

COMBINATIONAL COMPONENTS

We have discussed how:

- to analyze and synthesize single and multiple output networks with two or more levels
- to detect stuck-at faults
- to remove localized redundancy using fault detection techniques
- to use Boolean algebra to detect and remove certain timing effects – static and dynamic hazards

All of these are used to implement reliable building blocks that make up digital processors.

We need to investigate the combinational components of digital processors:

- adders
- logic units
- arithmetic units
- decoders/selectors (demux/mux)
- encoders
- ROMs and PLAs

Once these are available they can be used to implement the combinational portions of the processor (those parts that do not have internal memory)

All of these are discussed well in Chapter 5 which should be read carefully. We will go over a subset in class and in the notes.

RIPPLE-CARRY ADDER

Addition: Assume X and Y are two integers with binary representations $(x_k \dots x_1 x_0)$ and $(y_k \dots y_1 y_0)$ respectively.

- The sum $S = X + Y$ is denoted $(s_{k+1} s_k \dots s_1 s_0)$
- c_i will denote the binary carry from the production of s_{i-1} that must be added to the sum defining s_i .
- $c_0 = 0$ by definition and $s_{k+1} = c_{k+1}$
- $(c_{i+1}, s_i) = x_i + y_i + c_i$ (where the addition is base 2)
- We must convert the binary operations into Boolean operations.

Example:

$$\begin{array}{l} X : - 1 1 0 1 1 \\ Y : - 1 0 0 1 1 \\ C : 1 0 0 1 1 0 \\ S : 1 1 1 1 1 0 \end{array}$$

Determine Truth Tables for Sum and Carry

Each (c_{i+1}, s_i) pair is a function of three switching variables x_i , y_i and c_i . So we have 8 input patterns possible for which we must determine the two digit binary number that results from base 2 addition of the three binary digits.

x_i	0	0	0	0	1	1	1	1
y_i	0	0	1	1	0	0	1	1
c_i	+0	+1	+0	+1	+0	+1	+0	+1
(c_{i+1}, s_i)	00	01	01	10	01	10	10	11

Note this is essentially the truth tables for the two output functions (c_{i+1}, s_i) .

The Sum Function:

We can generate the canonical sum of products for s_i by considering the table.

$s_i = 1$ for the assignments

x_i	0	0	1	1
y_i	0	1	0	1
c_i	1	0	0	1

These correspond to the following SOP:

$$s_i(c_i, x_i, y_i) = c_i x_i' y_i' + c_i' x_i y_i' + c_i' x_i' y_i + c_i x_i y_i$$

We can now notice:

- The assignments are all at least 2 bits apart. So the fundamental products are all surrounded by 0s.
- There are no consensi w/r to c_i , x_i , or y_i and no coverage. So first Tison leaves the sum unchanged.

Both indicate the canonical sum is the complete sum, i.e., the terms are the PIs of s_i .

We can now notice:

- Since all PIs are isolated they are all essential PIs.
- There are no consensi w/r to c_i , x_i , or y_i and no coverage. So second Tison shows no redundancy in the complete sum, i.e., adding the monofom selection variables (A,B, . . .) to the products does not create consensi.

Both indicate the complete sum is the unique minimal sum.

So the minimal double-rail AND/OR implementation is

$$s_i(c_i, x_i, y_i) = c_i x_i' y_i' + c_i' x_i y_i' + c_i' x_i' y_i + c_i x_i y_i$$

The Carry Function:

$c_{i+1} = 1$ for the assignments where there are two or more 1s

x_i	0	1	1	1
y_i	1	1	0	1
c_i	1	0	1	1

$$c_{i+1}(c_i, x_i, y_i) = c_i x_i' y_i + c_i' x_i y_i + c_i x_i y_i' + c_i x_i y_i$$

Run first Tison:

$$\begin{aligned}c_{i+1} &= c'_i x_i y_i + c_i x_i y_i + c_i x'_i y_i + c_i x_i y'_i \\ \text{cons}(c_i) &= x_i y_i + c_i x'_i y_i + c_i x_i y'_i \\ \text{cons}(x_i) &= x_i y_i + c_i y_i + c_i x_i y'_i \\ \text{cons}(y_i) &= x_i y_i + c_i y_i + c_i x_i\end{aligned}$$

Now note that the complete sum has all monofom variables and is, therefore, unate.

c_{i+1} is unate and therefore, the complete sum is the unique minimal sum.

What about MOPIs?

$$s_i \bullet c_{i+1} = c_i x_i y_i$$

which is already a PI of s_i so the MOPIs are just the union of PIs for s_i and c_{i+1} .

It is easy to show that the individual minimals form the minimal multiple output form, i.e., using MOPIs does not reduce AND/OR form cost.

The AND/OR implementation is costly in terms of area and literals. We need to reexamine the functions and see if an alternate gate set would help.

Recall, assignments where $s_i = 1$.

x_i	0	0	1	1
y_i	0	1	0	1
c_i	1	0	0	1

Note $s_i = 1$ only when there are an odd number of 1s in the pattern. This is an odd parity check which we know is $s_i = c_i \oplus x_i \oplus y_i$. This is two two-input XOR gates.

We can also easily derive this from the minimal SOP:

$$\begin{aligned} s_i &= c_i x_i' y_i' + c_i x_i y_i + c_i' x_i' y_i + c_i' x_i y_i' \\ &= c_i (x_i' y_i' + x_i y_i) + c_i' (x_i' y_i + x_i y_i') \\ &= c_i (x_i \oplus y_i)' + c_i' (x_i \oplus y_i) \\ &= c_i \oplus x_i \oplus y_i \end{aligned}$$

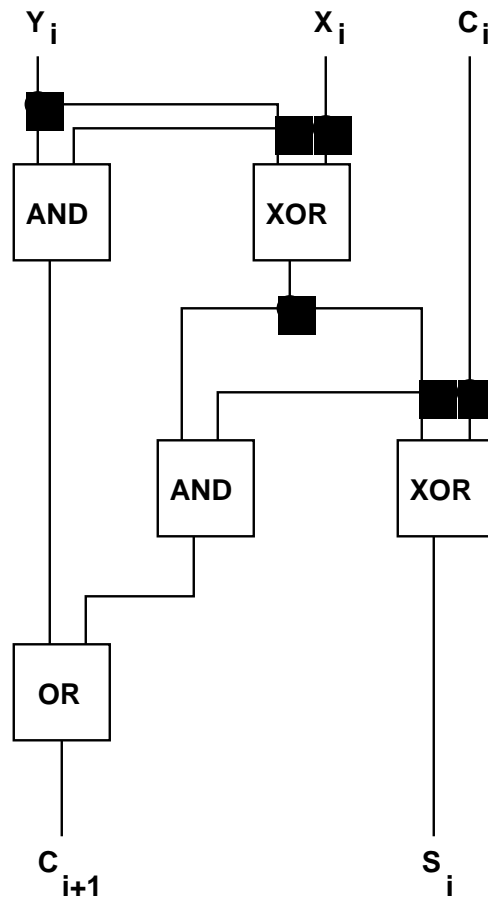
What about c_{i+1} ?

$$\begin{aligned} c_{i+1} &= c_i' x_i y_i + c_i x_i y_i + c_i x_i' y_i + c_i x_i y_i' \\ &= x_i y_i + c_i (x_i' y_i + x_i y_i') \\ &= x_i y_i + c_i (x_i \oplus y_i) \end{aligned}$$

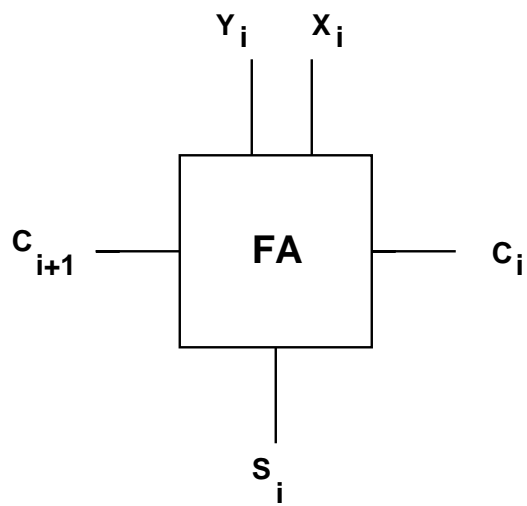
These two equations form the preferred form of a **FULL ADDER**. Two-input AND, OR and XOR gates are all that are required. The maximum fan-out is two. Very economical.

One is used for each bit of the sum.

Logic Network

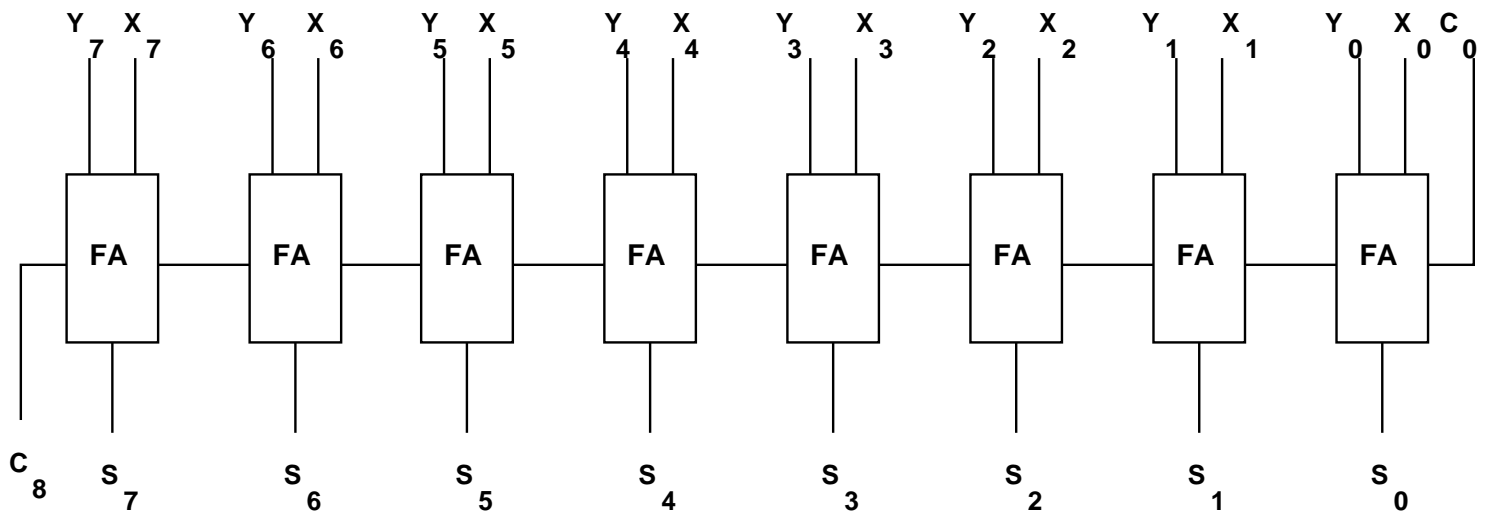


Component Symbol



8 Bit Ripple-carry Adder

$C_0 = 0$



SUBTRACTION

Subtraction uses 2's complement definition and ignores signs of operands.

To compute $X - Y$:

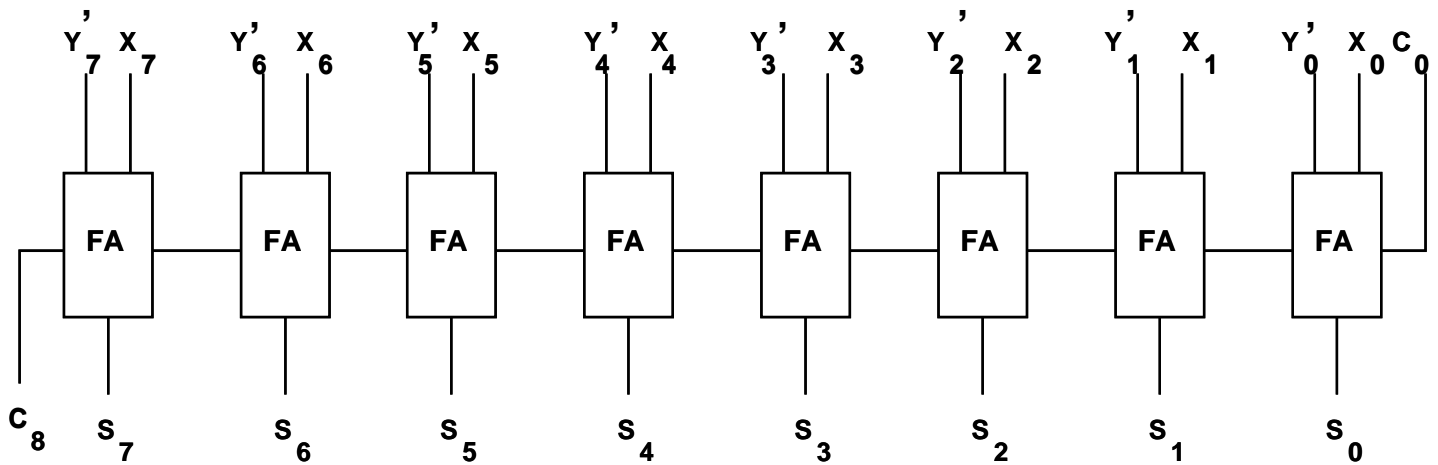
- form Y' the 2's complement of Y
- add $X + Y'$
- need to evaluate Y' efficiently
- get Z the 1's complement of Y
- set carry-in bit $c_0 = 1$

$$X = 7 = 0111, Y = 3 = 0011$$

$$\begin{array}{rcl} X & : & - 0 1 1 1 \\ Z & : & - 1 1 0 0 \\ C & : & 1 1 1 1 1 \\ S & : & - 0 1 0 0 \end{array}$$

8 Bit Ripple-Carry Adder

Performing Subtraction



$$C_0 = 1$$

We have assumed that Y'_i

are available

Two Problems

- We have two separate networks for subtraction and addition. We need a way of performing a conditional processing operation on the input to a single n -bit adder.
- For each FA in the RCA, there are 3 gate delays between x_i or y_i and c_{i+1} and 2 between c_i and c_{i+1} . So the time it takes to complete the n -bit addition is $2(n - 1) + 3$. This is unacceptable scaling for a combinational network.

The first is easy to handle.

We want to produce a circuit that takes a single bit input, S , that indicates if a sum or difference is to be performed.

$$\begin{aligned} S = 0 &\rightarrow F = X + Y \\ S = 1 &\rightarrow F = X - Y \\ &= X + Y' + 1 \end{aligned}$$

The subtraction is performed via the twos-complement technique as indicated.

The combined circuit is called an **ADDER/SUBTRACTOR**.

The problem is the conditional generation of y'_i bits.

The RCA uses XOR, AND, OR gates. The AND, OR gates do not have the complement functionality (recall our discussion of complete gate sets). So the XOR gate is the best candidate.

Consider the following:

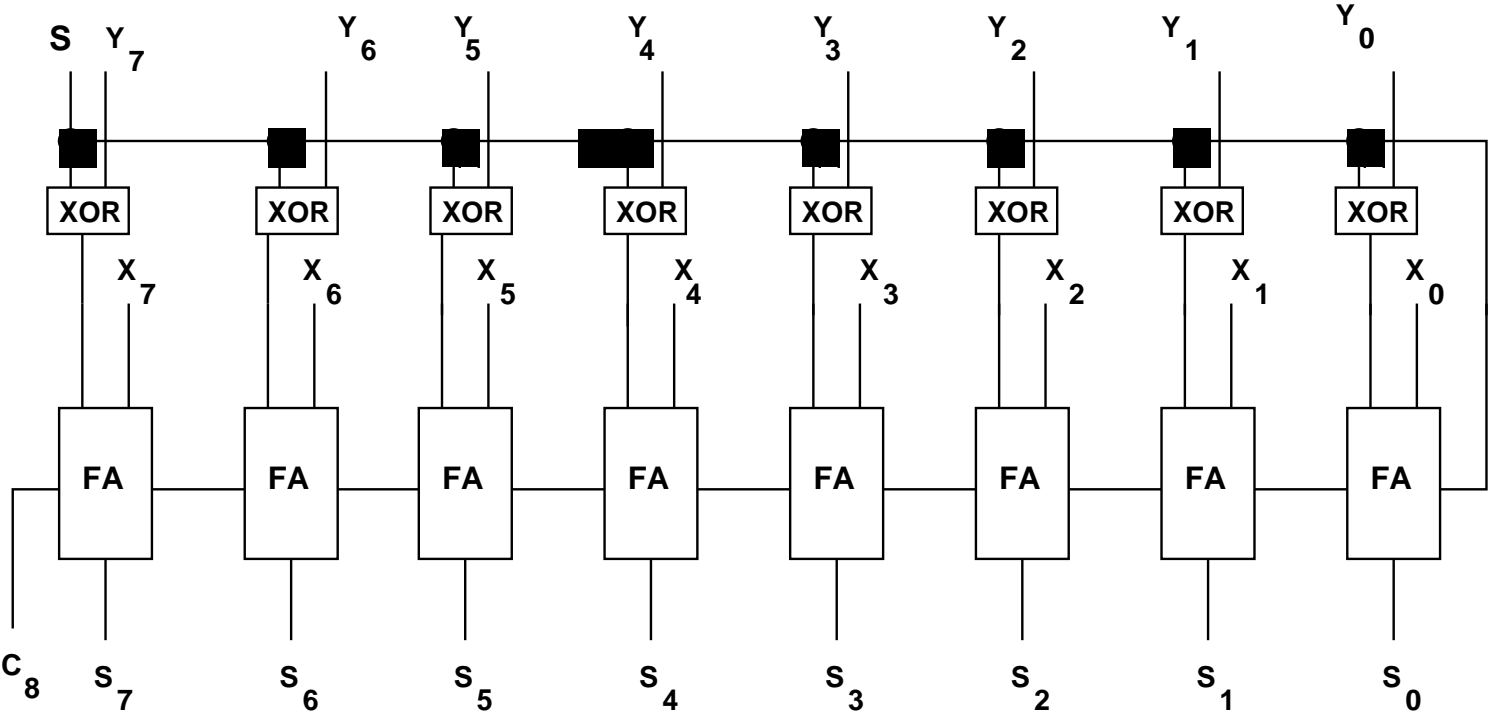
$$\begin{aligned} a \oplus b &= ab' + a'b \\ S \oplus y_i &= Sy'_i + S'y_i \\ 0 \oplus y_i &= 0 \bullet y'_i + 1 \bullet y_i \\ &= y_i \\ 1 \oplus y_i &= 1 \bullet y'_i + 0 \bullet y_i \\ &= y'_i \end{aligned}$$

When $C_0 = S$ is also used we have exactly what is needed.

- When $S = 0$ the output of the i -th XOR gate is y_i and the carry-in $C_0 = S = 0$. So $F = X + Y$ is computed.
- When $S = 1$ the output of the i -th XOR gate is y'_i and the carry-in $C_0 = S = 1$. So $F = X - Y$ is computed via the twos-complement algorithm.
- A single row of XOR gates doubles the functionality of a standard RCA.

The critical path of the circuit is still $O(n)$ however.

8 bit Ripple-carry Adder/Subtractor



$C_0 = S$

In order to reduce the delay time we must consider the critical path formed by the ripple-carry computations.

The carry bits form a recurrence in their present form:

$$\begin{aligned}c_{i+1} &= x_i y_i + c_i (x_i \oplus y_i) \\ &= g_i + c_i p_i\end{aligned}$$

So even though the $g_i = x_i y_i$ and $p_i = x_i \oplus y_i$ can all be computed simultaneously at a delay cost of one gate, c_{i+1} is not computed until after c_i is available when the recurrence above is used.

Note however that this is a combinational circuit so given assignments to the switching variables (x_{n-1}, \dots, x_0) and (y_{n-1}, \dots, y_0) **all of the carries are uniquely determined**, i.e., there is a mapping $\mathcal{C} : B^{2n} \rightarrow B^{n+1}$ that associates the carry values $(c_n, c_{n-1}, \dots, c_0)$ with the values of the (x_{n-1}, \dots, x_0) and (y_{n-1}, \dots, y_0) . So if we knew the truth tables for each of the c_i we could synthesize a multiple output network to evaluate all carries in two gate levels assuming double-rail logic.

Due to fan-in and fan-out limitations however this is not usually done for more than 4 carries at a time.

Given c_i , $(x_{i+3}, x_{i+2}, x_{i+1}x_i)$ and $(y_{i+3}, y_{i+2}, y_{i+1}y_i)$ we want to compute $(c_{i+4}, c_{i+3}, c_{i+2}c_{i+1})$ and then $(s_{i+3}, s_{i+2}, s_{i+1}s_i)$.

c_{i+4} is passed on to the next set of 4 bits to be processed in this manner and $(c_{i+3}, c_{i+2}c_{i+1}, c_i)$ are used to compute $(s_{i+3}, s_{i+2}, s_{i+1}s_i)$.

There are two ways to do this:

- deduce the truth tables relating X and Y to the carry values and synthesize a multiple output network.
- exploit the fact that the carries are given by a recurrence and construct functions for $(c_{i+4}, c_{i+3}, c_{i+2}, c_{i+1})$ from it.

For the first approach, there are 9 independent switching variables that can be assigned to form any one of the $2^9 = 512$ binary input vectors. The 4 output bits must be determined for each of them. Then the MOPIs generated. This is extremely tedious when done by hand. It also yields a solution that potentially involves high-fan-in gates (up to 9)

The second approach, however, is trivial by hand and the resulting fan-ins and fan-outs are bounded by 4.

We shall therefore take the second approach.

Recall, $g_i = x_i y_i$ and $p_i = x_i \oplus y_i$ and

$$\begin{aligned}c_{i+1} &= x_i y_i + c_i (x_i \oplus y_i) \\ &= g_i + p_i c_i\end{aligned}$$

Therefore we have the original recurrence,

$$\begin{aligned}c_{i+1} &= g_i + p_i c_i \\ c_{i+2} &= g_{i+1} + p_{i+1} c_{i+1} \\ c_{i+3} &= g_{i+2} + p_{i+2} c_{i+2} \\ c_{i+4} &= g_{i+3} + p_{i+3} c_{i+3}\end{aligned}$$

Note however that the first equation gives c_{i+1} in terms of input information only. So suppose we substitute it into the expression for c_{i+2} to eliminate the need to wait for c_{i+1} .

$$\begin{aligned}c_{i+2} &= g_{i+1} + p_{i+1} c_{i+1} \\ &= g_{i+1} + p_{i+1} (g_i + p_i c_i) \\ &= g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i\end{aligned}$$

Both c_{i+1} and c_{i+2} can now be evaluated simultaneously based on input data only using two independent two-level AND/OR networks.

We now have

$$\begin{aligned}c_{i+1} &= g_i + p_i c_i \\c_{i+2} &= g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i \\c_{i+3} &= g_{i+2} + p_{i+2} c_{i+2} \\c_{i+4} &= g_{i+3} + p_{i+3} c_{i+3}\end{aligned}$$

Substitute the new expression for c_{i+2} into the expression for c_{i+3} to get a formula for c_{i+3} in terms of input data only.

$$\begin{aligned}c_{i+3} &= g_{i+2} + p_{i+2} c_{i+2} \\&= g_{i+2} + p_{i+2} (g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i) \\&= g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i\end{aligned}$$

We now have

$$\begin{aligned}
 c_{i+1} &= g_i + p_i c_i \\
 c_{i+2} &= g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i \\
 c_{i+3} &= g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i \\
 c_{i+4} &= g_{i+3} + p_{i+3} c_{i+3}
 \end{aligned}$$

Substitute the new expression for c_{i+3} into the expression for c_{i+4} to get a formula for c_{i+4} in terms of input data only.

$$\begin{aligned}
 c_{i+4} &= g_{i+3} + p_{i+3} c_{i+3} \\
 &= g_{i+3} \\
 &\quad + p_{i+3} (g_{i+2} + p_{i+2} g_{i+1} \\
 &\quad + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i) \\
 &= g_{i+3} \\
 &\quad + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} \\
 &\quad + p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} p_{i+1} p_i c_i
 \end{aligned}$$

Note this requires an OR gate with a fan-in of 5 and an AND gate with a fan-in of 5. If we are limited to 4 as we assumed we must add another level of logic.

$$\begin{aligned}
 c_{i+4} &= A + B \\
 A &= g_{i+3} + p_{i+3} g_{i+2} \\
 &\quad + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i \\
 B &= p_{i+3} p_{i+2} p_{i+1} p_i c_i
 \end{aligned}$$

The resulting adder is called a **carry-lookahead adder**. For an n -bit slice of this technique there are

- $\frac{n^2+n}{2} + 1$ AND gates
- $n + 1$ OR gates
- the maximum fan-in of a gate is n
- the maximum fan-out of a line is slightly less than n^2 .
- the gate delay between x_i or y_i and c_{i+n} is 5 : 1 for g_i and p_i , 2 for the AND/OR network derived above and then 2 from the splitting of c_{i+4} to keep the fan-in under n .
- This is too costly to go much beyond $n = 4$.
- The 8-bit CLA has 10 gate delays vs. 17 for the RCA. (See the text for more detailed delay estimates that differentiate between gate types).
- The idea can be repeated in a multilevel technique. This is discussed in the text.