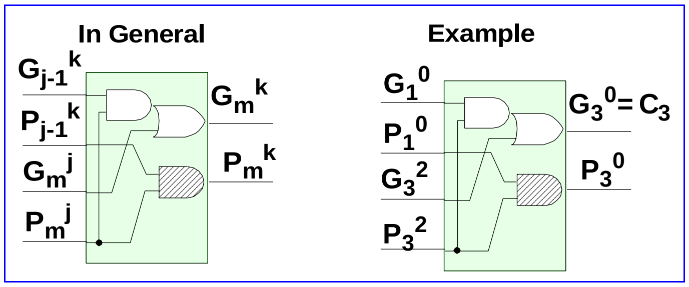


Here C_0 is taken as 0
so this adds 4 bit numbers.

The Generalized Carry Operator

Takes in two adjacent G and P pairs

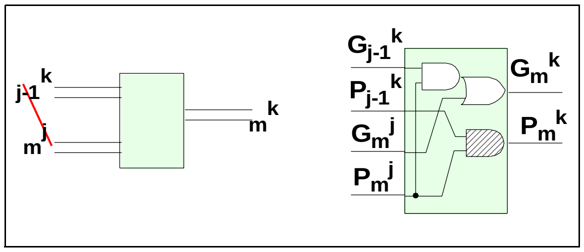
Puts out a merged G and P pair



What to Remember

The General Formula for combining carrys

The $\cancel{{}_{j-1}{}^{j}}$ are absorbed



The Brent-Kung Carry-Lookahead Adder

Combining P_m^k and G_m^k

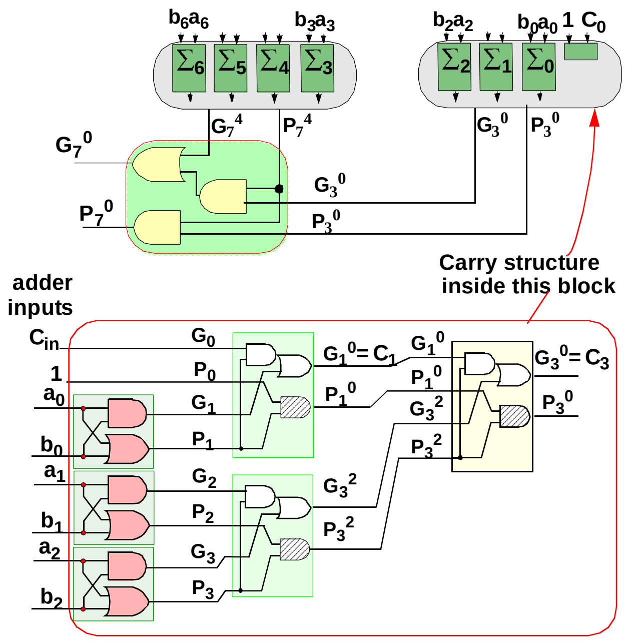
$$G_m^k = G_m^j + P_m^j G_{j-1}^k$$
$$G_7^0 = G_7^4 + P_7^4 G_3^0$$

$$P_m^k = P_m^j P_{j-1}^k$$
$$P_7^0 = P_7^4 P_3^0$$

$$P_m^m = P_m$$
$$G_m^m = G_m$$
$$G_m^0 = C_m$$

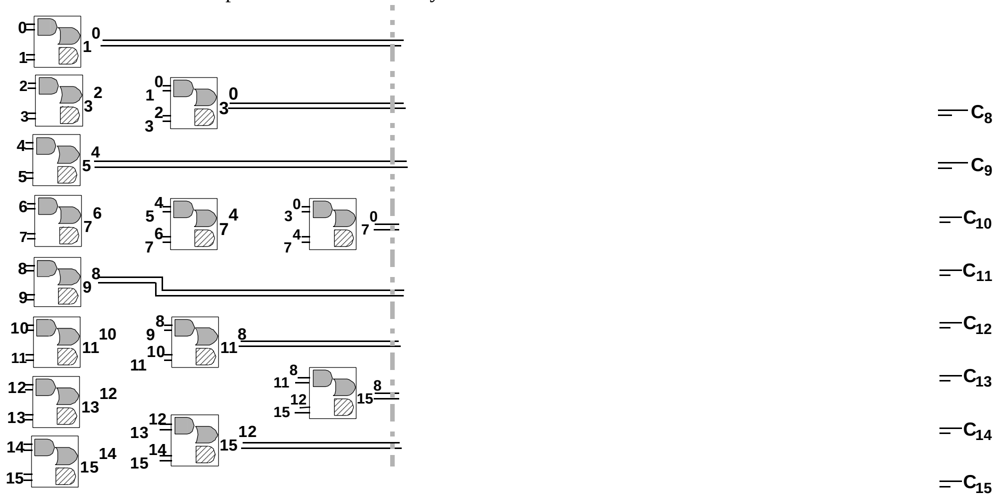
Carry in, C_0, uses the first adder position. This block can only add two 3-bit numbers.

If $C_0 \equiv 0$, always, then two 4-bit numbers can be added.



7. PROBLEM:

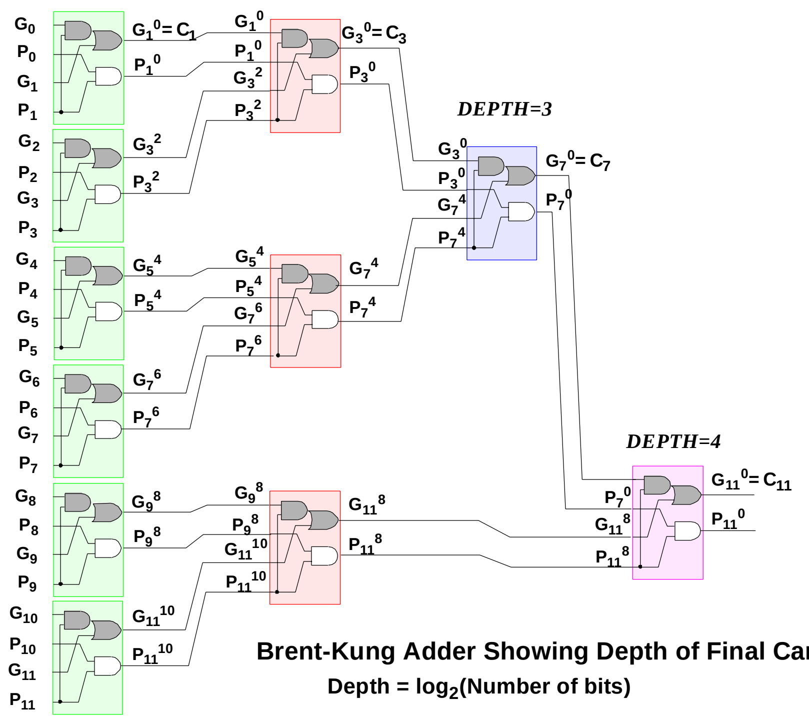
Assume the implied interconnections to the left of the dividing line.
Sketch the circuit with interconnections to calculate carrys C_8 through C_{15}
Do not increase the depth more than necessary.



8.• PROBLEM

Following the example of the compromise adder, Slide 58, show how to reduce the size of a few of the intermediate carrys in the Brent-Kung adder without reducing speed.One can only do the ones for the first n/2 bits. There is not as much saving here as in the compromise adder.

▪ Addition ▪



Brent-Kung Adder Showing Depth of Final Carry

Depth = $\log_2$(Number of bits)

A 12 Bit Brent-Krung Adder

There were 11 of the three-gate blocks when only a fast C_{12} was needed.

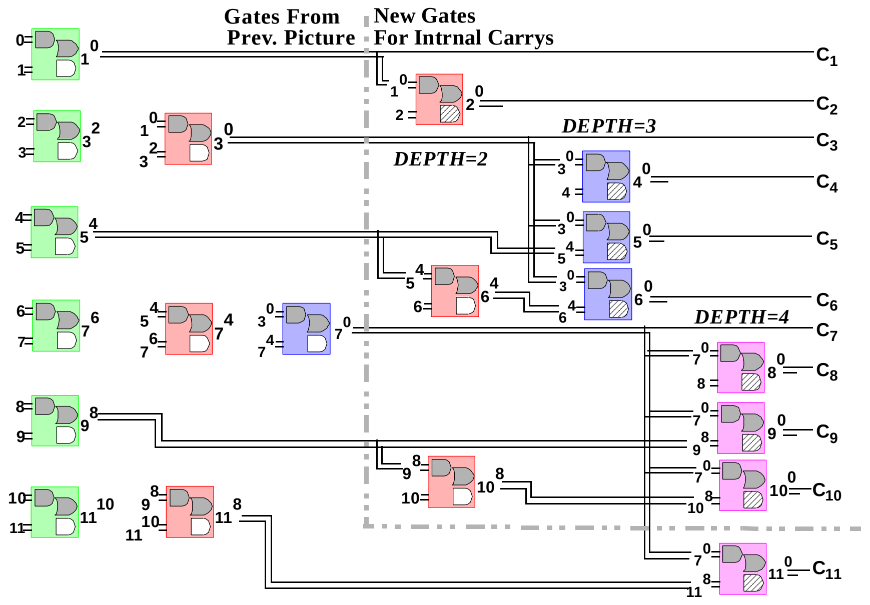
None of this extra circuitry was needed for a ripple-carry adder using P and G. The gates to generate the initial P and G are needed but not shown in the above diagram.

To get the other carries one must add another 9 three-gates pairs for a total of 20. This is shown on the next slide.

▪ Addition ▪

Brent-Kung Adder

Showing extra hardware for intermediate carrys
The depth is still 4 or less
The amount of hardware has nearly doubled.



Brent-Kung Adder Summary

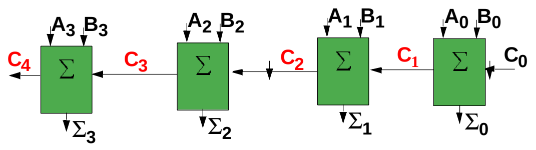
- The depth of the carry chain increases by 1 when the number of bits doubles. A depth of 4 would allow 9 to 16 bit words.
- Alternately the depth is the *ceiling*(*log*$_2$(n)) where n is the number of bits.[1] Remember that C_0 takes up one bit position unless it is always zero.
- The delay goes up more quickly because of the large fanout as n increases. On Slide 62, one generalized gate fans out to 5 gates. An n bit adder will have one gate that fans out to about n/2 gates.
- The number of generalized carry blocks is, for a (n/2)*log*$_2$(n), when n is a power of 2.
- For 32 bits or more, this adder has the maximum hardware of all the *carry-lookahead* adders. Its area is proportional to n*log2*(n). The area of the other carry look-ahead adders is proportional to n.
- It is a big power-hungry adder, but fast.

[1] The *ceiling* is the smallest integer larger than a number

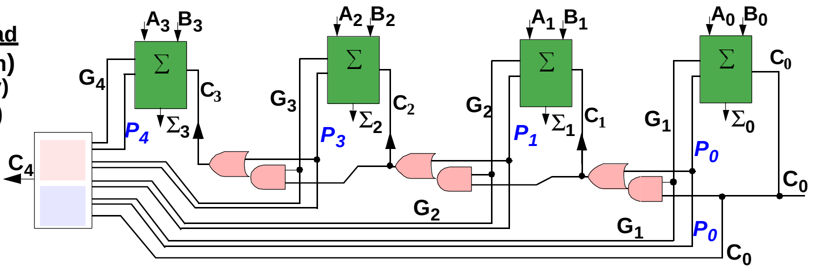
▪ Addition ▪

Summary Of Adders Types (Previously Covered)

(1) Ripple-Carry
slow; delay = O(n)
small; area = O(n)

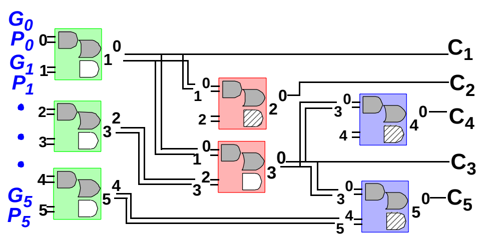


(2) Carry Look-Ahead
faster; delay = O(n)
0.5x(ripple delay)
large; area = O(n)
2x(ripple area)

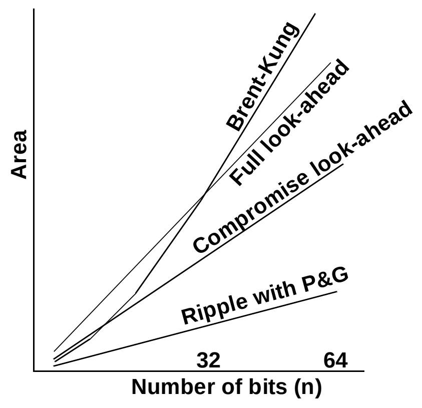


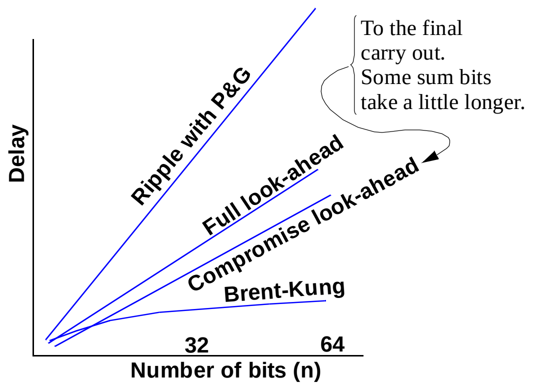
(3) Brent-Kung
fastest O(*log*$_2$(n))
largest O(n*log*$_2$(n))

area passes compromise
carry look-ahead about n=10



Properties of look-ahead adders





O(n) notation

The notation delay=O(n), means that the delay increases in proportion to n for large n.

Thus delay=13n is O(n), but so is delay = 26 + 13n because the 26 is negligible when n is large

More generally if some property = a + bn + cn^2 + dn^3
the property is said to be O(n^3) because it increases proportional to n^3 for large n.

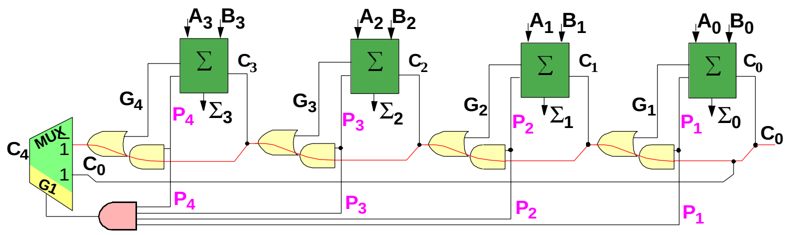
For n=4 or 8, small details in implementation, like buffer sizing, may make the look-ahead faster than Brent-Kung, but when n>32, it is clear that Brent-Kung will beat the pants off the others.

Summary Of Adders Types : (About to be covered)

Four New Adder Types

5) Carry Skip
Speeds up slowest C0->C4 path with little extra area.

Does not help other paths.



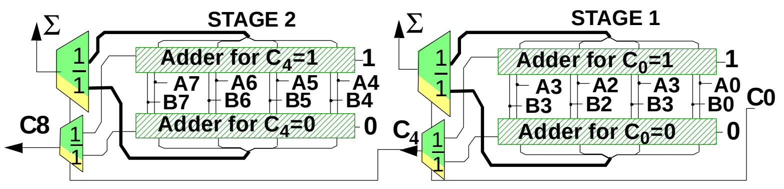
When $P_3P_2P_1P_0$=1, does not wait for the ripple carry.

6) Carry Select

Doubles area

Very fast path C_0->C_4

Still delay = O(n)



Calculate answers for both C_0=1 and for C_0=0
Select the correct one

Properties of these new adders

Carry-Bypass (Carry-Skip)

This is much like the compromise look-ahead adder.:

- The logic is a little smaller because the gate to calculate C_4 is smaller.
- The smaller gate will make the C_0-> C_4 path slightly faster.
- The path A_i,B_i -> C_4 will be slower.

Carry-Select

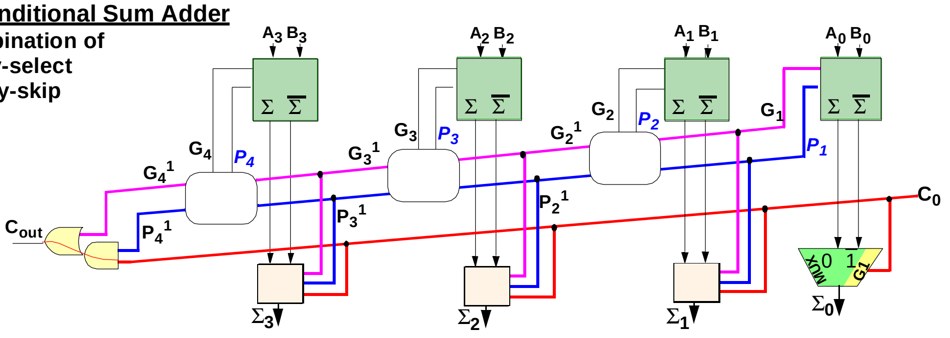
Calculate 4-bit sums with Cin=0 and Cin=1. Use Cin to select the correct one.

- Double the size of the base adder.
- The path C_0->C_4->C_8 is very fast.
  The delay is mainly in the 4-bit adder block.
- For cascade blocks, the adds are all done in parallel.
  C_0 beats the mux inputs at the C_4 mux,
  but the data is waiting at the C_8 mux when the C_4 control signal gets there.

Types of Adders Summary: (About to be covered, cont)

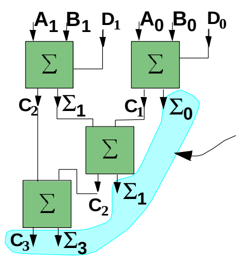
(7) Conditional Sum Adder

Combination of
carry-select
carry-skip



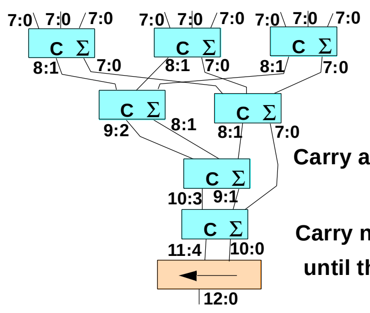
(8) Carry Save Adder

Used for multiple adds



Adding three
2-bit numbers

Final answer



Adds nine
8-bit numbers

Carry and Sum seperate

Carry not propagated
until the last adder.

Carleton UNIVERSITY

© John Knight

Revised; November 18, 2003

Properties of these new adders

Conditional Sum Adder

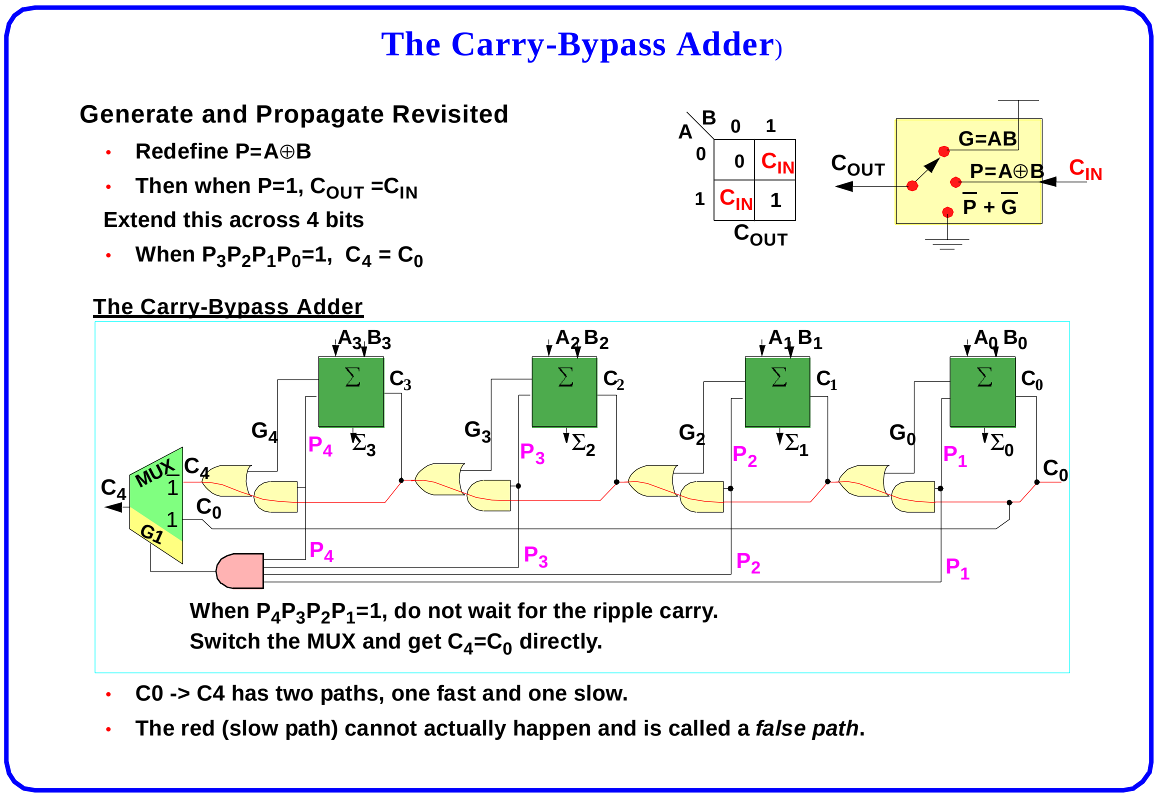
This is much like the compromise look-ahead adder.

- It calculates both sums and selects the correct one, like the carry-select adder.
- It calculates C_4 much like the carry-bypass adder.
- It is probably the fastest adder after the Brent-Kung.

Carry-Save Adder

An adder for a different purpose.

- It is good for adding several numbers, such as in multipliers.
- It uses the carry inputs in its adders to add a third number.
- Three numbers go in and two (a vector of carry bits and a vector of sum bits) come out.
- At the end one must add the two vectors together with a normal adder which propagates the carries. However it saves propagating carries during each two-number add.



The Carry-Bypass Adder)

Generate and Propagate Revisited

- Redefine P=A⊕B
- Then when P=1, C_{OUT} =C_{IN}

Extend this across 4 bits

- When $P_3P_2P_1P_0$=1, $C_4 = C_0$

<u>The Carry-Bypass Adder</u>

When $P_4P_3P_2P_1$=1, do not wait for the ripple carry.
Switch the MUX and get C_4=C_0 directly.

- C0 -> C4 has two paths, one fast and one slow.
- The red (slow path) cannot actually happen and is called a *false path*.

The Carry-Bypass Adder

This circuit allows the carry to bypass certain adder sections where the *propagate* signals are all asserted.

It speeds up the longest path where a carry propagates all the way from C_0 to C_4

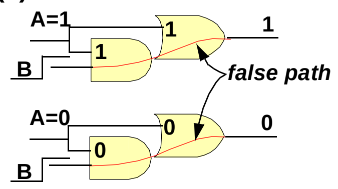
False Paths

An continuous path through combinational gates that:

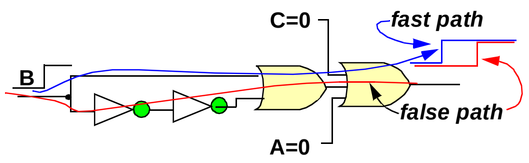
1) cannot be sensitized, or

2) always has a faster sensitized parallel path.

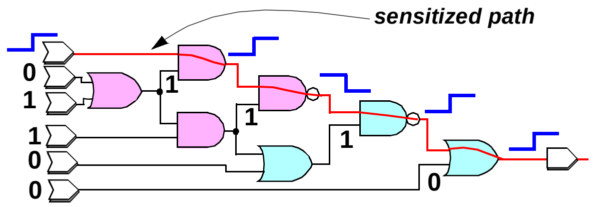
(1)



(2)



Sensitized Path



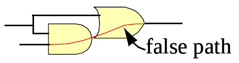
A path that can be made to propagate a signal change.

False Paths

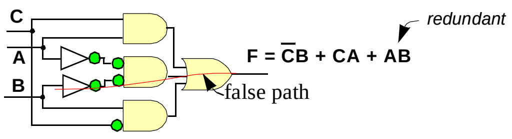
A false path is a connection through gates from the start to the end of the path which will never propagate a signal change (be sensitized) under proper operation.

.

False path when the complete path cannot be sensitized. The carry-bypass adder has that type of path.



False path due to redundant circuitry. The term **AB** is redundant. Any signal change through the inverter in the **B** path, will get to **F** faster through **$\overline{C}$B**.



False Paths

Paths that will never propagate a signal change

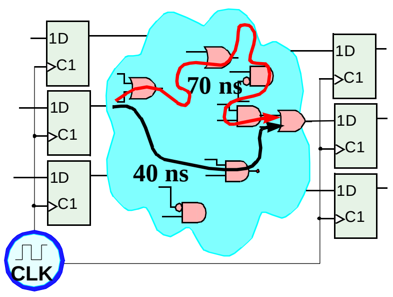
Long unused paths cause two problems, timing and testing

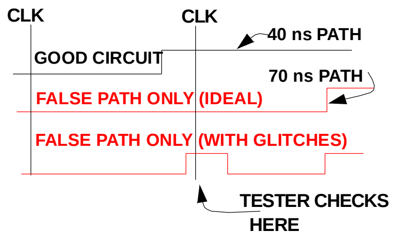
Timing problems

- **Static timing verification checks the delay of the longest combinational paths in a circuit.**
- **Path delay - input reg to output register - must be under a clock cycle.**
- **Here timing verification will say the clock period should be at least 70 ns.**

If the 70 ns path is a false path, and the next longest real path is 40 ns.

- **The verifier will state the clock period > 80ns.**
- **You will likely believe it!**





Testing Problems

Suppose the MUX in the carry-bypass adder was stuck up. The circuit would still work albeit more slowly.

- **One needs a test in which the 80ns path output is definitely wrong for the 60 ns or so.**
- **Generating this glitch free test is very difficult.**
- **Also testing usually not done at maximum speed.**

False Paths in the Carry-Bypass Adder

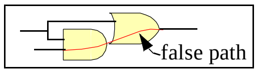
Timing Problems.

Synchronous logic

- In synchronous logic the input flip-flop outputs change just after the active clock edge.
- These changes propagate through the combinational logic (gates only, no flip-flops). The outputs of the gates change.They may go up and down several times.
- Eventually the changes will die out and the logic levels will stabilize.
- After that a new active clock edge may come and store these stable values in the output flip-flops.
- One must have:
  (The clock period) > (longest delay through the combinational logic).

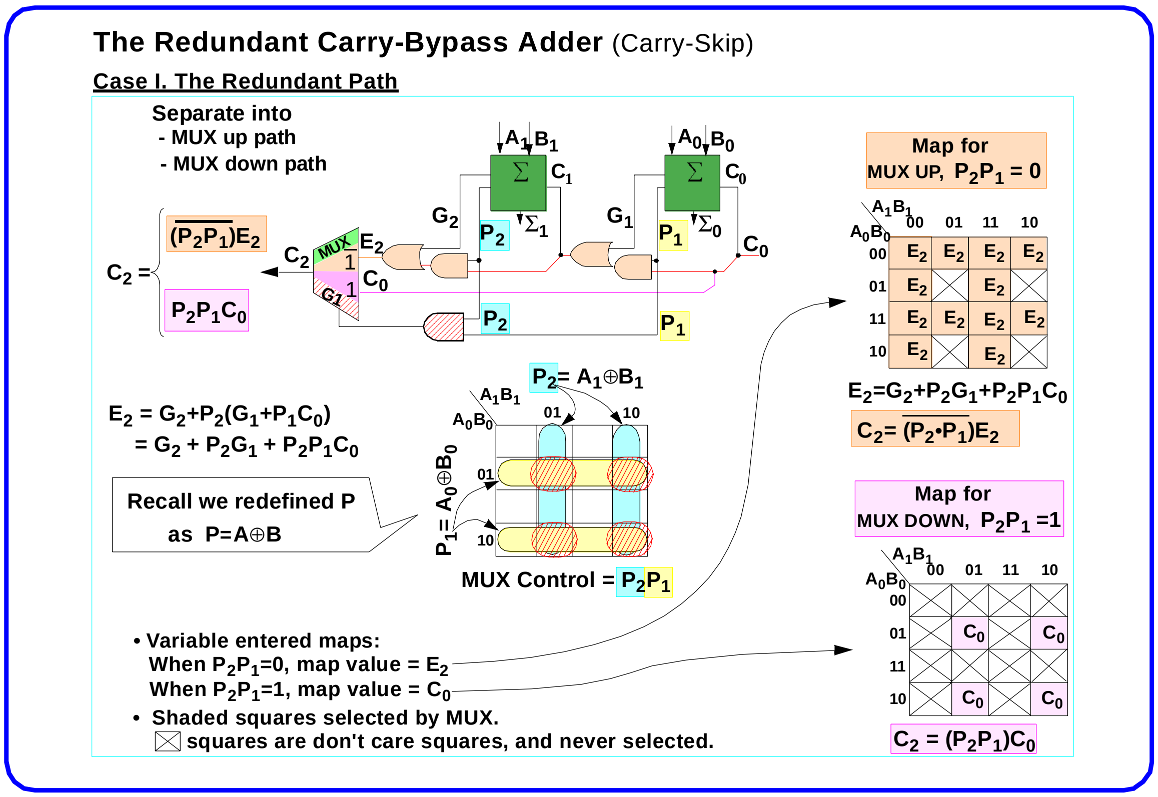
False Paths

- A false path is one which can never propagate a level change to an output.
- A common reason is the false path has a redundant parallel path. The output gets the correct answer from another path in less time than the propagation delay through the false path.
- Another reason is that the gates in the false path cannot all turn on at once.



Static Timing Verification

- After a circuit is designed and converted to a silicon layout, the delays in each gate can be calculated.
- A timing verifier is a program which goes through a logic circuit after all the gate delays have been estimated, and calculates that if all signals will be stable before the next clock edge.
- Unfortunately many of these programs only check the propagation delay along a path.
  They do not know if the output will be stabilized sooner by another parallel path.
  They do not know if the all the gates in the path can be turned on at once.
- Thus they will suggest making the clock slower than is actually needed.



The Redundant Carry-Bypass Adder (Carry-Skip)

Case I. The Redundant Path

Separate into
- MUX up path
- MUX down path

$C_2 = \begin{cases} (\overline{P_2P_1})E_2 \\ P_2P_1C_0 \end{cases}$

$E_2 = G_2+P_2(G_1+P_1C_0)$
$= G_2 + P_2G_1 + P_2P_1C_0$

Recall we redefined P as $P=A\oplus B$

MUX Control = P_2P_1

- Variable entered maps:
  When P_2P_1=0, map value = E_2
  When P_2P_1=1, map value = C_0
- Shaded squares selected by MUX.
  ☒ squares are don't care squares, and never selected.

Map for MUX UP, $P_2P_1 = 0$

A_0B_0 \ A_1B_1	00	01	11	10
00	E_2	E_2	E_2	E_2
01	E_2	☒	E_2	☒
11	E_2	E_2	E_2	E_2
10	E_2	☒	E_2	☒

$E_2=G_2+P_2G_1+P_2P_1C_0$

$C_2= (\overline{P_2\bullet P_1})E_2$

Map for MUX DOWN, $P_2P_1 =1$

A_0B_0 \ A_1B_1	00	01	11	10
00	☒	☒	☒	☒
01	☒	C_0	☒	C_0
11	☒	☒	☒	☒
10	☒	C_0	☒	C_0

$C_2 = (P_2P_1)C_0$

False Paths in the Carry-Bypass Adder (cont.)

Testing Problems

The tester would:
load the flip-flops with a test input,
trigger a clock edge,
wait for a clock period,
and then trigger another clock edge and read the outputs as captured by the flip-flops.

If the outputs are stable, it is easy to compare expected and actual signals. If the output is still active when the clock comes it is, difficult to predict what the actual signal will do. One needs to be sure the flip-flop will capture a wrong value if the faster path is defective. Designing such a test is difficult even for a single false path. Such tests cannot be done by normal test generation programs.

Most modern tests do not test at the full clock speed. *Scan tests*, to be discussed later, do not run at full speed.

Faster Circuits Do Not Have To Have False Paths

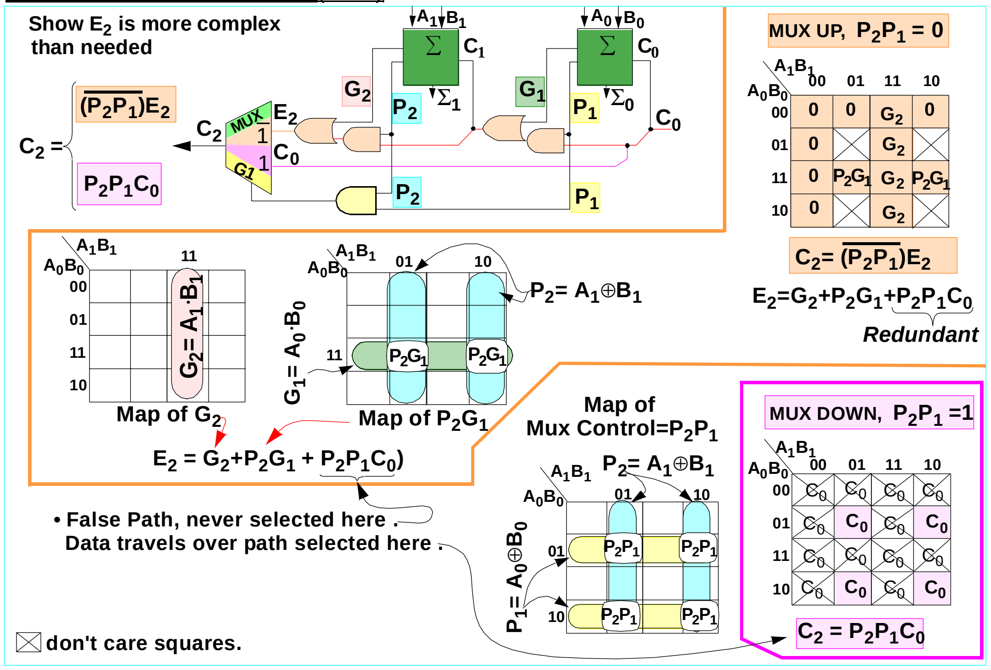
False paths are not necessary. It was proven in1991[1] that any redundant path, put in strictly to improve speed, could be replaced by a nonredundant circuit with no speed penalty.

False paths are not necessary

[1] K, Keutzer, S. Malik and A. Saldanha, "Is Redundancy Necessary to Reduce Delay?", IEEE Trans, on CAD, April 1991, pp 427-435.

The Redundant Carry-Bypass Adder (Carry-Skip)

Case I. The Redundant Path (cont.)



MUX UP, $P_2P_1 = 0$

$A_0B_0 \backslash A_1B_1$	00	01	11	10
00	0	0	G_2	0
01	0	⊠	G_2	⊠
11	0	P_2G_1	G_2	P_2G_1
10	0	⊠	G_2	⊠

MUX DOWN, $P_2P_1 = 1$

$A_0B_0 \backslash A_1B_1$	00	01	11	10
00	⊠	⊠	⊠	⊠
01	⊠	C_0	⊠	C_0
11	⊠	⊠	⊠	⊠
10	⊠	C_0	⊠	C_0

• False Path, never selected here.
Data travels over path selected here.

⊠ don't care squares.

The Redundant Carry-Bypass Adder

The Maps for E_2, the Upper MUX Input

- Three maps, G_2, P_2G_2 and P_2P_1, derive the expression for E_2. The left map encircles the G_2 term. These squares will be “1”s of E_2.
- The centre map encircles the two columns of P_2 and the row of G_1. The term is $P_2 \cdot G_1$ so the intersection of the circles will be “1”s of E_2.
- The lower centre map shows $P_2 \cdot P_1$ as the four squares at the intersections of the columns, P_2, and the rows, P_1.
- The OR of the three maps is E_2 and is shown in the “MUX UP” map.
- The MUX DOWN map shows $P_2 \cdot P_1 \cdot C_0$. The value C_0 is placed in those 4 squares. This avoids making a 5-variable map.

Don't Care Conditions Caused By Multiple Equations.

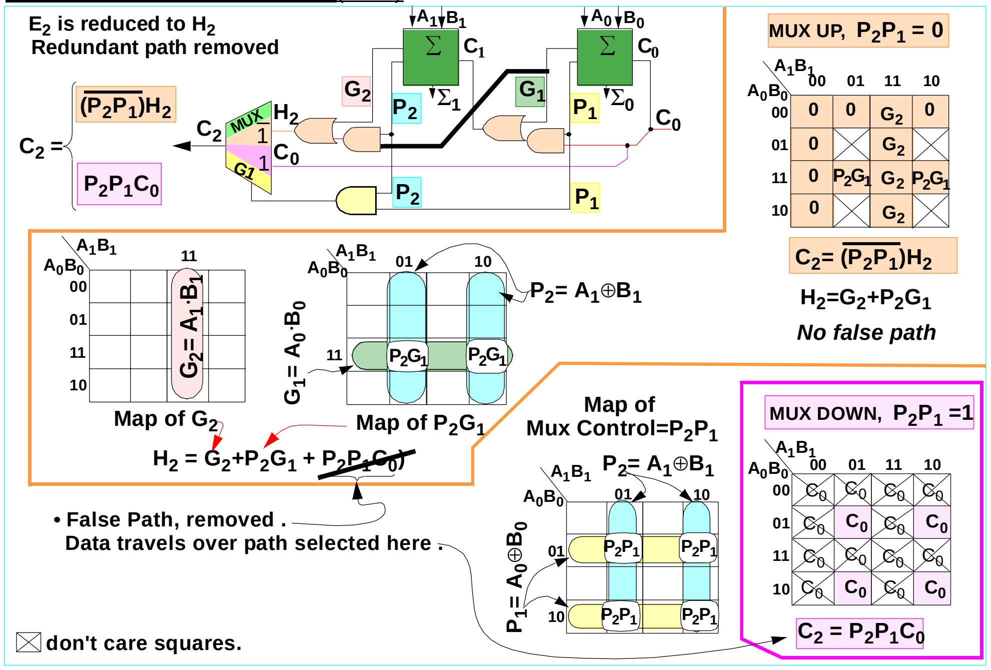
The four ⊠ squares in the “MUX UP” map are don’t care because the mux is always down (P_2P_1=1) for those four squares. For the function E_2, those squares contain the value of C_0, but who cares, they never transfer this value to the output C_2.

The twelve ⊠ squares in the “MUX DOWN” map are don’t care because the mux is down (P_1P_2=1) for only four squares. The map is actually filled with the value of C_0, but only the four useful ones are shown.

False Paths

The Redundant Carry-Bypass Adder (Carry-Skip)

Case II. A Nonredundant Path (cont.)



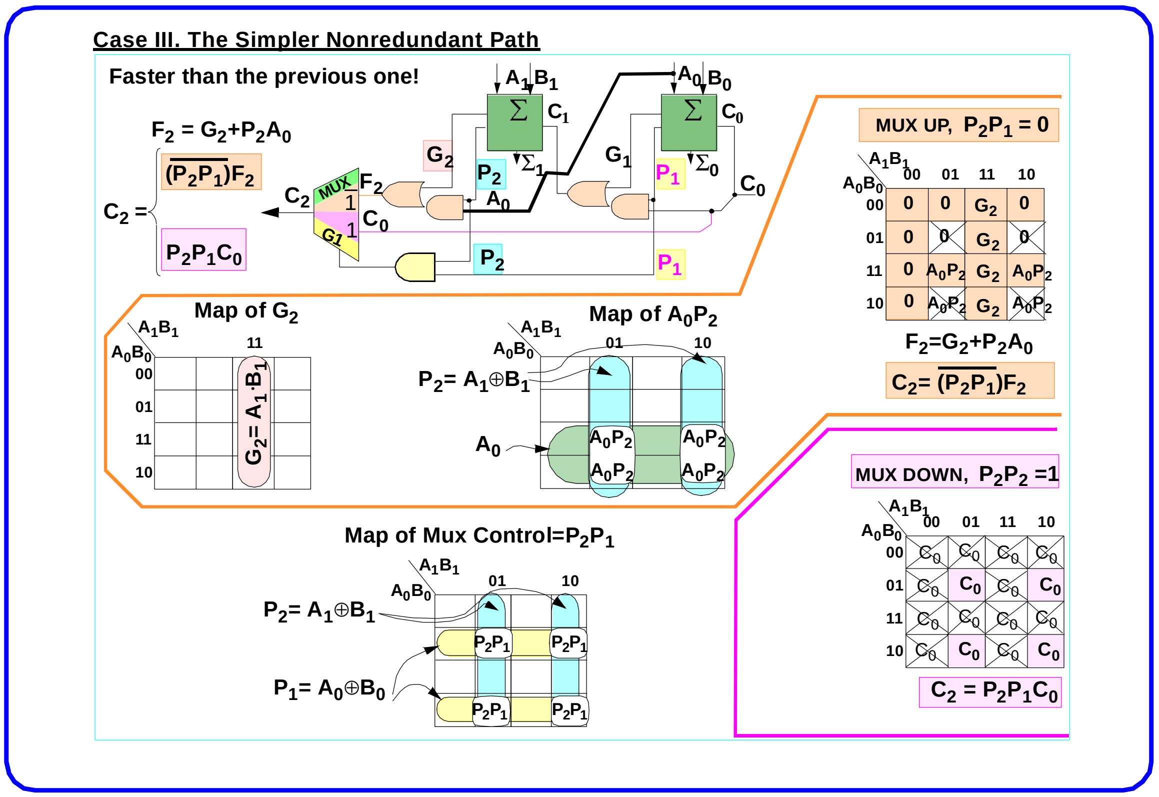
☒ don't care squares.

A Nonredundant Carry-Bypass Adder

The Upper MUX Input

- In the previous slide
  $E_2 = G_2 + P_2G_2 + P_2P_1C_0$
  The $P_2P_1C_0$ term was redundant because it was never “1” when the mux was in its upper position. This term is only used in the lower mux position.
- In this slide we drop $P_2P_1C_0$ from E_2 leaving
  $E_2 = G_2 + P_2G_2$
- This removes the false path.
  C0 no longer appears to have a path through the upper position in the mux.

False Paths



Carleton UNIVERSITY

© John Knight

Dig Cir p. 139 Revised; November 18, 2003 Slide 75

The Carry-Bypass Adder, Nonredundant Circuit

Here the $P_2P_1C_0$ term is replaced by A_0.
The new output F_2 is the same as the previous E_2 except for the four "don't care" ☒ squares.

Compare E2 and F2 equations and maps.

$E_2 = G_2 + P_2G_1 + P_2P_1C_0)$ $\qquad$ $F_2 = G_2+P_2A_0$

- The $P_2P_1C_0$ term only appeared in the don't care squares. It was removed.
- The P_2G_1only appeared in two squares. It was replaced by a P_2A_0 that made those two squares correct but changed the don't care squares.

Summary

- The don't care terms were caused by partitioning the logic into several functions.
- The don't care terms were utilized to remove redundant logic and false a path.
- Now both the static-timing verifier and the test engineer are happy. ☺

9. PROBLEM

Recall that here $P_2 = A_1 \oplus B_1$. The term P_2A_0 is on the time-critical path, and can be replaced by a slightly faster term. However it will cost a few extra transistors because one will not be able to utilize the adders XOR gate. Find this revised circuit.

The Carry Select Adder

Trade silicon for speed.

Uses Two Adders.

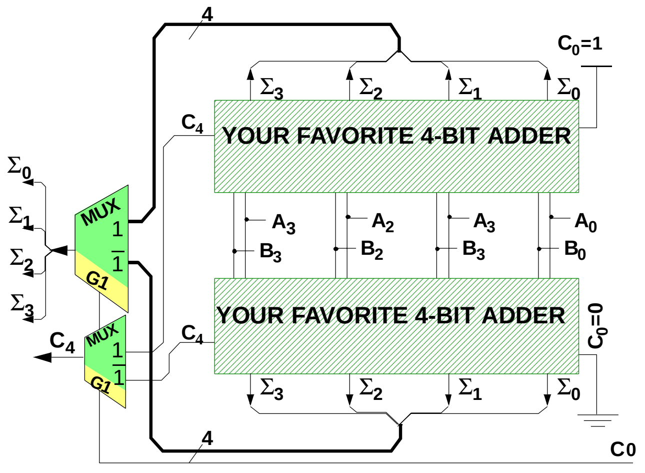
- Do 2 parallel adds
- One based on C_0=0. The other for C_0=1.
- Use C_0 to select the correct answer.

Time Saving

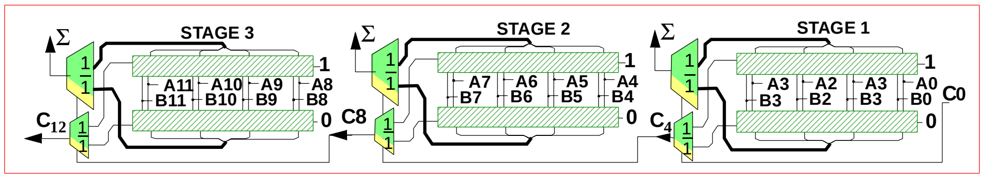
One stage saves little time.

Best with many stages. The add blocks are all done in parallel.

The carries ripple through the stages with one MUX delay per stage.



3-stage time delay = time for 4-bit add + 3*(delay thru MUX)



The Carry-Select Adder

A Fast, But Large, Adder

This adder consists of two normal adders in parallel. They might be ripple-carry or carry look-ahead, or any other type. They would usually be 4-bits adders or more.
One adder adds as if C_0=0, the other as if C_0=1. The real C_0 selects the correct answer with a MUX.

Speed

The 4-bit single MUX carry-select adder saves only one or two gate delays in the right-hand section. Probably about the same as the extra delay added by the MUX.

The carry-select adder is best for long word lengths broken into sections. For example 32 bits made of 8 sections of 4 bits each.

All the adds are done at the same time, so there is an initial delay for them to finish. Then the sums and carry outputs are available, but no one knows which to use.

Then the carry must propagate serially through the chain of MUXs, each carry switching a MUX which selects a carry, which in turn is used as the control for the next MUX.

This delay increases linearly with the number of MUXs. However it is faster than most other systems which increase linearly with the number of full adders.

10. PROBLEM

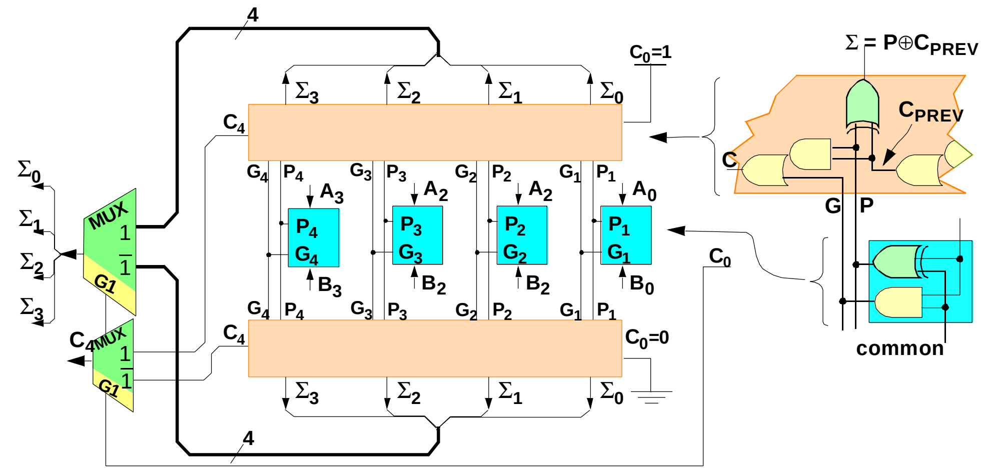
Does the carry-select adder contain redundant paths?

HINTS

Are there two apparent paths for the carry? Check the paths, is one ever turned off so a change cannot propagate through it, while the other is turned on? Alternately do two paths give the same answer but one path clearly always faster than the other.

The Carry-Select Adder (Cont.)

The two adders can share some circuitry



- The common blocks contain G=AB; P= A⊕B.
- The distinct (striped) block contain S=P⊕C$_{PREV}$; C=G+PC$_{OLD}$ for each bit
- About 60% more area than a single adder.

The Carry-Select Adder (Cont.)

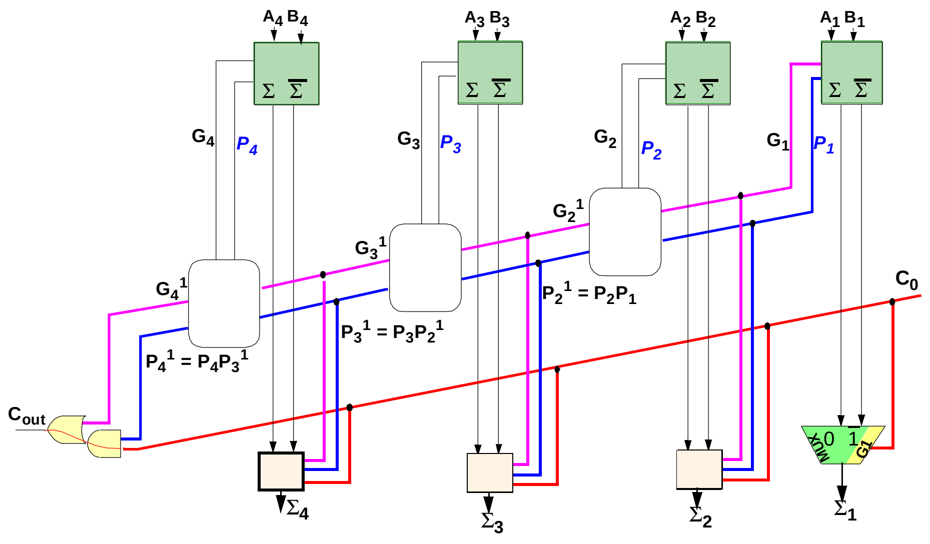
Sharing Circuitry

The propagate and generate circuits are common to the upper and lower adders because they do not use the carry. The other circuits involve carries and must be separate.

11. PROBLEM

Since the c_0 input is known to be 1 or 0, redesign the first full-adders in each 4-bit chain to utilize this fact.

Conditional Sum Adder



The Conditional-Sum Adder

At one time considered to be the fastest adder theoretically.
It combines features of the carry look-ahead, the carry-select, and the carry-bypass adders.

Each adder block calculates:

P = carry out if there is a carry in, Cout(C_0=1).
G= carry out if it is independent of a carry in, Cout(C_0=0).
Σ = sum out if carry in is 0, $\Sigma(C_{in}=0)$.
$\overline{\Sigma}$ = sum out if carry in is 1, $\Sigma(C_{in}=1)$.

Select the right carry and the right sum outside the adder block

Outside the adder block the previous P and G lines along with C0 are used to select the proper sum.
The proper carry out is G if C0=1 and P if C0=0
However the proper one is not selected immediately. Both are passed on to the next adder block.
The next block upgrades P and G and passes them on.

The carry out from a block of (usually four) adders is selected from the previous P and G by C_0.

Conditional Sum Adder

Calculate both sum and carry values

Sum $\Sigma_k = \begin{cases} \Sigma & \text{if } C_k=0 \\ \overline{\Sigma} & \text{if } C_k=1 \end{cases}$ Carry out $C_k = \begin{cases} G_k & \text{if } C_{k-1}=0 \\ P_k & \text{if } C_{k-1}=1 \end{cases}$

Calculate in the adder box

$\Sigma_{k-1} = A_{k-1} \oplus B_{k-1}$

$\overline{\Sigma}_{k-1} = A_{k-1} \oplus B_{k-1} \oplus 1$

$G_k = A_{k-1}B_{k-1}$

$P_k = A_{k-1}+B_{k-1}$

Select the correct sum to send out

Select the right carry to send on.

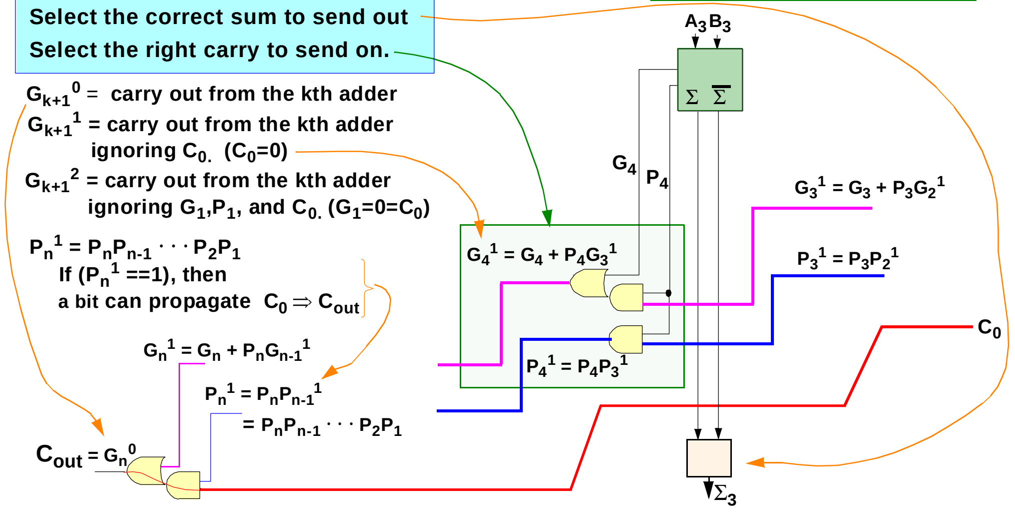
G_{k+1}^0 = carry out from the kth adder

G_{k+1}^1 = carry out from the kth adder ignoring C_0. ($C_0=0$)

G_{k+1}^2 = carry out from the kth adder ignoring G_1,P_1, and C_0. ($G_1=0=C_0$)

$P_n^1 = P_nP_{n-1} \cdots P_2P_1$

If (P_n^1 ==1), then a bit can propagate $C_0 \Rightarrow C_{out}$



Carry Calculations[1]

The initial carry C_0 bypasses all intermediate carry calculations

Two carries are calculated:

One G is value of the carry if C_0=0

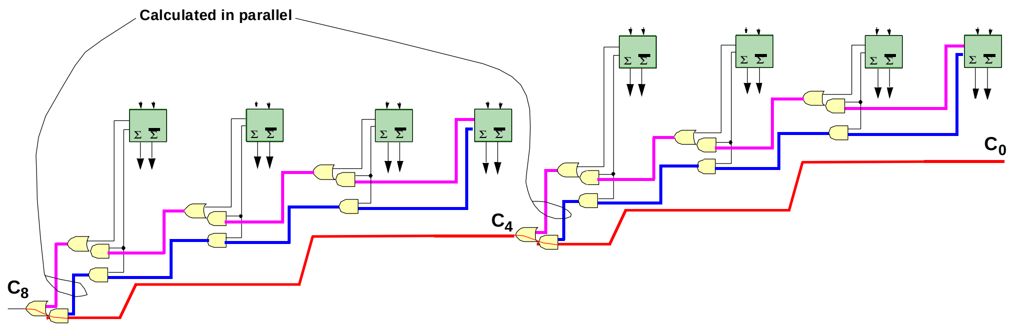
The other P is the value if C_0=1.

C_0 is not used to tell which carry is correct until the final output carry.

Notice that the propagation delay for P is exactly that of the carry-bypass adder, the delay of 4 ANDs and 2 ORs.

Also note that the next block takes in C_4 and sends out C_8.

The signals P and G are calculated in parallel with those in the first block, so C_8 does not have to wait extra time for its P and G. The delay for C_8 is that of 5 ANDs and 3 Ors

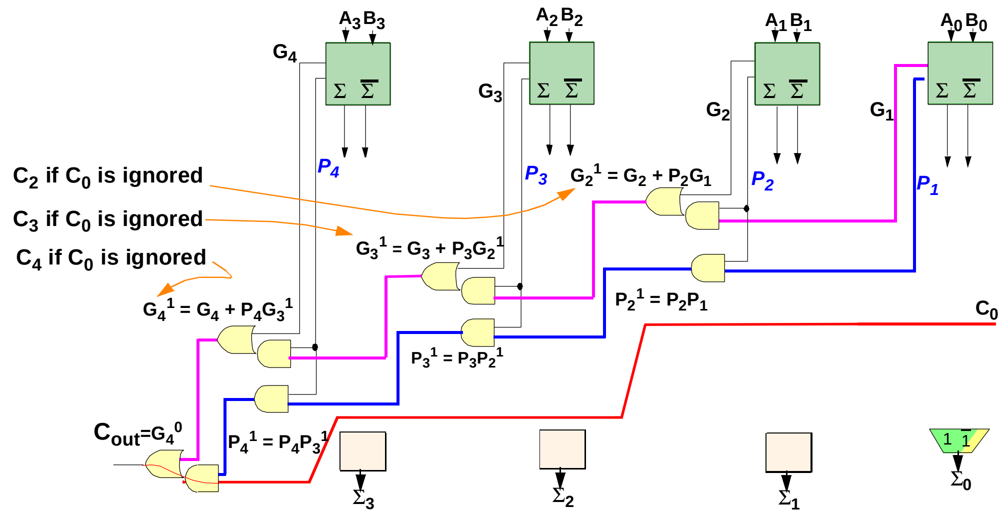


[1]. A. Bellaouar and M Elmasary, *Low-powered Digital VLSI Design Circuits and Systems*, Kluwer 1995, p.424 has a good summary of the csa..

Conditional Sum Adder

Carry Calculations

G_4^0 = carry out from the 4th adder = C_{out}
G_4^1 = carry out from the 4th adder with C_0 =0
$P_4^1 = P_4P_3P_2P_1$
If (P_4^1 =1) if a carry would propagate through all 4 adders
i.e $C_0 =1 \Rightarrow C_{out} =1$



Carry Calculations

The G chain

This is the same as the carry chain in the P and G ripple adder except it does not contain C_0.

It calculates C1, C2, C3 and C4, ignoring C0.

The P chain

This is the same as the P chain in the carry-bypass adder, except, as shown on the next page, it has taps to select the correct sum for individual full adders.

Comparison with other schemes

Combination of carry-select and carry-bypass adders

Like the carry-select adder, it calculates both $\Sigma(C_0=0)$ and $\Sigma(C_0=1)$.

It sends C_0 -> Cout directly if all propagate signals are true, like the carry-bypass adder. Thus the propagate time, if C_0 goes to Cout is almost the same as for the carry-bypass adder. The extra P line loading will slow it a little.

Note also this carry bypass is done for all the adders, for example Σ_4 is controlled by $P_3P_2P_1C_0$. This means the individual sum terms will be faster than in the carry look ahead adder which uses $G_3+P_3(G_2+P_2(G_1+P_1C_0)))$ to propagate C_0 into adder 4.

It uses the generalized generate signal G_n^k which signals if a carry comes from circuitry between adders n and k. This is like the Brent-Kung adder, except here it uses only G_n^1

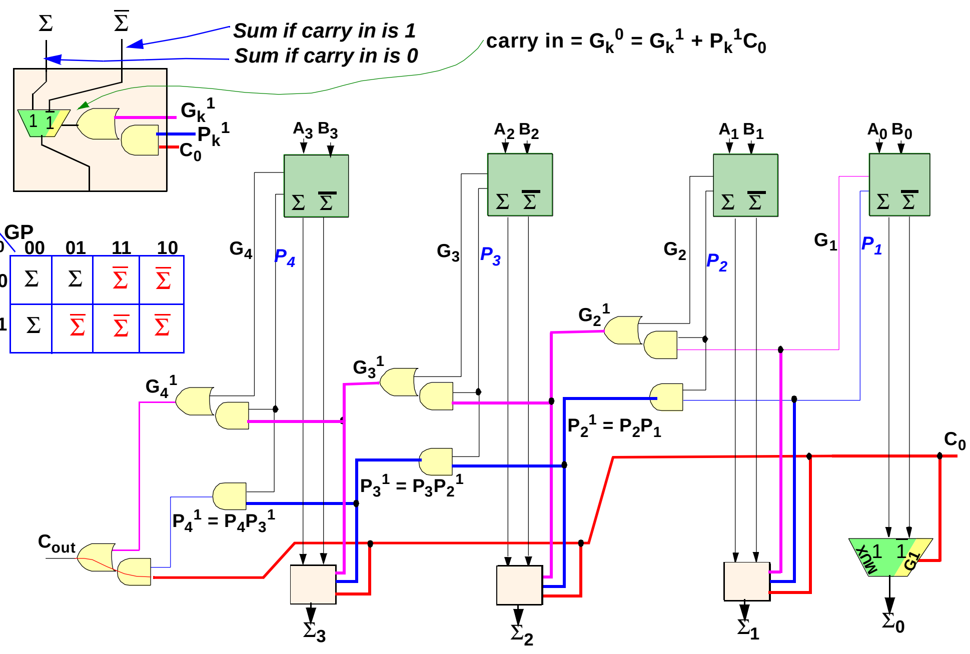
It does not use the logarithmic carry propagation so, for long word lengths, it will be slower than Brent-Kung.

There is an alternate implementation of the carry-select adder which uses transmission gates.[1]

[1]. See Jan M. Rabaey, *Digital Integrated Circuits,* Prentice Hall, 1996, Chapt. 7, Prob. 8, pp. 429-30.

Conditional Sum Adder

Selecting the sum terms



C_0 \ GP	00	01	11	10
0	Σ	Σ	$\overline{\Sigma}$	$\overline{\Sigma}$
1	Σ	$\overline{\Sigma}$	$\overline{\Sigma}$	$\overline{\Sigma}$

Conditional Sum Adder

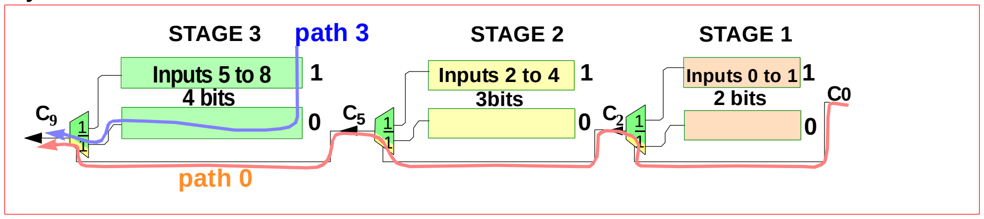
Reducing Delay By Gradually Increasing Stage Length

The stages calculate in parallel

- Their outputs reach the carry muxs at the same time (with equal lengths).
- Path 0 delay increases by a mux delay at each stage.

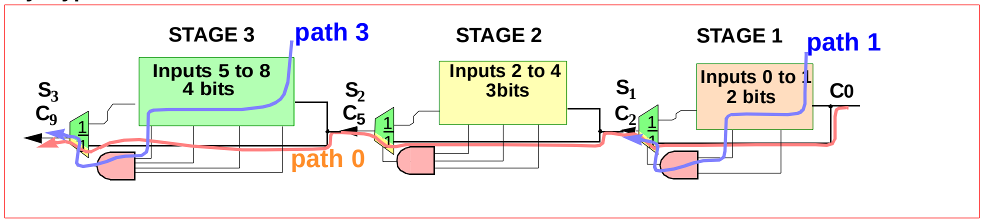
If (mux delay) ≈ (G P delay), can do one more add in each successive stage.

Carry-Select



Delay now increases as sqrt(n), O(√n), instead of linearly with n, O(n).

Carry-Bypass



Increasing Stage Length

Renumber the carrys by stage by making the output carry of stage k be $S_{k.}$ In the picture, S_1=C_2, S_2=C_4, ...

Balance delays so path 0 and path k are equal. Path k is the longest path from some stage k input to S_k.
τ_k is its path delay. Path 3 is shown.

In the first stage, signal C_0 will reach S_1 before the signal on path 1, so the delay to S_1 is $\tau_{1.}$

After that, as long as path k can reach mux k before the signal along path 0, then
(delay along path 0 up to carry S_k) = τ_{0k} = τ_1+k*τmux

The increase in delay between τ_{0k} and $\tau_{0(k+1)}$ is one mux delay as long as path k+1 can reach the mux k before the signal along path 0.

Thus we can increase the delay along each path k by one mux delay for each stage.

Thus $\tau_{0m,}$ the total carry-out delay for m stages, is τ_1+m*τ_{mux}

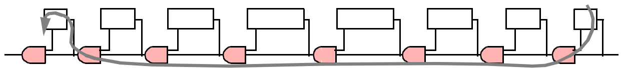
The number of full adders, n = 2+3+4+5+. . .+ m = m(m+1)/2 - 1. (Sum of arithmetic series)

Solve the quadratic equation to find m = -0.5 + √(2.25 + n)

For n>>2.25 $\tau_{0m} \approx \tau_{mux}\sqrt{n}$

Balancing the whole sum

Carry-Bypass



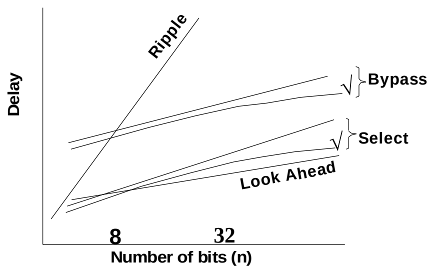
The circuits above have the delay to the final carry faster than to some of the high order bits of the sum. Then one of the critical paths is shown on the right. Here the stage length is decreased symmetrically on both ends to equalize the delay to the final sum bit in each stage.

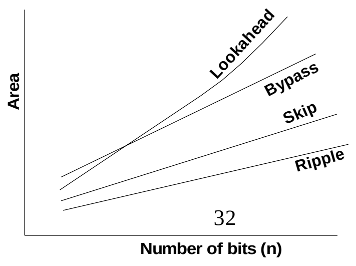
Carry-Select

Here the final sum is selected in parallel by the carry. Shortening the most-significant end is not necessary.

▪ Conditional Sum Adder ▪

Comparison of Adders[1]





Adapted from Rabaey.

The carry select adder area and speed depends on the stage "your favourite" adder type.

Details will change relative properties, particularly for low n.

[1] Jan M. Rabaey, Digital Integrated Circuits, Prentice Hall, 1995. pp. 399, 403, 424.

Summary of Adders

Ripple-carry adder is the smallest and the lowest power consumption, and for short words it may be fastest.

The bit-serial version is very small and very slow. It takes in and gives out bit streams. See Lealand Jackson, *Digital Filters and Signal Processing*, Kluwer 1989, pp 343-345.

The Brent-Kung adder is by far the fastest, but it gets very large.
The conditional-sum adder is the second choice for speed, and has much less area.

Experience with Small Adders

For small adders, O(n) approximations may be misleading.

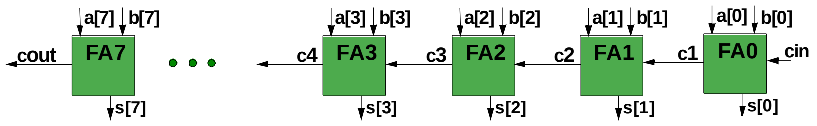
For 4 to 7 bit adds, using library Designware®, a Carleton graduate student, Youxing Zhao found:
The conditional sum adder (csa) was the fastest.
The ripple carry adder (rpl) was second and significantly slower.
The fast carry look-ahead (clf) was third.
The Brent-Kung (bk) and the carry look-ahead adder (cla) were last and about the same.

Verilog Adders

Ripple-Carry Adder

```
module ripple_add8(cout,s,a,b,cin);
   input  [7:0]  a, b,  input   cin;
   output [7:0]  s;   reg  [7:0]  s;
   output cout;     reg cout;
   integer i;
   wire   cy;

   always @(a or b or cin)
     begin
          cy=cin
          for (i=0; i<=7; i=i+1)
            begin
              {cy, s[i]) = a[i] + b[i] + cy;
            end
          cout=cy;
     end
   endmodule // ripple_add8
```



Verilog Adders

Ripple-Carry Adder

Connections

- The input and output ports, **a**, **b**, **cin**, **cout** and **s**, do not have to be declared again. The internal connection, **c1, c2, ...**, normally would be declared. However:
- Wires do not have to be declared explicitly if they serve as wiring between arguments of module instantiations. For example **c1, c2, ...**

Module Definitions

- We define a module **ripple_add8** and a module **fulladder**. **Ripple_add8** calls **fulladder** eight times.
- The definition of a module must be completely outside the definition of any other module. Note the **endmodule** statement for **ripple_add8** came before module **fulladder** started

Behavioural Model for Adder

The full adder was defined by logic equations rather than gates. This allows a logic synthesizer to choose how the gates are to be put together. For example it might factor the carry into:

$$a(b + c) + bc.$$

Normally the synthesizer will do a much better job than the designer. Two exceptions are:

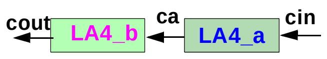
1. when custom cells available that are not in the synthesizer library.
2. When the logic is too much for the minimizer. Carry-select adders are probably too much.

▪ Verilog Adders ▪

Carry Lookahead Adder

```
module lookahead_add8(cout,s,a,b,cin);
    input  [7:0]  a, b;        input  cin;
    output [7:0]   s;          output cout;
    wire   ca;

    lookahead_4   LA4_a(ca,  s[3:0],a[3:0],b[3:0],cin],
                  LA4_b(cout, s[7:4],a[7:4],b[7:4],ca];
endmodule  //lookahead_add8
```



```
module lookahead_4(cout,s,a,b,cin);
    input   [3:0]  a, b, cin;
    output  [3:0]   s;       output cout;
    wires   [3:0]  c, p, g;

// Connect the carry, cin,  and carry lookahead  c[i].
    assign  c[0]  = cin,
            c[1]  = g[0]  | p[0]&c[0],
            c[2]  = g[1]  | p[1]&g[0]  | p[1]&p[0]&c[0],
            c[3]  = g[2]  | p[2]&g[1]  | p[2]&p[1]&g[0]
            | p[2]&p[1]&p[0]&c[0],
            cout = g[3]  | p[3]&g[2]  | p[3]&p[2]&g[1]
            | p[3]&p[2]&p[1]&g[0]  | p[3]&p[2]&p[1]&p[0]&c[0];

// Connect propagate and generate signals; connect the sum.
    assign  p = a^b,
            g = a&b,
            s = p^c;
end module // lookahead_4
```

Carry Lookahead Adder

Connections

- The input and output ports, **a**, **b**, **cin**, **cout** and **s**, do not have to be declared again. The internal connection, **ca**, was declared. However in this case it was optional (see below).
- Wires do not have to be declared explicitly if they serve as wiring between arguments of module instantiations. For example declaration of **ca** in **LA4_a** and **LA4_b**, is optional.

Nonprocedural Verilog Is a Circuit

- Note again that Verilog statements, except in procedures, are definitions of connections. The order of the statements does not matter any more than it matters which gate is put at the top of a wiring diagram.
- The two 4-bit sections are coupled by a ripple carry

The Carry Lookahead Code

- the equations were written to follow my guess at the fastest implementation. A good synthesizer may change the gate connections considerably.

Carry-Select Adder

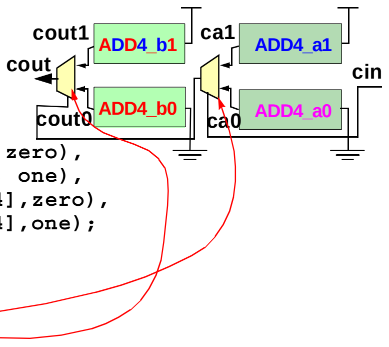
```
module select_add8(cout,s,a,b,cin);
    input  [7:0]  a, b;         input  cin;
    output [7:0]  s;            output cout;
    wire    [7:0] s0, s1;       wire  ca;
    parameter zero=0, one=1;

    add4   ADD4_a0(ca0,  s[3:0],a[3:0],b[3:0],zero),
           ADD4_a1(ca1,  s[3:0],a[3:0],b[3:0], one),
           ADD4_b0(cout0,  s[7:4],a[7:4],b[7:4],zero),
           ADD4_b1(cout1,  s[7:4],a[7:4],b[7:4],one);


//    The MUXs
    assign  ca   =  (cin) ? ca1 : ca0,
           cout  =  (ca)  ? cout1 : cout0,
           s[3:0]  =  (cin)  ? s1[3:0]  : s0[3:0],  // Not shown on diagram
           s[7:4]  =  (ca)   ? s1[7:4]  : s0[7:4];
endmodule  //select_add8


    module add4(cout,s,a,b,cin);
       input   [3:0]  a, b,     input   cin;
       output  [3:0]   s;       output  cout;

    // The 4-bit behavioural adder. Let the synthesizer decide.
            assign    {cout,s} = a + b + cin;
    endmodule  // add4
```



The Carry-Select Adder

Wire Declarations

- I tend to declare wires even when the default do not require it. It helps:
  a. to keep one from using the same symbol for two wires.
  b. to keep one confusing vectors and scalers, for example `cin` and `c[0]`.

Parameters

`parameter zero=0, one=1;`
This defines constants 0 and 1 at the start rather than deep inside the module. Then if one wants to change them, say the input should be asserted low logic, it is easy to do.

Concatenation Left of the “=”

Concatenation on the left side of an equal sign is handy:

```
          assign  {cout,s} = a + b + cin;
```

Precoded Verilog Adders

Libraries

Most sites have access to precoded operators.

In Synopsys a library is called Designware.

In dc one can see it by

```
> report_lib standard.sldb
```

In PKS one can see what is in the library using

```
> report_lib
```

The libaries will usually have:-
add, subtract,
various compares (signed, unsigned) >, <, <=, ==, ...
multiply

Adders, for example are differentiated according to:
- bit lengths of both operands
- two's compliment or unsigned (for overflow checking)
- carry propagation mechanism.

Module Generators and Libraries

The fast way to get circuits

Most logic synthesizers have several types of adders already created.
These may be Verilog descriptions coded as macros.
They may be already laid out.

Use a Behavioural Description with Libraries

The library or generator usually wants the simplest high-level description of the function:

```
{cout,s}= a + b;
```

Put it in a module like `add8`.

Tell the synthesizer you want the module implemented by a macro or cell.

Incrementers

Compare incrementers and Adders

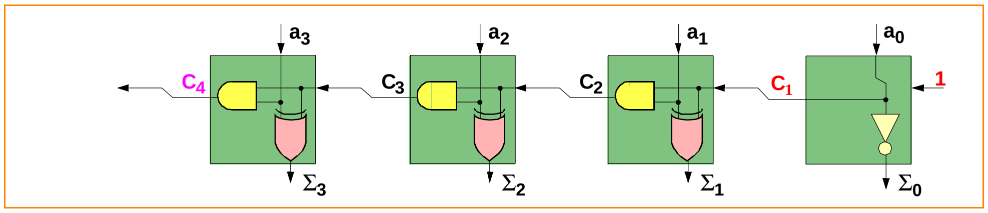
Full Adder

Add		$\Sigma = a \oplus b \oplus Cin$	$Cout = g + p\cdot Cin = a\cdot b + b\cdot Cin + a\cdot Cin$

Half Adder

Increment	b=0	$\Sigma = a \oplus 0 \oplus Cin$	$Cout = g + p\cdot Cin = a\cdot 0 + 0\cdot Cin + a\cdot Cin$
		$\Sigma = \underline{a} \oplus Cin$	$Cout = 0 + a\cdot Cin = a\cdot Cin$
First input	$C_0=1$	$\Sigma = a$	$C_1 = a_0$

4-Bit incrementer



Carry Lookahead

$C_4 = a_3 a_2 a_1 a_0$

Incrementer/Decrementer

Adders With One Input Fixed at One.

An adder can be used as an incrementer.
If a unit is to do nothing but increment, use a much simpler circuit.

- Adders are made of full adders. Incrementers are made of half adders.

Verilog and Incrementers

<u>How smart is your sythesizer?</u>

```
assign s = a +1;
```
// If the synthesizer knows about incrementers this should generate one.

```
parameter  one = 1;
  assign s = a +one;
```
// Maybe, but likely you will get an adder.

```
assign   cin = 1,
      s = a +cin;
```
// Unlikely

12. PROBLEM

Design a carry-select incrementer.

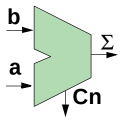
Hint: It is just a rearrangement of the AND gates.

Negative Numbers

Two’s Complement

The numbers

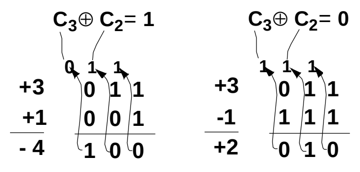
- **Positive numbers start with 0.**
- **Negative numbers start with 1.**
  The first bit tells the sign.
  -1 is always 1111 . . .
- **They use a normal positive-number adder.**



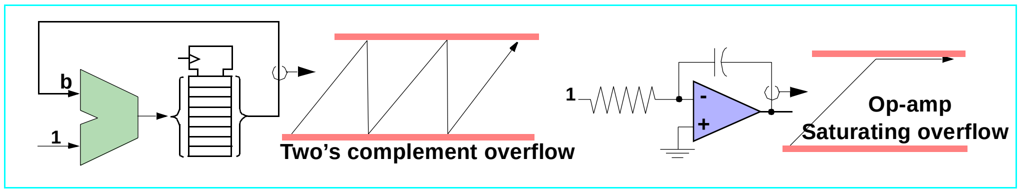
2’s complement	
+3	011
+2	010
+1	001
0	000
-1	111
-2	110
-3	101
-4	100

Overflow

- **Test for overflow is**
  Overflow = Cn ⊕ Cn-1
- **Maximum positive becomes maximum negative.**
  011 + 001 ⇒ 100   3 + 1 ⇒ -4



Example, accumulating adder v.s. integrator:

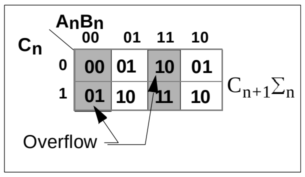


Subtraction

Common representations for signed numbers

1. Two’s compliment
   Uses a normal adder.
2. One’s complement
   Uses a normal adder except carry wraps around, Can double add times. Has two values representing zero.
3. Sign magnitude
   Cumbersome to implement.
   Normal output format for some A to Ds and some additive encoding compression schemes.

Overflow test for 2’s Complement



Adding numbers of opposite sign can never overflow.
Since **a[n]** and **b[n]** are the sign of a and b, **a[n]=b[n]**
is the only potential overflow.

Case (i) Numbers have same sign ie. **a[n]=b[n]**

If **a[n]=b[n]**, then **c[n]** = Σ**[n]**, (see map of Σ_n on right).
 ⇒ **c[n]** is the apparent sign of the number just as Σ**[n]** is.

The sum Σ**[n]** must have the common sign of **a[n]** and **b[n]** or there is overflow.
 But the sign Σ**[n]** = **c[n]**
 Further **c[n+1]** = 1 if **a[n]=1=b[n]**, **c[n+1]=0** if **a[n]=0=b[n]**

Deduce that **c[n+1]** ≠ **c[n]** ⇒ sign of Σ is opposite the common sign of **a** and **b** ⇒ overflow.
 That is **c[n+1]** ⊕ **c[n]=1** ⇒ overflow**.**

Case (ii) **a[n]**≠**b[n]**

If **a[n]**≠**b[n]**, then **c[n+1]** = **c[n]** and **c[n+1]** ⊕ **c[n]=0.** This agrees with no overflow.

Two's Complement (cont.)

Negating Numbers

a. Invert each bit.

b. Add1.

c. Ignore any off-end carry.

In Verilog

```
   wire [7:0] minus_a, a;
      minus_a = (~a)+1
```

+3	0 1 1	-1	1 1 1
~(+3)	1 0 0	~(-1)	0 0 0
	+ 1		+ 1
- 3	1 0 1	+1	0 0 1

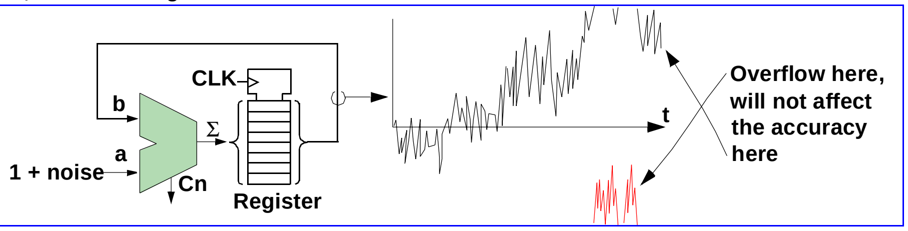
When Two's Complement Overflow Can Be Ignored

a. Do n-bit arithmetic on a signal

b. Allowed operations are +, -, and multiply by an integer x*17 OK ~~x*1.7~~.

c. If the correct output would not overflow n-bits, then internal overflows do not cause an error!

Example, accumulating adder



Two's Complement

Overflow

Two's complement overflow can be very bad because it goes from maximum positive to maximum negative.

On the other hand one can often recover from the overflow.

Recovery from overflow

Let x be a large number such that adding 3+x overflows.
Now immediately add -4 to the result. This will do a negative overflow and take the result back to x-1.
This is exactly the result if their had been no overflow.

Intermediate results which overflow cause no error if the correct final answer lies within range.

This applies only to addition and subtraction.
Multiplication by an integer is all right because that is equivalent to adding many times.
Multiplication by a fraction is not all right. There is an element of division destroys the overflow recovery.

Coding An Add/Subtract Unit

Coding for Synthesis

The Poor Way

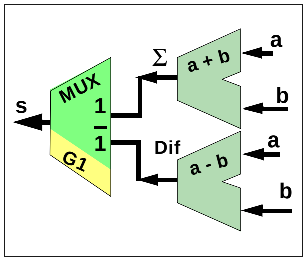
Depending on the value of cntr, s is assigned to either `a - b` or `a + b`.

```
module adder(s, cntr, a, b);
  output [7:0] s;
  input cntr;
  input [7:0] a , b;

    assign s = (cntr) ? (a+b) : (a-b);
```

Unless the synthesis tool is very smart. it will generate:

- **a mux from the conditional statement**
- **separate arithmetic units; one which adds, and one which subtracts.**



Subtraction

Making a subtractor from an adder

Converting Add to Subtract

To convert A + B into A - B:

1. Individually invert all the bits of B to ~B.
2. Apply ~B to the adder input. Remember we also had to add 1.
3. Do the adds A- B = A + (~ B + 1) = A + (~B) + 1.
   Do the +1 by sending 1 into C0 (Carry in)

Verilog Add/Subtract Circuit.

```
module add_subtract(overflw, cout, s, a, b, plus_minus_n)
// plus_minus_n =1  is add;    plus_minus_n = 0  is subtract.
   input [7:0] a,b; input plus_minus;
   output [7:0] s; output overflw, cout;
   wire [7:0] tildaB;
          assign tildaB = (plus_minus) ? b : ~b,                // or plus_minus ^ b
                 {cout,s} = a + tildaB + (~plus_minus),
                 overflw  = cout^((a[7]=tildaB[7])& s[7]);
endmodule
```

13. PROBLEM
    Estimate the area generated for this circuit vs the one on the next slide using XORs as a controlled inverter.

Coding for Synthesis

Smaller Faster Circuit

a - b can be performed by inverting b and adding "1".

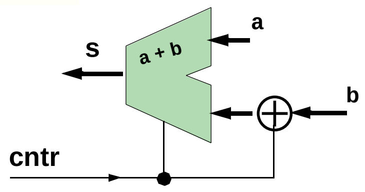
-b = ~b +1

Rewrite the code using `cntr` to:

- **form a conditional "add 1" (add the value of `cntr`)**
- **conditionally invert `b` by using XOR gates.**

```
module adder(s, cntr, a, b);
  output [7:0] s;
  input cntr;
  input [7:0] a , b;

    assign s = cntr + a +({8{cntr}} ^ b);
```



The XOR inverts b when cntr =1.

Coding For Synthesis

Having a reasonable concept of what the synthesizer will do will make smaller faster circuits.

Negating Two’s Complement Numbers

```
// ..........................................................................................
//     Starting at the least sig bit, x[0].  Pass it directly through to y
//     As long as x[i] is 0, don’t invert it.
//     After reaching the first x[i]=1, do not invert that x[i].
//     But do invert all bits  after the first x[i]=1.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    wire [N-1:0]cry, x, y;
   always @(x)
      begin
       //cry[0] = 0;   Tells whether to invert the next bit
       y[0]  = x[0];       //y[0] = x[0] ^0;
       cry[1]=x[0];
       y[1]  = x[1]^cry[1] ;
       cry[2]=x[1] | cry[1];
       y[2]  = x[2]^cry[2] ;
       cry[3]=x[2] | cry[2];
       . . .
      end
```

Negating Two’s Complement Numbers

The algorithm shown is fast and simple.

<u>**Examples**</u>

001010 = 10d

Start on the right, travel left.
As long as the bits are 0 leave them unchanged.

On the first 1 leave that unchanged,
but invert all bits from then on.
The colon shows the break between inversion and noninversion.

1101:10 =-10d

110100 =-12d

001:100 =-12d

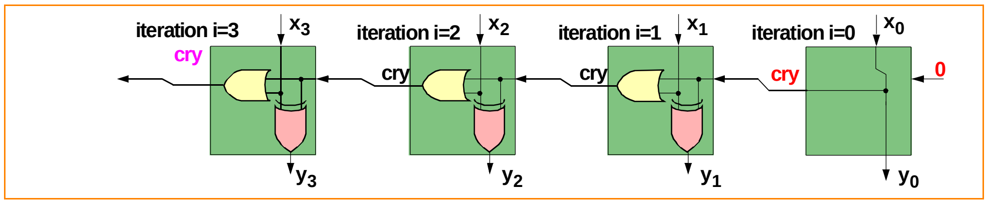
000101 = 5d

11101:1 =-5d

Circuit to find 2's Complememnt

```
reg  [N-1:0] y; // minus x
wire [N-1:0] x;
wire cry;
integer i;

always @(x)
  begin
     cry=0;      // Becomes 1 after the first  1 in x.l
    for(i=0; i<N; i=i+1)    // Start at least sig bit, move left.
    begin
       y[i]= x[i]^cry;      // Invert  x  next  i  after the first 1 in x is reached.
       cry = x[i] | cry;    // cry will remember when the first 1 was reached.
    end
  end
```



Finding the Two's Complement

This is a good circuit to use if you are not going to do an addition on the number after conversion.

If you are going to add the number immediately afterward, just invert each bit and make the carry-in for the adder equal 1.