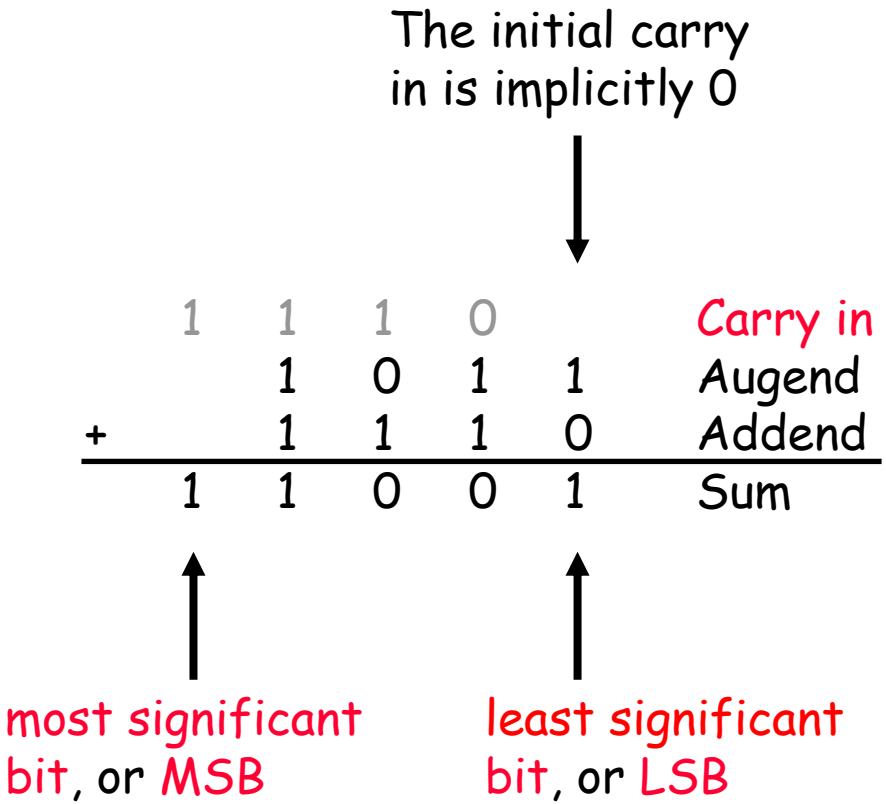
Chapter 5:Arithmetic Functions and Circuits

Binary Addition by Hand

- You can add two binary numbers one column at a time starting from the right, just as you add two decimal numbers
- But remember that it's binary. For example, 1 + 1 = 10 and you have to carry!

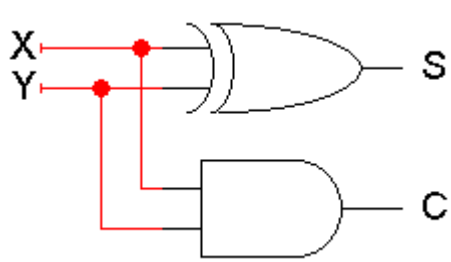


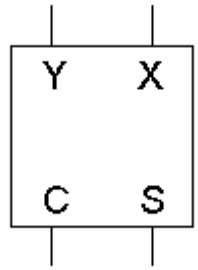
Adding Two Bits

- A hardware adder by copying the human addition algorithm
- Half adder: Adds two bits and produces a two-bit result: a sum (the right bit) and a carry out (the left bit)
- Here are truth tables, equations, circuit and block symbol

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

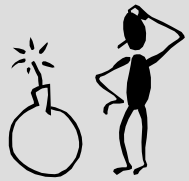
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10





$C = XY$

$S = X'Y + XY' = X \oplus Y$



Be careful! Now we're using + for both arithmetic addition and the logical OR operation.

Adding Three Bits

- But what we really need to do is add *three* bits: the augend and addend, and the *carry in* from the right.

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

X	Y	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

0 + 0 + 0 = 00
0 + 0 + 0 = 01
0 + 1 + 0 = 01
0 + 1 + 1 = 10
1 + 0 + 0 = 01
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11

Full Adder

- Full adder: Three bits of input, two-bit output consisting of a sum and a carry out
- Using Boolean algebra, we get the equations shown here
  - XOR operations simplify the equations a bit
  - We used algebra because you can't easily derive XORs from K-maps

X	Y	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

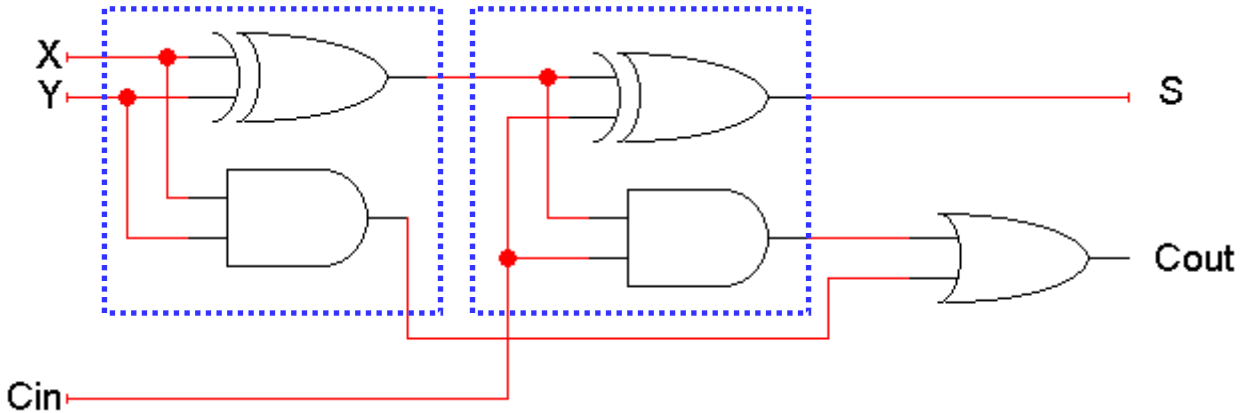
$$
\begin{aligned}
S &= \Sigma m(1,2,4,7) \\
&= X' Y' C_{in} + X' Y C_{in}' + X Y' C_{in}' + X Y C_{in} \\
&= X' (Y' C_{in} + Y C_{in}') + X (Y' C_{in}' + Y C_{in}) \\
&= X' (Y \oplus C_{in}) + X (Y \oplus C_{in})' \\
&= X \oplus Y \oplus C_{in}
\end{aligned}
$$

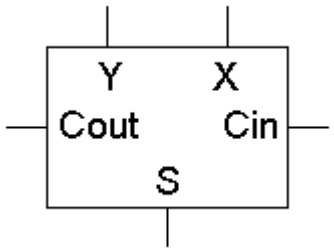
$$
\begin{aligned}
C_{out} &= \Sigma m(3,5,6,7) \\
&= X' Y C_{in} + X Y' C_{in} + X Y C_{in}' + X Y C_{in} \\
&= (X' Y + X Y') C_{in} + XY(C_{in}' + C_{in}) \\
&= (X \oplus Y) C_{in} + XY
\end{aligned}
$$

Full Adder Circuit

- These things are called half adders and full adders because you can build a full adder by putting together two half adders!

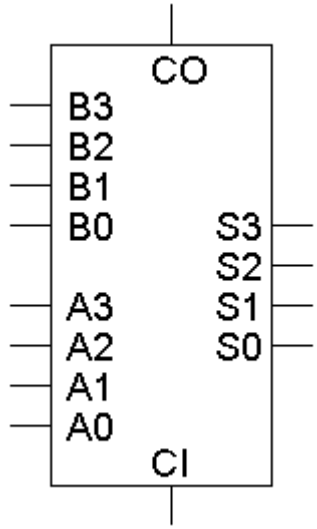
$$S = X \oplus Y \oplus C_{in}$$
$$C_{out} = (X \oplus Y)\, C_{in} + XY$$

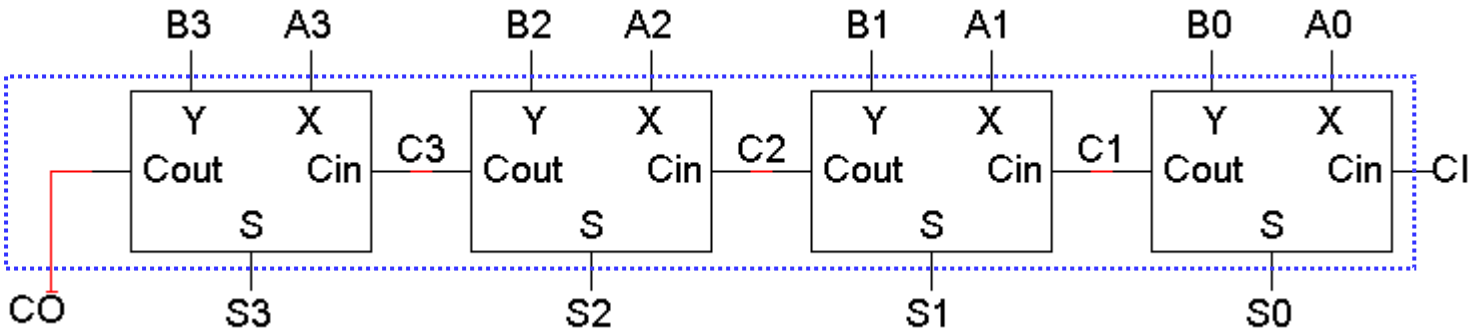




A 4-bit Adder

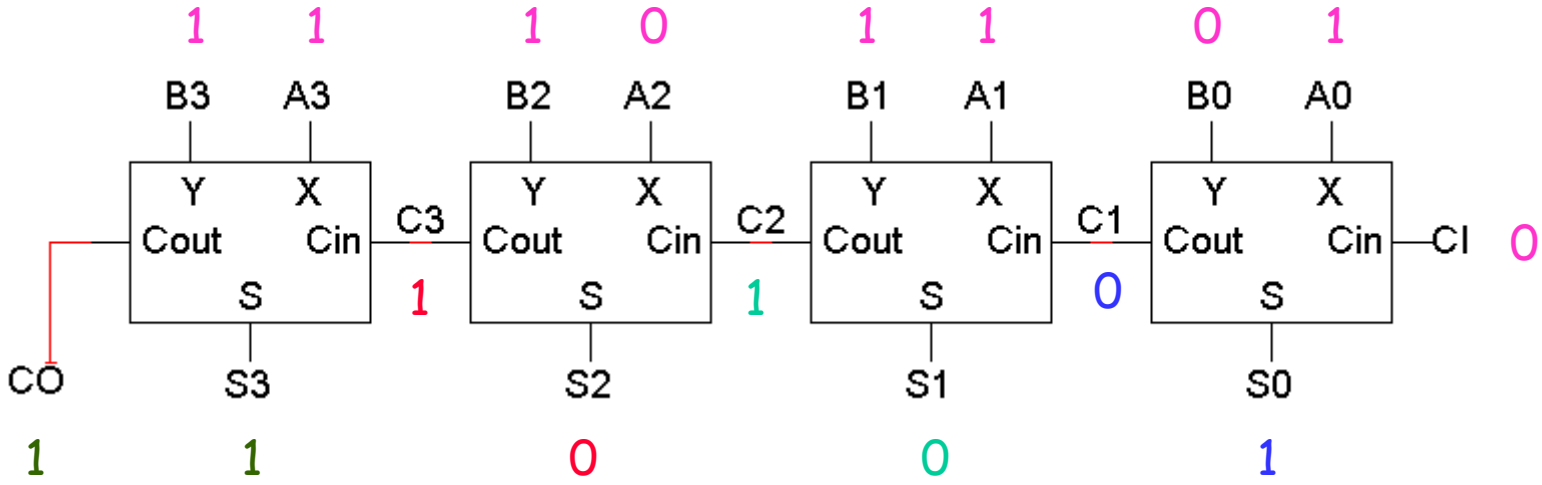
- Four full adders together make a 4-bit adder
- There are nine total inputs:
  - Two 4-bit numbers, A3 A2 A1 A0 and B3 B2 B1 B0
  - An initial carry in, CI
- The five outputs are:
  - A 4-bit sum, S3 S2 S1 S0
  - A carry out, CO
- Imagine designing a nine-input adder without this hierarchical structure—you'd have a 512-row truth table with five outputs!





An example of 4-bit addition

- Let's try our initial example: A=1011 (eleven), B=1110 (fourteen)



1. Fill in all the inputs, including CI=0
2. The circuit produces C1 and S0 (1 + 0 + 0 = 01)
3. Use C1 to find C2 and S1 (1 + 1 + 0 = 10)
4. Use C2 to compute C3 and S2 (0 + 1 + 1 = 10)
5. Use C3 to compute CO and S3 (1 + 1 + 1 = 11)

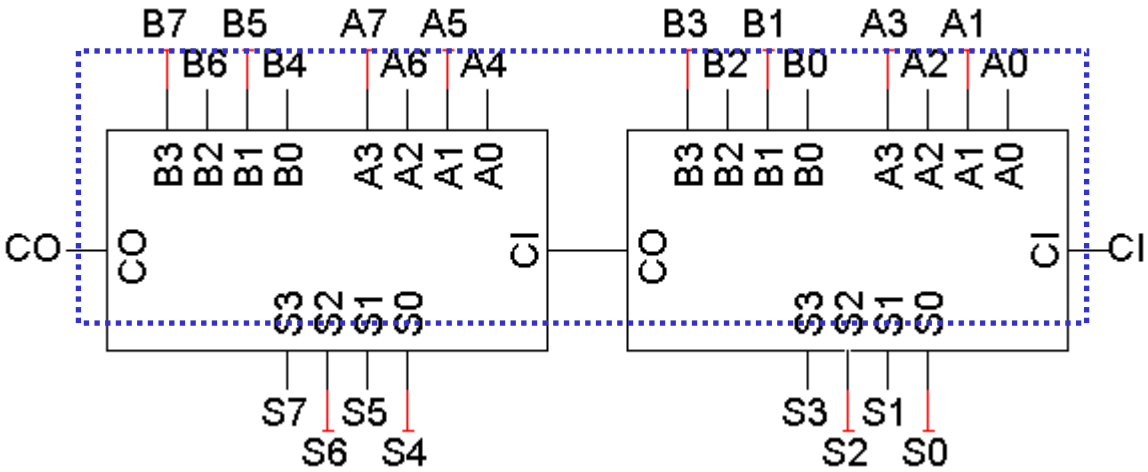
The final answer is 11001 (twenty-five)

Overflow

- In this case, note that the answer (11001) is *five* bits long, while the inputs were each only four bits (1011 and 1110). This is called overflow
- Although the answer 11001 is correct, we cannot use that answer in any subsequent computations with this 4-bit adder
- For **unsigned addition**, overflow occurs when the carry out is 1

Hierarchical Adder Design

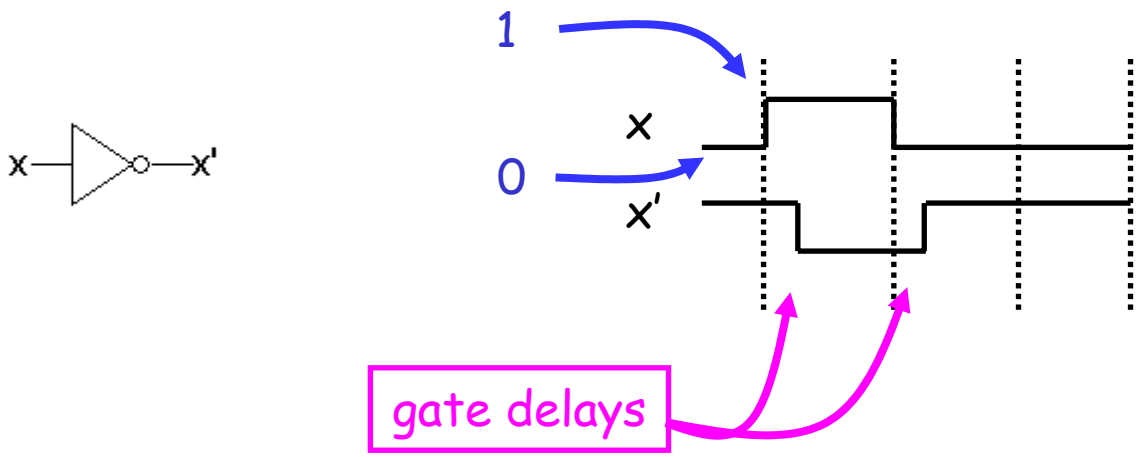
- When you add two 4-bit numbers the carry in is always 0, so why does the 4-bit adder have a CI input?
- One reason is so we can put 4-bit adders together to make even larger adders! This is just like how we put four full adders together to make the 4-bit adder in the first place
- Here is an 8-bit adder, for example.



- CI is also useful for subtraction!

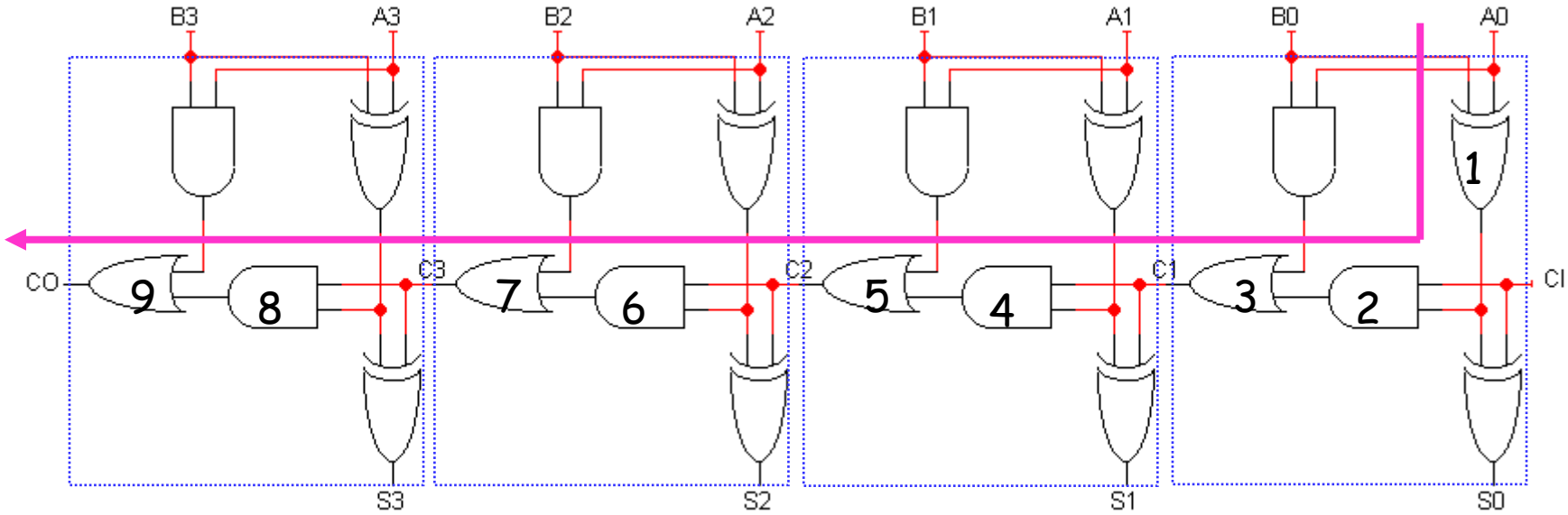
Gate Delays

- Every gate takes some small fraction of a second between the time inputs are presented and the time the correct answer appears on the outputs. This little fraction of a second is called a gate delay
- There are actually detailed ways of calculating gate delays that can get quite complicated, but for this class, let's just assume that there's some small constant delay that's the same for all gates
- We can use a timing diagram to show gate delays graphically



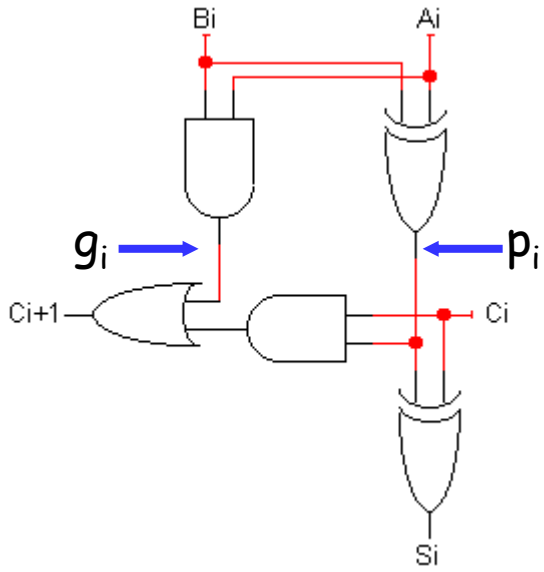
Delays in the Ripple Carry Adder

- The diagram below shows a 4-bit adder completely drawn out
- This is called a ripple carry adder, because the inputs A_0, B_0 and CI "ripple" leftwards until CO and S_3 are produced
- Ripple carry adders are slow!
  - Our example addition with 4-bit inputs required 5 "steps"
  - There is a very long path from A_0, B_0 and CI to CO and S_3
  - For an n-bit ripple carry adder, the longest path has 2n+1 gates
  - Imagine a 64-bit adder. The longest path would have 129 gates!



A faster way to compute carry outs

- Instead of waiting for the carry out from all the previous stages, we could compute it directly with a two-level circuit, thus minimizing the delay



- First we define two functions
  - The "generate" function g_i produces 1 when there *must* be a carry out from position i (i.e., when A_i and B_i are both 1).

$$g_i = A_i B_i$$

  - The "propagate" function p_i is true when, if there is an incoming carry, it is propagated (i.e, when A_i=1 or B_i=1, but not both).

$$p_i = A_i \oplus B_i$$

- Then we can rewrite the carry out function:

$$c_{i+1} = g_i + p_i c_i$$

A_i	B_i	C_i	C_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Algebraic Carry Out

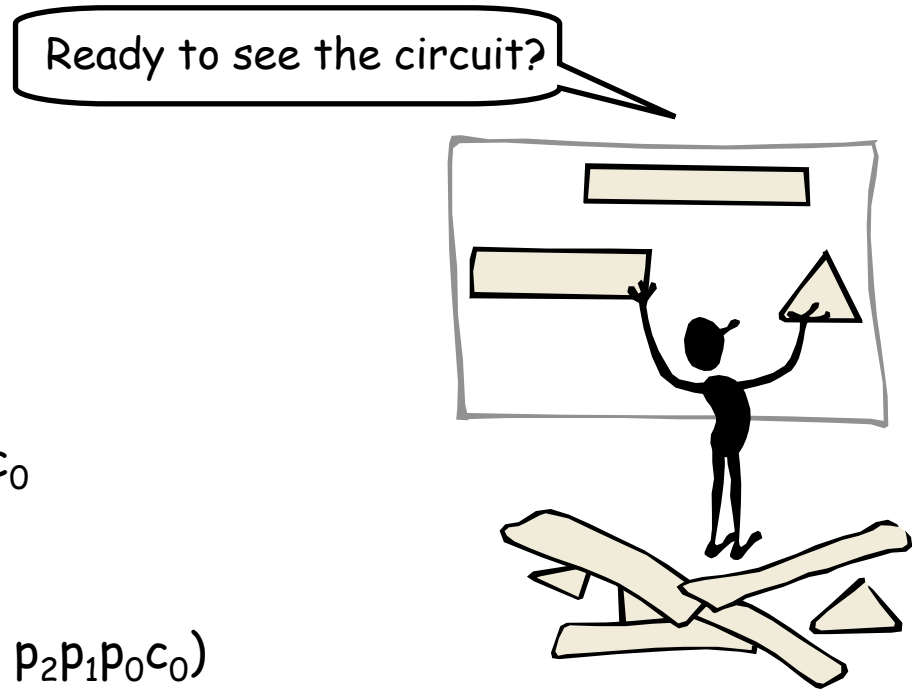
- Let's look at the carry out equations for specific bits, using the general equation from the previous page $c_{i+1} = g_i + p_i c_i$:

$c_1 = g_0 + p_0 c_0$

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

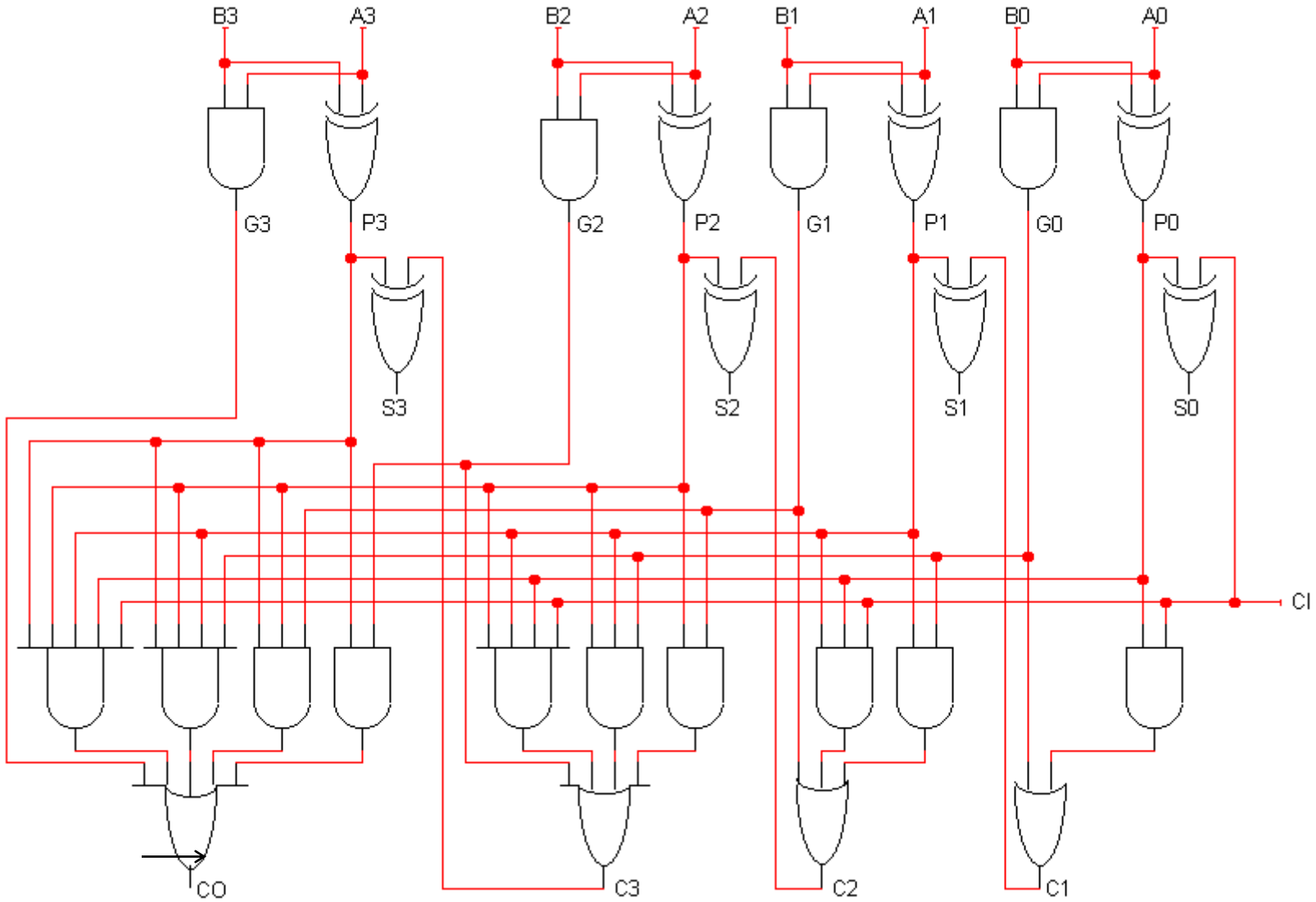
$c_3 = g_2 + p_2 c_2$
$= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0)$
$= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

$c_4 = g_3 + p_3 c_3$
$= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0)$
$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$



- These expressions are all sums of products, so we can use them to make a circuit with only a two-level delay.

A 4-bit CLA Circuit



Carry Lookahead Adders

- This is called a carry lookahead adder
- By adding more hardware, we reduced the number of levels in the circuit and sped things up
- We can "cascade" carry lookahead adders, just like ripple carry adders
- How much faster is this?
  - For a 4-bit adder, not much. CLA: 4 gates, RCA: 9 gates
  - But if we do the cascading properly, for a 16-bit carry lookahead adder, 8 gates vs. 33
  - Newer CPUs these days use 64-bit adders. That's 12 vs. 129 gates!
- The delay of a carry lookahead adder grows *logarithmically* with the size of the adder, while a ripple carry adder's delay grows *linearly*
- Trade-off between complexity and performance. Ripple carry adders are simpler, but slower. Carry lookahead adders are faster but more complex

Multiplication

- Multiplication can't be that hard!
  - It's just repeated addition
  - If we have adders, we can do multiplication also
- Remember that the AND operation is equivalent to multiplication on two bits:

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a×b
0	0	0
0	1	0
1	0	0
1	1	1

Binary multiplication example

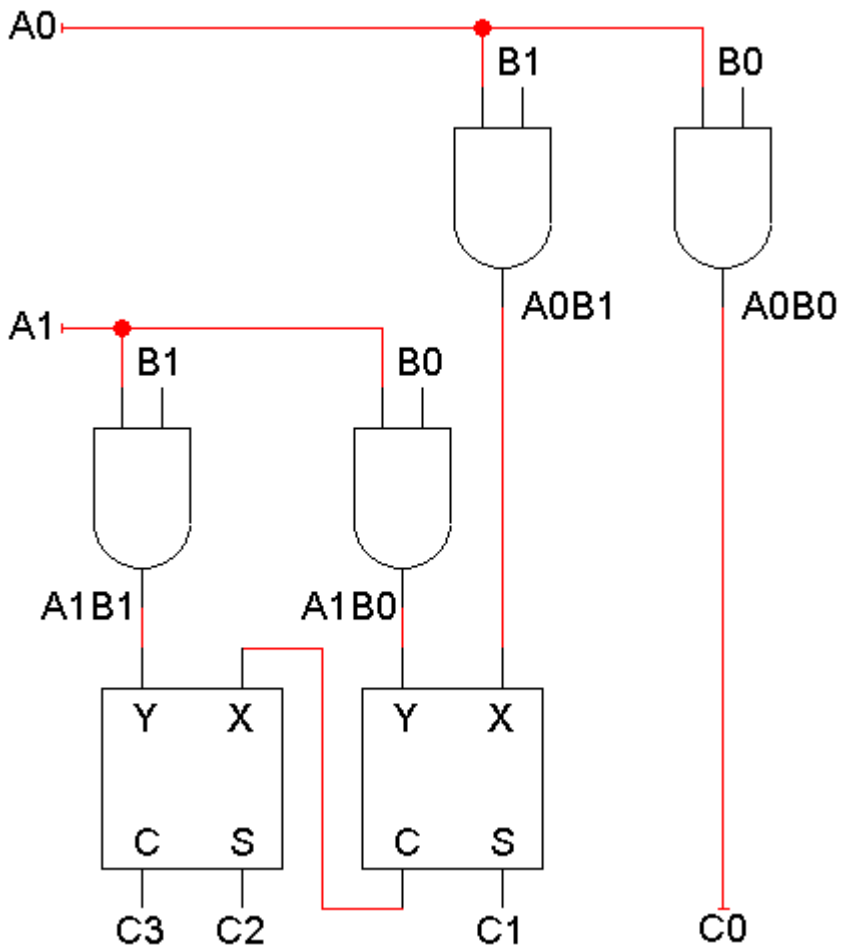
				1	1	0	1	Multiplicand
			x	0	1	1	0	Multiplier
				0	0	0	0	Partial products
			1	1	0	1		
		1	1	0	1			
+	0	0	0	0				
	1	0	0	1	1	1	0	Product

- Since we always multiply by either 0 or 1, the partial products are always either 0000 or the multiplicand (1101 in this example)
- There are four partial products which are added to form the result
  - We can add them in pairs, using three adders
  - Even though the product has up to 8 bits, we can use 4-bit adders if we "stagger" them leftwards, like the partial products themselves

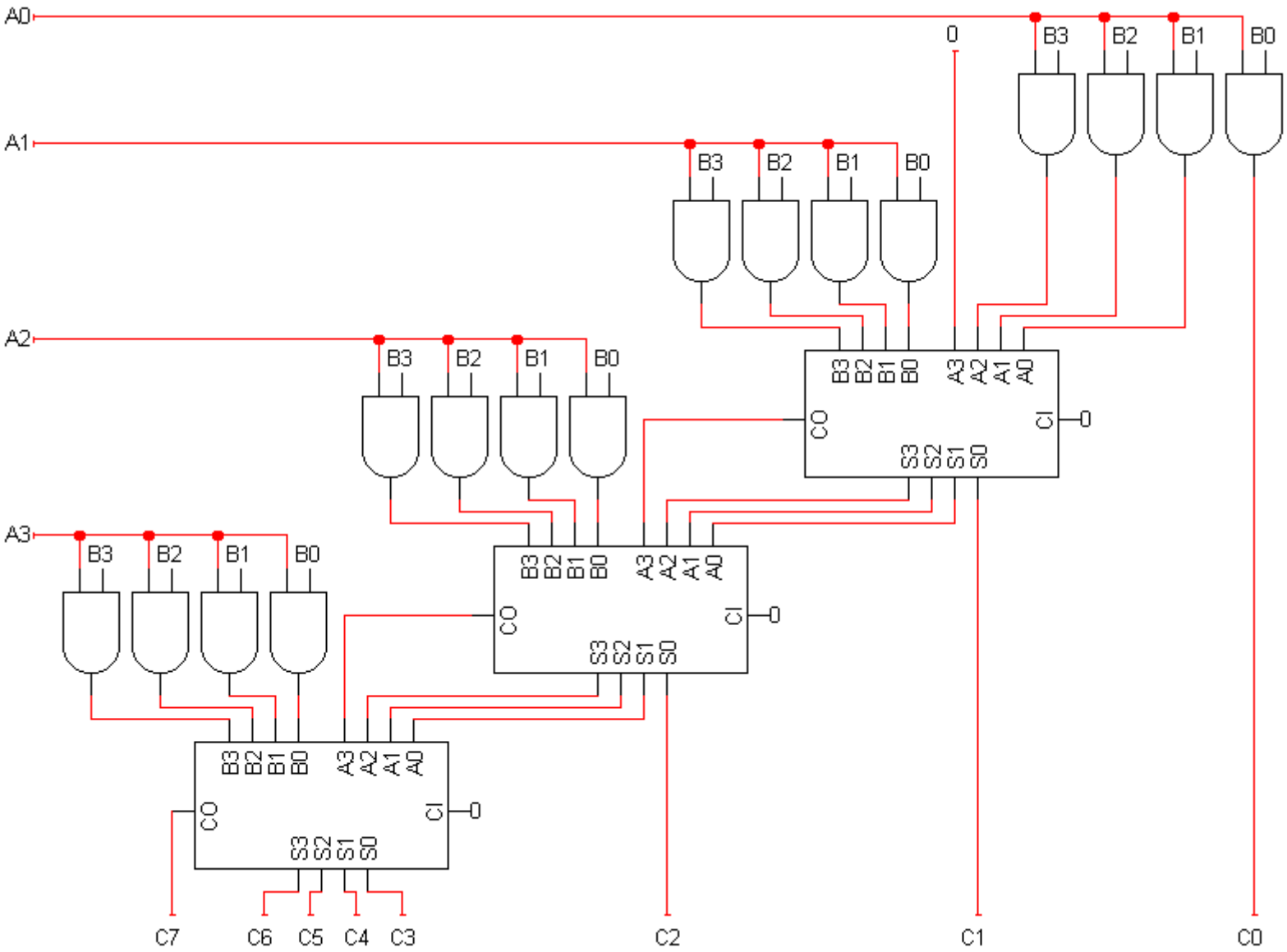
A 2x2 Binary Multiplier

- The AND gates produce the partial products
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products. In general, though, we'll need full adders
- Here C_3-C_0 are the product, not carries!

			B_1	B_0
		x	A_1	A_0
			A_0B_1	A_0B_0
+		A_1B_1	A_1B_0	
	C_3	C_2	C_1	C_0

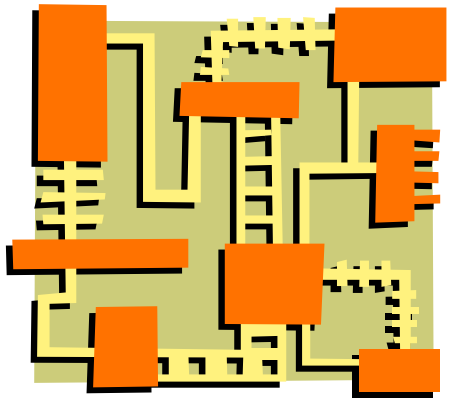


A 4x4 Multiplier Circuit



More on Multipliers

- Multipliers are very complex circuits
  - In general, when multiplying an m-bit number by an n-bit number:
    - There are n partial products, one for each bit of the multiplier
    - This requires n-1 adders, each of which can add m bits (the size of the multiplicand)
  - The circuit for 32-bit or 64-bit multiplication would be huge!



Multiplication: a special case

- In decimal, an easy way to multiply by 10 is to shift all the digits to the left, and tack a 0 to the right end

$$128 \times 10 = 1280$$

- We can do the same thing in binary. Shifting left is equivalent to multiplying by 2:

$$11 \times 10 = 110 \qquad \text{(in decimal, } 3 \times 2 = 6\text{)}$$

- Shifting left twice is equivalent to multiplying by 4:

$$11 \times 100 = 1100 \qquad \text{(in decimal, } 3 \times 4 = 12\text{)}$$

- As an aside, shifting to the *right* is equivalent to *dividing* by 2.

$$110 \div 10 = 11 \qquad \text{(in decimal, } 6 \div 2 = 3\text{)}$$

Addition and Multiplication Summary

- Adder and multiplier circuits mimic human algorithms for addition and multiplication
- Adders and multipliers are built hierarchically
  - We start with half adders or full adders and work our way up
  - Building these functions from scratch with truth tables and K-maps would be pretty difficult
- The arithmetic circuits impose a limit on the number of bits that can be added. Exceeding this limit results in overflow
- There is a tradeoff between simple but slow circuits (ripple carry adders) and complex but fast circuits (carry lookahead adders)
- Multiplication and division by powers of 2 can be handled with simple shifting

Signed Numbers

- The arithmetic we did so far was limited to unsigned (positive) integers
- How about negative numbers and subtraction?
- We'll look at three different ways of representing signed numbers
- How can we decide which representation is better?
  - The best one should result in the simplest and fastest operations
- We're mostly concerned with two particular operations:
  - Negating a signed number, or converting x into -x
  - Adding two signed numbers, or computing x + y

Signed Magnitude Representation

- Humans use a signed-magnitude system: we add + or - in front of a magnitude to indicate the sign
- We could do this in binary as well, by adding an extra sign bit to the front of our numbers. By convention:
  - A 0 sign bit represents a positive number
  - A 1 sign bit represents a negative number
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)
$0\,1101 = +13_{10}$ (a positive number in 5-bit signed magnitude)
$1\,1101 = -13_{10}$ (a negative number in 5-bit signed magnitude)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)
$0\,0100 = +4_{10}$ (a positive number in 5-bit signed magnitude)
$1\,0100 = -4_{10}$ (a negative number in 5-bit signed magnitude)

Signed Magnitude Operations

- Negating a signed-magnitude number is trivial: just change the sign bit from 0 to 1, or vice versa
- Adding numbers is difficult, though. Signed magnitude is basically what people use, so think about the grade-school approach to addition. It's based on comparing the signs of the augend and addend:
  - If they have the same sign, add the magnitudes and keep that sign
  - If they have different signs, then subtract the smaller magnitude from the larger one. The sign of the number with the larger magnitude is the sign of the result
- This method of subtraction would lead to a rather complex circuit.

```
  + 3  7  9          because        5  13  17
+ -6  4  7                        -  6   4   7
-----------                       ------------
 -2  6  8                          2   6   8
```

One's Complement Representation

- A different approach, one's complement, negates numbers by complementing each bit of the number
- We keep the sign bits: 0 for positive numbers, and 1 for negative. The sign bit is complemented along with the rest of the bits
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)
$01101 = +13_{10}$ (a positive number in 5-bit one's complement)
$10010 = -13_{10}$ (a negative number in 5-bit one's complement)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)
$00100 = +4_{10}$ (a positive number in 5-bit one's complement)
$11011 = -4_{10}$ (a negative number in 5-bit one's complement)

Why is it called "one's complement?"

- Complementing a single bit is equivalent to subtracting it from 1

  0′ = 1, and 1 - 0 = 1 1′ = 0, and 1 - 1 = 0

- Similarly, complementing each bit of an n-bit number is equivalent to subtracting that number from 2^n-1 (111...111)

- For example, we can negate the 5-bit number 01101
  - Here n=5, and 2^n-1 = 31_{10} = 11111_2
  - Subtracting 01101 from 11111 yields 10010

```
  1 1 1 1 1
- 0 1 1 0 1
-----------
  1 0 0 1 0
```

One's Complement Addition

- To add one's complement numbers:
  - First do unsigned addition on the numbers, *including* the sign bits
  - Then take the carry out and add it to the sum
- Two examples:

```
   0111     (+7)
+  1011   + (-4)
  ------   -----
 1 0010

   0010
+     1
  ------
   0011     (+3)
```

```
   0011     (+3)
+  0010   + (+2)
  ------   -----
 0 0101

   0101
+     0
  ------
   0101     (+5)
```

- This is simpler and more uniform than signed magnitude addition

Two's Complement

- Our final idea is two's complement. To negate a number, complement each bit (just as for ones' complement) and then add 1
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)
$0\,1101 = +13_{10}$ (a positive number in 5-bit two's complement)
$1\,0010 = -13_{10}$ (a negative number in 5-bit *ones'* complement)
$1\,0011 = -13_{10}$ (a negative number in 5-bit two's complement)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)
$0\,0100 = +4_{10}$ (a positive number in 5-bit two's complement)
$1\,1011 = -4_{10}$ (a negative number in 5-bit *ones'* complement)
$1\,1100 = -4_{10}$ (a negative number in 5-bit two's complement)

More about two's complement

- Two other equivalent ways to negate two's complement numbers:
  - You can subtract an n-bit two's complement number from 2^n

$$\begin{array}{r} 1\,00000 \\ -\ 01101 \quad (+13_{10}) \\ \hline 10011 \quad (-13_{10}) \end{array} \qquad \begin{array}{r} 1\,00000 \\ -\ 00100 \quad (+4_{10}) \\ \hline 11100 \quad (-4_{10}) \end{array}$$

  - You can complement all of the bits to the left of the rightmost 1

01101 = $+13_{10}$ (a positive number in two's complement)
10011 = -13_{10} (a negative number in two's complement)
00100 = $+4_{10}$ (a positive number in two's complement)
11100 = -4_{10} (a negative number in two's complement)

Two's Complement Addition

- Negating a two's complement number takes a bit of work, but addition is much easier than with the other two systems
- To find A + B, you just have to:
  - Do unsigned addition on A and B, including their sign bits
  - Ignore any carry out
- For example, to find 0111 + 1100, or (+7) + (-4):
  - First add 0111 + 1100 as unsigned numbers:

$$\begin{array}{r} 0\,1\,1\,1 \\ +\ 1\,1\,0\,0 \\ \hline 1\,0\,0\,1\,1 \end{array}$$

  - Discard the carry out (1)
  - The answer is 0011 (+3)

Another two's complement example

- To further convince you that this works, let's try adding two negative numbers—1101 + 1110, or (-3) + (-2) in decimal
- Adding the numbers gives 11011:

$$\begin{array}{r} 1\,1\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,1\,0\,1\,1 \end{array}$$

- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

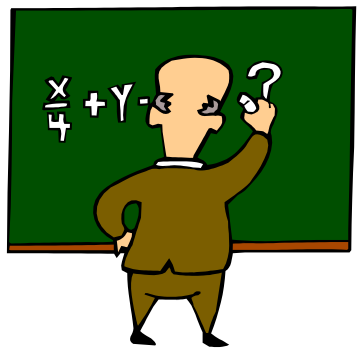
Why does this work?

- For n-bit numbers, the negation of B in two's complement is 2^n - B (this is one of the alternative ways of negating a two's-complement number)

$$\begin{aligned} A - B &= A + (-B) \\ &= A + (2^n - B) \\ &= (A - B) + 2^n \end{aligned}$$

- If $A \geq B$, then (A - B) is a positive number, and 2^n represents a carry out of 1. Discarding this carry out is equivalent to subtracting 2^n, which leaves us with the desired result (A - B)

- If $A < B$, then (A - B) is a negative number and we have 2^n - (B - A). This corresponds to the desired result, -(A - B), in two's complement form.



Comparing the signed number systems

- *Positive numbers are the same in all three representations*
- Signed magnitude and one's complement have *two* ways of representing 0. This makes things more complicated
- Two's complement has asymmetric ranges; there is one more negative number than positive number. Here, you can represent -8 but not +8
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest

Decimal	S.M.	1's comp.	2's comp.
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	—	—	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different systems on the previous page?

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- In general, with n-bit numbers including the sign, the ranges are:

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Converting signed numbers to decimal

- Convert 110101 to decimal, assuming this is a number in:

  (a) signed magnitude format

  (b) ones' complement

  (c) two's complement

Example solution

- Convert 110101 to decimal, assuming this is a number in:

  Since the sign bit is 1, this is a negative number. The easiest way to find the magnitude is to convert it to a positive number.

  (a) signed magnitude format

  Negating the original number, 110101, gives 010101, which is +21 in decimal. So 110101 must represent -21.

  (b) ones' complement

  Negating 110101 in ones' complement yields 001010 = $+10_{10}$, so the original number must have been -10_{10}.

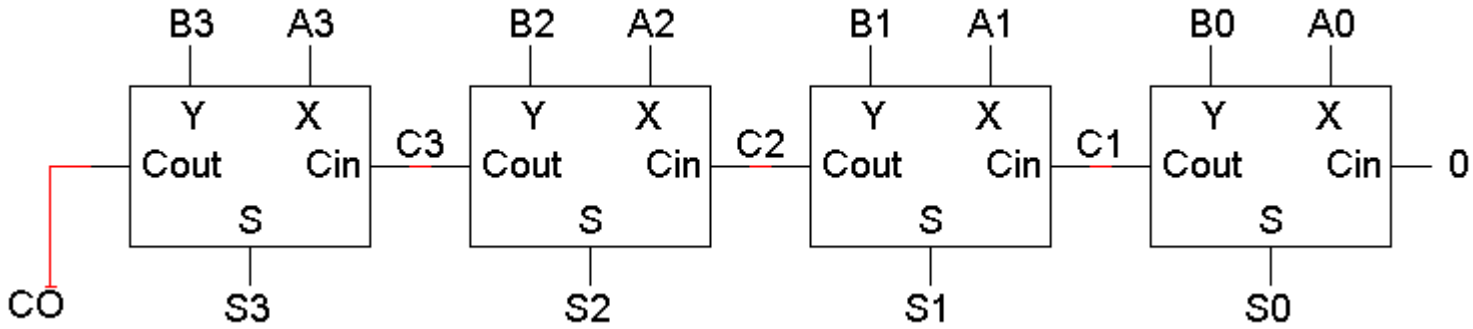
  (c) two's complement

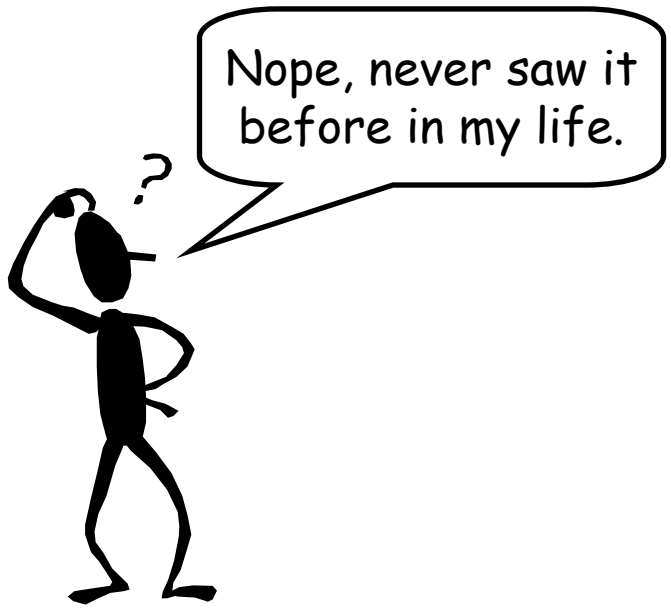
  Negating 110101 in two's complement gives 001011 = 11_{10}, which means 110101 = -11_{10}.

- The most important point here is that a binary number has *different* meanings depending on which representation is assumed

Our four-bit unsigned adder circuit

- Here is the four-bit unsigned addition circuit





Making a subtraction circuit

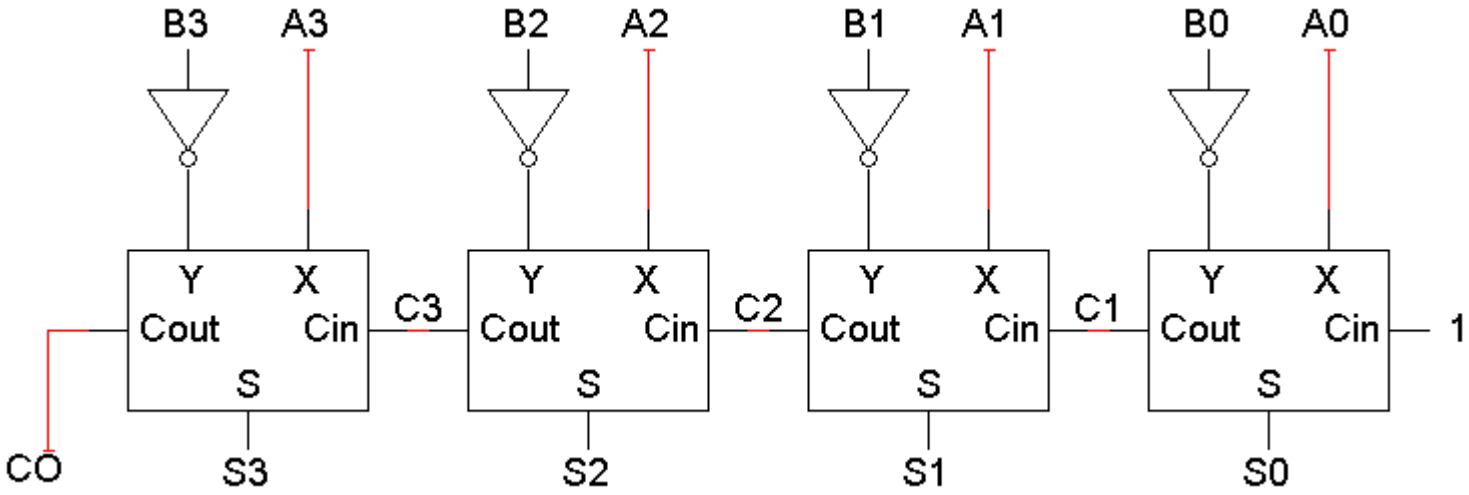
- We could build a subtraction circuit directly, similar to the way we made unsigned adders
- However, by using two's complement we can convert any subtraction problem into an addition problem. Algebraically,

$$A - B = A + (-B)$$

- So to subtract B from A, we can instead *add* the negation of B to A
- This way we can re-use the unsigned adder hardware

A two's complement subtraction circuit

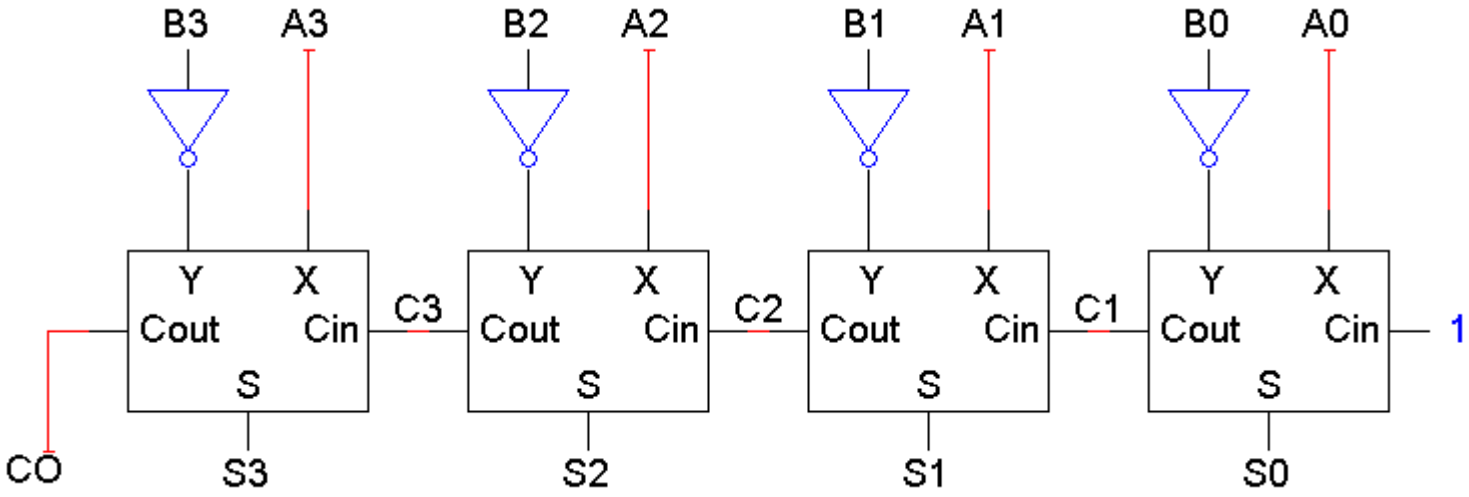
- To find A - B with an adder, we'll need to:
  - Complement each bit of B
  - Set the adder's carry in to 1
- The net result is A + B' + 1, where B' + 1 is the two's complement negation of B



- Remember that A3, B3 and S3 here are actually sign bits

Small differences

- The only differences between the adder and subtractor circuits are:
  - The subtractor has to negate B3 B2 B1 B0
  - The subtractor sets the initial carry in to 1, instead of 0



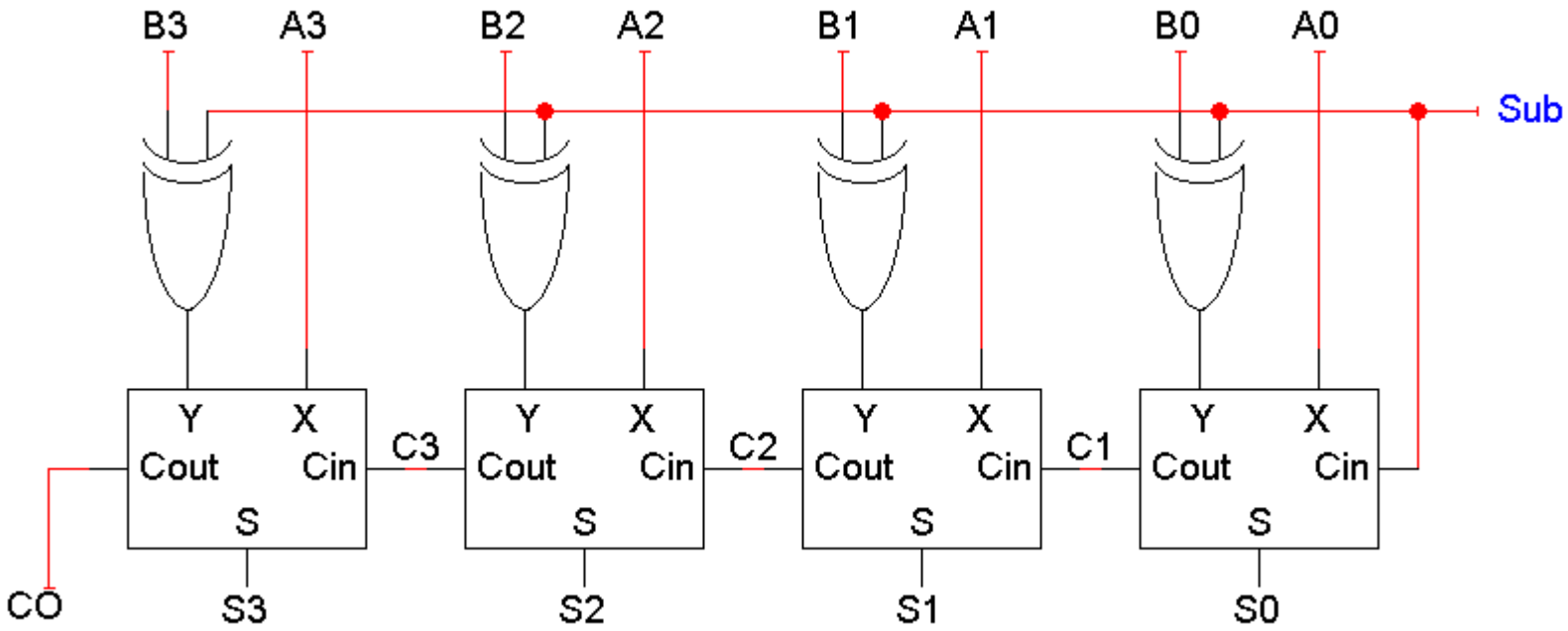
- Not too hard to make one circuit that does *both* addition and subtraction

An Adder-Subracter Circuit

- XOR gates let us selectively complement the B input

$$X \oplus 0 = X \qquad\qquad X \oplus 1 = X'$$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be *A* + B + 0, or just *A* + B
- When Sub = 1, the XOR gates output B3′ B2′ B1′ B0′ and the carry in is 1. Thus, the adder output will be a two's complement subtraction, *A* - B



Signed Overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8
- What if you try to compute 4 + 5, or (-4) + (-5)?

```
  0100   (+4)          1100   (-4)
+ 0101   (+5)        + 1011   (-5)
-------              -------
 01001   (-7)         10111   (+7)
```

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow
  - In the example on the left, the carry out is 0 but there *is* overflow
  - Conversely, there are situations where the carry out is 1 but there is *no* overflow

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits

```
  0100   (+4)        1100   (-4)
+ 0101   (+5)      + 1011   (-5)
-------            -------
 01001   (-7)       10111   (+7)
```

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result
  - If you add two *negative* numbers and get a *positive* result
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

Sign Extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0s in front of them. This helps in "lining up" numbers
- To subtract 231 and 3, for instance, you can imagine:

$$\begin{array}{r} 231 \\ -\ 003 \\ \hline 228 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude
- If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5)
  - But 1100 (-4) should become 1111 1100 (-4)
- The proper way to extend a signed binary number is to replicate the sign bit, so the sign is preserved

Subtraction Summary

- A good representation for negative numbers makes subtraction hardware much easier to design
  - Two's complement is used most often (although signed magnitude shows up sometimes, such as in floating-point systems)
  - Using two's complement, we can build a subtractor with minor changes to the adder
  - We can also make a single circuit which can both add and subtract
- Overflow is still a problem, but signed overflow is very different from the unsigned overflow
- Sign extension is needed to properly "lengthen" negative numbers