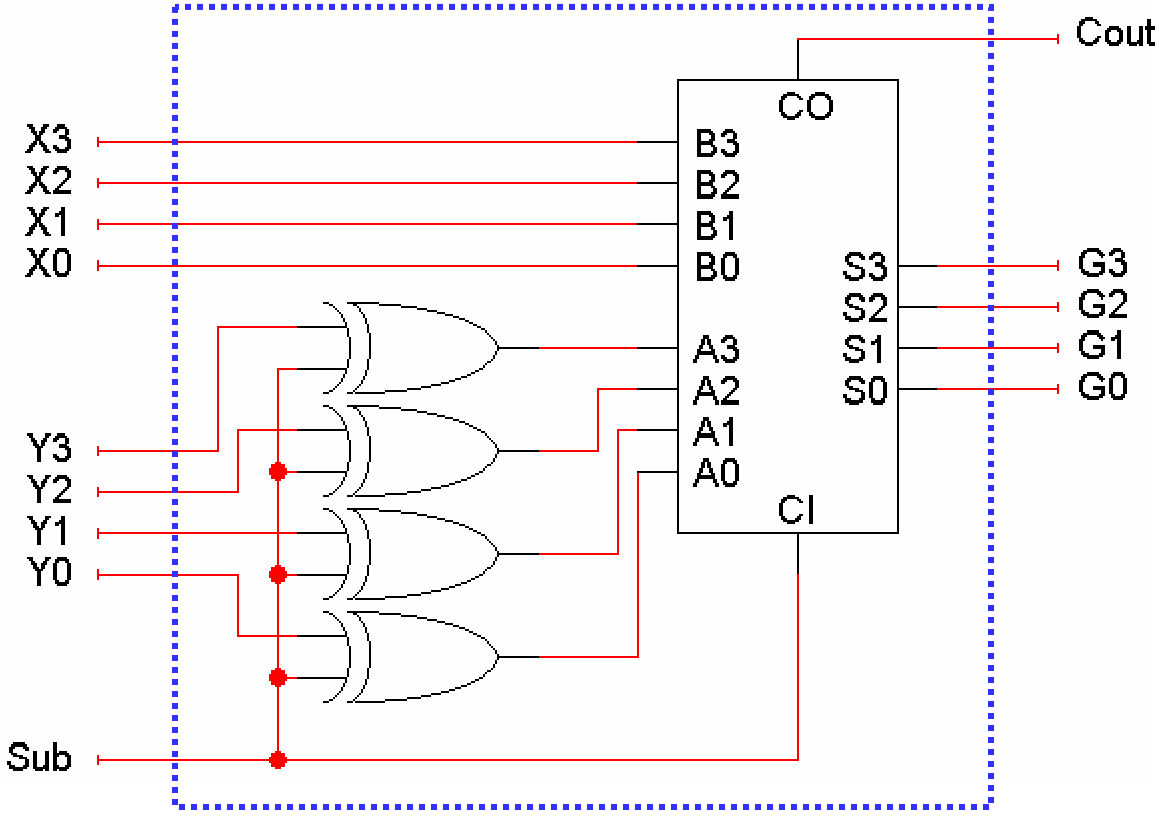
# Arithmetic-logic units

- An arithmetic-logic unit or ALU performs many different arithmetic and logical operations. The ALU is at the "heart" of a processor—you could say that everything else in the computer is there to support the ALU.
- Today we'll see how you can build your own ALU.
  - First we explain how several basic operations can be implemented using just an unsigned adder.
  - Then we'll talk about logical operations and how to build a corresponding circuit.
  - Finally, we'll put these two pieces together.
- We show the same examples as the book (pp. 360-365), but things are re-labelled to be clearer in LogicWorks. Some inputs (CI and S0) are also treated differently.

# It’s the adder-subtractor again!

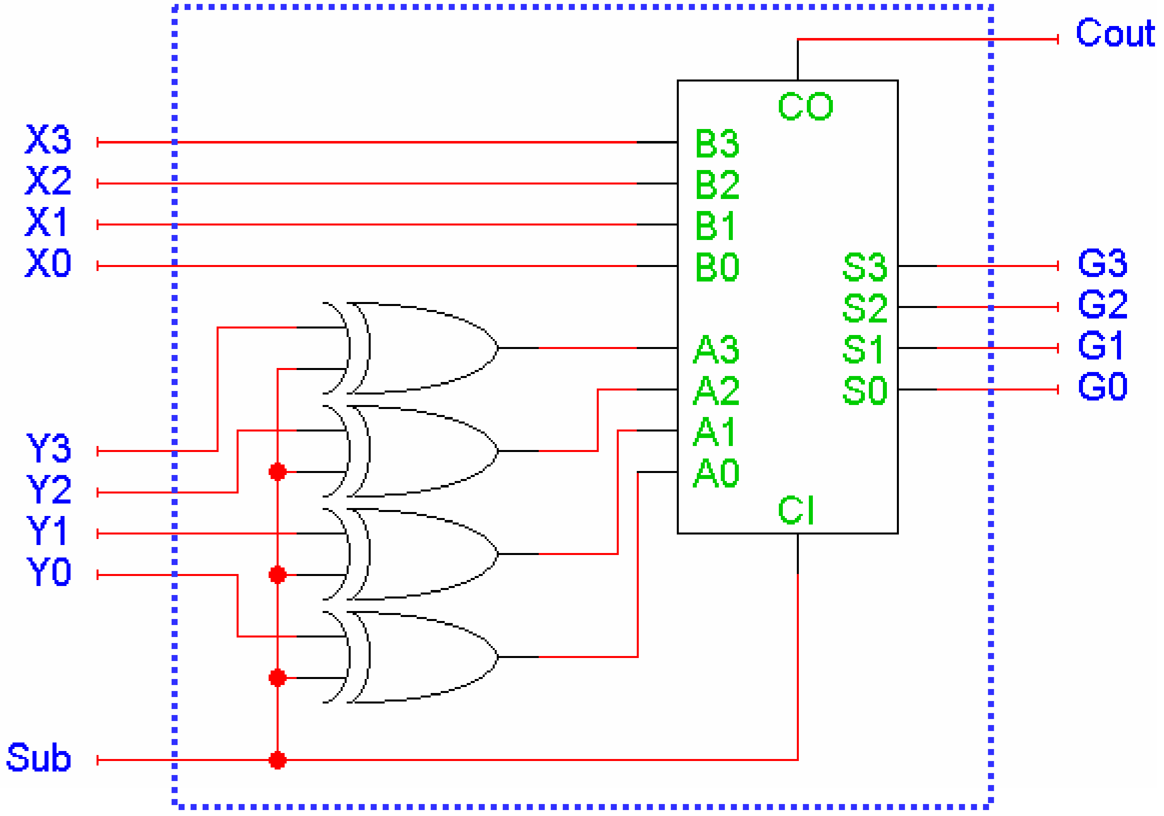
- An arithmetic unit is a circuit that supports several different arithmetic operations.
- We’ve already seen a simple arithmetic unit that supports two functions—addition and subtraction.
- That circuit has two four-bit data inputs X and Y, and a function selection input Sub.
- The four-bit output G is either X + Y or X – Y, depending on the value of the Sub input.



| Sub | G |
|---|---|
| 0 | X + Y |
| 1 | X – Y |

# Hierarchical circuit design

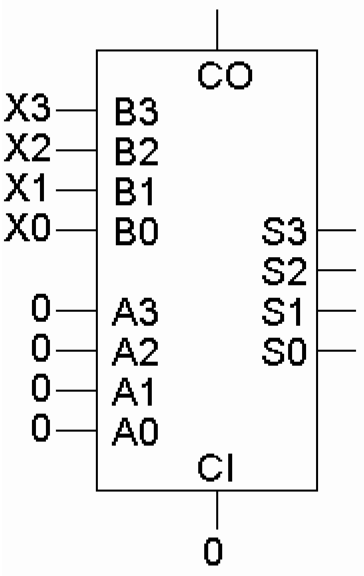
- This circuit is based on a four-bit unsigned adder, which *always* outputs the sum of its inputs, S = A + B + CI.
- To perform addition or subtraction, all we did was vary adder inputs A, B and CI according to the arithmetic unit inputs X, Y and Sub.
- The output G of the arithmetic unit comes right out of the adder.



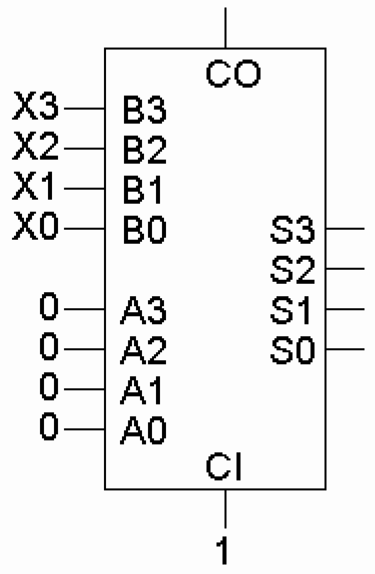
| Sub | Adder inputs A | B | CI | Adder output S |
|---|---|---|---|---|
| 0 | Y | X | 0 | X + Y |
| 1 | Y’ | X | 1 | X – Y |

# Some more possible functions

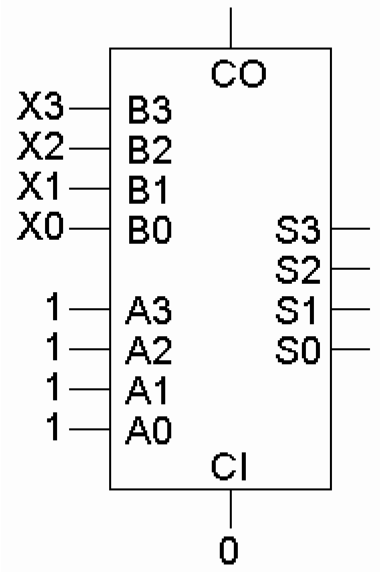
- We can follow the same approach to implement other functions with our four-bit unsigned adder as well!



S = X (transfer)



S = X + 1 (increment)



S = X – 1 (decrement)

| A | B | CI | S | |
|---|---|---|---|---|
| 0 | X | 0 | X | (transfer) |
| 0 | X | 1 | X + 1 | (increment) |
| -1 | X | 0 | X – 1 | (decrement) |

# The role of CI

- Notice that for the transfer and increment operations, the adder has the same A and B inputs, and only the CI input differs.
- In general we can always create additional functions by setting CI = 0 instead of CI = 1, and vice versa.
- Another example involves subtraction.
  - We already know that two's complement subtraction is performed by setting A = Y', B = X and CI = 1, so the adder outputs X + Y' + 1.
  - If we keep A = Y' and B = X, but set CI to 0 instead, we get the output X + Y'. This turns out to be a *ones' complement* subtraction.

# Table of arithmetic functions

- Here are the different arithmetic operations we've seen so far, and how they can be implemented by an unsigned four-bit adder.
- There are actually just four basic operations, but by setting CI to both 0 and 1 we come up with eight operations.
- Notice that the transfer function can be implemented in two ways, and appears twice in this table.

| Arithmetic operation A + B + CI | | Required adder inputs A | B | CI |
|---|---|---|---|---|
| X | (transfer) | 0000 | X | 0 |
| X + 1 | (increment) | 0000 | X | 1 |
| X + Y | (add) | Y | X | 0 |
| X + Y + 1 | | Y | X | 1 |
| X + Y' | (1C subtraction) | Y' | X | 0 |
| X + Y' + 1 | (2C subtraction) | Y' | X | 1 |
| X – 1 | (decrement) | 1111 | X | 0 |
| X | (transfer) | 1111 | X | 1 |

# Selection codes

- We can make a circuit that supports all eight of these functions, using just a single unsigned adder.
- First, we must assign a three-bit selection code S for each operation, so we can specify which function should be computed.

| Selection code | | | Arithmetic operation | | Required adder inputs | | |
|---|---|---|---|---|---|---|---|
| S2 | S1 | S0 | 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 |

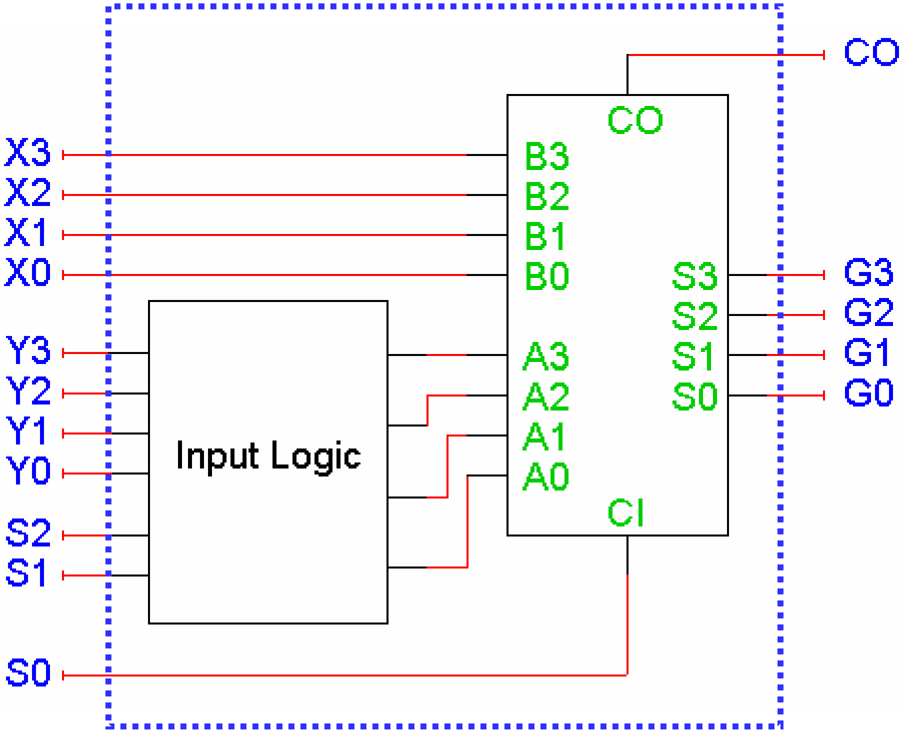
# Generating adder inputs

- The circuit for our arithmetic unit just has to generate the correct inputs to the adder (A, B and CI), based on the values of X, Y and S.
  - Adder input CI should always be the same as selection code bit S0.
  - The adder's input B is always equal to X.
  - Adder input A depends only on S2, S1 and Y.

| Selection code | | | Arithmetic operation | | Required adder inputs | | |
|---|---|---|---|---|---|---|---|
| S2 | S1 | S0 | 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

- Here is a diagram of our arithmetic unit so far.
- We've already set the adder inputs B to X and CI to S0, as explained on the previous page.
- All that's left is to generate adder input A from the arithmetic unit input Y and the function selection code bits S2 and S1.
- From the table on the last page, we can see that adder input A should be set to 0000, Y, Y' or 1111, depending on S2 and S1.



| S2 | S1 | A |
|---|---|---|
| 0 | 0 | 0000 |
| 0 | 1 | Y |
| 1 | 0 | Y' |
| 1 | 1 | 1111 |

# Primitive gate-based input logic

- We’ll build this circuit using primitive gates.
- If we want to use Karnaugh maps for simplification, then we should first expand the abbreviated truth table, since the Y in the output column is actually an input.
- Remember that A and Y are each four bits long! We are really describing *four* functions, A3, A2, A1 and A0, but they are each generated from Y3, Y2, Y1 and Y0 in the same way.

| S2 | S1 | A |
|---|---|---|
| 0 | 0 | 0000 |
| 0 | 1 | Y |
| 1 | 0 | Y’ |
| 1 | 1 | 1111 |

| S2 | S1 | $Y_i$ | $A_i$ |
|---|---|---|---|
| 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 |

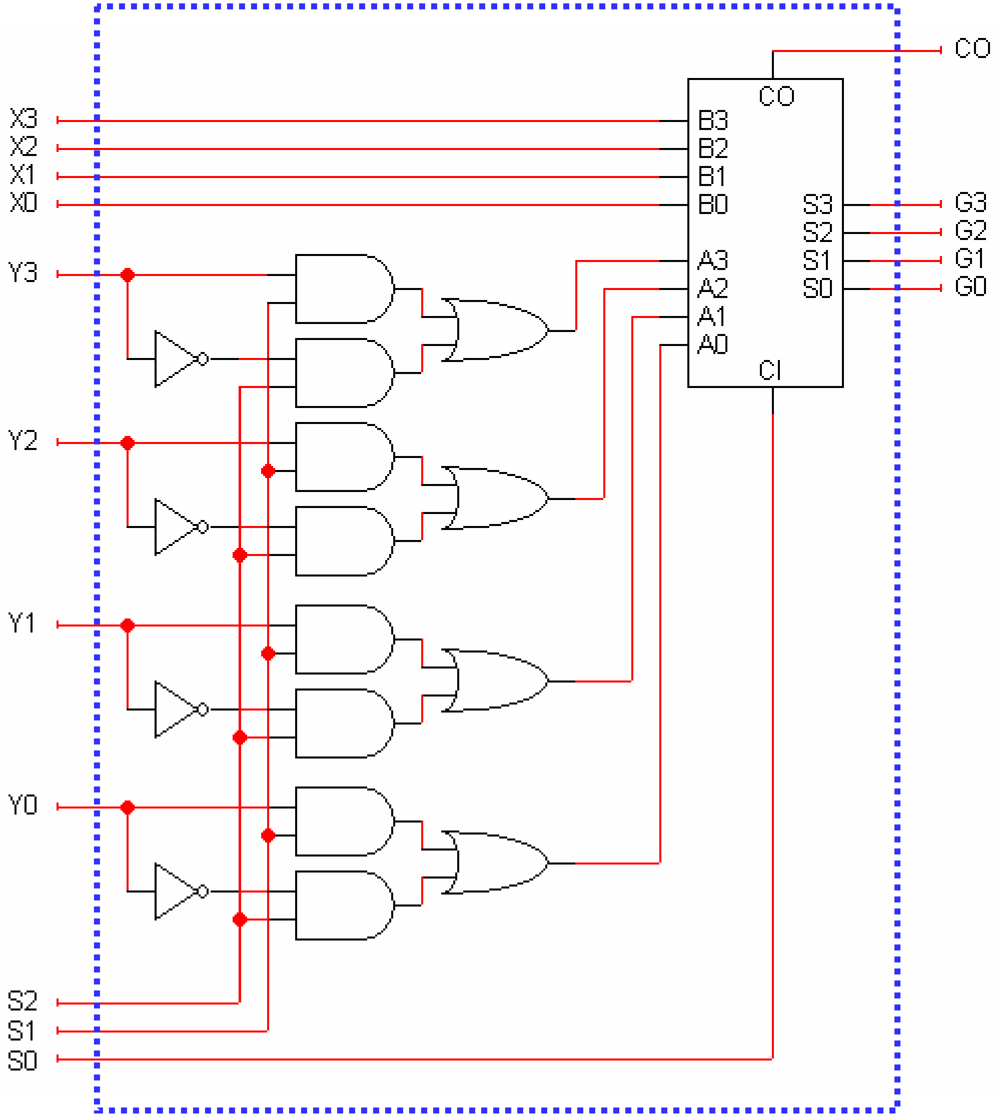
# Primitive gate implementation

- We can find a minimal sum of products from the truth table.

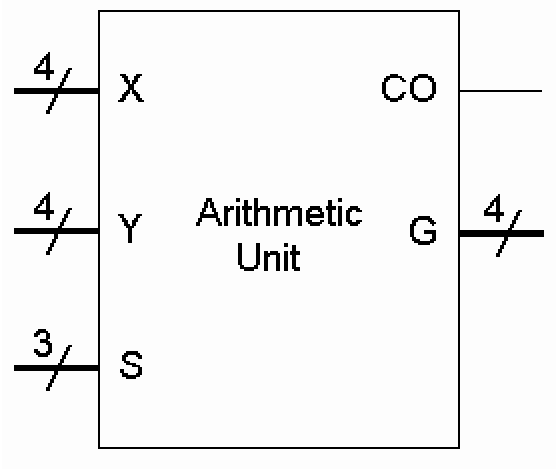
|  |  |  | S1 | S1 |
|---|---|---|---|---|
|  | 0 | 0 | 1 | 0 |
| S2 | 1 | 0 | 1 | 1 |
|  |  | $Y_i$ | $Y_i$ |  |

$A_i = S2\ Y_i' + S1\ Y_i$

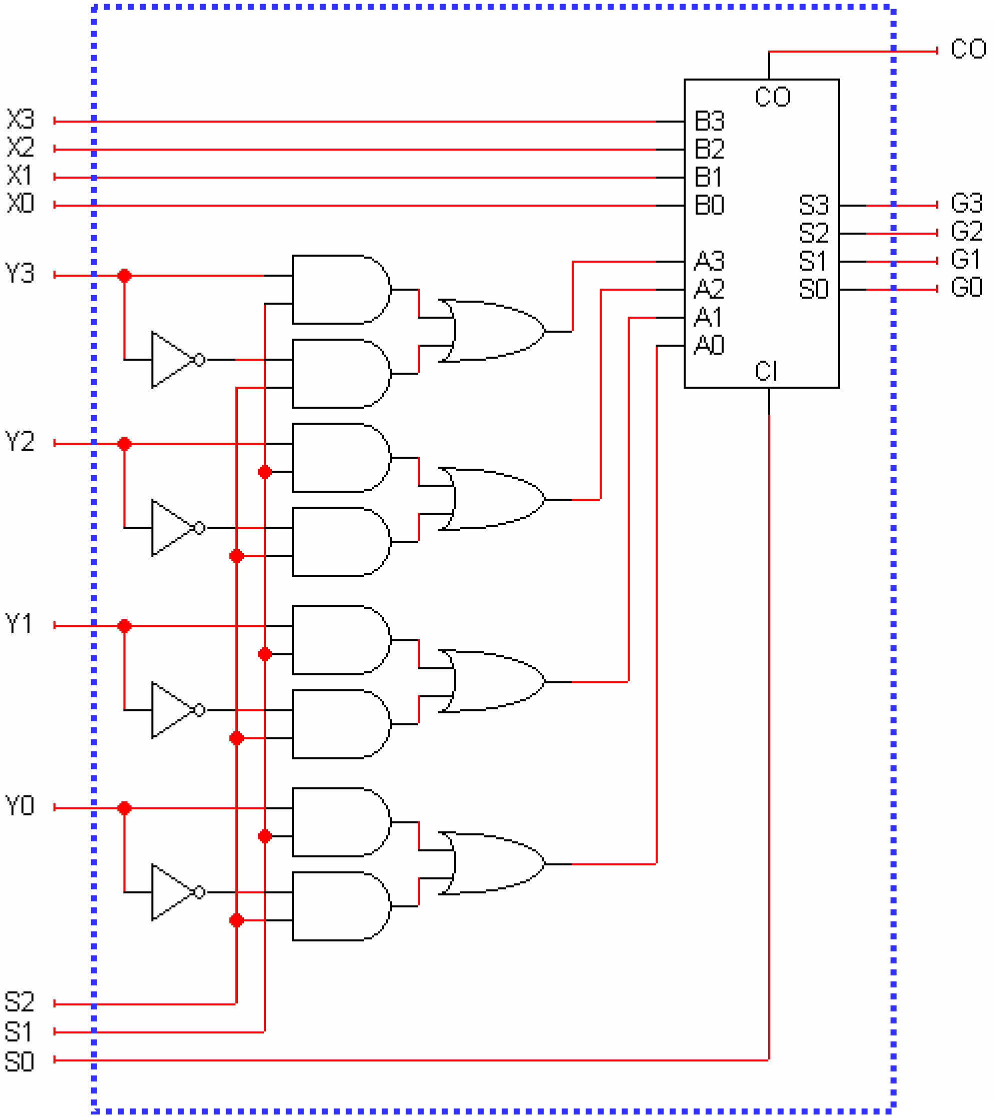
- Again, we have to repeat this once for each adder input bit A3-A0.
- You can see this repetition in the circuit diagram here.



# Our complete arithmetic unit



| S2 | S1 | S0 | G |
|---|---|---|---|
| 0 | 0 | 0 | X |
| 0 | 0 | 1 | X + 1 |
| 0 | 1 | 0 | X + Y |
| 0 | 1 | 1 | X + Y + 1 |
| 1 | 0 | 0 | X + Y’ |
| 1 | 0 | 1 | X + Y’ + 1 |
| 1 | 1 | 0 | X – 1 |
| 1 | 1 | 1 | X |



# Bitwise logical 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 $Y_i$.

$$\begin{array}{lcccc} & 1 & 0 & 1 & 1 \\ \text{AND} & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 0 \end{array} \qquad \begin{array}{lcccc} & 1 & 0 & 1 & 1 \\ \text{OR} & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 1 & 1 \end{array} \qquad \begin{array}{lcccc} & 1 & 0 & 1 & 1 \\ \text{XOR} & 1 & 1 & 1 & 0 \\ \hline & 0 & 1 & 0 & 1 \end{array}$$

- 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 C 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. For instance, to initialize a double-buffered, RGB-mode window with a depth buffer using OpenGL:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

# Bitwise operations in networking

- IP addresses are actually 32-bit binary numbers, and bitwise operations can be used to find network information.
- For example, you can bitwise-AND an address like 192.168.10.43 with a "subnet mask" to find the "network address," or which network the host is connected to.

```
  192.168. 10. 43 = 11000000.10101000.00001010.00101011
& 255.255.255.224 = 11111111.11111111.11111111.11100000
-------------------------------------------------------
  192.168. 10. 32 = 11000000.10101000.00001010.00100000
```

- You can use bitwise-OR to generate a "broadcast address" for sending data to all machines on a local network.

```
  192.168. 10. 43 = 11000000.10101000.00001010.00101011
|   0.  0.  0. 31 = 00000000.00000000.00000000.00011111
-------------------------------------------------------
  192.168. 10. 63 = 11000000.10101000.00001010.00111111
```

# Defining a logic unit

- A logic unit implements different logical functions on two multi-bit inputs X and Y, producing an output G.
- We'll design a simple four-bit logic unit that supports four operations.

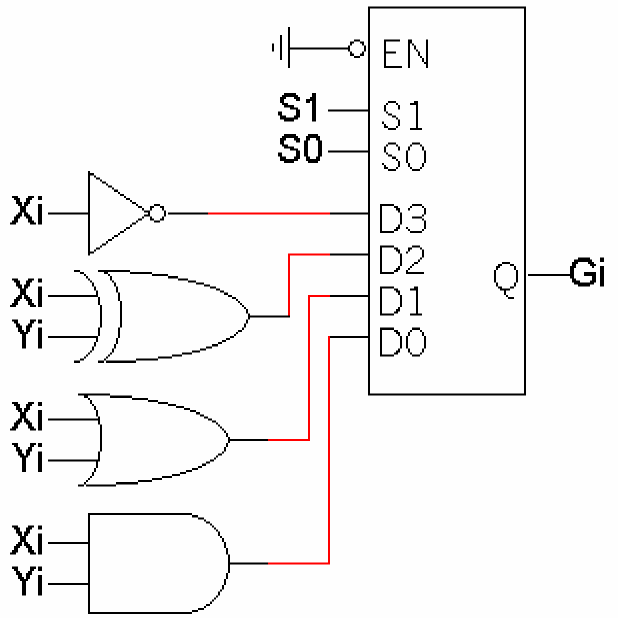
| S1 | S0 | G |
|---|---|---|
| 0 | 0 | XY |
| 0 | 1 | X + Y |
| 1 | 0 | X ⊕ Y |
| 1 | 1 | X' |

- Again, we have to assign a selection code S for each possible function.

# Implementing bitwise operations

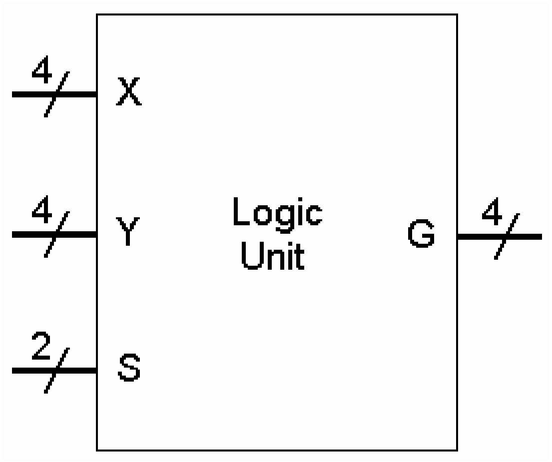
- Bitwise operations are applied to pairs of corresponding bits from inputs X and Y, so we can generate each bit of the output G in the same way.

| S1 | S0 | $G_i$ |
|---|---|---|
| 0 | 0 | $X_i Y_i$ |
| 0 | 1 | $X_i + Y_i$ |
| 1 | 0 | $X_i \oplus Y_i$ |
| 1 | 1 | $X_i'$ |

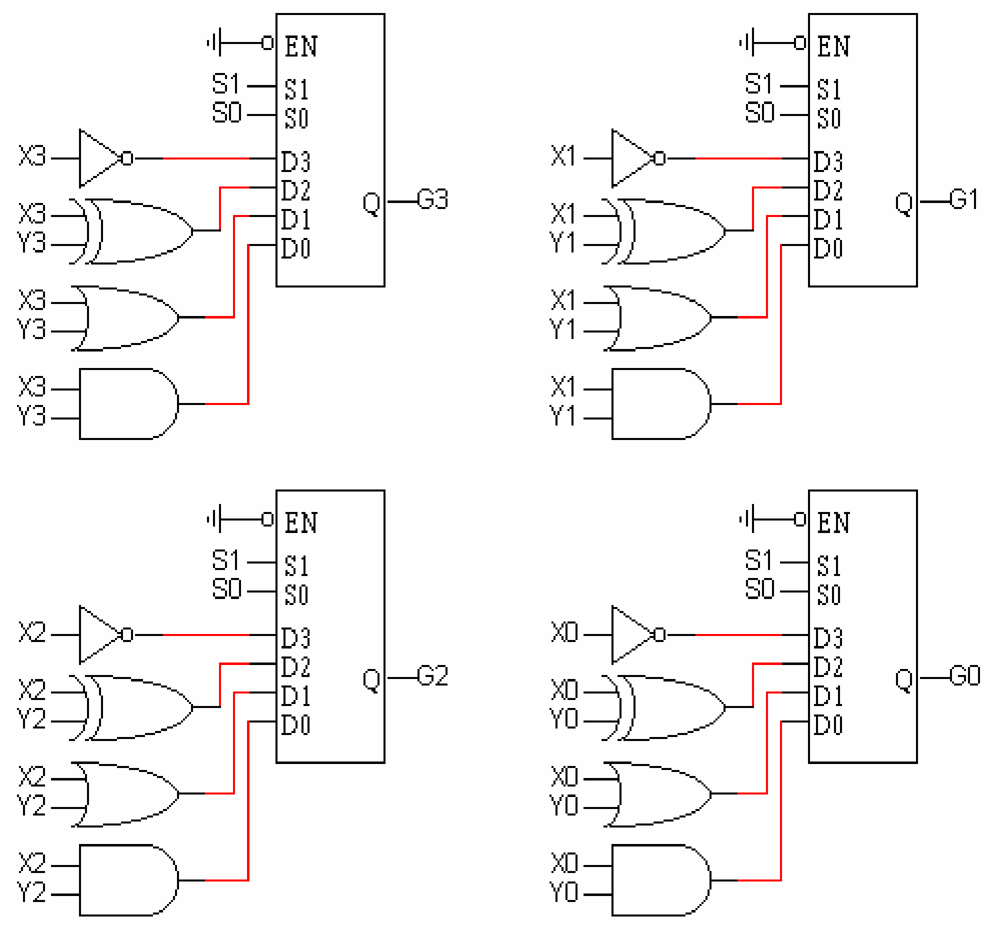


- We can implement each output bit $G_i$ as shown on the right, using a multiplexer to select the desired primitive operation.
- The complete logic unit diagram is given on the next page.

# Our complete logic unit



| S1 | S0 | G |
|---|---|---|
| 0 | 0 | XY |
| 0 | 1 | X + Y |
| 1 | 0 | X ⊕ Y |
| 1 | 1 | X’ |

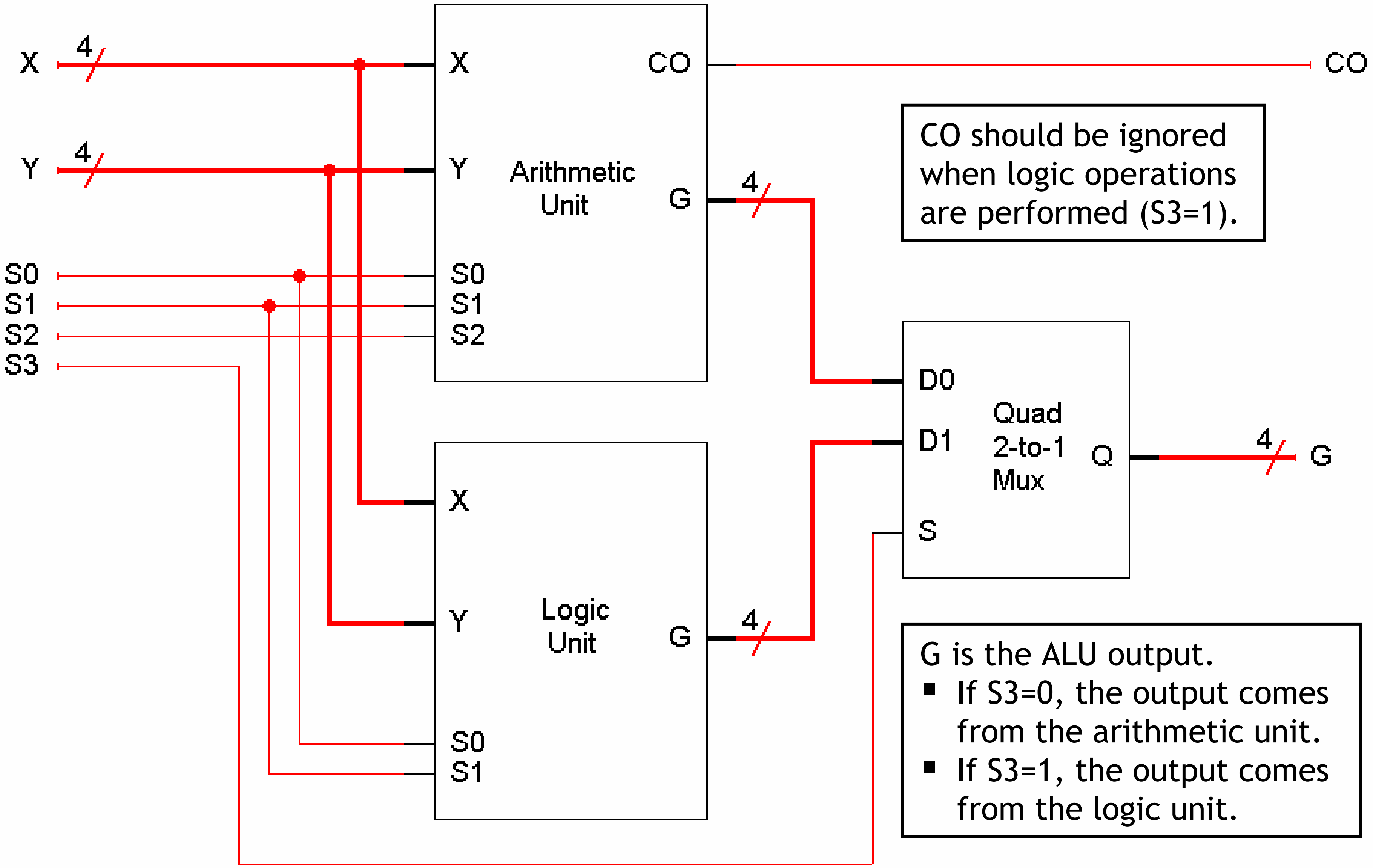


# Combining the arithmetic and logic units

- We can combine our arithmetic and logic units into a single arithmetic-logic unit, or ALU.
  - The ALU will accept two data inputs X and Y, and a selection code S.
  - It outputs a result G, as well as a carry out CO (only useful for the arithmetic operations).
- Since there are twelve total arithmetic and logical functions to choose from, we need a four-bit selection input.
- We've added selection bit S3, which chooses between arithmetic (S3=0) and logical (S3=1) operations.

| S3 | S2 | S1 | S0 | G |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X |
| 0 | 0 | 0 | 1 | X + 1 |
| 0 | 0 | 1 | 0 | X + Y |
| 0 | 0 | 1 | 1 | X + Y + 1 |
| 0 | 1 | 0 | 0 | X + Y' |
| 0 | 1 | 0 | 1 | X + Y' + 1 |
| 0 | 1 | 1 | 0 | X – 1 |
| 0 | 1 | 1 | 1 | X |
| 1 | x | 0 | 0 | X and Y |
| 1 | x | 0 | 1 | X or Y |
| 1 | x | 1 | 0 | X xor Y |
| 1 | x | 1 | 1 | X' |

# A complete ALU circuit



CO should be ignored when logic operations are performed (S3=1).

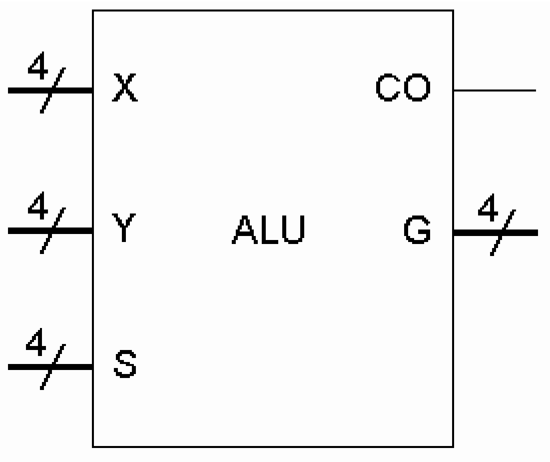
G is the ALU output.

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

# Comments on the multiplexer

- Both arithmetic and logic units are "active" and produce outputs.
  - The multiplexer determines whether the final result G comes from the arithmetic or logic unit.
  - The output of the other one is ignored.
- In programming, you'd use an if-then statement to select one operation or the other. This is useful since programs execute serially, and we want to avoid unnecessary work.
- Our hardware scheme may seem like wasted effort, but it's not really.
  - Disabling one unit or the other wouldn't save that much time.
  - We have to build the hardware for both units anyway.
  - You can think of this as a form of parallel processing, where several operations are done together.
- This is a very common use of multiplexers in logic design.

# An external view of the ALU



| S3 | S2 | S1 | S0 | G |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X |
| 0 | 0 | 0 | 1 | X + 1 |
| 0 | 0 | 1 | 0 | X + Y |
| 0 | 0 | 1 | 1 | X + Y + 1 |
| 0 | 1 | 0 | 0 | X + Y’ |
| 0 | 1 | 0 | 1 | X + Y’ + 1 |
| 0 | 1 | 1 | 0 | X – 1 |
| 0 | 1 | 1 | 1 | X |
| 1 | x | 0 | 0 | X and Y |
| 1 | x | 0 | 1 | X or Y |
| 1 | x | 1 | 0 | X xor Y |
| 1 | x | 1 | 1 | X’ |

# Summary

- In the last few lectures we looked at various arithmetic issues.
  - You can build adders hierarchically, starting with half adders.
  - A good representation of negative numbers simplifies subtraction.
  - Unsigned adders can implement many other arithmetic functions.
  - Logical operations can be applied to multi-bit values.
- Where are we now?
  - We started at the very bottom with primitive gates, but now we can understand ALUs, a critical part of any processor.
  - This all built upon our knowledge of Boolean algebra, Karnaugh maps, multiplexers, circuit analysis and design, and data representations.