Simplified

BGCITC

BALA G COMPUTER INSTITUTE & TRAINING CENTER

E- BOOK

DIGITAL ELECTRONICS

DIGITAL ELECTRONICS

Chapter-1

Boolean Algebra and Digital Circuits

Introduction

Boolean algebra (or Boolean logic) is a logical calculus of truth values, developed by George Boole in the late 1830s. It resembles the algebra of real numbers as taught in high school, but with the numeric operations of multiplication xy, addition x + y, and negation −x replaced by the respective logical operations of conjunction x^y, disjunction xvy, and complement ¬x. The Boolean operations are these and all other operations that can be built from these such x^(yvz). These turn out to coincide with the set of all operations on the set {0,1} that take only finitely many arguments; there are 2^{2^n} such operations when there are n arguments.

The laws of Boolean algebra can be defined axiomatically as certain equations called axioms together with their logical consequences called theorems, or semantically as those equations that are true for every possible assignment of 0 or 1 to their variables. The axiomatic approach is sound and complete in the sense that it proves respectively neither more nor fewer laws than the semantic approach.

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 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

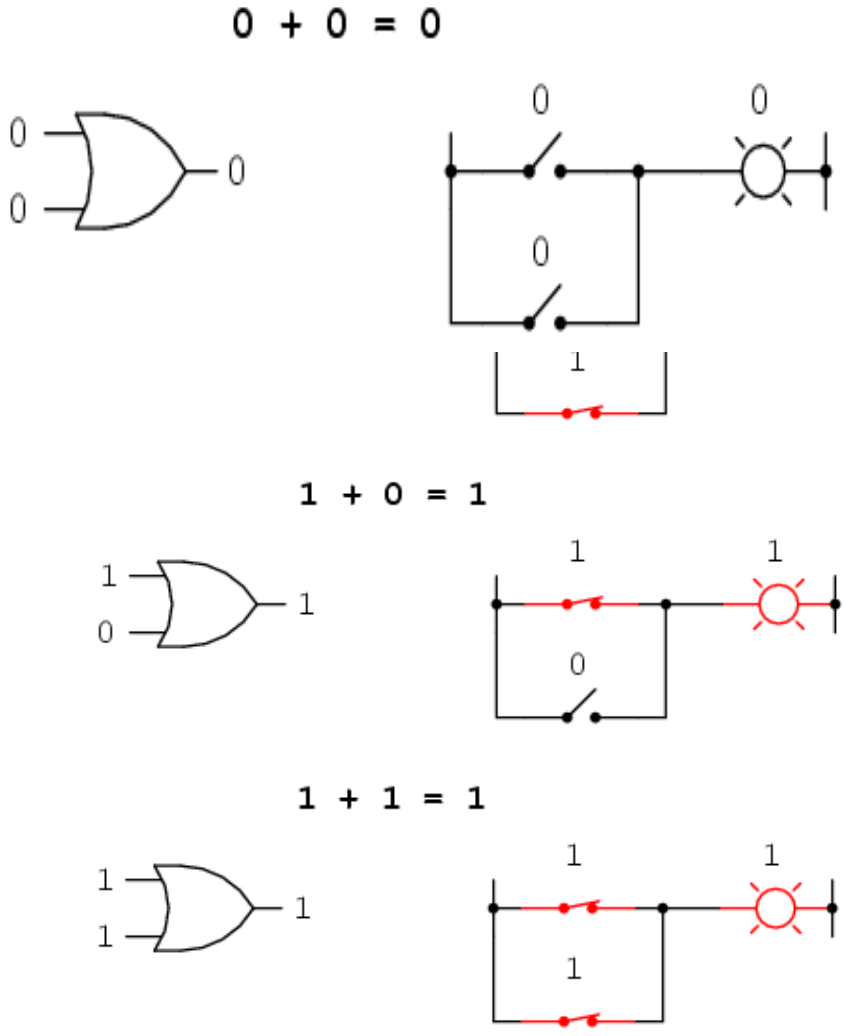
1 + 1 + 1 = 1

0 + 1 + 1 + 1 = 1

1 + 0 + 1 + 1 + 1 = 1

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:



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.

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:

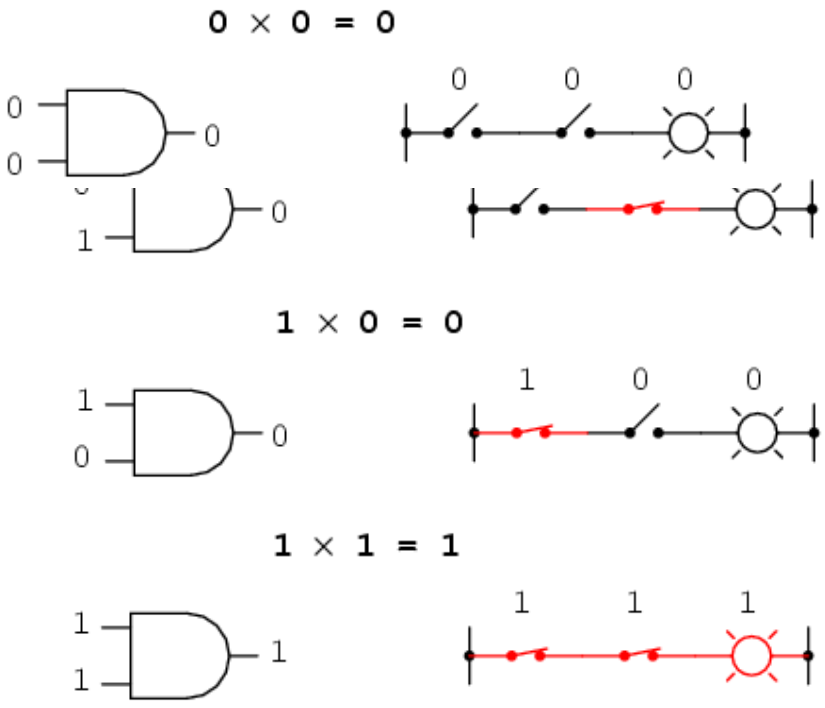
$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$

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:

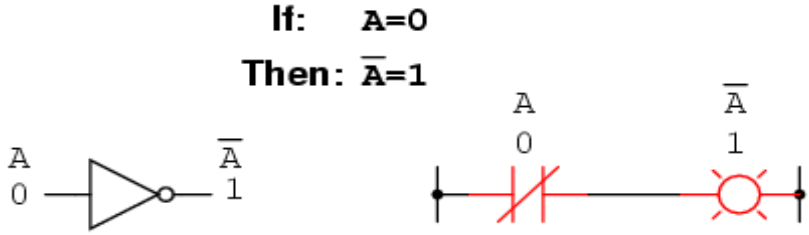
If: $A=0$

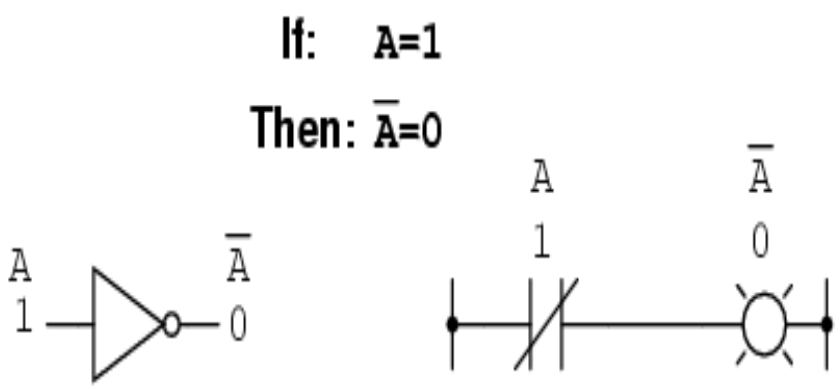
Then: $\overline{A}=1$

If: $A=1$

Then: $\overline{A}=0$

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.

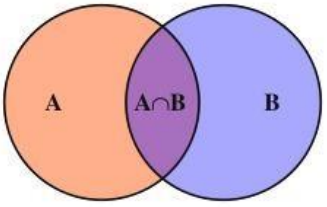
De Morgan's laws

In formal logic, **De Morgan's laws** are rules relating the logical operators 'and' and 'or' in terms of each other via negation.

NOT (P OR Q) = (NOT P) AND (NOT Q)
NOT (P AND Q) = (NOT P) OR (NOT Q)

In symbolic logic terms:

$$\neg(p \vee q) \iff (\neg p) \wedge (\neg q)$$
$$\neg(p \wedge q) \iff (\neg p) \vee (\neg q)$$



where:

- $\neg$ the negation operator (NOT)
- $\wedge$ the conjunction operator (AND)
- $\vee$ the disjunction operator (OR)
- $\iff$ logically equivalent (if and only if)

In set theory and Boolean algebra:

"*Union and intersection interchange under complementation.*":

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$
$$\overline{A \cup B} = \overline{A} \cap \overline{B}.$$

where:

- $\overline{A}$ the negation of A, the overline is written above the terms to be negated
- $\cap$ the intersection operator (AND)
- $\cup$ the union operator (OR)

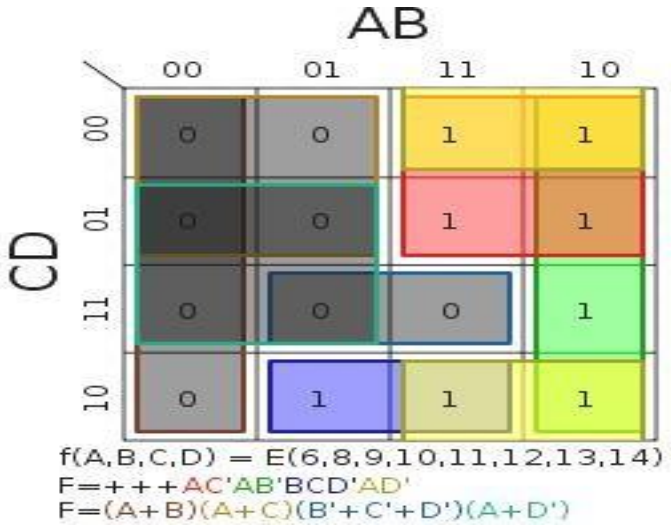
In set notation, De Morgan's law can be remembered using the mnemonic "break the line, change the sign".

Boolean expression

A Boolean expression is an expression that results in a Boolean value, that is, TRUE or FALSE. For example, the value for 5 > 3 is TRUE, the value for "An apple is not a fruit" is FALSE.
Boolean expressions are used also in document retrieval. For example, given a collection of documents we could select those according to a particular word content described by a Boolean expression such as (("cellular" OR "mobile") AND ("phone" OR "telephone")).

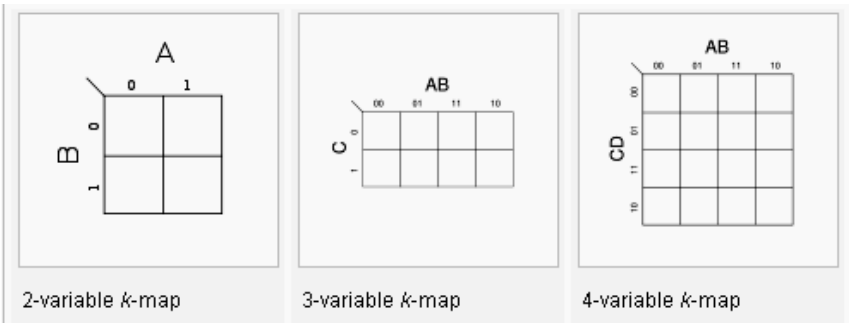
Karnaugh map

The Karnaugh map, also known as a Veitch diagram (KV-map or K-map for short), is a tool to facilitate the simplification of Boolean algebra integrated circuit expressions. The Karnaugh map reduces the need for extensive calculations by taking advantage of human pattern-recognition and permitting the rapid identification and elimination of potential race hazards.
The Karnaugh map was invented in 1952 by Edward W. Veitch. It was further developed in 1953 by Maurice Karnaugh, a physicist at Bell Labs, to help simplify digital electronic circuits.
In a Karnaugh map the boolean variables are transferred (generally from a truth table) and ordered according to the principles of Gray code in which only one variable changes in between squares. Once the table is generated and the output possibilities are transcribed, the data is arranged into the largest even group possible and the minterm is generated through the axiom laws of boolean algebra.



Size of map

The size of the Karnaugh map with *n* Boolean variables is determined by 2^n. The size of the group within a Karnaugh map with *n* Boolean variables and *k* number of terms in the resulting Boolean expression is determined by 2^{nk}. Common sized maps are of 2 variables which is a 2×2 map, 3 variables which is a 2×4 map, and 4 variables which is a 4×4 map.



2-variable *k*-map | 3-variable *k*-map | 4-variable *k*-map

Example

Karnaugh maps are used to facilitate the simplification of Boolean algebra functions. The following is an unsimplified Boolean Algebra function with Boolean variables *A*, *B*, *C*, *D*, and their inverses. They can be represented in two different functions:

- $f(A,B,C,D) = \sum(6,8,9,10,11,12,13,14)$ Note: The values inside $\sum$ are the minterms to map (i.e. which rows have output 1 in the truth table).

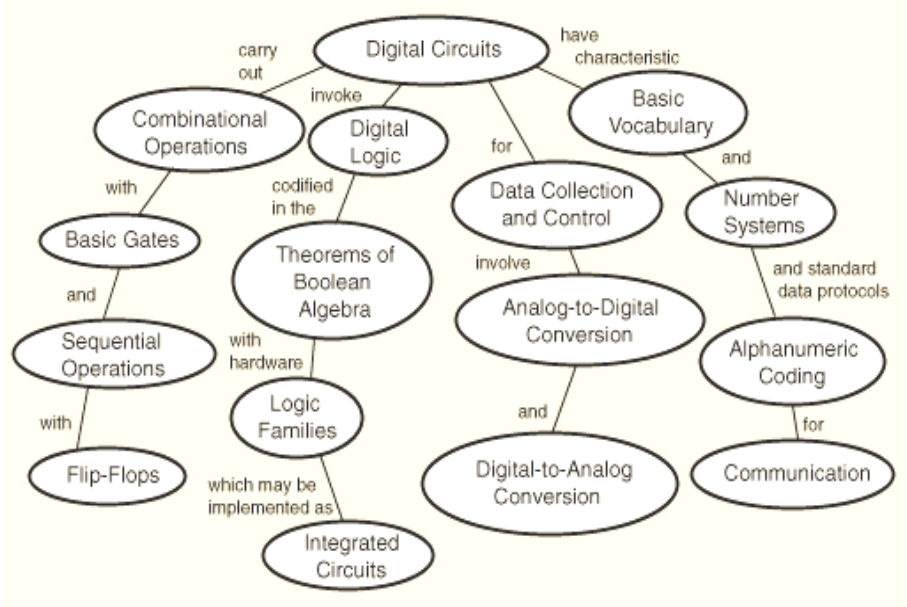
$$f(A,B,C,D) = (\overline{A}BC\overline{D})+(A\overline{B}\overline{C}\overline{D})+(A\overline{B}\overline{C}D)+(A\overline{B}C\overline{D})+(A\overline{B}CD)+(AB\overline{C}\overline{D})+(AB\overline{C}D)+(ABC\overline{D})$$

Truth table

Using the defined minterms, the truth table can be created:

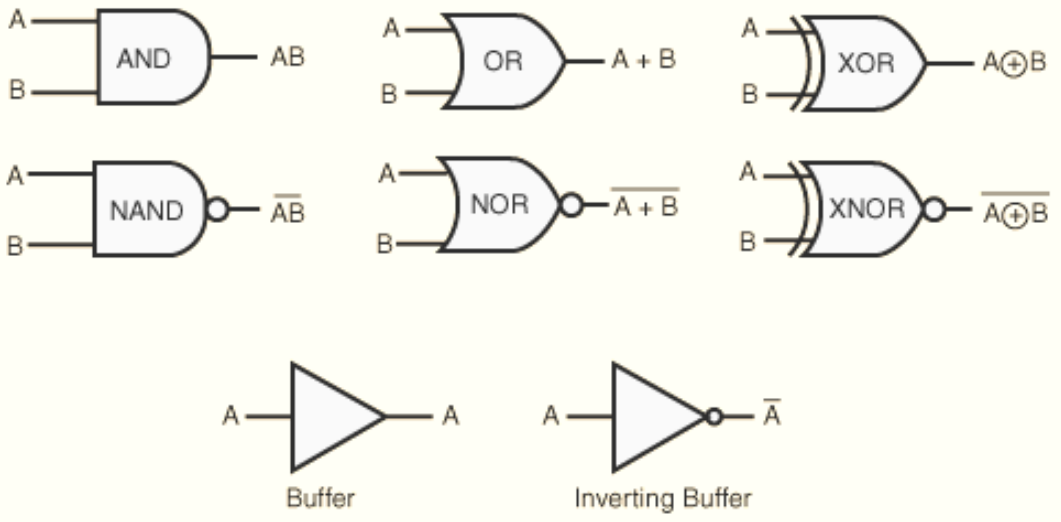
#	*A*	*B*	*C*	*D*	*f*(*A*,*B*,*C*,*D*)
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Digital Circuit



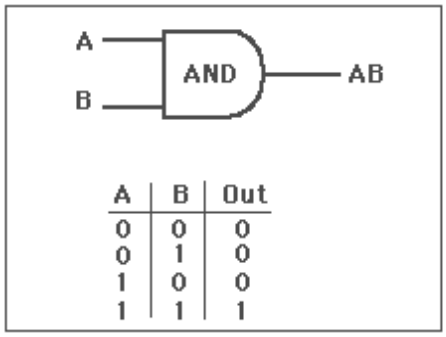
Basic Gates

This is an active graphic. Click on any of the gates for further information.



AND Gate

The output is high only when both inputs A and B are high.

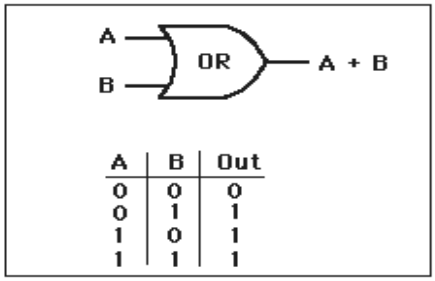


A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation will be signified by AB or A·B. Other common mathematical notations for it are $A \wedge B$ and $A \cap B$, called the intersection of A and B.

OR Gate

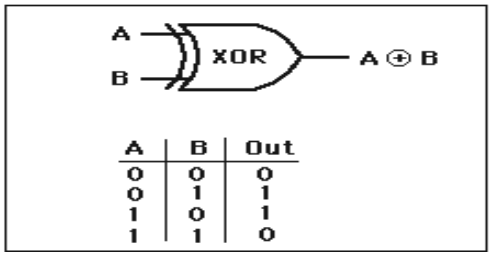
The output is high when either or both of inputs A or B is high. This is logically different from the exclusive OR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

The OR operation will be signified by A + B. Other common mathematical notations for it are $A \vee B$ and $A \cup B$, called the union of A and B.

Exclusive OR Gate

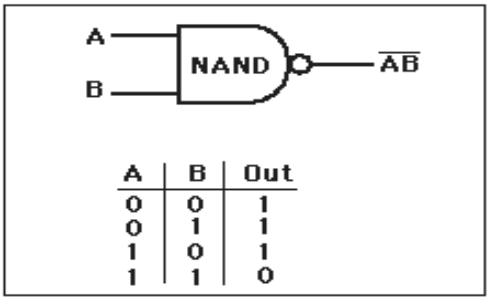


A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

The output is high when either of inputs A or B is high, but not if both A and B are high.

NAND Gate

The output is high when either of inputs A or B is high, or if neither is high. In other words, it is normally high, going low only if both A and B are high.

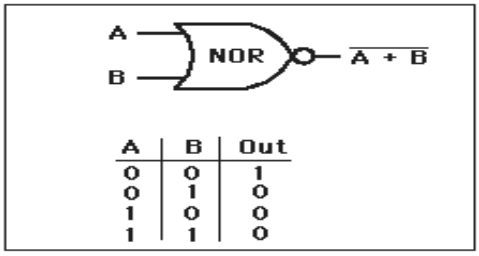


A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

The NAND gate and the NOR gate can be said to be universal gates since combinations of them can be used to accomplish any of the basic operations and can thus produce an inverter, an OR gate or an AND gate. The non-inverting gates do not have this versatility since they can't produce an invert.

NOR Gate

The output is high only when neither A nor B is high. That is, it is normally high but any kind of non-zero input will take it low.

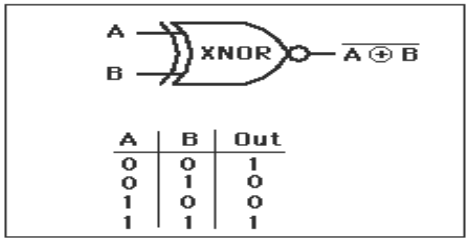


A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

The NOR gate and the NAND gate can be said to be universal gates since combinations of them can be used to accomplish any of the basic operations and can thus produce an inverter, an OR gate or an AND gate. The non-inverting gates do not have this versatility since they can't produce an invert.

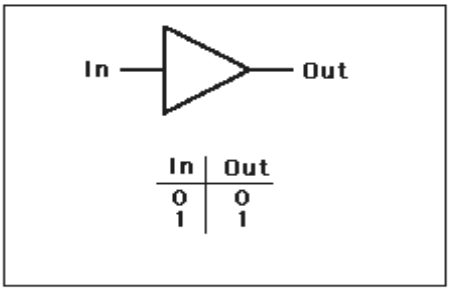
XNOR Gate

The output is high when both inputs A and B are high and when neither A nor B is high.



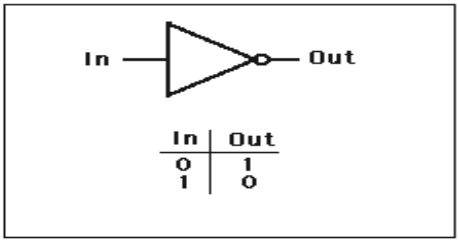
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	1

Buffer



The buffer is a single-input device which has a gain of 1, mirroring the input at the output. It has value for impedance matching and for isolation of the input and output.

Inverting Buffer



The inverting buffer is a single-input device which produces the state opposite the input. If the input is high, the output is low and vice versa.

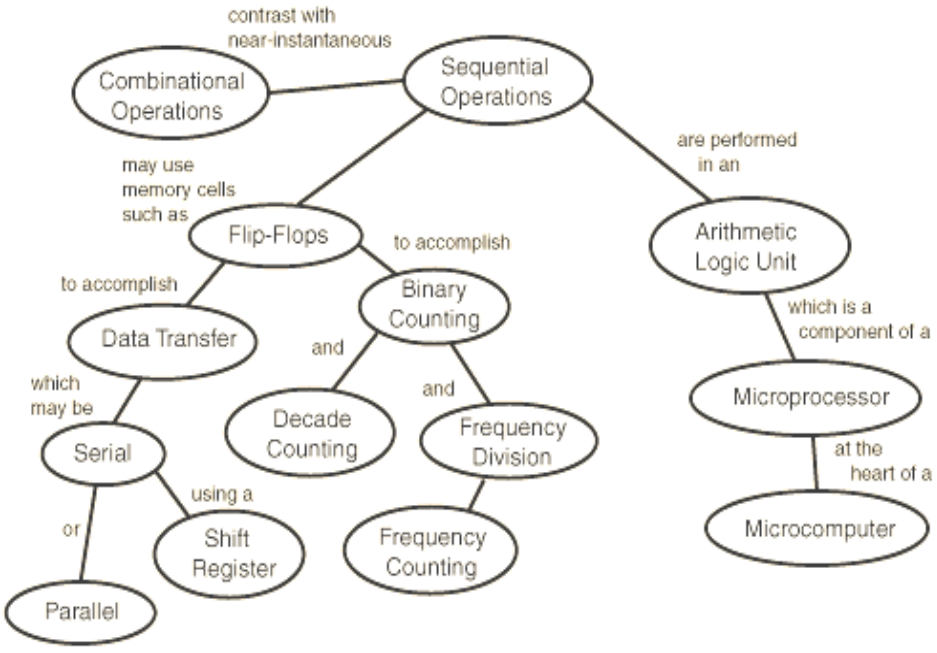
This device is commonly referred to as just an inverter.

Digital Logic

For two binary variables (taking values 0 and 1) there are 16 possible functions. The functions involve only three operations which make up Boolean algebra: AND, OR, and COMPLEMENT. They are symbolically represented as follows:

AND: A B OR: A + B COMPLEMENT: $\overline{A}$ = NOT A

These operations are like ordinary algebraic operations in that they are commutative, associative, and distributive. There is a group of useful theorems of Boolean algebra which help in developing the logic for a given operation.



Boolean Algebra Theorems

The applications of digital logic involve functions of the AND, OR, and NOT operations. These operations are subject to the following identities:

ABC = (AB)C = A(BC), A+B+C =(A+B)+C = A+(B+C):	AND, OR are associative.
AB = BA, A+B = B+A:	AND and OR operations are commutative.
A + BC = (A+B)(A+C), A(B+C) = AB + AC:	forms of the distributive property.
$\overline{A + B} = \bar{A}\bar{B}$:	a form of DeMorgan's Theorem.
$\overline{AB} = \bar{A} + \bar{B}$:	a form of DeMorgan's Theorem.
$AA = A$, $A+A = A$, $A + \bar{A} = 1$, $A\bar{A} = 0$, $A = \bar{\bar{A}}$:	Single Variable Theorems.
$A + AB = A$, $A + \bar{A}B = A+B$:	More two-variable theorems.
$A1 = A$, $A + 1 = 1$, $A + 0 = A$, $A0=0$, $\bar{1}=0$, $\bar{0}=1$:	Identity and Null operations.

Digital Logic

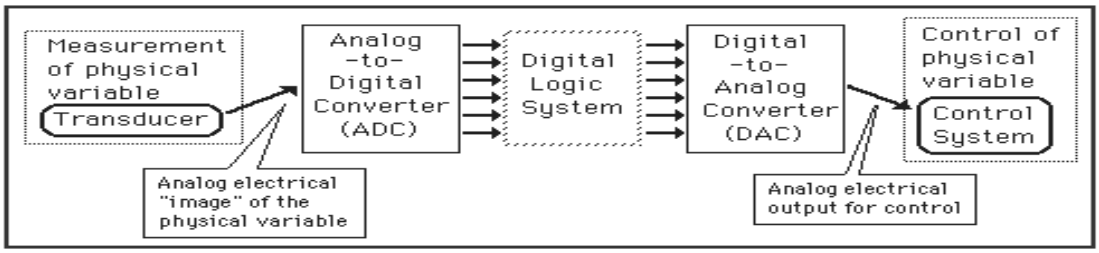
For two binary variables (taking values 0 and 1) there are 16 possible functions. The functions involve only three operations which make up Boolean algebra: AND, OR, and COMPLEMENT. They are symbolically represented as follows:

AND: A B OR: A + B COMPLEMENT: $\bar{A}$ = NOT A

These operations are like ordinary algebraic operations in that they are commutative, associative, and distributive. There is a group of useful theorems of Boolean algebra which help in developing the logic for a given operation.

Data Handling Systems

Both data about the physical world and control signals sent to interact with the physical world are typically "analog" or continuously varying quantities. In order to use the power of digital electronics, one must convert from analog to digital form on the experimental measurement end and convert from digital to analog form on the control or output end of a laboratory system.



Some Basic Digital Vocabulary

- Bit - a single binary digit, a "0" or a "1"
- Word - a series of bits grouped together to represent a number
- Byte - an eight-bit word
- Bus - a collection of conductors for transferring information within the computer.
- Baud - bits/second in communication. kB and MB are abbreviations used for kilobaud and megabaud.

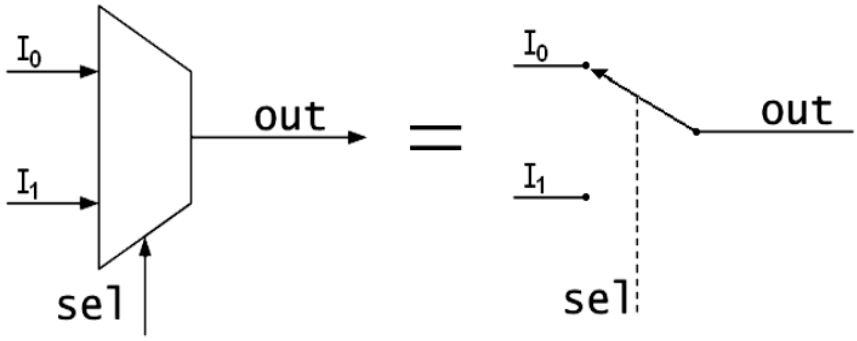
Chapter-2

Multiplexer and Demultiplexer

Multiplexer

In electronics, a **multiplexer** or **mux** (*occasionally the term **muldex** or **muldem** is also found, for a combination multiplexer-demultiplexer*) is a device that performs multiplexing; it selects one of many analog or digital input signals and outputs that into a single line. A multiplexer of 2^n inputs has n select bits, which are used to select which input line to send to the output.

An electronic multiplexer makes it possible for several signals to share one expensive device or other resource, for example one A/D converter or one communication line, instead of having one device per input signal.

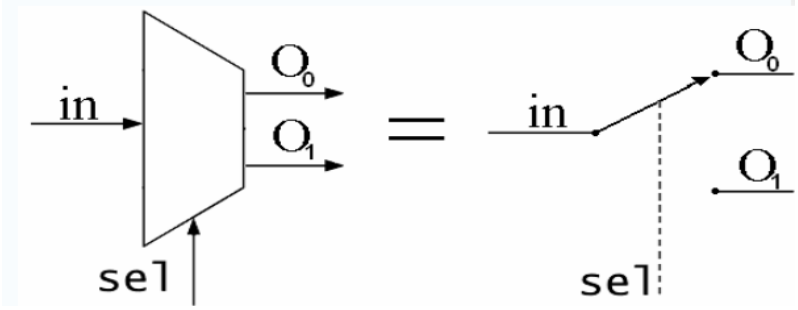


Schematic of a 2-to-1 Multiplexer. It can be equated to a controlled switch.

Demultiplexer

In electronics, a **demultiplexer** (or **demux**) is a device taking a single input signal and selecting one of many data-output-lines, which is connected to the single input. A multiplexer is often used with a complementary demultiplexer on the receiving end.

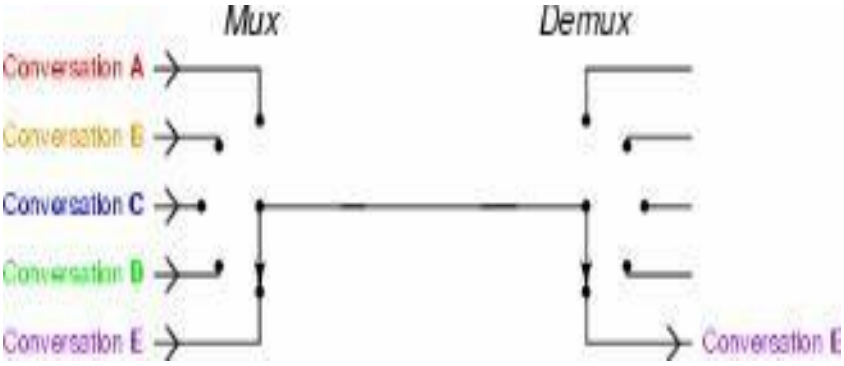
An electronic multiplexer can be considered as a multiple-input, single-output switch, and a demultiplexer as a single-input, multiple-output switch. The schematic symbol for a multiplexer is an isosceles trapezoid with the longer parallel side containing the input pins and the short parallel side containing the output pin. The schematic on the right shows a 2-to-1 multiplexer on the left and an equivalent switch on the right. The *sel* wire connects the desired input to the output.



Schematic of a 1-to-2 Demultiplexer. Like a multiplexer, it can be equated to a controlled switch.

Cost savings

One use for multiplexers is cost savings by connecting a multiplexer and a demultiplexer (or demux) together over a single channel (by connecting the multiplexer's single output to the demultiplexer's single input). The image to the right demonstrates this. In this case, the cost of implementing separate channels for each data source is more expensive than the cost and inconvenience of providing the multiplexing/demultiplexing functions. In a physical analogy, consider the merging behaviour of commuters crossing a narrow bridge; vehicles will take turns using the few available lanes. Upon reaching the end of the bridge they will separate into separate routes to their destinations.
At the receiving end of the data link a complementary demultiplexer is normally required to break single data stream back down into the original streams. In some cases, the far end system may have more functionality than a simple demultiplexer and so, whilst the demultiplexing still exists logically, it may never actually happen physically. This would be typical where a multiplexer serves a number of IP network users and then feeds directly into a router which immediately reads the content of the entire link into its routing processor and then does the demultiplexing in memory from where it will be converted directly into IP packets.
It is usual to combine a multiplexer and a demultiplexer together into one piece of equipment and simply refer to the whole thing as a "multiplexer". Both pieces of equipment are needed at both ends of a transmission link because most communications systems transmit in both directions.
A real world example is the creation of telemetry for transmission from the computer/instrumentation system of a satellite, space craft or other remote vehicle to a ground system.
In analog circuit design, a multiplexer is a special type of analog switch that connects one signal selected from several inputs to a single output.



The basic function of a multiplexer: combining multiple inputs into a single data stream. On the receiving side, a demultiplexer splits the single data stream into the original multiple signals.

DIGITAL MULTIPLEXERS

In digital circuit design, the selector wires are of digital value. In the case of a 2-to-1 multiplexer, a logic value of 0 would connect I_0 to the output while a logic value of 1 would connect I_1 to the output. In larger multiplexers, the number of selector pins is equal to $\lceil \log_2(n) \rceil$ where n is the number of inputs.

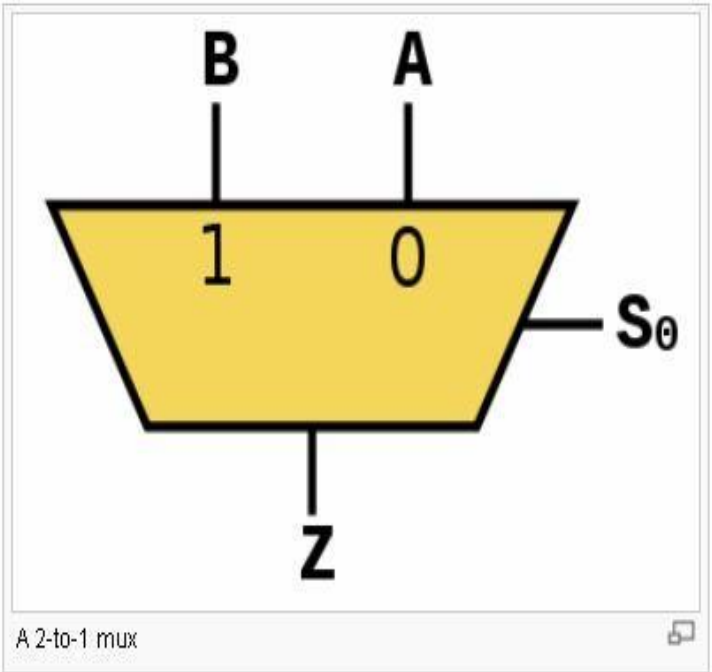
For example, 9 to 16 inputs would require no less than 4 selector pins and 17 to 32 inputs would require no less than 5 selector pins. The binary value expressed on these selector pins determines the selected input pin.

A 2-to-1 multiplexer has a boolean equation where A and B are the two inputs, S is the selector input, and Z is the output:

$$Z = (A \cdot \overline{S}) + (B \cdot S)$$

Which can be expressed as a truth table:

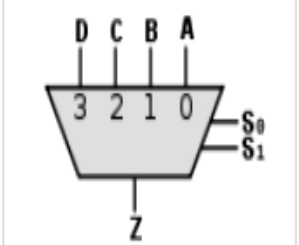
S	A	B	Z
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0



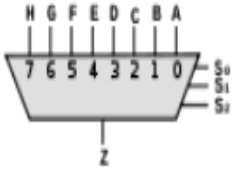
A 2-to-1 mux

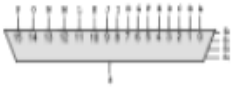
This truth table should make it quite clear that when S = 0 then Z = A but when S = 1 then Z = B. A straightforward realization of this 2-to-1 multiplexer would need 2 AND gates, an OR gate, and a NOT gate.

Larger multiplexers are also common and, as stated above, requires $\lceil \log_2(n) \rceil$ selector pins for n inputs. Other common sizes are 4-to-1, 8-to-1, and 16-to-1. Since digital logic uses binary values, powers of 2 are used (4, 8, 16) to maximally control a number of inputs for the given number of selector inputs.



4-to-1 mux

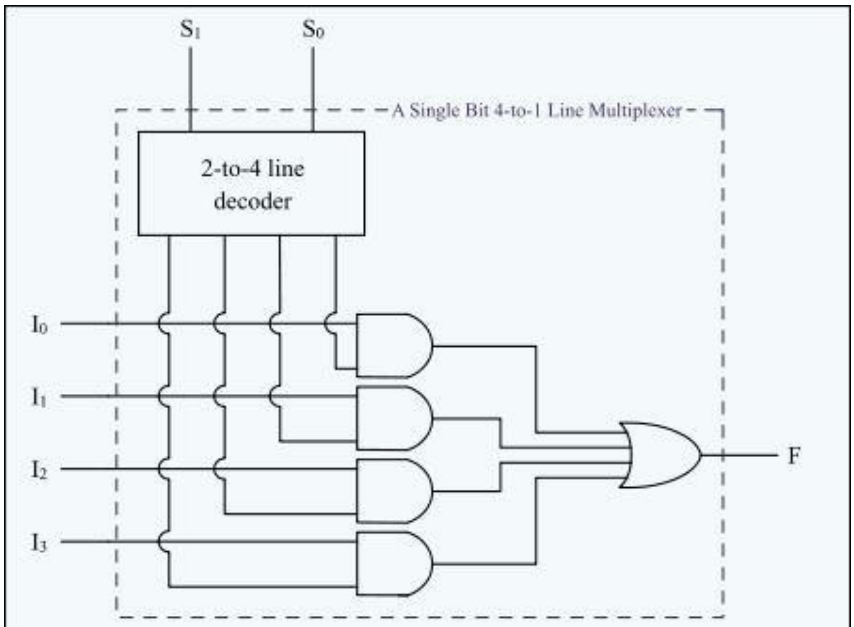

8-to-1 mux


16-to-1 mux

The boolean equation for a 4-to-1 multiplexer is:

$$F = (A \cdot \overline{S_0} \cdot \overline{S_1}) + (B \cdot S_0 \cdot \overline{S_1}) + (C \cdot \overline{S_0} \cdot S_1) + (D \cdot S_0 \cdot S_1)$$

Two realizations for creating a 4-to-1 multiplexer are shown below:



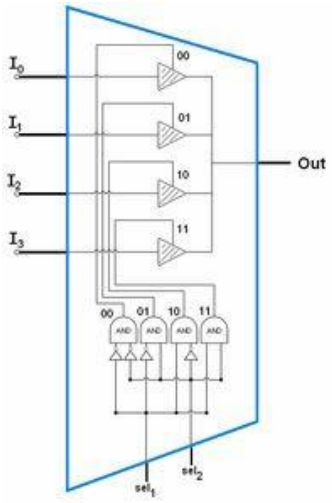
Truth Table

S_1 S_0	I_3	I_2	I_1	I_0	F
0 0	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	0
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	1
	0	1	1	0	0
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	1
	1	0	1	0	0
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	0
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
0 1	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	1
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	1
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	0
	1	1	1	0	1
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
1 0	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	0
	0	1	0	0	1
	0	1	0	1	1
	0	1	1	0	1
	0	1	1	1	1
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	0
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	1

S_1 S_0	I_3	I_2	I_1	I_0	F
1 1	0	0	0	0	0
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	0
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	0
	0	1	1	1	0
	1	0	0	0	1
	1	0	0	1	1
	1	0	1	0	1
	1	0	1	1	1
	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	1



These are two realizations of a 4-to-1 multiplexer:

- one realized from a decoder, AND gates, and an OR gate
- one realized from 3-state buffers and AND gates (the AND gates are acting as the decoder)

Note that the subscripts on the I_n inputs indicate the decimal value of the binary control inputs at which that input is let through.

Chaining multiplexers

Larger multiplexers can be constructed by using smaller multiplexers by chaining them together. For example, an 8-to-1 multiplexer can be made with two 4-to-1 and one 2-to-1 multiplexers. The two 4-to-1 multiplexer outputs are fed into the 2-to-1 with the selector pins on the 4-to-1's put in parallel giving a total number of selector inputs to 3, which is equivalent to an 8-to-1.

List of ICs which provide multiplexing

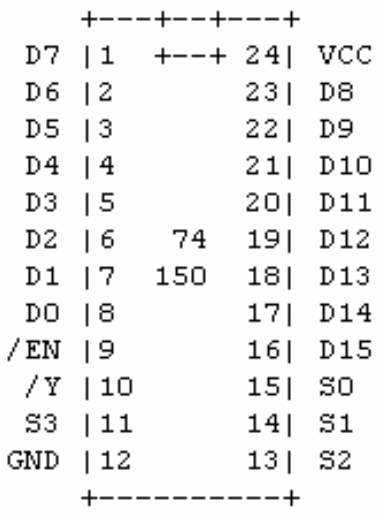
The 7400 series has several ICs that contain multiplexer(s):

S.No.	IC No.	Function	Output State
1	74157	Quad- 2:1 MUX	Output same as input given
2	74158	Quad- 2:1 MUX	Output is inverted input
3	74153	Dual- 4:1 MUX	Output same as input
4	74352	Dual- 4:1 MUX	Output is inverted input
5	74151A	8:1 MUX	Both outputs available ie. Complementary outputs
6	74151	8:1 MUX	Output is inverted input
7	74150	16:1 MUX	Output is inverted input

The 74150 IC

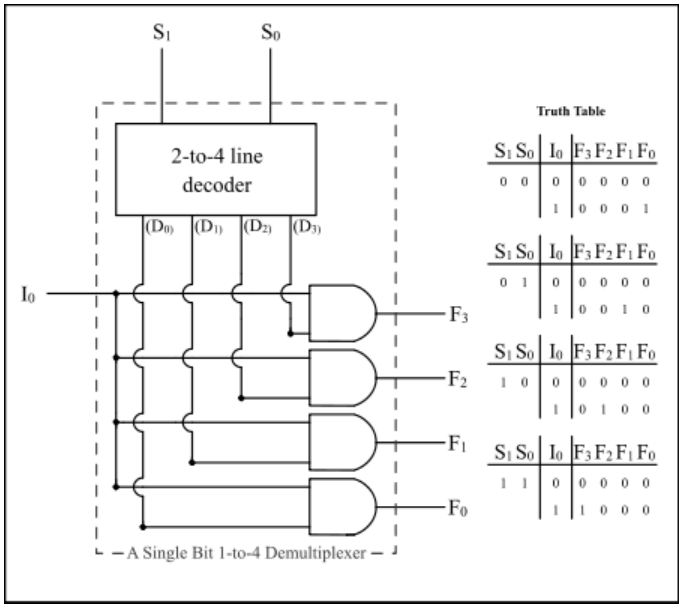
74150

16-to-1 line inverting data selector/multiplexer.



DIGITAL DEMULTIPLEXERS

Demultiplexers take one data input and a number of selection inputs, and they have several outputs. They forward the data input to one of the outputs depending on the values of the selection inputs. Demultiplexers are sometimes convenient for designing general purpose logic, because if the demultiplexer's input is always true, the demultiplexer acts as a decoder. This means that any function of the selection bits can be constructed by logically OR-ing the correct set of outputs.



List of ICs which provide demultiplexing

The 7400 series has several ICs that contain demultiplexer(s):

S.No.	IC No.	Function	Output State
1	74139	Dual 1:4 DEMUX	Output is inverted input
3	74156	Dual- 1:4 DEMUX	Output is open collector
4	74138	1:8 DEMUX	Output is inverted input
5	74154	1:16 DEMUX	Output is inverted input
6	74159	1:16 DEMUX	Output is open collector and same as input

Decoder

A decoder is a device which does the reverse of an encoder, undoing the encoding so that the original information can be retrieved. The same method used to encode is usually just reversed in order to decode. In digital electronics this would mean that a decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. e.g. n-to-2n, BCD decoders.

Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output code word. Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding.

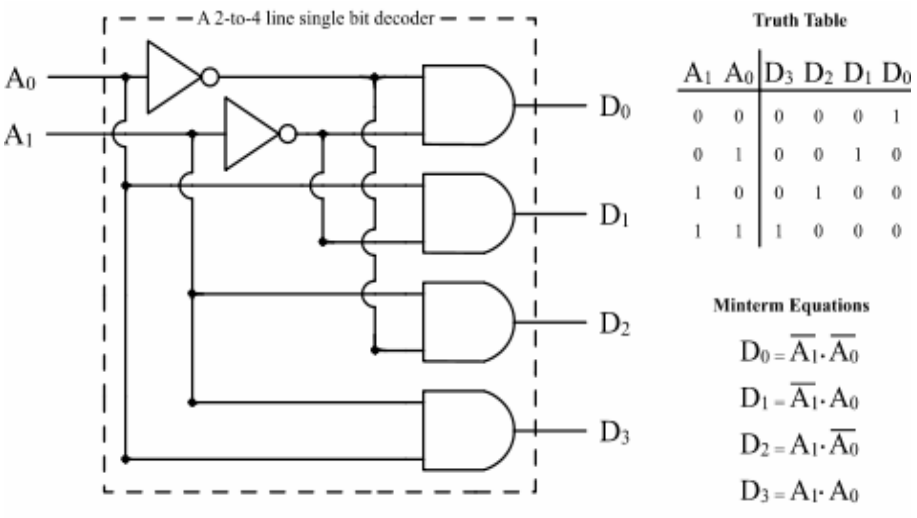
The decoder circuit would be an AND gate because the output of an AND gate is "High" (1) only when all its inputs are "High." Such output is called as "active High output". If instead of AND gate, the NAND gate is connected the output will be "Low" (0) only when all its inputs are "High". Such output is called as "active low output".

A slightly more complex decoder would be the n-to-2n type binary decoders. These type of decoders are combinational circuits that convert binary information from 'n' coded inputs to a maximum of 2n unique outputs. We say a maximum of 2n outputs because in case the 'n' bit coded information has unused bit combinations, the decoder may have less than 2n outputs. We can have 2-to-4 decoder, 3-to-8 decoder or 4-to-16 decoder. We can form a 3-to-8 decoder from two 2-to-4 decoders (with enable signals).

Similarly, we can also form a 4-to-16 decoder by combining two 3-to-8 decoders. In this type of circuit design, the enable inputs of both 3-to-8 decoders originate from a 4th input, which acts as a selector between the two 3-to-8 decoders. This allows the 4th input to enable either the top or bottom decoder, which produces outputs of D(0) through D(7) for the first decoder, and D(8) through D(15) for the second decoder.

It is important to note that a decoder that contains enable inputs is also known as a decoder-demultiplexer. Thus, we have a 4-to-16 decoder produced by adding a 4th input shared among both decoders, producing 16 outputs.

A Digitrax DH163AT DCC decoder in an Athearn locomotive before the shell goes on.



Example: A 2-to-4 Line Single Bit Decoder

IC 74154

IC 74154 is a 4-16 line decoder, it takes the 4 line BCD input and selects respective output one among the 16 output lines (click here to download datasheet). It is active low output IC so when any output line is selected it is indicated by active low signal, rest of the output lines will remain active high. This 4-line-to-16-line decoder utilizes TTL circuitry to decode four binary-coded inputs into one of sixteen mutually exclusive outputs when both the strobe inputs, G1 and G2, are low. The demultiplexing function is performed by using the 4 input lines to address the output line, passing data from one of the strobe inputs with the other strobe input low. When either strobe input is high, all outputs are high. These demultiplexer are ideally suited for implementing high-performance memory decoders.

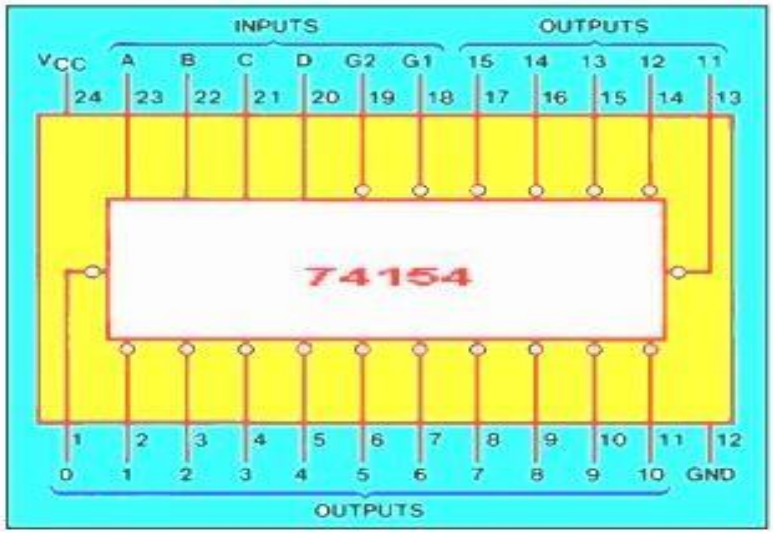


Figure G. IC 74154 4-16 line decoder

Chapter-3

Comparator, Error Detector and Code Conversion

COMPARATOR

In electronics, a comparator is a device which compares two voltages or currents and switches its output to indicate which is larger.
A dedicated voltage comparator will generally be faster than a general-purpose op-amp pressed into service as a comparator. A dedicated voltage comparator may also contain additional features such as an accurate, internal voltage reference, an adjustable hysteresis and a clock gated input.

Input voltage range

The input voltages must not exceed the power voltage range: $V_{S-} \leq V_+, V_- \leq V_{S+}$

In the case of TTL/CMOS logic output comparators, negative inputs are not allowed:

$$0 \leq V_+, V_- \leq V_{cc}$$

Op-amp implementation of voltage comparator

An op-amp combines both the null detector amplifier and the feedback signal source into one single unit. An op-amp has a well balanced difference input and a very high gain. The main difference between a comparator and an op-amp is that the op-amp is designed to operate in a linear region through a feedback control, while the comparator is designed to produce well-defined limit output voltages that respond quickly to changes in vs. The parallels in the characteristics allows the op-amps to serve as comparators in some functions.
A standard op-amp operating without negative feedback can be used as a comparator. When the non-inverting input (V+) is at a higher voltage than the inverting input (V-), the high gain of the op-amp causes it to output the most positive voltage it can. When the non-inverting input (V+) drops below the inverting input (V-), the op-amp outputs the most negative voltage it can. Since the output voltage is limited by the supply voltage, for an op-amp that uses a balanced, split supply, (powered by ± VS) this action can be written:
where sgn(x) is the sign function. Generally, the positive and negative supplies VS will not match absolute value:

$$V_{out} = V \cdot sgn(V_+ - V_-)$$

When else when

$$V_{out} \leq V_{S+} \text{ when } V_+ > V_- \text{ else } V_{S-} \text{ when } V_+ < V_-$$

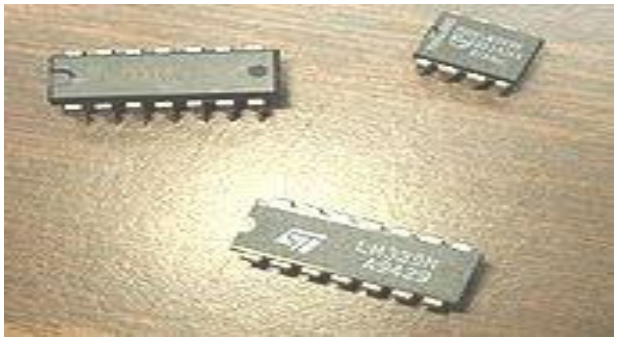
Equality of input values is very difficult to achieve in practice. The speed at which the change in output results from a change in input (often called the slew rate in operational amplifiers) is typically in the order of 10ns to 100ns, but can be as slow as a few tens of μs.

Dedicated voltage comparator chips

A dedicated voltage comparator chip, such as the LM339, is designed to interface directly to digital logic (for example TTL or CMOS). The output is a binary state, and it is often used to interface real world signals to digital circuitry (see analog to digital converter). If one of the voltages is fixed, for example because a DC adjustment is possible in a device earlier in the signal path, a comparator is just a cascade of amplifiers. When the voltages are nearly equal, the output voltage will not fall into one of the logic levels, thus analog signals will enter the digital domain with unpredictable results. To make this rangeas

small as possible the cascade is long and high gain, that is bipolar transistors instead of field effect transistors are used, except sometimes for the first stage. For high speed the input impedance of the stages is made low. This already reduces the saturation of the slow, large P-N junction of the bipolar transistors, which would otherwise lead to long recovery times. Fast that is small Schottky diodes as in binary logic are applied to improve matters even further. Also like in binary logic the speed is not as high as if the amplifiers would be used for analog signals. Slew rate has no meaning for these devices. For the application in flash ADCs after each amplifier the signal can be fanned out over 8 ports matched to the voltage and current gain and resistors are used as level-shifters.
The LM339 accomplishes this with an open collector output. When the inverting input is at a higher voltage than the non inverting input, the output of the comparator is connected to the negative power supply. When the non inverting input is higher than the inverting input, the output is floating (has a very high impedance to ground).

Several voltage comparator ICs



Inputs	Output
− > +	Negative
+ > −	Floating

With a pull-up resistor and a 0 to +5V power supply, the output takes on the voltages 0 or +5 and can be interfaced to TTL logic:

$V_{\text{out}} \leq V_{\text{CC}}$ when $V_+ \geq V_-$ else 0.

Comparators as detectors

Null detectors

Comparators are a type of amplifier, which is designed distinctively for null comparison measurements. A comparator is a very high gain amplifier with well-balanced inputs and controlled output limits. The comparator, as its name suggests, compares the two input voltages, determining which is larger. The inputs are an unknown voltage, and a reference voltage, usually referred to as v_u and v_r. The reference voltage is generally connected to the non-inverting input (+), while vu is usually connected to the inverting input (-). (The inputs are labeled according to the sign of the output when that input is greater than the other.) The output is given as either a positive or negative limit, for example +/-12V.

A null detector is one that functions to identify when a given value is zero. In this case, the idea is to detect when there is no difference between in the input voltages. This gives the identity of the unknown voltage, since the reference voltage is known.

When using a comparator as a null detector, there are limits as to the accuracy of the zero value measurable. Zero output is given when the magnitude of the difference in the voltages multiplied by the gain of the amplifier is less than the voltage limits. For example, if the gain of the amplifier is 10^6, and the voltage limits are +/-6V, then no output will be given if the difference in the voltages is less than 6μV. One could refer to this as a sort of uncertainty in the measurement.

1. ^ Malmstadt, Enke and Crouch, Electronics and Instrumentation for Scientists, The Benjamin/Cummings Publishing Company, Inc., 1981, ISBN 0-8053-6917-1, Chapter 5.
2. ^ Electronics and Instrumentation for Scientists. Malmstadt, Enke, and Crouch, The Benjamin/Cummings Publishing Co., In., 1981, p.108-110.

Zero-crossing detectors

For this type of detector, a comparator is used to detect each time an ac pulse changes polarity. The output of the comparator changes state each time the pulse changes its polarity. That is, the output is HI for a positive pulse and LO for a negative pulse. The comparator also amplifies and squares the input signal.

1. ^ Electronics and Instrumentation for Scientists. Malmstadt, Enke, and Crouch, The Benjamin/Cummings Publishing Co., In., 1981, p.230.

Digital comparator

A digital comparator is a hardware electronic device that compares two numbers in binary form and generates a one or a zero at its output depending on whether they are the same or not.
Comparators can be used in a central processing unit (CPU) or microcontroller in branching software. A comparator can be simulated by subtracting the two values (A & B) in question and checking if the result is zero. This works because if A = B then A - B = 0.
The analog equivalent is the comparator. Many microcontrollers have analog comparators on some of their inputs that can be read or trigger an interrupt.

The operation of a *single bit digital comparator* can be expressed as a truth table:

Inputs		Outputs		
A	B	$A < B$	$A = B$	$A > B$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

ERROR DETECTION

Parity bit

A parity bit is a bit that is added to ensure that the number of bits with value of one in a given set of bits is always even or odd. Parity bits are used as the simplest error detecting code.

As for binary digits, there are two variants of parity bits: even parity bit and odd parity bit. An even parity bit is set to 1 if the number of ones in a given set of bits is odd (making the total number of ones, including the parity bit, even). An odd parity bit is set to 1 if the number of ones in a given set of bits is even (making the total number of ones, including the parity bit, odd). Even parity is actually a special case of a cyclic redundancy check (CRC), where the 1-bit CRC is generated by the polynomial x+1.

If the parity bit is present but not used, it may be referred to as mark parity, where the parity bit is always 1, or as space parity, where the bit is always 0.

7 bits of data (number of 1s)	8 bits including parity	
	even	odd
0000000 (0)	**0**0000000	**1**0000000
1010001 (3)	**1**1010001	**0**1010001
1101001 (4)	**0**1101001	**1**1101001
1111111 (7)	**1**1111111	**0**1111111

If an odd number of bits (including the parity bit) are changed in transmission of a set of bits then parity bit will be incorrect and will thus indicate that an error in transmission has occurred. Therefore, parity bit is an error detecting code, but is not an error correcting code as there is no way to determine which particular bit is corrupted. The data must be discarded entirely, and re-transmitted from scratch. On a noisy transmission medium a successful transmission could take a long time, or even never occur. Parity does have the advantage, however, that it is about the best possible code that uses only a single bit of space and it requires only a number of XOR gates to generate.
Parity bit checking is used often for transmission of ASCII characters, since they have 7 bits and the 8th bit can conveniently be used as a parity bit.

For example, our parity bit can be computed as follows assuming we are sending a simple 4-bit value 1001, with the parity bit following on the right, and ^ denoting XOR gate:

Transmission sent using even parity:

```
A wants to transmit:          1001
A computes parity bit value:  1^0^0^1 = 0
A adds parity bit and sends:  10010
B receives:                   10010
B computes parity:            1^0^0^1^0 = 0
B reports correct transmission after observing expected even result.
```

Transmission sent using odd parity:

```
A wants to transmit:          1001
A computes parity bit value:  ~(1^0^0^1) = 1
A adds parity bit and sends:  10011
B receives:                   10011
B computes overall parity:    1^0^0^1^1 = 1
B reports correct transmission after observing expected odd result.
```

This mechanism enables the detection of single bit errors, because if one bit gets flipped due to line noise, there will be an incorrect number of ones in the received data. In the two examples above, B's calculated parity value matches the parity bit in its received value, indicating there are no single bit errors. Consider the following example with a transmission error in the second bit:

Transmission sent using even parity:

```
A wants to transmit:          1001
A computes parity bit value:  1^0^0^1 = 0
A adds parity bit and sends:  10010
*** TRANSMISSION ERROR ***
B receives:                   11010
B computes overall parity:    1^1^0^1^0 = 1
B reports incorrect transmission after observing unexpected odd result.
```

B's calculated parity value (1) does not match the parity bit (0) in its received value, indicating the bit error. Here's the same example but now the parity bit itself gets corrupted:

```
A wants to transmit:           1001
A computes even parity value: 1^0^0^1 = 0
A sends:                       10010
*** TRANSMISSION ERROR ***
B receives:                    10011
B computes overall parity:     1^0^0^1^1 = 1
B reports incorrect transmission after observing unexpected odd result.
```

Once again, B's computes an odd overall parity, indicating the bit error.
There is a limitation to parity schemes. A parity bit is only guaranteed to detect an odd number of bit errors. If an even number of bits have errors, the parity bit records the correct number of ones, even though the data is corrupt. (See also error detection and correction.) Consider the same example as before with an even number of corrupted bits:

```
A wants to transmit:           1001
A computes even parity value: 1^0^0^1 = 0
A sends:                       10010
*** TRANSMISSION ERROR ***
B receives:                    11011
B computes overall parity:     1^1^0^1^1 = 0
B reports correct transmission though actually incorrect.
```

B observes even parity, as expected, thereby failing to catch the two bit errors.

Use of Parity

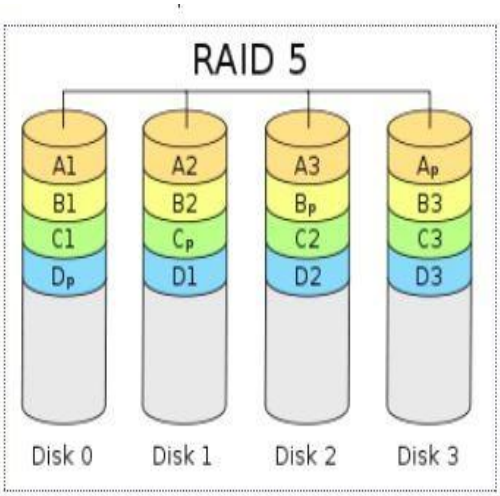
Because of its simplicity, parity is used in many hardware applications where an operation can be repeated in case of difficulty, or where simply detecting the error is helpful. For example, the SCSI and PCI buses use parity to detect transmission errors, and many microprocessor instruction caches include parity protection. Because the I-cache data is just a copy of main memory, it can be thrown away and re-fetched if it is found to be corrupted.
In serial data transmission, a common format is 7 data bit, an even parity bit, and one or two stop bits. This format neatly accommodates all the 7-bit ASCII characters in a convenient 8-bit byte. Other formats are possible; 8 bits of data plus a parity bit can convey all 8-bit byte values.
In serial communication contexts, parity is usually generated and checked by interface hardware (e.g., a UART) and, on reception, the result made available to the CPU (and so to, for instance, the operating system) via a status bit in a hardware register in the interface hardware. Recovery from the error is usually done by retransmitting the data, the details of which are usually handled by software (e.g., the operating system I/O routines).

Parity block

A parity block is used by certain RAID levels. Redundancy is achieved by the use of parity blocks. If a single drive in the array fails, data blocks and a parity block from the working drives can be combined to reconstruct the missing data.

Given the diagram below, where each column is a disk, assume A1 = 00000111, A2 = 00000101, and A3 = 00000000. Ap, generated by XORing A1, A2, and A3, will then equal 00000010. If the second drive fails, A2 will no longer be accessible, but can be reconstructed by XORing A1, A3, and Ap:

A1 XOR A3 XOR Ap = 00000101



Note: Data blocks are in the format A#, parity blocks Ap.

CODE CONVERSION

Binary To BCD Conversion

This model came about as a result of wondering if there was a simple hardware way of converting a binary number to Binary Coded Decimal (BCD). The clue required (courtesy of a quick search on the web) was found in a Xilinx Application note, XAPP029. This documents a serial binary to bcd conversion algorithm which, as usual, seems obvious once you've seen the idea!

The basic idea is to shift data serially into a shift register. As each bit is shifted in, the accumulated sum is collected. Each shift effectively doubles the value of the binary number in the four bit shift register which is going to hold the converted BCD digit.

Each time a bit is shifted in, the value in the shift register is doubled. After four bits have been shifted in, if the original value is 0, 1, 2, 3, or 4, then the result is within the 0-9 range of a BCD digit and there is no action required.

If the value is 5, 6, 7, 8, or 9, then the doubled result is greater than 10, so a carry out (called ModOut in the code) is generated to represent the overflow into the tens column (i.e. into the next BCD digit). Here's a little table showing how to double each bit in BCD. All numbers shown are decimal.

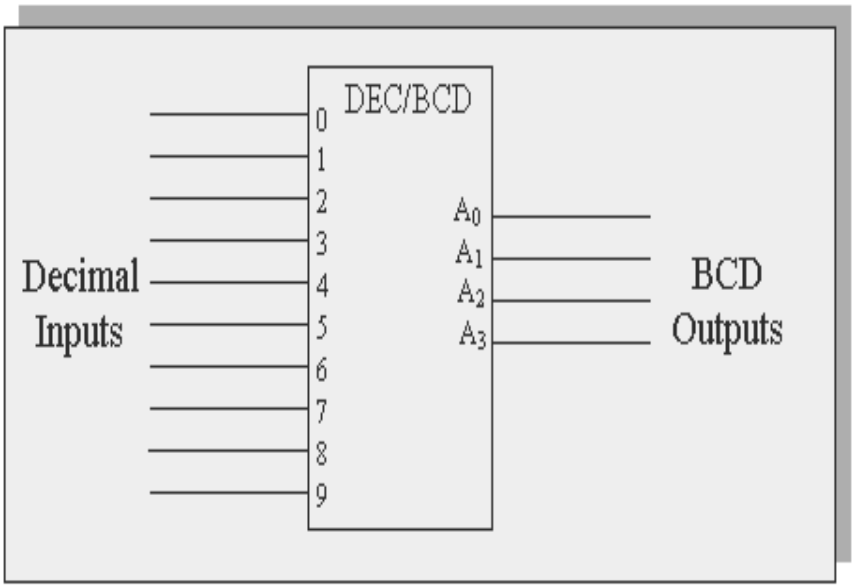
input	tens	units	result
0	0	0	0
1	0	2	2
2	0	4	4
3	0	6	6
4	0	8	8
5	1	0	10
6	1	2	12
7	1	4	14
8	1	6	16
9	1	8	18

The tens column thus represents an overflow into the next most significant BCD digit.

The Decimal-to-BCD Encoder

Recall that with BCD the ten decimal digits (0,1,…,9) are represented by their four-digit binary counterparts. Consequently, we expect the Decimal-to-BCD encoder to have 10 inputs and 4 outputs. The

logic symbol for this 10-line-to-4-line decoder is presented in Figure below and the associated conversion table listed in Table 2-13 with A3 representing the most significant bit.



Logic symbol for Decimal-to-BCD Encoder

Decimal Digit	BCD Code			
	A_3	A_2	A_1	A_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Decimal-to-BCD code table

Now, note that the MSB is defined by the function

A3 = 8 + 9

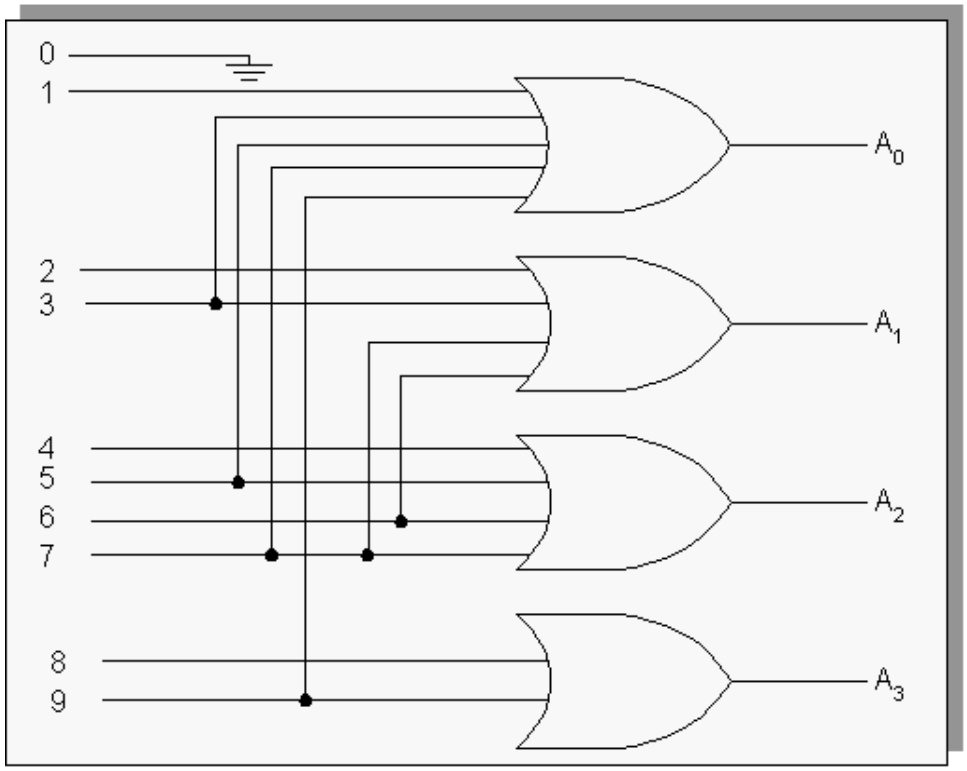
Similarly, A_2, A_1 and A_0 are defined by the functions

$$A_2 = 4 + 5 + 6 + 7$$

$$A_1 = 2 + 3 + 6 + 7$$

$$A_0 = 1 + 3 + 5 + 7 + 9$$

Now we can implement the logic circuit for the decimal-to-BCD-decoder as illustrated in Figure below:



Note that a connection for the decimal digit zero is not required as in this case the BCD outputs are all LOW when there are no HIGH inputs.

Octal-to-Binary Encoder

Octal-to-Binary take 8 inputs and provides 3 outputs, thus doing the opposite of what the 3-to-8 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of an Octal-to-binary encoder.

Truth Table

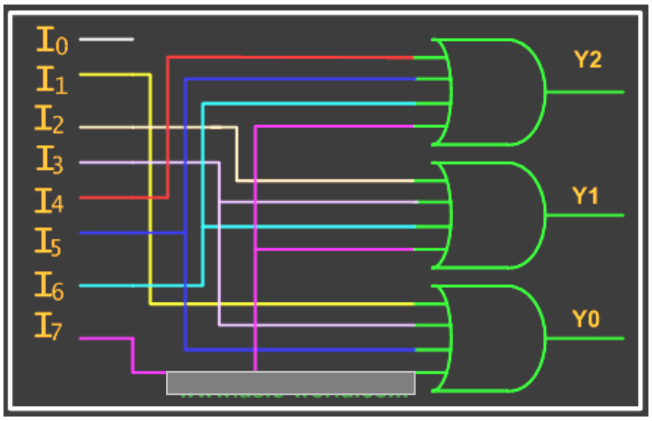
I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

Y0 = I1 + I3 + I5 + I7
Y1= I2 + I3 + I6 + I7
Y2 = I4 + I5 + I6 +I7

Based on the above equations, we can draw the circuit as shown below

Circuit



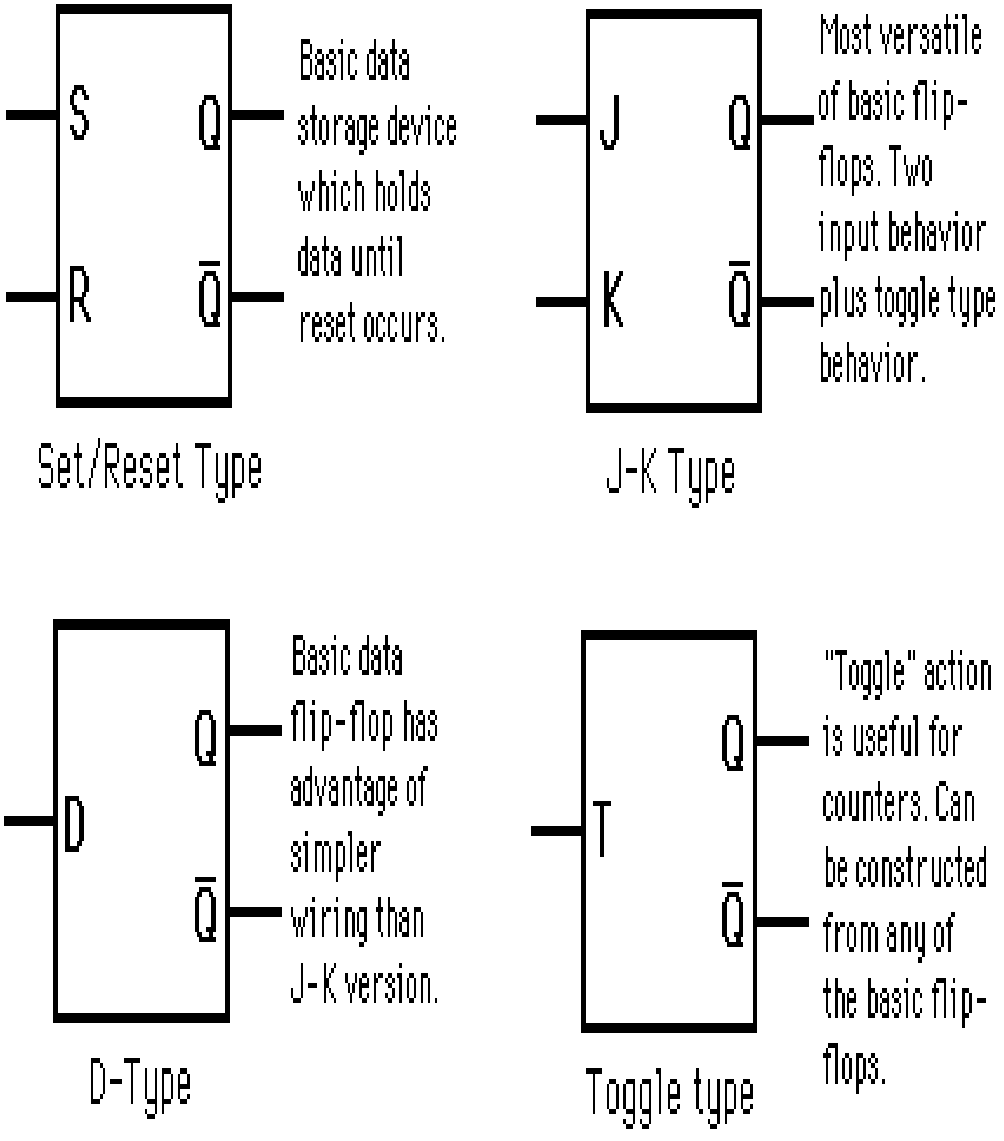
Chapter-4

Flip-Flop Devices

Flip-Flop

In digital circuits, a **flip-flop** is a term referring to an electronic circuit (a bistable multivibrator) that has two stable states and thereby is capable of serving as one bit of memory. Today, the term *flip-flop* has come to mostly denote *non-transparent* (*clocked* or *edge-triggered*) devices, while the simpler *transparent* ones are often referred to as latches; however, as this distinction is quite new, the two words are sometimes used interchangeably (see history).

A flip-flop is usually controlled by one or two control signals and/or a gate or clock signal. The output often includes the complement as well as the normal output. As flip-flops are implemented electronically, they require power and ground connections.



Set-Reset flip-flops (SR flip-flops)

The fundamental latch is the simple SR flip-flop , where S and R stand for set and reset respectively. It can be constructed from a pair of cross-coupled NOR logic gates. The stored bit is present on the output marked Q.
Normally, in storage mode, the S and R inputs are both low, and feedback maintains the Q and Q outputs in a constant state, with Q the complement of Q. If S (Set) is pulsed high while R is held low, then the Q output is forced high, and stays high even after S returns low; similarly, if R (Reset) is pulsed high while S is held low, then the Q output is forced low, and stays low even after R returns low.

SR Flip-Flop operation

Characteristic table			Excitation table				
S	R	Action	Q(t)	Q(t+1)	S	R	Action
0	0	Keep state	0	0	0	X	No change
0	1	Q = 0	0	1	1	0	Set
1	0	Q = 1	1	0	0	1	Reset
1	1	Unstable combination, see race condition	1	1	X	0	No change

('X' denotes a Don't care condition; meaning the signal is irrelevant)

Toggle flip-flops (T flip-flops)

If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. This behavior is described by the characteristic equation:

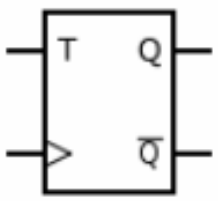
$Q_{next} = T \oplus Q$ (or, without benefit of the XOR operator,

the equivalent: $Q_{next} = T\overline{Q} + \overline{T}Q$)

and can be described in a truth table:

T Flip-Flop operation

Characteristic table				Excitation table			
T	Q	Q_{next}	Comment	Q	Q_{next}	T	Comment
0	0	0	hold state(no clk)	0	0	0	No change
0	1	1	hold state(no clk)	1	1	0	No change
1	0	1	toggle	0	1	1	Complement
1	1	0	toggle	1	0	1	Complement



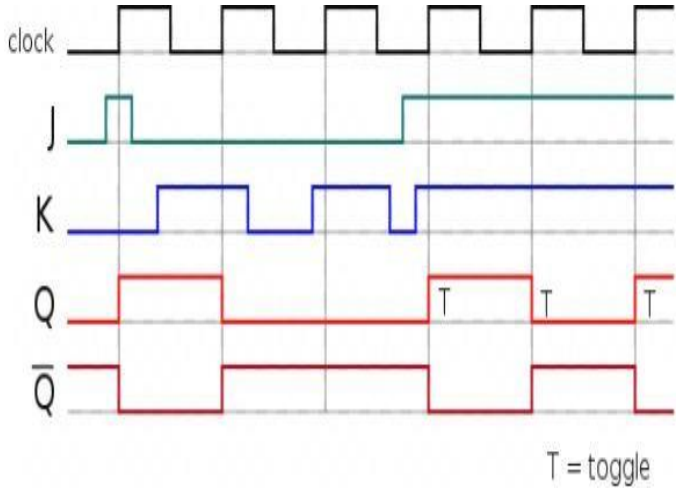
A circuit symbol for a T-type flip-flop, where > is the clock input, T is the toggle input and Q is the stored data output.

When T is held high, the toggle flip-flop divides the clock frequency by two; that is, if clock frequency is 4 MHz, the output frequency obtained from the flip-flop will be 2 MHz. This 'divide by' feature has application in various types of digital counters. A T flip-flop can also be built using a JK flip-flop (J & K pins are connected together and act as T) or D flip-flop (T input and Q is connected to the D input through an XOR gate).

JK flip-flop

The **JK** flip-flop augments the behavior of the SR flip-flop (J=Set, K=Reset) by interpreting the S = R = 1 condition as a "flip" or toggle command. Specifically, the combination J = 1, K = 0 is a command to set the flip-flop; the combination J = 0, K = 1 is a command to reset the flip-flop; and the combination J = K = 1 is a command to toggle the flip-flop, i.e., change its output to the logical complement of its current value. Setting J = K = 0 does NOT result in a D flip-flop, but rather, will hold the current state. To synthesize a D flip-flop, simply set K equal to the complement of J. The JK flip-flop is therefore a universal flip-flop, because it can be configured to work as an SR flip-flop, a D flip-flop, or a T flip-flop.

NOTE: The flip flop is positive edge triggered (Clock Pulse) as seen in the timing diagram.



JK flip-flop timing diagram

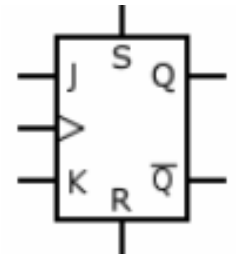
The characteristic equation of the JK flip-flop is:

$$Q_{next} = J\overline{Q} + \overline{K}Q$$

and the corresponding truth table is:

JK Flip Flop operation

Characteristic table				Excitation table				
J	K	Q_{next}	Comment	Q	Q_{next}	J	K	Comment
0	0	Q_{prev}	hold state	0	0	0	X	No change
0	1	0	reset	0	1	1	X	Set
1	0	1	set	1	0	X	1	Reset
1	1	$\overline{Q_{prev}}$	toggle	1	1	X	0	No change



A circuit symbol for a JK flip-flop, where > is the clock input, J and K are data inputs, Q is the stored data output, and Q' is the inverse of Q.

The origin of the name for the JK flip-flop is detailed by P. L. Lindley, a JPL engineer, in a letter to *EDN*, an electronics design magazine. The letter is dated June 13, 1968, and was published in the August edition of the newsletter. In the letter, Mr. Lindley explains that he heard the story of the JK flip-flop from Dr. Eldred Nelson, who is responsible for coining the term while working at Hughes Aircraft. Flip-flops in use at Hughes at the time were all of the type that came to be known as J-K. In designing a logical system, Dr. Nelson assigned letters to flip-flop inputs as follows: #1: A & B, #2: C & D, #3: E & F, #4: G & H, #5: J & K.

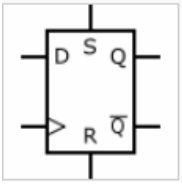
Another theory holds that the set and reset inputs were given the symbols "J" and "K" after one of the engineers that helped design the J-K flip-flop, Jack Kilby.

D flip-flop

The Q output always takes on the state of the D input at the moment of a rising clock edge. (or falling edge if the clock input is active low) It is called the **D** flip-flop for this reason, since the output takes the value of the **D** input or *Data* input, and *Delays* it by one clock count. The D flip-flop can be interpreted as a primitive memory cell, zero-order hold, or delay line.

Truth table:

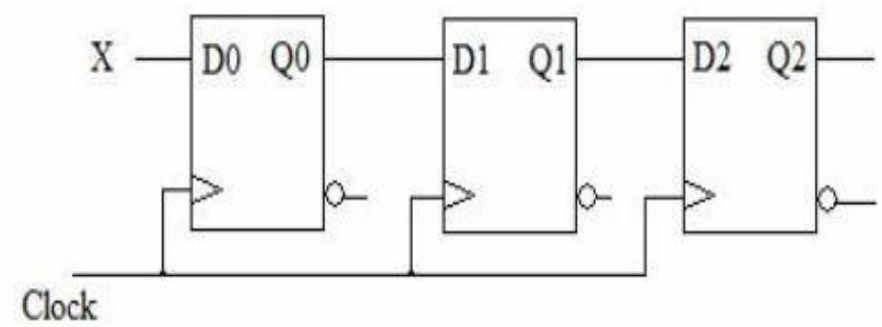
Clock	D	Q	Q_{prev}
Rising edge	0	0	X
Rising edge	1	1	X
Non-Rising	X	Q_{prev}	



D flip-flop symbol

('X' denotes a *Don't care* condition, meaning the signal is irrelevant)

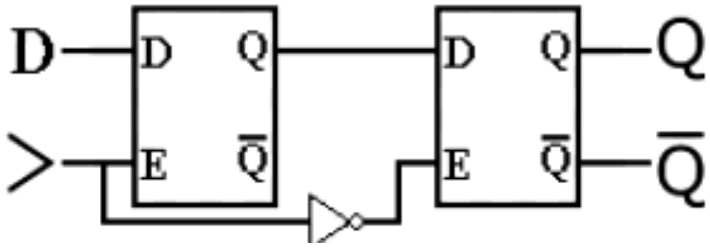
These flip flops are very useful, as they form the basis for shift registers, which are an essential part of many electronic devices. The advantage of the D flip-flop over the D-type latch is that it "captures" the signal at the moment the clock goes high, and subsequent changes of the data line do not influence Q until the next rising clock edge. An exception is that some flip-flops have a 'reset' signal input, which will reset Q (to zero), and may be either asynchronous or synchronous with the clock.
The above circuit shifts the contents of the register to the right, one bit position on each active transition of the clock. The input X is shifted into the leftmost bit position.



3-bit shift register

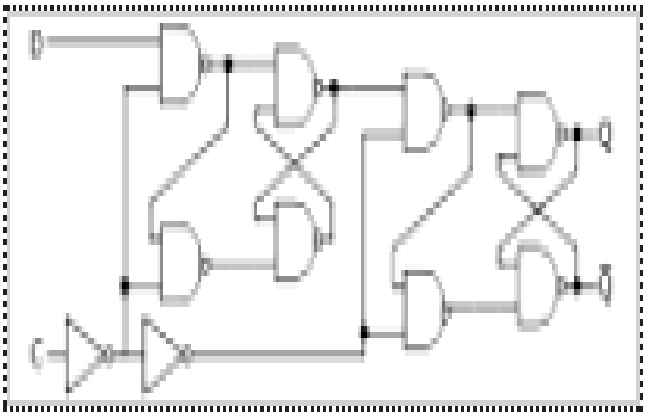
Master-slave (pulse-triggered) D flip-flop

A master-slave D flip-flop is created by connecting two gated D latches in series, and inverting the *enable* input to one of them. It is called master-slave because the second latch in the series only changes in response to a change in the first (master) latch. The term pulse triggered means that data are entered on the rising edge of the clock pulse, but the output doesn't reflect the change until the falling edge of the clock pulse.



A master slave D flip flop. It responds on the negative edge of the enable input (usually a clock).

For a positive-edge triggered master-slave D flip-flop, when the clock signal is low (logical 0) the "enable" seen by the first or "master" D latch (the inverted clock signal) is high (logical 1). This allows the "master" latch to store the input value when the clock signal transitions from low to high. As the clock signal goes high (0 to 1) the inverted "enable" of the first latch goes low (1 to 0) and the value seen at the input to the master latch is "locked". Nearly simultaneously, the twice inverted "enable" of the second or "slave" D latch transitions from low to high (0 to 1) with the clock signal. This allows the signal captured at the rising edge of the clock by the now "locked" master latch to pass through the "slave" latch. When the clock signal returns to low (1 to 0), the output of the "slave" latch is "locked", and the value seen at the last rising edge of the clock is held while the "master" latch begins to accept new values in preparation for the next rising clock edge.



By removing the left-most inverter in the above circuit, a D-type flip flop that strobes on the *falling edge* of a clock signal can be obtained. This has a truth table like this:

D	Q	>	Q_{next}
0	X	Falling	0
1	X	Falling	1

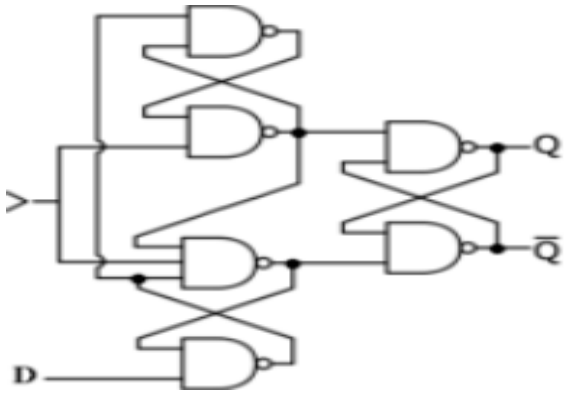
Most D-type flip-flops in ICs have the capability to be set and reset, much like an SR flip-flop. Usually, the illegal S = R = 1 condition is resolved in D-type flip-flops.

Inputs				Outputs	
S	R	D	>	Q	Q'
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	1	1

By setting S = R = 0, the flip-flop can be used as described above.

Edge-triggered D flip-flop

A more efficient way to make a D flip-flop is not so easy to understand, but it works the same way. While the master-slave D flip flop is also triggered on the edge of a clock, its components are each triggered by clock levels. The "edge-triggered D flip flop" does not have the master slave properties.



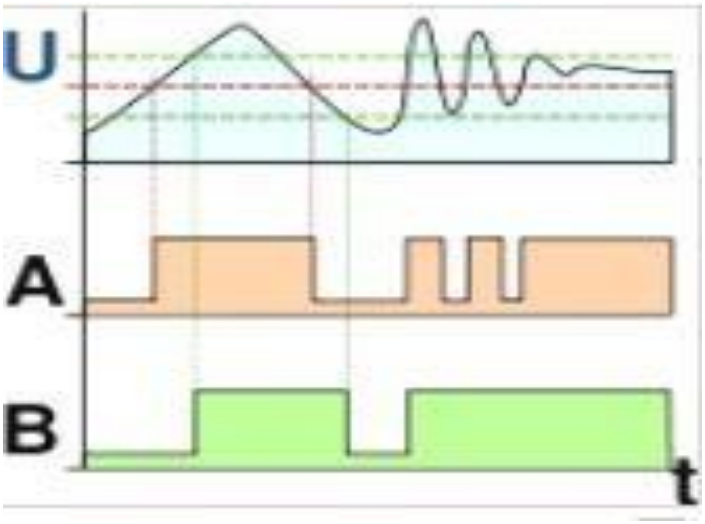
A positive-edge-triggered D flip-flop.

Schmitt trigger

In electronics, a **Schmitt trigger** is a comparator circuit that incorporates positive feedback.

When the input is higher than a certain chosen threshold, the output is high; when the input is below another (lower) chosen threshold, the output is low; when the input is between the two, the output retains its value. The trigger is so named because the output retains its value until the input changes sufficiently to trigger a change. This dual threshold action is called *hysteresis*, and implies that the Schmitt trigger has some memory.

The benefit of a Schmitt trigger over a circuit with only a single input threshold is greater stability (noise immunity). With only one input threshold, a noisy input signal near that threshold could cause the output to switch rapidly back and forth from noise alone. A noisy Schmitt Trigger input signal near one threshold can cause only one switch in output value, after which it would have to move beyond the other threshold in order to cause another switch.



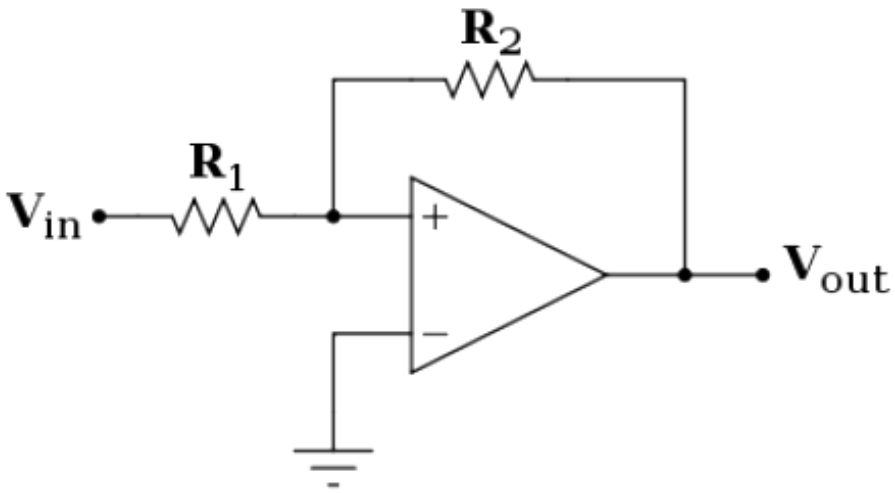
The effect of using a Schmitt trigger (B) instead of a comparator (A).

The symbol for Schmitt triggers in circuit diagrams is a triangle with a hysteresis symbol. The symbol depicts a typical hysteresis curve.

Standard Schmitt trigger Inverting Schmitt trigger

When the Schmitt trigger is inverting (i.e., when very negative inputs lead to positive outputs and vice versa), the hysteresis symbol is top–bottom mirrored.

Today Schmitt triggers are typically built around comparators, connected to have positive feedback instead of the usual negative feedback. For this circuit the switching occurs near ground, with the amount of hysteresis controlled by the resistances of R_1 and R_2:



The comparator gives out the highest voltage it can, $+V_S$, when the non-inverting (+) input is at a higher voltage than the inverting (-) input, and then switches to the lowest output voltage it can, $-V_S$, when the positive input drops below the negative input. For very negative inputs, the output will be low, and for very positive inputs, the output will be high, and so this is an implementation of a "non-inverting" Schmitt trigger.

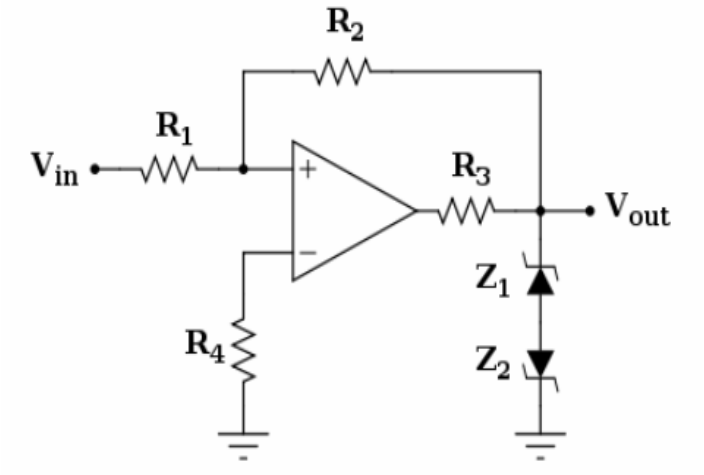
For instance, if the Schmitt trigger is currently in the high state, the output will be at the positive power supply rail ($+V_S$). V_+ is then a voltage divider between V_{in} and $+V_S$. The comparator will switch when $V_+=0$ (ground). Current conservation shows that this requires

$$V_{in}/R_1 = -V_S/R_2,$$

and so V_{in} must drop below $-(R_1/R_2)V_S$ to get the output to switch. Once the comparator output has switched to $-V_S$, the threshold becomes $+(R_1/R_2)V_S$ to switch back to high.

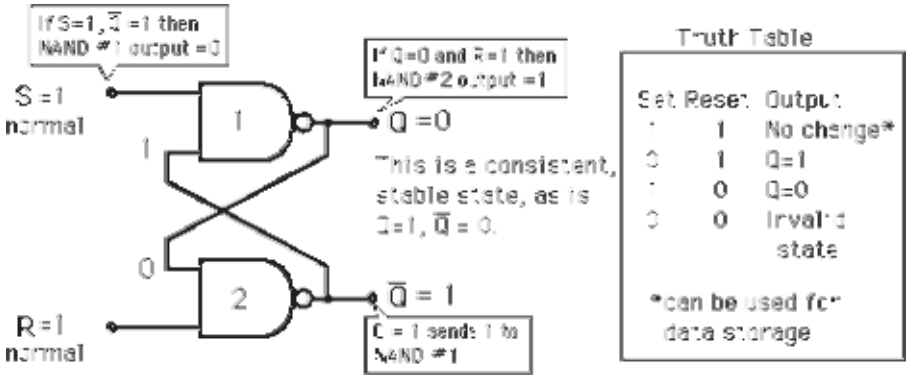
So this circuit creates a switching band centered around zero, with trigger levels $\pm(R_1/R_2)V_S$. The input voltage must rise above the top of the band, and then below the bottom of the band, for the output to switch on and then back off. If R_1 is zero or R_2 is infinity (i.e., an open circuit), the band collapses to zero width, and it behaves as a standard comparator. The output characteristic is shown in the picture on the right. The value of the threshold T is given by $(R_1/R_2)V_S$ and the maximum value of the output M is the power supply rail.

A possible structure of a more realistic configuration is the following:

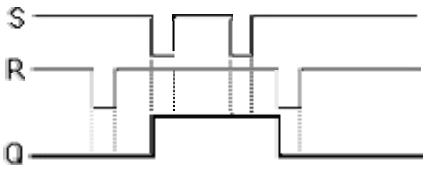


The output characteristic has exactly the same shape of the previous basic configuration and the threshold values are the same as well. On the other hand, in the previous case the output voltage was depending on the power supply, while now it is defined by the Zener diodes: this way the output can be modified and it is much more stable. The resistor R_3 is there to limit the current through the diodes, while R_4 is there to minimize the input voltage offset caused by the op-amp's input bias currents (see *Limitations of real op-amps*).

NAND-gate Latch

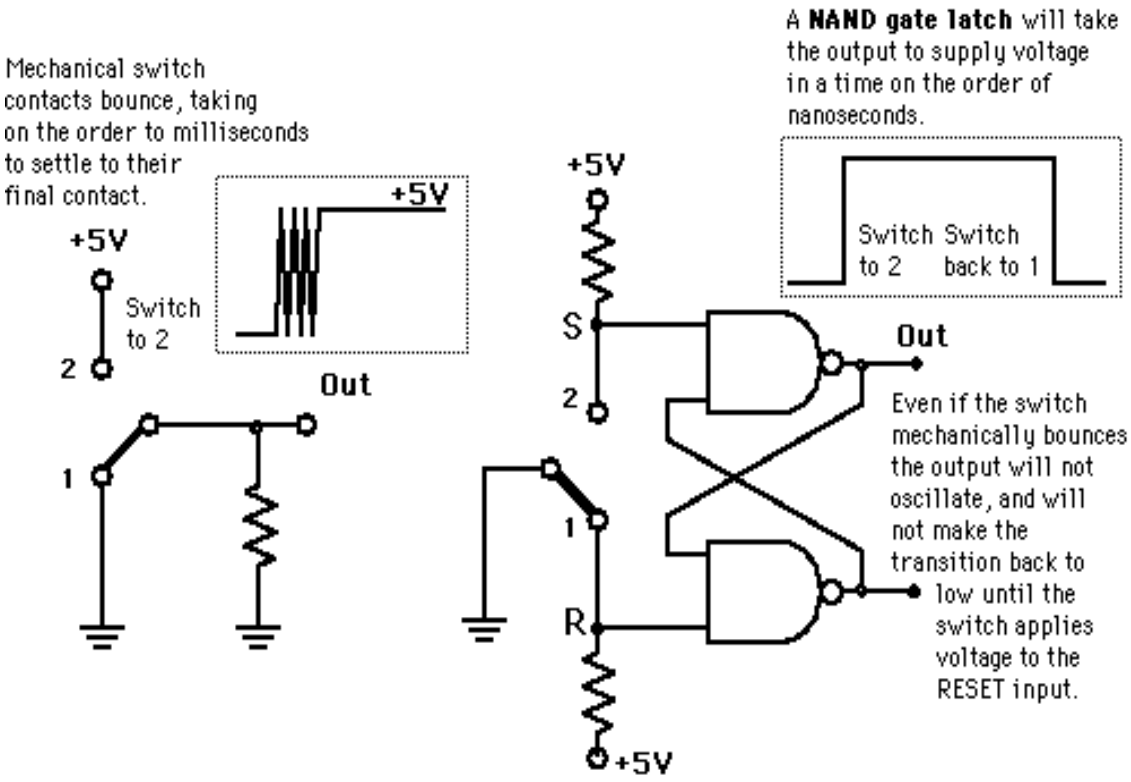


The time sequence at right shows the conditions under which the set and reset inputs cause a state change, and when they don't.



The concept of a "latch" circuit is important to creating memory devices. The function of such a circuit is to "latch" the value created by the input signal to the device and hold that value until some other signal changes it.

Switch Debouncing



Chapter-5

Registers and Counters

Introduction

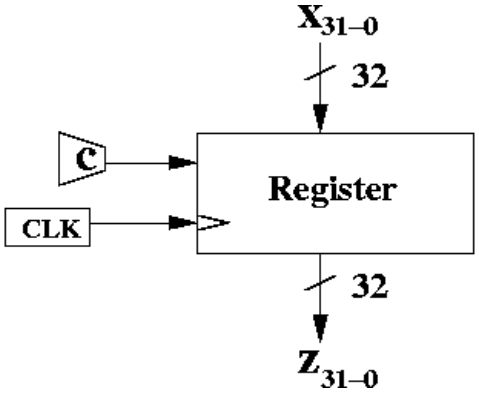
In computer architecture, a processor register is a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere. Most, but not all, modern computer architectures operate on the principle of moving data from main memory into registers, operating on them, then moving the result back into main memory—a so-called load-store architecture. A common property of computer programs is locality of reference: the same values are often accessed repeatedly; and holding these frequently used values in registers improves program execution performance.
Processor registers are at the top of the memory hierarchy, and provide the fastest way for a CPU to access data. The term is often used to refer only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. More properly, these are called the "architectural registers". For instance, the x86 instruction set defines a set of eight 32-bit registers, but a CPU that implements the x86 instruction set will often contain many more registers than just these eight.
Allocating frequently used variables to registers can be critical to a program's performance. This action, namely register allocation is performed by a compiler in the code generation phase.

A CPU consists of registers. Registers on modern CPUs usually store 32 bit or 64 bits. Registers are made from flip flops.

We're going to look at two ways to make a parallel-load register. A parallel-load register has two basic operations, which are summarized by the following table.

c	Operation
0	Hold
1	Parallel Load ($z_{31-0} = x_{31-0}$)

Here's a black box diagram for a parallel load register.



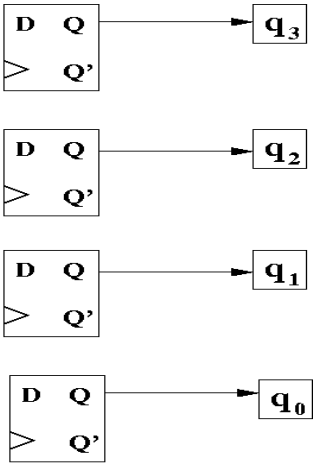
A 32 bit register has 32 bits of data input, x_{31-0}. It also has 32 bits of output, which is labelled, z_{31-0}.

There's also a clock input. The register can only be updated at positive clock edges. If **c = 0** (at the positive clock edge), then the output of the register is unchanged. If **c = 1** (at the positive clock edge), x_{31-0}, is loaded in parallel, and those 32 bits are output as z_{31-0}.

Implementing a 4-bit parallel load register with D flip flops

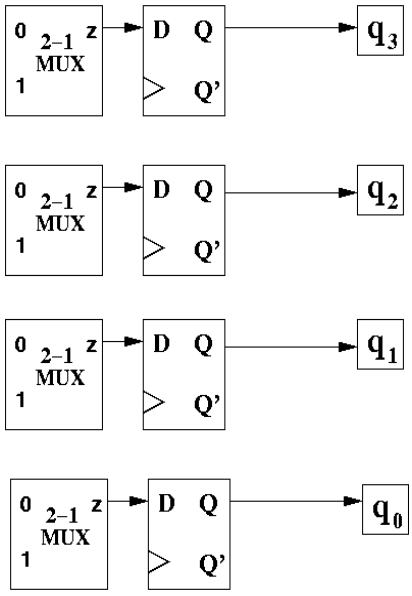
Implementing a 32-bit register is like implementing a 4-bit parallel load register, except with more flip flops. So, we'll do it with 4 flip flops, instead. (Although, by the same reasoning, we could do it with, say, 2 flip flops).

Since we need to store 4 bits, we'll need 4 flip flops. Here's the initial diagram.



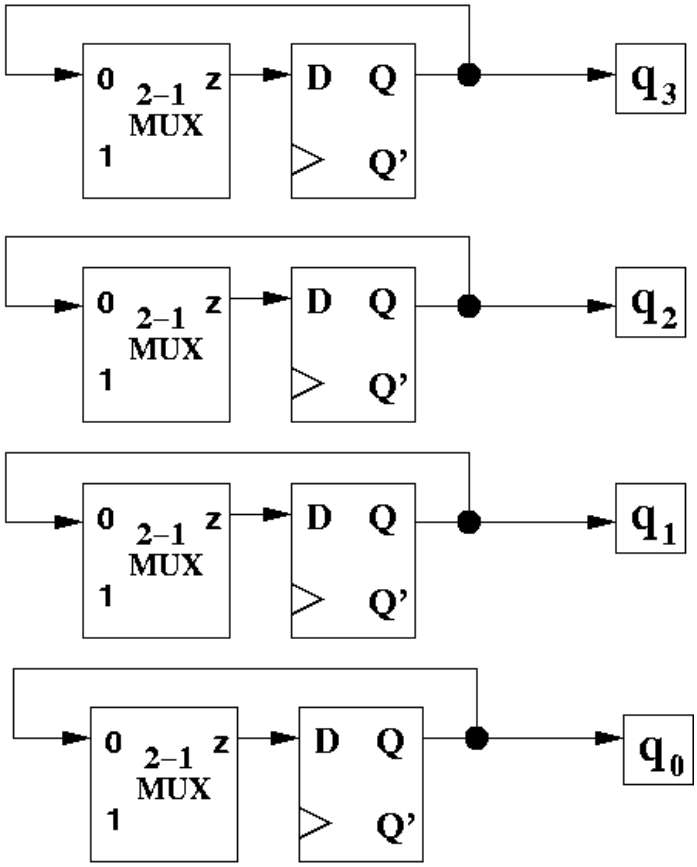
At this point, we need to figure out how to make a choice between two options: we either want to hold, or we want to parallel load.

What combinational logic device can we use that allows us to pick between one of many different inputs (or in this case, between one of two inputs)? We should use a 2-1 MUX!

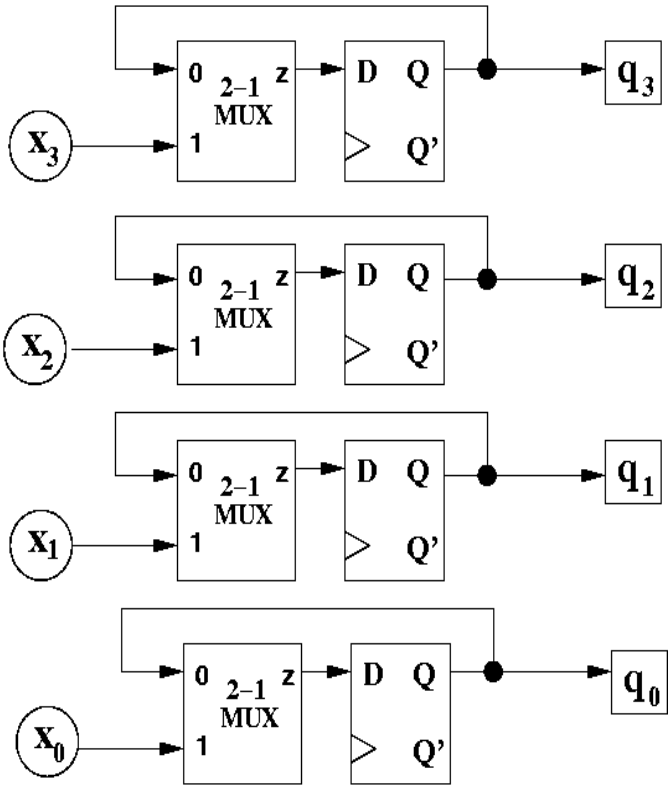


For the "0" input, we want to *hold* the value. What does that mean? Each flip flop is currently outputting a bit. We want to maintain that same value.

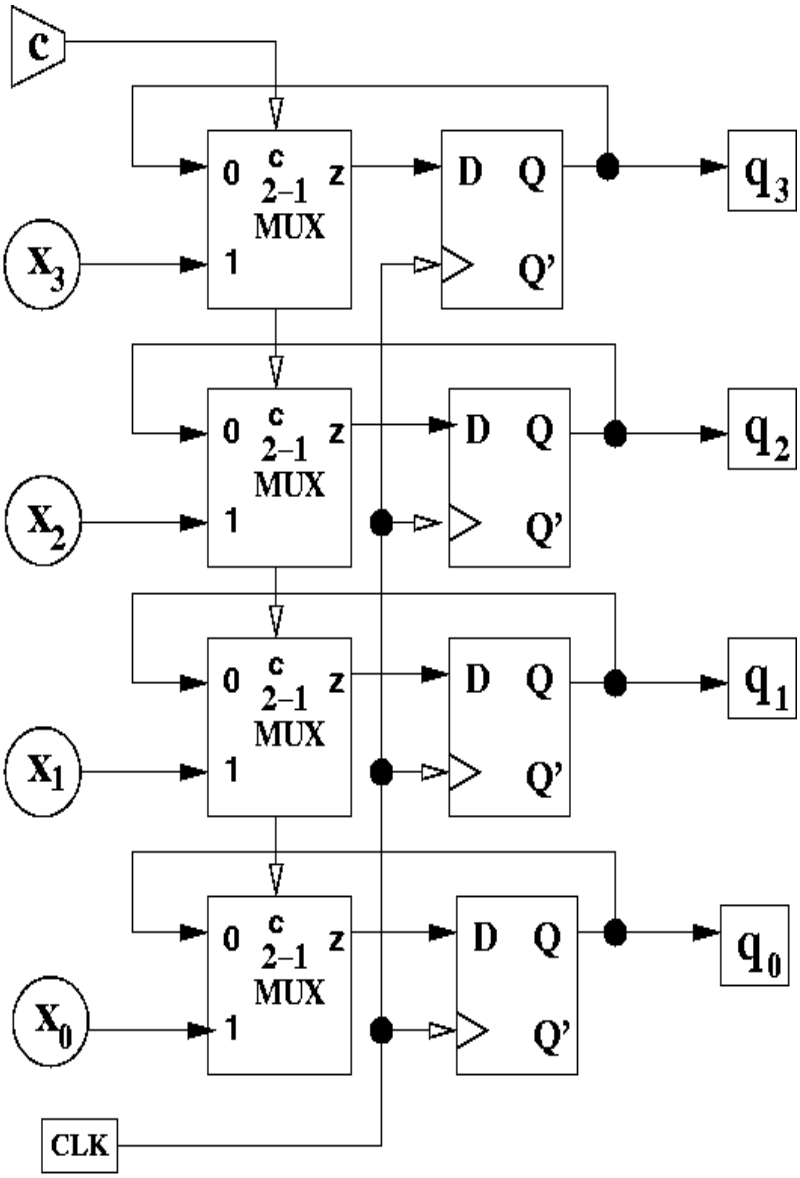
How can we do that with a D flip flop? Recall that a D flip flop reads in the value of D, and makes that the output, **Q**. If we want to have the output stay the same, we need to feed back the output back into the input, through input 0 of the MUX.



When **c = 1** we want to parallel load the input. We accomplish this by feeding in the input x_{3-0} into the "1" input of the 2-1 MUX.



The circuit isn't quite complete. We should add the clock and the control input, as well.



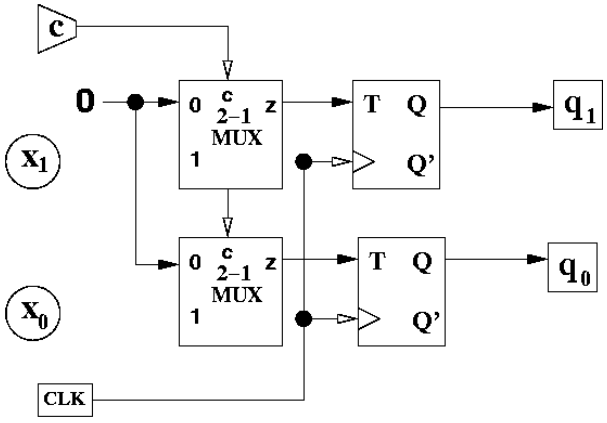
The control input is drawn in a "cheating" way. I draw the line going in at the top MUX, and being drawn "straight through" to the other MUXes. You can choose to draw it this way, if you want, or draw it less ambiguously.

Implementing a 2-bit parallel load register with T flip flops

Now, let's implement a parallel load register with T flip flops, instead. To save on space, we'll use 2 bits instead of 4. It should be easy to convert it to 4 bits if necessary.

The first question is how to "hold" using T flip flops. For D flip flops, we did that by feeding the output of the D flip flop back into the input (through the MUX).

We don't want to feed the output back, because that's not how T flip flops work (try to figure out happens if you *do* feed it back). Recall the two operations of a T flip flop: hold and toggle. You already have a way to "hold" the value. Input the value 0 into the T input!



The hard part is to make a T flip flop parallel load. It's quite easy to do it for D flip flop, since all you do is feed in the input directly to D.

So what does it mean to parallel load? It means to get a flip flop to output the value of **x**. So, let's think about what that means. Suppose the flip flop current outputs 1, and we want to parallel load 0. This means, we need to toggle the output, so that it changes the output from 1 to 0.

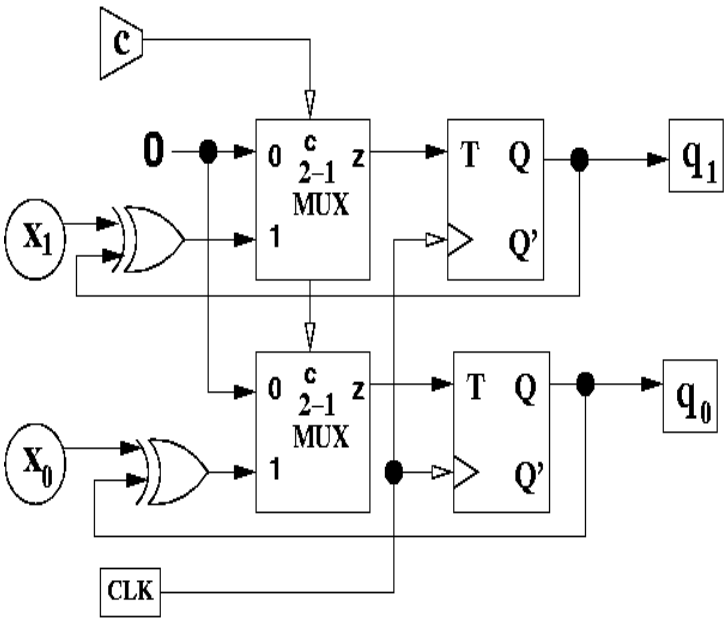
Let's come up with a table to list out all the possibilities.

What we want to load (x)	What is currently being output (q)	What we need to do (T)
x = 0	**q = 0**	HOLD (**T = 0**)
x = 0	**q = 1**	TOGGLE (**T = 1**)
x = 1	**q = 0**	TOGGLE (**T = 1**)
x = 1	**q = 1**	HOLD (**T = 0**)

We can derive a Boolean expression for T. If you look carefully, it's just the XOR of **x** and **q**.

```
T = x XOR q
```

Here's the circuit:



Variations

If you want to work on a similar problem, you can add features to the parallel load register. For example, here's an expanded table.

c_1c_0	Operation
00	Hold
01	Parallel Load ($z_{31\text{-}0} = x_{31\text{-}0}$)
10	Logical Shift Left 1 bit
11	Logical Shift Right 1 bit

This requires a 4-1 MUX.

Categories of registers

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". Registers are now usually implemented as a register file, but they have also been implemented using individual flip-flops, high speed core memory, thin film memory, and other ways in various machines.

A processor often contains several kinds of registers, that can be classified according to their content or instructions that operate on them:

- **User-accessible Registers** - The most common division of user-accessible registers is into data registers and address registers.
- **Data registers** are used to hold numeric values such as integer and floating-point values. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
- **Address registers** hold addresses and are used by instructions that indirectly access memory.
    - Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist.
    - A stack pointer, sometimes called a stack register, is the name given to a register that can be used bysome instructions to maintain a stack (data structure).
- **Conditional registers** hold truth values often used to determine whether some instruction should or should not be executed.
- **General purpose registers** (**GPR**s) can store both data and addresses, i.e., they are combined Data/Address registers.
- **Floating point registers** (**FPR**s) store floating point numbers in many architectures.
- **Constant registers** hold read-only values such as zero, one, or pi.
- **Vector registers** hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- **Special purpose registers** hold program state; they usually include the program counter (aka instruction pointer), stack pointer, and status register (aka processor status word). In embedded microprocessors, they can also correspond to specialised hardware elements.
    - **Instruction registers** store the instruction currently being executed.
- In some architectures, **model-specific registers** (also called *machine-specific registers*) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.

- **Control and status registers** - It has three types. Program counter, instruction registers, Program status word (PSW).
- Registers related to fetching information from RAM, a collection of storage registers located on separate chips from the CPU (unlike most of the above, these are generally not *architectural* registers):
    - Memory buffer register
    - Memory data register
    - Memory address register
    - Memory Type Range Registers

Hardware registers are similar, but occur outside CPUs.

SHIFT REGISTER

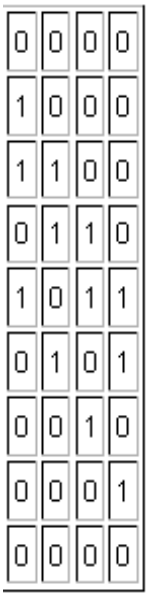
In digital circuits, a shift register is a group of flip flops set up in a linear fashion which have their inputs and outputs connected together in such a way that the data is shifted down the line when the circuit is activated.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as serial-in, parallel-out (SIPO) or as parallel-in, serial-out (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also bi-directional shift registers which allow you to vary the direction of the shift register. The serial input and outputs of a register can also be connected together to create a circular shift register. One could also create multi-dimensional shift registers, which can perform more complex computation.

Serial-in, serial-out

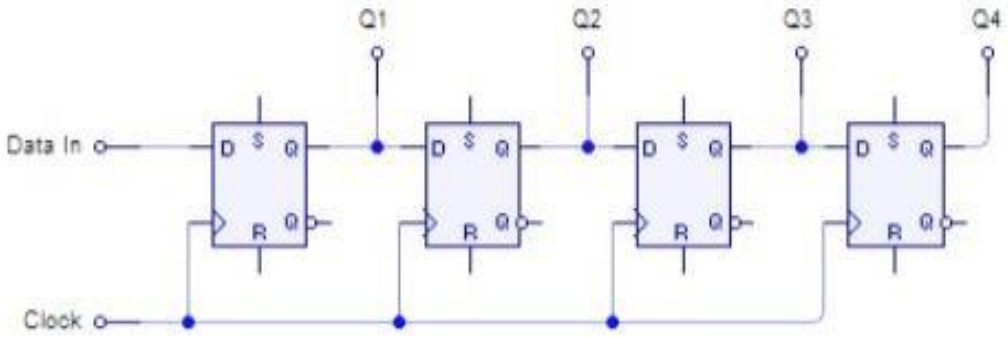
Destructive readout

These are the simplest kind of shift register. The data string is presented at 'Data In', and is shifted right one stage each time 'Data Advance' is brought high. At each advance, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output. The bit on the far right (i.e. 'Data Out') is shifted out and lost.
The data are stored after each flip-flop on the 'Q' output, so there are four storage 'slots' available in this arrangement, hence it is a 4-Bit Register. To give an idea of the shifting pattern, imagine that the register holds 0000 (so all storage slots are empty). As 'Data In' presents 1,1,0,1,0,0,0,0 (in that order, with a pulse at 'Data Advance' each time. This is called clocking or strobing) to the register, this is the result. The left hand column corresponds to the left-most flip-flop's output pin, and so on.
So the serial output of the entire register is 11010000 (). As you can see if we were to continue to input data, we would get exactly what was put in, but offset by four 'Data Advance' cycles. This arrangement is the hardware equivalent of a queue. Also, at any time, the whole register can be set to zero by bringing the reset (R) pins high.
This arrangement performs destructive readout - each datum is lost once it been shifted out of the right-most bit.



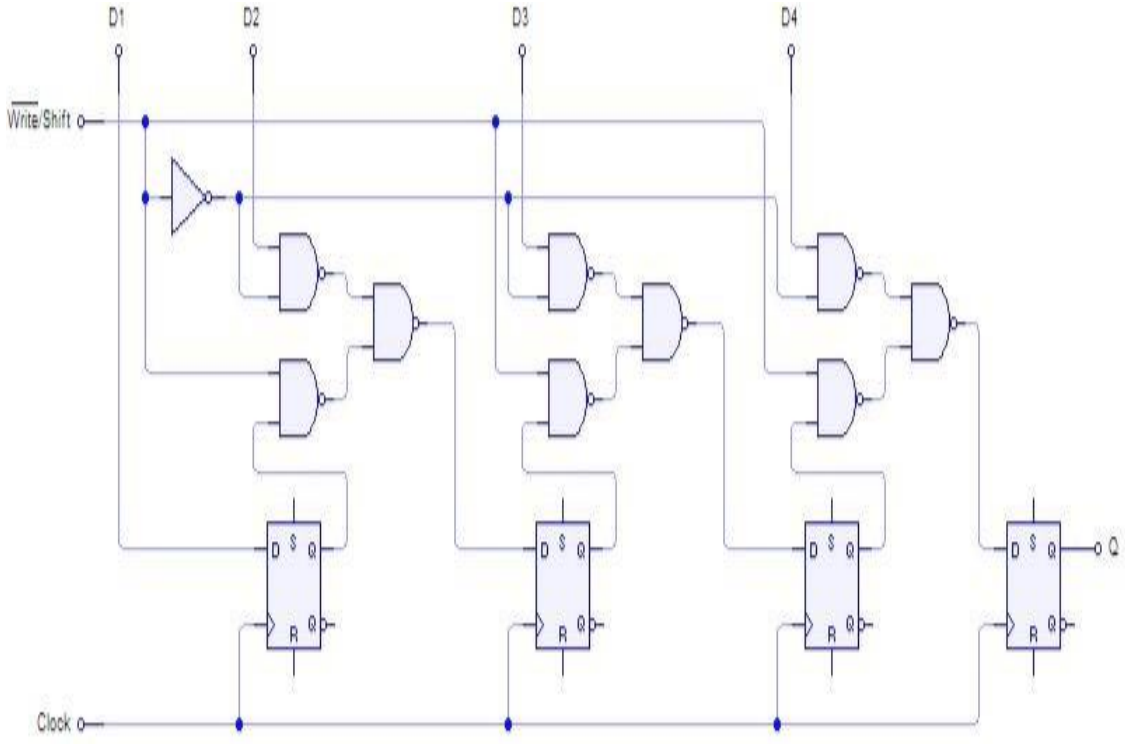
Serial-in, parallel-out

This configuration allows conversion from serial to parallel format. Data are input serially, as described in the SISO section above. Once the data has been input, it may be either read off at each output simultaneously, or it can be shifted out and replaced.



Parallel-in, serial-out

This configuration has the data input on lines D1 through D4 in parallel format. To write the data to the register, the Write/Shift control line must be held LOW. To shift the data, the W/S control line is brought HIGH and the registers are clocked. The arrangement now acts as a SISO shift register, with D1 as the Data Input. However, as long as the number of clock cycles is not more than the length of the data-string, the Data Output, Q, will be the parallel data read off in order.



4-Bit PISO Shift Register

COUNTER

In digital logic and computing, a **counter** is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. In practice, there are two types of counters:

- up counters, which increase (increment) in value
- down counters, which decrease (decrement) in value

In electronics, counters can be implemented quite easily using register-type circuits such as the flip-flop, and a wide variety of designs exist, e.g:

- Asynchronous (ripple) counters
- Synchronous counters
- Johnson counters
- Decade counters
- Up-Down counters

This counter counts both orders up and down by changing the supply.

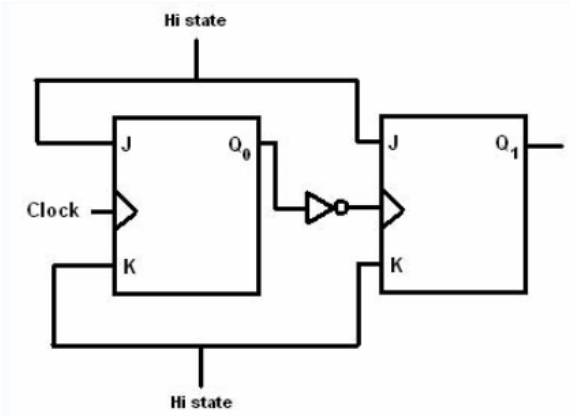
- Ring counters

It is a counter which is formed by enclosed shift registers

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in binary, or sometimes binary coded decimal. Many types of counter circuit are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Asynchronous (ripple) counter

The simplest counter circuit is a single D-type flip-flop, with its D (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), you will get another 1 bit counter that counts half as fast. Putting them together yields a two bit counter:



Asynchronous Counter created from JK flip-flops.

You can continue to add additional flip-flops, always inverting the output to its own input, and using the output from the previous flip-flop as the clock signal. The result is called a ripple counter, which can count to 2^n-1 where n is the number of bits (flip-flop stages) in the counter. Ripple counters suffer from unstable outputs as the overflows "ripple" from stage to stage, but they do find frequent application as dividers for clock signals, where the instantaneous count is unimportant, but the division ratio overall is. (To clarify this, a 1-bit counter is exactly equivalent to a divide by two circuit – the output frequency is exactly half that of the input when fed with a regular train of clock pulses).

The use of flip-flop outputs as clocks leads to timing skew between the count data bits, making this ripple technique incompatible with normal synchronous circuit design styles.

Cycle	Q1	Q0	(Q1:Q0)dec
0	0	0	0
1	0	1	1
2	1	0	2
3	1	1	3
4	0	0	0

Synchronous counter

Where a stable count value is important across several bits, which is the case in most counter systems, synchronous counters are used. These also use flip-flops, either the D-type or the more complex J-K type, but here, each stage is clocked simultaneously by a common clock signal. Logic gates between each stage of the circuit control data flow from stage to stage so that the desired count behavior is realized. Synchronous counters can be designed to count up or down, or both according to a direction input, and may be presetable via a set of parallel "jam" inputs. Most types of hardware-based counter are of this type. A simple way of implementing the logic for each bit of an ascending counter (which is what is shown in the image to the right) is for each bit to toggle when all of the less significant bits are at a logic high state. For example, bit 1 toggles when bit 0 is logic high; bit 2 toggles when both bit 1 and bit 0 are logic high; bit 3 toggles when bit 2, bit 1 and bit 0 are all high; and so on.

Synchronous counters can also be implemented with hardware finite state machines, which are more complex but allow for smoother, more stable transitions.

Ring counter

A ring counter is a shift register (a cascade connection of flip-flops) with the output of the last one connected to the input of the first, that is, in a ring. Typically a pattern consisting of a single 1 bit is circulated, so the state repeats every N clock cycles if N flip-flops are used. It can be used as a cycle counter of N states.

Johnson counter

A Johnson counter (or switchtail ring counter, twisted-ring counter, walking-ring counter, or Moebius counter) is a modified ring counter, where the output from the last stage is inverted and fed back as input to the first stage. A pattern of bits equal in length to twice the length of the shift register thus circulates indefinitely. These counters find specialist applications, including those similar to the decade counter, digital to analogue conversion, etc.

Decade counter

A decade counter is one that counts in decimal digits, rather than binary. A decimal counter may have each digit binary encoded (that is, it may count in binary-coded decimal, as the 7490 integrated circuit did) or other binary encodings (such as the bi-quinary encoding of the 7490 integrated circuit). Alternatively, it may have a "fully decoded" or one-hot output code in which each output goes high in turn; the 4017 was such a circuit. The latter type of circuit finds applications in multiplexers and demultiplexers, or wherever a scanning type of behaviour is useful. Similar counters with different numbers of outputs are also common.

Up–down counter

A counter that can change state in either direction, under control an up–down selector input, is known as an up–down counter. When the selector is in the up state, the counter increments its value; when the selector is in the down state, the counter decrements the count.

In computer science

In computability theory, a **counter** is considered a type of memory. A counter stores a single natural number (initially zero) and can be arbitrarily many digits long. A counter is usually considered in conjunction with a finite state machine (FSM), which can perform the following operations on the counter:

- Check whether the counter is zero
- Increment the counter by one
- Decrement the counter by one (if it's already zero, this leaves it unchanged).

The following machines are listed in order of power, with each one being strictly more powerful than the one below it:

1. Deterministic or non-deterministic FSM plus two counters
2. Non-deterministic FSM plus one stack
3. Non-deterministic FSM plus one counter
4. Deterministic FSM plus one counter
5. Deterministic or non-deterministic FSM

For the first and last, it doesn't matter whether the FSM is deterministic or non-deterministic (see determinism). They have equivalent power. The first two and the last one are levels of the Chomsky hierarchy.

The first machine, an FSM plus two counters, is equivalent in power to a Turing machine. See the article on register machines for a proof.

In the Internet

A web counter counts how many time a website or certain page has been viewed. They are usually accurate but some controversies have started regarding if they are accurate or not. This is usually because most counters count the number of unique hits a page gets while others count how many times a page was viewed even if the same person viewed it repeatedly.

Chapter-6

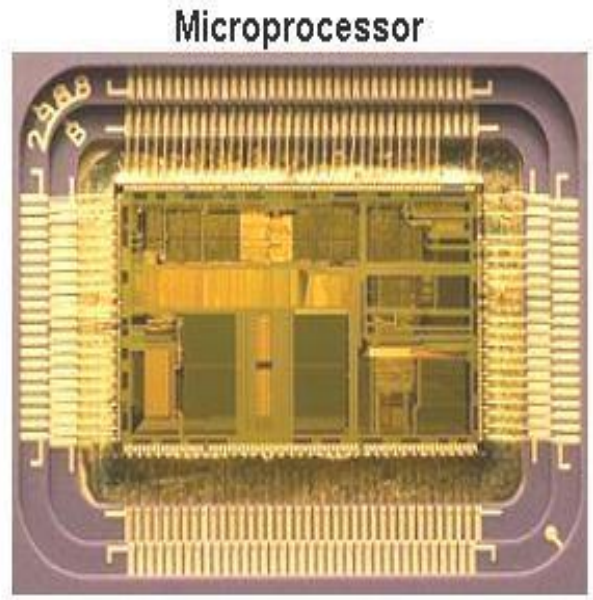
Microprocessors

Introduction

A microprocessor incorporates most or all of the functions of a central processing unit (CPU) on a single integrated circuit (IC). The first microprocessors emerged in the early 1970s and were used for electronic calculators, using Binary-coded decimal (BCD) arithmetic on 4-bit words. Other embedded uses of 4- and 8-bit microprocessors, such as terminals, printers, various kinds of automation etc, followed rather quickly. Affordable 8-bit microprocessors with 16-bit addressing also led to the first general purpose microcomputers in the mid-1970s.

Computer processors were for a long period constructed out of small and medium-scale ICs containing the equivalent of a few to a few hundred transistors. The integration of the whole CPU onto a single VLSI chip therefore greatly reduced the cost of processing capacity. From their humble beginnings, continued increases in microprocessor capacity have rendered other forms of computers almost completely obsolete (see history of computing hardware), with one or more microprocessor as processing element in everything from the smallest embedded systems and handheld devices to the largest mainframes and supercomputers.

Since the early 1970s, the increase in capacity of microprocessors has been known to generally follow Moore's Law, which suggests that the complexity of an integrated circuit, with respect to minimum component cost, doubles every two years. In the late 1990s, and in the high performance microprocessor segment, heat generation (TDP), due to switching losses, static current leakage, and other factors, emerged as a leading developmental constraint.



Date invented	Late 1960s/Early 1970s (see article for explanation)
Connects to	Printed circuit boards via sockets, soldering, or other methods.
Architectures	PowerPC, x86, x86-64, and many others (see below, and article)
Common manufacturers	AMD, Analog Devices, Atmel, Cypress, Fairchild, Fujitsu, Hitachi, IBM, Infineon, Intel, Intersil, ITT, Maxim, Microchip, Mitsubishi, Mostek, Motorola, National, NEC, NXP (Philips), OKI, Renesas, Samsung, Sharp, Siemens, Signetics, STM, Synertek, Texas, Toshiba, TSMC, UMC, Winbond, Zilog, and others.

8085 Microprocessor

The Intel 8085 is an 8-bit microprocessor introduced by Intel in 1977. It was binary-compatible with the more-famous Intel 8080 but required less supporting hardware, thus allowing simpler and less expensive microcomputer systems to be built.

The "5" in the model number came from the fact that the 8085 required only a +5-volt (V) power supply rather than the +5V, -5V and +12V supplies the 8080 needed. Both processors were sometimes used in computers running the CP/M operating system, and the 8085 later saw use as a microcontroller (much by

virtue of its component count reducing feature). Both designs were eclipsed for desktop computers by the compatible but more capable Zilog Z80, which took over most of the CP/M computer market as well as taking a large share of the booming home computer market in the early-to-mid-1980s. The 8085 had a very long life as a controller. Once designed into such products as the DECtape controller and the VT100 video terminal in the late 1970s, it continued to serve for new production throughout the life span of those products (generally many times longer than the new manufacture lifespan of desktop computers).

Intel 8085

An Intel 8085AH processor.

Produced	From 1977 to 1990s
Common manufacturer(s)	Intel and several others
Max. CPU clock	3,5 and 6 MHz
Instruction set	pre x86
Package(s)	40 pin DIP

Buses

- Address bus - 16 line bus accessing 2^{16} memory locations (64 KB) of memory.
- Data bus - 8 line bus accessing one (8-bit) byte of data in one operation. Data bus width is the traditional measure of processor bit designations, as opposed to address bus width, resulting in the 8-bit microprocessor designation.
- Control buses - Carries the essential signals for various operations.

8086 Microprocessor

The **8086** is a 16-bit microprocessor chip designed by Intel and introduced on the market in 1978, which gave rise to the x86 architecture. Intel 8088, released in 1979, was essentially the same chip, but with an external 8-bit data bus (allowing the use of cheaper and fewer supporting logic chips), and is notable as the processor used in the original IBM PC.

Intel 8086

Produced	From 1978 to 1990s
Common manufacturer(s)	Intel, AMD, NEC, Fujitsu, Harris (Intersil), OKI, Siemens AG, Texas Instruments.
Max. CPU clock	5 MHz to 10 MHz
Instruction set	x86-16
Package(s)	40 pin DIP

The 8086 was originally intended as a temporary substitute for the ambitious iAPX 432 project in an attempt to draw attention from the less-delayed 16 and 32-bit processors of other manufacturers (such as Motorola, Zilog, and National Semiconductor) and at the same time to top the successful Z80 (designed by former Intel employees). Both the architecture and the physical chip were therefore developed quickly (in a little more than two years), using the same basic microarchitecture elements and physical implementation techniques as employed by the older 8085, and for which it also functioned as its continuation. Marketed as source compatible, it was designed so that assembly language for the 8085, 8080, or 8008 could be automatically converted into equivalent (sub-optimal) 8086 source code, with little or no hand-editing.

This was possible because the programming model and instruction set was (loosely) based on the 8085. However, the 8086 design was expanded to support full 16-bit processing, instead of the fairly basic 16-bit capabilities of the 8080/8085. New kinds of instructions were added as well; self-repeating operations and instructions to better support nested ALGOL-family languages such as Pascal, among others.

The 8086 was sequenced using a mix of random logic and microcode and was implemented using depletion load nMOS circuitry with approximately 20,000 active transistors (29,000 counting all ROM and PLA sites). It was soon moved to a new refined nMOS manufacturing process called HMOS (for High performance MOS) that Intel originally developed for manufacturing of fast static RAM products[8]. This was followed by HMOS-II, HMOS-III, and eventually a CMOS version. The original chip measured 33 mm² and minimum feature size was 3.2 μm.

The architecture was defined by *Stephen P. Morse* and *Bruce Ravenel*. *Peter A.Stoll* was lead engineer of the development team and *William Pohlman* the manager. While less known than the 8088 chip, the legacy of the 8086 is enduring; references to it can still be found on most modern computers in the form of the Vendor ID entry for all Intel devices, which is 8086_H (hexadecimal). It also lent its last two digits to Intel's later extended versions of the design, such as the 286 and the 386, all of which eventually became known as the x86 family.

Buses and operation

All internal registers as well as internal and external data buses were 16 bits wide, firmly establishing the "16-bit microprocessor" identity of the 8086. A 20-bit external address bus gave an 1 MB (segmented) physical address space (220 = 1,048,576). The data bus was multiplexed with the address bus in order to fit a standard 40-pin dual in-line package. 16-bit I/O addresses meant 64 KB of separate I/O space (216 = 65,536). The maximum linear address space were limited to 64 KB, simply because internal registers were only 16 bits wide. Programming over 64 KB boundaries involved adjusting segment registers (see below) and were therefore fairly awkward (and remained so until the 80386).
Some of the control pins, which carry essential signals for all external operations, had more than one function depending upon whether the device was operated in "min" or "max" mode. The former were intended for small single processor systems whilst the latter were for medium or large systems, using more than one processor.

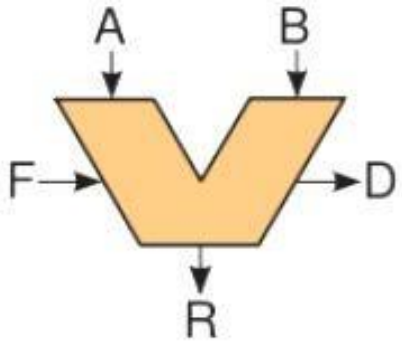
Architecture of a Microprocessor

Arithmetic logic unit

In computing, an arithmetic logic unit (ALU) is a digital circuit that performs arithmetic and logical operations. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simples microprocessors contain one for purposes such as maintaining timers. The processors found inside modern CPUs and graphics processing units (GPUs) have inside them very powerful and very complex ALUs; a single component may contain a number of ALUs.

Mathematician John von Neumann proposed the ALU concept in 1945, when he wrote a report on the foundations for a new computer called the EDVAC.

Arithmetic Logic Unit schematic symbol



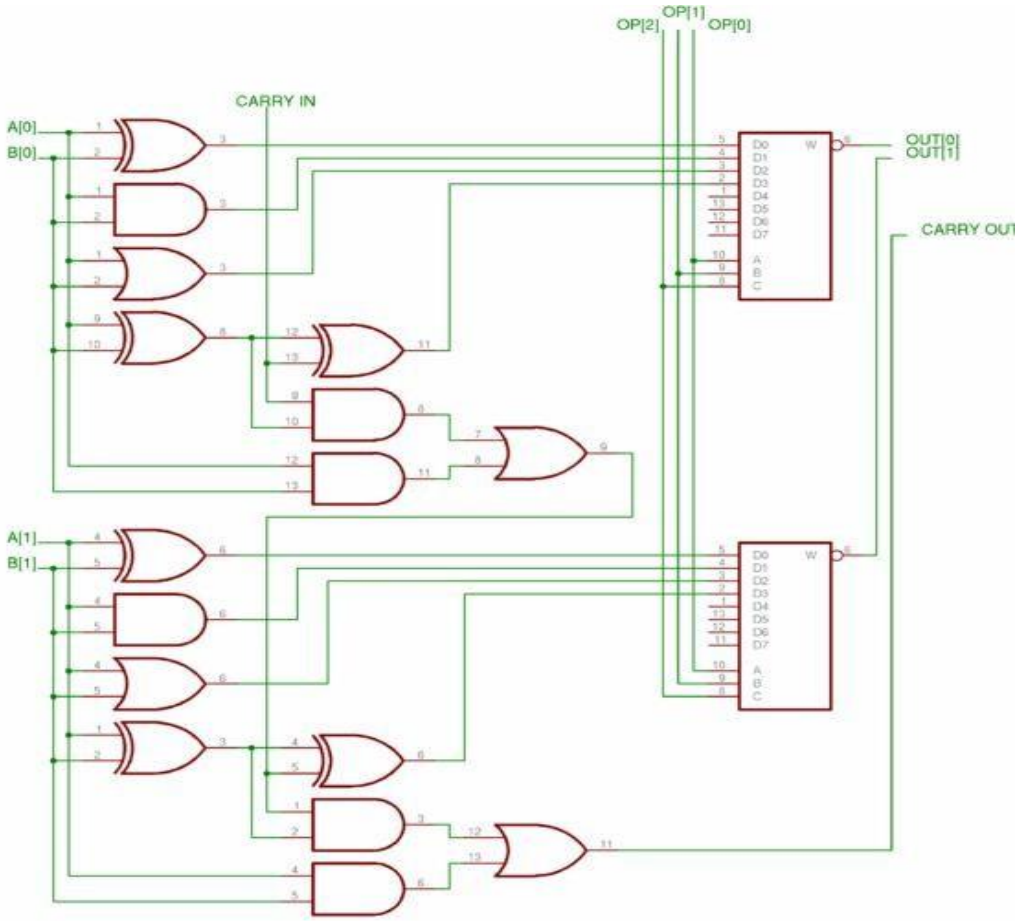
An ALU must process numbers using the same format as the rest of the digital circuit. The format of modern processors is almost always the two's complement binary number representation. Early computers used a wide variety of number systems, including one's complement, sign-magnitude format, and even true decimal systems, with ten tubes per digit.

ALUs for each one of these numeric systems had different designs, and that influenced the current preference for two's complement, as this is the representation that makes it easier for the ALUs to calculate additions and subtractions.

The two's-compliment number system allows for subtraction to be accomplished by adding the negative of a number in a very simple way which negates the need for specialised circuits to do subtraction.

The inputs to the ALU are the data to be operated on (called operands) and a code from the control unit indicating which operation to perform. Its output is the result of the computation.

In many designs the ALU also takes or generates as inputs or outputs a set of condition codes from or to a status register. These codes are used to indicate cases such as carry-in or carry-out, overflow, divide-by-zero, etc.



A simple example arithmetic logic unit (2-bit ALU) that does AND, OR, XOR, and addition.

Control unit

A **control unit** in general is a central (or sometimes distributed but clearly distinguishable) part of whatsoever machinery that controls its operation, provided that a piece of machinery is complex and organized enough to contain any such unit. One domain in which the term is specifically used is the area of computer design.

The functions performed by the control unit vary greatly by the internal architecture of the CPU, since the control unit really implements this architecture. On a regular processor that executes x86 instructions natively the control unit performs the tasks of fetching, decoding, managing execution and then storing results. On a x86 processor with a RISC core, the control unit has significantly more work to do. It manages the translation of x86 instructions to RISC micro-instructions, manages scheduling the micro-instructions between the various execution units, and juggles the output from these units to make sure they end up where they are supposed to go. On one of these processors the control unit may be broken into other units (such as a scheduling unit to handle scheduling and a retirement unit to deal with results coming from the pipeline) due to the complexity of the job it must perform.

Instruction set

An **instruction set** is a list of all the instructions, and all their variations, that a processor (or in the case of a virtual machine, an interpreter) can execute.

Instructions include:

- Arithmetic such as **add** and **subtract**
- Logic instructions such as **and**, **or**, and **not**
- Data instructions such as **move**, **input**, **output**, **load**, and **store**
- Control flow instructions such as **goto**, **if ... goto**, **call**, and **return**.

An **instruction set**, or **instruction set architecture** (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular CPU design.

Instruction set architecture is distinguished from the microarchitecture, which is the set of processor design techniques used to implement the instruction set. Computers with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

This concept can be extended to unique ISAs like TIMI (Technology-Independent Machine Interface) present in the IBM System/38 and IBM AS/400. TIMI is an ISA that is implemented as low-level software and functionally resembles what is now referred to as a virtual machine. It was designed to increase the longevity of the platform and applications written for it, allowing the entire platform to be moved to very different hardware without having to modify any software except that which comprises TIMI itself. This allowed IBM to move the AS/400 platform from an older CISC architecture to the newer POWER architecture without having to recompile any parts of the OS or software associated with it. Nowadays there are several open source Operating Systems which could be easily ported on any existing general purpose CPU, because the compilation is the essential part of their design (e.g. new software installation).

Machine language

Machine language is built up from discrete *statements* or *instructions*. on the processing architecture, a given instruction may specify:

- Particular registers for arithmetic, addressing, or control functions
- Particular memory locations or offsets
- Particular addressing modes used to interpret the operands

More complex operations are built up by combining these simple instructions, which (in a von Neumann machine) are executed sequentially, or as otherwise directed by control flow instructions.

Some operations available in most instruction sets include:

- moving
  - set a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
  - move data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
  - read and write data from hardware devices
- computing
  - add, subtract, multiply, or divide the values of two registers, placing the result in a register
  - perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register
  - compare two values in registers (for example, to see if one is less, or if they are equal)
- affecting program flow
  - jump to another location in the program and execute instructions there
  - jump to another location if a certain condition holds
  - jump to another location, but save the location of the next instruction as a point to return to (a *call*)

Some computers include "complex" instructions in their instruction set. A single "complex" instruction does something that may take many instructions on other computers. Such instructions are typified by instructions that take multiple steps, control multiple functional units, or otherwise appear on a larger scale than the bulk of simple instructions implemented by the given processor. Some examples of "complex" instructions include:

- saving many registers on the stack at once
- moving large blocks of memory
- complex and/or floating-point arithmetic (sine, cosine, square root, etc.)
- performing an atomic test-and-set instruction
- instructions that combine ALU with an operand from memory rather than a register

A complex instruction type that has become particularly popular recently is the SIMD or Single-Instruction Stream Multiple-Data Stream operation or vector instruction, an operation that performs the same arithmetic operation on multiple pieces of data at the same time. SIMD have the ability of manipulating large vectors and matrices in minimal time. SIMD instructions allow easy parallelization of algorithms commonly involved in sound, image, and video processing. Various SIMD implementations have been brought to market under trade names such as MMX, 3DNow! and AltiVec.

The design of instruction sets is a complex issue. There were two stages in history for the microprocessor. One using CISC or complex instruction set computer where many instructions were implemented. In the 1970s places like IBM did research and found that many instructions were used that could be eliminated. The result was the RISC, reduced instruction set computer, architecture which uses a smaller set of instructions. A simpler instruction set may offer the potential for higher speeds, reduced processor size, and reduced power consumption; a more complex one may optimize common operations, improve memory/cache efficiency, or simplify programming.