**Data Types**

* Registers contain either data or control information
* Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation
* Data are numbers and other binary-coded information that are operated on
* Possible data types in registers:

1. Numbers used in computations
2. Letters of the alphabet used in data processing
   1. Other discrete symbols used for specific purposes

* All types of data, except binary numbers, are represented in binary-coded form
* A number system of *base*, or *radix, r* is a system that uses distinct symbols for *r* digits
* Numbers are represented by a string of digit symbols
* The string of digits 724.5 represents the quantity

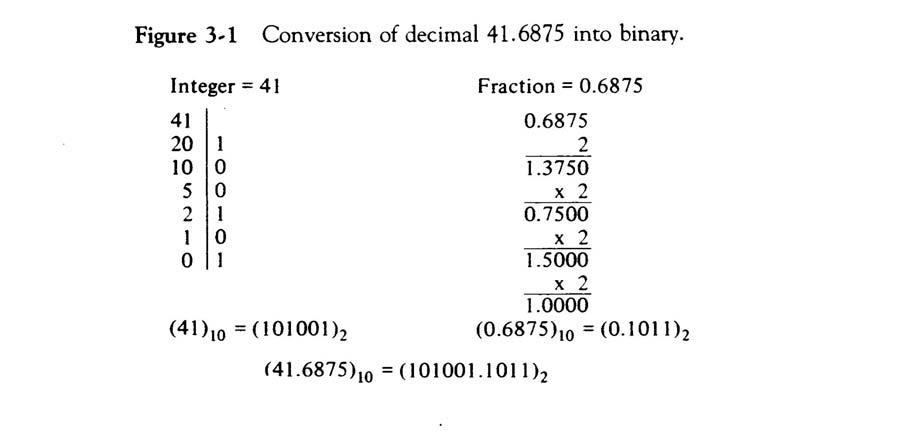
7 x 102 + 2 x 101 + 4 x 100 + 5 x 10-1

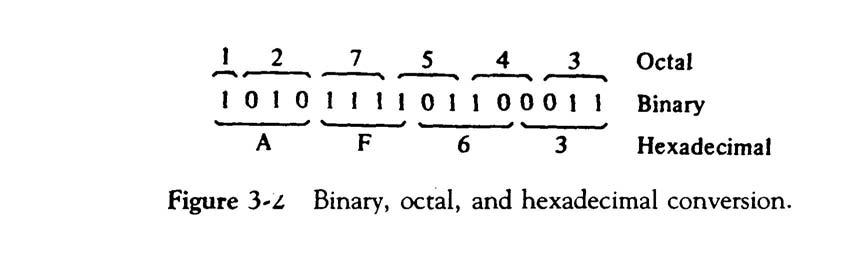
* The string of digits 101101 in the binary number system represents the quantity
  1. x 25 + 0 x 24 + 1 x 23 + 1 x 22 + 0 x 21 + 1 x 20 = 45
* (101101)2 = (45)10
* We will also use the octal (radix 8) and hexa decimal (radix 16) number systems

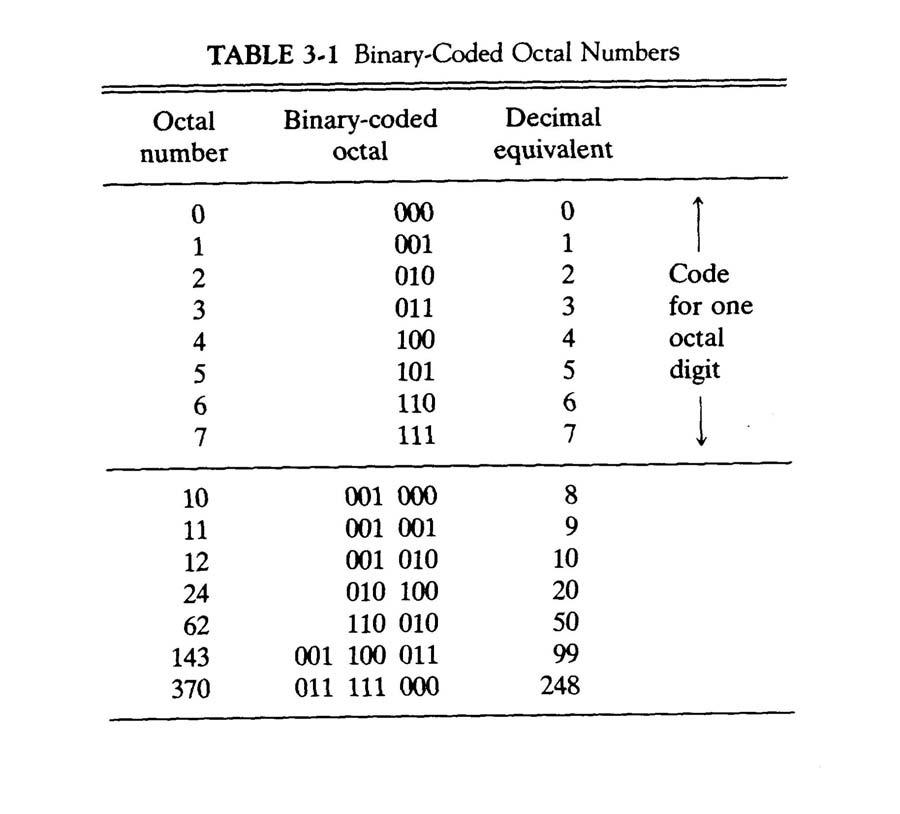
(736.4)8 = 7 x 82 + 3 x 81 + 6 x 80 + 4 x 8-1 = (478.5)10

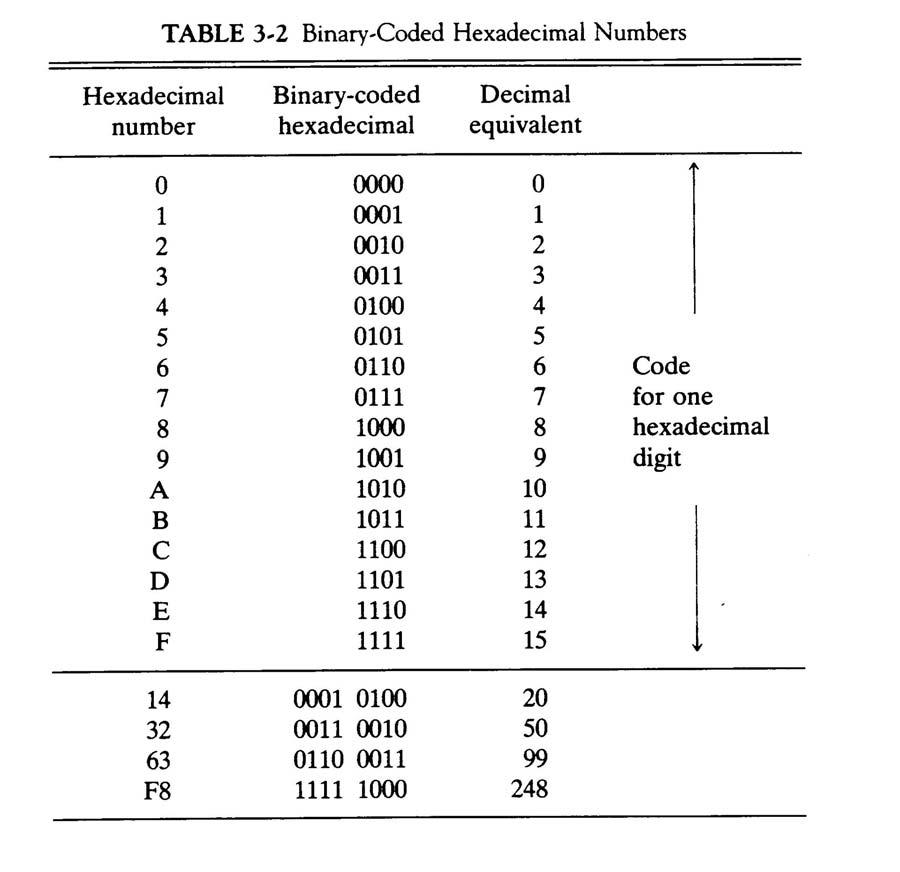
(F3)16 = F x 161 + 3 x 160 = (243)10

* Conversion from decimal to radix *r* system is carried out by separating the number into its integer and fraction parts and converting each part separately
* Divide the integer successively by *r* and accumulate the remainders
* Multiply the fraction successively by *r* until the fraction becomes zero

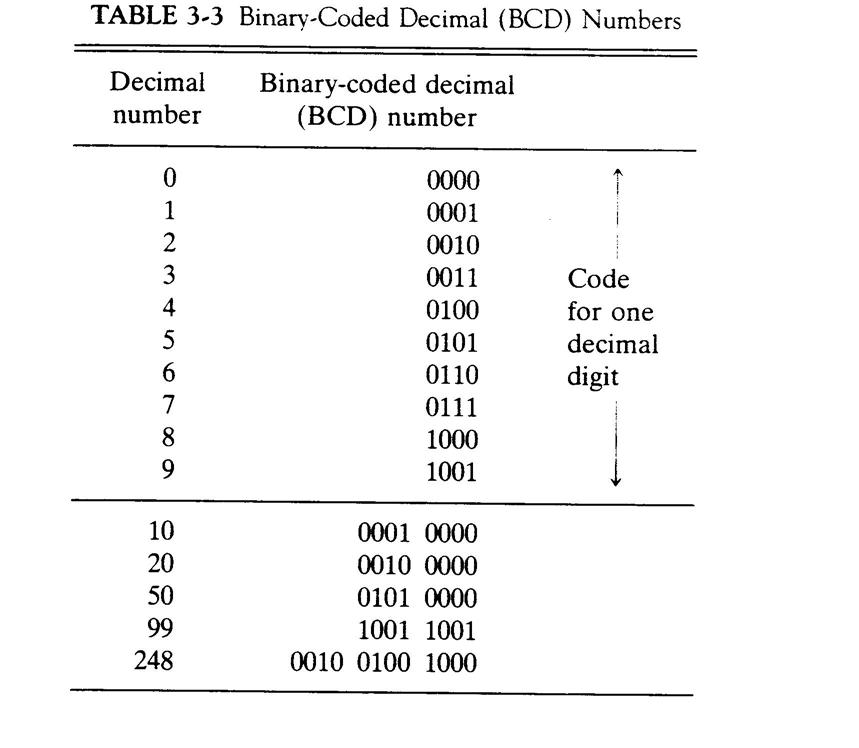


* Each octal digit corresponds to three binary digits
* Each hexadecimal digit corresponds to four binary digits
* Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or ¼, respectively

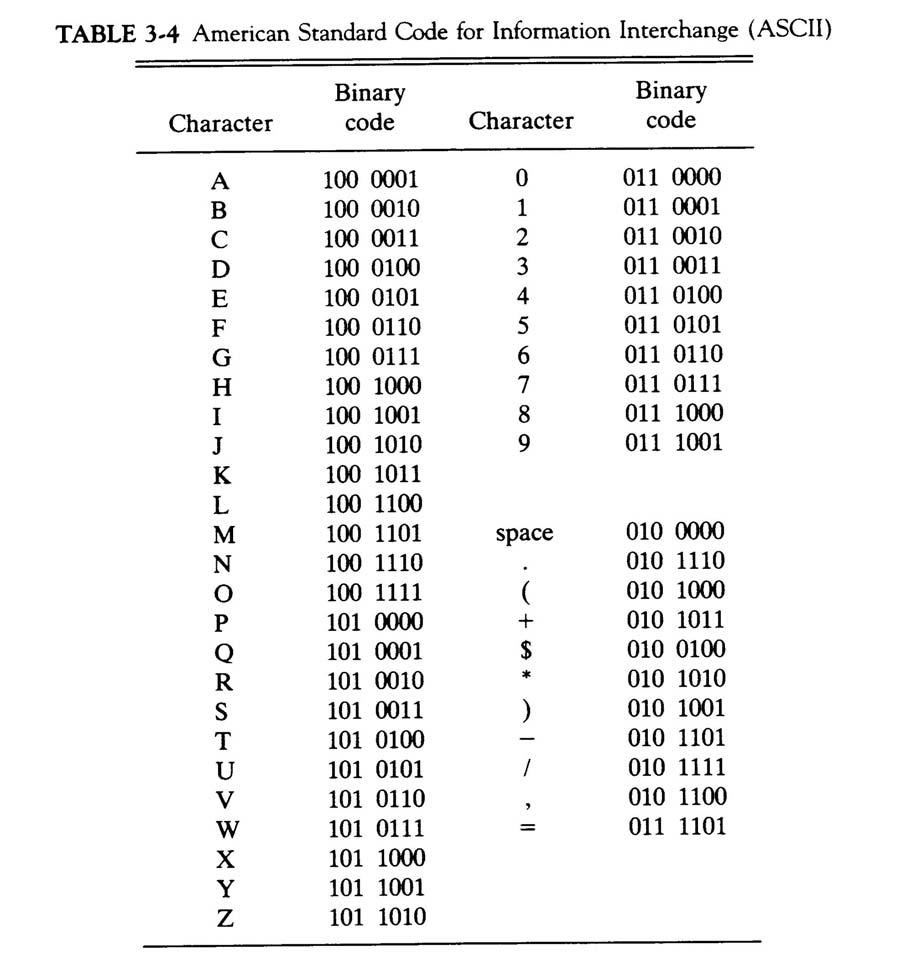




* A binary code is a group of *n* bits that assume up to 2*n* distinct combinations
* A four bit code is necessary to represent the ten decimal digits – 6 are unused
* The most popular decimal code is called *binary-coded decimal* (BCD)
* BCD is different from converting a decimal number to binary
* For example 99, when converted to binary, is 1100011
* 99 when represented in BCD is 1001 1001



* The standard alphanumeric binary code is ASCII
* This uses seven bits to code 128 characters
* Binary codes are required since registers can hold binary information only



**Complements**

* Complements are used in digital computers for simplifying subtraction and logical manipulation
* Two types of complements for each base *r* system: *r*’s complement and (*r* – 1)’s complement
* Given a number *N* in base *r* having *n* digits, the (*r* – 1)’s complement of *N* is defined as (*rn* – 1) – *N*
* For decimal, the 9’s complement of *N* is (10*n* – 1) – *N*
* The 9’s complement of 546700 is 999999 – 546700 = 453299
* The 9’s complement of 453299 is 999999 – 453299 = 546700
* For binary, the 1’s complement of *N* is (2*n* – 1) – *N*
* The 1’s complement of 1011001 is 1111111 – 1011001 = 0100110
* The 1’s complement is the true complement of the number – just toggle all bits
* The *r*’s complement of an *n*-digit number *N* in base *r* is defined as *rn* – *N*
* This is the same as adding 1 to the (*r* – 1)’s complement
* The 10’s complement of 2389 is 7610 + 1 = 7611
* The 2’s complement of 101100 is 010011 + 1 = 010100
* Subtraction of unsigned *n*-digit numbers: *M – N*

1. Add *M* to the *r*’s complement of *N* – this results in *M* + (*rn* – *N*) = *M – N + rn*
2. If *M* ≥ *N*, the sum will produce an end carry *rn* which is discarded o If *M* < *N*, the sum does not produce an end carry and is equal to

*rn* – (*N – M*), which is the *r*’s complement of (*N – M*). To obtain theanswer in a familiar form, take the *r*’s complement of the sum and place a negative sign in front.

Example: 72532 – 13250 = 59282. The 10’s complement of 13250 is 86750.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M | = | | 72352 |  |  |  |
| 10’s comp. of N | = +86750 | | |  |  |  |
| Sum |  | = 159282 | |  |  |  |
| Discard end carry | = -100000 | | | |  |  |
| Answer |  | = | 59282 |  |  |  |
| Example for M < N: 13250 – 72532 = -59282 | | | | | | |
| M | = | | 13250 |  |  |  |
| 10’s comp. of N | = +27468 | | |  |  |  |
| Sum |  | = | 40718 |  |  |  |
| No end carry |  |  |  |  |  |  |
| Answer |  | = -59282 (10’s comp. of 40718) | | | | |
| Example for X = 1010100 and Y = 1000011 | | | | | | |
| X | = | | 1010100 | |  |  |
| 2’s comp. of Y | = +0111101 | | | |  |  |
| Sum |  | = 10010001 | | |  |  |
| Discard end carry | = -10000000 | | | | | |
| Answer X – Y |  | = | 0010001 | | |  |
| Y | = | | 1000011 | | |  |
| 2’s comp. of X | = +0101100 | | | | |  |
| Sum |  | = | 1101111 | |  |  |
| No end carry  Answer  = -0010001 (2’s comp. of 1101111) |  |  |  | | | |
|  |  |  |  |  |  |  |

**Fixed-Point Representation**

* Positive integers and zero can be represented by unsigned numbers
* Negative numbers must be represented by signed numbers since + and – signs are not available, only 1’s and 0’s are
* Signed numbers have msb as 0 for positive and 1 for negative – msb is the sign bit
* Two ways to designate binary point position in a register

1. Fixed point position
   1. Floating-point representation

* Fixed point position usually uses one of the two following positions
  1. A binary point in the extreme left of the register to make it a fraction
  2. A binary point in the extreme right of the register to make it an integer
  3. In both cases, a binary point is not actually present
* The floating-point representations uses a second register to designate the position of the binary point in the first register
* When an integer is positive, the msb, or sign bit, is 0 and the remaining bits represent the magnitude
* When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of three ways
  1. Signed-magnitude representation
  2. Signed-1’s complement representation
  3. Signed-2’s complement representation
* Consider an 8-bit register and the number +14
  1. The only way to represent it is 00001110
* Consider an 8-bit register and the number –14

|  |  |  |
| --- | --- | --- |
| o | Signed magnitude: | 1 0001110 |
| o | Signed 1’s complement: | 1 1110001 |
| o | Signed 2’s complement: | 1 1110010 |

* Typically use signed 2’s complement
* Addition of two signed-magnitude numbers follow the normal rules

1. If same signs, add the two magnitudes and use the common sign
2. Differing signs, subtract the smaller from the larger and use the sign of the larger magnitude
   1. Must compare the signs and magnitudes and then either add or subtract

* Addition of two signed 2’s complement numbers does not require a comparison or subtraction – only addition and complementation
  1. Add the two numbers, including their sign bits

1. Discard any carry out of the sign bit position
2. All negative numbers must be in the 2’s complement form
3. If the sum obtained is negative, then it is in 2’s complement form

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| +6 | 00000110 | -6 | | 11111010 |
| +13 | 00001101 | +13 | | 00001101 |
| +19 | 00010011 |  | +7 | 00000111 |
| +6 | 00000110 | -6 | | 11111010 |
| -13 | 11110011 | -13 | | 11110011 |
| -7 | 11111001 |  | -19 | 11101101 |

* Subtraction of two signed 2’s complement numbers is as follows
  1. Take the 2’s complement form of the subtrahend (including sign bit) o Add it to the minuend (including the sign bit)
  2. A carry out of the sign bit position is discarded
* An *overflow* occurs when two numbers of *n* digits each are added and the sum occupies *n* + 1 digits
* Overflows are problems since the width of a register is finite
* Therefore, a flag is set if this occurs and can be checked by the user
* Detection of an overflow depends on if the numbers are signed or unsigned
* For unsigned numbers, an overflow is detected from the end carry out of the msb
* For addition of signed numbers, an overflow cannot occur if one is positive and one is negative – both have to have the same sign
* An overflow can be detected if the carry into the sign bit position and the carry out of the sign bit position are not equal

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| +70 | 0 1000110 | -70 | | 1 0111010 |
| +80 | 0 1010000 | -80 | | 1 0110000 |
| +150 | 1 0010110 |  | -150 | 0 1101010 |

* The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit
* A 4-bit decimal code requires four flip-flops for each decimal digit
* This takes much more space than the equivalent binary representation and the circuits required to perform decimal arithmetic are more complex
* Representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary
* Either signed magnitude or signed complement systems
* The sign of a number is represented with four bits
  1. 0000 for + o 1001 for –
* To obtain the 10’s complement of a BCD number, first take the 9’s complement and then add one to the least significant digit
* Example: (+375) + (-240) = +135

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 375 | (0000 | 0011 | 0111 | 1010)BCD | |
| +9 | 760 | (1001 | 0111 | 0110 | 0000) | BCD |
| 0 | 135 | (0000 | 0001 | 0011 | 0101) |  |
| BCD |

**Floating-Point Representation**

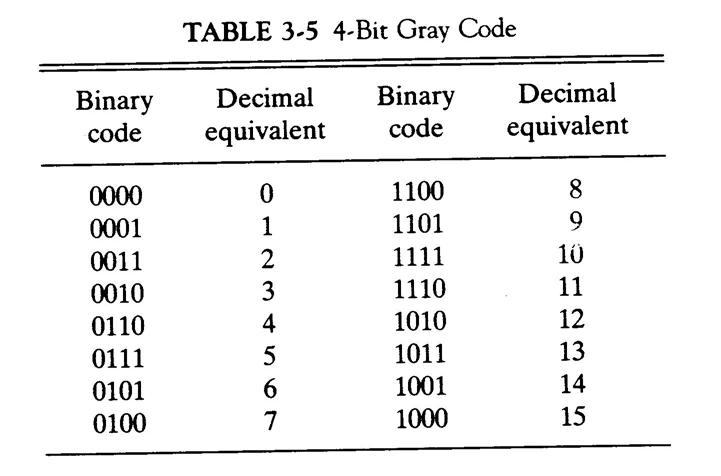
* The floating-point representation of a number has two parts
* The first part represents a signed, fixed-point number – the *mantissa*
* The second part designates the position of the binary point – the *exponent*
* The mantissa may be a fraction or an integer
* Example: the decimal number +6132.789 is
  1. Fraction: +0.6123789
  2. Exponent: +04
  3. Equivalent to +0.6132789 x 10+4
* A floating-point number is always interpreted to represent *m* x *re*
* Example: the binary number +1001.11 (with 8-bit fraction and 6-bit exponent)
  1. Fraction: 01001110

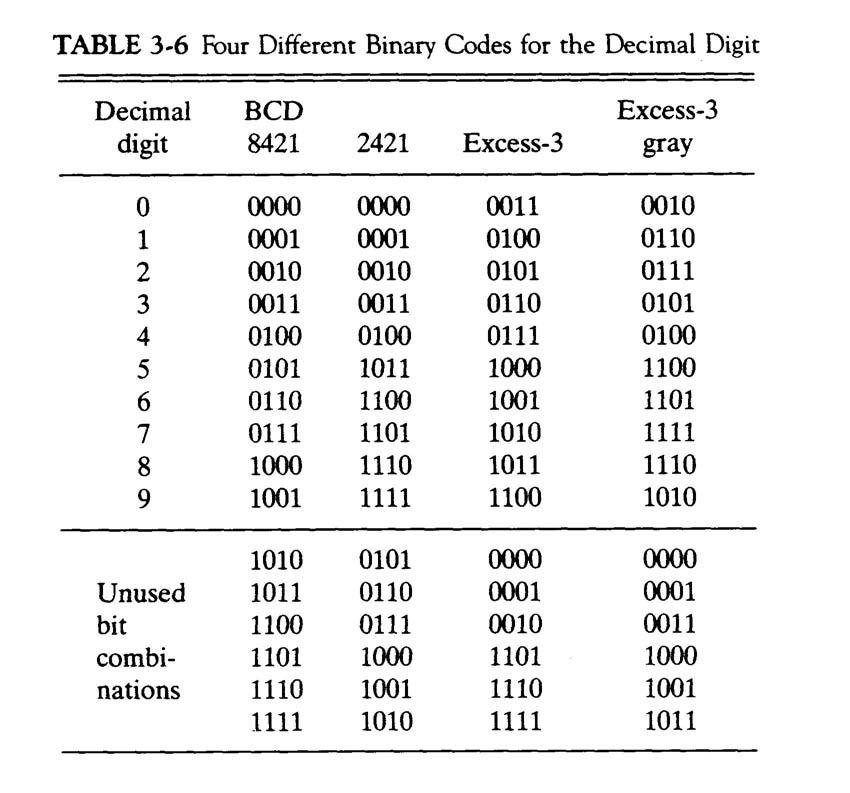
1. Exponent: 000100
   1. Equivalent to +(.1001110)2 x 2+4

* A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero
* The decimal number 350 is normalized, 00350 is not
* The 8-bit number 00011010 is not normalized
* Normalize it by fraction = 11010000 and exponent = -3
* Normalized numbers provide the maximum possible precision for the floating-point number

**Other Binary Codes**

* Digital systems can process data in discrete form only
* Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter
* The reflected binary or *Gray code*, is sometimes used for the converted digital data
* The Gray code changes by only one bit as it sequences from one number to the next
* Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system

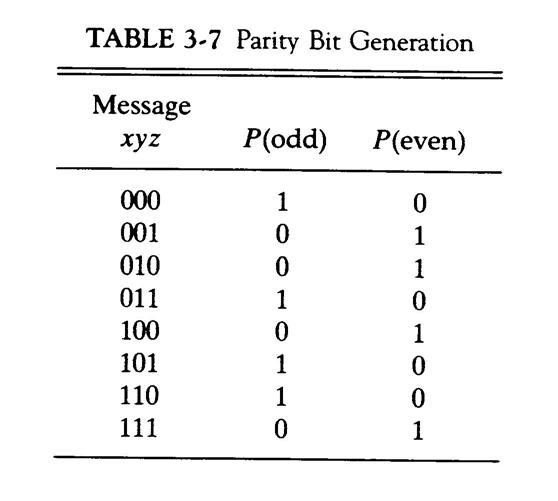




* Binary codes for decimal digits require a minimum of four bits
* Other codes besides BCD exist to represent decimal digits
* The 2421 code and the excess-3 code are both *self-complementing*
* The 9’s complement of each digit is obtained by complementing each bit in the code
* The 2421 code is a *weighted code*
* The bits are multiplied by indicated weights and the sum gives the decimal digit
* The excess-3 code is obtained from the corresponding BCD code added to 3

**Error Detection Codes**

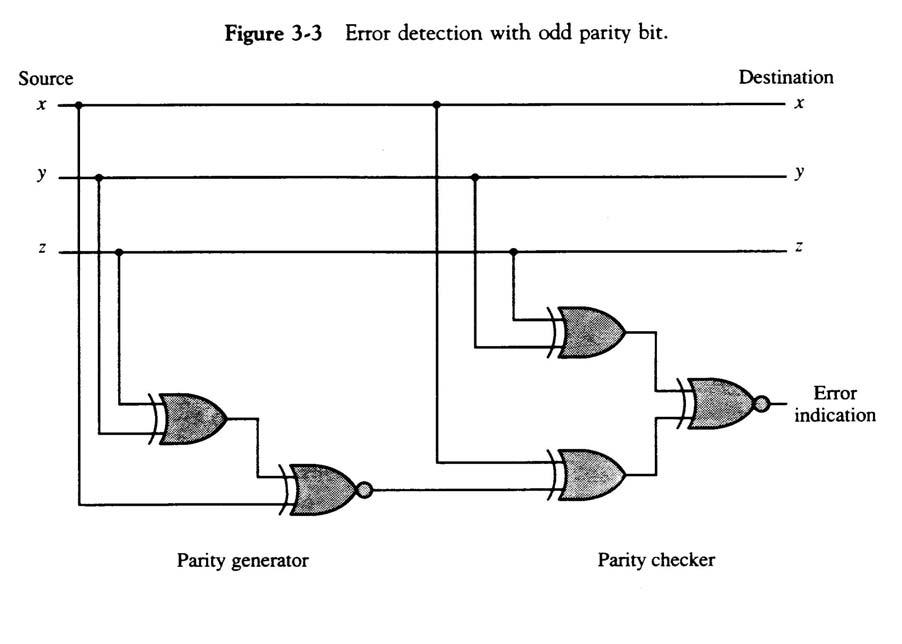
* Transmitted binary information is subject to noise that could change bits 1 to 0 and vice versa
* An *error detection code* is a binary code that detects digital errors during transmission
* The detected errors cannot be corrected, but can prompt the data to be retransmitted
* The most common error detection code used is the *parity bit*
* A parity bit is an extra bit included with a binary message to make the total number of 1’s either odd or even



* The P(odd) bit is chosen to make the sum of 1’s in all four bits odd
* The even-parity scheme has the disadvantage of having a bit combination of all 0’s
* Procedure during transmission:
  1. At the sending end, the message is applied to a *parity generator* o The message, including the parity bit, is transmitted

o At the receiving end, all the incoming bits are applied to a *parity checker*

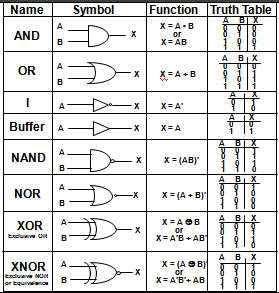
* 1. Any odd number of errors are detected
* Parity generators and checkers are constructed with XOR gates (odd function)
* An odd function generates 1 iff an odd number if input variables are 1



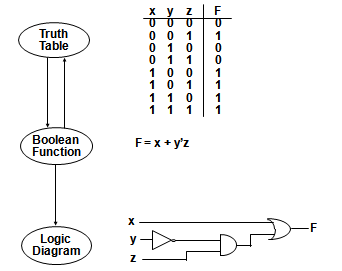
**Digital Computers**

Imply that the computer deals with digital information, i.e., it deals with the information that is represented by binary digits.Why BINARY ? instead of Decimal or other number system ? Consider the calculation cost – Add

**Logic Gates**

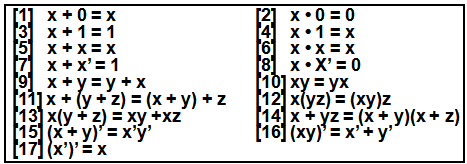
****

**Boolean Algebra**

* Algebra with Binary(Boolean) Variable and Logic Operations
* Boolean Algebra is useful in Analysis and Synthesis of
  + Digital Logic Circuits
    - Input and Output signals can be
  + represented by Boolean Variables, and
    - Function of the Digital Logic Circuits can be represented by
    - Logic Operations, i.e., Boolean Function(s)
    - From a Boolean function, a logic diagram
  + can be constructed using AND, OR, and I
* Truth Table
* The most elementary specification of the function of a Digital Logic
  + Circuit is the Truth Table
    - Table that describes the Output Values for all the combinations
    - of the Input Values, called *MINTERMS*
    - n input variables → 2n minterms

LOGIC CIRCUIT DESIGN

BASIC IDENTITIES OF BOOLEAN ALGEBRA

COMPLEMENT OF FUNCTIONS

A Boolean function of a digital logic circuit is represented by only using

logical variables and AND, OR, and Invert operators.

* Complement of a Boolean function

* Replace all the variables and sub expressions in the parentheses

appearing in the function expression with their respective complements

A,B,...,Z,a,b,...,z -> A’,B’,...,Z’,a’,b’,...,z’

(p + q) -> (p + q)’

* Replace all the operators with their respective complementary operators

AND -> OR

OR -> AND

* Basically, extensive applications of the De Morgan’s theorem

(x1 + x2 + ... + xn )’ -> x1’x2’... xn’

(x1x2 ... xn)' -> x1' + x2' +...+ xn'

Simplification from Boolean function

* Finding an equivalent expression that is least expensive to implement
* For a simple function, it is possible to obtain a simple expression for low cost implementation But, with complex functions, it is a very difficult task

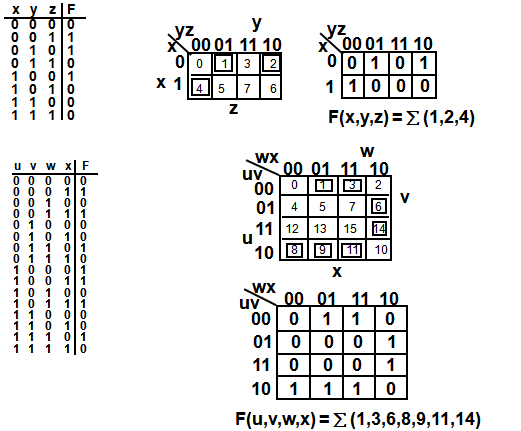
**KARNAUGH MAP**

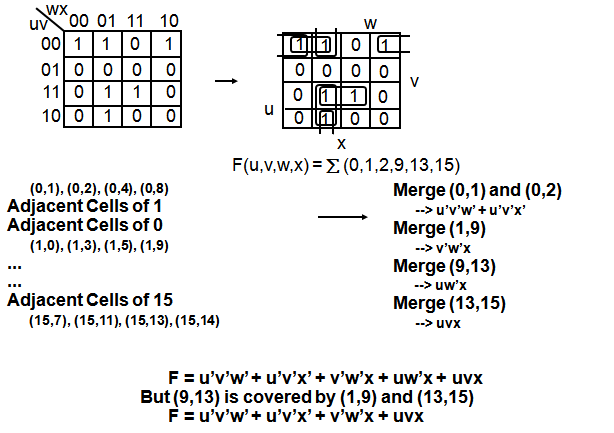
Karnaugh Map for an n-input digital logic circuit (n-variable sum-of-products form of Boolean Function, or Truth Table) is

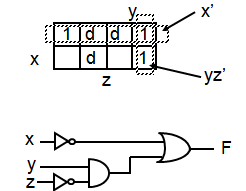
* Rectangle divided into 2n cells
* Each cell is associated with a Minterm
* An output(function) value for each input value associated with a

mintern is written in the cell representing the minterm 1-cell, 0-cell

Each Minterm is identified by a decimal number whose binary representation is identical to the binary interpretation of the input values of the minterm.





Don’t-care conditions are useful in minimizing the logic functions using K-map.Can be considered either 1 or 0.Thus increases the chances of merging cells into the larger cells Reduce the number of variables in the product terms.

