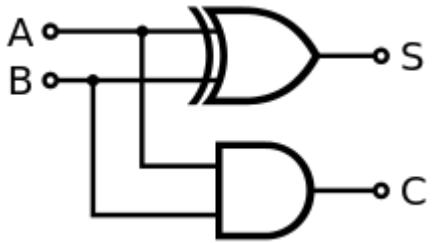
# Types of Adders

# Addition in Binary

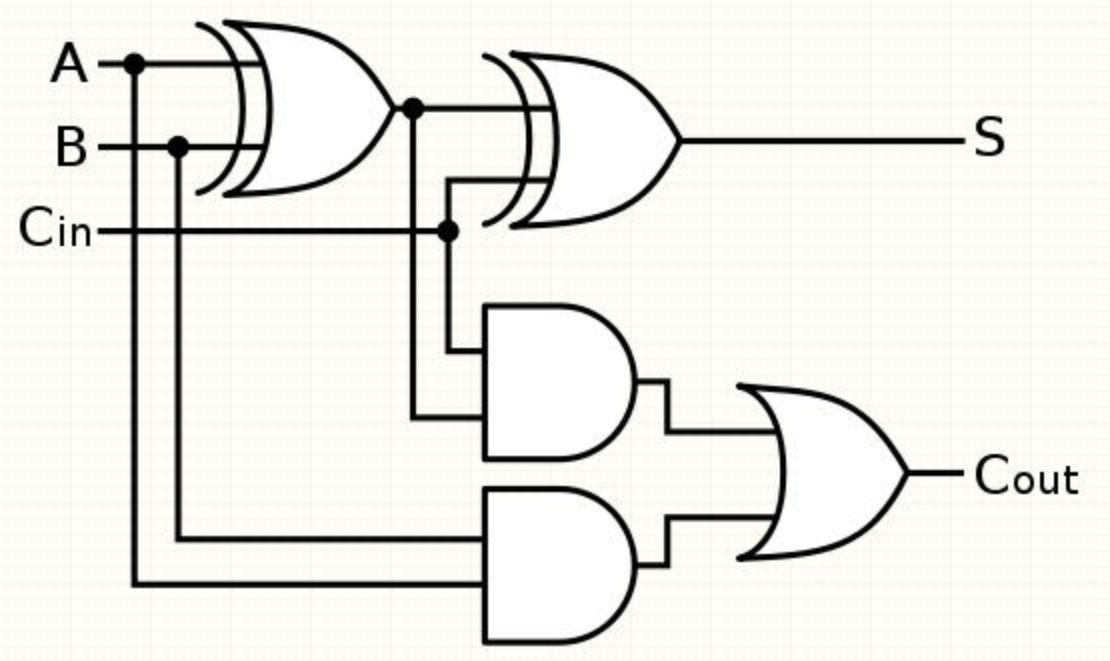
$$\begin{array}{r} {\scriptstyle 1\ \ 1\ \ \ \ } \\ 1011 \\ +\ 0110 \\ \hline 10001 \end{array}$$

| A | B | S | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

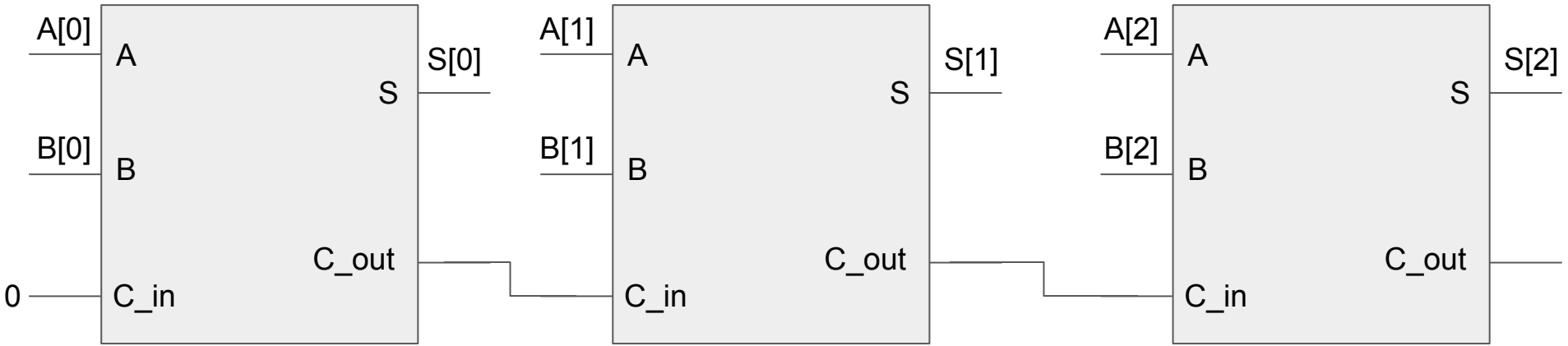


Same as the half adder table

| A | B | Carry In | S | Carry Out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |


A
B
Cin
S
Cout

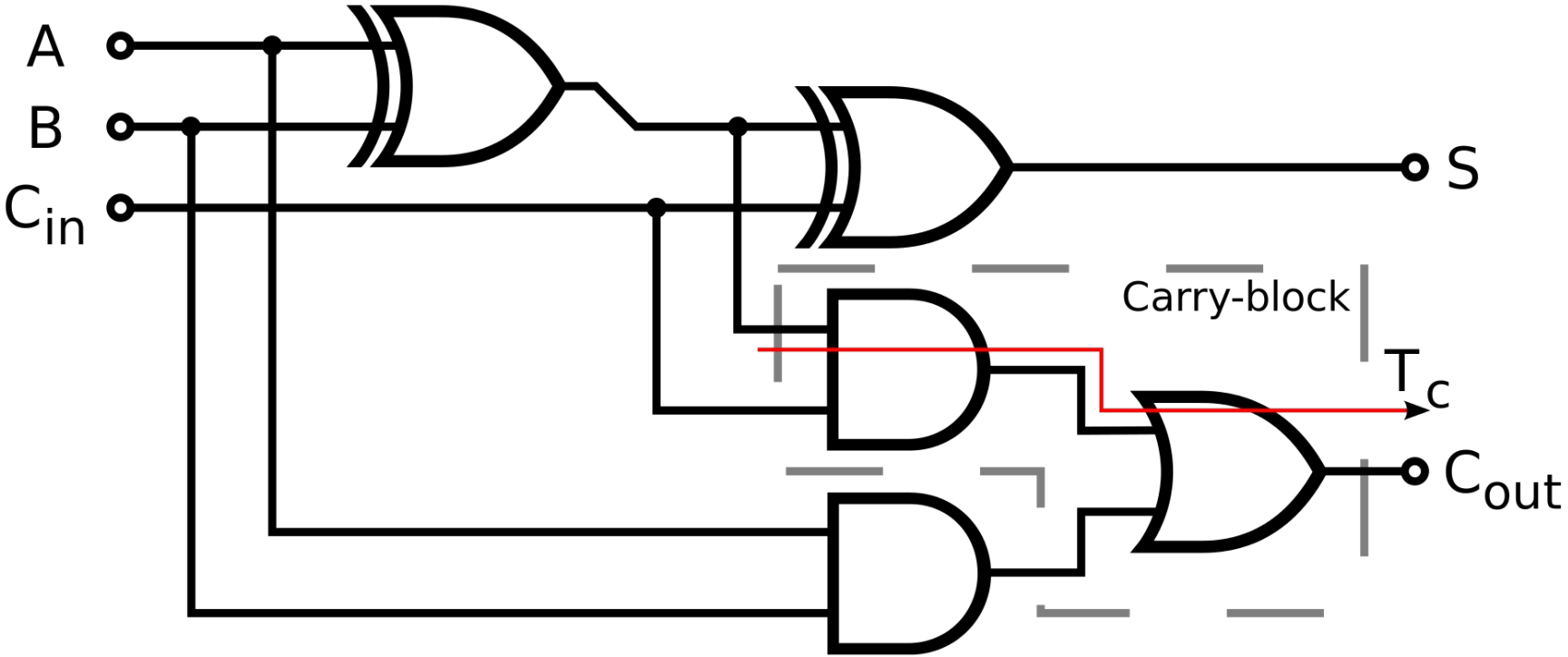
# Ripple Carry Adder



# And there we go!

We can add binary numbers of *N* bits.

# But...



# Another thing you could do: Carry Save Addition

First, get the sum:

$$\begin{array}{r} 1011 \\ +\ 0110 \\ \hline 1101 \end{array}$$

Next, get carries:

$$\begin{array}{r} 1011 \\ +\ 0110 \\ \hline 00100 \end{array}$$

Finally, add the two:

$$\begin{array}{r} 1101 \\ +\ 00100 \\ \hline 10001 \end{array}$$

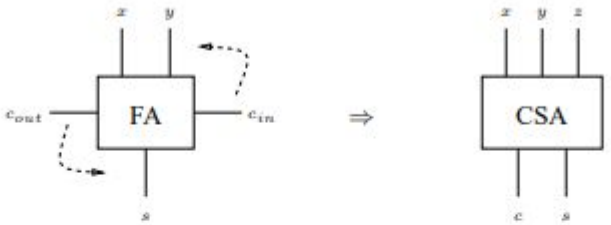
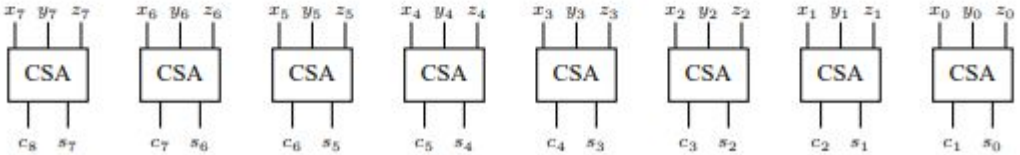



Figure 1: The carry save adder block is the same circuit as the full adder.



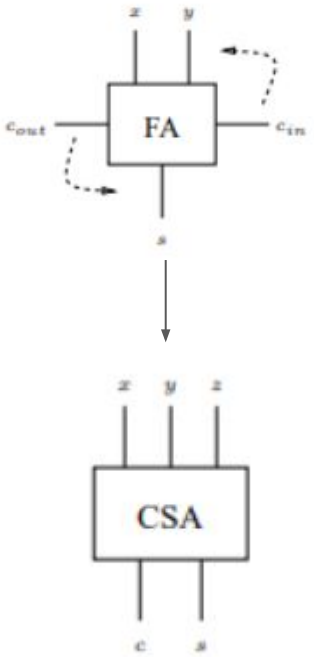

Loh, *Carry Save Addition,* https://people.clarkson.edu/~linh/algorithms/carry%20save%20addition.pdf

# But wait, why?

Say we want to add three numbers together.

With a ripple carry adder, that means two carry chains, which could potentially take a long time.

With a carry save, that carry_in bit is open, and so we can actually just add a third number.

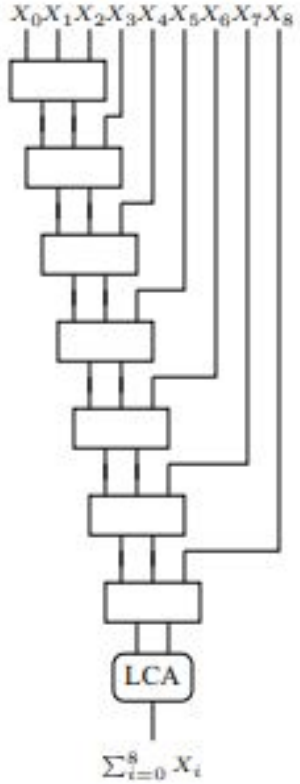


# Wallace Trees

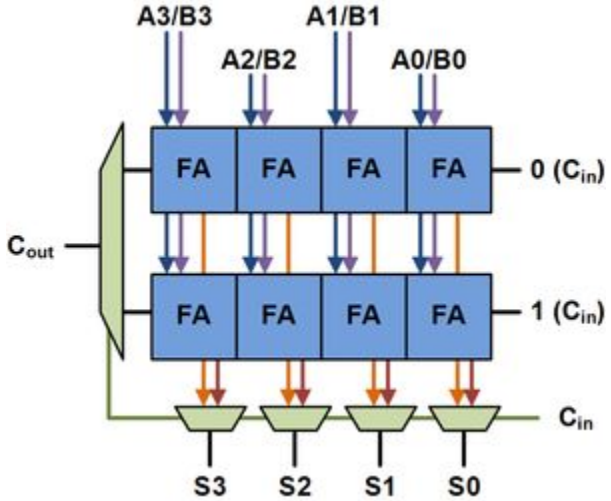
Because each carry save adder takes in three numbers and outputs two, we can chain them together to add a lot of numbers.

Then at the end, we only have to go through one carry chain instead of a lot.

This works best for really large numbers (like 512 to 2048 bits), where the ripple carry could take a long time.



# Another idea: Carry Select



# Clever things to do instead

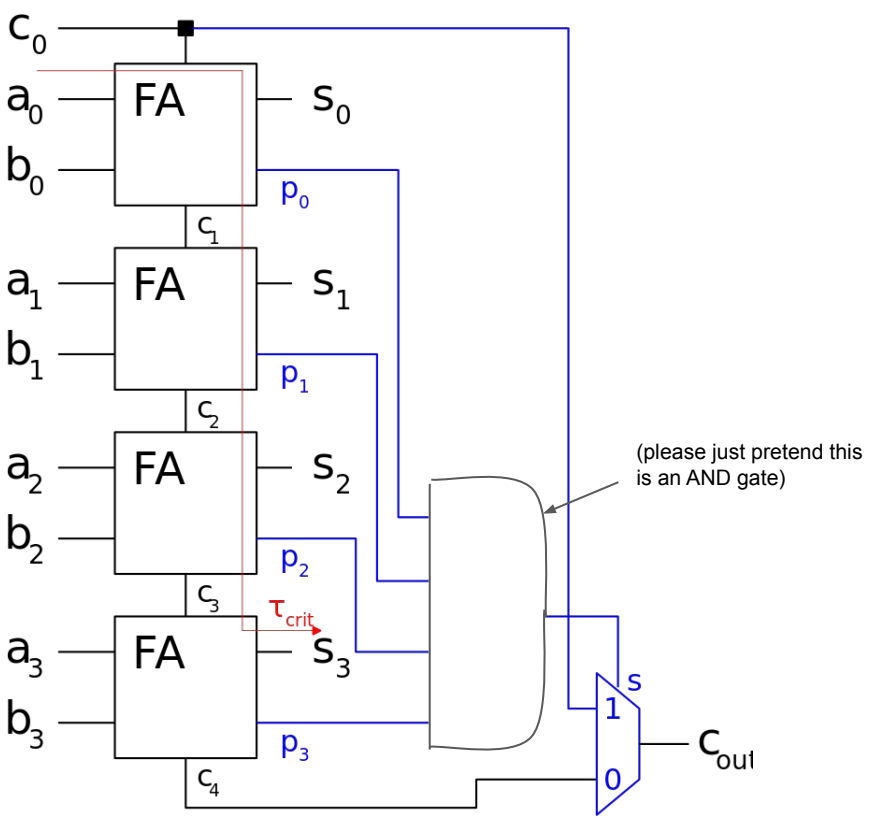
The general idea is that we can very easily tell whether a single bit will propagate, generate, or stop a carry bit.

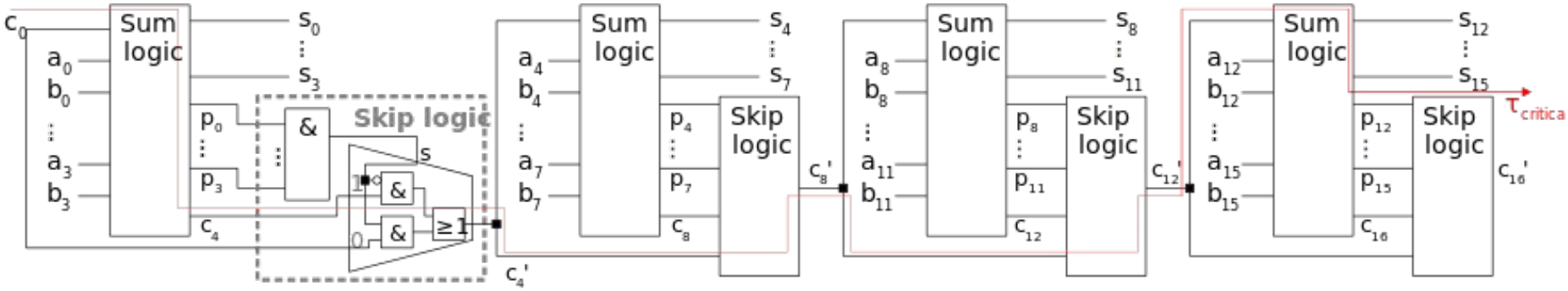
This information can be collected and used on groups of bits instead of one at a time.

| A | B | Carry In | Propagate | Generate |
|---|---|---|---|---|
| 0 | 0 | ? | 0 | 0 |
| 0 | 1 | ? | 1 | 0 |
| 1 | 0 | ? | 1 | 0 |
| 1 | 1 | ? | 0 | 1 |

# Carry Skip Adders

- Collect propagate bits in an AND gate
- Use the output as the select in a multiplexer, to choose either:
  - The last carry out bit, as normal, or
  - The carry in, bypassing the entire carry chain
- Very easy, improves worst case scenario






Now, instead of only shortening the time for the longest path, any time one of these four bit chunks are all 1’s, the operation is sped up.

# Carry Look Ahead

Ok, surely we can be smarter about this. Let's re-define what the outputs are, in terms of propagate, generate, and carry:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

# What to do with this

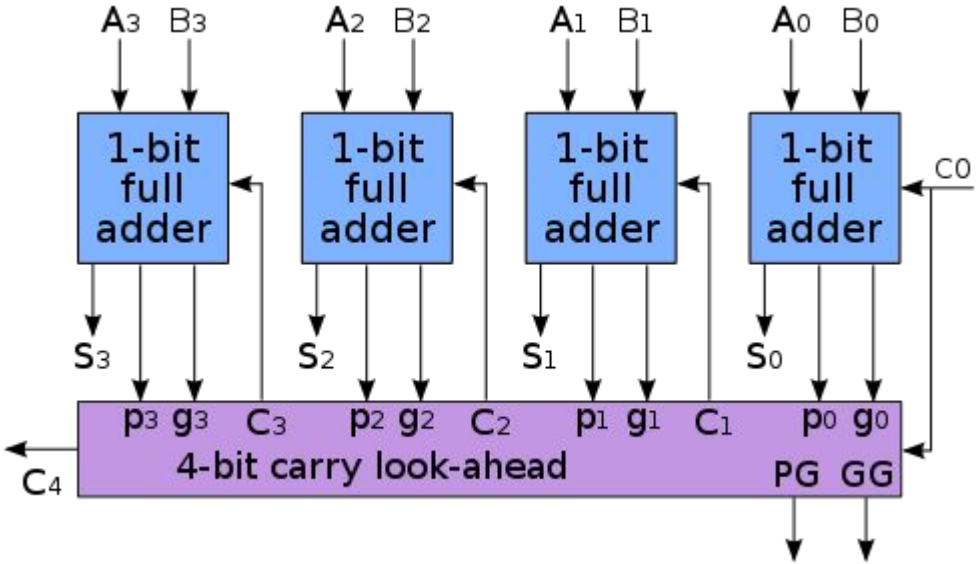
We know $C_0$ - that's our carry in. So we can use our equation, $C_{i+1} = G_i + P_i C_i$ to do this:

$$C_1 = G_0 + P_0 \cdot C_0,$$

$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1,$$

$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2,$$

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3.$$




Typically, we use several of these in a row, speeding up the addition even more.