Fundamentals of digital Systems

ERMIT Departement, M. Bidnyi, Lecture synopsis, 2020

Number Systems

Roman Numerals

The Romans devised a system that was a substantial improvement over hash marks, because it used a variety of symbols (or *ciphers*) to represent increasingly large quantities.

The notation for 1 is the capital letter I. The notation for 5 is the capital letter V. Other ciphers possess increasing values:

X = 10

L = 50

C = 100

D = 500

M = 1000

If a cipher is accompanied by another cipher of equal or lesser value to the immediate right of it, with no ciphers greater than that other cipher to the right of that other cipher, that other cipher’s value is added to the total quantity.

Thus, VIII symbolizes the number 8, and CLVII symbolizes the number 157. On the other hand, if a cipher is accompanied by another cipher of lesser value to the immediate left, that other cipher’s value is *subtracted* from the first. Therefore, IV symbolizes the number 4 (V minus I), and CM symbolizes the number 900 (M minus C).

You might have noticed that ending credit sequences for most motion pictures contain a notice for the date of production, in Roman numerals. For the year 1987, it would read: MCMLXXXVII. Let’s break this numeral down into its constituent parts, from left to right:

M = 1000

+

CM = 900

+

L = 50

+

XXX = 30

+

V = 5

+

II = 2

Aren’t you glad we don’t use this system of numeration? Large numbers are very difficult to denote this way, and the left vs. right / subtraction vs. addition of values can be very confusing, too.

Another major problem with this system is that there is no provision for representing the number zero or negative numbers, both very important concepts in mathematics.

Roman culture, however, was more pragmatic with respect to mathematics than most, choosing only to develop their numeration system as far as it was necessary for use in daily life.

Place Value

We owe one of the most important ideas in numeration to the ancient Babylonians, who were the first (as far as we know) to develop the concept of cipher position, or place value, in representing larger numbers.

Instead of inventing new ciphers to represent larger numbers, as the Romans did, they re-used the same ciphers, placing them in different positions from right to left.

Our own decimal numeration system uses this concept, with only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in “weighted” positions to represent very large and very small numbers.

Each cipher represents an integer quantity, and each place from right to left in the notation represents a multiplying constant, or *weight*, for each integer quantity.

For example, if we see the decimal notation “1206”, we known that this may be broken down into its constituent weight-products as such:

1206 = 1000 + 200 + 6

1206 = (1 x 1000) + (2 x 100) + (0 x 10) + (6 x 1)

Each cipher is called a *digit* in the decimal numeration system, and each weight, or place value, is ten times that of the one to the immediate right.

So, we have a *ones* place, a tens place, a *hundreds* place, a thousands place, and so on, working from right to left.

Right about now, you’re probably wondering why I’m laboring to describe the obvious. Who needs to be told how decimal numeration works, after you’ve studied math as advanced as algebra and trigonometry?

The reason is to better understand other numeration systems, by first knowing the how’s and why’s of the one you’re already used to.

The decimal numeration system uses ten ciphers, and place-weights that are multiples of ten. What if we made a numeration system with the same strategy of weighted places, except with fewer or more ciphers?

Binary Numeration

The binary numeration system is such a system. Instead of ten different cipher symbols, with each weight constant being ten times the one before it, we only have *two* cipher symbols, and each weight constant is *twice* as much as the one before it.

The two allowable cipher symbols for the binary system of numeration are “1” and “0,” and these ciphers are arranged right-to-left in doubling values of weight. The rightmost place is the *ones* place, just as with decimal notation. Proceeding to the left, we have the *twos* place, the *fours* place, the *eights* place, the *sixteens* place, and so on.

For example, the following binary number can be expressed, just like the decimal number 1206, as a sum of each cipher value times its respective weight constant:

11010 = 2 + 8 + 16 = 26

11010 = (1 x 16) + (1 x 8) + (0 x 4) + (1 x 2) + (0 x 1)

This can get quite confusing, as I’ve written a number with binary numeration (11010), and then shown its place values and total in standard, decimal numeration form (16 + 8 + 2 = 26). In the above example, we’re mixing two different kinds of numerical notation.

To avoid unnecessary confusion, we have to denote which form of numeration we’re using when we write (or type!). Typically, this is done in subscript form, with a “2” for binary and a “10” for decimal, so the binary number 11010_2 is equal to the decimal number 26_{10}.

The subscripts are not mathematical operation symbols like superscripts (exponents) are. All they do is indicate what system of numeration we’re using when we write these symbols for other people to read. If you see “3_{10}”, all this means is the number three written using *decimal* numeration.

However, if you see “3^{10}”, this means something completely different: three to the tenth power (59,049). As usual, if no subscript is shown, the cipher(s) are assumed to be representing a decimal number.

Commonly, the number of cipher types (and therefore, the place-value multiplier) used in a numeration system is called that system’s base. Binary is referred to as “base two” numeration, and decimal as “base ten.”

Additionally, we refer to each cipher position in binary as a *bit* rather than the familiar word *digit* used in the decimal system.

Now, why would anyone use binary numeration? The decimal system, with its ten ciphers, makes a lot of sense, being that we have ten fingers on which to count between our two hands. (It is interesting that some ancient central American cultures used numeration systems with a base of twenty.

Presumably, they used both fingers and toes to count!!). But the primary reason that the binary numeration system is used in modern electronic computers is because of the ease of representing two cipher states (0 and 1) electronically.

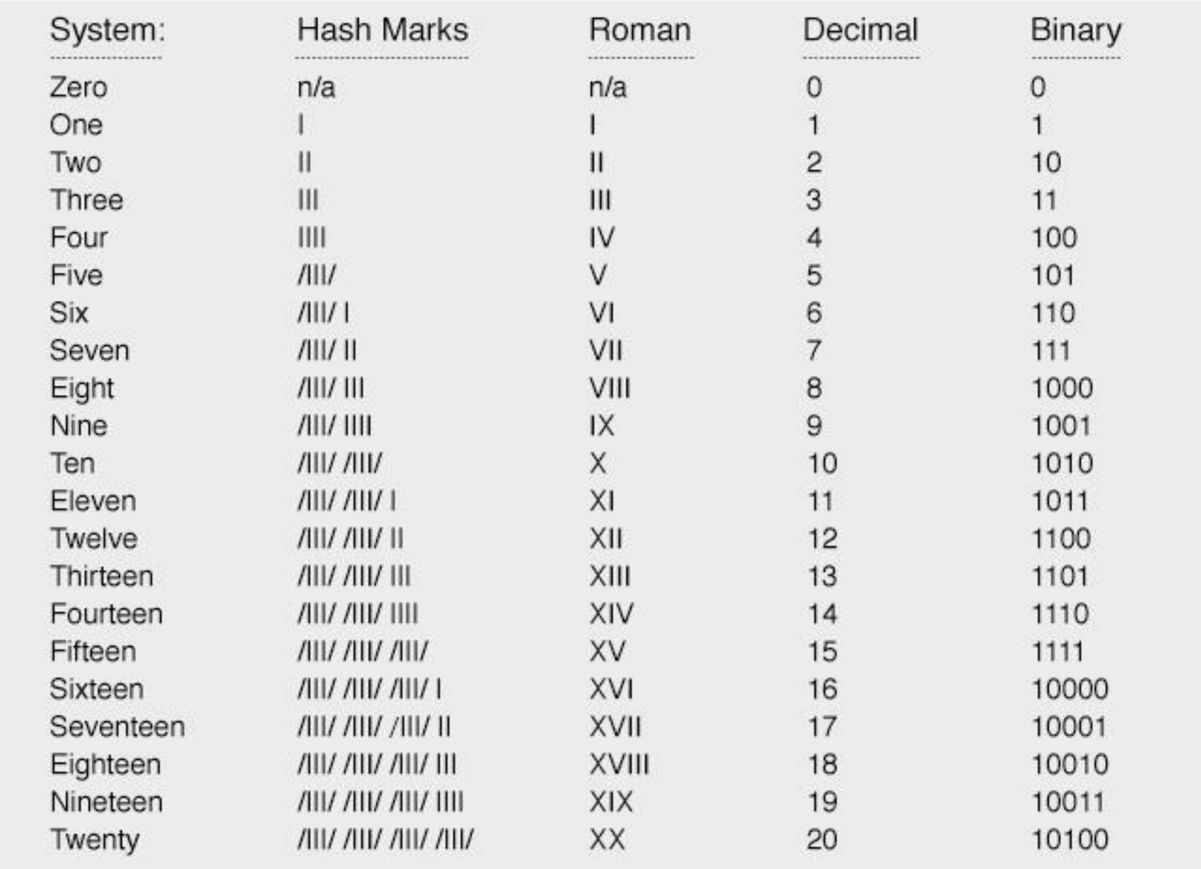
With relatively simple circuitry, we can perform mathematical operations on binary numbers by representing each bit of the numbers by a circuit which is either on (current) or off (no current). Just like the abacus with each rod representing another decimal digit, we simply add more circuits to give us more bits to symbolize larger numbers.

Binary numeration also lends itself well to the storage and retrieval of numerical information: on magnetic tape (spots of iron oxide on the tape either being magnetized for a binary "1" or demagnetized for a binary "0"), optical disks (a laser-burned pit in the aluminum foil representing a binary "1" and an unburned spot representing a binary "0"), or a variety of other media types.

Before we go on to learning exactly how all this is done in digital circuitry, we need to become more familiar with binary and other associated systems of numeration.

Decimal versus Binary Numeration

Let's count from zero to twenty using four different kinds of numeration systems: hash marks, Roman numerals, decimal, and binary:



System:	Hash Marks	Roman	Decimal	Binary
Zero	n/a	n/a	0	0
One	I	I	1	1
Two	II	II	2	10
Three	III	III	3	11
Four	IIII	IV	4	100
Five	/III/	V	5	101
Six	/III/ I	VI	6	110
Seven	/III/ II	VII	7	111
Eight	/III/ III	VIII	8	1000
Nine	/III/ IIII	IX	9	1001
Ten	/III/ /III/	X	10	1010
Eleven	/III/ /III/ I	XI	11	1011
Twelve	/III/ /III/ II	XII	12	1100
Thirteen	/III/ /III/ III	XIII	13	1101
Fourteen	/III/ /III/ IIII	XIV	14	1110
Fifteen	/III/ /III/ /III/	XV	15	1111
Sixteen	/III/ /III/ /III/ I	XVI	16	10000
Seventeen	/III/ /III/ /III/ II	XVII	17	10001
Eighteen	/III/ /III/ /III/ III	XVIII	18	10010
Nineteen	/III/ /III/ /III/ IIII	XIX	19	10011
Twenty	/III/ /III/ /III/ /III/	XX	20	10100

Neither hash marks nor the Roman system are very practical for symbolizing large numbers. Obviously, place-weighted systems such as decimal and binary are more efficient for the task.

Notice, though, how much shorter decimal notation is over binary notation, for the same number of quantities. What takes five bits in binary notation only takes two digits in decimal notation.

This raises an interesting question regarding different numeration systems: how large of a number can be represented with a limited number of cipher positions, or places? With the crude hash-mark system, the number of places IS the largest number that can be represented, since one hash mark "place" is required for every integer step.

For place-weighted systems of numeration, however, the answer is found by taking base of the numeration system (10 for decimal, 2 for binary) and raising it to the power of the number of places.

For example, 5 digits in a decimal numeration system can represent 100,000 different integer number values, from 0 to 99,999 (10 to the 5th power = 100,000). 8 bits in a binary numeration system can represent 256 different integer number values, from 0 to 11111111 (binary), or 0 to 255 (decimal), because 2 to the 8th power equals 256.

With each additional place position to the number field, the capacity for representing numbers increases by a factor of the base (10 for decimal, 2 for binary).

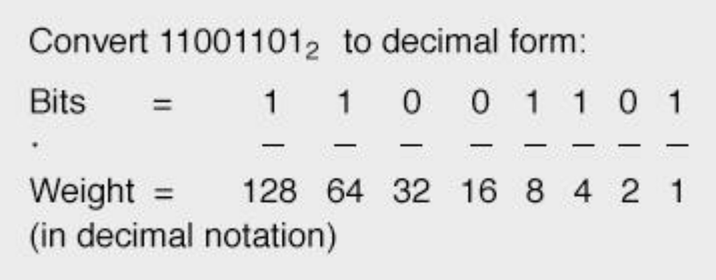
An interesting footnote for this topic is the one of the first electronic digital computers, the Eniac.

The designers of the Eniac chose to represent numbers in decimal form, digitally, using a series of circuits called "ring counters" instead of just going with the binary numeration system, in an effort to minimize the number of circuits required to represent and calculate very large numbers.

This approach turned out to be counter-productive, and virtually all digital computers since then have been purely binary in design.

Binary to Decimal Conversion

To convert a number in binary numeration to its equivalent in decimal form, all you have to do is calculate the sum of all the products of bits with their respective place-weight constants. To illustrate:



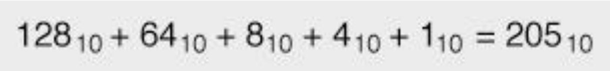
Convert 11001101_2 to decimal form:

Bits	=	1	1	0	0	1	1	0	1
.		—	—	—	—	—	—	—	—
Weight (in decimal notation)	=	128	64	32	16	8	4	2	1

The bit on the far right side is called the Least Significant Bit (LSB), because it stands in the place of the lowest weight (the one's place).

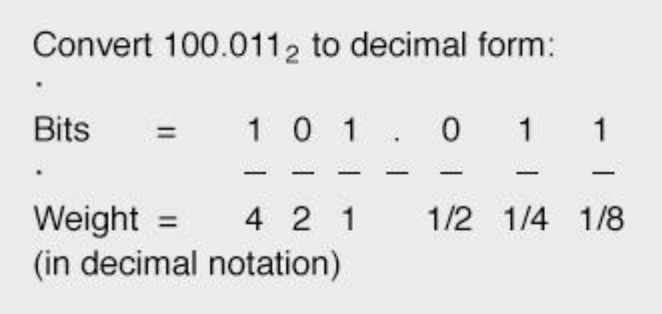
The bit on the far left side is called the Most Significant Bit (MSB), because it stands in the place of the highest weight (the one hundred twenty-eight's place).

Remember, a bit value of "1" means that the respective place weight gets added to the total value, and a bit value of "0" means that the respective place weight does *not* get added to the total value. With the above example, we have:



$$128_{10} + 64_{10} + 8_{10} + 4_{10} + 1_{10} = 205_{10}$$

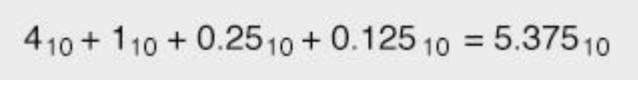
If we encounter a binary number with a dot (.), called a "binary point" instead of a decimal point, we follow the same procedure, realizing that each place weight to the right of the point is one-half the value of the one to the left of it (just as each place weight to the right of a *decimal* point is one-tenth the weight of the one to the left of it). For example:



Convert 100.011_2 to decimal form:

.

Bits	=	1	0	1	.	0	1	1
.		—	—	—	—	—	—	—
Weight (in decimal notation)	=	4	2	1		1/2	1/4	1/8



$$4_{10} + 1_{10} + 0.25_{10} + 0.125_{10} = 5.375_{10}$$

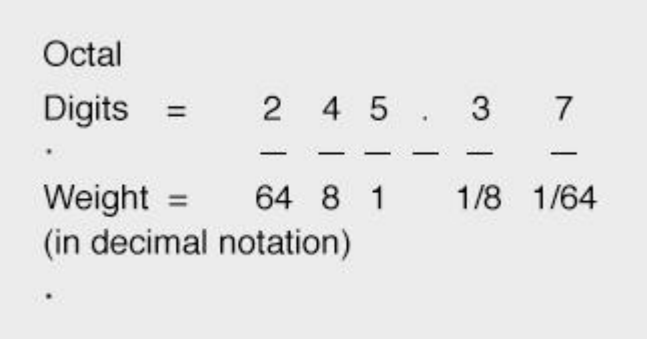
Octal and Hexadecimal to Decimal Conversion

Although the prime intent of octal and hexadecimal numeration systems is for the “shorthand” representation of binary numbers in digital electronics, we sometimes have the need to convert from either of those systems to decimal form.

Of course, we could simply convert the hexadecimal or octal format to binary, then convert from binary to decimal, since we already know how to do both, but we can also convert directly.

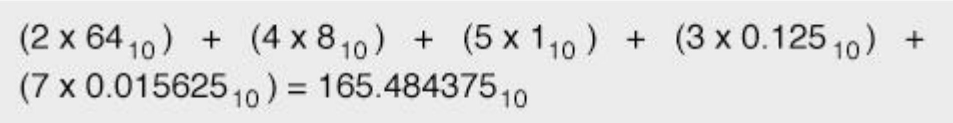
Because octal is a base-eight numeration system, each place-weight value differs from either adjacent place by a factor of eight.

For example, the octal number 245.37 can be broken down into place values as such:



Octal						
Digits =	2	4	5	.	3	7
.	—	—	—	—	—	—
Weight =	64	8	1		1/8	1/64
(in decimal notation)						
.						

The decimal value of each octal place-weight times its respective cipher multiplier can be determined as follows:



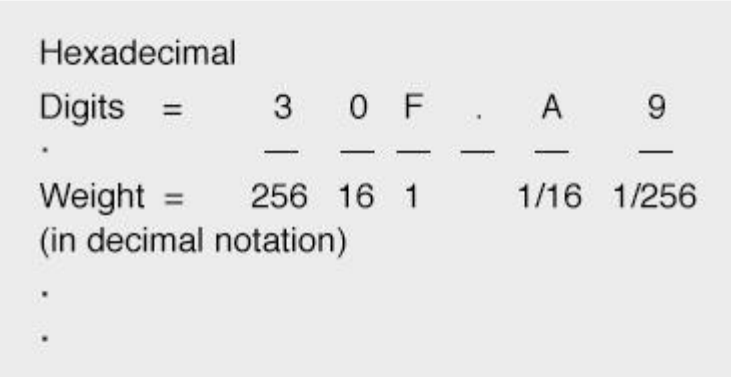
$$(2 \times 64_{10}) + (4 \times 8_{10}) + (5 \times 1_{10}) + (3 \times 0.125_{10}) + (7 \times 0.015625_{10}) = 165.484375_{10}$$

Hexadecimal to Decimal Conversion

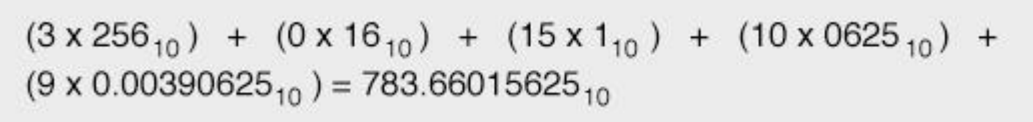
The technique for converting hexadecimal notation to decimal is the same, except that each successive place-weight changes by a factor of sixteen.

Simply denote each digit’s weight, multiply each hexadecimal digit value by its respective weight (in decimal form), then add up all the decimal values to get a total.

For example, the hexadecimal number $30F.A9_{16}$ can be converted like this:



Hexadecimal							
Digits =	3	0	F	.	A	9
.	—	—	—	—	—	—
Weight =	256	16	1		1/16	1/256
(in decimal notation)						
.						
.						



$(3 \times 256_{10}) + (0 \times 16_{10}) + (15 \times 1_{10}) + (10 \times 0625_{10}) + (9 \times 0.00390625_{10}) = 783.66015625_{10}$

These basic techniques may be used to convert a numerical notation of *any* base into decimal form, if you know the value of that numeration system's base.

Conversion From Decimal Numeration

Because octal and hexadecimal numeration systems have bases that are multiples of binary (base 2), conversion back and forth between either hexadecimal or octal and binary is very easy.

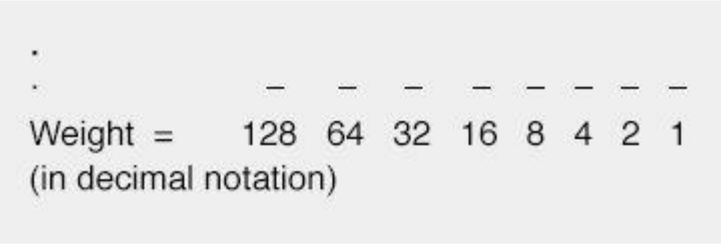
Also, because we are so familiar with the decimal system, converting binary, octal, or hexadecimal to decimal form is relatively easy (simply add up the products of cipher values and place-weights).

However, conversion from decimal to any of these "strange" numeration systems is a different matter.

Trial-and-Fit Method

The method which will probably make the most sense is the "trial-and-fit" method, where you try to "fit" the binary, octal, or hexadecimal notation to the desired value as represented in decimal form.

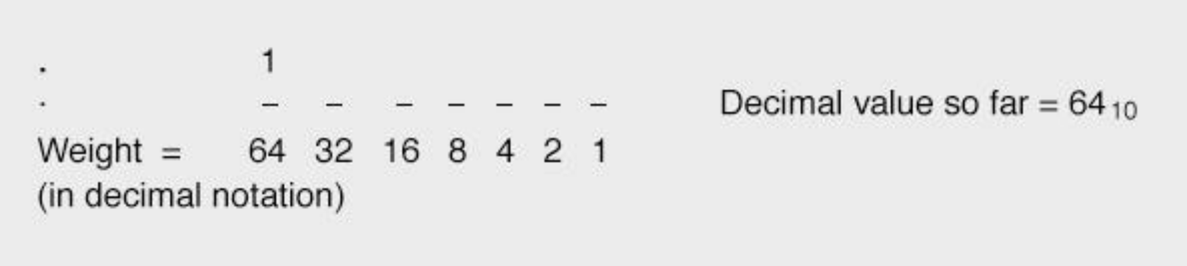
For example, let's say that I wanted to represent the decimal value of 87 in binary form. Let's start by drawing a binary number field, complete with place-weight values:



```
.
.                _    _    _    _   _  _  _  _
Weight =        128   64   32   16  8  4  2  1
(in decimal notation)
```

Well, we know that we won't have a "1" bit in the 128's place, because that would immediately give us a value greater than 87.

However, since the next weight to the right (64) is less than 87, we know that we must have a "1" there.

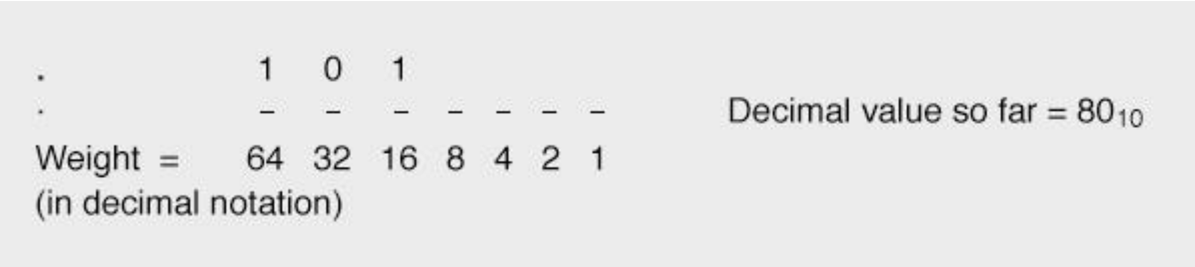


```
.               1
.               _    _    _   _  _  _  _
Weight =        64   32   16  8  4  2  1
(in decimal notation)
```

Decimal value so far = 64_{10}

If we were to make the next place to the right a "1" as well, our total value would be $64_{10} + 32_{10}$, or 96_{10}. This is greater than 87_{10}, so we know that this bit must be a "0".

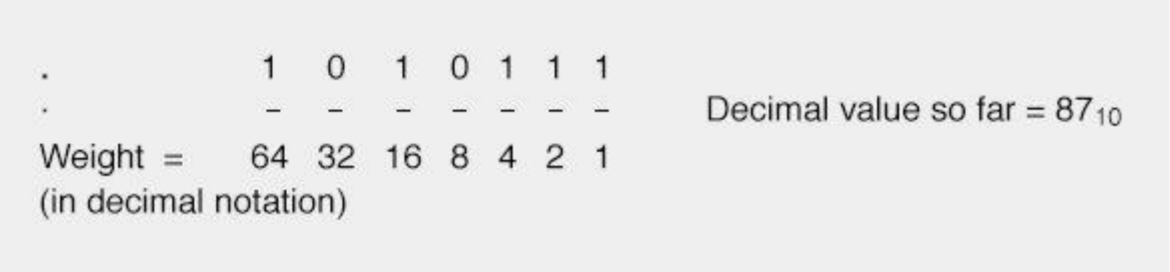
If we make the next (16's) place bit equal to "1," this brings our total value to $64_{10} + 16_{10}$, or 80_{10}, which is closer to our desired value (87_{10}) without exceeding it:



```
.               1    0    1
.               _    _    _   _  _  _  _
Weight =        64   32   16  8  4  2  1
(in decimal notation)
```

Decimal value so far = 80_{10}

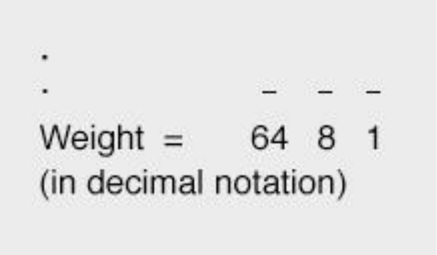
By continuing in this progression, setting each lesser-weight bit as we need to come up to our desired total value without exceeding it, we will eventually arrive at the correct figure:



.	1	0	1	0	1	1	1	
.	–	–	–	–	–	–	–	Decimal value so far = 87_{10}
Weight =	64	32	16	8	4	2	1	
(in decimal notation)								

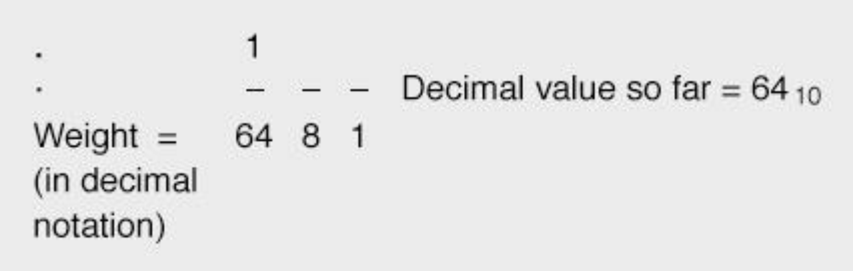
Trial-and-Fit Method in Octal and Hexadecimal

This trial-and-fit strategy will work with octal and hexadecimal conversions, too. Let's take the same decimal figure, 87_{10}, and convert it to octal numeration:



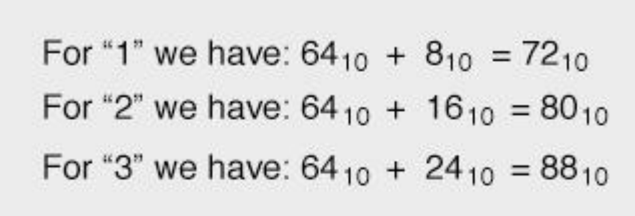
.			
.	–	–	–
Weight =	64	8	1
(in decimal notation)			

If we put a cipher of "1" in the 64's place, we would have a total value of 64_{10} (less than 87_{10}). If we put a cipher of "2" in the 64's place, we would have a total value of 128_{10} (greater than 87_{10}). This tells us that our octal numeration must start with a "1" in the 64's place:



.	1			
.	–	–	–	Decimal value so far = 64_{10}
Weight =	64	8	1	
(in decimal notation)				

Now, we need to experiment with cipher values in the 8's place to try and get a total (decimal) value as close to 87 as possible without exceeding it. Trying the first few cipher options, we get:



For "1" we have: $64_{10} + 8_{10} = 72_{10}$

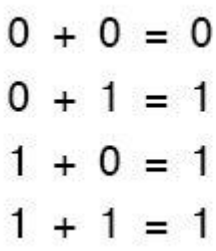
For "2" we have: $64_{10} + 16_{10} = 80_{10}$

For "3" we have: $64_{10} + 24_{10} = 88_{10}$

Logic Gates

Boolean Arithmetic

Let us begin our exploration of Boolean algebra by adding numbers together:



$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

The first three sums make perfect sense to anyone familiar with elementary addition.

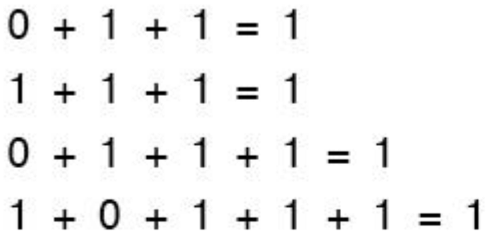
The last sum, though, is quite possibly responsible for more confusion than any other single statement in digital electronics, because it seems to run contrary to the basic principles of mathematics.

Well, it does contradict the principles of addition for real numbers, but not for Boolean numbers.

Remember that in the world of Boolean algebra, there are only two possible values for any quantity and for any arithmetic operation: 1 or 0.

There is no such thing as "2" within the scope of Boolean values. Since the sum "1 + 1" certainly isn't 0, it must be 1 by process of elimination.

It does not matter how many or few terms we add together, either. Consider the following sums:



$$0 + 1 + 1 = 1$$
$$1 + 1 + 1 = 1$$
$$0 + 1 + 1 + 1 = 1$$
$$1 + 0 + 1 + 1 + 1 = 1$$

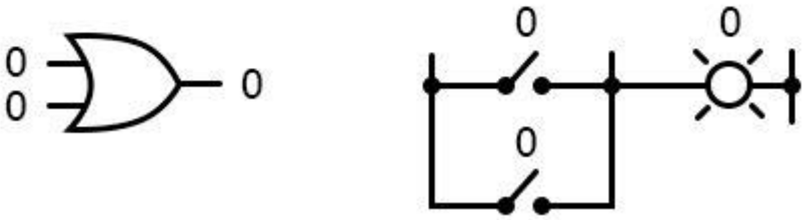
OR Gate

Take a close look at the two-term sums in the first set of equations.

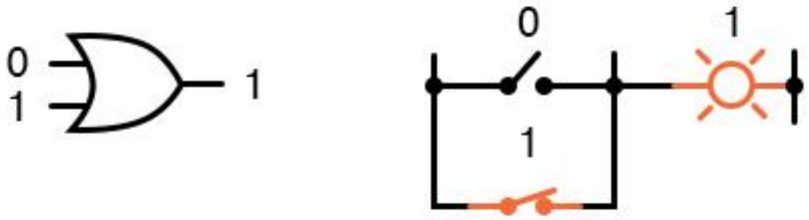
Does that pattern look familiar to you? It should! It is the same pattern of 1's and 0's as seen in the truth table for an OR gate.

In other words, Boolean addition corresponds to the logical function of an "OR" gate, as well as to parallel switch contacts:

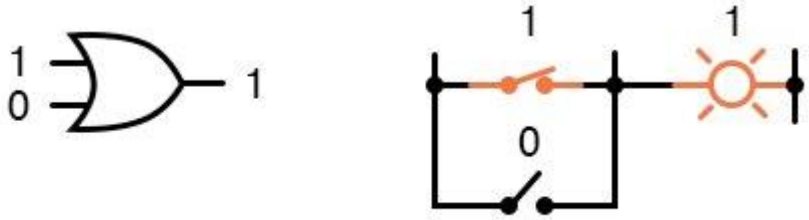
0 + 0 = 0



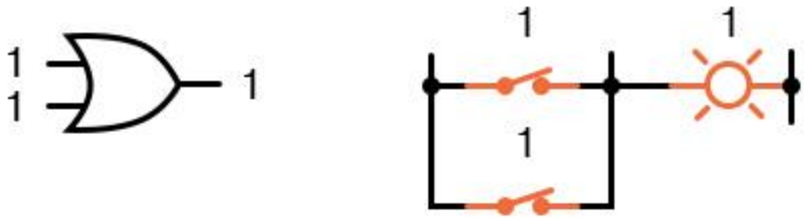
0 + 1 = 1



1 + 0 = 1



1 + 1 = 1



There is no such thing as subtraction in the realm of Boolean mathematics.

Subtraction implies the existence of negative numbers: **5 - 3** is the same thing as **5 + (-3)**, and in Boolean algebra negative quantities are forbidden.

There is no such thing as division in Boolean mathematics, either, since division is really nothing more than *compounded subtraction*, in the same way that multiplication is *compounded addition*.

AND Gate

Multiplication is valid in Boolean algebra, and thankfully it is the same as in real-number algebra: anything multiplied by **0** is **0**, and anything multiplied by **1** remains unchanged:

0 x 0 = 0

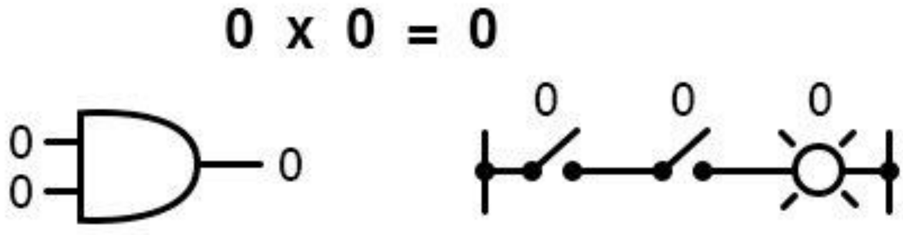
0 x 1 = 0

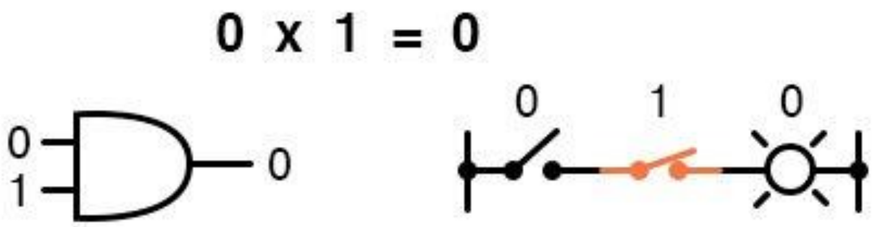
1 x 0 = 0

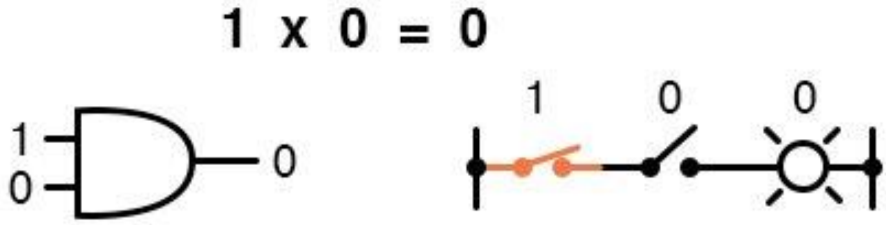
1 x 1 = 1

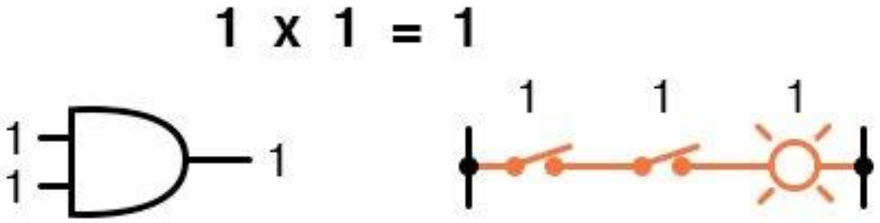
This set of equations should also look familiar to you: it is the same pattern found in the truth table for an AND gate.

In other words, Boolean multiplication corresponds to the logical function of an "**AND**" gate, as well as to series switch contacts:









Like “normal” algebra, Boolean algebra uses alphabetical letters to denote variables.

Unlike “normal” algebra, though, Boolean variables are always CAPITAL letters, never lower-case.

Because they are allowed to possess only one of two possible values, either **1** or **0**, each and every variable has a *complement*: the opposite of its value.

For example, if variable “**A**” has a value of **0**, then the complement of **A** has a value of **1**.

Boolean notation uses a bar above the variable character to denote complementation, like this:

If: $A = 0$

Then: $\overline{A} = 1$

If: $A = 1$

Then: $\overline{A} = 0$

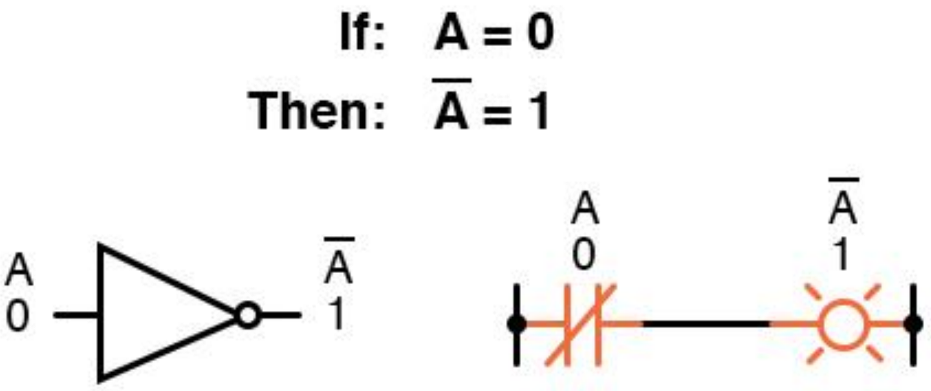
NOT Gate

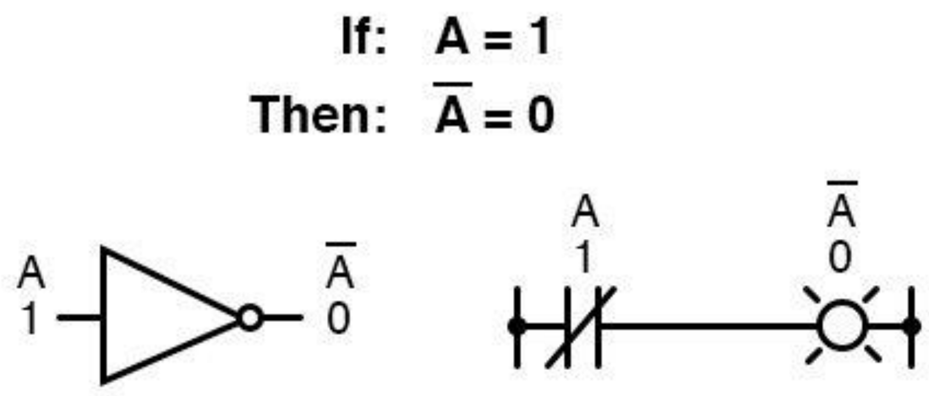
In written form, the complement of “**A**” denoted as “**A-not**” or “**A-bar**”. Sometimes a “prime” symbol is used to represent complementation.

For example, **A**’ would be the complement of **A**, much the same as using a prime symbol to denote differentiation in calculus rather than the fractional notation *d/dt*.

Usually, though, the “bar” symbol finds more widespread use than the “*prime*” symbol, for reasons that will become more apparent later in this chapter.

Boolean complementation finds equivalency in the form of the **NOT gate**, or a normally-closed switch or relay contact:





The basic definition of Boolean quantities has led to the simple rules of addition and multiplication, and has excluded both subtraction and division as valid arithmetic operations.

We have a symbology for denoting Boolean variables, and their complements. In the next section we will proceed to develop Boolean identities.

REVIEW:

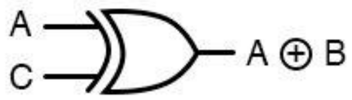
- Boolean addition is equivalent to the *OR* logic function, as well as parallel switch contacts.
- Boolean multiplication is equivalent to the *AND* logic function, as well as series switch contacts.
- Boolean complementation is equivalent to the *NOT* logic function, as well as *normally-closed* relay contacts.

The Exclusive-OR Function: The XOR Gate

One element conspicuously missing from the set of Boolean operations is that of Exclusive-OR, often represented as XOR.

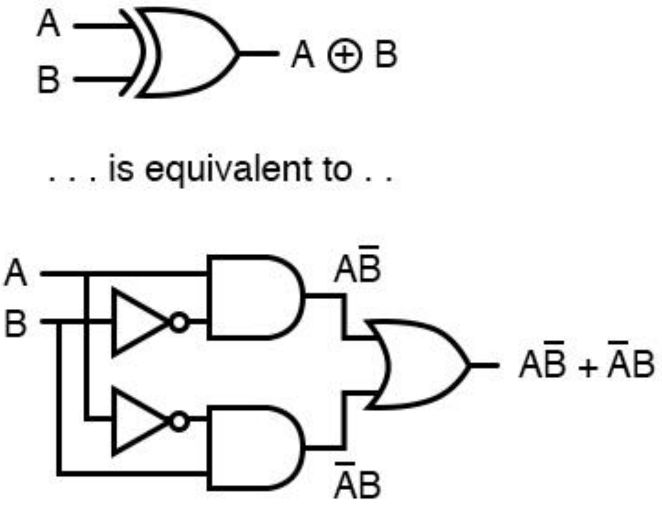
Whereas the OR function is equivalent to Boolean addition, the AND function to Boolean multiplication, and the NOT function (inverter) to Boolean complementation, there is no direct Boolean equivalent for Exclusive-OR.

This hasn’t stopped people from developing a symbol to represent this logic gate, though:



This logic gate symbol is seldom used in Boolean expressions because the identities, laws, and rules of simplification involving addition, multiplication, and complementation do not apply to it.

However, there is a way to represent the Exclusive-OR function in terms of OR and AND, as has been shown in previous chapters: AB’ + A’B



$$A \oplus B = A\bar{B} + \bar{A}B$$

As a Boolean equivalency, this rule may be helpful in simplifying some Boolean expressions.

Any expression following the AB’ + A’B form (two AND gates and an OR gate) may be replaced by a single Exclusive-OR gate.

DeMorgan’s Theorems

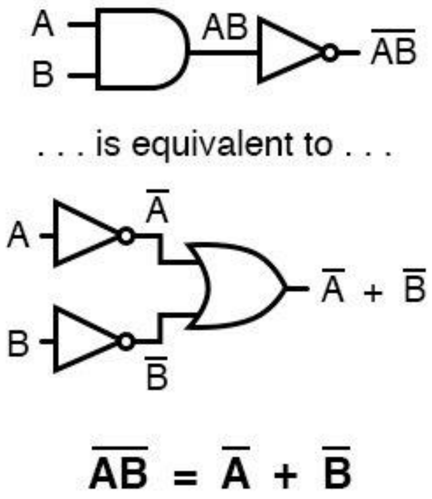
A mathematician named DeMorgan developed a pair of important rules regarding group complementation in Boolean algebra.

Group complementation is to the complement of a group of terms, represented by a long bar over more than one variable.

Logic gates that inverting all inputs to a gate reverses that gate’s essential function from AND to OR, or vice versa, and also inverts the output.

So, an OR gate with all inputs inverted (a Negative-OR gate) behaves the same as a NAND gate, and an AND gate with all inputs inverted (a Negative-AND gate) behaves the same as a NOR gate.

DeMorgan’s theorems state the same equivalence in “backward” form: that inverting the output of any gate results in the same function as the opposite type of gate (AND vs. OR) with inverted inputs:


A
B
AB
$\overline{AB}$
. . . is equivalent to . . .
A
$\overline{A}$
B
$\overline{B}$
$\overline{A} + \overline{B}$
$\overline{AB} = \overline{A} + \overline{B}$

A long bar extending over the term AB acts as a grouping symbol, and as such is entirely different from the product of A and B independently inverted.

In other words, (AB)’ is not equal to A’B’. Because the “prime” symbol (’) cannot be stretched over two variables like a bar can, we are forced to use parentheses to make it apply to the whole term AB in the previous sentence.

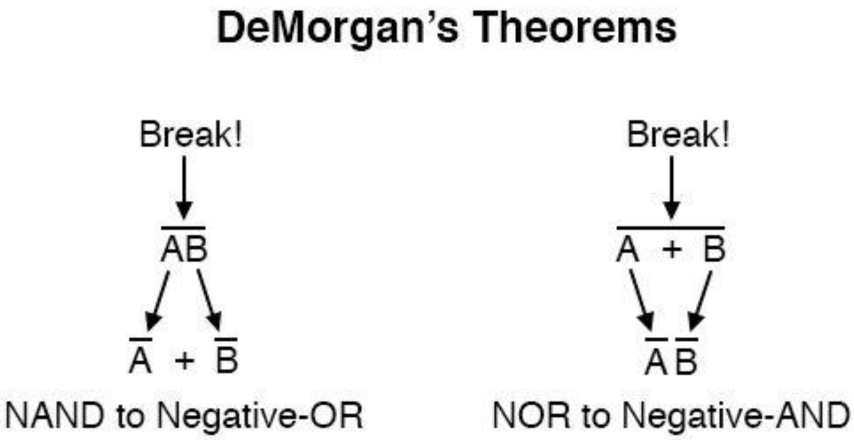
A bar, however, acts as its own grouping symbol when stretched over more than one variable.

This has profound impact on how Boolean expressions are evaluated and reduced, as we shall see.

DeMorgan’s Theorem

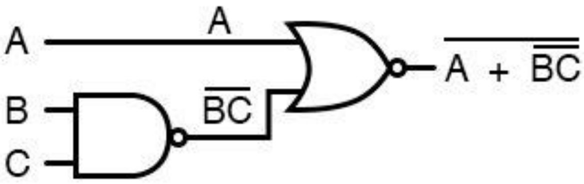
DeMorgan’s theorem may be thought of in terms of breaking a long bar symbol.

When a long bar is broken, the operation directly underneath the break changes from addition to multiplication, or vice versa, and the broken bar pieces remain over the individual variables. To illustrate:

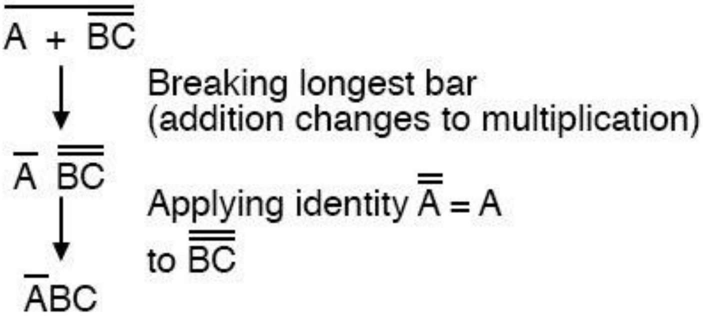


When multiple "layers" of bars exist in an expression, you may only break one bar at a time, and it is generally easier to begin simplification by breaking the longest (uppermost) bar first.

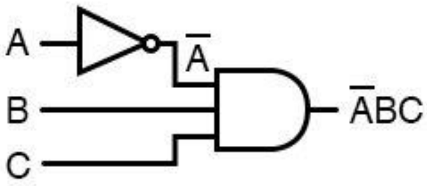
To illustrate, let's take the expression (A + (BC)')' and reduce it using DeMorgan's Theorems:



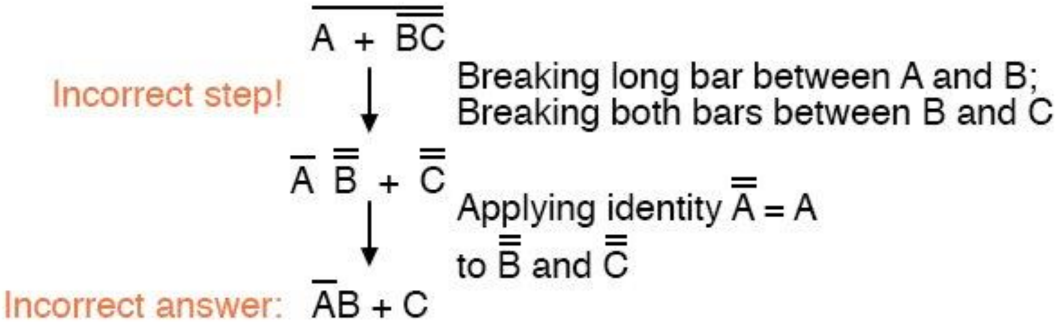
Following the advice of breaking the longest (uppermost) bar first, we'll begin by breaking the bar covering the entire expression as a first step:


$\overline{A + \overline{BC}}$
Breaking longest bar
(addition changes to multiplication)
$\overline{A}\,\overline{\overline{BC}}$
Applying identity $\overline{\overline{A}} = A$
to $\overline{\overline{BC}}$
$\overline{A}BC$

As a result, the original circuit is reduced to a three-input AND gate with the A input inverted:

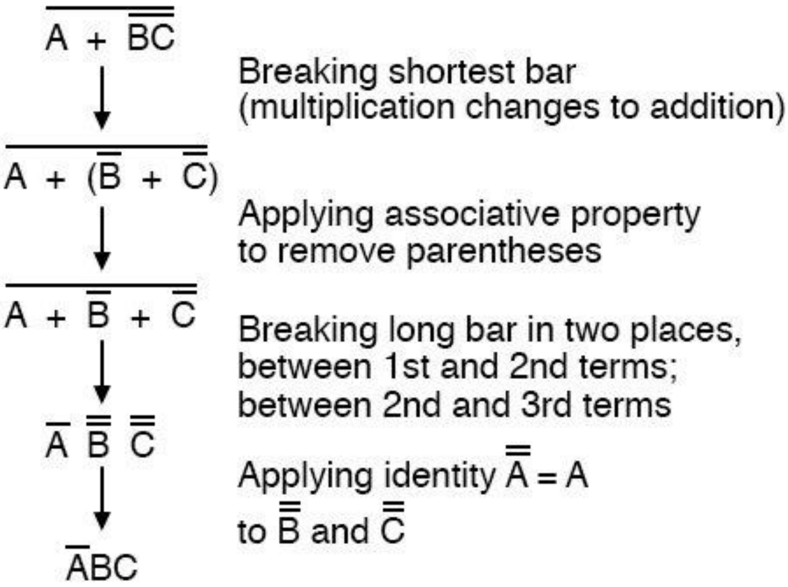


You should never break more than one bar in a single step, as illustrated here:


$\overline{A + \overline{BC}}$
Incorrect step!
Breaking long bar between A and B;
Breaking both bars between B and C
$\overline{A}\,\overline{\overline{B}} + \overline{\overline{C}}$
Applying identity $\overline{\overline{A}} = A$
to $\overline{\overline{B}}$ and $\overline{\overline{C}}$
Incorrect answer:
$\overline{A}B + C$

As tempting as it may be to conserve steps and break more than one bar at a time, it often leads to an incorrect result, so don’t do it!

It is possible to properly reduce this expression by breaking the short bar first, rather than the long bar first:


$\overline{A + \overline{BC}}$
Breaking shortest bar
(multiplication changes to addition)
$\overline{A + (\overline{B} + \overline{C})}$
Applying associative property
to remove parentheses
$\overline{A + \overline{B} + \overline{C}}$
Breaking long bar in two places,
between 1st and 2nd terms;
between 2nd and 3rd terms
$\overline{A}\,\overline{\overline{B}}\,\overline{\overline{C}}$
Applying identity $\overline{\overline{A}} = A$
to $\overline{\overline{B}}$ and $\overline{\overline{C}}$
$\overline{A}BC$

The end result is the same, but more steps are required compared to using the first method, where the longest bar was broken first.

Note how in the third step we broke the long bar in two places.

This is a legitimate mathematical operation, and not the same as breaking two bars in one step!

The prohibition against breaking more than one bar in one step is not a prohibition against breaking a bar in more than one place.

Breaking in more than one place in a single step is okay; breaking more than one bar in a single step is not.

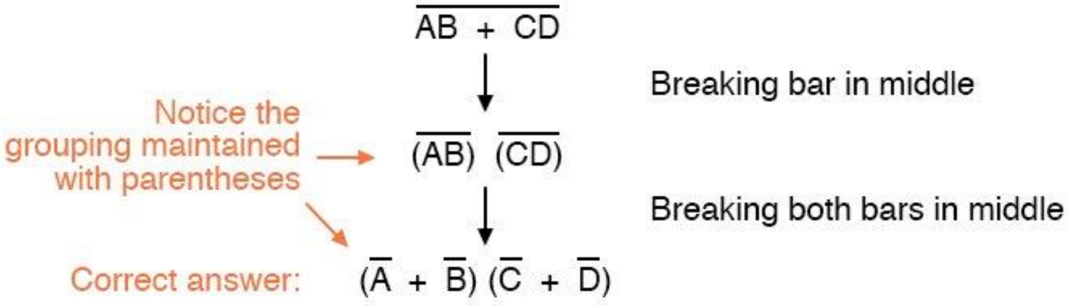
You might be wondering why parentheses were placed around the sub-expression B’ + C’, considering the fact that I just removed them in the next step.

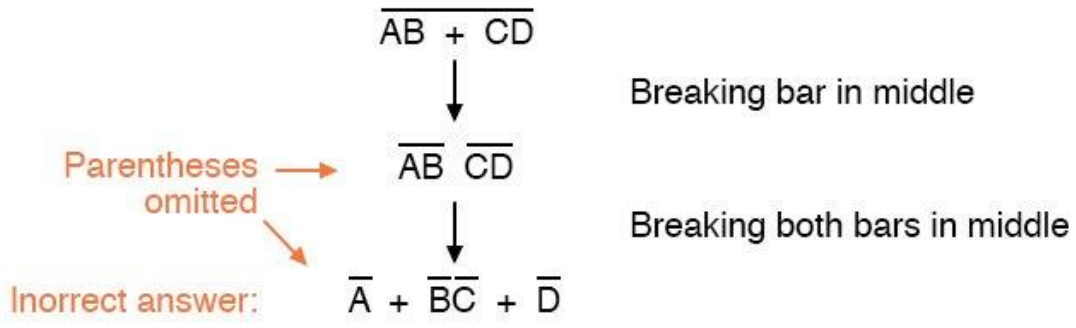
This is done to emphasize an important but easily neglected aspect of DeMorgan’s theorem.

Since a long bar functions as a grouping symbol, the variables formerly grouped by a broken bar must remain grouped lest proper precedence (order of operation) be lost.

In this example, it really wouldn’t matter if I forgot to put parentheses in after breaking the short bar, but in other cases it might.

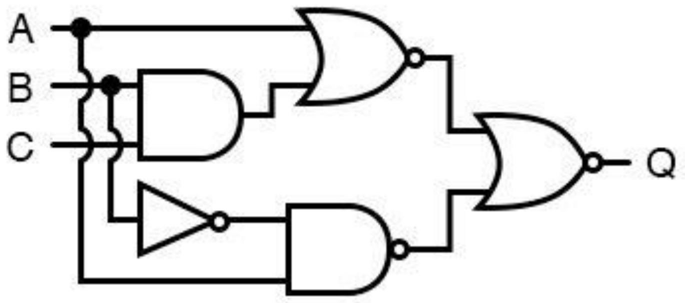
Consider this example, starting with a different expression:


$\overline{AB + CD}$
Breaking bar in middle
Notice the grouping maintained with parentheses
$\overline{(AB)}\ \overline{(CD)}$
Breaking both bars in middle
Correct answer:
$(\overline{A} + \overline{B})(\overline{C} + \overline{D})$


$\overline{AB + CD}$
Breaking bar in middle
Parentheses omitted
$\overline{AB}\ \overline{CD}$
Breaking both bars in middle
Inorrect answer:
$\overline{A} + \overline{B}\,\overline{C} + \overline{D}$

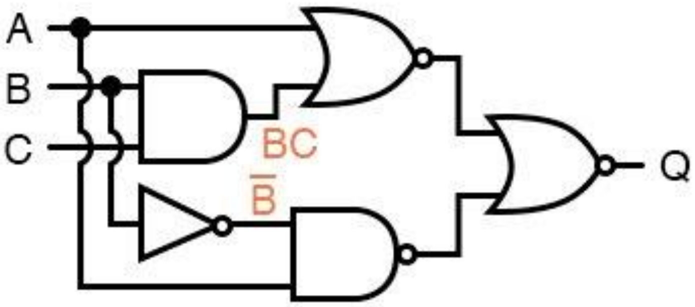
As you can see, maintaining the grouping implied by the complementation bars for this expression is crucial to obtaining the correct answer.

Let’s apply the principles of DeMorgan’s theorems to the simplification of a gate circuit:



As always, our first step in simplifying this circuit must be to generate an equivalent Boolean expression.

We can do this by placing a sub-expression label at the output of each gate, as the inputs become known. Here’s the first step in this process:

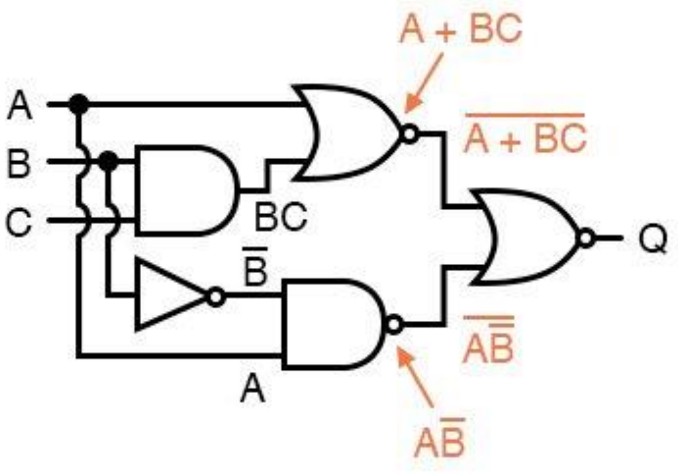


Next, we can label the outputs of the first NOR gate and the NAND gate.

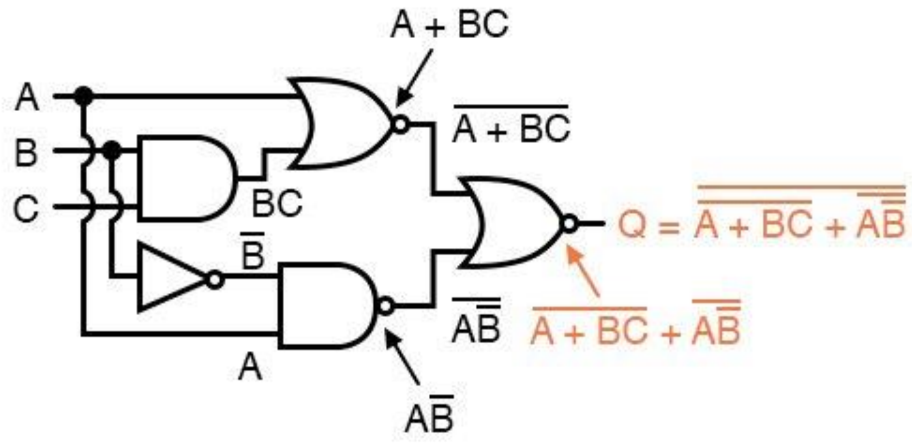
When dealing with inverted-output gates it easier to write an expression for the gate's output without the final inversion, with an arrow pointing to just before the inversion bubble.

Then, at the wire leading out of the gate (after the bubble), I write the full, complemented expression.

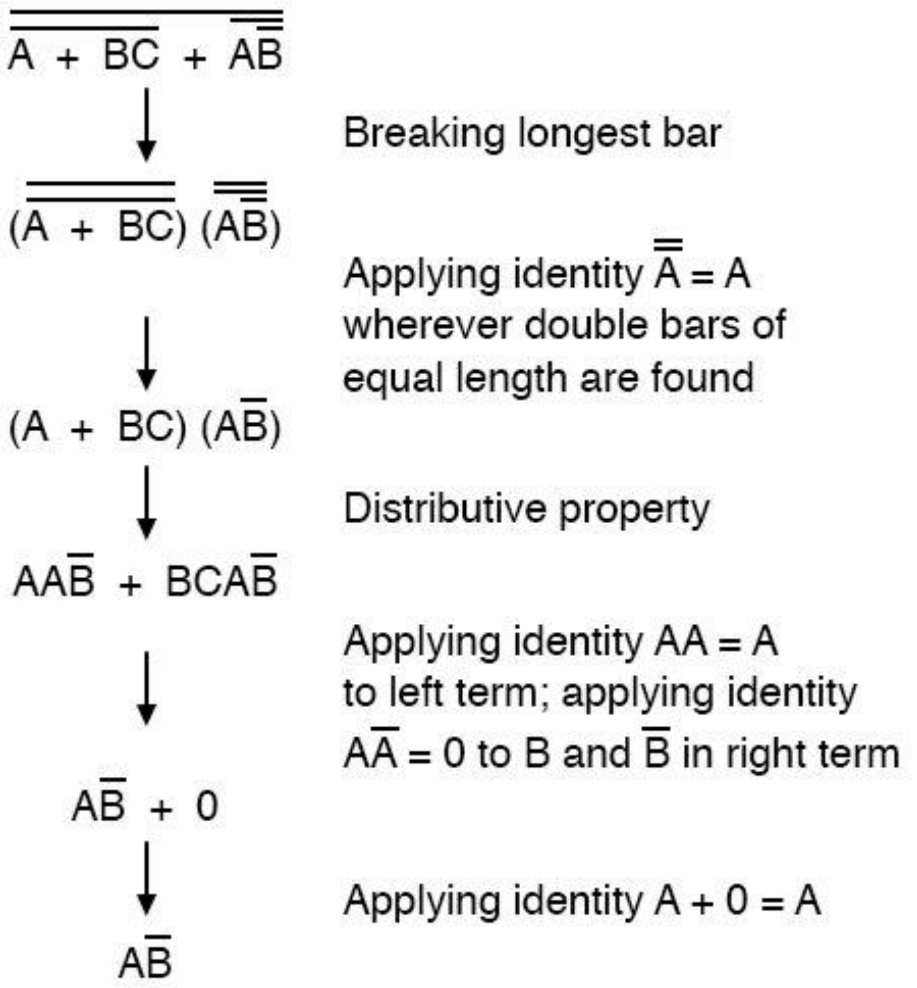
This helps ensure I don't forget a complementing bar in the sub-expression, by forcing myself to split the expression-writing task into two steps:



Finally, we write an expression (or pair of expressions) for the last NOR gate:


A + BC
A
B
C
$\overline{A+BC}$
BC
$\overline{B}$
$Q = \overline{\overline{A+BC}+\overline{A\overline{B}}}$
$\overline{A\overline{B}}$
$\overline{A+BC}+\overline{A\overline{B}}$
A
$A\overline{B}$

Now, we reduce this expression using the identities, properties, rules, and theorems (DeMorgan’s) of Boolean algebra:



$$\overline{\overline{A+BC}+\overline{A\overline{B}}}$$

↓ Breaking longest bar

$$(\overline{\overline{A+BC}})\,(\overline{\overline{A\overline{B}}})$$

Applying identity $\overline{\overline{A}} = A$ wherever double bars of equal length are found

↓

$$(A+BC)\,(A\overline{B})$$

↓ Distributive property

$$AA\overline{B}+BCA\overline{B}$$

Applying identity $AA = A$ to left term; applying identity $A\overline{A} = 0$ to B and $\overline{B}$ in right term

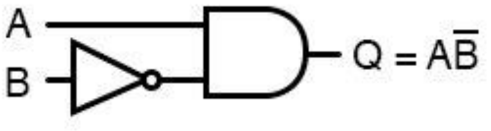
↓

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

↓ Applying identity $A + 0 = A$

$$A\overline{B}$$

The equivalent gate circuit for this much-simplified expression is as follows:



REVIEW:

- DeMorgan's Theorems describe the equivalence between gates with inverted inputs and gates with inverted outputs. Simply put, a NAND gate is equivalent to a Negative-OR gate, and a NOR gate is equivalent to a Negative-AND gate.
- When "breaking" a complementation bar in a Boolean expression, the operation directly underneath the break (addition or multiplication) reverses, and the broken bar pieces remain over the respective terms.
- It is often easier to approach a problem by breaking the longest (uppermost) bar before breaking any bars under it. You must never attempt to break two bars in one step!
- Complementation bars function as grouping symbols. Therefore, when a bar is broken, the terms underneath it must remain grouped. Parentheses may be placed around these grouped terms as a help to avoid changing precedence.

Converting Truth Tables into Boolean Expressions

In designing digital circuits, the designer often begins with a truth table describing what the circuit should do.

The design task is largely to determine what type of circuit will perform the function described in the truth table.

While some people seem to have a natural ability to look at a truth table and immediately envision the necessary logic gate or relay logic circuitry for the task, there are procedural techniques available for the rest of us.

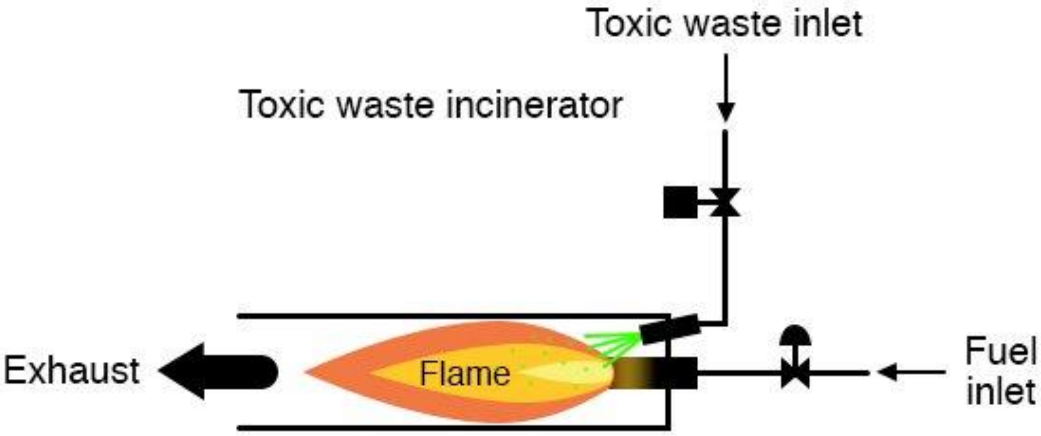
Here, Boolean algebra proves its utility in a most dramatic way.

To illustrate this procedural method, we should begin with a realistic design problem.

Suppose we were given the task of designing a flame detection circuit for a toxic waste incinerator.

The intense heat of the fire is intended to neutralize the toxicity of the waste introduced into the incinerator.

Such combustion-based techniques are commonly used to neutralize medical waste, which may be infected with deadly viruses or bacteria:



So long as a flame is maintained in the incinerator, it is safe to inject waste into it to be neutralized.

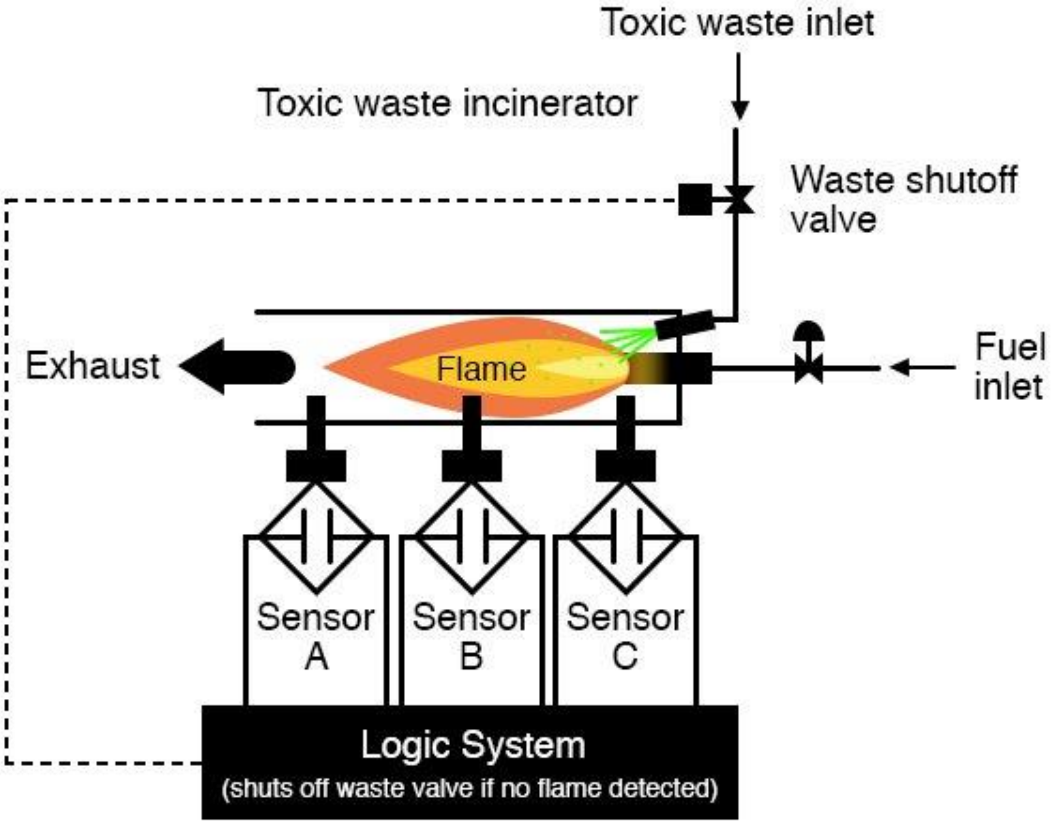
If the flame were to be extinguished, however, it would be unsafe to continue to inject waste into the combustion chamber, as it would exit the exhaust un-neutralized, and pose a health threat to anyone in close proximity to the exhaust.

What we need in this system is a sure way of detecting the presence of a flame, and permitting waste to be injected only if a flame is “proven” by the flame detection system.

Several different flame-detection technologies exist: optical (detection of light), thermal (detection of high temperature), and electrical conduction (detection of ionized particles in the flame path), each one with its unique advantages and disadvantages.

Suppose that due to the high degree of hazard involved with potentially passing un-neutralized waste out the exhaust of this incinerator, it is decided that the flame detection system be made redundant (multiple sensors), so that failure of a single sensor does not lead to an emission of toxins out the exhaust.

Each sensor comes equipped with a normally-open contact (open if no flame, closed if flame detected) which we will use to activate the inputs of a logic system:



Our task, now, is to design the circuitry of the logic system to open the waste valve if and only if there is good flame proven by the sensors.

First, though, we must decide what the logical behavior of this control system should be.

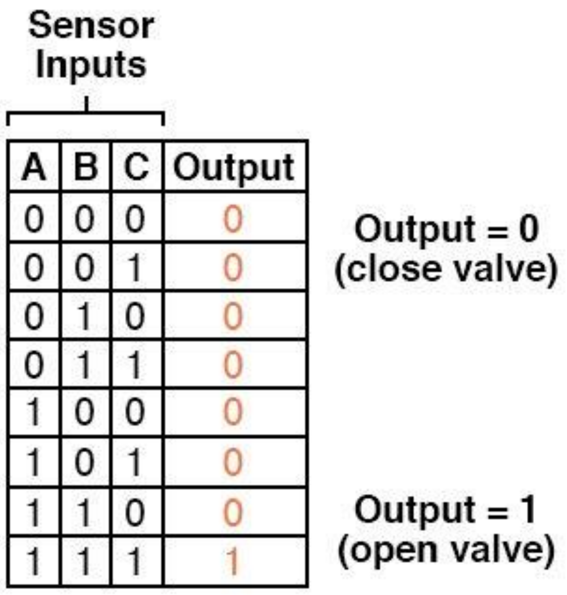
Do we want the valve to be opened if only one out of the three sensors detects flame? Probably not, because this would defeat the purpose of having multiple sensors.

If any one of the sensors were to fail in such a way as to falsely indicate the presence of flame when there was none, a logic system based on the principle of "any one out of three sensors showing flame" would give the same output that a single-sensor system would with the same failure.

A far better solution would be to design the system so that the valve is commanded to open if and only if *all three sensors* detect a good flame.

This way, any single, failed sensor falsely showing flame could not keep the valve in the open position; rather, it would require all three sensors to be failed in the same manner—a highly improbable scenario—for this dangerous condition to occur.

Thus, our truth table would look like this:



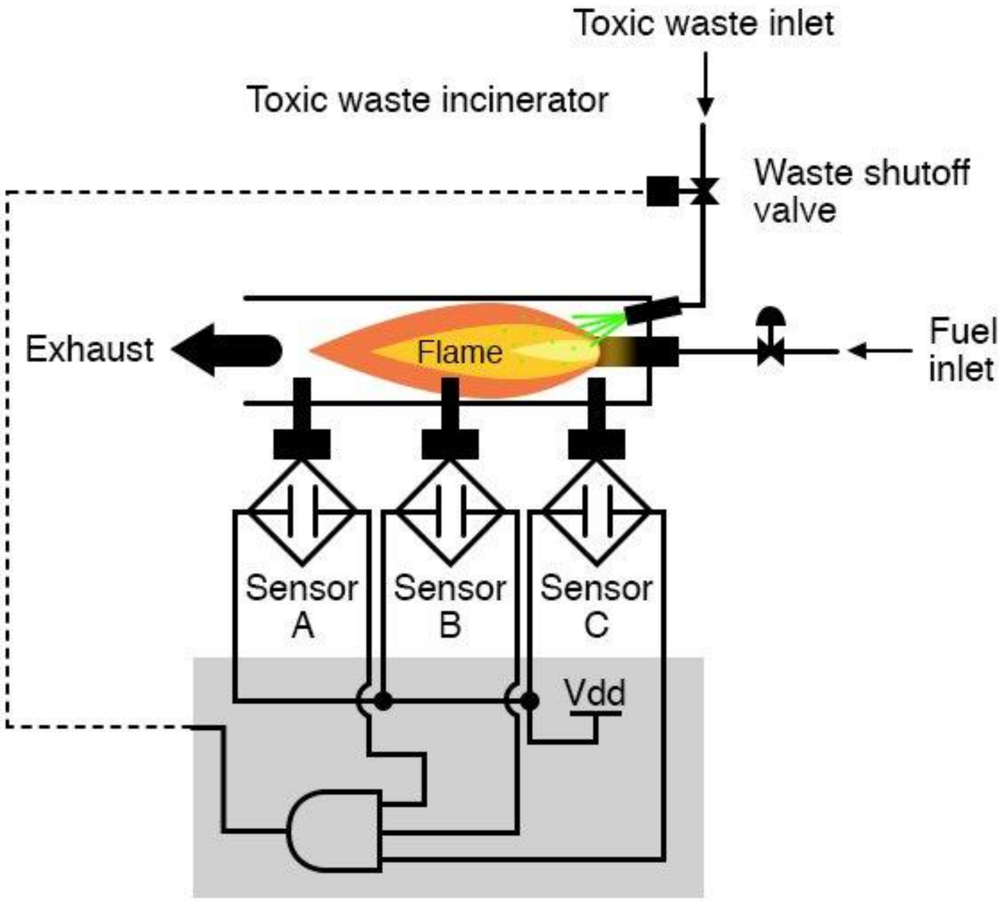
Sensor Inputs

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

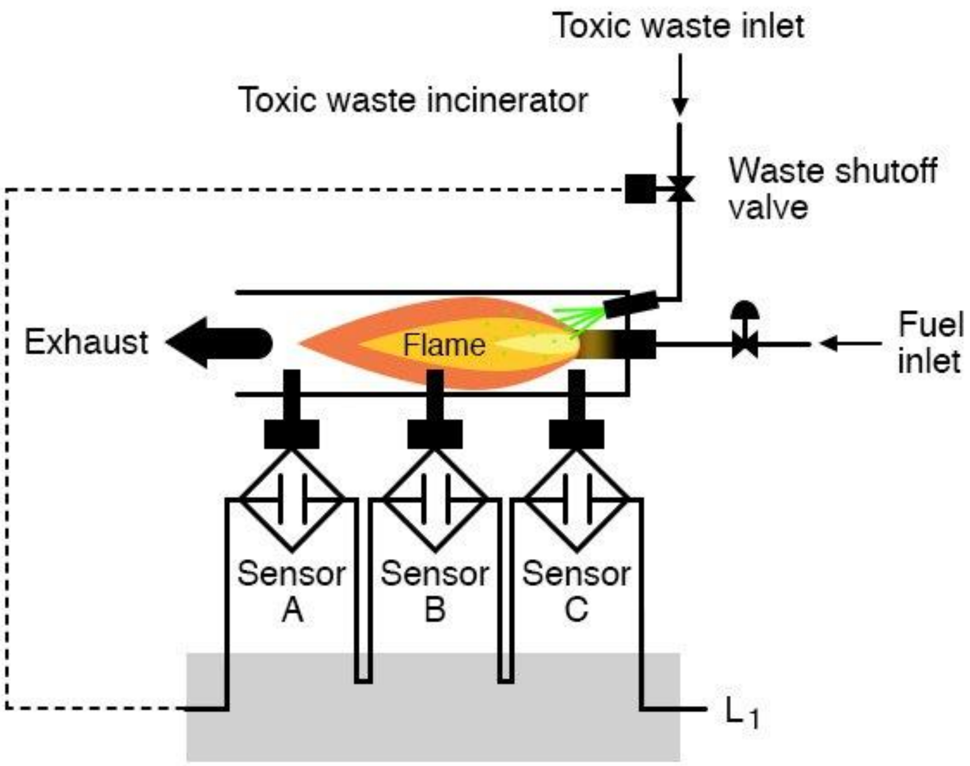
Output = 0
(close valve)

Output = 1
(open valve)

It does not require much insight to realize that this functionality could be generated with a three-input AND gate: the output of the circuit will be "high" if and only if input A *AND* input B *AND* input C are all "high:"



If using relay circuitry, we could create this AND function by wiring three relay contacts in series, or simply by wiring the three sensor contacts in series, so that the only way electrical power could be sent to open the waste valve is if all three sensors indicate flame:



While this design strategy maximizes safety, it makes the system very susceptible to sensor failures of the opposite kind.

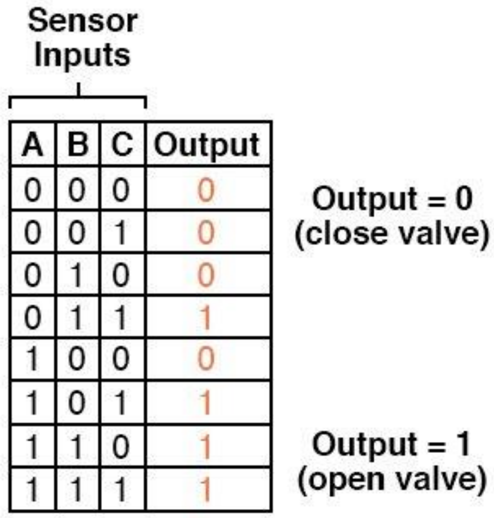
Suppose that one of the three sensors were to fail in such a way that it indicated no flame when there really was a good flame in the incinerator’s combustion chamber.

That single failure would shut off the waste valve unnecessarily, resulting in lost production time and wasted fuel (feeding a fire that wasn’t being used to incinerate waste).

It would be nice to have a logic system that allowed for this kind of failure without shutting the system down unnecessarily, yet still provide sensor redundancy so as to maintain safety in the event that any single sensor failed “high” (showing flame at all times, whether or not there was one to detect).

A strategy that would meet both needs would be a “two out of three” sensor logic, whereby the waste valve is opened if at least two out of the three sensors show good flame.

The truth table for such a system would look like this:



Sensor Inputs
Output = 0 (close valve)
Output = 1 (open valve)

A	B	C	Output
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

Using Sum-Of-Products

Here, it is not necessarily obvious what kind of logic circuit would satisfy the truth table.

However, a simple method for designing such a circuit is found in a standard form of Boolean expression called the *Sum-Of-Products,* or *SOP*, form.

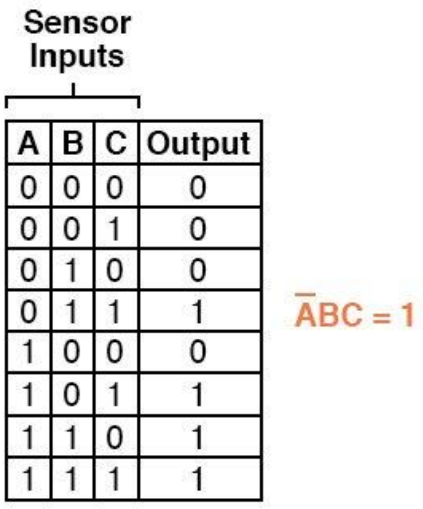
As you might suspect, a Sum-Of-Products Boolean expression is literally a set of Boolean terms added (*summed*) together, each term being a multiplicative (*product*) combination of Boolean variables.

An example of an SOP expression would be something like this: ABC + BC + DF, the sum of products “ABC,” “BC,” and “DF.”

Sum-Of-Products expressions are easy to generate from truth tables.

All we have to do is examine the truth table for any rows where the output is “high” (1), and write a Boolean product term that would equal a value of 1 given those input conditions.

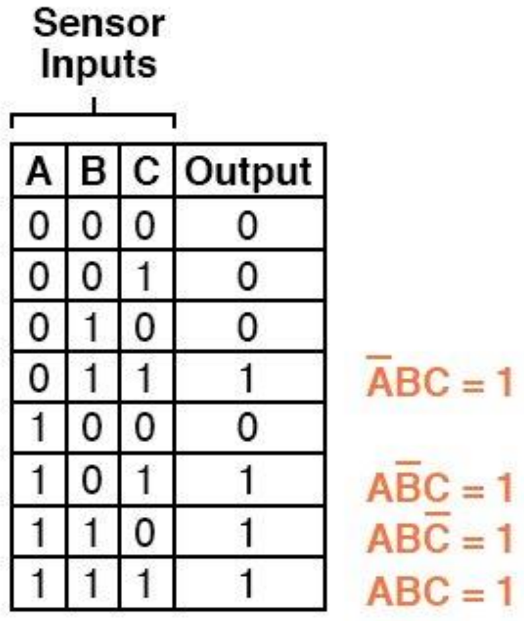
For instance, in the fourth row down in the truth table for our two-out-of-three logic system, where A=0, B=1, and C=1, the product term would be A’BC, since that term would have a value of 1 if and only if A=0, B=1, and C=1 (it is minterm):



Sensor Inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC = 1$
1	0	0	0	
1	0	1	1	
1	1	0	1	
1	1	1	1	

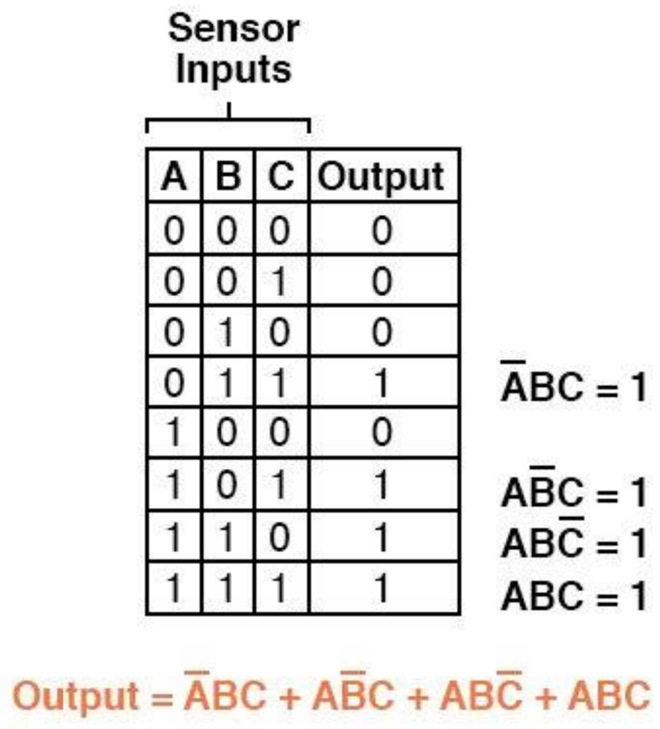
Three other rows of the truth table have an output value of 1, so those rows also need Boolean product expressions to represent them:



Sensor Inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\overline{B}C = 1$
1	1	0	1	$AB\overline{C} = 1$
1	1	1	1	$ABC = 1$

Finally, we join these four Boolean product expressions together by addition, to create a single Boolean expression describing the truth table as a whole:

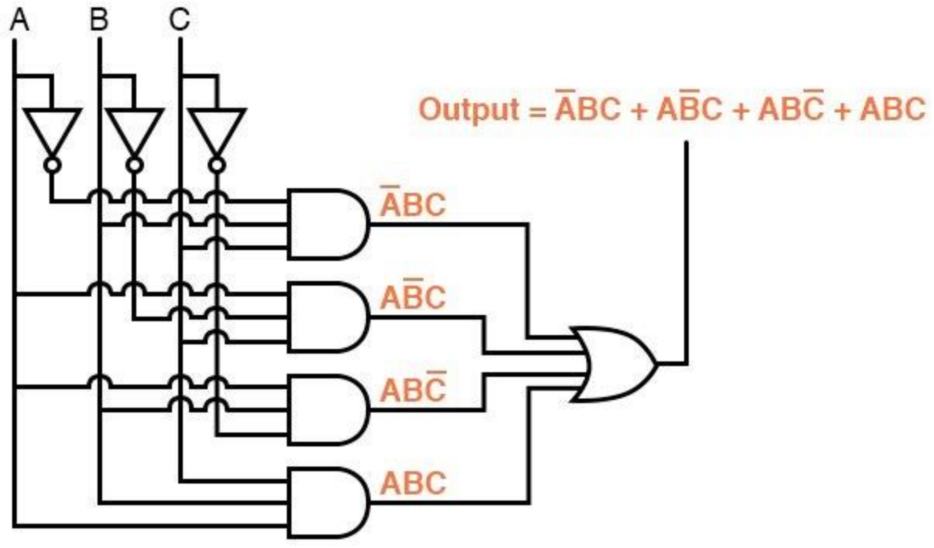


Sensor Inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\overline{B}C = 1$
1	1	0	1	$AB\overline{C} = 1$
1	1	1	1	$ABC = 1$

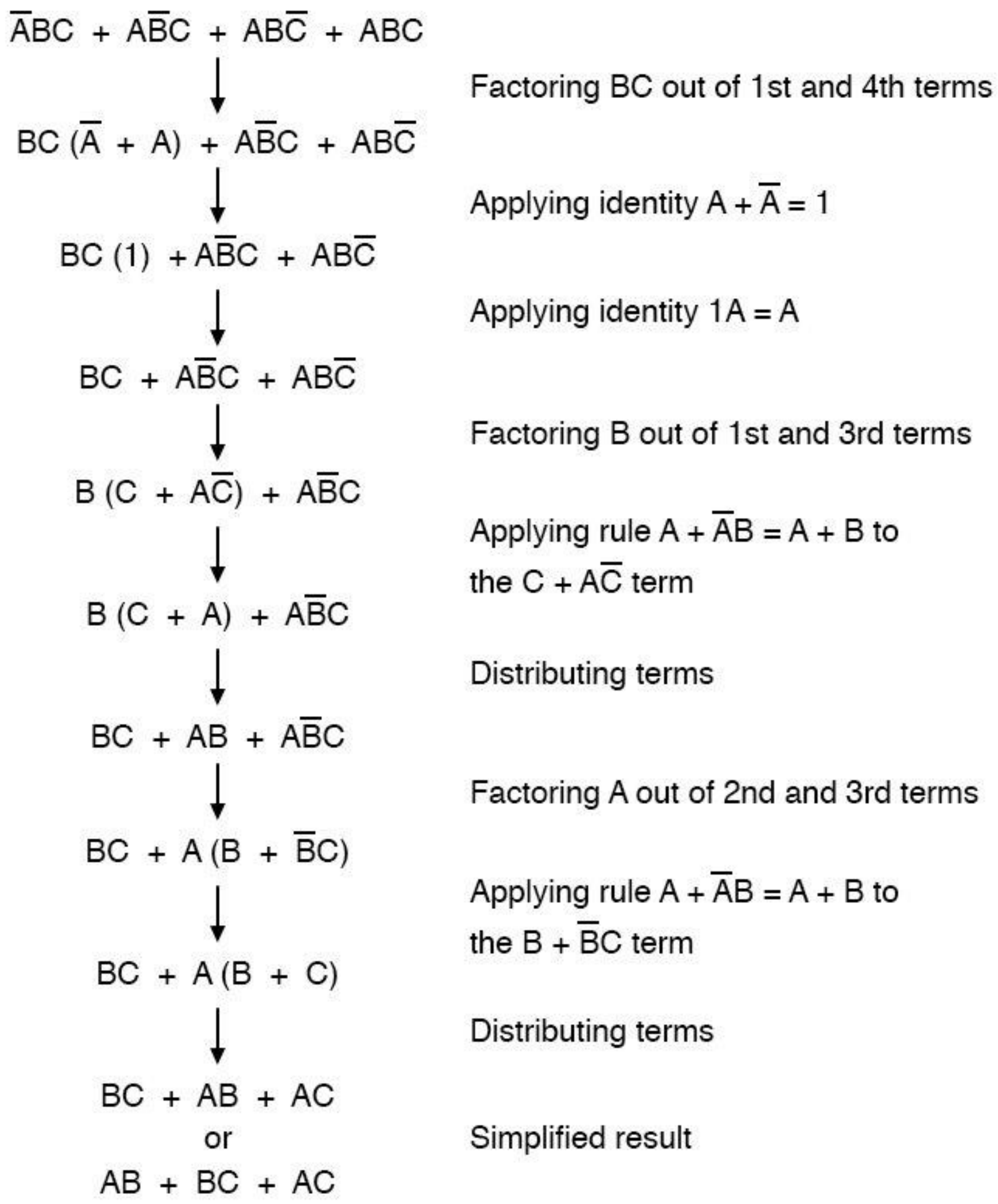
Output = $\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

Now that we have a Boolean Sum-Of-Products expression for the truth table's function, we can easily design a logic gates circuit based on that expression:


A
B
C
Output = $\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$
$\overline{A}BC$
$A\overline{B}C$
$AB\overline{C}$
ABC

Unfortunately, this circuit is quite complex, and could benefit from simplification.

Using Boolean algebra techniques, the expression may be significantly simplified:



$\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

Factoring BC out of 1st and 4th terms

$BC(\overline{A} + A) + A\overline{B}C + AB\overline{C}$

Applying identity $A + \overline{A} = 1$

$BC(1) + A\overline{B}C + AB\overline{C}$

Applying identity $1A = A$

$BC + A\overline{B}C + AB\overline{C}$

Factoring B out of 1st and 3rd terms

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

Applying rule $A + \overline{A}B = A + B$ to
the $C + A\overline{C}$ term

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

Distributing terms

$BC + AB + A\overline{B}C$

Factoring A out of 2nd and 3rd terms

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

Applying rule $A + \overline{A}B = A + B$ to
the $B + \overline{B}C$ term

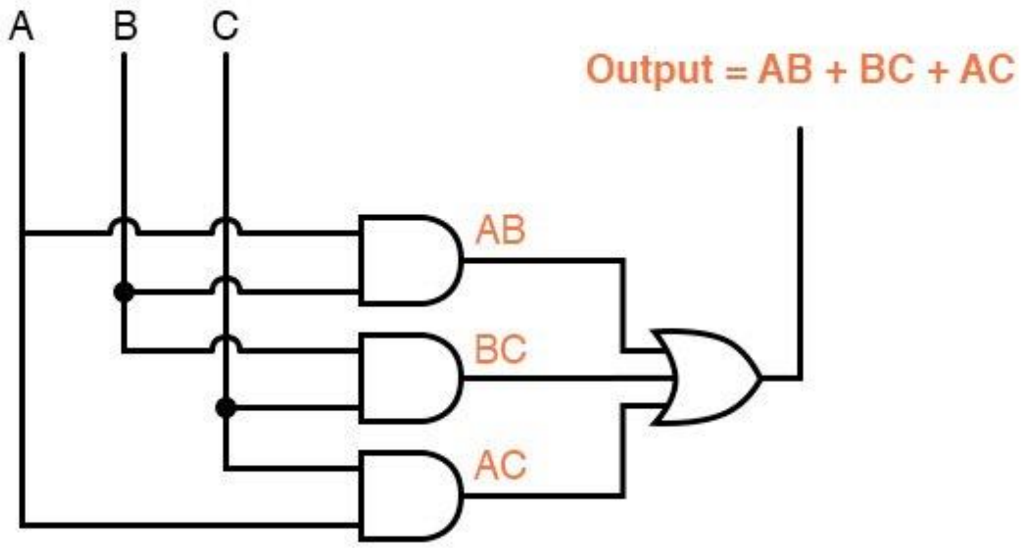
$BC + A(B + C)$

Distributing terms

$BC + AB + AC$
or
$AB + BC + AC$

Simplified result

As a result of the simplification, we can now build much simpler logic circuits performing the same function:



Either one of these circuits will adequately perform the task of operating the incinerator waste valve based on a flame verification from two out of the three flame sensors.

At minimum, this is what we need to have a safe incinerator system.

We can, however, extend the functionality of the system by adding to it logic circuitry designed to detect if any one of the sensors does not agree with the other two.

If all three sensors are operating properly, they should detect flame with equal accuracy.

Thus, they should either all register “low” (000: no flame) or all register “high” (111: good flame).

Any other output combination (001, 010, 011, 100, 101, or 110) constitutes a disagreement between sensors, and may therefore serve as an indicator of a potential sensor failure.

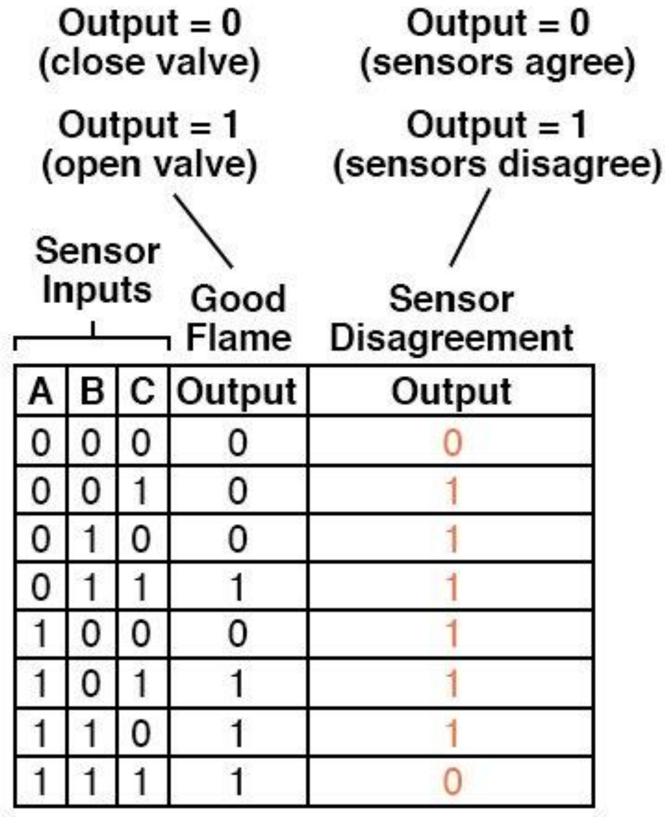
If we added circuitry to detect any one of the six “sensor disagreement” conditions, we could use the output of that circuitry to activate an alarm.

Whoever is monitoring the incinerator would then exercise judgment in either continuing to operate with a possible failed sensor (inputs: 011, 101, or 110), or shut the incinerator down to be absolutely safe.

Also, if the incinerator is shut down (no flame), and one or more of the sensors still indicates flame (001, 010, 011, 100, 101, or 110) while the other(s) indicate(s) no flame, it will be known that a definite sensor problem exists.

The first step in designing this “sensor disagreement” detection circuit is to write a truth table describing its behavior.

Since we already have a truth table describing the output of the “good flame” logic circuit, we can simply add another output column to the table to represent the second circuit, and make a table representing the entire logic system:



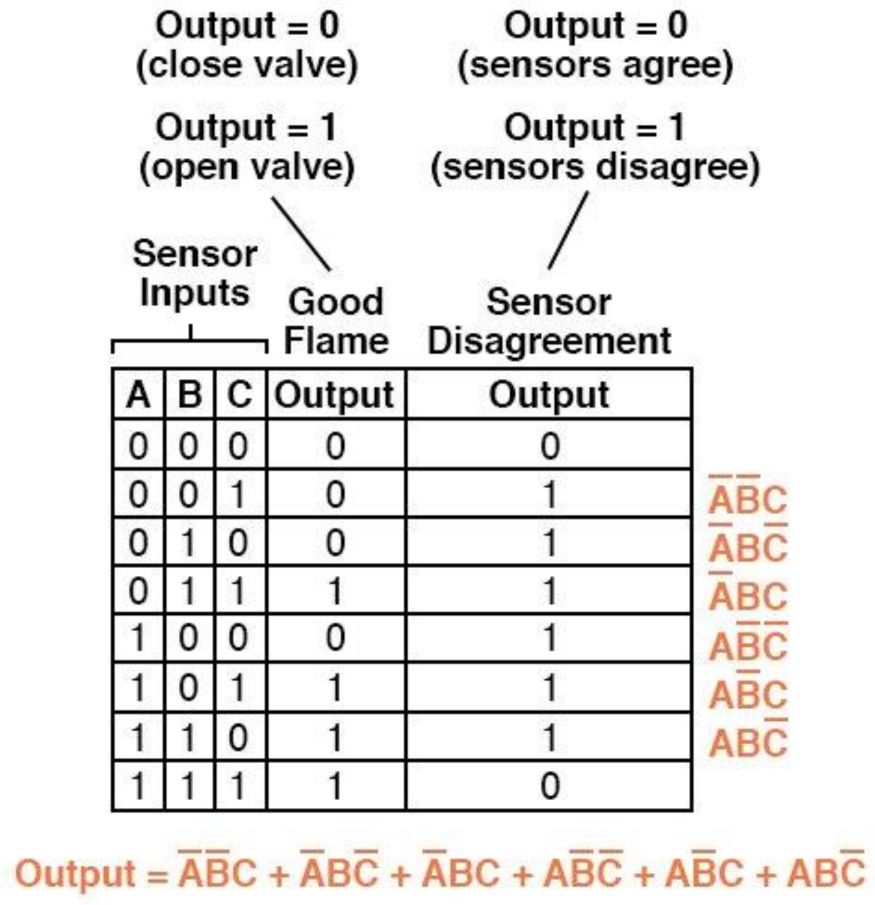
Output = 0 (close valve)
Output = 1 (open valve)

Output = 0 (sensors agree)
Output = 1 (sensors disagree)

Sensor Inputs			Good Flame	Sensor Disagreement
A	B	C	Output	Output
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

While it is possible to generate a Sum-Of-Products expression for this new truth table column, it would require six terms, of three variables each!

Such a Boolean expression would require many steps to simplify, with a large potential for making algebraic errors:


Output = 0
(close valve)
Output = 1
(open valve)
Output = 0
(sensors agree)
Output = 1
(sensors disagree)
Sensor Inputs
Good Flame
Sensor Disagreement

A	B	C	Output	Output	
0	0	0	0	0	
0	0	1	0	1	$\overline{A}\overline{B}C$
0	1	0	0	1	$\overline{A}B\overline{C}$
0	1	1	1	1	$\overline{A}BC$
1	0	0	0	1	$A\overline{B}\overline{C}$
1	0	1	1	1	$A\overline{B}C$
1	1	0	1	1	$AB\overline{C}$
1	1	1	1	0	

$$\text{Output} = \overline{A}\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C}$$

Using Product-Of-Sums

An alternative to generating a Sum-Of-Products expression to account for all the “high” (1) output conditions in the truth table is to generate a *Product-Of-Sums*, or *POS*, expression, to account for all the “low” (0) output conditions instead.

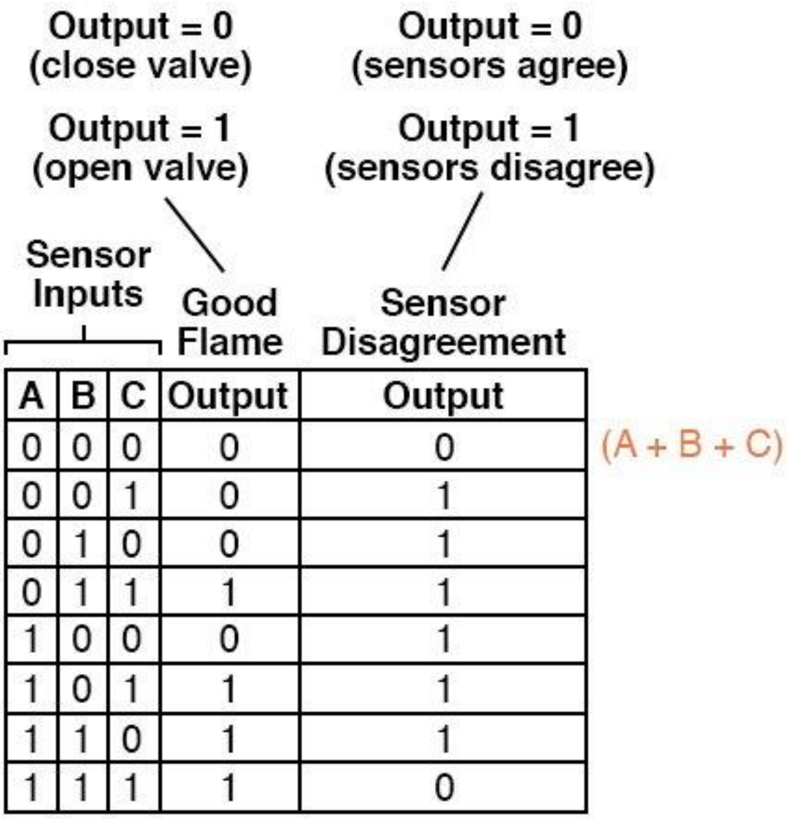
Being that there are much fewer instances of a “low” output in the last truth table column, the resulting Product-Of-Sums expression should contain fewer terms.

As its name suggests, a Product-Of-Sums expression is a set of added terms (*sums*), which are multiplied (*product*) together.

An example of a POS expression would be (A + B)(C + D), the product of the sums “A + B” and “C + D”.

To begin, we identify which rows in the last truth table column have “low” (0) outputs, and write a Boolean sum term that would equal 0 for that row’s input conditions.

For instance, in the first row of the truth table, where A=0, B=0, and C=0, the sum term would be (A + B + C), since that term would have a value of 0 if and only if A=0, B=0, and C=0:



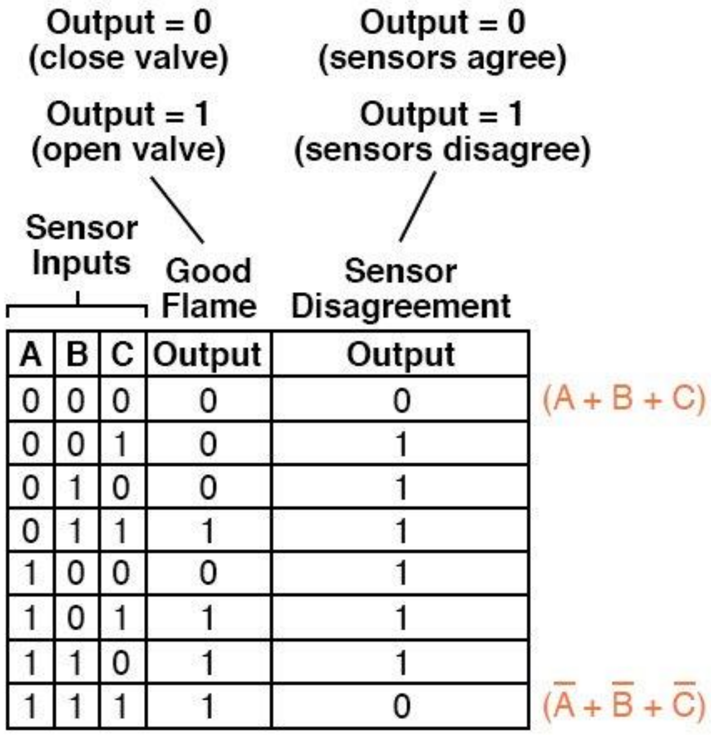
Output = 0
(close valve)
Output = 1
(open valve)
Output = 0
(sensors agree)
Output = 1
(sensors disagree)
Sensor Inputs
Good Flame
Sensor Disagreement
(A + B + C)

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

Only one other row in the last truth table column has a “low” (0) output, so all we need is one more sum term to complete our Product-Of-Sums expression.

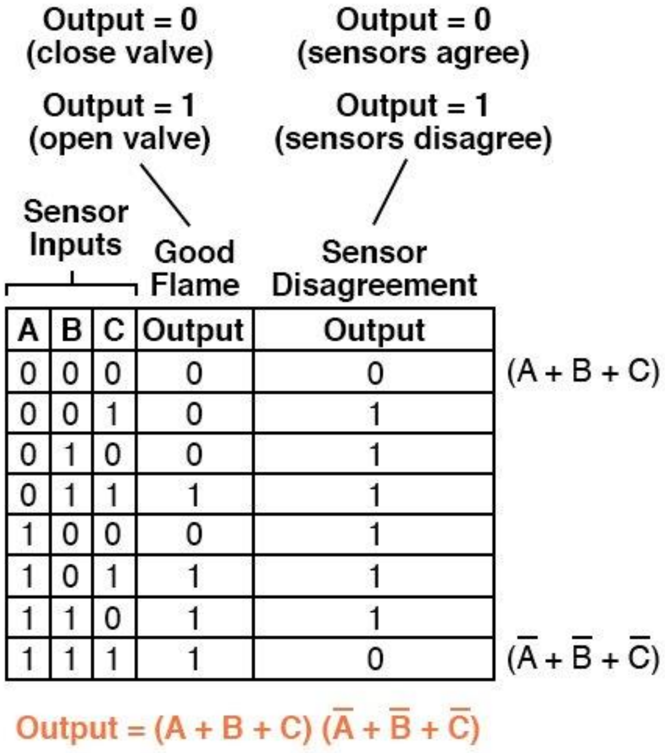
This last sum term represents a 0 output for an input condition of A=1, B=1 and C=1.

Therefore, the term (Maxterm) must be written as (A’ + B’+ C’), because only the sum of the complemented input variables would equal 0 for that condition only:


Output = 0
(close valve)
Output = 1
(open valve)
Output = 0
(sensors agree)
Output = 1
(sensors disagree)
Sensor
Inputs
Good
Flame
Sensor
Disagreement

A	B	C	Output	Output	
0	0	0	0	0	$(A + B + C)$
0	0	1	0	1	
0	1	0	0	1	
0	1	1	1	1	
1	0	0	0	1	
1	0	1	1	1	
1	1	0	1	1	
1	1	1	1	0	$(\overline{A} + \overline{B} + \overline{C})$

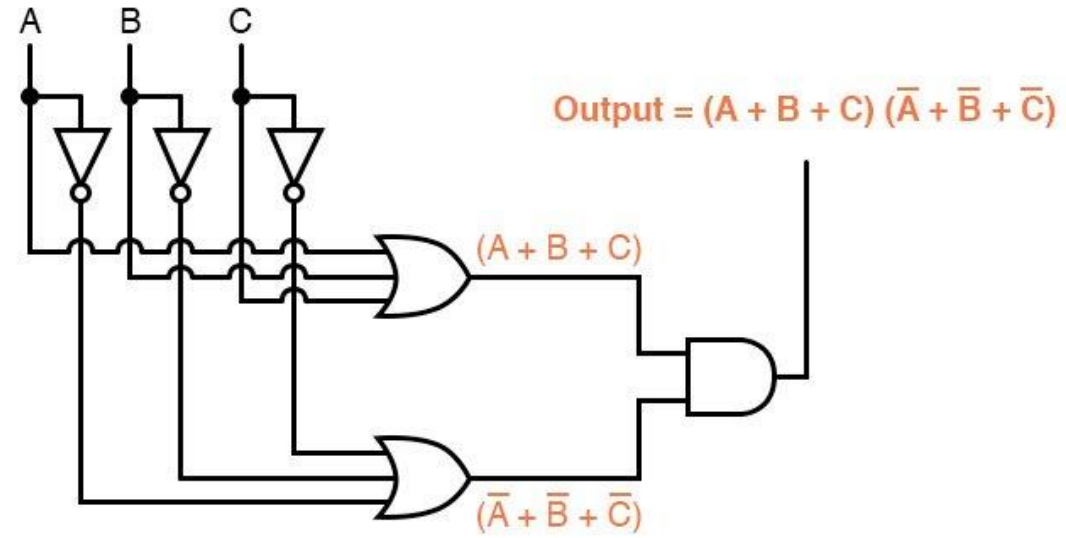
The completed Product-Of-Sums expression, of course, is the multiplicative combination of these two sum Maxterms:


Output = 0
(close valve)
Output = 1
(open valve)
Output = 0
(sensors agree)
Output = 1
(sensors disagree)
Sensor
Inputs
Good
Flame
Sensor
Disagreement

A	B	C	Output	Output	
0	0	0	0	0	$(A + B + C)$
0	0	1	0	1	
0	1	0	0	1	
0	1	1	1	1	
1	0	0	0	1	
1	0	1	1	1	
1	1	0	1	1	
1	1	1	1	0	$(\overline{A} + \overline{B} + \overline{C})$

$\text{Output} = (A + B + C)(\overline{A} + \overline{B} + \overline{C})$

Whereas a Sum-Of-Products expression could be implemented in the form of a set of AND gates with their outputs connecting to a single OR gate, a Product-Of-Sums expression can be implemented as a set of OR gates feeding into a single AND gate:


A
B
C
Output = (A + B + C) ($\overline{A}$ + $\overline{B}$ + $\overline{C}$)
(A + B + C)
($\overline{A}$ + $\overline{B}$ + $\overline{C}$)

As you can see, both the Sum-Of-Products and Products-Of-Sums standard Boolean forms are powerful tools when applied to truth tables.

They allow us to derive a Boolean expression—and ultimately, an actual logic circuit—from nothing but a truth table, which is a written specification for what we want a logic circuit to do.

To be able to go from a written specification to an actual circuit using simple, deterministic procedures means that it is possible to automate the design process for a digital circuit.

REVIEW:

- Sum-Of-Products, or SOP, Boolean expressions may be generated from truth tables quite easily, by determining which rows of the table have an output of 1, writing one product term for each row, and finally summing all the product terms. This creates a Boolean expression representing the truth table as a whole.
- Sum-Of-Products expressions lend themselves well to implementation as a set of AND gates (products) feeding into a single OR gate (sum).
- Product-Of-Sums, or POS, Boolean expressions may also be generated from truth tables quite easily, by determining which rows of the table have an output of 0, writing one sum term for each row, and finally multiplying all the sum terms. This creates a Boolean expression representing the truth table as a whole.
- Product-Of-Sums expressions lend themselves well to implementation as a set of OR gates (sums) feeding into a single AND gate (product).

Karnaugh Maps, Truth Tables, and Boolean Expressions

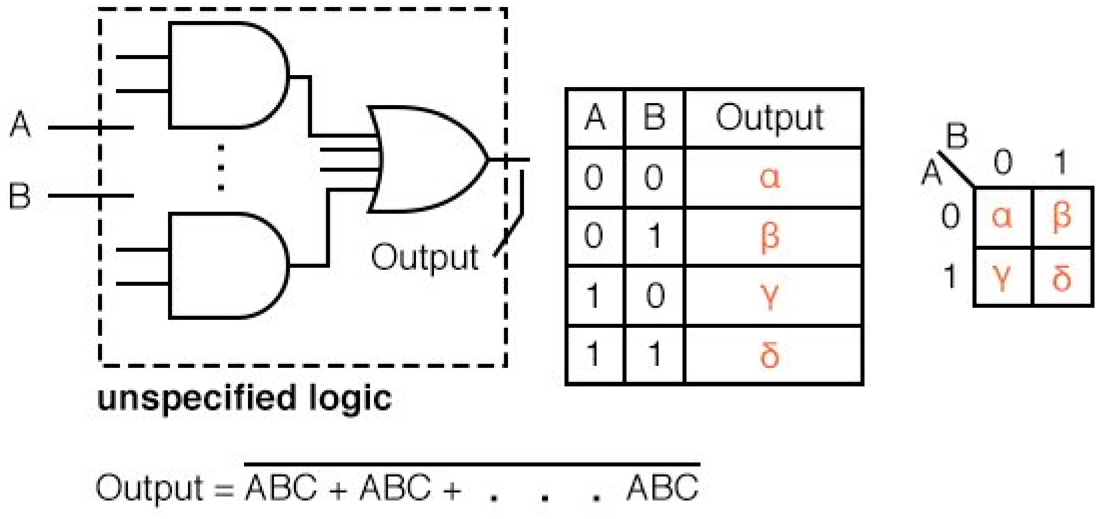
Maurice Karnaugh, a telecommunications engineer, developed the Karnaugh map at Bell Labs in 1953 while designing digital logic based telephone switching circuits.

The Use of Karnaugh Map

Karnaugh maps reduce logic functions more quickly and easily compared to Boolean algebra. By reduce we mean simplify, reducing the number of gates and inputs.

We like to simplify logic to a lowest cost form to save costs by elimination of components. We define lowest cost as being the lowest number of gates with the lowest number of inputs per gate.

Given a choice, most students do logic simplification with Karnaugh maps rather than Boolean algebra once they learn this tool.



A	B	Output
0	0	α
0	1	β
1	0	γ
1	1	δ

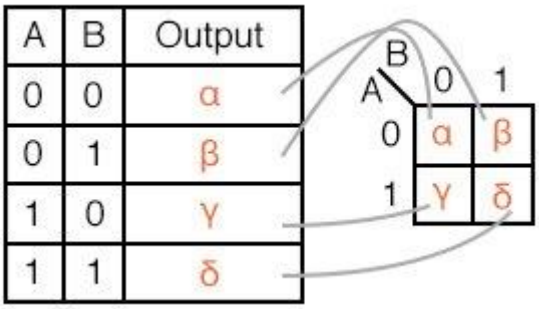
A \ B	0	1
0	α	β
1	γ	δ

$$\text{Output} = \overline{ABC + ABC + \;.\;\;.\;\;.\; ABC}$$

Two inputs A and B can take on values of either 0 or 1, high or low, open or closed, True or False, as the case may be. There are 2^2 = 4 combinations of inputs producing an output.

These outputs may be recorded in the truth table, or in the Karnaugh map. Look at the Karnaugh map as being a rearranged truth table.

The Output of the Boolean equation may be computed by the laws of Boolean algebra and transfered to the truth table or Karnaugh map.

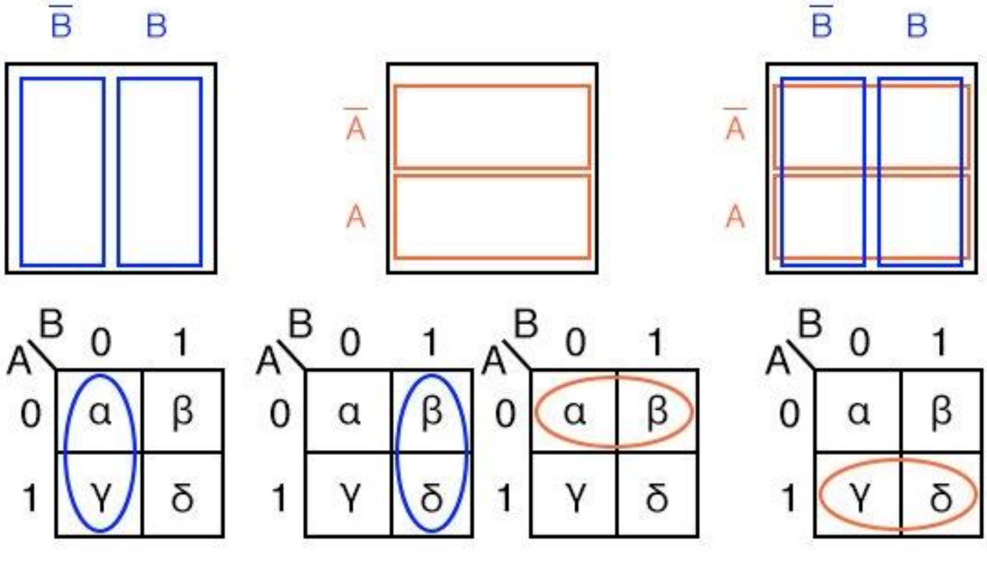


A	B	Output
0	0	α
0	1	β
1	0	γ
1	1	δ

The outputs of a truth table correspond on a one-to-one basis to Karnaugh map entries. Starting at the top of the truth table, the A=0, B=0 inputs produce an output α.

Note that this same output α is found in the Karnaugh map at the A=0, B=0 cell address, upper left corner of K-map where the A=0 row and B=0 column intersect. The other truth table outputs β, χ, δ from inputs AB=01, 10, 11 are found at corresponding K-map locations.

Below, we show the adjacent 2-cell regions in the 2-variable K-map like Boolean regions.



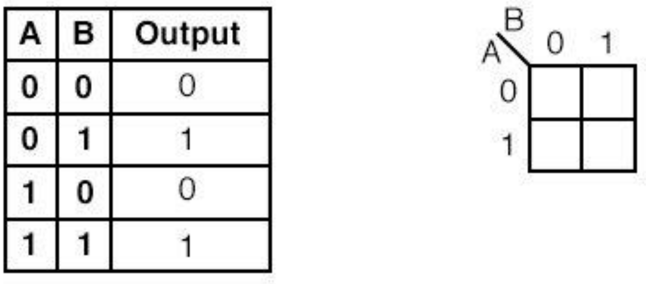
Cells α and χ are adjacent in the K-map as ellipses in the left most K-map below. Referring to the previous truth table, this is not the case. There is another truth table entry (β) between them. Which brings us to the whole point of the organizing the K-map into a square array, cells with any Boolean variables in common need to be close to one another so as to present a pattern that jumps out at us.

For cells α and χ they have the Boolean variable B’ in common. We know this because B=0 (same as B’) for the column above cells α and χ. Compare this to the square Venn diagram above the K-map.

A similar line of reasoning shows that β and δ have Boolean B (B=1) in common. Then, α and β have Boolean A’ (A=0) in common. Finally, χ and δ have Boolean A (A=1) in common. Compare the last two maps to the middle square Venn diagram.

To summarize, we are looking for commonality of Boolean variables among cells. The Karnaugh map is organized so that we may see that commonality. Let’s try some examples.

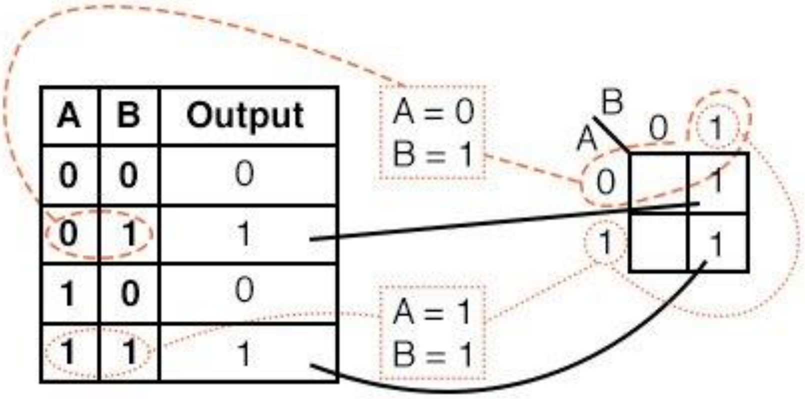
Examples



A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

Example:

Transfer the contents of the truth table to the Karnaugh map above.



A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

Solution:

The truth table contains two 1s. the K- map must have both of them. locate the first 1 in the 2nd row of the truth table above.

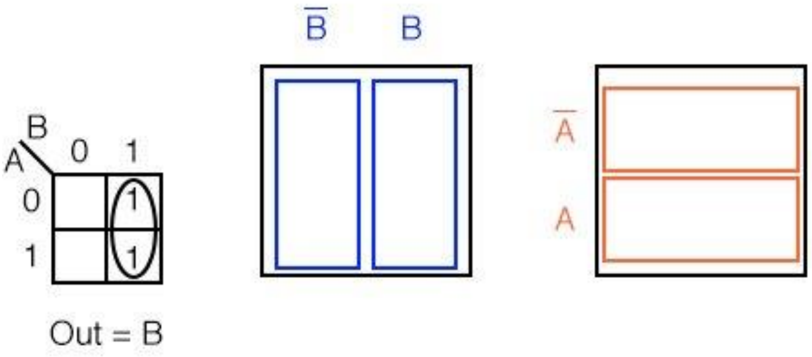
- note the truth table AB address

- locate the cell in the K-map having the same address
- place a 1 in that cell

Repeat the process for the 1 in the last line of the truth table.

Example:

For the Karnaugh map in the above problem, write the Boolean expression. Solution is below.



Solution:

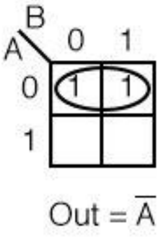
Look for adjacent cells, that is, above or to the side of a cell. Diagonal cells are not adjacent. Adjacent cells will have one or more Boolean variables in common.

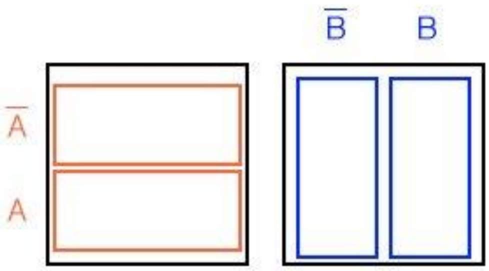
- Group (circle) the two 1s in the column
- Find the variable(s) top and/or side which are the same for the group, Write this as the Boolean result. It is B in our case.
- Ignore variable(s) which are not the same for a cell group. In our case A varies, is both 1 and 0, ignore Boolean A.
- Ignore any variable not associated with cells containing 1s. B’ has no ones under it. Ignore B’
- Result Out = B

This might be easier to see by comparing to the Venn diagrams to the right, specifically the B column.

Example:

Write the Boolean expression for the Karnaugh map below.



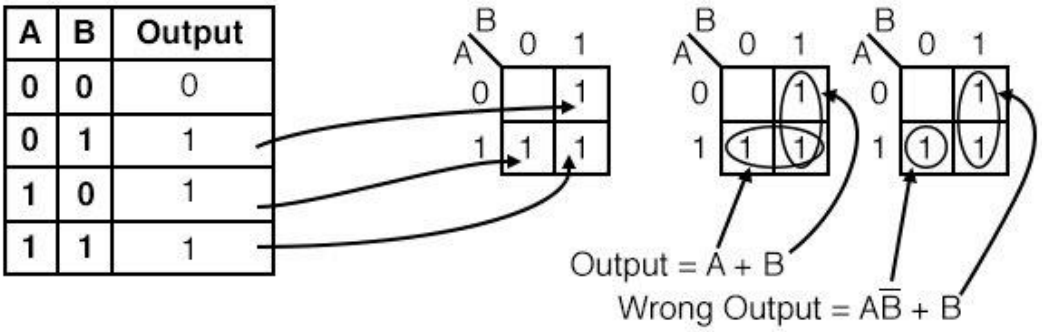


Solution: (above)

- Group (circle) the two 1’s in the row
- Find the variable(s) which are the same for the group, Out = A’

Example:

For the Truth table below, transfer the outputs to the Karnaugh, then write the Boolean expression for the result.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Solution:

Transfer the 1s from the locations in the Truth table to the corresponding locations in the K-map.

- Group (circle) the two 1’s in the column under B=1
- Group (circle) the two 1’s in the row right of A=1
- Write product term for first group = B
- Write product term for second group = A
- Write Sum-Of-Products of above two terms Output = A+B

The solution of the K-map in the middle is the simplest or lowest cost solution. A less desirable solution is at far right. After grouping the two 1s, we make the mistake of forming a group of 1-cell. The reason that this is not desirable is that:

- The single cell has a product term of AB’

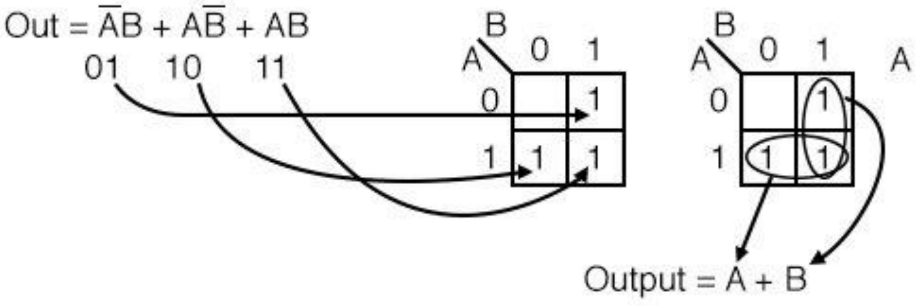
- The corresponding solution is Output = AB’ + B
- This is not the simplest solution

The way to pick up this single 1 is to form a group of two with the 1 to the right of it as shown in the lower line of the middle K-map, even though this 1 has already been included in the column group (B). We are allowed to re-use cells in order to form larger groups. In fact, it is desirable because it leads to a simpler result.

We need to point out that either of the above solutions, Output or Wrong Output, are logically correct. Both circuits yield the same output. It is a matter of the former circuit being the lowest cost solution.

Example:

Fill in the Karnaugh map for the Boolean expression below, then write the Boolean expression for the result.



Solution: (above)

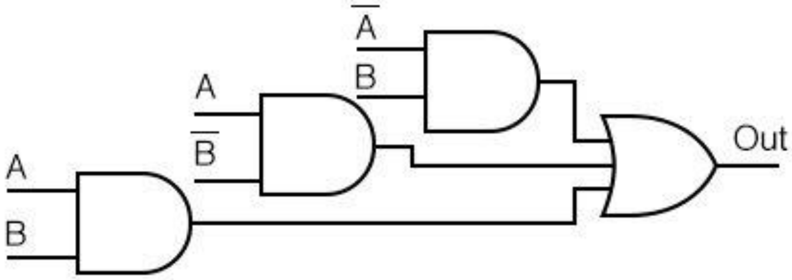
The Boolean expression has three product terms. There will be a 1 entered for each product term. Though, in general, the number of 1s per product term varies with the number of variables in the product term compared to the size of the K-map.

The product term is the address of the cell where the 1 is entered. The first product term, A’B, corresponds to the 01 cell in the map. A 1 is entered in this cell. The other two P-terms are entered for a total of three 1s

Next, proceed with grouping and extracting the simplified result as in the previous truth table problem.

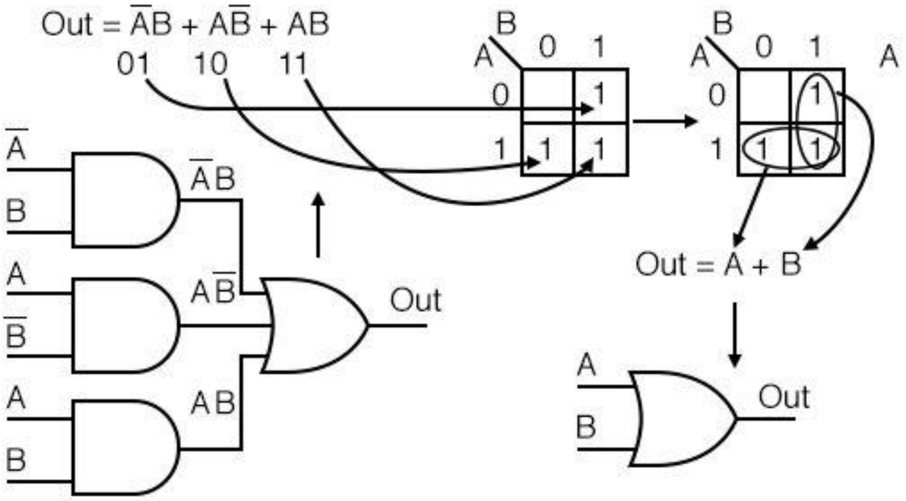
Example:

Simplify the logic diagram below.

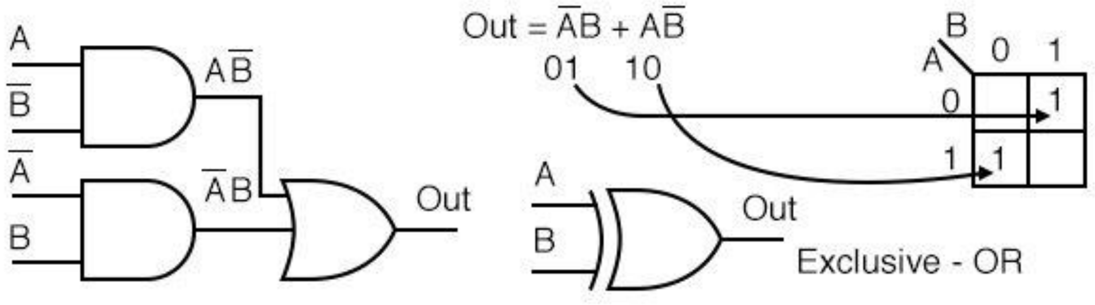


Solution: (Figure below)

- Write the Boolean expression for the original logic diagram as shown below
- Transfer the product terms to the Karnaugh map
- Form groups of cells as in previous examples
- Write Boolean expression for groups as in previous examples
- Draw simplified logic diagram



Example: Simplify the logic diagram below.



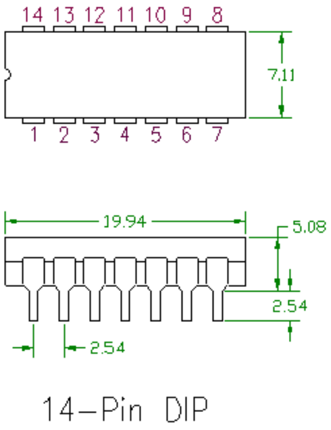
Solution:

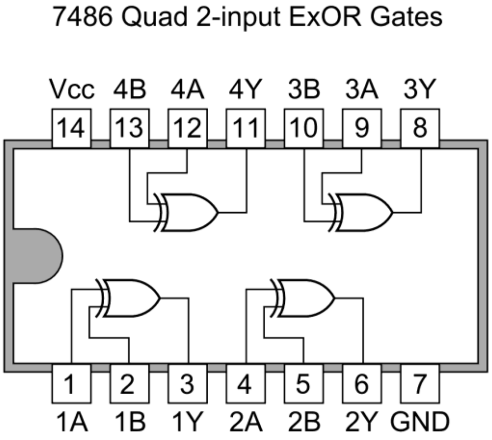
- Write the Boolean expression for the original logic diagram shown above
- Transfer the product terms to the Karnaugh map.
- It is not possible to form groups.
- No simplification is possible; leave it as it is.

No logic simplification is possible for the above diagram. This sometimes happens. Neither the methods of Karnaugh maps nor Boolean algebra can simplify this logic further.

We show an Exclusive-OR schematic symbol above; however, this is not a logical simplification. It just makes a schematic diagram look nicer.

Since it is not possible to simplify the Exclusive-OR logic and it is widely used, it is provided by manufacturers as a basic integrated circuit (7486).





Logic Simplification With Karnaugh Maps

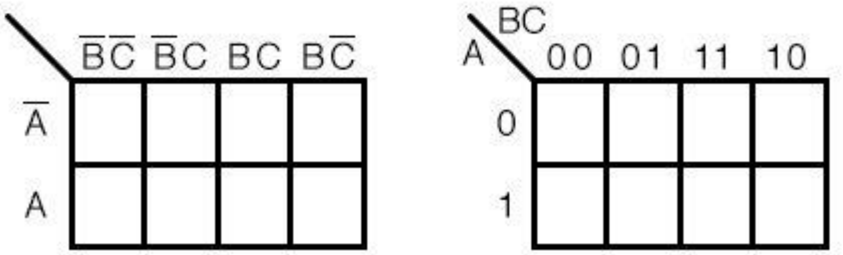
The logic simplification examples that we have done so far could have been performed with Boolean algebra about as quickly. Real world logic simplification problems call for larger Karnaugh maps so that we may do serious work.

We will work some contrived examples below. By contrived, we mean examples which illustrate techniques.

This approach will develop the tools we need to transition to the more complex applications.

Karnaugh Maps and Gray Code Sequence

We show our previously developed Karnaugh map. We will use the form on the right.



Note the sequence of numbers across the top of the map. It is not in binary sequence which would be **00, 01, 10, 11**. It is **00, 01, 11 10**, which is Gray code sequence. Gray code sequence only changes one binary bit as we go from one number to the next in the sequence, unlike binary.

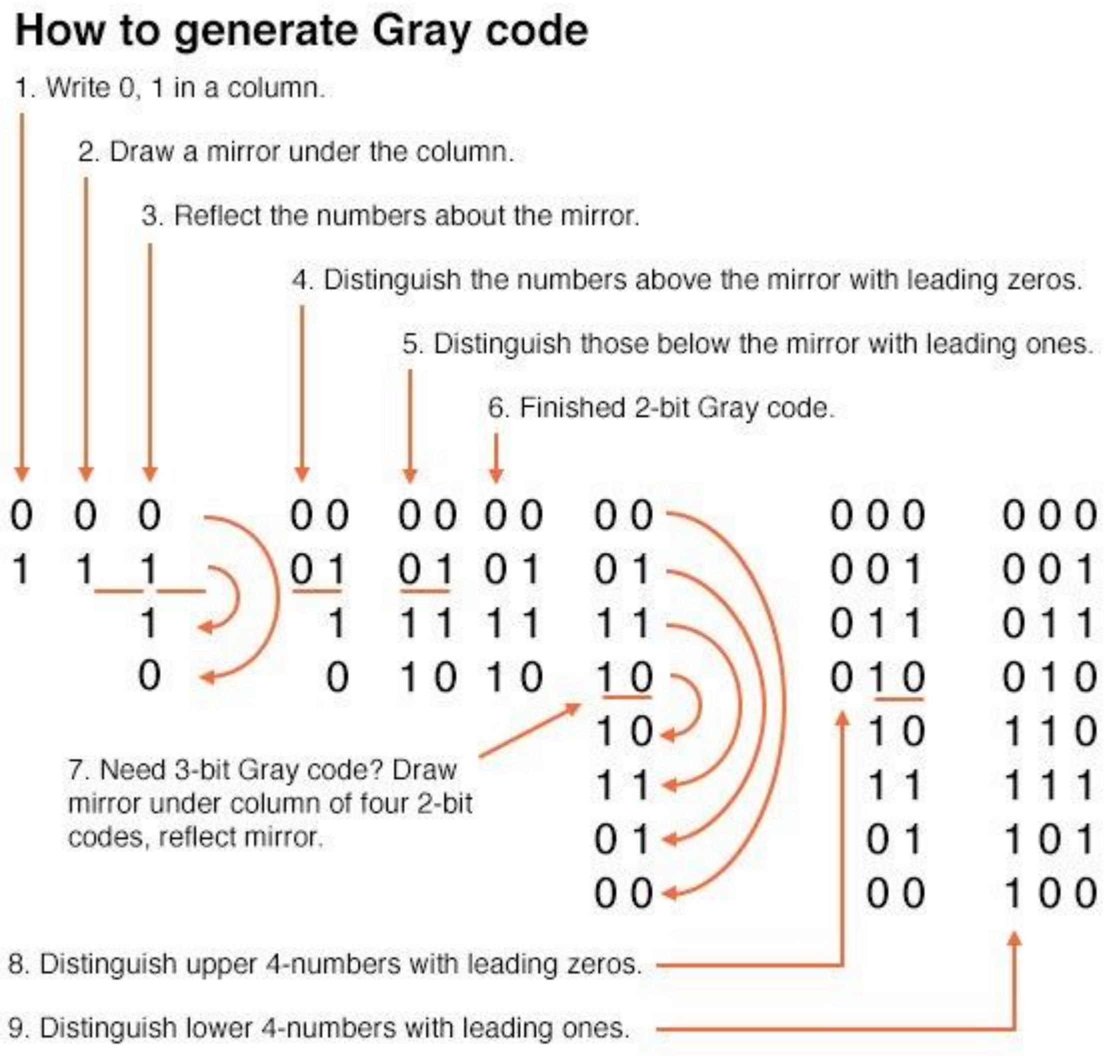
That means that adjacent cells will only vary by one bit, or Boolean variable. This is what we need to organize the outputs of a logic function so that we may view commonality.

Moreover, the column and row headings must be in Gray code order, or the map will not work as a Karnaugh map. Cells sharing common Boolean variables would no longer be adjacent, nor show visual patterns.

Adjacent cells vary by only one bit because a Gray code sequence varies by only one bit.

Generating Gray Code

If we sketch our own Karnaugh maps, we need to generate Gray code for any size map that we may use. This is how we generate Gray code of any size.

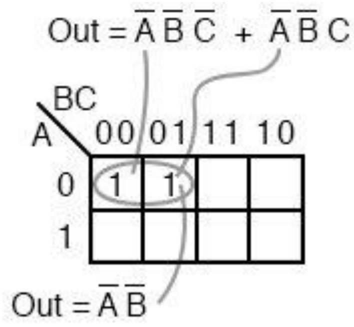

How to generate Gray code
1. Write 0, 1 in a column.
2. Draw a mirror under the column.
3. Reflect the numbers about the mirror.
4. Distinguish the numbers above the mirror with leading zeros.
5. Distinguish those below the mirror with leading ones.
6. Finished 2-bit Gray code.
7. Need 3-bit Gray code? Draw mirror under column of four 2-bit codes, reflect mirror.
8. Distinguish upper 4-numbers with leading zeros.
9. Distinguish lower 4-numbers with leading ones.

Note that the Gray code sequence, above right, only varies by one bit as we go down the list, or bottom to top up the list. This property of Gray code is often useful for digital electronics in general. In particular, it is applicable to Karnaugh maps.

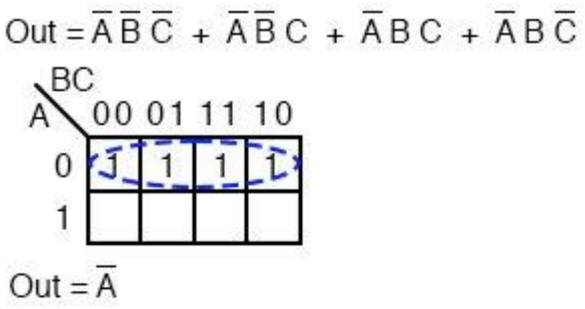
Examples of Simplification with Karnaugh Maps

Let us move on to some examples of simplification with 3-variable Karnaugh maps. We show how to map the product terms of the unsimplified logic to the K-map.

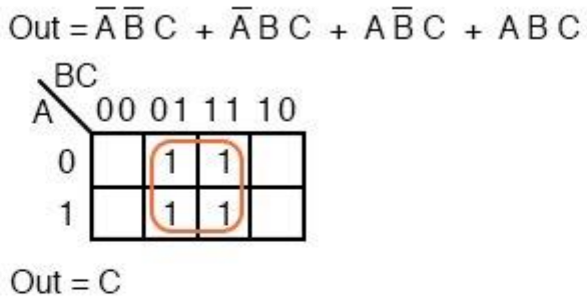
We illustrate how to identify groups of adjacent cells which leads to a Sum-of-Products simplification of the digital logic.


Out = $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C$
BC
A
00 01 11 10
0
1
1 1
Out = $\overline{A}\,\overline{B}$

Above we, place the 1’s in the K-map for each of the product terms, identify a group of two, then write a *p-term* (product term) for the sole group as our simplified result.


Out = $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,B\,\overline{C}$
BC
A
00 01 11 10
0
1
1 1 1 1
Out = $\overline{A}$

Mapping the four product terms above yields a group of four covered by Boolean **A’**

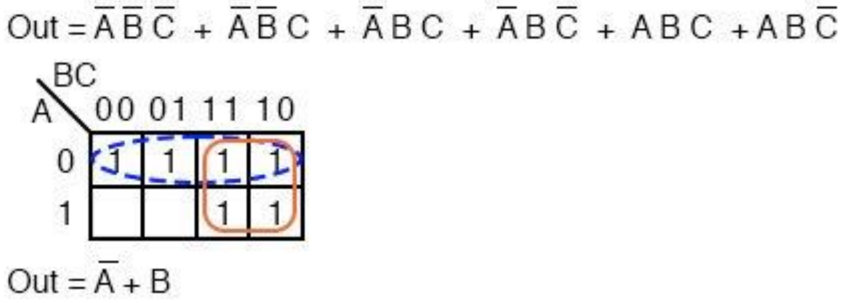


$Out = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,C$

A \ BC	00	01	11	10
0		1	1	
1		1	1	

$Out = C$

Mapping the four p-terms yields a group of four, which is covered by one variable **C**.



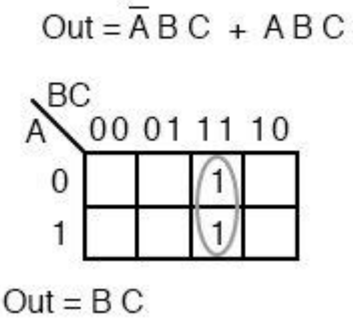
$Out = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,B\,\overline{C} + A\,B\,C + A\,B\,\overline{C}$

A \ BC	00	01	11	10
0	1	1	1	1
1			1	1

$Out = \overline{A} + B$

After mapping the six p-terms above, identify the upper group of four, pick up the lower two cells as a group of four by sharing the two with two more from the other group. Covering these two with a group of four gives a simpler result.

Since there are two groups, there will be two p-terms in the Sum-of-Products result **A'+B**

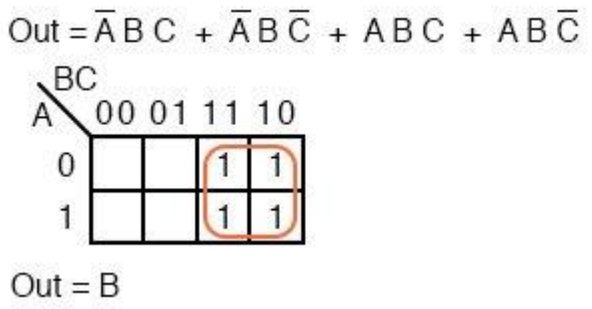


$Out = \overline{A}\,B\,C + A\,B\,C$

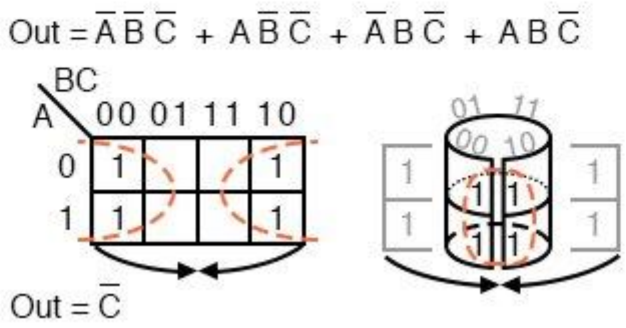
A \ BC	00	01	11	10
0			1	
1			1	

$Out = B\,C$

The two product terms above form one group of two and simplifies to **BC**

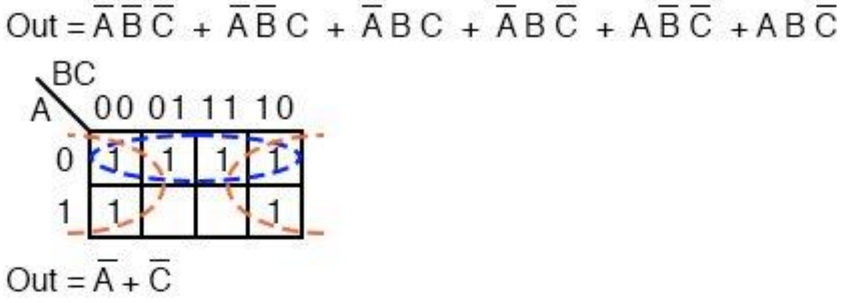

$\text{Out} = \overline{A}BC + \overline{A}B\overline{C} + ABC + AB\overline{C}$
BC
A
00 01 11 10
0
1
1 1
1 1
Out = B

Mapping the four p-terms yields a single group of four, which is **B**


$\text{Out} = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + \overline{A}B\overline{C} + AB\overline{C}$
BC
A
00 01 11 10
0
1
1 1
1 1
01 11
00 10
1 1
1 1
$\text{Out} = \overline{C}$

Mapping the four p-terms above yields a group of four. Visualize the group of four by rolling up the ends of the map to form a cylinder, then the cells are adjacent. We normally mark the group of four as above left.

Out of the variables A, B, C, there is a common variable: C’. C’ is a 0 overall four cells. The final result is **C’**

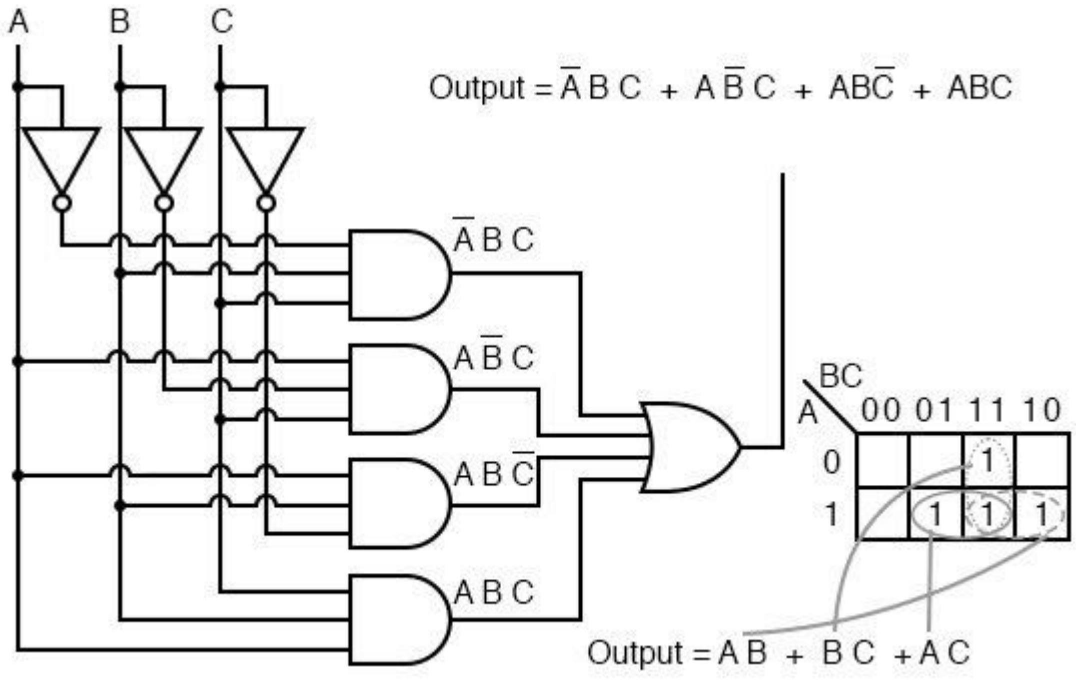

$\text{Out} = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$
BC
A
00 01 11 10
0
1
1 1 1 1
1 1
$\text{Out} = \bar{A} + \bar{C}$

.

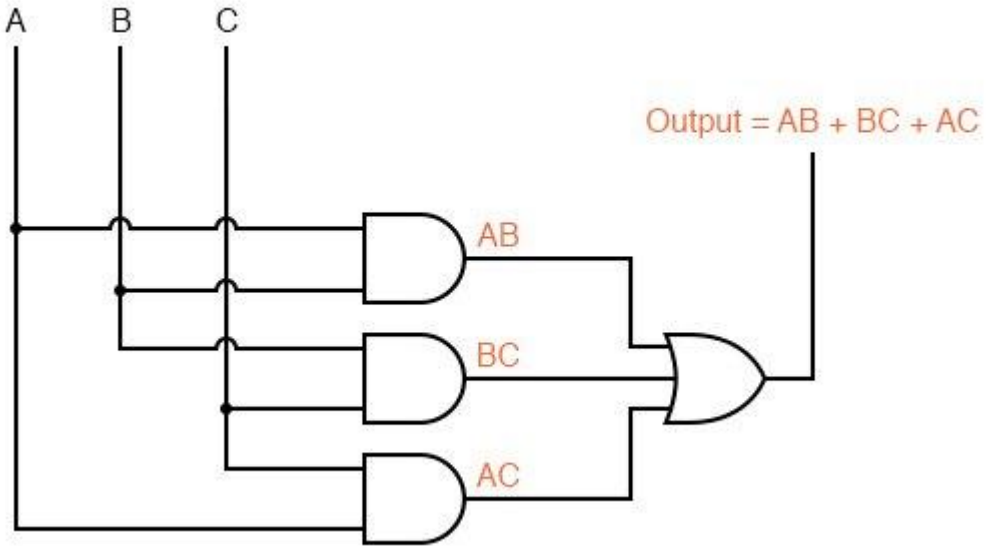
The six cells above from the unsimplified equation can be organized into two groups of four. These two groups should give us two p-terms in our simplified result of **A’ + C’**.

Simplifying Boolean Equations with Karnaugh Maps

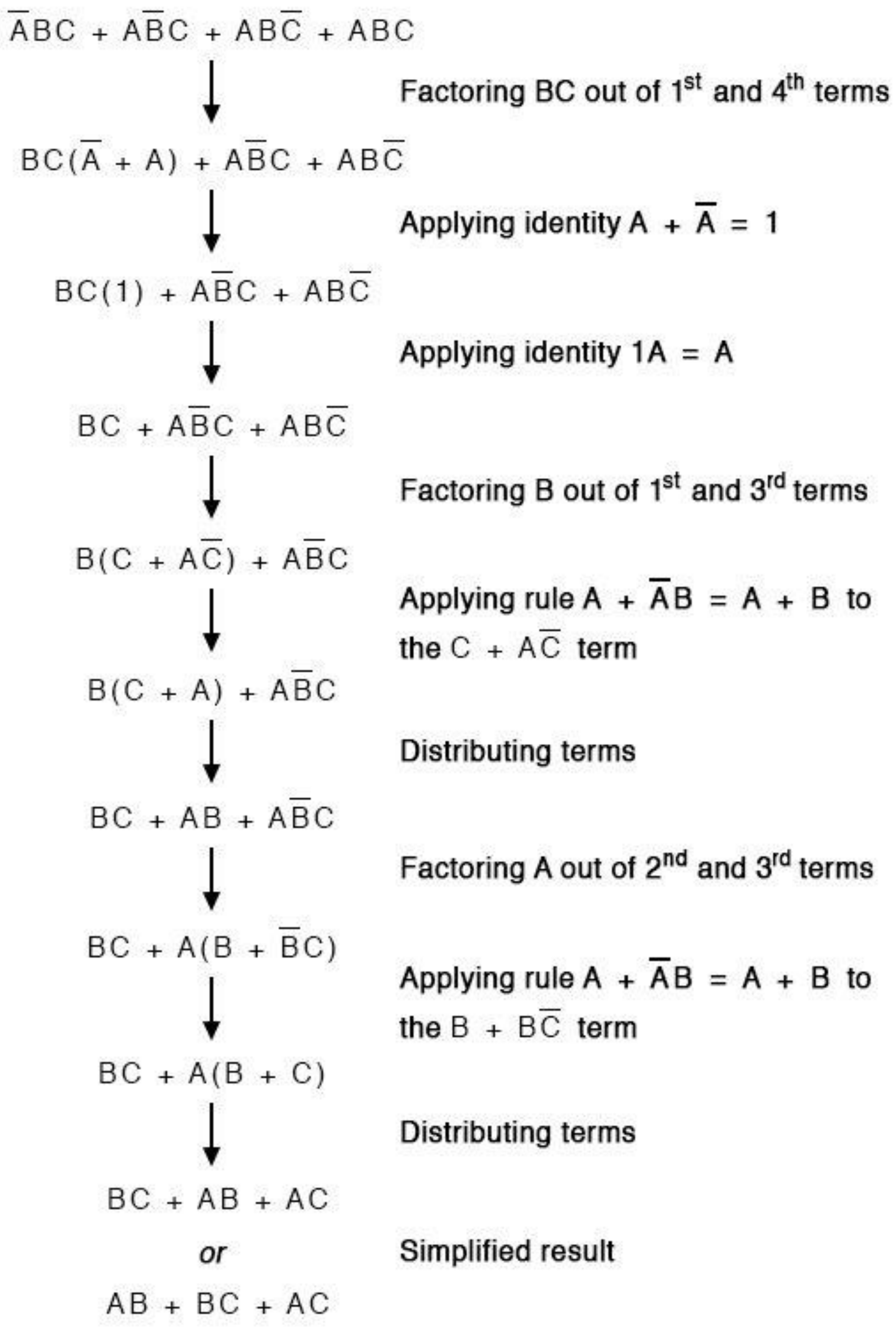
Below, we revisit the toxic waste incinerator from the Boolean algebra chapter. See Boolean algebra chapter for details on this example. We will simplify the logic using a Karnaugh map.


A
B
C
$\text{Output} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$
$\bar{A}BC$
$A\bar{B}C$
$AB\bar{C}$
ABC
BC
A
00 01 11 10
0
1
1
1 1 1
$\text{Output} = AB + BC + AC$

The Boolean equation for the output has four product terms. Map four 1's corresponding to the p-terms. Forming groups of cells, we have three groups of two. There will be three p-terms in the simplified result, one for each group. See Converting Truth Tables into Boolean Expressions from chapter 7 for a gate diagram of the result, which is reproduced below.



Below we repeat the Boolean algebra simplification of the toxic waste incinerator for comparison.



$\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

Factoring BC out of 1st and 4th terms

$BC(\overline{A} + A) + A\overline{B}C + AB\overline{C}$

Applying identity $A + \overline{A} = 1$

$BC(1) + A\overline{B}C + AB\overline{C}$

Applying identity $1A = A$

$BC + A\overline{B}C + AB\overline{C}$

Factoring B out of 1st and 3rd terms

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

Applying rule $A + \overline{A}B = A + B$ to the $C + A\overline{C}$ term

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

Distributing terms

$BC + AB + A\overline{B}C$

Factoring A out of 2nd and 3rd terms

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

Applying rule $A + \overline{A}B = A + B$ to the $B + B\overline{C}$ term

$BC + A(B + C)$

Distributing terms

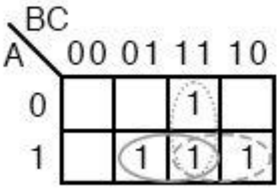
$BC + AB + AC$

or

$AB + BC + AC$

Simplified result

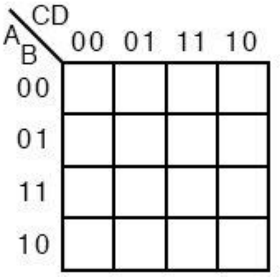
This case illustrates why the Karnaugh map is widely used for logic simplification.



Output $= AB + BC + AC$

Larger 4-variable Karnaugh Maps

Knowing how to generate Gray code should allow us to build larger maps. Actually, all we need to do is look at the left to right sequence across the top of the 3-variable map, and copy it down the left side of the 4-variable map. See below.

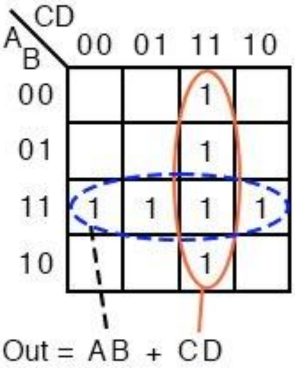


Reduction of 4 Variable K Maps

The following four variable Karnaugh maps illustrate the reduction of Boolean expressions too tedious for Boolean algebra. Reductions could be done with Boolean algebra.

However, the Karnaugh map is faster and easier, especially if there are many logic reductions to do.

$$\text{Out} = \overline{A}\,\overline{B}CD + \overline{A}BCD + ABCD + A\overline{B}CD + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABC\overline{D}$$



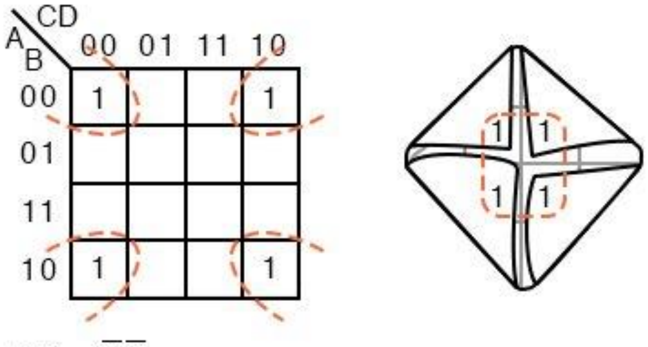
The above Boolean expression has seven product terms. They are mapped top to bottom and left to right on the K-map above. For example, the first P-term **A’B’CD** is the first row, 3rd cell, corresponding to map location **A=0, B=0, C=1, D=1**.

The other product terms are placed in a similar manner. Encircling the largest groups possible, two groups of four are shown above.

The dashed horizontal group corresponds to the simplified product term **AB**. The vertical group corresponds to Boolean CD. Since there are two groups, there will be two product terms in the Sum-Of-Products result of **Out=AB+CD**.

Fold up the corners of the map below like it is a napkin to make the four cells physically adjacent.

$$Out = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}C\overline{D}$$



$$Out = \overline{B}\,\overline{D}$$

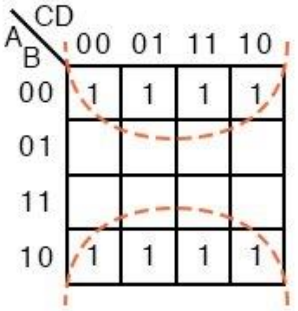
The four cells above are a group of four because they all have the Boolean variables **B’** and **D’** in common. In other words, **B=0** for the four cells, and **D=0** for the four cells.

The other variables **(A, C)** are **0** in some cases, **1** in other cases with respect to the four corner cells.

Thus, these variables **(A, C)** are not involved with this group of four. This single group comes out of the map as one product term for the simplified result: **Out=B’D’**

For the K-map below, roll the top and bottom edges into a cylinder forming eight adjacent cells.

$$Out = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}CD + A\overline{B}C\overline{D}$$

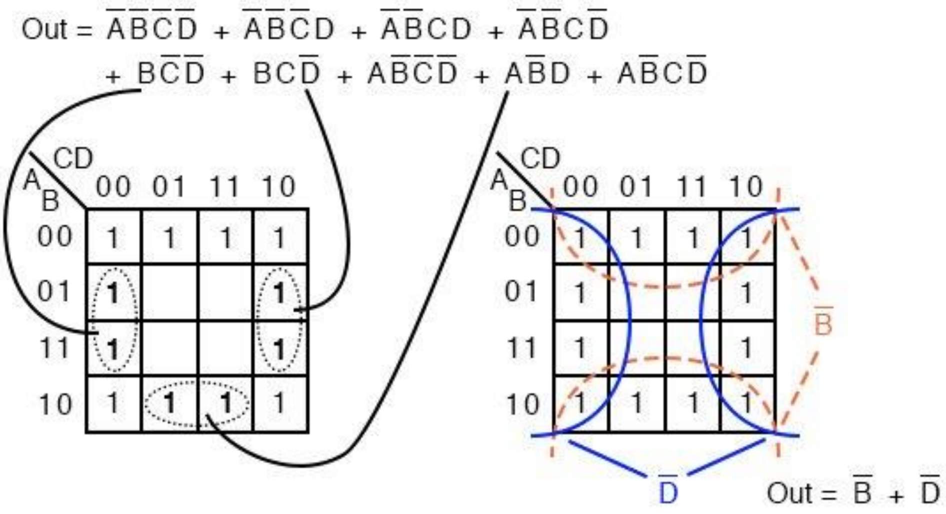


$$Out = \overline{B}$$

The above group of eight has one Boolean variable in common: **B=0**. Therefore, the one group of eight is covered by one p-term: **B’**. The original eight-term Boolean expression simplifies to **Out=B’**

P-Terms in 4 Variable K Maps

The Boolean expression below has nine p-terms, three of which have three Booleans instead of four. The difference is that while four Boolean variable product terms cover one cell, the three Boolean p-terms cover a pair of cells each.


$\text{Out} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}C\overline{D}$
$+ B\overline{C}\,\overline{D} + BC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}D + A\overline{B}C\overline{D}$
CD
AB
00 01 11 10
00
01
11
10
1 1 1 1
1 1
1 1
1 1 1 1
CD
AB
00 01 11 10
00
01
11
10
1 1 1 1
1 1
1 1
1 1 1 1
$\overline{B}$
$\overline{D}$
$\text{Out} = \overline{B} + \overline{D}$

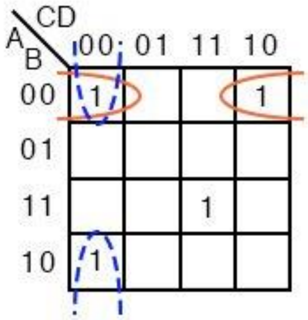
The six product terms of four Boolean variables map in the usual manner above as single cells. The three Boolean variable terms (three each) map as cell pairs, which is shown above.

Note that we are mapping p-terms into the K-map, not pulling them out at this point.

For the simplification, we form two groups of eight. Cells in the corners are shared with both groups. This is fine. In fact, this leads to a better solution than forming a group of eight and a group of four without sharing any cells. Final Solution is **Out=B’+D’**

Below we map the unsimplified Boolean expression to the Karnaugh map.

$$Out = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + A\overline{B}\overline{C}\overline{D} + ABCD$$

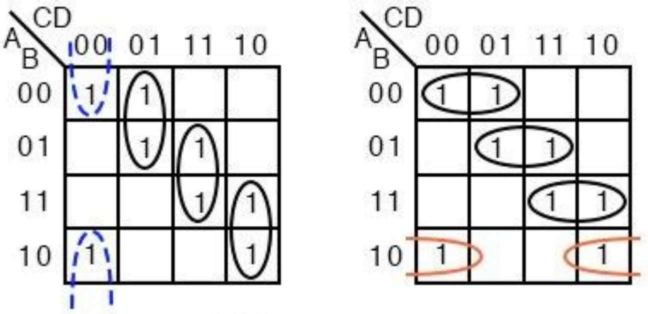


$$Out = \overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{D} + ABCD$$

Above, three of the cells form into groups of two cells. A fourth cell cannot be combined with anything, which often happens in “real world” problems. In this case, the Boolean p-term **ABCD** is unchanged in the simplification process. Result: **Out= B’C’D’+A’B’D’+ABCD**

Often times there is more than one minimum cost solution to a simplification problem. Such is the case illustrated below.

$$Out = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}B\overline{C}D + \overline{A}BCD$$
$$+ ABCD + ABC\overline{D} + A\overline{B}\overline{C}\overline{D} + A\overline{B}C\overline{D}$$



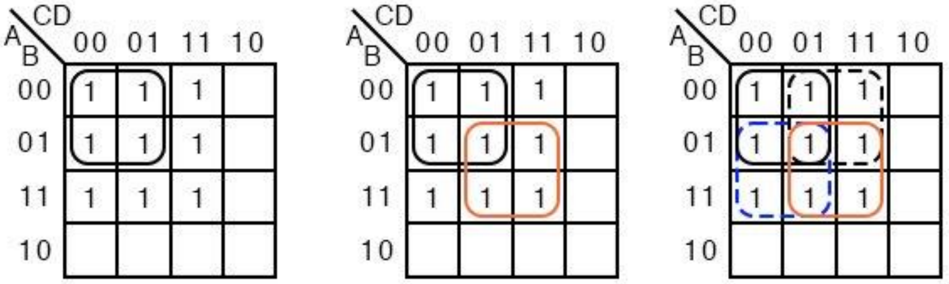
$$Out = \overline{B}\overline{C}\overline{D} + \overline{A}\overline{C}D + BCD + AC\overline{D}$$
$$Out = \overline{A}\overline{B}\overline{C} + \overline{A}BD + ABC + A\overline{B}\overline{D}$$

Both results above have four product terms of three Boolean variable each. Both are equally valid *minimal* cost solutions. The difference in the final solution is due to how the cells are grouped as shown above.

A minimal cost solution is a valid logic design with the minimum number of gates with the minimum number of inputs.

Below we map the unsimplified Boolean equation as usual and form a group of four as a first simplification step. It may not be obvious how to pick up the remaining cells.

$$\text{Out} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD$$



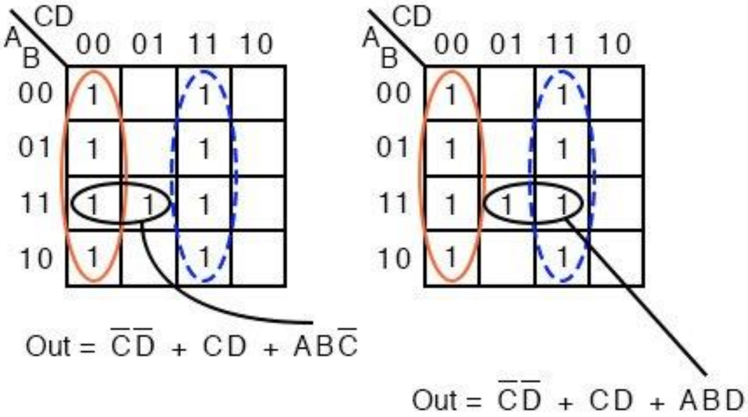
$$\text{Out} = \overline{A}\,\overline{C} + \overline{A}D + B\overline{C} + BD$$

Pick up three more cells in a group of four, center above. There are still two cells remaining. the minimal cost method to pick up those is to group them with neighboring cells as groups of four as at above right.

On a cautionary note, do not attempt to form groups of three. Groupings must be powers of 2, that is, 1, 2, 4, 8 ...

Below we have another example of two possible minimal cost solutions. Start by forming a couple of groups of four after mapping the cells.

$$\text{Out} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}BCD + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}CD$$



$$\text{Out} = \overline{C}\,\overline{D} + CD + AB\overline{C}$$

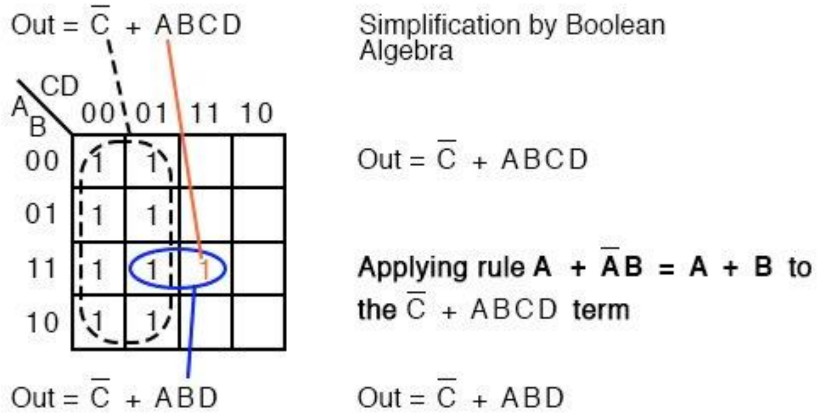
$$\text{Out} = \overline{C}\,\overline{D} + CD + ABD$$

The two solutions depend on whether the single remaining cell is grouped with the first or the second group of four as a group of two cells. That cell either comes out as either **ABC’** or **ABD**, your choice.

Either way, this cell is covered by either Boolean product term. Final results are shown above.

Below we have an example of a simplification using the Karnaugh map at left or Boolean algebra at right. Plot **C’** on the map as the area of all cells covered by address **C=0**, the 8-cells on the left of the map. Then, plot the single **ABCD** cell.

That single cell forms a group of 2-cell as shown, which simplifies to P-term **ABD**, for an end result of **Out = C’ + ABD**.



Simplification by Boolean Algebra

Out = $\overline{C}$ + ABCD

Applying rule $A + \overline{A}B = A + B$ to the $\overline{C}$ + ABCD term

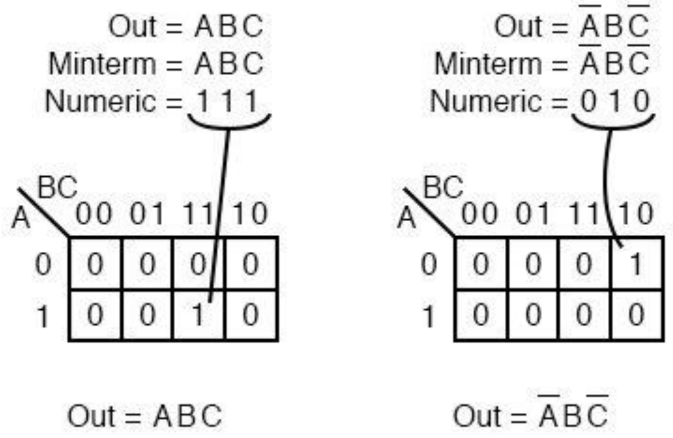
Out = $\overline{C}$ + ABD

This (above) is a rare example of a four-variable problem that can be reduced with Boolean algebra without a lot of work, assuming that you remember the theorems.

Minterm and Maxterm Solution

So far we have been finding Sum-Of-Product (SOP) solutions to logic reduction problems. For each of these SOP solutions, there is also a Product-Of-Sums solution (POS), which could be more useful, depending on the application.

Before working a Product-Of-Sums solution, we need to introduce some new terminology. We just want to establish a formal procedure for minterms for comparison to the new procedure for maxterms.



Minterm

A *minterm* is a Boolean expression resulting in **1** for the output of a single cell, and **0**s for all other cells in a Karnaugh map, or truth table. If a minterm has a single **1** and the remaining cells as **0**s, it would appear to cover a minimum area of **1**s.

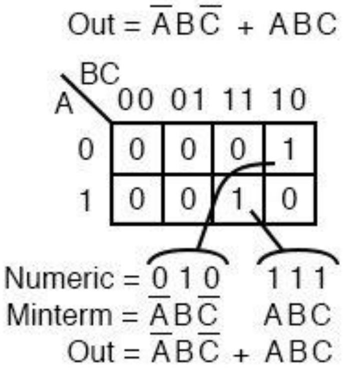
The illustration above left shows the minterm **ABC**, a single product term, as a single **1** in a map that is otherwise **0**s. We have not shown the **0**s in our Karnaugh maps up to this point, as it is customary to omit them unless specifically needed. Another minterm **A'BC'** is shown above right.

The point to review is that the address of the cell corresponds directly to the minterm being mapped. That is, the cell **111** corresponds to the minterm **ABC** above left.

Above right we see that the minterm **A'BC'** corresponds directly to the cell **010**. A Boolean expression or map may have multiple minterms.

Referring to the above figure, Let's summarize the procedure for placing a minterm in a K-map:

- Identify the minterm (product term) term to be mapped.
- Write the corresponding binary numeric value.
- Use binary value as an address to place a 1 in the K-map
- Repeat steps for other minterms (P-terms within a *Sum-Of-Products*).


Out = $\overline{A}B\overline{C}$ + ABC
BC
A
00 01 11 10
0
1
0 0 0 1
0 0 1 0
Numeric = 0 1 0 1 1 1
Minterm = $\overline{A}B\overline{C}$ ABC
Out = $\overline{A}B\overline{C}$ + ABC

A Boolean expression will more often than not consist of multiple minterms corresponding to multiple cells in a Karnaugh map as shown above. The multiple minterms in this map are the individual minterms which we examined in the previous figure above.

The point we review for reference is that the **1**s come out of the K-map as a binary cell address which converts directly to one or more product terms.

By directly we mean that a **0** corresponds to a complemented variable, and a **1** corresponds to a true variable. Example: **010** converts directly to **A'BC'**.

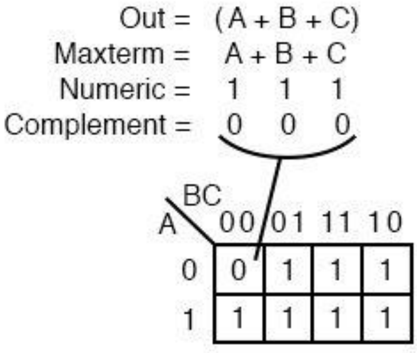
There was no reduction in this example. Though, we do have a Sum-Of-Products result from the minterms.

Referring to the above figure, Let's summarize the procedure for writing the Sum-Of-Products reduced Boolean equation from a K-map:

- Form largest groups of **1**s possible covering all minterms. Groups must be a power of 2.
- Write binary numeric value for groups.
- Convert binary value to a product term.
- Repeat steps for other groups. Each group yields a p-terms within a Sum-Of-Products.

Nothing new so far, a formal procedure has been written down for dealing with minterms. This serves as a pattern for dealing with maxterms.

Next we check the Boolean function which is 0 for a single cell and 1s for all others.



Maxterm

A *maxterm* is a Boolean expression resulting in a **0** for the output of a single cell expression, and **1**s for all other cells in the Karnaugh map, or truth table. The illustration above left shows the maxterm **(A+B+C)**, a single sum term, as a single **0** in a map that is otherwise **1**s.

If a maxterm has a single **0** and the remaining cells as **1**s, it would appear to cover a maximum area of **1**s.

There are some differences now that we are dealing with something new, maxterms. The maxterm is a **0**, not a **1** in the Karnaugh map. A maxterm is a sum term, **(A+B+C)** in our example, not a product term. It also looks strange that **(A+B+C)** is mapped into the cell **000**.

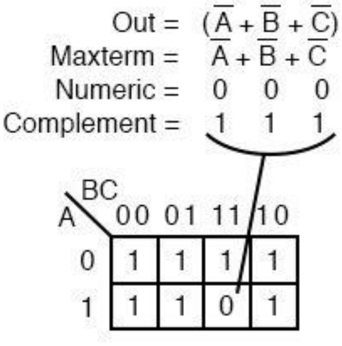
For the equation **Out=(A+B+C)=0**, all three variables **(A, B, C)** must individually be equal to **0**. Only **(0+0+0)=0** will equal **0**. Thus we place our sole **0** for minterm **(A+B+C)** in cell **A,B,C=000** in the K-map, where the inputs are all **0** .

This is the only case which will give us a **0** for our maxterm. All other cells contain **1**s because any input values other than (**(0,0,0)** for **(A+B+C)** yields **1**s upon evaluation.

Referring to the above figure, the procedure for placing a maxterm in the K-map is:

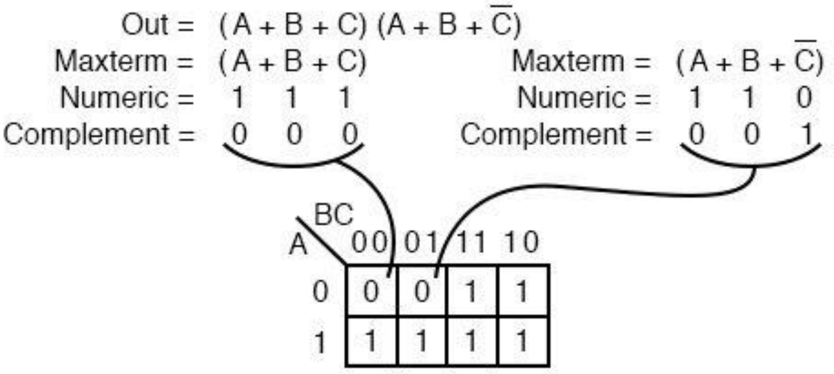
- Identify the Sum term to be mapped.
- Write corresponding binary numeric value.

- Form the complement
- Use the complement as an address to place a **0** in the K-map
- Repeat for other maxterms (Sum terms within Product-of-Sums expression).


Out = $(\overline{A} + \overline{B} + \overline{C})$
Maxterm = $\overline{A} + \overline{B} + \overline{C}$
Numeric = 0 0 0
Complement = 1 1 1
BC
A
00 01 11 10
0 1 1 1 1
1 1 1 0 1

Another maxterm **A’+B’+C’** is shown above. Numeric **000** corresponds to **A’+B’+C’**. The complement is **111**. Place a **0** for maxterm **(A’+B’+C’)** in this cell **(1,1,1)** of the K-map as shown above.

Why should **(A’+B’+C’)** cause a **0** to be in cell **111**? When **A’+B’+C’** is **(1’+1’+1’)**, all **1**s in, which is **(0+0+0)** after taking complements, we have the only condition that will give us a **0**. All the **1**s are complemented to all **0**s, which is **0** when **OR**ed.

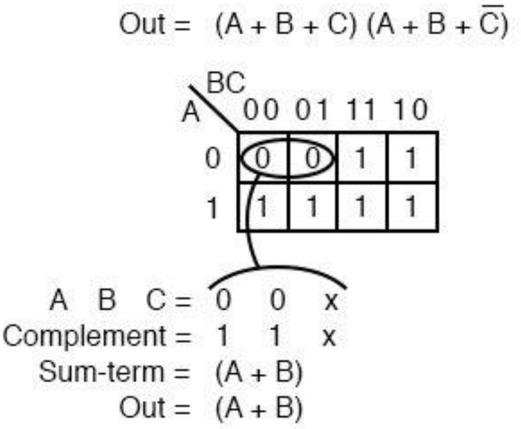


A Boolean Product-Of-Sums expression or map may have multiple maxterms as shown above. Maxterm **(A+B+C)** yields numeric **111** which complements to **000**, placing a **0** in cell **(0,0,0)**. Maxterm **(A+B+C’)** yields numeric **110** which complements to **001**, placing a **0** in cell **(0,0,1)**.

Now that we have the k-map setup, what we are really interested in is showing how to write a Product-Of-Sums reduction. Form the **0**s into groups. That would be a group of two below. Write the binary value corresponding to the sum-term which is **(0,0,X)**.

Both A and B are **0** for the group. But, **C** is both **0** and **1** so we write an **X** as a place holder for **C**. Form the complement **(1,1,X)**. Write the Sum-term **(A+B)** discarding the **C** and the **X** which held its’ place.

In general, expect to have more sum-terms multiplied together in the Product-Of-Sums result. Though, we have a simple example here.



A \ BC	00	01	11	10
0	0	0	1	1
1	1	1	1	1

Let’s summarize the procedure for writing the Product-Of-Sums Boolean reduction for a K-map:

- Form largest groups of 0s possible, covering all maxterms. Groups must be a power of 2.
- Write binary numeric value for group.
- Complement binary numeric value for group.
- Convert complement value to a sum-term.
- Repeat steps for other groups. Each group yields a sum-term within a Product-Of-Sums result.

Examples

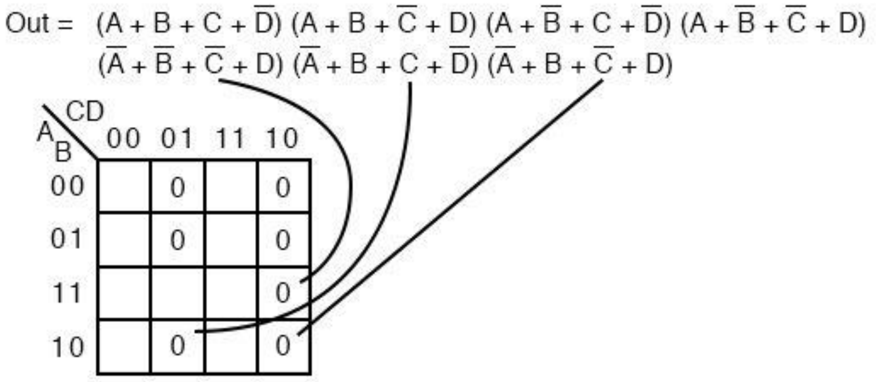
Example:

Simplify the Product-Of-Sums Boolean expression below, providing a result in POS form.

$$\text{Out}= (A+B+C+\overline{D})(A+B+\overline{C}+D)(A+\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D)$$
$$(\overline{A}+\overline{B}+\overline{C}+D)(\overline{A}+B+C+\overline{D})(\overline{A}+B+\overline{C}+D)$$

Solution:

Transfer the seven maxterms to the map below as **0**s. Be sure to complement the input variables in finding the proper cell location.

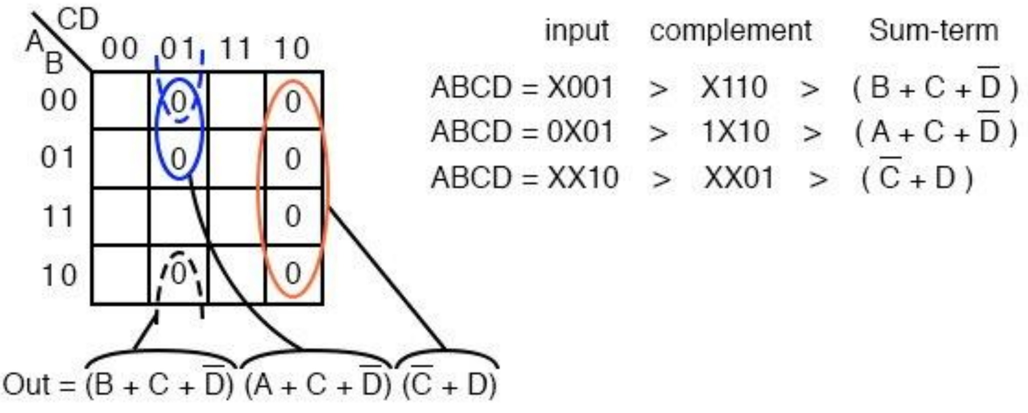


$\text{Out} = (A+B+C+\overline{D})(A+B+\overline{C}+D)(A+\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D)$
$(\overline{A}+\overline{B}+\overline{C}+D)(\overline{A}+B+C+\overline{D})(\overline{A}+B+\overline{C}+D)$

AB \ CD	00	01	11	10
00		0		0
01		0		0
11				0
10		0		0

We map the **0**s as they appear left to right top to bottom on the map above. We locate the last three maxterms with leader lines..

Once the cells are in place above, form groups of cells as shown below. Larger groups will give a sum-term with fewer inputs. Fewer groups will yield fewer sum-terms in the result.



AB \ CD	00	01	11	10
00		0		0
01		0		0
11				0
10		0		0

$\text{Out} = (B+C+\overline{D})(A+C+\overline{D})(\overline{C}+D)$

input		complement		Sum-term
ABCD = X001	>	X110	>	$(B+C+\overline{D})$
ABCD = 0X01	>	1X10	>	$(A+C+\overline{D})$
ABCD = XX10	>	XX01	>	$(\overline{C}+D)$

We have three groups, so we expect to have three sum-terms in our POS result above. The group of 4-cells yields a 2-variable sum-term. The two groups of 2-cells give us two 3-variable sum-terms.

Details are shown for how we arrived at the Sum-terms above. For a group, write the binary group input address, then complement it, converting that to the Boolean sum-term. The final result is product of the three sums.

Example:

Simplify the Product-Of-Sums Boolean expression below, providing a result in SOP form.

$$\text{Out} = (A+B+C+\overline{D})(A+B+\overline{C}+D)(A+\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D)$$
$$(\overline{A}+\overline{B}+\overline{C}+D)(\overline{A}+B+C+\overline{D})(\overline{A}+B+\overline{C}+D)$$

Solution:

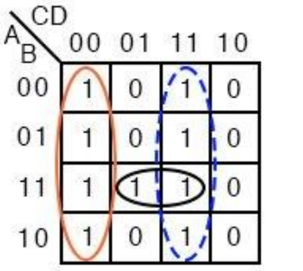
This looks like a repeat of the last problem. It is except that we ask for a Sum-Of-Products Solution instead of the Product-Of-Sums which we just finished. Map the maxterm **0**s from the Product-Of-Sums given as in the previous problem, below left.

$$\text{Out} = (A+B+C+\overline{D})(A+B+\overline{C}+D)(A+\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D)$$
$$(\overline{A}+\overline{B}+\overline{C}+D)(\overline{A}+B+C+\overline{D})(\overline{A}+B+\overline{C}+D)$$

AB \ CD	00	01	11	10
00		0		0
01		0		0
11				0
10		0		0

AB \ CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	1	1	1	0
10	1	0	1	0

Then fill in the implied **1**s in the remaining cells of the map above right.

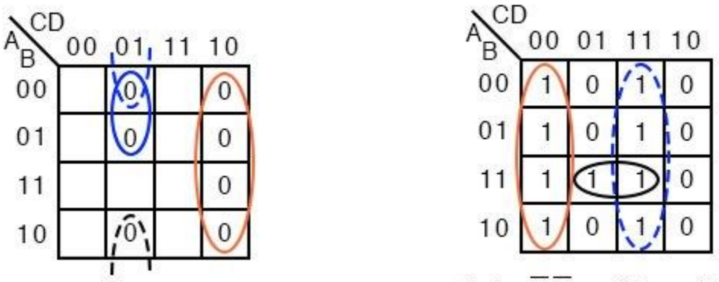


AB \ CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	1	1	1	0
10	1	0	1	0

$$\text{Out} = \overline{C}\,\overline{D} + CD + ABD$$

Form groups of **1**s to cover all **1**s. Then write the Sum-Of-Products simplified result as in the previous section of this chapter. This is identical to a previous problem.

$$\text{Out} = (A + B + C + \overline{D})\,(A + B + \overline{C} + D)\,(A + \overline{B} + C + \overline{D})\,(A + \overline{B} + \overline{C} + D)$$
$$(\overline{A} + \overline{B} + \overline{C} + D)\,(\overline{A} + B + C + \overline{D})\,(\overline{A} + B + \overline{C} + D)$$



AB \ CD	00	01	11	10
00		0		0
01		0		0
11				0
10		0		0

AB \ CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	1	1	1	0
10	1	0	1	0

$\text{Out} = \overline{C}\,\overline{D} + CD + ABD$

$\text{Out} = (B+C+\overline{D})\;(A+C+\overline{D})\;(\overline{C}+D)$

Above we show both the Product-Of-Sums solution, from the previous example, and the Sum-Of-Products solution from the current problem for comparison.

Which is the simpler solution? The POS uses 3-OR gates and 1-AND gate, while the SOP uses 3-AND gates and 1-OR gate. Both use four gates each.

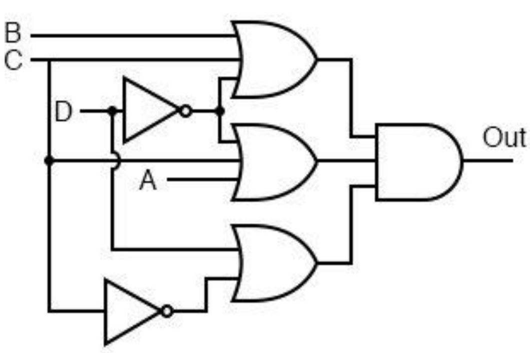
Taking a closer look, we count the number of gate inputs. The POS uses 8-inputs; the SOP uses 7-inputs. By the definition of minimal cost solution, the SOP solution is simpler.

This is an example of a technically correct answer that is of little use in the real world.

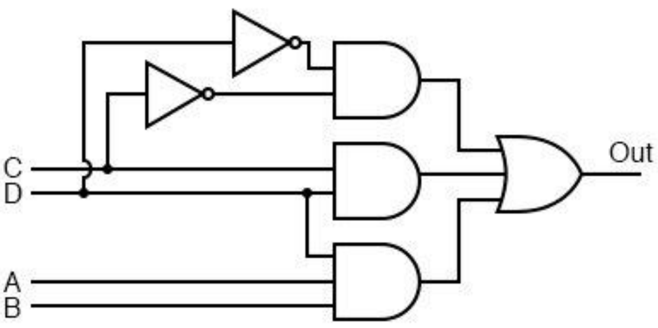
The better solution depends on complexity and the logic family being used. The SOP solution is usually better if using the TTL logic family, as NAND gates are the basic building block, which works well with SOP implementations.

On the other hand, A POS solution would be acceptable when using the CMOS logic family since all sizes of NOR gates are available.

$\text{Out} = (B + C + \overline{D})\,(A + C + \overline{D})\,(\overline{C} + D)$

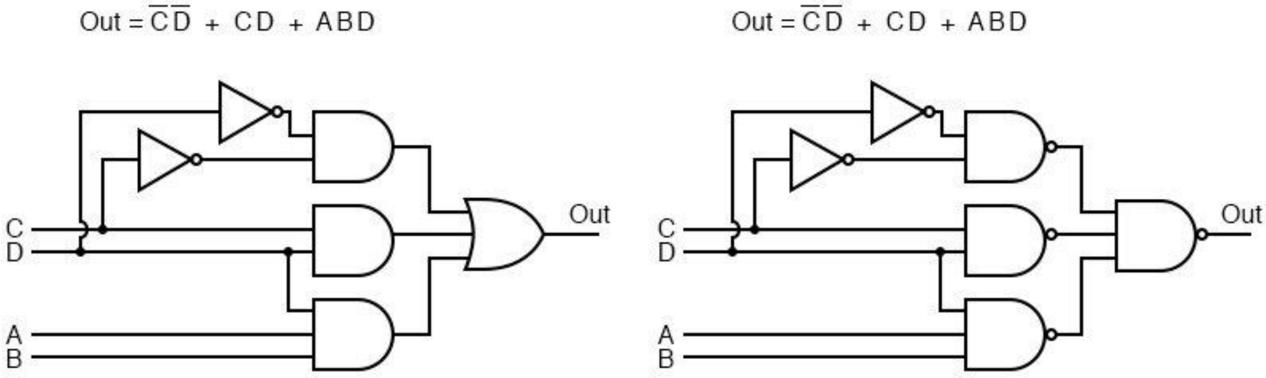


$\text{Out} = \overline{C}\,\overline{D} + CD + ABD$

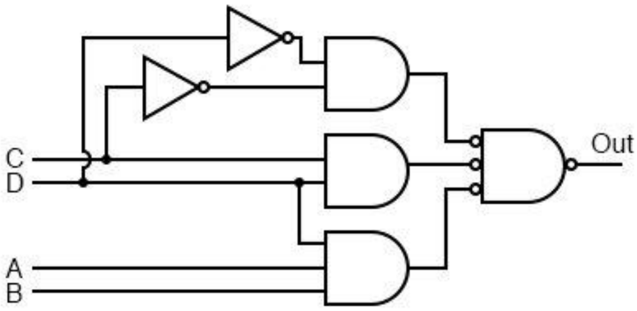


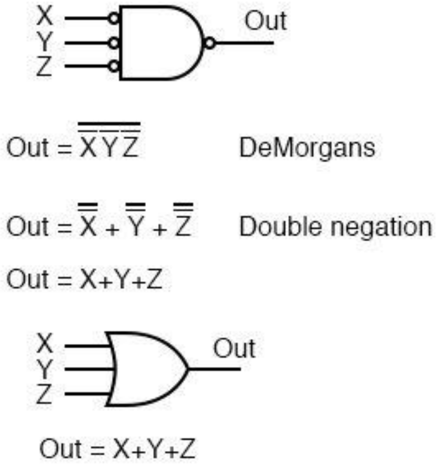
The gate diagrams for both cases are shown above, Product-Of-Sums left, and Sum-Of-Products right.

Below, we take a closer look at the Sum-Of-Products version of our example logic, which is repeated at left.


Out = $\overline{C}\,\overline{D}$ + CD + ABD
C
D
A
B
Out
Out = $\overline{C}\,\overline{D}$ + CD + ABD
C
D
A
B
Out

Above all AND gates at left have been replaced by NAND gates at right.. The OR gate at the output is replaced by a NAND gate. To prove that AND-OR logic is equivalent to NAND-NAND logic, move the inverter invert bubbles at the output of the 3-NAND gates to the input of the final NAND as shown in going from above right to below left.




X
Y
Z
Out
Out = $\overline{\overline{X}\,\overline{Y}\,\overline{Z}}$
DeMorgans
Out = $\overline{\overline{X}} + \overline{\overline{Y}} + \overline{\overline{Z}}$
Double negation
Out = X+Y+Z
X
Y
Z
Out
Out = X+Y+Z

Above right we see that the output NAND gate with inverted inputs is logically equivalent to an OR gate by DeMorgan’s theorem and double negation.

This information is useful in building digital logic in a laboratory setting where TTL logic family NAND gates are more readily available in a wide variety of configurations than other types.

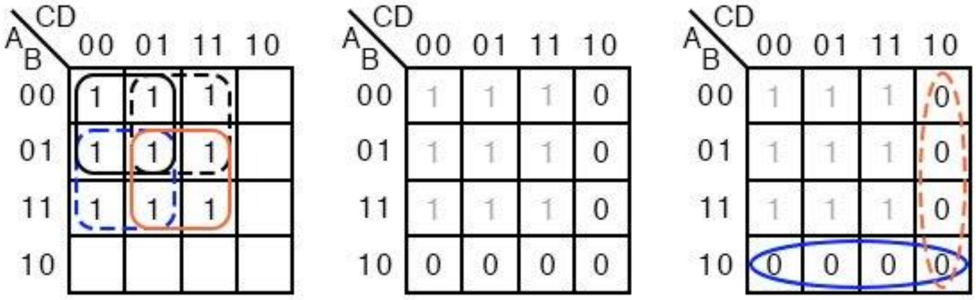
The Procedure for constructing NAND-NAND logic, in place of AND-OR logic is as follows:

- Produce a reduced Sum-Of-Products logic design.
- When drawing the wiring diagram of the SOP, replace all gates (both AND and OR) with NAND gates.
- Unused inputs should be tied to logic High.
- In case of troubleshooting, internal nodes at the first level of NAND gate outputs do NOT match AND-OR diagram logic levels, but are inverted. Use the NAND-NAND logic diagram. Inputs and final output are identical, though.
- Label any multiple packages U1, U2,.. etc.
- Use data sheet to assign pin numbers to inputs and outputs of all gates.

Example:

Let us revisit a previous problem involving an SOP minimization. Produce a Product-Of-Sums solution. Compare the POS solution to the previous SOP.

$$\begin{aligned}\text{Out} = {} & \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD \\ & + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD \\ & + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD\end{aligned}$$



AB \ CD	00	01	11	10
00	1	1	1	
01	1	1	1	
11	1	1	1	
10				

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	0	0	0

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	0	0	0

$$\text{Out} = \overline{A}\,\overline{C} + \overline{A}D + B\overline{C} + BD$$

$$\text{Out} = (\overline{A}+B)\ (\overline{C}+D)$$

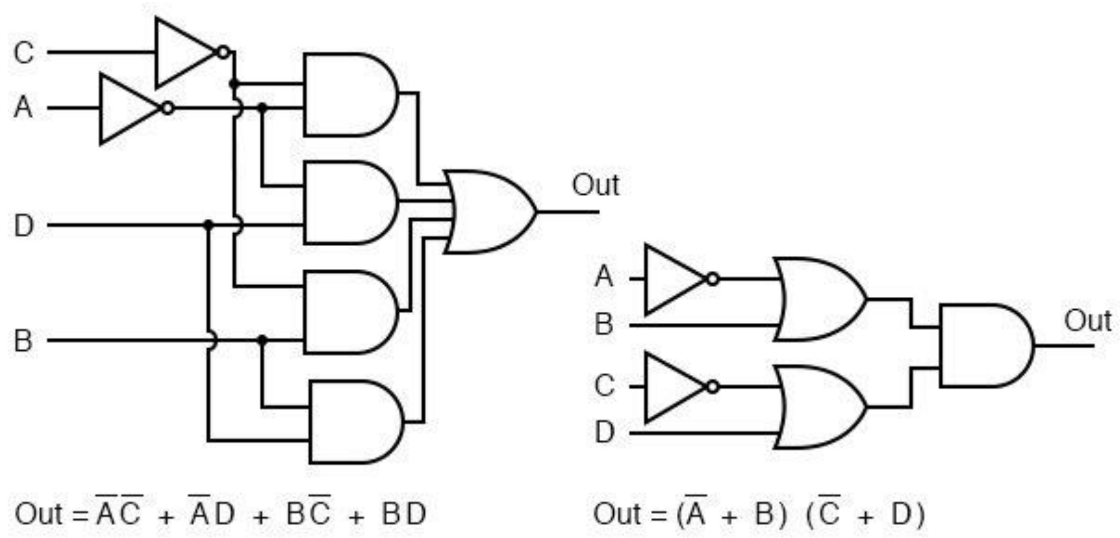
Solution:

Above left we have the original problem starting with a 9-minterm Boolean unsimplified expression. Reviewing, we formed four groups of 4-cells to yield a 4-product-term SOP result, lower left.

In the middle figure, above, we fill in the empty spaces with the implied **0**s. The **0**s form two groups of 4-cells. The solid blue group is **(A’+B)**, the dashed red group is **(C’+D)**. This yields two sum-terms in the Product-Of-Sums result, above right **Out = (A’+B)(C’+D)**

Comparing the previous SOP simplification, left, to the POS simplification, right, shows that the POS is the least cost solution. The SOP uses 5-gates total, the POS uses only 3-gates.

This POS solution even looks attractive when using TTL logic due to simplicity of the result. We can find AND gates and an OR gate with 2-inputs.

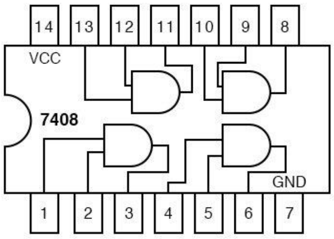

C
A
D
B
Out
A
B
C
D
Out
Out = $\overline{A}\,\overline{C} + \overline{A}D + B\overline{C} + BD$
Out = $(\overline{A} + B)(\overline{C} + D)$

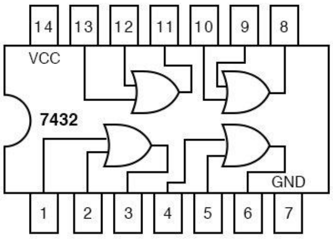
The SOP and POS gate diagrams are shown above for our comparison problem.

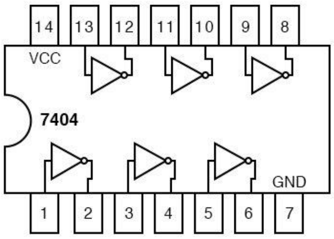
Given the pin-outs for the TTL logic family integrated circuit gates below, label the maxterm diagram above right with Circuit designators (U1-a, U1-b, U2-a, etc), and pin numbers.

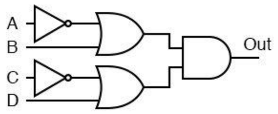
Each integrated circuit package that we use will receive a circuit designator: U1, U2, U3. To distinguish between the individual gates within the package, they are identified as a, b, c, d, etc.

The 7404 hex-inverter package is U1. The individual inverters in it are are U1-a, U1-b, U1-c, etc. U2 is assigned to the 7432 quad OR gate. U3 is assigned to the 7408 quad AND gate.







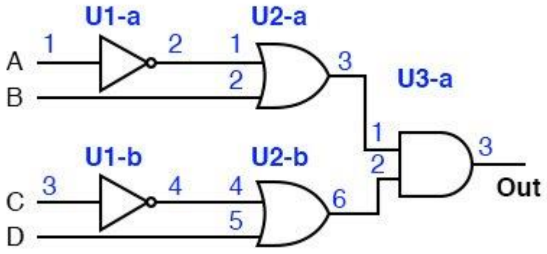


Out = $(\overline{A} + B)(\overline{C} + D)$

With reference to the pin numbers on the package diagram above, we assign pin numbers to all gate inputs and outputs on the schematic diagram below.

We can now build this circuit in a laboratory setting. Or, we could design a printed circuit board for it. A printed circuit board contains copper foil “wiring” backed by a non conductive substrate of phenolic, or epoxy-fiberglass.

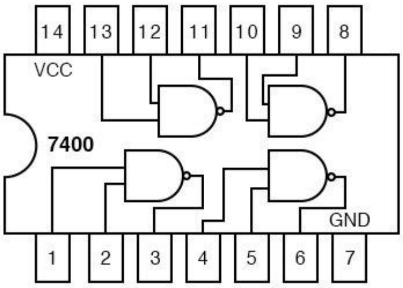
Printed circuit boards are used to mass produce electronic circuits. Ground the inputs of unused gates.

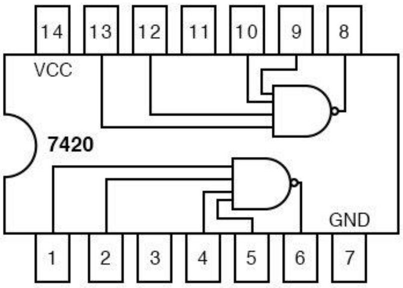


Out = $(\overline{A} + B)(\overline{C} + D)$

UI = 7404
U2 = 7432
U3 = 7408

Label the previous POS solution diagram above left (third figure back) with Circuit designators and pin numbers. This will be similar to what we just did.





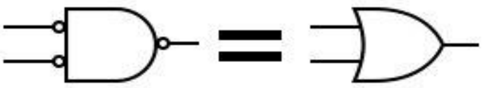
We can find 2-input AND gates, 7408 in the previous example. However, we have trouble finding a 4-input OR gate in our TTL catalog.

The only kind of gate with 4-inputs is the 7420 NAND gate shown above right.

We can make the 4-input NAND gate into a 4-input OR gate by inverting the inputs to the NAND gate as shown below. So we will use the 7420 4-input NAND gate as an OR gate by inverting the inputs.

$$\overline{Y} = \overline{A}\,\overline{B} = \overline{A + B}$$
$$Y = A + B$$

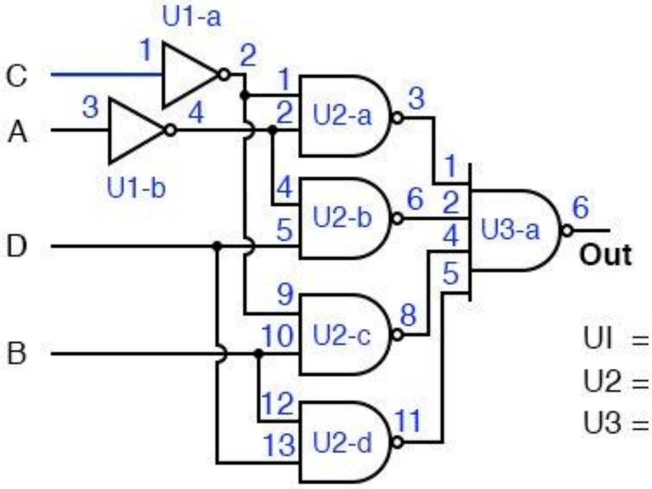
DeMorgan's
Double negation



We will not use discrete inverters to invert the inputs to the 7420 4-input NAND gate, but will drive it with 2-input NAND gates in place of the AND gates called for in the SOP, minterm, solution.

The inversion at the output of the 2-input NAND gates supply the inversion for the 4-input OR gate.

The result is shown below. It is the only practical way to actually build it with TTL gates by using NAND-NAND logic replacing AND-OR logic.



UI = 7404
U2 = 7400
U3 = 7420

$$Out = \overline{(\overline{\overline{A}\,\overline{C}})\,(\overline{\overline{A}\,D})\,(\overline{B\,\overline{C}})\,(\overline{B\,D})}$$ **Boolean from diagram**

$$Out = \overline{\overline{\overline{A}\,\overline{C}}} + \overline{\overline{\overline{A}\,D}} + \overline{\overline{B\,\overline{C}}} + \overline{\overline{B\,D}}$$ **DeMorgan's**

$$Out = \overline{A}\,\overline{C} + \overline{A}\,D + B\,\overline{C} + B\,D$$ **Double negation**

Don’t Care Cells in the Karnaugh Map

Up to this point we have considered logic reduction problems where the input conditions were completely specified. That is, a 3-variable truth table or Karnaugh map had 2n = 23 or 8-entries, a full table or map.

It is not always necessary to fill in the complete truth table for some real-world problems. We may have a choice to not fill in the complete table.

For example, when dealing with BCD (Binary Coded Decimal) numbers encoded as four bits, we may not care about any codes above the BCD range of (0, 1, 2…9). The 4-bit binary codes for the hexadecimal numbers (Ah, Bh, Ch, Eh, Fh) are not valid BCD codes.

Thus, we do not have to fill in those codes at the end of a truth table, or K-map, if we do not care to.

We would not normally care to fill in those codes because those codes (1010, 1011, 1100, 1101, 1110, 1111) will never exist as long as we are dealing only with BCD encoded numbers. These six invalid codes are don’t cares as far as we are concerned.

That is, we do not care what output our logic circuit produces for these don’t cares.

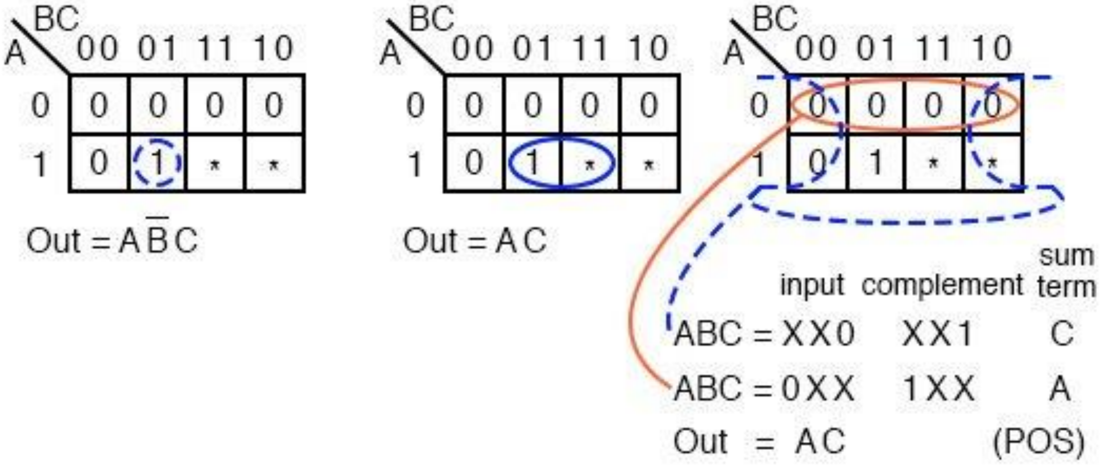
Don’t Cares

Don’t cares in a Karnaugh map, or truth table, may be either 1s or 0s, as long as we don’t care what the output is for an input condition we never expect to see. We plot these cells with an asterisk, *, among the normal 1s and 0s.

When forming groups of cells, treat the don’t care cell as either a 1 or a 0, or ignore the don’t cares.

This is helpful if it allows us to form a larger group than would otherwise be possible without the don’t cares. There is no requirement to group all or any of the don’t cares.

Only use them in a group if it simplifies the logic.

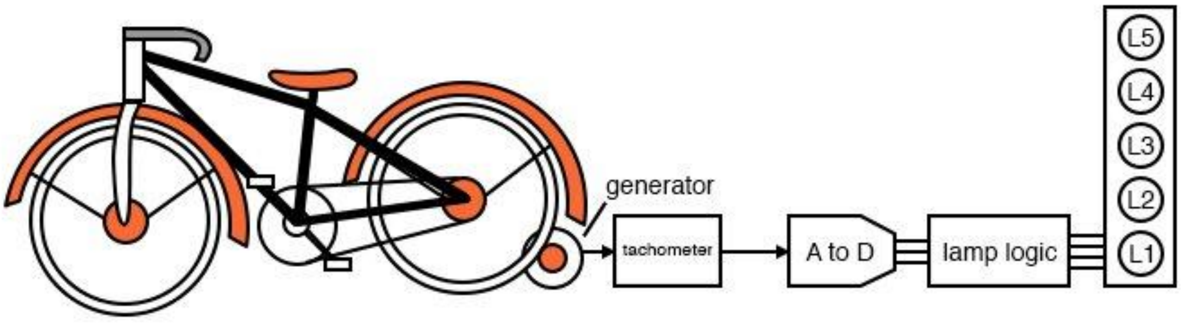


Above is an example of a logic function where the desired output is **1** for input **ABC = 101** over the range from **000 to 101**. We do not care what the output is for the other possible inputs (**110, 111)**. Map those two as don't cares. We show two solutions.

The solution on the right Out = AB'C is the more complex solution since we did not use the don't care cells. The solution in the middle, Out=AC, is less complex because we grouped a don't care cell with the single **1** to form a group of two.

The third solution, a Product-Of-Sums on the right, results from grouping a don't care with three zeros forming a group of four 0s. This is the same, less complex, **Out=AC**.

We have illustrated that the don't care cells may be used as either **1**s or **0**s, whichever is useful.



Let has been asked to build the lamp logic for a stationary bicycle exhibit at the local science museum. As a rider increases his pedaling speed, lamps will light on a bar graph display.

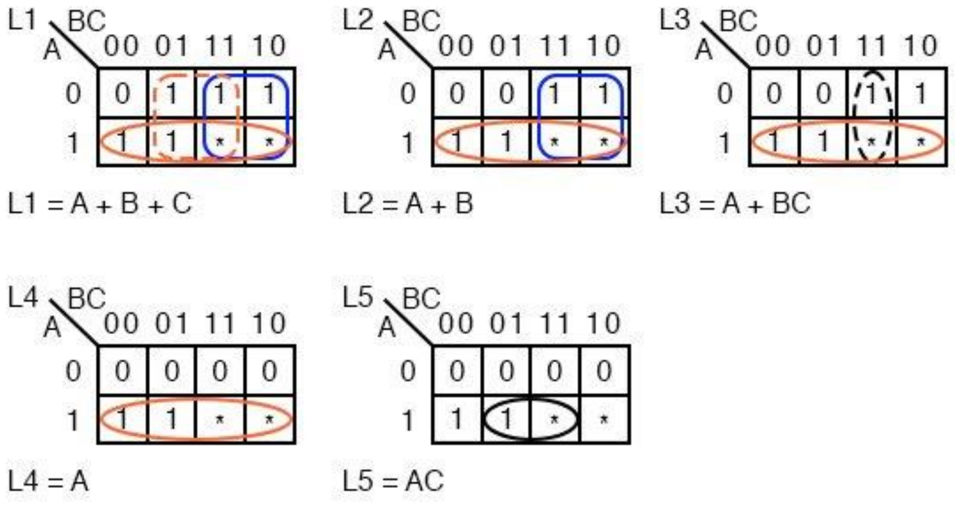
No lamps will light for no motion. As speed increases, the lower lamp, L1 lights, then L1 and L2, then, L1, L2, and L3, until all lamps light at the highest speed. Once all the lamps illuminate, no further increase in speed will have any effect on the display.

A small DC generator coupled to the bicycle tire outputs a voltage proportional to speed. It drives a tachometer board which limits the voltage at the high end of speed where all lamps light. No further increase in speed can increase the voltage beyond this level.

This is crucial because the downstream A to D (Analog to Digital) converter puts out a 3-bit code, **ABC**, 2^3 or 8-codes, but we only have five lamps. **A** is the most significant bit, **C** the least significant bit.

The lamp logic needs to respond to the six codes out of the A to D. For **ABC=000**, no motion, no lamps light. For the five codes **(001 to 101)** lamps L1, L1&L2, L1&L2&L3, up to all lamps will light, as speed, voltage, and the A to D code (ABC) increases.

We do not care about the response to input codes **(110, 111)** because these codes will never come out of the A to D due to the limiting in the tachometer block. We need to design five logic circuits to drive the five lamps.



L1 = A + B + C

L2 = A + B

L3 = A + BC

L4 = A

L5 = AC

Since, none of the lamps light for **ABC=000** out of the A to D, enter a **0** in all K-maps for cell **ABC=000**. Since we don't care about the never to be encountered codes **(110, 111)**, enter asterisks into those two cells in all five K-maps. Instead of asterisks, you can put a minus (-).

Lamp L5 will only light for code **ABC=101**. Enter a **1** in that cell and five **0**s into the remaining empty cells of L5 K-map.

L4 will light initially for code **ABC=100**, and will remain illuminated for any code greater, **ABC=101**, because all lamps below L5 will light when L5 lights. Enter **1**s into cells **100** and **101** of the L4 map so that it will light for those codes. Four **0**'s fill the remaining L4 cells

L3 will initially light for code **ABC=011**. It will also light whenever L5 and L4 illuminate. Enter three **1**s into cells **011, 100, 101** for L3 map. Fill three **0**s into the remaining L3 cells.

L2 lights for **ABC=010** and codes greater. Fill **1**s into cells **010, 011, 100, 101**, and two **0**s in the remaining cells.

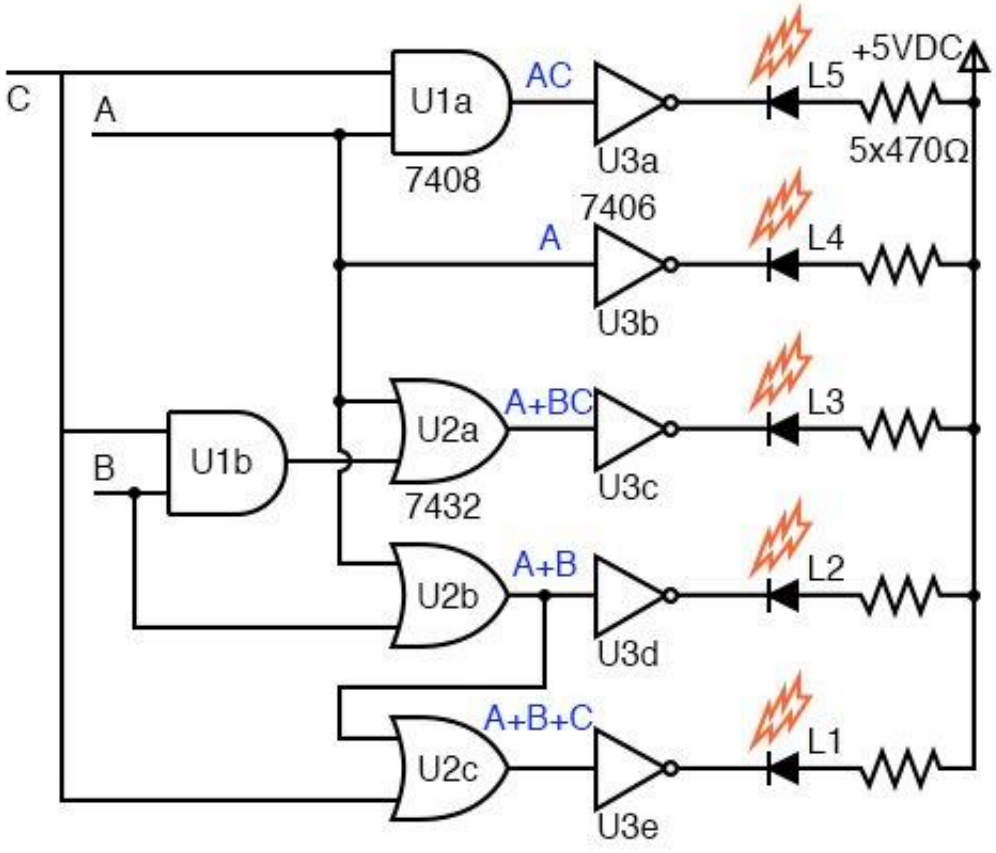
The only time L1 is not lighted is for no motion. There is already a **0** in cell **ABC=000**. All the other five cells receive **1**s.

Group the **1**'s as shown above, using don't cares whenever a larger group results. The L1 map shows three product terms, corresponding to three groups of 4-cells.

We used both don't cares in two of the groups and one don't care on the third group. The don't cares allowed us to form groups of four.

In a similar manner, the L2 and L4 maps both produce groups of 4-cells with the aid of the don't care cells. The L4 reduction is striking in that the L4 lamp is controlled by the most significant bit from the A to D converter, **L5=A**.

No logic gates are required for lamp L4. In the L3 and L5 maps, single cells form groups of two with don't care cells. In all five maps, the reduced Boolean equation is less complex than without the don't cares.



The gate diagram for the circuit is above. The outputs of the five K-map equations drive inverters. Note that the L1 **OR** gate is not a 3-input gate but a 2-input gate having inputs **(A+B), C**, outputting **A+B+C** The *open collector* inverters, **7406**, are desirable for driving LEDs, though, not part of the K-map logic design.

The output of an open collector gate or inverter is open-circuited at the collector internal to the integrated circuit package so that all collector current may flow through an external load. An active high into any of the inverters pulls the output low, drawing current through the LED and the current limiting resistor.

The LEDs would likely be part of a solid-state relay driving 220 V AC lamps for a museum exhibit, not shown here.

Decoder

A decoder is a circuit that changes a code into a set of signals. It is called a decoder because it does the reverse of encoding, but we will begin our study of encoders and decoders with decoders because they are simpler to design.

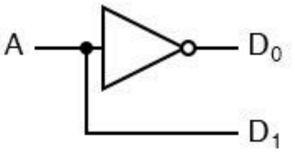
Types of Decoders

Line Decoder

A common type of decoder is the line decoder which takes an n-digit binary number and decodes it into 2^n data lines. The simplest is the 1-to-2 line decoder. The truth table is:

A	D_1	D_0
0	0	1
1	1	0

A is the address and D is the dataline. D0 is NOT A and D1 is A. The circuit looks like the Figures below.

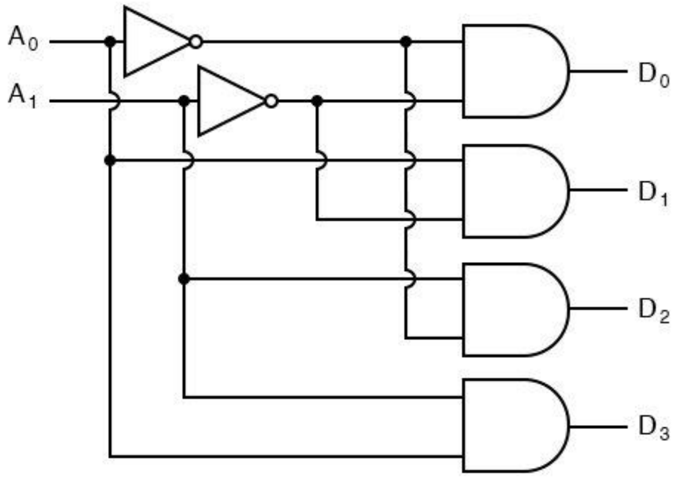


2-to-4 Line Decoder

Only slightly more complex is the 2-to-4 line decoder. The truth table is:

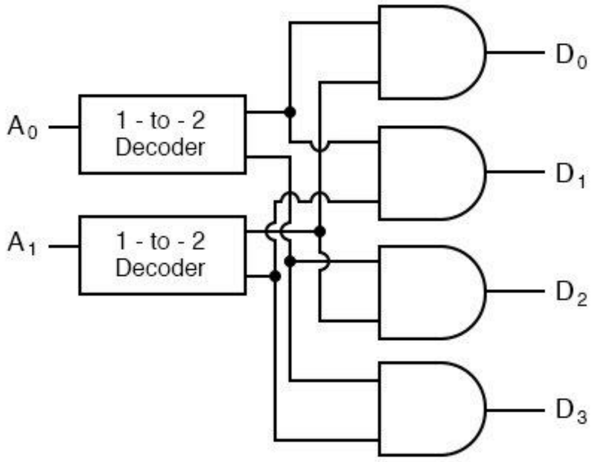
A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Developed into a circuit it looks like the Figure below.

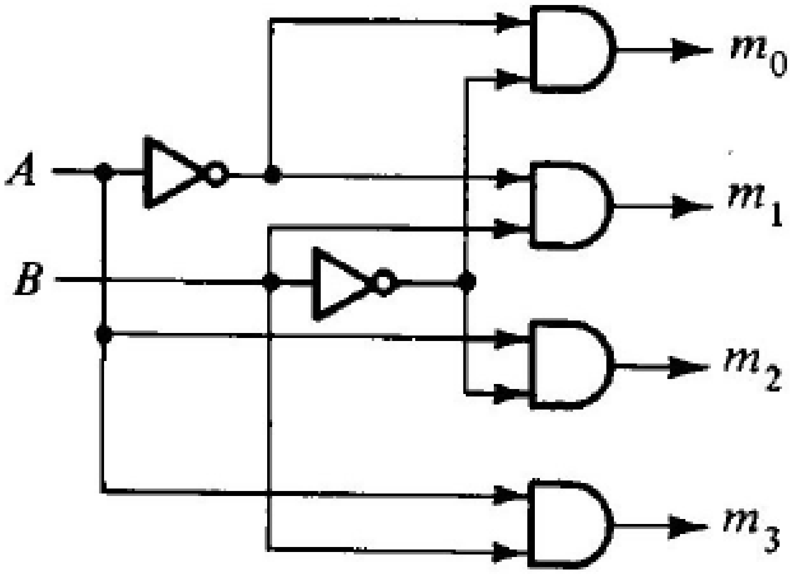


Larger Line Decoders

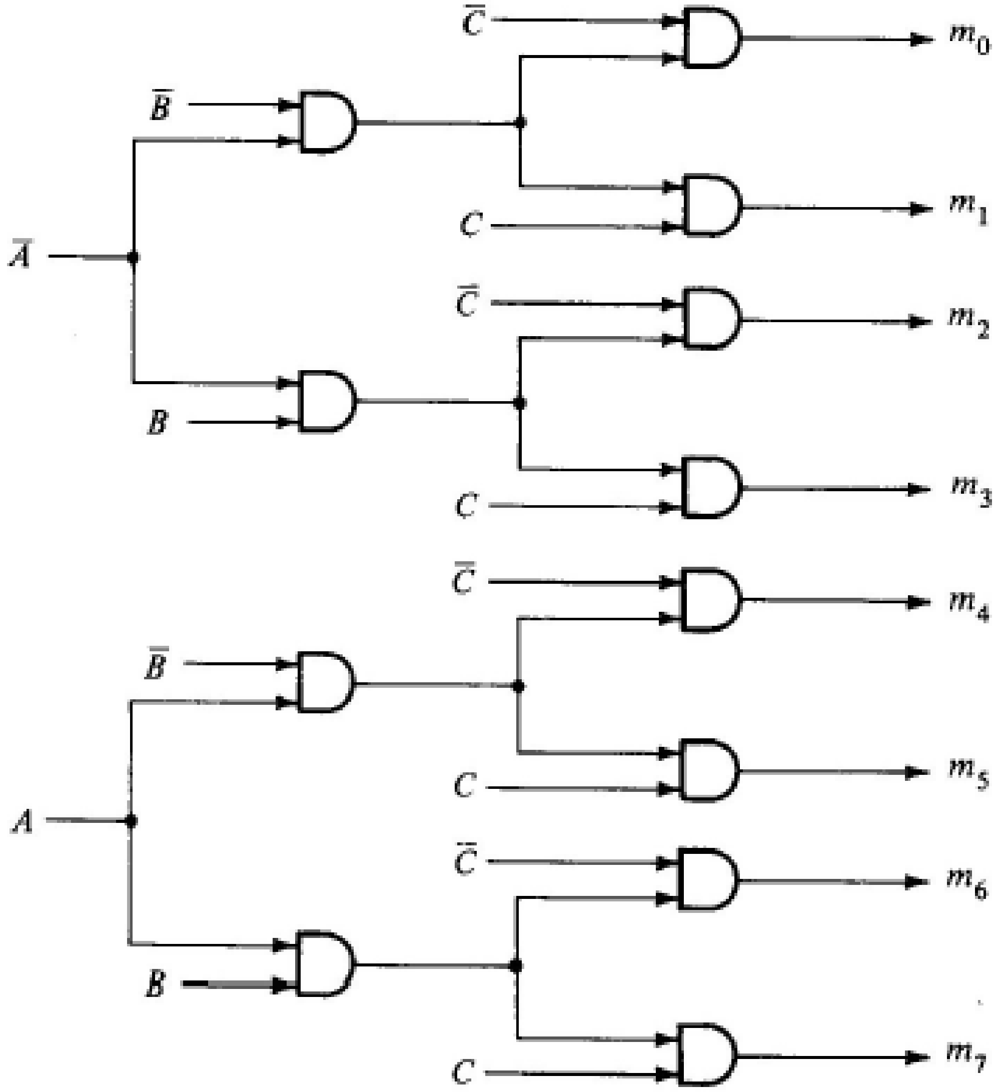
Larger line decoders can be designed in a similar fashion, but just like with the binary adder there is a way to make larger decoders by combining smaller decoders. An alternate circuit for the 2-to-4 line decoder is:



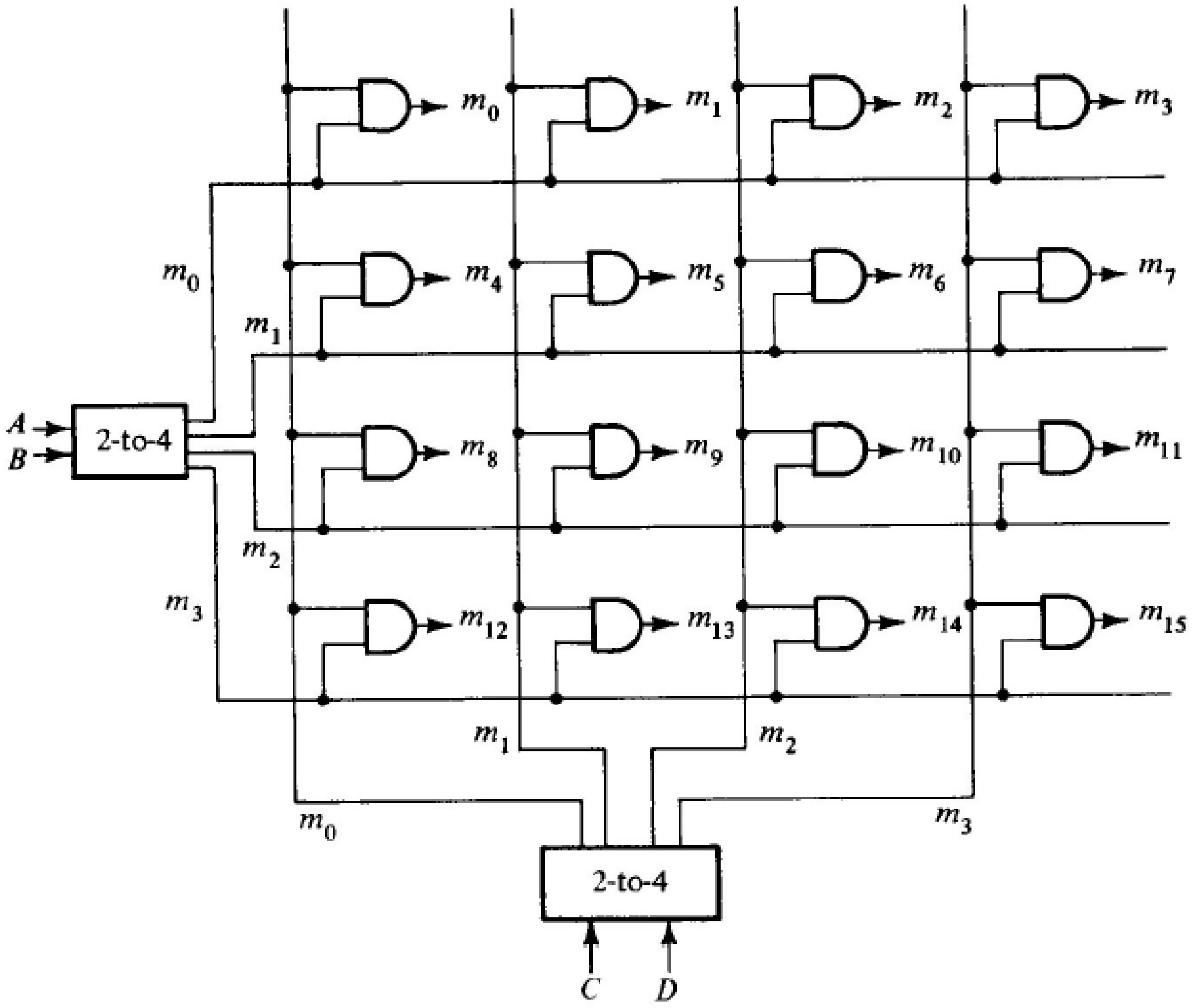
Outputs are minterms.



Replacing the 1-to-2 Decoders with their circuits will show that both circuits are equivalent. In a similar fashion a 3-to-8 line decoder can be made from a 1-to-2 line decoder and a 2-to-4 line decoder. It is named tree structure decoder.



A 4-to-16 line decoder can be made from two 2-to-4 line decoders. It is named dual-tree structure decoder



A typical application of a line decoder circuit is to select among multiple devices. A circuit needing to select among sixteen devices could have sixteen control lines to select which device should “listen”. With a decoder only four control lines are needed.

Encoder

An encoder is a circuit that changes a set of signals into a code. Let’s begin making a 2-to-1 line encoder truth table by reversing the 1-to-2 decoder truth table.

D_1	D_0	A
0	1	0
1	0	1

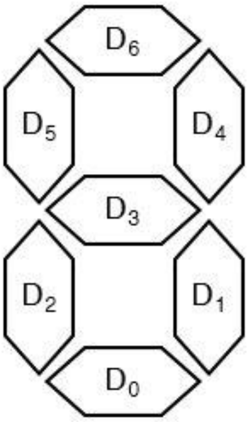
This truth table is a little short. A complete truth table would be:

D_1	D_0	A
0	0	-
0	1	0
1	0	1
1	1	-

One question we need to answer is what to do with those other inputs? Do we ignore them? Do we have them generate an additional error output? In many circuits, this problem is solved by adding sequential logic in order to know not just what input is active but also which order the inputs became active.

Encoder Design Applications

A more useful application of combinational encoder design is a binary to 7-segment encoder. The seven segments are given according to:

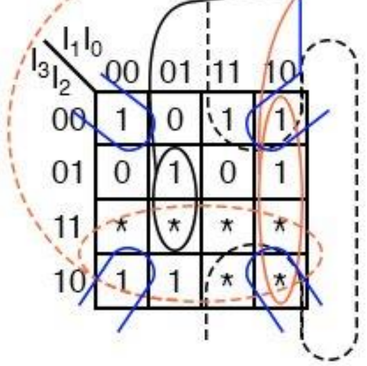


Truth table is:

I_3	I_2	I_1	I_0	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0	1
0	0	1	1	1	0	1	1	0	1	1
0	1	0	0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	1	0	1	1
0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	1	0	1	0	0	1	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

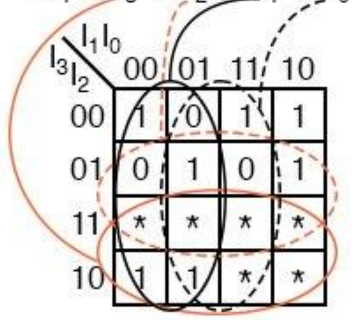
Deciding what to do with the remaining six entries of the truth table is easier with this circuit. This circuit should not be expected to encode an undefined combination of inputs, so we can leave them as “don’t care” when we design the circuit. The equations were simplified with Karnaugh maps.

$D_0 = I_3 + \overline{I}_2 I_1 + \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0 + I_2 \overline{I}_1 I_0$



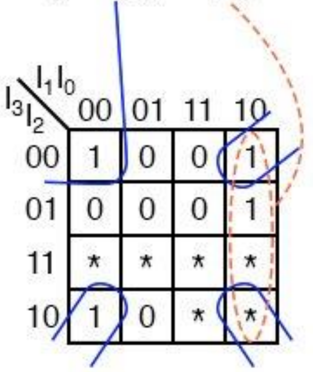
$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	*	*	*	*
10	1	1	*	*

$D_1 = I_3 + I_2 + \overline{I}_1 + I_0$



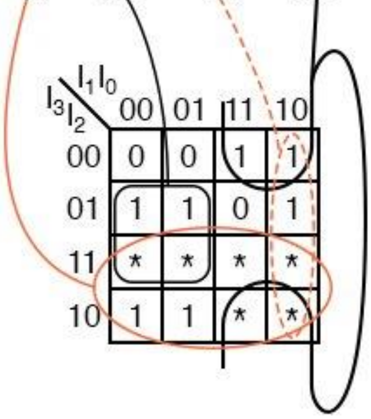
$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	*	*	*	*
10	1	1	*	*

$D_2 = \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0$



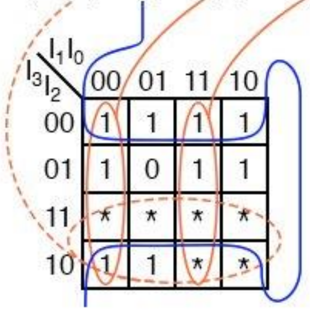
$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	*	*	*	*
10	1	0	*	*

$D_3 = I_3 + I_2 \overline{I}_1 + I_1 \overline{I}_0 + \overline{I}_2 I_1$



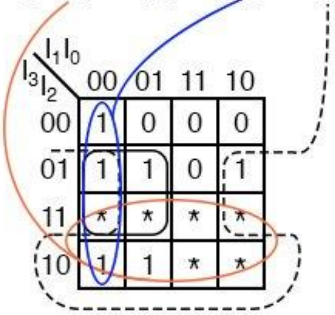
$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	*	*	*	*
10	1	1	*	*

$D_4 = I_3 + \overline{I}_2 + \overline{I}_1 \overline{I}_0 + I_1 I_0$



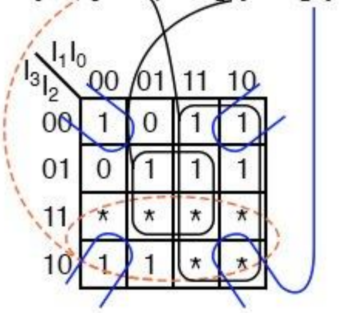
$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	1	1	1
01	1	0	1	1
11	*	*	*	*
10	1	1	*	*

$D_5 = I_3 + I_2 \overline{I}_1 + \overline{I}_1 \overline{I}_0 + I_2 \overline{I}_0$



$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	*	*	*	*
10	1	1	*	*

$D_6 = I_3 + I_1 + I_2 I_0 + \overline{I}_2 \overline{I}_0$



$I_3I_2 \backslash I_1I_0$	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	*	*	*	*
10	1	1	*	*

Equation Collection Summary

The collection of equations is summarized here:

$$D_0 = I_3 + \overline{I}_2 I_1 + \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0 + I_2 \overline{I}_1 I_0$$

$$D_1 = I_3 + I_2 + \overline{I}_1 + I_0$$

$$D_2 = \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0$$

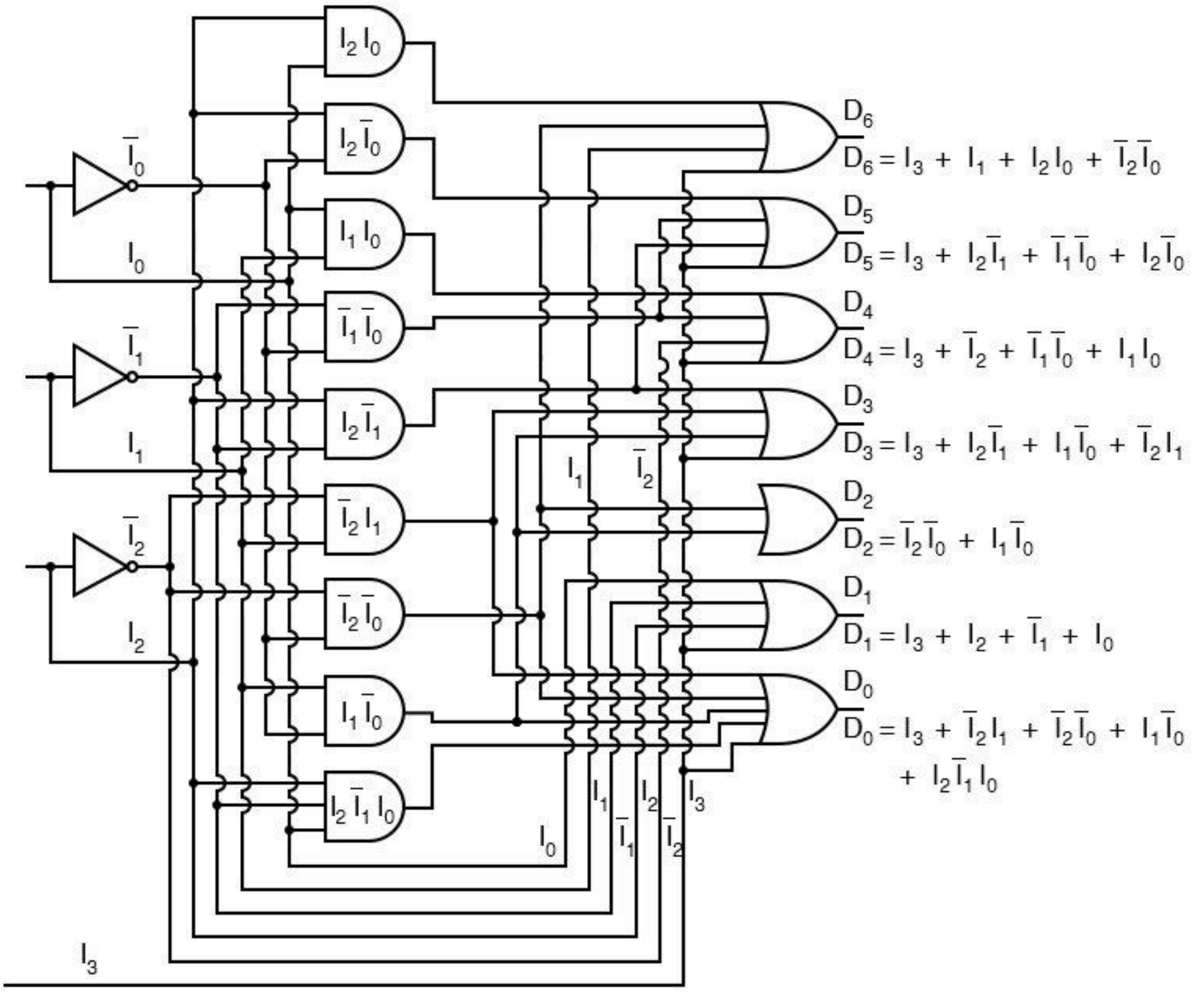
$$D_3 = I_3 + I_2 \overline{I}_1 + I_1 \overline{I}_0 + \overline{I}_2 I_1$$

$$D_4 = I_3 + \overline{I}_2 + \overline{I}_1 \overline{I}_0 + I_1 I_0$$

$$D_5 = I_3 + I_2 \overline{I}_1 + \overline{I}_1 \overline{I}_0 + I_2 I_0$$

$$D_6 = I_3 + I_1 + I_2 I_0 + \overline{I}_2 \overline{I}_0$$

The circuit is:

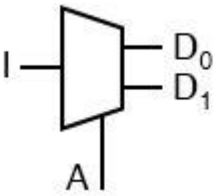


$I_2 I_0$
$I_2 \overline{I}_0$
$I_1 I_0$
$\overline{I}_1 \overline{I}_0$
$I_2 \overline{I}_1$
$\overline{I}_2 I_1$
$\overline{I}_2 \overline{I}_0$
$I_1 \overline{I}_0$
$I_2 \overline{I}_1 I_0$
$\overline{I}_0$
I_0
$\overline{I}_1$
I_1
$\overline{I}_2$
I_2
I_3
I_1
$\overline{I}_2$
I_0
I_1
$\overline{I}_1$
I_2
$\overline{I}_2$
I_3
D_6
$D_6 = I_3 + I_1 + I_2 I_0 + \overline{I}_2 \overline{I}_0$
D_5
$D_5 = I_3 + I_2 \overline{I}_1 + \overline{I}_1 \overline{I}_0 + I_2 \overline{I}_0$
D_4
$D_4 = I_3 + \overline{I}_2 + \overline{I}_1 \overline{I}_0 + I_1 I_0$
D_3
$D_3 = I_3 + I_2 \overline{I}_1 + I_1 \overline{I}_0 + \overline{I}_2 I_1$
D_2
$D_2 = \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0$
D_1
$D_1 = I_3 + I_2 + \overline{I}_1 + I_0$
D_0
$D_0 = I_3 + \overline{I}_2 I_1 + \overline{I}_2 \overline{I}_0 + I_1 \overline{I}_0 + I_2 \overline{I}_1 I_0$

Demultiplexers

A demultiplexer, sometimes abbreviated dmux, is a circuit that has one input and more than one output. It is used when a circuit intends to send a signal to one of many devices. This description sounds similar to the description given for a decoder, but a decoder is used *to* ***select*** among many devices while a demultiplexer is used *to* ***send a signal*** among many devices.

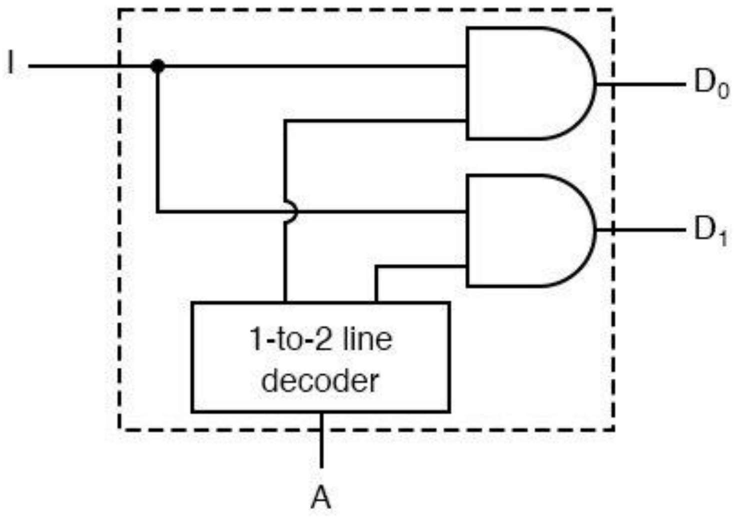
A demultiplexer is used often enough that it has its own schematic symbol (Figure below)



The truth table for a 1-to-2 demultiplexer is:

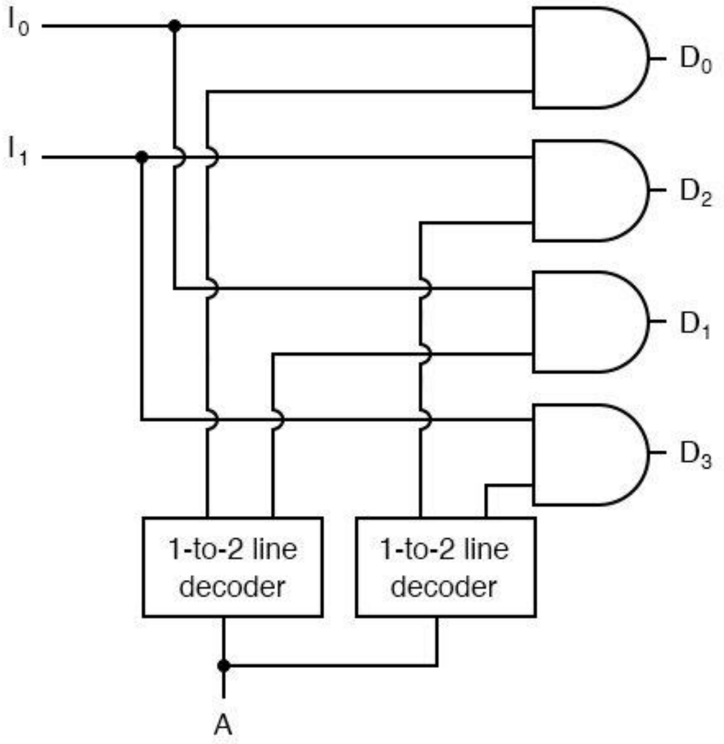
I	A	D_0	D_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Using our 1-to-2 decoder as part of the circuit, we can express this circuit easily as:

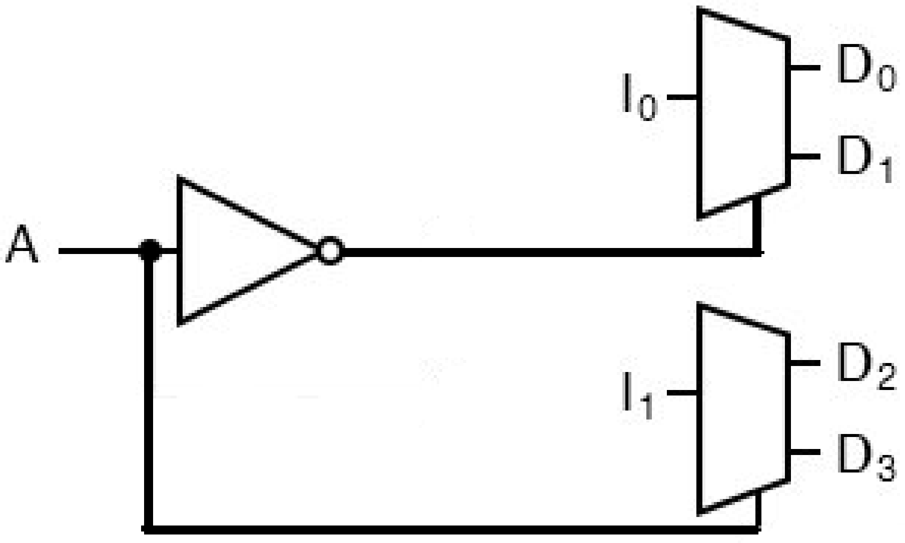


This circuit can be expanded into two different ways. You can increase the number of signals that get transmitted, or you can increase the number of inputs that get passed through. To increase the number of inputs that get passed through just requires a larger line decoder. Increasing the number of signals that get transmitted is even easier.

As an example, a device that passes one set of two signals among four signals is a “two-bit 1-to-2 demultiplexer”. Its circuit is:

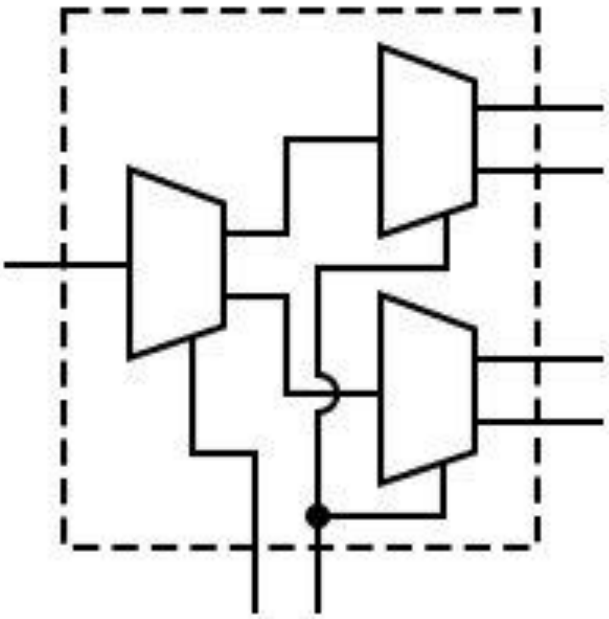


or by expressing the circuit as,



shows that it could be two one-bit 1-to-2 demultiplexers without changing its expected behavior.

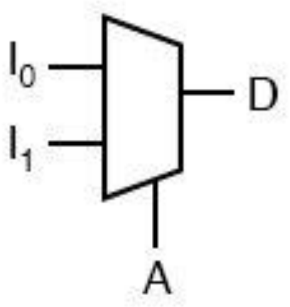
A 1-to-4 demultiplexer can easily be built from 1-to-2 demultiplexers as follows.



Multiplexers

A multiplexer, abbreviated MUX or MS, is a device that has multiple inputs and one output.

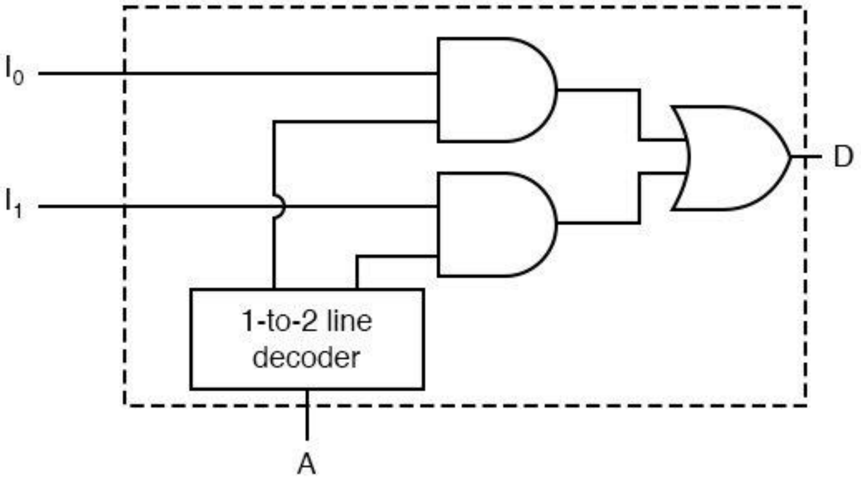
The schematic symbol for multiplexers is



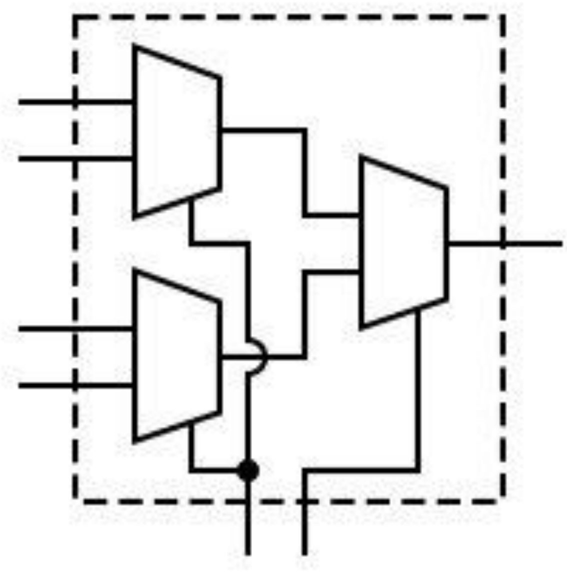
The truth table for a 2-to-1 multiplexer is

I_1	I_0	A	D
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

Using a 1-to-2 decoder as part of the circuit, we can express this circuit easily.



Multiplexers can also be expanded as demultiplexers. A 4-to-1 multiplexer circuit is

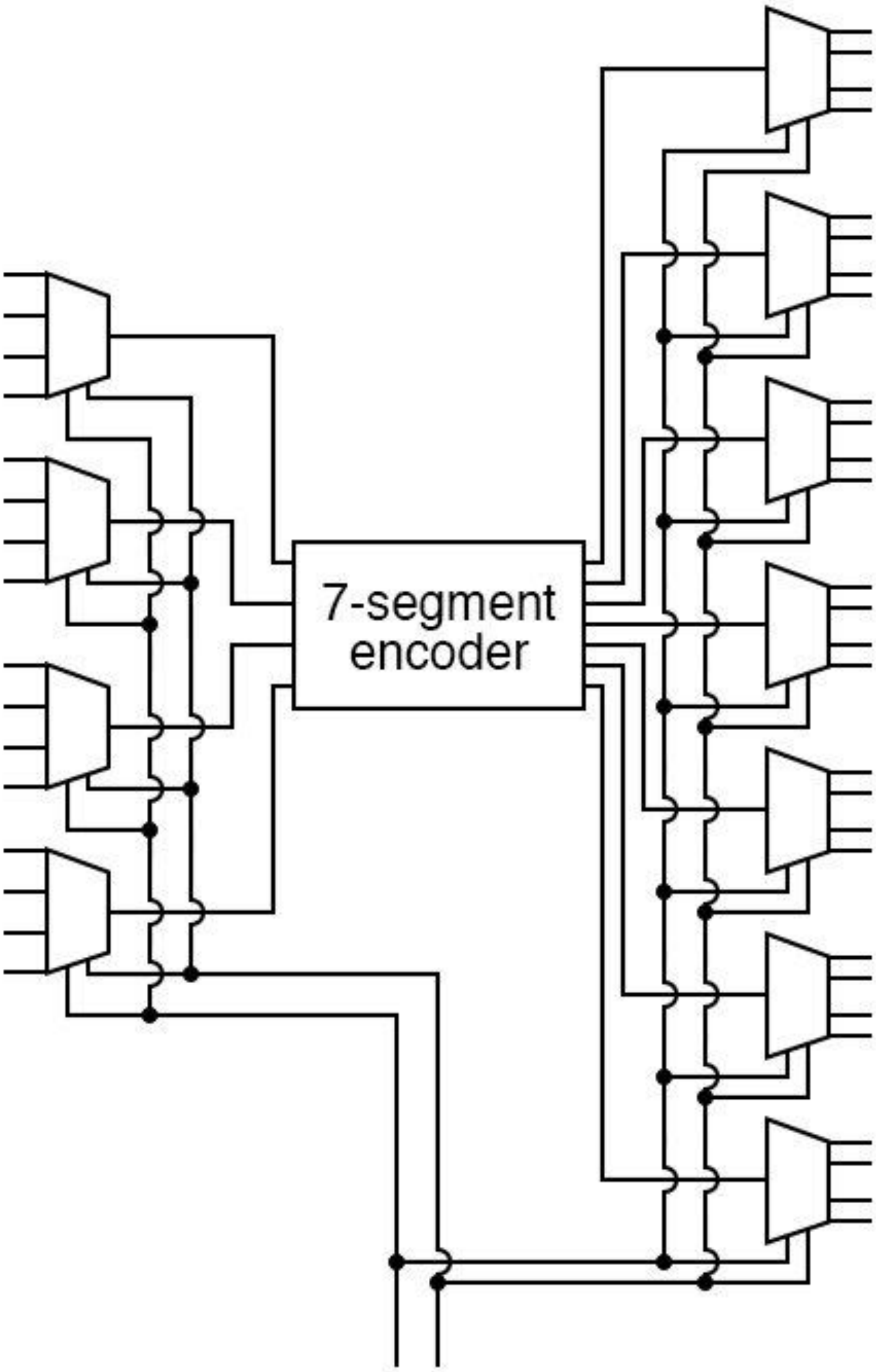


That is the formal definition of a multiplexer. Informally, there are a lot of confusions. Both demultiplexers and multiplexers have similar names, abbreviations, schematic symbols and circuits, so confusion is easy. The term multiplexer, and the abbreviation mux are often used to also mean a demultiplexer, or a multiplexer and a demultiplexer working together. So when you hear about a multiplexer, it may mean something quite different.

Using Multiple Combinational Circuits

As an example #1 of using several circuits together, we are going to make a device that will have 16 inputs, representing a four-digit number, to a four-digit 7-segment display but using just one binary-to-7-segment encoder.

First, the overall architecture of our circuit provides what looks like our description provided.



Follow this circuit through and you can confirm that it matches the description given above. There are 16 primary inputs and two more inputs used to select which digit will be displayed.

There are 28 outputs to control the four-digit 7-segment display. Only four of the primary inputs are encoded at a time. You may have noticed a potential question though.

When one of the digits is selected, what do the other three digits display? Review the circuit for the demultiplexers and notice that any line not selected by the A input is zero.

So the other three digits are blank. We don’t have a problem, only one digit displays at a time.

Notice how quickly this large circuit was developed from smaller parts. This is true of most complex circuits: they are composed of smaller parts allowing a designer to abstract away some complexity and understand the circuit as a whole.

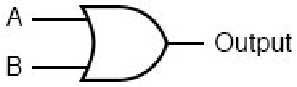
Sometimes a designer can even take components that others have designed and remove the detailed design work.

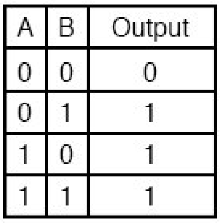
In addition to the added quantity of gates, this design suffers from one additional weakness. You can only see one display one digit at a time.

If there was some way to rotate through the four digits quickly, you could have the appearance of all four digits being displayed at the same time. That is a job for a sequential circuit.

Digital Logic With Feedback or sequential devices

With simple gate and combinational logic circuits, there is a definite output state for any given input state. Take the truth table of an OR gate, for instance:

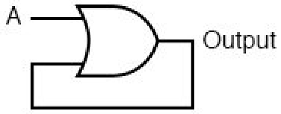


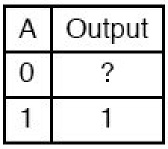


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

For each of the four possible combinations of input states (0-0, 0-1, 1-0, and 1-1), there is one, definite, unambiguous output state. Whether we're dealing with a multitude of cascaded gates or a single gate, that output state is determined by the truth table(s) for the gate(s) in the circuit, and nothing else.

However, if we alter this gate circuit so as to give signal feedback from the output to one of the inputs, strange things begin to happen:





A	Output
0	?
1	1

We know that if A is 1, the output must be 1, as well. Such is the nature of an OR gate: any "high" (1) input forces the output "high" (1). If A is "low" (0), however, we cannot guarantee the logic level or state of the output in our truth table.

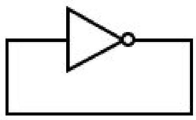
Since the output feeds back to one of the OR gate's inputs, and we know that any 1 input to an OR gates makes the output 1, this circuit will "latch" in the 1 output state after any time that A is 1. When A is 0, the output could be either 0 or 1, *depending on the circuit's prior state*!

The proper way to complete the above truth table would be to insert the word *latch* in place of the question mark, showing that the output maintains its last state when A is 0.

Any digital circuit employing feedback may be called a *multivibrator*. The example we just explored with the OR gate was a very simple example of what is called a *bistable* multivibrator. It is called "bistable" because it can hold stable in one of two possible output states, either 0 or 1. There are also *monostable* multivibrators, which have only one stable output state (that other state being momentary), which we'll explore later; and *astable* multivibrators, which have no stable state (oscillating back and forth between an output of 0 and 1).

A very simple astable multivibrator is an inverter with the output fed directly back to the input:

Inverter with feedback



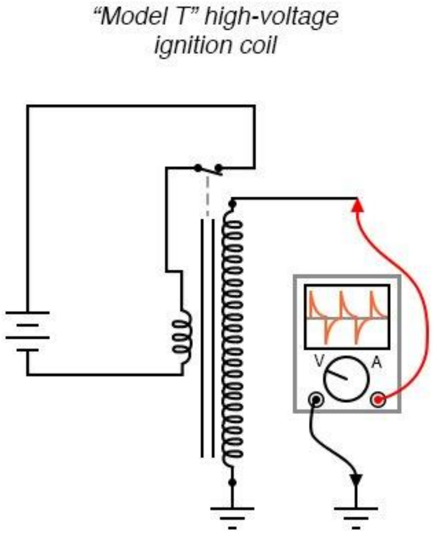
When the input is 0, the output switches to 1. That 1 output gets fed back to the input as a 1. When the input is 1, the output switches to 0. That 0 output gets fed back to the input as a 0, and the cycle repeats itself.

The result is a high frequency (several megahertz) oscillator, if implemented with a solid-state (semiconductor) inverter gate:

If implemented with relay logic, the resulting oscillator will be considerably slower, cycling at a frequency well within the audio range.

The buzzer or vibrator circuit thus formed was used extensively in early radio circuitry, as a way to convert steady, low-voltage DC power into pulsating DC power which could then be stepped up in voltage through a transformer to produce the high voltage necessary for operating the vacuum tube amplifiers.

Henry Ford's engineers also employed the buzzer/transformer circuit to create continuous high voltage for operating the spark plugs on Model T automobile engines:

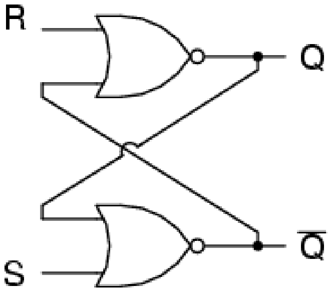


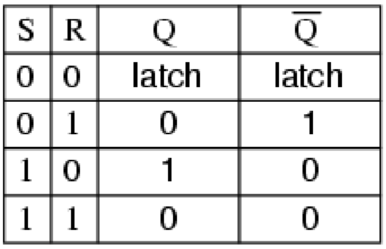
Borrowing terminology from the old mechanical buzzer (vibrator) circuits, solid-state circuit engineers referred to any circuit with two or more vibrators linked together as a multivibrator. The astable multivibrator mentioned previously, with only one “vibrator,” is more commonly implemented with multiple gates, as we’ll see later.

The most interesting and widely used multivibrators are of the bistable variety.

The S-R Latch

A bistable latch has *two* stable states, as indicated by the prefix ***bi*** in its name. Typically, one state is referred to as ***set*** and the other as ***reset***. The simplest bistable device, therefore, is known as a ***Set-Reset***, or **S-R**, latch. To create an S-R latch, we can wire two NOR gates in such a way that the output of one feeds back to the input of another, and vice versa, like this:





S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

The Q and not-Q outputs are supposed to be in opposite states. I say "supposed to" because making both the S and R inputs equal to 1 results in both Q and not-Q being 0. For this reason, having both S and R equal to 1 is called an *invalid* or *illegal* state for the S-R latch.

Otherwise, making S=1 and R=0 "sets" the latch so that Q=1 and not-Q=0. Conversely, making R=1 and S=0 "resets" the latch in the opposite state. When S and R are both equal to 0, the latch's outputs "latch" in their prior states.

By definition, a condition of Q=1 and not-Q=0 is *set*. A condition of Q=0 and not-Q=1 is *reset*. These terms are universal in describing the output states of any latch circuit. The astute observer will note that the initial power-up condition of either the gate of S-R latch is such that both gates start in the de-energized mode.

As such, one would expect that the circuit will start up in an invalid condition, with both Q and not-Q outputs being in the same state. Actually, this is true! However, the invalid condition is unstable with both S and R inputs inactive, and the circuit will quickly stabilize in either the set or reset condition because one gate is bound to react a little faster than the other.

If both gates were precisely identical, they would oscillate between high and low like an astable latch upon power-up without ever reaching a point of stability!

Fortunately for cases like this, such a precise match of components is a rare possibility.

It must be noted that although an astable (continually oscillating) condition would be extremely rare, there will most likely be a cycle or two of oscillation in the above circuit, and the final state of the circuit (set or reset) after power-up would be unpredictable.

A race condition occurs when two mutually-exclusive events are simultaneously initiated through different circuit elements by a single cause.

Race conditions should be avoided in circuit design primarily for the unpredictability that will be created.

One way to avoid such a condition is to insert a time-delay element into the circuit to disable one of the competing relays for a short time, giving the other one a clear advantage.

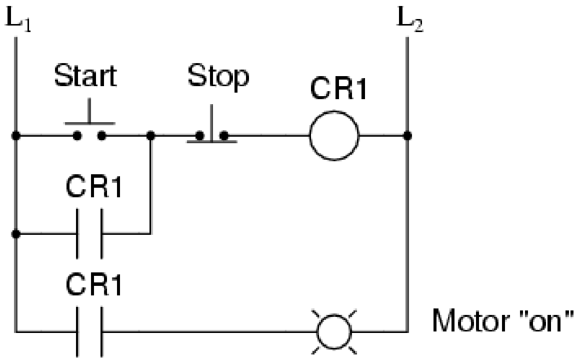
In other words, by purposely slowing down the de-energization of one relay, we ensure that the other relay will always “win” and the race results will always be predictable.

The end result is that the circuit powers up cleanly and predictably in the reset state with S=0 and R=0.

Complex computer programs, for that matter, may also incur race problems if improperly designed. Race problems are a possibility for any sequential system, and may not be discovered until some time after initial testing of the system. They can be very difficult problems to detect and eliminate.

A practical application of an S-R latch circuit might be for starting and stopping a motor, using normally-open, momentary pushbutton switch contacts for both start (S) and stop (R) switches.

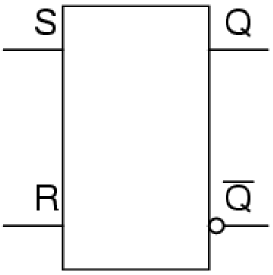
Normally, a simple ladder logic circuit is employed, such as this:



In the above motor start/stop circuit, the CR1 contact in parallel with the start switch contact is referred to as a “seal-in” contact, because it “seals” or latches control relay CR1 in the energized state after the start switch has been released.

To break the “seal,” or to “unlatch” or “reset” the circuit, the stop pushbutton is pressed, which de-energizes CR1 and restores the seal-in contact to its normally open status. Notice, however, that this circuit performs much the same function as the S-R latch.

Also, note that this circuit has no inherent instability problem (if even a remote possibility). In semiconductor form, S-R latches come in prepackaged units so that you don’t have to build them from individual gates. They are symbolized as such:



REVIEW:

- A *bistable* latch is one with *two* stable output states.
- In a bistable latch, the condition of Q=1 and not-Q=0 is defined as *set*. A condition of Q=0 and not-Q=1 is conversely defined as *reset*. If Q and not-Q happen to be forced to the same state (both 0 or both 1), that state is referred to as *invalid*.

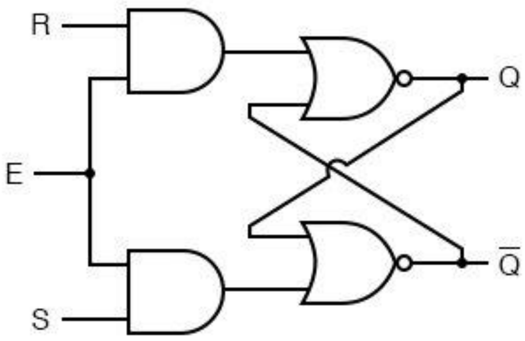
- In an S-R latch, activation of the S input sets the circuit, while activation of the R input resets the circuit. If both S and R inputs are activated simultaneously, the circuit will be in an invalid condition.
- A *race condition* is a state in a sequential system where two mutually-exclusive events are simultaneously initiated by a single cause.

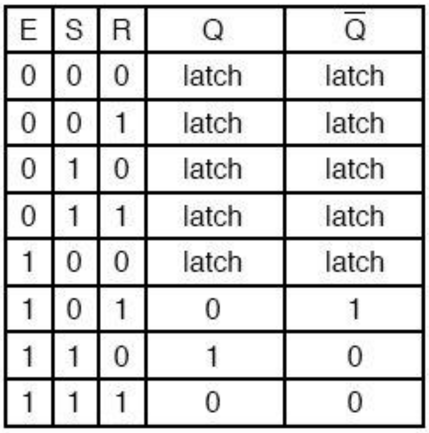
The Gated S-R Latch

It is sometimes useful in logic circuits to have a latch which changes state only when certain conditions are met, regardless of its S and R input states.

The conditional input is called the enable, and is symbolized by the letter E (Enable). Study the following example to see how this works:

Gated SR- Latch Truth Table





E	S	R	Q	$\overline{Q}$
0	0	0	latch	latch
0	0	1	latch	latch
0	1	0	latch	latch
0	1	1	latch	latch
1	0	0	latch	latch
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

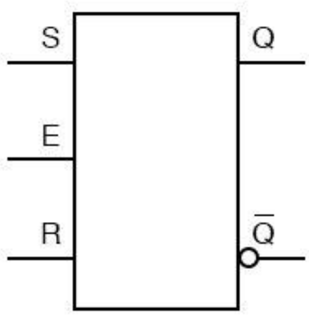
When the E=0, the outputs of the two AND gates are forced to 0, regardless of the states of either S or R. Consequently, the circuit behaves as though S and R were both 0, latching the Q and not-Q outputs in their last states.

Only when the enable input is activated (1) will the latch respond to the S and R inputs.

A practical application of this might be the same motor control circuit (with two normally-open push button switches for start and stop), except with the addition of a master lockout input (E) that disables both push buttons from having control over the motor when its low (0).

Once again, these latch (multivibrator) circuits are available as prepackaged semiconductor devices ("chips"), and are symbolized as such:

S-R Gated Latch Symbol



It is also common to see the enable input designated by the letters "EN" instead of just "E."

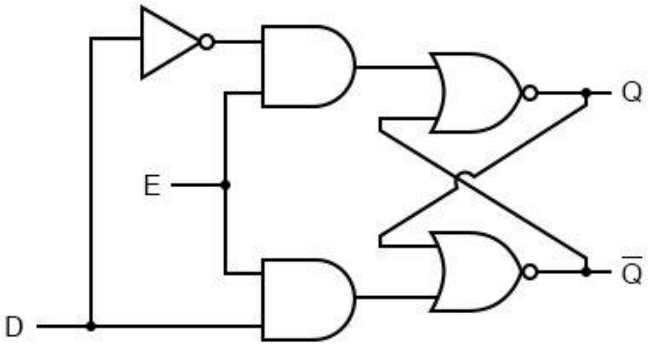
REVIEW:

- The *enable* input on a latch (multivibrator) must be activated for either S or R inputs to have any effect on the output state.
- This enable input is sometimes labeled "E", and other times as "EN".

The D Latch

Since the enable input on a gated S-R latch provides a way to latch the Q and not-Q outputs without regard to the status of S or R, we can eliminate one of those inputs to create a multivibrator latch circuit with no "illegal" input states.

Such a circuit is called a D latch, and its internal logic looks like this:

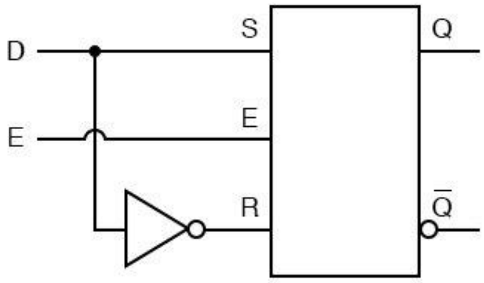


E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

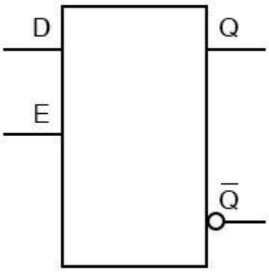
Note that the R input has been replaced with the complement (inversion) of the old S input, and the S input has been renamed to D. As with the gated S-R latch, the D latch will not respond to a signal input if the enable input is 0—it simply stays latched in its last state. When the enable input is 1, however, the Q output follows the D input.

Since the R input of the S-R circuitry has been done away with, this latch has no “invalid” or “illegal” state. Q and not-Q are always opposite of one another.

If the above diagram is confusing at all, the next diagram should make the concept simpler:



Like both the S-R and gated S-R latches, the D latch circuit may be found as its own prepackaged circuit, complete with a standard symbol:



The D latch is nothing more than a gated S-R latch with an inverter added to make R the complement (inverse) of S.

An application for the D latch is a 1-bit memory circuit. You can “write” (store) a 0 or 1 bit in this latch circuit by making the enable input high (1) and setting D to whatever you want the stored bit to be. When the enable input is made low (0), the latch ignores the status of the D input and merrily holds the stored bit value, outputting at the stored value at Q, and its inverse on output not-Q.

REVIEW:

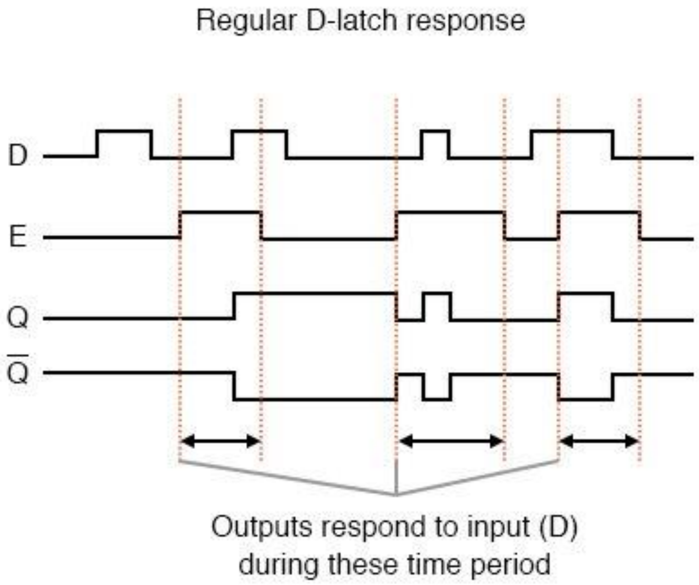
- A D latch is like an S-R latch with only one input: the “D” input. Activating the D input sets the circuit, and de-activating the D input resets the circuit. Of course, this is only if the enable input (E) is activated as well. Otherwise, the output(s) will be latched, unresponsive to the state of the D input.
- D latches can be used as 1-bit memory circuits, storing either a “high” or a “low” state when disabled, and “reading” new data from the D input when enabled.

Edge-triggered Latches: Flip-Flops

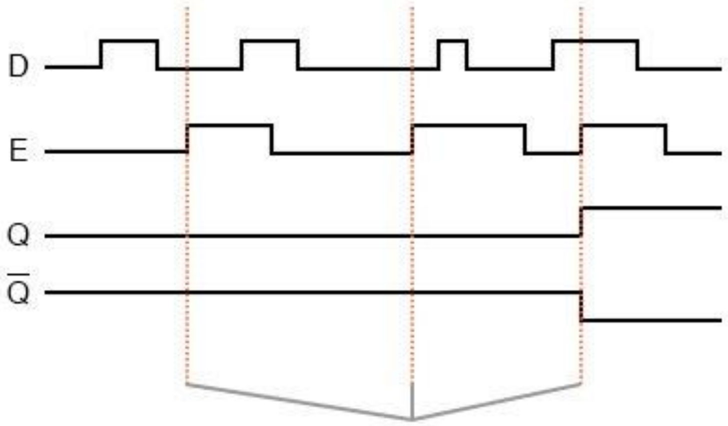
So far, we've studied both S-R and D latch circuits with enable inputs. The latch responds to the data inputs (S-R or D) only when the enable input is activated. In many digital applications, however, it is desirable to limit the responsiveness of a latch circuit to a very short period of time instead of the entire duration that the enabling input is activated.

One method of enabling a multivibrator circuit is called *edge triggering*, where the circuit's data inputs have control only during the time that the enable input is *transitioning* from one state to another.

Let's compare timing diagrams for a normal D latch versus one that is edge-triggered:



Positive edge-triggered D-latch response



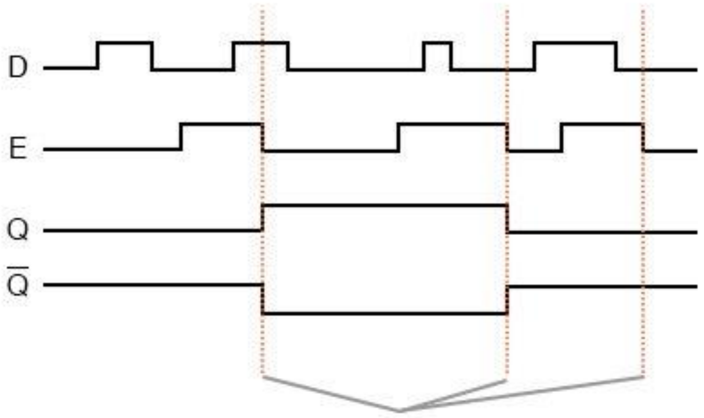
Outputs respond to input (D)
only when enable signal transitions
from low to high

In the first timing diagram, the outputs respond to input D whenever the enable (E) input is high, for however long it remains high. When the enable signal falls back to a low state, the circuit remains latched.

In the second timing diagram, we note a distinctly different response in the circuit output(s): it only responds to the D input during that brief moment of time when the enable signal *changes*, or *transitions*, from low to high. This is known as *positive* edge-triggering.

There is such a thing as *negative* edge triggering as well, and it produces the following response to the same input signals:

Negative edge-triggered D-latch response



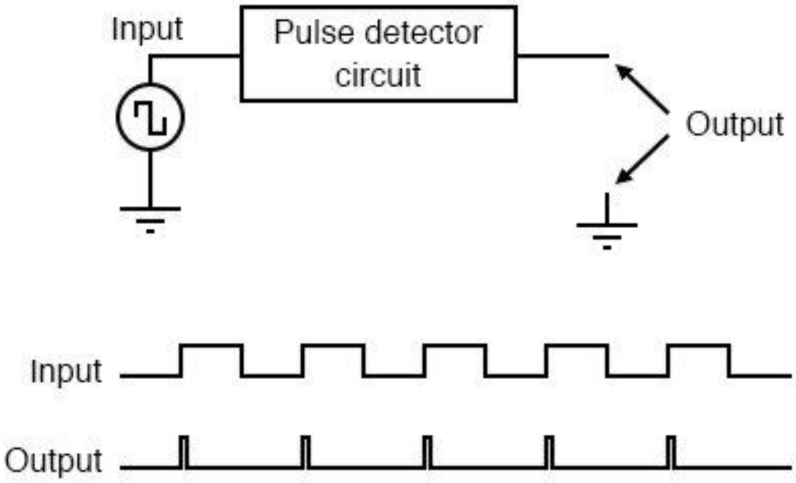
Outputs respond to input (D)
only when enable signal transitions
from high to low

Whenever we enable a multivibrator circuit on the transitional edge of a square-wave enable signal, we call it a ***flip-flop*** instead of a *latch*.

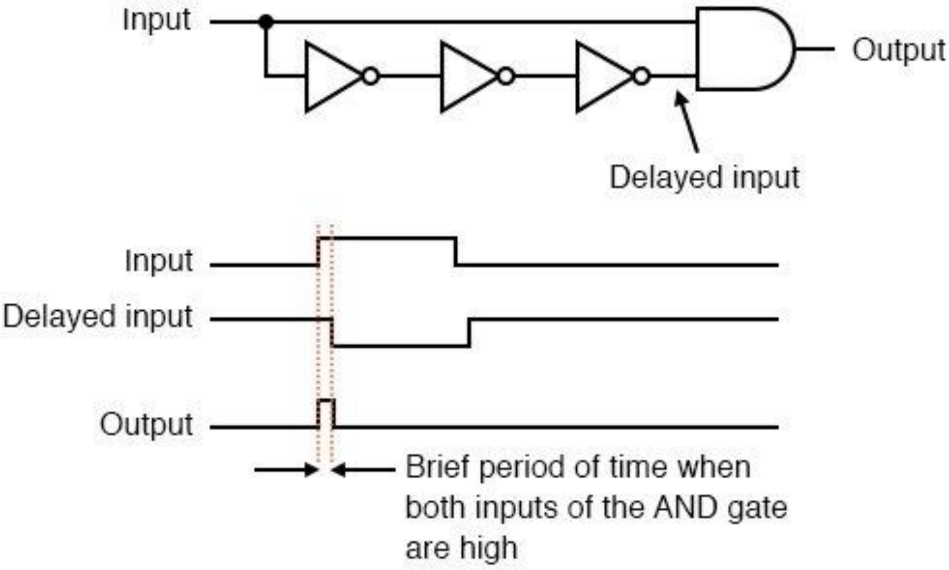
Consequently, and edge-triggered S-R circuit is more properly known as an S-R flip-flop, and an edge-triggered D circuit as a D flip-flop. The enable signal is renamed to be the clock signal. Also, we refer to the data inputs (S, R, and D, respectively) of these flip-flops as *synchronous* inputs, because they have effect only at the time of the clock pulse edge (transition), thereby synchronizing any output changes with that clock pulse, rather than at the whim of the data inputs.

But, how do we actually accomplish this edge-triggering? To create a "gated" S-R latch from a regular S-R latch is easy enough with a couple of AND gates, but how do we implement logic that only pays attention to the *rising* or *falling edge* of a changing digital signal?

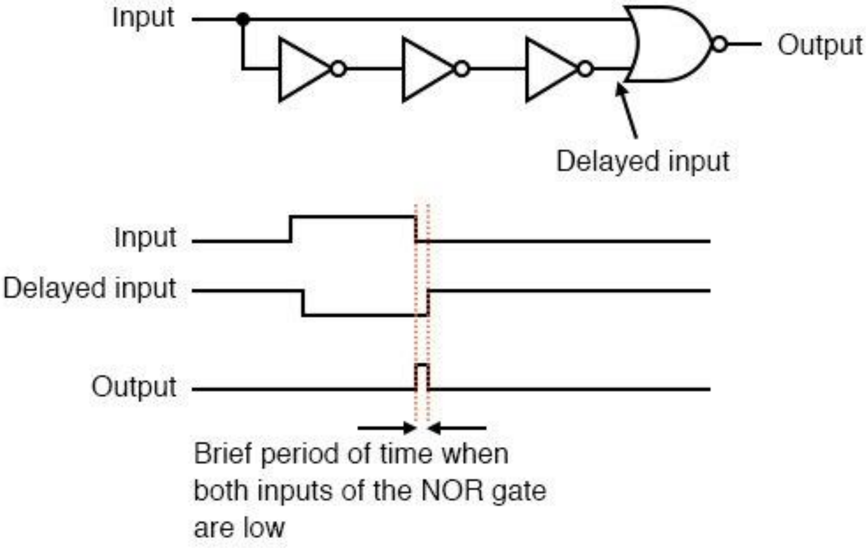
What we need is a digital circuit that outputs a brief pulse whenever the input is activated for an arbitrary period of time, and we can use the output of this circuit to briefly enable the latch. We're getting a little ahead of ourselves here, but this is actually a kind of monostable multivibrator, which for now we'll call a *pulse detector*.



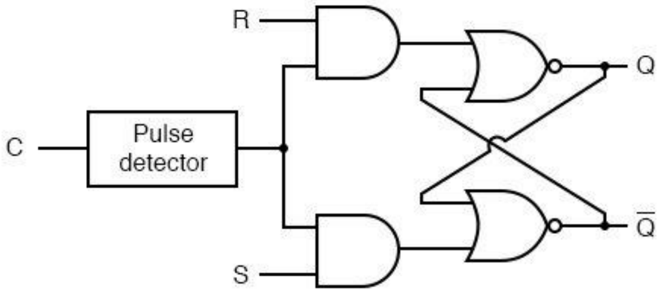
Implementing this timing function with semiconductor components is actually quite easy, as it exploits the inherent time delay within every logic gate (known as propagation delay). What we do is take an input signal and split it up two ways, then place a gate or a series of gates in one of those signal paths just to delay it a bit, then have both the original signal and its delayed counterpart enter into a two-input gate that outputs a high signal for the brief moment of time that the delayed signal has not yet caught up to the low-to-high change in the non-delayed signal. An example circuit for producing a clock pulse on a low-to-high input signal transition is shown here:


Input
Output
Delayed input
Input
Delayed input
Output
Brief period of time when both inputs of the AND gate are high

This circuit may be converted into a negative-edge pulse detector circuit with only a change of the final gate from AND to NOR:


Input
Output
Delayed input
Input
Delayed input
Output
Brief period of time when both inputs of the NOR gate are low

Now that we know how a pulse detector can be made, we can show it attached to the enable input of a latch to turn it into a flip-flop. In this case, the circuit is a S-R flip-flop:

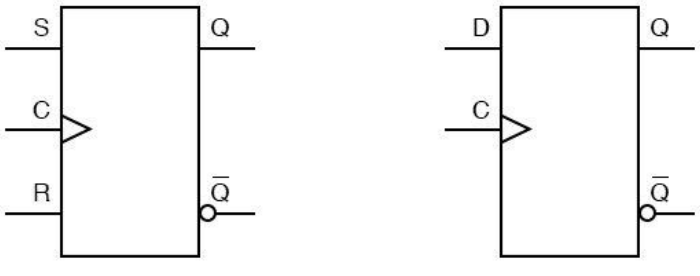


C	S	R	Q	$\overline{Q}$
↑	0	0	latch	latch
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	0	0
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch

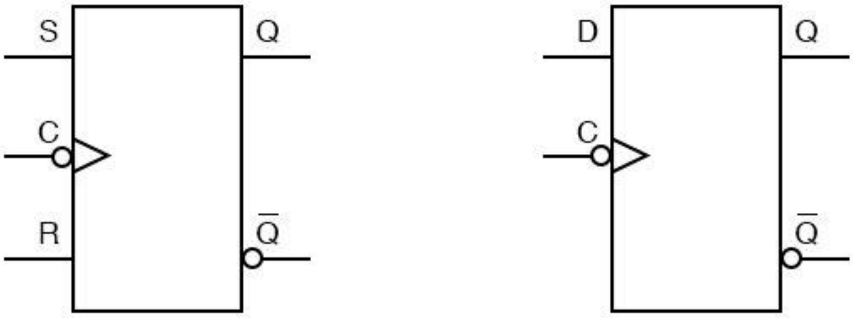
Only when the clock signal (C) is transitioning from low to high is the circuit responsive to the S and R inputs. For any other condition of the clock signal (“x”) the circuit will be latched.

It is important to note that the invalid state for the S-R flip-flop is maintained only for the short period of time that the pulse detector circuit allows the latch to be enabled. After that brief time period has elapsed, the outputs will latch into either the set or the reset state. Once again, the problem of a *race condition* manifests itself. With no enable signal, an invalid output state cannot be maintained. However, the valid “latched” states of the multivibrator—set and reset—are mutually exclusive to one another. Therefore, the two gates of the multivibrator circuit will “race” each other for supremacy, and whichever one attains a high output state first will “win.”

The block symbols for flip-flops are slightly different from that of their respective latch counterparts:



The triangle symbol next to the clock inputs tells us that these are edge-triggered devices, and consequently that these are flip-flops rather than latches. The symbols above are positive edge-triggered: that is, they “clock” on the rising edge (low-to-high transition) of the clock signal. Negative edge-triggered devices are symbolized with a bubble on the clock input line:



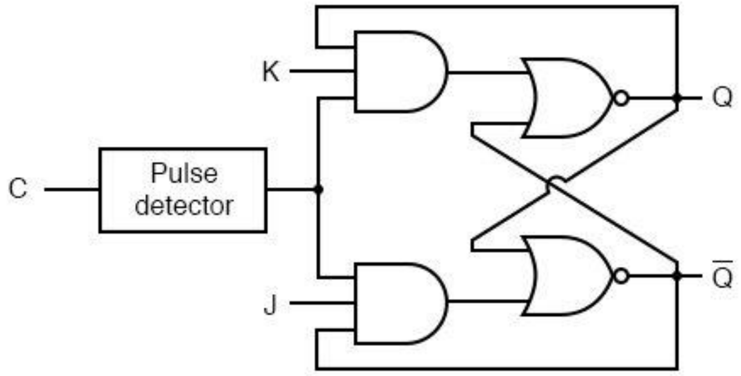
Both of the above flip-flops will "clock" on the falling edge (high-to-low transition) of the clock signal.

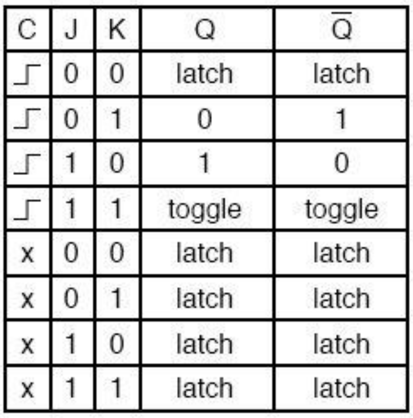
REVIEW:

- A *flip-flop* is a latch circuit with a "pulse detector" circuit connected to the enable (E) input, so that it is enabled only for a brief moment on either the rising or falling edge of a clock pulse.
- Pulse detector circuits may be made from time-delay relays for ladder logic applications, or from semiconductor gates (exploiting the phenomenon of *propagation delay*).

The J-K Flip-Flop

Another variation on a theme of bistable multivibrators is the J-K flip-flop. Essentially, this is a modified version of an S-R flip-flop with no "invalid" or "illegal" output state. Look closely at the following diagram to see how this is accomplished:





C	J	K	Q	$\overline{Q}$
↑	0	0	latch	latch
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	toggle	toggle
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch

The J and K Inputs

What used to be the S and R inputs are now called the J and K inputs, respectively. The old two-input AND gates have been replaced with 3-input AND gates, and the third input of each gate receives feedback from the Q and not-Q outputs.

What this does for us is permit the J input to have effect only when the circuit is reset, and permit the K input to have effect only when the circuit is set.

In other words, the two inputs are *interlocked*, to use a relay logic term, so that they cannot both be activated simultaneously.

If the circuit is “set,” the J input is inhibited by the 0 status of not-Q through the lower AND gate; if the circuit is “reset,” the K input is inhibited by the 0 status of Q through the upper AND gate.

When both J and K inputs are 1, however, something unique happens. Because of the selective inhibiting action of those 3-input AND gates, a “set” state inhibits input J so that the flip-flop acts as if J=0 while K=1 when in fact both are 1.

On the next clock pulse, the outputs will switch (“toggle”) from set (Q=1 and not-Q=0) to reset (Q=0 and not-Q=1). Conversely, a “reset” state inhibits input K so that the flip-flop acts as if J=1 and K=0 when in fact both are 1. The next clock pulse toggles the circuit again from reset to set.

Logical Sequence of J-K Flip-Flop

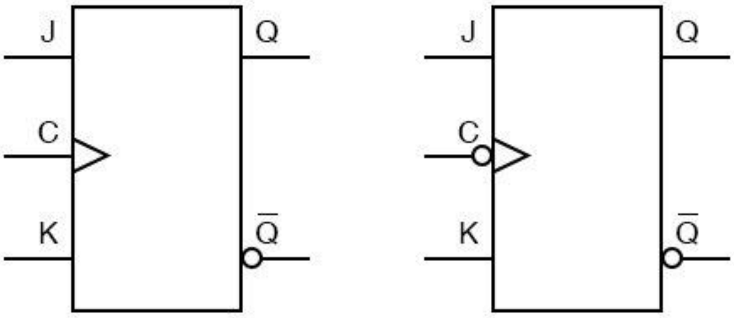
The end result is that the S-R flip-flop’s “invalid” state is eliminated (along with the race condition it engendered) and we get a useful feature as a bonus: the ability to toggle between the two (bistable) output states with every transition of the clock input signal.

There is no such thing as a J-K latch, only J-K flip-flops. Without the edge-triggering of the clock input, the circuit would continuously toggle between its two output states when both J and K were held high (1), making it an astable device instead of a bistable device in that circumstance.

If we want to preserve bistable operation for all combinations of input states, we must use edge-triggering so that it toggles only when we tell it to, one step (clock pulse) at a time.

The Block Symbol for J-K Flip-Flops

The block symbol for a J-K flip-flop is a whole lot less frightening than its internal circuitry, and just like the S-R and D flip-flops, J-K flip-flops come in two clock varieties (negative and positive edge-triggered):



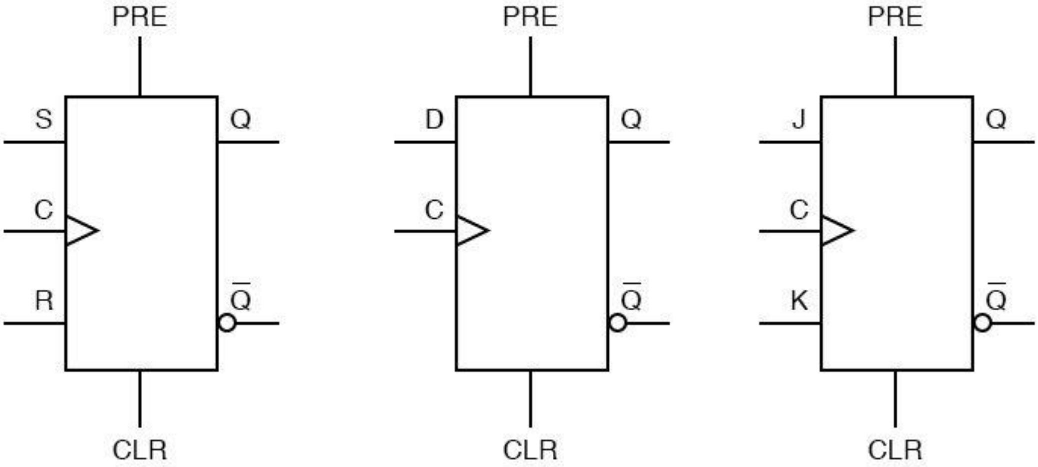
REVIEW:

- A J-K flip-flop is nothing more than an S-R flip-flop with an added layer of feedback. This feedback selectively enables one of the two set/reset inputs so that they cannot both carry an active signal to the multivibrator circuit, thus eliminating the invalid condition.
- When both J and K inputs are activated, and the clock input is pulsed, the outputs (Q and not-Q) will swap states. That is, the circuit will *toggle* from a set state to a reset state or vice versa.

Asynchronous Flip-Flop Inputs

The normal data inputs to a flip flop (D, S and R, or J and K) are referred to as synchronous inputs because they have an effect on the outputs (Q and not-Q) only in step, or in sync, with the clock signal transitions.

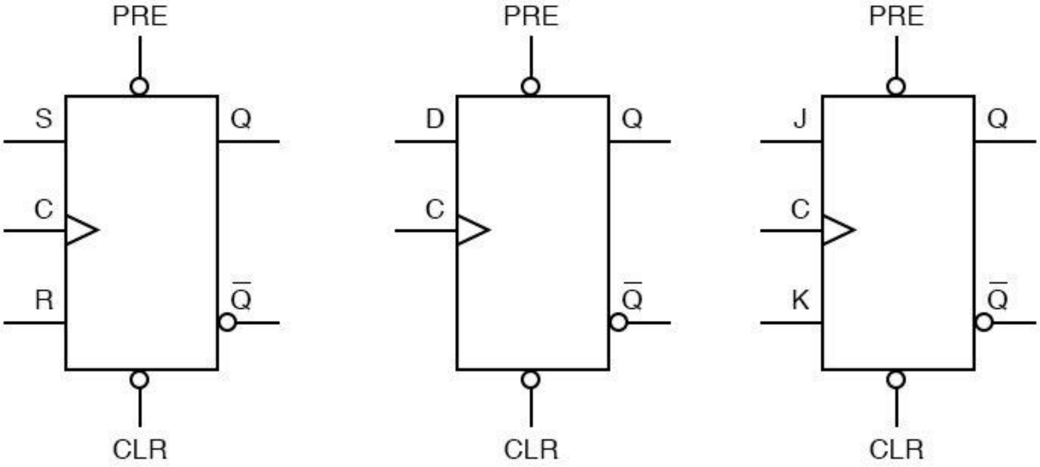
These extra inputs are called asynchronous because they can set or reset the flip-flop regardless of the status of the clock signal. Typically, they’re called preset and clear:



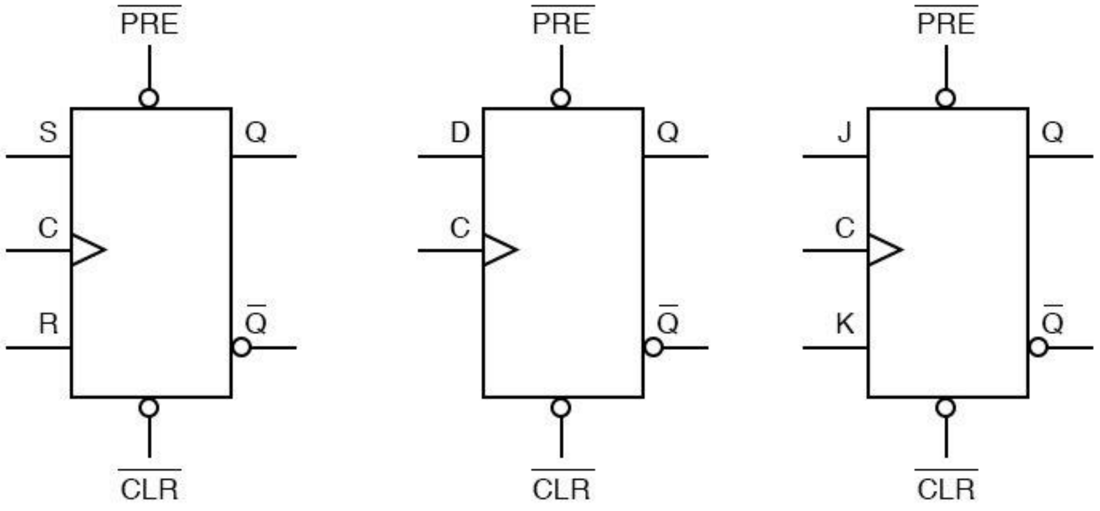
When the preset input is activated, the flip-flop will be set (Q=1, not-Q=0) regardless of any of the synchronous inputs or the clock. When the clear input is activated, the flip-flop will be reset (Q=0, not-Q=1), regardless of any of the synchronous inputs or the clock.

So, what happens if both preset and clear inputs are activated? Surprise, surprise: we get an invalid state on the output, where Q and not-Q go to the same state, the same as our old friend, the S-R latch! Preset and clear inputs find use when multiple flip-flops are ganged together to perform a function on a multi-bit binary word, and a single line is needed to set or reset them all at once.

Asynchronous inputs, just like synchronous inputs, can be engineered to be active-high or active-low. If they're active-low, there will be an inverting bubble at that input lead on the block symbol, just like the negative edge-trigger clock inputs.



Sometimes the designations “PRE” and “CLR” will be shown with inversion bars above them, to further denote the negative logic of these inputs:



REVIEW:

- *Asynchronous* inputs on a flip-flop have control over the outputs (Q and not-Q) regardless of clock input status.
- These inputs are called the *preset* (PRE) and *clear* (CLR). The preset input drives the flip-flop to a set state while the clear input drives it to a reset state.
- It is possible to drive the outputs of a J-K flip-flop to an invalid condition using the asynchronous inputs, because all feedback of circuit is overridden.

Monostable Multivibrators

We've already seen one example of a monostable multivibrator in use: the pulse detector used within the circuitry of flip-flops, to enable the latch portion for a brief time when the clock input signal transitions from either low to high or high to low.

The pulse detector is classified as a monostable multivibrator because it has only one stable state. Stable state is a state of output where the device is able to latch or hold to forever, without external prodding.

A latch or flip-flop, being a bistable device, can hold in either the "set" or "reset" state for an indefinite period of time. Once its set or reset, it will continue to latch in that state unless prompted to change by an external input.

A mechanical analogy of a monostable device would be a momentary contact pushbutton switch, which spring-returns to its normal (stable) position when pressure is removed from its button actuator.

Likewise, a standard wall (toggle) switch, such as the type used to turn lights on and off in a house, is a bistable device. It can latch in one of two modes: on or off.

All monostable multivibrators are timed devices. That is, their unstable output state will hold only for a certain minimum amount of time before returning to its stable state.

With semiconductor monostable circuits, this timing function is typically accomplished through the use of resistors and capacitors, making use of the exponential charging rates of RC circuits.

A comparator is often used to compare the voltage across the charging (or discharging) capacitor with a steady reference voltage, and the on/off output of the comparator used for a logic signal.

With ladder logic, time delays are accomplished with time-delay relays, which can be constructed with semiconductor/RC circuits like that just mentioned, or mechanical delay devices which impede the immediate motion of the relay’s armature.

No matter how long the input signal stays high (logic 1), the output remains high for just 1 second of time, then returns to its normal (stable) low state.

For some applications, it is necessary to have a monostable device that outputs a longer pulse than the input pulse which triggers it.

One-shot multivibrators of both the retriggerable and non-retriggerable variety find wide application in industry for siren actuation and machine sequencing, where an intermittent input signal produces an output signal of a set time.

REVIEW:

- A *monostable* multivibrator has only one stable output state. The other output state can only be maintained temporarily.
- One-shot circuits with very short time settings may be used to *debounce* the “dirty” signals created by mechanical switch contacts.

Finite State Machines

Until now (before flip-flops) every presented circuit was a combinatorial circuit. That means that its output is dependent only by its current inputs. Previous inputs for that type of circuits have no effect on the output.

However, there are many applications where there is a need for our circuits to have “memory”; to remember previous inputs and calculate their outputs according to them. A circuit whose output depends not only on the present input but also on the history of the input is called a sequential circuit.

Now we will learn how to design and build such sequential circuits. In order to see how this procedure works, we will use an example, on which we will study our topic.

So let’s suppose we have a digital quiz game that works on a clock and reads an input from a manual button. However, we want the switch to transmit only one HIGH pulse to the circuit. If we hook the button directly on the game circuit it will transmit HIGH for as few clock cycles as our finger can achieve. On a common clock frequency our finger can never be fast enough.

The design procedure has specific steps that must be followed in order to get the work done.

Step 1

The first step of the design procedure is to define with simple but clear words what we want our circuit to do:

“Our mission is to design a secondary circuit that will transmit a HIGH pulse with duration of only one cycle when the manual button is pressed, and won’t transmit another pulse until the button is depressed and pressed again.”

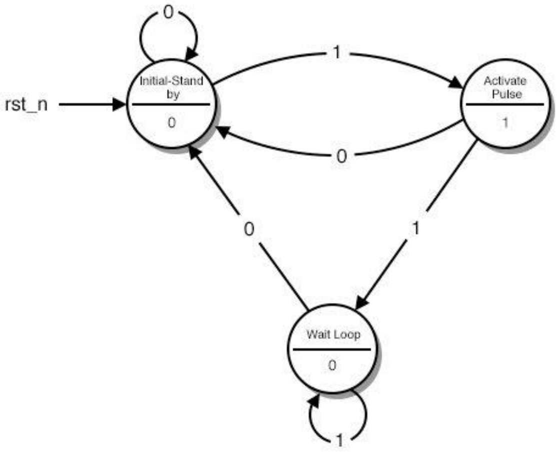
Step 2

The next step is to design a State Diagram.

This is a diagram that is made from circles and arrows and describes visually the operation of our circuit. In mathematic terms, this diagram that describes the operation of our sequential circuit is a Finite State Machine. Make a note that this is a Moore Finite State Machine.

Its output is a function of only its current state, not its input. That is in contrast with the Mealy Finite State Machine, where input affects the output.

The State Diagram of our circuit is the following: (Figure below)



A State Diagram

Every circle represents a “state”, a well-defined condition that our machine can be found at. In the upper half of the circle we describe that condition. The description helps us remember what our circuit is supposed to do at that condition.

- The first circle is the “stand-by” condition. This is where our circuit starts from and where it waits for another button press.
- The second circle is the condition where the button has just been just pressed and our circuit needs to transmit a HIGH pulse.
- The third circle is the condition where our circuit waits for the button to be released before it returns to the “stand-by” condition.

In the lower part of the circle is the output of our circuit. If we want our circuit to transmit a HIGH on a specific state, we put a 1 on that state. Otherwise we put a 0.

Every arrow represents a “transition” from one state to another. A transition happens once every clock cycle. Depending on the current Input, we may go to a different state each time. Notice the number in the middle of every arrow. This is the current Input.

For example, when we are in the “Initial-Stand by” state and we “read” a 1, the diagram tells us that we have to go to the “Activate Pulse” state. If we read a 0 we must stay on the “Initial-Stand by” state.

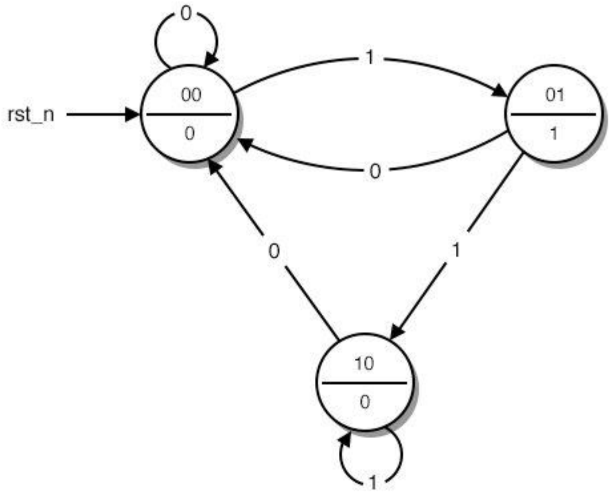
So, what does our “Machine” do exactly? It starts from the “Initial - Stand by” state and waits until a 1 is read at the Input. Then it goes to the “Activate Pulse” state and transmits a HIGH pulse on its output. If the button keeps being pressed, the circuit goes to the third state, the “Wait Loop”.

There it waits until the button is released (Input goes 0) while transmitting a LOW on the output. Then it’s all over again!

This is possibly the most difficult part of the design procedure, because it cannot be described by simple steps. It takes exprerience and a bit of sharp thinking in order to set up a State Diagram, but the rest is just a set of predetermined steps.

Step 3

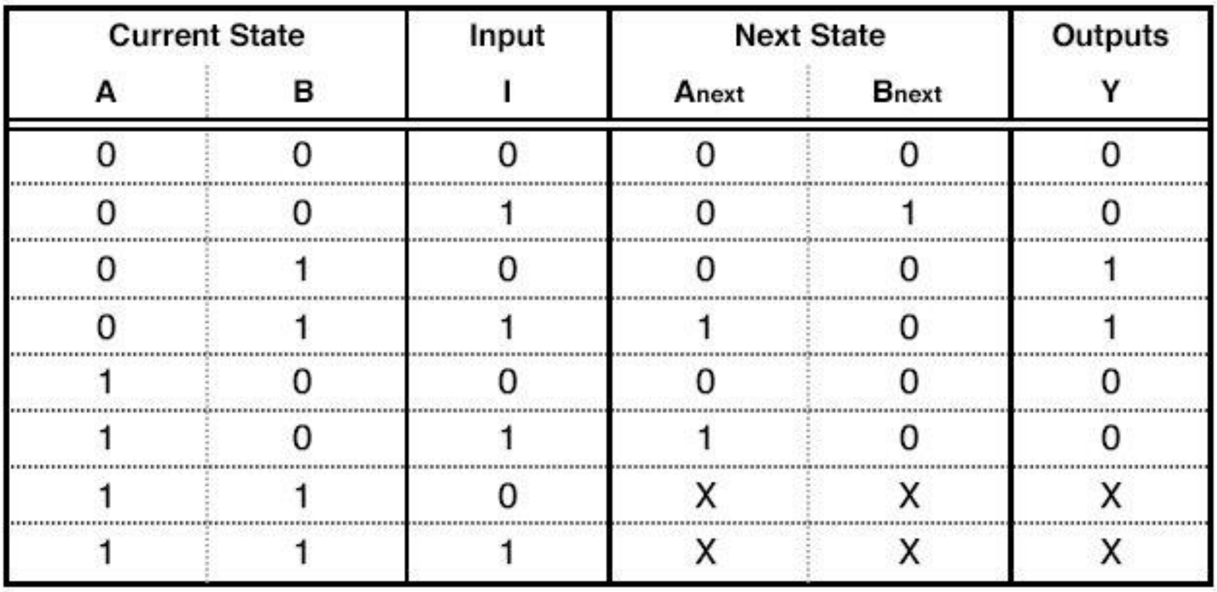
Next, we replace the words that describe the different states of the diagram with *binary* numbers. We start the enumeration from 0 which is assigned on the initial state. We then continue the enumeration with any state we like, until all states have their number. The result looks something like this: (Figure below)



A State Diagram with Coded States

Step 4

Afterwards, we fill the *State Table*. This table has a very specific form. I will give the table of our example and use it to explain how to fill it in. (Figure below)



Current State		Input	Next State		Outputs
A	B	I	A_{next}	B_{next}	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	X	X	X
1	1	1	X	X	X

A State Table

The first columns are as many as the bits of the highest number we assigned the State Diagram. If we had 5 states, we would have used up to the number 100, which means we would use 3 columns. For our example, we used up to the number 10, so only 2 columns will be needed. These columns describe the *Current State* of our circuit.

To the right of the Current State columns we write the *Input Columns*. These will be as many as our Input variables. Our example has only one Input.

Next, we write the *Next State Columns*. These are as many as the Current State columns.

Finally, we write the *Outputs Columns*. These are as many as our outputs. Our example has only one output. Since we have built a More Finite State Machine, the output is dependent on only the current input states. This is the reason the outputs column has two 1: to result in an output Boolean function that is independant of input I. Keep on reading for further details. The Current State and Input columns are the Inputs of our table. We fill them in with all the binary numbers from 0 to:

$2^{(\text{Number of Current State columns + Number of Input columns})} - 1$

It is simpler than it sounds fortunately. Usually there will be more rows than the actual States we have created in the State Diagram, but that’s ok.

Each row of the Next State columns is filled as follows: We fill it in with the state that we reach when, in the State Diagram, from the Current State of the same row we follow the Input of the same row. If have to fill in a row whose Current State number doesn't correspond to any actual State in the State Diagram we fill it with Don't Care terms (X). After all, we don't care where we can go from a State that doesn't exist. We wouldn't be there in the first place! Again it is simpler than it sounds.

The outputs column is filled by the output of the corresponding Current State in the State Diagram.

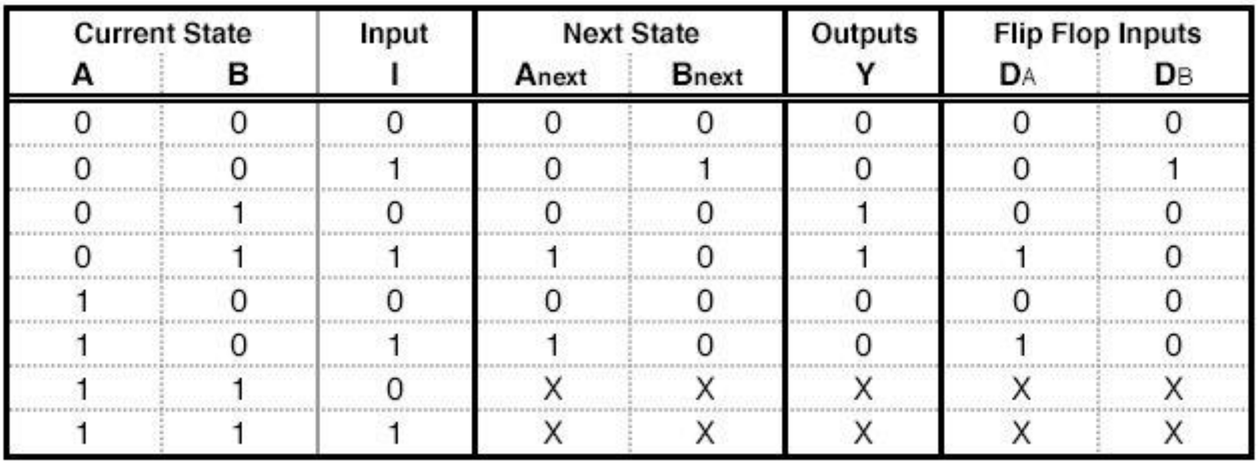
The State Table is complete! It describes the behaviour of our circuit as fully as the State Diagram does.

Step 5a

The next step is to take that theoretical "Machine" and implement it in a circuit. Most often than not, this implementation involves Flip Flops. This guide is dedicated to this kind of implementation and will describe the procedure for both D - Flip Flops as well as JK - Flip Flops. T - Flip Flops will not be included as they are too similar to the two previous cases. The selection of the Flip Flop to use is arbitrary and usually is determined by cost factors. The best choice is to perform both analysis and decide which type of Flip Flop results in minimum number of logic gates and lesser cost.

First we will examine how we implement our "Machine" with D-Flip Flops.

We will need as many D - Flip Flops as the State columns, 2 in our example. For every Flip Flop we will add one more column in our State table (Figure below) with the name of the Flip Flop's input, "D" for this case. The column that corresponds to each Flip Flop describes **what input we must give the Flip Flop in order to go from the Current State to the Next State**. For the D - Flip Flop this is easy: The necessary input is equal to the Next State. In the rows that contain X's we fill X's in this column as well.

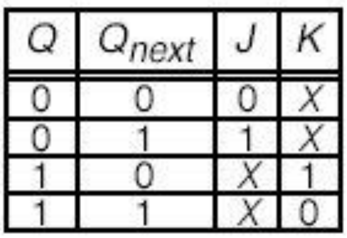


Current State		Input	Next State		Outputs	Flip Flop Inputs	
A	B	I	A_{next}	B_{next}	Y	D_A	D_B
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	0	0	1	0	0
0	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	1	0
1	1	0	X	X	X	X	X
1	1	1	X	X	X	X	X

A State Table with D - Flip Flop Excitations

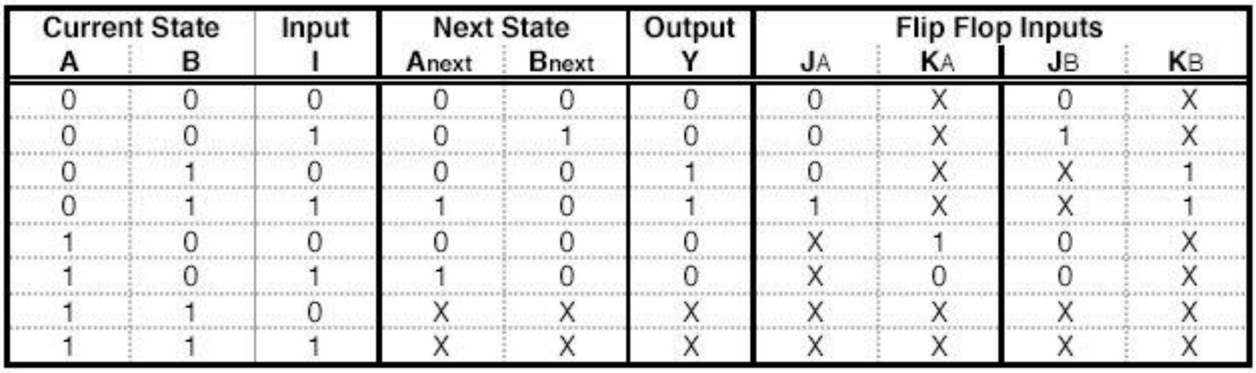
Step 5b

We can do the same steps with JK - Flip Flops. There are some differences however. A JK - Flip Flop has two inputs, therefore we need to add two columns for each Flip Flop. The content of each cell is dictated by the JK's excitation table:



Q	Q_{next}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

This table says that if we want to go from State Q to State Qnext, we need to use the specific input for each terminal. For example, to go from 0 to 1, we need to feed J with 1 and we **don't care** which input we feed to terminal K.



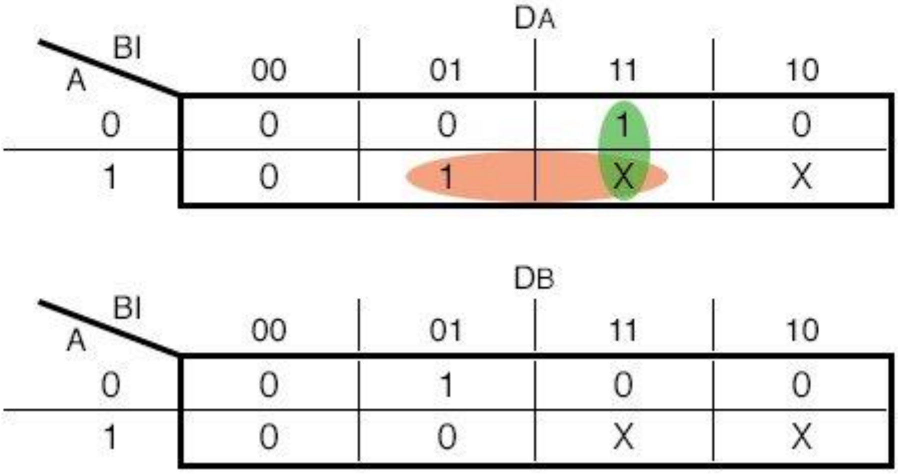
Current State		Input	Next State		Output	Flip Flop Inputs			
A	B	I	A_{next}	B_{next}	Y	J_A	K_A	J_B	K_B
0	0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	0	X	1	X
0	1	0	0	0	1	0	X	X	1
0	1	1	1	0	1	1	X	X	1
1	0	0	0	0	0	X	1	0	X
1	0	1	1	0	0	X	0	0	X
1	1	0	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X

A State Table with JK - Flip Flop Excitations

Step 6

We are in the final stage of our procedure. What remains, is to determine the Boolean functions that produce the inputs of our Flip Flops and the Output. We will extract one Boolean funtion for each Flip Flop input we have. This can be done with a Karnaugh Map. The input variables of this map are the Current State variables **as well as** the Inputs.

That said, the input functions for our D - Flip Flops are the following: (Figure below)



DA

A \ BI	00	01	11	10
0	0	0	1	0
1	0	1	X	X

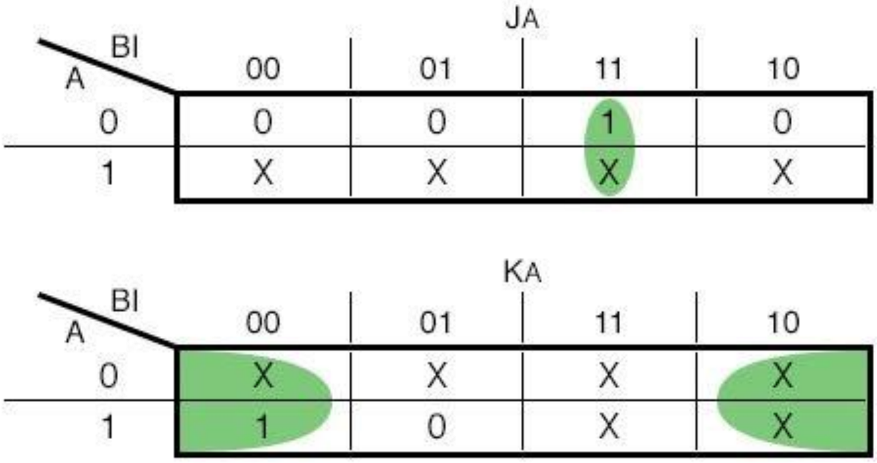
DB

A \ BI	00	01	11	10
0	0	1	0	0
1	0	0	X	X

Karnaugh Maps for the D - Flip Flop Inputs

$$D_A = A \cdot I + B \cdot I = (A + B) \cdot I$$
$$D_B = \overline{A} \cdot \overline{B} \cdot I$$

If we chose to use JK - Flip Flops our functions would be the following: (Figure below)

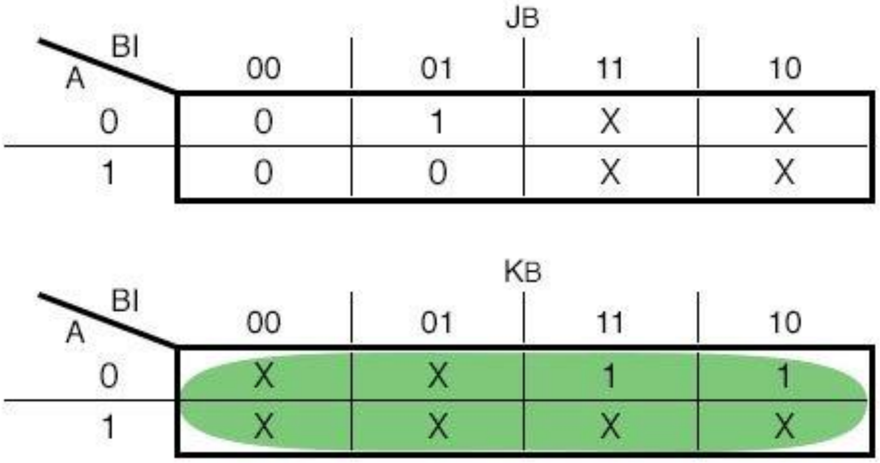


JA

A \ BI	00	01	11	10
0	0	0	1	0
1	X	X	X	X

KA

A \ BI	00	01	11	10
0	X	X	X	X
1	1	0	X	X



JB

A \ BI	00	01	11	10
0	0	1	X	X
1	0	0	X	X

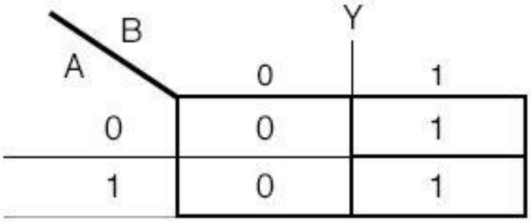
KB

A \ BI	00	01	11	10
0	X	X	1	1
1	X	X	X	X

Karnaugh Map for the JK - Flip Flop Input

$$J_A = B \cdot I$$
$$K_A = \overline{I}$$
$$J_B = \overline{A} \cdot I$$
$$K_B = 1$$

A Karnaugh Map will be used to determine the function of the Output as well: (Figure below)



Y

A \ B	0	1
0	0	1
1	0	1

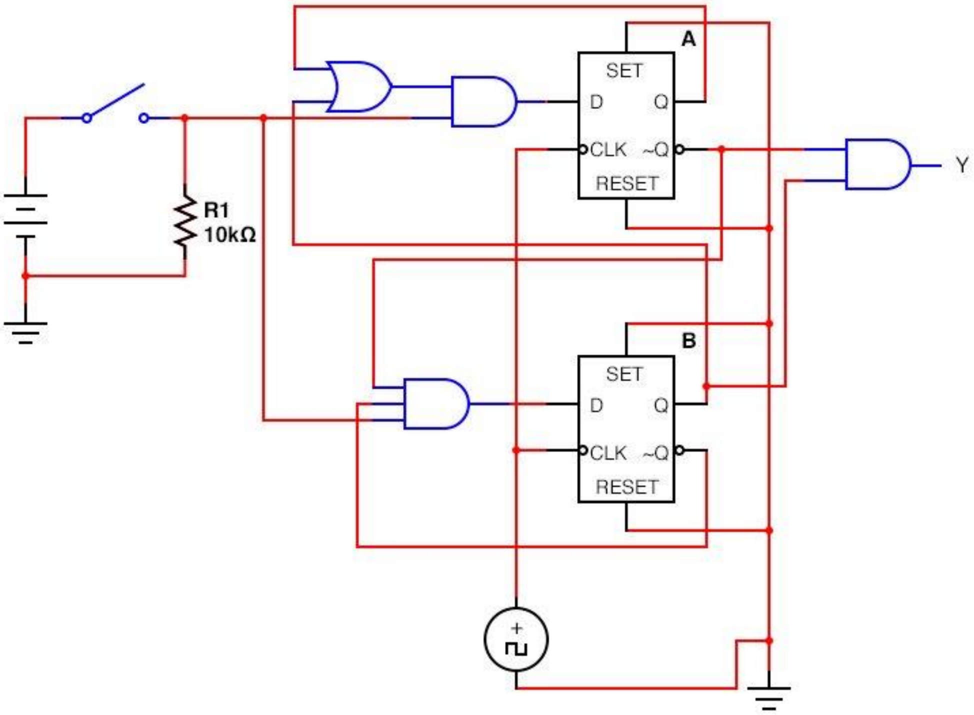
Karnaugh Map for the Output variable Y

$$Y = \overline{A} \cdot B$$

Step 7

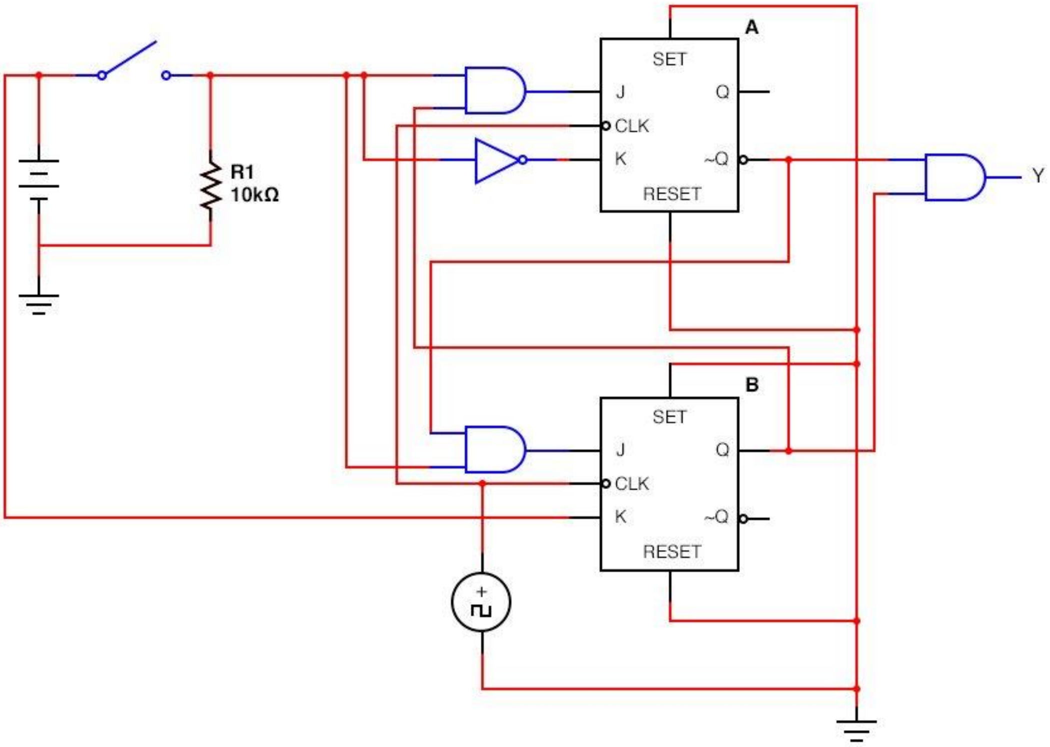
We design our circuit. We place the Flip Flops and use logic gates to form the Boolean functions that we calculated. The gates take input from the output of the Flip Flops and the Input of the circuit. Don’t forget to connect the clock to the Flip Flops!

The D - Flip Flop version: (Figure below)



The completed D - Flip Flop Sequential Circuit

The JK - Flip Flop version: (Figure below)



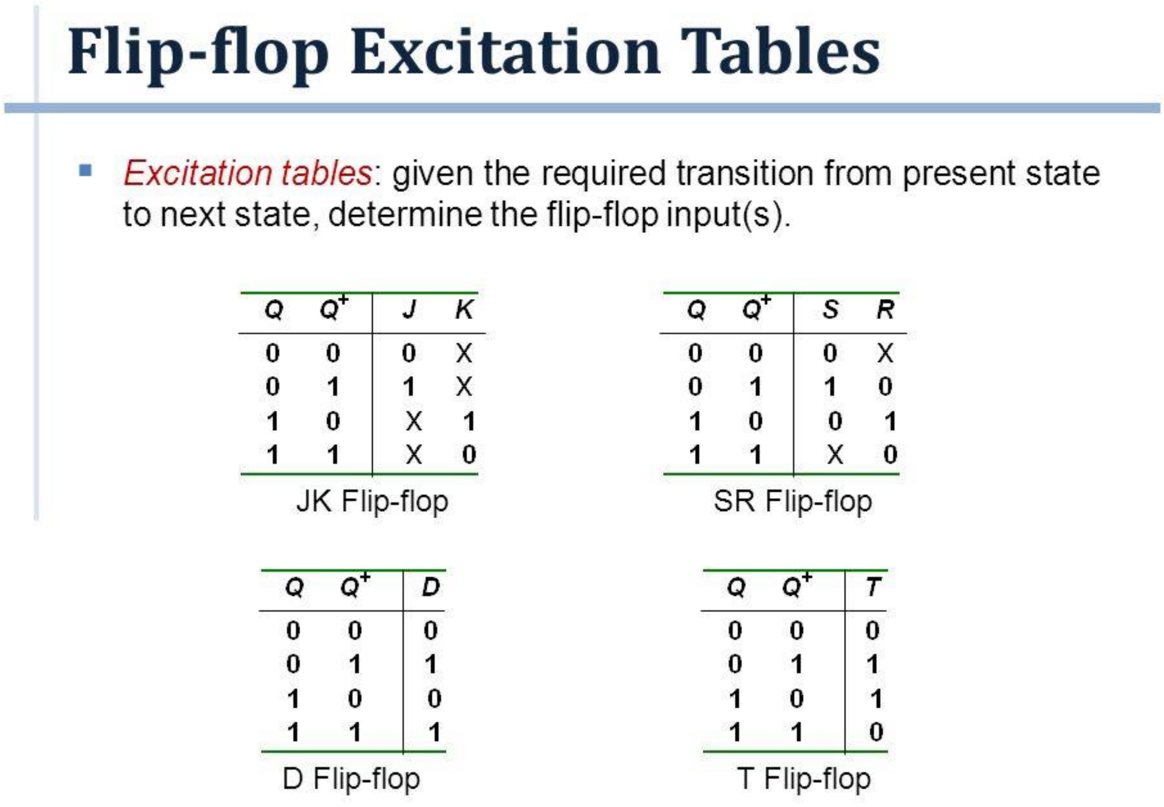
The completed JK - Flip Flop Sequential Circuit

We have successfully designed and constructed a Sequential Circuit. At first it might seem a daunting task, but after practice and repetition the procedure will become trivial. Sequential Circuits can come in handy as control parts of bigger circuits and can perform any sequential logic task that we can think of.

REVIEW:

- A Sequential Logic function has a “memory” feature and takes into account past inputs in order to decide on the output.
- The Finite State Machine is an abstract mathematical model of a sequential logic function. It has finite inputs, outputs and number of states.
- FSMs are implemented in real-life circuits through the use of Flip Flops
- The implementation procedure needs a specific order of steps (algorithm), in order to be carried out.

Flip-Flop Excitation or state tables



Flip-flop Excitation Tables

- *Excitation tables*: given the required transition from present state to next state, determine the flip-flop input(s).

Q	*Q*⁺	*J*	*K*
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

Q	*Q*⁺	*S*	*R*
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR Flip-flop

Q	*Q*⁺	*D*
0	0	0
0	1	1
1	0	0
1	1	1

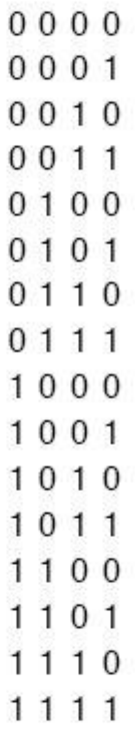
D Flip-flop

Q	*Q*⁺	*T*
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

Binary Count Sequence

If we examine a four-bit binary count sequence from 0000 to 1111, a definite pattern will be evident in the “oscillations” of the bits between 0 and 1:

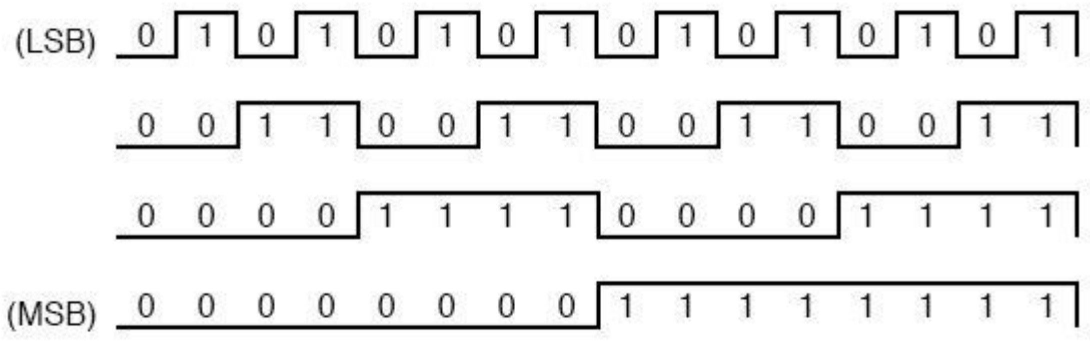


```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

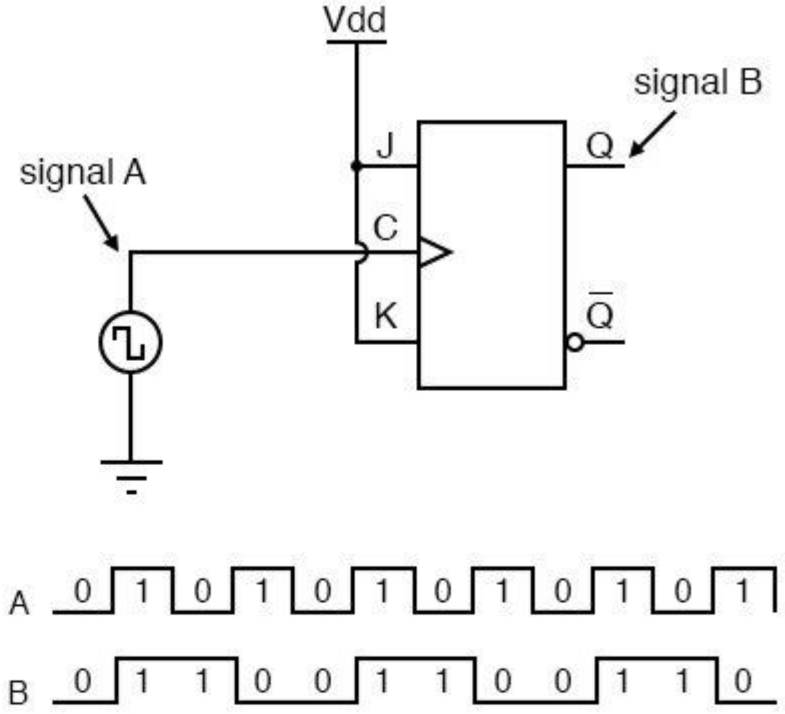
Note how the least significant bit (LSB) toggles between 0 and 1 for every step in the count sequence, while each succeeding bit toggles at one-half the frequency of the one before it.

The most significant bit (MSB) only toggles once during the entire sixteen-step count sequence: at the transition between 7 (0111) and 8 (1000).

If we wanted to design a digital circuit to “count” in four-bit binary, all we would have to do is design a series of frequency divider circuits, each circuit dividing the frequency of a square-wave pulse by a factor of 2:


(LSB) 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
(MSB) 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

J-K flip-flops are ideally suited for this task, because they have the ability to "toggle" their output state at the command of a clock pulse when both J and K inputs are made "high" (1):



If we consider the two signals (A and B) in this circuit to represent two bits of a binary number, signal A being the LSB and signal B being the MSB, we see that the count sequence is backward: from 11 to 10 to 01 to 00 and back again to 11.

Although it might not be counting in the direction we might have assumed, at least it counts!

Later'll explore different types of counter circuits, all made with J-K flip-flops, and all based on the exploitation of that flip-flop's toggle mode of operation.

REVIEW:

- Binary count sequences follow a pattern of octave frequency division: the frequency of oscillation for each bit, from LSB to MSB, follows a divide-by-two pattern. In other words, the LSB will oscillate at the highest frequency, followed by the next bit at one-half the LSB's frequency, and the next bit at one-half the frequency of the bit before it, etc.
- Circuits may be built that "count" in a binary sequence, using J-K flip-flops set up in the "toggle" mode.

Asynchronous Counters

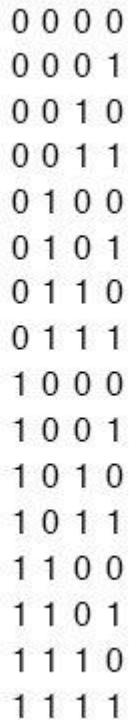
Above we saw a circuit using one J-K flip-flop that counted backward in a two-bit binary sequence, from 11 to 10 to 01 to 00.

Since it would be desirable to have a circuit that could count forward and not just backward, it would be worthwhile to examine a forward count sequence again and look for more patterns that might indicate how to build such a circuit.

Since we know that binary count sequences follow a pattern of octave (factor of 2) frequency division, and that J-K flip-flops set up for the “toggle” mode are capable of performing this type of frequency division, we can envision a circuit made up of several J-K flip-flops, cascaded to produce four bits of output.

The main problem facing us is to determine how to connect these flip-flops together so that they toggle at the right times to produce the proper binary sequence.

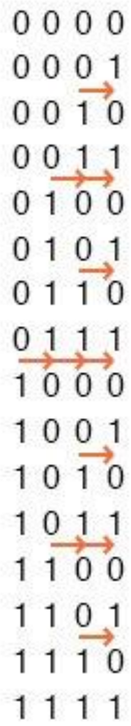
Examine the following binary count sequence, paying attention to patterns preceding the “toggling” of a bit between 0 and 1:



```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Note that each bit in this four-bit sequence toggles when the bit before it (the bit having a lesser significance, or place-weight), toggles in a particular direction: from 1 to 0.

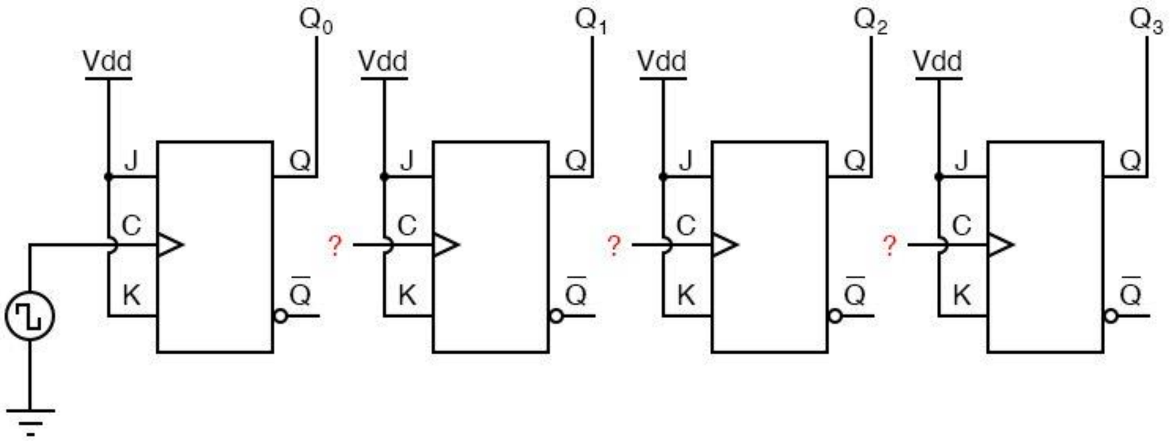
Small arrows indicate those points in the sequence where a bit toggles, the head of the arrow pointing to the previous bit transitioning from a “high” (1) state to a “low” (0) state:



```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

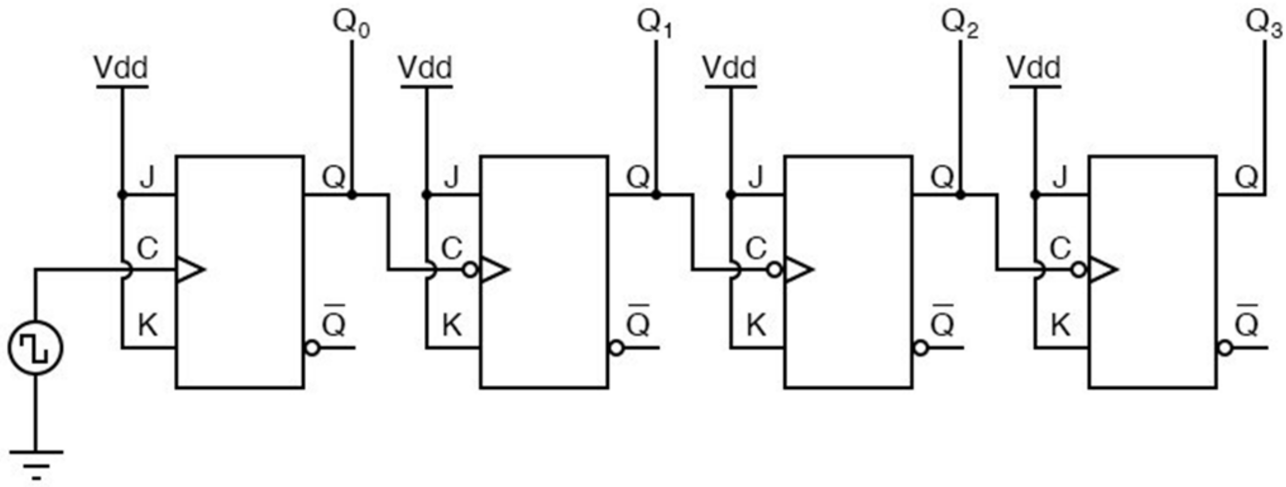
Starting with four J-K flip-flops connected in such a way to always be in the “toggle” mode, we need to determine how to connect the clock inputs in such a way so that each succeeding bit toggles when the bit before it transitions from 1 to 0.

The Q outputs of each flip-flop will serve as the respective binary bits of the final, four-bit count:

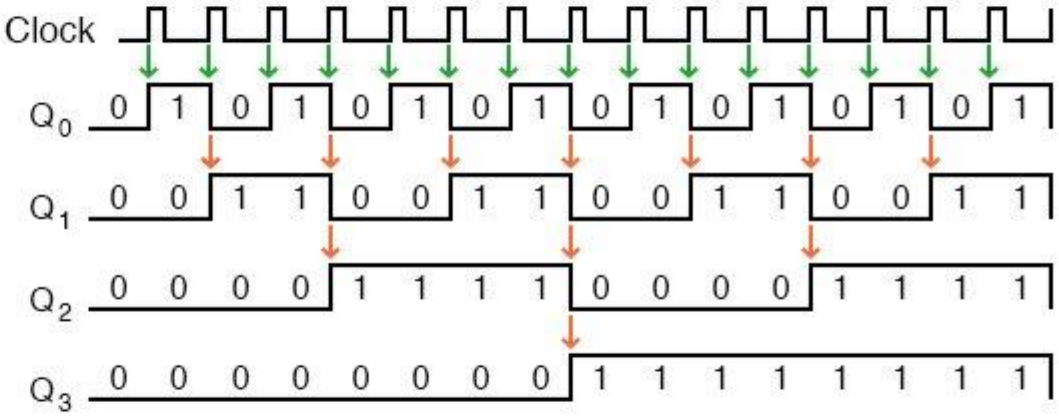


If we used flip-flops with negative-edge triggering (bubble symbols on the clock inputs), we could simply connect the clock input of each flip-flop to the Q output of the flip-flop before it, so that when the bit before it changes from a 1 to a 0, the “falling edge” of that signal would “clock” the next flip-flop to toggle the next bit.

Four-bit "Up" Counter



This circuit would yield the following output waveforms, when "clocked" by a repetitive source of pulses from an oscillator:



The first flip-flop (the one with the Q0 output), has a positive-edge triggered clock input, so it toggles with each rising edge of the clock signal.

Notice how the clock signal in this example has a duty cycle less than 50%.

The signal is shown in such a way as to demonstrate that the clock signal need not be symmetrical to obtain reliable, "clean" output bits in our four-bit binary sequence.

Using one J-K flip-flop for each output bit, however, relieves us of the necessity of having a symmetrical clock signal, allowing the use of practically any variety of high/low waveform to increment the count sequence.

As indicated by all the other arrows in the pulse diagram, each succeeding output bit is toggled by the action of the preceding bit transitioning from “high” (1) to “low” (0).

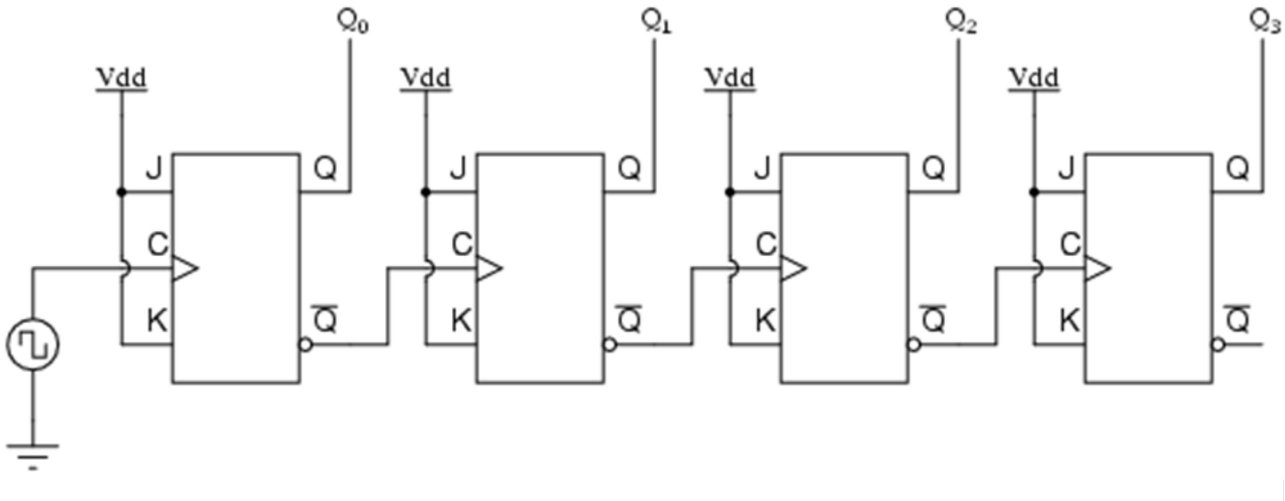
This is the pattern necessary to generate an “up” count sequence.

A less obvious solution for generating an “up” sequence using positive-edge triggered flip-flops is to “clock” each flip-flop using the Q’ output of the preceding flip-flop rather than the Q output.

Since the Q’ output will always be the exact opposite state of the Q output on a J-K flip-flop (no invalid states with this type of flip-flop), a high-to-low transition on the Q output will be accompanied by a low-to-high transition on the Q’ output.

In other words, each time the Q output of a flip-flop transitions from 1 to 0, the Q’ output of the same flip-flop will transition from 0 to 1, providing the positive-going clock pulse we would need to toggle a positive-edge triggered flip-flop at the right moment:

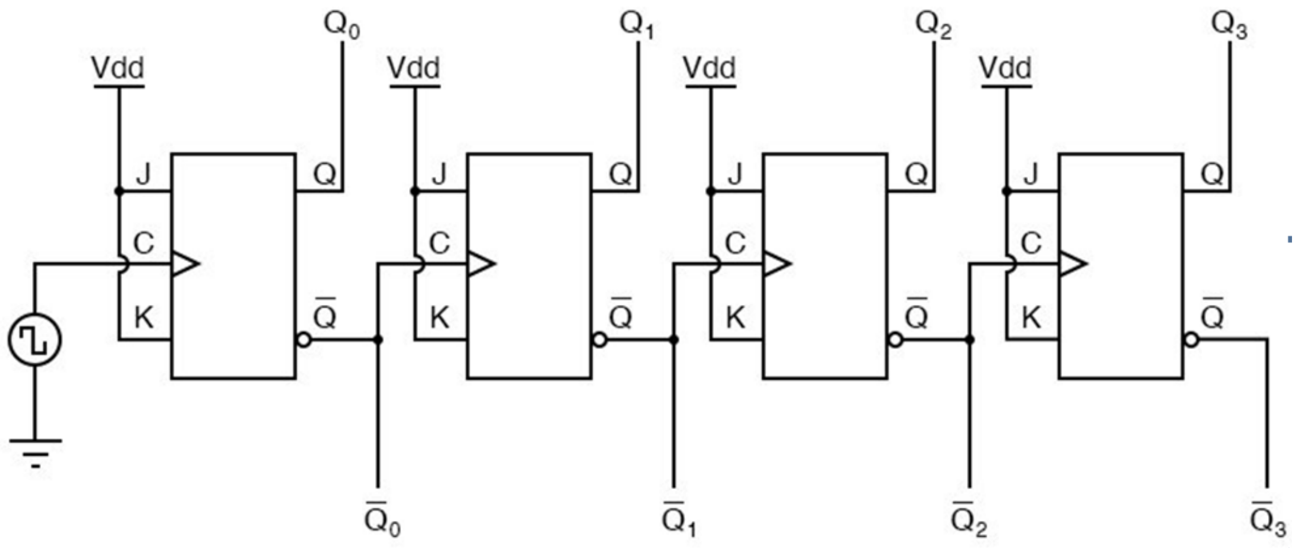
Alternative Four-bit “Up” Counter

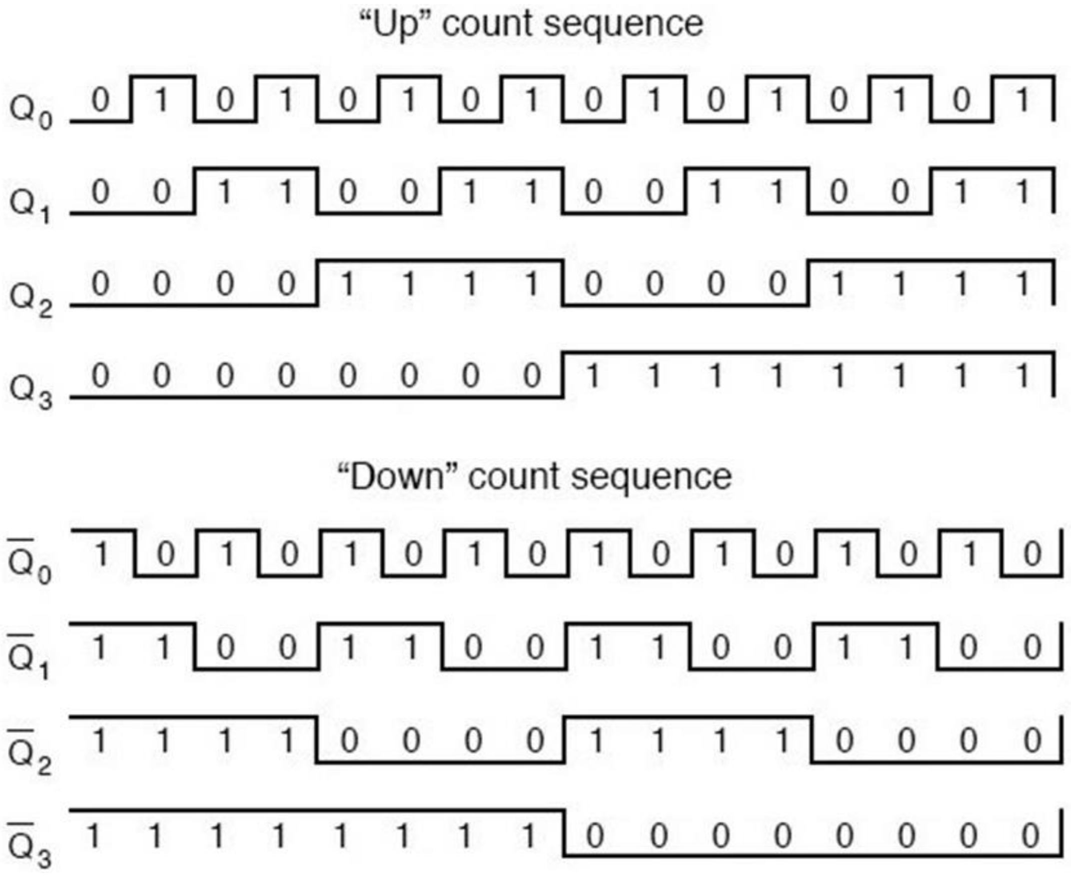


One way we could expand the capabilities of either of these two counter circuits is to regard the Q’ outputs as another set of four binary bits.

If we examine the pulse diagram for such a circuit, we see that the Q’ outputs generate a *down*-counting sequence, while the Q outputs generate an *up*-counting sequence:

Simultaneous “Up” and “Down” Counter





Unfortunately, all of the counter circuits shown thus far share a common problem: the *ripple* effect.

This effect is seen in certain types of binary adder and data conversion circuits, and is due to accumulative propagation delays between cascaded gates.

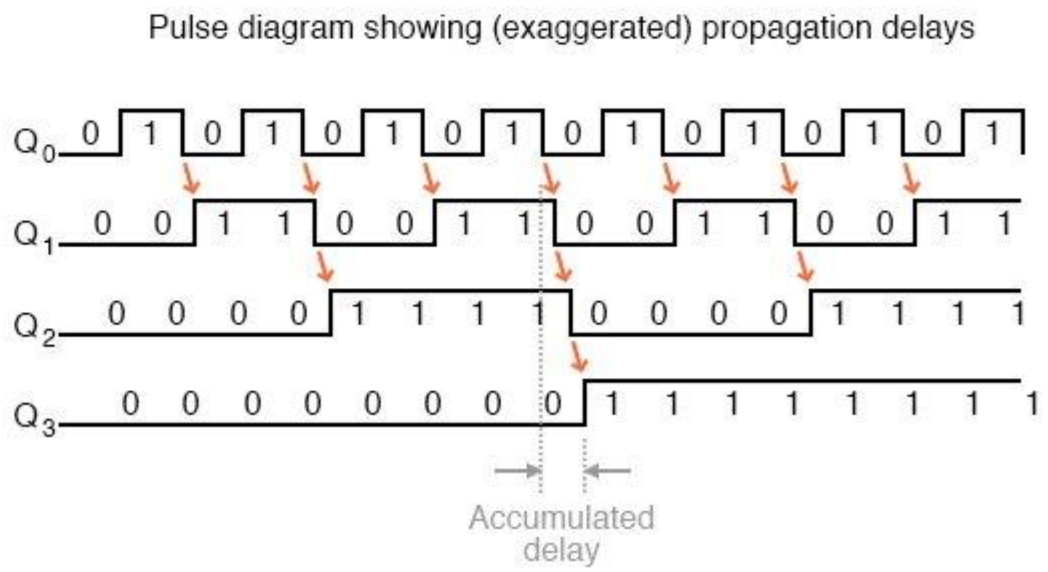
When the Q output of a flip-flop transitions from 1 to 0, it commands the next flip-flop to toggle.

If the next flip-flop toggle is a transition from 1 to 0, it will command the flip-flop after it to toggle as well, and so on.

However, since there is always some small amount of propagation delay between the command to toggle (the clock pulse) and the actual toggle response (Q and Q’ outputs changing states), any subsequent flip-flops to be toggled will toggle some time *after* the first flip-flop has toggled.

Thus, when multiple bits toggle in a binary count sequence, they will not all toggle at exactly the same time:

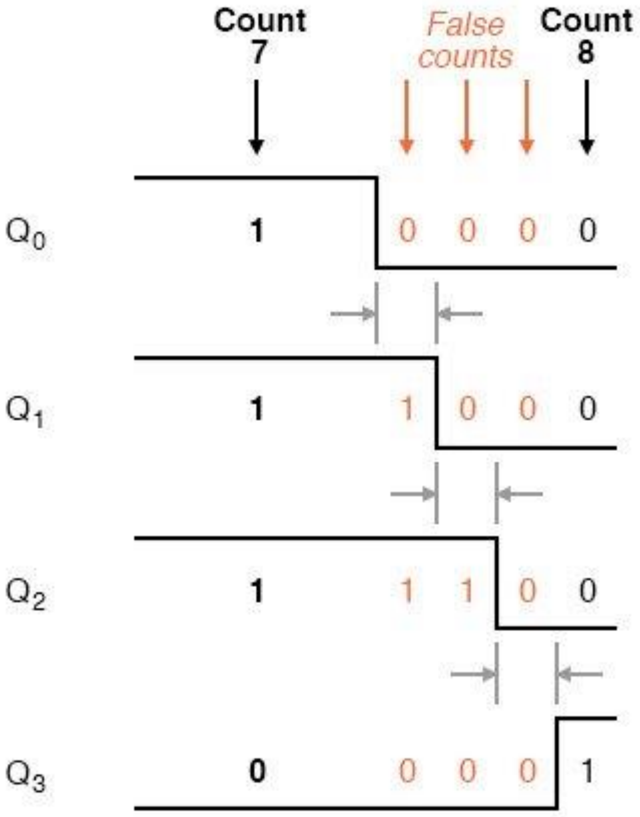
Disadvantage of Asynchronous Counter Circuit: Propagation Delay


Pulse diagram showing (exaggerated) propagation delays
Q0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Q1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Q2 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
Q3 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
Accumulated delay

As you can see, the more bits that toggle with a given clock pulse, the more severe the accumulated delay time from LSB to MSB.

When a clock pulse occurs at such a transition point (say, on the transition from 0111 to 1000), the output bits will “ripple” in sequence from LSB to MSB, as each succeeding bit toggles and commands the next bit to toggle as well, with a small amount of propagation delay between each bit toggle.

If we take a close-up look at this effect during the transition from 0111 to 1000, we can see that there will be false output counts generated in the brief time period that the “ripple” effect takes place:



Instead of cleanly transitioning from a “0111” output to a “1000” output, the counter circuit will very quickly ripple from 0111 to 0110 to 0100 to 0000 to 1000, or from 7 to 6 to 4 to 0 and then to 8. This behavior earns the counter circuit the name of *ripple* counter, or *asynchronous* counter.

Strobe Signal Counter Circuit

In many applications, this effect is tolerable, since the ripple happens very, very quickly (the width of the delays has been exaggerated here as an aid to understanding the effects).

If all we wanted to do was drive a set of light-emitting diodes (LEDs) with the counter’s outputs, for example, this brief ripple would be of no consequence at all.

However, if we wished to use this counter to drive the “select” inputs of a multiplexer, index a memory pointer in a microprocessor (computer) circuit, or perform some other task where false outputs could cause spurious errors, it would not be acceptable.

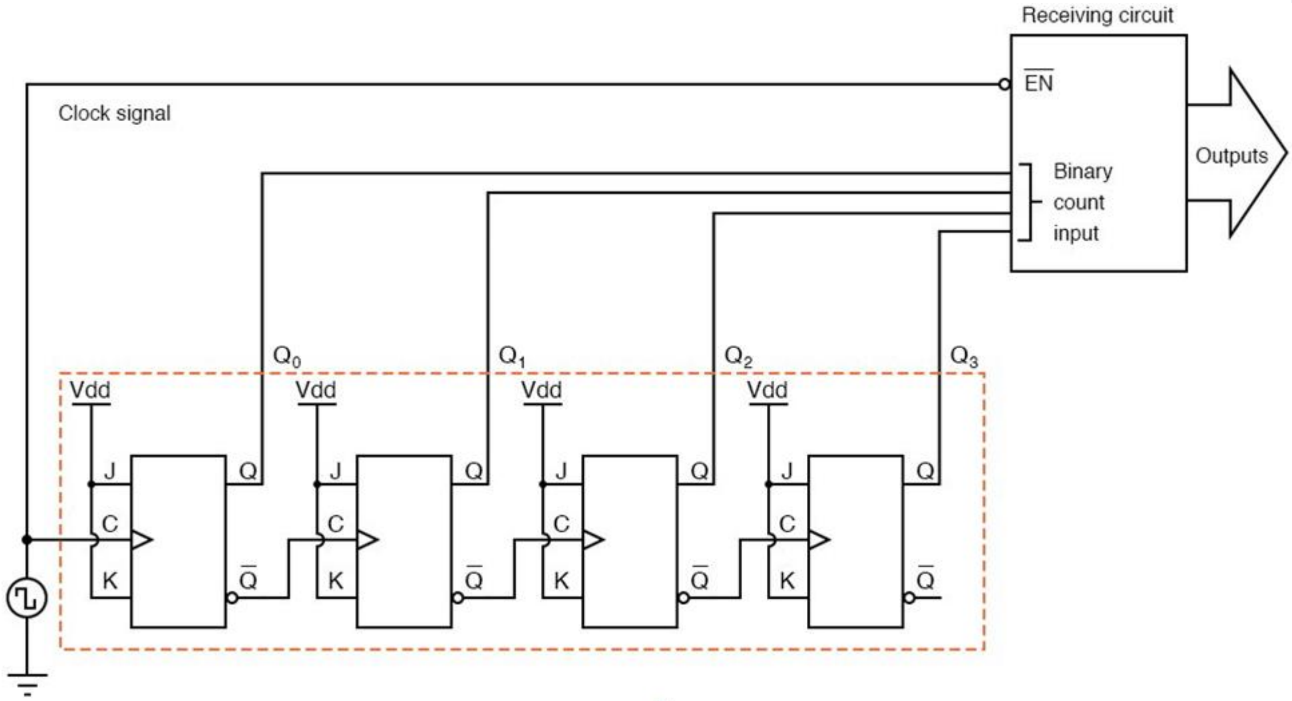
There is a way to use this type of counter circuit in applications sensitive to false, ripple-generated outputs, and it involves a principle known as *strobing*.

Most decoder and multiplexer circuits are equipped with at least one input called the “enable.”

The output(s) of such a circuit will be active only when the enable input is made active.

We can use this enable input to *strobe* the circuit receiving the ripple counter’s output so that it is disabled (and thus not responding to the counter output) during the brief period of time in which the counter outputs might be rippling, and enabled only when sufficient time has passed since the last clock pulse that all rippling will have ceased.

In most cases, the strobing signal can be the same clock pulse that drives the counter circuit:



With an active-low Enable input, the receiving circuit will respond to the binary count of the four-bit counter circuit only when the clock signal is “low.”

As soon as the clock pulse goes “high,” the receiving circuit stops responding to the counter circuit’s output.

Since the counter circuit is positive-edge triggered (as determined by the *first* flip-flop clock input), all the counting action takes place on the low-to-high transition of the clock signal, meaning that the receiving circuit will become disabled just before any toggling occurs on the counter circuit’s four output bits.

The receiving circuit will not become enabled until the clock signal returns to a low state, which should be a long enough time *after* all rippling has ceased to be “safe” to allow the new count to have effect on the receiving circuit.

The crucial parameter here is the clock signal’s “high” time: it must be at least as long as the maximum expected ripple period of the counter circuit.

If not, the clock signal will prematurely enable the receiving circuit, while some rippling is still taking place.

Disadvantage of Asynchronous Counter Circuit: Limited Speed

Another disadvantage of the asynchronous, or ripple, counter circuit is limited speed.

While all gate circuits are limited in terms of maximum signal frequency, the design of asynchronous counter circuits compounds this problem by making propagation delays additive.

Thus, even if strobing is used in the receiving circuit, an asynchronous counter circuit cannot be clocked at any frequency higher than that which allows the greatest possible accumulated propagation delay to elapse well before the next pulse.

The solution to this problem is a counter circuit that avoids ripple altogether.

Such a counter circuit would eliminate the need to design a “strobing” feature into whatever digital circuits use the counter output as an input, and would also enjoy a much greater operating speed than its asynchronous equivalent.

REVIEW:

- An “up” counter may be made by connecting the clock inputs of positive-edge triggered J-K flip-flops to the Q’ outputs of the preceding flip-flops. Another way is to use negative-edge triggered flip-flops, connecting the clock inputs to the Q outputs of the preceding flip-flops. In either case, the J and K inputs of all flip-flops are connected to V_{cc} or V_{dd} so as to always be “high.”
- Counter circuits made from cascaded J-K flip-flops where each clock input receives its pulses from the output of the previous flip-flop invariably exhibit a *ripple effect*, where false output counts are generated between some steps of

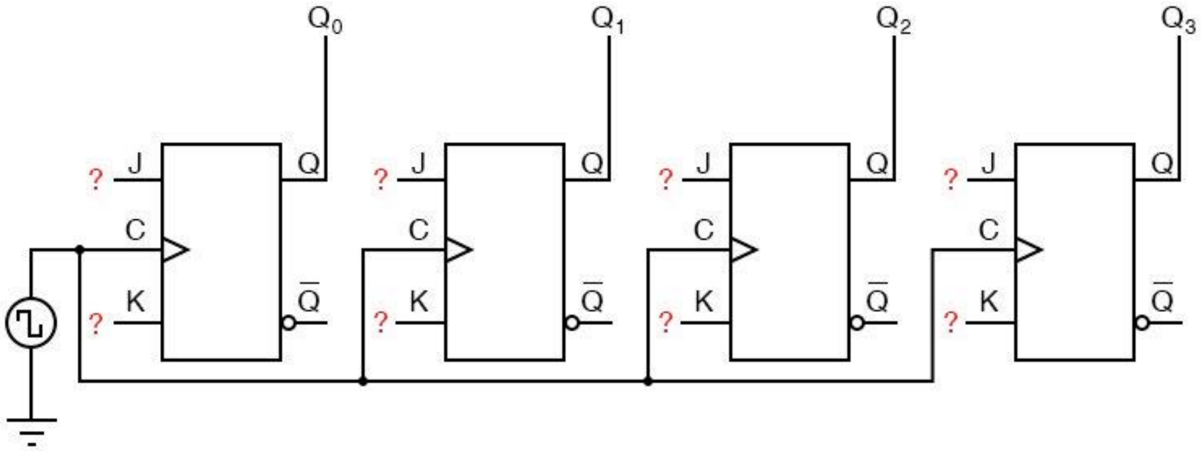
the count sequence. These types of counter circuits are called *asynchronous counters*, or *ripple counters*.

- *Strobing* is a technique applied to circuits receiving the output of an asynchronous (ripple) counter, so that the false counts generated during the ripple time will have no ill effect. Essentially, the *enable* input of such a circuit is connected to the counter's clock pulse in such a way that it is enabled only when the counter outputs are not changing, and will be disabled during those periods of changing counter outputs where ripple occurs.

Synchronous Counters

A *synchronous counter*, in contrast to an *asynchronous counter*, is one whose output bits change state simultaneously, with no ripple.

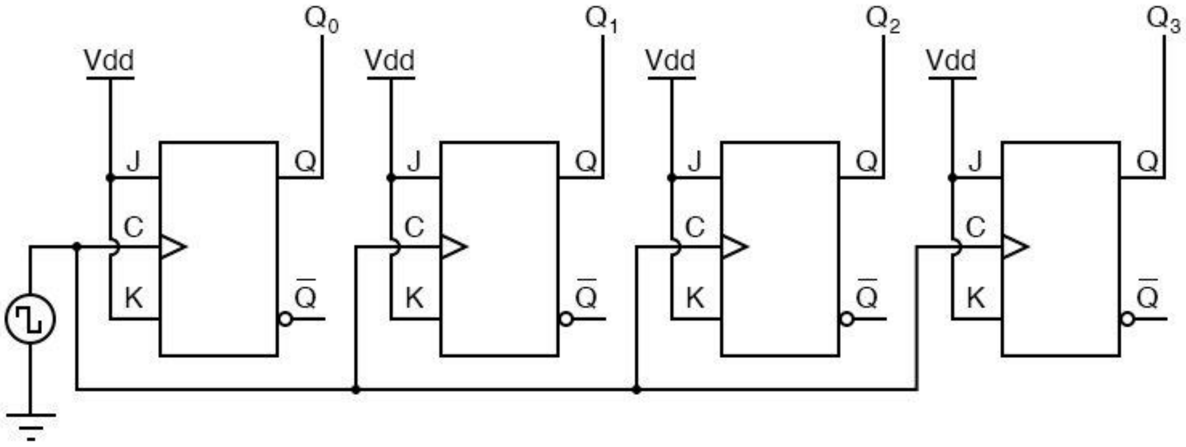
The only way we can build such a counter circuit from J-K flip-flops is to connect all the clock inputs together, so that each and every flip-flop receives the exact same clock pulse at the exact same time:



Now, the question is, what do we do with the J and K inputs? We know that we still have to maintain the same divide-by-two frequency pattern in order to count in a binary sequence, and that this pattern is best achieved utilizing the "toggle" mode of the flip-flop, so the fact that the J and K inputs must both be (at times) "high" is clear.

However, if we simply connect all the J and K inputs to the positive rail of the power supply as we did in the asynchronous circuit, this would clearly not work because all the flip-flops would toggle at the same time: with each and every clock pulse!

This circuit will not function as a counter!



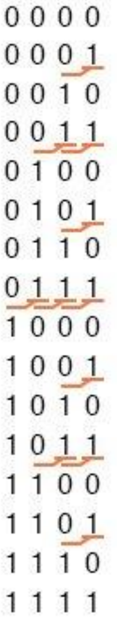
Let’s examine the four-bit binary counting sequence again, and see if there are any other patterns that predict the toggling of a bit.

Asynchronous counter circuit design is based on the fact that each bit toggle happens at the same time that the preceding bit toggles from a “high” to a “low” (from 1 to 0).

Since we cannot clock the toggling of a bit based on the toggling of a previous bit in a synchronous counter circuit (to do so would create a ripple effect) we must find some other pattern in the counting sequence that can be used to trigger a bit toggle.

Examining the four-bit binary count sequence, another predictive pattern can be seen.

Notice that just before a bit toggles, all preceding bits are “high:”



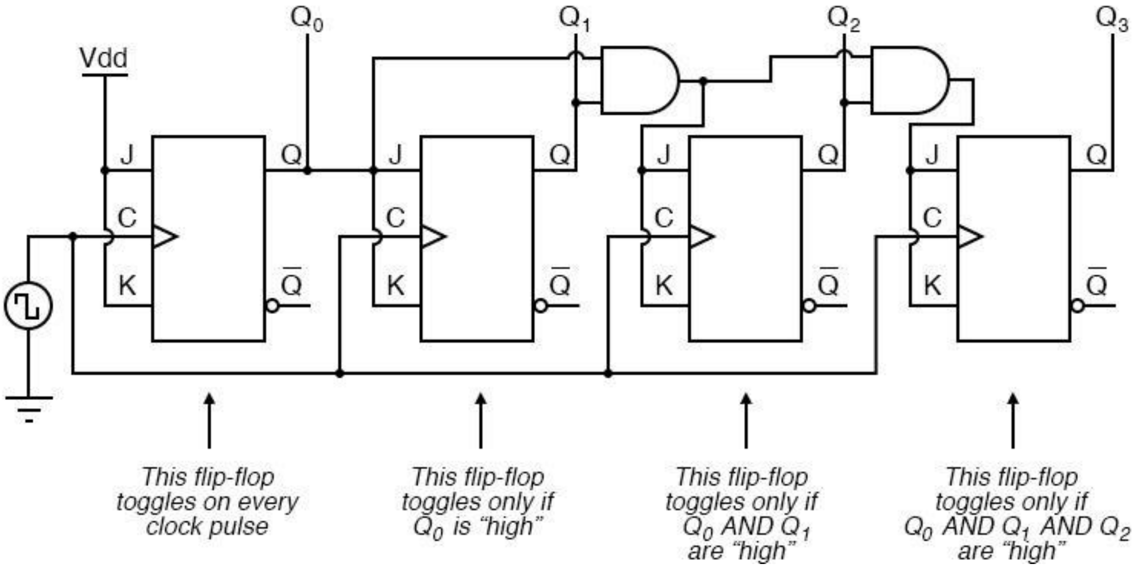
```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

This pattern is also something we can exploit in designing a counter circuit.

Synchronous "Up" Counter

If we enable each J-K flip-flop to toggle based on whether or not all preceding flip-flop outputs (Q) are "high," we can obtain the same counting sequence as the asynchronous circuit without the ripple effect, since each flip-flop in this circuit will be clocked at exactly the same time:

A four-bit synchronous "up" counter


Vdd
Q_0
Q_1
Q_2
Q_3
J
C
K
Q
$\overline{Q}$
This flip-flop toggles on every clock pulse
This flip-flop toggles only if Q_0 is "high"
This flip-flop toggles only if Q_0 AND Q_1 are "high"
This flip-flop toggles only if Q_0 AND Q_1 AND Q_2 are "high"

The result is a four-bit *synchronous* “up” counter. Each of the higher-order flip-flops are made ready to toggle (both J and K inputs “high”) if the Q outputs of all previous flip-flops are “high.”

Otherwise, the J and K inputs for that flip-flop will both be “low,” placing it into the “latch” mode where it will maintain its present output state at the next clock pulse.

Since the first (LSB) flip-flop needs to toggle at every clock pulse, its J and K inputs are connected to Vcc or Vdd, where they will be “high” all the time.

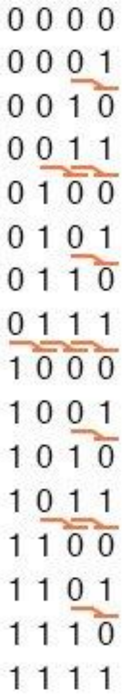
The next flip-flop need only “recognize” that the first flip-flop’s Q output is high to be made ready to toggle, so no AND gate is needed.

However, the remaining flip-flops should be made ready to toggle only when all lower-order output bits are “high,” thus the need for AND gates.

Synchronous “Down” Counter

To make a synchronous “down” counter, we need to build the circuit to recognize the appropriate bit patterns predicting each toggle state while counting down.

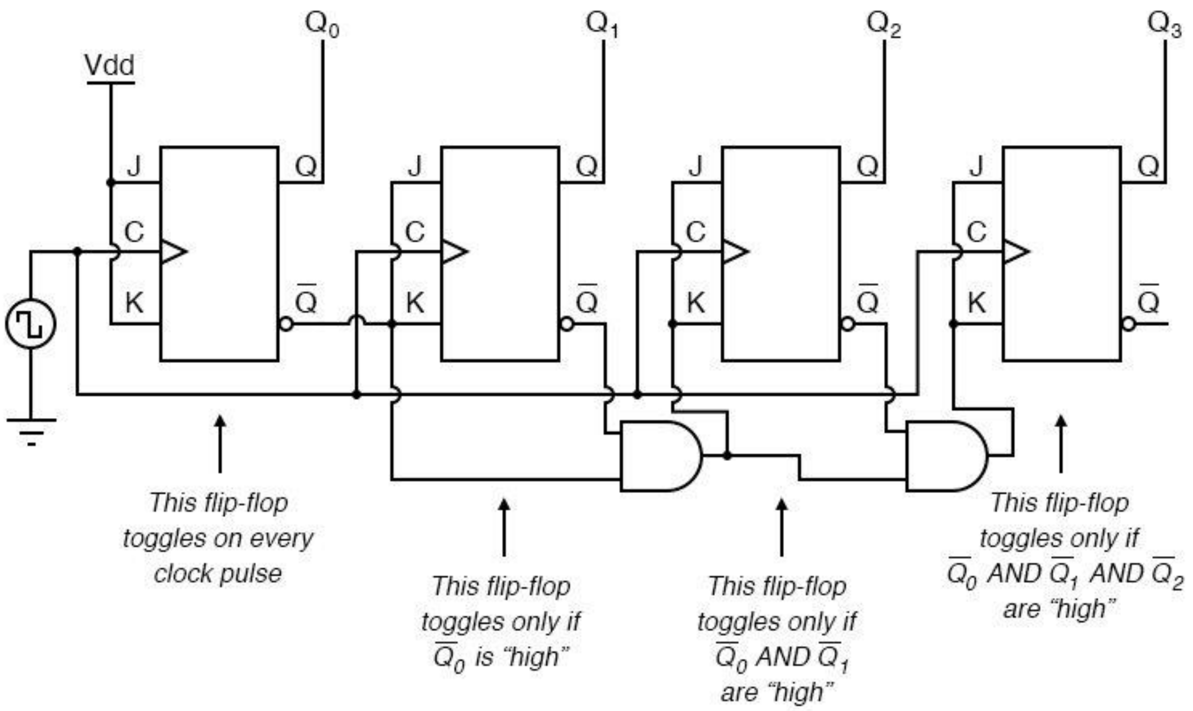
Not surprisingly, when we examine the four-bit binary count sequence, we see that all preceding bits are “low” prior to a toggle (following the sequence from bottom to top):



```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Since each J-K flip-flop comes equipped with a Q’ output as well as a Q output, we can use the Q’ outputs to enable the toggle mode on each succeeding flip-flop, being that each Q’ will be “high” every time that the respective Q is “low:”

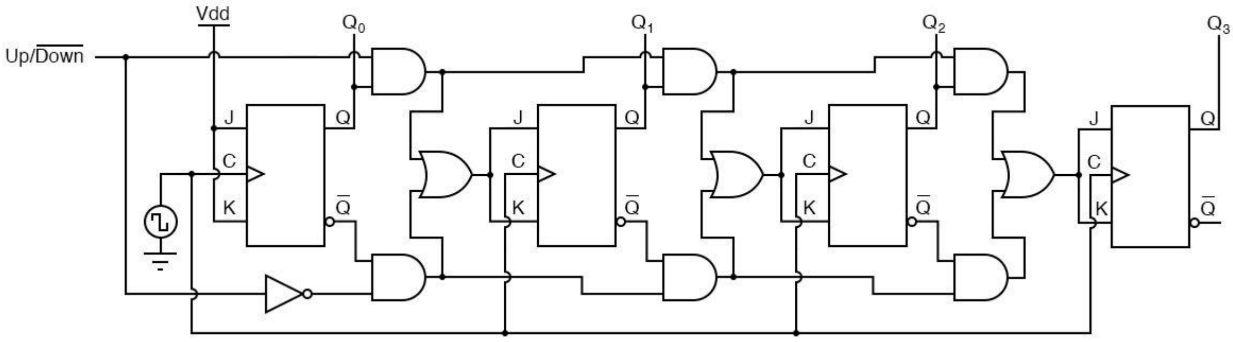
A four-bit synchronous “down” counter


Vdd
Q0
Q1
Q2
Q3
J
C
K
Q
$\overline{Q}$
This flip-flop toggles on every clock pulse
This flip-flop toggles only if $\overline{Q}_0$ is “high”
This flip-flop toggles only if $\overline{Q}_0$ AND $\overline{Q}_1$ are “high”
This flip-flop toggles only if $\overline{Q}_0$ AND $\overline{Q}_1$ AND $\overline{Q}_2$ are “high”

Counter Circuit with Selectable “up” and “down” Count Modes

Taking this idea one step further, we can build a counter circuit with selectable between “up” and “down” count modes by having dual lines of AND gates detecting the appropriate bit conditions for an “up” and a “down” counting sequence, respectively, then use OR gates to combine the AND gate outputs to the J and K inputs of each succeeding flip-flop:

A four-bit synchronous “up/down” counter



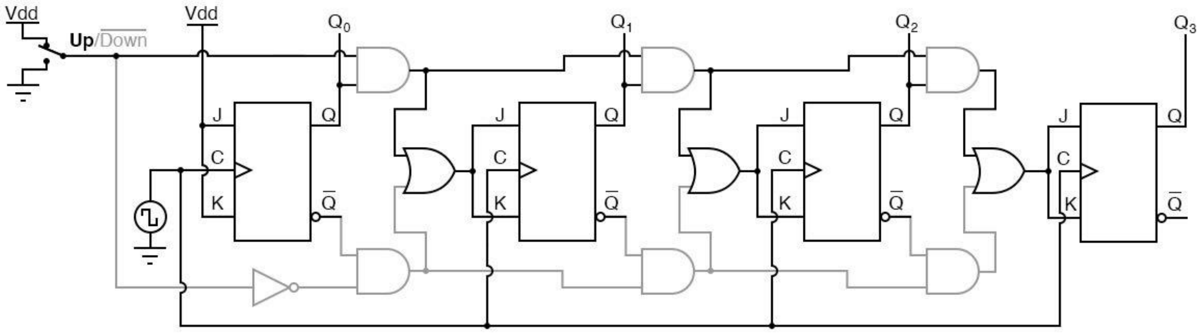
This circuit isn’t as complex as it might first appear. The Up/Down control input line simply enables either the upper string or lower string of AND gates to pass the Q/Q’ outputs to the succeeding stages of flip-flops.

If the Up/Down control line is “high,” the top AND gates become enabled, and the circuit functions exactly the same as the first (“up”) synchronous counter circuit shown above.

If the Up/Down control line is made “low,” the bottom AND gates become enabled, and the circuit functions identically to the second (“down” counter) circuit shown above.

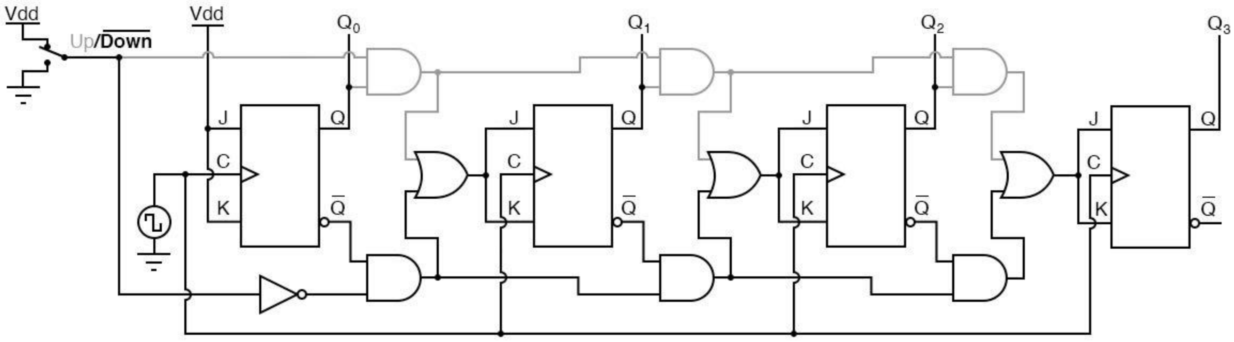
To illustrate, here is a diagram showing the circuit in the “up” counting mode (all disabled circuitry shown in grey rather than black):

Counter in “up” counting mode

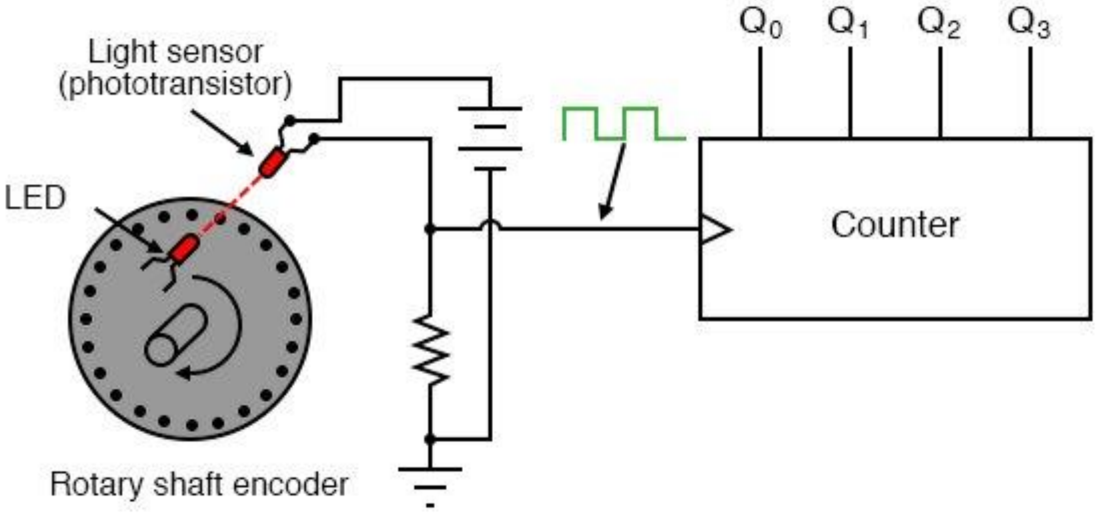


Here, shown in the “down” counting mode, with the same grey coloring representing disabled circuitry:

Counter in “down” counting mode



Up/down counter circuits are very useful devices. A common application is wheel motion control, where devices called *rotary shaft encoders* convert mechanical rotation into a series of electrical pulses, these pulses “clocking” a counter circuit to track total motion:



As the wheel moves, it turns the encoder shaft, making and breaking the light beam between LED and phototransistor, thereby generating clock pulses to increment the counter circuit.

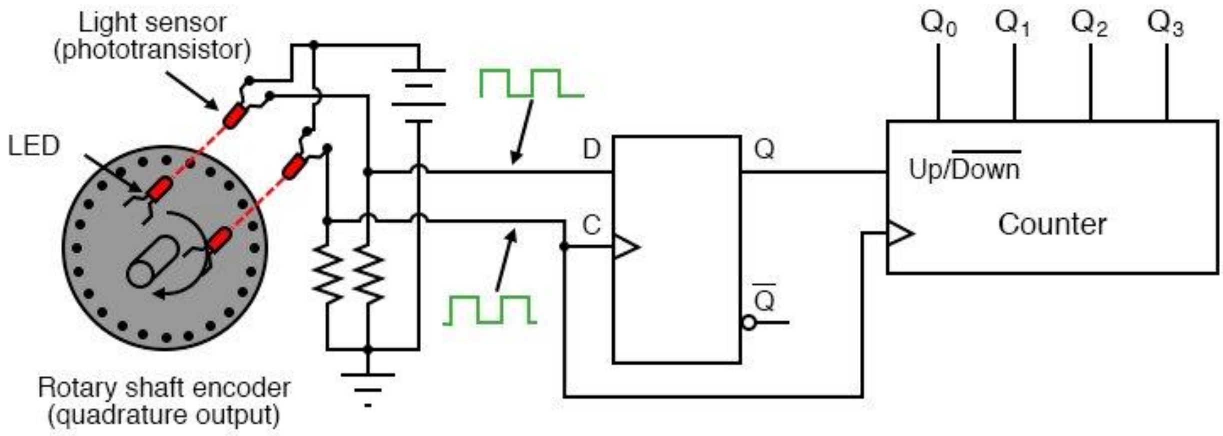
Thus, the counter integrates, or accumulates, total motion of the shaft, serving as an electronic indication of how far the wheel has moved.

If all we care about is tracking total motion, and do not care to account for changes in the direction of motion, this arrangement will suffice.

However, if we wish the counter to *increment* with one direction of motion and *decrement* with the reverse direction of motion, we must use an up/down counter, and an encoder/decoding circuit having the ability to discriminate between different directions.

If we re-design the encoder to have two sets of LED/phototransistor pairs, those pairs aligned such that their square-wave output signals are 90o out of phase with each other, we have what is known as a *quadrature* output encoder (the word “quadrature” simply refers to a 90o angular separation).

A phase detection circuit may be made from a D-type flip-flop, to distinguish a clockwise pulse sequence from a counter-clockwise pulse sequence:



When the encoder rotates clockwise, the “D” input signal square-wave will lead the “C” input square-wave, meaning that the “D” input will already be “high” when the “C” transitions from “low” to “high,” thus *setting* the D-type flip-flop (making the Q output “high”) with every clock pulse.

A “high” Q output places the counter into the “Up” count mode, and any clock pulses received by the clock from the encoder (from either LED) will increment it.

Conversely, when the encoder reverses rotation, the “D” input will lag behind the “C” input waveform, meaning that it will be “low” when the “C” waveform transitions from “low” to “high,” forcing the D-type flip-flop into the reset state (making the Q output “low”) with every clock pulse.

This “low” signal commands the counter circuit to decrement with every clock pulse from the encoder.

This circuit, or something very much like it, is at the heart of every position-measuring circuit based on a pulse encoder sensor.

Such applications are very common in robotics and other applications involving the measurement of reversible, mechanical motion.

Introduction to Shift Registers

Shift registers, like counters, are a form of *sequential* logic.

Sequential logic, unlike combinational logic is not only affected by the present inputs, but also, by the prior history.

In other words, sequential logic remembers past events.

Shift registers produce a discrete delay of a digital signal or waveform.

A waveform synchronized to a *clock*, a repeating square wave, is delayed by "***n***" discrete clock times, where "***n***" is the number of shift register stages.

Thus, a four stage shift register delays "data in" by four clocks to "data out".

The stages in a shift register are *delay stages*, typically type "D" Flip-Flops or type "JK" Flip-flops.

Formerly, very long (several hundred stages) shift registers served as digital memory.

This obsolete application is reminiscent of the acoustic mercury delay lines used as early computer memory.

Serial data transmission, over a distance of meters to kilometers, uses shift registers to convert parallel data to serial form.

Serial data communications replaces many slow parallel data wires with a single serial high speed circuit.

Serial data over shorter distances of tens of centimeters, uses shift registers to get data into and out of microprocessors.

Numerous peripherals, including analog to digital converters, digital to analog converters, display drivers, and memory, use shift registers to reduce the amount of wiring in circuit boards.

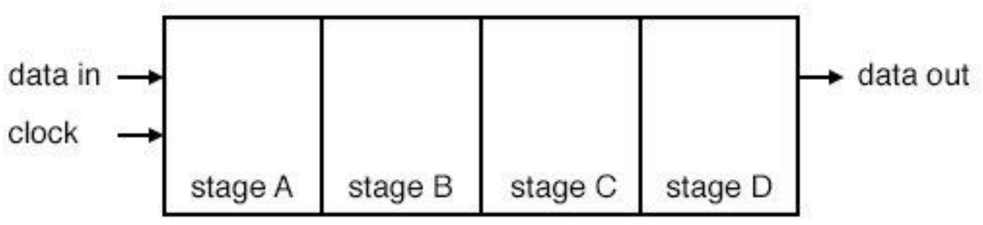
Some specialized counter circuits actually use shift registers to generate repeating waveforms.

Longer shift registers, with the help of feedback generate patterns so long that they look like random noise, *pseudo-noise*.

Basic shift registers are classified by structure according to the following types:

- Serial-in/serial-out
- Parallel-in/serial-out

- Serial-in/parallel-out
- Universal parallel-in/parallel-out
- Ring counter



Serial-in, serial-out shift register with 4-stages

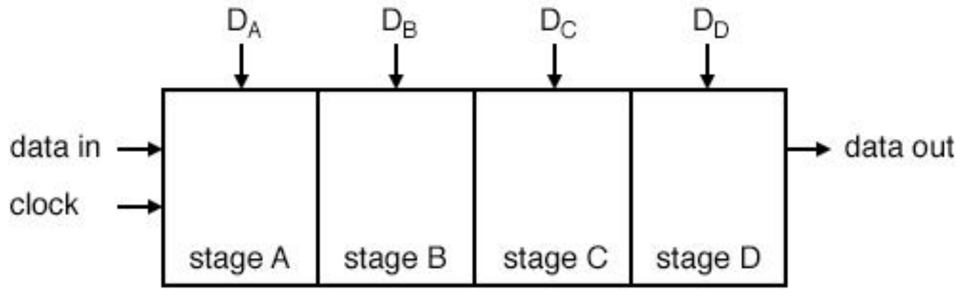
Above we see a block diagram of a serial-in/serial-out shift register, which is 4-stages long.

Data at the input will be delayed by four clock periods from the input to the output of the shift register.

Data at “data in”, above, will be present at the Stage A output after the first clock pulse. After the second pulse stage A data is transfered to stage B output, and “data in” is transfered to stage A output. After the third clock, stage C is replaced by stage B; stage B is replaced by stage A; and stage A is replaced by “data in”.

After the fourth clock, the data originally present at “data in” is at stage D, “output”.

The “first in” data is “first out” as it is shifted from “data in” to “data out”.



Parallel-in, serial-out shift register with 4-stages

Data is loaded into all stages at once of a parallel-in/serial-out shift register.

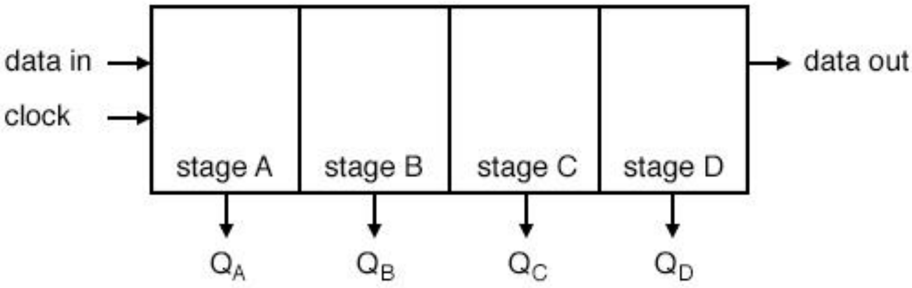
The data is then shifted out via “data out” by clock pulses. Since a 4- stage shift register is shown above, four clock pulses are required to shift out all of the data.

In the diagram above, stage D data will be present at the “data out” up until the first clock pulse; stage C data will be present at “data out” between the first clock and the second clock pulse; stage B data will be present between the second clock and the third clock; and stage A data will be present between the third and the fourth clock.

After the fourth clock pulse and thereafter, successive bits of “data in” should appear at “data out” of the shift register after a delay of four clock pulses.

If four switches were connected to DA through DD, the status could be read into a microprocessor using only one data pin and a clock pin.

Since adding more switches would require no additional pins, this approach looks attractive for many inputs.

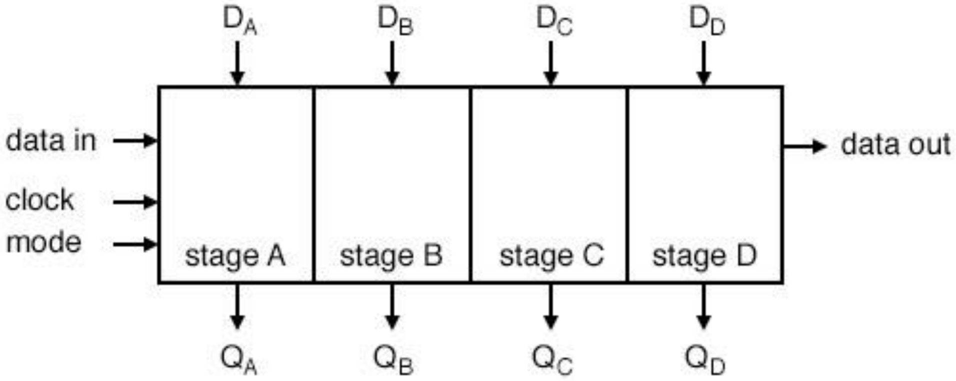


Serial-in, serial-out shift register with 4-stages

Above, four data bits will be shifted in from “data in” by four clock pulses and be available at QA through QD for driving external circuitry such as LEDs, lamps, relay drivers, and horns. After the first clock, the data at “data in” appears at QA.

After the second clock, The old QA data appears at QB; QA receives next data from “data in”. After the third clock, QB data is at QC.

After the fourth clock, QC data is at QD. This stage contains the data first present at “data in”. The shift register should now contain four data bits.



Parallel-in, parallel-out shift register with 4 stages

A parallel-in/parallel-out shift register combines the function of the parallel-in, serial-out shift register with the function of the serial-in, parallel-out shift register to yield the universal shift register.

The “do anything” shifter comes at a price– the increased number of I/O (Input/Output) pins may reduce the number of stages which can be packaged.

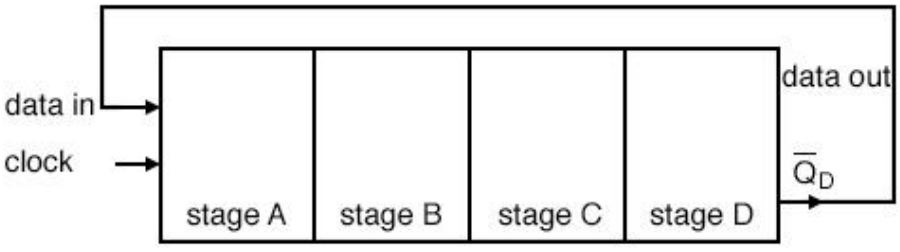
Data presented at DA through DD is parallel loaded into the registers.

This data at QA through QD may be shifted by the number of pulses presented at the clock input.

The shifted data is available at QA through QD.

The “mode” input, which may be more than one input, controls parallel loading of data from DA through DD, shifting of data, and the direction of shifting.

There are shift registers which will shift data either left or right.



Ring Counter. shift register output fed back to input

If the serial output of a shift register is connected to the serial input, data can be perpetually shifted around the ring as long as clock pulses are present.

If the output is inverted before being fed back as shown above, we do not have to worry about loading the initial data into the “ring counter”.

Shift Registers: Serial-in, Serial-out

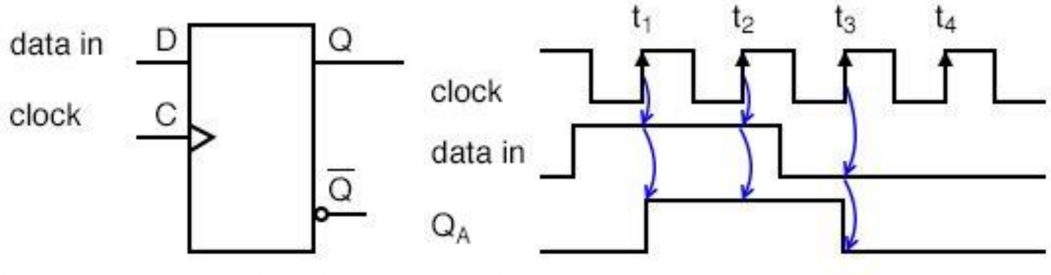
Serial-in, serial-out shift registers delay data by one clock time for each stage.

They will store a bit of data for each register. A serial-in, serial-out shift register may be one to 64 bits in length, longer if registers or packages are cascaded.

Below is a single stage shift register receiving data which is not synchronized to the register clock.

The "data in" at the D pin of the type D FF (Flip-Flop) does not change levels when the clock changes for low to high.

We may want to synchronize the data to a system-wide clock in a circuit board to improve the reliability of a digital logic circuit.



Data present at clock time is transferred from **D** to **Q**.

The obvious point (as compared to the figure below) illustrated above is that whatever "data in" is present at the D pin of a type D FF is transfered from D to output Q at clock time.

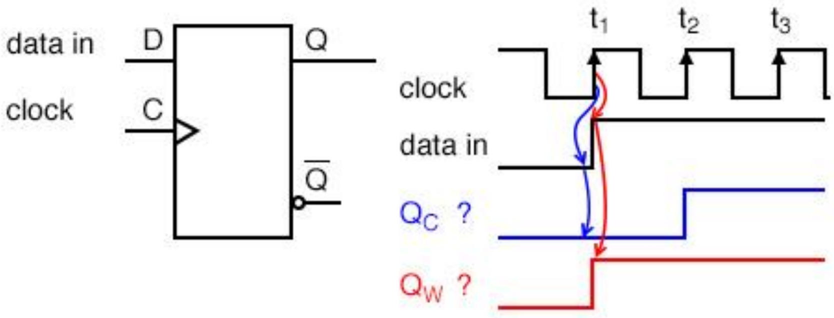
Since our example shift register uses positive edge sensitive storage elements, the output Q follows the D input when the clock transitions from low to high as shown by the up arrows on the diagram above.

There is no doubt what logic level is present at clock time because the data is stable well before and after the clock edge.

This is seldom the case in multi-stage shift registers. But, this was an easy example to start with. We are only concerned with the positive, low to high, clock edge.

The falling edge can be ignored. It is very easy to see Q follow D at clock time above.

Compare this to the diagram below where the “data in” appears to change with the positive clock edge.



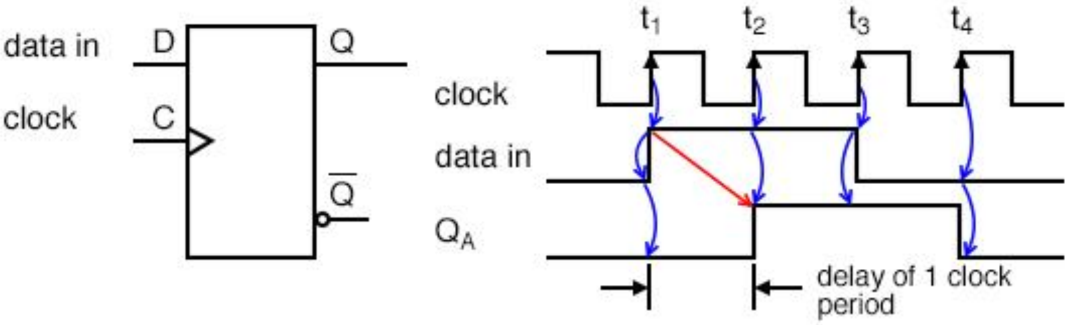
Does the clock t_1 see a 0 or a 1 at data in at D? Which output is correct, Q_C or Q_W?

Since “data in” appears to changes at clock time t1 above, what does the type D FF see at clock time?

The short oversimplified answer is that it sees the data that was present at D prior to the clock.

That is what is transferred to Q at clock time t1. The correct waveform is QC. At t1 Q goes to a zero if it is not already zero.

The D register does not see a one until time t2, at which time Q goes high.



Data present t_H before clock time at **D** is transferred to **Q**.

Since data, above, present at D is clocked to Q at clock time, and Q cannot change until the next clock time, the D FF delays data by one clock period, provided that the data is already synchronized to the clock. The QA waveform is the same as “data in” with a one clock period delay.

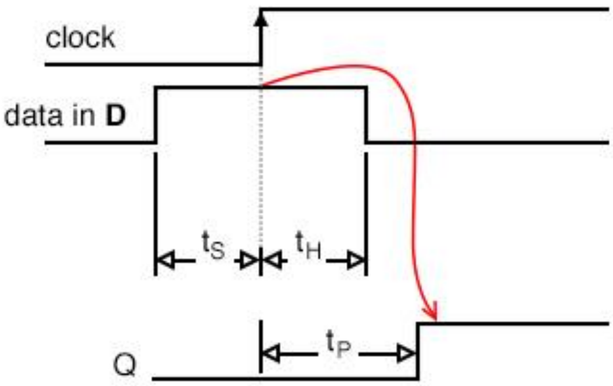
A more detailed look at what the input of the type D Flip-Flop sees at clock time follows.

Refer to the figure below. Since "data in" appears to changes at clock time (above), we need further information to determine what the D FF sees.

If the "data in" is from another shift register stage, another same type D FF, we can draw some conclusions based on *data sheet* information.

Manufacturers of digital logic make available information about their parts in data sheets, formerly only available in a collection called a data book.

Data books are still available; though, the manufacturer's web site is the modern source.



Data must be present (t_S) before the clock and after (t_H) the clock. Data is delayed from D to Q by propagation delay (t_P).

The following data was extracted from the CD4006b data sheet for operation at 5VDC, which serves as an example to illustrate timing.

- t_S=100ns
- t_H=60ns
- t_P=200-400ns typ/max

tS is the setup time, the time data must be present before clock time. In this case, data must be present at D 100ns prior to the clock.

Furthermore, the data must be held for hold time tH=60ns after clock time. These two conditions must be met to reliably clock data from D to Q of the Flip-Flop.

There is no problem meeting the setup time of 60ns as the data at D has been there for the whole previous clock period if it comes from another shift register stage.

For example, at a clock frequency of 1 Mhz, the clock period is 1000 µs, plenty of time.

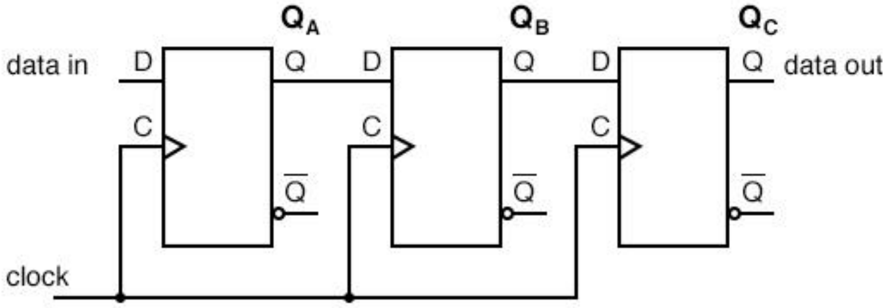
Data will actually be present for 1000µs prior to the clock, which is much greater than the minimum required tS of 60ns.

The hold time tH=60ns is met because D connected to Q of another stage cannot change any faster than the propagation delay of the previous stage tP=200ns.

Hold time is met as long as the propagation delay of the previous D FF is greater than the hold time.

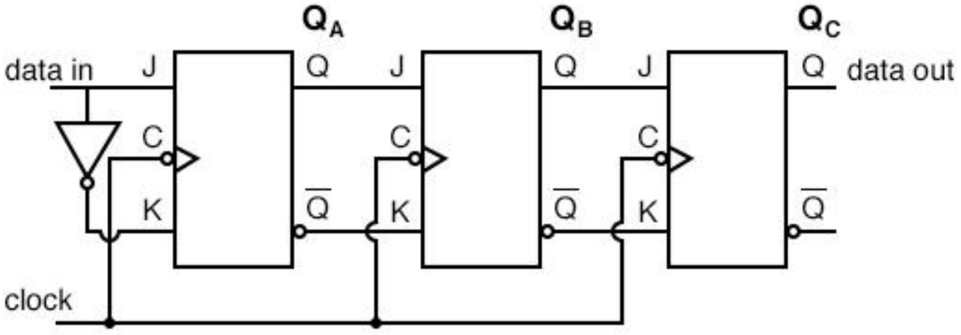
Data at D driven by another stage Q will not change any faster than 200ns for the CD4006b.

To summarize, output Q follows input D at nearly clock time if Flip-Flops are cascaded into a multi-stage shift register.



Serial-in, serial out shift register using type "**D**" storage elements

Three type D Flip-Flops are cascaded Q to D and the clocks paralleled to form a three-stage shift register above.



Serial-in, serial out shift register using type "**JK**" storage elements

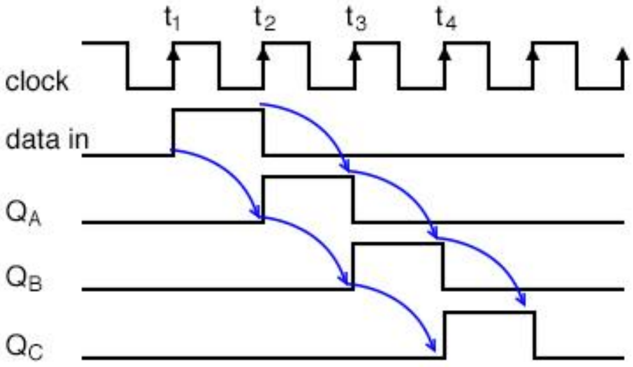
Type JK Flip Flopss cascaded Q to J, Q' to K with clocks in parallel to yield an alternate form of the shift register above.

A serial-in/serial-out shift register has a clock input, a data input, and a data output from the last stage.

In general, the other stage outputs are not available Otherwise, it would be a serial-in, parallel-out shift register.

The waveforms below are applicable to either one of the preceding two versions of the serial-in, serial-out shift register.

The three pairs of arrows show that a three-stage shift register temporarily stores 3-bits of data and delays it by three clock periods from input to output.



At clock time t_1 a "data in" of 0 is clocked from D to Q of all three stages. In particular, D of stage A sees a logic 0, which is clocked to Q_A where it remains until time t_2.

At clock time t_2 a "data in" of 1 is clocked from D to Q_A. At stages B and C, a 0, fed from preceding stages is clocked to Q_B and Q_C.

At clock time t_3 a "data in" of 0 is clocked from D to Q_A. Q_A goes low and stays low for the remaining clocks due to "data in" being 0. Q_B goes high at t_3 due to a 1 from the previous stage. Q_C is still low after t_3 due to a low from the previous stage.

Q_C finally goes high at clock t_4 due to the high fed to D from the previous stage Q_B. All earlier stages have 0s shifted into them. And, after the next clock pulse at t_5, all logic 1s will have been shifted out, replaced by 0s.

Serial-in/serial-out devices

We will take a closer look at the following parts available as integrated circuits.

For complete device data sheets follow the links.

CD4006b 18-bit serial-in/ serial-out shift register

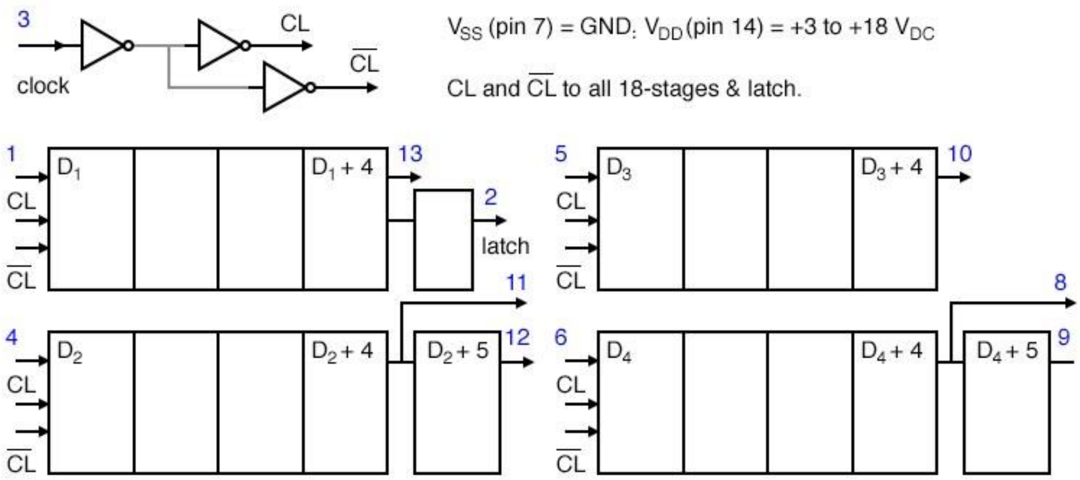
CD4031b 64-bit serial-in/ serial-out shift register

CD4517b dual 64-bit serial-in/ serial-out shift register

The following serial-in/ serial-out shift registers are 4000 series CMOS (Complementary Metal Oxide Semiconductor) family parts.

As such, They will accept a V_{DD}, positive power supply of 3-Volts to 15-Volts. The V_{SS} pin is grounded.

The maximum frequency of the shift clock, which varies with V_{DD}, is a few megahertz.

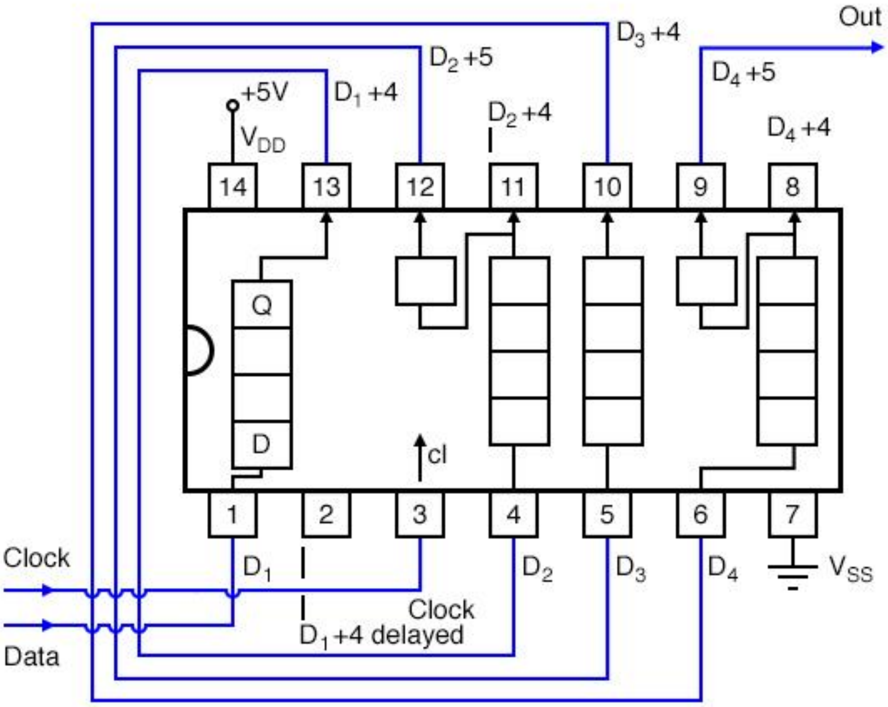


CD4006b Serial-in/ serial out shift register

The 18-bit CD4006b consists of two stages of 4-bits and two more stages of 5-bits with a an output tap at 4-bits.

Thus, the 5-bit stages could be used as 4-bit shift registers.

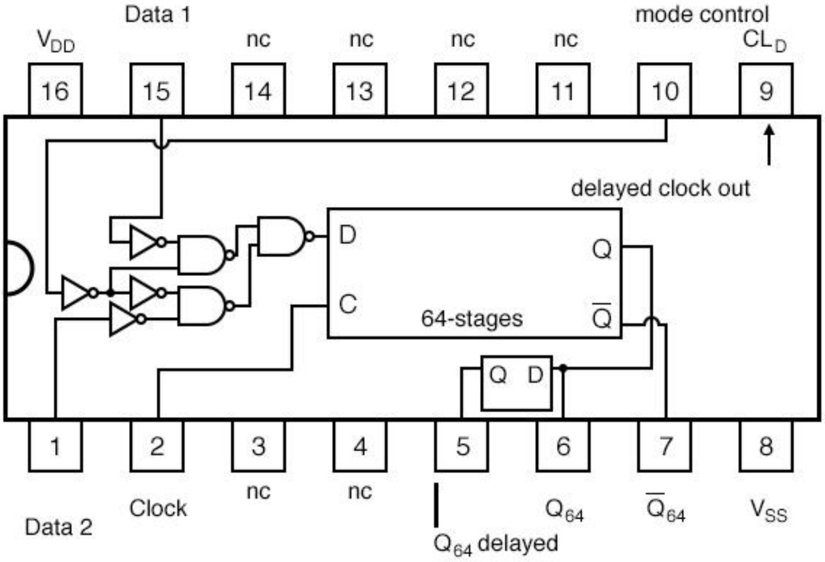
To get a full 18-bit shift register the output of one shift register must be cascaded to the input of another and so on until all stages create a single shift register as shown below.



A CD4031 64-bit serial-in/ serial-out shift register is shown below.

A number of pins are not connected (nc). Both Q and Q' are available from the 64th stage, actually Q_{64} and Q'_{64}.

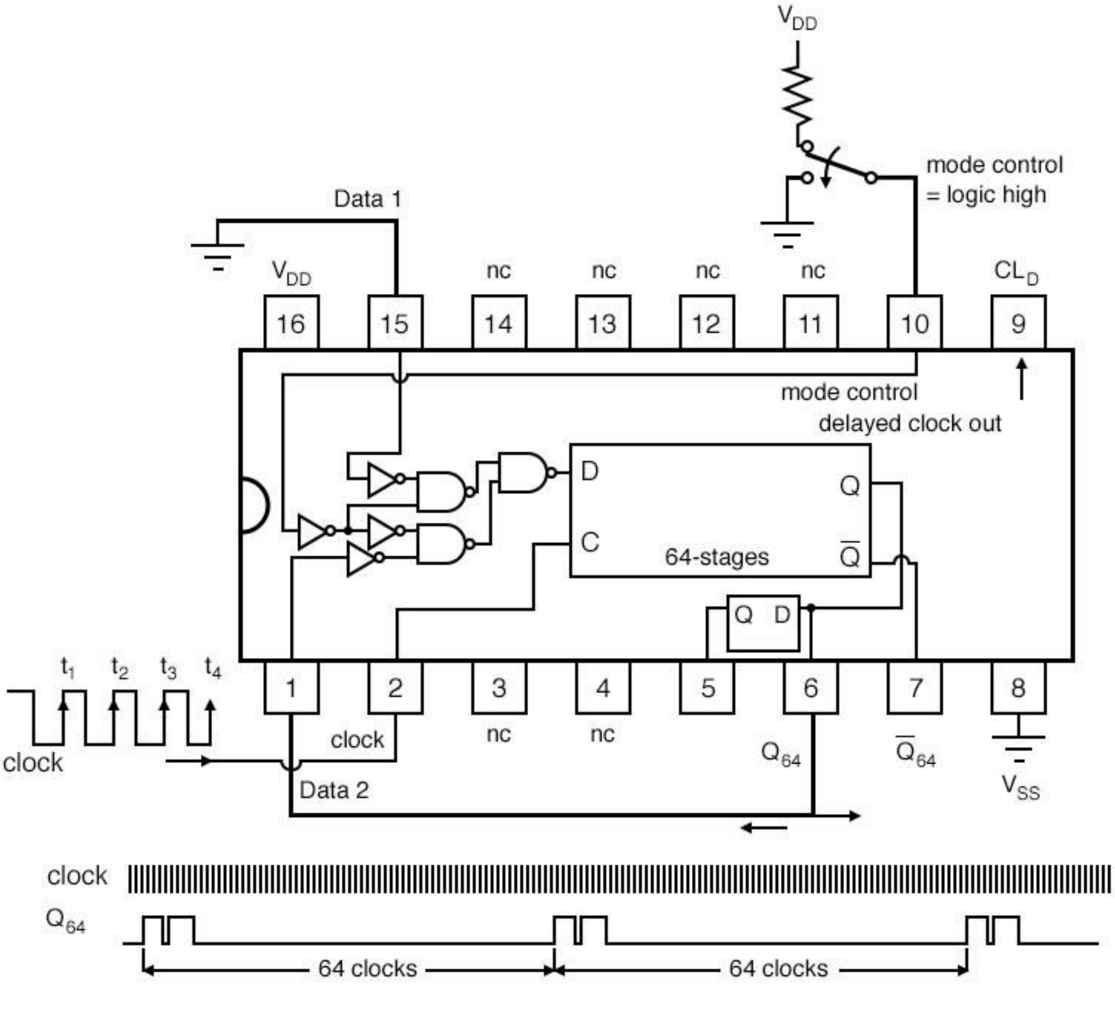
There is also a Q_{64} "delayed" from a half stage which is delayed by half a clock cycle. A major feature is a data selector which is at the data input to the shift register.



CD4031 64-bit serial in/ serial out shift register

The “mode control” selects between two inputs: data 1 and data 2. If “mode control” is high, data will be selected from “data 2” for input to the shift register.

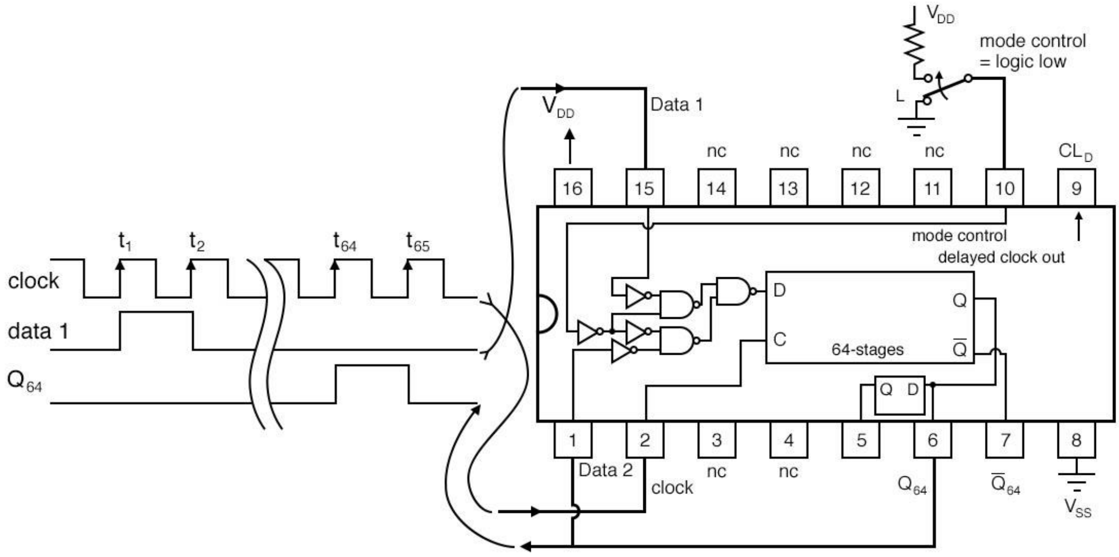
In the case of “mode control” being logic low, the “data 1” is selected. Examples of this are shown in the two figures below.



CD4031 64-bit serial-in/ serial-out shift register recirculating data

The “data 2” above is wired to the Q_{64} output of the shift register. With “mode control” high, the Q_{64} output is routed back to the shifter data input D.

Data will recirculate from output to input. The data will repeat every 64 clock pulses as shown above.



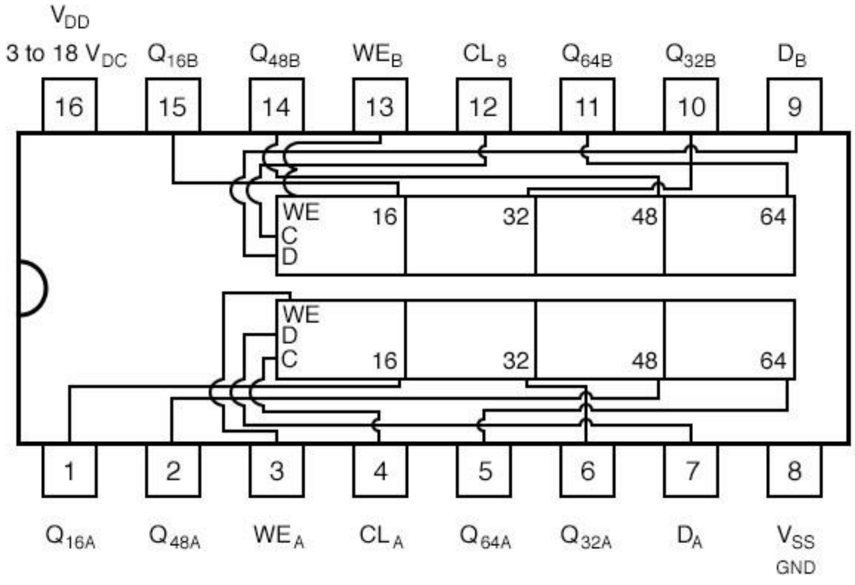
CD4031 64-bit serial-in/ serial-out shift register load new data at Data 1.

With “mode control” low, the CD4031 “data 1” is selected for input to the shifter.

The output, Q_{64}, is not recirculated because the lower data selector gate is disabled.

By disabled we mean that the logic low “mode select” inverted twice to a low at the lower NAND gate prevents it for passing any signal on the lower pin (data 2) to the gate output.

Thus, it is disabled.



CD4517b dual 64-bit serial-in/ serial-out shift register

A CD4517b dual 64-bit shift register is shown above. Note the taps at the 16th, 32nd, and 48th stages.

That means that shift registers of those lengths can be configured from one of the 64-bit shifters.

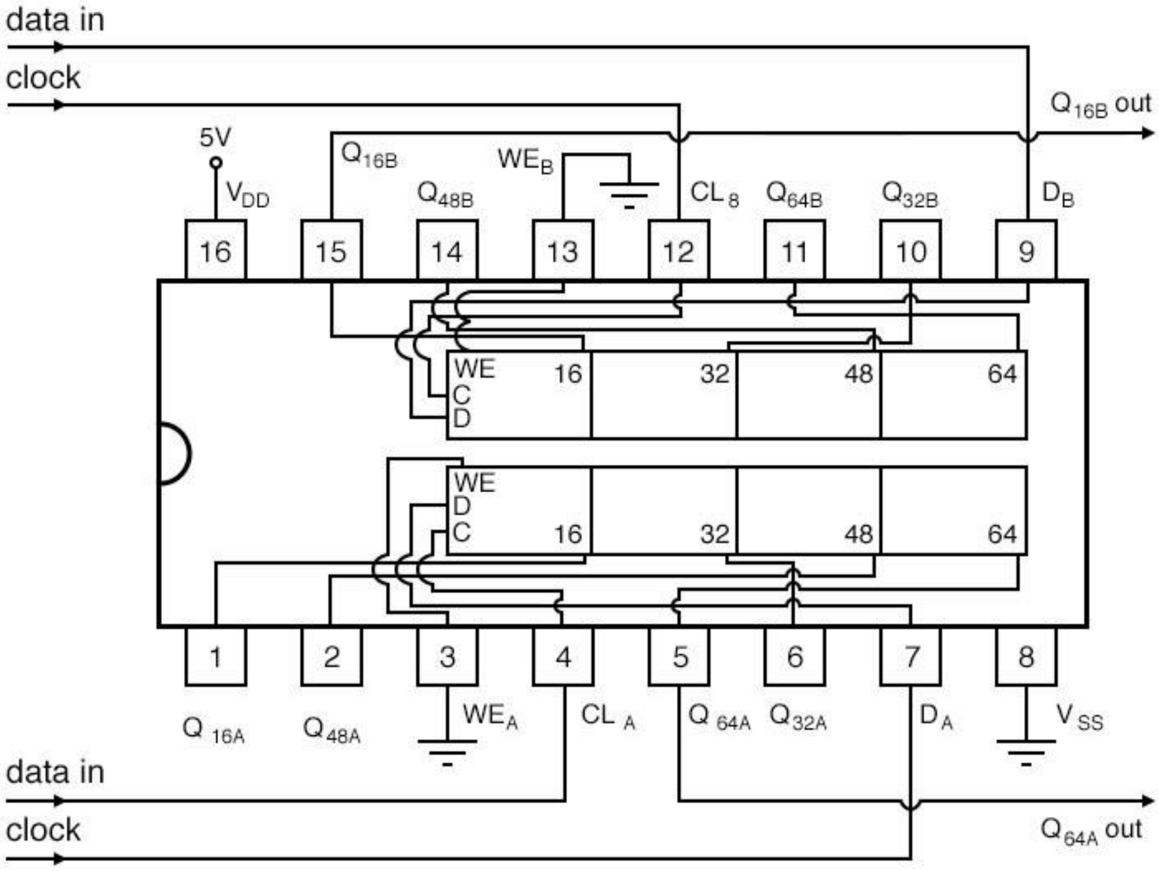
Of course, the 64-bit shifters may be cascaded to yield an 80-bit, 96-bit, 112-bit, or 128-bit shift register.

The clock CL_A and CL_B need to be paralleled when cascading the two shifters. WE_B and WE_B are grounded for normal shifting operations.

The data inputs to the shift registers A and B are D_A and D_B respectively.

Suppose that we require a 16-bit shift register.

Can this be configured with the CD4517b? How about a 64-shift register from the same part?



CD4517b dual 64-bit serial-in/serial-out shift register wired as 16-bit and 64-bit shift registers

Above we show A CD4517b wired as a 16-bit shift register for section B.

The clock for section B is CL_B. The data is clocked in at CL_B. And the data delayed by 16-clocks is picked of off Q_{16B}. WE_B , the write enable, is grounded.

Above we also show the same CD4517b wired as a 64-bit shift register for the independent section A.

The clock for section A is CL_A. The data enters at CL_A. The data delayed by 64-clock pulses is picked up from Q_{64A}. WE_A, the write enable for section A, is grounded.

Shift Registers: Parallel-in, Serial-out (PISO) Conversion

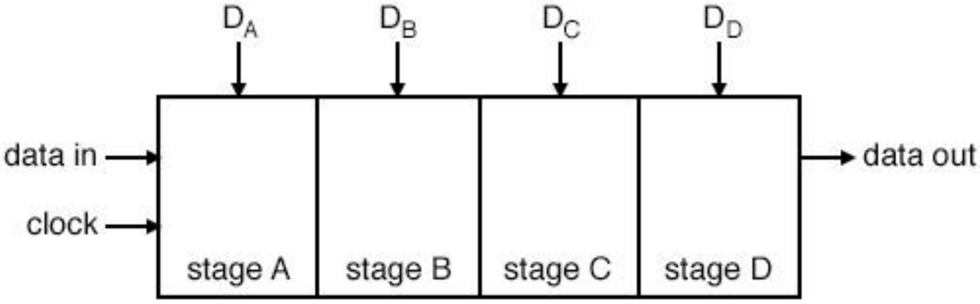
Parallel-in/ serial-out shift registers do everything that the previous serial-in/ serial-out shift registers do plus input data to all stages simultaneously.

The parallel-in/ serial-out shift register stores data, shifts it on a clock by clock basis, and delays it by the number of stages times the clock period.

In addition, parallel-in/ serial-out really means that we can load data in parallel into all stages before any shifting ever begins.

This is a way to convert data from a *parallel* format to a *serial* format. By parallel format we mean that the data bits are present simultaneously on individual wires, one for each data bit as shown below.

By serial format we mean that the data bits are presented sequentially in time on a single wire or circuit as in the case of the “data out” on the block diagram below.



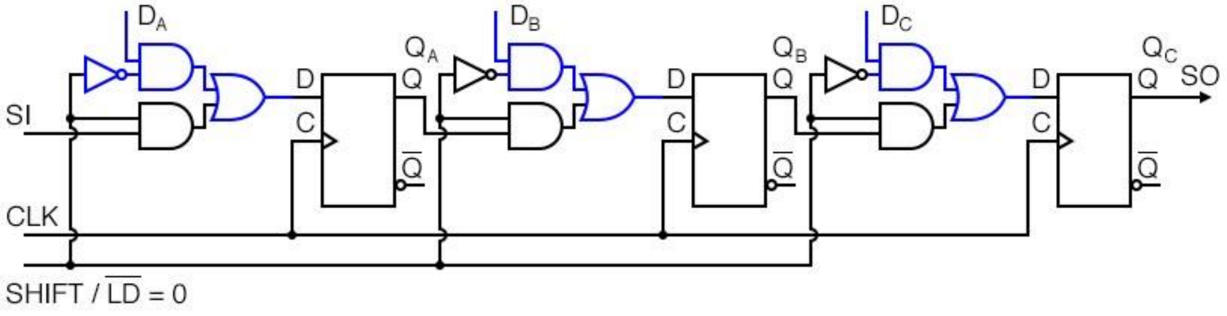
Parallel-in, serial-out shift register with 4-stages

Below we take a close look at the internal details of a 3-stage parallel-in/ serial-out shift register.

A stage consists of a type D Flip-Flop for storage, and an AND-OR selector to determine whether data will load in parallel, or shift stored data to the right.

In general, these elements will be replicated for the number of stages required. We show three stages due to space limitations.

Four, eight or sixteen bits is normal for real parts.



Parallel-in/ serial-out shift register showing parallel load path

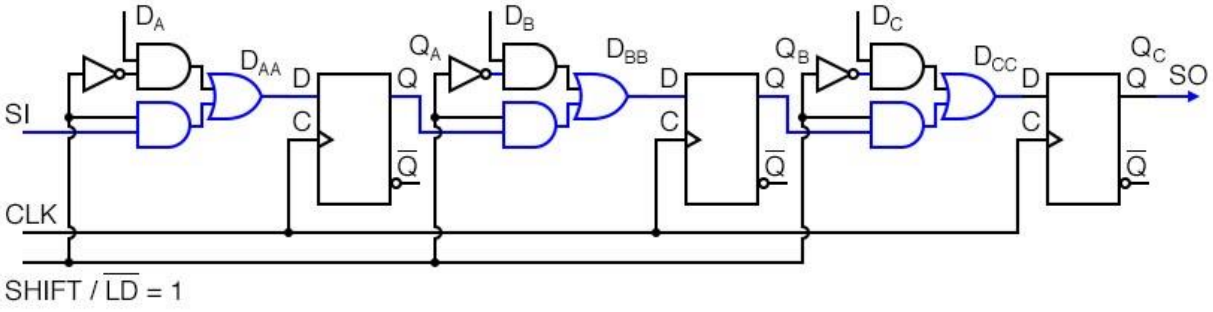
Above we show the parallel load path when SHIFT/LD' is logic low. The upper NAND gates serving D_A D_B D_C are enabled, passing data to the D inputs of type D Flip-Flops Q_A Q_B D_C respectively.

At the next positive going clock edge, the data will be clocked from D to Q of the three FFs. Three bits of data will load into Q_A Q_B D_C at the same time.

The type of parallel load just described, where the data loads on a clock pulse is known as *synchronous load* because the loading of data is synchronized to the clock.

This needs to be differentiated from asynchronous load where loading is controlled by the preset and clear pins of the Flip-Flops which does not require the clock.

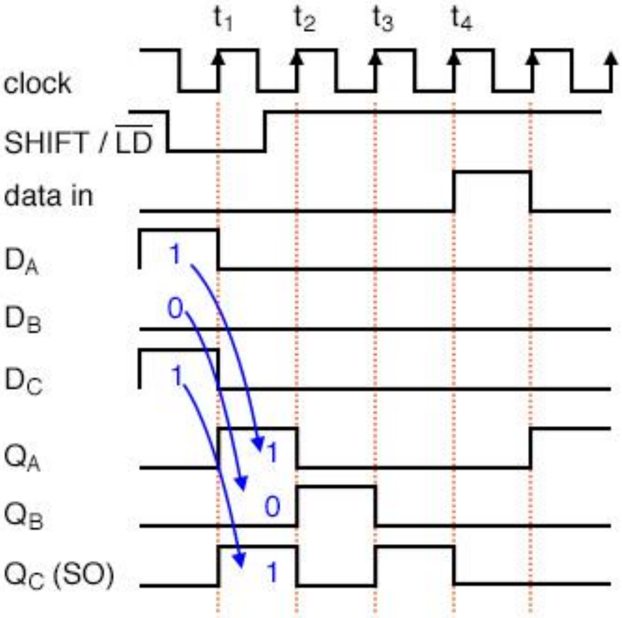
Only one of these load methods is used within an individual device, the synchronous load being more common in newer devices.



Parallel-in/ serial-out shift register showing shift path

The shift path is shown above when SHIFT/LD' is logic high. The lower AND gates of the pairs feeding the OR gate are enabled giving us a shift register connection of SI to D_A , Q_A to D_B , Q_B to D_C , Q_C to SO. Clock pulses will cause data to be right shifted out to SO on successive pulses.

The waveforms below show both parallel loading of three bits of data and serial shifting of this data. Parallel data at D_A D_B D_C is converted to serial data at SO.



Parallel-in/ serial-out shift register load/ shift waveforms

What we previously described with words for parallel loading and shifting is now set down as waveforms above.

As an example we present **101** to the parallel inputs D_{AA} D_{BB} D_{CC}. Next, the SHIFT/LD' goes low enabling loading of data as opposed to shifting of data.

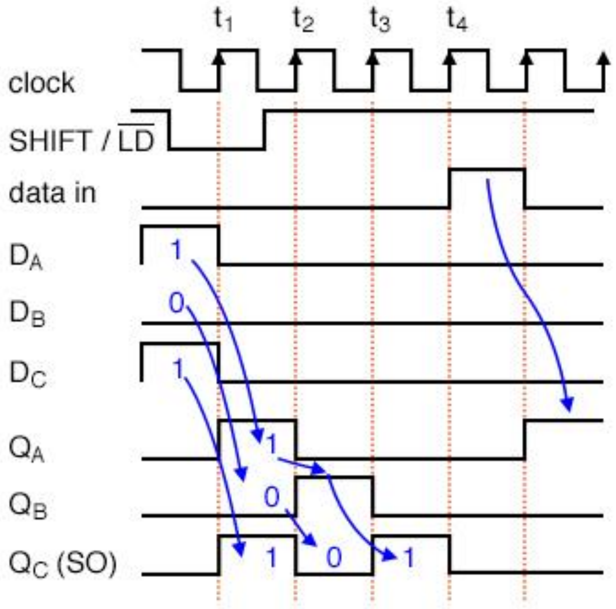
It needs to be low a short time before and after the clock pulse due to setup and hold requirements. It is considerably wider than it has to be.

Though, with synchronous logic it is convenient to make it wide. We could have made the active low SHIFT/LD’ almost two clocks wide, low almost a clock before t_1 and back high just before t_3.

The important factor is that it needs to be low around clock time t_1 to enable parallel loading of the data by the clock.

Note that at t1 the data **101** at D_A D_B D_C is clocked from D to Q of the Flip-Flops as shown at Q_A Q_B Q_C at time t_1.

This is the parallel loading of the data synchronous with the clock.



Parallel-in/ serial-out shift register load/ shift waveforms

Now that the data is loaded, we may shift it provided that SHIFT/LD’ is high to enable shifting, which it is prior to t_2.

At t_2 the data **0** at Q_C is shifted out of SO which is the same as the Q_C waveform. It is either shifted into another integrated circuit, or lost if there is nothing connected to SO.

The data at Q_B, a **0** is shifted to Q_C. The **1** at Q_A is shifted into Q_B. With “data in” a **0**, Q_A becomes **0**. After t_2, Q_A Q_B Q_C = **010**.

After t_3, Q_A Q_B Q_C = **001**. This **1**, which was originally present at Q_A after t_1, is now present at SO and Q_C.

The last data bit is shifted out to an external integrated circuit if it exists. After t_4 all data from the parallel load is gone.

At clock t_5 we show the shifting in of a data **1** present on the SI, serial input.

Q: Why provide SI and SO pins on a shift register?

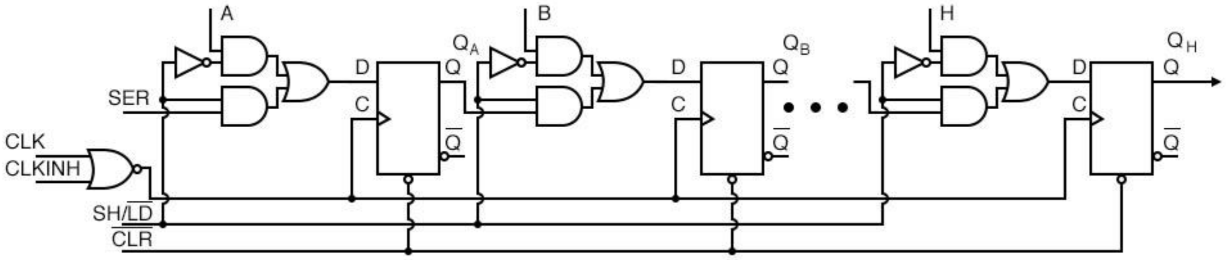
A: These connections allow us to cascade shift register stages to provide large shifters than available in a single IC (Integrated Circuit) package. They also allow serial connections to and from other ICs like microprocessors.

Let's take a closer look at parallel-in/ serial-out shift registers available as integrated circuits.

Parallel-in/serial-out devices

- SN74ALS166 parallel-in/ serial-out 8-bit shift register, synchronous load
- SN74ALS165 parallel-in/ serial-out 8-bit shift register, asynchronous load
- CD4014B parallel-in/ serial-out 8-bit shift register, synchronous load
- SN74LS647 parallel-in/ serial-out 16-bit shift register, synchronous load

(here SN74… are TTL, CD… are CMOS devices)



SN74ALS166 Parallel-in/ serial-out 8-bit shift register

The SN74ALS166 shown above is the closest match of an actual part to the previous parallel-in/ serial out shifter figures.

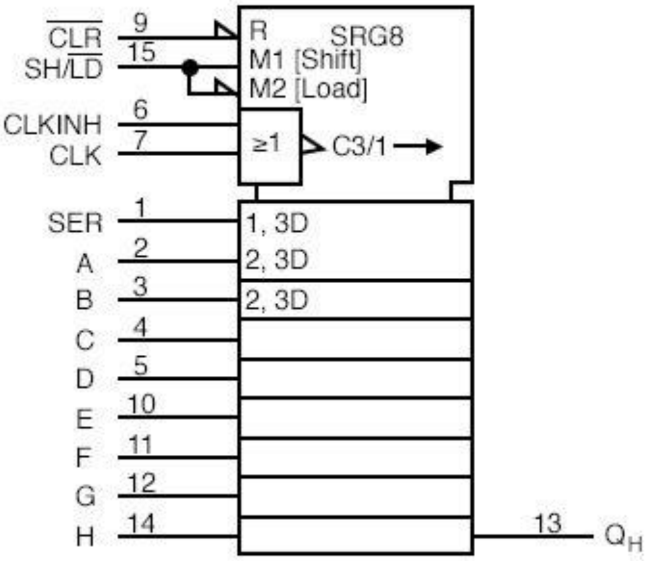
Let us note the minor changes to our figure above. First of all, there are 8-stages. We only show three.

The manufacturer labels the data inputs A, B, C, and so on to H.

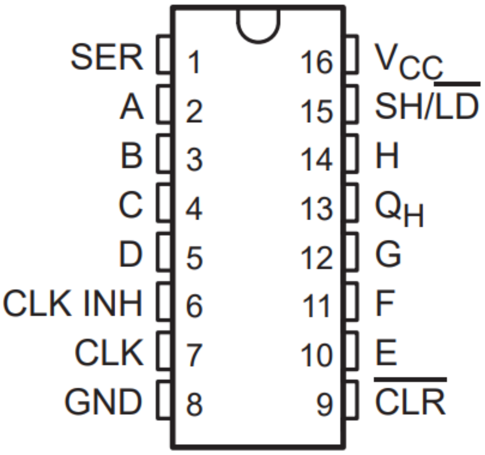
The SHIFT/LOAD control is called SH/LD'. It is abbreviated from our previous terminology, but works the same: parallel load if low, shift if high.

The shift input (serial data in) is SER on the ALS166 instead of SI. The clock CLK is controlled by an inhibit signal, CLKINH.

If CLKINH is high, the clock is inhibited, or disabled. Otherwise, this “real part” is the same as what we have looked at in detail.



SN74ALS166 ANSI Symbol



Above is the ANSI (American National Standards Institute) symbol for the SN74ALS166 as provided on the data sheet.

Once we know how the part operates, it is convenient to hide the details within a symbol. There are many general forms of symbols.

The advantage of the ANSI symbol is that the labels provide hints about how the part operates.

The large notched block at the top of the ‘74ASL166 is the control section of the ANSI symbol. There is a reset indicted by R.

There are three control signals: M1 (Shift), M2 (Load), and C3/1 (arrow) (inhibited clock). The clock has two functions.

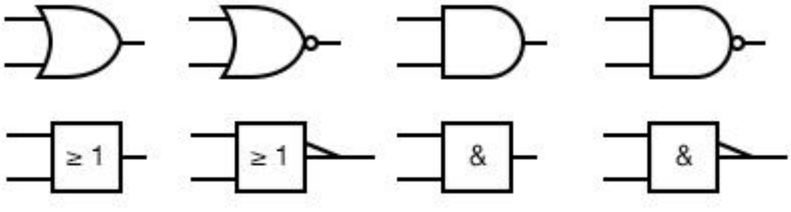
First, C3 for shifting parallel data wherever a prefix of 3 appears. Second, whenever M1 is asserted, as indicated by the 1 of C3/1 (arrow), the data is shifted as indicated by the right pointing arrow.

The slash (/) is a separator between these two functions. The 8-shift stages, as indicated by title SRG8, are identified by the external inputs A, B, C, to H.

The internal 2, 3D indicates that data, D, is controlled by M2 [Load] and C3 clock. In this case, we can conclude that the parallel data is loaded synchronously with the clock C3.

The upper stage at A is a wider block than the others to accommodate the input SER.

The legend 1, 3D implies that SER is controlled by M1 [Shift] and C3 clock. Thus, we expect to clock in data at SER when shifting as opposed to parallel loading.



ANSI gate symbols

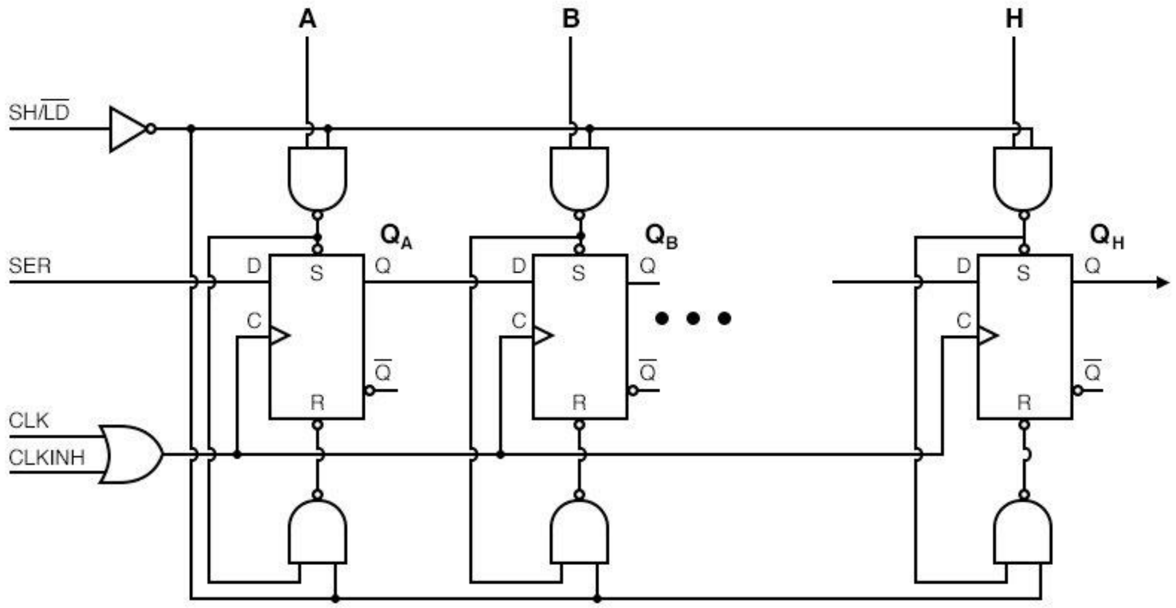
The ANSI/IEEE basic gate rectangular symbols are provided above for comparison to the more familiar shape symbols so that we may decipher the meaning of the symbology associated with the CLKINH and CLKpins on the previous ANSI SN74ALS166 symbol.

The CLK and CLKINH feed an OR gate on the SN74ALS166 ANSI symbol. OR is indicated by => on the rectangular inset symbol.

The long triangle at the output indicates a clock. If there was a bubble with the arrow this would have indicated shift on negative clock edge (high to low).

Since there is no bubble with the clock arrow, the register shifts on the positive (low to high transition) clock edge.

The long arrow, after the legend C3/1 pointing right indicates shift right, which is down the symbol.



SN74ALS165 Parallel-in/ serial-out 8-bit shift register, asynchronous load

Part of the internal logic of the SN74ALS165 parallel-in/ serial-out, asynchronous load shift register is reproduced from the data sheet above.

We have not looked at asynchronous loading of data up to this point.

First of all, the loading is accomplished by application of appropriate signals to the Set (preset) and Reset (clear) inputs of the Flip-Flops.

The upper NAND gates feed the Set pins of the FFs and also cascades into the lower NAND gate feeding the Reset pins of the FFs.

The lower NAND gate inverts the signal in going from the Set pin to the Reset pin.

First, SH/LD’ must be pulled Low to enable the upper and lower NAND gates.

If SH/LD’ were at a logic high instead, the inverter feeding a logic low to all NAND gates would force a High out, releasing the “active low” Set and Reset pins of all FFs.

There would be no possibility of loading the FFs.

With SH/LD’ held Low, we can feed, for example, a data 1 to parallel input A, which inverts to a zero at the upper NAND gate output, setting FF Q_A to a 1.

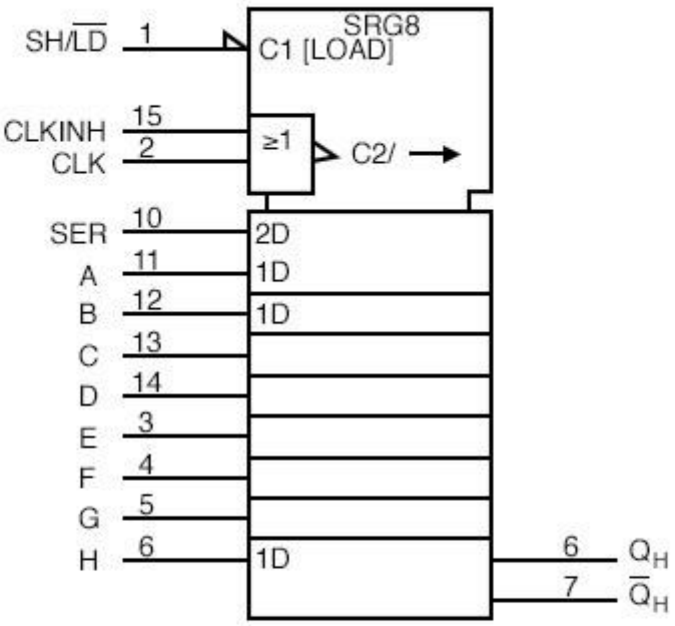
The 0 at the Set pin is fed to the lower NAND gate where it is inverted to a 1 , releasing the Reset pin of Q_A.

Thus, a data A=1 sets Q_A=1. Since none of this required the clock, the loading is asynchronous with respect to the clock.

We use an asynchronous loading shift register if we cannot wait for a clock to parallel load data, or if it is inconvenient to generate a single clock pulse.

The only difference in feeding a data 0 to parallel input A is that it inverts to a 1 out of the upper gate releasing Set.

This 1 at Set is inverted to a 0 at the lower gate, pulling Reset to a Low, which resets Q_A=0.



SN74ALS165 ANSI Symbol

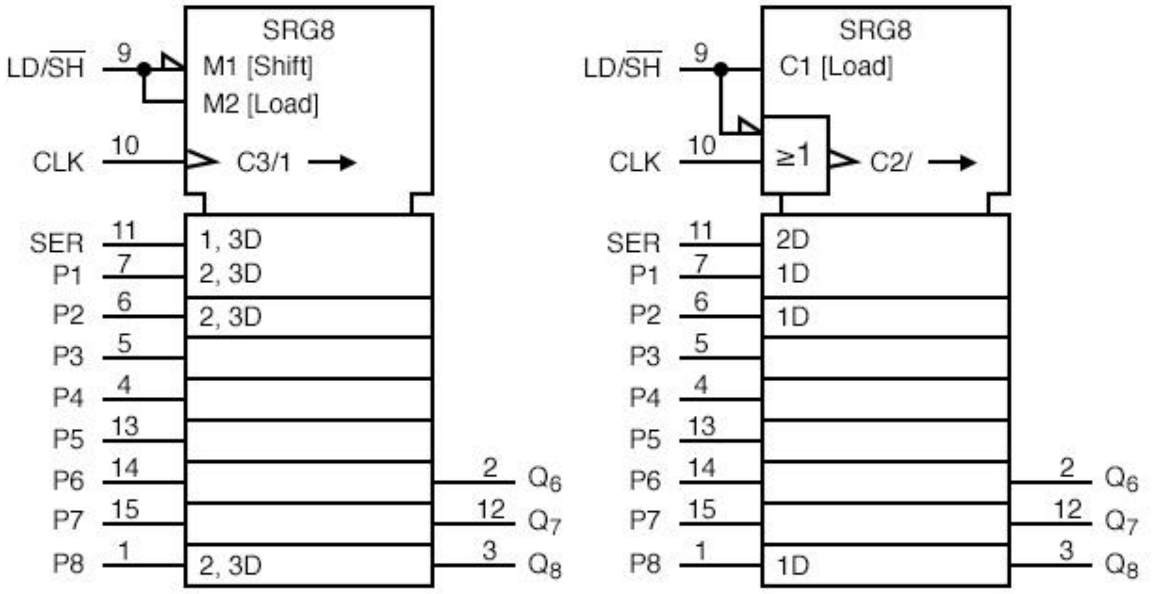
The ANSI symbol for the SN74ALS166 above has two internal controls C1 [LOAD] and C2 clock from the OR function of (CLKINH, CLK).

SRG8 says 8-stage shifter. The arrow after C2 indicates shifting right or down. SER input is a function of the clock as indicated by internal label 2D.

The parallel data inputs A, B, C to H are a function of C1 [LOAD], indicated by internal label 1D.

C1 is asserted when sh/LD’ =0 due to the half-arrow inverter at the input.

Compare this to the control of the parallel data inputs by the clock of the previous synchronous ANSI SN75ALS166. Note the differences in the ANSI Data labels.



CD4014B, synchronous load

CD4021B, asynchronous load

CMOS Parallel-in/ serial-out shift registers, 8-bit ANSI symbols

On the CD4014B above, M1 is asserted when LD/SH'=0. M2 is asserted when LD/SH'=1.

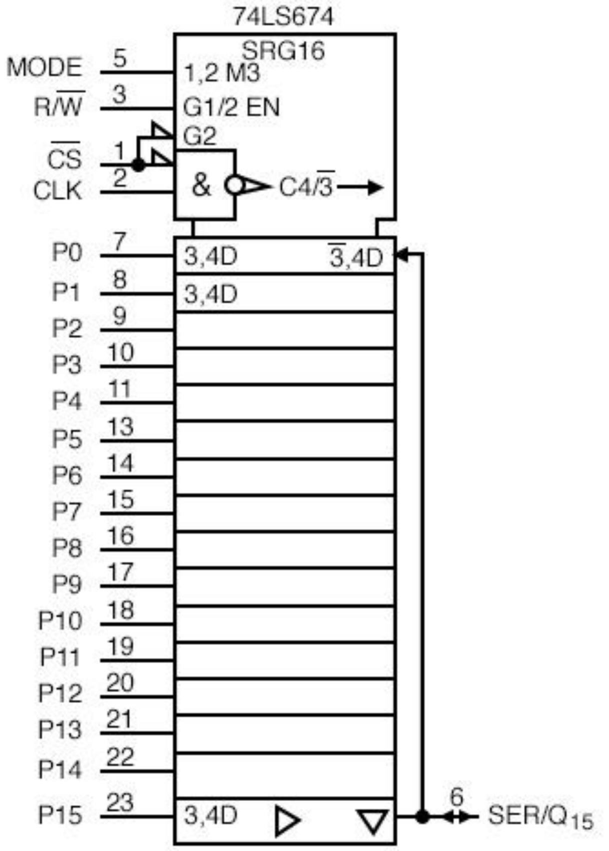
Clock C3/1 is used for parallel loading data at 2, 3D when M2 is active as indicated by the 2,3 prefix labels.

Pins P3 to P7 are understood to have the smae internal 2,3 prefix labels as P2 and P8. At SER, the 1,3D prefix implies that M1 and clock C3 are necessary to input serial data.

Right shifting takes place when M1 active is as indicated by the 1 in C3/1 arrow.

The CD4021B is a similar part except for asynchronous parallel loading of data as implied by the lack of any 2 prefix in the data label 1D for pins P1, P2, to P8.

Of course, prefix 2 in label 2D at input SER says that data is clocked into this pin. The OR gate inset shows that the clock is controlled by LD/SH'.



SN74LS674, Parallel-in/ serial-out, synchronous load

The above SN74LS674 internal label SRG 16 indicates 16-bit shift register.

The MODE input to the control section at the top of the symbol is labeled 1,2 M3. Internal M3 is a function of input MODE and G1 and G2as indicated by the 1,2 preceding M3.

The base label G indicates an AND function of any such G inputs. Input R/W' is internally labeled G1/2 EN.

This is an enable EN (controlled by G1 AND G2) for tristate devices used elsewhere in the symbol.

We note that CS' on (pin 1) is internal G2. Chip select CS' also is ANDed with the input CLK to give internal clock C4.

The bubble within the clock arrow indicates that activity is on the negative (high to low transition) clock edge.

The slash (/) is a separator implying two functions for the clock. Before the slash, C4 indicates control of anything with a prefix of 4.

After the slash, the 3’ (arrow) indicates shifting. The 3’ of C4/3’ implies shifting when M3 is de-asserted (MODE=0). The long arrow indicates shift right (down).

Moving down below the control section to the data section, we have external inputs P0-P15, pins (7-11, 13-23).

The prefix 3,4 of internal label 3,4D indicates that M3 and the clock C4 control loading of parallel data.

The D stands for Data. This label is assumed to apply to all the parallel inputs, though not explicitly written out.

Locate the label 3’,4D on the right of the P0 (pin7) stage. The complemented-3 indicates thatM3=MODE=0 inputs (shifts) SER/Q15 (pin5) at clock time, (4 of 3’,4D) corresponding to clock C4.

In other words, with MODE=0, we shift data into Q0 from the serial input (pin 6). All other stages shift right (down) at clock time.

Moving to the bottom of the symbol, the triangle pointing right indicates a buffer between Q and the output pin.

The Triangle pointing down indicates a tri-state device. We previously stated that the tristate is controlled by enable EN, which is actually G1 AND G2 from the control section.

If R/W=0, the tri-state is disabled, and we can shift data into Q0 via SER (pin 6), a detail we omitted above. We actually need MODE=0, R/W’=0, CS’=0

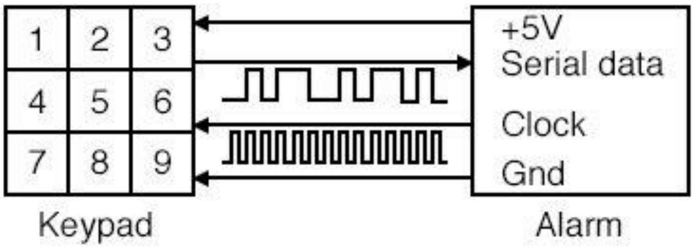
The internal logic of the SN74LS674 and a table summarizing the operation of the control signals is available in the link in the bullet list, top of section.

If R/W’=1, the tristate is enabled, Q15 shifts out SER/Q15 (pin 6) and recirculates to the Q0 stage via the right hand wire to 3’,4D.

We have assumed that CS’ was low giving us clock C4/3’ and G2 to ENable the tri-state.

Practical Applications

An application of a parallel-in/ serial-out shift register is to read data into a microprocessor.

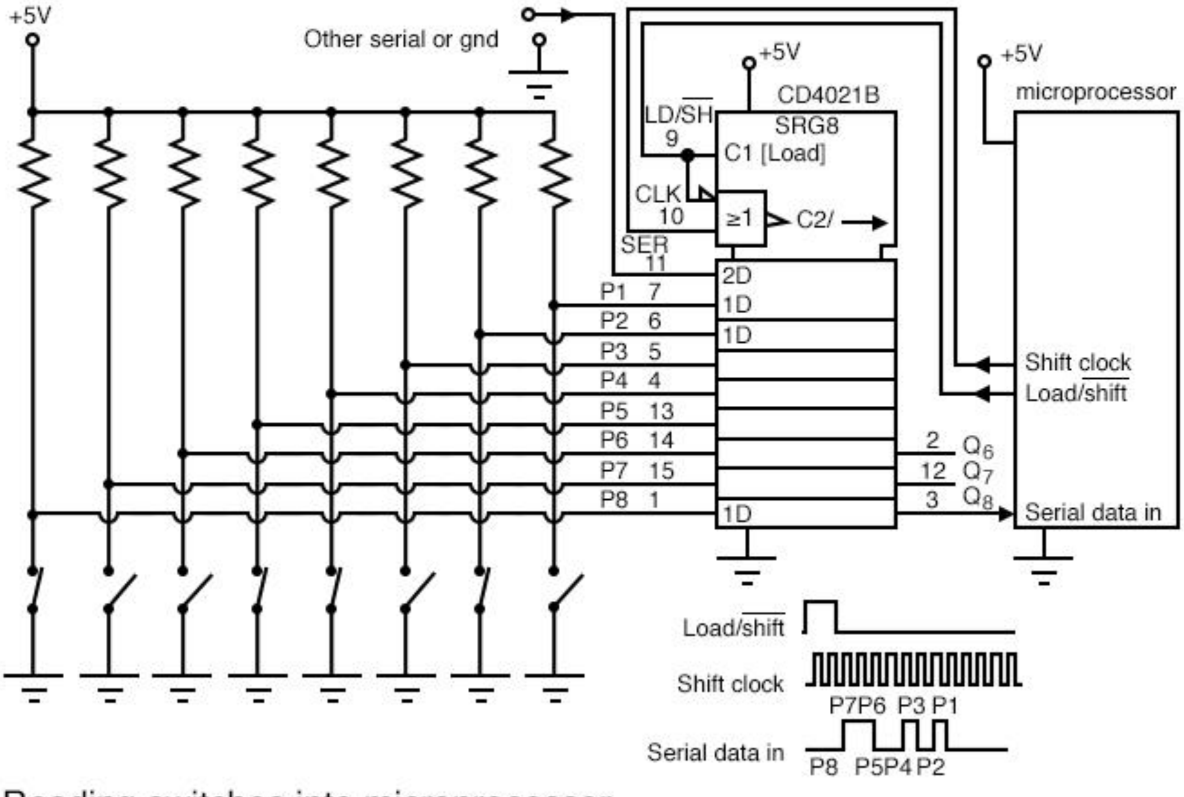


Alarm with remote keypad

The Alarm is controlled by a remote keypad. The alarm box supplies +5V and ground to the remote keypad to power it.

The alarm reads the remote keypad every few tens of milliseconds by sending shift clocks to the keypad which returns serial data showing the status of the keys via a parallel-in/ serial-out shift register.

Thus, we read nine key switches with four wires. How many wires would be required if we had to run a circuit for each of the nine keys?



Reading switches into microprocessor

A practical application of a parallel-in/ serial-out shift register is to read many switch closures into a microprocessor on just a few pins.

Some low end microprocessors only have 6-I/O (Input/Output) pins available on an 8-pin package.

Or, we may have used most of the pins on an 84-pin package. We may want to reduce the number of wires running around a circuit board, machine, vehicle, or building.

This will increase the reliability of our system. It has been reported that manufacturers who have reduced the number of wires in an automobile produce a more reliable product.

In any event, only three microprocessor pins are required to read in 8-bits of data from the switches in the figure above.

We have chosen an asynchronous loading device, the CD4021B because it is easier to control the loading of data without having to generate a single parallel load clock.

The parallel data inputs of the shift register are pulled up to +5V with a resistor on each input.

If all switches are open, all 1s will be loaded into the shift register when the microprocessor moves the LD/SH' line from low to high, then back low in anticipation of shifting.

Any switch closures will apply logic 0s to the corresponding parallel inputs. The data pattern at P1-P7 will be parallel loaded by the LD/SH'=1 generated by the microprocessor software.

The microprocessor generates shift pulses and reads a data bit for each of the 8-bits.

This process may be performed totally with software, or larger microprocessors may have one or more serial interfaces to do the task more quickly with hardware.

With LD/SH'=0, the microprocessor generates a 0 to 1 transition on the Shift clock line, then reads a data bit on the Serial data in line. This is repeated for all 8-bits.

The SER line of the shift register may be driven by another identical CD4021B circuit if more switch contacts need to be read.

In which case, the microprocessor generates 16-shift pulses. More likely, it will be driven by something else compatible with this serial data format, for example, an analog to digital converter, a temperature sensor, a keyboard scanner, a serial read-only memory.

As for the switch closures, they may be limit switches on the carriage of a machine, an over-temperature sensor, a magnetic reed switch, a door or window switch, an air or water pressure switch, or a solid state optical interrupter.

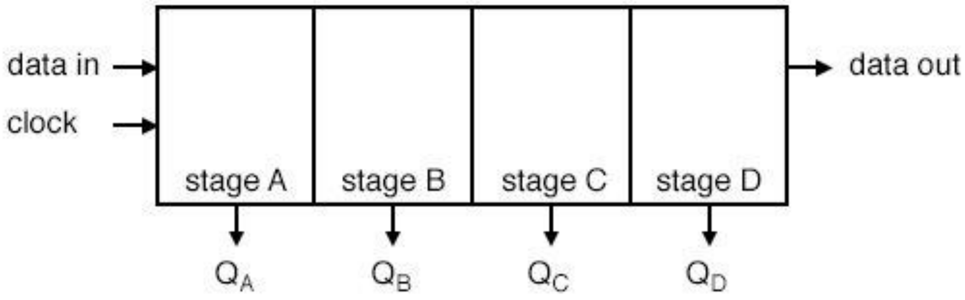
Shift Registers: Serial-in, Parallel-out (SIPO) Conversion

A serial-in, parallel-out shift register is similar to the serial-in, serial-out shift register in that it shifts data into internal storage elements and shifts data out at the serial-out, data-out, pin.

It is different in that it makes all the internal stages available as outputs. Therefore, a serial-in, parallel-out shift register converts data from serial format to parallel format.

An Example of Using Serial-in, Parallel-out Shift Register

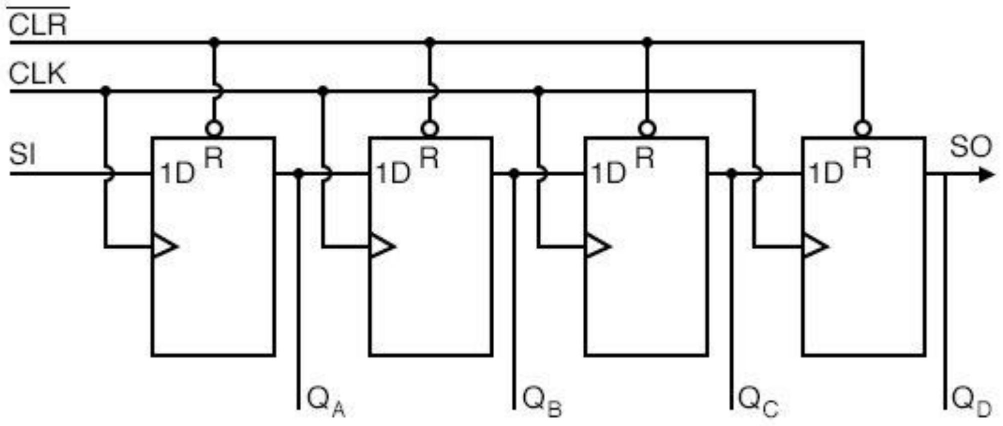
If four data bits are shifted in by four clock pulses via a single wire at data-in, below, the data becomes available simultaneously on the four Outputs Q_A to Q_D after the fourth clock pulse.



Serial-in, parallel-out shift register with 4-stages

The practical application of the serial-in, parallel-out shift register is to convert data from serial format on a single wire to parallel format on multiple wires.

Let’s illuminate four LEDs (light emitting diodes) with the four outputs (Q_A Q_B Q_C Q_D).



Serial-in/ Parallel-out shift register details

The above details of the serial-in, parallel-out shift register are fairly simple. It looks like a serial-in, serial-out shift register with taps added to each stage output.

Serial data shifts in at SI (Serial Input). After a number of clocks equal to the number of stages, the first data bit in appears at SO (Q_D) in the above figure.

In general, there is no SO pin. The last stage (Q_D above) serves as SO and is cascaded to the next package if it exists.

Serial-in, Parallel-out vs. Serial-in, Serial-out Shift Register

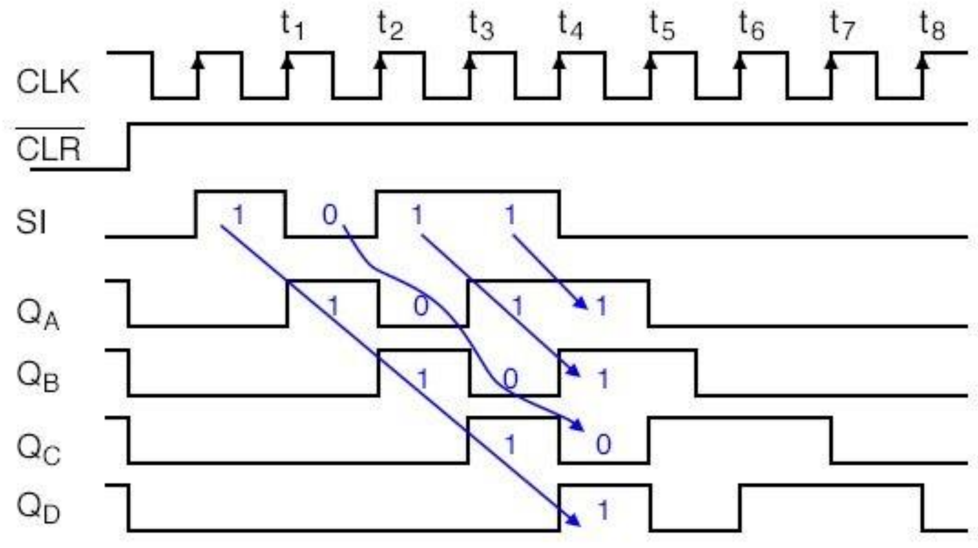
If a serial-in, parallel-out shift register is so similar to a serial-in, serial-out shift register, why do manufacturers bother to offer both types?

Why not just offer the serial-in, parallel-out shift register?

The answer is that they actually only offer the serial-in, parallel-out shift register, as long as it has no more than 8-bits.

Note that serial-in, serial-out shift registers come in bigger than 8-bit lengths of 18 to 64-bits.

It is not practical to offer a 64-bit serial-in, parallel-out shift register requiring that many output pins. See waveforms (timing diagrams) below for above shift register.



Serial-in/ parallel-out shift register waveforms

The shift register has been cleared prior to any data by CLR', an active low signal, which clears all type D Flip-Flops within the shift register.

Note the serial data 1011 pattern presented at the SI input. This data is synchronized with the clock CLK.

This would be the case if it is being shifted in from something like another shift register, for example, a parallel-in, serial-out shift register (not shown here).

On the first clock at t_1, the data 1 at SI is shifted from D to Q of the first shift register stage. After t_2 this first data bit is at Q_B.

After t_3 it is at Q_C. After t_4 it is at Q_D. Four clock pulses have shifted the first data bit all the way to the last stage Q_D.

The second data bit a 0 is at Q_C after the 4th clock. The third data bit a 1 is at Q_B. The fourth data bit another 1 is at Q_A.

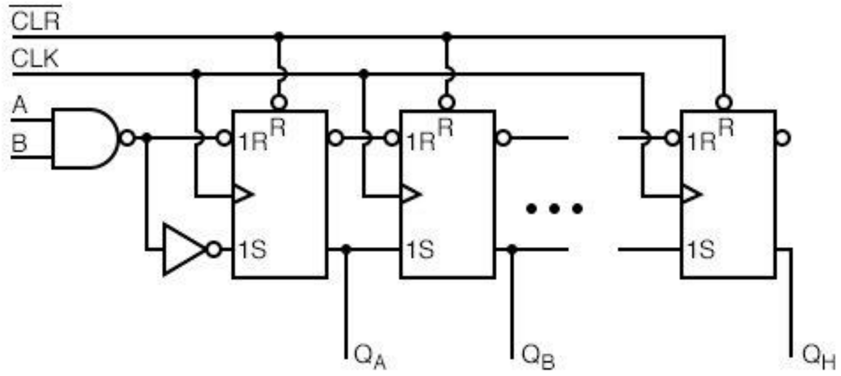
Thus, the serial data input pattern 1011 is contained in (Q_D Q_C Q_B Q_A). It is now available on the four outputs.

It will available on the four outputs from just after clock t_4 to just before t_5. This parallel data must be used or stored between these two times, or it will be lost due to shifting out the QD stage on following clocks t_5 to t_8 as shown above.

Serial-in, Parallel-out Devices

Let’s take a closer look at serial-in, parallel-out shift registers available as integrated circuits.

- SN74ALS164A serial-in/ parallel-out 8-bit shift register
- SN74AHC594 serial-in/ parallel-out 8-bit shift register with output register
- SN74AHC595 serial-in/ parallel-out 8-bit shift register with output register
- CD4094 serial-in/ parallel-out 8-bit shift register with output register

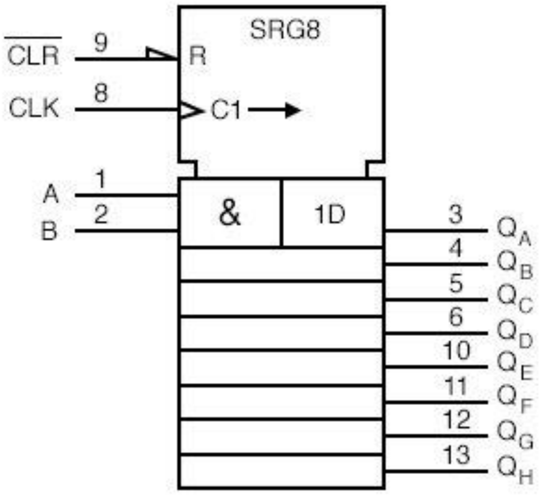


Serial-in/ Parallel-out shift register details

The 74ALS164A is almost identical to our prior diagram with the exception of the two serial inputs A and B.

The unused input should be pulled high to enable the other input. We do not show all the stages above.

However, all the outputs are shown on the ANSI symbol below, along with the pin numbers.

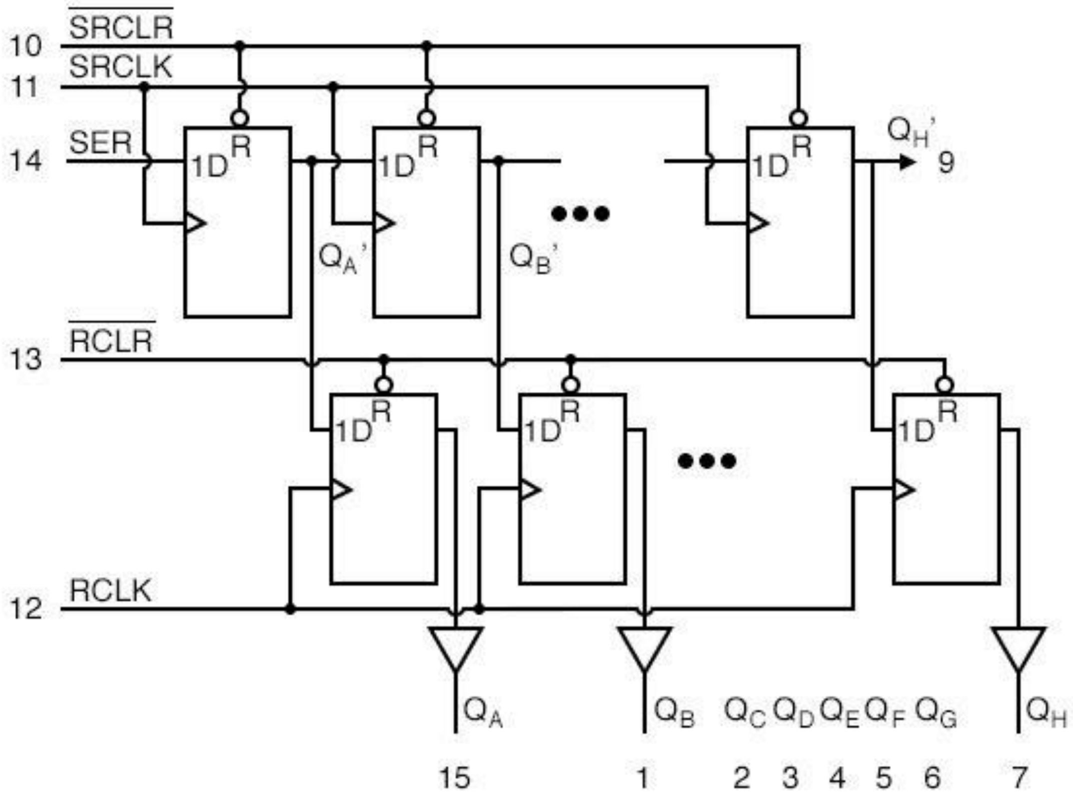


SN74ALS164A ANSI Symbol

The CLK input to the control section of the above ANSI symbol has two internal functions C1, control of anything with a prefix of 1.

This would be clocking in of data at 1D. The second function, the arrow after the slash (/) is right (down) shifting of data within the shift register.

The eight outputs are available to the right of the eight registers below the control section. The first stage is wider than the others to accommodate the A&B input.



The above internal logic diagram is adapted from the TI (Texas Instruments) data sheet for the 74AHC594. The type “D” FFs in the top row comprise a serial-in, parallel-out shift register.

This section works like the previously described devices. The outputs (Q_A’ Q_B’ to Q_H’) of the shift register half of the device feed the type “D” FFs in the lower half in parallel. Q_H’ (pin 9) is shifted out to any optional cascaded device package.

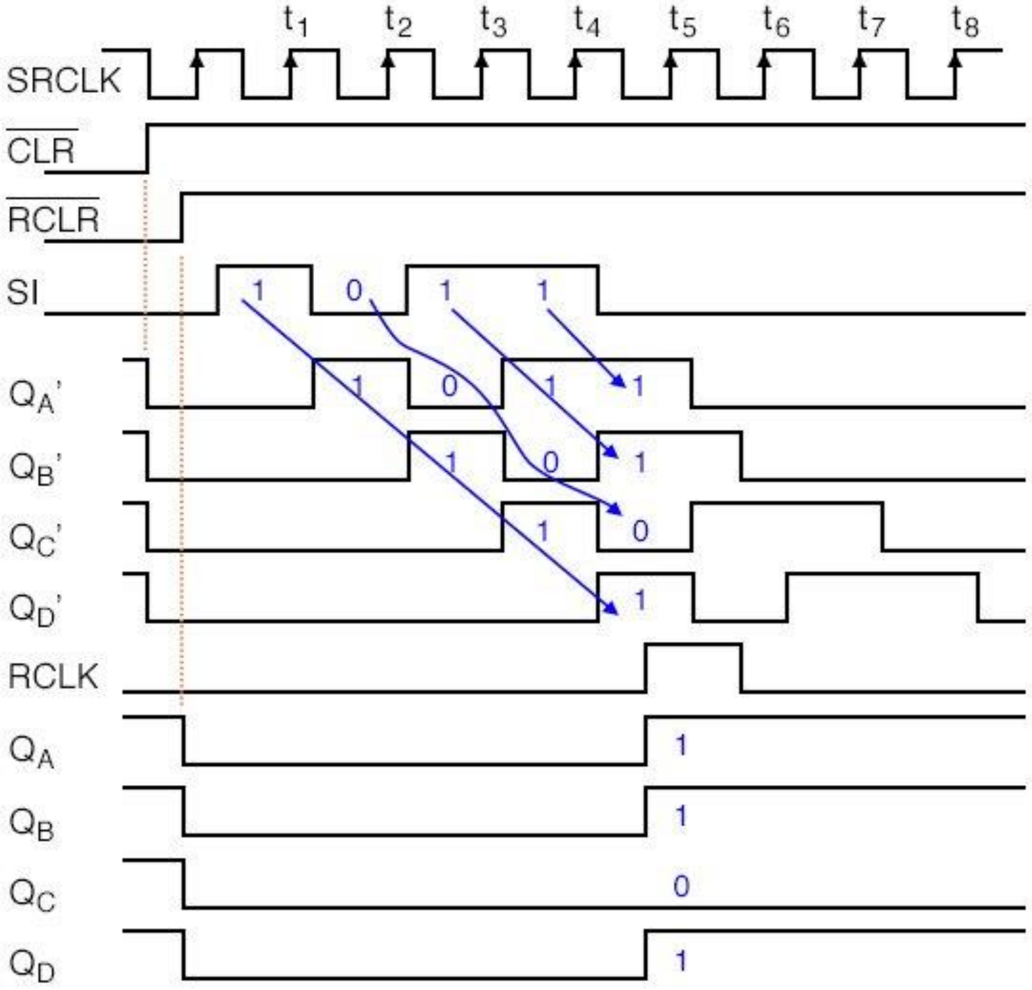
A single positive clock edge at RCLK will transfer the data from D to Q of the lower FFs. All 8-bits transfer in parallel to the output register (a collection of storage elements).

The purpose of the output register is to maintain a constant data output while new data is being shifted into the upper shift register section.

This is necessary if the outputs drive relays, valves, motors, solenoids, horns, or buzzers. This feature may not be necessary when driving LEDs as long as flicker during shifting is not a problem.

Note that the 74AHC594 has separate clocks for the shift register (SRCLK) and the output register (RCLK). Also, the shifter may be cleared by SRCLR and, the output register by RCLR.

It desirable to put the outputs in a known state at power-on, in particular, if driving relays, motors, etc. The waveforms below illustrate shifting and latching of data.



Waveforms for 74AHC594 serial-in/ parallel-out shift register with latch

The above waveforms show shifting of 4-bits of data into the first four stages of 74AHC594, then the parallel transfer to the output register.

In actual fact, the 74AHC594 is an 8-bit shift register, and it would take 8-clocks to shift in 8-bits of data, which would be the normal mode of operation.

However, the 4-bits we show saves space and adequately illustrates the operation.

We clear the shift register half a clock prior to t0 with SRCLR’=0. SRCLR’ must be released back high prior to shifting.

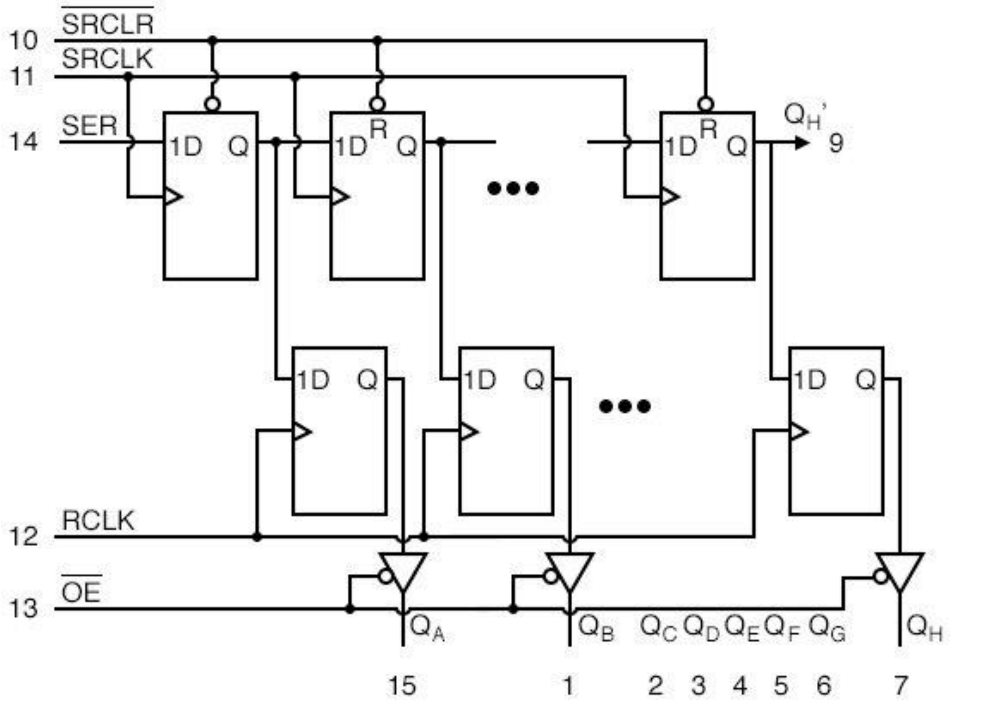
Just prior to t0 the output register is cleared by RCLR’=0. It, too, is released (RCLR’=1).

Serial data 1011 is presented at the SI pin between clocks t_0 and t_4. It is shifted in by clocks t_1 t_2 t_3 t_4 appearing at internal shift stages Q_A’ Q_B’ Q_C’ Q_D’ .

This data is present at these stages between t_4 and t_5. After t_5 the desired data (1011) will be unavailable on these internal shifter stages.

Between t_4 and t_5 we apply a positive going RCLK transferring data 1011 to register outputs Q_A Q_B Q_C Q_D .

This data will be frozen here as more data (0s) shifts in during the succeeding SRCLKs (t_5 to t_8). There will not be a change in data here until another RCLK is applied.



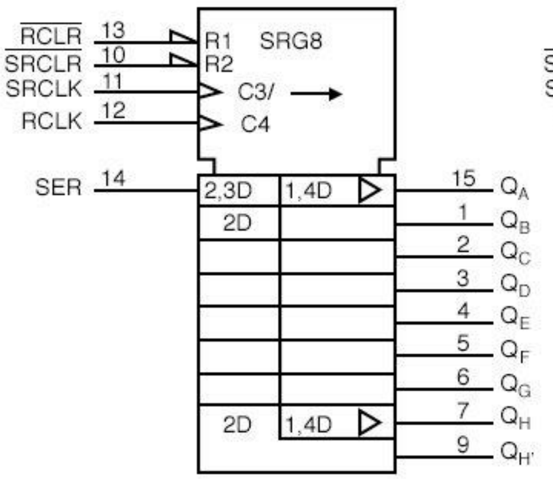
74AHC595 Serial-in/ Parallel-out 8-bit shift register with output registers

The 74AHC595 is identical to the ‘594 except that the RCLR’ is replaced by an OE’ enabling a tri-state buffer at the output of each of the eight output register bits.

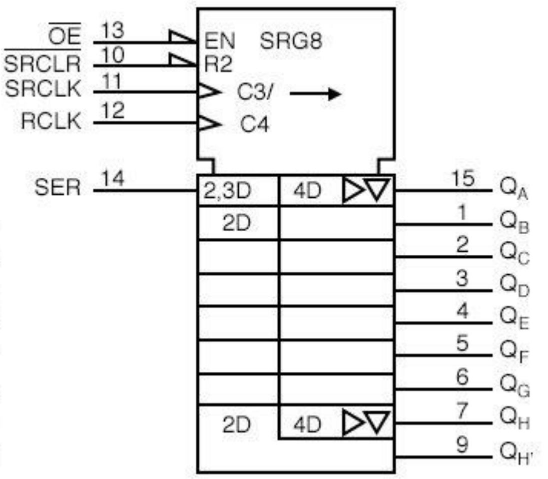
Though the output register cannot be cleared, the outputs may be disconnected by OE’=1.

This would allow external pull-up or pull-down resistors to force any relay, solenoid, or valve drivers to a known state during a system power-up.

Once the system is powered-up and, say, a microprocessor has shifted and latched data into the ‘595, the output enable could be asserted (OE’=0) to drive the relays, solenoids, and valves with valid data, but, not before that time.



SN74AHC594 ANSI Symbol



SN74AHC595 ANSI Symbol

Above are the proposed ANSI symbols for these devices. C3 clocks data into the serial input (external SER) as indicated by the 3 prefix of 2,3D.

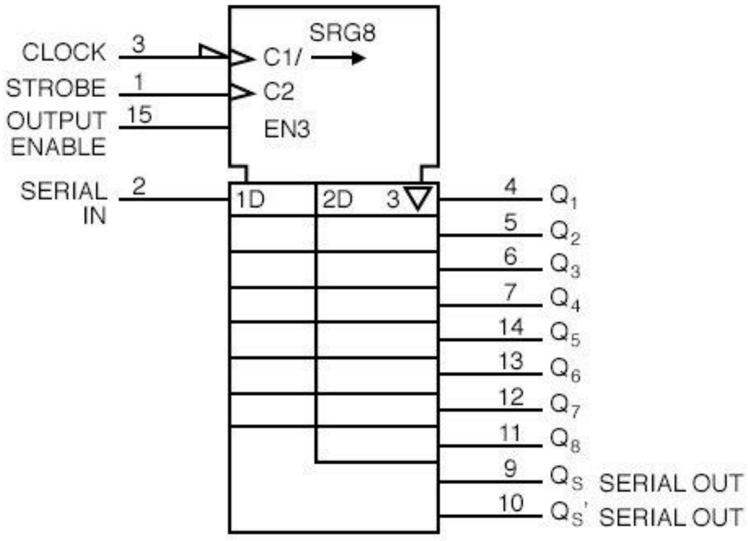
The arrow after C3/ indicates shifting right (down) of the shift register, the 8-stages to the left of the ‘595symbol below the control section.

The 2 prefix of 2,3D and 2D indicates that these stages can be reset by R2 (external SRCLR’).

The 1 prefix of 1,4D on the ‘594 indicates that R1 (external RCLR’) may reset the output register, which is to the right of the shift register section.

The ‘595, which has an EN at external OE’ cannot reset the output register. But, the EN enables tristate (inverted triangle) output buffers.

The right pointing triangle of both the ‘594 and‘595 indicates internal buffering. Both the ‘594 and‘595 output registers are clocked by C4 as indicated by 4 of 1,4D and 4D respectively.



CD4094B/ 74HCT4094 ANSI Symbol

The CD4094B is a 3 to 15VDC capable latching shift register alternative to the previous 74AHC594 devices.

CLOCK, C1, shifts data in at SERIAL IN as implied by the 1 prefix of 1D.

It is also the clock of the right shifting shift register (left half of the symbol body) as indicated by the /(right-arrow) of C1/(arrow) at the CLOCK input.

STROBE, C2 is the clock for the 8-bit output register to the right of the symbol body. The 2 of 2D indicates that C2 is the clock for the output register.

The inverted triangle in the output latch indicates that the output is tristated, being enabled by EN3.

The 3 preceding the inverted triangle and the 3 of EN3 are often omitted, as any enable (EN) is understood to control the tristate outputs. Q_S and Q_S’ are non-latched outputs of the shift register stage.

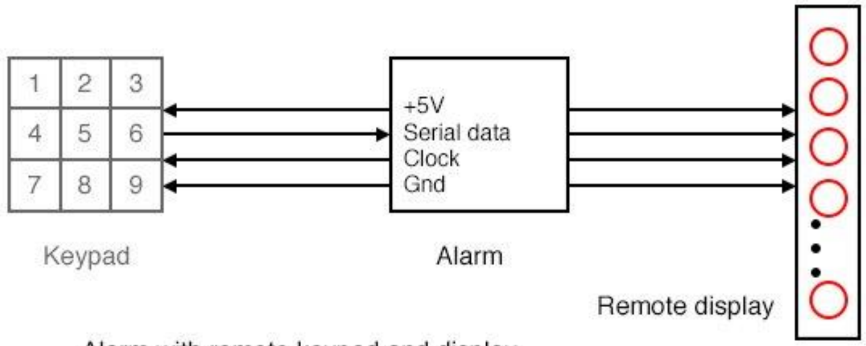
Q_S could be cascaded to SERIAL IN of a succeeding device.

Practical Applications

A real-world application of the serial-in, parallel-out shift register is to output data from a microprocessor to a remote panel indicator.

Or, another remote output device which accepts serial format data.

The figure “Alarm with remote key pad” is repeated here from the parallel-in, serial-out section with the addition of the remote display.



Alarm with remote keypad and display

Thus, we can display, for example, the status of the alarm loops connected to the main alarm box.

If the Alarm detects an open window, it can send serial data to the remote display to let us know.

Both the keypad and the display would likely be contained within the same remote enclosure, separate from the main alarm box. However, we will only look at the display panel here.

If the display were on the same board as the Alarm, we could just run eight wires to the eight LEDs along with two wires for power and ground.

These eight wires are much less desirable on a long run to a remote panel. Using shift registers, we only need to run five wires- clock, serial data, a strobe, power, and ground.

If the panel were just a few inches away from the main board, it might still be desirable to cut down on the number of wires in a connecting cable to improve reliability.

Also, we sometimes use up most of the available pins on a microprocessor and need to use serial techniques to expand the number of outputs.

Some integrated circuit output devices, such as Digital to Analog converters contain serial-in, parallel-out shift registers to receive data from microprocessors.

We have chosen the 74AHC594 serial-in, parallel-out shift register with output register; though, it requires an extra pin, RCLK, to parallel load the shifted-in data to the output pins.

This extra pin prevents the outputs from changing while data is shifting in. This is not much of a problem for LEDs. But, it would be a problem if driving relays, valves, motors, etc.

Code executed within the microprocessor would start with 8-bits of data to be output. One bit would be output on the “Serial data out” pin, driving SER of the remote 74AHC594.

Next, the microprocessor generates a low to high transition on “Shift clock”, driving SRCLK of the ‘595 shift register.

This positive clock shifts the data bit at SER from “D” to “Q” of the first shift register stage.

This has no effect on the Q_A LED at this time because of the internal 8-bit output register between the shift register and the output pins (Q_A to Q_H).

Finally, “Shift clock” is pulled back low by the microprocessor. This completes the shifting of one bit into the ‘595.

The above procedure is repeated seven more times to complete the shifting of 8-bits of data from the microprocessor into the 74AHC594 serial-in, parallel-out shift register.

To transfer the 8-bits of data within the internal ‘595 shift register to the output requires that the microprocessor generate a low to high transition on RCLK, the output register clock.

This applies new data to the LEDs. The RCLK needs to be pulled back low in anticipation of the next 8-bit transfer of data.

The data present at the output of the ‘595 will remain until the process is repeated for a new 8-bits of data.

In particular, new data can be shifted into the ‘595 internal shift register without affecting the LEDs. The LEDs will only be updated with new data with the application of the RCLK rising edge.

What if we need to drive more than eight LEDs? Simply cascade another 74AHC594 SER pin to the QH’ of the existing shifter.

Parallel the SRCLK and RCLK pins. The microprocessor would need to transfer 16-bits of data with 16-clocks before generating an RCLK feeding both devices.

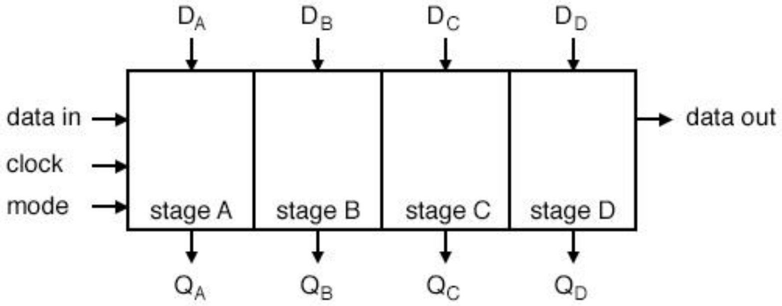
The discrete LED indicators, which we show, could be 7-segment LEDs. Though, there are LSI (Large Scale Integration) devices capable of driving several 7-segment digits.

This device accepts data from a microprocessor in a serial format, driving more LED segments than it has pins by multiplexing the LEDs.

Universal Shift Registers: Parallel-in, Parallel-out

The purpose of the parallel-in/ parallel-out shift register is to take in parallel data, shift it, then output it as shown below.

A universal shift register is a do-everything device in addition to the parallel-in/ parallel-out function.



Parallel-in,parallel-out shift register with 4-stages

Above we apply four bit of data to a parallel-in/ parallel-out shift register at D_A D_B D_C D_D. The mode control, which may be multiple inputs, controls parallel loading vs shifting.

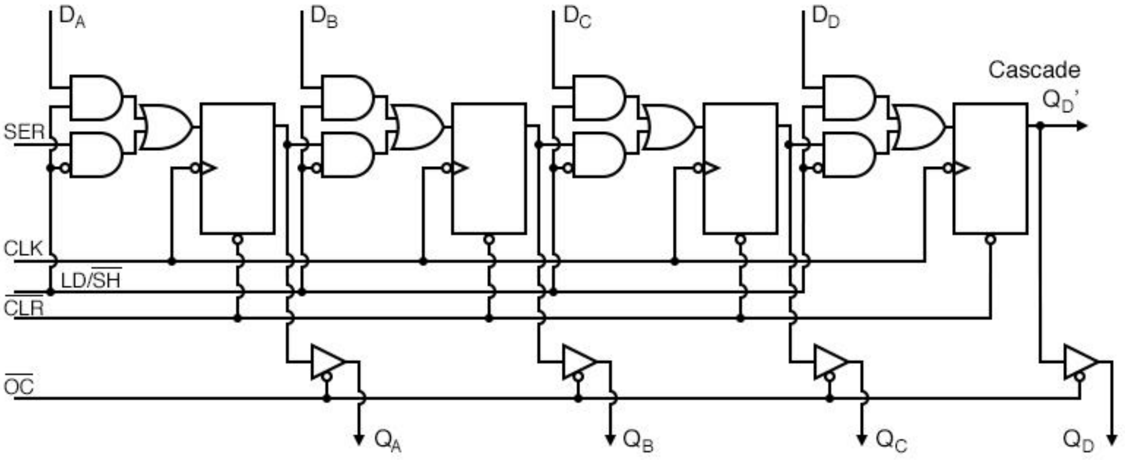
The mode control may also control the direction of shifting in some real devices. The data will be shifted one bit position for each clock pulse.

The shifted data is available at the outputs Q_A Q_B Q_C Q_D . The “data in” and “data out” are provided for cascading of multiple stages.

Though, above, we can only cascade data for right shifting. We could accommodate cascading of left-shift data by adding a pair of left pointing signals, “data in” and “data out”, above.

The internal details of a right shifting parallel-in/ parallel-out shift register are shown below.

The tri-state buffers are not strictly necessary to the parallel-in/ parallel-out shift register, but are part of the real-world device shown below.



74LS395 parallel-in/ parallel-out shift register with tri-state output

The 74LS395 so closely matches our concept of a hypothetical right shifting parallel-in/ parallel-out shift register that we use an overly simplified version of the data sheet details above.

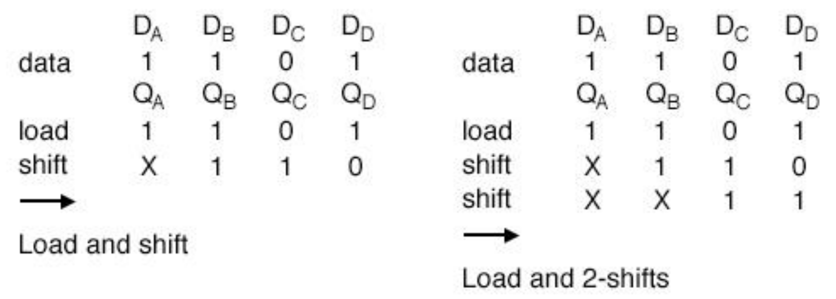
LD/SH' controls the AND-OR multiplexer at the data input to the FF's. If LD/SH'=1, the upper four AND gates are enabled allowing application of parallel inputs D_A D_B D_C D_D to the four FF data inputs.

Note the inverter bubble at the clock input of the four FFs. This indicates that the 74LS395 clocks data on the negative going clock, which is the high to low transition.

The four bits of data will be clocked in parallel from D_A D_B D_C D_D to Q_A Q_B Q_C Q_D at the next negative going clock. In this "real part", OC' must be low if the data needs to be available at the actual output pins as opposed to only on the internal FFs.

The previously loaded data may be shifted right by one bit position if LD/SH'=0 for the succeeding negative going clock edges.

Four clocks would shift the data entirely out of our 4-bit shift register. The data would be lost unless our device was cascaded from Q_D' to SER of another device.



	D_A	D_B	D_C	D_D
data	1	1	0	1
	Q_A	Q_B	Q_C	Q_D
load	1	1	0	1
shift	X	1	1	0

→

Load and shift

	D_A	D_B	D_C	D_D
data	1	1	0	1
	Q_A	Q_B	Q_C	Q_D
load	1	1	0	1
shift	X	1	1	0
shift	X	X	1	1

→

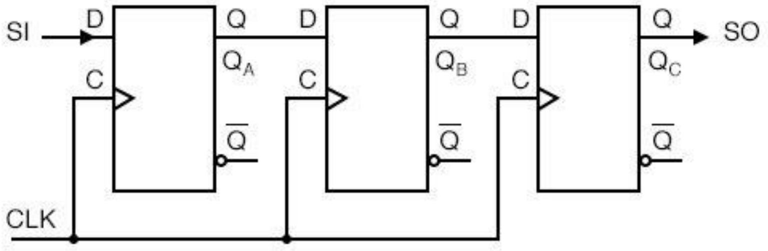
Load and 2-shifts

Parallel-in/ parallel-out shift register

Above, a data pattern is presented to inputs D_A D_B D_C D_D. The pattern is loaded to Q_A Q_B Q_C Q_D . Then it is shifted one bit to the right.

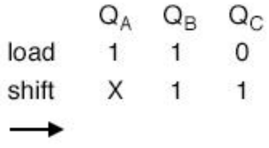
The incoming data is indicated by X, meaning the we do no know what it is. If the input (SER) were grounded, for example, we would know what data (0) was shifted in.

Also shown, is right shifting by two positions, requiring two clocks.



Shift right

The above figure serves as a reference for the hardware involved in right shifting of data.

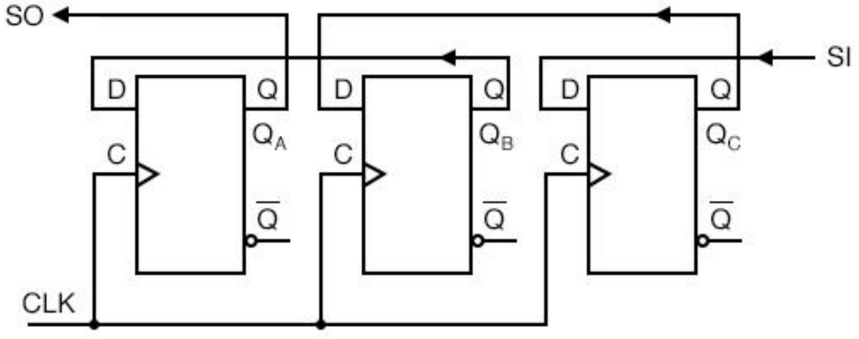


	Q_A	Q_B	Q_C
load	1	1	0
shift	X	1	1

→

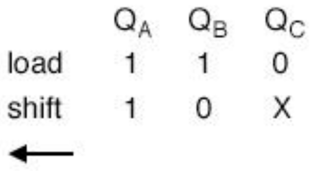
Load and right shift

Right shifting of data is provided above for reference to the previous right shifter.



Shift left

If we need to shift left, the FFs need to be rewired. Compare to the previous right shifter. Also, SI and SO have been reversed. SI shifts to Q_C. Q_C shifts to Q_B. Q_B shifts to Q_A. Q_A leaves on the SO connection, where it could cascade to another shifter SI. This left shift sequence is backwards from the right shift sequence.



	Q_A	Q_B	Q_C
load	1	1	0
shift	1	0	X

←

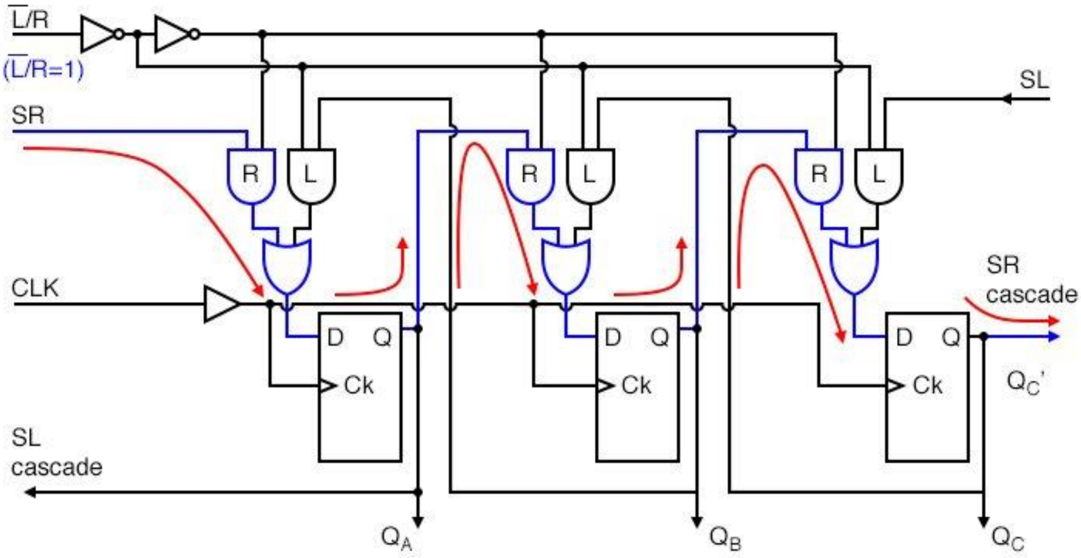
Load and left shift

Above we shift the same data pattern left by one bit.

There is one problem with the “shift left” figure above. Nobody manufactures a shift-left part.

A “real device” which shifts one direction can be wired externally to shift the other direction. Or, should we say there is no left or right in the context of a device which shifts in only one direction.

However, there is a market for a device which will shift left or right on command by a control line. Of course, left and right are valid in that context.



Shift left/ right, right action

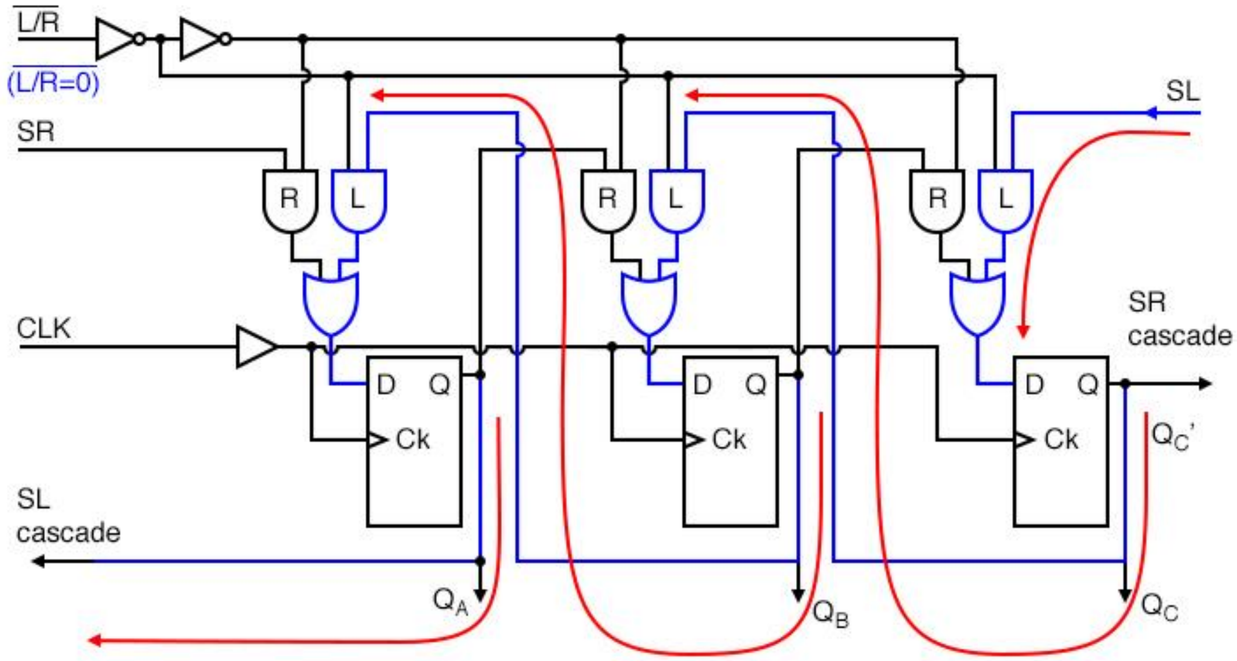
What we have above is a hypothetical shift register capable of shifting either direction under the control of L’/R.

It is setup with L’/R=1 to shift the normal direction, right. L’/R=1 enables the multiplexer AND gates labeled R.

This allows data to follow the path illustrated by the arrows, when a clock is applied. The connection path is the same as the"too simple” “shift right” figure above.

Data shifts in at SR, to Q_A, to Q_B, to Q_C, where it leaves at SR cascade. This pin could drive SR of another device to the right.

What if we change L’/R to L’/R=0?



Shift left/ right register, left action

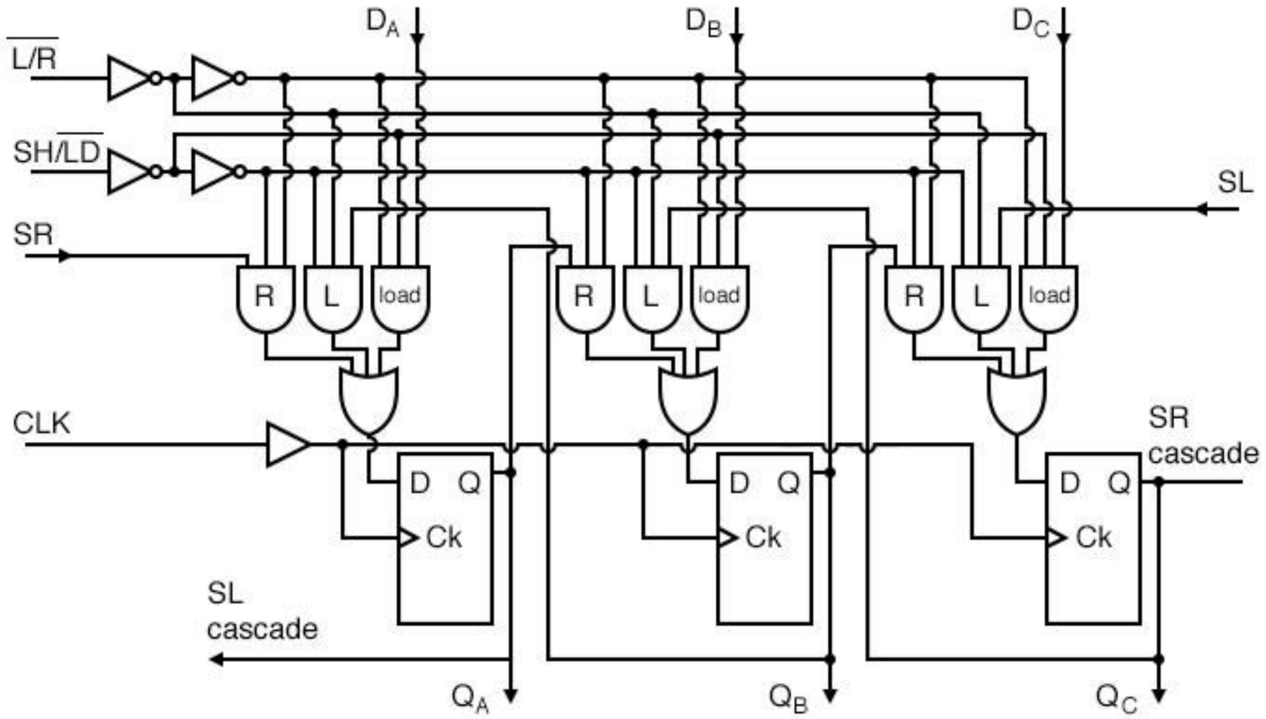
With L’/R=0, the multiplexer AND gates labeled L are enabled, yielding a path, shown by the arrows, the same as the above “shift left” figure.

Data shifts in at SL, to Q_C, to Q_B, to Q_A, where it leaves at SL cascade. This pin could drive SL of another device to the left.

The prime virtue of the above two figures illustrating the “shift left/ right register” is simplicity.

The operation of the left right control L’/R=0 is easy to follow.

The parallel data loading implied by the section title. This appears in the figure below.



Shift left/ right/ load

Now that we can shift both left and right via L’/R, let us add SH/LD’, shift/ load, and the AND gates labeled “load” to provide for parallel loading of data from inputs D_A D_B D_C.

When SH/LD’=0, AND gates R and L are disabled, AND gates “load” are enabled to pass data D_A D_B D_C to the FF data inputs. the next clock CLK will clock the data to Q_A Q_B Q_C.

As long as the same data is present it will be re-loaded on succeeding clocks. However, data present for only one clock will be lost from the outputs when it is no longer present on the data inputs.

One solution is to load the data on one clock, then proceed to shift on the next four clocks. This problem is remedied in the 74ALS299 by the addition of another AND gate to the multiplexer.

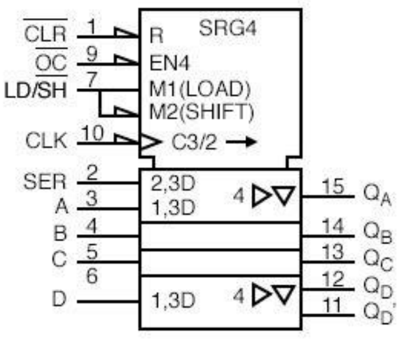
If SH/LD’ is changed to SH/LD’=1, the AND gates labeled “load” are disabled, allowing the left/ right control L’/R to set the direction of shift on the L or R AND gates. Shifting is as in the previous figures.

The only thing needed to produce a viable integrated device is to add the fourth AND gate to the multiplexer as alluded for the 74ALS299.

Parallel-in/ parallel-out and universal devices

Let’s take a closer look at Serial-in/ parallel-out shift registers available as integrated circuits.

- SN74LS395A parallel-in/ parallel-out 4-bit shift register
- SN74ALS299 parallel-in/ parallel-out 8-bit universal shift register



SN74LS395A ANSI Symbol

We have already looked at the internal details of the SN74LS395A, see above previous figure, 74LS395 parallel-in/ parallel-out shift register with tri-state output.

Directly above is the ANSI symbol for the 74LS395.

Why only 4-bits, as indicated by SRG4 above? Having both parallel inputs, and parallel outputs, in addition to control and power pins, does not allow for any more I/O (Input/Output) bits in a 16-pin DIP (Dual Inline Package).

R indicates that the shift register stages are reset by input CLR’ (active low-inverting half arrow at input) of the control section at the top of the symbol. OC’, when low, (invert arrow again) will enable (EN4) the four tristate output buffers (QA QB QC QD) in the data section.

Load/shift’ (LD/SH’) at pin (7) corresponds to internals M1 (load) and M2 (shift). Look for prefixes of 1 and 2 in the rest of the symbol to ascertain what is controlled by these.

The negative edge sensitive clock (indicated by the invert arrow at pin-10) C3/2has two functions.

First, the 3 of C3/2 affects any input having a prefix of 3, say 2,3D or 1,3D in the data section.

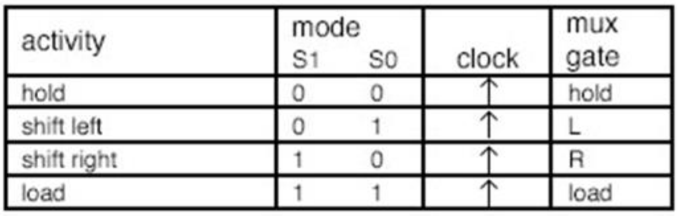
This would be parallel load at A, B, C, D attributed to M1 and C3 for 1,3D. Second, 2 of C3/2-right-arrow indicates data clocking wherever 2 appears in a prefix (2,3D at pin-2).

Thus we have clocking of data at SER into QA with mode 2. The right arrow after C3/2 accounts for shifting at internal shift register stages QA QB QC QD.

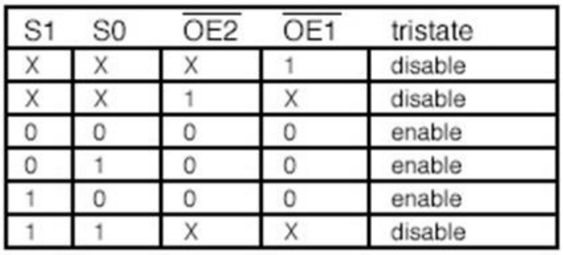
The right pointing triangles indicate buffering; the inverted triangle indicates tri-state, controlled by the EN4.

Note, all the 4s in the symbol associated with the EN are frequently omitted. Stages QB QC are understood to have the same attributes as QD. QD' cascades to the next package's SER to the right.

The table below, condensed from the data '299 data sheet, summarizes the operation of the 74ALS299 universal shift/ storage register.



activity	mode		clock	mux
	S1	S0		gate
hold	0	0	↑	hold
shift left	0	1	↑	L
shift right	1	0	↑	R
load	1	1	↑	load



S1	S0	$\overline{OE2}$	$\overline{OE1}$	tristate
X	X	X	1	disable
X	X	1	X	disable
0	0	0	0	enable
0	1	0	0	enable
1	0	0	0	enable
1	1	X	X	disable

The Multiplexer gates R, L, load operate as in the previous "shift left/ right register" figures.

The difference is that the mode inputs S1 and S0 select shift left, shift right, and load with mode set to S1 S0 = to 01, 10, and 11respectively as shown in the table, enabling multiplexer gates L, R, and load respectively.

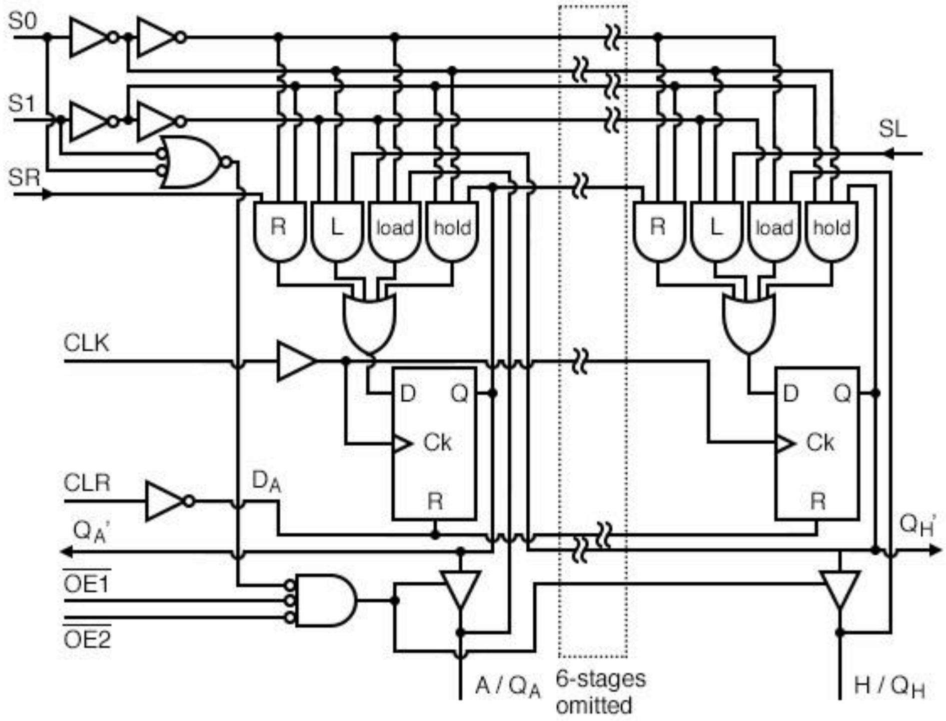
See table. A minor difference is the parallel load path from the tri-state outputs. Actually the tri-state buffers are (must be) disabled by S1 S0 = 11 to float the I/O bus for use as inputs.

A bus is a collection of similar signals. The inputs are applied to A, B through H (same pins as QA, QB, through QH) and routed to the load gate in the multiplexers, and on the the D inputs of the FFs. Data is parallel load on a clock pulse.

The one new multiplexer gate is the AND gate labeled hold, enabled by S1 S0 = 00. The hold gate enables a path from the Q output of the FF back to the hold gate, to the D input of the same FF. The result is that with mode S1 S0 = 00, the output is continuously re-loaded with each new clock pulse. Thus, data is held. This is summarized in the table.

To read data from outputs QA, QB, through QH, the tri-state buffers must be enabled by OE2’, OE1’ =00 and mode =S1 S0 = 00, 01, or 10.

That is, mode is anything except load. See second table.



74ALS299 universal shift/ storage register with tri-state outputs

Right shift data from a package to the left, shifts in on the SR input. Any data shifted out to the right from stage Q_H cascades to the right via Q_H’.

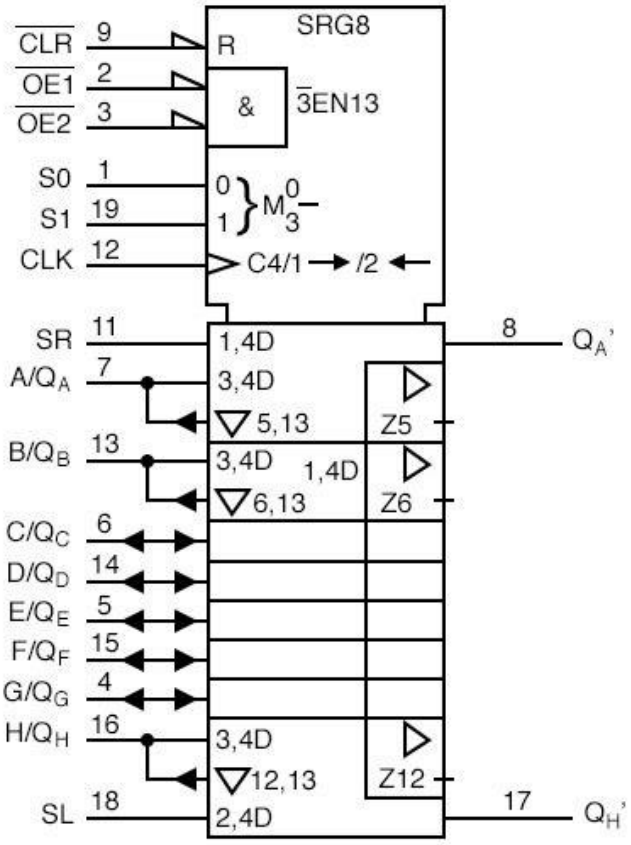
This output is unaffected by the tri-state buffers. The shift right sequence for S1 S0 = 10 is:

$$SR > Q_A > Q_B > Q_C > Q_D > Q_E > Q_F > Q_G > Q_H\ (Q_H')$$

Left shift data from a package to the right shifts in on the SL input. Any data shifted out to the left from stage Q_A cascades to the left via Q_A’, also unaffected by the tri-state buffers. The shift left sequence for S1 S0 = 01 is:

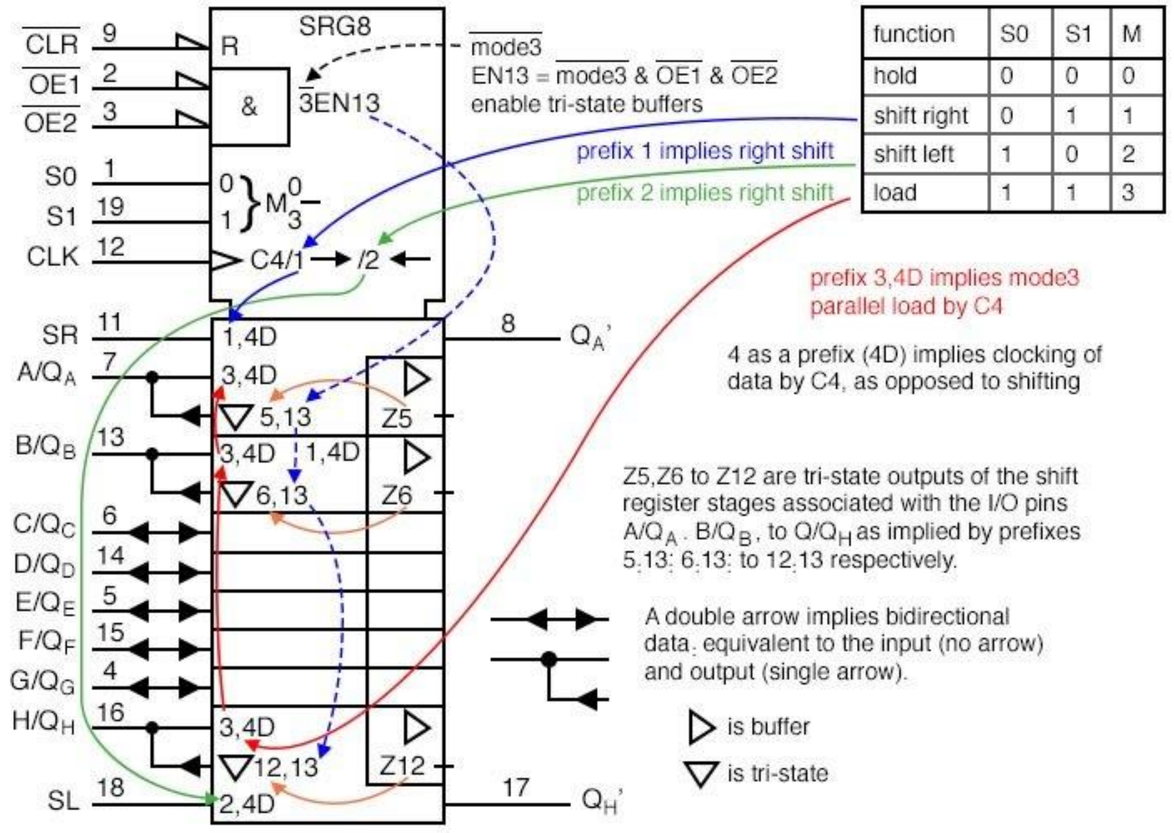
$$(Q_A')\ Q_A < Q_B < Q_C < Q_D < Q_E < Q_F < Q_G < Q_H\ (Q_{SL}')$$

Shifting may take place with the tri-state buffers disabled by one of OE2’ or OE1’ = 1. Though, the register contents outputs will not be accessible. See table.



SN74ALS299 ANSI Symbol

The “clean” ANSI symbol for the SN74ALS299 parallel-in/ parallel-out 8-bit universal shift register with tri-state output is shown for reference above.



EN13 = mode3 & OE1 & OE2 enable tri-state buffers

prefix 1 implies right shift

prefix 2 implies right shift

prefix 3,4D implies mode3 parallel load by C4

4 as a prefix (4D) implies clocking of data by C4, as opposed to shifting

Z5,Z6 to Z12 are tri-state outputs of the shift register stages associated with the I/O pins A/Q_A . B/Q_B , to Q/Q_H as implied by prefixes 5.13: 6.13: to 12.13 respectively.

A double arrow implies bidirectional data. equivalent to the input (no arrow) and output (single arrow).

function	S0	S1	M
hold	0	0	0
shift right	0	1	1
shift left	1	0	2
load	1	1	3

SN74ALS299 ANSI Symbol, annotated

The annotated version of the ANSI symbol is shown to clarify the terminology contained therein.

Note that the ANSI mode (S0 S1) is reversed from the order (S1 S0) used in the previous table.

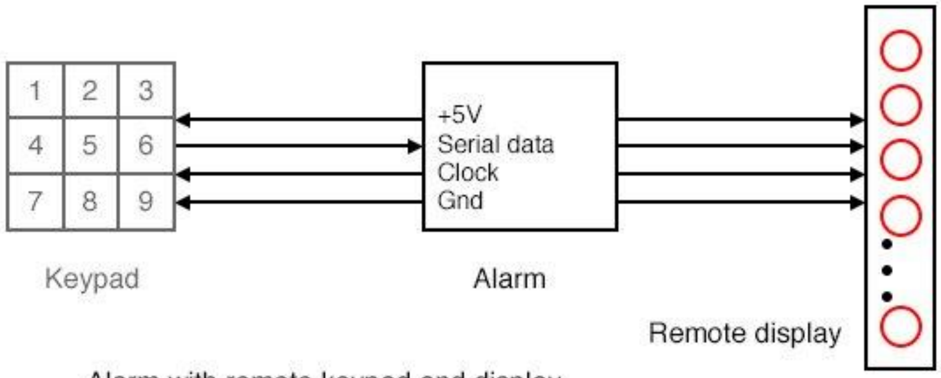
That reverses the decimal mode numbers (1 & 2).

Practical applications

The Alarm with remote keypad block diagram is repeated below. Previously, we built the keypad reader and the remote display as separate units.

Now we will combine both the keypad and display into a single unit using a universal shift register.

Though separate in the diagram, the Keypad and Display are both contained within the same remote enclosure.

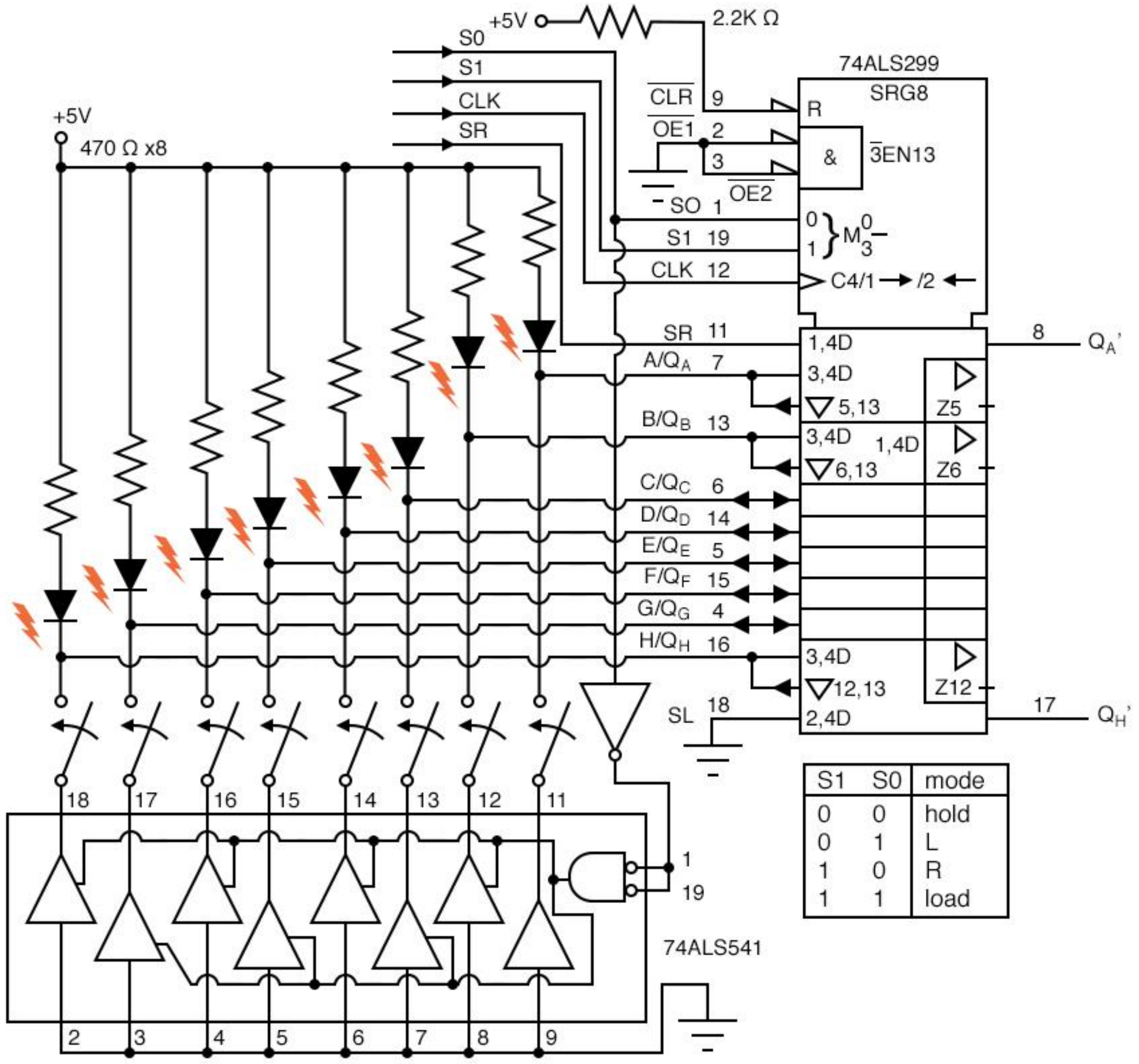


Alarm with remote keypad and display

We will parallel load the keyboard data into the shift register on a single clock pulse, then shift it out to the main alarm box.

At the same time , we will shift LED data from the main alarm to the remote shift register to illuminate the LEDs.

We will be simultaneously shifting keyboard data out and LED data into the shift register.



S1	S0	mode
0	0	hold
0	1	L
1	0	R
1	1	load

74ALS299 universal shift register reads switches, drives LEDs

Eight LEDs and current limiting resistors are connected to the eight I/O pins of the 74ALS299 universal shift register.

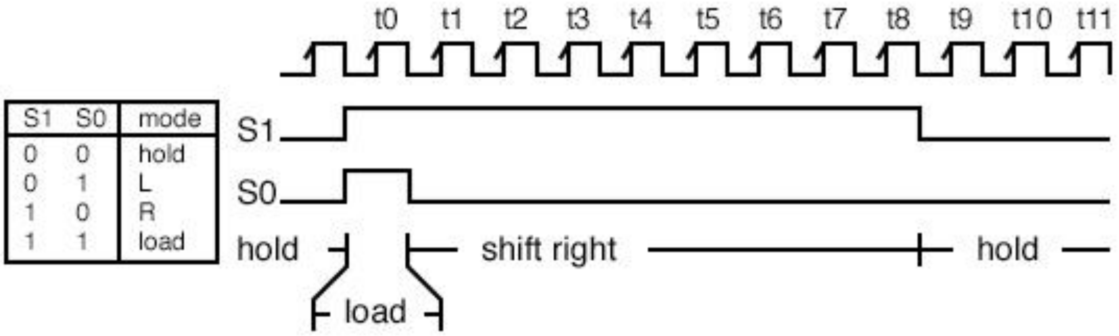
The LEDS can only be driven during Mode 3 with S1=0 S0=0. The OE1’ and OE2’ tristate enables are grounded to permenantly enable the tristate outputs during modes 0, 1, 2.

That will cause the LEDS to light (flicker) during shifting. If this were a problem the EN1’ and EN2’ could be ungrounded and paralleled with S1 and S0 respectively to only enable the tristate buffers and light the LEDS during hold, mode 3. Let’s keep it simple for this example.

During parallel loading, S0=1 inverted to a 0, enables the octal tristate buffers to ground the switch wipers.

The upper, open, switch contacts are pulled up to logic high by the resister-LED combination at the eight inputs.

Any switch closure will short the input low. We parallel load the switch data into the ‘299 at clock t0 when both S0 and S1 are high. See waveforms below.



S1	S0	mode
0	0	hold
0	1	L
1	0	R
1	1	load

Load (t0) & shift (t1-t8) switches out of Q_H', shift LED data into SR

Once S0 goes low, eight clocks (t0 tot8) shift switch closure data out of the ‘299 via the Qh’ pin.

At the same time, new LED data is shifted in at SR of the 299 by the same eight clocks. The LED data replaces the switch closure data as shifting proceeds.

After the 8th shift clock, t8, S1 goes low to yield hold mode (S1 S0 = 00). The data in the shift register remains the same even if there are more clocks, for example, t9, t10, etc.

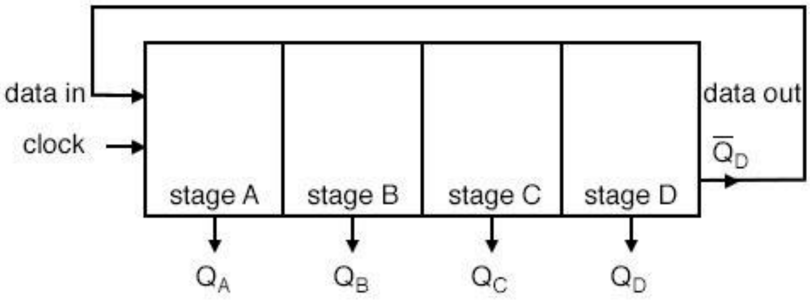
Where do the waveforms come from? They could be generated by a microprocessor if the clock rate were not over 100 kHz, in which case, it would be inconvenient to generate any clocks after t8.

If the clock was in the megahertz range, the clock would run continuously. The clock, S1 and S0 would be generated by digital logic, not shown here.

Ring Counters

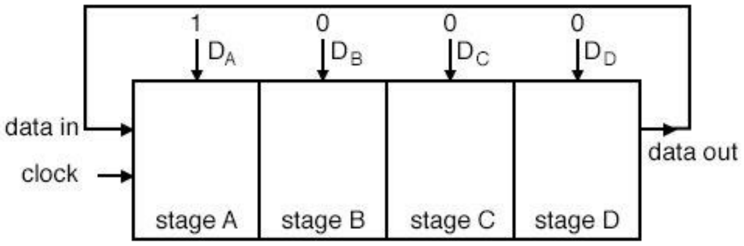
If the output of a shift register is fed back to the input. a ring counter results. The data pattern contained within the shift register will recirculate as long as clock pulses are applied.

For example, the data pattern will repeat every four clock pulses in the figure below. However, we must load a data pattern. All 0‘s or all 1‘s doesn’t count. Is a continuous logic level from such a condition useful?



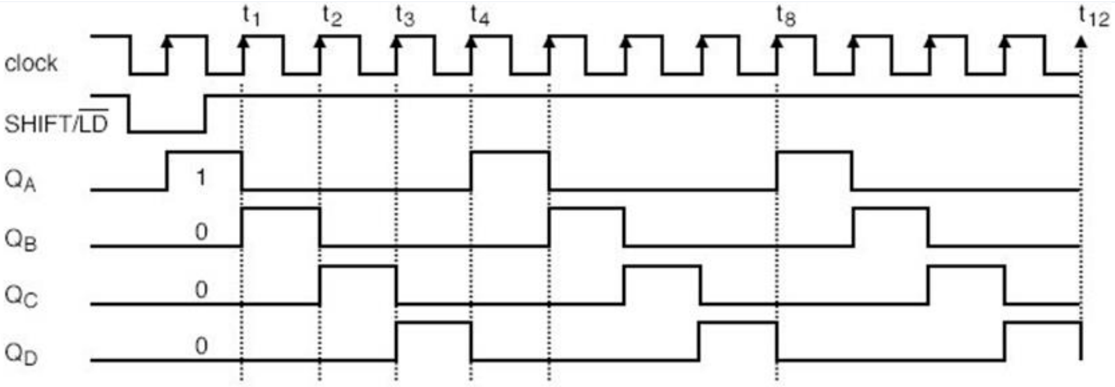
Ring Counter: shift register output fed back to input

We make provisions for loading data into the parallel-in/ serial-out shift register configured as a ring counter below. Any random pattern may be loaded. The most generally useful pattern is a single 1.



Parallel-in, serial-out shift register configured as a ring counter

Loading binary 1000 into the ring counter, above, prior to shifting yields a viewable pattern. The data pattern for a single stage repeats every four clock pulses in our 4-stage example. The waveforms for all four stages look the same, except for the one clock time delay from one stage to the next. See figure below.

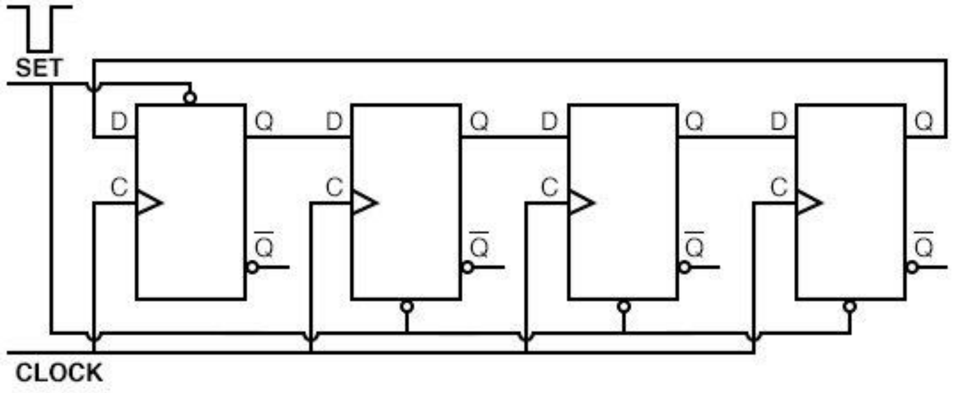


Load 1000 into 4-stage ring counter and shift

The circuit above is a divide by 4 counter. Comparing the clock input to any one of the outputs, shows a frequency ratio of 4:1.

Q: How may stages would we need for a divide by 10 ring counter?

A: Ten stages would recirculate the 1 every 10 clock pulses.



Set one stage, clear three stages

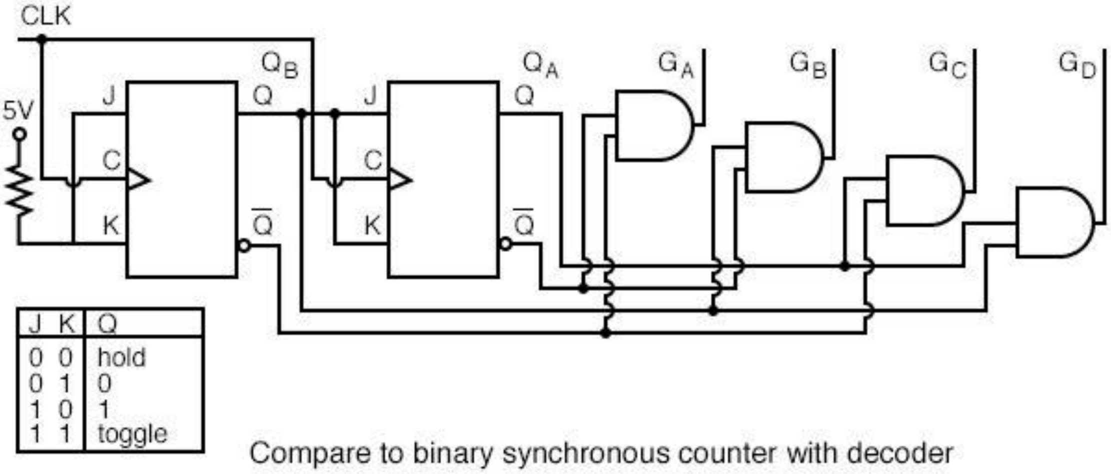
An alternate method of initializing the ring counter to 1000 is shown above. The shift waveforms are identical to those above, repeating every fourth clock pulse.

The requirement for initialization is a disadvantage of the ring counter over a conventional counter.

At a minimum, it must be initialized at power-up since there is no way to predict what state flip-flops will power up in.

In theory, initialization should never be required again. In actual practice, the flip-flops could eventually be corrupted by noise, destroying the data pattern.

A "self correcting" counter, like a conventional synchronous binary counter would be more reliable.



J	K	Q
0	0	hold
0	1	0
1	0	1
1	1	toggle

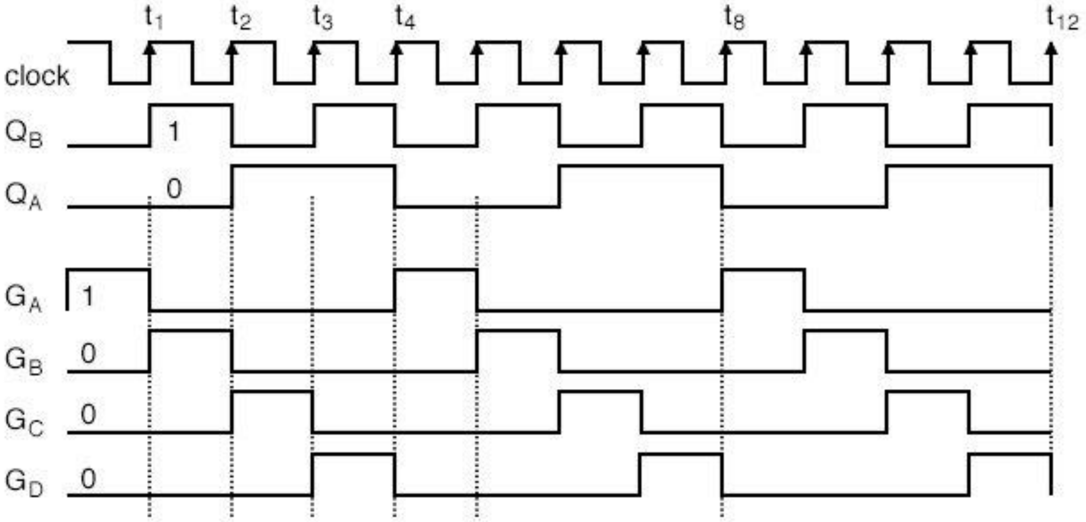
Compare to binary synchronous counter with decoder

The above binary synchronous counter needs only two stages, but requires decoder gates.

The ring counter had more stages, but was self decoding, saving the decode gates above.

Another disadvantage of the ring counter is that it is not “self starting”.

If we need the decoded outputs, the ring counter looks attractive, in particular, if most of the logic is in a single shift register package. If not, the conventional binary counter is less complex without the decoder.



Compare to binary synchronous counter with decode, waveforms

The waveforms decoded from the synchronous binary counter are identical to the previous ring counter waveforms.

The counter sequence is (Q_A Q_B) = (00 01 10 11).

Johnson Counters

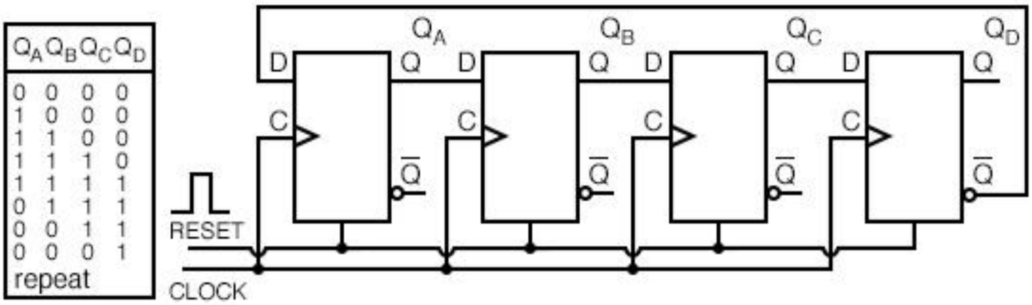
The *switch-tail ring counter*, also know as the *Johnson counter*, overcomes some of the limitations of the ring counter.

Like a ring counter a Johnson counter is a shift register fed back on its' self. It requires half the stages of a comparable ring counter for a given division ratio.

If the complement output of a ring counter is fed back to the input instead of the true output, a Johnson counter results.

The difference between a ring counter and a Johnson counter is which output of the last stage is fed back (Q or Q').

Carefully compare the feedback connection below to the previous ring counter.



Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
repeat			

Johnson counter (note the $\overline{Q}_D$ to D_A feedback connection)

This "reversed" feedback connection has a profound effect upon the behavior of the otherwise similar circuits.

Recirculating a single 1 around a ring counter divides the input clock by a factor equal to the number of stages.

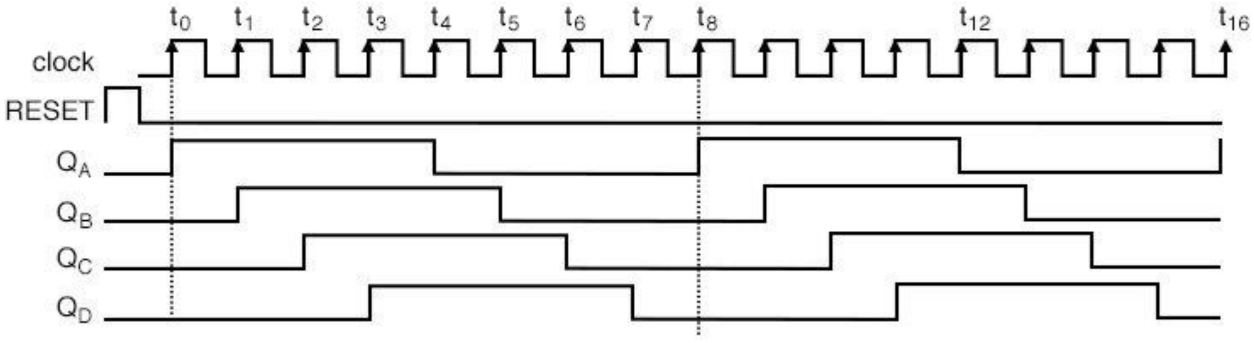
Whereas, a Johnson counter divides by a factor equal to twice the number of stages.

For example, a 4-stage ring counter divides by 4. A 4-stage Johnson counter divides by 8.

Start a Johnson counter by clearing all stages to 0s before the first clock. This is often done at power-up time.

Referring to the figure below, the first clock shifts three 0s from (Q_A Q_B Q_C) to the right into (Q_B Q_C Q_D). The 1 at Q_D' (the complement of Q) is shifted back into Q_A.

Thus, we start shifting 1s to the right, replacing the 0s. Where a ring counter recirculated a single 1, the 4-stage Johnson counter recirculates four 0s then four 1s for an 8-bit pattern, then repeats.



Four stage Johnson counter waveforms

The above waveforms illustrates that multi-phase square waves are generated by a Johnson counter.

The 4-stage unit above generates four overlapping phases of 50% duty cycle. How many stages would be required to generate a set of three phase waveforms?

For example, a three stage Johnson counter, driven by a 360 Hertz clock would generate three 120o phased square waves at 60 Hertz.

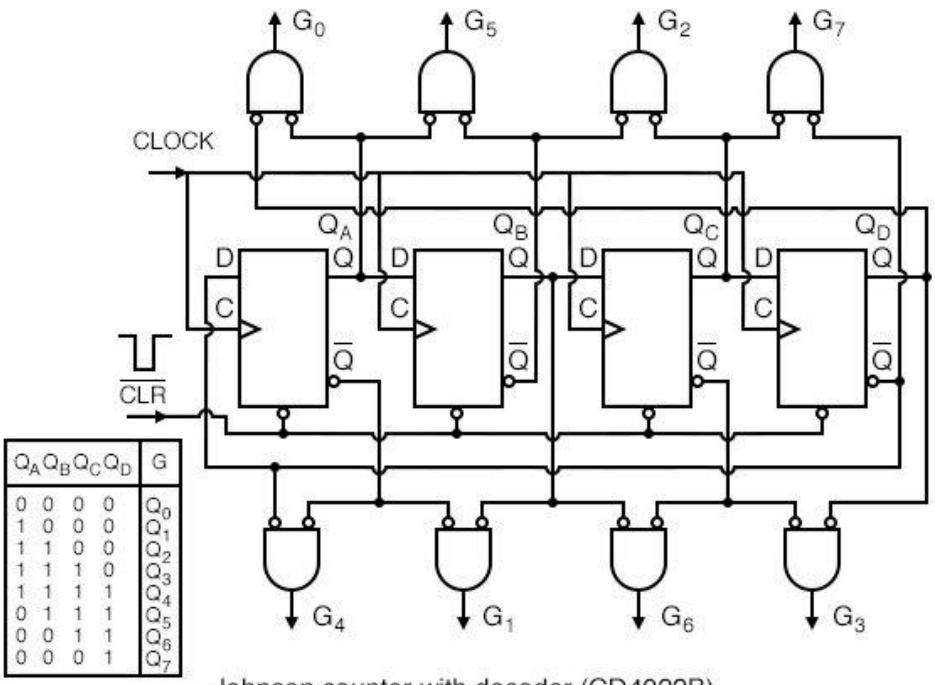
The outputs of the flop-flops in a Johnson counter are easy to decode to a single state.

Below for example, the eight states of a 4-stage Johnson counter are decoded by no more than a two input gate for each of the states.

In our example, eight of the two input gates decode the states for our example Johnson counter.

No matter how long the Johnson counter, only 2-input decoder gates are needed.

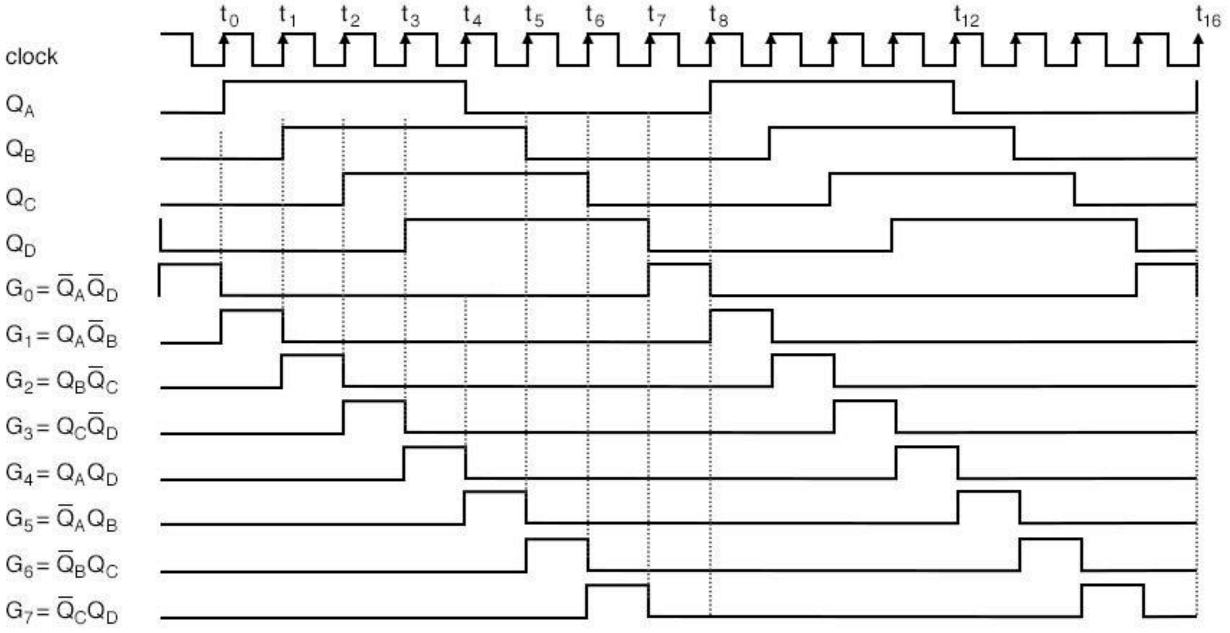
Note, we could have used uninverted inputs to the AND gates by changing the gate inputs from true to inverted at the FFs, Q to Q’, (and vice versa).



Q_A	Q_B	Q_C	Q_D	G
0	0	0	0	Q_0
1	0	0	0	Q_1
1	1	0	0	Q_2
1	1	1	0	Q_3
1	1	1	1	Q_4
0	1	1	1	Q_5
0	0	1	1	Q_6
0	0	0	1	Q_7

Johnson counter with decoder (CD4022B)

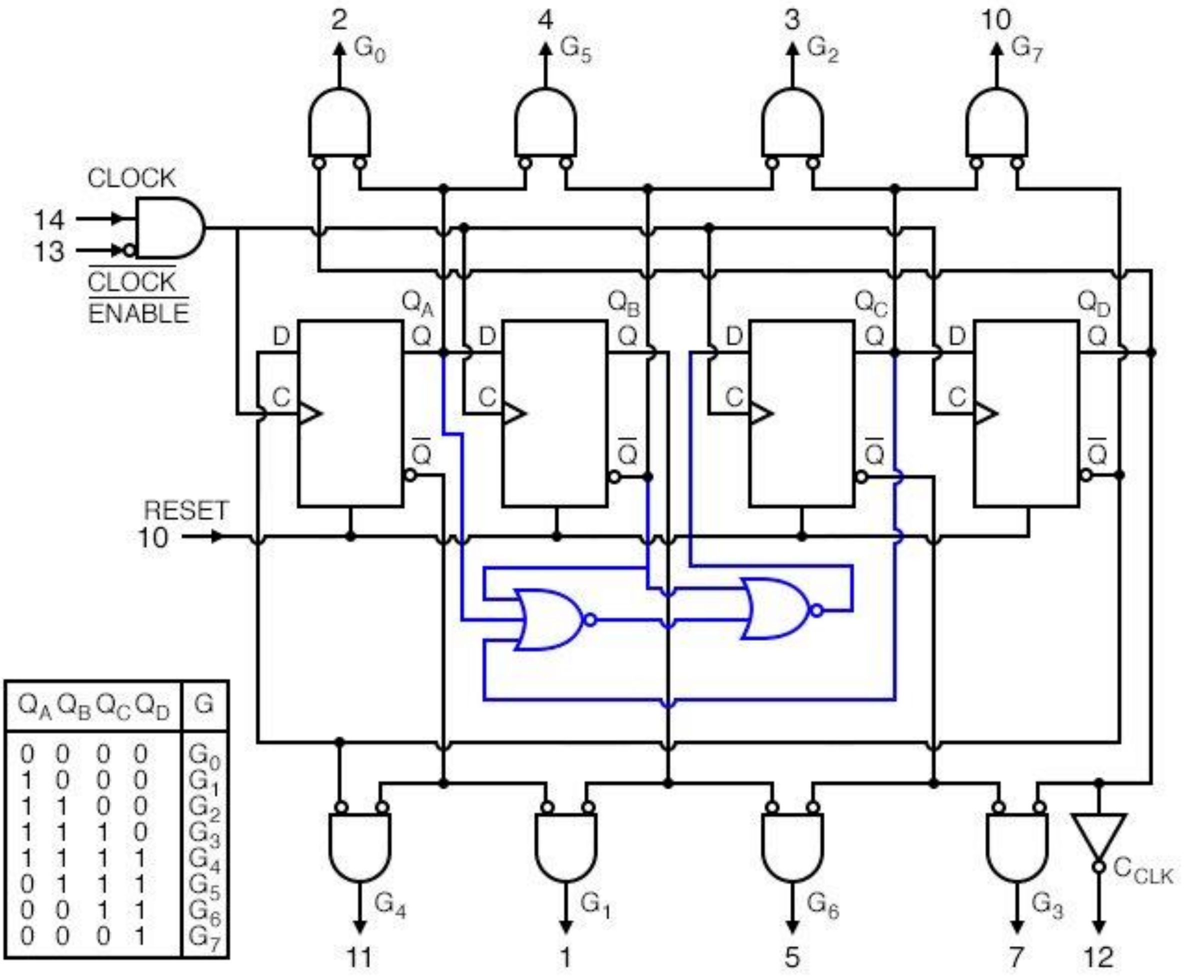
However, we are trying to make the diagram above match the data sheet for the CD4022B, as closely as practical.



Four stage (8-state) Johnson counter decoder waveforms

Above, our four phased square waves Q_A to Q_D are decoded to eight signals (G_0 to G_7) active during one clock period out of a complete 8-clock cycle.

For example, G_0 is active high when both Q_A and Q_D are low. Thus, pairs of the various register outputs define each of the eight states of our Johnson counter example.



Q_A	Q_B	Q_C	Q_D	G
0	0	0	0	G_0
1	0	0	0	G_1
1	1	0	0	G_2
1	1	1	0	G_3
1	1	1	1	G_4
0	1	1	1	G_5
0	0	1	1	G_6
0	0	0	1	G_7

NOR gate unused state detector: Q_A Q_B Q_C = 010 forces the **1** to a **0**

CD4022B modulo-8 Johnson counter with unused state detector

Above is the more complete internal diagram of the CD4022B Johnson counter. See the manufacturers’ data sheet for minor details omitted.

The major new addition to the diagram as compared to previous figures is the *disallowed state detector* composed of the two NOR gates.

Take a look at the inset state table. There are 8-permissible states as listed in the table.

Since our shifter has four flip-flops, there are a total of 16-states, of which there are 8-disallowed states. That would be the ones not listed in the table.

In theory, we will not get into any of the disallowed states as long as the shift register is RESET before first use.

However, in the “real world” after many days of continuous operation due to unforeseen noise, power line disturbances, near lightning strikes, etc, the Johnson counter could get into one of the disallowed states.

For high reliability applications, we need to plan for this slim possibility. More serious is the case where the circuit is not cleared at power-up.

In this case there is no way to know which of the 16-states the circuit will power up in.

Once in a disallowed state, the Johnson counter will not return to any of the permissible states without intervention. That is the purpose of the NOR gates.

Examine the table for the sequence (Q_A Q_B Q_C) = (010). Nowhere does this sequence appear in the table of allowed states.

Therefore (010) is disallowed. It should never occur. If it does, the Johnson counter is in a disallowed state, which it needs to exit to any allowed state.

Suppose that (Q_A Q_B Q_C) = (010). The second NOR gate will replace Q_B = 1 with a 0 at the D input to FF Q_C.

In other words, the offending 010 is replaced by 000. And 000, which does appear in the table, will be shifted right.

There are may triple-0 sequences in the table. This is how the NOR gates get the Johnson counter out of a disallowed state to an allowed state.

Not all disallowed states contain a 010 sequence. However, after a few clocks, this sequence will appear so that any disallowed states will eventually be escaped.

If the circuit is powered-up without a RESET, the outputs will be unpredictable for a few clocks until an allowed state is reached.

If this is a problem for a particular application, be sure to RESET on power-up.

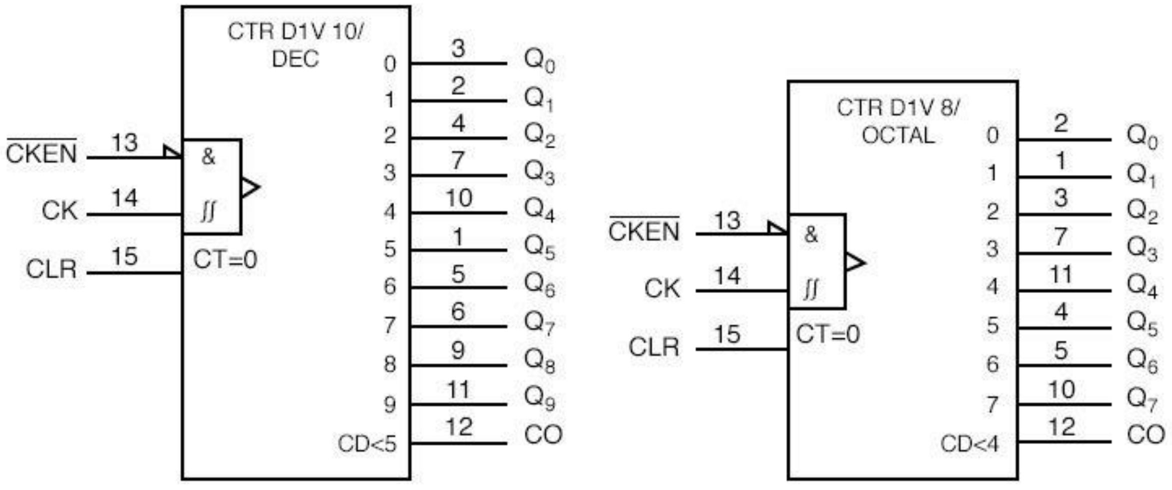
Johnson Counter Devices

A pair of integrated circuit Johnson counter devices with the output states decoded is available.

We have already looked at the CD4017 internal logic in the discussion of Johnson counters.

The 4000 series devices can operate from 3V to 15V power supplies. The the 74HC’ part, designed for a TTL compatiblity, can operate from a 2V to 6V supply, count faster, and has greater output drive capability.

- CD4017 Johnson counter with 10 decoded outputs CD4022 Johnson counter with 8 decoded outputs
- 74HC4017 Johnson counter, 10 decoded outputs



CD4017B, 74HC4017

CD4022B

The ANSI symbols for the *modulo-10* (divide by 10) and modulo-8 Johnson counters are shown above.

The symbol takes on the characteristics of a counter rather than a shift register derivative, which it is.

Waveforms for the CD4022 modulo-8 and operation were shown previously. The CD4017B/ 74HC4017 decade counter is a 5-stage Johnson counter with ten decoded outputs.

The operation and waveforms are similar to the CD4017. In fact, the CD4017 and CD4022 are both detailed on the same data sheet.

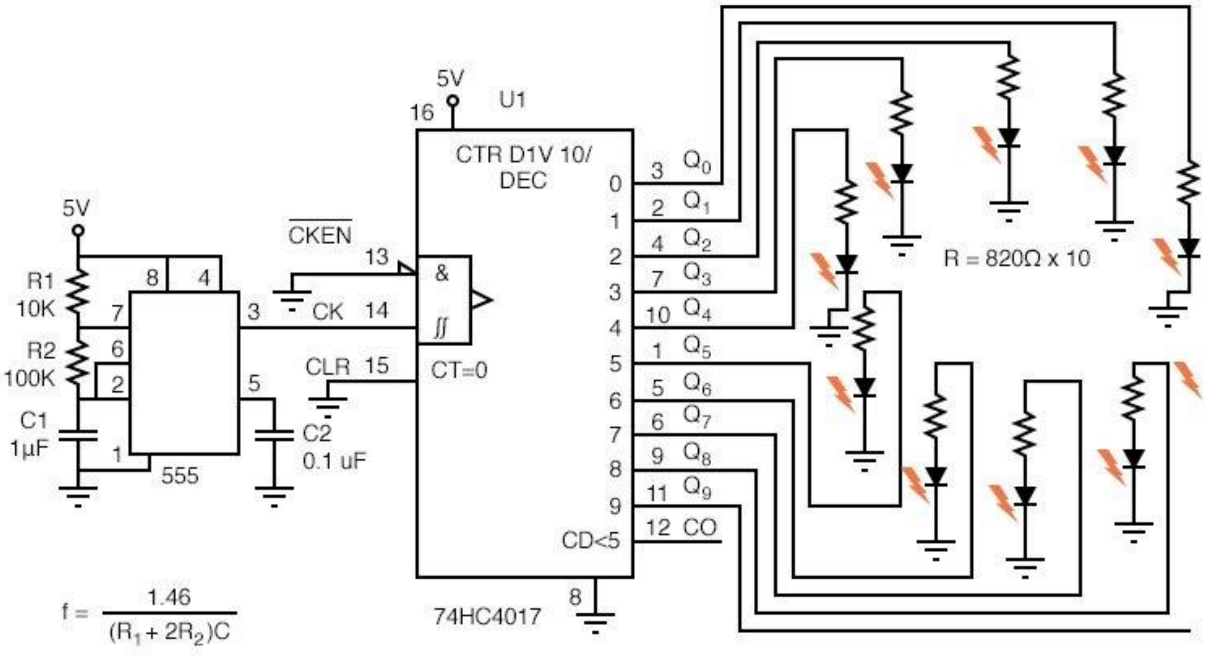
The 74HC4017 is a more modern version of the decade counter.

These devices are used where decoded outputs are needed instead of the binary or BCD (Binary Coded Decimal) outputs found on normal counters.

By decoded, we mean one line out of the ten lines is active at a time for the ‘4017 in place of the four bit BCD code out of conventional counters.

See previous waveforms for 1-of-8 decoding for the ‘4022 Octal Johnson counter.

Practical Applications



Decoded ring counter drives walking LED

The above Johnson counter shifts a lighted LED each fifth of a second around the ring of ten.

Note that the 74HC4017 is used instead of the ‘40017 because the former part has more current drive capability.

From the data sheet, operating at VCC= 5V, the VOH= 4.6V at 4ma.

In other words, the outputs can supply 4 ma at 4.6 V to drive the LEDs. Keep in mind that LEDs are normally driven with 10 to 20 ma of current.

Though, they are visible down to 1 ma. This simple circuit illustrates an application of the ‘HC4017.

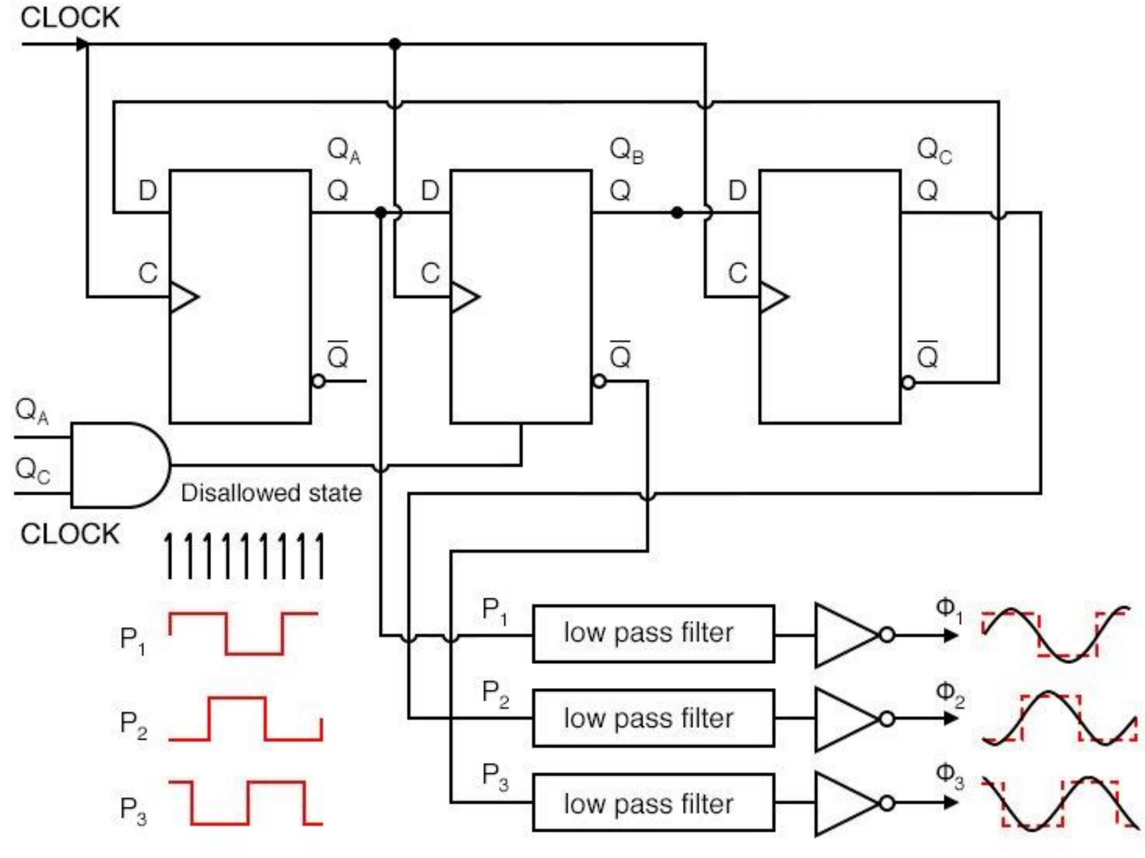
Need a bright display for an exhibition? Then, use inverting buffers to drive the cathodes of the LEDs pulled up to the power supply by lower value anode resistors.

The 555 timer, serving as an astable multivibrator, generates a clock frequency determined by R1 R2 C1.

This drives the 74HC4017 a step per clock as indicated by a single LED illuminated on the ring.

Note, if the 555 does not reliably drive the clock pin of the ‘4015, run it through a single buffer stage between the 555 and the ‘4017.

A variable R2 could change the step rate. The value of decoupling capacitor C2 is not critical. A similar capacitor should be applied across the power and ground pins of the ‘4017.



Three phase square/ sine wave generator.

The Johnson counter above generates 3-phase square waves, phased 60^o apart with respect to (Q_A Q_B Q_C).

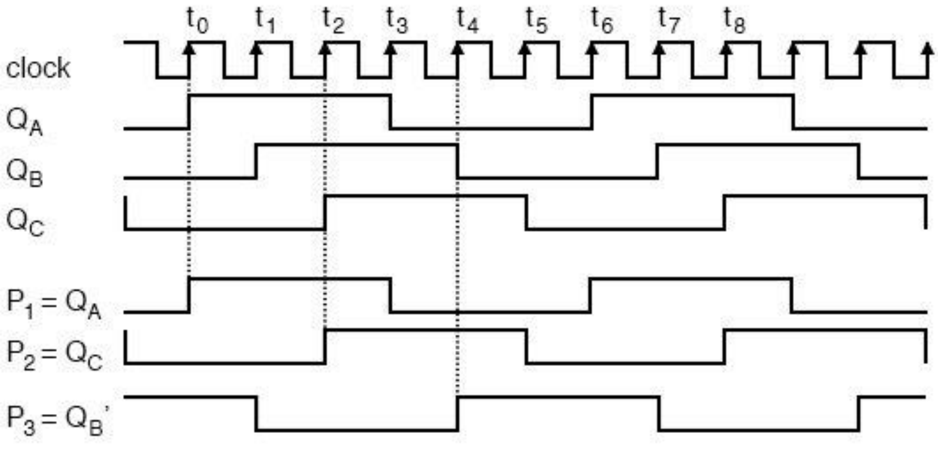
However, we need 120^o phased waveforms of power applications.

Choosing P_1=Q_A P_2=Q_C P_3=Q_B’ yields the 120^o phasing desired. See figure below.

If these (P_1 P_2 P_3) are low-pass filtered to sine waves and amplified, this could be the beginnings of a 3-phase power supply.

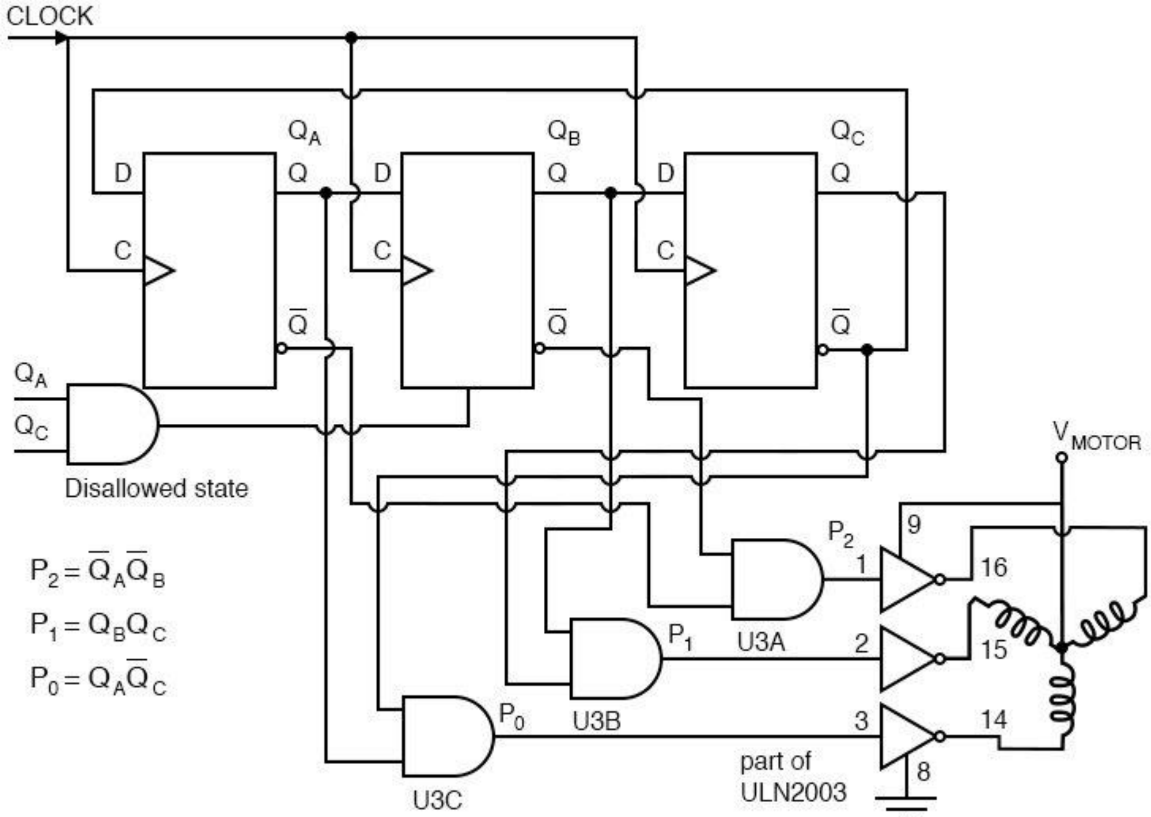
For example, do you need to drive a small 3-phase 400 Hz aircraft motor?

Then, feed 6x 400Hz to the above circuit CLOCK. Note that all these waveforms are 50% duty cycle.



3-stage Johnson counter generates 3-Φ waveform

The circuit below produces 3-phase nonoverlapping, less than 50% duty cycle, waveforms for driving 3-phase stepper motors.

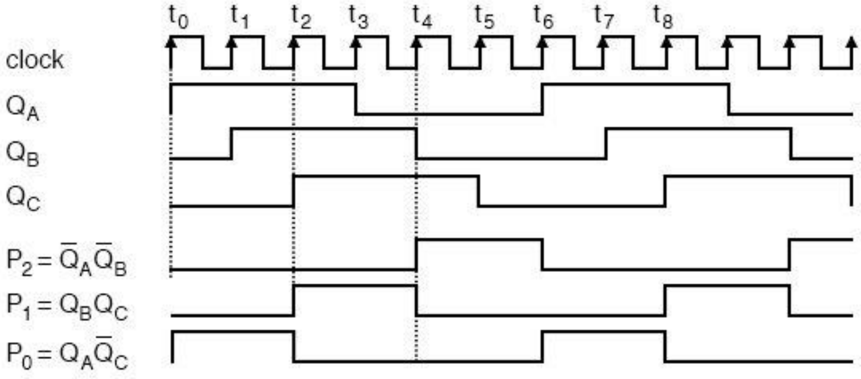


3-stage (6-stage) Johnson counter decoded for 3-Φ stepper motor.

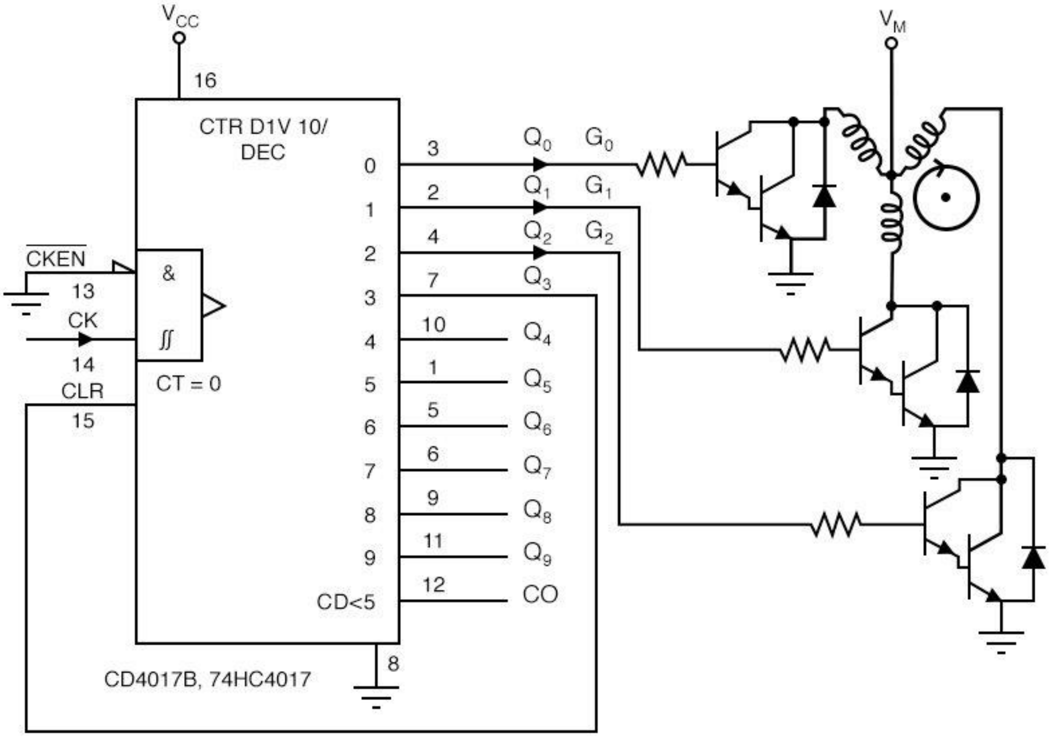
Above we decode the overlapping outputs Q_A Q_B Q_C to non-overlapping outputs P_0 P_1 P_2 as shown below.

These waveforms drive a 3-phase stepper motor after suitable amplification from the milliamp level to the fractional amp level using the ULN2003 drivers shown above, or the discrete component Darlington pair driver shown in the circuit which follow.

Not counting the motor driver, this circuit requires three IC (Integrated Circuit) packages: two dual type “D” FF packages and a quad NAND gate.



3-stage Johnson counter generates 3-Φ stepper waveform

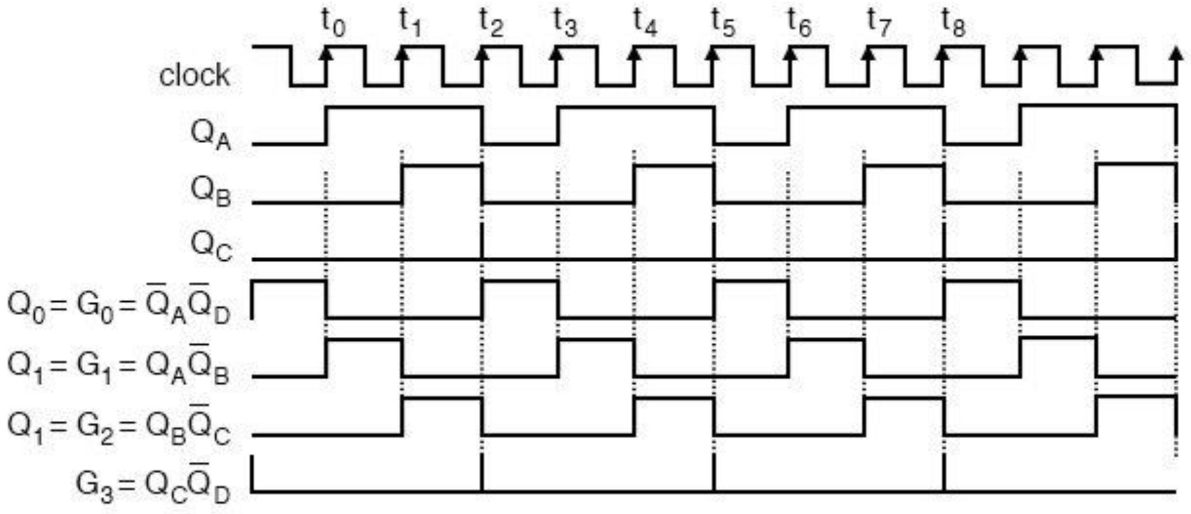


Johnson sequence terminated early by reset at Q_3, which is high for nano seconds

A single CD4017, above, generates the required 3-phase stepper waveforms in the circuit above by clearing the Johnson counter at count 3.

Count 3 persists for less than a microsecond before it clears its' self. The other counts (Q_0=G_0 Q_1=G_1 Q_2=G_2) remain for a full clock period each.

The Darlington bipolar transistor drivers shown above are a substitute for the internal circuitry of the ULN2003.

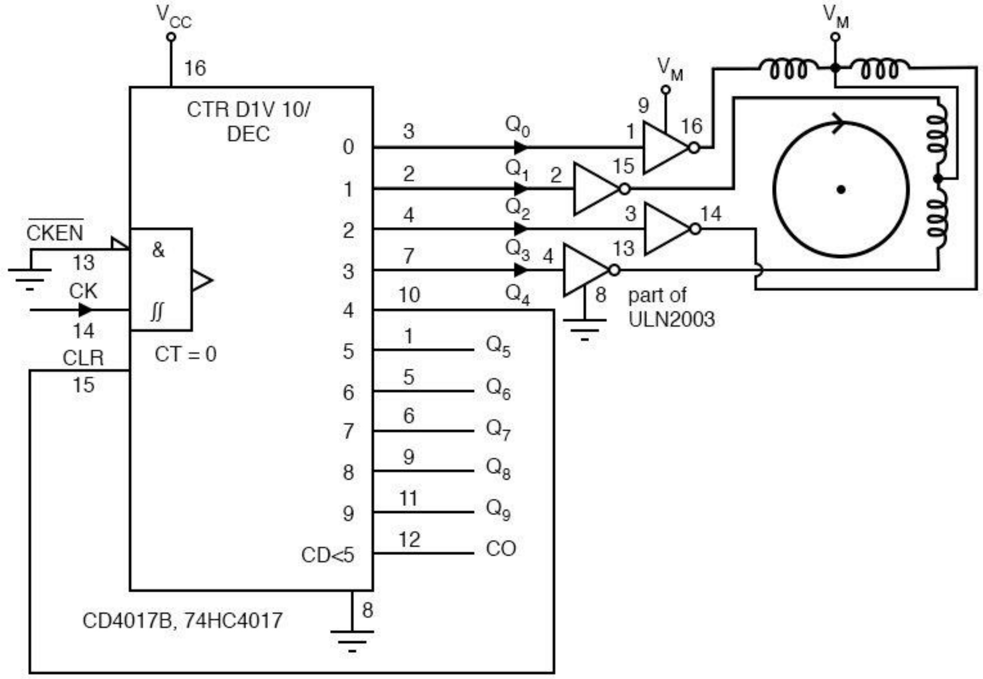


CD4017B 5-stage (10-state) Johnson counter resetting at $Q_CQ_BQ_A$= 100 generates 3-Φ stepper waveform.

The above waveforms make the most sense in the context of the internal logic of the CD4017 shown earlier in this section.

Though, the AND gating equations for the internal decoder are shown. The signals QA QB QC are Johnson counter direct shift register outputs not available on pin-outs.

The QD waveform shows resetting of the ‘4017 every three clocks. Q0 Q1 Q2, etc. are decoded outputs which actually are available at output pins.

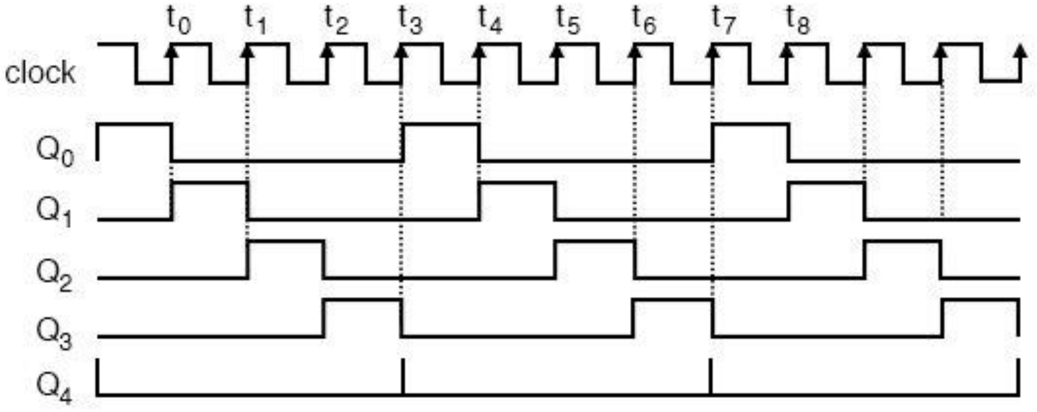


Johnson counter drives unipolar stepper motor.

Above we generate waveforms for driving a *unipolar stepper motor*, which only requires one polarity of driving signal.

That is, we do not have to reverse the polarity of the drive to the windings. This simplifies the power driver between the ‘4017 and the motor.

Darlington pairs from a prior diagram may be substituted for the ULN3003.



Johnson counter unipolar stepper motor waveforms.

Once again, the CD4017B generates the required waveforms with a reset after the teminal count.

The decoded outputs Q_0 Q_1 Q_2 Q_3 sucessively drive the stepper motor windings, with Q_4 reseting the counter at the end of each group of four pulses.

Reference

1. An introduction to computer logic, Nagle H. Troy, Prentice-Hall, Inc., Englewood Cliffs, N.J., Prentice-Hall International, Inc., London.

2. Digital design and computer architecture**,** David M. Harris, Sarah L. Harris, Stanford University**,** Copyrighted Material.

3. Site: https://www.allaboutcircuits.com/textbook/digital/