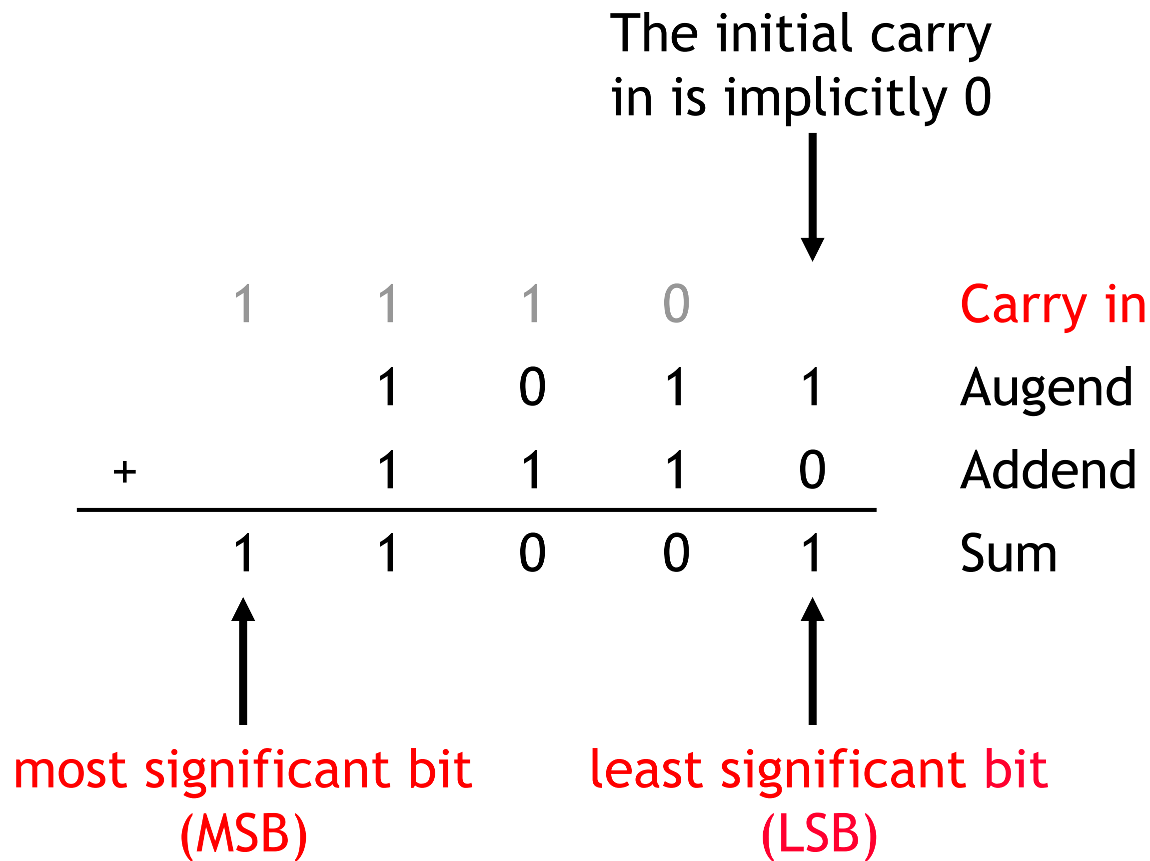
Addition and multiplication

- Arithmetic is the most basic thing you can do with a computer, but it's not as easy as you might expect!
- These next few lectures focus on addition, subtraction, multiplication and arithmetic-logic units or ALUs, which are the "heart" of CPUs.
- Arithmetic hardware is an excellent example of many issues that we've discussed, such as Boolean algebra, circuit analysis, data representation, and hierarchical, modular design.

©2000-2003 Howard Huang

Binary addition by hand

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



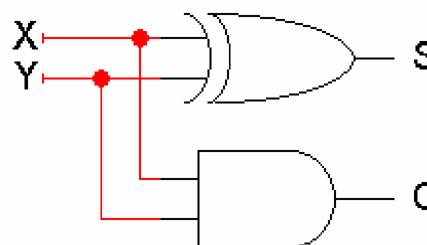
Adding two bits

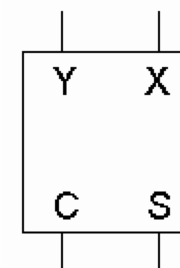
- We'll make a hardware adder based on our human addition algorithm.
- We start with a half adder, which adds two bits X and Y and produces a two-bit result: a sum S (the right bit) and a carry out C (the left bit).
- Here are truth tables, equations, circuit and block symbol.

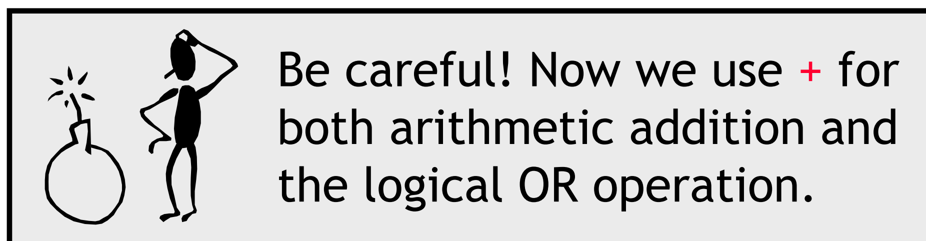
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$C = XY$

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







Adding three bits

- But what we really need to do is add *three* bits: the augend and addend bits, *and* the carry in from the right.
- A full adder circuit takes three inputs X, Y and C_{in}, and produces a two-bit output consisting of a sum S and a carry out C_{out}.
- This truth table should look familiar, as it was an example in the decoder and multiplexer lectures.

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

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

Full adder equations

- Using Boolean algebra, we can simplify S and C_{out} as shown here.

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

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

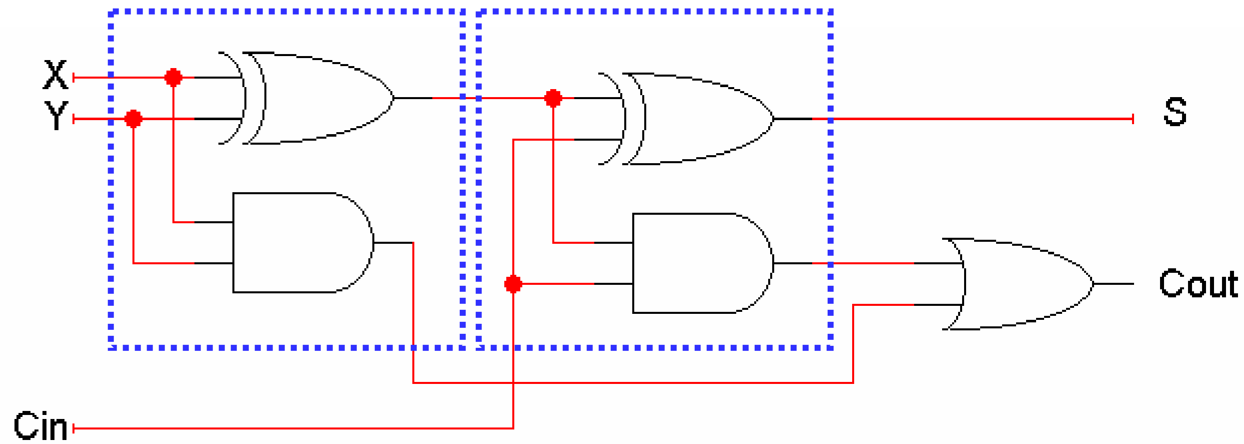
$$
\begin{aligned}
C_{out} &= \Sigma m(3,5,6,7) \\
&= X'YC_{in} + XY'C_{in} + XYC_{in}' + XYC_{in} \\
&= (X'Y + XY')C_{in} + XY(C_{in}' + C_{in}) \\
&= (X \oplus Y)C_{in} + XY
\end{aligned}
$$

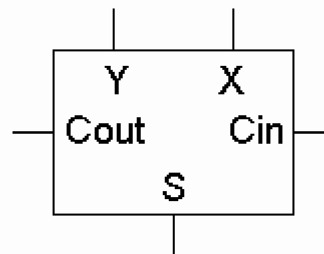
- Notice that XOR operations simplify things a bit, but we had to resort to using algebra since it’s hard to find XOR-based expressions with K-maps.

Full adder circuit

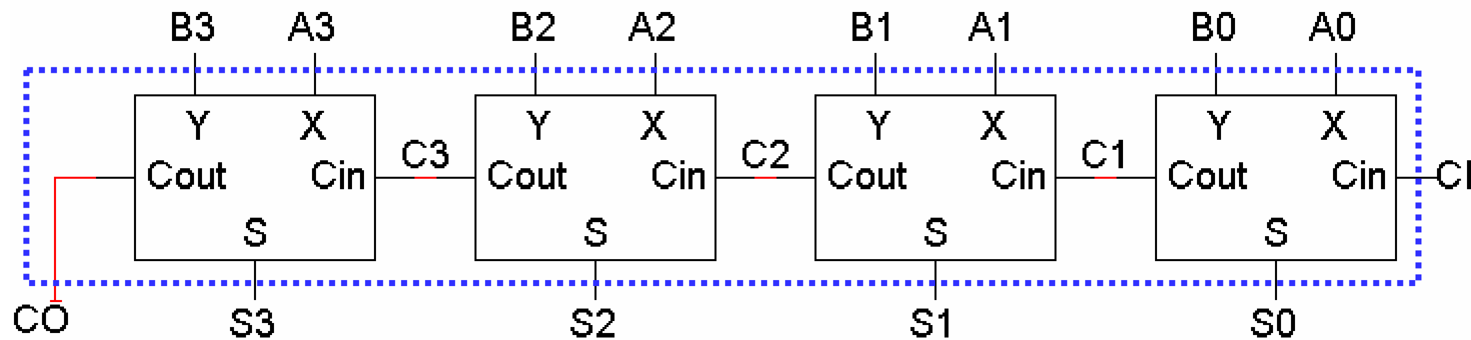
- We write the equations this way to highlight the hierarchical nature of adder circuits—you can build a full adder by combining two half adders!

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

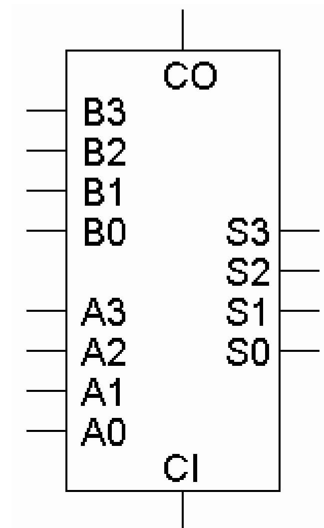




A four-bit adder

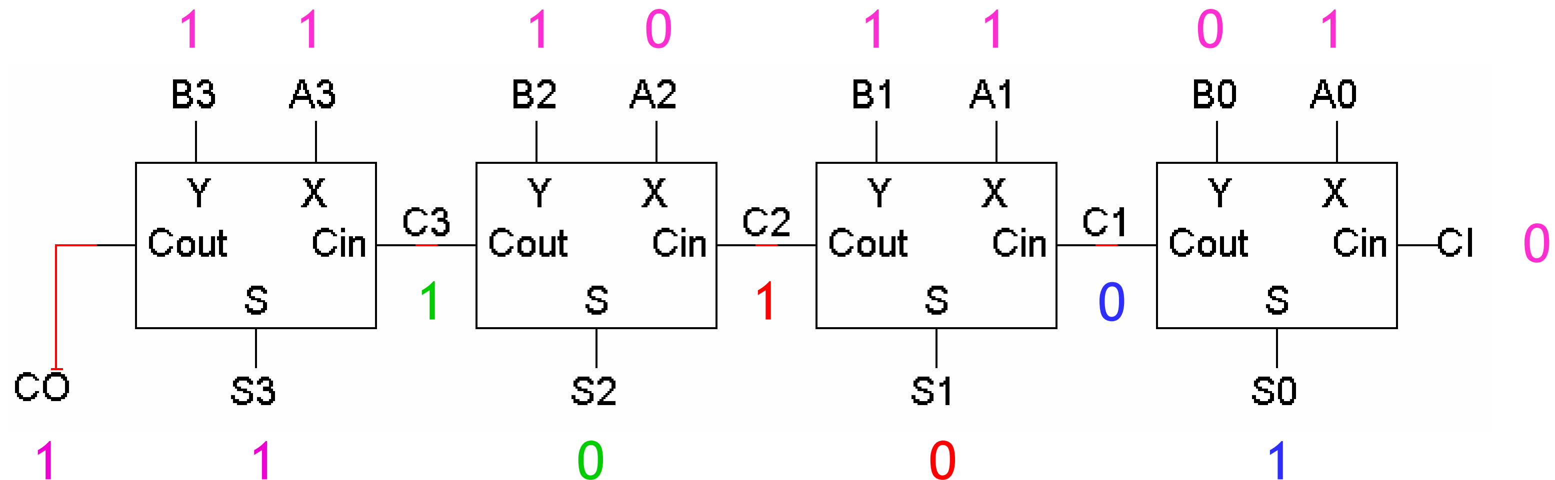


- Similarly, we can cascade four full adders to build a four-bit adder.
  - The inputs are two four-bit numbers (A3A2A1A0 and B3B2B1B0) and a carry in CI.
  - The two outputs are a four-bit sum S3S2S1S0 and the carry out CO.
- If you designed this adder without taking advantage of the hierarchical structure, you'd end up with a 512-row truth table with five outputs!



An example of 4-bit addition

- Let's put our initial example into this circuit, with A=1011 and B=1110.

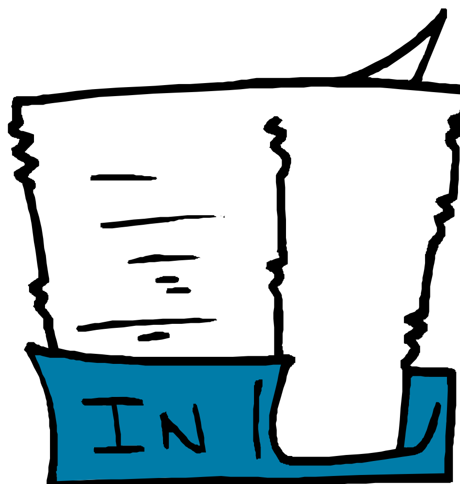


1. Fill in all the inputs, including CI=0
2. The circuit produces C1 and S0 (1 + 0 + 0 = 01)
3. Use C1 to find C2 and S1 (1 + 1 + 0 = 10)
4. Use C2 to compute C3 and S2 (0 + 1 + 1 = 10)
5. Use C3 to compute CO and S3 (1 + 1 + 1 = 11)

Woohoo! The final answer is 11001.

Overflow

- This particular example illustrates a case of overflow.
  - The answer 11001 is *five* bits long, but the inputs 1011 and 1110 were each only four bits wide.
  - Although the five-bit answer 11001 is correct, we cannot use it in any subsequent computations with this four-bit adder.
- For "unsigned" addition, overflow occurs when the carry out is 1.
- Overflows can never be prevented because computer hardware is finite. Even though we can make adders of arbitrary sizes, there will always be another number that won't fit.

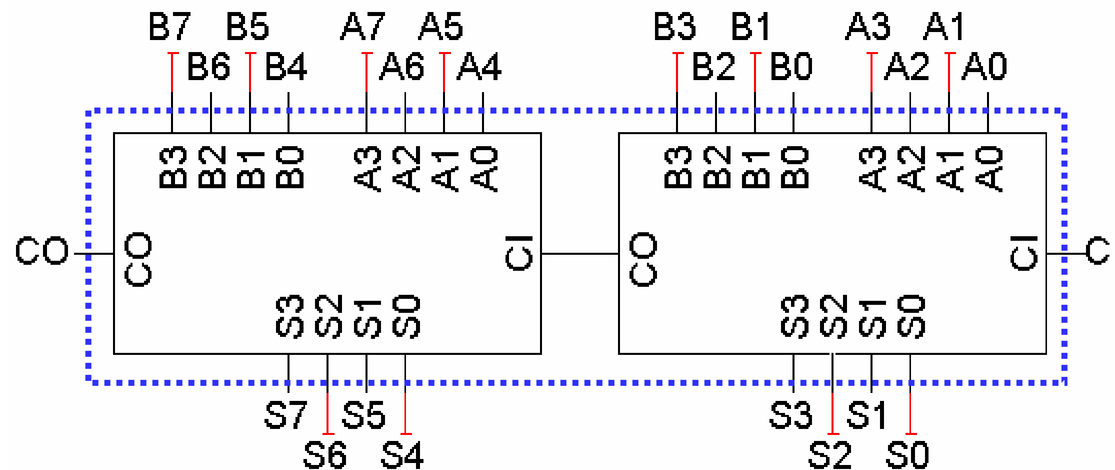


Ariane 5

- In 1996, the European Space Agency's Ariane 5 rocket was launched for the first time... and it exploded 40 seconds after liftoff.
- It turns out the Ariane 5 used software designed for the older Ariane 4.
  - The Ariane 4 stored its horizontal velocity as a 16-bit signed integer.
  - But the Ariane 5 reaches a much higher velocity, which caused an overflow in the 16-bit quantity.
- The overflow error was never caught, so incorrect instructions were sent to the rocket boosters and main engine.

Hierarchical adder design

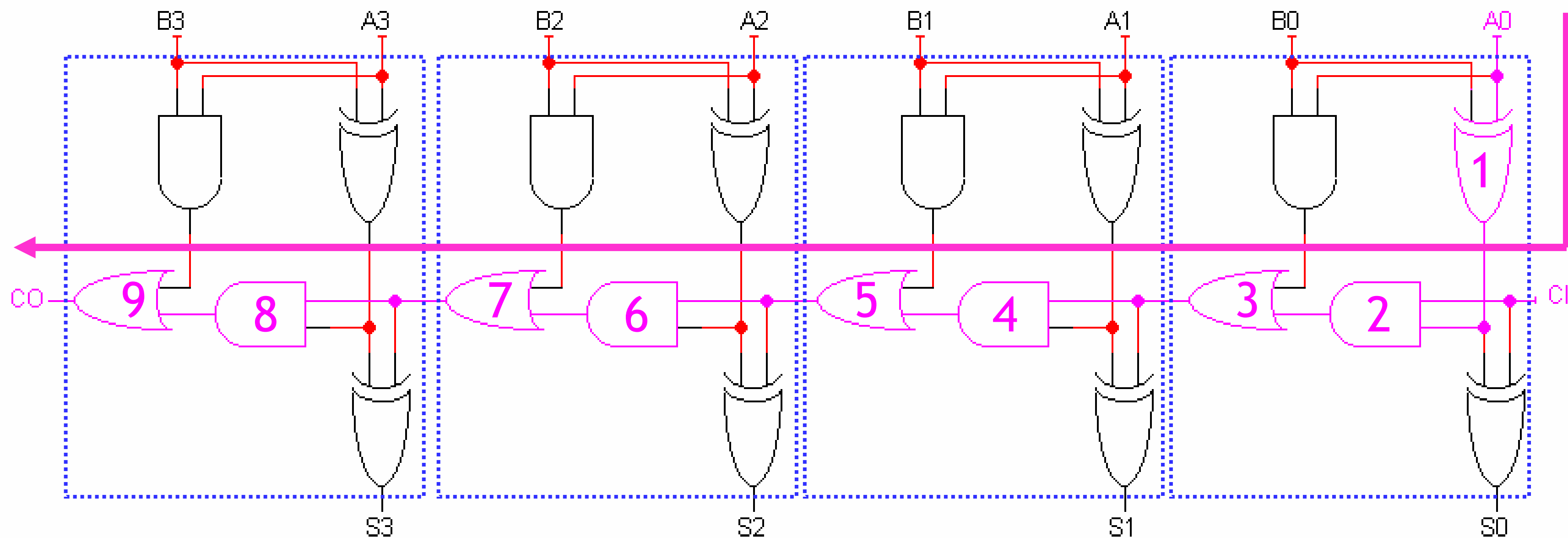
- When you add two 4-bit numbers the carry in is always 0, so why does the four-bit adder have a CI input?
- We can use CI to combine four-bit adders together to make even larger adders, just like we combined half adders and full adders earlier.
- Here is one way to build an eight-bit adder, for example.



- CI is also useful for subtraction, as we'll see next time.

Ripple carry delays

- The diagram below shows our four-bit adder completely drawn out.
- This is called a ripple carry adder, because the inputs A0, B0 and CI "ripple" leftwards until CO and S3 are produced.
- Ripple carry adders are slow!
  - There is a very long path from A0, B0 and CI to CO and S3.
  - For an *n*-bit ripple carry adder, the longest path has 2*n*+1 gates.
  - The longest path in a 64-bit adder would include 129 gates!



A faster way to compute carry outs

- Instead of waiting for the carry out from each previous stage, we can minimize the delay by computing it directly with a two-level circuit.
- First we'll 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).

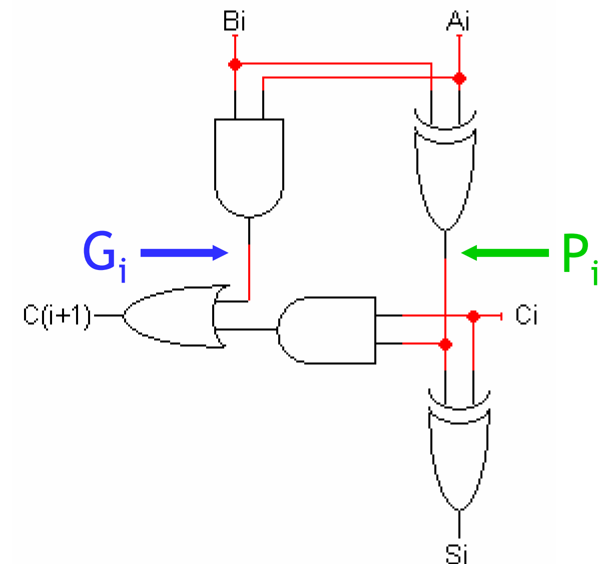
$$G_i = A_iB_i$$

  - The "propagate" function P_i is true when an incoming carry 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.

$$C_{i+1} = G_i + P_iC_i$$



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 hocus-pocus

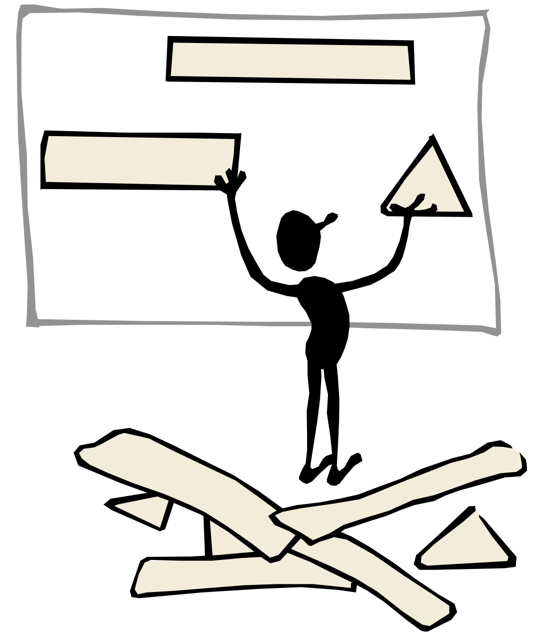
- Let's look at the carry out equations for specific bits, using the general equation from the previous page $C_{i+1} = G_i + P_iC_i$.

$$C_1 = G_0 + P_0C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1C_1 \\ &= G_1 + P_1(G_0 + P_0C_0) \\ &= G_1 + P_1G_0 + P_1P_0C_0 \end{aligned}$$

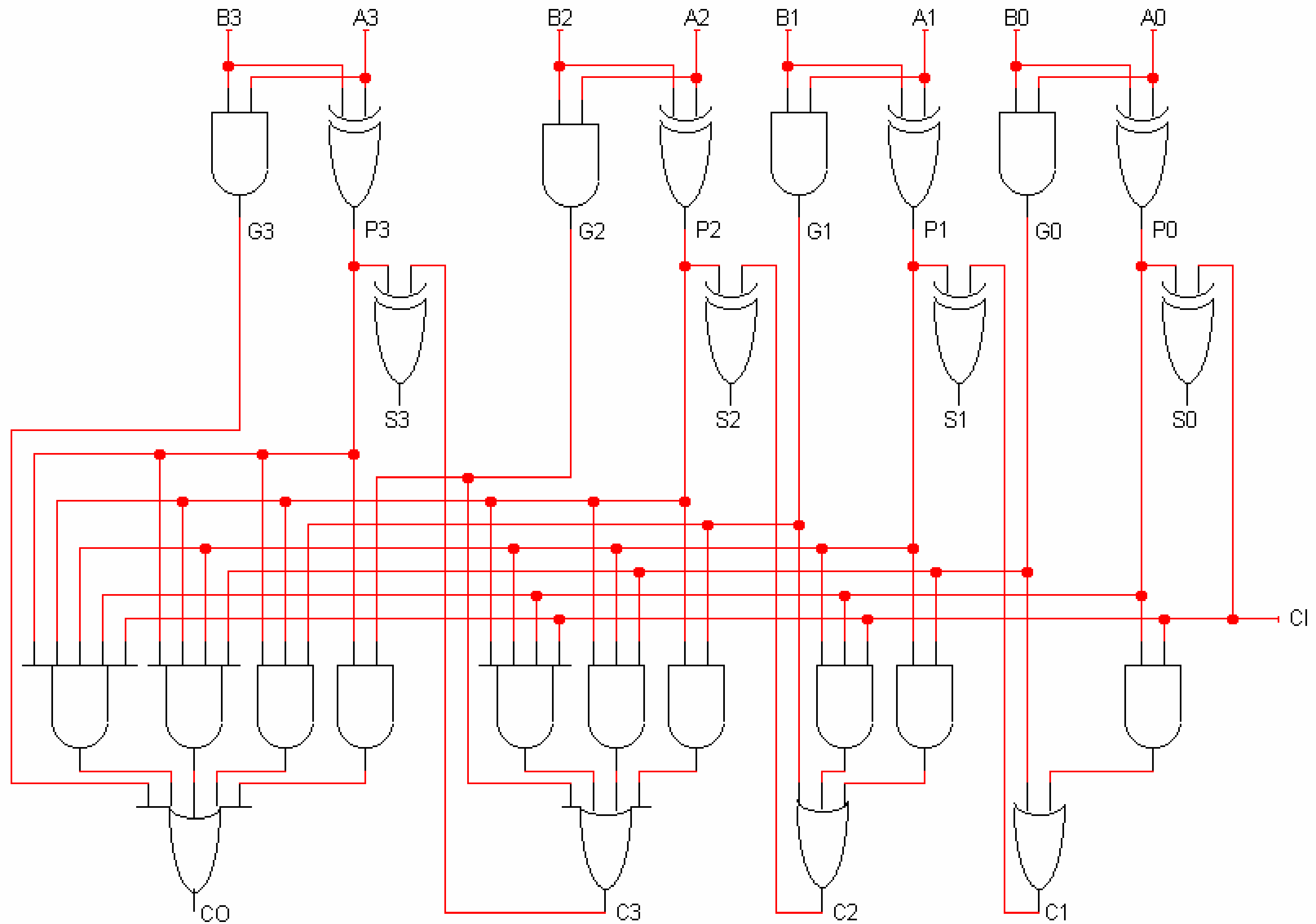
$$\begin{aligned} C_3 &= G_2 + P_2C_2 \\ &= G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \end{aligned}$$

$$\begin{aligned} C_4 &= G_3 + P_3C_3 \\ &= G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$



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

A faster four-bit adder



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 combine carry lookahead adders just like ripple carry adders, and we can do carry lookahead *between* the adders too.
- How much faster is this?
  - For a four-bit adder, not much. There are 4 gates in the longest path of a carry lookahead adder, versus 9 gates for a ripple carry adder.
  - If we do the cascading properly, a 16-bit carry lookahead adder will have only 8 gates in the longest path, as opposed to 33 with a ripple carry adder.
  - The newest processors 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*.

Binary multiplication by hand

- Multiplication can't be that hard! It's just repeated addition, so if we have adders, we should be able to do multiplication also.
- Here's an example of binary multiplication to get the mojo flowing.

				1	1	0	1
			×	0	1	1	0
				0	0	0	0
			1	1	0	1	
		1	1	0	1		
+	0	0	0	0			
	1	0	0	1	1	1	0

Binary multiplication

			1	1	0	1	Multiplicand
		×	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.
  - The product can have up to 8 bits, but we can use four-bit adders if we stagger them leftwards, like the partial products themselves.

2×2 binary multiplication

- Here is an outline of multiplying the two-bit numbers A1A0 and B1B0, to produce the four-bit product P3-P0.

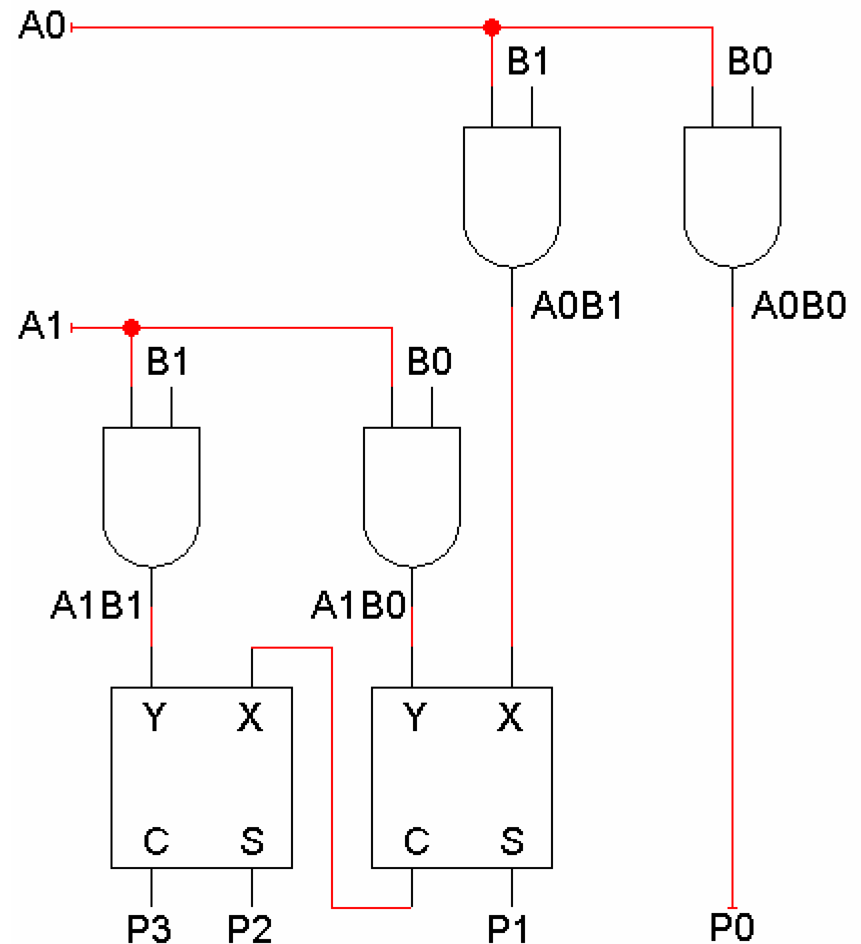
			B1	B0
		×	A1	A0
			A0B1	A0B0
+		A1B1	A1B0	
	P3	P2	P1	P0

- The bits of each partial product are computed by multiplying two bits of the input.
- Since two-bit multiplication is the same as the logical AND operation, we can use AND gates to generate the partial products.

A	B	A×B	A•B
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

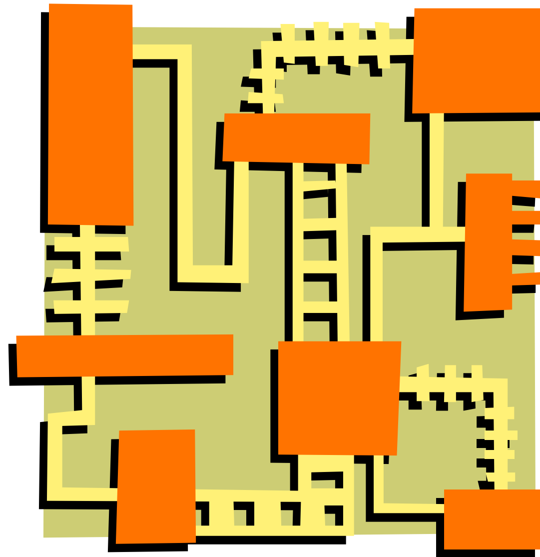
A 2×2 binary multiplier

- Here is a circuit that multiplies the two-bit numbers A1A0 and B1B0, resulting in the four-bit product P3-P0.
- For a 2×2 multiplier we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- The diagram on the next page shows how this can be extended to a four-bit multiplier, taking inputs A3-A0 and B3-B0 and outputting the product P7-P0.



A 4×4 binary multiplier

Complexity of multiplication circuits



- In general, when multiplying an m-bit number by an n-bit number:
  - There will be n partial products, one for each bit of the multiplier.
  - This requires n-1 adders, each of which can add m bits.
- The circuit for 32-bit or 64-bit multiplication would be huge!

Shifty arithmetic operations

- 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 a similar thing in binary. Shifting left once multiplies by two.

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

- Shifting left twice is equivalent to multiplying by four.

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

- Similarly, shifting once to the *right* is equivalent to *dividing* by two.

$$1100 / 10 = 100 \qquad \text{(in decimal, } 12 / 2 = 6\text{)}$$

Addition and multiplication summary

- Adder and multiplier circuits reflect human algorithms for addition and multiplication.
- Adders and multipliers are built hierarchically.
  - We start with half adders and full adders and work our way up.
  - Building these circuits from scratch using truth tables and K-maps would be pretty difficult.
- Adder circuits are limited in the number of bits that can be handled. An overflow occurs when a result exceeds this limit.
- There is a tradeoff between simple but slow ripple carry adders and more complex but faster carry lookahead adders.
- Multiplying and dividing by powers of two can be done with simple shifts.