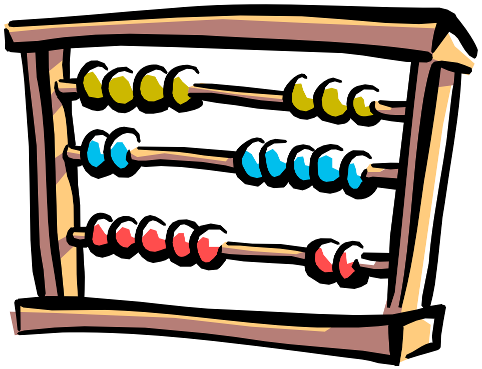
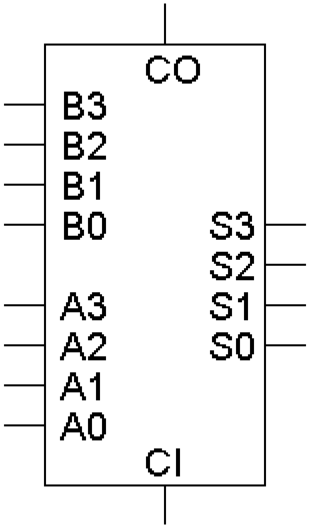
Arithmetic-Logic Units (ALUs)



The four-bit adder

- The basic four-bit adder *always* computes $S = A + B + CI$

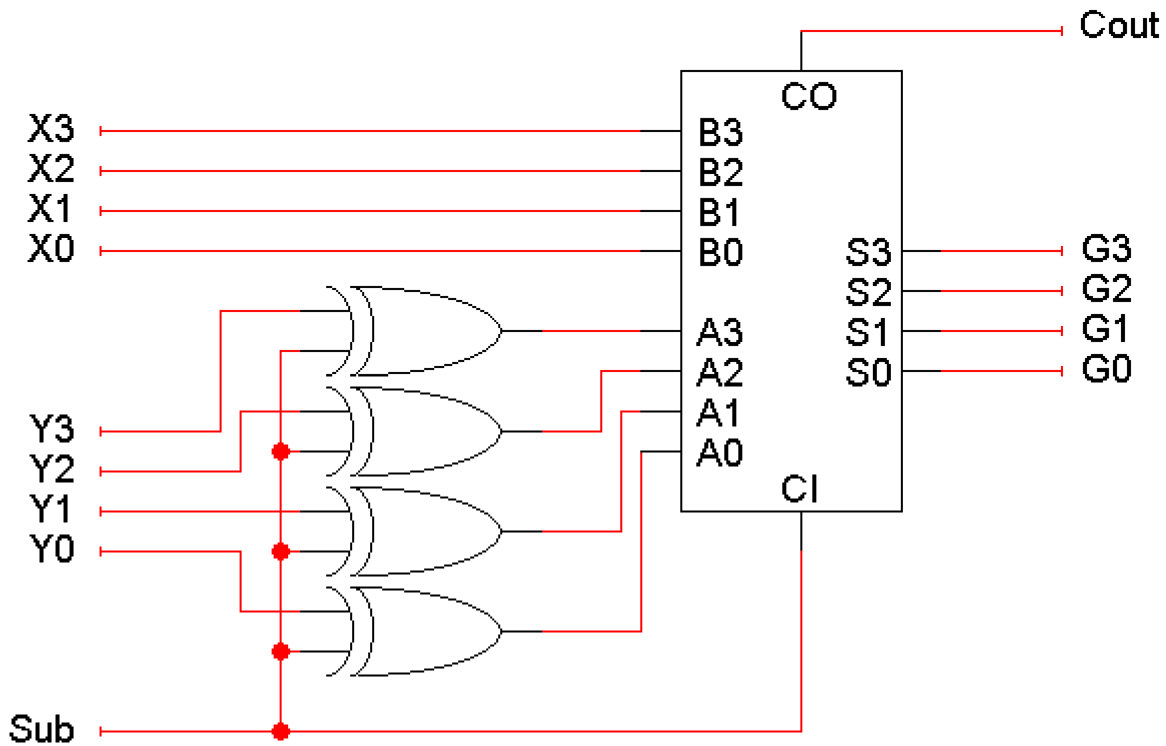


- But by changing what goes into the adder inputs A, B and CI, we can change the adder output S

- This is also what we did to build the combined adder-subtractor circuit

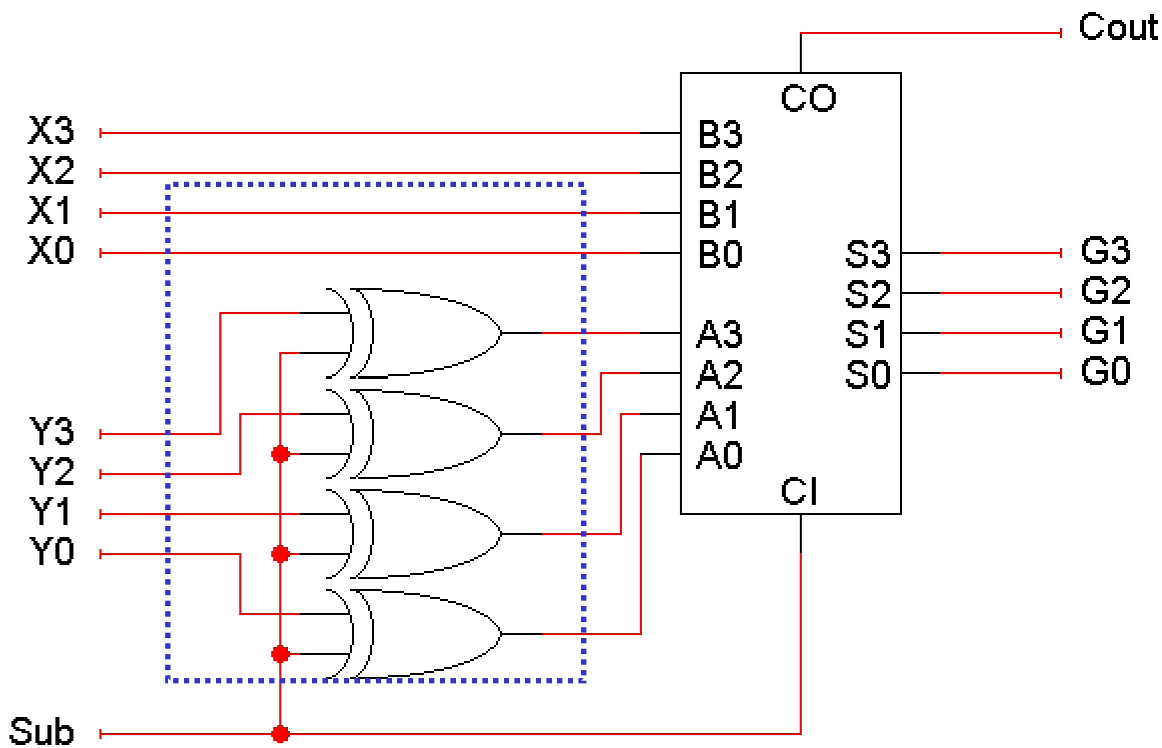
It's the adder-subtractor again!

- Here the signal Sub and some XOR gates alter the adder inputs
  - When Sub = 0, the adder inputs A, B, CI are Y, X, 0, so the adder produces G = X + Y + 0, or just X + Y
  - When Sub = 1, the adder inputs are Y', X and 1, so the adder output is G = X + Y' + 1, or the two's complement operation X - Y



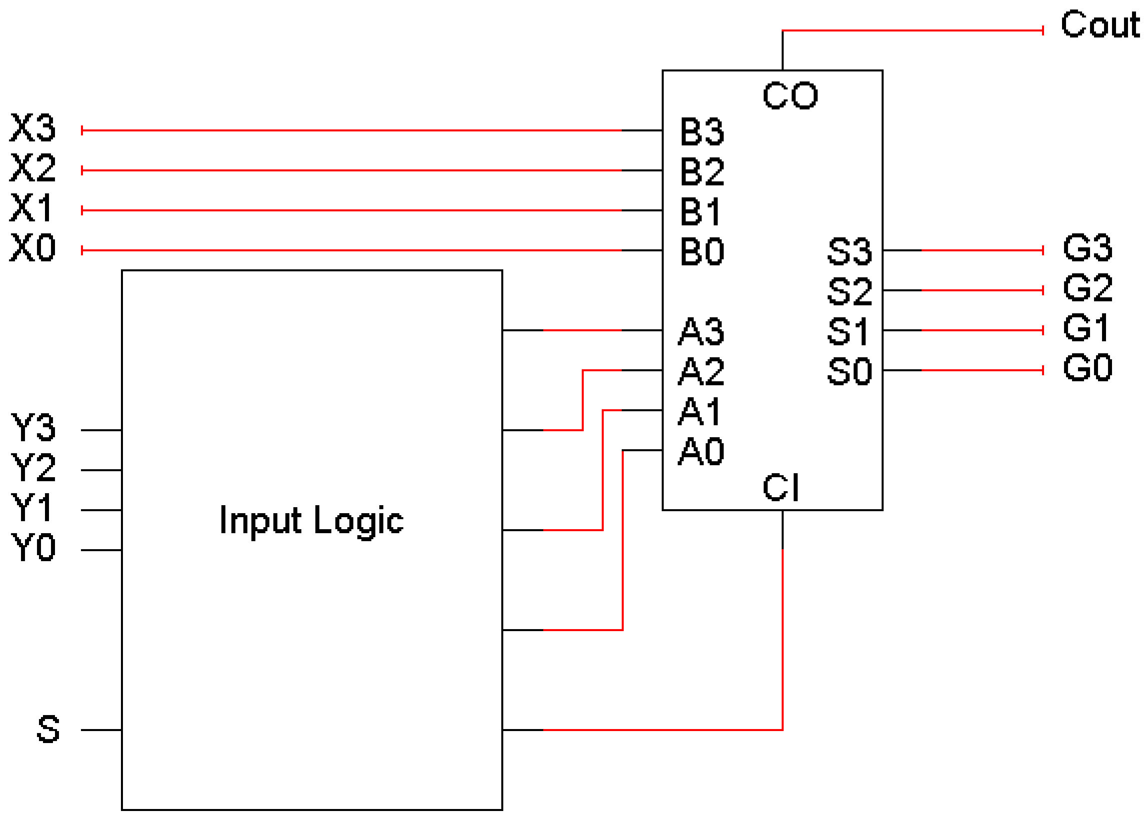
The multi-talented adder

- So we have one adder performing two separate functions
- "Sub" acts like a function select input which determines whether the circuit performs addition or subtraction
- Circuit-wise, all "Sub" does is modify the adder's inputs *A* and *CI*



Modifying the adder inputs

- By following the same approach, we can use an adder to compute *other* functions as well
- We just have to figure out which functions we want, and then put the right circuitry into the "Input Logic" box



Some more possible functions

- We already saw how to set adder inputs A, B and CI to compute either X + Y or X – Y
- How can we produce the increment function G = X + 1?

  One way: Set A = 0000, B = X, and CI = 1

- How about decrement: G = X - 1?

  A = 1111 (-1), B = X, CI = 0

- How about transfer: G = X?

  A = 0000, B = X, CI = 0

  This is almost the same as the increment function!

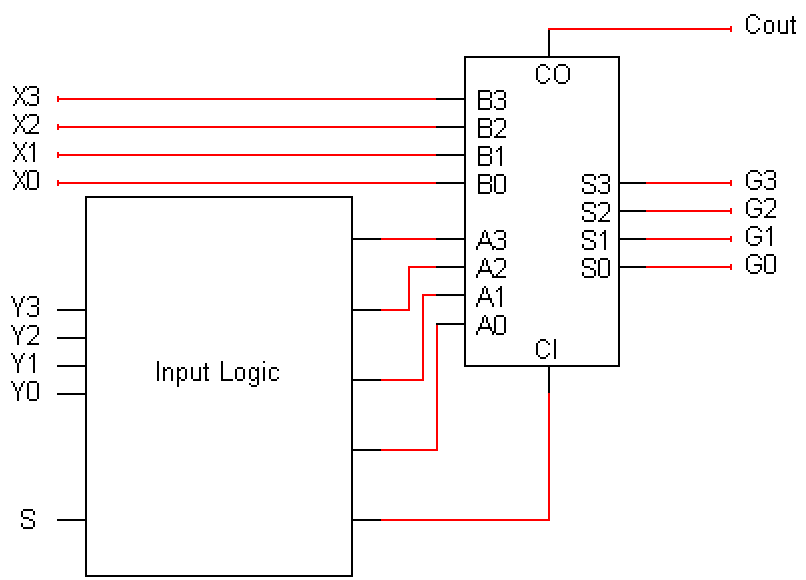


Table of arithmetic functions

- Here are some of the different possible arithmetic operations
- We'll need some way to specify which function we're interested in, so we've randomly assigned a selection code to each operation

S_2	S_1	S_0	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	$X + 1$	(increment)
0	1	0	$X + Y$	(add)
0	1	1	$X + Y + 1$	
1	0	0	$X + Y'$	(1C subtraction)
1	0	1	$X + Y' + 1$	(2C subtraction)
1	1	0	$X - 1$	(decrement)
1	1	1	X	(transfer)

Mapping the table to an adder

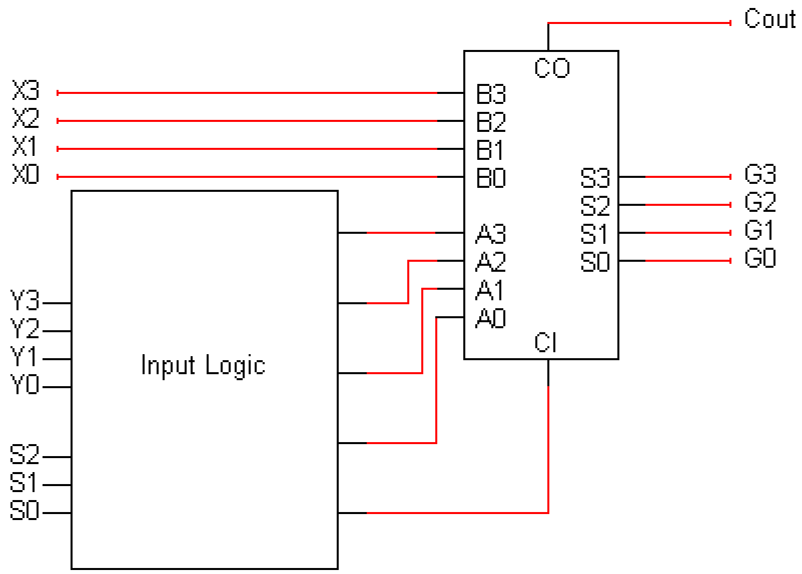
- This second table shows what the adder's inputs should be for each of our eight desired arithmetic operations
  - Adder input CI is always the same as selection code bit S_0
  - B is always set to X
  - A depends only on S_2 and S_1
- These equations depend on both the desired operations and the assignment of selection codes

Selection code			Desired arithmetic operation		Required adder inputs		
S_2	S_1	S_0	G (A + B + CI)		A	B	CI
0	0	0	X	(transfer)	0000	X	0
0	0	1	X + 1	(increment)	0000	X	1
0	1	0	X + Y	(add)	Y	X	0
0	1	1	X + Y + 1		Y	X	1
1	0	0	X + Y'	(1C subtraction)	Y'	X	0
1	0	1	X + Y' + 1	(2C subtraction)	Y'	X	1
1	1	0	X – 1	(decrement)	1111	X	0
1	1	1	X	(transfer)	1111	X	1

Building the input logic

- All we need to do is compute the adder input A, given the arithmetic unit input Y and the function select code S (actually just S_2 and S_1)
- Here is an abbreviated truth table:

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111



- We want to pick one of these four possible values for A, depending on S_2 and S_1

Primitive gate-based input logic

- We could build this circuit using primitive gates
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table
  - The Y that appears in the output column (A) is actually an input
  - We make that explicit in the table on the right
- Remember A and Y are each 4 bits long!

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111

S_2	S_1	Y_i	A_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

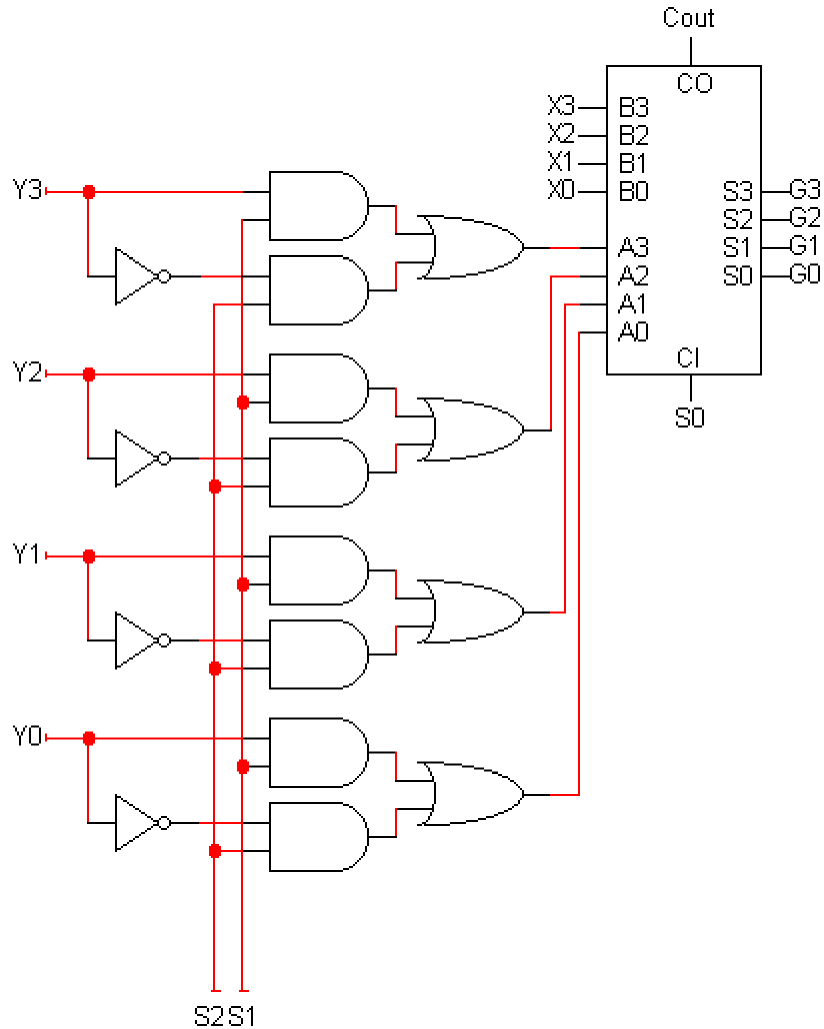
Primitive gate implementation

- From the truth table, we can find an MSP:

			S_1	
	0	0	1	0
S_2	1	0	1	1
		Y_i	Y_i	

$$A_i = S_2 Y_i' + S_1 Y_i$$

- Again, we have to repeat this once for each bit Y3-Y0, connecting to the adder inputs A3-A0
- This completes our arithmetic unit



Bitwise operations

- Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit words instead of just single bits
- To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:

	AND		OR		XOR
	1 0 1 1		1 0 1 1		1 0 1 1
AND	1 1 1 0	OR	1 1 1 0	XOR	1 1 1 0
	1 0 1 0		1 1 1 1		0 1 0 1

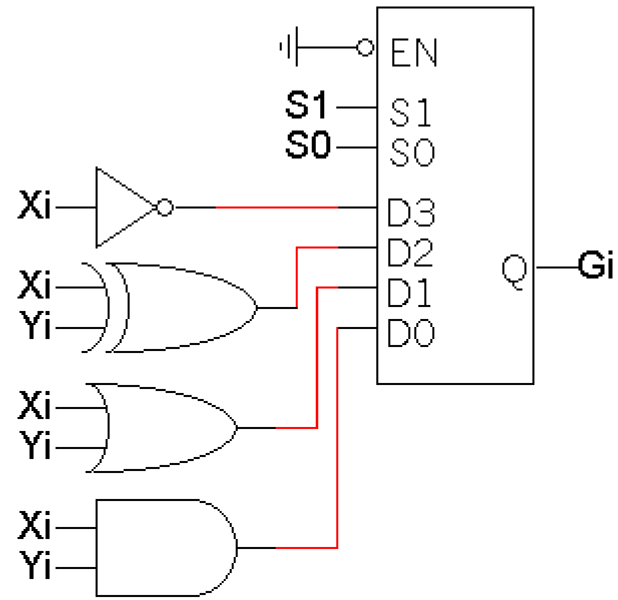
- We've already seen this informally in two's-complement arithmetic, when we talked about "complementing" all the bits in a number

Defining a logic unit

- A logic unit supports different logical functions on two multi-bit inputs X and Y, producing an output G

- This abbreviated table shows four possible functions and assigns a selection code S to each

S_1	S_0	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$

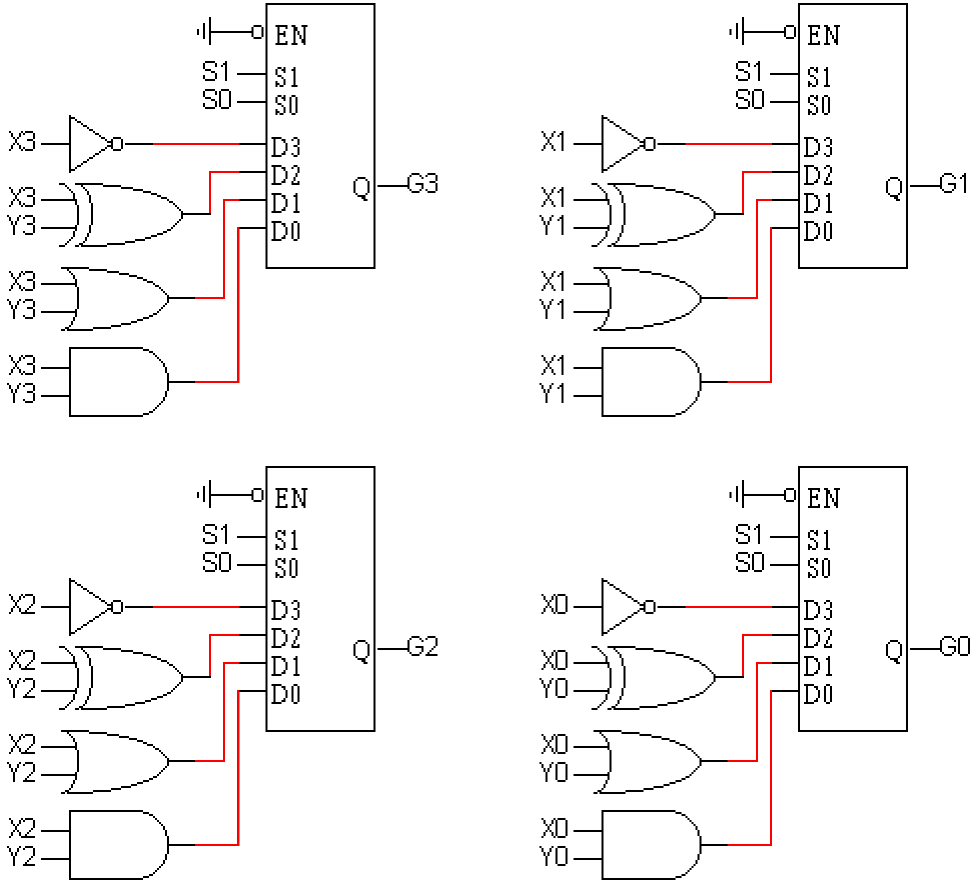


- We'll just use multiplexers and some primitive gates to implement this

- Again, we need one multiplexer for *each bit* of X and Y

Our simple logic unit

- Inputs:
  - X (4 bits)
  - Y (4 bits)
  - S (2 bits)
- Outputs:
  - G (4 bits)



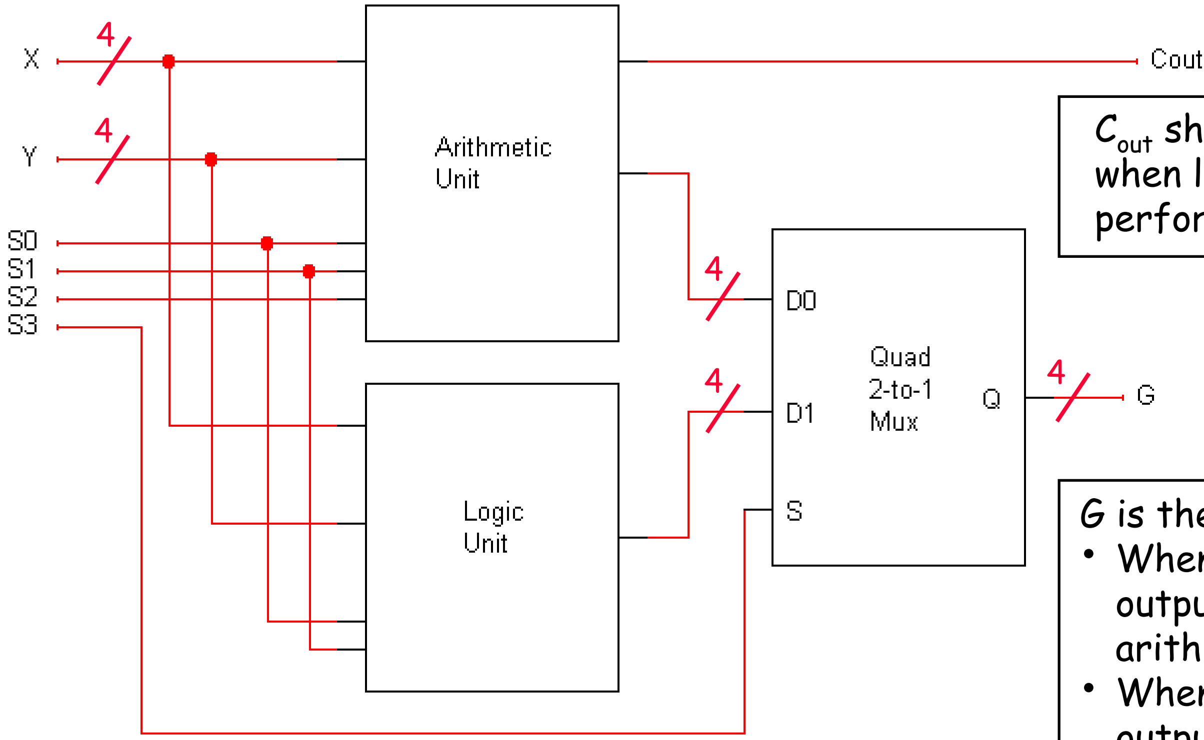
Our ALU function table

- This table shows a sample function table for an ALU
- All of the arithmetic operations have S_3=0, and all of the logical operations have S_3=1
- These are the same functions we saw when we built our arithmetic and logic units a few minutes ago
- Since our ALU only has 4 logical operations, we don't need S_2. The operation done by the logic unit depends only on S_1 and S_0

S_3	S_2	S_1	S_0	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X$ and Y
1	x	0	1	$G = X$ or Y
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

A complete ALU circuit

The / and 4 on a line indicate that it's actually *four* lines.



C_{out} should be ignored when logic operations are performed (when S3=1).

G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.

The completed ALU

- This ALU is a good example of hierarchical design
  - With the 12 inputs, the truth table would have had $2^{12} = 4096$ lines. That's an awful lot of paper.
  - Instead, we were able to use components that we've seen before to construct the entire circuit from a couple of easy-to-understand components
- As always, we encapsulate the complete circuit in a "black box" so we can reuse it in fancier circuits.