Binary
Decimal
Octal and
Hexadecimal number systems

A number can be represented with different base values. We are familiar with the numbers in the base 10 (known as decimal numbers), with digits taking values 0,1,2,…,8,9.

A computer uses a Binary number system which has a base 2 and digits can have only TWO values: 0 and 1.

A decimal number with a few digits can be expressed in binary form using a large number of digits. Thus the number 65 can be expressed in binary form as 1000001.

The binary form can be expressed more compactly by grouping 3 binary digits together to form an octal number. An octal number with base 8 makes use of the EIGHT digits 0,1,2,3,4,5,6 and 7.

A more compact representation is used by Hexadecimal representation which groups 4 binary digits together. It can make use of 16 digits, but since we have only 10 digits, the remaining 6 digits are made up of first 6 letters of the alphabet. Thus the hexadecimal base uses 0,1,2,….8,9,A,B,C,D,E,F as digits.

To summarize
Decimal : base 10
Binary : base 2
Octal: base 8
Hexadecimal : base 16

Decimal, Binary, Octal, and Hex Numbers

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversion of binary to decimal (base 2 to base 10)

Each position of binary digit can be replaced by an equivalent power of 2 as shown below.

2^{n-1}	2^{n-2}	……	……	2^3	2^2	2^1	2^0

Thus to convert any binary number replace each binary digit (bit) with its power and add up.

Example: convert $(1011)_2$ to its decimal equivalent

Represent the weight of each digit in the given number using the above table.

2^{n-1}	2^{n-2}	……	……	2^3	2^2	2^1	2^0
				1	0	1	1

Now add up all the powers after multiplying by the digit values, 0 or 1

$(1011)_2$

$= 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

$= 8 + 0 + 2 + 1$

$= 11$

Example2: convert $(1000100)_2$ to its decimal equivalent

$= 2^6 \times 1 + 2^5 \times 0 + 2^4 \times 0 + 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 0$

$= 64 + 0 + 0 + 0 + 4 + 0 + 0$

$= (68)_{10}$

Conversion of decimal to binary (base 10 to base 2)

Here we keep on dividing the number by 2 recursively till it reduces to zero. Then we print the remainders in reverse order.

Example: convert $(68)_{10}$ to binary

68/2 = 34 remainder is 0
34/ 2 = 17 remainder is 0
17 / 2 = 8 remainder is 1
8 / 2 = 4 remainder is 0
4 / 2 = 2 remainder is 0
2 / 2 = 1 remainder is 0
1 / 2 = 0 remainder is 1

We stop here as the number has been reduced to zero and collect the remainders in reverse order.

Answer = 1 0 0 0 1 0 0

Note: the answer is read from bottom (MSB, most significant bit) to top (LSB least significant bit) as $(1000100)_2$.

<u>You should be able to write a recursive function to convert a binary integer into its decimal equivalent.</u>

Conversion of binary fraction to decimal fraction

In a binary fraction, the position of each digit(bit) indicates its relative weight as was the case with the integer part, except the weights to in the reverse direction. Thus after the decimal point, the first digit (bit) has a weight of ½ , the next one has a weight of ¼ , followed by 1/8 and so on.

2^0 .	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…	….	…
.	1	0	1	1	0	0	0

The decimal equivalent of this binary number 0.1011 can be worked out by considering the weight of each bit. Thus in this case it turns out to be

(1/2) x 1 + (1/4) x 0 + (1/8) x 1 + (1/16) x 1.

Conversion of decimal fraction to binary fraction

To convert a decimal fraction to its binary fraction, multiplication by 2 is carried out repetitively and the integer part of the result is saved and placed after the decimal point. The fractional part is taken and multiplied by 2. The process can be stopped any time after the desired accuracy has been achieved.

Example: convert $(0.68)_{10}$ to binary fraction.

0.68 * 2 = 1.36 integer part is 1
Take the fractional part and continue the process
0.36 * 2 = 0.72 integer part is 0
0.72 * 2 = 1.44 integer part is 1
0.44 * 2 = 0.88 integer part is 0

The digits are placed in the order in which they are generated, and not in the reverse order. Let us say we need the accuracy up to 4 decimal places. Here is the result.
Answer = 0. 1 0 1 0…..

Example: convert $(70.68)_{10}$ to binary equivalent.
First convert 70 into its binary form which is 1000110. Then convert 0.68 into binary form upto 4 decimal places to get 0.1010. Now put the two parts together.

Answer = 1 0 0 0 1 1 0 . 1 0 1 0….

Octal Number System

•Base or radix 8 number system.

•1 octal digit is equivalent to 3 bits.

•Octal numbers are 0 to7. (see the chart down below)

•Numbers are expressed as powers of 8. See this table

8^{n-1}	8^{n-2}	……	……	8^3	8^2	8^1	8^0
					6	3	2

Conversion of octal to decimal (base 8 to base 10)

Example: convert $(632)_8$ to decimal

$= (6 \times 8^2) + (3 \times 8^1) + (2 \times 8^0)$

$= (6 \times 64) + (3 \times 8) + (2 \times 1)$

$= 384 + 24 + 2$

$= (410)_{10}$

Conversion of decimal to octal (base 10 to base 8)

Example: convert $(177)_{10}$ to octal equivalent

177 / 8 = 22 remainder is 1
22 / 8 = 2 remainder is 6
2 / 8 = 0 remainder is 2

Answer = 2 6 1

Note: the answer is read from bottom to top as $(261)_8$, the same as with the binary case.

Conversion of decimal fraction to octal fraction is carried out in the same manner as decimal to binary except that now the multiplication is carried out by 8.

Example: convert $(0.523)_{10}$ to octal equivalent up to 3 decimal places

0.523 x 8 = 4.184 ,its integer part is 4
0.184 x 8 = 1.472, its integer part is 1
0.472 x 8 = 3.776 , its integer part is 3

So the answer is $(0.413..)_8$

Conversion of decimal to binary (using octal)

When the numbers are large, conversion to binary would take a large number of division by 2. It can be simplified by first converting the number to octal and then converting each octal into its binary form:

Example: convert $(177)_{10}$ to its binary equivalent using octal form

Step 1: convert it to the octal form first as shown above

This yields $(2\ 6\ 1)_8$

Step 2: Now convert each octal code into its 3 bit binary form, thus 2 is replaced by 010, 6 is replaced by 110 and 1 is replaced by 001. The binary equivalent is $(010\ 110\ 001)_2$

Example: convert $(177.523)_{10}$ to its binary equivalent up to 6 decimal places using octal form.

Step 1: convert 177 to its octal form first, to get $(2\ 6\ 1)_8$ and then convert that to the binary form as shown above, which is $(010\ 110\ 001)_2$

Step 2: convert 0.523 to its octal form which is $(0.413..)_8$

Step 3: convert this into the binary form, digit by digit. This yields (0.100 001 011…)

Step 4: Now put it all together

$(010\ 110\ 001\ .\ 100\ 001\ 011\ldots)_2$

Conversion of binary to decimal (using octal)

First convert the binary number into its octal form. Conversion of binary numbers to octal simply requires grouping bits in the binary number into groups of three bits

•Groups are formed beginning with the Least Significant Bit and progressing to the MSB. Start from right hand side and proceed to left. If the left most group contains only a single digit or a double digit, add zeroes to make it 3 digits.

•Thus

$11\ 100\ 111_2$

$= 011\ \ 100\ \ 111_2$

$= \ 3\ \ \ 4\ \ \ 7_8$

And

$1\ 100\ 010\ 101\ 010\ 010\ 001_2$

$= 001\ 100\ 010\ 101\ 010\ 010\ 001_2$

$= 1425221_8$

Now it can be converted into the decimal form.

Hexadecimal Number System

•Base or radix 16 number system.

•1 hex digit is equivalent to 4 bits.

•Numbers are 0,1,2…..8,9, A, B, C, D, E, F.

B is 11, E is 14

•Numbers are expressed as powers of 16.

•$16^0 = 1$, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$, …

Conversion of hex to decimal (base 16 to base 10)

Example: convert $(F4C)_{16}$ to decimal

$= (F \times 16^2) + (4 \times 16^1) + (C \times 16^0)$

$= (15 \times 256) + (4 \times 16) + (12 \times 1)$

Conversion of decimal to hex (base 10 to base 16)

Example: convert $(4768)_{10}$ to hex.

= 4768 / 16 = 298 remainder 0

= 298 / 16 = 18 remainder 10 (A)

= 18 / 16 = 1 remainder 2

= 1 / 16 = 0 remainder 1

Answer: 1 2 A 0

Note: the answer is read from bottom to top , same as with the binary case.

= 3840 + 64 + 12 + 0
= $(3916)_{10}$

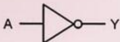
Conversion of binary to hex

•Conversion of binary numbers to hex simply requires grouping bits in the binary numbers into groups of four bits.

•Groups are formed beginning with the LSB and progressing to the MSB.

•1110 0111_2 = $E7_{16}$

•1 1000 1010 1000 0111_2
= 0001 1000 1010 1000 0111_2
= 1 8 A 8 7_{16}

Gate	Symbol	Truth table	Expression
Buffer		A Y 0 0 1 1	Y = A
Inverter (NOT gate)		A Y 0 1 1 0	$Y = \overline{A}$
2-input AND gate		A B Y 0 0 0 0 1 0 1 0 0 1 1 1	$Y = A \bullet B$
2-input NAND gate		A B Y 0 0 1 0 1 1 1 0 1 1 1 0	$Y = \overline{A \bullet B}$
2-input OR gate		A B Y 0 0 0 0 1 1 1 0 1 1 1 1	$Y = A + B$
2-input NOR gate		A B Y 0 0 1 0 1 0 1 0 0 1 1 0	$Y = \overline{A + B}$
2-input EX-OR gate		A B Y 0 0 0 0 1 1 1 0 1 1 1 0	$Y = A \oplus B$
2-input EX-NOR gate		A B Y 0 0 1 0 1 0 1 0 0 1 1 1	$Y = \overline{A \oplus B}$

DeMorgan’s Theory

DeMorgan’s Theorems are basically two sets of rules or laws developed from the Boolean expressions for AND, OR and NOT using two input variables, A and B. These two rules or theorems allow the input variables to be negated and converted from one form of a Boolean function into an opposite form.

DeMorgan’s first theorem states that two (or more) variables NOR´ed together is the same as the two variables inverted (Complement) and AND´ed, while the second theorem states that two (or more) variables NAND´ed together is the same as the two terms inverted (Complement) and OR´ed. That is replace all the OR operators with AND operators, or all the AND operators with an OR operators.

DeMorgan’s First Theorem

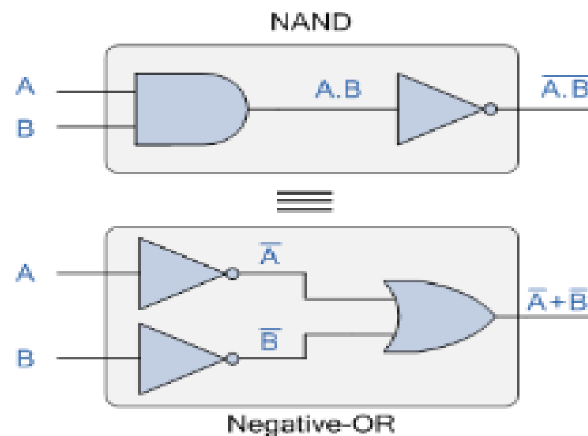
DeMorgan’s First theorem proves that when two (or more) input variables are AND’ed and negated, they are equivalent to the OR of the complements of the individual variables. Thus the equivalent of the NAND function will be a negative-OR function, proving that A.B = A+B. We can show this operation using the following table.

Verifying DeMorgan’s First Theorem using Truth Table

Inputs		Truth Table Outputs For Each Term				
B	A	A.B	$\overline{A.B}$	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	0	1	1	1	1
0	1	0	1	0	1	1
1	0	0	1	1	0	1
1	1	1	0	0	0	0

We can also show that $\overline{A.B} = \overline{A}+\overline{B}$ using logic gates as shown.

DeMorgan's First Law Implementation using Logic Gates



The top logic gate arrangement of: $\overline{A.B}$ can be implemented using a standard NAND gate with inputs A and B. The lower logic gate arrangement first inverts the two inputs producing $\overline{A}$ and $\overline{B}$. These then become the inputs to the OR gate. Therefore the output from the OR gate becomes: $\overline{A}+\overline{B}$

Then we can see here that a standard OR gate function with inverters (NOT gates) on each of its inputs is equivalent to a NAND gate function. So an individual NAND gate can be represented in this way as the equivalency of a NAND gate is a negative-OR.

DeMorgan's Second Theorem

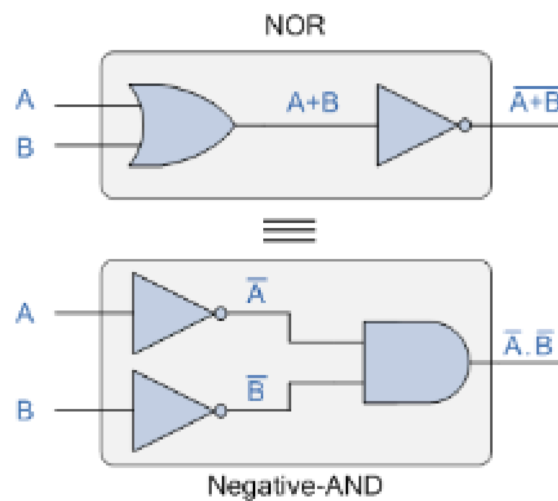
DeMorgan's Second theorem proves that when two (or more) input variables are OR'ed and negated, they are equivalent to the AND of the complements of the individual variables. Thus the equivalent of the NOR function is a negative-AND function proving that $\overline{A+B} = \overline{A}.\overline{B}$, and again we can show operation this using the following truth table.

Verifying DeMorgan's Second Theorem using Truth Table

Inputs		Truth Table Outputs For Each Term				
B	A	A+B	$\overline{A+B}$	$\overline{A}$	$\overline{B}$	$\overline{A}.\overline{B}$
0	0	0	1	1	1	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	0	0	0	0

We can also show that $\overline{A+B} = \overline{A}.\overline{B}$ using the following logic gates example.

DeMorgan's Second Law Implementation using Logic Gates



The top logic gate arrangement of: $\overline{A+B}$ can be implemented using a standard NOR gate function using inputs A and B. The lower logic gate arrangement first inverts the two inputs, thus producing $\overline{A}$ and $\overline{B}$. Thus then become the inputs to the AND gate. Therefore the output from the AND gate becomes: $\overline{A}.\overline{B}$

Then we can see that a standard AND gate function with inverters (NOT gates) on each of its inputs produces an equivalent output condition to a standard NOR gate function, and an individual NOR gate can be represented in this way as the equivalency of a NOR gate is a negative-AND.

Although we have used DeMorgan's theorems with only two input variables A and B, they are equally valid for use with three, four or more input variable expressions, for example:

For a 3-variable input

$$\overline{A.B.C} = \overline{A}+\overline{B}+\overline{C}$$

and also

$$\overline{A+B+C} = \overline{A}.\overline{B}.\overline{C}$$

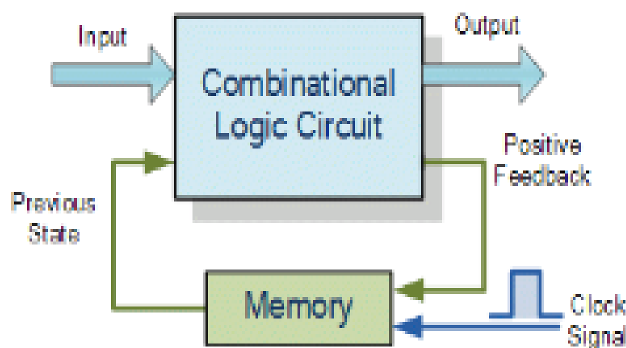
For a 4-variable input

$$\overline{A.B.C.D} = \overline{A}+\overline{B}+\overline{C}+\overline{D}$$

and also

$$\overline{A+B+C+D} = \overline{A}.\overline{B}.\overline{C}.\overline{D}$$

and so on.



Sequential Logic Circuits

Sequential Logic Circuits use flip-flops as memory elements and in which their output is dependent on the input state

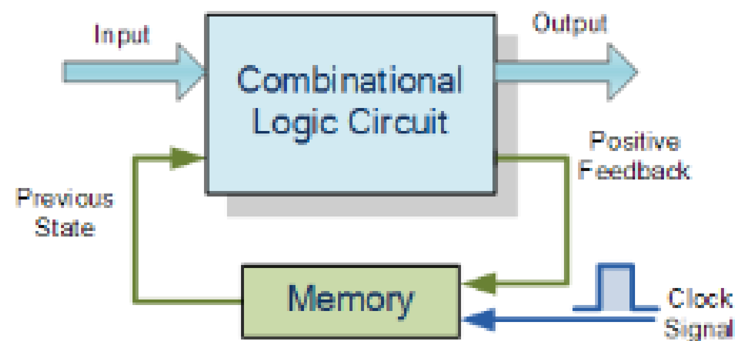
Unlike Combinational Logic circuits that change state depending upon the actual signals being applied to their inputs at that time, **Sequential Logic** circuits have some form of inherent "Memory" built in.

This means that sequential logic circuits are able to take into account their previous input state as well as those actually present, a sort of "before" and "after" effect is involved with sequential circuits.

In other words, the output state of a "sequential logic circuit" is a function of the following three states, the "present input", the "past input" and/or the "past output". *Sequential Logic circuits* remember these conditions and stay fixed in their current state until the next clock signal changes one of the states, giving sequential logic circuits "Memory".

Sequential logic circuits are generally termed as *two state* or Bistable devices which can have their output or outputs set in one of two basic states, a logic level "1" or a logic level "0" and will remain "latched" (hence the name latch) indefinitely in this current state or condition until some other input trigger pulse or signal is applied which will cause the bistable to change its state once again.

Sequential Logic Representation

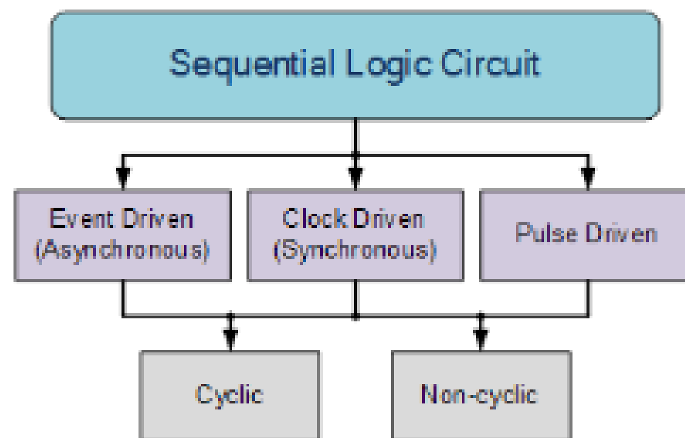


The word "Sequential" means that things happen in a "sequence", one after another and in **Sequential Logic** circuits, the actual clock signal determines when things will happen next. Simple sequential logic circuits can be constructed from standard **Bistable** circuits such as: *Flip-flops*, *Latches* and *Counters* and which themselves can be made by simply connecting together universal NAND Gates and/or NOR Gates in a particular combinational way to produce the required sequential circuit.

Classification of Sequential Logic

As standard logic gates are the building blocks of combinational circuits, bistable latches and flip-flops are the basic building blocks of sequential logic circuits. Sequential logic circuits can be constructed to produce either simple edge-triggered flip-flops or more complex sequential circuits such as storage registers, shift registers, memory devices or counters. Either way sequential logic circuits can be divided into the following three main categories:

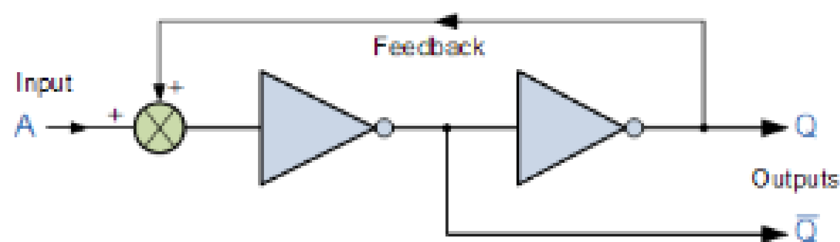
1. Event Driven – asynchronous circuits that change state immediately when enabled.
2. Clock Driven – synchronous circuits that are synchronised to a specific clock signal.
3. Pulse Driven – which is a combination of the two that responds to triggering pulses.



As well as the two logic states mentioned above logic level "1" and logic level "0", a third element is introduced that separates **sequential logic** circuits from their **combinational logic** counterparts, namely *TIME*. Sequential logic circuits return back to their original steady state once reset and sequential circuits with loops or feedback paths are said to be "cyclic" in nature.

We now know that in sequential circuits changes occur only on the application of a clock signal making it synchronous, otherwise the circuit is asynchronous and depends upon an external input. To retain their current state, sequential circuits rely on feedback and this occurs when a fraction of the output is fed back to the input and this is demonstrated as:

Sequential Feedback Loop



The two inverters or NOT gates are connected in series with the output at Q fed back to the input. Unfortunately, this configuration never changes state because the output will always be the same, either a "1" or a "0", it is permanently set. However, we can see how feedback works by examining the most basic sequential logic components, called the SR flip-flop.

SR Flip-Flop

The **SR flip-flop**, also known as a *SR Latch*, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will "SET" the device (meaning the output = "1"), and is labelled **S** and one which will "RESET" the device (meaning the output = "0"), labelled R.

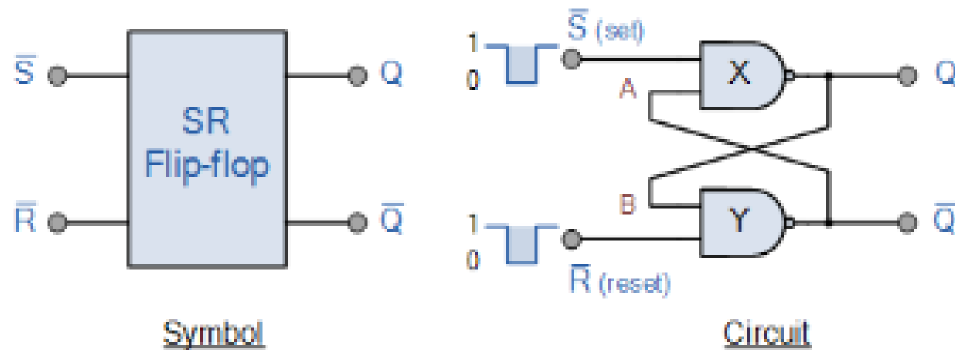
Then the SR description stands for "Set-Reset". The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level "1" or logic "0" depending upon this set/reset condition.

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the SR flip-flop actually has three inputs, Set, Reset and its current output Q relating to it's current state or history. The term "Flip-flop" relates to the actual operation of the device, as it can be "flipped" into one logic Set state or "flopped" back into the opposing logic Reset state.

The NAND Gate SR Flip-Flop

The simplest way to make any basic single bit set-reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND gates as shown, to form a Set-Reset Bistable also known as an active LOW SR NAND Gate Latch, so that there is feedback from each output to one of the other NAND gate inputs. This device consists of two inputs, one called the *Set*, S and the other called the *Reset*, R with two corresponding outputs Q and its inverse or complement $\overline{Q}$ (not-Q) as shown below.

The Basic SR Flip-flop



The Set State

Consider the circuit shown above. If the input R is at logic level "0" (R = 0) and input S is at logic level "1" (S = 1), the NAND gate *Y* has at least one of its inputs at logic "0" therefore, its output $\overline{Q}$ must be at a logic level "1" (NAND Gate principles). Output $\overline{Q}$ is also fed back to input "A" and so both inputs to NAND gate *X* are at logic level "1", and therefore its output Q must be at logic level "0".

Again NAND gate principals. If the reset input R changes state, and goes HIGH to logic "1" with S remaining HIGH also at logic level "1", NAND gate *Y* inputs are now R = "1" and B = "0". Since one of its inputs is still at logic level "0" the output at $\overline{Q}$ still remains HIGH at logic level "1" and there is no change of state. Therefore, the flip-flop circuit is said to be "Latched" or "Set" with $\overline{Q}$ = "1" and Q = "0".

Reset State

In this second stable state, $\overline{Q}$ is at logic level "0", (not Q = "0") its inverse output at Q is at logic level "1", (Q = "1"), and is given by R = "1" and S = "0". As gate *X* has one of its inputs at logic "0" its output Q must equal logic level "1" (again NAND gate principles). Output Q is fed back to input "B", so both inputs to NAND gate *Y* are at logic "1", therefore, $\overline{Q}$ = "0".

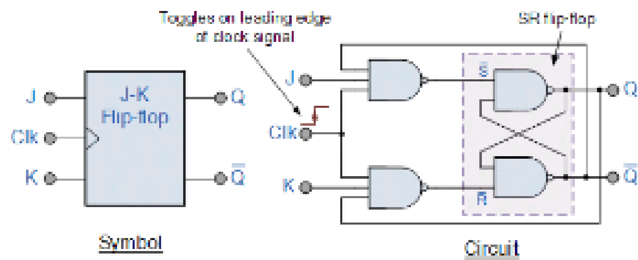
If the set input, S now changes state to logic "1" with input R remaining at logic "1", output $\overline{Q}$ still remains LOW at logic level "0" and there is no change of state. Therefore, the flip-flop circuits "Reset" state has also been latched and we can define this "set/reset" action in the following truth table.

Truth Table for this Set-Reset Function

State	S	R	Q		Description
Set	1	0	0	1	Set $\overline{Q}$ » 1
	1	1	0	1	no change
Reset	0	1	1	0	Reset $\overline{Q}$ » 0
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

It can be seen that when both inputs S = "1" and R = "1" the outputs Q and $\overline{Q}$ can be at either logic level "1" or "0", depending upon the state of the inputs S or R BEFORE this input condition existed. Therefore the condition of S = R = "1" does not change the state of the outputs Q and $\overline{Q}$.

However, the input state of S = "0" and R = "0" is an undesirable or invalid condition and must be avoided. The condition of S = R = "0" causes both outputs Q and $\overline{Q}$ to be HIGH together at logic level "1" when we would normally want $\overline{Q}$ to be the inverse of Q. The result is that the flip-flop looses control of Q and $\overline{Q}$, and if the two inputs are now switched "HIGH" again after this condition to logic "1", the flip-flop becomes unstable and switches to an unknown data state based upon the unbalance as shown in the following switching diagram.



The JK Flip Flop

The JK Flip-flop is similar to the SR Flip-flop but there is no change in state when the J and K inputs are both LOW

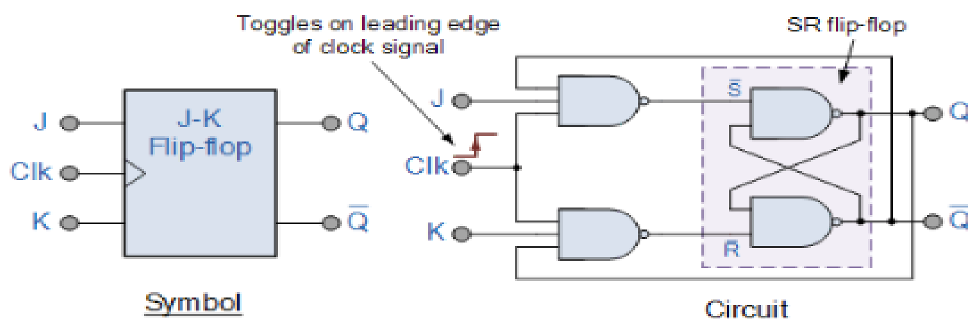
Then to overcome these two fundamental design problems with the SR flip-flop design, the **JK flip Flop** was developed.

This simple **JK flip Flop** is the most widely used of all the flip-flop designs and is considered to be a universal flip-flop circuit. The two inputs labelled "J" and "K" are not shortened abbreviated letters of other words, such as "S" for Set and "R" for Reset, but are themselves autonomous letters chosen by its inventor Jack Kilby to distinguish the flip-flop design from other types.

The sequential operation of the JK flip flop is exactly the same as for the previous SR flip-flop with the same "Set" and "Reset" inputs. The difference this time is that the "JK flip flop" has no invalid or forbidden input states of the SR Latch even when S and R are both at logic "1".

The **JK flip flop** is basically a gated SR flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic level "1". Due to this additional clocked input, a JK flip-flop has four possible input combinations, "logic 1", "logic 0", "no change" and "toggle". The symbol for a JK flip flop is similar to that of an *SR Bistable Latch* as seen in the previous tutorial except for the addition of a clock input.

The Basic JK Flip-flop



Both the S and the R inputs of the previous SR bistable have now been replaced by two inputs called the J and K inputs, respectively after its inventor Jack Kilby. Then this equates to: J = S and K = R.

The two 2-input AND gates of the gated SR bistable have now been replaced by two 3-input NAND gates with the third input of each gate connected to the outputs at Q and $\overline{Q}$. This cross coupling of the SR flip-flop allows the previously invalid condition of S = "1" and R = "1" state to be used to produce a "toggle action" as the two inputs are now interlocked.

If the circuit is now "SET" the J input is inhibited by the "0" status of $\overline{Q}$ through the lower NAND gate. If the circuit is "RESET" the K input is inhibited by the "0" status of Q through the upper NAND gate. As Q and $\overline{Q}$ are always different we can use them to control the input. When both inputs J and K are equal to logic "1", the JK flip flop toggles as shown in the following truth table.

The Truth Table for the JK Function

	Clock	Input		Output		Description
	Clk	J	K	Q	$\overline{Q}$	
same as for the SR Latch	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	↓	0	1	1	0	Reset Q » 0
	X	0	1	0	1	
	↓	1	0	0	1	Set Q » 1
	X	1	0	1	0	
toggle action	↓	1	1	0	1	Toggle
	↓	1	1	1	0	

Then the JK flip-flop is basically an SR flip flop with feedback which enables only one of its two input terminals, either SET or RESET to be active at any one time under normal switching thereby eliminating the invalid condition seen previously in the SR flip flop circuit.

However, if both the J and K inputs are HIGH at logic "1" (J = K = 1), when the clock input goes HIGH, the circuit will "toggle" as its outputs switch and change state complementing each other. This results in the JK flip-flop acting more like a T-type toggle flip-flop when both terminals are "HIGH". However, as the outputs are fed back to the inputs, this can cause the output at Q to oscillate between SET and RESET continuously after being complemented once.

While this JK flip-flop circuit is an improvement on the clocked SR flip-flop it also suffers from timing problems called "race" if the output Q changes state before the timing pulse of the clock input has time to go "OFF". To avoid this the timing pulse period (T) must be kept as short as possible (high frequency). As this is sometimes not possible with basic JK flip-flops built using basic NAND or NOR gates, far more advanced master-slave (edge-triggered) flip-flops were developed which are more stable.

Master-Slave JK Flip-flop

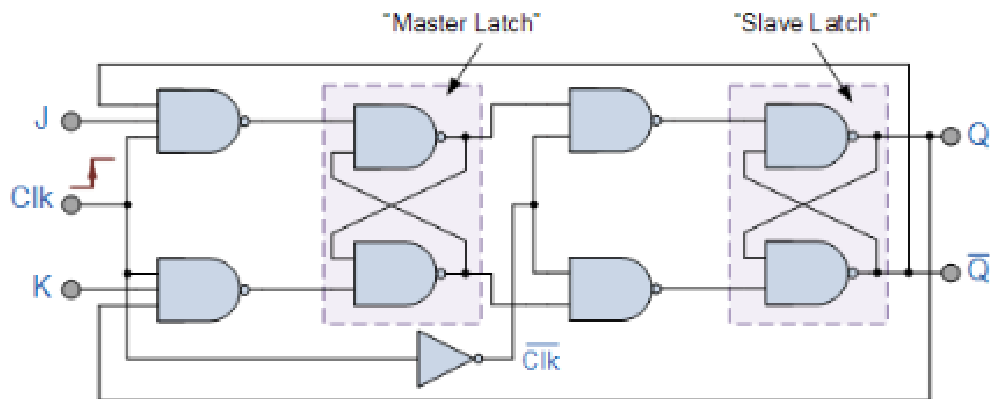
The master-slave flip-flop eliminates all the timing problems by using two SR flip-flops connected together in a series configuration. One flip-flop acts as the "Master" circuit, which triggers on the leading edge of the clock pulse while the other acts as the "Slave" circuit, which triggers on the falling edge of the clock pulse. This results in the two sections, the master section and the slave section being enabled during opposite half-cycles of the clock signal.

The TTL 74LS73 is a Dual JK flip-flop IC, which contains two individual JK type bistable's within a single chip enabling single or master-slave toggle flip-flops to be made. Other JK flip flop IC's include the 74LS107 Dual JK flip-flop with clear, the 74LS109 Dual positive-edge triggered JK flip flop and the 74LS112 Dual negative-edge triggered flip-flop with both preset and clear inputs.

The Master-Slave JK Flip-flop

The **Master-Slave Flip-Flop** is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse. The outputs from Q and $\overline{Q}$ from the "Slave" flip-flop are fed back to the inputs of the "Master" with the outputs of the "Master" flip flop being connected to the two inputs of the "Slave" flip flop. This feedback configuration from the slave's output to the master's input gives the characteristic toggle of the JK flip flop as shown below.

The Master-Slave JK Flip Flop



The input signals J and K are connected to the gated "master" SR flip flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip flop is the inverse (complement) of the "master" clock input, the "slave" SR flip flop does not toggle. The outputs from the "master" flip flop are only "seen" by the gated "slave" flip flop when the clock input goes "LOW" to logic level "0".

When the clock is "LOW", the outputs from the "master" flip flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip flop now responds to the state of its inputs passed over by the "master" section.

Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flip flop are fed through to the gated inputs of the "slave" flip flop and on the "High-to-Low" transition the same inputs are reflected on the output of the "slave" making this type of flip flop edge or pulse-triggered.

Then, the circuit accepts input data when the clock signal is "HIGH", and passes the data to the output on the falling-edge of the clock signal. In other words,

the **Master-Slave JK Flip flop** is a “Synchronous” device as it only passes data with the timing of the clock signal.

One of the main disadvantages of the basic **SR NAND Gate Bistable** circuit is that the indeterminate input condition of SET = "0" and RESET = "0" is forbidden.

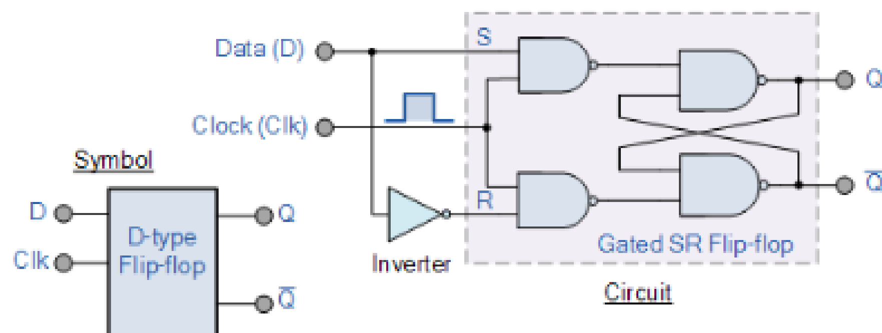
This state will force both outputs to be at logic "1", over-riding the feedback latching action and whichever input goes to logic level "1" first will lose control, while the other input still at logic "0" controls the resulting state of the latch.

But in order to prevent this from happening an inverter can be connected between the "SET" and the "RESET" inputs to produce another type of flip flop circuit known as a *Data Latch*, *Delay flip flop*, *D-type Bistable*, *D-type Flip Flop* or just simply a **D Flip Flop** as it is more generally called.

The **D Flip Flop** is by far the most important of the clocked flip-flops as it ensures that ensures that inputs S and R are never equal to one at the same time. The D-type flip flop are constructed from a gated SR flip-flop with an inverter added between the S and the R inputs to allow for a single D (Data) input.

Then this single data input, labelled "D" and is used in place of the "Set" signal, and the inverter is used to generate the complementary "Reset" input thereby making a level-sensitive D-type flip-flop from a level-sensitive SR-latch as now S = D and R = not D as shown.

D-type Flip-Flop Circuit



We remember that a simple SR flip-flop requires two inputs, one to "SET" the output and one to "RESET" the output. By connecting an inverter (NOT gate) to the SR flip-flop we can "SET" and "RESET" the flip-flop using just one input as now the two input signals are complements of each other. This complement avoids the ambiguity inherent in the SR latch when both inputs are LOW, since that state is no longer possible.

Thus this single input is called the "DATA" input. If this data input is held HIGH the flip flop would be "SET" and when it is LOW the flip flop would change and become "RESET". However, this would be rather pointless since the output of the flip flop would always change on every pulse applied to this data input.

To avoid this an additional input called the "CLOCK" or "ENABLE" input is used to isolate the data input from the flip flop's latching circuitry after the desired data has been stored. The effect is that D input condition is only copied to the output Q when the clock input is active. This then forms the basis of another sequential device called a **D Flip Flop.**

The "D flip flop" will store and output whatever logic level is applied to its data terminal so long as the clock input is HIGH. Once the clock input goes LOW the "set" and "reset" inputs of the flip-flop are both held at logic level "1" so it will not change state and store whatever data was present on its output before the clock transition occurred. In other words the output is "latched" at either logic "0" or logic "1".

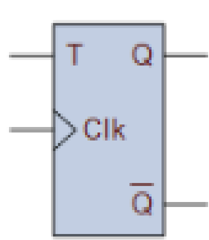
Truth Table for the D-type Flip Flop

Clk	D	Q	$\overline{Q}$	Description
↓ » 0	X	Q	$\overline{Q}$	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

↓ and ↑ indicates direction of clock pulse as it is assumed D-type flip flops are edge triggered

Toggle Flip-flop

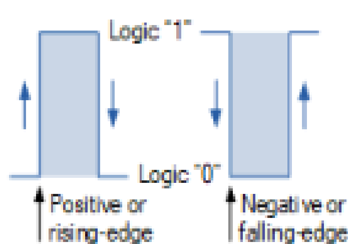
While the *Data* (D) flip-flop is a variation of a clocked SR flip-flop constructed using either NAND or NOR gates, the *Toggle* (T) flip-flop is a variation of the clocked JK flip-flop. The toggle or T-type flip-flop gets its name from the fact that its two outputs Q and $\overline{Q}$ invert from their previous state as it toggles back and forth every time it is triggered (T = 1).



That is, the Q and $\overline{Q}$ outputs change to a "1" if it was "0", and "0" if it was previously a "1" but only when the "T" input changes HIGH, otherwise they do not change, and its this asynchronous toggling action we are interested in here.

The JK is renamed T for T-type or Toggle flip-flop and is generally represented by the logic or graphical symbol shown. The Toggle schematic symbol has two inputs available, one represents the "toggle" (T) input and the other the "clock" (CLK) input.

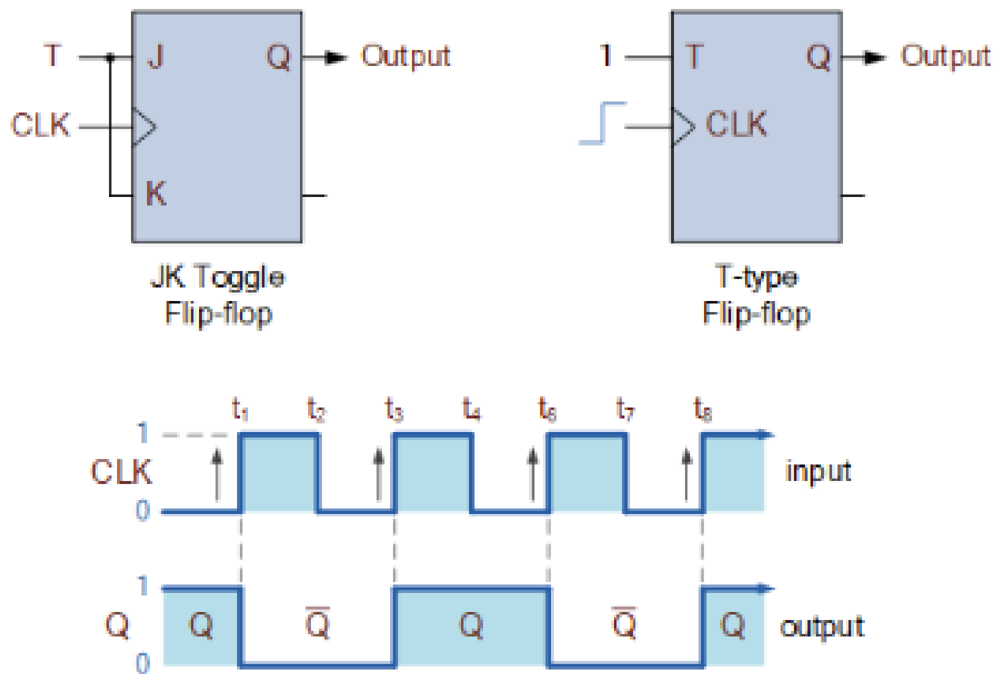
Also, just like the 74LS73 JK flip-flop, the T-type can also be configured to have an enable input called EN or CE (clock enable) allowing it to hold the last data state stored on its outputs indefinately. Thus with the clock enable input set, the application of any new clock pulses prevents toggling of the outputs. But this "enable" feature, if required, must be implimented using additional logic gates.



The triangle of chevron on the input of either type of T-type flip-flop indicates that it is an *edge-triggered* device. If there is a small bubble or circle at the input, then it indicates that the flip-flop toggles on the negative falling edge (HIGH-to-LOW) of each pulse, otherwise, it changes state on the positive or rising transistional edge (LOW-to-HIGH) of each input pulse.

Then we can create the logic circuit of a single bit toggle flip-flop using the basic JK flip-flop by connecting the J and K data inputs together where the common point at the connection of the two inputs is designated T, as shown.

The Toggle Flip-flop



Suppose that initially CLK and input T are both LOW (CLK = T = 0), and that output Q is HIGH (Q = 1). At the rising edge or falling edge of a CLK pulse, the logic "0" condition present at T prevents the output at Q from changing state. Thus the output remains unchanged when T = 0.

Now let's suppose that input T is HIGH (T = 1) and CLK is LOW (CLK = 0). At the rising edge (assuming positive transistion) of a CLK pulse at time t_1, the output at Q changes state and becomes LOW, making $\overline{Q}$ HIGH. The negative transistion of the clock pulse from HIGH to LOW at time t_2 has no effect on the output at Q as the flip-flop is reset into one stable state.

At the next rising edge of the clock signal at time t_3, the logic "1" at T passes to Q, changing its state making output Q HIGH again. The negative transistion of the CLK pulse at time t_4 from HIGH to LOW once again has no effect on the output. Thus the Q output of the flip-flop "toggles" at each positive going edge (for this example) of the CLK pulse.

Characteristics Table for the Toggle Function

CLK	T	Q	Q+1
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0

Then we can define the switching action of the toggle flip-flop in Boolean form as being:

$$Q+1 = T.\overline{Q} + \overline{T}.Q$$

Where: Q represents the present steady state of the flip-flop and Q+1 is the next switching state.

You may have noticed that the characteristic equation given in Boolean form for the toggle flip-flop above will produce an output HIGH for the next state (Q+1) if the two inputs of T and Q are different, and a LOW output if these inputs are the same.

This idea of Q+1 is HIGH only when either of the inputs is HIGH but not when both inputs are HIGH, that is either input but not both represents the same Boolean Algebra expression of an **Exclusive-OR Function** which is given as:

$$Q+1 = \overline{T}Q + T\overline{Q} = T \text{ XOR } Q = T \oplus Q$$

Then we can represent the switching action of a toggle flip-flop using a 2-input Exclusive-OR (Ex-OR) gate.