Logic Design

Goal: to become literate in most common concepts and terminology of digital electronics

Important concepts:

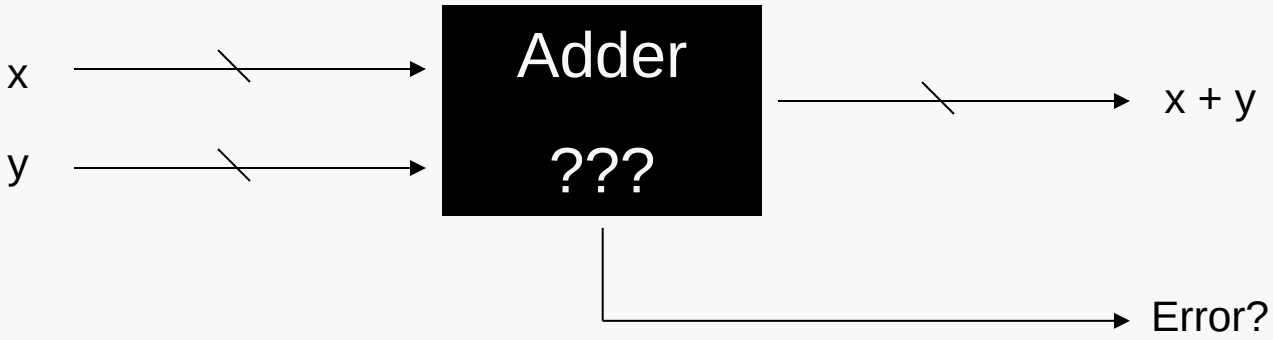
- use abstraction and composition to implement complicated functionality with very simple digital electronics
- keep things as simple, regular, and small as possible

Things we will not explore:

- physics
- chip fabrication
- layout
- tools for chip specification and design

Motivation

Consider the external view of addition:



What kind of circuitry would go into the "black box" adder to produce the correct results?

How would it be designed? What modular components might be used?

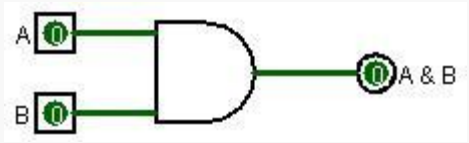
Basic Logic Gates

Fundamental building blocks of circuits; mirror the standard logical operations:

NOT gate

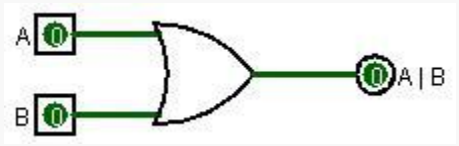
A	Out
0	1
1	0

AND gate



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

OR gate



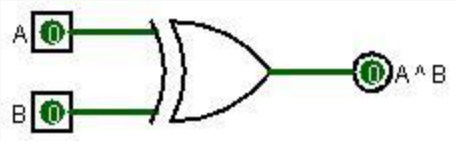
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Note the outputs of the AND and OR gates are commutative with respect to the inputs.

Multi-way versions of the AND and OR gates are commonly assumed in design.

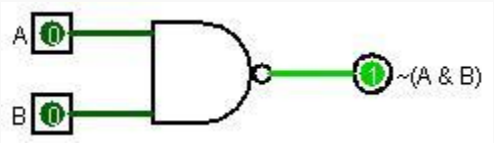
Additional Common Logic Gates

XOR gate



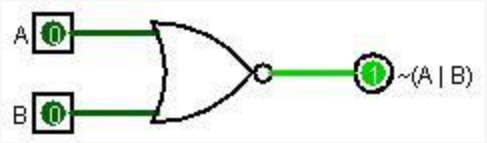
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

NAND gate



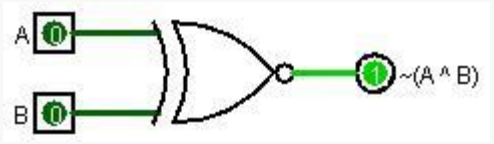
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NOR gate



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

XNOR gate



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	1

Combinational and Sequential Circuits

A *combinational circuit* is one with no "memory". That is, its output depends only upon the current state of its inputs, and not at all on the current state of the circuit itself.

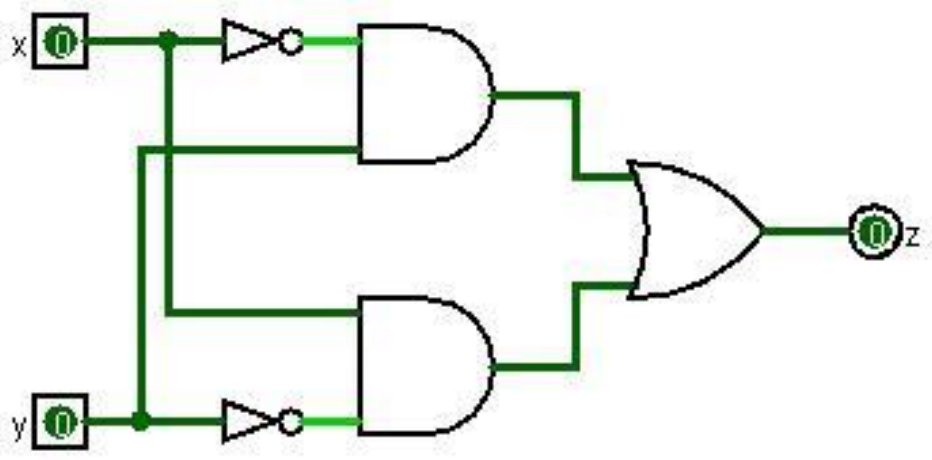
A *sequential circuit* is one whose output depends not only upon the current state of its inputs, but also on the current state of the circuit itself.

For now, we will consider only combinational circuits.

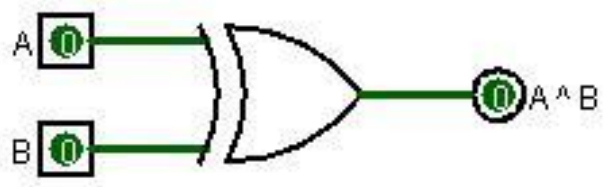
From Function to Combinational Circuit

Given a simple Boolean function, it is relatively easy to design a circuit composed of the basic logic gates to implement the function:

$$z: \quad x \cdot \bar{y} + \bar{x} \cdot y$$



This circuit implements the *exclusive or* (XOR) function, often represented as a single logic gate:



Sum-of-Products Form

A Boolean expression is said to be in *sum-of-products form* if it is expressed as a sum of terms, each of which is a product of variables and/or their complements:

$$a \cdot b + \overline{a} \cdot \overline{b}$$

It's relatively easy to see that every Boolean expression can be written in this form.

Why?

The summands in the sum-of-products form are called *minterms*.

- each minterm contains each of the variables, or its complement, exactly once
- each minterm is unique, and therefore so is the representation (aside from order)

Sum-of-Products Form

Given a truth table for a Boolean function, construction of the sum-of-products representation is trivial:

- for each row in which the function value is 1, form a product term involving all the variables, taking the variable if its value is 1 and the complement if the variable's value is 0
- take the sum of all such product terms

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$\bar{x} \cdot \bar{y} \cdot z$

$\bar{x} \cdot y \cdot \bar{z}$

$x \cdot \bar{y} \cdot \bar{z}$

$x \cdot y \cdot z$

$$F = \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot z$$

Equivalence

$F(x,y,z) = \bar{x}\cdot y\cdot z + x\cdot \bar{y}\cdot z + x\cdot y\cdot \bar{z} + x\cdot y\cdot z$	Given
$= \bar{x}\cdot y\cdot z + x\cdot \bar{y}\cdot z + x\cdot y\cdot \bar{z} + x\cdot y\cdot z + x\cdot y\cdot z + x\cdot y\cdot z$	Idempotence, twice
$= \left(\bar{x}\cdot y\cdot z + x\cdot y\cdot z\right) + \left(x\cdot \bar{y}\cdot z + x\cdot y\cdot z\right) + \left(x\cdot y\cdot \bar{z} + x\cdot y\cdot z\right)$	Commutativity, Associativity
$= \left(\bar{x} + x\right)\cdot y\cdot z + \left(\bar{y} + y\right)\cdot x\cdot z + \left(\bar{z} + z\right)\cdot x\cdot y$	Commutativity, Distributivity
$= 1\cdot y\cdot z + 1\cdot x\cdot z + 1\cdot x\cdot y$	Boundedness
$= x\cdot y + x\cdot z + y\cdot z$	Boundedness, Commutativity
$= G(x,y,z)$	

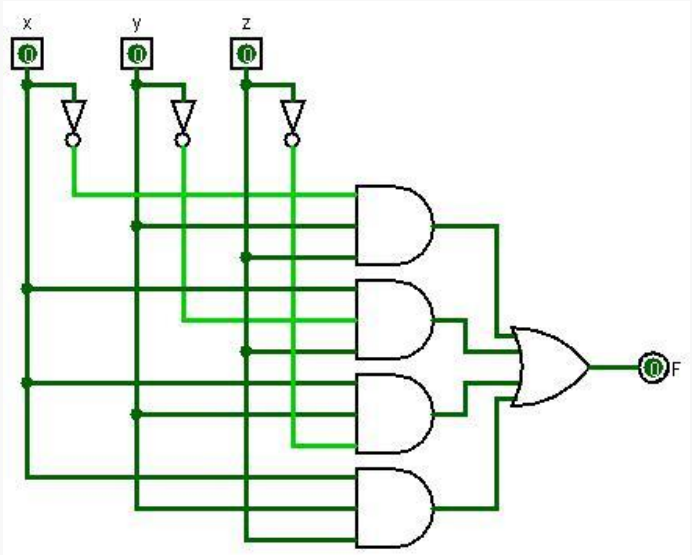
Efficiency of Expression

While the sum-of-products form is arguably natural, it is not necessarily the simplest way form, either in:

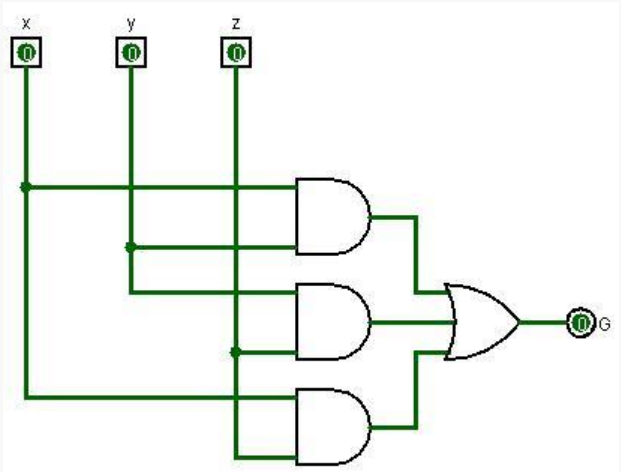
- number of gates (space)
- depth of circuit (time)

$$F(x,y,z) = \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z$$

$$+ x \cdot y \cdot \bar{z} + x \cdot y \cdot z$$



$$G(x,y,z) = x \cdot y + y \cdot z + x \cdot z$$



1-bit Half Adder

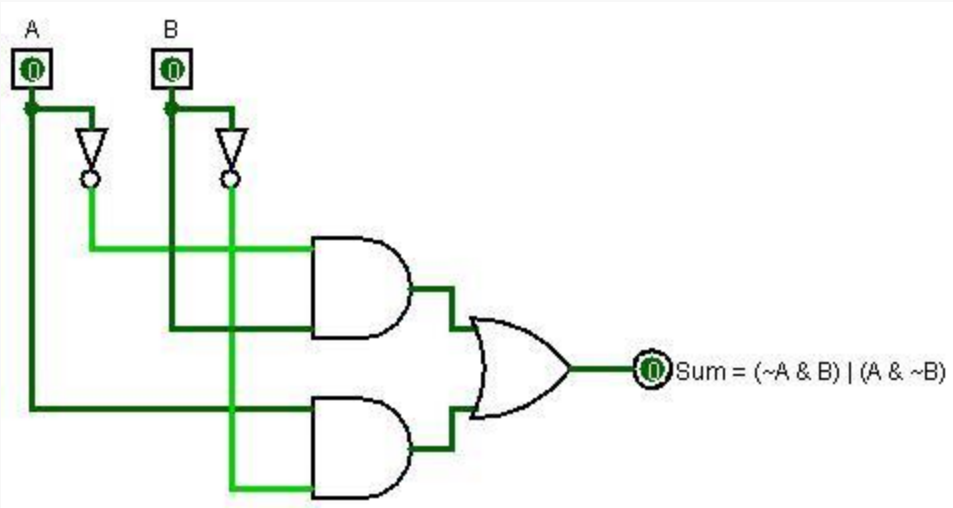
Let's make a 1-bit adder (*half adder*)… we can think of it as a Boolean function with two inputs and the following defining table:

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

Here's the resulting circuit.

It's equivalent to the XOR circuit seen earlier.

But… in the final row of the truth table above, we've ignored the fact that there's a carry-out bit.



Dealing with the Carry

The carry-out value from the 1-bit sum can also be expressed via a truth table.

However, the result won't be terribly useful unless we also take into account a carry-in.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

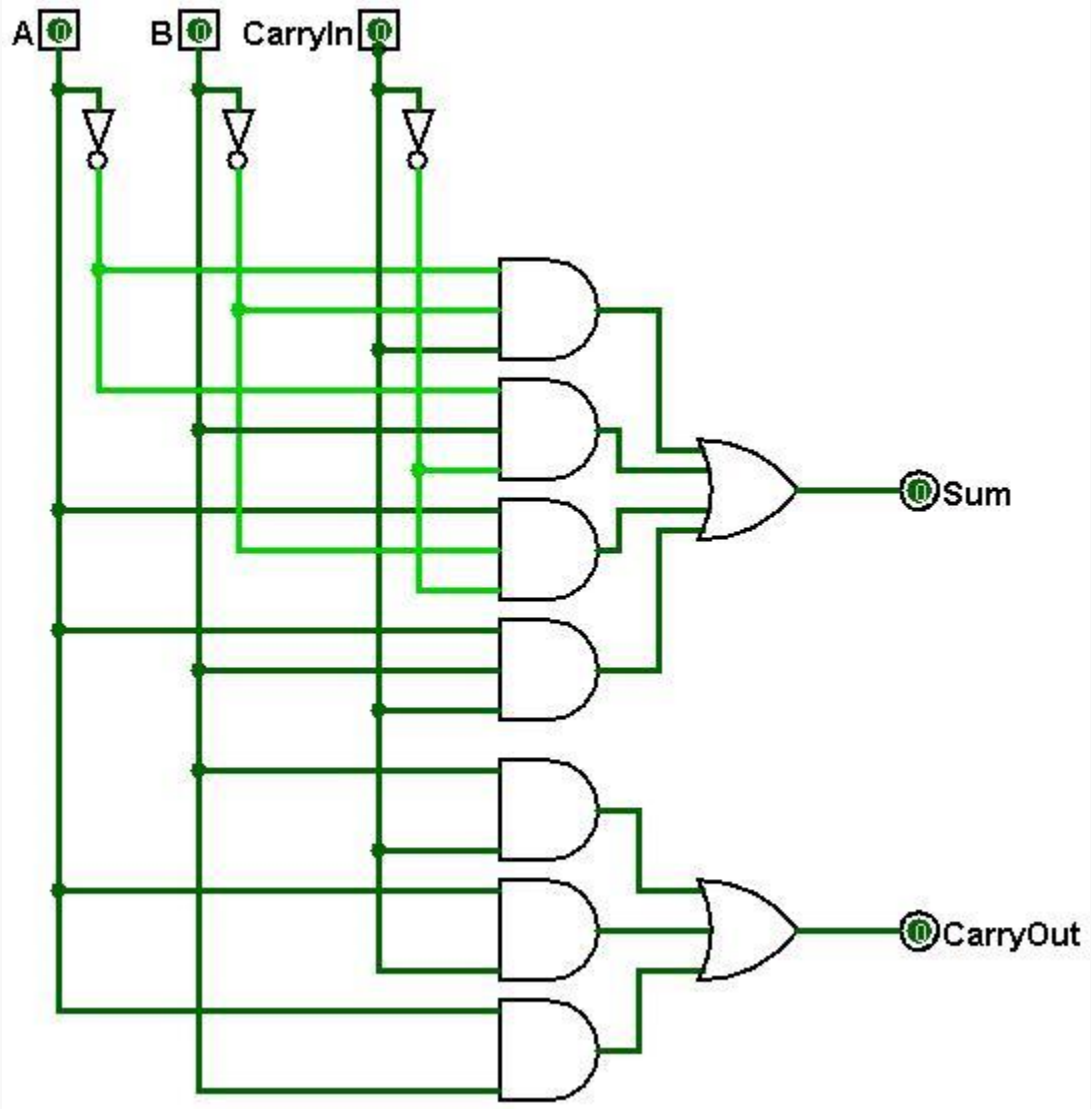
The resulting sum-of-products expressions are:

$$Sum = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in}$$

$$\begin{aligned} C_{out} &= \overline{A} \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} \\ &= \overline{A} \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot C_{in} + A \cdot B \cdot \left(\overline{C_{in}} + C_{in} \right) \\ &= \overline{A} \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot C_{in} + A \cdot B \\ &= A \cdot C_{in} + B \cdot C_{in} + A \cdot B \end{aligned}$$

1-bit Full Adder

The expressions for the sum and carry lead to the following unified implementation:



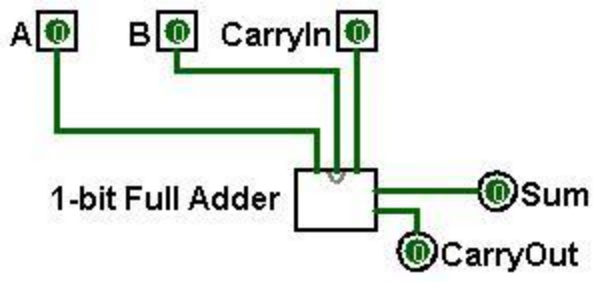
$$Sum = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}}$$

$$+ A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in}$$

$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

1-bit Full Adder as a Module

When building more complex circuits, it is useful to consider sub-circuits as individual, "black-box" modules. For example:

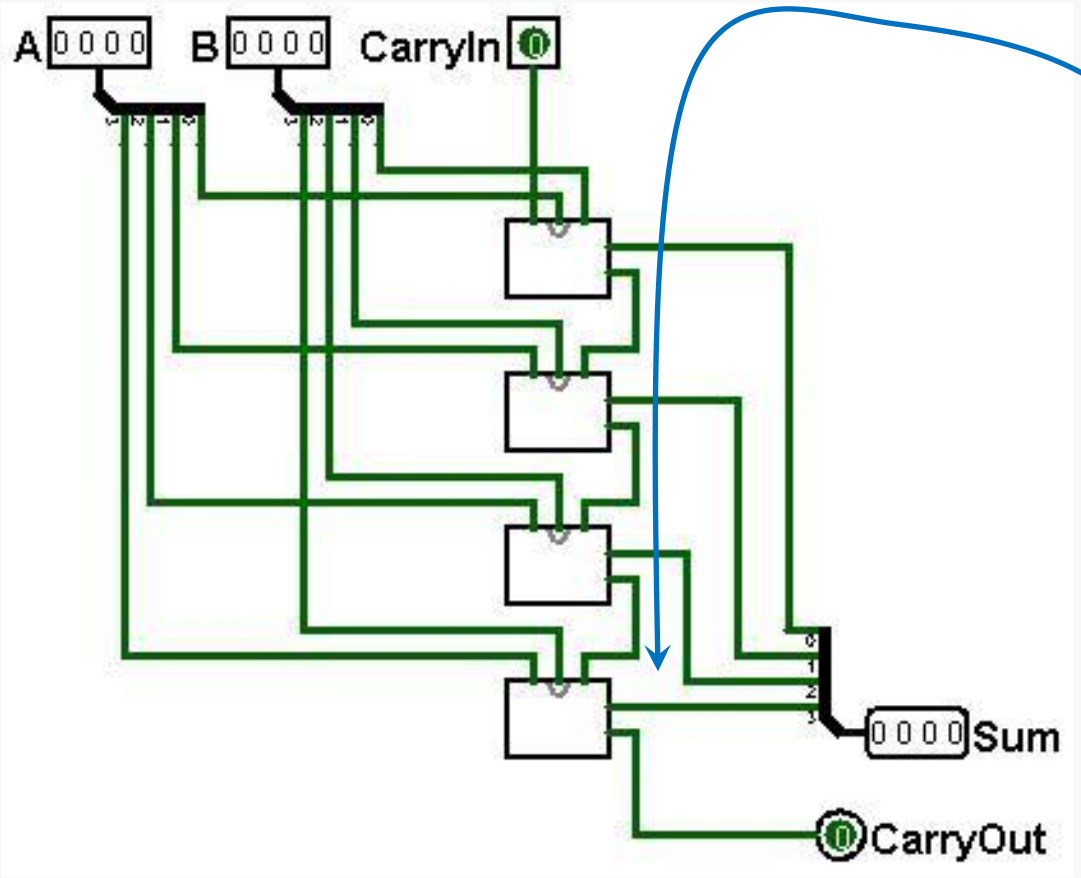


$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

$$Sum = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot \overline{B} \cdot \overline{C_{in}} + A \cdot B \cdot C_{in}$$

Chaining a 4-bit Adder

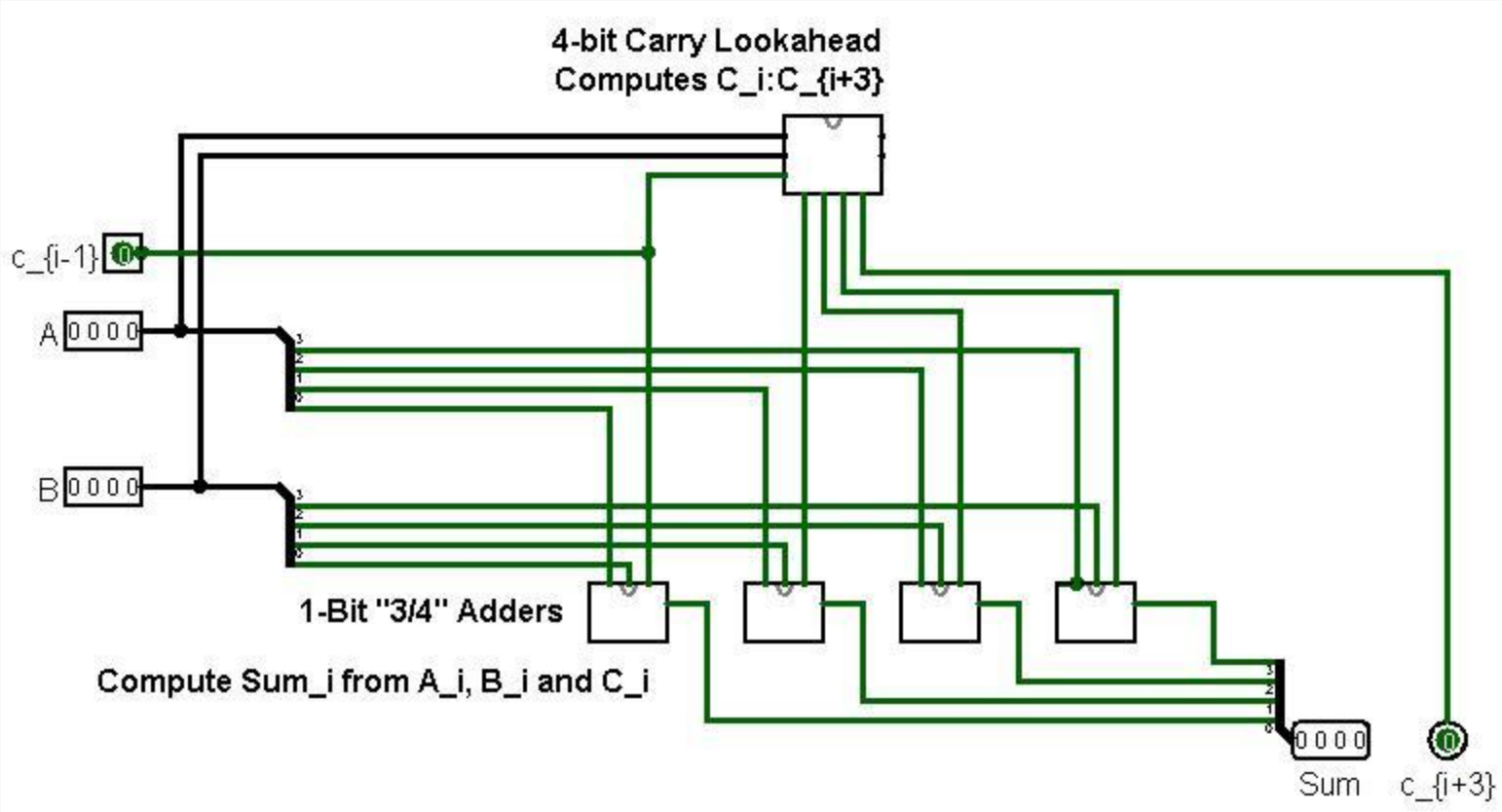
An 4-bit adder built by chaining 1-bit adders:



This has one serious shortcoming. The carry bits must *ripple* from top to bottom, creating a lag before the result will be obtained for the final sum bit and carry.

Carry-Lookahead Adder

Perhaps surprisingly, it's possible to compute all the carry bits before any sum bits are computed... and that leads to a faster adder design:



Why is this faster than the ripple-carry approach?

Latency

The answer lies in the concept of *gate latency*.

Each logic gate takes a certain amount of time (usually measured in picoseconds) to stabilize on the correct output... we call that the latency of the gate.

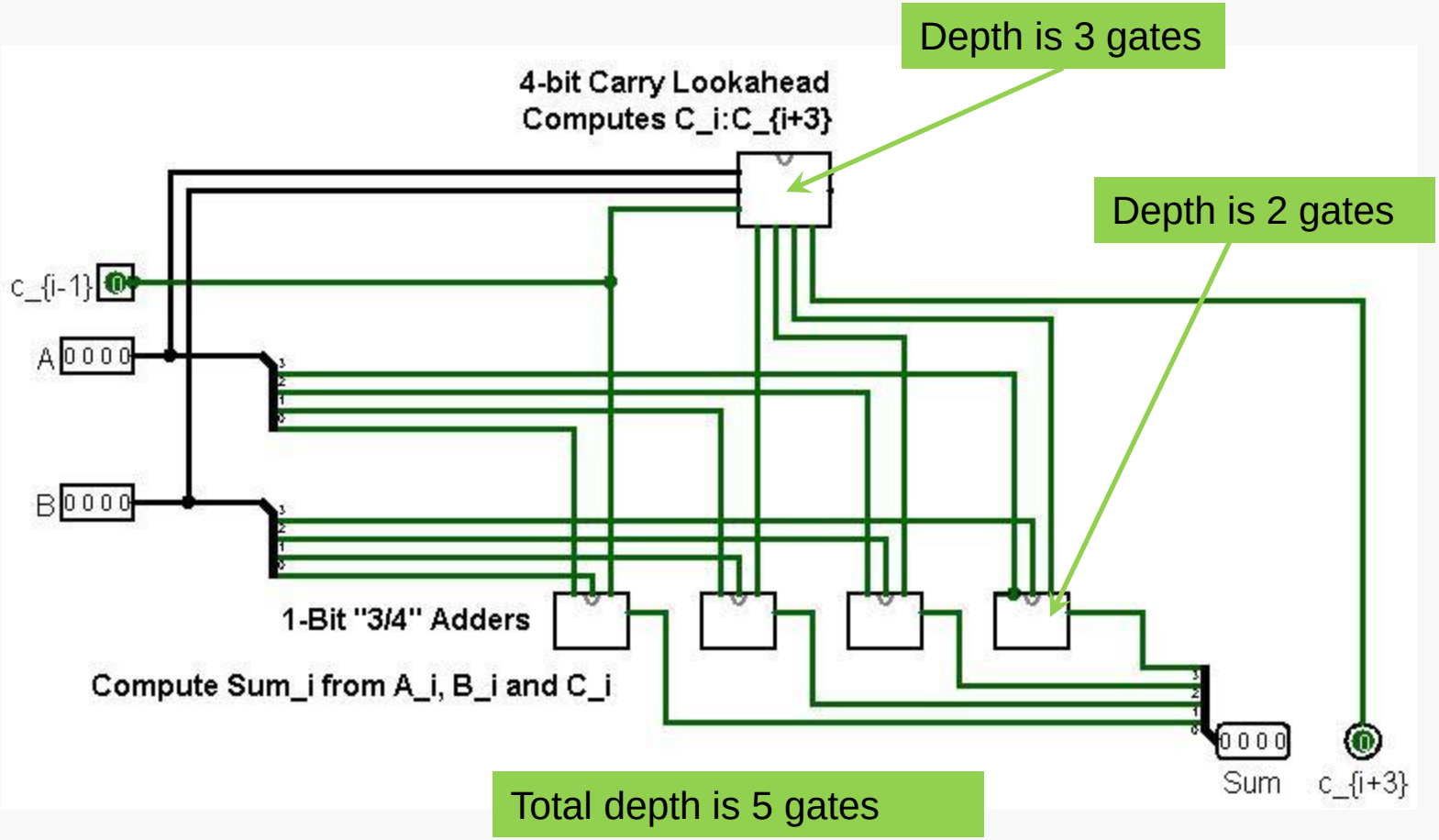
For simplicity, we'll assume in this course that all gates (except inverters) have the same latency, and that inverters are so fast they can be igored.

Then, the idea is that the latency of a circuit can be measured by the maximum number of gates a signal passes through within the circuit... called the *depth* of the circuit.

So, the 1-bit full adder we saw earlier has a depth of 2.

Carry-Lookahead Adder Latency

Without going into details:



How does that compare to the ripple-carry approach?

Ripple-carry Latency

A 4-bit ripple-carry design would have 4 1-bit full adders, and we've seen that each of those has a depth of 2 gates.

But those adders fire sequentially, so running one after the other would entail a total depth of 8 gates.

So, the ripple-carry design would be 1.6 times as "deep" and it's not unreasonable to say it would take about 1.6 times as long to compute the result.

Just how you'd implement the computation of those carry bits is an interesting question...

Carry-Lookahead Logic

Let's look at just how the carry bits depend on the summand bits:

```
  c4  c3   c2   c1   c0
      a3   a2   a1   a0
      b3   b2   b1   b0
      ------------------
      s3   s2   s1   s0
```

We will allow for a carry-in in the low-order position (c_0).

It's clear that $c_1 = 1$ if and only if at least two of the bits in the previous column are 1.

Since this relationship holds for every carry bit (except c_0), we have the following general Boolean equation for carry bits:

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

(Note that • represents AND and + represents OR.)

Carry-Lookahead Logic

Now, this relationship doesn't seem to help until we look at it a bit more deeply:

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i = a_i \cdot b_i + (a_i + b_i) \cdot c_i$$

If we define

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

then we get the following relationships:

$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

Now, we can calculate all of the g_i and p_i terms at once, from the bits of the two summands, and c_0 will be given, so we can compute c_1 and c_2 before we actually do the addition!

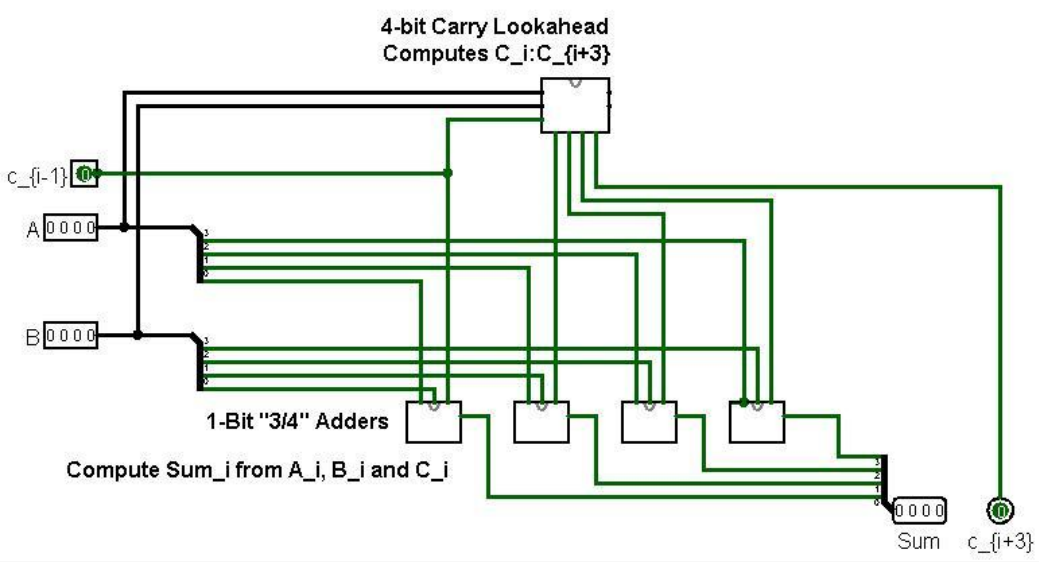
Carry-Lookahead Logic

Finally, here's how we can calculate c_3 and c_4:

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

$$
\begin{aligned}
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}
$$

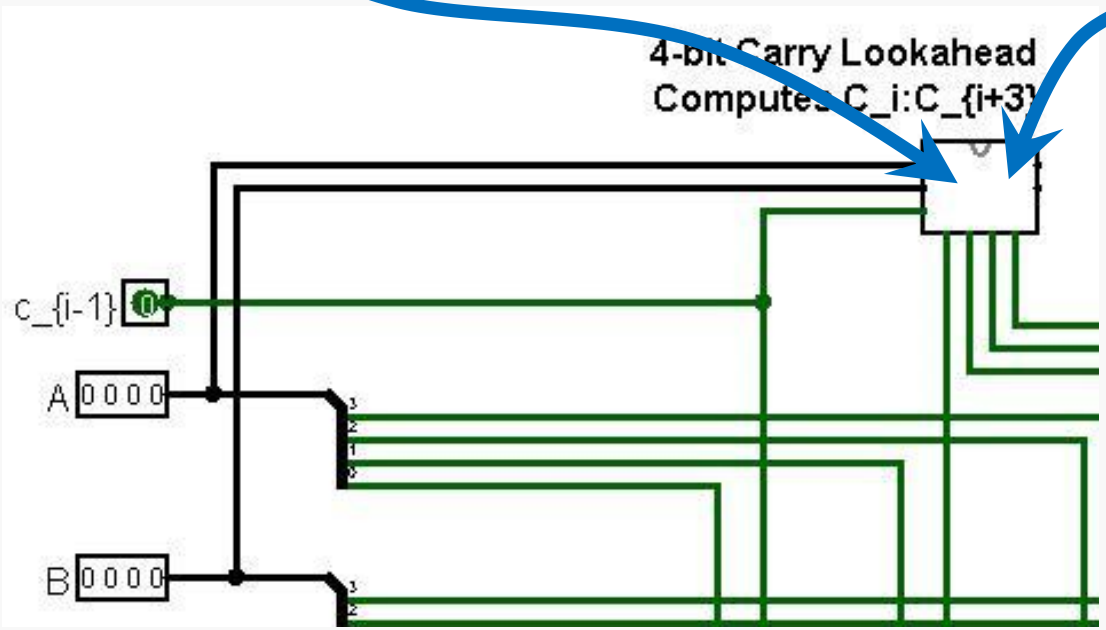
So, we have the necessary logic to implement the 4-bit Carry Lookahead unit for our 4-bit Carry Lookahead Adder:



Carry-Lookahead Logic

$$g_i = a_i \cdot b_i$$
$$p_i = a_i + b_i$$

$$c_1 = g_0 + p_0 \cdot c_0$$
$$c_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$
$$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$



Abstraction

The g_i and p_i bits represent an abstract view of how carry bits are generated and propagate during addition:

$$g_i = a_i \cdot b_i$$

generate bit for *i*-th column

adding the summand bits generates a carry-out bit iff both summand bits are 1

$$p_i = a_i + b_i$$

propagate bit for *i*-th column

if $c_i = 1$ (the carry-out bit from the previous column), there's a carry-out into the next column iff at least one of the summand bits is 1

Abstraction

So, here's why the formulas we've derived make sense intuitively:

$$c_1 = g_0 + p_0 \cdot c_0$$

c_1 is 1 iff:

c_0 was 1 and column 0 propagated it

or

column 0 generated a carry-out

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

c_4 is 1 iff:

c_0 was 1 and columns 0 to 3 propagated it, or

column 0 generated a carry-out and columns 1 to 3 propagated it, or

column 1 generated a carry-out and columns 2 to 3 propagated it, or

column 2 generated a carry-out and column 3 propagated it, or

column 3 generated a carry-out