



College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 10

Arithmetic-logic units

Remember:

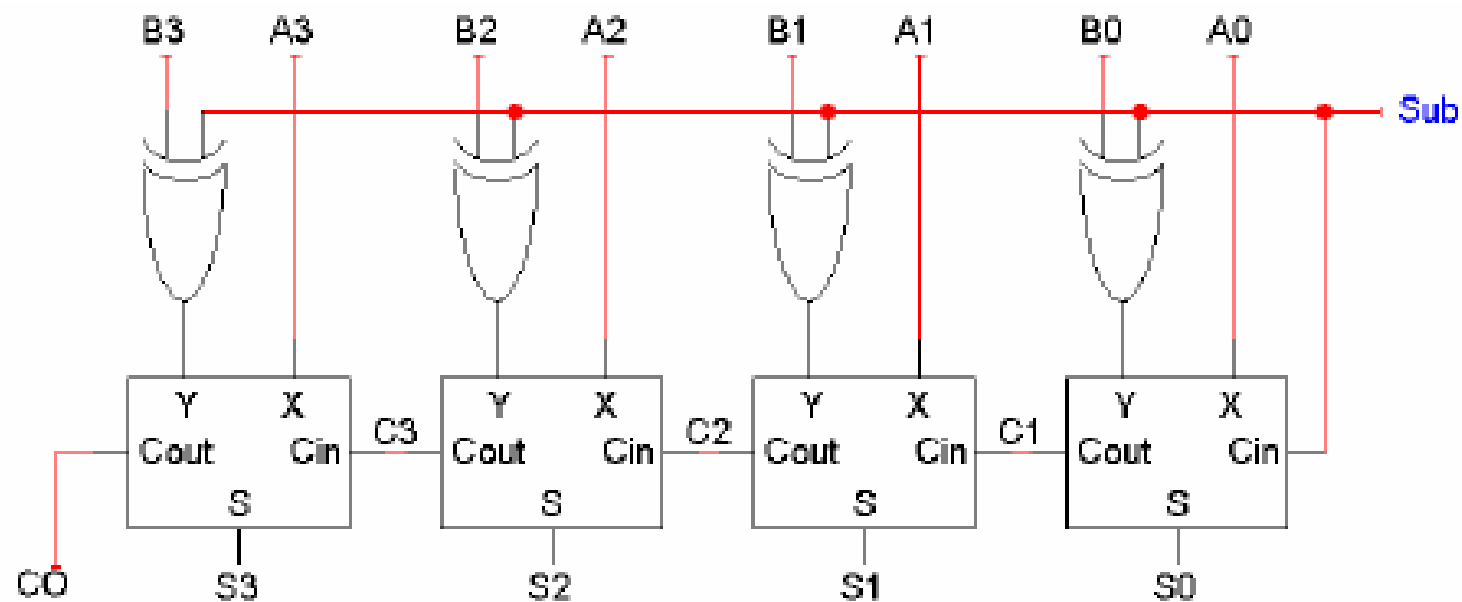
An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

$$X \oplus 0 = X$$

$$X \oplus 1 = X'$$

- When $Sub = 0$, the XOR gates output $B_3 B_2 B_1 B_0$ 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 $B_3' B_2' B_1' B_0'$ and the carry in is 1. Thus, the adder output will be a two's complement subtraction, $A - B$.

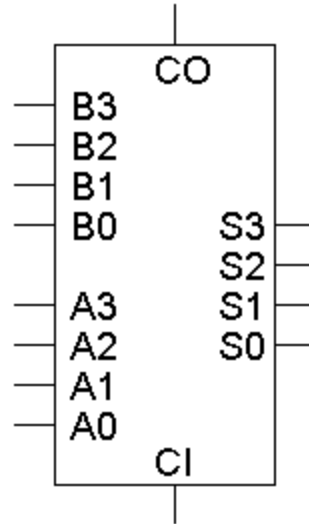


Arithmetic-logic units

- **An arithmetic-logic unit, or ALU, performs many different arithmetic and logic operations. The ALU is the “heart” of a processor—you could say that everything else in the CPU is there to support the ALU.**
- **Here’s the plan:**
 - We’ll show an arithmetic unit first, by building off ideas from the adder-subtractor circuit.
 - Then we’ll talk about logic operations a bit, and build a logic unit.
 - Finally, we put these pieces together using multiplexers.
- **We show the same examples as from the book.**

The four-bit adder

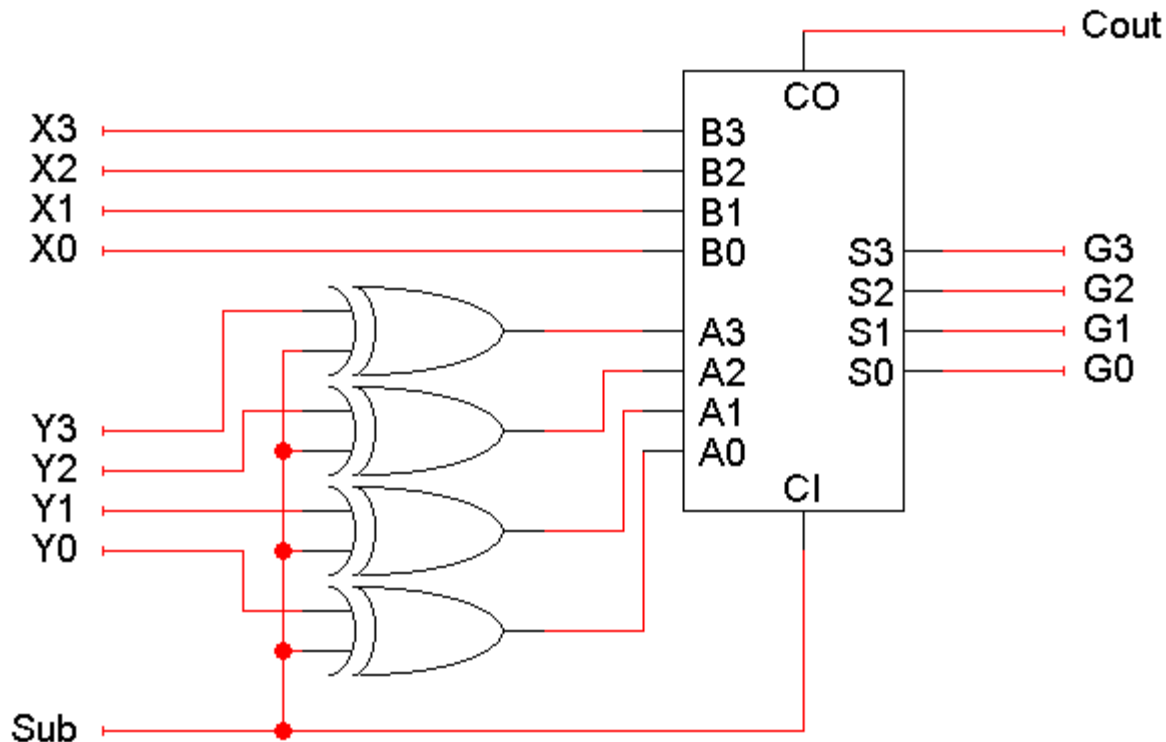
- The basic four-bit adder *always* computes $S = A + B + CI$.



- 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.

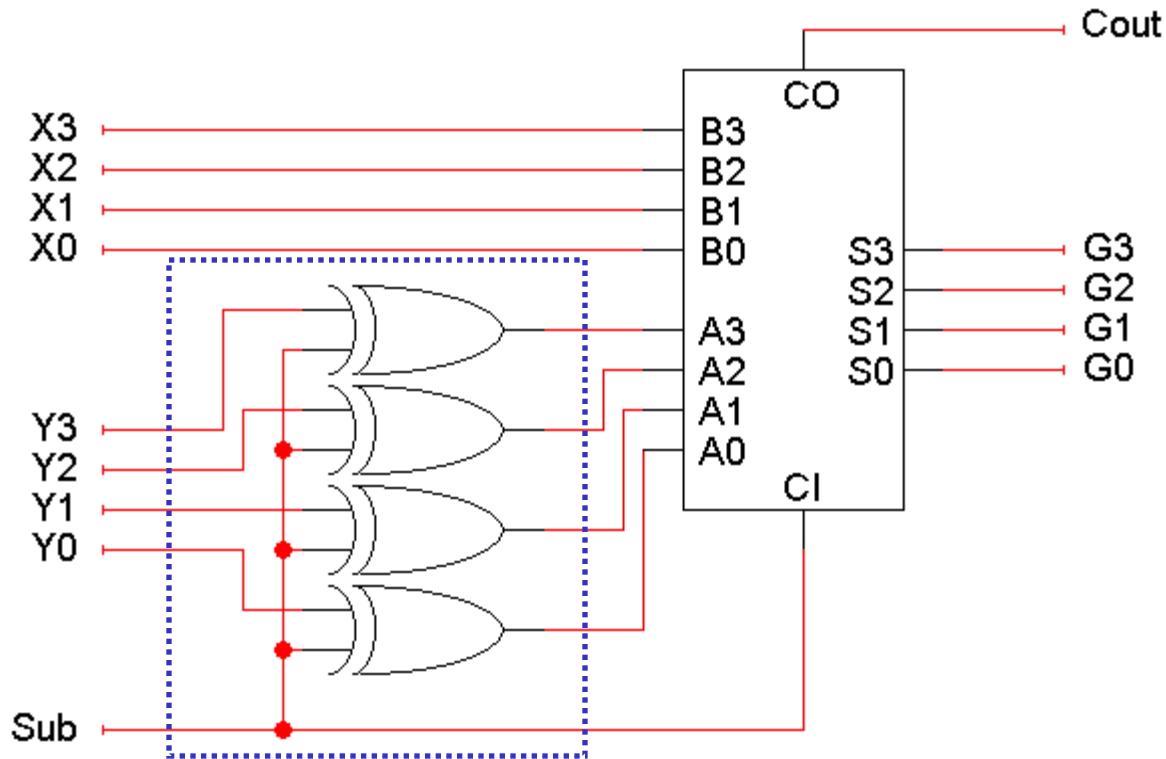
It's the adder-subtractor **again!**

- 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.



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 $CI = 1$

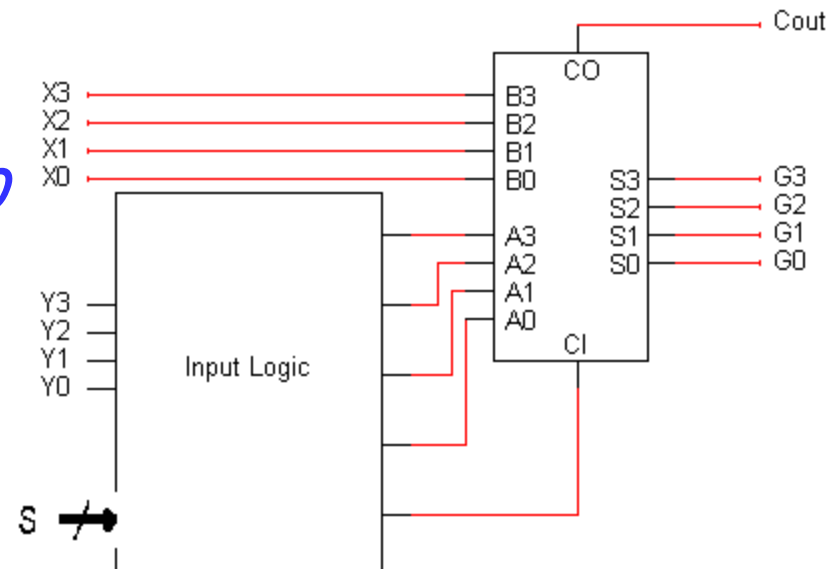
- How about decrement: $G = X - 1$?

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

- How about transfer: $G = X$?
(This can be useful.)

$A = 0000$, $B = X$, $CI = 0$

This is almost the same as the increment function!



The role of CI

- The transfer and increment operations have the same A and B inputs, and differ only in the CI input.
- In general we can get additional functions (not all of them useful) by using both $CI = 0$ and $CI = 1$.
- Another example:
 - Two's-complement subtraction is obtained by setting $A = Y'$, $B = X$, and $CI = 1$, so $G = X + Y' + 1$.
 - If we keep $A = Y'$ and $B = X$, but set CI to 0 , we get $G = X + Y'$. This turns out to be a ones' complement subtraction operation.

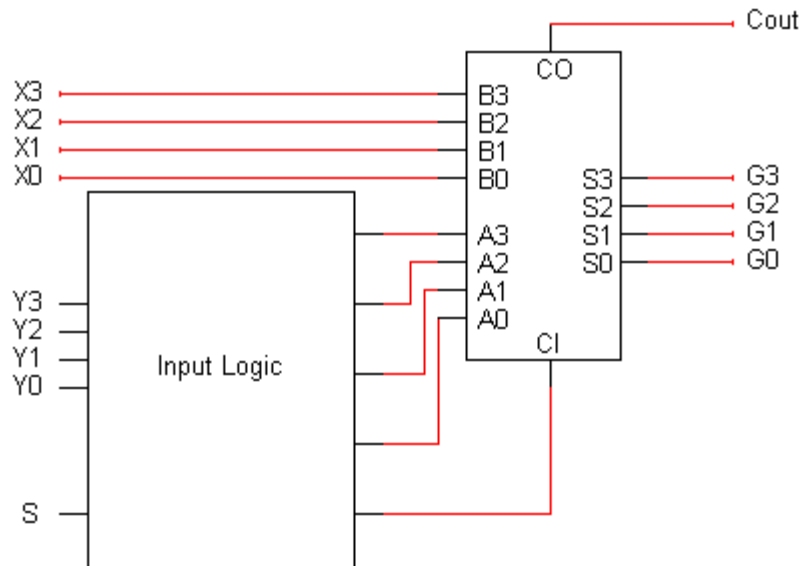
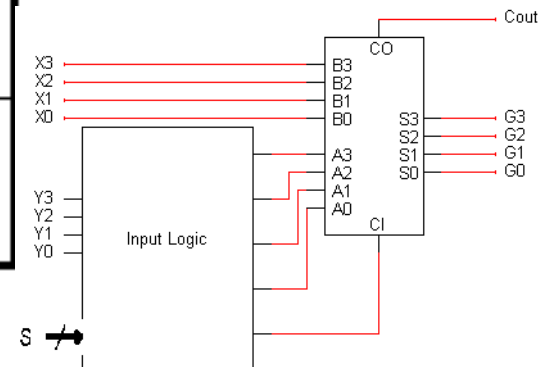


Table of arithmetic functions

- 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.

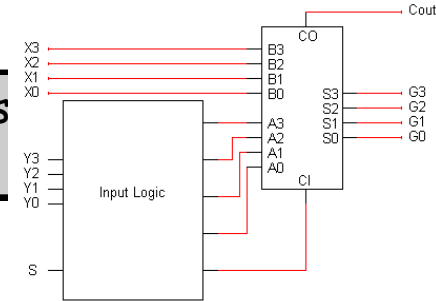
S_2	S_1	S_0	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	$X + 1$	(increment)
0	1	0	$X + Y$	(add)
0	1	1	$X + Y + 1$	
1	0	0	$X + Y'$	(1C subtraction)
1	0	1	$X + Y' + 1$	(2C subtraction)
1	1	0	$X - 1$	(decrement)
1	1	1	X	(transfer)



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.

Selection code			Desired arithmetic operation $G(A + B + CI)$	Required adder inputs		
S_2	S_1	S_0		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

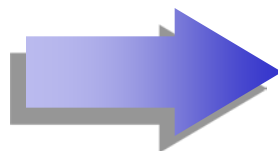


- Adder input CI is always the same as selection code bit S_0 .
 - B is always set to X.
 - A depends only on S_2 and S_1 .
- These equations depend on both the desired operations and the assignment of selection codes.

Building the input logic

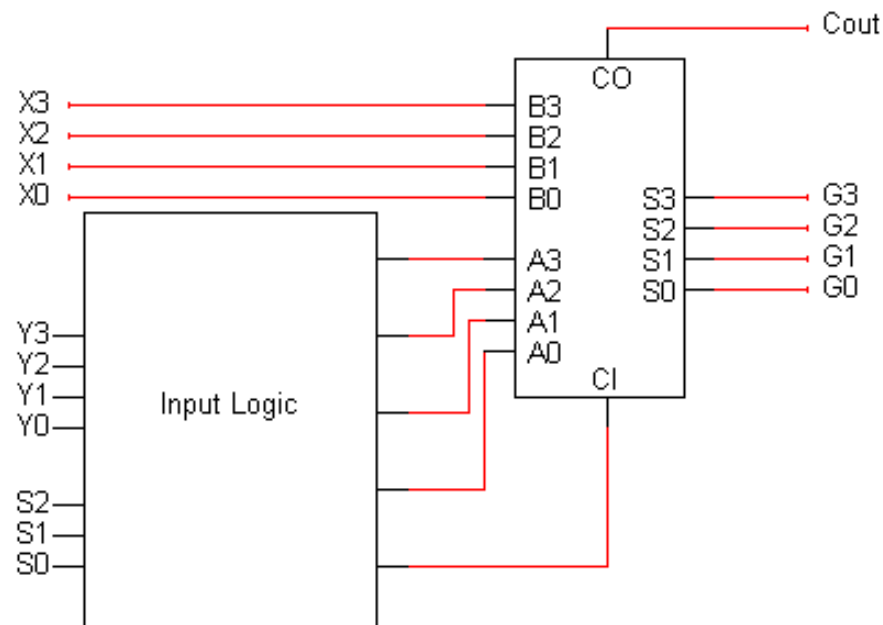
- All we need to do is compute the adder input A , given the arithmetic unit input Y and the function select code S (actually just S_2 and S_1).
- Here is an abbreviated truth table:

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



inputs output

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

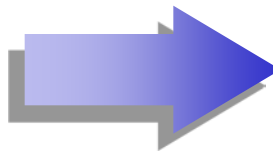


- We want to pick one of these four possible values for A , depending on S_2 and S_1 .

Primitive gate-based input logic

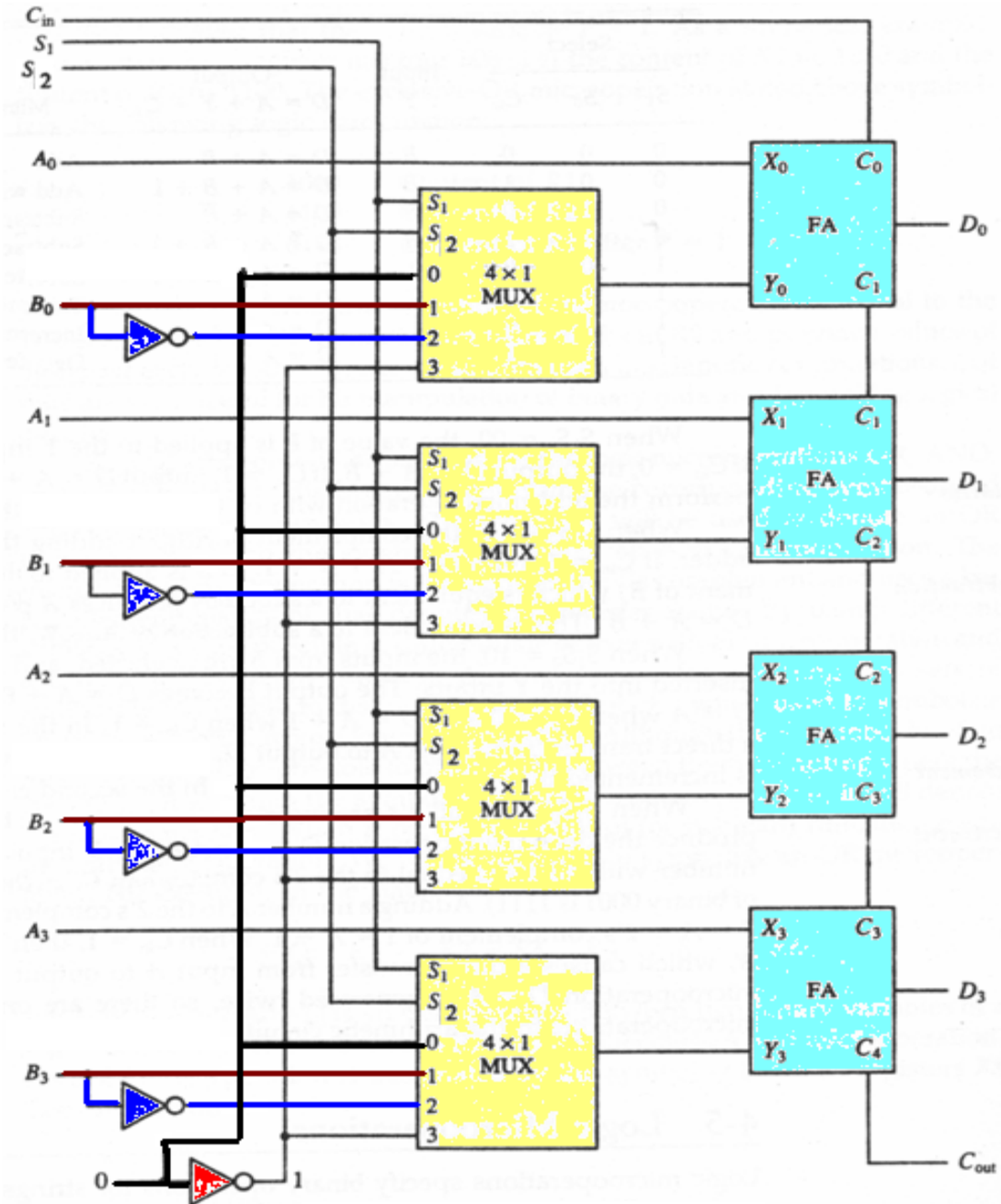
- We could build this circuit using primitive gates.
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table.
 - The Y that appears in the output column (A) is actually an input.
 - We make that explicit in the table on the right.
- Remember A and Y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

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



inputs			output
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

Selection code			Desired arithmetic operation $D = X + Y + Cin$	Required adder inputs		
S_2	S_1	S_0		X	Y	Cin
0	0	0	A (transfer)	0000	X	0
0	0	1	$A+1$ (increment)	0000	X	1
0	1	0	$A+B$ (add)	B	X	0
0	1	1	$A+B+1$	B	X	1
1	0	0	$A+B'$ (1C subtraction)	B'	X	0
1	0	1	$A+B'+1$ (2C subtraction)	B'	X	1
1	1	0	$A-1$ (decrement)	1111	X	0
1	1	1	A (transfer)	1111	X	1



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

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 :

$$\begin{array}{r} \text{AND} \quad 1\ 0\ 1\ 1 \\ \quad 1\ 1\ 1\ 0 \\ \hline \quad 1\ 0\ 1\ 0 \end{array}$$

$$\begin{array}{r} \text{OR} \quad 1\ 0\ 1\ 1 \\ \quad 1\ 1\ 1\ 0 \\ \hline \quad 1\ 1\ 1\ 1 \end{array}$$

$$\begin{array}{r} \text{XOR} \quad 1\ 0\ 1\ 1 \\ \quad 1\ 1\ 1\ 0 \\ \hline \quad 0\ 1\ 0\ 1 \end{array}$$

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

Bitwise operations in programming

- Languages like C, C++ and Java provide bitwise logical operations:

& (AND) **|** (OR) **^** (XOR) **~** (NOT)

- These operations treat each integer as a bunch of individual bits:

13 & 25 = 9 because **01101 & 11001 = 01001**

- They are *not* the same as the operators **&&**, **||** and **!**, which treat each integer as a single logical value (0 is false, everything else is true):

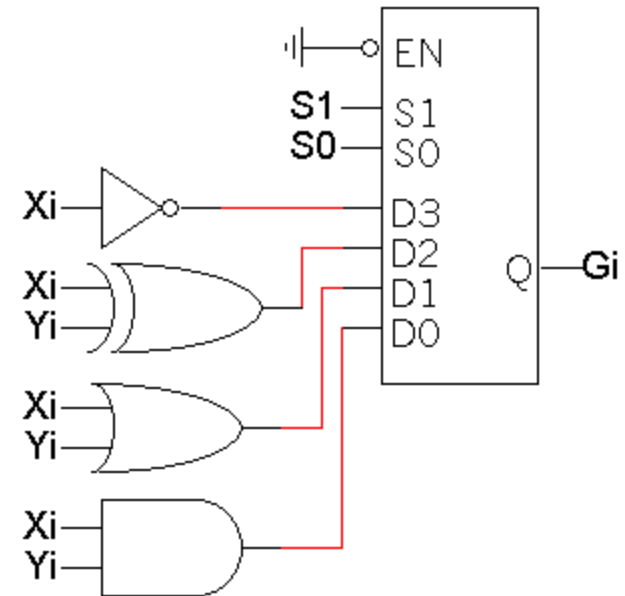
13 && 25 = 1 because **true && true = true**

- Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.
- Easy to represent sets of fixed universe size with bits:
 - **1**: is member, **0** not a member. Unions: **OR**, Intersections: **AND**

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.

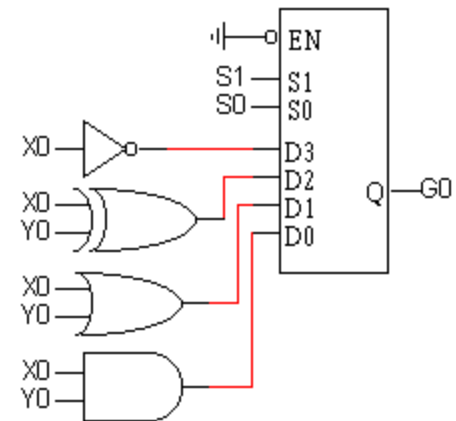
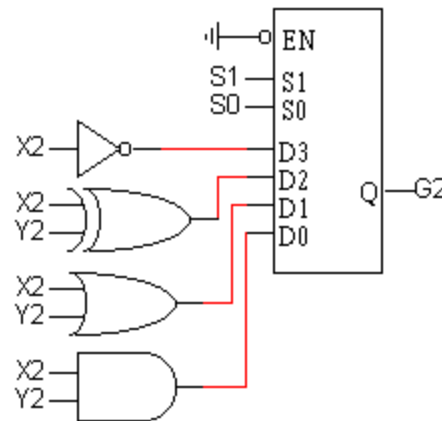
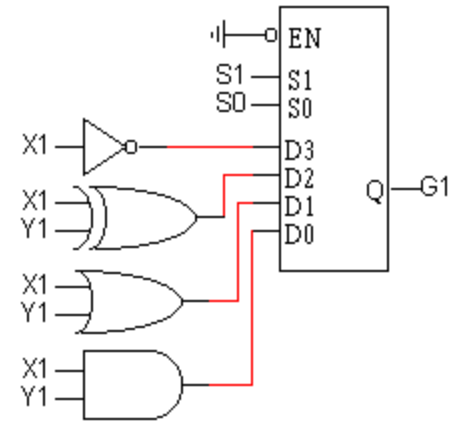
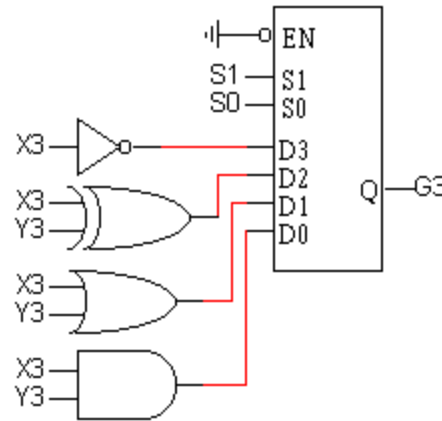
S_1	S_0	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$



- 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)**



Combining the arithmetic and logic units

- Now we have two pieces of the puzzle:
 - **An arithmetic unit that can compute eight functions on 4-bit inputs.**
 - **A logic unit that can perform four functions on 4-bit inputs.**
- We can combine these together into a single circuit, an arithmetic-logic unit (ALU).

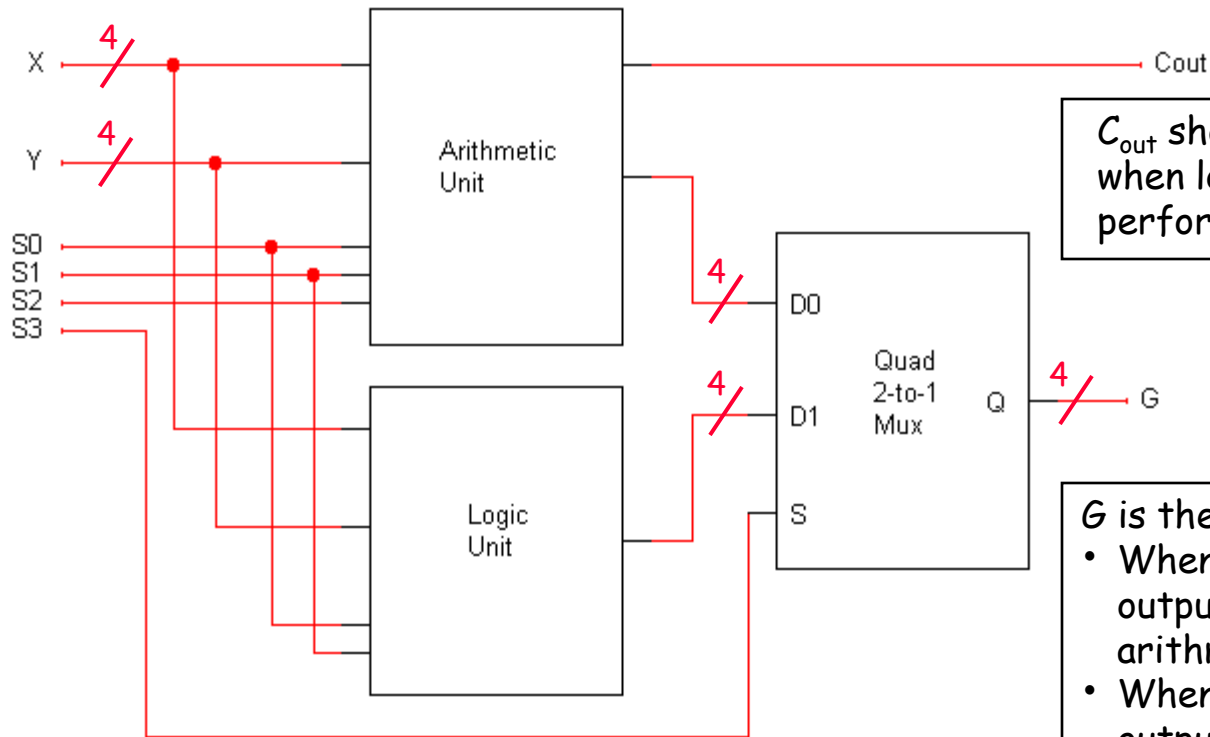
Our ALU function table

- 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	$G = X \text{ and } Y$
1	x	0	1	$G = X \text{ or } Y$
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 $S2$.

Comments on the multiplexer

- ***Both* the arithmetic unit and the logic unit are “active” and produce outputs.**
 - **The mux determines whether the final result comes from the arithmetic or logic unit.**
 - **The output of the other one is effectively ignored.**
- **Our hardware scheme may seem like wasted effort, but it’s not really.**
 - **“Deactivating” one or the other wouldn’t save that much time.**
 - **We have to build hardware for both units anyway, so we might as well run them together.**
- **This is a very common use of multiplexers in logic design.**

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.**

