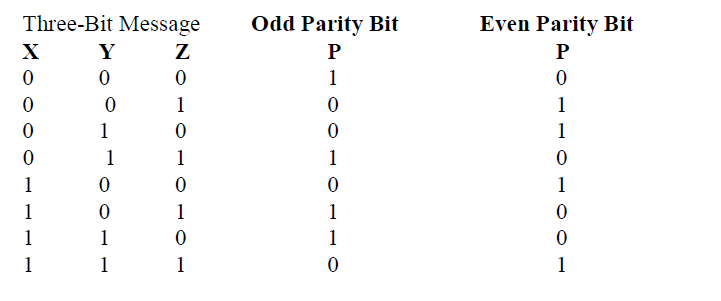
- **Error-Detection Codes**

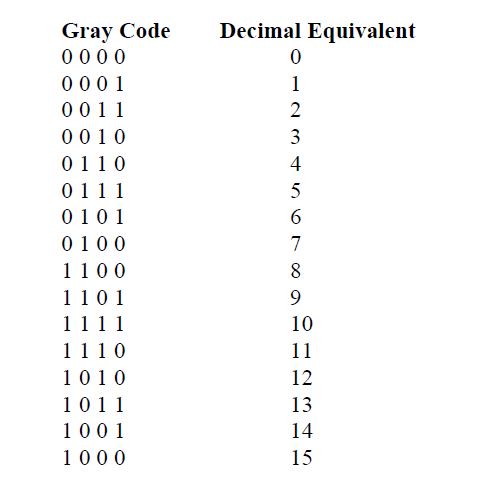
Binary information may be transmitted through some communication medium, e.g. using wires or wireless media. A corrupted bit will have its value changed from 0 to 1 or vice versa. To be able to detect errors at the receiver end, the sender sends an extra bit (parity bit) with the original binary message. A parity bit is an extra bit included with the n-bit binary message to make the total number of 1’s in this message (including the parity bit) either odd or even. If the *parity bit* makes the total number of 1’s an odd (even) number, it is called odd (even) parity. The table shows the ***required*** odd (***even*) *parity*** for a 3-bit message



No error is detectable if the transmitted message has 2 bits in error since the total number of 1’s will remain even (or odd) as in the original message. In general, a transmitted message with even number of errors cannot be detected by the parity bit.

- **Gray code**

The Gray code consist of 16 4-bit code words to represent the decimal Numbers 0 to 15. For Gray code, successive code words differ by only one bit from one to the next



**Binary Logic**

**Introduction**

Binary logic deals with variables that assume discrete values and with operators that assume logical meaning.

While each logical element or condition must always have a logic value of either "0" or "1", we also need to have ways to combine different logical signals or conditions to provide a logical result. For example, consider the logical statement: "If I move the switch on the wall up, the light will turn on." At first glance, this seems to be a correct statement. However, if we look at a few other factors, we realize that there's more to it than this. In this example, a more complete statement would be: "If I move the switch on the wall up *and* the light bulb is good *and* the power is on, the light will turn on." If we look at these two statements as logical expressions and use logical terminology, we can reduce the first statement to:

Light = Switch This means nothing more than that the light will follow the action of the switch, so that when the switch is up/on/true/1 the light will also be on/true/1. Conversely, if the switch is down/off/false/0 the light will also be off/false/0. Looking at the second version of the statement, we have a slightly more complex expression:

Light = Switch *and* Bulb *and* Power When we deal with logical circuits (as in computers), we not only need to deal with logical functions; we also need some special symbols to denote these functions in a logical diagram. There are three fundamental logical operations, from which all other functions, no matter how complex, can be derived. These functions are named *and*, *or*, and *not*. Each of these has a specific symbol and a clearly-defined behaviour.

AND. The AND operation is represented by a dot(.) or by the absence of an operator. E.g. x.y=z xy=z are all read as x AND y=z. the logical operation AND is interpreted to mean that z=1 if and only if x=1 and y=1 otherwise z=0

OR. The operation is represented by a + sign for example, x+y=z is interpreted as x OR y=z meaning that z=1 if x=1 or y=1 or if both x=1 and y=1. If both x and y are 0, then z=0

NOT. This operation is represented by a bar or a prime. For example x′==z is interpreted as NOT x =z meaning that z is what x is not *x*

It should be noted that although the AND and the OR operation have some similarity with the multiplication and addition respectively in binary arithmetic , however one should note that an arithmetic variable may consist of many digits. A binary logic variable is always 0 or 1.

e.g. in binary arithmetic, 1+1=10 while in binary logic 1+1=1

**Basic Gate**

The basic building blocks of a computer are called *logical gates* or just *gates*. Gates are basic circuits that have at least one (and usually more) *input* and exactly one output. Input and output values are the logical values *true* and *false*. In computer architecture it is common to use 0 for false and 1 for true. Gates have no *memory*. The value of the output depends only on the current value of the inputs. A useful way of describing the relationship between the inputs of gates

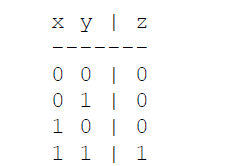
and their output is the truth table. In a truth table, the value of each output is tabulated for every possible combination of the input values. We usually consider three basic kinds of gates, *and*-gates, *or*-gates, and *not*-gates (or *inverters*).

- **The AND Gate**

The AND gate implements the AND function. With the gate shown to the left, both inputs must have logic 1 signals applied to them in order for the output to be a logic 1. With either input at logic 0, the output will be held to logic 0.



The truth table for an *and*-gate with two inputs looks like this:



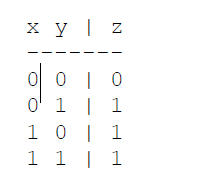
There is no limit to the number of inputs that may be applied to an AND function, so there is no functional limit to the number of inputs an AND gate may have. However, for practical reasons, commercial AND gates are most commonly manufactured with 2, 3, or 4 inputs. A standard Integrated Circuit (IC) package contains 14 or 16 pins, for practical size and handling. A standard 14-pin package can contain four 2-input gates, three 3-input gates, or two 4-input gates, and still have room for two pins for power supply connections.

- **The OR Gate**

The OR gate is sort of the reverse of the AND gate. The OR function, like its verbal counterpart, allows the output to be true (logic 1) if any one or more of its inputs are true. Verbally, we might say, "If it is raining OR if I turn on the sprinkler, the lawn will be wet." Note that the lawn will still be wet if the sprinkler is on and it is also raining. This is correctly reflected by the basic OR function. In symbols, the OR function is designated with a plus sign (+). In logical diagrams, the symbol below designates the OR gate.



The truth table for an *or*-gate with two inputs looks like this:

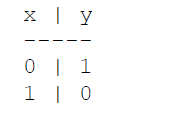


- **The NOT Gate, or Inverter**

The inverter is a little different from AND and OR gates in that it always has exactly one input as well as one output. Whatever logical state is applied to the input, the opposite state will appear at the output.



The truth table for an *inverter* looks like this:

\