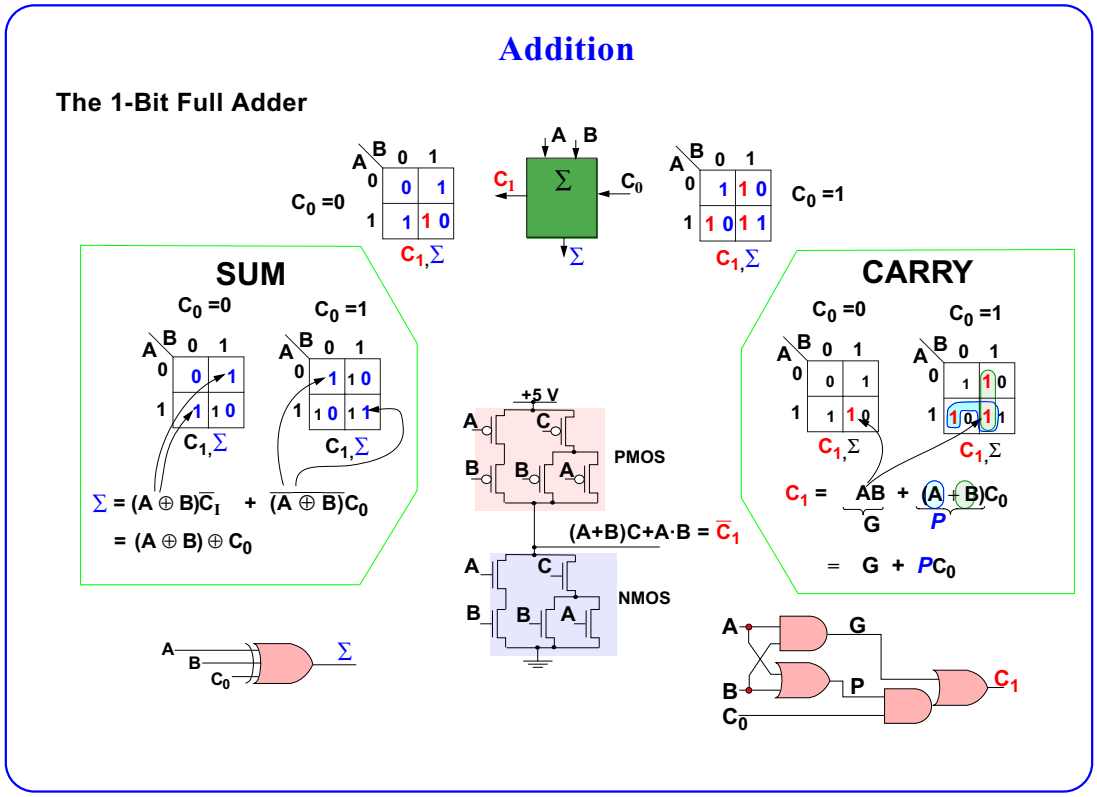
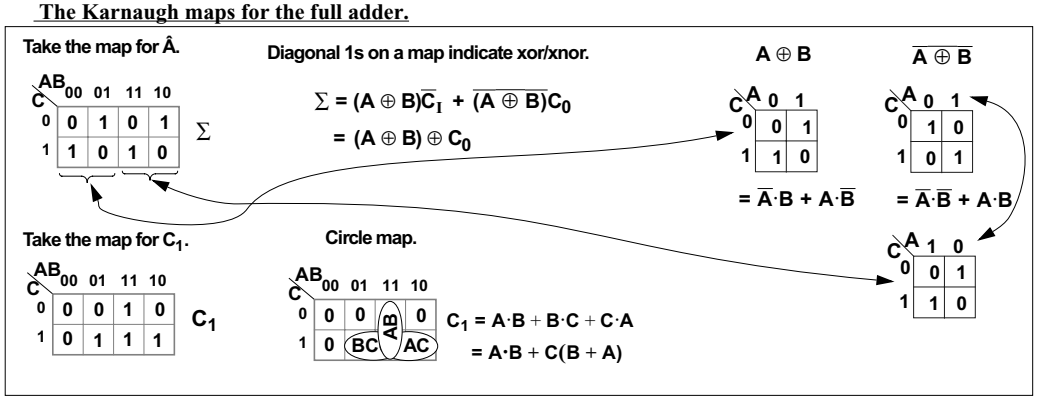


The 1-Bit Full Adder

It adds 3 bits, A + B +C_0

The Karnaugh maps for the full adder.



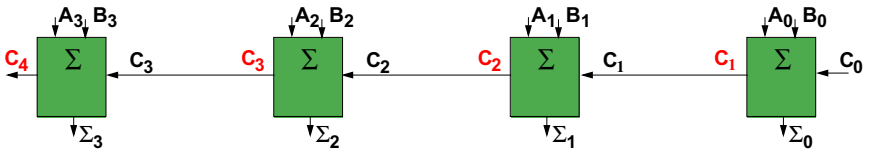
The CMOS carry circuit

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

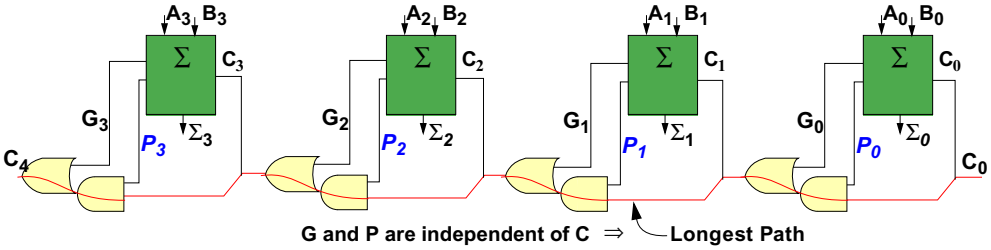
1. PROBLEM
   a. Find the expression it does implement.
   b. Show that it really is $\overline{C}_1$.

The 4-Bit Adder

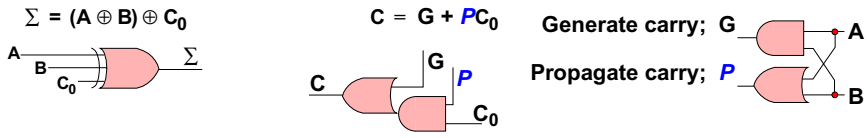
Ripple-Carry Adder



Ripple Carry Using Propagate and Generate



Basic Adder Blocks



2. PROBLEM

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

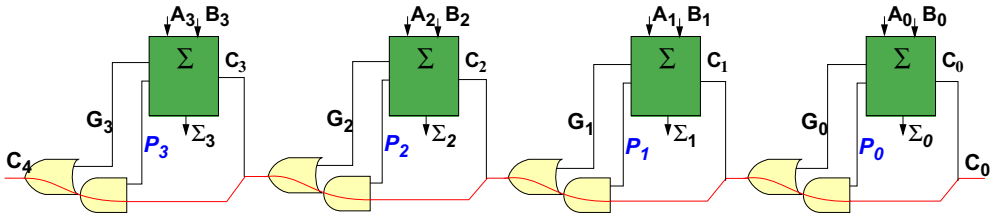
   Solution:

   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-skip adder demand that P=XOR

See “Using the Sum of Products (Σ of Π) for the PMOS function” on page 25.

Derivation of Carry Look-Ahead Blocks

Ripple Carry Using P and G (repeated)

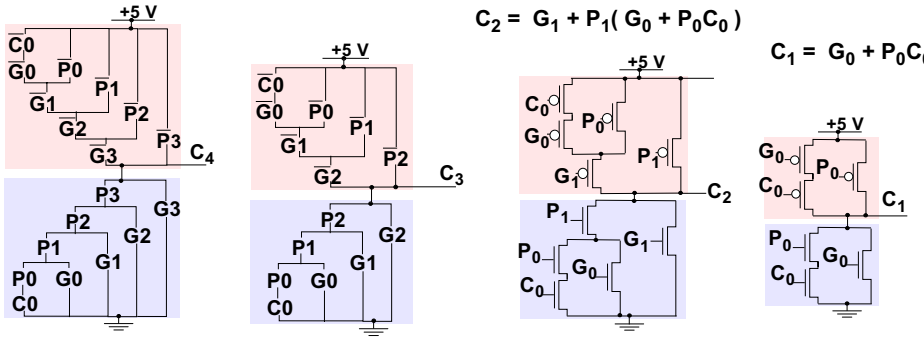


$C_4 = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0C_0)))$

$C_3 = G_2 + P_2(G_1 + P_1(G_0 + P_0C_0))$

$C_2 = G_1 + P_1(G_0 + P_0C_0)$

$C_1 = G_0 + P_0C_0)$



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 and reduce the delay to one complex gate. However for four adders in series, the complex gate will have 5 series transistors. This will also slow down the carry propagation.
Three to four adders is the maximum that can be cascaded with carry look-ahead.

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:

$$C_3 = G_2 + P_2 (G_1 + P_1 (G_0 + P_0C_0))$$

if one follows the methods in "Using the Sum of Products (Σ of Π) for the PMOS function" on page 25

Remember that $G_i = A_iB_i$, and $P_i = A_i + B_i$, hence one will never get the case $G_i,P_i = 1,0$.
These inputs become don't care outputs, i.e don't care squares on the map, for the C_3 equation.

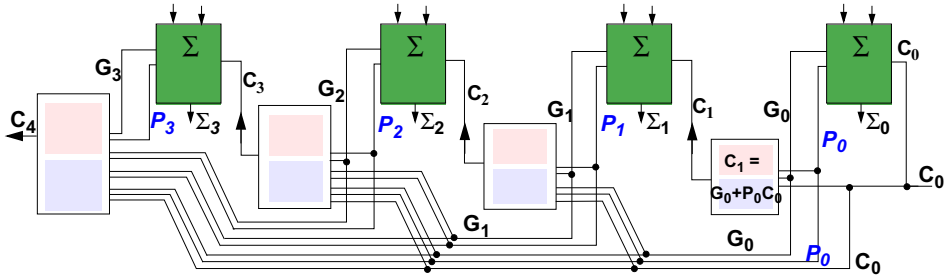
When the outputs for these inputs is ignored, the equation for the PMOS circuit can be derived.

3. PROBLEM

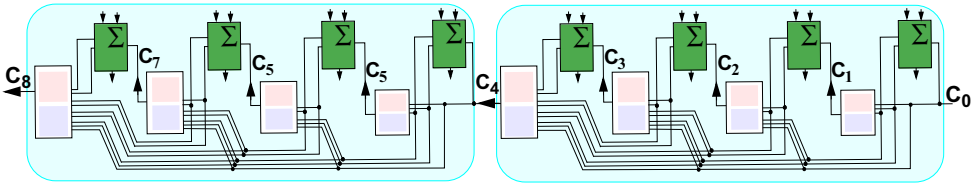
Derive the equation for the PMOS circuit from the NMOS one.
It may be easier to try C_2 rather than C_3.

The 4-Bit Carry Look-Ahead Adder

Carry Lookahead Using P and G



Look-Ahead Carry Blocks, Connected By Ripple carry



Carry Look-Ahead (cont)

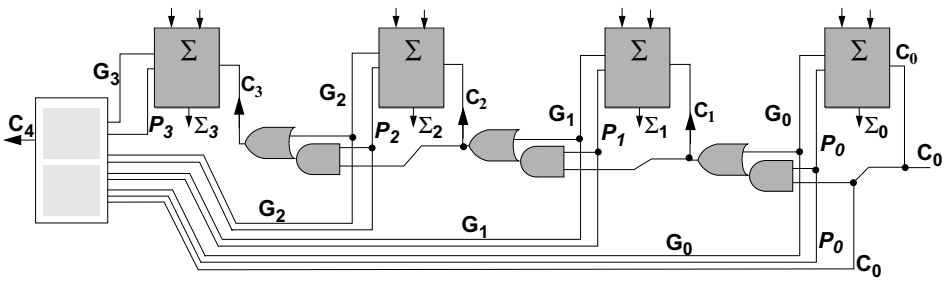
Look-Ahead

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*.

4. PROBLEM

There are five outputs, Σ_1, Σ_2. Σ_3, Σ_4 and C_4.
Rate each output as being slower, the same, or faster than the *carry look-ahead adder.*
Rate each output as being slower, the same, or faster than the *P and G ripple-carry adder.*
Compare the area with the the *carry look-ahead* and the *P and G ripple-carry adders*.



Describe a senerio where this might be the best of the three adders. Hint: *Consider look-ahead carry blocks*.

The Brent-Krung Carry-Lookahead Adder

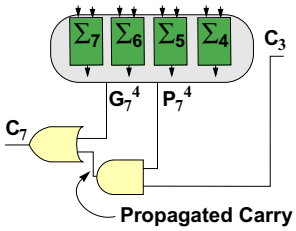
Generalized *generate* and *propagate*.

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

If $P_m^k=1$ then C_{k-1} can propagate thru adders m to k
It comes out as C_m

G_m^k is a *generate* for adders m thru k.

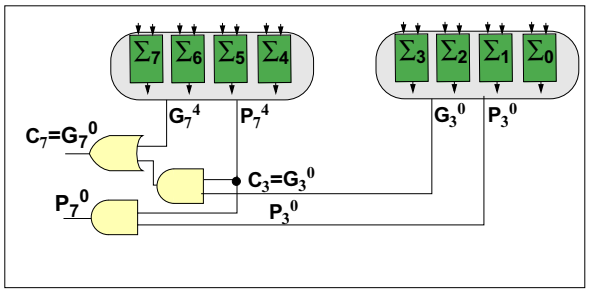
If $G_m^k=1$ a carry was generated by adders m thru k.

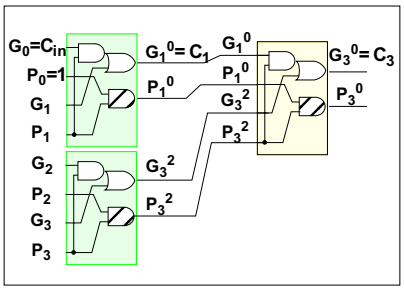


Combining P_m^k and G_m^k

$$G_m^k = G_m^j + P_m^j G_{j-1}^k$$

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





The Brent-Krung 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, and is defined by:-

$P_m = a_m + b_m$, $\quad P_m^k = P_m P_{m-1}^k = P_m^j P_{j-1}^k$ $\quad P_0 = 1$

Examples:

$P_5^2 = P_5 P_4 P_3 P_2$ $\quad P_m^0 = P_m P_{m-1} P_{m-2} \ldots P_0$

Also G_m^k is a *composite generate*

$G_m = a_m b_m$, $\quad G_m^k = G_m + P_m G_{m-1}^k = G_m^j + P_m^j G_{j-1}$ $\quad G_0 = C_0$ which is often 0

Examples:

$G_m^0 = C_m$

$$\begin{aligned} G_5^3 &= G_5 + P_5 G_4^3 \\ &= G_5 + P_5(G_4 + P_4 G_3^3) \\ &= (G_5 + P_5 G_4) + P_5 P_4 G_3 \\ &= G_5^4 + P_5^4 G_3 \end{aligned}$$

Larger Example

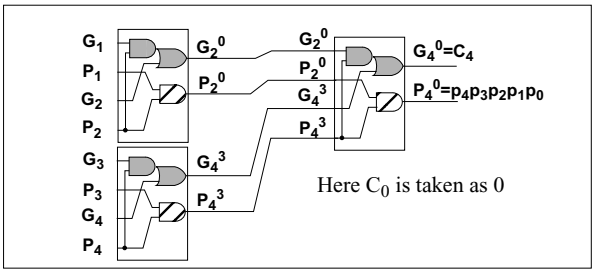
$G_1^0 = C_1 = G_1 + P_1 C_0 = G_1 + 0$

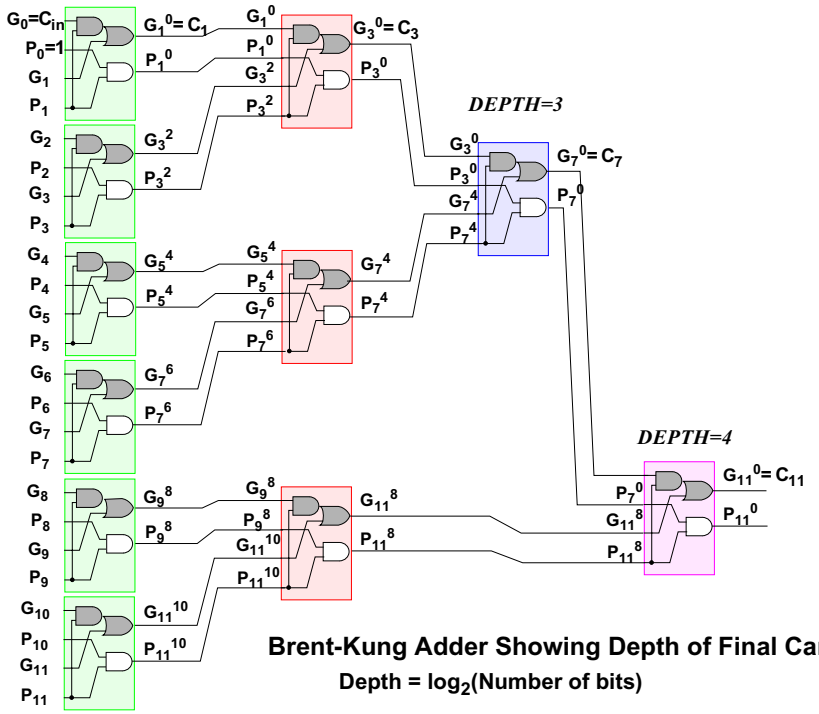
$G_2^0 = C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1^0$

$G_4^3 = G_4 + P_4 G_3$

$G_4^0 = G_4^3 + P_4^3 G_2^0$

$= (G_4 + P_4 G_3) + (P_4 P_3)(G_2 + P_2 G_1^0)$



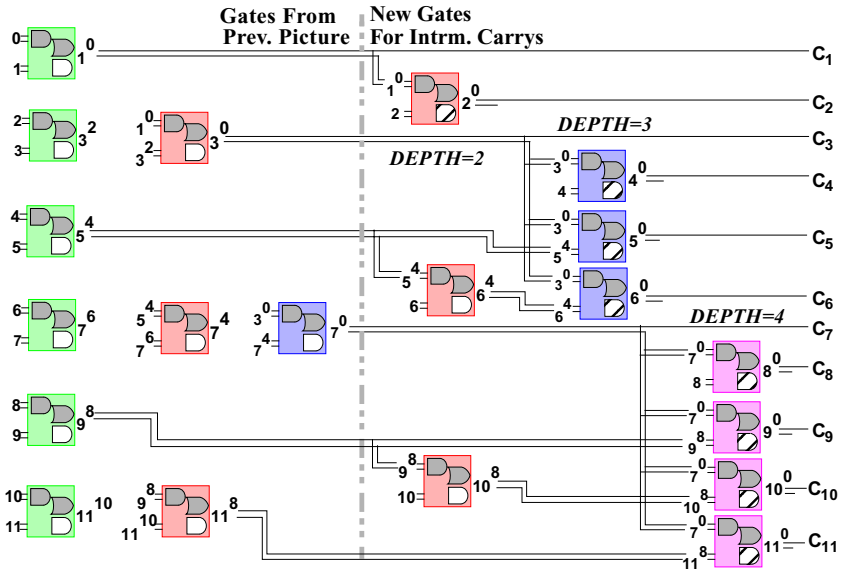


The Brent-Kung Adder (Cont)

- 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 *log* $_{\text{base 2}}$ of the number of bits.
- This adder has the minimum hardware of all the *carry-lookahead* adders.
- It is still a big power-hungry adder, but fast

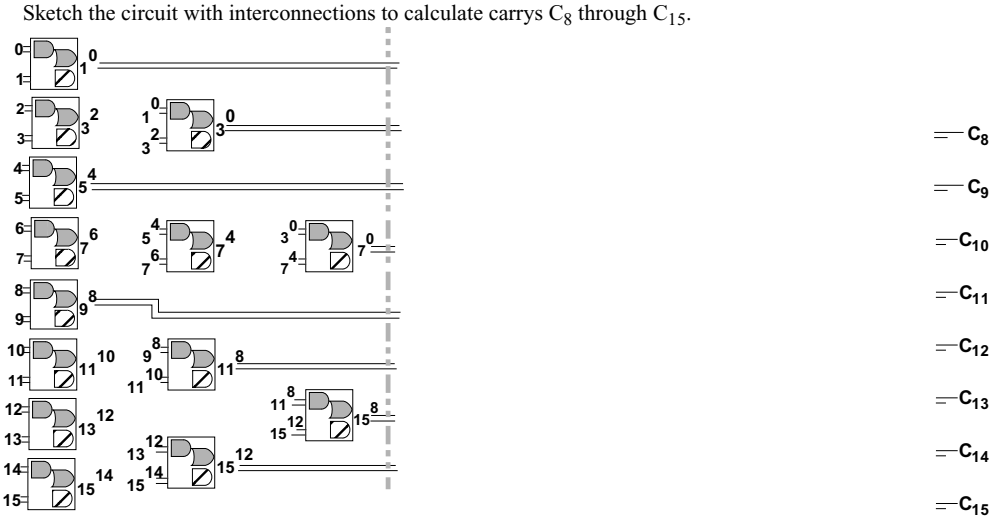
Brent-Kung Adder

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



5. 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}.



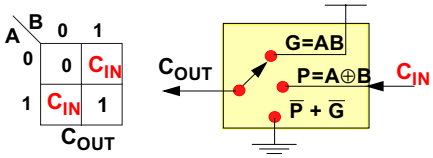
The Carry-Skip Adder)

Generate and Propagate Revisited

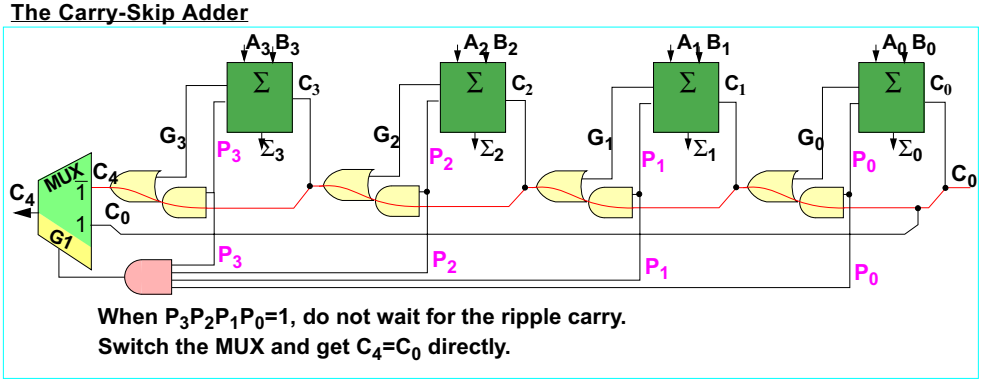
- Redefine P=A⊕B
- Then when P=1, $C_{OUT} = C_{IN}$

Extend this across 4 bits

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



The Carry-Skip Adder



When $P_3P_2P_1P_0=1$, 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-Skip Adder

This circuit allows the carry to skip over 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

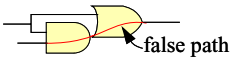
6. PROBLEM

Considering how the number of series transistors and the amount of capacitive load effect the propagation delay, which circuit would be faster to deliver C_4 in the the worst case:
The *carry look-ahead* adder?
The the combined *carry look-ahead* / *ripple-carry* adder of PROBLEM 4.?
The *carry-skip* adder?

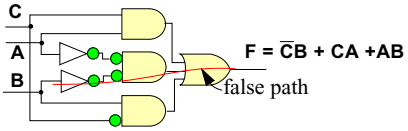
False Paths

A false path is one which will never propagate a signal change under proper operation.
However there is a connection through gates from the start to the end of the path.

False path when the complete path cannot be turned on at once. The carry-skip 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 $\overline{\mathbf{C}}\mathbf{B}$.



False Paths

Paths that will never propagate a signal change

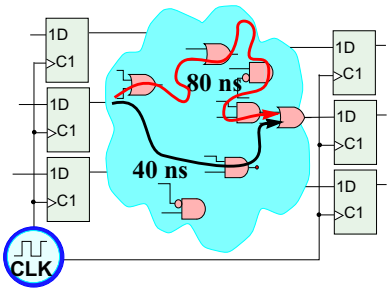
Long unused paths cause two problems, timing and testing

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 80 ns.

If the 80 ns path is a redundant carry path, and the next longest real path is 40 ns.

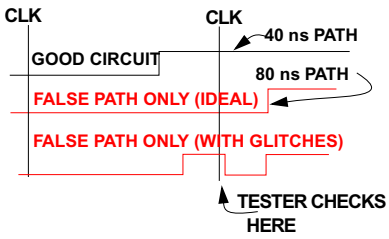
- The verifier will state the clock period > 80ns.
- You will likely believe it!



Testing Problems

Suppose the MUX in the carry-skip 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-Skip Adder

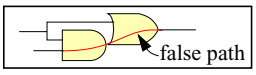
Timing Problems.

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.

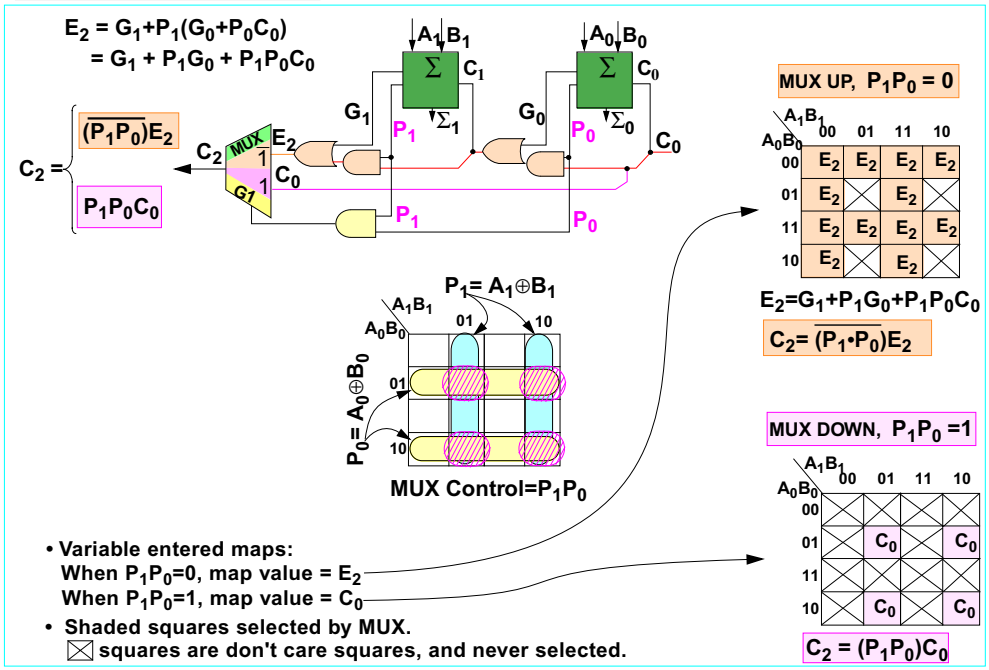


Static Timing Verification

- 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 tured on at once.
- Thus they will suggest making the clock slower than is actually needed.

The Redundant Carry-Skip Adder (Carry-Bypass)

Case I. The Redundant Path



False Paths in the Carry-Skip 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.

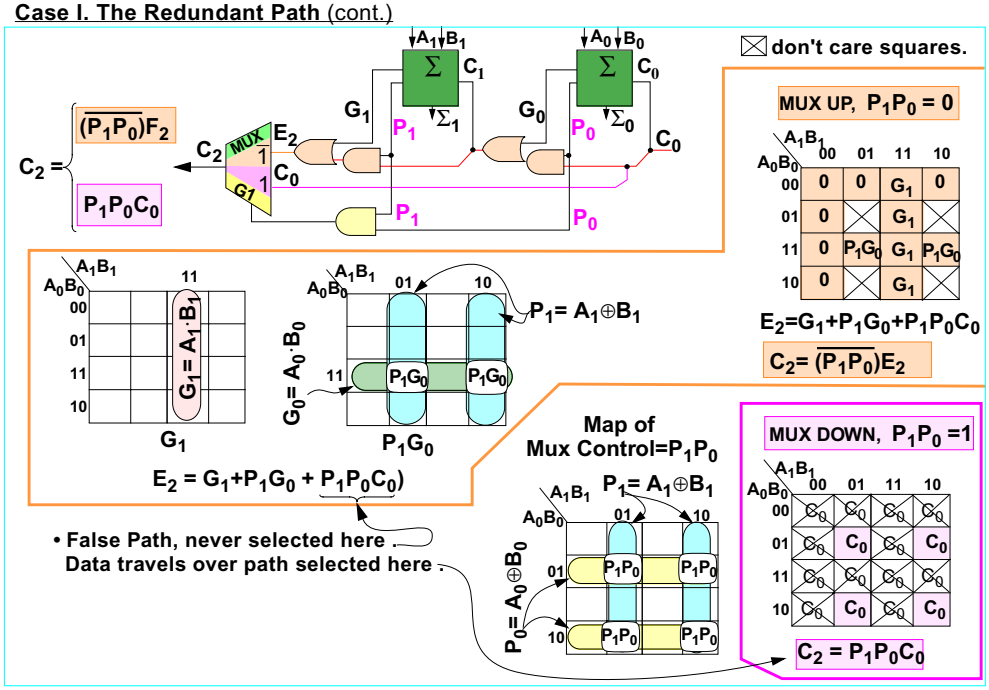
Faster Circuits Do Not Have To Have False Paths

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.

[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-Skip Adder (Carry-Bypass)

Case I. The Redundant Path (cont.)



The Nonredundant Carry-Skip Adder

The Maps for E_2, the Upper MUX Input

- The left hand three maps derive the expression for E_2. The left map encircles the G_1 term. These squares will be "1"s of E_2.
- The centre map encircles the two columns of P_1 and the row of G_0. The term is $P_1 \cdot G_0$ so the intersection of the circles will be "1"s of E_2.
- The right map shows $P_1 \cdot P_0 \cdot C0$. $P_1 \cdot P_0$ is four squares at the intersections of the columns, P1, and the rows, P0. The value C_0 is placed in those 4 squares. This avoids making a 5-variable map.
- The OR of the three maps is E_2 and is shown in the "MUX UP" 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_1P_2=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.

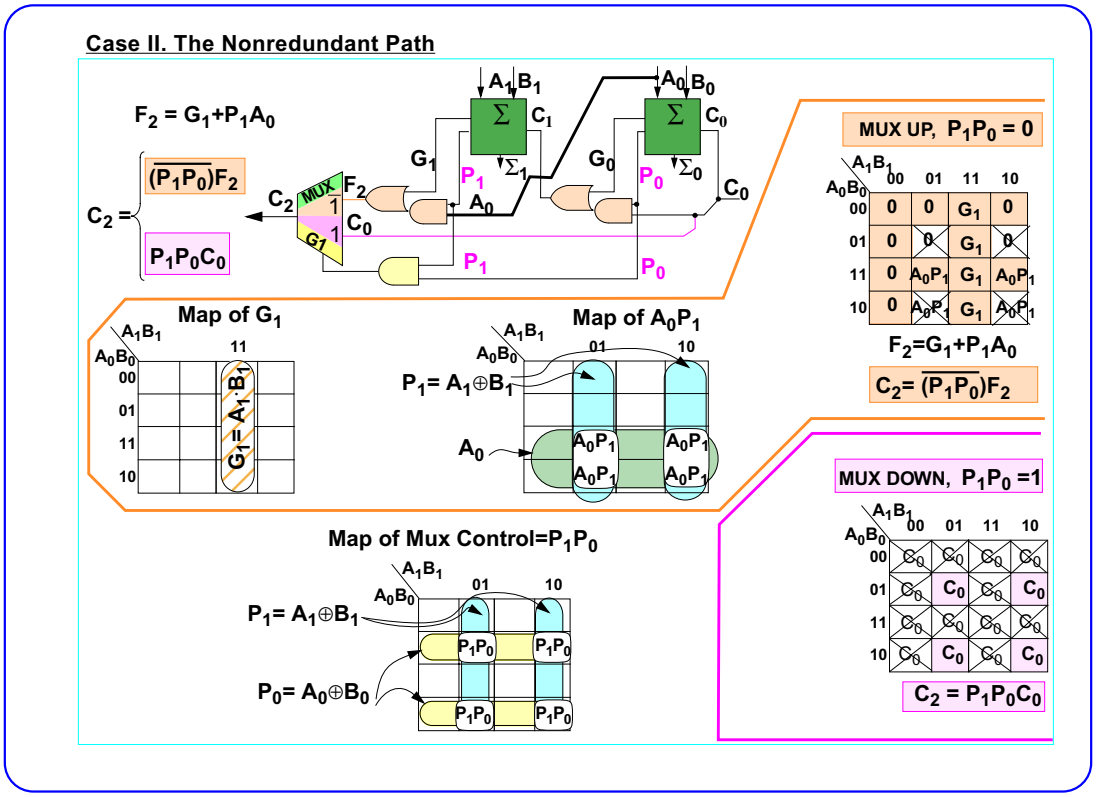
The Carry-Skip Adder, Nonredundant Circuit

Here the $P_1P_0C_0$ 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.

$E_2 = G_1 + P_1G_0 + P_1P_0C_0)$ $\qquad$ $F_2 = G_1 + P_1A_0$

- The $P_1P_0C_0$ term only appeared in the don't care squares. It was removed.
- The P_1G_0 only appeared in two squares. It was replaced by a P_1A_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. ☺

7. PROBLEM

Recall that here $P_1 = A_1 \oplus B_1$. The term P_1A_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

More silicon for speed.

Uses Two Adders.

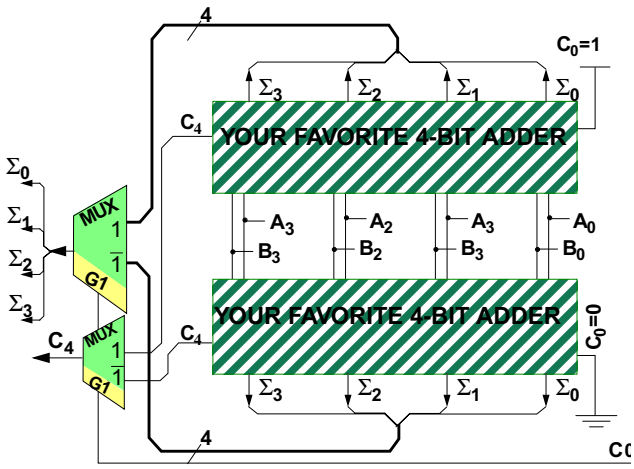
- Do 2 parallel adds
- One based on C_0=0. The other for C_0=1.
- Use C_0 to select the correct answer.

Time Saving

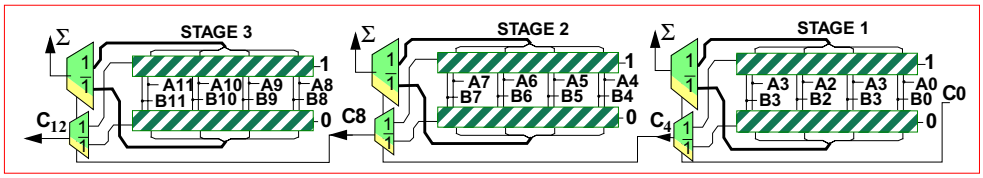
One stage saves little time.

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

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



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.

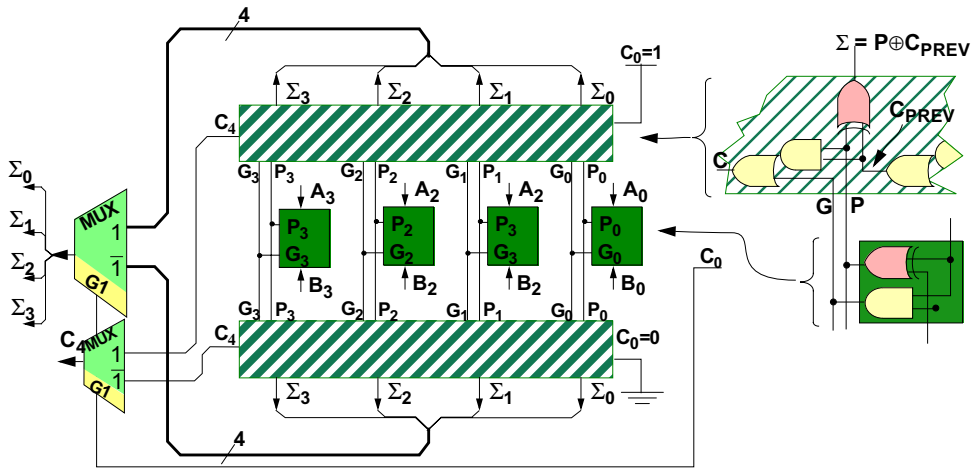
This delay increases linearly with the number of MUXs. However it is faster than most other systems which increase linearly with the number of full adders.

8. PROBLEM

Does the carry-select adder contain redundant paths?

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.

9. PROBLEM

Since the c0 input is known to be 1 or 0, redesign the first full-adders in each 4-bit chain to utilize this fact.

Summary of Adders

Ripple-carry adder is the smallest and the lowest power consumption.

The carry-select adder achieves more speed for more power.

Some other adders

The fastest adder theoretically is said to be the conditional-sum adder, described in:
Bellaouar and Elmasary, *Low-Power Digita VLSI Design*, Kluwer1995, pp 423-428.

The smallest and slowest adder is the bit-serial adder, which takes in and gives out bit streams. See Lealand Jackson, *Digital Filters and Signal Processing*, Kluwer 1989, pp 343-345.

Experience with Adders

For a 4 to 7 bit adds in a Viterbi decoder, a Carleton graduate student, Youxing Zhao found:
The conditional sum adder (csa) was the fastest[1].
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.

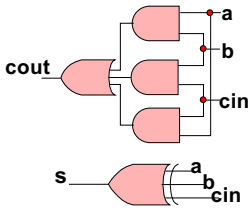
[1] A. Bellaouar and M Elmasary, *Low-powered Digital VLSI Design Circuits and Systems*, Kluwer 1995, p.424 has a good summary of the csa. Each full adder calculates (S1,C1) and (S0, C0) for a carry-in of 1 and 0 respectively. Then muxs are used to select the answer.

Verilog Adders

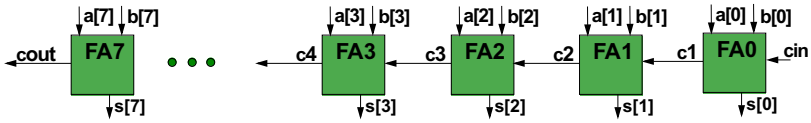
Ripple-Carry Adder

```
module ripple_add8(cout,s,a,b,cin);
   input [7:0] a, b,  input   cin;
   output[7:0]  s;    output cout;

   fulladder FA0(c1,  s[0],a[0],b[0] ,cin),
             FA1(c2,  s[1],a[1],b[1] ,c1),
             FA2(c3,  s[2],a[2],b[2] ,c2),
             FA3(c4,  s[3],a[3],b[3] ,c3),
             FA4(c5,  s[4],a[4],b[4] ,c4),
             FA5(c6,  s[5],a[5],b[5] ,c5),
             FA6(c7,  s[6],a[6],b[6] ,c6),
             FA7(cout,s[7],a[7],b[7] ,c7);
endmodule // ripple_add8
```




```
module fulladder (cout, s, a, b, cin);
   input a, b, cin;
   output s,cout;
      assign s=a^b^cin,   cout= a&b | a&cin | b&cin;
endmodule // fulladder
```



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:

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

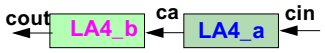
Normally the synthesizer will do a much better job than the designer. Two exceptions are:

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.

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] p, g, c;

   // 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

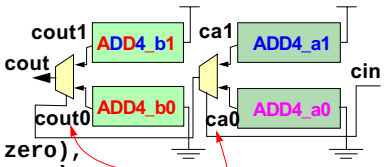
```
module select_add8(cout,s,a,b,cin);
   input [7:0] a, b;        input  cin;
   output[7:0]  s;          output cout;
   wire  [7:0] s0, s1;      wire  ca;
   parameter zero=0, one=1;

   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`

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

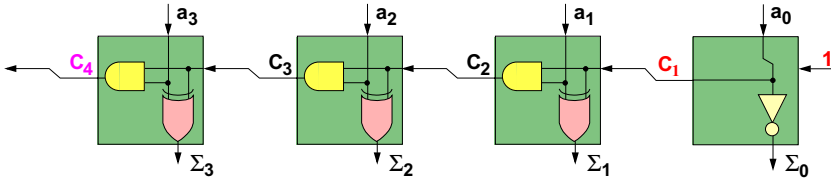
Full Adder

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

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

How smart is your sythesizer?

```
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

10. PROBLEM

Design a carry-select incrementer.

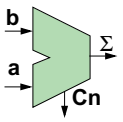
Hint: It is just a rearrangement of the AND gates.

Subtraction

Two's Complement

The numbers

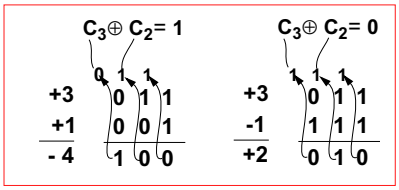
- Positive numbers start with **0**.
- Negative numbers start with 1.
  The first bit tells the sign.
  -1 is always 1111 . . .
- They use a normal positive-number adder.



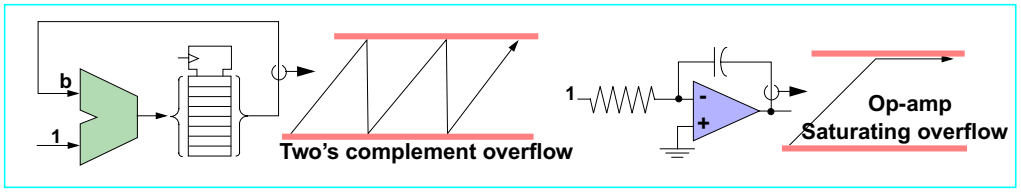
2's complement	
+3	011
+2	010
+1	001
0	000
-1	111
-2	110
-3	101
-4	100

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:



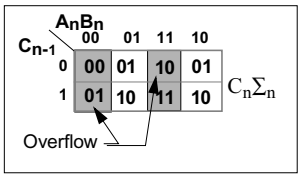
Subtraction

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.

+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	

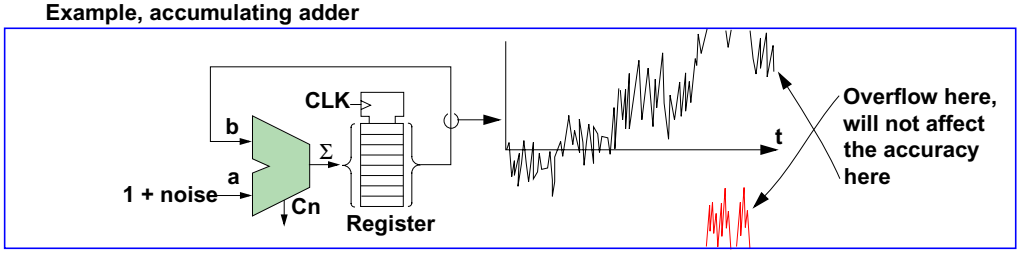
In Verilog

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

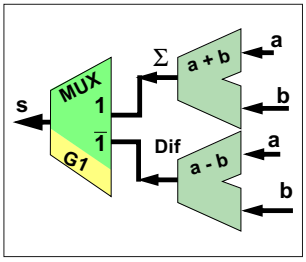
Depending on the value of cntr, S is assigned to either a - b or a + b.

```
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
```

11. 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

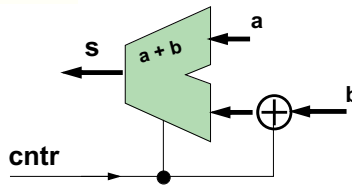
`a - b` can be performed by inverting `b` and adding "1".
`-b = ~b +1`

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

Negating Two's Complement Numbers

```
// ..........................................................................................
//       Starting at the lsb, x[0]:
//       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 x[i] bits checked after the first x[i]=1.
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   always @(x)
     begin
         minus_x[0] = x[0]^0    // cry[0]=0;
         cry[1]=x[0];
         minus_x[1] = x[1]^cry[1] ;
         cry[2]=x[1]&cry[1];
         minus_x[2] = x[2]^cry[2] ;
         cry[3]=x[2]&cry[2];
         . . .
   end

   endmodule    // negate
```


Negating Two's Complement Numbers

The algorithm shown is fast and simple.

Examples

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

111011 =-5d