BINARY NUMBER SYSTEM

Introduction

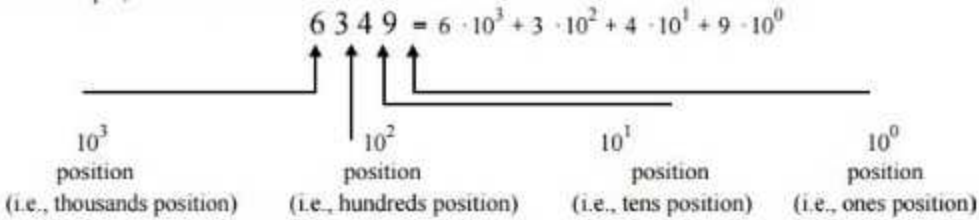
The number system that you are familiar with, that you use every day, is the decimal number system, also commonly referred to as the base-10 system. When you perform computations such as 3 + 2 = 5, or 21 – 7 = 14, you are using the decimal number system. This system, which you likely learned in first or second grade, is ingrained into your subconscious; it's the natural way that you think about numbers. Evidence exists that Egyptians were using a decimal number system five thousand years ago. The Roman numeral system, predominant for hundreds of years, was also a decimal number system (though organized differently from the Arabic base-10 number system that we are most familiar with). Indeed, base-10 systems, in one form or another, have been the most widely used number systems ever since civilization started counting.

In dealing with the inner workings of a computer, though, you are going to have to learn to think in a different number system, the binary number system, also referred to as the base-2 system.

Consider a child counting a pile of pennies. He would begin: "One, two, three, …, eight, nine." Upon reaching nine, the next penny counted makes the total one single group of ten pennies. He then keeps counting: "One group of ten pennies… two groups of ten pennies… three groups of ten pennies … eight groups of ten pennies … nine groups of ten pennies…" Upon reaching nine groups of ten pennies plus nine additional pennies, the next penny counted makes the total thus far: one single group of one hundred pennies. Upon completing the task, the child might find that he has three groups of one hundred pennies, five groups of ten pennies, and two pennies left over: 352 pennies.

More formally, the base-10 system is a positional system, where the rightmost digit is the ones position (the number of ones), the next digit to the left is the tens position (the number of groups of 10), the next digit to the left is the hundreds position (the number of groups of 100), and so forth. The base-10 number system has 10 distinct symbols, or digits (0, 1, 2, 3,…8, 9). In decimal notation, we write a number as a string of symbols, where each symbol is one of these ten digits, and to interpret a decimal number, we multiply each digit by the power of 10 associated with that digit's position.

For example, consider the decimal number: 6349. This number is:



Consider: Computers are built from transistors, and an individual transistor can only be ON or OFF (two options). Similarly, data storage devices can be optical or magnetic. Optical storage devices store data in a specific location by controlling whether light is reflected off that location or is not reflected off that location (two options). Likewise, magnetic storage devices store data in a specific location by magnetizing the particles in that location with a specific orientation. We can have the north magnetic pole pointing in one direction, or the opposite direction (two options).

Computers can most readily use two symbols, and therefore a base-2 system, or binary number system, is most appropriate. The base-10 number system has 10 distinct symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The base-2 system has exactly two symbols: 0 and 1. The base-10 symbols are termed digits. The base-2 symbols are termed binary digits, or bits for short. All base-10 numbers are built as strings of digits (such as 6349). All binary numbers are built as strings of bits (such as 1101). Just as we would say that the decimal number 12890 has five digits, we would say that the binary number 11001 is a five-bit number.

2 The Binary Number System

> Consider again the example of a child counting a pile of pennies, but this time in binary.

He would begin with the first penny: "1." The next penny counted makes the total one single group of two pennies. What number is this?

When the base-10 child reached nine (the highest symbol in his scheme), the next penny gave him "one group of ten", denoted as 10, where the "1" indicated one collection of ten.

Similarly, when the base-2 child reaches one (the highest symbol in his scheme), the next penny gives him "one group of two", denoted as 10, where the "1" indicates one collection of two.

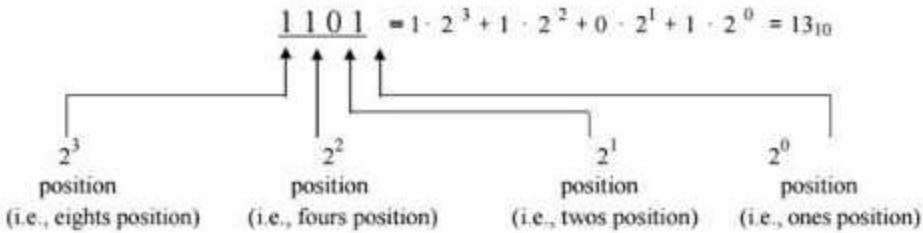
Back to the base-2 child: The next penny makes one group of two pennies and one additional penny: "11." The next penny added makes two groups of two, which is one group of 4: "100." The "1" here indicates a collection of two groups of two, just as the "1" in the base-10 number 100 indicates ten groups of ten.

Upon completing the counting task, base -2 child might find that he has one group of four pennies, no groups of two pennies, and one penny left over: 101 pennies. The child counting the same pile of pennies in base-10 would conclude that there were 5 pennies. So, 5 in base-10 is equivalent to `101` in base-2. To avoid confusion when the base in use if not clear from the context, or when using multiple bases in a single expression, we append a subscript to the number to indicate the base, and write:

$$5_{10} = 101_2$$

Just as with decimal notation, we write a binary number as a string of symbols, but now each symbol is a `0` or a `1`. To interpret a binary number, we multiply each digit by the power of 2 associated with that digit's position.

For example, consider the binary number `1101`. This number is:

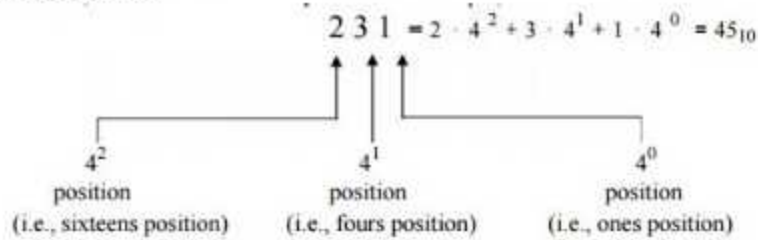


Since binary numbers can only contain the two symbols 0 and 1, numbers such as 25 and 1114000 cannot be binary numbers.

We say that all data in a computer is stored in binary—that is, as 1's and 0's. It is important to keep in mind that values of 0 and 1 are logical values, not the values of a physical quantity, such as a voltage. The actual physical binary values used to store data internally within a computer might be, for instance, 5 volts and 0 volts, or perhaps 3.3 volts and 0.3 volts or perhaps reflection and no reflection. The two values that are used to physically store data can differ within different portions of the same computer. All that really matters is that there are two different symbols, so we will always refer to them as 0 and 1.

A string of eight bits (such as 11000110) is termed a byte. A collection of four bits (such as 1011) is smaller than a byte, and is hence termed a nibble. (This is the sort of nerd-humor for which engineers are famous.)

The idea of describing numbers using a positional system, as we have illustrated for base-10 and base-2, can be extended to any base. For example, the base-4 number 231 is:



3 Converting Between Binary Numbers and Decimal Numbers

We humans about numbers using the decimal number system, whereas computers use the binary number system. We need to be able to readily shift between the binary and decimal number representations.
Converting a Binary Number to a Decimal Number

To convert a binary number to a decimal number, we simply write the binary number as a sum of powers of 2. For example, to convert the binary number `1011` to a decimal number, we note that the rightmost position is the ones position and the bit value in this position is a `1`. So, this rightmost bit has the decimal value of $1 \cdot 2^0$. The next position to the left is the twos position, and the bit value in this position is also a `1`. So, this next bit has the decimal value of $1 \cdot 2^1$. The next position to the left is the fours position, and the bit value in this position is a 0. The leftmost position is the eights position, and the bit value in this position is a `1`. So, this leftmost bit has the decimal value of $1 \cdot 2^3$. Thus:

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 2 + 1 = 11_{10}$$

1. The binary number 110110 as a decimal number. Solution:

For example, to convert the binary number 10101 to decimal, we annotate the position values below the bit values:

1	0	1	0	1
16	8	4	2	1

Then we add the position values for those positions that have a bit value of 1: 16 + 4 + 1 = 21. Thus

$$10101_2 = 21_{10}$$

You should "memorize" the binary representations of the decimal digits 0 through 15 shown below.

Decimal Number	Binary Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal Number	Binary Number
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

You may be wondering about the leading zeros in the table above. For example, the decimal number 5 is represented in the table as the binary number 0101. We could have represented the binary equivalent of 5 as 101, 00101, 0000000101, or with any other number of leading zeros. All answers are correct.

Sometimes, though, you will be given the size of a storage location. When you are given the size of the storage location, include the leading zeros to show all bits in the storage location. For example, if told to represent decimal 5 as an 8-bit binary number, your answer should be 00000101.

Converting a Decimal Number to a Binary Number: Method 2

The second method of converting a decimal number to a binary number entails repeatedly dividing the decimal number by 2, keeping track of the remainder at each step. To convert the decimal number x to binary:

Step 1. Divide x by 2 to obtain a quotient and remainder. The remainder will be 0 or 1.

Step 2. If the quotient is zero, you are finished: Proceed to Step 3. Otherwise, go back to Step 1, assigning x to be the value of the most-recent quotient from Step 1.

Step 3. The sequence of remainders forms the binary representation of the number.

4 Hexadecimal Numbers

In addition to binary, another number base that is commonly used in digital systems is base 16. This number system is called hexadecimal, and each digit position represents a power of 16. For any number base greater than ten, a problem occurs because there are more than ten symbols needed to represent the numerals for that number base. It is customary in these cases to use the ten decimal numerals followed by the letters of the alphabet beginning with A to provide the needed numerals. Since the hexadecimal system is base 16, there are sixteen numerals required. The following are the hexadecimal numerals:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The following are some examples of hexadecimal numbers:

10_{16} 47_{16} $3FA_{16}$ $A03F_{16}$

The reason for the common use of hexadecimal numbers is the relationship between the numbers 2 and 16. Sixteen is a power of 2 ($16 = 2^4$). Because of this relationship, four digits in a binary number can be represented with a single hexadecimal digit. This makes conversion between binary and hexadecimal numbers very easy, and hexadecimal can be used to write large binary numbers with much fewer digits. When working with large digital systems, such as computers, it is common to find binary numbers with 8, 16 and even 32 digits. Writing a 16 or 32 bit binary number would be quite tedious and error prone. By using hexadecimal, the numbers can be written with fewer digits and much less likelihood of error.

To convert a binary number to hexadecimal, divide it into groups of four digits starting with the rightmost digit. If the number of digits isn't a multiple of 4, prefix the number with 0's so that each group contains 4 digits. For each four digit group, convert the 4 bit binary number into an equivalent hexadecimal digit. (See the Binary, BCD, and Hexadecimal Number Tables at the end of this document for the correspondence between 4 bit binary patterns and hexadecimal digits)

2. Convert the binary number 10110101 to a hexadecimal number

Divide into groups for 4 digits	1011 0101
Convert each group to hex digit	B 5
	$B5_{16}$

3. Convert the binary number 0110101110001100 to hexadecimal

Divide into groups of 4 digits	0110 1011 1000 1100
Convert each group to hex digit	6 B 8 C
	$6B8C_{16}$

To convert a hexadecimal number to a binary number, convert each hexadecimal digit into a group of 4 binary digits.

4. Convert the hex number 374F into binary

	3 7 4 F
Convert the hex digits to binary	0011 0111 0100 1111
	0011011101001111_2

There are several ways in common use to specify that a given number is in hexadecimal representation rather than some other radix. In cases where the context makes it absolutely clear that numbers are represented in hexadecimal, no indicator is used. In much written material where the context doesn't make it clear what the radix is, the numeric subscript 16 following the hexadecimal number is used. In most programming languages, this method isn't really feasible, so there are several conventions used depending on the language. In the C and C++ languages, hexadecimal constants are represented with a '0x' preceding the number, as in: 0x317F, or 0x1234, or 0xAF. In assembler programming languages that follow the Intel style, a hexadecimal constant begins with a numeric character (so that the assembler can distinguish it from a variable name), a leading '0' being used if necessary. The letter 'h' is then suffixed onto the number to inform the assembler that it is a hexadecimal constant. In Intel style assembler format: 371Fh and

0FABCh are valid hexadecimal constants. Note that: A37h isn't a valid hexadecimal constant. It doesn't begin with a numeric character, and so will be taken by the assembler as a variable name. In assembler programming languages that follow the Motorola style, hexadecimal constants begin with a '$' character. So in this case: $371F or $FABC or $01 are valid hexadecimal constants.

5 Binary Coded Decimal Numbers

Another number system that is encountered occasionally is Binary Coded Decimal. In this system, numbers are represented in a decimal form, however each decimal digit is encoded using a four bit binary number.

The decimal number 136 would be represented in BCD as follows: 136 = 0001 0011 0110

1 3 6

Conversion of numbers between decimal and BCD is quite simple. To convert from decimal to BCD, simply write down the four bit binary pattern for each decimal digit. To convert from BCD to decimal, divide the number into groups of 4 bits and write down the corresponding decimal digit for each 4 bit group.

There are a couple of variations on the BCD representation, namely packed and unpacked. An unpacked BCD number has only a single decimal digit stored in each data byte. In this case, the decimal digit will be in the low four bits and the upper 4 bits of the byte will be 0. In the packed BCD representation, two decimal digits are placed in each byte. Generally, the high order bits of the data byte contain the more significant decimal digit.

6. The following is a 16 bit number encoded in packed BCD format:

   01010110 10010011

   This is converted to a decimal number as follows: 0101 0110 1001 0011

   5 6 9 3 The value is 5693 decimal

7. The same number in unpacked BCD (requires 32 bits)

   00000101 00000110 00001001 00000011

   5 6 9 3

The use of BCD to represent numbers isn't as common as binary in most computer systems, as it is not as space efficient. In packed BCD, only 10 of the 16 possible bit patterns in each 4 bit unit are used. In unpacked BCD, only 10 of the 256 possible bit patterns in each byte are used. A 16 bit quantity can represent the range 0-65535 in binary, 0-9999 in packed BCD and only 0-99 in unpacked BCD.

Fixed Precision and Overflow

we haven't considered the maximum size of the number. We have assumed that as many bits are available as needed to represent the number. In most computer systems, this isn't the case. Numbers in computers are typically represented using a fixed number of bits. These sizes are typically 8 bits, 16 bits, 32 bits, 64 bits and 80 bits. These sizes are generally a multiple of 8, as most computer memories are organized on an 8 bit byte basis. Numbers in which a specific number of bits are used to represent the value are called fixed precision numbers. When a specific number of bits are used to represent a number, that determines the range of possible values that can be represented. For example, there are 256 possible combinations of 8 bits, therefore an 8 bit number can represent 256 distinct numeric values and the range is typically considered to be 0-255. Any number larger than 255 can't be represented using 8 bits. Similarly,

16 bits allows a range of 0-65535.

When fixed precision numbers are used, (as they are in virtually all computer calculations) the concept of overflow must be considered. An overflow occurs when the result of a calculation can't be represented with the number of bits available. For example when adding the two eight bit quantities: 150 + 170, the result is 320. This is outside the range 0-255, and so the result can't be represented using 8 bits. The result has overflowed the available range. When overflow occurs, the low order bits of the result will remain valid, but the high order bits will be lost. This results in a value that is significantly smaller than the correct result.

When doing fixed precision arithmetic (which all computer arithmetic involves) it is necessary to be conscious of the possibility of overflow in the calculations.

Signed and Unsigned Numbers.

we have only considered positive values for binary numbers. When a fixed precision binary number is used to hold only positive values, it is said to be unsigned. In this case, the range of positive values that can be represented is 0 -- 2^n-1, where n is the number of bits used. It is also possible to represent signed (negative as well as positive) numbers in binary. In this case, part of the total range of values is used to represent positive values, and the rest of the range is used to represent negative values.

There are several ways that signed numbers can be represented in binary, but the most common representation used today is called two's complement. The term two's complement is somewhat ambiguous, in that it is used in two different

ways. First, as a representation, two's complement is a way of interpreting and assigning meaning to a bit pattern contained in a fixed precision binary quantity. Second, the term two's complement is also used to refer to an operation that can be performed on the bits of a binary quantity. As an operation, the two's complement of a number is formed by inverting all of the bits and adding 1. In a binary number being interpreted using the two's complement representation, the high order bit of the number indicates the sign. If the sign bit is 0, the number is positive, and if the sign bit is 1, the number is negative. For positive numbers, the rest of the bits hold the true magnitude of the number. For negative numbers, the lower order bits hold the complement (or bitwise inverse) of the magnitude of the number. It is important to note that two's complement representation can only be applied to fixed precision quantities, that is, quantities where there are a set number of bits.

Two's complement representation is used because it reduces the complexity of the hardware in the arithmetic-logic unit of a computer's CPU. Using a two's complement representation, all of the arithmetic operations can be performed by the same hardware whether the numbers are considered to be unsigned or signed. The bit operations performed are identical, the difference comes from the interpretation of the bits. The interpretation of the value will be different depending on whether the value is considered to be unsigned or signed.

8. Find the 2's complement of the following 8 bit number

00101001

```
 11010110      First, invert the bits
+ 00000001     Then, add 1
= 11010111
```

The 2's complement of 00101001 is 11010111

9. Find the 2's complement of the following 8 bit number 10110101

```
 01001010      Invert the bits
+ 00000001     then add 1
= 01001011
```

The 2's complement of 10110101 is 01001011

The counting sequence for an eight bit binary value using 2's complement representation appears as follows:

			largest magnitude positive
01111111	7Fh	127	number
01111110	7Eh	126	
01111101	7Dh	125	
...			
00000011	03h		
00000010	02h		
00000001	01h		
00000000	00h		
11111111	0FFh	-1	
11111110	0FEh	-2	
11111101	0FDh	-3	
...			
10000010	82h	-126	
10000001	81h	-127	
10000000	80h	-128	largest magnitude negative number

Counting up from 0, when 127 is reached, the next binary pattern in the sequence corresponds to -128. The values jump from the greatest positive number to the greatest negative number, but that the sequence is as expected after that. (i.e. adding 1 to –128 yields –127, and so on.). When the count has progressed to 0FFh (or the largest unsigned magnitude possible) the count wraps around to 0. (i.e. adding 1 to –1 yields 0).

ASCII Character Encoding

The name ASCII is an acronym for: American Standard Code for Information Interchange. It is a character encoding standard developed several decades ago to provide a standard way for digital machines to encode characters. The ASCII code provides a mechanism for encoding alphabetic characters, numeric digits, and punctuation marks for use in representing text and numbers written using the Roman alphabet. As originally designed, it was a seven bit code. The seven bits allow the representation of 128 unique characters. All of the alphabet, numeric digits and standard English punctuation marks are encoded. The ASCII standard was later extended to an eight bit code (which allows 256 unique code patterns) and various additional symbols were added, including characters with diacritical marks (such as accents) used in European languages, which don't appear in English. There are also numerous non-standard extensions to ASCII giving different encoding for the upper 128 character codes than the standard. For example, The character set encoded into the display card for the original IBM PC had a non-standard encoding for the upper character set. This is a non-standard extension that is in very wide spread use, and could be considered a standard in itself.

Some important things to points about ASCII code:

The numeric digits, 0-9, are encoded in sequence starting at 30h The upper case alphabetic characters are sequential beginning at 41h The lower case alphabetic characters are sequential beginning at 61h

The first 32 characters (codes 0-1Fh) and 7Fh are control characters. They do not have a standard symbol (glyph) associated with them. They are used for carriage control, and protocol purposes. They include 0Dh (CR or carriage return), 0Ah (LF or line feed), 0Ch (FF or form feed), 08h (BS or backspace).

Most keyboards generate the control characters by holding down a control key (CTRL) and simultaneously pressing an alphabetic character key. The control code will have the same value as the lower five bits of the alphabetic key pressed. So, for example, the control character 0Dh is carriage return. It can be generated by pressing CTRL-M. To get the full 32 control characters a few at the upper end of the range are generated by pressing CTRL and a punctuation key in combination. For example, the ESC (escape) character is generated by pressing CTRL-[(left square bracket).

Conversions Between Upper and Lower Case ASCII Letters.

ASCII code chart that the uppercase letters start at 41h and that the lower case letters begin at 61h. In each case, the rest of the letters are consecutive and in alphabetic order. The difference between 41h and 61h is 20h. Therefore the conversion between upper and lower case involves either adding or subtracting 20h to the character code. To convert a lower case letter to upper case, subtract 20h, and conversely to convert upper case to lower case, add 20h. It is important to note that you need to first ensure that you do in fact have an alphabetic character before performing the addition or subtraction. Ordinarily, a check should be made that the character is in the range 41h–5Ah for upper case or 61h-7Ah for lower case.

Conversion Between ASCII and BCD

ASCII code chart that the numeric characters are in the range 30h-39h. Conversion between an ASCII encoded digit and an unpacked BCD digit can be

accomplished by adding or subtracting 30h. Subtract 30h from an ASCII digit to get BCD, or add 30h to a BCD digit to get ASCII. Again, as with upper and lower case conversion for alphabetic characters, it is necessary to ensure that the character is in fact a numeric digit before performing the subtraction. The digit characters are in the range 30h-39h.

LOGIC GATES

All digital systems are made from a few basic digital circuits that we call logic gates. These circuits perform the basic logic functions that we will describe in this session. The physical realization of these logic gates has changed over the years from mechanical relays to electronic vacuum tubes to transistors to integrated circuits containing thousands of transistors.

In this appendix you will learn:
Definitions of the basic gates in terms of truth tables and logic equations
DeMorgan's Theorem

How gates defined in terms of positive and negative logic are related To use multiple-input gates

How to perform a sum of products and a product of sums design from a truth table specification

1 The Three Basic Logic Gates

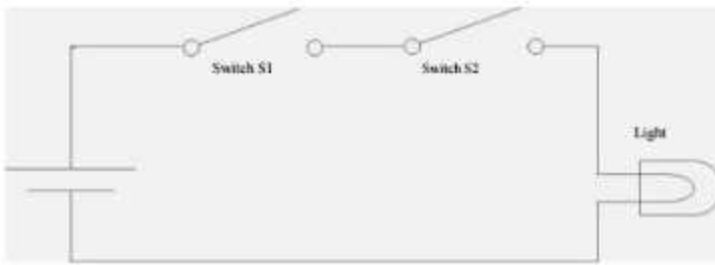
Much of a computer's hardware is comprised of digital logic circuits. Digital logic circuits are built from just a handful of primitive elements, called logic gates, combined in various ways.

In a digital logic circuit, only two values may be present. The values may be −5 and + 5 volts. Or the values may be 0.5 and 3.5 volts. Or the values may be… you get the picture. To allow consideration of all of these possibilities, we will say that digital logic circuits allow the presence of two logical values: 0 and 1.

So, signals in a digital logic circuit take on the values of 0 or 1. Logic gates are devices which compute functions of these binary signals.

The AND Gate

Consider the circuit below which consists of a battery, a light, and two switches in series:



When will the light turn on? It should be clear that the light will turn on only if both switch S1 and switch S2 are shut.

It is quite likely that you encounter the and operation in some shape or form hundreds of times each day. Consider the simple action of withdrawing funds from your checking account at an ATM. You will only be able to complete the transaction if you have a checking account and you have money in it. The ATM will only permit the transaction if you have your ATM card and you enter your correct 4-digit PIN. To enter the correct PIN, you have to enter the first digit correctly and enter the second digit correctly and enter the third digit correctly and enter the fourth digit correctly.

Returning to the circuit above, we can represent the light's operation using a table:

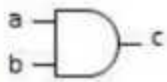
S1	S2	Light
open	open	off
open	closed	off
closed	open	off
closed	closed	on

The switch is a binary device: it can be open or closed. Let's represent these two states as 0 and 1. Likewise, the light is a binary device with two states: off and on, which we will represent as 0 and 1. Rewriting the table above with this notation, we have:

S1	S2	Light
0	0	0
0	1	0
1	0	0
1	1	1

This table, which displays the output for all possible combinations of the input, is termed the truth table for the AND operation. In a computer, this and functionality is implemented with a circuit called an AND gate. The simplest

AND gate has two inputs and one output and is represented pictorially by the symbol:



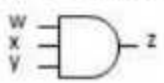
where the inputs have been labeled *a* and *b*, and the output has been labeled *c*. If both inputs are 1 then the output is 1. Otherwise, the output is 0.

We represent the and operation by using either the multiplication symbol (i.e., " · ") or by writing the inputs together. Thus, for the AND gate shown above, we would write the output *c* as c = a b or as c = ab. This would be pronounced: "*c* = *a* and *b*."

The truth table for the AND gate is shown below. The output *c* = ab is equal to 1 if and only if (iff) a is 1 and b is 1. Otherwise, the output is 0.

a	*b*	*c*
0	0	0
0	1	0
1	0	0
1	1	1

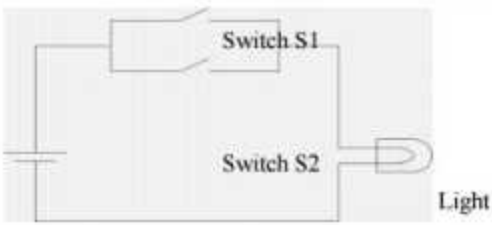
AND gates can have more than one input (however, an AND gate always has just a single output). Let's consider a three-input AND gate:



w	*x*	*y*	*z*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

The OR Gate

Now consider the circuit shown below, that has 2 switches in parallel.



It is evident that the light will turn on when either switch S_1 is shut or switch S_2 is shut or both are shut.

It is quite likely that you encounter the ***or*** operation in some shape or form hundreds of times each day. Consider the simple action of sitting on your couch at home at two in the morning studying for your Digital Logic class. Your phone will ring if you get a call from Alice or from Bob. Your home's security alarm will go off if the front door opens or the back door opens. You will drink a cup of coffee if you are drowsy ***or*** you are thirsty.

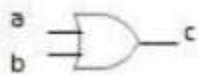
We can represent the light's operation using a table

S1	S2	Light
open	open	off
open	closed	on
closed	open	on
closed	closed	on

Changing the words *open* and *off* to 0 and the words *shut* and *on* to 1 and the table becomes:

S1	S2	Light
0	0	0
0	1	1
1	0	1
1	1	1

This is the truth table for the OR operation. This ***or*** functionality is implemented with a circuit called an OR gate. The simplest OR gate has two inputs and one output and is represented pictorially by the symbol:



If either or both inputs are 1, the output is 1. Otherwise, the output is 0.

We represent the or operation by using the addition symbol. Thus, for the OR gate above, we would write the output c as c = a + b. This would be pronounced: "*c* = *a* or *b*."

The truth table for the OR gate is shown below. The output is 1 if *a* is 1 ***or*** *b* is 1; otherwise, the output is 0.

a	*b*	*c*
0	0	0
0	1	1
1	0	1
1	1	1

The NOT Gate

The last of our basic logic gates is the NOT gate. The NOT gate always has one input and one output. If the input is 1, the output is 0. If the input is 0, the output is 1. This operation— chaging the value of the binary input—is called complementation, negation or inversion. The mathematical symbol for negation is an apostrophe. If the input to a NOT gate is P, the output, termed the complement, is denoted as P'.

The pictorial symbol for a NOT gate is intended to depict an amplifier followed by a bubble, shown below. Sometimes the NOT operation is represented by just the bubble, without the amplifier.

The truth table for the NOT gate is shown below:

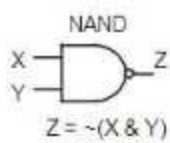
P	P'
0	1
1	0

Three New Gates

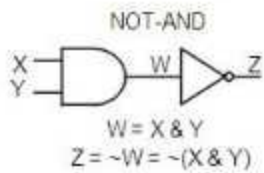
Three new gates, NAND, NOR, and Exclusive-OR, can be formed from our three basic gates: NOT, AND, and OR.

NAND Gate

The logic symbol for a NAND gate is like an AND gate with a small circle (or bubble) on the output.we see that the output of a NAND gate is 0 (low) only if both inputs are 1 (high) . The NAND gate is equivalent to an AND gate followed by an inverter (NOT-AND).



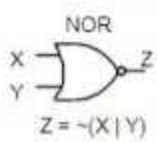
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0



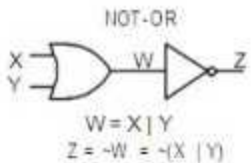
X	Y	W	Z
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

NOR Gate

The logic symbol for a NOR gate is like an OR gate with a small circle (or bubble) on the output. From the truth table .we see that the output of a NOR gate is 1 (high) only if both inputs are 0 (low). The NOR gate is equivalent to an OR gate followed by an inverter (NOT-OR), as shown by the two truth tables.



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

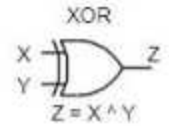


X	Y	W	Z
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

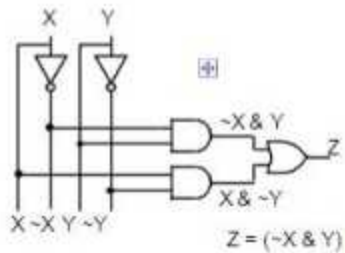
Exclusive-OR Gate.

The XOR gate logic symbol is like an OR gate symbol with an extra curved vertical line on the input. From the truth table .we see that the output Z of an XOR gate is 1 (true or high) if either input, X or Y, is 1 (true or high), but not both. The output Z will be zero if both X and Y are the same (either both 1 or both 0).

The equation for the XOR gate is given as Z = X ^ Y. In this book we will use the symbol ^ as the XOR operator. Sometimes the symbol or the dollar sign $ is used to denote Exclusive-OR. We will use the symbol ^ because that is the symbol recognized by the Verilog software used to program a CPLD.



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0



X	Y	~X	~Y	X & Y	X & ~Y	Z
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

Z = (~X & Y) | (X & ~Y)

BOOLEAN ALGEBRA

Boolean algebra is an algebraic structure defined by a set of elements, B, together with two binary operators, + and., provider that the following postulates are satisfied.

T1: Commutative Law

(a)A+B = B+A

(b) A B = BA

T2: Associative Law

(a) (A+B) +C = A+ (B+C)

(b) (A B) C = A (B C)

T3: Distributive Law

(a) A (B +C) = A B + AC

(b)A + (B C) = (A +B) (A+C)

T4: Identity Law

(a) A+A =A

(b)A A =A

T5: Negative Law

(a) (A') =A'

(b)(A'') = A

T6: Redundant Law

(a) A+AB=A

(b) A (A +B) =A

T7: Null Law

(a)0 + A = A

(b) 1 A = A

(c) 1 + A = 1

(d) 0 A = 0

T8: Double Negation Law

(a) A' +A=1

(b) A' A=0

T9: Absorption Law

(a) A+A'B =A+B

(b) A (A' + B) =AB

T10: De Morgan's Theorem

(a) (A+B)' = A' B'

(b)(AB)' = A'+B'

Example 1:

Using theorems,

A + A' B = A 1 + A' B

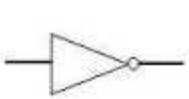
= A (1 + B) + A'B

=A + AB + A'B

=A + B (A + A')

= A + B

Using Truth Table



Using Truth Table

A	B	A+B	A'B	A+A'B
0	0	0	0	0
0	1	1	1	1
1	0	1	0	1
1	1	1	0	1

1 Verification Of De Morgan's Theorems:

- De Morgan's First Theorem states:

The complement of a product of variables is equal to the sum of the complements of the individual variables

- De Morgan's Second Theorem states:

The complement of sum of variables is equal to the product of the complements of the dividable variables

- **FIRST LAW**

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

Figure: De Morgan's First Law

- **SECOND LAW**

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

Figure: De Morgan's Second Law

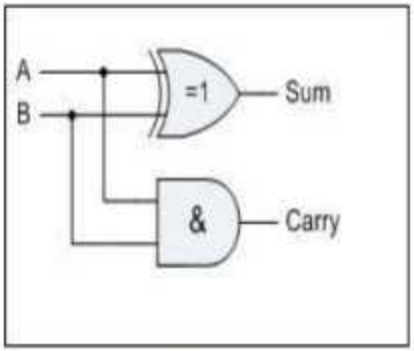
ADDER

1 Half Adder

Half adder is a circuit that will add two bits & produce a sum & a carry bit. It needs two input bits & two output bits.Fig.4.1 shows the block diagram of a half adder.

Figure: Block diagram of a Half Adder

Ex-OR gate will only produce an output "1" when "EITHER" input is at logic "1", so we need an additional output to produce a carry output, "1" when "BOTH" inputs "A" and "B" are at logic "1" and a standard AND Gate fits the bill nicely. By combining the Ex-OR gate with the AND gate results in a simple digital binary adder circuit known commonly as the "Half Adder" circuit.



INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure: Logic diagram & Truth table for half adder

2 Full Adder

A half adder has only two inputs &there is no provision to add a carry coming from the lower order bits when multi addition is performed. For this purpose, a full adder is designed.

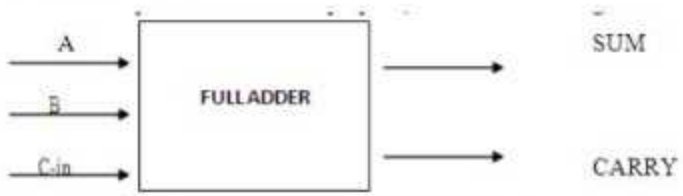


Figure: Block diagram of a Half Adder

The 1-bit Full Adder circuit is basically two half adders connected together and consists of three Ex-OR gates, two AND gates and an OR gate, six logic gates in total. The truth table for the full adder includes an additional column to take into account the Carry-in input as well as the summed output and carry-output.

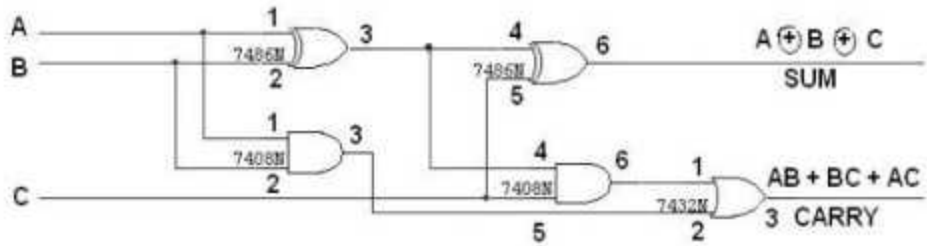


Figure: Logic diagram of a Full adder using two Half Adders

Table: Truth Table for Full Adder

Table: Truth Table for Full Adder

INPUTS			OUTPUTS	
A	B	C	CARRY	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

FLIP FLOP

1 RS Flip Flop

RS Flip Flop have two inputs, S and R. S is called set and R is called reset. The S input is used to produce HIGH on Q (i.e. store binary 1 in flip-flop). The R input is used to produce LOW on Q (i.e. store binary 0 in flip-flop). Q' is Q complementary output, so it always holds the opposite value of Q. The output of the S-R Flip Flop depends on current as well as previous inputs or state, and its state (value stored) can change as soon as its inputs change. The circuit and the truth table of RS Flip Flop is shown below.

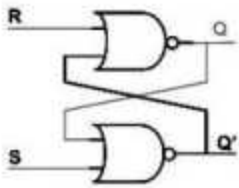


Figure : RS Flip Flop

Table: Truth table for RS Flip Flop

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

The operation has to be analyzed with the 4 inputs combinations together with the 2 possible previous states.

When S = 0 and R = 0: If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So it is clear that when both S and R inputs are LOW, the output is retained as before the application of inputs. (i.e. there is no state change).
When S = 1 and R = 0: If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. So in simple words when S is HIGH and R is LOW, output Q is HIGH.

When S = 0 and R = 1: If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So in simple words when S is LOW and R is HIGH, output Q is LOW.
When S = 1 and R =1 : No matter what state Q and Q' are in, application of 1 at input of NOR gate always results in 0 at output of NOR gate, which results in both Q and Q' set to LOW (i.e. Q = Q'). LOW in both the outputs basically is wrong, so this case is invalid.

It is possible to construct the RS Flip Flop using NAND gates (of course as seen in Logic gates section). The only difference is that NAND is NOR gate dual form (Did I say that in Logic gates section?). So in this case the R = 0 and S = 0 case becomes the invalid case. The circuit and Truth table of RS Flip Flop using NAND is shown below.

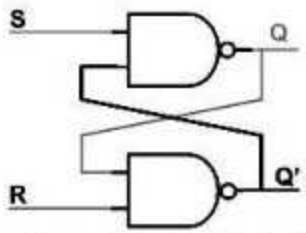


Figure : S-R using NAND Gates

Table Truth table for SR Flip Flop

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

If you look closely, there is no control signal, so this kind of Flip Flopes or flip-flops are called asynchronous logic elements. Since all the sequential circuits are built around the RS Flip Flop, we will concentrate on synchronous circuits and not on asynchronous circuits.

2 RS Flip Flop with Clock

We have seen this circuit earlier with two possible input configurations: one with level sensitive input and one with edge sensitive input. The circuit below shows the level sensitive RS Flip Flop. Control signal "Enable" E is used to gate the input S and R to the RS Flip Flop. When Enable E is HIGH, both the AND gates act as buffers and thus R and S appears at the RS Flip Flop input and it functions like a normal RS Flip Flop. When Enable E is LOW, it drives LOW to both inputs of RS Flip Flop. As we saw in previous page, when both inputs of a NOR Flip Flop are low, values are retained (i.e. the output does not change).

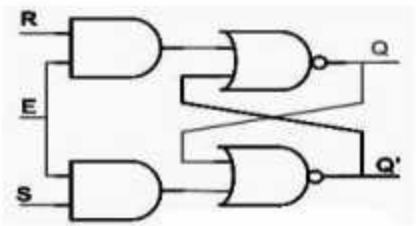


Figure : S-R with Edge Sensitive and Level sensitive

Set up and Hold time

For synchronous flip-flops, we have special requirements for the inputs with respect to clock signal input. They are

Setup Time: Minimum time period during which data must be stable before the clock makes a valid transition. For example, for a posedge triggered flip-flop, with a setup time of 2 ns, Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 2 ns before clock makes transition from 0 to 1.

Hold Time: Minimum time period during which data must be stable after the clock has made a valid transition. For example, for a posed triggered flip-flop, with a hold time of 1 ns. Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 1 ns after clock has made transition from 0 to 1.

If data makes transition within this setup window and before the hold window, then the flip-flop output is not predictable, and flip-flop enters what is known as meta stable state. In this state flip-flop output oscillates between 0 and 1. It takes some time for the flip-flop to settle down. The whole process is called Meta stability. You could refer to tidbits section to know more information on this topic. The waveform below shows input S (R is not shown), and CLK and output Q (Q' is not shown) for a SR posed flip-flop.

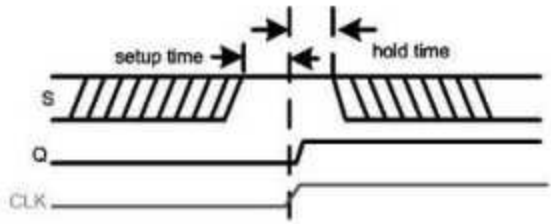


Figure: Waveform for S-R and CLK

Figure: Waveform for S-R and CLK

3 D Flip Flop

The RS Flip Flop seen earlier contains ambiguous state; to eliminate this condition we can ensure that S and R are never equal. This is done by connecting S and R together with an inverter. Thus we have D Flip Flop: the same as the RS Flip Flop, with the only difference that there is only one input, instead of two (R and S). This input is called D or Data input. D Flip Flop is called D transparent Flip Flop for the reasons explained earlier. Delay flip-flop or delay latch is another name used. Below is the truth table and circuit of D Flip Flop.

In real world designs (ASIC/FPGA Designs) only D latches/Flip-Flops are used.

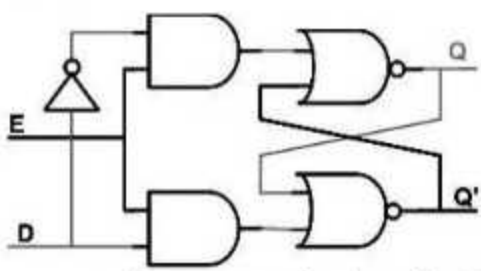


Figure 2.12: D Flip Flop with Edge Sensitive and Level sensitive

Table: Truth table for D Flip Flop

D	Q	Q+
1	X	1
0	X	0

Below is the D Flip Flop waveform, which is similar to the RS Flip Flop one, but with R removed.

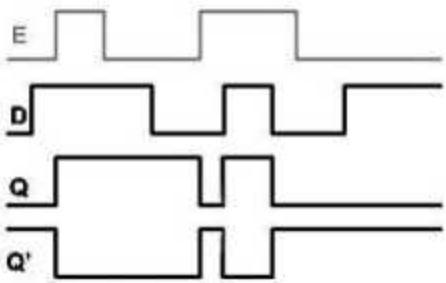


Figure: D Flip Flop waveform

Figure: D Flip Flop waveform

5 JK Flip Flop

The ambiguous state output in the RS Flip Flop was eliminated in the D Flip Flop by joining the inputs with an inverter. But the D Flip Flop has a single input. JK Flip Flop is similar to RS Flip Flop in that it has 2 inputs J and K as shown Figurer below. The ambiguous state has been eliminated here: when both inputs are high, output toggles. The only difference we see here is output feedback to inputs, which is not there in the RS Flip Flop.

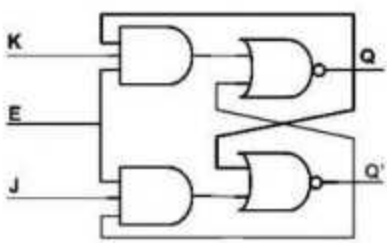


Figure: JK Flip Flop

Table: Truth table for JK Flip Flop

J	K	Q
1	1	0
1	1	1
1	0	1
0	1	0

4 T Flip Flop

When the two inputs of JK Flip Flop are shorted, a T Flip Flop is formed. It is called T Flip Flop as, when input is held HIGH, output toggles.

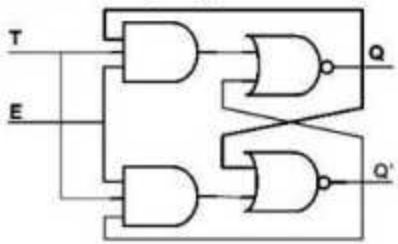


Figure : T Flip Flop

Table: T Flip Flop

T	Q	Q+
1	0	1
1	1	0
0	1	1
0	0	0

6 JK Master Slave Flip-Flop

All sequential circuits that we have seen in the last few pages have a problem (All level sensitive sequential circuits have this problem). Before the enable input changes state from HIGH to LOW (assuming HIGH is ON and LOW is OFF state), if inputs changes, then another state transition occurs for the same enable pulse. This sort of multiple transition problem is called racing.

If we make the sequential element sensitive to edges, instead of levels, we can overcome this problem, as input is evaluated only during enable/clock edges.

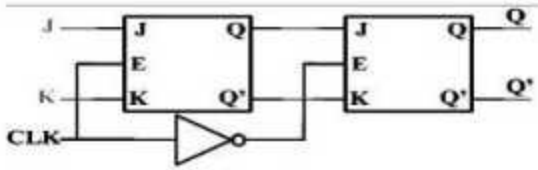


Figure: JK Master Slave Flip Flop

Figure: JK Master Slave Flip Flop

In the Figure above there are two Flip Flop, the first Flip Flop on the left is called master Flip Flop and the one on the right is called slave Flip Flop. Master Flip Flop is positively clocked and slave Flip Flop is negatively clocked.

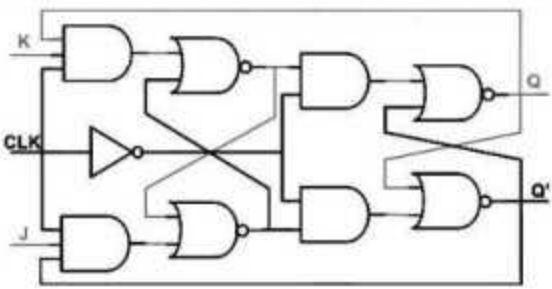


Figure : JK Master Slave Flip Flop

Figure : JK Master Slave Flip Flop

COUNTERS

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the "output."
- The output value increases by one on each clock cycle.
- After the largest value, the output "wraps around" back to 0.

Benefits of counters

- Counters can act as simple clocks to keep track of "time."

- You may need to record how many times something has happened.
  - How many bits have been sent or received?
  - How many steps have been performed in some computation?
- All processors contain a program counter, or PC.
  - Programs consist of a list of instructions that are to be executed one after another (for the most part).
  - The PC keeps track of the instruction currently being executed.
  - The PC increments once on each clock cycle, and the next program instruction is then executed.

Counter Types

Asynchronous Counter (Ripple or Serial Counter)

Each FF is triggered one at a time with output of one FF serving as clock input of next FF in the chain.

Synchronous Counter (a.k.a. Parallel Counter)

All the FF" s in the counter are clocked at the same time.

Up Counter

Counter counts from zero to a maximum count.

Down Counter

Counter counts from a maximum count down to zero.

BCD Counter

Counter counts from 0000 to 1001 before it recycles.

Pre-settable Counter

Counter that can be preset to any starting count either synchronously or asynchronously

Ring Counter

Shift register in which the output of the last FF is connected back to the input of the first FF.

Johnson Counter

Shift register in which the inverted output of the last FF is connected to the input of the first FF.

1 Synchronous Counter

There is a problem with the ripple counter just discussed. The output stages of the flip-flops further down the line (from the first clocked flip-flop) take time to respond to changes that occur due to the initial clock signal. This is a result of the internal propagation delay that occurs within a given flip-flop.

A standard TTL flip-flop may have an internal propagation delay of 30 ns. If you join four flip-flops to create a MOD-16 counter, the accumulative propagation delay at the highest-order output will be 120 ns. When used in high-precision synchronous systems, such large delays can lead to timing problems.

To avoid large delays, you can create what is called a synchronous counter. Synchronous counters, unlike ripple (asynchronous) counters, contain flip-flops whose clock inputs are driven at the same time by a common clock line. This means that output transitions for each flip-flop will occur at the same time. Now, unlike the ripple counter, you must use some additional logic circuitry placed between various flip-flop inputs and outputs to give the desired count waveform.

For example, to create a 4-bit MOD-16 synchronous counter requires adding two additional AND gates, as shown below. The AND gates act to keep a flip-flop in hold mode (if both input of the gate are low) or toggle mode (if both inputs of the gate are high). So, during the 0–1 count, the first flip-flop is in toggle mode (and always is); all the rest are held in hold mode. When it is time for the 2–4 count, the first and second flip-flops are placed in toggle mode; the last two are held in hold mode.

When it is time for the 4–8 count, the first AND gate is enabled, allowing the third flip-flop to toggle. When it is time for the 8–15 count, the second AND gate is enabled, allowing the last flip-flop to toggle

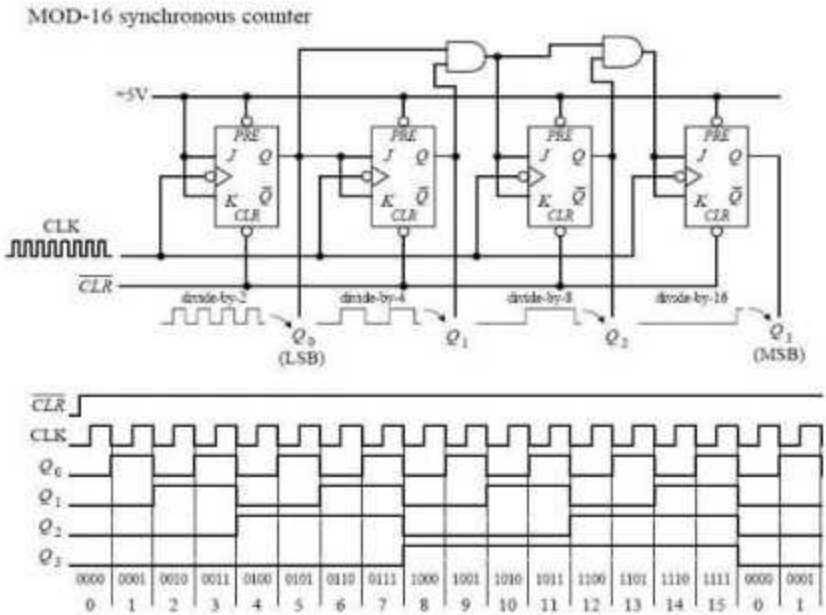


Figure: Mod 16 Synchronous Counters and Cycle Diagram

The ripple (asynchronous) and synchronous counters discussed so far are simple but hardly ever used. In practice, if you need a counter, be it ripple or synchronous, you go out and purchase a counter IC. These ICs are often MOD-16 or MOD-10 counters and usually come with many additional features. For example, many ICs allow you to preset the count to a desired number via parallel input lines.

Synchronous Up /Down Counter

The down counter counts in reverse from 1111 to 0000 and then goes to 1111. If we inspect the count cycle, we find that each flip-flop will complement when the previous flip- flops are all 0 (this is the opposite of the up counter). The down counter can be implemented similar to the up counter, except that the AND gate input is taken from Q' instead of Q. This is shown in the following Figure of a 4-bit up-down counter using T flip-flops.

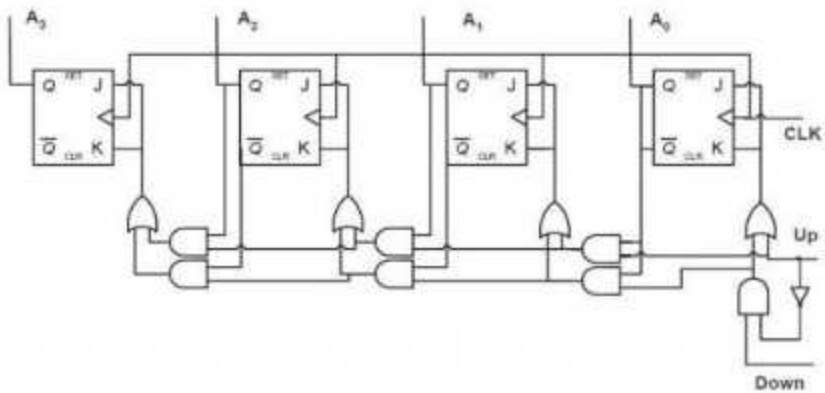


Figure: Synchronous Up /Down Counter

Figure: Synchronous Up /Down Counter

2 Asynchronous Up /Down Counter:

In certain applications, a counter must be able to count both up and down. The circuit below is a 3-bit up-down counter. It counts up or down depending on the status of the control signals UP and DOWN. When the UP input is at 1 and the DOWN input is at 0, the NAND network between FF0 and FF1 will gate the non-inverted output (Q) of FF0 into the clock input of FF1. Similarly, Q of FF1 will be gated through the other NAND network into the clock input of FF2. Thus the counter will count up.

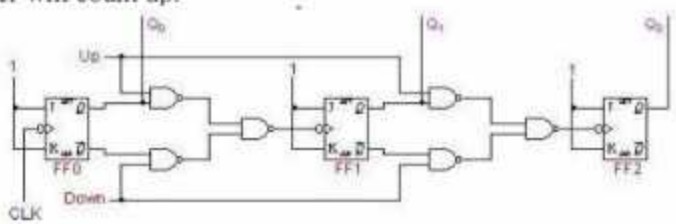


Figure: Asynchronous Up /Down Counter

When the control input UP is at 0 and DOWN is at 1, the inverted outputs of FF_0 and FF_1 are gated into the clock inputs of FF_1 and FF_2 respectively. If the flip-flops are initially y reset to 0's, then the counter will go through the following sequence as input pulses are applied

Notice that an asynchronous up-down counter is slower than an up counter or a down counter because of the additional propagation delay introduced by the NAND networks.

Design of Synchronous Counters

This section begins our study of designing an important class of clocked sequential logic circuits-synchronous fi ni t e -state machines. Like all sequential circuits, a finite-state machine determines its outputs and its next state from its current inputs and current state. A synchronous finite state machine changes state only on the clocking event.

ANALOG TO DIGITAL CONVERSION

A comparator compares the unknown voltage with a known value of voltage and then produces proportional output (i.e. it will produce either a 1 or a 0). This principle is basically used in the above circuit. Here three comparators are used. Each has two inputs. One input of each comparator is connected to analog input voltage. The other input terminals are connected to fixed reference voltage like +3/4V, +V/2 and +V/4. Now the circuit can convert analog voltage into equivalent digital signal. Since the analog output voltage is connected in parallel to all the comparators, the circuit is also called as parallel A/D converter.

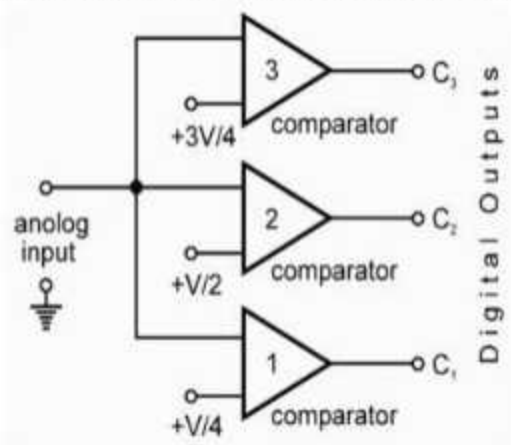


Figure: ADC Conversion

Working – Here each comparator is connected to a reference voltage of +3/4V, +V/2 and +V/4 with their outputs as $C_3C_2C_1$ respectively. Now suppose the analog input voltage change from 0 – 4V, then the actual values of reference voltages will be +3/4V = 3V, +V/2 =2V and +V/4 =1V. Now there will be following conditions of outputs of the circuit

1) When input voltage is between 0 and 1V, the output will be $C_3C_2C_1 = 000$.

2) When input voltage > 1V £ 2V, the output will be $C_3C_2C_1 = 001$.
3) When input voltage > 2V £ 3V, the output will be $C_3C_2C_1 = 011$.
4) When input voltage > 3V £ 4V, the output will be $C_3C_2C_1 = 111$.

In this way, the circuit can convert the analog input voltage into its equivalent or proportional binary number in digital style.

1 Successive Approximation Technique

The basic drawback of counter method (given above) is that it has longer conversion time. Because it always starts from 0000 at every measurement, until the analog voltage is matched. This drawback is removed in successive approximation method. In the adjacent figure, the method of successive approximation technique is shown. When unknown voltage (V_a) is applied, the circuit starts up from 0000, as shown above. The output of SAR advances with each MSB. The output of SAR does not increase step–by–step in BCD bus pattern, but individual bit becomes high–starting from MSB. Then by comparison, the bit is fixed or removed. Thus, it sets first MSB (1000), then the second MSB (0100) and so on. Every time, the output of SAR is converted to equivalent analog voltage by binary ladder. It is then compared with applied unknown voltage (V_a). The comparison process goes on, in binary search style, until the binary equivalent of analog voltage is obtained. In this way following steps are carried out during conversion.

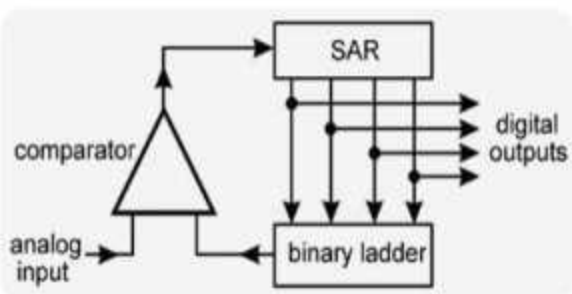


Figure: Successive Approximation Technique

Figure: Successive Approximation Technique

Now refer the following figure and the given steps -

1) The unknown analog voltage (V_a) is applied.
2) Starts up from 0000 and sets up first MSB 1000.
3) If $V_a >= 1000$, the first MSB is fixed.
4) If $V_a < 1000$, the first MSB is removed and second MSB is set
5) The fixing and removing the MSBs continues up to last bit (LSB), until equivalent binary output is obtained.

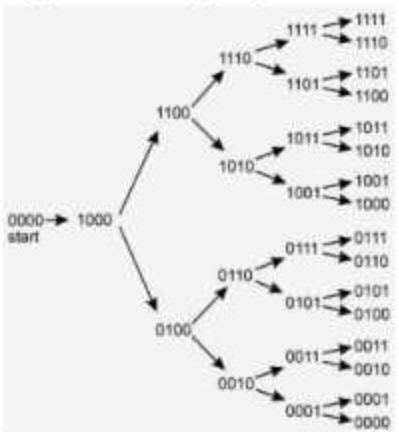


Figure 3.38 Equivalent Binary Output

Figure 3.38 Equivalent Binary Output

2 Flash ADC

Also called the parallel A/D converter, this circuit is the simplest to understand. It is formed of a series of comparators, each one comparing the input signal to a unique reference voltage. The comparator outputs connect to the inputs of a priority encoder circuit, which then produces a binary output.

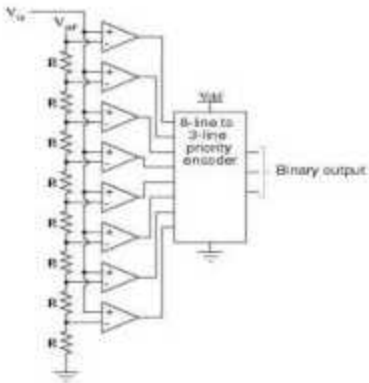


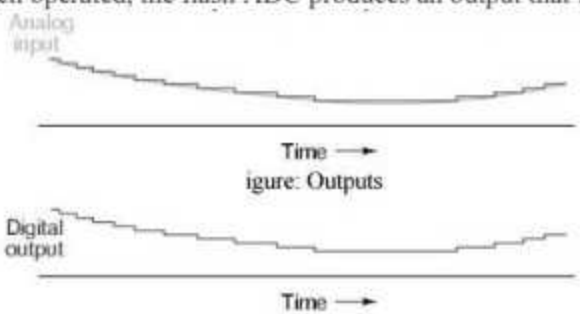
Figure: Flash ADC

Figure: Flash ADC

The following illustration shows a 3-bit flash ADC circuit:

V_{ref} is a stable reference voltage provided by a precision voltage regulator as part of the converter circuit, not shown in the schematic. As the analog input voltage exceeds the reference voltage at each comparator, the comparator outputs will sequentially saturate to a high state. The priority encoder generates a binary number based on the highest-order active input, ignoring all other active inputs.

When operated, the flash ADC produces an output that looks something like this



igure: Outputs

DIGITAL TO ANALOG CONVERTER(DAC)

The process of converting digital signal into equivalent analog signal is called D/A conversion. The electronics circuit, which does this process, is called D/A converter. The circuit has „n' number of digital data inputs with only one

output. Basically, there are two types of D/A converter circuits: Weighted resistors D/A converter circuit and Binary ladder or R–2R ladder D/A converter circuit.

1 Weighted resistors D/A converter

Here an OPAMP is used as summing amplifier. There are four resistors R, 2R, 4R and 8R at the input terminals of the OPAMP with R as feedback resistor. The network of resistors at the input terminal of OPAMP is called as variable resistor network. The four inputs of the circuit are D, C, B & A. Input D is at MSB and A is at LSB. Here we shall connect 8V DC voltage as logic–1 level. So we shall assume that 0 = 0V and 1 = 8V.

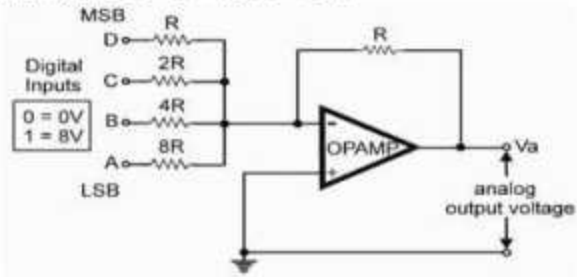


Figure: Weighted resistors D/A converter

Figure: Weighted resistors D/A converter

Now the working of the circuit is as follows. Since the circuit is summing amplifier, its output is given by the following equation

$$v_0 = R\left(\frac{D}{R} + \frac{C}{2R} + \frac{B}{4R} + \frac{A}{8R}\right)$$

Working of the circuit

When input DCBA = 0000, then putting these value in above equation (1) we get

$$v_0 = R\left(\frac{0}{R} + \frac{0}{2R} + \frac{0}{4R} + \frac{0}{8R}\right) = 0V$$

When digital input of the circuit DCBA = 0001, then putting these value in above equation (1) we get

$$v_0 = R\left(\frac{0}{R} + \frac{0}{2R} + \frac{0}{4R} + \frac{0}{8R}\right) = 0V$$

When digital input of the circuit DCBA = 0010, then putting these value in above equation (1) we get

$$v_0 = R\left(\frac{0}{R} + \frac{0}{2R} + \frac{8V}{4R} + \frac{0}{8R}\right) = -R\frac{8V}{4R} = -2V$$

..... so on.

In this way, when digital input changes from 0000 to 1111 (in BCD style), output voltage (Vo) changes proportionally. This is given in the conversion chart. There are some main disadvantages of the circuit.

They are

1) Each resistor in the circuit has different value.
2) So error in value of each resistor adds up.
3) The value of resistor at MSB is the lowest. Hence, it draws more current.
4) Also, its heat & power dissipation is very high.
5) There is the problem of impedance matching due to different values of resistors.

2 R–2R Ladder D/A Converter

It is modern type of resistor network. It has only two values of resistors the R and 2R. These values repeat throughout in the circuit. The OPAMP is used at output for scaling the output voltage. The working of the circuit can be understood as follows. For simplicity, we ignore the OPAMP in the above circuit (this is because its gain is unity). Now consider the circuit, without OPAMP. Suppose the digital input is DCBA = 1000. Then the circuit is reduced to a small circuit.

$$output = \left(\frac{2R}{2R + 2R}\right) \times (+V) = \frac{V}{2}$$

Its output is given by –

Reduced circuit of R-2R ladder, when we consider that all inputs=0

Now suppose digital input of the same circuit is changed to DCBA = 0100. Then the output voltage will be V/4, when DCBA = 0010, output voltage will be V/8, for DCBA = 0001, output voltage will be V/16 and so on. The general formula for the above circuit of R–2R ladder, including the OPAMP also, will be –