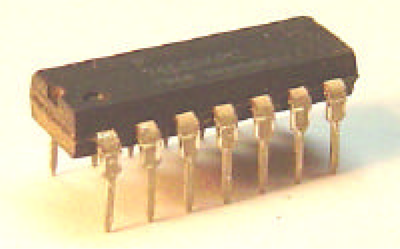
Today's Topics

- **Using TTL logic chips**
- **Multiplexer and Decoder circuits**
- **Programmable Logic Arrays**
- **Comparators, Shifters and Adders**
- **Improving the performance of an Adder circuit**
- **Building an Arithmetic Logic Unit "Bit Slice"**
- **Combining Bit Slices to form a complete ALU**
- **How the ALU function codes work**
- **Signed vs. Unsigned binary numbers**
- **Storing negative numbers using two's complement notation**

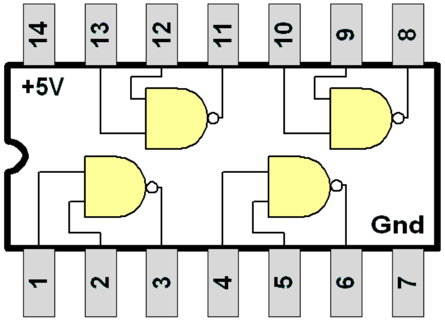
TTL Logic Chips

In the 1970’s the semiconductor industry developed a line of chips which had various types of Boolean logic gates on them. They used a technology called “Transistor-Transistor Logic”, and they became known as TTL Logic Chips.

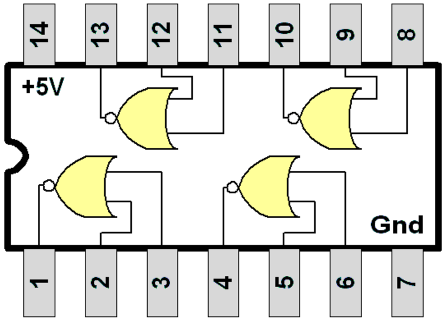


TTL Chip

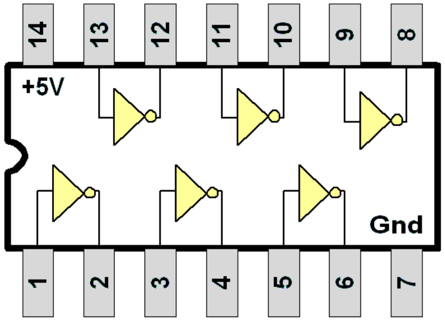
They are often packaged in 14-pin DIP (Dual In-line Package) chips. The pin-outs (diagrams that show how the pins on the chip are connected) for several chips with simple Boolean logic gates are shown below:



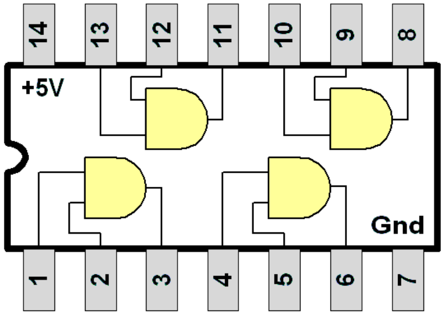
7400
Quad NAND Gate



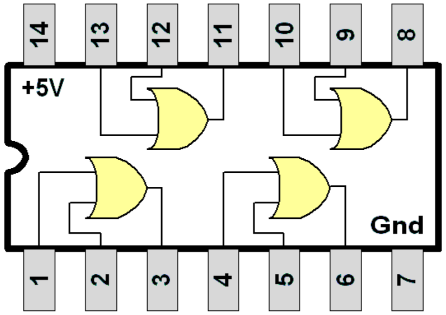
7402
Quad NOR Gate



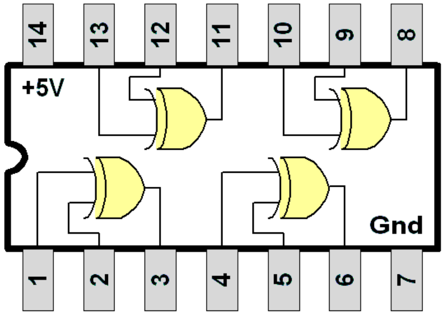
7404
Hex Inverter



7408
Quad AND Gate



7432
Quad OR Gate



7486
Quad XOR Gate

Characteristics of TTL Chips

3.2.1

- **The names of TTL chips include phrases like "Quad" and "Hex" to indicate how many of a particular type of gate they have. "Hex Inverter" means a chip with six inverters.**

- **The individual gates on a TTL chips are completely independent of each other. The inputs and outputs of one gate have nothing to do with the inputs and outputs of another gate (unless the output from one gate is connected to the input of another).**

- **Each chip has two pins that are used to power the chip. One pin is connected to a 5V power supply (i.e., the positive terminal of a battery) while the other is connected to "ground" (0V, or the negative terminal of a battery).**

- **TTL chips accept a 0V or 5V signal on any of their input pins (0V = FALSE, 5V = TRUE), and produce a 0V or 5V signal on the output pins.**

- **For a circuits like NAND, NOR and inverters, the chip can produce a 5V output signal when the inputs to the gate are 0V. The way it does this is by switching the 5V power supply line to the output of the gate.**

- **A single signal can be sent to the inputs of several TTL gates (there is a limit to the number of inputs that a single signal can drive, but for the purposes of solving logic problems we're going to ignore it).**

- **The outputs of two gates cannot be connected together. If one gate tried to output a 5V (TRUE) signal and it was connected to another output that tried to output a 0V (FALSE) signal, the value of the signal would be garbled and the gates could be damaged.**

Combinational Circuits

We'll start our tour of computer circuits by looking at some "combinational circuits". A combinational circuit is one whose outputs depend entirely on the current inputs. If the same inputs are applied to such a circuit, it will always produce the same outputs.

An example of a "combinational circuit" is the adder circuit used within the CPU of a computer. The adder will always produce the same answer if it's given the same inputs.

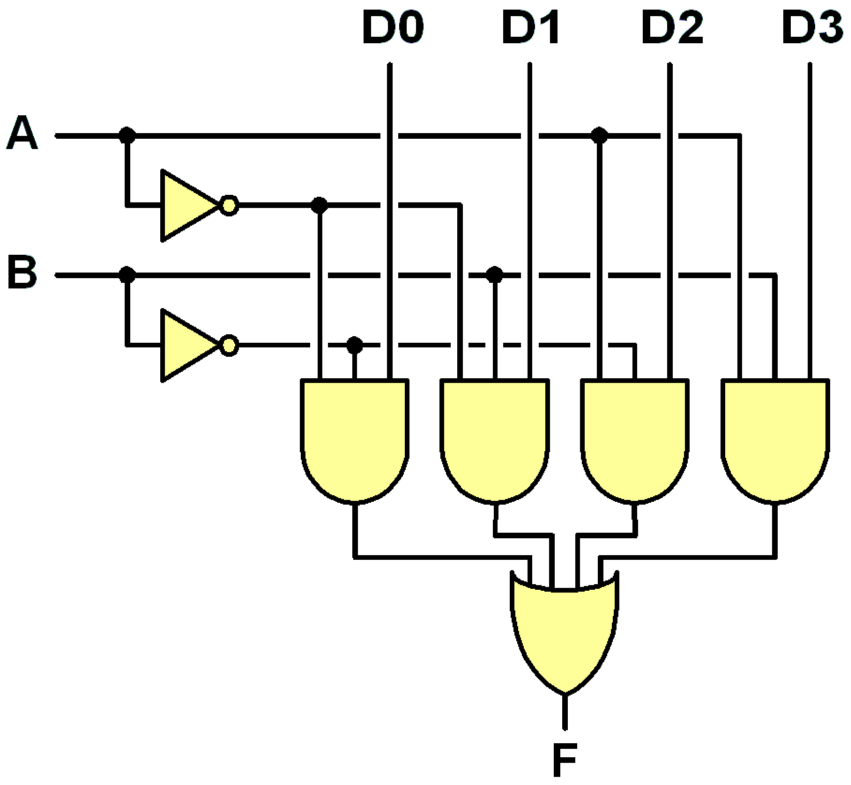
Digital computers use a variety of standard combinational circuits such as adders. But they also rely on non-combinational circuits such as memories. The output of a memory circuit depends not only on the current inputs, but also on what data is stored within the circuit.

The first combinational circuit we'll look at is the multiplexer. A multiplexer chooses one of several data inputs depending on what control inputs it sees.

The "A" and "B" inputs are the control inputs. They select which data input will be used.

The "D0" through "D3" inputs are the data inputs. One of these inputs will be selected by the control inputs.

The "F" output will have the same value as the selected data input.

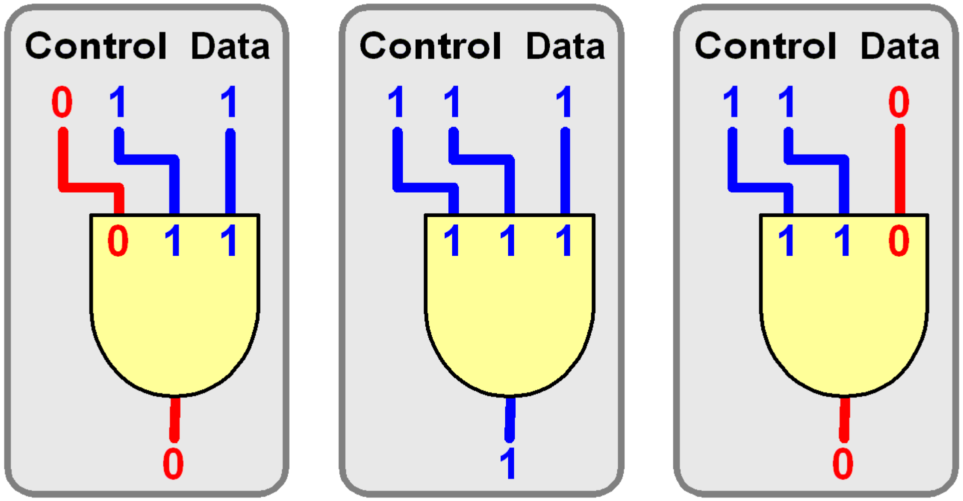


Multiplexer Operation

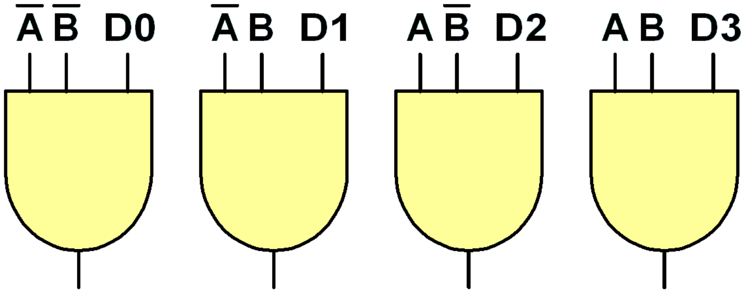
The AND logic elements are part of the key to the operation of a multiplexer circuit. They perform a “Gate” function and either pass data through or block it. In this 4-input multiplexer, each AND gate has three inputs. The left two inputs are the control inputs, and the right input is the data input.

The rules for the AND gate are: “Output true only if ALL inputs are true”. So if either control input is false, the gate cannot activate (left diagram).

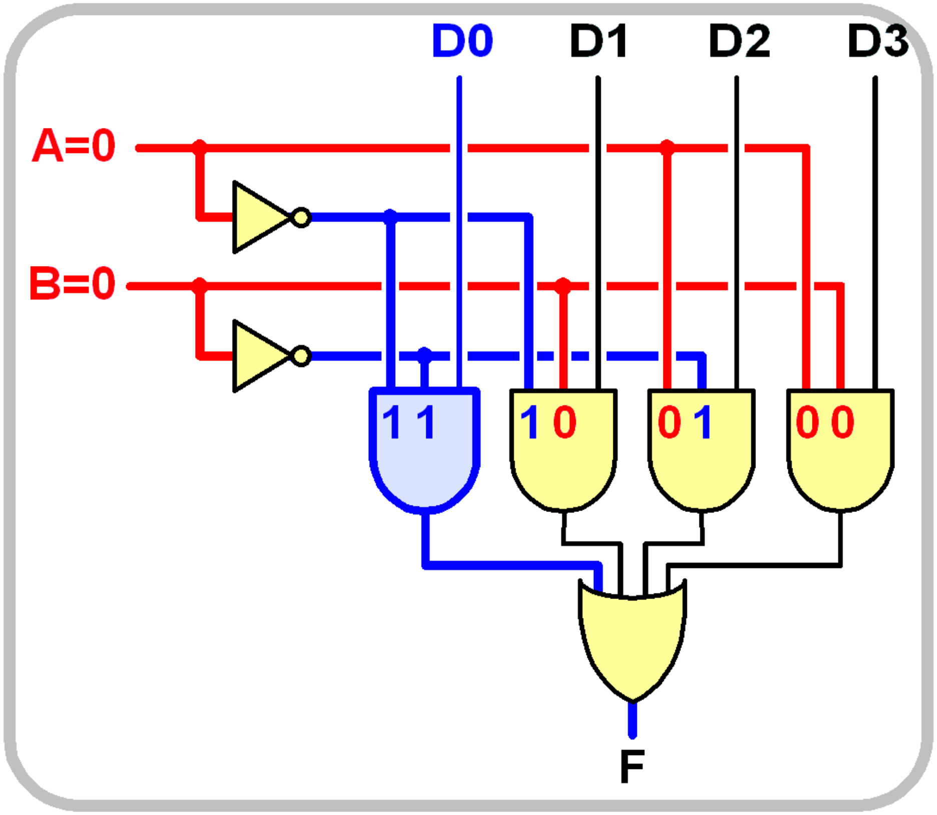
When both control inputs are true, then the output of the gate is the same as the data input. If the data input is true, the output is true (middle diagram). If the data input is false, the output is false (right diagram). So we can say that the data input “passed through” the gate when it’s activated.

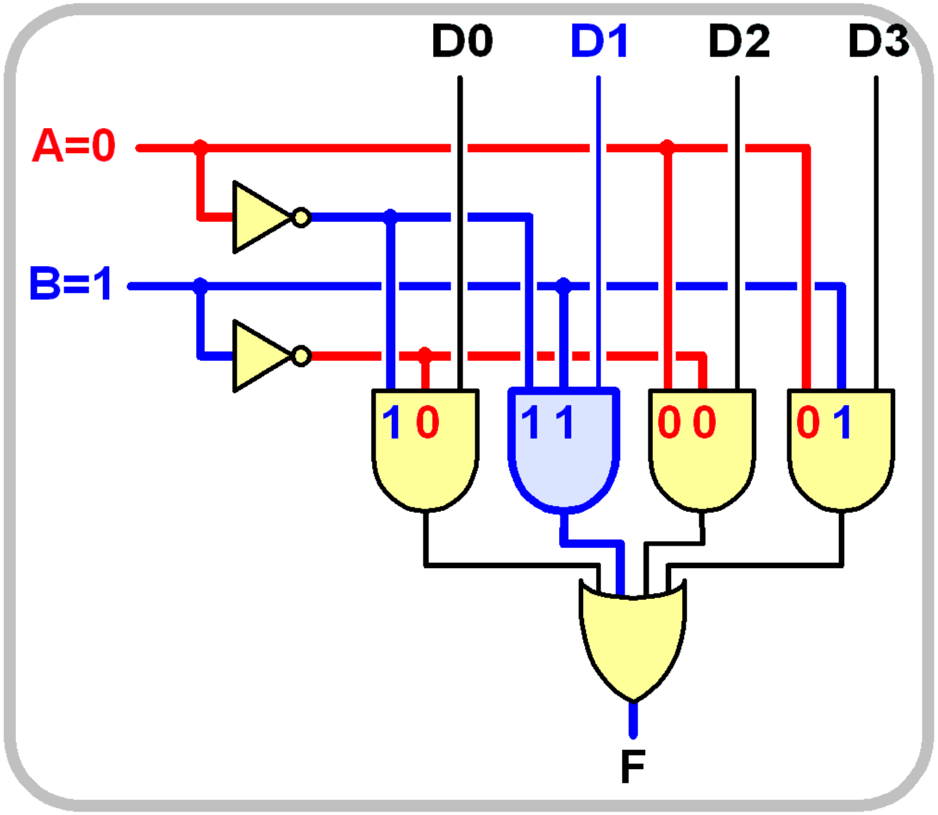


There are four AND gates in the multiplexer circuit we’re looking at. The “selection” part of the multiplexer (which data input it selects) works because of the way the control inputs are connected to each of the AND gates:



Multiplexer Operation



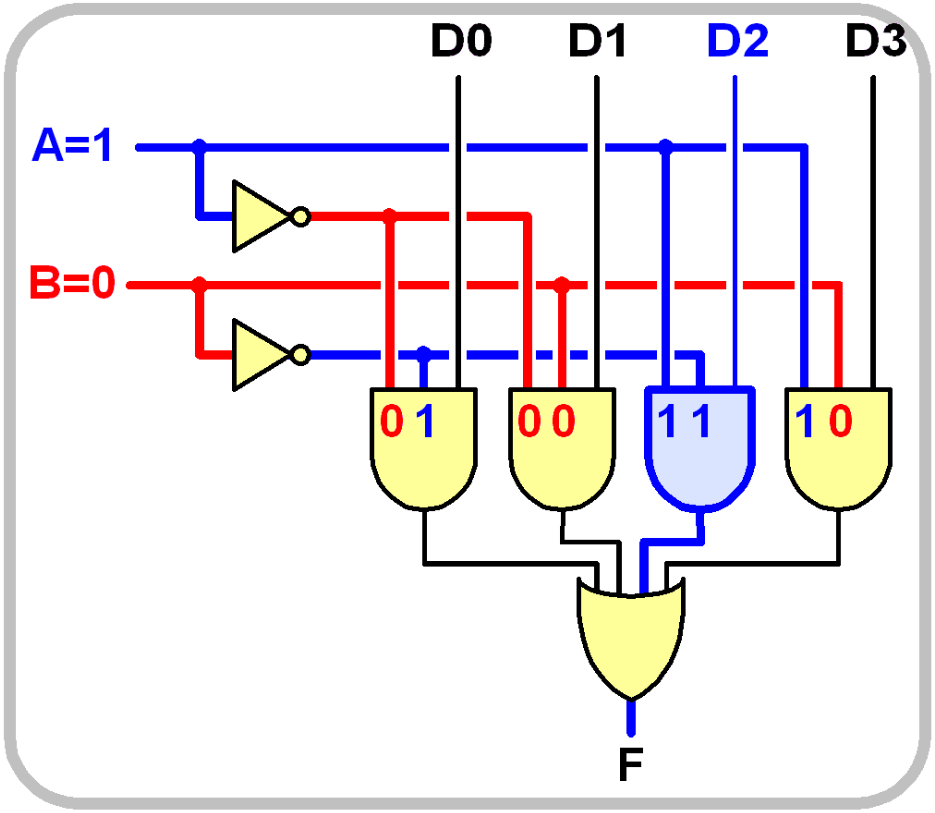


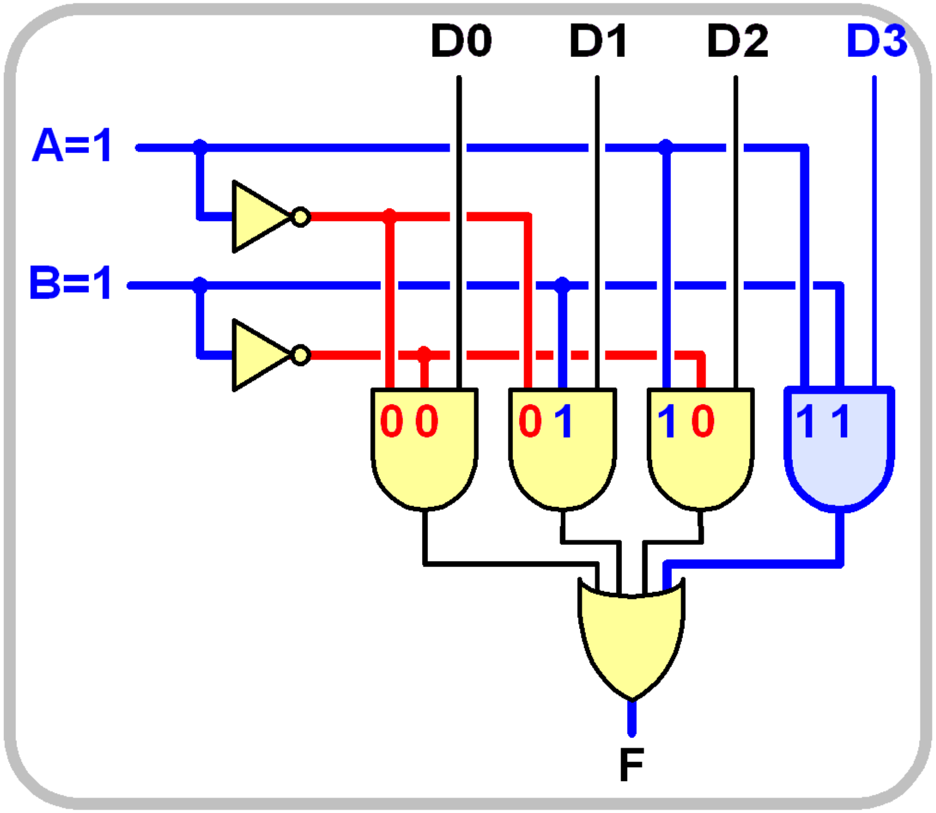
When A=0 and B=0, only the leftmost AND gate has both it's control lines set to TRUE, so only that gate allows the data signal to pass through to the output.

When A=0 and B=1, only the next-to-leftmost AND gate has both it's control lines set to TRUE, so only that gate allows the data signal to pass through.

All of the AND gates produce a zero output except the selected one, so the final OR gate simply passes through the TRUE or FALSE from the one active AND gate.

Multiplexer Operation





When A=1 and B=0, only the next-to-rightmost AND gate has both it's control lines set to TRUE, so only that gate allows the data signal to pass through.

When A=1 and B=1, only the rightmost AND gate has both it's control lines set to TRUE, so only that gate allows the data signal to pass through.

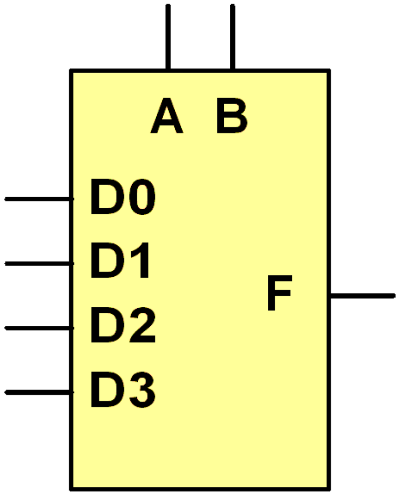
Multiplexer Circuits

We can describe what a multiplexer circuit does using a truth table:

A	B	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

The table shows, for example, that when A=0 and B=0, output "F" is the same as input "D0"

When we draw a circuit that uses a multiplexer we can use the following symbol instead of drawing all of the individual gates:



What we've been looking at is called a "4-input multiplexer". You can buy TTL chips with complete 4-input multiplexer circuits built into them.

There are also TTL chips with 8-input and 16-input multiplexers. An 8-input multiplexer needs three control lines, because three binary signals gives 8 possible control combinations (2^3=8). The textbook shows the circuitry for an 8-input multiplexer in figure 3-11 on page 149.

Similarly, a 16-input multiplexer needs four control lines, since four binary signals allows for the 16 different combinations required to select one of the inputs (2^4=16).

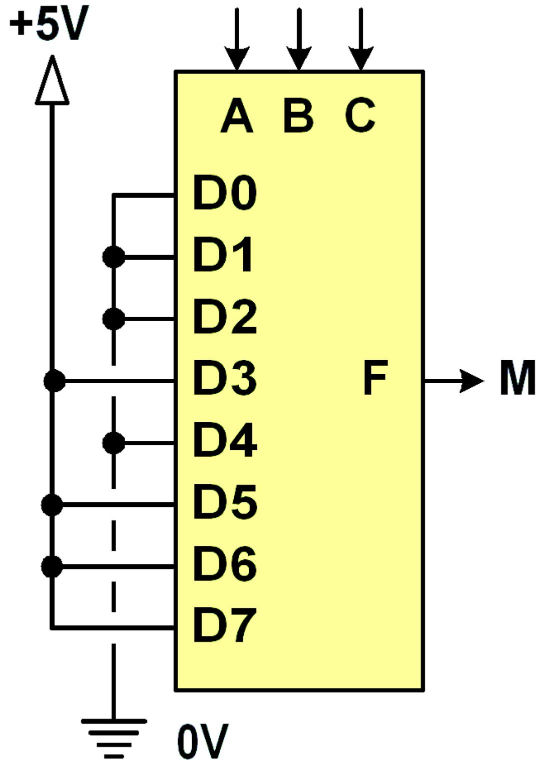
Using a Multiplexer to Implement Logic

A multiplexer with "n" control inputs can be used as an easy solution for any Boolean logic problem with "n" inputs. Below is an example of an 8-input multiplexer wired to solve the 3-input "Majority Function":

We've connected the A, B and C inputs of our majority function to the control inputs of the multiplexer. And we've "hardwired" the eight data inputs to 5V (TRUE) or 0V (FALSE) depending on what output the majority function is supposed to produce for the various control combinations.

For example, our majority function is supposed to produce a TRUE result when A=0, B=1 and C=1. Under those A,B and C input conditions the multiplexer will select the "D3" input. Therefore, we've hardwired the D3 input to a TRUE value.

A	B	C	M
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

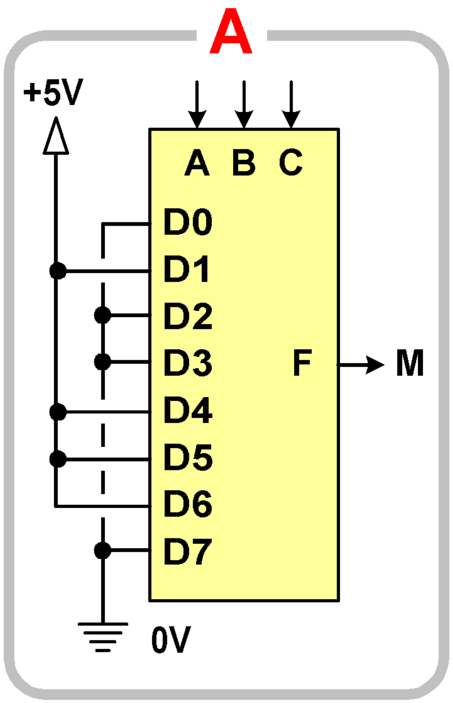


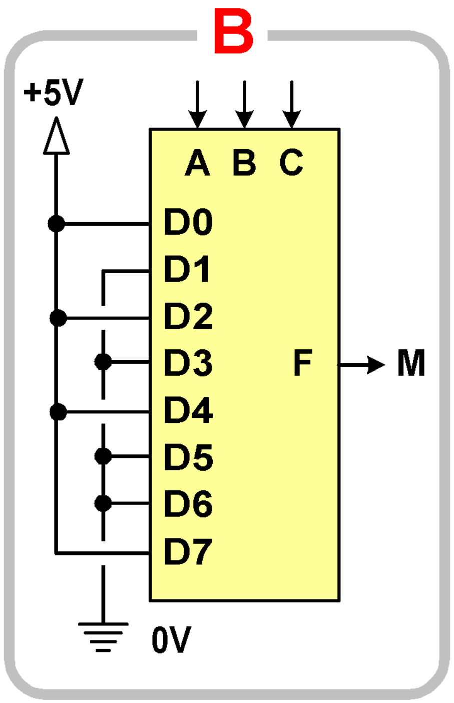
This gives us a very simple way to solve any Boolean problem with 3 input conditions (or, for a 16-input multiplexer, any Boolean problem with 4 input conditions).

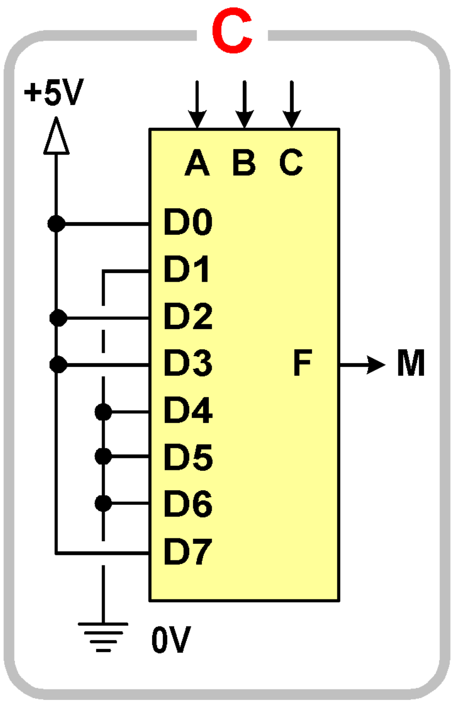
Exercise 1 - Multiplexer

Which of the three multiplexer circuits on the right corresponds to the truth table on the left?

A	B	C	M
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

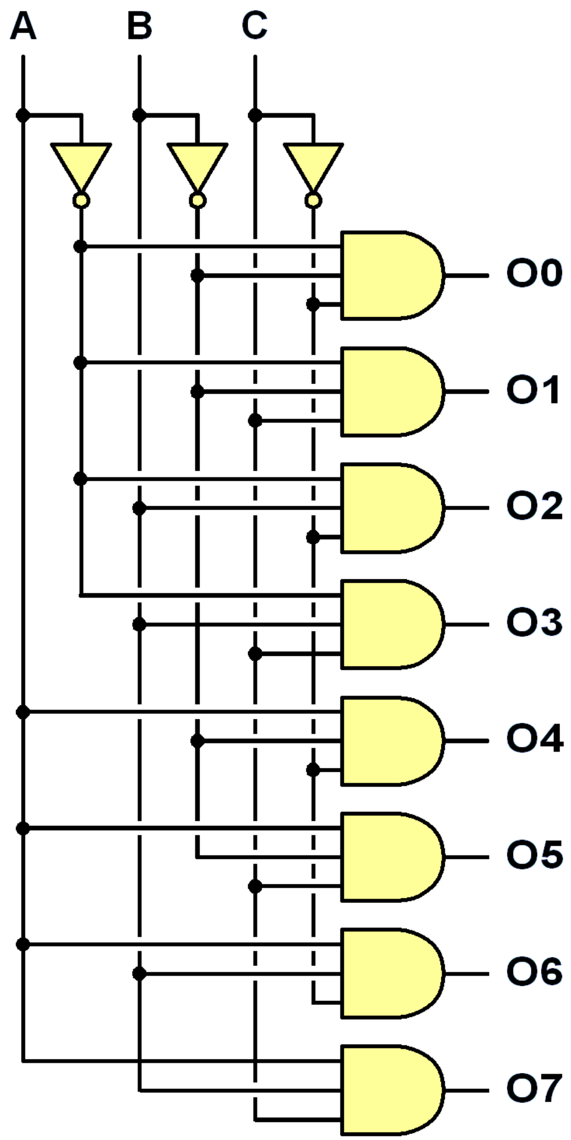






Decoder Circuit

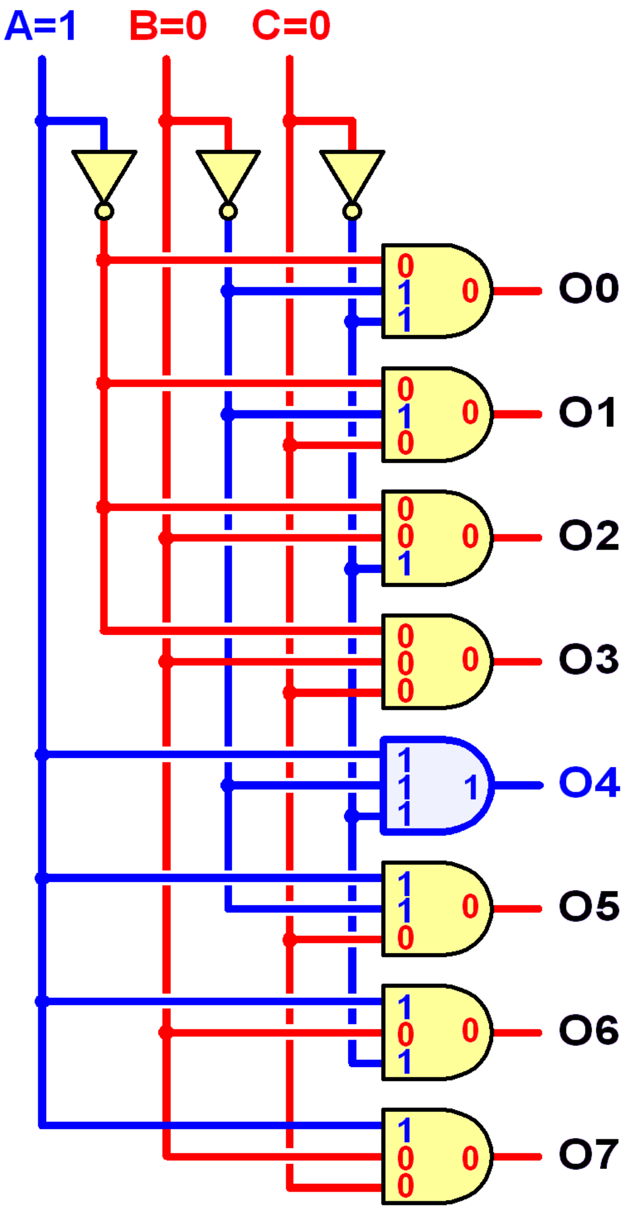
Decoders are a very common feature of computer circuits. A decoder chooses one output based on a set of control inputs:



A	B	C	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder Circuit Example

The diagram below shows how output O4 is selected when A=1, B=0 and C=0:



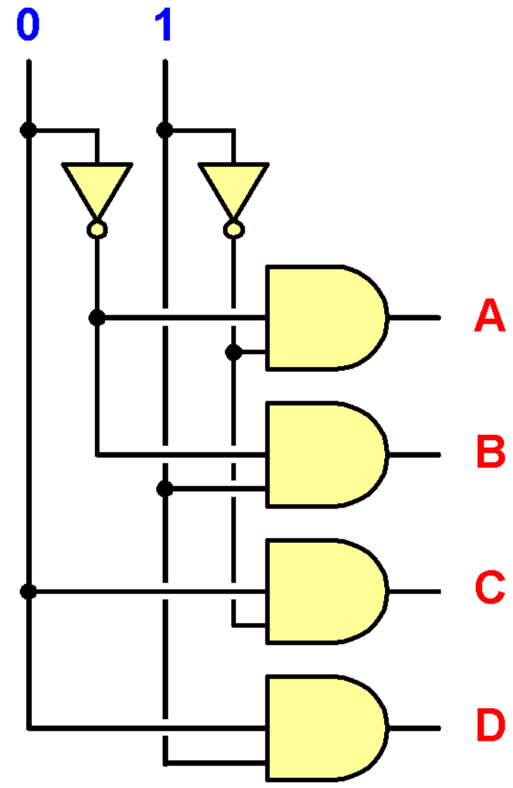
The connections between the control inputs and the AND gates are alternated so that each of the 8 gates has a different combination of A, NOT-A, B, NOT-B, C, and NOT-C inputs. When A=1, B=0 and C=0, only the fifth AND gate see all three of it's inputs as TRUE, and so it's the only one that activates.

This example shows a 8-output decoder, but you can also build other decoders – for example a 4-output decoder would require 2 control inputs ($2^2 = 4$), and a 32-output decoder would require 5 control inputs ($2^5 = 32$).

Decoders are found in many places in a computer. A good example is in memory circuits – decoders are used to take part of a memory address and select which memory chip or which memory cell needs to be accessed.

Exercise 2 - Decoder Circuit

For this 4-output decoder with the given inputs, which of the outputs is TRUE?



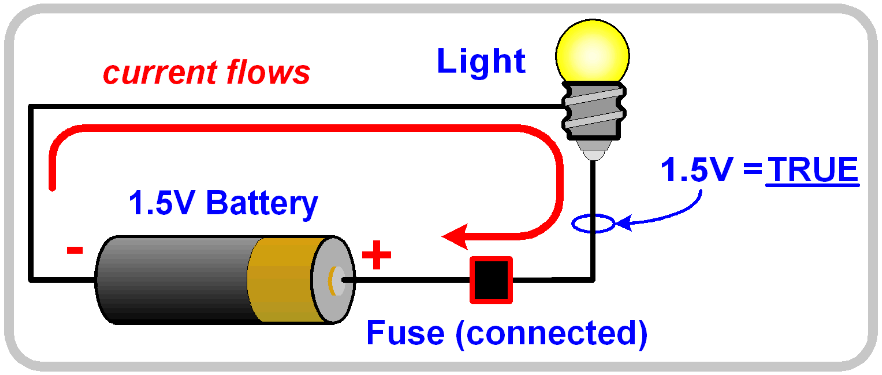
Programmable Logic Arrays

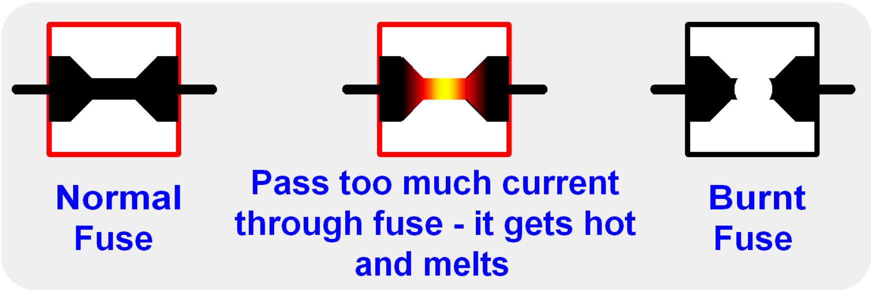
A Programmable Logic Array (PLA) is a general-purpose logic chip whose function can be "programmed" to solve a Boolean problem. The "programming" of a PLA chip is not the same as programming a computer system. In the chip industry, "programmable" chips are chips whose circuitry can be changed after the chip is manufactured.

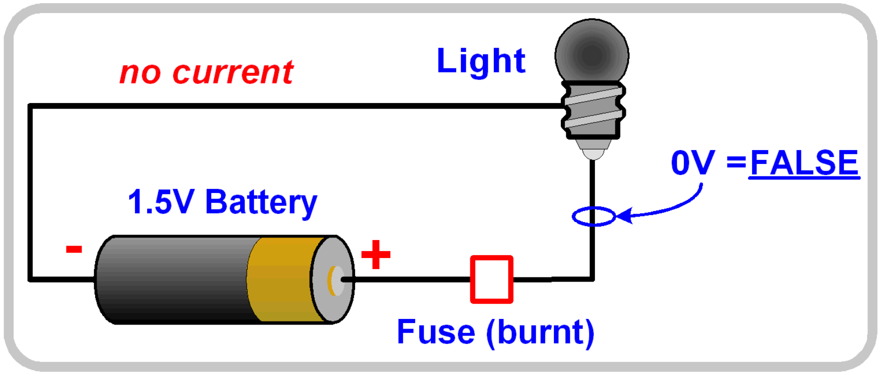
Programmable chips contain fuses to connect certain circuits. A fuse contains a very narrow link of metal which normally conducts electricity – but it can be easily melted by passing lots of electric current through it. Once the metal link has been melted, electricity can no longer flow.

Fuses allow some of the circuits in the chip to be changed after it's been manufactured and delivered to the user. "Programmable" chips that use this technique can be customized to perform whatever functions are needed.

But – once a fuse has been burned it cannot be undone!







Programmable Logic Arrays

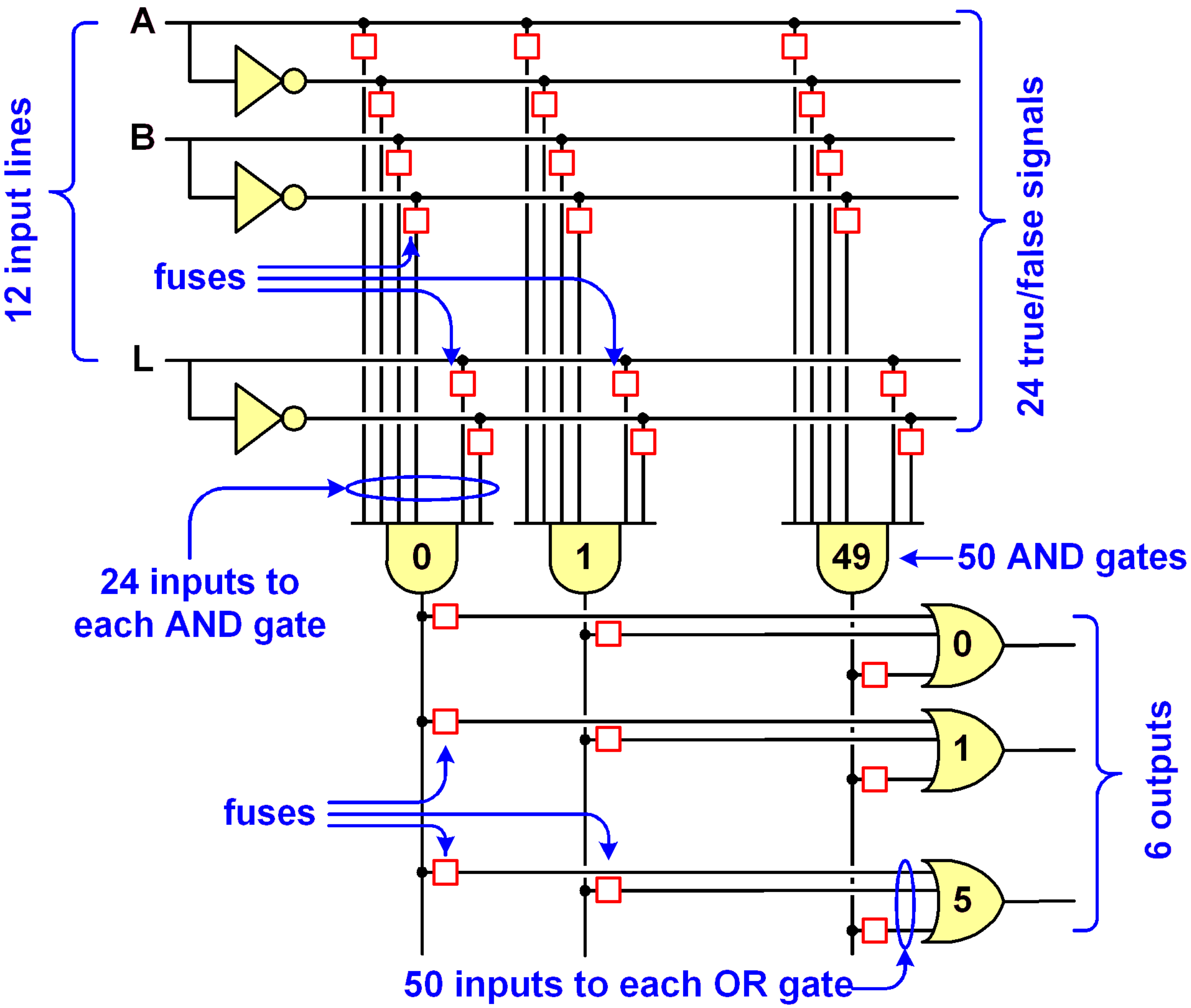
PLA chips make it easy to solve Boolean problems by implementing a "sum of products" solution. Here is a sample of what a PLA chip can look like:

This is a sample of a chip with 12 inputs, 50 AND gates, and 5 outputs.

One set of fuses allow any of the 12 inputs or their negative values to be connected or not connected to any of the 50 AND gates.

A second set of fuses allow the output of the AND gates to be connected or not connected to any of the six output OR gates.

Since there are six outputs, up to six independent Boolean "sum of products" circuits can be created.



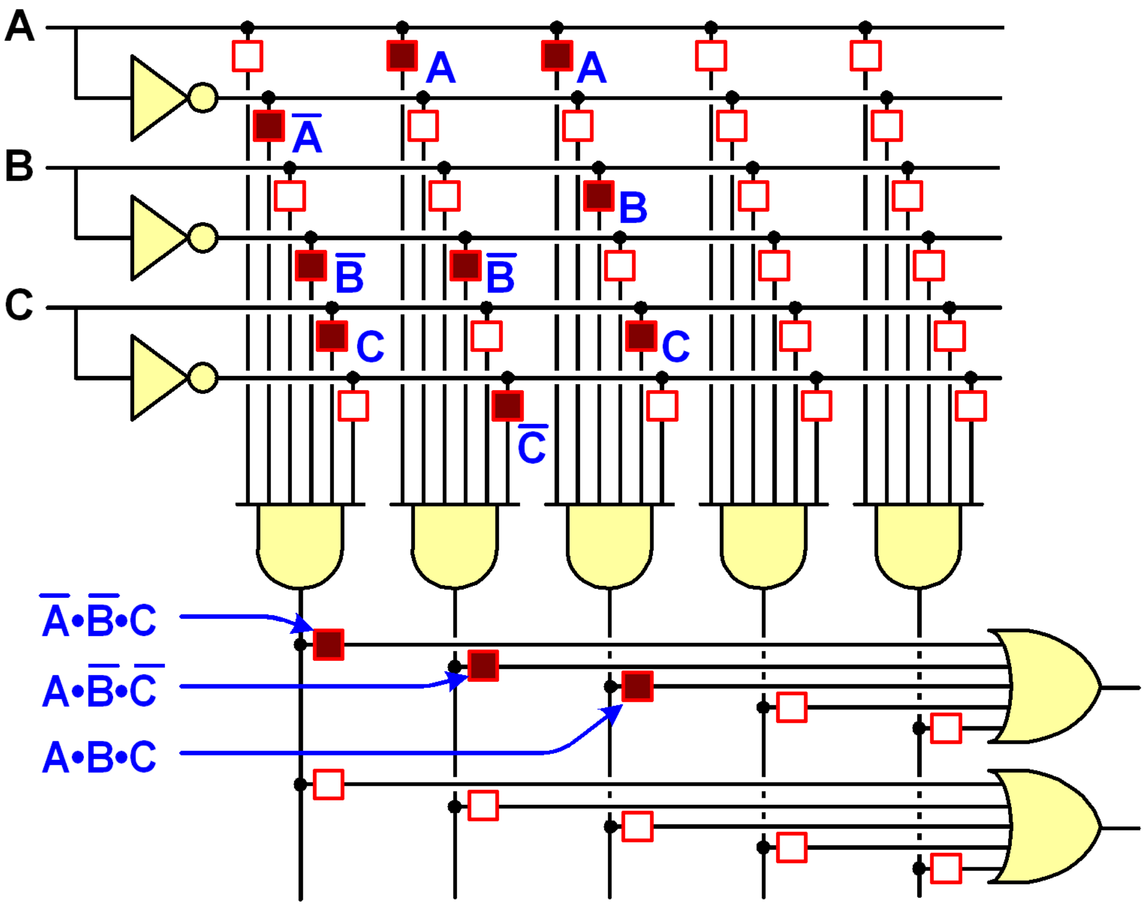
Programmable Logic Arrays

Here's a sample of how a PLA can be used to implement a Boolean function:

This is the truth table that shows the results we need to produce for three inputs:

A	B	C	Result	
0	0	0	0	
0	0	1	1	$\overline{A}\bullet\overline{B}\bullet C$
0	1	0	0	
0	1	1	0	
1	0	0	1	$A\bullet\overline{B}\bullet\overline{C}$
1	0	1	0	
1	1	0	0	
1	1	1	1	$A\bullet B\bullet C$

$(\overline{A}\bullet\overline{B}\bullet C) + (A\bullet\overline{B}\bullet\overline{C}) + (A\bullet B\bullet C)$

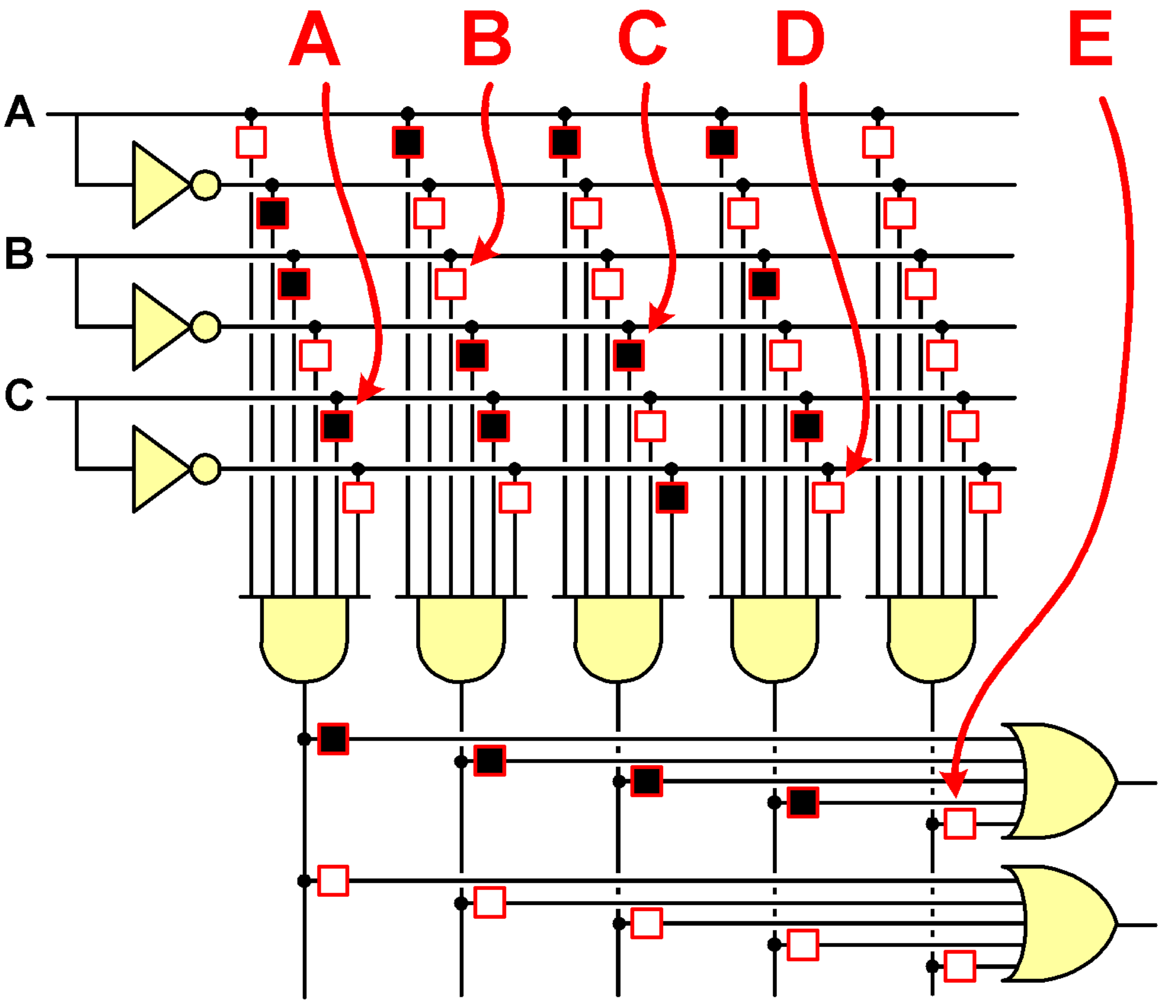


The filled-in boxes show which fuses are connected. The connected fuses pass the appropriate signals from the inputs through to the AND gates, and then connect the output of the AND gates to the inputs of the OR gate. We're only producing one result, so we only need to use one output OR gate.

Exercise 3 - Programmable Logic Array

The following PLA has been programmed to perform the "Majority" function for thee inputs. Which of the connections is wrong?

$$(\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$



Arithmetic Circuits

Arithmetic circuits are used to manipulate groups of bits which represent numeric values. Remember that computers represent numbers in binary format. For example, the decimal number “42” can be represented by a group of 8 binary signals, or “bits”, as follows:

bit contents	0	0	1	0	1	0	1	0
bit number	7	6	5	4	3	2	1	0
bit value	2^7 (128)	2^6 (64)	2^5 (32)	2^4 (16)	2^3 (8)	2^2 (4)	2^1 (2)	2^0 (1)

32 + 8 + 2 = 42

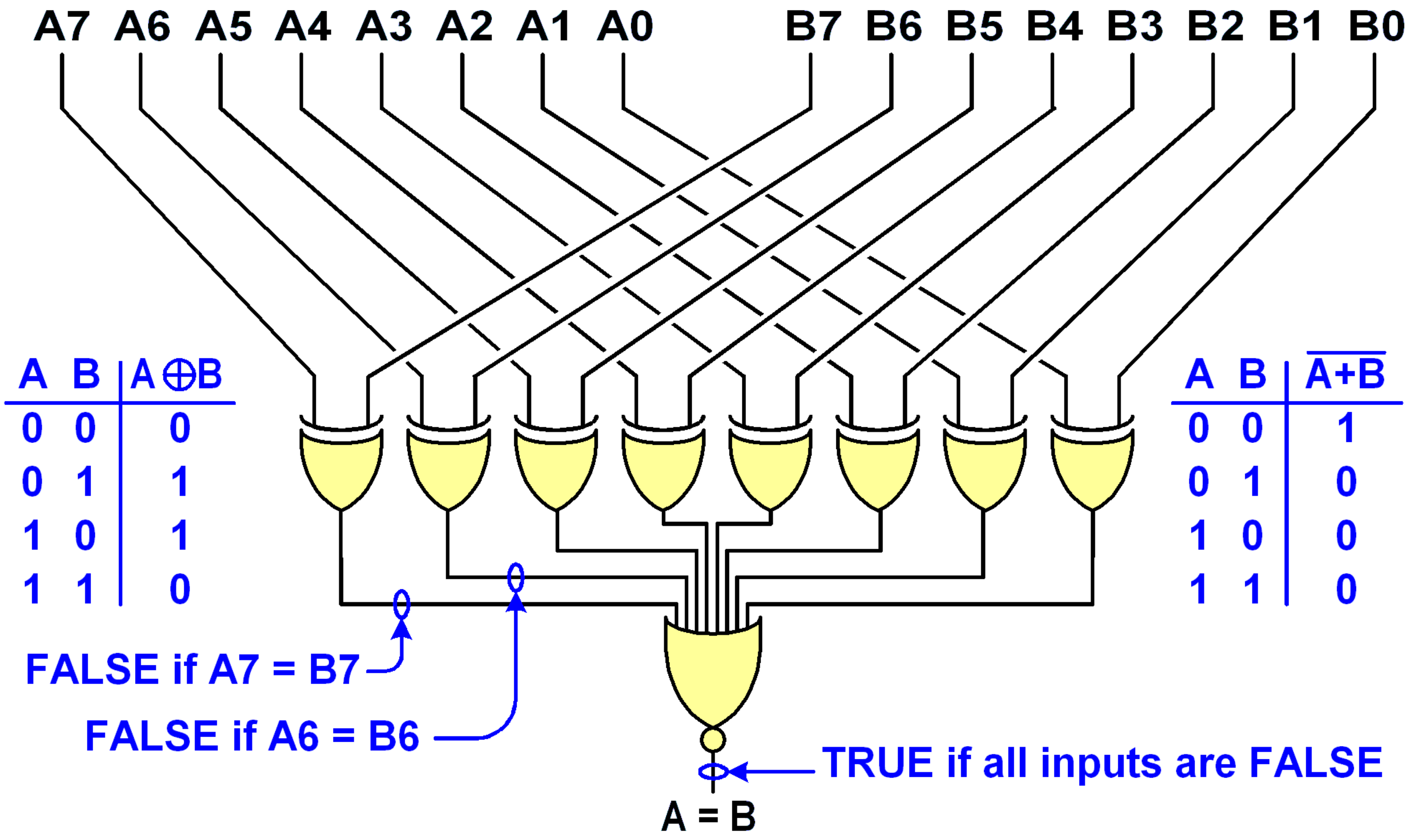
Arithmetic circuits can take a group of signals and manipulate their values according to the rules for binary arithmetic. Arithmetic circuits include:

- Comparators – circuits which compare two numbers to see if they have the same value
- Shifters – circuits which multiply or divide numbers by powers of 2
- Adders – circuits which can add two numbers together

We’ll be using examples of circuits which work on groups of 8 bits – but the circuits can easily be extended to work on larger numbers.

Comparators

A comparator is one of the simplest arithmetic circuits – it accepts input words and produces an output which is TRUE if the two words are identical:



A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

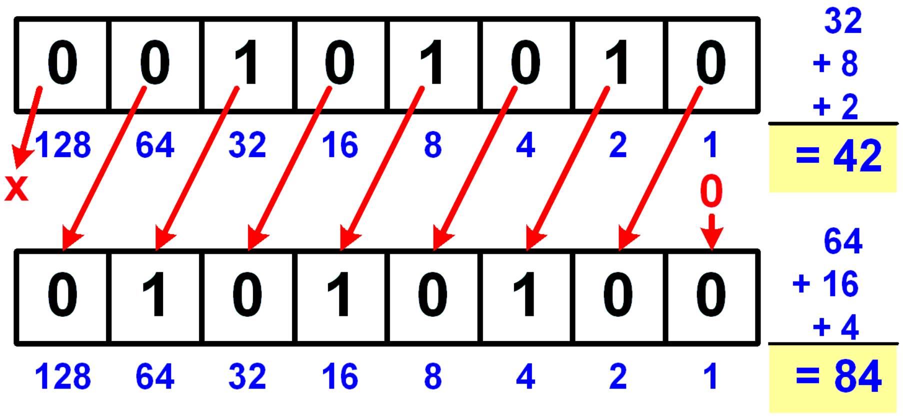
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

The heart of the comparator is the series of XOR gates which compare each bit. XOR produces a TRUE output if one input is true or the other input is true but not if both are true. So a TRUE output from an XOR gate means that the two inputs are different.

Shifters

A shifter is a circuit which can divide or multiply a binary number by 2. Shifting means to move all of the bits in a number one position to the left or to the right. Moving a bit to the left increases it's value by twofold, while moving a bit to the right decreases it's value by half:

Shifting to the Left



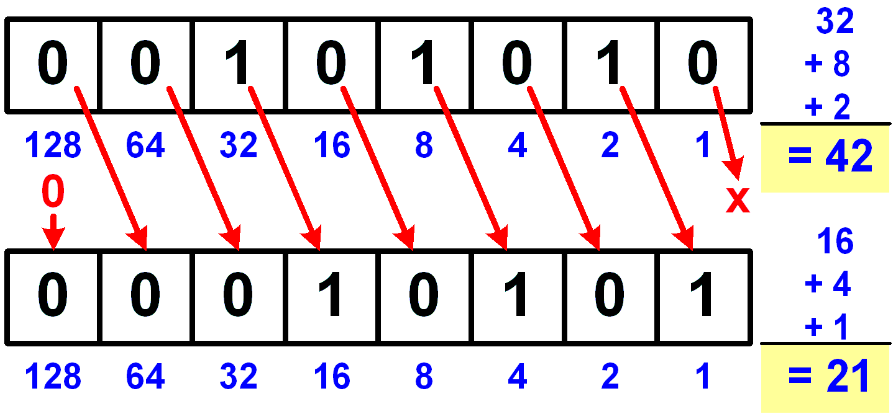
0	0	1	0	1	0	1	0
128	64	32	16	8	4	2	1

32 + 8 + 2 = 42

0	1	0	1	0	1	0	0
128	64	32	16	8	4	2	1

64 + 16 + 4 = 84

Shifting to the Right



0	0	1	0	1	0	1	0
128	64	32	16	8	4	2	1

32 + 8 + 2 = 42

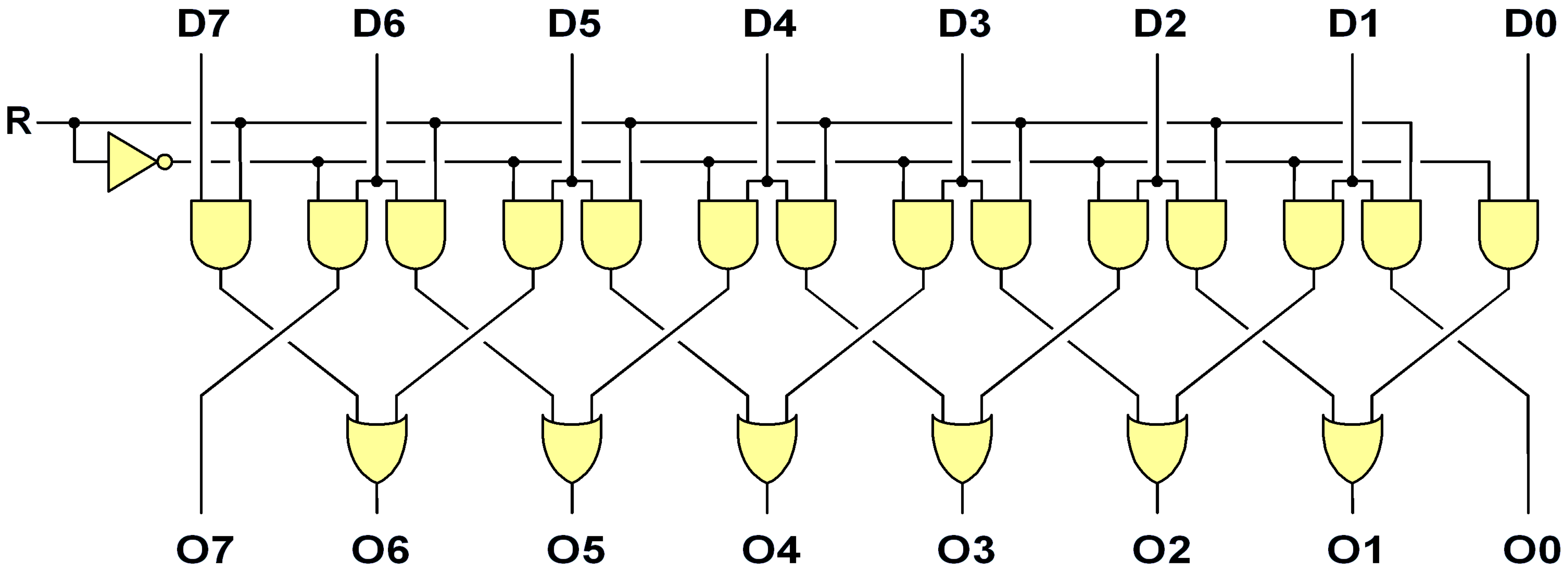
0	0	0	1	0	1	0	1
128	64	32	16	8	4	2	1

16 + 4 + 1 = 21

Notice that a "0" bit is inserted at one end of the number, and the bit which is "shifted out" of the number simply disappears.

Shifters

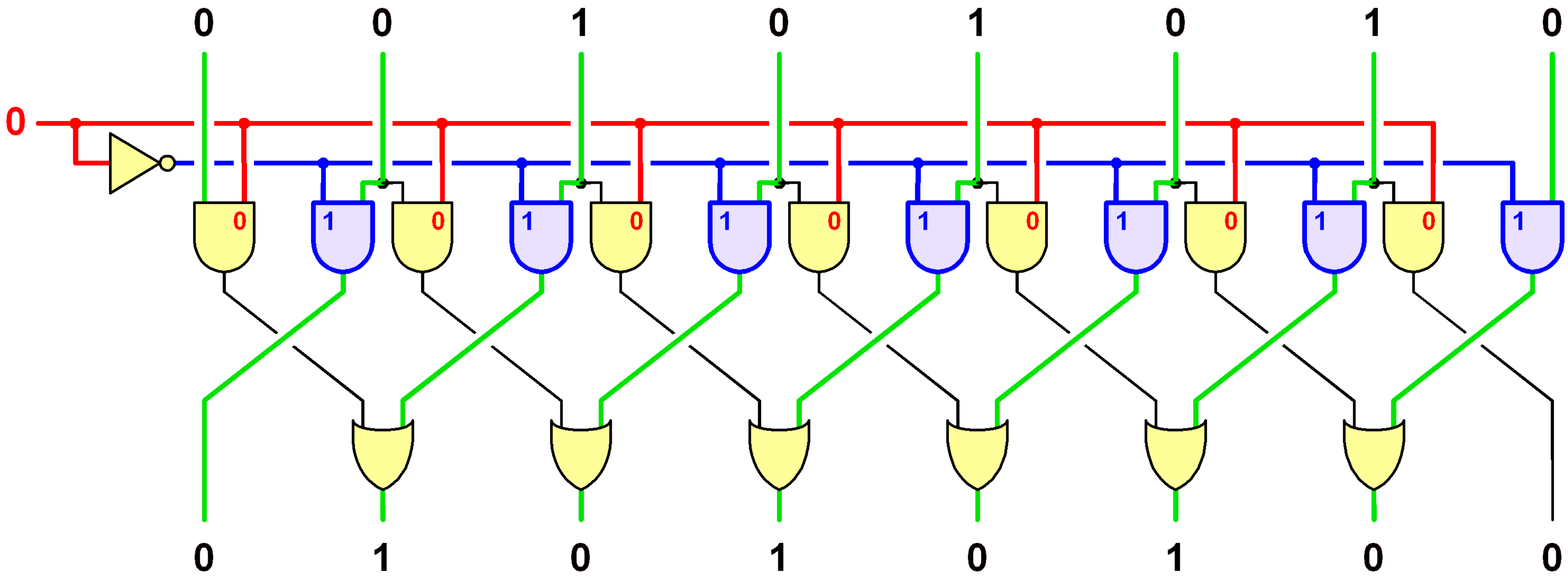
This is how a shifter circuit is built. This shifter accepts two 8-bit input words and a control line which tells the circuit to shift either to the left or to the right:



The "R" input controls the direction of the shift. If the input is TRUE, then the input value is shifted one position to the right. If this input is FALSE, the input value is shifted to the left.

Shifters – Left Shift Example

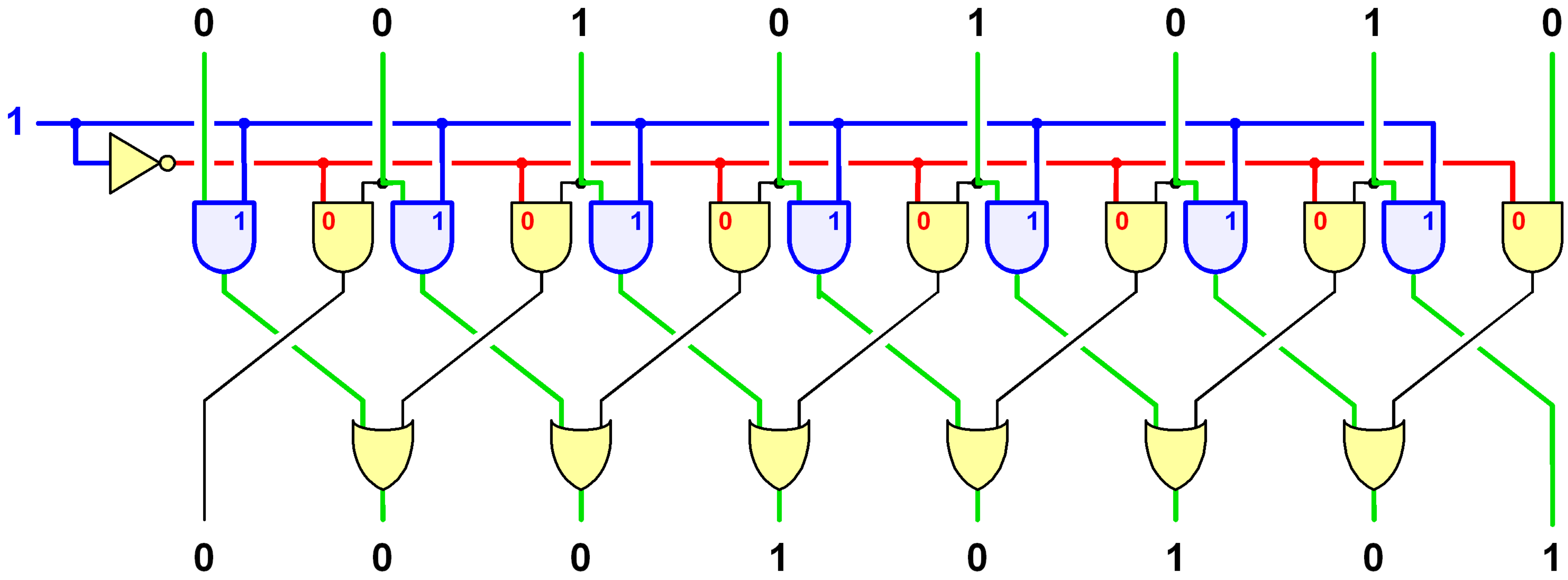
This diagram shows how the shifter works when it is shifting bits to the left:



When the control input is zero, the left-hand AND gate for each input bit is activated. Those gates accept the data input and send it to the output position one bit to the left.

Shifters – Right Shift Example

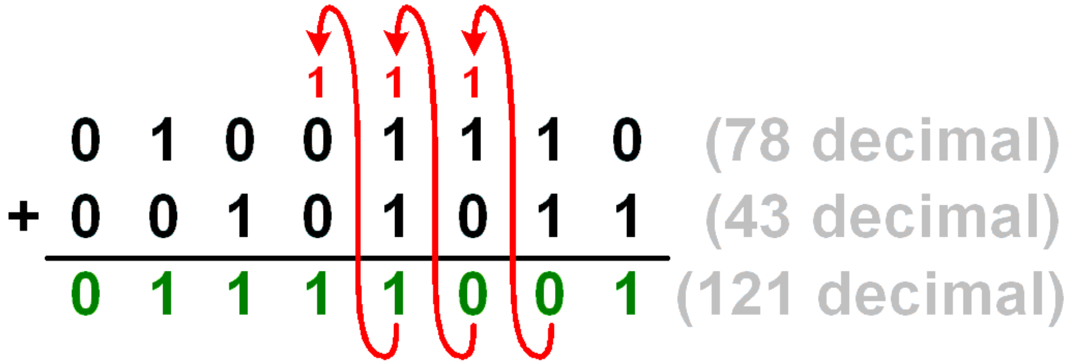
This diagram shows how the shifter works when it is shifting bits to the right:



When the control input is one, the right-hand AND gate for each input bit is activated. Those gates accept the data input and send it to the output position one bit to the right.

Adders

The ability to add two binary numbers together is an essential piece of a computer system. Although each individual signal can only hold a “1” or a “0”, computers put multiple bits together to form multi-bit numbers. Here’s an example of two 8-bit numbers being added:



```
         1   1   1
  0 1 0 0 1 1 1 0   (78 decimal)
+ 0 0 1 0 1 0 1 1   (43 decimal)
  ---------------
  0 1 1 1 1 0 0 1  (121 decimal)
```

The procedure for adding binary numbers is the same as for decimal numbers:

- **Start with the rightmost column**
- **Add the two digits in the column together to create a result**
- **If there’s a carry, add it to the next column**
- **Repeat, moving one column to the left each time**

Adders

3.2.3

In decimal, there are lots of combinations of two digits with 10 possible values for each. But in binary, since each digit can only be “1” or “0”, the rules for addition are very simple:

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

(and carry 1)

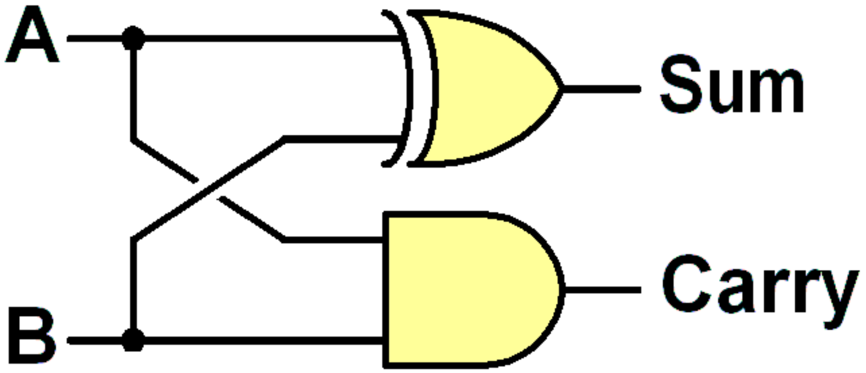
We can describe the rules for binary addition using a truth table. Our truth table shows two inputs A and B (the two binary digits to be added) and two outputs (Sum and Carry):

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

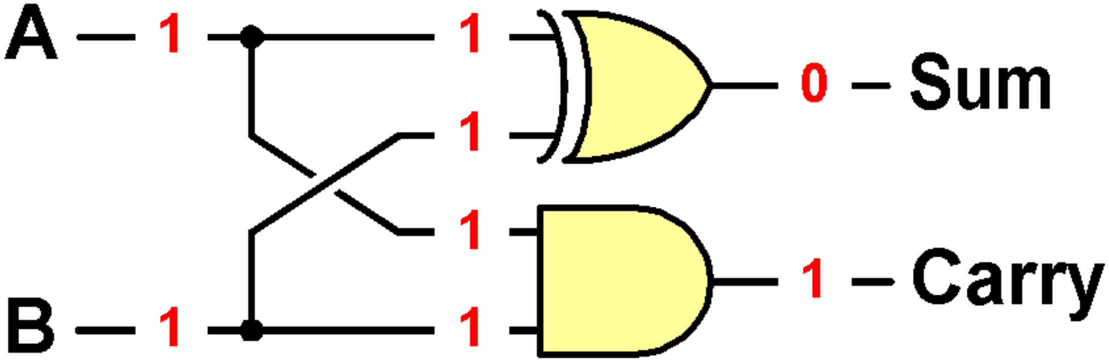
Note that the results required to produce the “Sum” are the same results that an XOR gate produces. Also notice that the results for “Carry” are the same as an AND gate.

Adders

Since our truth table shows that XOR produces the sum of two binary digits and AND produces the carry, a adder circuit is very simple – it just use these gates with the same two inputs for each:



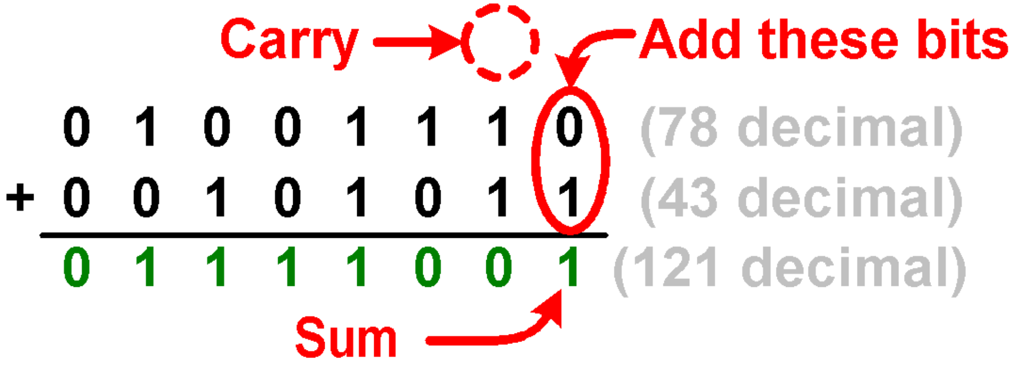
Here's an example of the circuit adding "1 + 1":



The sum is "0" and there is a carry into the next digit position.

Multi-bit Adders

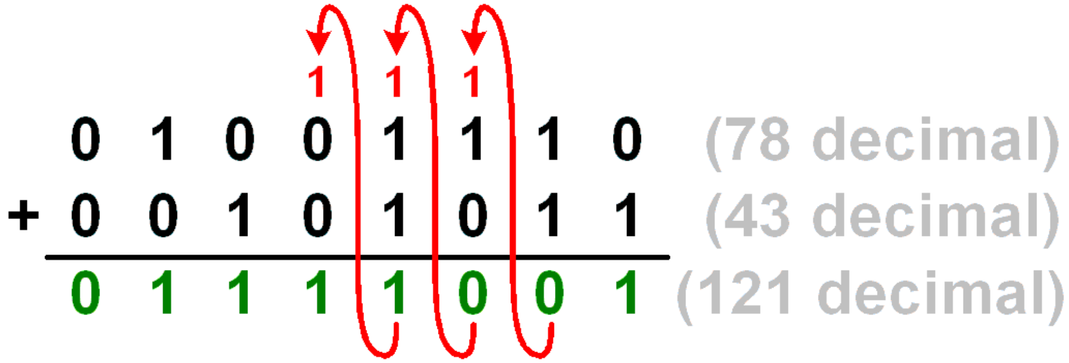
A computer system needs the ability to add multi-bit numbers. The adder we just designed can add a single column of bits to produce a sum for that column and a carry which can be added into the next column:



```
  0 1 0 0 1 1 1 0  (78 decimal)
+ 0 0 1 0 1 0 1 1  (43 decimal)
  ---------------
  0 1 1 1 1 0 0 1  (121 decimal)
```

We must use a separate copy of the adder circuit for each column of a multi-bit number.

Our adder works for the rightmost column because it has only two bits to add, but for the remaining columns we need to add two bits plus a potential carry from the previous column:



```
          1 1 1
  0 1 0 0 1 1 1 0  (78 decimal)
+ 0 0 1 0 1 0 1 1  (43 decimal)
  ---------------
  0 1 1 1 1 0 0 1  (121 decimal)
```

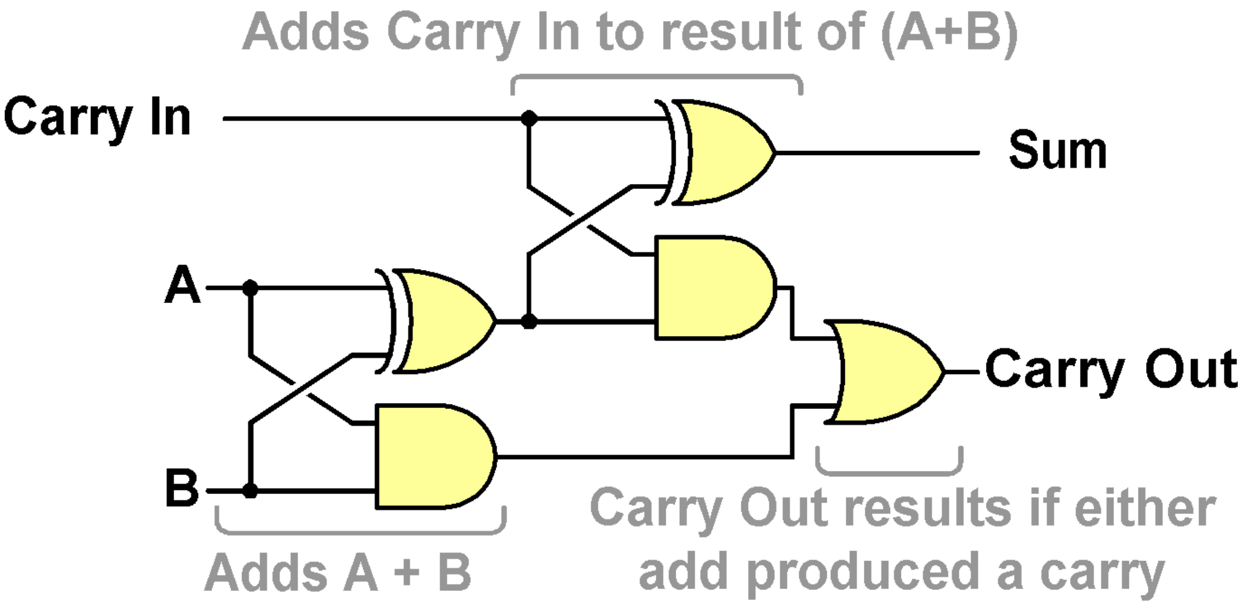
A Full Adder

The simple adder we just designed is called a "half adder". It can add only two bits.

For the rest of the columns we need to design a three-input adder to handle two data bits as well as a "carry in" from the previous column. This is known as a "full adder".

The truth table for a full adder is shown at the right, and below is it's circuit:

Carry In	A	B	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

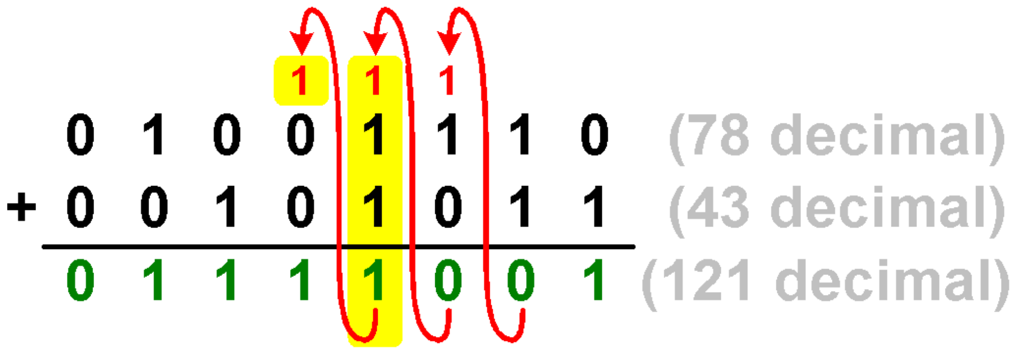

Adds Carry In to result of (A+B)

Carry Out results if either add produced a carry

Adds A + B

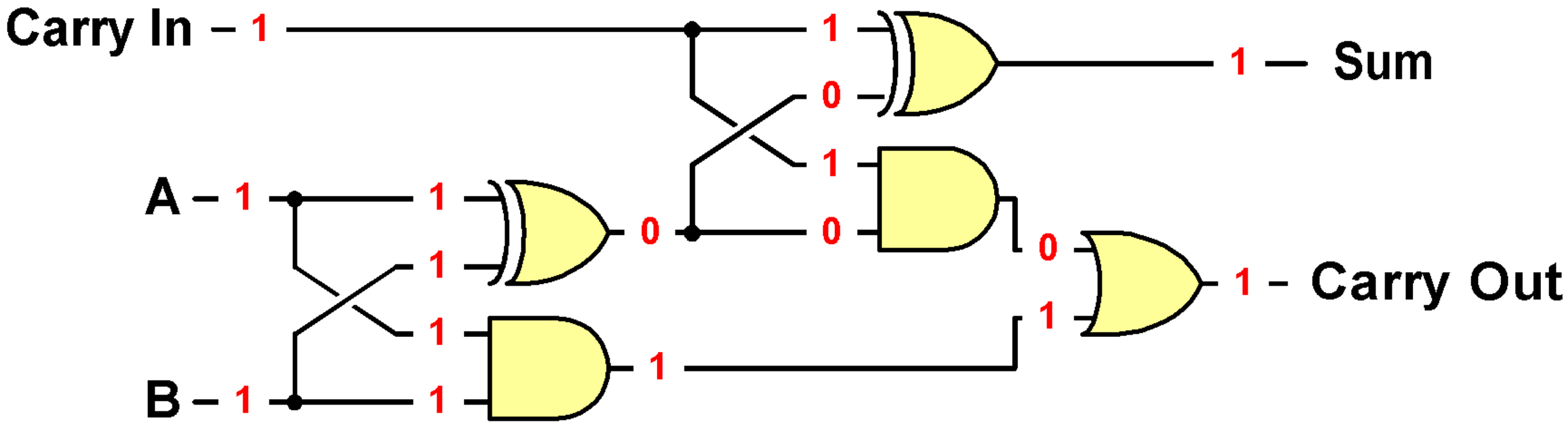
Full Adder Example

Here’s an example of our full adder adding one of the columns in a multi-bit number:



```
                1   1   1
    0   1   0   0   1   1   1   0   (78 decimal)
+   0   0   1   0   1   0   1   1   (43 decimal)
    0   1   1   1   1   0   0   1   (121 decimal)
```

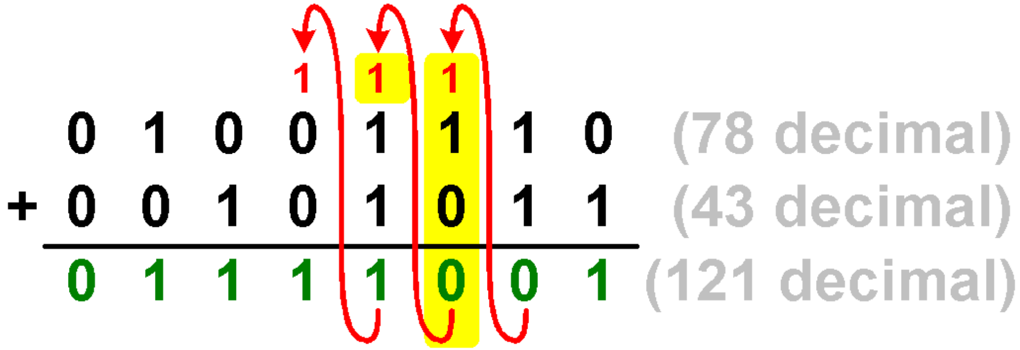
The input data bits (A and B) are “1”s, as is the Carry In bit:



The resulting sum is “1”, as is the Carry Out.

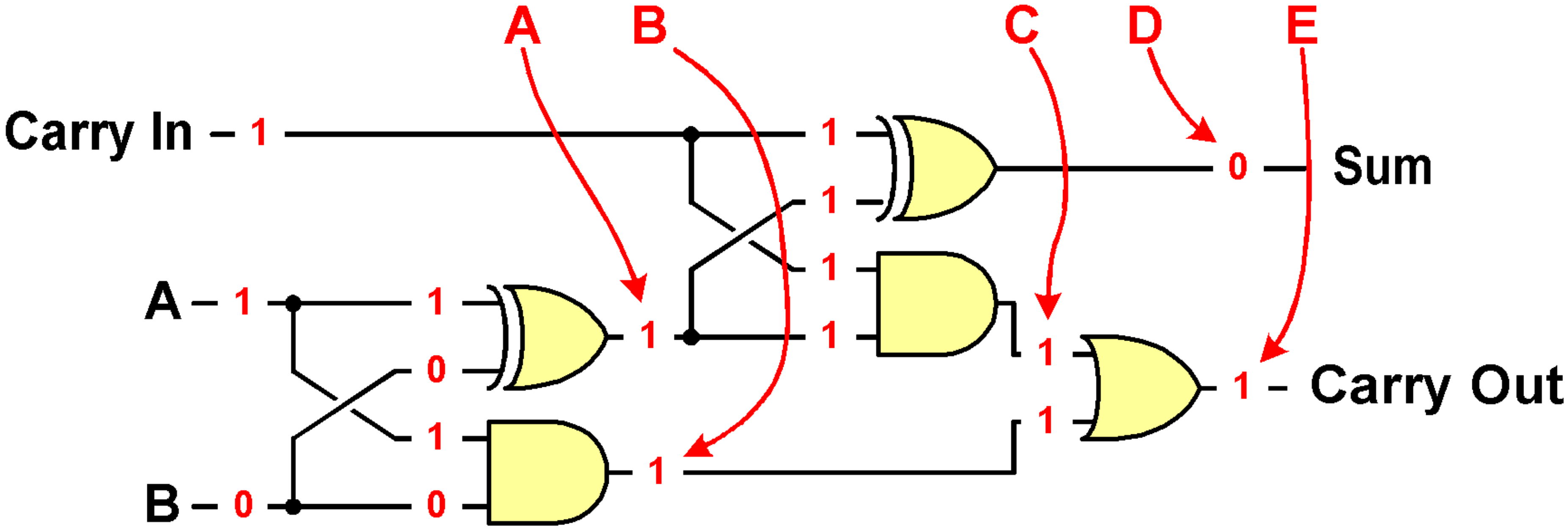
Exercise 4 - Full Adder

Our Full Adder is adding together the highlighted column of two multi-bit numbers:



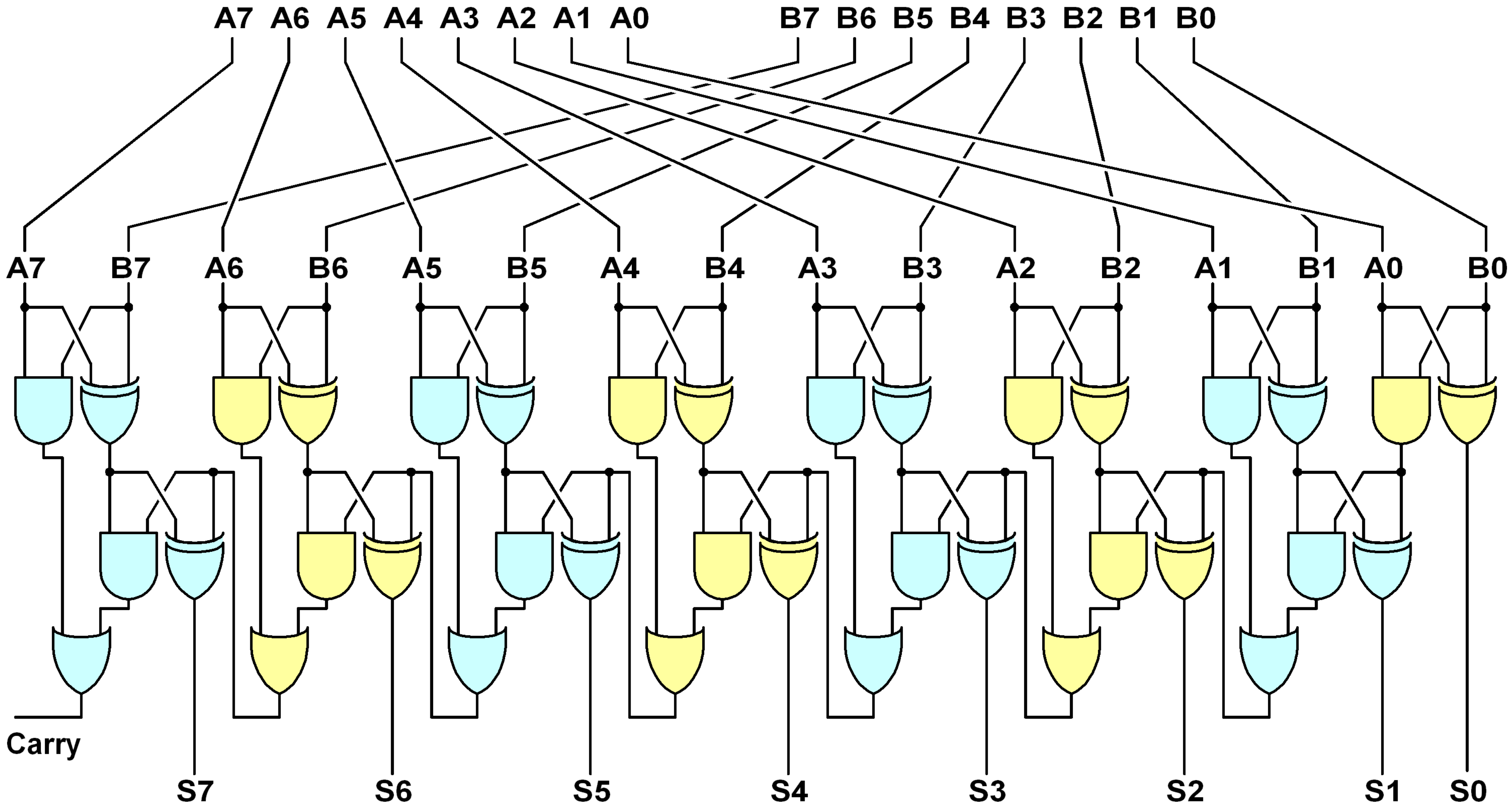
```
            1   1   1
  0 1 0 0   1   1   1 0   (78 decimal)
+ 0 0 1 0   1   0   1 1   (43 decimal)
  0 1 1 1   1   0   0 1   (121 decimal)
```

Which of the indicated values is incorrect?



An 8-bit Adder

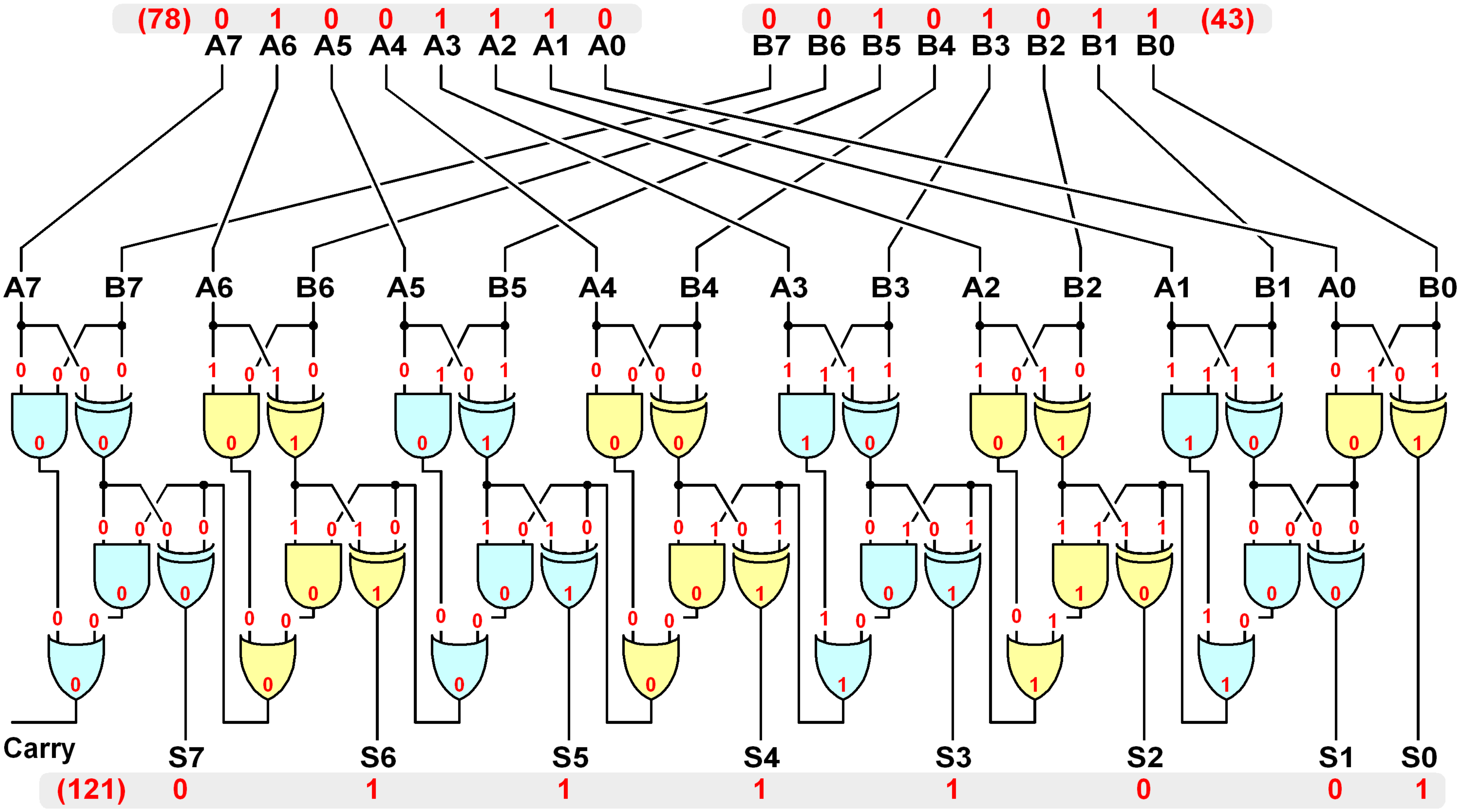
Here is an example of how to use multiple full adders to add multi-bit numbers:



Each stage of the adder has been given a different colour to make it clearer which gates each bit goes through. Notice that a half-adder is used for the low-order (rightmost bit) and full adders are used for all the other bits.

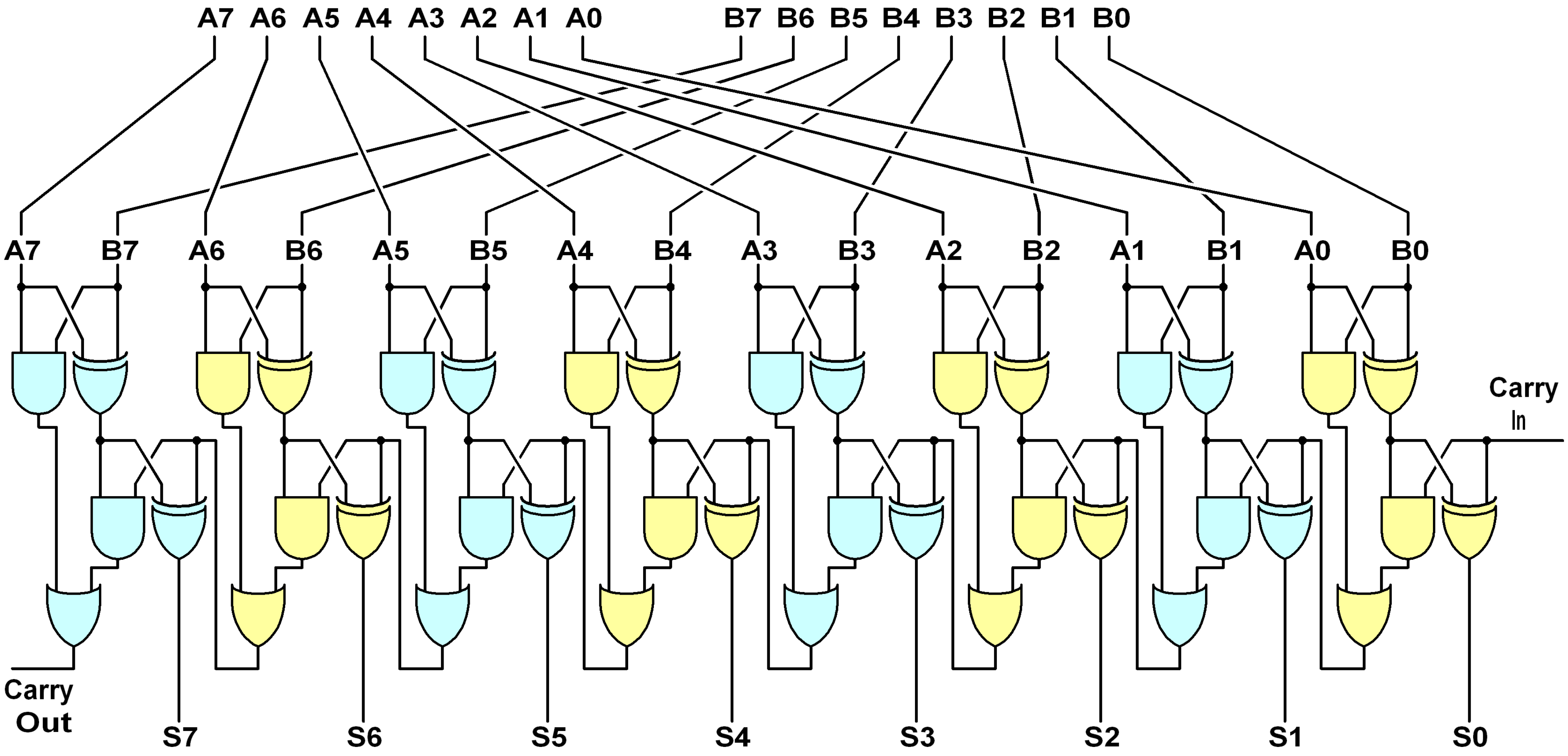
An 8-bit Adder - Example

This diagram shows an 8-bit adder adding the binary values for 78 and 43 and producing a binary result of 121. The flow of signals through the entire circuit is shown by the red “1”s and “0”s:



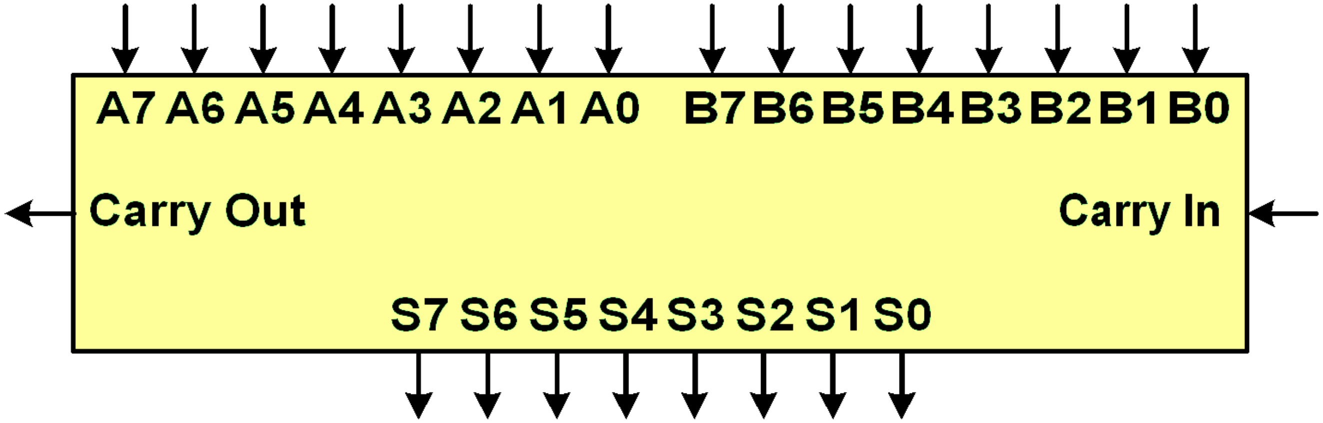
An 8-bit full Adder

In practice multi-bit adders are usually built with full adder stages even for the rightmost bit. This allows multiple adders to be "ganged" together to create an adder that can handle even larger numbers:

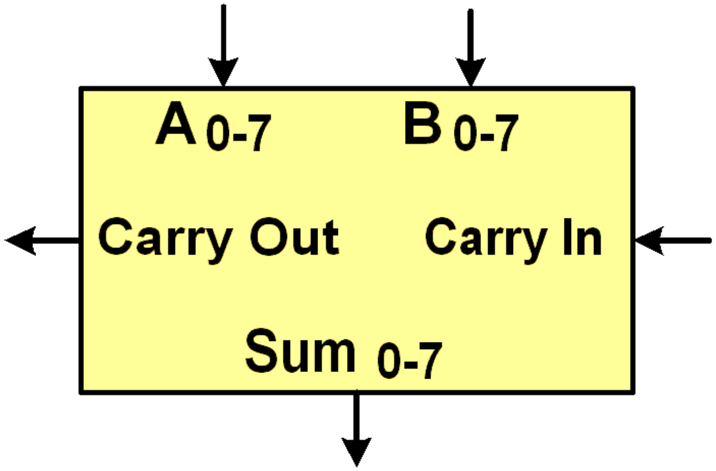


An 8-bit Adder

We can represent the entire 8-bit adder circuit with the following symbol:

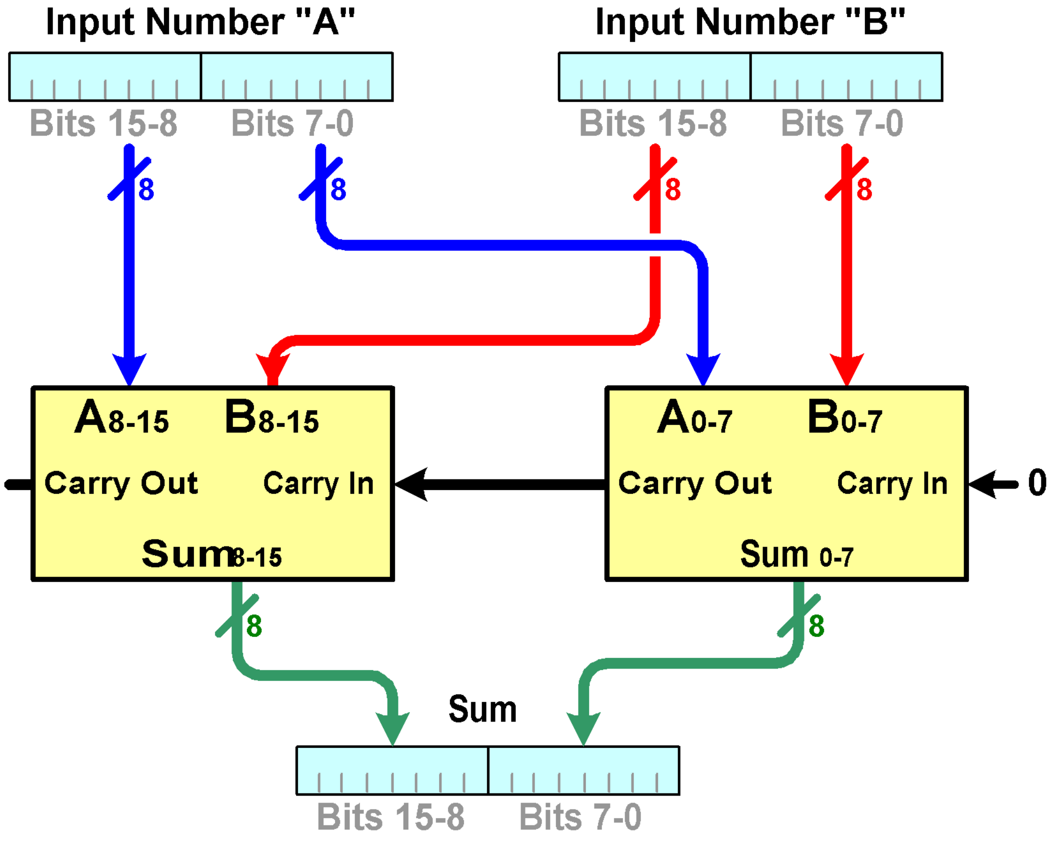


We can simplify the diagram even further by combining the 8 lines for the inputs and sum into single lines in our diagram. There are still actually 16 input lines and 8 output lines, but this is now an “abbreviated” diagram:



“Ganged” 8-bit Adders

We can combine two 8-bit adders together to add 16-bit numbers:

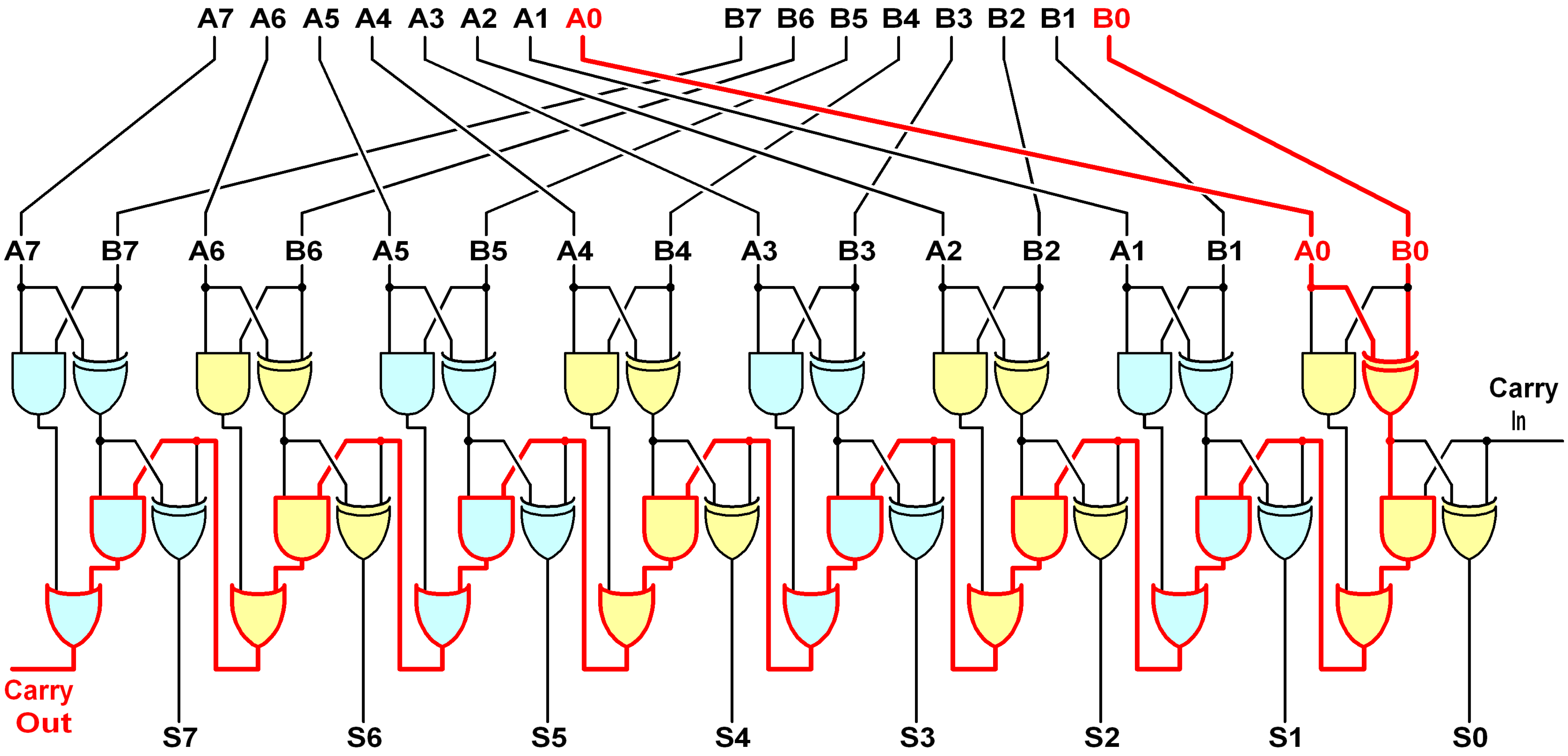


This diagram shows that:

- **The low-order (rightmost bits) of each of the two 16-bit input numbers go one adder, and the high-order (leftmost bits) go to the other.**
- **The carry out from the low-order adder goes to the carry-in of the high-order adder.**
- **The outputs from both adders are combined to form a 16-bit sum**

Adder Carry Delay

It takes a finite amount of time for the signals to pass though each gate – the correct answer isn't produced at the adder outputs until the input signals have passed through all the gates.



Bits are added in parallel, but the carry signal has to ripple through all of the stages and so it limits the speed of the adder. The more stages you add, the slower the adder is.

Speeding up the Adder

One way to speed up the adder is to do some of the work in parallel using extra circuitry. Let’s see an example of how this can work with decimal numbers in order to understand the technique.

Let’s say we want to add the following two decimal numbers together:

$$\begin{array}{r} 5\,2\,4\,6\,9\,1 \\ \underline{+\,2\,7\,3\,1\,0\,8} \end{array}$$

If it takes somebody one second to add each digit, then it will take a total of 6 seconds to add this number. To reduce the time needed, we can break the numbers in half – a high-order half and a low-order half – and add the two halves in parallel:

$$\begin{array}{rcr} \text{High-Order} & & \text{Low-Order} \\ 5\,2\,4 & & 6\,9\,1 \\ +\,\underline{2\,7\,3} & \text{and} & +\,\underline{1\,0\,8} \\ 7\,9\,7 & & 7\,9\,9 \end{array}$$

…then we can put the two results back together again to get the correct sum: 797799

If two people do these two additions at the same time (in parallel), the time it takes to add the two numbers together is now only 3 seconds.

But **– what happens if there is a carry out of the low order part of the number?**

$$\begin{array}{rcr} \text{High-Order} & & \text{Low-Order} \\ 5\,2\,4 & & 6\,9\,1 \\ +\,\underline{2\,7\,3} & \text{and} & +\,\underline{4\,0\,8} \\ 7\,9\,7 & & 1\quad 0\,9\,9 \end{array}$$

Speeding up the Adder

A carry out of the low order part of the number will ruin our scheme. But note that there are only two possibilities for the high-order part of the sum: the sum <u>with</u> a carry from the low-order part of the number, or the sum <u>without</u> a carry from the low-order part.

We can make our technique work by doing <u>three</u> additions in parallel:

- **The low-order part of the number**
- **The high-order part of the number with <u>no</u> carry in**
- **The high-order part of the number <u>with</u> a carry in (in other words, "+1")**

Here's an example showing how we'd add these two numbers:

$$\begin{array}{r} 5\,2\,4\,6\,9\,1 \\ \underline{+\,2\,7\,3\,4\,0\,8} \end{array}$$

We break these into low and high-order halves and add the high-order half twice, once with and once without a carry in:

High Order with Carry	High-Order without Carry	Low-Order
1		
5 2 4	5 2 4	6 9 1
+ 2 7 3	+ 2 7 3	+ 4 0 8
7 9 8	7 9 7	1 0 9 9

After doing the three additions, we look at the low-order sum and see that there was a carry out from it. So we choose the high-order sum that included the carry and put the results back together: 798099

We now need 3 people to do additions in parallel, but we can get the right result in 3 seconds.

Carry Select Adder

A carry select adder uses exactly this same technique. It breaks the addition of two binary numbers into two halves – a low-order half and a high-order half:

Instead of:

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

The carry select adder performs three additions:

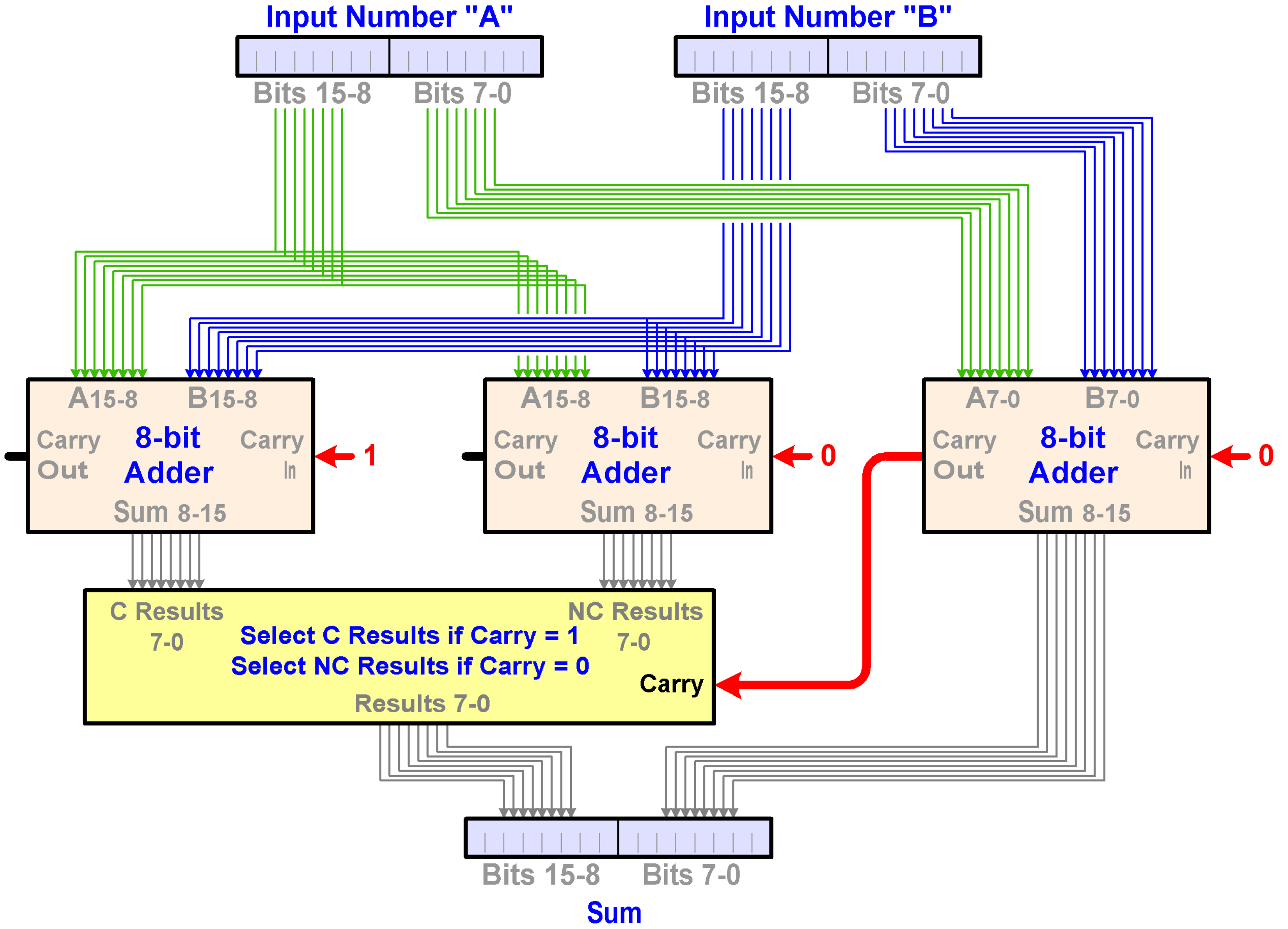
- **The low-order part of the number**
- **The high-order part of the number with no carry in**
- **The high-order part of the number with a carry in (in other words, "+1")**

Hi-Order + carry		High-Order		Low-Order
1 0 0 1 0 1 1 0		1 0 0 1 0 1 1 0		
0 1 0 0 1 0 1 1	and	0 1 0 0 1 0 1 1	and	1 0 0 1 0 1 0 1
+ 1		+ 0		+ 0 1 0 0 1 0 1 1
1 1 1 0 0 0 1 0		1 1 1 0 0 0 0 1		1 1 1 0 0 0 0 0

Finally, it chooses which high-order half of the number to use based on whether there was a carry out of the low-order half.

Carry Select Adder

Here's how to build a 16-bit carry select adder using three 8-bit adder circuits:

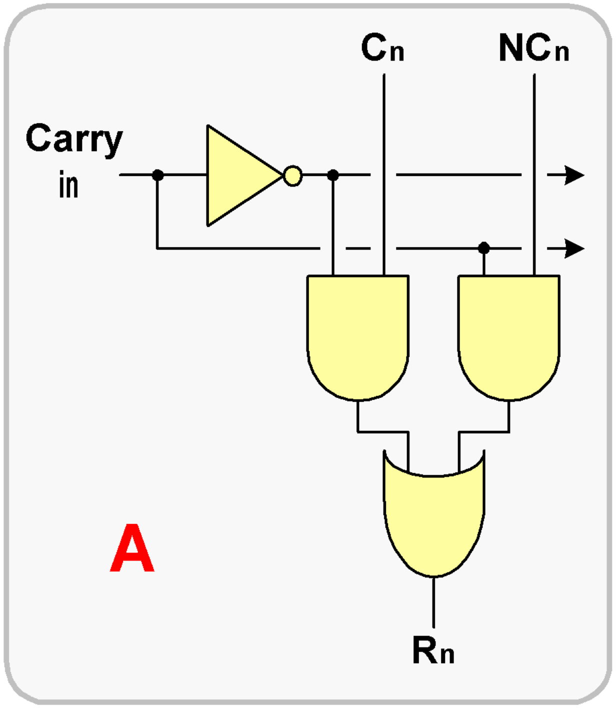


The low-order 8 bits are added as usual, but the high order 8 bits are added twice – with and without a Carry in. The yellow circuit selects which of the high-order sums to use in the result.

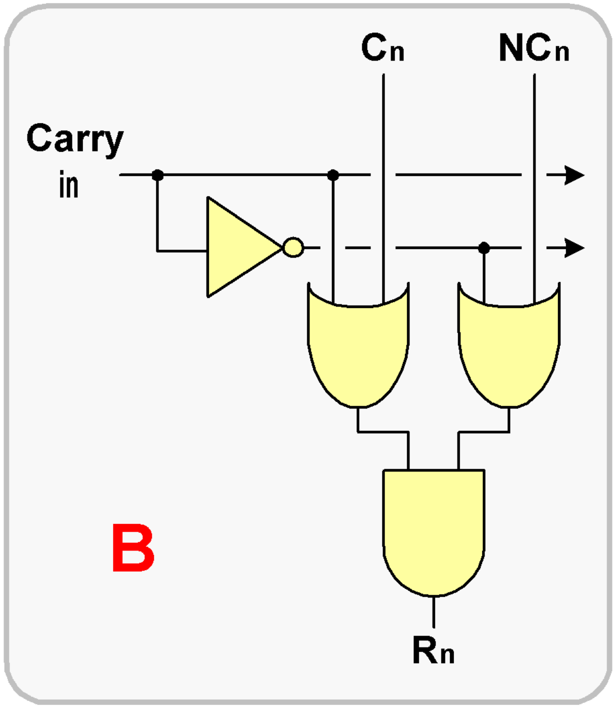
Exercise 5 - Carry Select Adder

Which of the following circuits represents the logic of the select portion of the Carry Select Adder (yellow box on the previous page)? Remember the purpose of the circuit is:

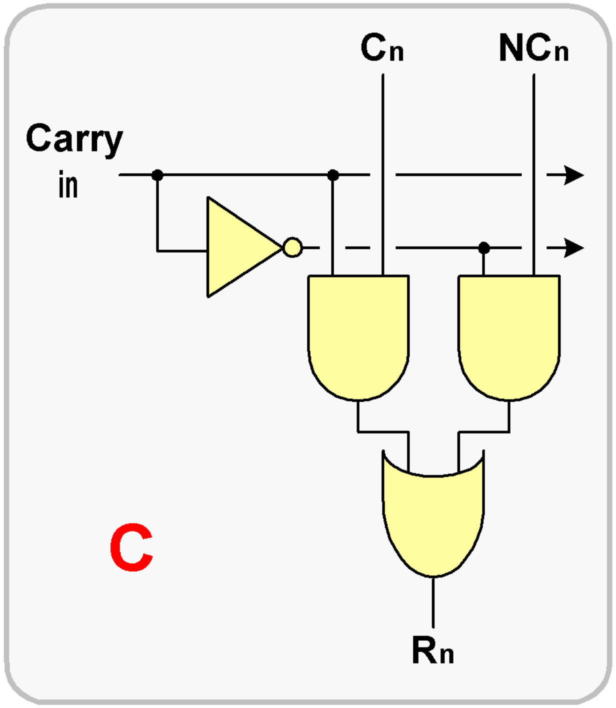
- **When Carry = FALSE, connect the "NCn" (no carry) inputs to the "Rn" (result) outputs**
- **When Carry = TRUE, connect the "Cn" (carry) inputs to the "Rn" (result) outputs**



A



B



C

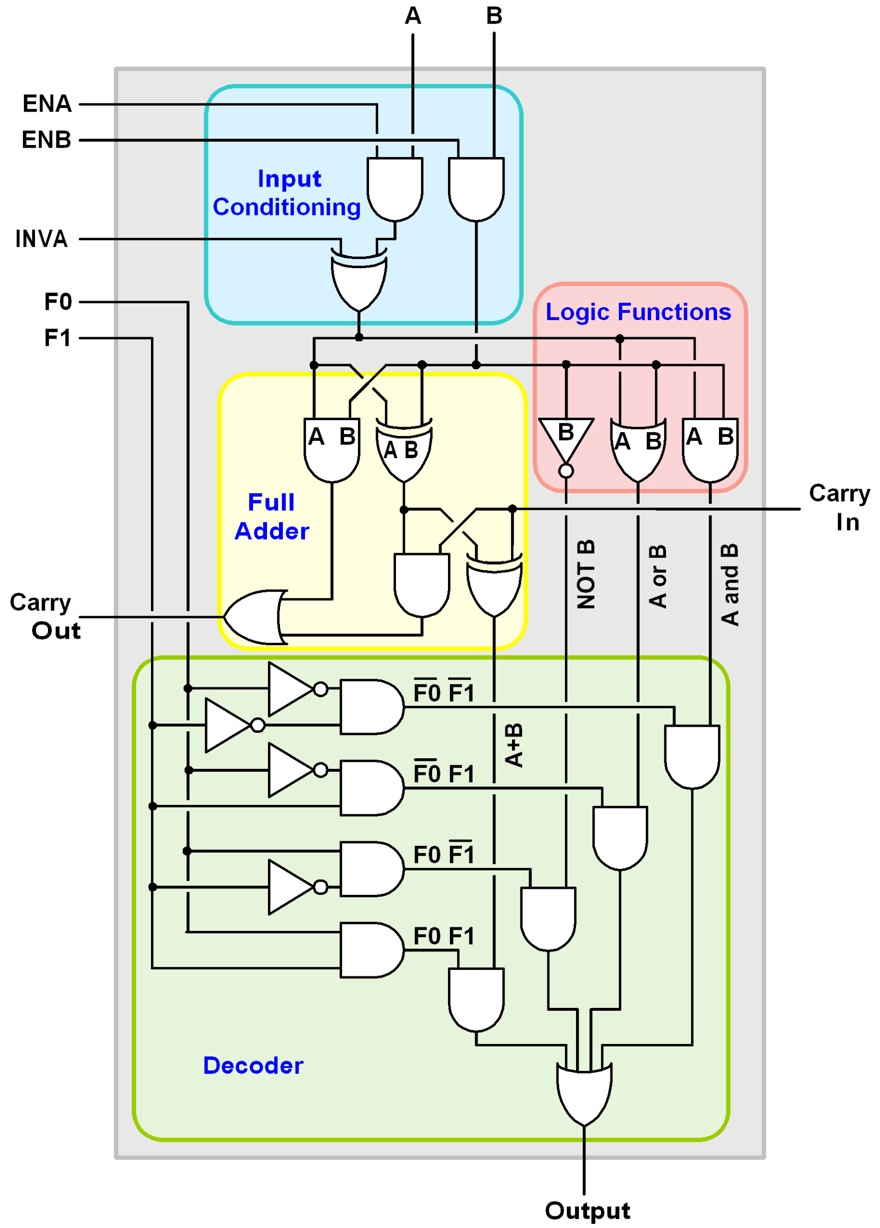
An ALU Bit Slice

An Arithmetic Logic Unit (ALU) is the heart of the computer system. It handles the arithmetic and logic functions needed to execute instructions.

An ALU is built from “bit slices” that perform the processing for a single bit of a multi-bit number. A diagram of an ALU bit slice is shown at right. This is equivalent to the circuit shown on page 156 of the textbook.

The ALU has the following input and output signals:

A,B	The input data
Carry In	Carry from the previous bit position
Carry Out	Carry to the next bit position
ENA, ENB, INVA	Input conditioning
F0, F1	Function code
Output	The ALU result

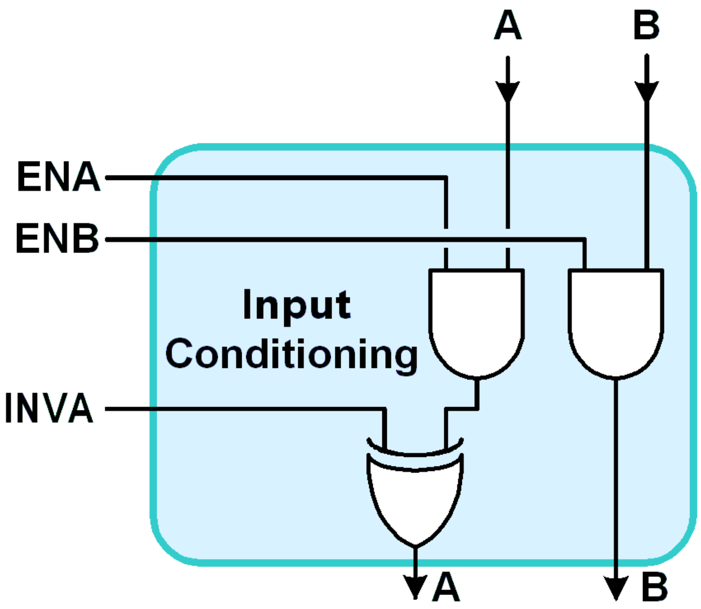


ALU Bit Slice – Input Conditioning

The input conditioning logic accepts three control signals which can be used to modify the A and B inputs.

- **ENA** – Passes the A input through unchanged if TRUE, or forces it to zero if FALSE.
- **ENB** – Passes the B input through unchanged if TRUE, or forces it to zero if FALSE
- **INVA** – Passes the A input unchanged if FALSE, or inverts it's value if TRUE

The truth tables below show how the Boolean AND and XOR functions are used to achieve these results:



ENA	A	ENA · A
0	0	0
0	1	0
1	0	0
1	1	1

Result = A
when ENA is TRUE

ENB	B	ENB · B
0	0	0
0	1	0
1	0	0
1	1	1

Result = B
when ENB is TRUE

INVA	A	INVA Å A
0	0	0
0	1	1
1	0	1
1	1	0

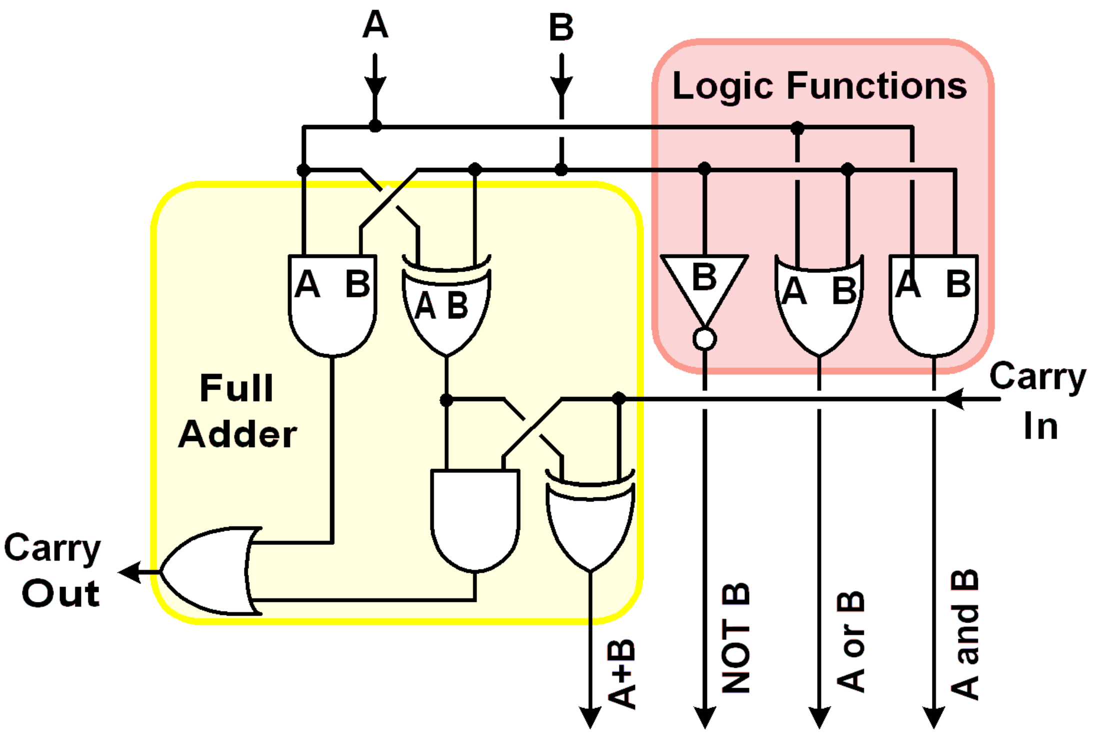
Result = NOT-A
when INVA is true

Being able to control the A and B inputs like this adds a lot of flexibility to our ALU and allows it to do some things that would otherwise be quite difficult, as we shall soon see.

ALU Bit Slice – Functional Units

After passing through the input conditioning logic, the A and B inputs are sent to the inputs of four different functional units at the same time:

- **Adder** – Produces the arithmetic sum of the A and B inputs. (This unit also handles carry in and carry out from other ALU stages).
- **NOT** – Produces the Boolean NOT of the B input only
- **OR** – Produces the Boolean OR of the A and B inputs
- **AND** – Produces the Boolean AND of the A and B inputs

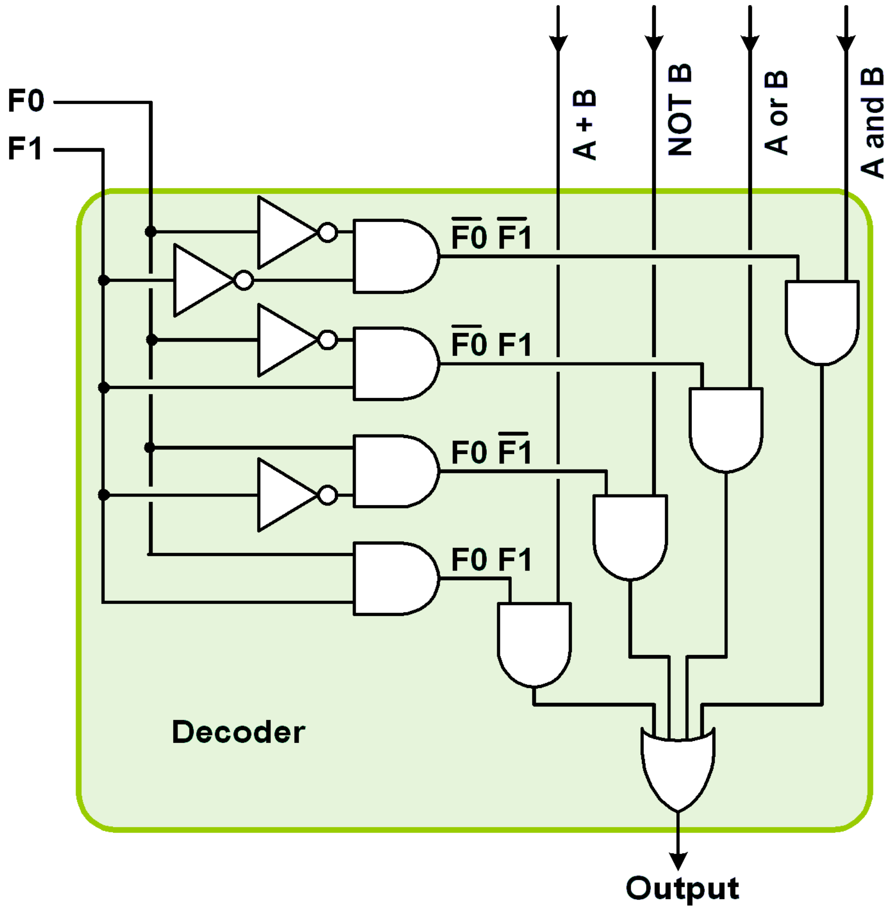


The ALU produces all four of these functions at the same time and passes them onward to the decoder.

ALU Bit Slice - Decoder

The decoder chooses which of the four functions to pass through to the output of the ALU. It's controlled by two function control lines called F0 and F1. The state of these two control lines selects which logic function is selected:

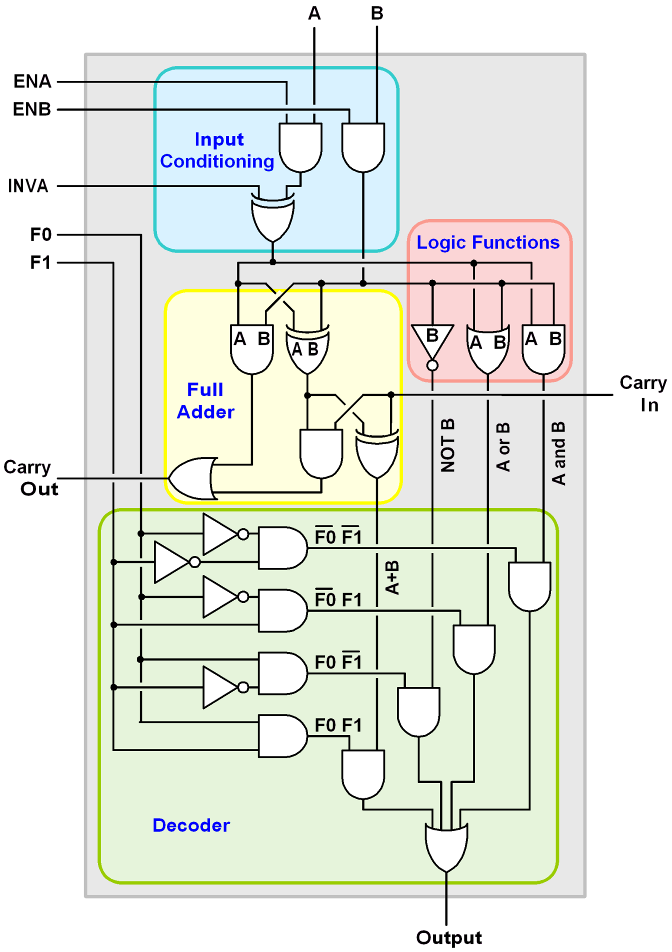
F0	F1	Logic Function
0	0	A and B
0	1	A or B
1	0	NOT B
1	1	A + B

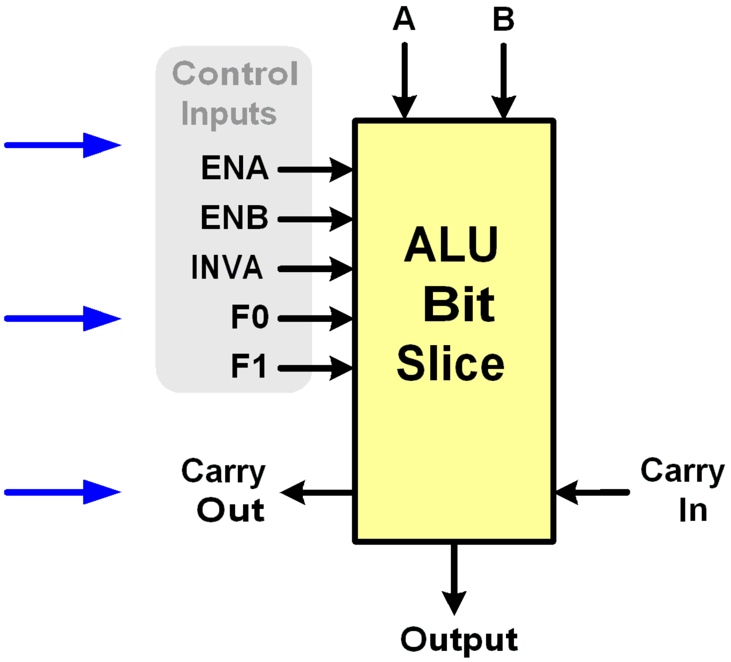


Although the ALU always generates all four different results, the decoder selects only one of them which is passed through to the output stage. And so the F0 and F1 inputs control which logic operation is used within the ALU.

An ALU Bit Slice

All of the logic inside the entire ALU bit slice on the left can be represented by a single symbol as shown on the right:

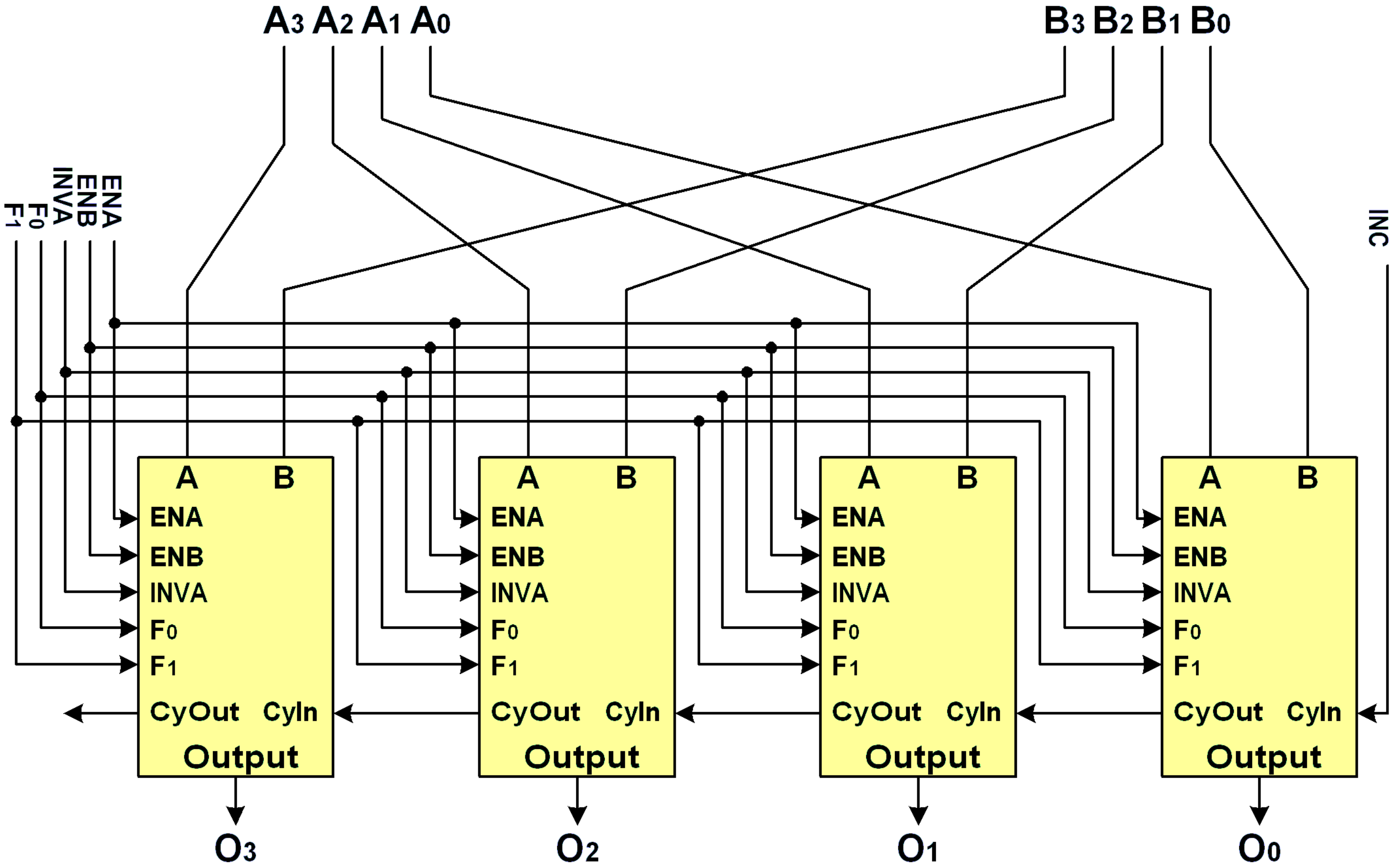




The five control input signal connections control what function the ALU bit slice performs, while the other signals carry data into and out of the slice.

Complete 4-bit ALU

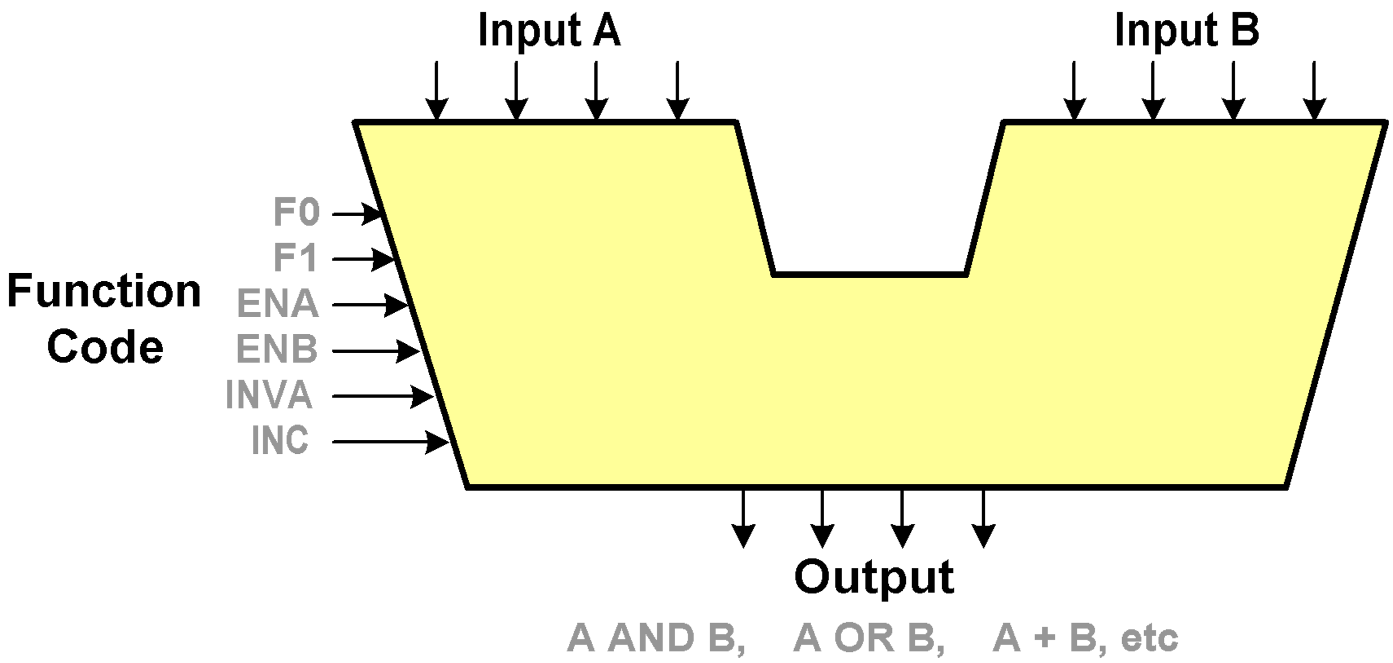
Multiple ALU bit slices can be combined together to form a complete ALU which can perform operations on multi-bit numbers. Here's an example of an ALU made from 4 bit slices which can handle 4-bit numbers:



The five control input signals are connected to all four bit slices so they all perform the same function. A sixth control signal called "INC" is connected to the "Carry In" for the low-order bit slice – this signal can be used to increment (add 1 to) a number.

Symbol for a Complete ALU

The following symbol is used to show a complete Arithmetic Logic Unit built from four bit slices:



An ALU has:

- **Multiple-bit "A" and "B" inputs to accept two numbers**
- **A number of function code inputs that determine what function the ALU performs**
- **A multiple-bit output which produces the resulting number according to the selected function**

By adding more bit slices you can create an ALU that has 8, 16, or 32-bit inputs and outputs, or even larger.

ALU Function Codes

3.2.3

Figure 4-2 on page 234 of the textbook shows the various functions that the ALU can perform for various combinations of control inputs:

F0	F1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	NOT-A
1	0	1	1	0	0	NOT-B
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

ALU Function Codes – Logical Functions

The function code inputs cause the ALU to performs various logical functions as described below. Note that the first two combinations allow the A or B input to pass through the ALU without being changed.

F0	F1	ENA	ENB	INVA	INC	Function	Explanation
0	1	1	0	0	0	A	Only A is enabled, so B is input as “0”s The OR function is selected, the A bits are not changed when they’re ORed with “0”s.
0	1	0	1	0	0	B	Only B is enabled, so A is input as “0”s The OR function is selected, the B bits are not changed when they’re ORed with “0”s.
0	1	1	0	1	0	NOT-A	Only A is enabled, so B is input as “0”s. INVA is enabled, this inverts the A bits. The OR function is selected, and the inverted A bits aren’t changed when ORed with “0”s.
1	0	1	1	0	0	NOT-B	The “NOT-B” function is selected, so A inputs are ignored and the inverted B bits are output.
0	0	1	1	0	0	A AND B	A and B inputs are both enabled and the AND function is selected
0	1	1	1	0	0	A OR B	A and B inputs are both enabled and the OR function is selected.

Exercise 6 - ALU Function Codes

What is the output for the following set of ALU function codes?

F0	F1	ENA	ENB	INVA	INC
0	0	1	0	0	0

F0	F1	Logic Function
0	0	A and B
0	1	A or B
1	0	NOT B
1	1	A + B

A – the result is the A input

B – the result is the B input

C – the result is A AND B

D – the result is A OR B

E – none of the above

ALU Function Codes – Unsigned Constants

The function code inputs can be set to cause two unsigned constant values to come out of the ALU as described below:

F0	F1	ENA	ENB	INVA	INC	Function	Explanation
0	1	0	0	0	0	0	Neither A or B inputs are enabled, so both numbers are input as “0”s. The “OR” function is selected, and “0”s ORed with “0”s produces a result of “0”s
1	1	0	0	0	1	1	Neither A nor B inputs are enabled, so both numbers are input as “0”s. The “ADD” function is selected and INC is also set. INC is connected to the “carry in” of the low-order bit, which is equivalent to adding one to the ALU value. The result is: 0(A) + 0(B) + 1(INC) = 1

ALU Function Codes – Arithmetic Functions

The ALU performs logical functions as follows:

F0	F1	ENA	ENB	INVA	INC	Function	Explanation
1	1	1	1	0	0	A + B	The A and B inputs are both enabled and the function code is “ADD”
1	1	1	1	0	1	A + B + 1	The A and B inputs are both enabled and the function code is “ADD”. In addition, the INC signal is set and this acts as if an additional “1” is added to the ALU result.
1	1	1	0	0	1	A + 1	The B input is disabled and so it is input as “0”s. The function code is “ADD”, and the INC signal is set and this acts as if an additional “1” is added to the ALU result.
1	1	0	1	0	1	B + 1	The A input is disabled and so it is input as “0”s. The function code is “ADD”, and the INC signal is set and this acts as if an additional “1” is added to the ALU result.

There are also several combinations of control signals which allow operations such as subtraction.   To understand how these work, we must learn the format for signed numbers.

Signed Binary Numbers

Until now, all of the binary numbers we've seen have been <u>unsigned numbers</u>. Unsigned numbers can <u>never be less than zero</u>.

Computers must often deal with numbers which can have negative values. These types of numbers are called <u>signed numbers</u>, and we need some way to store such numbers and perform calculations with them.

Most computers store signed numbers using <u>two's complement</u> notation. This notation is used for signed numbers because the circuitry to handle them is very simple.

A number stored in two's complement notation can be <u>positive or negative</u>. Note that signed numbers are not always negative!

The leftmost bit (the high order bit) of the number tells you if the number is positive or negative:

When the leftmost bit is "<u>1</u>", the number is <u>negative</u>

When the leftmost bit is "<u>0</u>", the number is <u>positive</u>

0	1	1	0	0	1	1	1

A positive signed 8-bit number

1	0	0	1	0	1	1	0

A negative signed 8-bit number

Signed vs. Unsigned Binary Numbers

A number is not necessarily negative just because it's leftmost bit is "1". In an unsigned number, the high-order bit holds part of the value of the number. Consider this unsigned number, for example:

1	0	0	1	0	1	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

The value of this number is:

$$\begin{aligned} & 2^7 + 2^4 + 2^2 + 2^1 \\ = \; & 128 + 16 + 4 + 2 \\ = \; & 150 \end{aligned}$$

So how can you tell if a number is signed or unsigned?

You can't tell just by looking at the number any more than you could tell if "APT203" written on a piece of paper is an apartment number or a license plate number.

Converting Between Negative and Positive

For signed numbers, you can convert a number between it's positive and negative form by performing these two steps in order:

- **Invert all of the bits of the number (0's become 1's, 1's become 0's)**
- **Add 1 to the number**

This is called "taking the two's complement of the number". Here is an example:

(decimal +12)		0	0	0	0	1	1	0	0
Step 1 - Invert all of the bits:		1	1	1	1	0	0	1	1
Step 2 - Add 1:	+	0	0	0	0	0	0	0	1
(decimal –12)		1	1	1	1	0	1	0	0

Note that the two's complement operation works both ways:

Step 1 - Invert all of the bits:		0	0	0	0	1	0	1	1
Step 2 - Add 1:	+	0	0	0	0	0	0	0	1
(decimal +12)		0	0	0	0	1	1	0	0

These examples show 8-bit numbers, but 16, 32 or 64-bit numbers work exactly the same way.

Negative One

The two’s complement value for “–1” has a special, easily recognizable value:

(decimal +1)		0	0	0	0	0	0	0	1
Step 1 - Invert all of the bits:		1	1	1	1	1	1	1	0
Step 2 - Add 1:	+	0	0	0	0	0	0	0	1
(decimal –1)		1	1	1	1	1	1	1	1

And, of course, we can go the other way too:

Step 1 - Invert all of the bits:		0	0	0	0	0	0	0	0
Step 2 - Add 1:	+	0	0	0	0	0	0	0	1
(decimal +1)		0	0	0	0	0	0	0	1

This pattern holds true no matter how many bits (8, 16, 32, etc) are in the number.

So, when you’re dealing with signed numbers, remember that a number whose bits are all “1”s is the value “–1”.

Using Two's Complement for Subtraction

You can subtract one number from another by adding it's two's complement. This makes it possible to perform subtraction using the same circuits that are used for addition. Let's see an example:

(carries)	1	1	1	1					
(decimal +19)		0	0	0	1	0	0	1	1
(decimal –12)		1	1	1	1	0	1	0	0
(decimal +7)		**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**

Note that there is a carry out of the high order bit position. For unsigned numbers a carry out of the high order bit position indicates that the resulting number is too big to fit.

But for signed numbers a carry out of the high-order bit position is not necessarily an error. In fact, for signed numbers, the indication of an error is that the carry into the high order bit is different than the carry out of the high order bit. This is often called an "overflow".

Exercise 7 - Signed Numbers

1: What is the 8-bit binary value that’s equivalent to decimal –6?

(decimal +6)	0	0	0	0	0	1	1	0

A	1	0	0	0	0	1	1	0
B	1	1	1	1	1	0	1	0
C	1	1	1	1	1	0	0	1

2: Convert the following 8-bit signed number to decimal:

1	1	1	1	1	0	0	1

A: 5 B: 6 C: 7 D: 8

E: –5 F: –6 G: –7 H: –8

3: What is the sum of the following two 8-bit signed numbers?

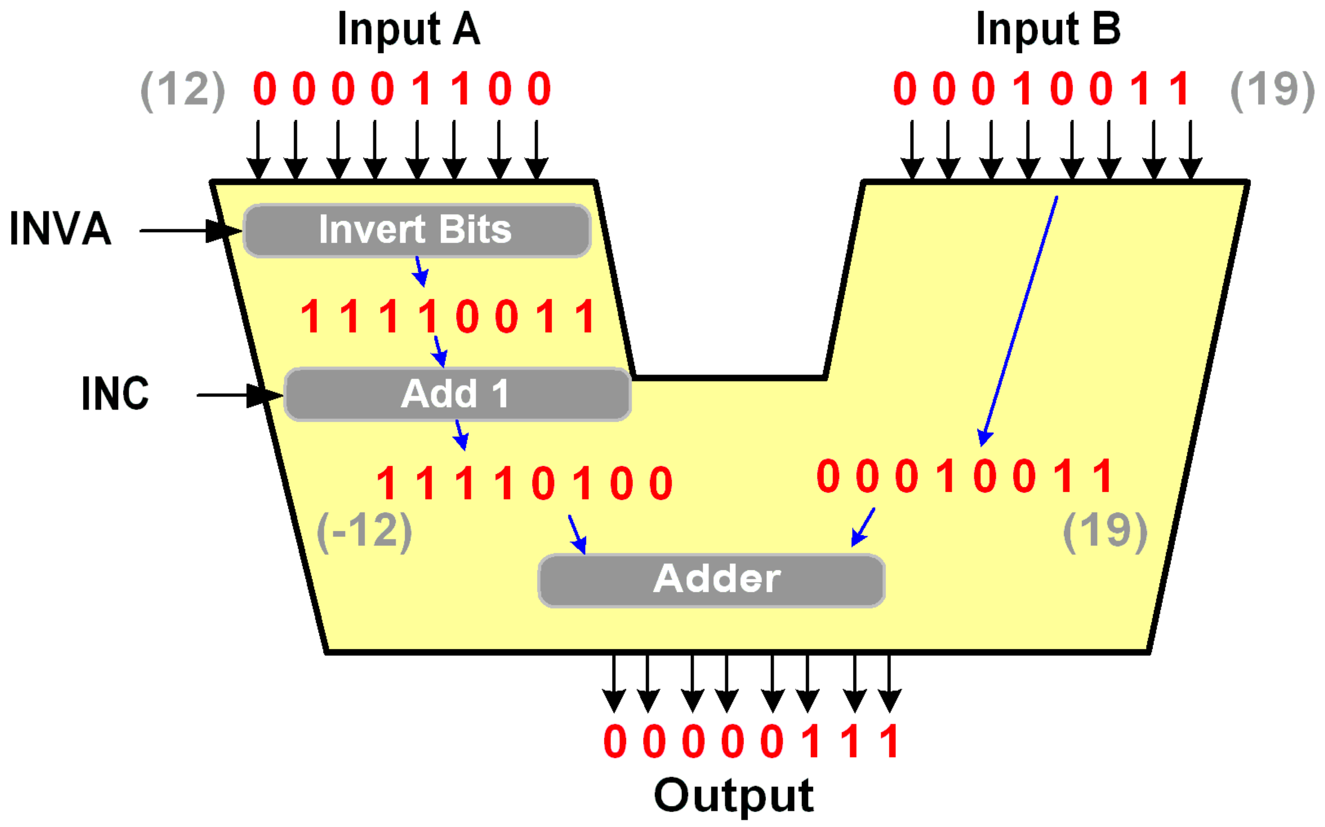
(decimal –42)	1	1	0	1	0	1	1	0
(decimal –17)	1	1	1	0	1	1	1	1

A	1	1	0	0	0	1	0	1
B	0	1	0	0	0	1	0	1
C	1	0	1	1	1	0	1	1

Understanding how the ALU subtracts

Let's look in more detail at how the ALU subtracts using two's complement notation. This diagram shows how an 8-bit ALU performs the calculation "19 – 12 = 7":

The input conditioning logic of the ALU is used to perform a two's-complement operation on the A input, negating it's value. The negative value of A is added to the B, and the result is the same as "B – A".



Note - the "Add 1" isn't really done in a separate stage as shown here – it's actually done in the "Adder" stage because INC is connected to the low-order "Carry In" input of the adder. But the result is the same no matter where the INC is done.

ALU Function Codes – Signed Functions

Now that we know how signed numbers are stored, we can understand how the ALU can subtract and deal with signed numbers even though it has no special circuitry for doing so:

F0	F1	ENA	ENB	INVA	INC	Function	Explanation
1	1	1	1	1	1	B – A	The A and B inputs are both enabled and the "ADD" function code is selected. In addition, the "INVA" and "INC" signals are also set – this has the effect of inverting the A input and adding 1 to it – so the A input is converted to it's two's complement (negative) form. When –A is added to B the result is B – A.
1	1	0	1	1	0	B – 1	The A input is not enabled so it is input as "0" bits. The INVA signal is on, so the "0" bits are inverted and become "1" bits. Therefore, the "A" input has a signed value of "–1" (all bits are "1"). The function code is "add", and adding "–1" to B gives the result B – 1.

Note that our ALU cannot perform an "A – B" or "A – 1" function, because there is no "INVB" capability built into it. An ALU with this function could be built, but our ALU doesn't have it.

ALU Function Codes – Signed Functions (cont’d)

Here are the last two functions that the ALU can perform:

F0	F1	ENA	ENB	INVA	INC	Function	Explanation
1	1	1	0	1	1	–A	The B input is disabled and is input as “0”s. the “INVA” and “INC” signals are also set – this has the effect of inverting the A input and adding 1 to it – so the A input is converted to it’s two’s complement (negative) form. The function code is “add”, and zero plus “–A” gives the result “–A”.
0	1	0	0	1	0	–1	The A and B inputs are both disabled and are input as “0”s. The “INVA” signal causes the “0” A input to be converted to all “1” bits. The function code is “OR”. After the A “1” bits are ORed with the B “0” bits, the resulting output is all “1” bits. This is the same as a signed value of “–1”.

Key Concepts

- **How multiplexers and decoder circuits work**
- **Using a multiplexer circuit to solve Boolean logic problems**
- **Using programmable logic arrays to solve Boolean logic problems**
- **How comparator and shifter circuits work**
- **How a half-adder circuit works**
- **The difference between a half-adder and a full adder**
- **How a carry select adder improves performance**
- **How an ALU bit slice works**
- **How ALU bit slices are combined to handle multi-bit numbers**
- **How the ALU function codes work**
- **Signed vs. Unsigned binary numbers and 2's complement notation**
- **How the ALU is able to subtract by using two's complement notation**

What’s Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 3 – “Disk Technology”**
- **Study for Quiz 2, which covers:**
  - **Week 3 lecture**
  - **WebCT module 3**
- **Continue working on Assignment 1 (for the material covered so far)**
- **Pre-read week 4 material**