Arithmetic circuits

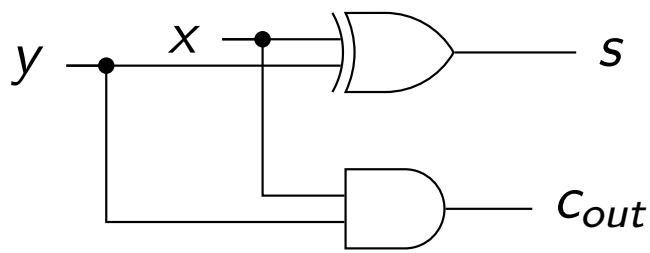
- Since we can represent numbers in base-2, we can design various types of circuits to perform arithmetic operations.
- We will consider a few different circuits.

Half adder circuits

- Design a circuit that takes two bits x and y as input and adds $(x + y)$ them together. The circuit should produce two outputs — a sum bit s and a carry out bit c_{out}.
- The resulting circuit is known as a **half adder**.
- Truth table(s) for half adder:

x	y	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- From the truth tables, we see that $s = x \oplus y$ and $c_{out} = xy$.
- Circuit...



Full adder

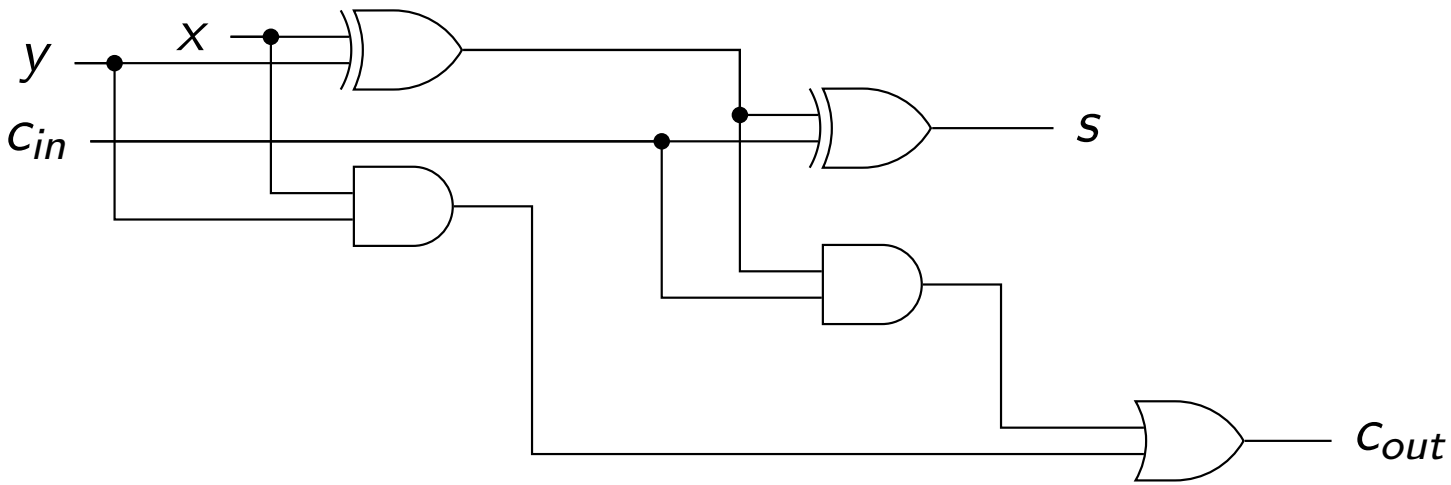
- If we want to add numbers (which consist of multiple bits), then we need a circuit that "supports" a carry in c_{in} which would come from the addition of previous bits.
- The resulting circuit is a **full adder**.
- Truth table for full adder:

x	y	c_{in}	s	c_{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

- From the truth tables, we see that $s = x \oplus y \oplus c_{in}$ and $c_{out} = xy + c_{in}(x \oplus y)$.

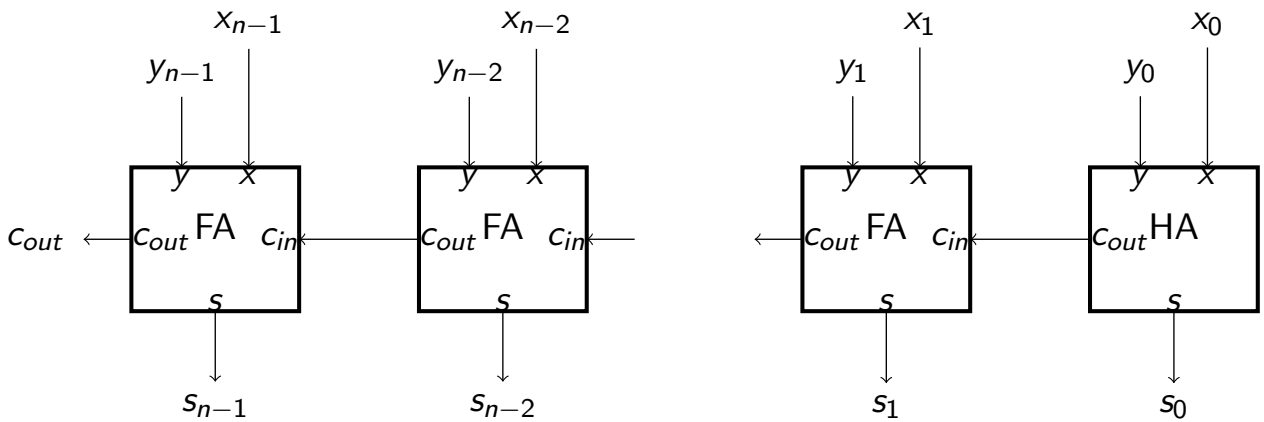
Full adder

- We can see that a full adder can be made using half adders.



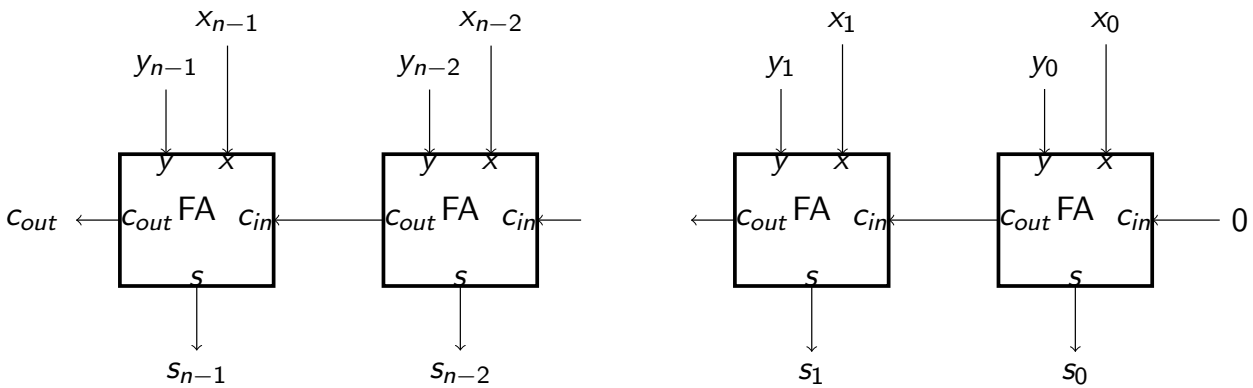
Ripple adder

- Say we want to add two n-bit numbers $x = (x_{n-1}, x_{n-2}, \cdots, x_1 x_0)$ and $y = (y_{n-1}, y_{n-2}, \cdots, y_1 y_0)$.
- We can accomplish this task using one half adder and $n-1$ full adders.



Ripple adder

- Can use only full adders if we force the carry in of the least significant bit to be 0.



Performance of the ripple adder

- Recall that, after a logic gate's input change, the gate output takes a bit of time to change due to delay inside of the gate.
- We can ask ourselves how long must we wait for the output of a ripple adder to be correct (i.e., outputs stop changing).
- To determine how long we must wait, we need to find the *longest combinational path* from any input to any output.
- For the n-bit ripple adder, let us assume that the inputs are all present at time 0. We can see that bit i cannot correctly compute its sum and carry out until bit $i-1$ has correctly computed *its carry out*. Further, note that the carry out from the i-bit depends on all the inputs from i down to 0.
- Therefore, the longest potential path through the ripple adder is from either a_0 or b_0 to the ultimate carry cout c_{out} from bit $n-1$.

Performance of the ripple adder

- Let's assume that we are using the ripple adder consisting of only full adders.
  - For the 0-th bit, from either a_0 or b_0 to the carry out in the worst case signals must propagate through 1 **XOR**, 1 **AND** and 1 **OR**.
  - For the i-th bit, from the carry in to the carry out, in the worst case signals must propagate through 1 **AND** and 1 **OR**.
- Therefore if we measure delay in terms of the number of gates through which signals must propagate and cause potential changes in values, the worst possible delay of the circuit is

$$(1\textbf{XOR} + 1\textbf{AND} + 1\textbf{OR}) + (n-1) * (1\textbf{AND} + 1\textbf{OR})$$

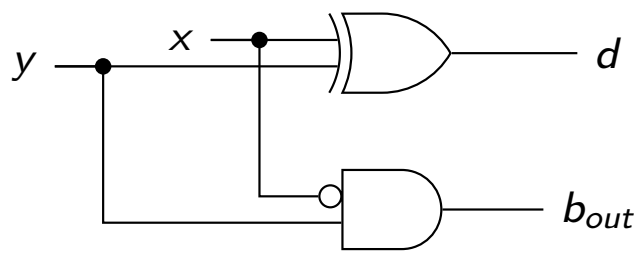
- If the delay of all gates is the same, then the delay is a total of $2n + 1$ gate delays.
- The performance of the ripple adder is extremely bad for large n due to the need for values to *ripple* down the *carry chain*.

Half subtractor circuits

- Design a circuit that takes two bits x and y as input and subtracts $(x - y)$. The circuit should produce two outputs — a difference bit d and a borrow bit b_{out}.
- The b_{out} is 1 when the circuit would *require a borrow*.
- The resulting circuit is known as a **half subtractor**.
- Truth table(s) for half subtractor:

x	y	d	b_{out}	
0	0	0	0	
0	1	1	1	$\leftarrow$ we would require a borrow
1	0	1	0	
1	1	0	0	

- From the truth tables, we see that $d = x \oplus y$ and $b_{out} = \overline{x}y$.
- Circuit...



- Note the similarity of this circuit to the half adder circuit.

Full subtractor

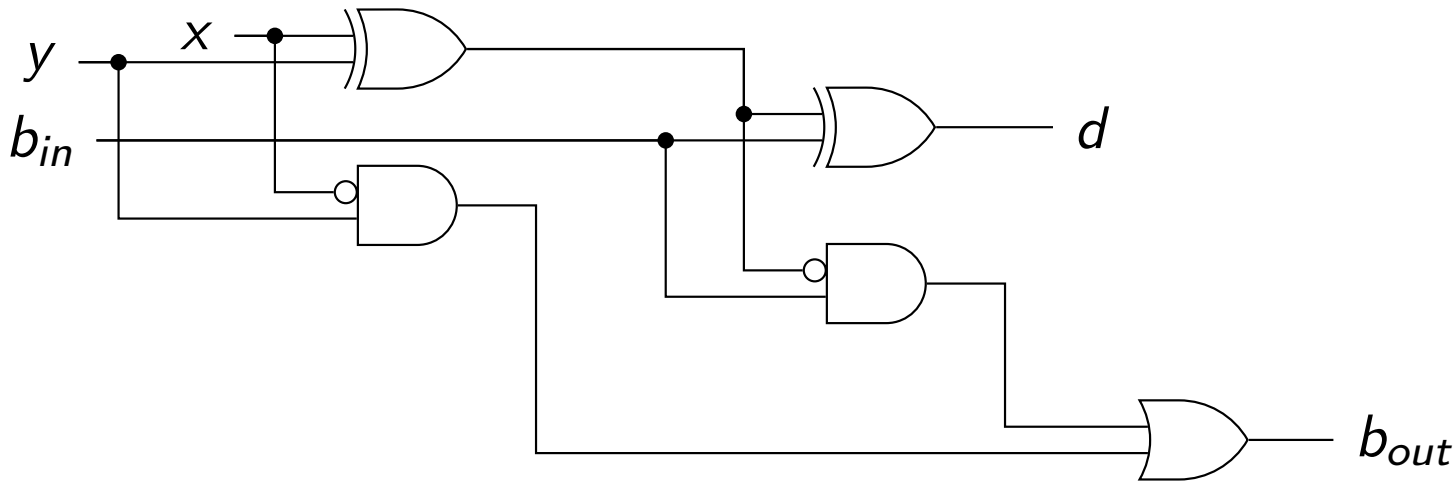
- If we want to subtract numbers (which consist of multiple bits), then we need a circuit that "supports" a borrow in b_{in} from the next higher bits in the subtraction.
- The b_{out} is 1 when the circuit would *require a borrow*.
- When b_{in} is 1 if means the circuit would *supply a borrow*.
- The resulting circuit is a **full subtractor**.
- Truth table for full subtractor:

x	y	b_{in}	d	b_{out}
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

- From the truth tables, we see that $d = x \oplus y \oplus b_{in}$ and $b_{out} = \overline{x}y + b_{in}\overline{(x \oplus y)}$.
- Note that there is similarity with a full adder circuit.

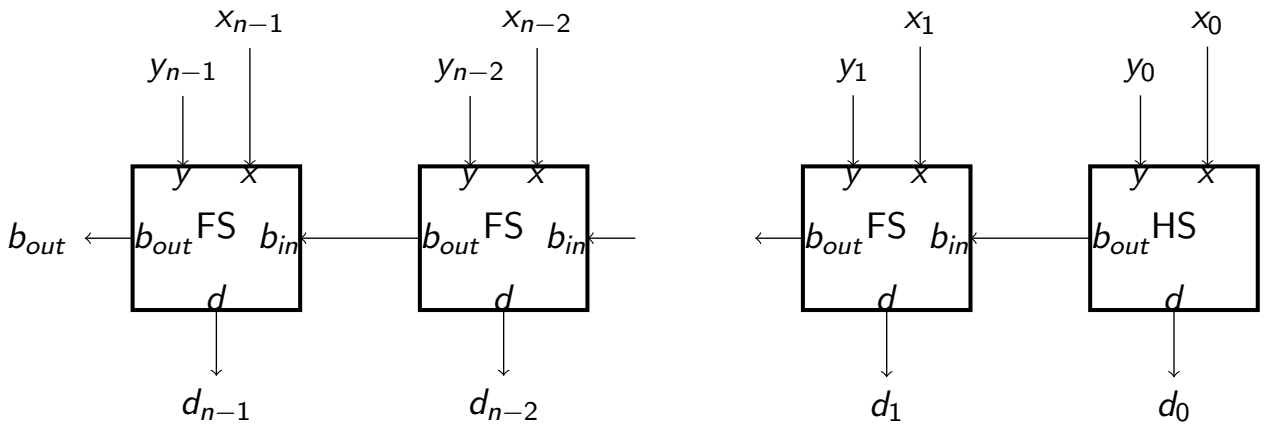
Full subtractor

- Schematic for a full subtractor circuit:



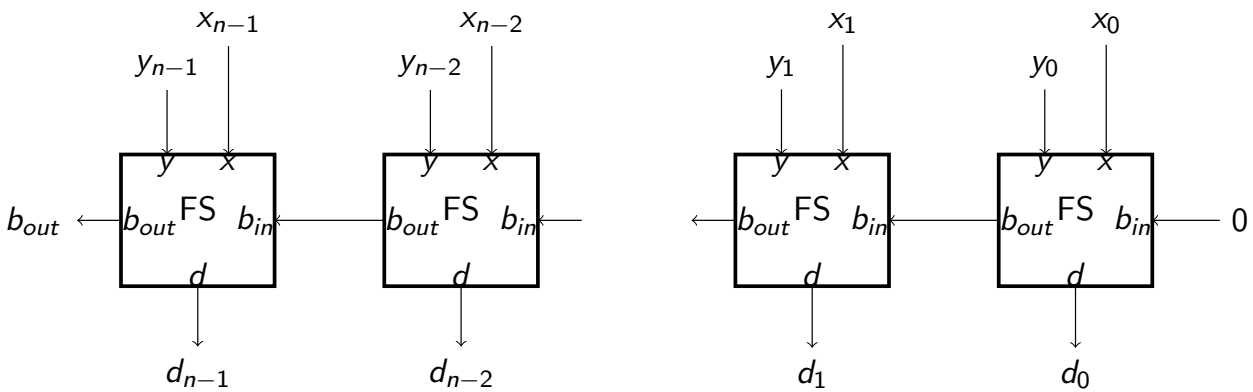
Ripple subtractor

- Say we want to subtract two n-bit numbers $x = (x_{n-1}, x_{n-2}, \cdots, x_1 x_0)$ and $y = (y_{n-1}, y_{n-2}, \cdots, y_1 y_0)$ and get $x - y$.
- We can accomplish this task using one half subtractor and $n - 1$ full subtractors.



Ripple subtractor

- Can use only full subtractors if we force the borrow in of the least significant bit to be 0.

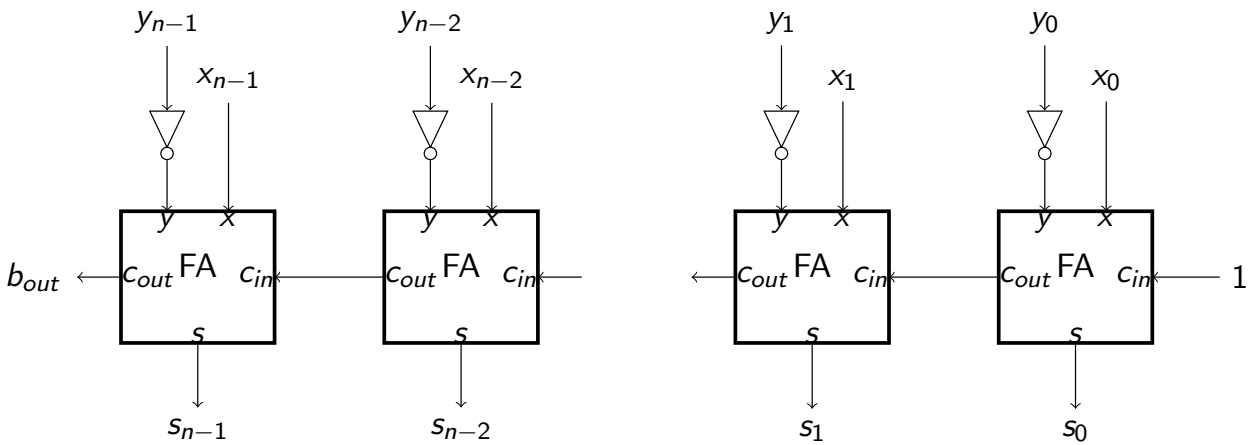


Performance of the ripple subtractor

- Similar to the ripple adder. Need to follow down the *borrow chain* from the least significant inputs x_0 or y_0.

Subtraction using addition circuit

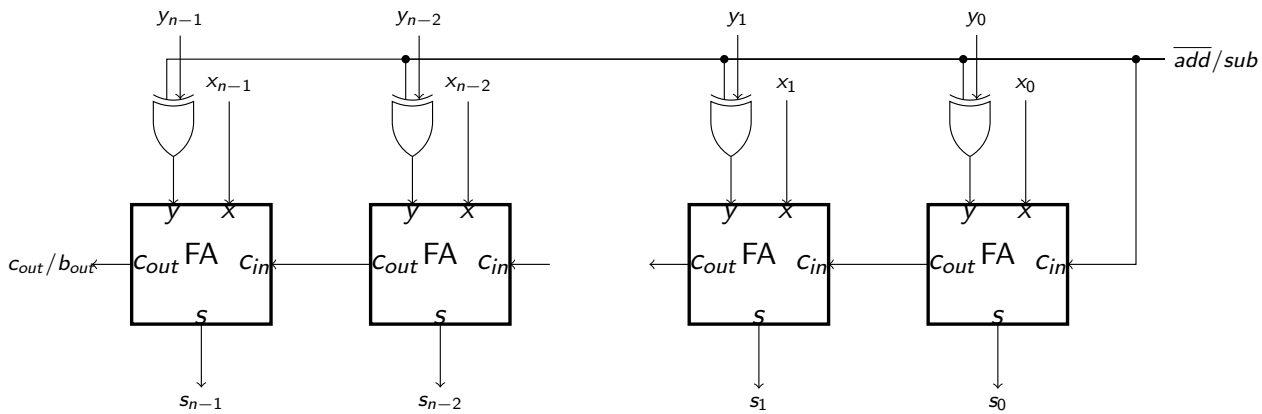
- Silly to have a different circuit for subtraction.
- Recall that subtraction is equivalent to adding the 2s complement of the subtrahend to the minuend.
- Further, taking the 2s complement of a binary number is equivalent to flipping the bits and adding 1.
- Consider this circuit instead for subtraction:



- Note outputs are still labelled as "sum" since they are outputs from adder circuits, but this "sum" is actually subtraction.

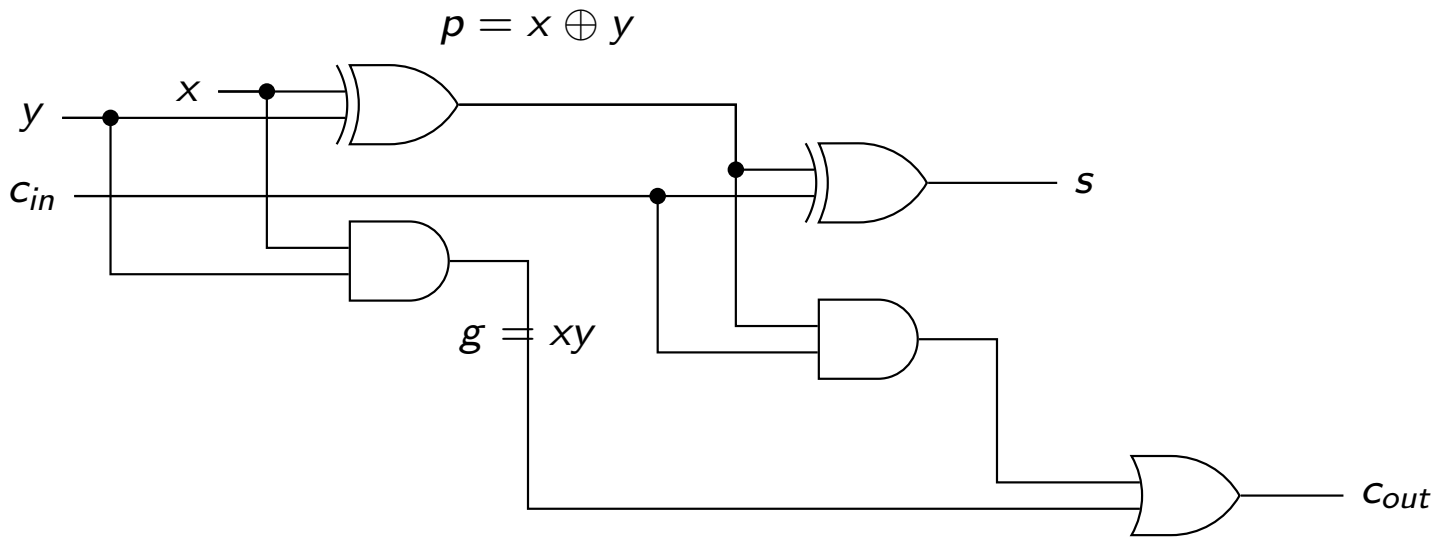
Addition and subtraction together

- Can combine the adder and subtractor together by introducing a *control line* — $\overline{add}/sub$.
- When $\overline{add}/sub = 0$, the circuit adds. When $\overline{add}/sub = 1$, the circuit subtracts.
- Note the creative use of **XOR** to perform inversion when doing subtraction!



Carry lookahead adders (CLA)

- Can make adders faster using *carry lookahead* circuitry.
- Let $p = x \oplus y$ be the *propagate* signal and let $g = xy$ be the *generate* signal.



- We can then write the carry out in terms of p and g as $c = g + c_{in}p$.

Carry lookahead adders (CLA)

- The ripple adder is slow because high bits must "wait" for carries to be generated. Note that the carries are produced as signals propagate through many "levels of gates".
- To speed up computation of the carries, we can first generate all the p and g for all bits (p and g only depend on x and y).
- Then, we can compute all carries at the *same time* by "collapsing" the multiple levels of logic; e.g.,

$$\begin{aligned}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 c_1 \\
&= g_1 + p_1(g_0 + p_0 c_0) \\
&= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
c_3 &= g_2 + p_2 c_2 \\
&= g_1 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0) \\
&= g_1 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
\ldots &= \ldots
\end{aligned}$$

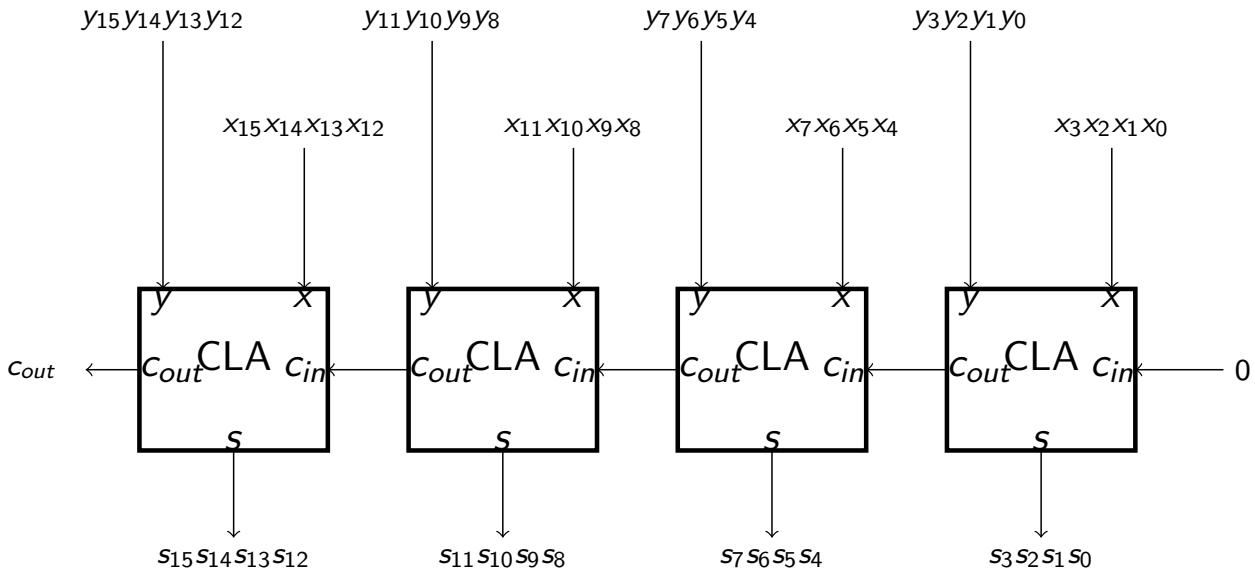
- Note that in removing the parentheses "(" and ")" we are, in effect, removing levels of gates from the equations and collapsing the carry signals back to 2-level SOPs!

Performance of carry lookahead adders (CLA)

- Sufficient to consider how fast the carry signals can be generated.
- Note that all p and g are generated from x and y after 1 gate delay;
- **All** carry signals are generated via SOP expressions 2 gate delays after the p and g;
- Consequently, all carries are available at the same time after 3 gate delays.
  - Regardless of the number of bits n that we are adding, the delay of a full carry lookahead adder is fixed.
- Of course, this is impractical in reality. **Why**? It's impractical because we see that the carry lookahead circuitry requires larger and larger gates (i.e., more inputs) as we more to higher order carry signals. Totally impractical.

Compromise between ripple adder and CLA

- Use a combination of CLA and ripple adder; build a reasonably sized CLA and then connect multiple CLA in a chain like a ripple adder.
- Example: 16-bit adder using 4, 4-bit CLA:



Array multipliers

- Multiplication of binary numbers works just like in decimal.
- Example multiplication of 2, 3-bit numbers $x = (x_2 x_1 x_0)$ and $y = (y_2 y_1 y_0)$ to get $x \times y = p$ where $p = (p_5 p_4 p_3 p_2 p_1 p_0)$.
- Note that multiplication requires twice the number of bits for the result.

Array multipliers

- Details worked out...

			x_2	x_1	x_0	← multiplicand
		$\times$	y_2	y_1	y_0	← multiplier
			y_0x_2	y_0x_1	y_0x_0	← partial product
	$+$	y_1x_2	y_1x_1	y_1x_0		
	pp_4^1	pp_3^1	pp_2^1	pp_1^1	pp_0^1	← partial product
$+$	y_2x_2	y_2x_1	y_2x_0			
pp_5^2	pp_4^2	pp_3^2	pp_2^2	pp_1^2	pp_0^2	← partial product

Array multipliers

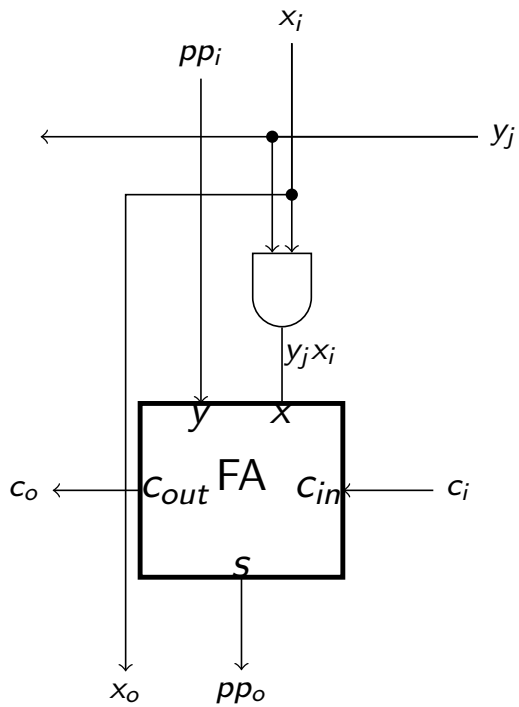
- Numerical example... 7×6 (or 111×110)...

$$
\begin{array}{cccccc}
 & & & 1 & 1 & 1 \\
 & & \times & 1 & 1 & 0 \\
\hline
 & & & 0 & 0 & 0 \\
 & + & 1 & 1 & 1 & \\
\hline
 & 0 & 1 & 1 & 1 & 0 \\
+ & 1 & 1 & 1 & & \\
\hline
1 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

- Result is 101010 or 42.
- The multiplication of 2 bits is simply **AND**.

Array multipliers

- Since bit multiplication is just **AND** we can make an array multiplier from **AND** gates and 1-bit full adders.
- Let's build this block first...



Array multipliers

- Using previous block, can build an array multipler; e.g., 3-bits...