Digital Logic Design

Introduction

A digital computer stores data in terms of digits (numbers) and proceeds in discrete steps from one state to the next. The states of a digital computer typically involve binary digits which may take the form of the presence or absence of magnetic markers in a storage medium , on-off switches or relays. In digital computers, even letters, words and whole texts are represented digitally.

Digital Logic is the basis of electronic systems, such as computers and cell phones. Digital Logic is rooted in binary code, a series of zeroes and ones each having an opposite value. This system facilitates the design of electronic circuits that convey information, including logic gates. Digital Logic gate functions include and, or and not. The value system translates input signals into specific output. Digital Logic facilitates computing, robotics and other electronic applications.

Digital Logic Design is foundational to the fields of electrical engineering and computer engineering. Digital Logic designers build complex electronic components that use both electrical and computational characteristics. These characteristics may involve power, current, logical function, protocol and user input. Digital Logic Design is used to develop hardware, such as circuit boards and microchip processors. This hardware processes user input, system protocol and other data in computers, navigational systems, cell phones or other high-tech systems.

Data Representation and Number system

Numeric systems

The numeric system we use daily is the decimal system, but this system is not convenient for machines since the information is handled codified in the shape of on or off bits; this way of codifying takes us to the necessity of knowing the positional calculation which will allow us to express a number in any base where we need it.

Radix number systems

The numeric system we use daily is the decimal system, but this system is not convenient for machines since the information is handled codified in the shape of on or off bits; this way of codifying takes us to the necessity of knowing the positional calculation which will allow us to express a number in any base where we need it.

A base of a number system or radix defines the range of values that a digit may have.

In the binary system or base 2, there can be only two values for each digit of a number, either a "0" or a "1".

In the octal system or base 8, there can be eight choices for each digit of a number:

"0", "1", "2", "3", "4", "5", "6", "7".

In the decimal system or base 10, there are ten different values for each digit of a number:

"0", "1", "2", "3", "4", "5", "6", "7", "8", "9".

In the hexadecimal system, we allow 16 values for each digit of a number:

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

Where "A" stands for 10, "B" for 11 and so on.

Conversion among radices

- Convert from Decimal to Any Base

Let's think about what you do to obtain each digit. As an example, let's start with a decimal number 1234 and convert it to decimal notation. To extract the last digit, you move the decimal point left by one digit, which means that you divide the given number by its base 10.

1234/10 = 123 + 4/10

The remainder of 4 is the last digit. To extract the next last digit, you again move the decimal point left by one digit and see what drops out.

123/10 = 12 + 3/10

The remainder of 3 is the next last digit. You repeat this process until there is nothing left. Then you stop. In summary, you do the following:

```
Quotient  Remainder

          ----------------------------
  1234/10 =   123       4 --------+
   123/10 =    12       3 ------+ |
    12/10 =     1       2 ----+ | |
     1/10 =     0       1 --+ | | |(Stop when the quotient is 0)
                            | | | |
                            1 2 3 4   (Base 10)
```

Now, let's try a nontrivial example. Let's express a decimal number 1341 in binary notation. Note that the desired base is 2, so we repeatedly divide the given decimal number by **2**.

```
           Quotient  Remainder
          ----------------------------
  1341/2  =   670       1 ---------------------+
   670/2  =   335       0 -------------------+ |
   335/2  =   167       1 -----------------+ | |
   167/2  =    83       1 ---------------+ | | |
    83/2  =    41       1 -------------+ | | | |
    41/2  =    20       1 -----------+ | | | | |
    20/2  =    10       0 ---------+ | | | | | |
    10/2  =     5       0 -------+ | | | | | | |
     5/2  =     2       1 ------+ | | | | | | | |
     2/2  =     1       0 ----+ | | | | | | | | |
     1/2  =     0       1 --+ | | | | | | | | | |(Stop when the
                            | | | | | | | | | | | quotient is 0)
                            1 0 1 0 0 1 1 1 1 0 1  (BIN; Base 2)
```

Let's express the same decimal number 1341 in octal notation.

```
           Quotient  Remainder
          ----------------------------
  1341/8  =   167       5 --------+
   167/8  =    20       7 ------+ |
    20/8  =     2       4 ----+ | |
     2/8  =     0       2 --+ | | |   (Stop when the quotient is 0)
                           | | | |
                           2 4 7 5   (OCT; Base 8)
```

Let's express the same decimal number 1341 in hexadecimal notation.

```
           Quotient  Remainder
          ----------------------------
  1341/16 =    83      13 ------+
    83/16 =     5       3 ----+ |
     5/16 =     0       5 --+ | |   (Stop when the quotient is 0)
                           | | |
                           5 3 D   (HEX; Base 16)
```

In conclusion, the easiest way to convert fixed point numbers to any base is to convert each part separately. We begin by separating the number into its integer and fractional part. The integer part is converted using the remainder method, by using a successive division of the number by the base until a zero is obtained. At each division, the reminder is kept and then the new number in the base r is obtained by reading the remainder from the lat remainder upwards.

The conversion of the fractional part can be obtained by successively multiplying the fraction with the base. If we iterate this process on the remaining fraction, then we will obtain successive significant digit. This methods form the basis of

the multiplication methods of converting fractions between bases

Example. Convert the decimal number 3315 to hexadecimal notation. What about the hexadecimal equivalent of the decimal number 3315.3?

Solution:

```
                Quotient  Remainder
    ----------------------------
    3315/16 =   207        3 ------+
     207/16 =    12       15 ----+ |
      12/16 =     0       12 --+ | |  (Stop when the quotient is 0)
                              | | |
                            C F 3  (HEX; Base 16)


                                      (HEX; Base 16)
            Product  Integer Part   0.4 C C C ...
    -------------------------------   | | | |
    0.3*16   =   4.8        4      ----+ | | | | |
    0.8*16   =  12.8       12      ------+ | | | |
    0.8*16   =  12.8       12      --------+ | | |
    0.8*16   =  12.8       12      ----------+ | |
             :                    --------------------+
             :
    Thus, 3315.3 (DEC) --> CF3.4CCC... (HEX)
```

- Convert From Any Base to Decimal

Let's think more carefully what a decimal number means. For example, 1234 means that there are four boxes (digits); and there are 4 one's in the right-most box (least significant digit), 3 ten's in the next box, 2 hundred's in the next box, and finally 1 thousand's in the left-most box (most significant digit). The total is 1234:

```
    Original Number:      1     2     3     4
                          |     |     |     |
    How Many Tokens:      1     2     3     4
    Digit/Token Value: 1000   100    10     1
    Value:             1000 + 200  + 30  + 4  = 1234
```

or simply, 1*1000 + 2*100 + 3*10 + 4*1 = 1234

Thus, each digit has a value: $10^0=1$ for the least significant digit, increasing to $10^1=10$, $10^2=100$, $10^3=1000$, and so forth.

Likewise, the least significant digit in a hexadecimal number has a value of $16^0=1$ for the least significant digit, increasing to $16^1=16$ for the next digit, $16^2=256$ for the next, $16^3=4096$ for the next, and so forth. Thus, 1234 means that there are four boxes (digits); and there are 4 one's in the right-most box (least significant digit), 3 sixteen's in the next box, 2 256's in the next, and 1 4096's in the left-most box (most significant digit). The total is:

$1*4096 + 2*256 + 3*16 + 4*1 = 4660$

In summary, the conversion from any base to base 10 can be obtained from the formulae

$x = \sum\limits^{n-1} d\, b^i$ Where b is the base, d the digit at position i, m the number of digit after the decimal point, n the number

$$1\ 0 \qquad i \qquad\qquad\qquad i$$
$$i=-m$$

of digits of the integer part and X_{10} is the obtained number in decimal. This form the basic of the polynomial method of converting numbers from any base to decimal

Example. Convert 234.14 expressed in an octal notation to decimal.

$2*8^2 + 3*8^1 + 4*8^0 + 1*8^{-1} + 4*8^{-2} \qquad = 2*64 + 3*8 + 4*1 + 1/8 + 4/64 = 156.1875$

Example. Convert the hexadecimal number 4B3 to decimal notation. What about the decimal equivalent of the hexadecimal number 4B3.3?

Solution:

```
Original Number:     4     B     3  .  3
                     |     |     |     |
How Many Tokens:     4    11     3     3
Digit/Token Value: 256    16     1     0.0625
Value:            1024 +176   + 3   + 0.1875  = 1203.1875
```

Example. Convert 234.14 expressed in an octal notation to decimal.

Solution:

```
Original Number:     2     3     4  .  1         4
                     |     |     |     |          |
How Many Tokens:     2     3     4     1          4
Digit/Token Value:  64     8     1     0.125      0.015625
Value:             128 + 24   + 4   + 0.125 + 0.0625    = 156.1875
```

- Relationship between Binary - Octal and Binary-hexadecimal

As demonstrated by the table bellow, there is a direct correspondence between the binary system and the octal system, with three binary digits corresponding to one octal digit. Likewise, four binary digits translate directly into one hexadecimal digit.

```
BIN    OCT    HEX    DEC
-----------------------
0000   00      0      0
0001   01      1      1
0010   02      2      2
0011   03      3      3
0100   04      4      4
0101   05      5      5
0110   06      6      6
0111   07      7      7
-----------------------
1000   10      8      8
1001   11      9      9
1010   12      A     10
1011   13      B     11
1100   14      C     12
1101   15      D     13
1110   16      E     14
1111   17      F     15
```

With such relationship. In order to convert a binary number to octal, we partition the base 2 number into groups of three

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

starting from the radix point, and pad the outermost groups with 0's as needed to form triples. Then, we convert each triple to the octal equivalent.

For conversion from base 2 to base 16, we use groups of four.

Consider converting 10110_2 to base 8:

$10110_2 = 010_2\ 110_2 = 2_8\ 6_8 = 26_8$

Notice that the leftmost two bits are padded with a 0 on the left in order to create a full triplet.

Now consider converting 10110110_2 to base 16:

$10110110_2 = 1011_2\ 0110_2 = B_{16}\ 6_{16} = B6_{16}$

(Note that 'B' is a base 16 digit corresponding to 11_{10}. B is not a variable.)

The conversion methods can be used to convert a number from any base to any other base, but it may not be very intuitive to convert something like 513.03 to base 7. As an aid in performing an unnatural conversion, we can convert to the more familiar base 10 form as an intermediate step, and then continue the conversion from base 10 to the target base. As a general rule, we use the polynomial method when converting *into* base 10, and we use the remainder and multiplication methods when converting *out* of base 10.

Numeric complements

The radix complement of an n digit number y in radix b is, by definition, $b^n - y$. Adding this to x results in the value $x + b^n - y$ or $x - y + b^n$. Assuming $y \le x$, the result will always be greater than b^n and dropping the initial '1' is the same as subtracting b^n, making the result $x - y + b^n - b^n$ or just $x - y$, the desired result.

The radix complement is most easily obtained by adding 1 to the diminished radix complement, which is $(b^n - 1) - y$. Since $(b^n - 1)$ is the digit $b - 1$ repeated n times (because $b^n - 1 = b^n - 1^n = (b - 1)(b^{n-1} + b^{n-2} + ... + b + 1) = (b - 1)b^{n-1} + ... + (b - 1)$, see also binomial numbers), the diminished radix complement of a number is found by complementing each digit with respect to $b - 1$ (that is, subtracting each digit in y from $b - 1$). Adding 1 to obtain the radix complement can be done separately, but is most often combined with the addition of x and the complement of y.

In the decimal numbering system, the radix complement is called the *ten's complement* and the diminished radix complement the *nines' complement*.

In binary, the radix complement is called the *two's complement* and the diminished radix complement the *ones' complement*. The naming of complements in other bases is similar.

- Decimal example

To subtract a decimal number y from another number x using the method of complements, the ten's complement of y (nines' complement plus 1) is added to x. Typically, the nines' complement of y is first obtained by determining the complement of each digit. The complement of a decimal digit in the nines' complement system is the number that must be added to it to produce 9. The complement of 3 is 6, the complement of 7 is 2, and so on. Given a subtraction problem:

```
   873   (x)
 - 218   (y)
```
The nines' complement of y (218) is 781. In this case, because y is three digits long, this is the same as subtracting y

from 999. (The number of 9's is equal to the number of digits of *y*.)

Next, the sum of *x*, the nines' complement of *y*, and 1 is taken:

```
  873   (x)
+ 781   (complement of y)
+   1   (to get the ten's complement of y)
=====
 1655
```

The first "1" digit is then dropped, giving 655, the correct answer.

If the subtrahend has fewer digits than the minuend, leading zeros must be added which will become leading nines when the nines' complement is taken. For example:

```
  48032   (x)
-   391   (y)
```
becomes the sum:
```
  48032   (x)
+ 99608   (nines' complement of y)
+     1   (to get the ten's complement)
=======
 147641
```
Dropping the "1" gives the answer: 47641

- Binary example

The method of complements is especially useful in binary (radix 2) since the ones' complement is very easily obtained by inverting each bit (changing '0' to '1' and vice versa). And adding 1 to get the two's complement can be done by simulating a carry into the least significant bit. For example:

```
  01100100   (x, equals decimal 100)
- 00010110   (y, equals decimal 22)
```

becomes the sum:
```
  01100100   (x)
+ 11101001   (ones' complement of y)
+        1   (to get the two's complement)
==========
 101001110
```

Dropping the initial "1" gives the answer: 01001110 (equals decimal 78)

Signed fixed point numbers

Up to this point we have considered only the representation of unsigned fixed point numbers. The situation is quite different in representing *signed* fixed point numbers. There are four different ways of representing signed numbers that are commonly used: sign-magnitude, one's complement, two's complement, and excess notation. We will cover each in turn, using integers for our examples. The Table below shows for a 3-bit number how the various representations appear.

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

Decimal	Unsigned	Sign–Mag.	1's Comp.	2's Comp.	Excess 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	–	100	111	000	100
-1	–	101	110	111	011
-2	–	110	101	110	010
-3	–	111	100	101	001
-4	–	–	–	100	000

Table1. 3 bit number representation

- Signed Magnitude Representation

The signed magnitude (also referred to as sign and magnitude) representation is most familiar to us as the base 10 number system. A plus or minus sign to the left of a number indicates whether the number is positive or negative as in $+12_{10}$ or -12_{10}. In the binary signed magnitude representation, the leftmost bit is used for the sign, which takes on a value of 0 or 1 for '+' or '−', respectively. The remaining bits contain the absolute magnitude.

Consider representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format:

$(+12)_{10} = (00001100)_2$

$(-12)_{10} = (10001100)_2$

The negative number is formed by simply changing the sign bit in the positive number from 0 to 1. Notice that there are both positive and negative representations for zero: +0= 00000000 and -0= 10000000.

- One's Complement Representation

The one's complement operation is trivial to perform: convert all of the 1's in the number to 0's, and all of the 0's to 1's. See the fourth column in Table1 for examples. We can observe from the table that in the one's complement representation the leftmost bit is 0 for positive numbers and 1 for negative numbers, as it is for the signed magnitude representation. This negation, changing 1's to 0's and changing 0's to 1's, is known as complementing the bits. Consider again representing $(+12)10$ and $(-12)10$ in an eight-bit format, now using the one's complement representation:

$(+12)10 = (00001100)2$

$(-12)10 = (11110011)2$

Note again that there are representations for both +0 and −0, which are 00000000 and 11111111, respectively. As a result, there are only $2^8 - 1 = 255$ different numbers that can be represented even though there are 2^8 different bit patterns.

The one's complement representation is not commonly used. This is at least partly due to the difficulty in making comparisons when there are two representations for 0. There is also additional complexity involved in adding numbers.

- Two's Complement Representation

The two's complement is formed in a way similar to forming the one's complement: complement all of the bits in the number, but then add 1, and if that addition results in a carry-out from the most significant bit of the number, discard the carry-out.

Examination of the fifth column of Table above shows that in the two's complement representation, the leftmost bit is again 0 for positive numbers and is 1 for negative numbers. However, this number format does not have the unfortunate characteristic of signed-magnitude and one's complement representations: it has only one representation for zero. To see that this is true, consider forming the negative of $(+0)_{10}$, which has the bit pattern: $(+0)_{10} = (00000000)_2$

Forming the one's complement of $(00000000)_2$ produces $(11111111)_2$ and adding

1 to it yields $(00000000)_2$, thus $(-0)_{10} = (00000000)_2$. The carry out of the leftmost position is discarded in two's complement addition (except when detecting an overflow condition). Since there is only one representation for 0, and since all bit patterns are valid, there are $2^8 = 256$ different numbers that can be represented.

Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, this time using the two's complement representation. Starting with $(+12)_{10} = (00001100)_2$, complement, or negate the number, producing $(11110011)_2$.
Now add 1, producing $(11110100)_2$, and thus $(-12)_{10} = (11110100)_2$:

$(+12)_{10} = (00001100)_2$

$(-12)_{10} = (11110100)_2$

There is an equal number of positive and negative numbers provided zero is considered to be a positive number, which is reasonable because its sign bit is 0. The positive numbers start at 0, but the negative numbers start at −1, and so the magnitude of the most negative number is one greater than the magnitude of the most positive number. The positive number with the largest magnitude is +127, and the negative number with the largest magnitude is −128. There is thus no positive number that can be represented that corresponds to the negative of −128. If we try to form the two's complement negative of −128, then we will arrive at a negative number, as shown below:

$(-128)_{10} = (10000000)_2$
$(-128)_{10} = (01111111$
$(-128)_{10} + (+0000001)2$
$(-128)_{10}$ ——————)2
$(-128)10 = (10000000)_2$

The two's complement representation is the representation most commonly used in conventional computers.

- Excess Representation

In the excess or biased representation, the number is treated as unsigned, but is "shifted" in value by subtracting the bias from it. The concept is to assign the smallest numerical bit pattern, all zeros, to the negative of the bias, and assign the remaining numbers in sequence as the bit patterns increase in magnitude. A convenient way to think of an excess representation is that a number is represented as the sum of its two's complement form and another number, which is known as the "excess," or "bias." Once again, refer to Table 2.1, the rightmost column, for examples.

Consider again representing $(+12)10$ and $(-12)10$ in an eight-bit format but now using an excess 128 representation. An excess 128 number is formed by adding 128 to the original number, and then creating the unsigned binary version. For $(+12)_{10}$, we compute $(128 + 12 = 140)_{10}$ and produce the bit pattern $(10001100)_2$. For $(-12)_{10}$, we compute $(128 + -12 = 116)_{10}$ and produce the bit pattern $(01110100)_2$

$(+12)_{10} = (10001100)_2$

$(-12)_{10} = (01110100)_2$

Note that there is no numerical significance to the excess value: it simply has the effect of shifting the representation of the two's complement numbers.

There is only one excess representation for 0, since the excess representation is simply a shifted version of the two's complement representation. For the previous case, the excess value is chosen to have the same bit pattern as the largest negative number, which has the effect of making the numbers appear in numerically sorted order if the numbers are viewed in an unsigned binary representation.

Thus, the most negative number is $(-128)_{10} = (00000000)_2$ and the most positive number is $(+127)_{10} = (11111111)_2$. This representation simplifies making comparisons between numbers, since the bit patterns for negative numbers have numerically smaller values than the bit patterns for positive numbers. This is important for representing the exponents of floating point numbers, in which exponents of two numbers are compared in order to make them equal for addition and subtraction.

choosing a bias:

The bias chosen is most often based on the number of bits (n) available for representing an integer. To get an approximate equal distribution of true values above and below 0, the bias should be $2^{(n-1)}$ or $2^{(n-1)} - 1$

Floating point representation

Floating point is a numerical representation system in which a string of digits represent a real number. The name floating point refers to the fact that the radix point (decimal point or more commonly in computers, binary point) can be placed anywhere relative to the digits within the string. A fixed point is of the form $a \times b^n$ where a is the fixed point part often referred to as the mantissa, or significand of the number b represents the base and n the exponent. Thus a floating point number can be characterized by a triple of numbers: sign, exponent, and significand.

- Normalization

A potential problem with representing floating point numbers is that the same number can be represented in different ways, which makes comparisons and arithmetic operations difficult. For example, consider the numerically equivalent forms shown below:

$3584.1 \times 10^0 = 3.5841 \times 10^3 = .35841 \times 10^4$.

In order to avoid multiple representations for the same number, floating point numbers are maintained in normalized form. That is, the radix point is shifted to the left or to the right and the exponent is adjusted accordingly until the radix point is to the left of the leftmost nonzero digit. So the rightmost number above is the normalized one. Unfortunately, the number zero cannot be represented in this scheme, so to represent zero an exception is made. The exception to this rule is that zero is represented as all 0's in the mantissa.

If the mantissa is represented as a binary, that is, base 2, number, and if the normalization condition is that there is a leading "1" in the normalized mantissa, then there is no need to store that "1" and in fact, most floating point formats do *not* store it. Rather, it is "chopped off " before packing up the number for storage, and it is restored when unpacking the number into exponent and mantissa. This results in having an additional bit of precision on the right of the number, due

to removing the bit on the left. This missing bit is referred to as the hidden bit, also known as a hidden 1.

For example, if the mantissa in a given format is 1.1010 after normalization, then the bit pattern that is stored is 1010—the left-most bit is truncated, or hidden.

Possible floating point format.

In order to choose a possible floating point format for a given computer, the programmer must take into consideration the following:

The number of words used (i.e. the total number of bits used.)
The representation of the mantissa (2s complement etc.)
The representation of the exponent (biased etc.)
The total number of bits devoted for the mantissa and the exponent
The location of the mantissa (exponent first or mantissa first)

Because of the five points above, the number of ways in which a floating point number may be represented is legion. Many representation use the format of sign bit to represent a floating point where the leading bit represents the sign

Sign	Exponent	Mantissa

- The IEEE standard for floating point

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point format arithmetic in mini and micro computers.(i.e. ANSI/IEEE 754-1985). This standard specifies how single precision (32 bit) , double precision (64 bit) and Quadruple (128 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

General layout

The three fields in an IEEE 754 float

Sign	Exponent	Fraction

Binary floating-point numbers are stored in a sign-magnitude form where the most significant bit is the sign bit, exponent is the biased exponent, and "fraction" is the significand without the most significant bit.

Exponent biasing

The exponent is biased by $(2^{e-1}) - 1$, where e is the number of bits used for the exponent field (e.g. if e=8, then $(2^{8-1}) - 1 = 128 - 1 = 127$). Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this, the exponent is biased before being stored by adjusting its value to put it within an unsigned range suitable for comparison.

For example, to represent a number which has exponent of 17 in an exponent field 8 bits wide:

$exponent field = 17 + (2^{8-1}) - 1 = 17 + 128 - 1 = 144.$

Single Precision

The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the

final 23 bits are the fraction 'F':

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1        8 9                   31
```

To convert decimal 17.15 to IEEE Single format:

Convert decimal 17 to binary 10001. Convert decimal 0.15 to the repeating binary fraction 0.001001 Combine integer and fraction to obtain binary 10001.001001 Normalize the binary number to obtain 1.0001001001×2^4 Thus, $M = m-1 = 0001001001$ and $E = e+127 = 131 = 10000011$.

The number is positive, so S=0.   Align the values for M, E, and S in the correct fields.

```
0   10000011    00010010011001100110011
```

Note that if the exponent does not use all the field allocated to it, there will be leading 0's while for the mantissa, the zero's will be filled at the end.

Double Precision

The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F':

```
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
1           11 12
```

Quad Precision

The IEEE Quad precision floating point standard representation requires a 128 bit word, which may be represented as numbered from 0 to 127, left to right. The first bit is the sign bit, S, the next fifteen bits are the exponent bits, 'E', and the final 128 bits are the fraction 'F':

```
S EEEEEEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1             15 16
```

	Single	Double	Quadruple
No. of sign bit	1	1	1
No. of exponent bit	8	11	15
No. of fraction	23	52	111
Total bits used	32	64	128
Bias	127	1023	16383

Table2 Basic IEEE floating point format

Binary code

Internally, digital computers operate on binary numbers. When interfacing to humans, digital processors, e.g. pocket calculators, communication is decimal-based. Input is done in decimal then converted to binary for internal processing.

For output, the result has to be converted from its internal binary representation to a decimal form. Digital system represents and manipulates not only binary number but also many other discrete elements of information.

-Binary coded Decimal

In computing and electronic systems, binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding. It occupies more space than a pure binary representation**.**In BCD, a digit is usually represented by four bits which, in general, represent the values/digits/characters 0-9

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

```
Decimal:    0     1     2     3     4     5     6     7     8     9

BCD:      0000  0001  0010  0011  0100  0101  0110  0111  1000  1001
```

Thus, the BCD encoding for the number 127 would be:

```
 0001 0010 0111
```

The position weights of the BCD code are 8, 4, 2, 1. Other codes (shown in the table) use position weights of 8, 4, -2, -1 and 2, 4, 2, 1.

An example of a non-weighted code is the excess-3 code where digit codes is obtained from

their binary equivalent after adding 3. Thus the code of a decimal 0 is 0011, that of 6 is 1001, etc

Decimal Digit	8 4 2 1 Code	8 4 -2 -1 code	2 4 2 1 code	Excess-3 code
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 1 1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 1 0	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 0 1	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 0 0	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 1 1	1 0 1 1	1 0 0 0
6	0 1 1 0	1 0 1 0	1 1 0 0	1 0 0 1
7	0 1 1 1	1 0 0 1	1 1 0 1	1 0 1 0
8	1 0 0 0	1 0 0 0	1 1 1 0	1 0 1 1
9	1 0 0 1	1 1 1 1	1 1 1 1	1 1 0 0

it is very important to understand the difference between the conversion of a decimal number to binary and the binary coding of a decimal number. In each case, the final result is a series of bits. The bits obtained from conversion are binary digit. Bits obtained from coding are combinations of 1's and 0's arranged according to the rule of the code used. e.g. the binary conversion of 13 is 1101; the BCD coding of 13 is 00010011

- **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

Three-Bit Message			**Odd Parity Bit**	**Even Parity Bit**
X	**Y**	**Z**	**P**	**P**
0	0	0	1	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	0	1

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

Gray Code	Decimal Equivalent
0 0 0 0	0
0 0 0 1	1
0 0 1 1	2
0 0 1 0	3
0 1 1 0	4
0 1 1 1	5
0 1 0 1	6
0 1 0 0	7
1 1 0 0	8
1 1 0 1	9
1 1 1 1	10
1 1 1 0	11
1 0 1 0	12
1 0 1 1	13
1 0 0 1	14
1 0 0 0	15

Character Representation

Even though many people used to think of computers as "number crunchers", people figured out long ago that it's just as important to handle character data.

Character data isn't just alphabetic characters, but also numeric characters, punctuation, spaces, etc. Most keys on the central part of the keyboard (except shift, caps lock) are characters. Characters need to represented. In particular, they need to be represented in binary. After all, computers store and manipulate 0's and 1's (and even those 0's and 1's are just

abstractions. The implementation is typically voltages).

Unsigned binary and two's complement are used to represent unsigned and signed integer respectively, because they have nice mathematical properties, in particular, you can add and subtract as you'd expect.

However, there aren't such properties for character data, so assigning binary codes for characters is somewhat arbitrary. The most common character representation is ASCII, which stands for *American Standard Code for Information Interchange*.

There are two reasons to use ASCII. First, we need some way to represent characters as binary numbers (or, equivalently, as bitstring patterns). There's not much choice about this since computers represent everything in binary.

If you've noticed a common theme, it's that we need representation schemes for everything. However, most importantly, we need representations for numbers and characters. Once you have that (and perhaps pointers), you can build up everything you need.

The other reason we use ASCII is because of the letter "S" in ASCII, which stands for "standard". Standards are good because they allow for common formats that everyone can agree on.

Unfortunately, there's also the letter "A", which stands for American. ASCII is clearly biased for the English language character set. Other languages may have their own character set, even though English dominates most of the computing world (at least, programming and software).

Even though character sets don't have mathematical properties, there are some nice aspects about ASCII. In particular, the lowercase letters are contiguous ('a' through 'z' maps to 97_{10} through 122_{10}). The upper case letters are also contiguous ('A' through 'Z' maps to 65_{10} through 90_{10}). Finally, the digits are contiguous ('0' through '9' maps to 48_{10} through 57_{10}).

Since they are contiguous, it's usually easy to determine whether a character is lowercase or uppercase (by checking if the ASCII code lies in the range of lower or uppercase ASCII codes), or to determine if it's a digit, or to convert a digit in ASCII to an integer value.

ASCII Code (Decimal)

```
 0 nul   16 dle   32 sp    48 0     64 @     80 P     96 `     112 p
 1 soh   17 dc1   33 !     49 1     65 A     81 Q     97 a     113 q
 2 stx   18 dc2   34 "     50 2     66 B     82 R     98 b     114 r
 3 etx   19 dc3   35 #     51 3     67 C     83 S     99 c     115 s
 4 eot   20 dc4   36 $     52 4     68 D     84 T     100 d    116 t
 5 enq   21 nak   37 %     53 5     69 E     85 U     101 e    117 u
 6 ack   22 syn   38 &     54 6     70 F     86 V     102 f    118 v
 7 bel   23 etb   39 '     55 7     71 G     87 W     103 g    119 w
 8 bs    24 can   40 (     56 8     72 H     88 X     104 h    120 x
 9 ht    25 em    41 )     57 9     73 I     89 Y     105 i    121 y
10 nl    26 sub   42 *     58 :     74 J     90 Z     106 j    122 z
11 vt    27 esc   43 +     59 ;     75 K     91 [     107 k    123 {
12 np    28 fs    44 ,     60 <     76 L     92 \     108 l    124 |
13 cr    29 gs    45 -     61 =     77 M     93 ]     109 m    125 }
14 so    30 rs    46 .     62 >     78 N     94 ^     110 n    126 ~
15 si    31 us    47 /     63 ?     79 O     95 _     111 o    127 del
```

The characters between 0 and 31 are generally not printable (control characters, etc). 32 is the space character. Also note that there are only 128 ASCII characters. This means only 7 bits are required to represent an ASCII character. However, since the smallest size representation on most computers is a byte, a byte is used to store an ASCII character. The Most Significant bit(MSB) of an ASCII character is 0.

ASCII Code (Hex)

```
00 nul   10 dle   20 sp    30 0     40 @     50 P     60 `     70 p
01 soh   11 dc1   21 !     31 1     41 A     51 Q     61 a     71 q
02 stx   12 dc2   22 "     32 2     42 B     52 R     62 b     72 r
03 etx   13 dc3   23 #     33 3     43 C     53 S     63 c     73 s
04 eot   14 dc4   24 $     34 4     44 D     54 T     64 d     74 t
05 enq   15 nak   25 %     35 5     45 E     55 U     65 e     75 u
06 ack   16 syn   26 &     36 6     46 F     56 V     66 f     76 v
07 bel   17 etb   27 '     37 7     47 G     57 W     67 g     77 w
08 bs    18 can   28 (     38 8     48 H     58 X     68 h     78 x
09 ht    19 em    29 )     39 9     49 I     59 Y     69 i     79 y
0a nl    1a sub   2a *     3a :     4a J     5a Z     6a j     7a z
0b vt    1b esc   2b +     3b ;     4b K     5b [     6b k     7b {
0c np    1c fs    2c ,     3c <     4c L     5c \     6c l     7c |
0d cr    1d gs    2d -     3d =     4d M     5d ]     6d m     7d }
0e so    1e rs    2e .     3e >     4e N     5e ^     6e n     7e ~
0f si    1f us    2f /     3f ?     4f O     5f _     6f o     7f del
```

The difference in the ASCII code between an uppercase letter and its corresponding lowercase letter is 20_{16}. This makes it easy to convert lower to uppercase (and back) in hex (or binary).

Other Character Codes

While ASCII is still popularly used, another character representation that was used (especially at IBM) was EBCDIC, which stands for *Extended Binary Coded Decimal Interchange Code* (yes, the word "code" appears twice). This character set has mostly disappeared. EBCDIC does not store characters contiguously, so this can create problems alphabetizing "words".

One problem with ASCII is that it's biased to the English language. This generally creates some problems. One common solution is for people in other countries to write programs in ASCII.

Other countries have used different solutions, in particular, using 8 bits to represent their alphabets, giving up to 256 letters, which is plenty for most alphabet based languages (recall you also need to represent digits, punctuation, etc).

However, Asian languages, which are word-based, rather than character-based, often have more words than 8 bits can represent. In particular, 8 bits can only represent 256 words, which is far smaller than the number of words in natural languages.

Thus, a new character set called Unicode is now becoming more prevalent. This is a 16 bit code, which allows for about 65,000 different representations. This is enough to encode the popular Asian languages (Chinese, Korean, Japanese, etc.). It also turns out that ASCII codes are preserved. What does this mean? To convert ASCII to Unicode, take all one byte ASCII codes, and zero-extend them to 16 bits. That should be the Unicode version of the ASCII characters.

The biggest consequence of using Unicode from ASCII is that text files double in size. The second consequence is that endianness begins to matter again. Endianness is the ordering of individually addressable sub-units (words, bytes, or even bits) within a longer data word stored in external memory. The most typical cases are the ordering of bytes within a 16-, 32-, or 64-bit word, where endianness is often simply referred to as byte order. The usual contrast is between most versus least significant byte first, called big-endian and little-endian respectively.

Big-endian places the most significant bit, digit, or byte in the first, or leftmost, position. Little-endian places the most significant bit, digit, or byte in the last, or rightmost, position. Motorola processors employ the big-endian approach, whereas Intel processors take the little-endian approach. Table bellow illustrates how the decimal value 47,572 would be expressed in hexadecimal and binary notation (two octets) and how it would be stored using these two methods.

Table : Endianess

Number Hexadecimal	Big-Endian	Little-Endian
B9D4	B9D4	4D9B
Binary		
10111001	10111001	11010100
11010100	11010100	10111001

With single bytes, there's no need to worry about endianness. However, you have to consider that with two byte quantities.

While C and C++ still primarily use ASCII, Java has already used Unicode. This means that Java must create a byte type, because char in Java is no longer a single byte. Instead, it's a 2 byte Unicode representation.

Exercise

1. The state of a 12-bit register is 010110010111. what is its content if it represents :
      i. Decimal digits in BCD code   ii.   Decimal digits in excess-3 code

      iii. Decimal digits in 8 4-2-1 code   iv. Natural binary number

2. The result of an experiment fall in the range -4 to +6. A scientist wishes to read the result into a computer and then process them. He decides to use a 4-bit binary code to represents each of the possible inputs. Devise a 4-bit binary code of representing numbers in the range of -4 to 6

3. The (r-1)'s complement of a base-6 numbers is called the 5's complement. Explain the procedure for obtaining the 5's complement of base 6 numbers. Obtain the 5's complement of $(3210)_6$

4. Design a three bit code to represent each of the six digits of the base 6 number system.

5. Represent the decimal number –234.125 using the IEEE 32- bit (single) format

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 $x' = \overline{x}$ =z is interpreted as NOT x =z meaning that z is what x is not

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 gatesand 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.

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design
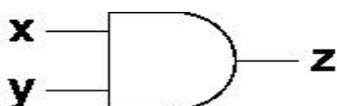
The truth table for an *and*-gate with two inputs looks like this:

```
x y | z
-------
0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1
```

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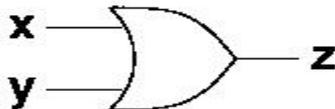
The truth table for an *or*-gate with two inputs looks like this:

```
x y | z
-------
0 0 | 0
0 1 | 1
1 0 | 1
1 1 | 1
```

As with the AND function, the OR function can have any number of inputs. However, practical commercial OR gates are mostly limited to 2, 3, and 4 inputs, as with AND gates.

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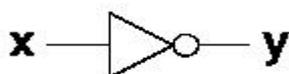
The truth table for an *inverter* looks like this:

Digital System Design   **17EE35**

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| Digital System Design |

```
x | y
-----
0 | 1
1 | 0
```

The NOT function, as it is called, is necesasary in many applications and highly useful in others. A practical verbal application might be:

> The door is NOT locked = You may enter

In the inverter symbol, the triangle actually denotes only an amplifier, which in digital terms means that it "cleans up" the signal but does not change its logical sense. It is the circle at the output which denotes the logical inversion. The circle could have been placed at the input instead, and the logical meaning would still be the same

Combined gates

Sometimes, it is practical to combine functions of the basic gates into more complex gates, for instance in order to save space in circuit diagrams. In this section, we show some such combined gates together with their truth tables.

- **The *nand*-gate**

The *nand*-gate is an *and*-gate with an inverter on the output. So instead of drawing several gates like this:
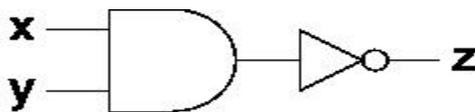
We draw a single *and*-gate with a little ring on the output like this:
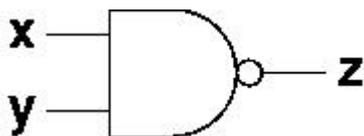
The *nand*-gate, like the *and*-gate can take an arbitrary number of inputs.

The truth table for the *nand*-gate is like the one for the *and*-gate, except that all output values have been inverted:

```
x y | z
-------
0 0 | 1
0 1 | 1
1 0 | 1
1 1 | 0
```

The truth table clearly shows that the NAND operation is the complement of the AND

- **The *nor*-gate**

The *nor*-gate is an *or*-gate with an inverter on the output. So instead of drawing several gates like this:

17EE35
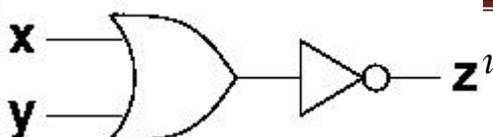
We draw a single *or*-gate with a little ring on the output like this:
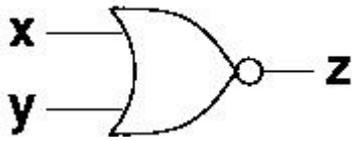
The *nor*-gate, like the *or*-gate can take an arbitrary number of inputs.

The truth table for the *nor*-gate is like the one for the *or*-gate, except that all output values have been inverted:

```
x y | z
-------
0 0 | 1
0 1 | 0
1 0 | 0
1 1 | 0
```

- **The *exclusive-or*-gate**

The *exclusive-or*-gate is similar to an *or*-gate. It can have an arbitrary number of inputs, and its output value is 1 if and only if *exactly one input* is 1 (and thus the others 0). Otherwise, the output is 0.

We draw an *exclusive-or*-gate like this:

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

```
x   y | z
-------
0   0 | 0
0   1 | 1
1   0 | 1
1   1 | 0
```

- **The *exclusive-Nor*-gate**

The *exclusive-Nor*-gate is similar to an N *or*-gate. It can have an arbitrary number of inputs, and its output value is 1 if and only if the two *input are of the same values* (1 and 1 or 0 and 0). Otherwise, the output is 0.

We draw an *exclusive-Nor*-gate like this:

em Design   17EE35

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical

Academic

Notes

Digital System Design
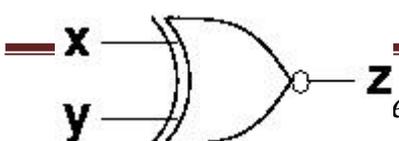
The truth table for an *exclusive-nor*-gate with two inputs looks like this:

```
x y | z
-------
0  0 | 1
0  1 | 0
1  0 | 0
1  1 | 1
```

Let us limit ourselves to gates with *n* inputs. The truth tables for such gates have 2^n lines. Such a gate is completely defined by the output column in the truth table. The output column can be viewed as a string of 2^n binary digits. How many different strings of binary digits of length 2^n are there? The answer is 2^{2n}, since there are 2^k different strings of *k* binary digits, and if $k=2^n$, then there are 2^{2n} such strings. In particular, if *n*=2, we can see that there are 16 different types of gates with 2 inputs.

Families of logic gates

There are several different families of logic gates. Each family has its capabilities and limitations, its advantages and disadvantages. The following list describes the main logic families and their characteristics. You can follow the links to see the circuit construction of gates of each family.

- **Diode Logic (DL)**

Diode logic gates use diodes to perform AND and OR logic functions. Diodes have the property of easily passing an electrical current in one direction, but not the other. Thus, diodes can act as a logical switch.

Diode logic gates are very simple and inexpensive, and can be used effectively in specific situations. However, they cannot be used extensively, as they tend to degrade digital signals rapidly. In addition, they cannot perform a NOT function, so their usefulness is quite limited.

- **Resistor-Transistor Logic (RTL)**

Resistor-transistor logic gates use Transistors to combine multiple input signals, which also amplify and invert the resulting combined signal. Often an additional transistor is included to re-invert the output signal. This combination provides clean output signals and either inversion or non-inversion as needed.

RTL gates are almost as simple as DL gates, and remain inexpensive. They also are handy because both normal and inverted signals are often available. However, they do draw a significant amount of current from the power supply for each gate. Another limitation is that RTL gates cannot switch at the high speeds used by today's computers, although they are still useful in slower applications.

Although they are not designed for linear operation, RTL integrated circuits are sometimes used as inexpensive small-signal amplifiers, or as interface devices between linear and digital circuits.

- **Diode-Transistor Logic (DTL)**

By letting diodes perform the logical AND or OR function and then amplifying the result with a transistor, we can avoid some of the limitations of RTL. DTL takes diode logic gates and adds a transistor to the output, in order to provide logic inversion and to restore the signal to full logic levels.

- **Transistor-Transistor Logic (TTL)**

The physical construction of integrated circuits made it more effective to replace all the input diodes in a DTL gate with a transistor, built with multiple emitters. The result is transistor-transistor logic, which became the standard logic circuit

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

in most applications for a number of years.

As the state of the art improved, TTL integrated circuits were adapted slightly to handle a wider range of requirements, but their basic functions remained the same. These devices comprise the 7400 family of digital ICs.

- Emitter-Coupled Logic (ECL)

Also known as Current Mode Logic (CML), ECL gates are specifically designed to operate at extremely high speeds, by avoiding the "lag" inherent when transistors are allowed to become saturated. Because of this, however, these gates demand substantial amounts of electrical current to operate correctly.

- CMOS Logic

One factor is common to all of the logic families we have listed above: they use significant amounts of electrical power. Many applications, especially portable, battery-powered ones, require that the use of power be absolutely minimized. To accomplish this, the CMOS (Complementary Metal-Oxide-Semiconductor) logic family was developed. This family uses enhancement-mode MOSFETs as its transistors, and is so designed that it requires almost no current to operate.

CMOS gates are, however, severely limited in their speed of operation. Nevertheless, they are highly useful and effective in a wide range of battery-powered applications.

Most logic families share a common characteristic: their inputs require a certain amount of current in order to operate correctly. CMOS gates work a bit differently, but still represent a capacitance that must be charged or discharged when the input changes state. The current required to drive any input must come from the output supplying the logic signal. Therefore, we need to know how much current an input requires, and how much current an output can reliably supply, in order to determine how many inputs may be connected to a single output.

However, making such calculations can be tedious, and can bog down logic circuit design. Therefore, we use a different technique. Rather than working constantly with actual currents, we determine the amount of current required to drive one standard input, and designate that as a standard load on any output. Now we can define the number of standard loads a given output can drive, and identify it that way. Unfortunately, some inputs for specialized circuits require more than the usual input current, and some gates, known as buffers, are deliberately designed to be able to drive more inputs than usual. For an easy way to define input current requirements and output drive capabilities, we define two new terms: fan-in and fan-out

Fan-in

Fan-in is a term that defines the maximum number of digital inputs that a single logic gate can accept. Most transistor-transistor logic (TTL) gates have one or two inputs, although some have more than two. A typical logic gate has a fan-in of 1 or 2.

In some digital systems, it is necessary for a single TTL logic gate to drive several devices with fan-in numbers greater than 1. If the total number of inputs a transistor-transistor logic (TTL) device must drive is greater than 10, a device called a buffer can be used between the TTL gate output and the inputs of the devices it must drive. A logical inverter (also called a NOT gate) can serve this function in most digital circuits.

Fan-out

Fan-out is a term that defines the maximum number of digital inputs that the output of a single logic gate can feed. Most transistor-transistor logic (TTL) gates can feed up to 10 other digital gates or devices. Thus, a typical TTL gate has a fan-out of 10.

In some digital systems, it is necessary for a single TTL logic gate to drive more than 10 other gates or devices. When this is the case, a device called a buffer can be used between the TTL gate and the multiple devices it must drive. A buffer of this type has a fan-out of 25 to 30. A logical inverter (also called a NOT gate) can serve this function in most digital circuits.

Remember, fan-in and fan-out apply directly only within a given logic family. If for any reason you need to interface between two different logic families, be careful to note and meet the drive requirements and limitations of both families, within the interface circuitry

Boolean Algebra

One of the primary requirements when dealing with digital circuits is to find ways to make them as simple as possible. This constantly requires that complex logical expressions be reduced to simpler expressions that nevertheless produce the same results under all possible conditions. The simpler expression can then be implemented with a smaller, simpler circuit, which in turn saves the price of the unnecessary gates, reduces the number of gates needed, and reduces the power and the amount of space required by those gates.

One tool to reduce logical expressions is the mathematics of logical expressions, introduced by George Boole in 1854 and known today as *Boolean Algebra*. The rules of Boolean Algebra are simple and straight-forward, and can be applied to any logical expression. The resulting reduced expression can then be readily tested with a Truth Table, to verify that the reduction was valid.

Boolean algebra is an algebraic structure defined on a set of elements B, together with two binary operators(+, **.**) provided the following postulates are satisfied.

1. Closure with respect to operator + and Closure with respect to operator **.**
2. An identity element with respect to + designated by 0: X+0= 0+X=X

   An identity element with respect to . designated by 1: X.1= 1.X=X

3. Commutative with respect to +: X=Y=Y+X

   Commutative with respect to **.**: X.Y=Y.X

4. . distributive over +: X.(Y+Z)=X.Y+X.Z

   + distributive over .: X+(Y.Z)=(X+Y).(X+Z)

5. For every element x belonging to B, there exist an element x′ or $\bar{x}$ called the complement of x such that x. x′=0 and x+ x′=1
6. There exists at least two elements x,y belonging to B such that x≠y

The two valued Boolean algebra is defined on a set B={0,1} with two binary operators + and**.**

X	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	x+y
0	0	0
0	1	1
1	0	1
1	1	0

x	x′
0	1
1	0

Closure. from the tables, the result of each operation is either 0 or 1 and 1 ,0 belongs to B

Identity. From the truth table we see that 0 is the identity element for + and 1 is the identity element for **.**.

Commutative law is obvious from the symmetry of binary operators table.

Distributive Law. x.(y+z)=x.y+x.z

x	y	z	y+z	x.(y+z)	x.y	x.z	x.y+x.z
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Distributive of + over . can be shown as in the truth table above

From the complement table we can see that x+ x′=1 i.e 1+0=1 and x. x′=0 i.e 1.0=0

Principle of duality of Boolean algebra

The principle of duality state that every algebraic expression which can be deduced from the postulates of Boolean algebra remains valid if the operators and the identity elements are interchanged. This mean that the dual of an expression is obtained changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa

Laws of Boolean Algebra

Postulate 2 :

(a) *0 + A = A*   (b) *1.A = A*

Postulate 5 :

(a) *A + A′ =1*   (b) *A. A′=0*

Theorem1 : Identity Law

(a) A + A = A   (b) *A A = A*

Theorem2

(a) *1 + A = 1*   (b) *0. A = 0*

Theorem3: involution

A″=A

Combinational circuit

Introduction

The combinational circuit consist of logic gates whose outputs at any time is determined directly from the present combination of input without any regard to the previous input. A combinational circuit performs a specific information processing operation fully specified logically by a set of Boolean functions.

A *combinatorial circuit* is a generalized gate. In general such a circuit has *m* inputs and *n* outputs. Such a circuit can always be constructed as *n* separate combinatorial circuits, each with exactly one output. For that reason, some texts only discuss combinatorial circuits with exactly one output. In reality, however, some important *sharing of intermediate signals* may take place if the entire *n*-output circuit is constructed at once. Such sharing can significantly reduce the number of gates required to build the circuit.

When we build a combinatorial circuit from some kind of specification, we always try to make it *as good as possible*. The only problem is that the definition of "as good as possible" may vary greatly. In some applications, we simply want to minimize the number of gates (or the number of transistors, really). In other, we might be interested in as short a *delay* (the time it takes a signal to traverse the circuit) as possible, or in as low *power consumption* as possible. In general, a mixture of such criteria must be applied.

Describing existing circuits using Truth tables

To specify the exact way in which a combinatorial circuit works, we might use different methods, such as logical expressions or *truth tables*.

A truth table is a complete enumeration of all possible combinations of input values, each one with its associated output value.

When used to describe an existing circuit, output values are (of course) either 0 or 1. Suppose for instance that we wish to make a truth table for the following circuit:
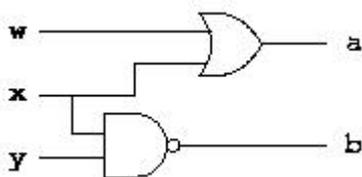
All we need to do to establish a truth table for this circuit is to compute the output value for the circuit for each possible combination of input values. We obtain the following truth table:

```
w x y | a b
-----------
0 0 0 | 0 1
0 0 1 | 0 1
0 1 0 | 1 1
0 1 1 | 1 0
1 0 0 | 1 1
1 0 1 | 1 1
1 1 0 | 1 1
1 1 1 | 1 0
```

Specifying circuits to build

When used as a *specification* for a circuit, a table may have some output values that are not specified, perhaps because the corresponding combination of input values can never occur in the particular application. We can indicate such unspecified output values with a dash -.

For instance, let us suppose we want a circuit of four inputs, interpreted as two nonnegative binary integers of two binary digits each, and two outputs, interpreted as the nonnegative binary integer giving the quotient between the two input numbers. Since division is not defined when the denominator is zero, we do not care what the output value is in this case. Of the sixteen entries in the truth table, four have a zero denominator. Here is the truth table:

```
x1 x0 y1 y0 | z1 z0
-------------------
 0  0  0  0 |-  -
 0  0  0  1 |0  0
 0  0  1  0 |0  0
 0  0  1  1 |0  0
 0  1  0  0 |-  -
 0  1  0  1 |0  1
 0  1  1  0 |0  0
 0  1  1  1 |0  0
 1  0  0  0 |-  -
 1  0  0  1 |1  0
 1  0  1  0 |0  1
 1  0  1  1 |0  0
 1  1  0  0 |-  -
 1  1  0  1 |1  1
 1  1  1  0 |0  1
 1  1  1  1 |0  1
```

Unspecified output values like this can greatly decrease the number of circuits necessary to build the circuit. The reason is simple: when we are free to choose the output value in a particular situation, we choose the one that gives the fewest total number of gates.

Circuit minimization is a difficult problem from complexity point of view. Computer programs that try to optimize circuit design apply a number of *heuristics* to improve speed. In this course, we are not concerned with optimality. We are therefore only going to discuss a simple method that works for all possible combinatorial circuits (but that can waste large numbers of gates).

A separate single-output circuit is built for each output of the combinatorial circuit.

Our simple method starts with the truth table (or rather *one of the acceptable truth tables*, in case we have a choice). Our circuit is going to be a two-layer circuit. The first layer of the circuit will have at most 2^n AND-gates, each with n inputs (where n is the number of inputs of the combinatorial circuit). The second layer will have a single OR-gate with as many inputs as there are gates in the first layer. For each line of the truth table with an output value of 1, we put down a AND-gate with n inputs. For each input entry in the table with a 1 in it, we connect an input of the AND-gate to the corresponding input. For each entry in the table with a 0 in it, we connect an input of the AND-gate to the corresponding input *inverted*.

The output of each AND-gate of the fist layer is then connected to an input of the OR-gate of the second layer.

As an example of our general method, consider the following truth table (where a - indicates that we don't care what value is chosen):

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

```
x y z | a b
-----------
0 0 0 | - 0
0 0 1 | 1 1
0 1 0 | 1 -
0 1 1 | 0 0
1 0 0 | 0 1
1 0 1 | 0 -
1 1 0 | - -
1 1 1 | 1 0
```

The first step is to arbitrarily choose values for the undefined outputs. With out simple method, the best solution is to choose a 0 for each such undefined output. We get this table:

```
x y z | a b
-----------
0 0 0 | 0 0
0 0 1 | 1 1
0 1 0 | 1 0
0 1 1 | 0 0
1 0 0 | 0 1
1 0 1 | 0 0
1 1 0 | 0 0
1 1 1 | 1 0
```

Now, we have to build two separate single-output circuits, one for the a column and one for the b column.

$A = x'y'z + x'yz' + xyz$

$B = x'y'z + xy'z'$

For the first column, we get three 3-input *AND*-gates in the first layer, and a 3-input *OR*-gate in the second layer. We get three *AND* -gates since there are three rows in the a column with a value of 1. Each one has 3-inputs since there are three inputs, x, y, and z of the circuit. We get a 3-input *OR*-gate in the second layer since there are three *AND* -gates in the first layer.

Here is the complete circuit for the first column:
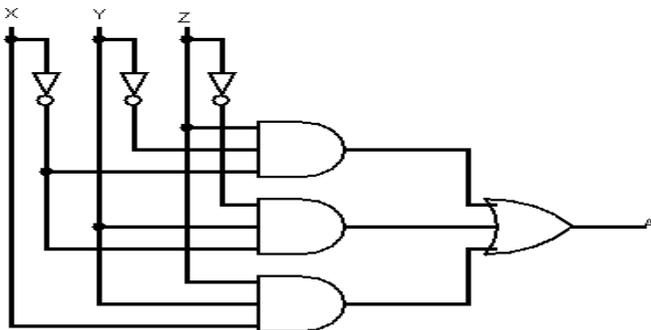
For the second column, we get two 3-input *AND* -gates in the first layer, and a 2-input *OR*-gate in the second layer. We get two *AND*-gates since there are two rows in the b column with a value of 1. Each one has 3-inputs since again there are three inputs, x, y, and z of the circuit. We get a 2-input *AND*-gate in the second layer since there are two *AND*-gates in the first layer.

Here is the complete circuit for the second column:
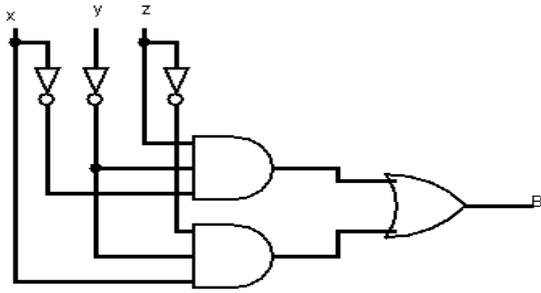
Now, all we have to do is to combine the two circuits into a single one:
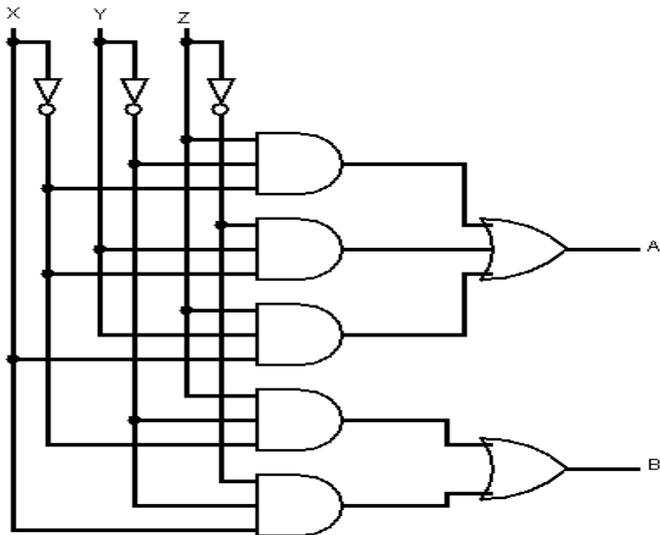
While this circuit works, it is not the one with the fewest number of gates. In fact, since both output columns have a 1 in the row correspoding to the inputs 0 0 1, it is clear that the gate for that row can be shared between the two subcircuits:
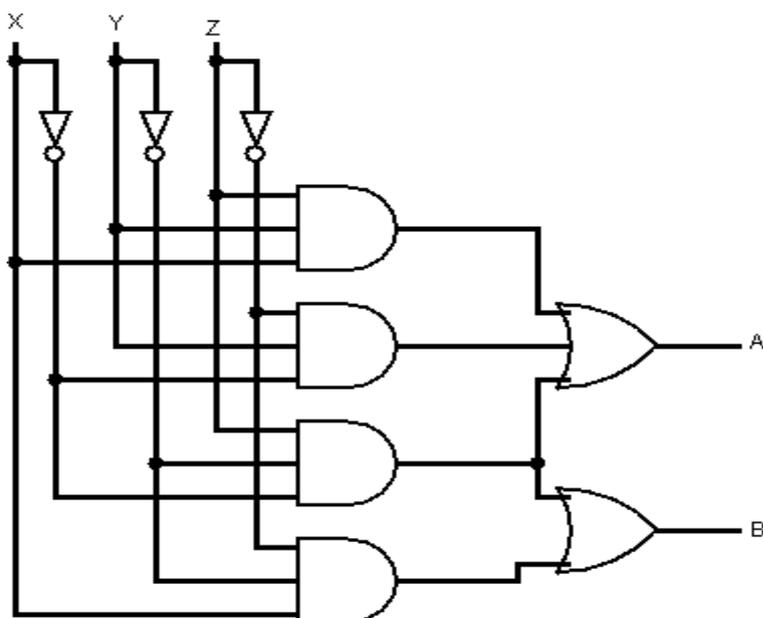
In some cases, even smaller circuits can be obtained, if one is willing to accept more layers (and thus a higher circuit delay).

Digital System Design   *17EE35*

Boolean functions

Operations of binary variables can be described by mean of appropriate mathematical function called Boolean function. A Boolean function define a mapping from a set of binary input values into a set of output values. A Boolean function is formed with binary variables, the binary operators AND and OR and the unary operator NOT.

For example , a Boolean function $f(x_1,x_2,x_3,\ldots,x_n) = y$ defines a mapping from an arbitrary combination of binary input values $(x_1,x_2,x_3,\ldots,x_n)$ into a binary value y. a binary function with n input variable can operate on 2^n distincts values. Any such function can be described by using a truth table consisting of 2^n rows and n columns. The content of this table are the values produced by that function when applied to all the possible combination of the n input variable.

Example

x	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

The function f, representing x.y, that is f(x,y)=xy. Which mean that f=1 if x=1 and y=1 and f=0 otherwise.

For each rows of the table, there is a value of the function equal to 1 or 0. The function f is equal to the sum of all rows that gives a value of 1.

A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gate. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate and each term is implemented with a logic gate . e.g.

F=xyz

F=x+y′z
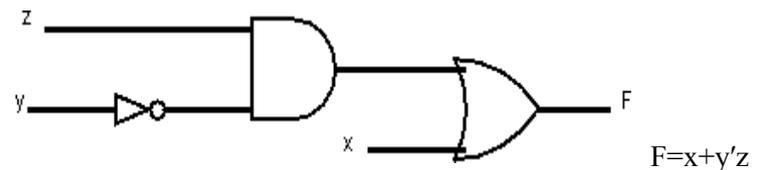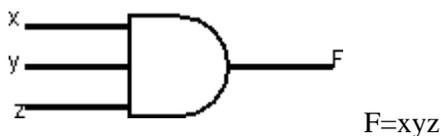
Complement of a function

The complement of a function F is F′ and is obtained from an interchange of 0's to 1's and 1's to 0's in the value of F. the complement of a function may be derived algebraically trough De Morgan's theorem

$(A+B+C+\ldots)' = A'B'C'\ldots$
$(ABC\ldots)' = A'+ B'+C'\ldots$

The generalized form of de Morgan's theorem state that the complement of function is obtained by interchanging AND and OR and complementing each literal.

F=X′YZ′+X′Y′Z′
F′=(X′YZ′+X′Y′Z′)′
       =(X′YZ′)′.(X′Y′Z′)′

Digital System Design   *17EE35*

$=(X''+Y'+Z'')(X''+Y''+Z'')$

$=(X+Y'+Z)(X+Y+Z)$

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

Canonical form(Minterns and Maxterms)

A binary variable may appear either in it normal form or in it complement form . consider two binary variables x and y combined with AND operation. Since each variable may appears in either form there are four possible combinations: $x'y'$, $x'y$, xy',xy. Each of the term represent one distinct area in the Venn diagram and is called minterm or a standard product. With n variable, 2^n minterms can be formed.

In a similar fashion, n variables forming an OR term provide 2^n possible combinations called maxterms or standard sum. Each maxterm is obtained from an OR term of the n variables, with each variable being primed if the corresponding bit is 1 and un-primed if the corresponding bit is 0. Note that each maxterm is the complement of its corresponding minterm and vice versa.

X	Y	Z	Minterm	maxterm
0	0	0	$x'y'z'$	$X+y+z$
0	0	1	$X'y'z$	$X+y+z'$
0	1	0	$X'yz'$	$X+y'+z$
0	1	1	$X'yz$	$X+y'+z'$
1	0	0	$Xy'z'$	$X'+y+z$
1	0	1	$Xy'z$	$X'+y+z'$
1	1	0	Xyz'	$X'+y'+z$
1	1	1	Xyz	$X'+y'+z'$

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of variable that produce a 1 and taken the OR of those terms.

Similarly, the same function can be obtained by forming the maxterm for each combination of variable that produces 0 and then taken the AND of those term.

It is sometime convenient to express the bolean function when it is in sum of minterms, in the following notation:

$F(X,Y,Z)=\sum(1,4,5,6,7)$. the summation symbol$\sum$ stands for the ORing of the terms; the number follow ing it are the minterms of the function. The letters in the parenthesis following F form list of the variables in the order taken when the minterm is converted to an AND term.

So, $F(X,Y,Z)=\sum(1,4,5,6,7) = X'Y'Z+XY'Z'+XY'Z+XYZ'+XYZ$

Sometime it is convenient to express a Boolean function in its sum of minterm. If it is not in that case, the expression is expanded into the sum of AND term and if there is any missing variable, it is ANDed with an expression such as $x+x'$ where x is one of the missing variable.

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This can be done by using distributive law $x+xz=(x+y)(x+z)$. then if there is any missing variable, say x in each OR term is ORded with xx'.

e.g. represent $F=xy+x'z$ as a product of maxterm

$=(xy +x')(xy+z)$
$(x+x')(y+x')(x+z)(y+z)$
$(y+x')(x+z)(y+z)$

Digital System Design   *17EE35*

Adding missing variable in each term

$(y+x')= x'+y+zz'$ $=(x'+y+z)(x'+y+z')$

$(x+z)= x+z+yy'$ $=(x+y+z)(x+y'+z)$

$(y+z)= y+z+xx'$ $=(x+y+z)(x'+y+z)$

$F= (x+y+z)(x+y'+z) (x'+y+z)(x'+y+z')$

A convenient way to express this function is as follow :

$F(x,y,z)= \prod (0,2,4,5)$

Standard form

Another way to express a boolean function is in satndard form. Here the term that form the function may contains one, two or nay number of literals. There are two types of standard form. The sum of product and the product of sum.

The sum of product(SOP) is a Boolean expression containing AND terms called product term of one or more literals each. The sum denotes the ORing of these terms

e.g. $F=x+xy'+x'yz$

the product of sum (POS)is a Boolean expression containing OR terms called SUM terms. Each term may have any number of literals. The product denotes the ANDing of these terms

e.g. $F= x(x+y')(x'+y+z)$

a boolean function may also be expressed in a non standard form. In that case, distributive law can be used to remove the parenthesis

$F=(xy+zw)(x'y'+z'w')$

$= xy(x'y'+z'w')+zw(x'y'+z'w')$

$=Xyx'y +xyz'w' +zwx'y' +zwz'w'$

$=xyz'w'+zwx'y'$

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only methods is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar

Describing existing circuits using Logic expressions

To define what a combinatorial circuit does, we can use a *logic expression* or an *expression* for short. Such an expression uses the two constants 0 and 1, variables such as *x*, *y*, and *z* (sometimes with suffixes) as names of inputs and outputs, and the operators +, **.** and a horizontal bar or a prime (which stands for *not*). As usual, multiplication is considered to have higher priority than addition. Parentheses are used to modify the priority.

If Boolean functions in either Sum of Product or Product of Sum forms can be implemented using 2-Level

Digital System Design   *17EE35*

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| Digital System Design |

implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

Examples:

$(X'+Y)(Y+XZ')'+X(YZ)'$

The equation is neither in sum of product nor in product of sum. The implementation is as follow
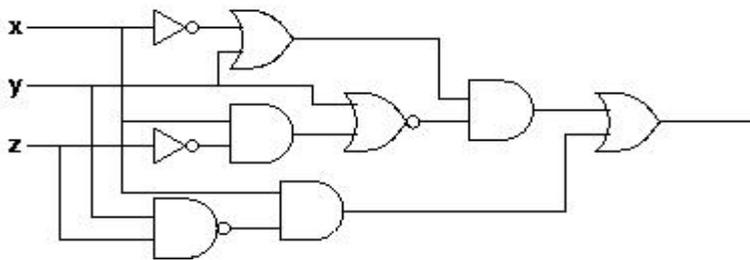
$X_1X_2'X_3+X_1'X_2'X_2+X_1'X_2X_3'$

The equation is in sum of product. The implementation is in 2-Levels. AND gates form the first level and a single OR gate the second level.
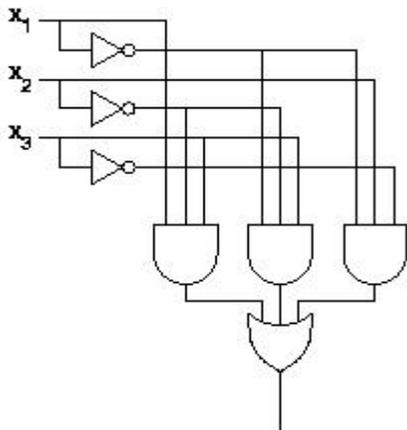
$(X+1)(Y+0Z)$

The equation is neither in sum of product nor in product of sum. The implementation is as follow

Digital System Design   *17EE35*
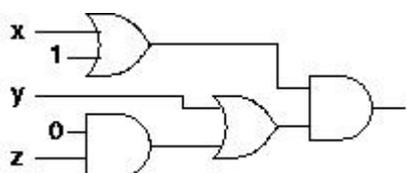
Power of logic expressions

A valid question is: *can logic expressions describe all possible combinatorial circuits?*. The answer is *yes* and here is why:

You can trivially convert the truth table for an arbitrary circuit into an expression. The expression will be in the form of a sum of products of variables and there inverses. Each row with output value of 1 of the truth table corresponds to one term in the sum. In such a term, a variable having a 1 in the truth table will be uninverted, and a variable having a 0 in the truth table will be inverted.

Take the following truth table for example:

```
 x y z | f
 ---------
 0 0 0 | 0
 0 0 1 | 0
 0 1 0 | 1
 0 1 1 | 0
 1 0 0 | 1
 1 0 1 | 0
 1 1 0 | 0
 1 1 1 | 1
```

The corresponding expression is:

$X'Y'Z + XY'Z' + XYZ$

Since you can describe any combinatorial circuit with a truth table, and you can describe any truth table with an expression, you can describe any combinatorial circuit with an expression.

Simplicity of logic expressions

There are many logic expressions (and therefore many circuits) that correspond to a certain truth table, and therefore to a certain function computed. For instance, the following two expressions compute the same function:

$X(Y+Z)$ $XY+XZ$

The left one requires two gates, one *and*-gate and one *or*-gate. The second expression requires two *and*-gates and one *or*-gate. It seems obvious that the first one is preferable to the second one. However, this is not always the case. It is not always true that the number of gates is the only way, nor even the best way, to determine simplicity.

We have, for instance, assumed that gates are ideal. In reality, the signal takes some time to propagate through a gate. We call this time the gate *delay*. We might be interested in circuits that minimize the total gate delay, in other words, circuits that make the signal traverse the fewest possible gates from input to output. Such circuits are not necessarily the same ones that require the smallest number of gates.

Circuit minimization

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, it can appear in many different forms when expressed algebraically.

Digital System Design   *17EE35*

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

Simplification through algebraic manipulation

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation as stated above. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only methods is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar

e.g. simplify $x+x'y$

$x+x'y=(x+x')(x+y)=x+y$
simplify $x'y'z+x'yz+xy'$
$x'y'z+x'yz+xy'=x'z(y+y')+xy'$
$=x'z+xy'$

Simplify $xy +x'z+yz$
$xy +x'z+yz= xy +x'z+yz(x+x')$
$xy +x'z+yzx+yzx'$
$xy(1+z) +x'z(1+y)$
$=xy+x'z$

Karnaugh map

The Karnaugh map also known as Veitch diagram or simply as K map is a two dimensional form of the truth table, drawn in such a way that the simplification of Boolean expression can be immediately be seen from the location of 1's in the map. The map is a diagram made up of squares , each sqare represent one minterm. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognised graphically in the map from the area enclosed by those squares whose minterms are included in the function.

 A two variable Boolean function can be represented as follow
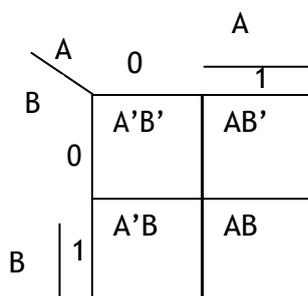
A three variable function can be represented as follow

Digital System Design   17EE35

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

A'B'C'	A'BC'	ABC'	AB'C'
A'B'C	A'BC	ABC	AB'C

C | 1
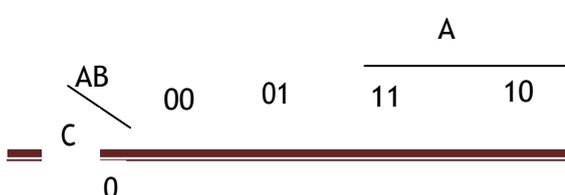
B

A four variable Boolean function can be represented in the map bellow

A

AB
00   01   11   10
CD

A'B'C'D'	A'BC'D'	ABC'D'	AB'C'D'
A'B'C'D	A'BC'D	ABC'D	AB'C'D
A'B'CD	A'BCD	ABCD	AB'CD
A'B'CD'	A'BCD'	ABCD'	AB'CD'

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

00

01

D

01

C

01

B
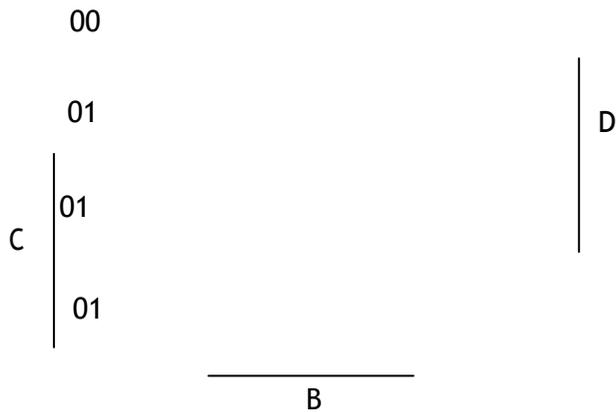
To simplify a Boolean function using karnaugh map, the first step is to plot all ones in the function truth table on the map. The next step is to combine adjacent 1's into a group of one, two, four, eight, sixteen. The group of minterm should be as large as possible. A single group of four minterm yields a simpler expression than two groups of two minterms.

In a four variable karnaugh map,

1 variable product term is obtained if 8 adjacent squares are covered
2 variable product term is obtained if 4 adjacent squares are covered
3 variable product term is obtained if 2 adjacent squares are covered
1 variable product term is obtained if 1 square is covered
A square having a 1 may belong to more than one term in the sum of product expression

The final stage is reached when each of the group of minterms are ORded together to form the simplified sum of product expression

The karnaugh map is not a square or rectangle as it may appear in the diagram. The top edge is adjacent to the bottom edge and the left hand edge adjacent to the right hand edge. Consequent, two squares in karnaugh map are said to be adjacent if they differ by only one variable

Implicant

In Boolean logic, an implicant is a "covering" (sum term or product term) of one or more minterms in a sum of products (or maxterms in a product of sums) of a boolean function. Formally, a product term P in a sum of products is an implicant of the Boolean function F if P implies F. More precisely:

> *P* implies *F* (and thus is an implicant of *F*) if *F* also takes the value 1 whenever *P* equals 1.

where
- *F* is a Boolean of *n* variables.
- *P* is a product term

This means that P < = F with respect to the natural ordering of the Boolean space. For instance, the function

> $f(x,y,z,w) = xy + yz + w$ is implied by *xy*, by *xyz*, by *xyzw*, by *w* and many others; these are the implicants of *f*.

Digital System Design   *17EE35*

Prime implicant

A prime implicant of a function is an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant. W.V. Quine defined a prime implicant of F to be an implicant that is minimal - that is, if the removal of any literal from P results in a non-implicant for F. Essential prime implicants are prime implicants that cover an output of the function that no combination of other prime implicants is able to cover.
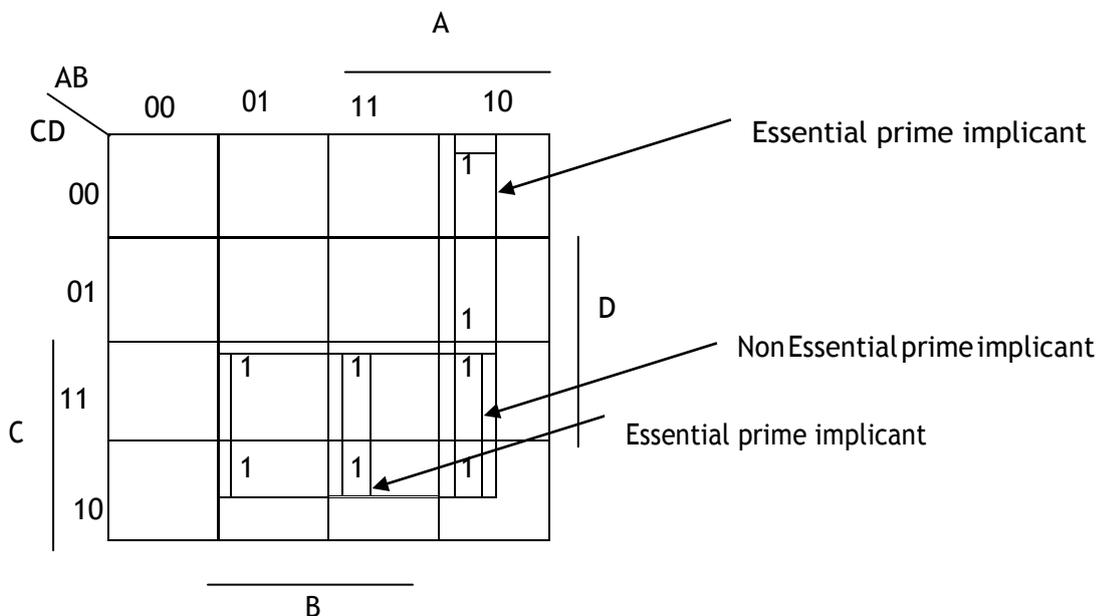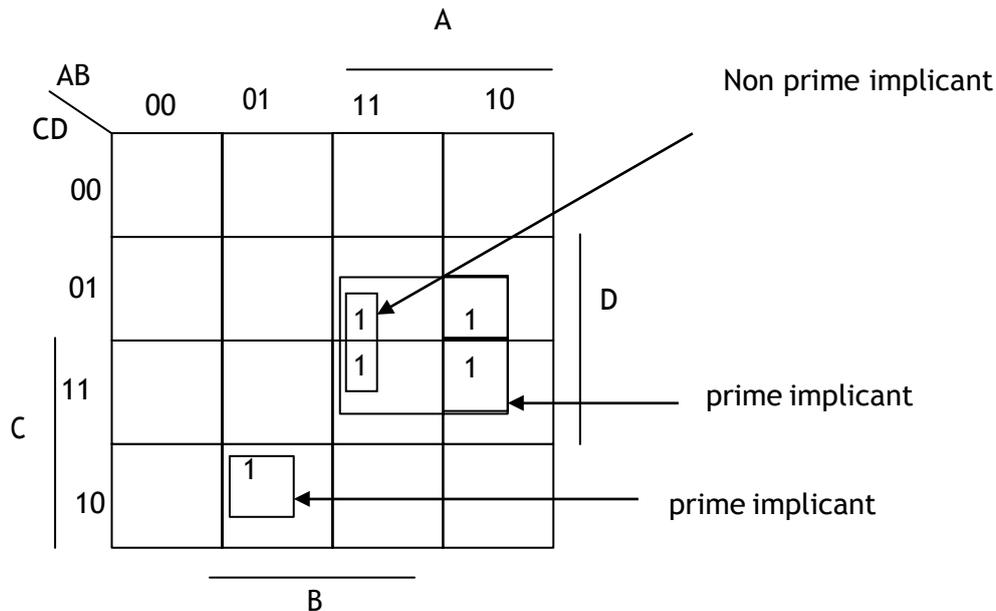
In simplifying a Boolean function using karnaugh map, non essential prime implicant are not needed

Digital System Design   *17EE35*

Minimization of Boolean expressions using Karnaugh maps.

Given the following truth table for the majority function.

a	b	C	M(output)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The Boolean algebraic expression is

$m = a'bc + ab'c + abc' + abc$.

 the minimization using algebraic manipulation can be done as follows.

$m = a'bc + abc + ab'c + abc + abc' + abc$

$= (a' + a)bc + a(b' + b)c + ab(c' + c)$

$= bc + ac + ab$

The **abc** term was replicated and combined with the other terms.

To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 101.

The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 110 (**a** = 1, **b** = 1, **c** = 0).

The minimization is done by drawing circles around sets of adjacent 1s. Adjacency is horizontal, vertical, or both. The circles must always contain 2^n 1s where n is an integer.

Digital System Design   17EE35
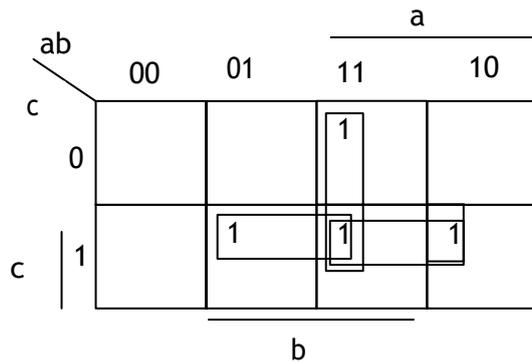
We have circled two 1s. The fact that the circle spans the two possible values of **a**

(0 and 1) means that the **a** term is eliminated from the Boolean expression corresponding to this circle.

Now we have drawn circles around all the 1s. Thus the expression reduces to

bc + ac + ab

as we saw before.

What is happening? What does adjacency and grouping the 1s together have to do with minimization? Notice that the 1 at position 111 was used by all 3 circles. This 1 corresponds to the abc term that was replicated in the original algebraic minimization. Adjacency of 2 1s means that the terms corresponding to those 1s differ in one variable only. In one case that variable is negated and in the other it is not.

The map is easier than algebraic minimization because we just have to recognize patterns of 1s in the map instead of using the algebraic manipulations. Adjacency also applies to the edges of the map.

Now for 4 Boolean variables. The Karnaugh map is drawn as shown below.

Digital System Design　　　　　*17EE35*
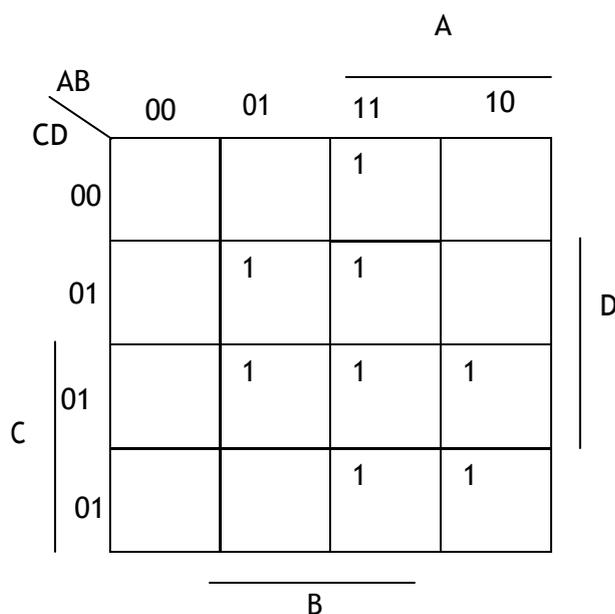
The following corresponds to the Boolean expression

Q = A′BC′D + A′BCD + ABC′D′ + ABC′D + ABCD + ABCD′ + AB′CD + AB′CD′

RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s.

Grouping the 1s together results in the following.
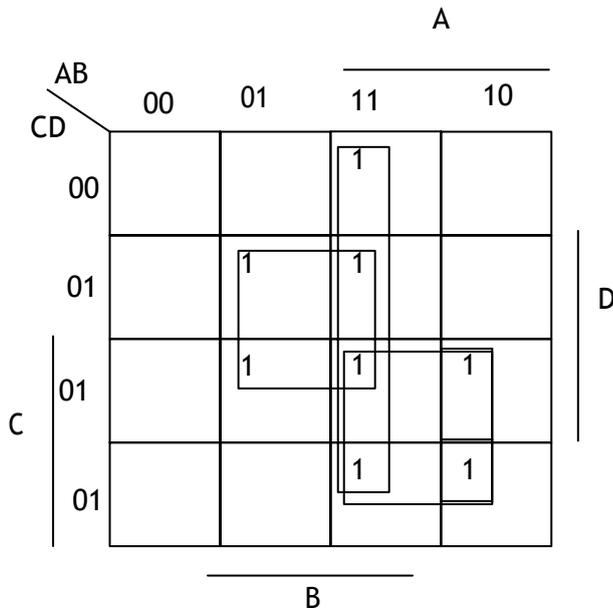
The expression for the groupings above is

Q = BD + AC + AB

This expression requires 3 2-input **AND** gates and 1 3-input **OR** gate.

Other examples

1. **F=A′B+AB**

=B

2. F=A′B′C′+A′B′C+A′BC′+ABC′+ABC

Digital System Design   *17EE35*

A

AB

	00	01	11	10
C 0	1	1	1	
C 1	1		1	

B

$=A'B'+BC'+AB$

3.   F=AB+A′BC′D+A′BCD+AB′C′D′

A

AB

CD

	00	01	11	10
00			1	1
01		1	1	
01		1	1	
01			1	

B

D $=BD+AB+AC'D'$

C

4.   F=AC′D′+A′B′C+A′C′D+AB′D

A

AB

CD

	00	01	11	10

Digital System Design   *17EE35*

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

| Electrical |
| Academic |
| Notes |
| **Digital System Design** |

$$D = B'D + AC'D' + A'C'D + A'B'C$$

5. $F = A'B'C'D' + AB'C'D' + A'BC'D + ABC'D + A'BCD + ABCD$

$$D = BD + D'B'$$

Obtaining a Simplified product of sum using Karnaugh map

The simplification of the product of sum follows the same rule as the product of sum. However, adjacent cells to be combined are the cells containing 0. In this approach, the obtained simplified function is F'. since F is represented by the square marked with 1. The function F can be obtained in product of sum by applying de morgan's rule on F'.

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

F=A′B′C′D′+A′BC′D′+AB′C′D′+A′BC′D+A′B′CD′+A′BCD′+AB′CD′

The obtained simplified F′=AB+CD+BD′. Since F″=F, By applying de morgan's rule to F′, we obtain

F″=(AB+CD+BD′)′

=(A′+B′)(C′+D′)(B′+D) which is he simplified F in product of sum.

Don't Care condition

Sometimes we do not care whether a 1 or 0 occurs for a certain set of inputs. It may be that those inputs will never occur so it makes no difference what the output is. For example, we might have a BCD (binary coded decimal) code which consists of 4 bits to encode the digits 0 (0000) through 9 (1001). The remaining codes (1010 through 1111) are not used. If we had a truth table for the prime numbers 0 through 9, it would be

A	B	C	D	F
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	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$F = A'B'CD' + A'B'CD + A'BC'D + A'BCD$

The X in the above stand for "don't care", we don't care whether a 1 or 0 is the value for that combination of inputs because (in this case) the inputs will never occur.

$F = BD + B'C$
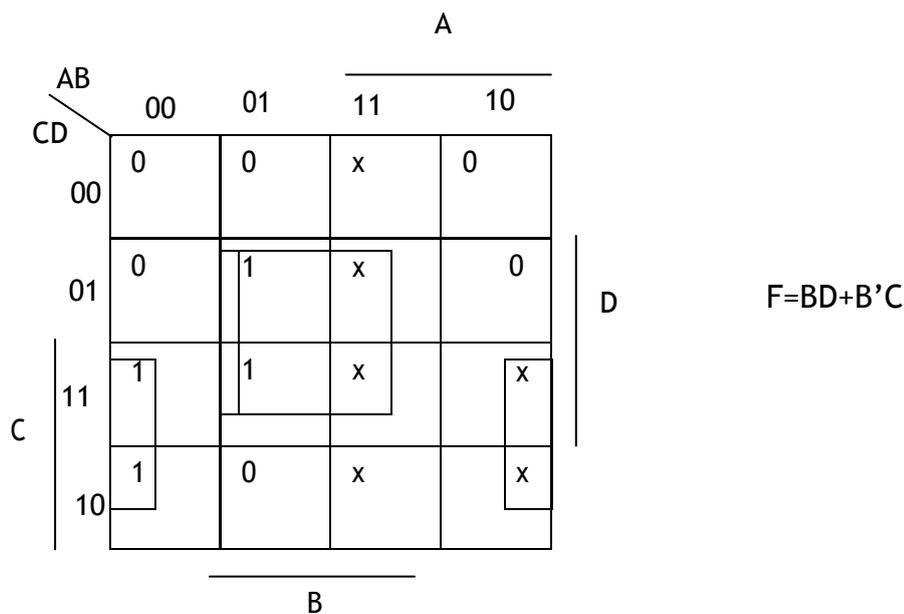
The tabulation method(Quine-McCluskey)

For function of five or more variables, it is difficult to be sure that the best selection is made. In such case, the tabulation method can be used to overcome such difficulty. The tabulation method was first formulated by Quine and later improved by McCluskey. It is also known as Quine-McCluskey method.

The Quine–McCluskey algorithm (or the method of prime implicants) is a method used for minimization of boolean functions. It is functionally identical to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a Boolean function has been reached.

The method involves two steps:

Finding all prime implicants of the function.

Use those prime implicants in a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function.

Step 1: finding prime implicants

Minimizing an arbitrary function:

```
      A B C D  f
 m0  0 0 0 0   0
 m1  0 0 0 1   0
 m2  0 0 1 0   0
 m3  0 0 1 1   0
 m4  0 1 0 0   1
 m5  0 1 0 1   0
 m6  0 1 1 0   0
 m7  0 1 1 1   0
 m8  1 0 0 0   1
 m9  1 0 0 1   x
m10  1 0 1 0   1
m11  1 0 1 1   1
m12  1 1 0 0   1
m13  1 1 0 1   0
m14  1 1 1 0   x
m15  1 1 1 1   1
```

One can easily form the canonical sum of products expression from this table, simply by summing the minterms (leaving out don't-care terms) where the function evaluates to one:

F(A,B,C,D) = A′BC′D′ + AB′C′D′ + AB′CD′ + AB′CD + ABC′D′ + ABCD

Of course, that's certainly not minimal. So to optimize, all minterms that evaluate to one are first placed in a minterm table. Don't-care terms are also added into this table, so they can be combined with minterms:

Number of 1s Minterm Binary Representation

```
-------------------------------------------
1       m4      0100
        m8      1000
-------------------------------------------
2       m9      1001
        m10     1010
        m12     1100
-------------------------------------------
3       m11     1011
        m14     1110
-------------------------------------------
4       m15     1111
```

At this point, one can start combining minterms with other minterms. If two terms vary by only a single digit changing, that digit can be replaced with a dash indicating that the digit doesn't matter. Terms that can't be combined any more are marked with a "*". When going from Size 2 to Size 4, treat '-' as a third bit value. Ex: -110 and -100 or -11- can be combined, but not -110 and 011-. (Trick: Match up the '-' first.)

```
Number of 1s Minterm  0-Cube      | Size 2 Implicants   | Size 4 Implicants
-----------------------------     |-------------------  |----------------------
1       m4      0100              | m(4,12)  -100*      | m(8,9,10,11)  10--*
        m8      1000              | m(8,9)  100-        | m(8,10,12,14)  1--0*
-----------------------------     | m(8,10)  10-0       |----------------------
2       m9      1001              | m(8,12)  1-00       | m(10,11,14,15) 1-1-*
        m10     1010              |---------------------   |
        m12     1100              | m(9,11)  10-1       |
-----------------------------     | m(10,11) 101-       |
3       m11     1011              | m(10,14) 1-10       |
        m14     1110              | m(12,14) 11-0       |
-----------------------------     |-------------------  |
4       m15     1111              | m(11,15) 1-11       |
                                  | m(14,15) 111-       |
```

At this point, the terms marked with * can be seen as a solution. That is the solution is

F=AB′+AD′+AC+BC′D′

If the karnaugh map was used, we should have obtain an expression simplier than this. To obtain a minimal form, we need to use the prime implicant chart

Step 2: prime implicant chart

None of the terms can be combined any further than this, so at this point we construct an essential prime implicant table. Along the side goes the prime implicants that have just been generated, and along the top go the minterms specified earlier. The don't care terms are not placed on top - they are omitted from this section because they are not necessary inputs.

	4	8	10	11	12	15	
m(4,12)	X				X		-100 (BC′D′)
m(8,9,10,11)		X	X	X			10--(AB′)
m(8,10,12,14)		X	X		X		1--0 (AD′)
m(10,11,14,15)			X	X		X	1-1- (AC)

In the prime implicant table shown above, there are 5 rows, one row for each of the prime implicant and 6 columns, each representing one minterm of the function. X is placed in each row to indicate the minterms contained in the prime implicant of that row. For example, the two X in the first row indicate that minterm 4 and 12 are contained in the prime implicant represented by (-100) i.e. BC′D′

The completed prime implicant table is inspected for columns containing only a single x. in this example, there are two minterms whose column have a single x. 4,15. The minterm 4 is covered by prime implicant BC′D′. that is the selection of prime implicant BC′D′ guarantee that minterm 4 is included in the selection. Similarly, for minterm 15 is covered by prime implicant AC. Prime implicants that cover minterms with a single X in their column are called essential prime implicants.

Those essential prime implicant must be selected.

Now we find out each column whose minterm is covered by the selected essential prime implicant

For this example, essential prime implicant BC′D′ covers minterm 4 and 12. Essential prime implicant AC covers 10, 11 and 15. An inspection of the implicant table shows that, all the minterms are covered by the essential prime implicant except the minterms 8. The minterms not selected must be included by the selection of one or more prime implicants. From this example, we have only one minterm which is 8. It can be included in the selection either by including the prime implicant AB′ or AD′. Since both of them have minterm 8 in their selection. We have thus found the minimum set of prime implicants whose sum gives the required minimized function:

F=BC′D′+AD′+AC OR F= BC′D′+AB′+AC.

Both of those final equations are functionally equivalent to this original (very area-expensive) equation:

F(A,B,C,D) = A′BC′D′ + AB′C′D′ + AB′CD′ + AB′CD + ABC′D′ + ABCD

Implimenting logical circuit using NAND and NOR gate only.

In addition to AND, OR, and NOT gates, other logic gates like NAND and NOR are also used in the design of digital circuits.

The NAND gate represents the complement of the AND operation. Its name is an

abbreviation of NOT AND. The graphic symbol for the NAND gate consists of an AND symbol with a bubble on the output, denoting that a complement operation is performed on the output of the AND gate as shown earlier

The NOR gate represents the complement of the OR operation. Its name is an abbreviation of NOT OR. The graphic symbol for the NOR gate consists of an OR symbol with a bubble on the output, denoting that a complement operation is performed on the output of the OR gate as shown earlier.

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families. In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around.

Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around.

NAND Gate is a Universal Gate

To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates. A universal gate is a gate which can implement any Boolean function without need to use any other gate type.

Implementing an Inverter Using only NAND Gate

The figure shows two ways in which a NAND gate can be used as an inverter (NOT gate).

1. All NAND input pins connect to the input signal A gives an output A′.
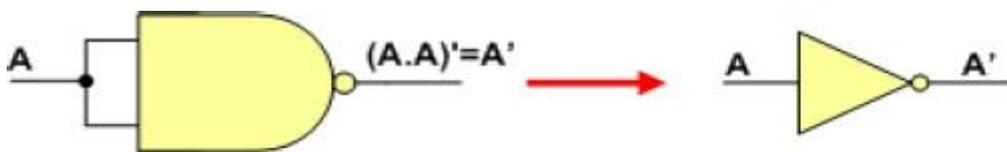
2. One NAND input pin is connected to the input signal A while all other input pins are connected to logic 1. The output will be A′.

Implementing AND Using only NAND Gates

An AND gate can be replaced by NAND gates as shown in the figure (The AND is

replaced by a NAND gate with its output complemented by a NAND gate inverter).

Digital System Design   *17EE35*
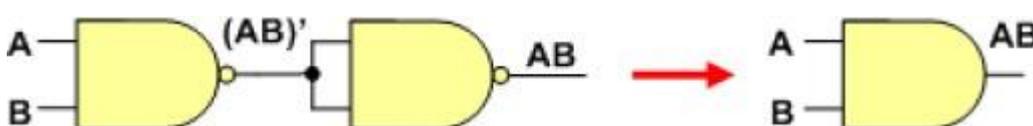
Implementing OR Using only NAND Gates

An OR gate can be replaced by NAND gates as shown in the figure (The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters).
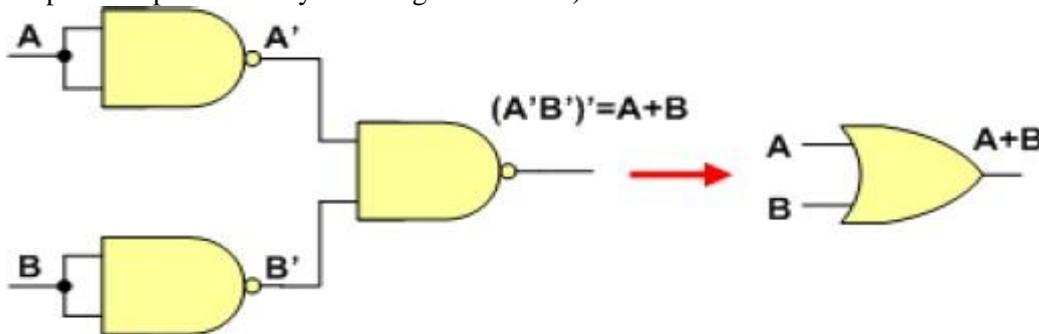
Thus, the NAND gate is a universal gate since it can implement the AND, OR and NOT functions.

NOR Gate is a Universal Gate:

To prove that any Boolean function can be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

Implementing an Inverter Using only NOR Gate

The figure shows two ways in which a NOR gate can be used as an inverter (NOT gate).

1.All NOR input pins connect to the input signal A gives an output A′.
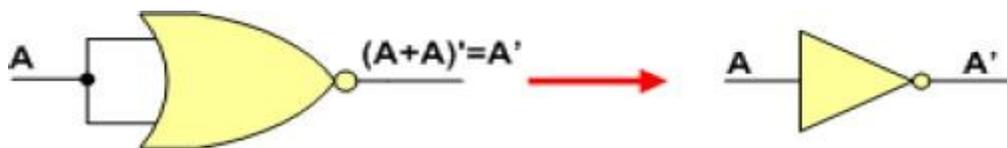
2. One NOR input pin is connected to the input signal A while all other input pins are connected to logic 0. The output will be A′.
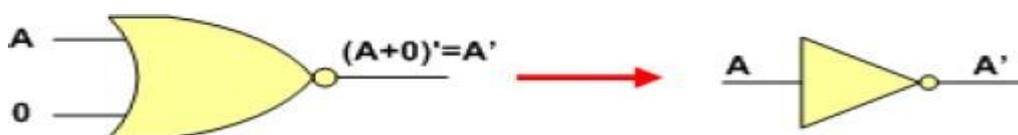
Implementing OR Using only NOR Gates

An OR gate can be replaced by NOR gates as shown in the figure (The OR is replaced by a NOR gate with its output complemented by a NOR gate inverter)
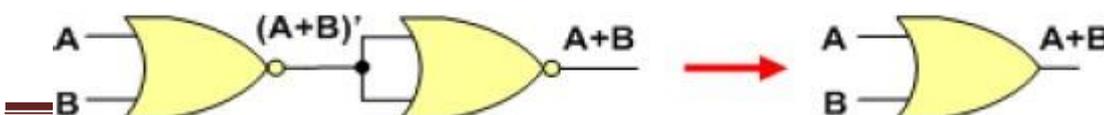
Implementing AND Using only NOR Gates

An AND gate can be replaced by NOR gates as shown in the figure (The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters)
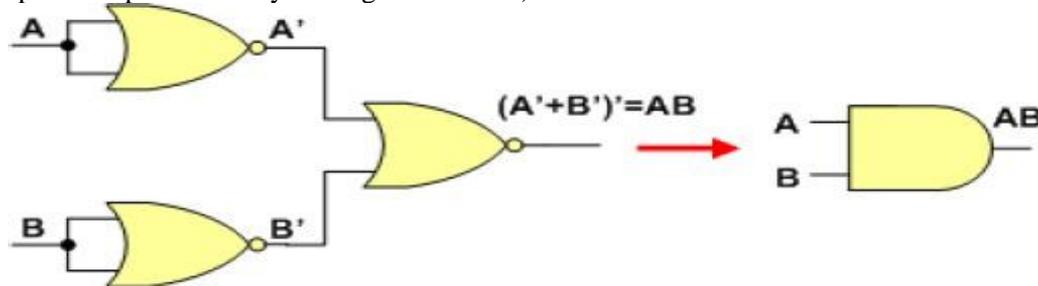
Thus, the NOR gate is a universal gate since it can implement the AND, OR and NOT functions.

Equivalent Gates:

The shown figure summarizes important cases of gate equivalence. Note that bubbles indicate a complement operation (inverter).

A NAND gate is equivalent to an inverted-input OR gate.

An AND gate is equivalent to an inverted-input NOR gate.

A NOR gate is equivalent to an inverted-input AND gate.

An OR gate is equivalent to an inverted-input NAND gate.

Two NOT gates in series are same as a buffer because they cancel each other as $A'' = A$.

Two-Level Implementations:

We have seen before that Boolean functions in either SOP or POS forms can be implemented using 2-Level implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

To implement a function using NAND gates only, it must first be simplified to a sum of product and to implement a function using NOR gates only, it must first be simplified to a product of sum

We will show that SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only NOR gates through examples.

Example 1: Implement the following SOP function using NAND gate only

F = XZ + Y′Z + X′YZ

Being an SOP expression, it is implemented in 2-levels as shown in the figure.
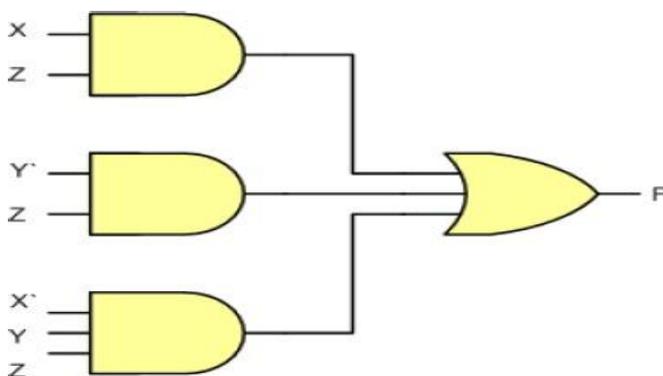
Introducing two successive inverters at the inputs of the OR gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.
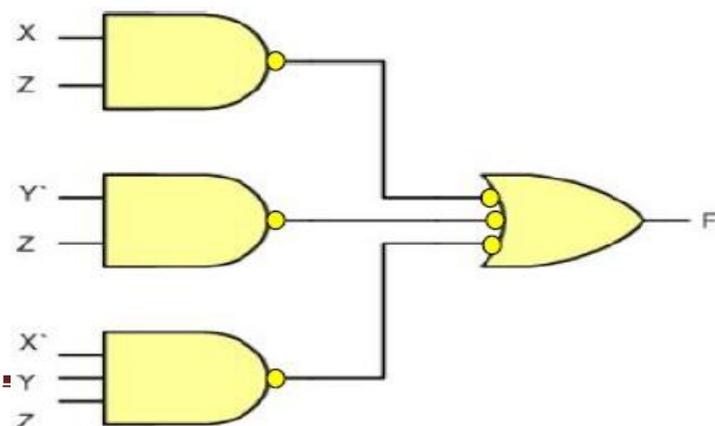
By associating one of the inverters with the output of the first level AND gate and the other with the input of the OR gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NAND gates as shown in Figure.
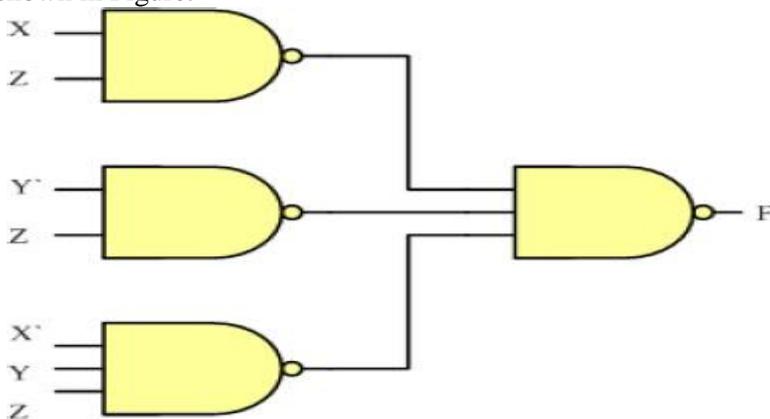
Example 2: Implement the following POS function using NOR gates only

F = (X+Z) (Y′+Z) (X′+Y+Z)

Being a POS expression, it is implemented in 2-levels as shown in the figure.
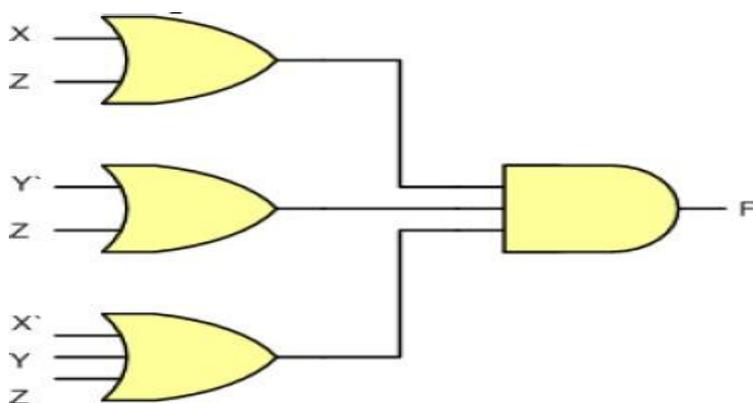
Introducing two successive inverters at the inputs of the AND gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.

Digital System Design   *17EE35*
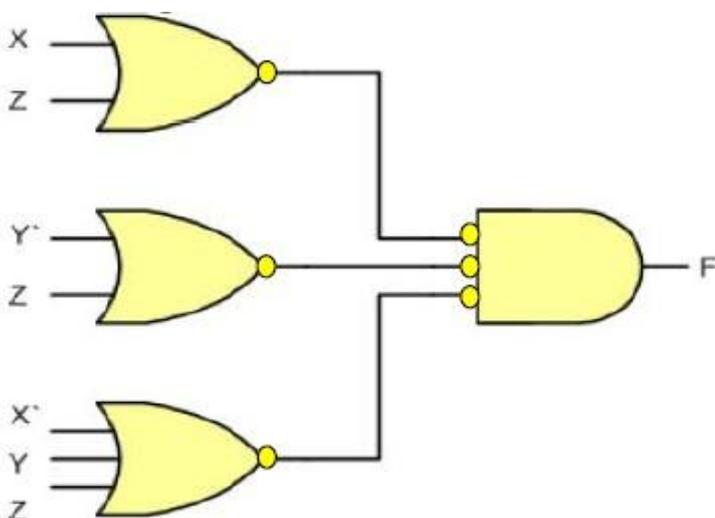
By associating one of the inverters with the output of the first level OR gates and the other with the input of the AND gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NOR gates as shown in Figure.
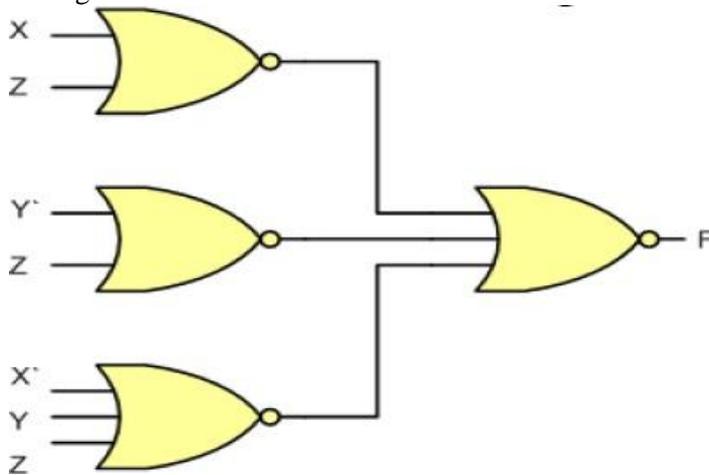
There are some other types of 2-level combinational circuits which are

• NAND-AND

• AND-NOR,

• NOR-OR,

• OR-NAND

These are explained by examples.

AND-NOR functions:

Example 3: Implement the following function $F=(XZ+Y'Z+X'YZ)'$ OR $\quad\quad\quad F'=XZ+Y'Z+X'YZ$

Since F′ is in SOP form, it can be implemented by using NAND-NAND circuit.

By complementing the output we can get F, or by using *NAND-AND* circuit as shown in the figure.

Digital System Design   17EE35
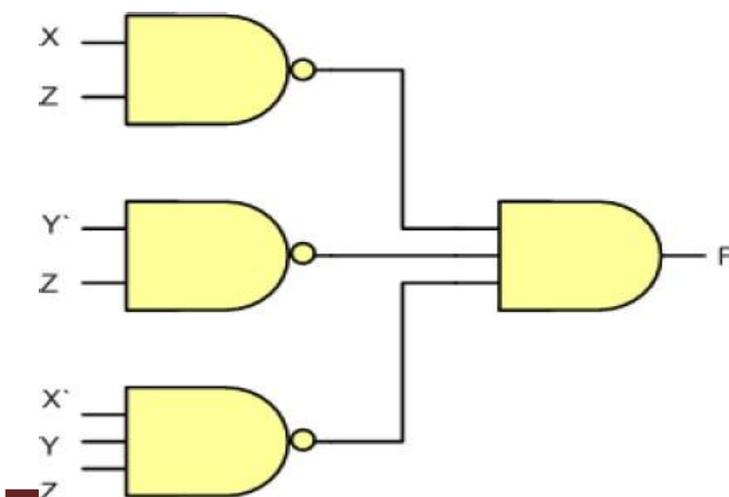
It can also be implemented using *AND-NOR* circuit as it is equivalent to NAND- AND circuit as shown in the figure.
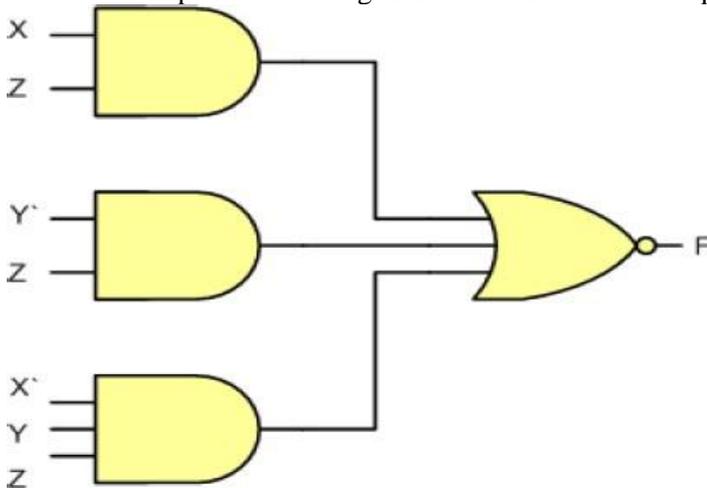
OR-NAND functions:

Example 4: Implement the following function

F=((X+Z)(Y′+Z)(X′+Y+Z)) ′ orF′ (X+Z)(Y′+Z)(X′+Y+Z)

Since **F′** is in POS form, it can be implemented by using NOR-NOR circuit.

By complementing the output we can get **F**, **or** by using *NOR-OR* circuit as shown in the figure.
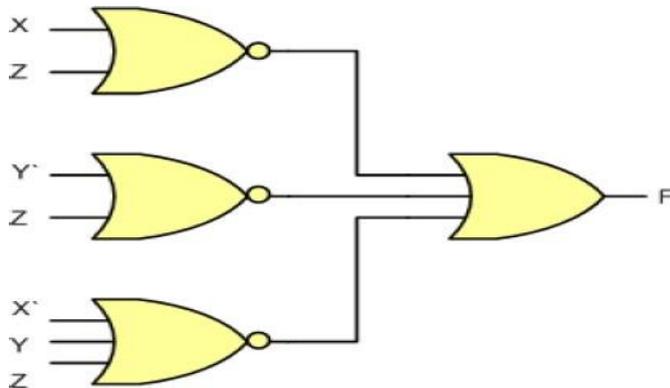
It can also be implemented using ***OR-NAND*** circuit as it is equivalent to NOR-OR circuit as shown in the figure
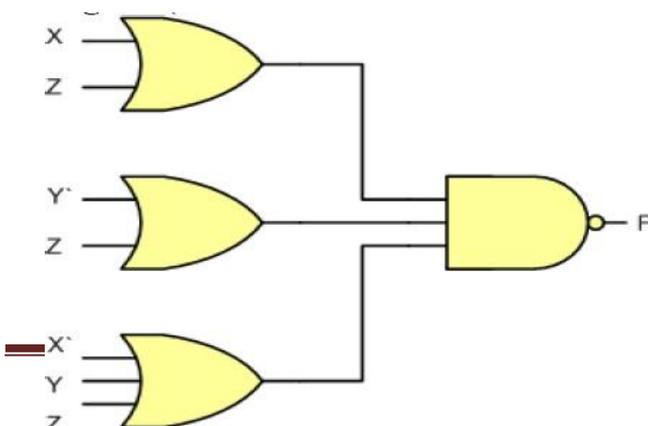
Designing Combinatorial Circuits

The design of a combinational circuit starts from the verbal outline of the problem and ends with a logic circuit diagram or a set of Boolean functions from which the Boolean function can be easily obtained. The procedure involves the following steps:

- The problem is stated
- The number of available input variables and required output variables is determined.
- The input and output variable are assigned their letter symbol
- The truth table that defines the required relationship between the inputs and the outputs is derived.
- The simplified Boolean function for each output is obtained
- The logic diagram is drawn.

Example of combinational circuit

Adders

In electronics, an adder or summer is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where twos complement or ones complement is being used to represent negative numbers, it is trivial to modify an adder into an adder-subtracter. Other signed number representations require a more complex adder.

-Half Adder

A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

A half adder has two inputs, generally labelled A and B, and two outputs, the sum S and carry C. S is the two-bit XOR of A and B, and C is the AND of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs.

The drawback of this circuit is that in case of a multibit addition, it cannot include a carry.

Following is the truth table for a half adder:

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Equation of the Sum and Carry.

   Sum=A′B+AB′   Carry=AB

One can see that Sum can also be implemented using XOR gate as A$\oplus$B
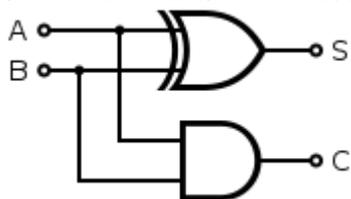
-Full Adder.

 A full adder has three inputs *A*, *B*, and a carry in *C*, such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled C_i or C_{in} while the carry out is labelled C_o or C_{out}.

A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits. It can be combined with other full adders or work on its own.

Input			Output	
A	**B**	C_i	C_o	S
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

$C_o = A'BC_i + AB'C_i + ABC_i' + ABC_i$

$S = A'B'C_i + A'BC_i' + ABC_i' + ABC_i$

A full adder can be trivially built using our   ng
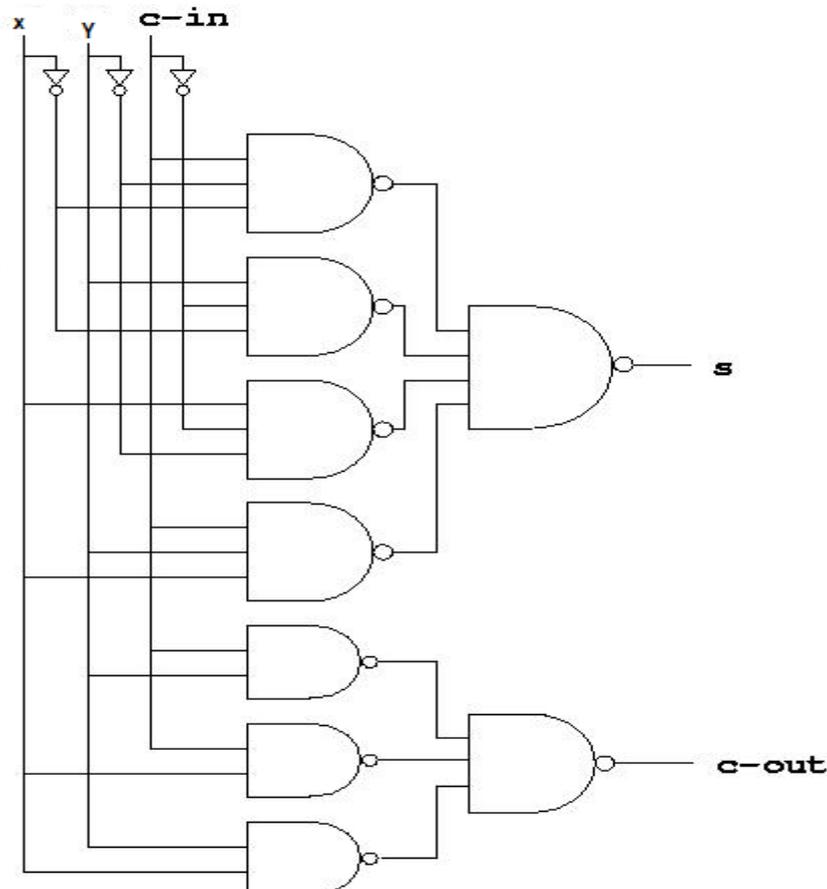
circuit diagram using NAND gates only:

$C_o = A'BC_i + AB'C_i + ABC_i' + ABC_i$ by manipulating C_o, we can see that $C_o = C_i(A \oplus B) + AB$

$S = A'B'C_i + A'BC_i' + ABC_i' + ABC_i$ By manipulating S, we can see that $S = C_i \oplus (A \oplus B)$

Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and XOR gates occurs when both inputs are 1; for the adder shown here, this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips.

A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, connecting the sum from that to an input to the second adder, connecting Ci to the other input and OR the two carry outputs. Equivalently, S could be made the three-bit xor of A, B, and Ci and Co could be made the three-bit majority function of A, B, and Ci. The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.

Digital System Design   *17EE35*
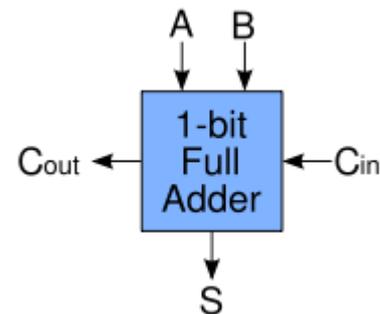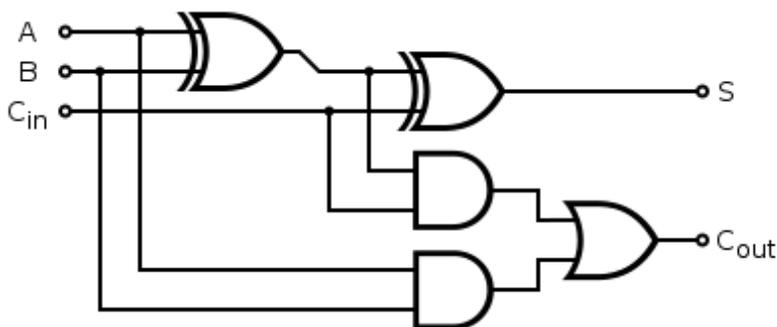
Ripple carry adder

It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a *ripple carry adder*, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder.
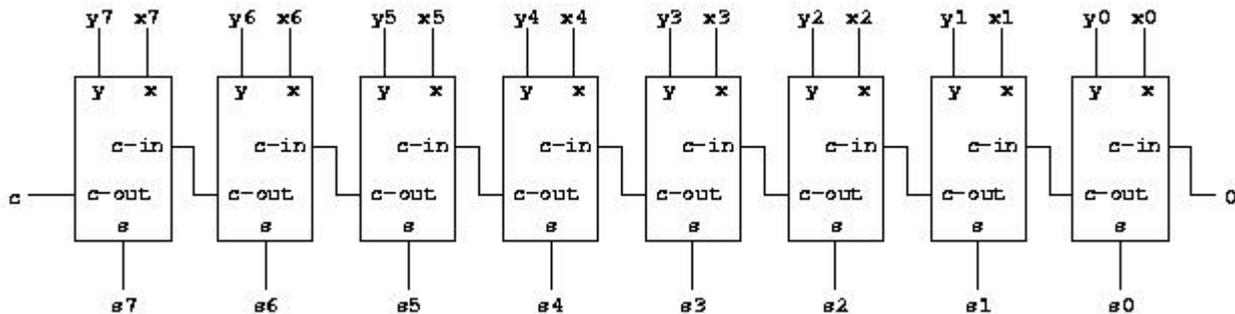
The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Following the path from C_{in} to C_{out} shows 2 gates that must be passed through. Therefore, a 32-bit adder requires 31 carry computations and the final sum calculation for a total of 31 * 2 + 1 = 63 gate delays.

Subtractor

In electronics, a subtractor can be designed using the same approach as that of an adder. The binary subtraction process is summarized below. As with an adder, in the general case of calculations on multi-bit numbers, three bits are involved in performing the subtraction for each bit: the minuend (X_i), subtrahend (Y_i), and a borrow in from the previous (less significant) bit order position (B_i). The outputs are the difference bit (D_i) and borrow bit $B_i + 1$.

Half subtractor

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow). Such a circuit is called a half-subtractor because it enables a borrow out of the current arithmetic operation but no borrow in from a previous arithmetic operation.

The truth table for the half subtractor is given below.

X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0
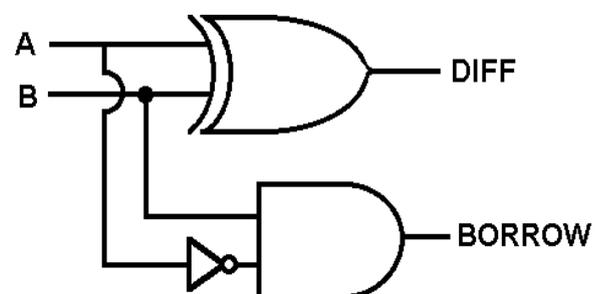
D=X′Y+XY′   or D= X $\oplus$ Y

B=X′Y

Full Subtractor

As in the case of the addition using logic gates , a *full subtractor* is made by combining two half-subtractors and an additional OR-gate. A full subtractor has the borrow in capability (denoted as **BOR$_{IN}$** in the diagram below) and so allows *cascading* which results in the possibility of **multi-bit subtraction**.

The final truth table for a full subtractor looks like:

A	B	BOR$_{IN}$	D	BOR$_{OUT}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Find out the equations of the borrow and the difference

The circuit diagram for a full subtractor is given below.
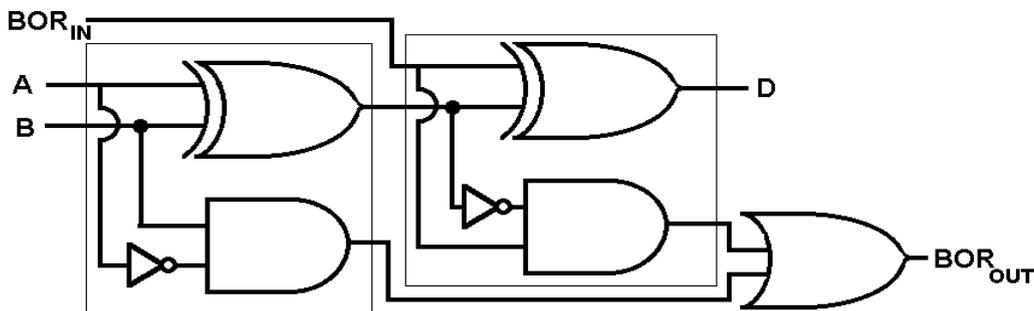
For a wide range of operations many circuit elements will be required. A neater solution will be to use subtraction via addition using *complementing* as was discussed in the binary arithmetic topic. In this case only adders are needed as shown bellow.

Binary subtraction using adders

Our binary adder can already handle negative numbers as indicated in the section on binary arithmetic But we have not discussed how we can get it to handle subtraction. To see how this can be done, notice that in order to compute the expression *x - y*, we can compute the expression *x + -y* instead. We know from the section on binary arithmetic how to negate a number by inverting all the bits and adding 1. Thus, we can compute the expression as *x + inv(y) + 1*. It suffices to invert all the inputs of the second operand before they reach the adder, but how do we add the 1. That seems to require another adder just for that. Luckily, we have an unused carry-in signal to position 0 that we can use. Giving a 1 on this input in effect adds one to the result. The complete circuit with addition and subtraction looks like this:

Digital System Design   17EE35
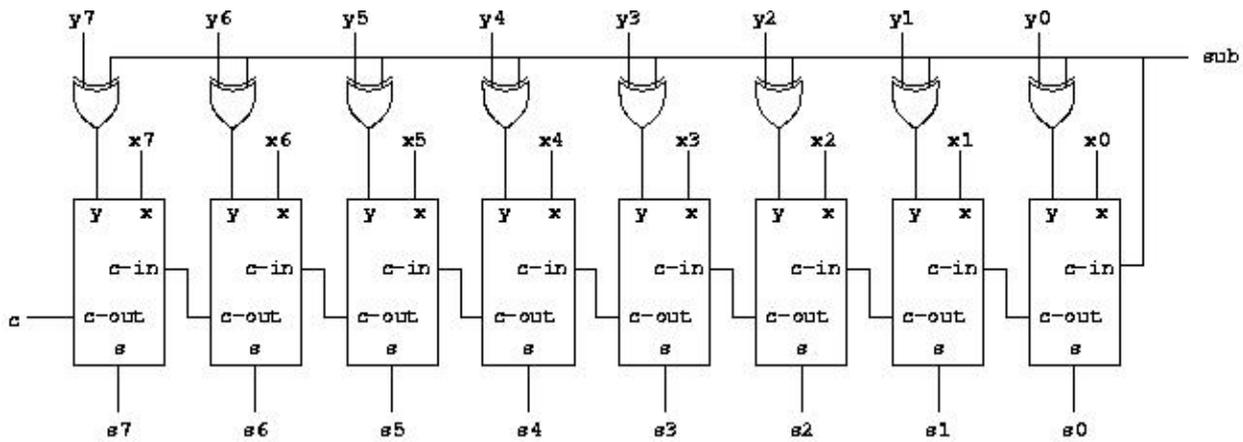
Exercise. Generate the truth table and Draw a logic circuit for a 3 bit message Parity Checker and generator seen in data representation section

Medium Scale integration component

The purpose of circuit minimization is to obtain an algebraic expression that, when implemented results in a low cost circuit. Digital circuit are constructed with integrated circuit(IC). An IC is a small silicon semiconductor crystal called chip containing the electronic component for digital gates. The various gates are interconnected inside the chip to form the required circuit. Digital IC are categorized according to their circuit complexity as measured by the number of logic gates in a single packages.

- Small scale integration (SSI). SSi devices contain fewer than 10 gates. The input and output of the gates are connected directly to the pins in the package.
- Medium Scale Integration. MSI devices have the complexity of approximately 10 to 100 gates in a single package
- Large Scale Integration (LSI). LSI devices contain between 100 and a few thousand gates in a single package
- Very Large Scale Integration(VLSI). VLSI devices contain thousand of gates within a single package. VLSI devices have revolutionized the computer system design technology giving the designer the capabilities to create structures that previously were uneconomical.

Multiplexer

A multiplexer is a combinatorial circuit that is given a certain number (usually a power of two) *data inputs*, let us say 2^n, and n *address inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the selected data input.

In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input.

The truth table for a multiplexer is huge for all but the smallest values of n. We therefore use an abbreviated version of the truth table in which some inputs are replaced by `-' to indicate that the input value does not matter.

Here is such an abbreviated truth table for n = 3. The full truth table would have $2^{(3 + 23)} = 2048$ rows.

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

```
       SELECT            INPUT
a₂ a₁ a₀ d₇ d₆ d₅ d₄ d₃ d₂ d₁ d₀ |      x
-  -  -  -  -  -  -  -  -  -  -  --- -
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
```

We can abbreviate this table even more by using a letter to indicate the value of the selected input, like this:

```
a₂ a₁ a₀ d₇ d₆ d₅ d₄ d₃ d₂ d₁ d₀ |   x
-  -  -  -  -  -  -  -  -  -  -  --- -
0  0  0  -  -  -  -  -  -  -  c |   c
0  0  1  -  -  -  -  -  -  c  - |   c
0  1  0  -  -  -  -  -  c  -  - |   c
0  1  1  -  -  -  -  c  -  -  - |   c
1  0  0  -  -  -  c  -  -  -  - |   c
1  0  1  -  -  c  -  -  -  -  - |   c
1  1  0  -  c  -  -  -  -  -  - |   c
1  1  1  c  -  -  -  -  -  -  - |   c
```

The same way we can simplify the truth table for the multiplexer, we can also simplify the corresponding circuit. Indeed, our simple design method would yield a very large circuit. The simplified circuit looks like this:

Digital System Design   17EE35
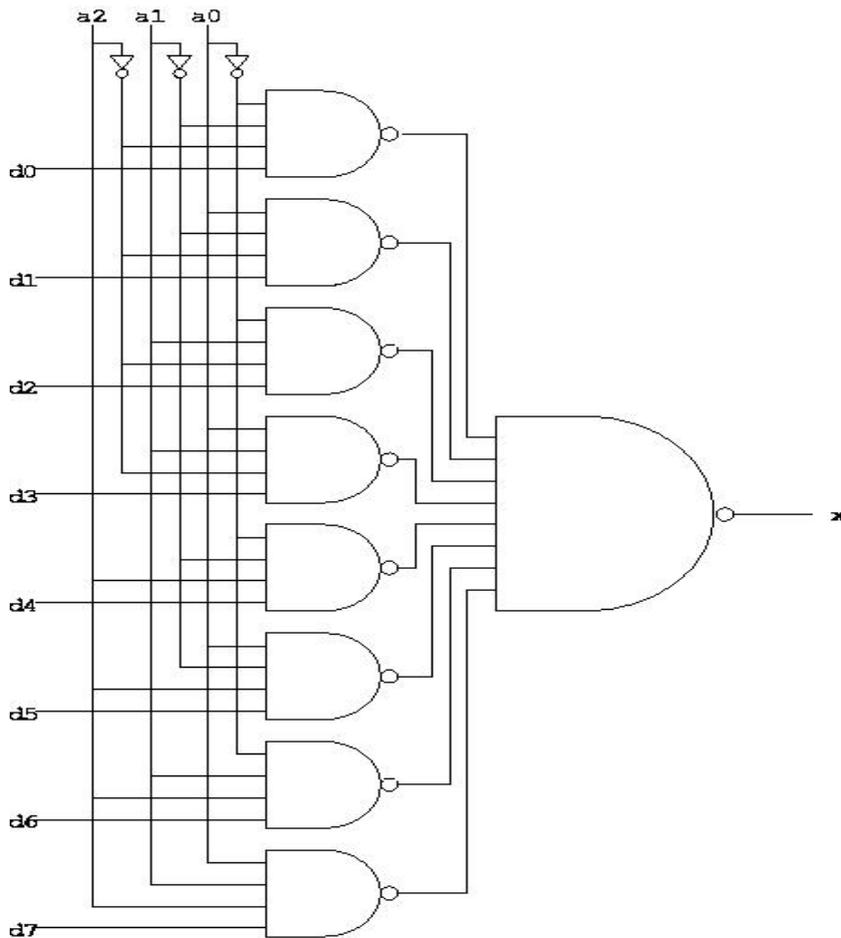
Demultiplexer

The demultiplexer is the inverse of the multiplexer, in that it takes a single data input and n address inputs. It has 2^n outputs. The address input determine which data output is going to have the same value as the data input. The other data outputs will have the value 0.

Here is an abbreviated truth table for the demultiplexer. We could have given the full table since it has only 16 rows, but we will use the same convention as for the multiplexer where we abbreviated the values of the data inputs.

```
 a2 a1 a0 d | x7 x6 x5 x4 x3 x2 x1 x0
 ------------------------------------
  0  0  0  c |  0  0  0  0  0  0  0  c
  0  0  1  c |  0  0  0  0  0  0  c  0
  0  1  0  c |  0  0  0  0  0  c  0  0
  0  1  1  c |  0  0  0  0  c  0  0  0
  1  0  0  c |  0  0  0  c  0  0  0  0
  1  0  1  c |  0  0  c  0  0  0  0  0
  1  1  0  c |  0  c  0  0  0  0  0  0
  1  1  1  c |  c  0  0  0  0  0  0  0
```
Here is one possible circuit diagram for the demultiplexer:

Digital System Design   17EE35
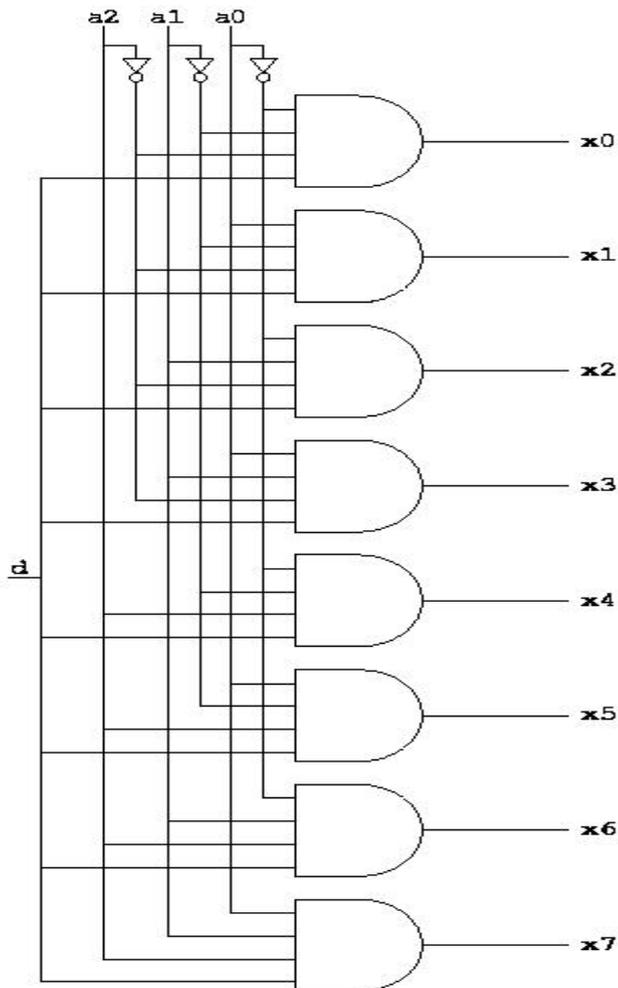
Decoder

In both the multiplexer and the demultiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of n digits to 2^n outputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0.

It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*. It has the following truth table (for n = 3):

```
 a2 a1 a0 | x7 x6 x5 x4 x3 x2 x1 x0
 ----------------------------------
  0  0  0 | 0  0  0  0  0  0  0  1
  0  0  1 | 0  0  0  0  0  0  1  0
  0  1  0 | 0  0  0  0  0  1  0  0
  0  1  1 | 0  0  0  0  1  0  0  0
  1  0  0 | 0  0  0  1  0  0  0  0
  1  0  1 | 0  0  1  0  0  0  0  0
  1  1  0 | 0  1  0  0  0  0  0  0
  1  1  1 | 1  0  0  0  0  0  0  0
```
Here is the circuit diagram for the decoder:

Digital System Design   *17EE35*
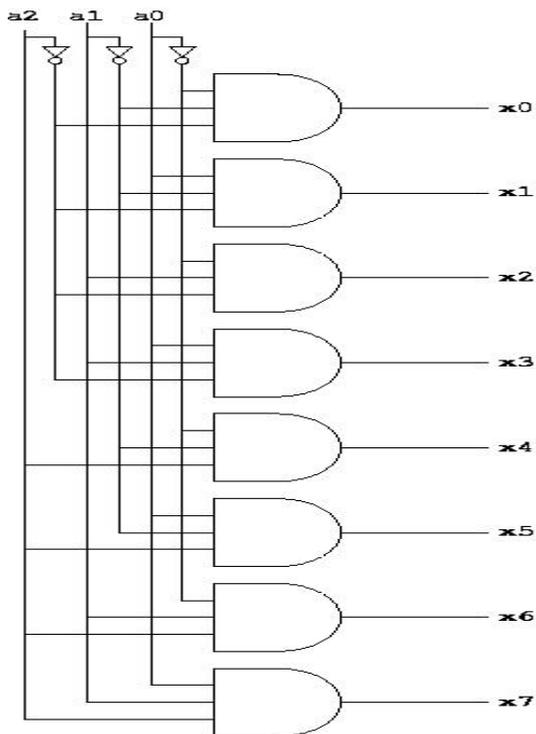
Encoder

An encoder has 2^n input lines and n output lines. The output lines generate a binary code corresponding to the input value. For example a single bit 4 to 2 encoder takes in 4 bits and outputs 2 bits. It is assumed that there are only 4 types of input signals these are : 0001, 0010, 0100, 1000.

I_3	I_2	I_1	I_0	F_1	F_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

4 to 2 encoder

Digital System Design   *17EE35*
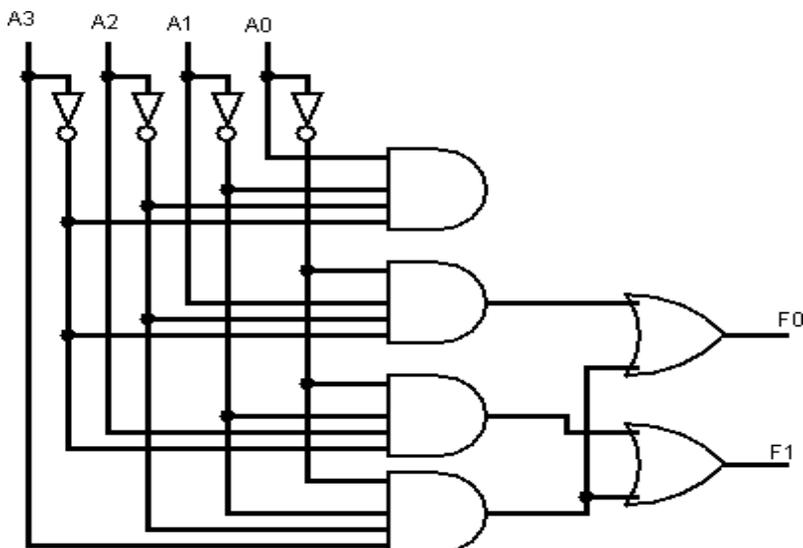
The encoder has the limitation that only one input can be active at any given time. If two inputs are simultaneously active, the output produces an undefined combination. To prevent this we make use of the priority encoder.

A priority encoder is such that if two or more inputs are given at the same time, the input having the highest priority will take precedence. An example of a single bit 4 to 2 encoder is shown.

I_3	I_2	I_1	I_0	F_1	F_0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

4 to 2 priority encoder

The X's designate the don't care condition designating that fact that the binary value may be equal either to 0 or 1. For example, the input I_3 has the highest priority so regarded the value of other inputs, if the value of I3 is 1, the output for F1F0=11(binary 3)

Exercise

1 By using algebraic manipulation, Show that:
  - A'B'C+A'BC'+AB'C'+ABC= A $\odot$ (B $\odot$ C)   - AB+C'D = (A+B+C)(A+B'+C)(A'+B+C)(A'+B+C')

2. A circuit has four inputs D,C,B,A encoded in natural binary form where A is the least significant bit. The inputs in the range 0000=0 to 1011=11 represents the months of the year from January (0) to December (11). Input in the range 1100-1111(i.e.12 to 15) cannot occur. The output of the circuit is true if the month represented by the input has 31 days. Otherwise the output is false. The output for inputs in the range 1100 to 1111 is undefined.
    - Draw the truth table to represent the problem and obtain the function F as a Sum of minterm.
    - Use the Karnaugh map to obtain a simplified expression for the function F.
    - Construct the circuit to implements the function using NOR gates only.

3. A circuit has four inputs P,Q,R,S, representing the natural binary number 0000=0, to 1111=15. P is the most significant bit. The circuit has one output, X, which is true if the input to the circuit represents is a prime number and false otherwise (A prime number is a number which is only divisible by 1 and by itself. Note that zero(0000) and one(0001) are not considered as prime numbers)
    i. Design a true table for this circuit, and hence obtain an expression for X in terms of P,Q,R,S.
    ii. Design a circuit diagram to implement this function using NOR gate only

4. A combinational circuit is defined by the following three Boolean functions: F1=x'y'z'+xz F2=xy'z'+x'y F3=x'y'z+xy Design the circuit that implements the functions

5. A circuit implements the Boolean function F=A'B'C'D'+A'BCD'+AB'C'D'+ABC'D It is found that the circuit input combinations A'B'CD', A'BC'D', AB'CD' can never occur.
    i. Find a simpler expression for F using the proper don't care condition.
    ii. Design the circuit implementing the simplified expression of F

6. A combinational circuit is defined by the following three Boolean functions: F1=x'y'z'+xz F2=xy'z'+x'y F3=x'y'z+xy Design the circuit with a decoder and external gates.

7. A circuit has four inputs P,Q,R,S, representing the natural binary number 0000=0, to 1111=15. P is the most significant bit. The circuit has one output, X, which is true if the number represented is divisible by three (Regard zero as being indivisible by three.)
Design a true table for this circuit, and hence obtain an expression for X in terms of P,Q,R,S as a product of maxterms and also as a sum of minterms
Design a circuit diagram to implement this function

Digital System Design   *17EE35*

8. Plot the following function on K map and use the K map to simplify the expression.

$$F = \overline{A}BC + \overline{A}\overline{B}C + A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C} + A\overline{B}C + \overline{A}B\overline{C}$$

$$F = A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C}$$ Simplify the following expressions by means of Boolean algebra

$$F = ABCD + \overline{A}\overline{B}\overline{C}D + A\overline{B}\overline{C}D + A\overline{B}CD + ABCD + ABCD + ABCD + ABCD$$

$$F = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C} + AB\overline{C} + AB\overline{C}$$

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical	
Academic	
Notes	
Digital System Design	

Sequential circuit
Introduction

In the previous session, we said that the output of a combinational circuit depends solely upon the input. The implication is that combinational circuits have no memory. In order to build sophisticated digital logic circuits, including computers, we need more a powerful model. We need circuits whose output depends upon both the input of the circuit and its previous state. In other words, we need circuits that have *memory.*

For a device to serve as a memory, it must have three characteristics:

* the device must have two stable states
* there must be a way to read the state of the device
* there must be a way to set the state at least once.

It is possible to produce circuits with memory using the digital logic gates we've already seen. To do that, we need to introduce the concept of *feedback.* So far, the logical flow in the circuits we've studied has been from input to output. Such a circuit is called *acyclic.* Now we will introduce a circuit in which the output is fed back to the input, giving the circuit memory. (There are other memory technologies that store electric charges or magnetic fields; these do not depend on feedback.)
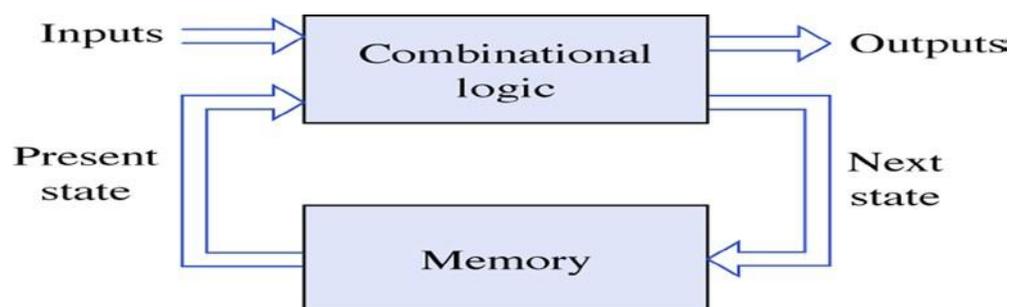
Latches and flip-flops

In the same way that gates are the building blocks of combinatorial circuits, *latches* and *flip-flops* are the building blocks of sequential circuits.

While gates had to be built directly from transistors, latches can be built from gates, and flip-flops can be built from latches. This fact will make it somewhat easier to understand latches and flip-flops.

Both latches and flip-flops are circuit elements whose output depends not only on the current inputs, but also on previous inputs and outputs. The difference between a latch and a flip-flop is that a latch does not have a *clock signal*, whereas a flip-flop always does.

Latches

How can we make a circuit out of gates that is not combinatorial? The answer is *feed-back*, which means that we create *loops* in the circuit diagrams so that output values depend, indirectly, on themselves. If such feed-back is *positive* then the circuit tends to have stable states, and if it is *negative* the circuit will tend to oscillate.

In order for a logical circuit to "remember" and retain its logical state even after the controlling input signal(s) have been removed, it is necessary for the circuit to include some form of feedback. We might start with a pair of inverters, each having its input connected to the other's output. The two outputs will always have opposite logic

Digital System Design   17EE35

levels.

The problem with this is that we don't have any additional inputs that we can use to change the logic states if we want. We can solve this problem by replacing the inverters with NAND or NOR gates, and using the extra input lines to control the circuit.

The circuit shown below is a basic NAND latch. The inputs are generally designated "S" and "R" for "Set" and "Reset" respectively. Because the NAND inputs must normally be logic 1 to avoid affecting the latching action, the inputs are considered to be inverted in this circuit.

The outputs of any single-bit latch or memory are traditionally designated Q and Q'. In a commercial latch circuit, either or both of these may be available for use by other circuits. In any case, the circuit itself is:
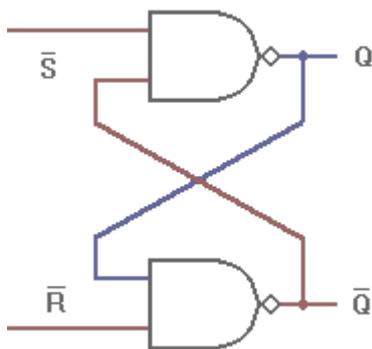
For the NAND latch circuit, both inputs should normally be at a logic 1 level. Changing an input to a logic 0 level will force that output to a logic 1. The same logic 1 will also be applied to the second input of the other NAND gate, allowing that output to fall to a logic 0 level. This in turn feeds back to the second input of the original gate, forcing its output to remain at logic 1.

Applying another logic 0 input to the same gate will have no further effect on this circuit. However, applying a logic 0 to the *other* gate will cause the same reaction in the other direction, thus changing the state of the latch circuit the other way.

Note that it is forbidden to have both inputs at a logic 0 level at the same time. That state will force both outputs to a logic 1, overriding the feedback latching action. In this condition, whichever input goes to logic 1 first will lose control, while the other input (still at logic 0) controls the resulting state of the latch. If both inputs go to logic 1 simultaneously, the result is a "race" condition, and the final state of the latch cannot be determined ahead of time.

 The same functions can also be performed using NOR gates. A few adjustments must be made to allow for the difference in the logic function, but the logic involved is quite similar.

The circuit shown below is a basic NOR latch. The inputs are generally designated "S" and "R" for "Set" and "Reset" respectively. Because the NOR inputs must normally be logic 0 to avoid overriding the latching action, the inputs are not inverted in this circuit. The NOR-based latch circuit is:

For the NOR latch circuit, both inputs should normally be at a logic 0 level. Changing an input to a logic 1 level will force that output to a logic 0. The same logic 0 will also be applied to the second input of the other NOR gate, allowing that output to rise to a logic 1 level. This in turn feeds back to the second input of the original gate, forcing its output to remain at logic 0 even after the external input is removed.

Applying another logic 1 input to the same gate will have no further effect on this circuit. However, applying a logic 1 to the *other* gate will cause the same reaction in the other direction, thus changing the state of the latch circuit the other way.

Note that it is forbidden to have both inputs at a logic 1 level at the same time. That state will force both outputs to a logic 0, overriding the feedback latching action. In this condition, whichever input goes to logic 0 first will lose control, while the other input (still at logic 1) controls the resulting state of the latch. If both inputs go to logic 0 simultaneously, the result is a "race" condition, and the final state of the latch cannot be determined ahead of time.
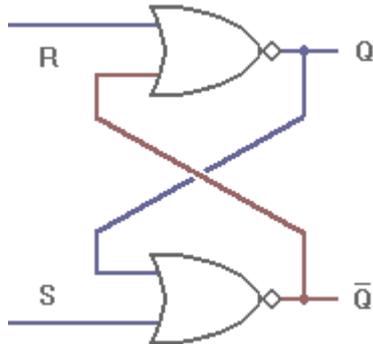
One problem with the basic RS NOR latch is that the input signals actively drive their respective outputs to a logic 0, rather than to a logic 1. Thus, the S input signal is applied to the gate that produces the Q' output, while the R input signal is applied to the gate that produces the Q output. The circuit works fine, but this reversal of inputs can be confusing when you first try to deal with NOR-based circuits.

Flip-flops

Latches are asynchronous, which means that the output changes very soon after the input changes. Most computers today, on the other hand, are synchronous, which means that the outputs of all the sequential circuits change simultaneously to the rhythm of a global clock signal.

A flip-flop is a synchronous version of the latch.

A flip-flop circuit can be constructed from two NAND gates or two NOR gates. These flip-flops are shown in Figure 2 and Figure 3. Each flip-flop has two outputs, Q and Q', and two inputs, set and reset. This type of flip-flop is referred to as an SR flip-flop or SR latch. The flip-flop in Figure 2 has two useful states. When Q=1 and Q'=0, it is in the set state (or 1-state). When Q=0 and Q=1, it is in the clear state (or 0 -state). The outputs Q and Q are complements of each other and are referred to as the normal and complement outputs, respectively. The binary state of the flip-flop is taken to be the value of the normal output.

When a 1 is applied to both the set and reset inputs of the flip-flop in Figure 2, both Q and Q outputs go to 0. This condition violates the fact that both outputs are complements of each other. In normal operation this condition must be avoided by making sure that 1's are not applied to both inputs simultaneously.
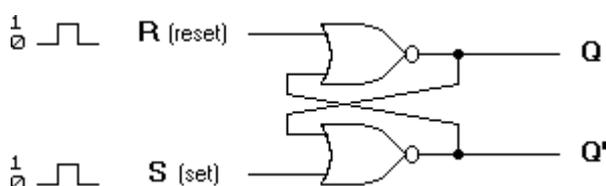
(a) Logic diagram

```
S R │ Q Q'
1 0 │ 1 0
0 0 │ 1 0   (after S=1, R=0)
0 1 │ 0 1
0 0 │ 0 1   (after S=0, R=1)
1 1 │ 0 0
```

(b) Truth table

Figure 2. Basic flip-flop circuit with NOR gates
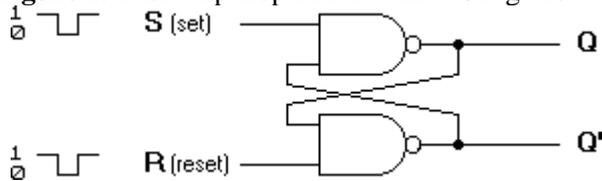
(a) Logic diagram

```
S R │ Q Q'
1 0 │ 0 1
1 1 │ 0 1   (after S=1, R=0)
0 1 │ 1 0
1 1 │ 1 0   (after S=0, R=1)
0 0 │ 1 1
```

(b) Truth table

Figure 3. Basic flip-flop circuit with NAND gates

The NAND basic flip-flop circuit in Figure 3(a) operates with inputs normally at 1 unless the state of the flip-flop has to be changed. A 0 applied momentarily to the set input causes Q to go to 1 and Q′ to go to 0, putting the flip-flop in the set state. When both inputs go to 0, both outputs go to 1. This condition should be avoided in normal operation.

Clocked SR Flip-Flop

The clocked SR flip-flop shown in Figure 4 consists of a basic NOR flip-flop and two AND gates. The outputs of the two AND gates remain at 0 as long as the clock pulse (or CP) is 0, regardless of the S and R input values. When the clock pulse goes to 1, information from the S and R inputs passes through to the basic flip-flop. With both S=1 and R=1, the occurrence of a clock pulse causes both outputs to momentarily go to 0. When the pulse is removed, the state of the flip-flop is indeterminate, ie., either state may result, depending on whether the set or reset input of the flip-flop remains a 1 longer than the transition to 0 at the end of the pulse.

(a) Logic diagram

Digital System Design   *17EE35*
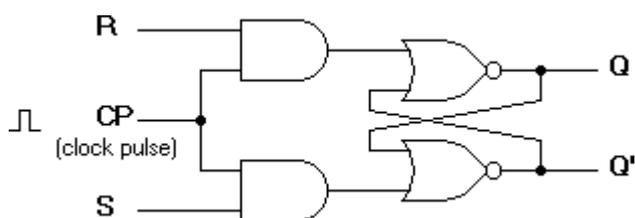
```
Q S R │ Q(t+1)
0 0 0 │    0
0 0 1 │    0
0 1 0 │    1
0 1 1 │ indeterminate
1 0 0 │    1
1 0 1 │    0
1 1 0 │    1
1 1 1 │ indeterminate
```

(b) Truth table

Figure 4. Clocked SR flip-flop

D Flip-Flop

The D flip-flop shown in Figure 5 is a modification of the clocked SR flip-flop. The D input goes directly into the S input and the complement of the D input goes to the R input. The D input is sampled during the occurrence of a clock pulse. If it is 1, the flip-flop is switched to the set state (unless it was already set). If it is 0, the flip-flop switches to the clear state.
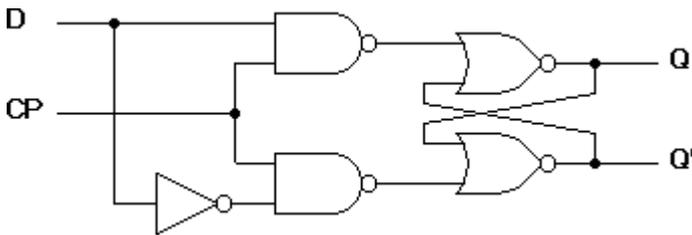
(a) Logic diagram with NAND gates
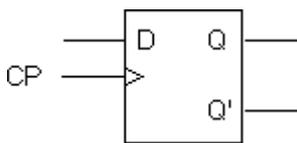
(b) Graphical symbol

```
Q D │ Q(t+1)
0 0 │    0
0 1 │    1
1 0 │    0
1 1 │    1
```

(c) Transition table

Figure 5. Clocked D flip-flop

JK Flip-Flop

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate state of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop (note that in a JK flip-flop, the letter J is for set and the letter K is for clear). When logic 1 inputs are applied to both J and K simultaneously, the flip-flop switches to its complement state, ie., if Q=1, it switches to Q=0 and vice versa.

A clocked JK flip-flop is shown in Figure 6. Output Q is ANDed with K and CP inputs so that the flip-flop is cleared during a clock pulse only if Q was previously 1. Similarly, ouput Q′ is ANDed with J and CP inputs so that the flip-flop is set with a clock pulse only if Q′ was previously 1.

Digital System Design   17EE35

Note that because of the feedback connection in the JK flip-flop, a CP signal which remains a 1 (while J=K=1) after the outputs have been complemented once will cause repeated and continuous transitions of the outputs. To avoid this, the clock pulses must have a time duration less than the propagation delay through the flip-flop. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction. The same reasoning also applies to the T flip-flop presented next.
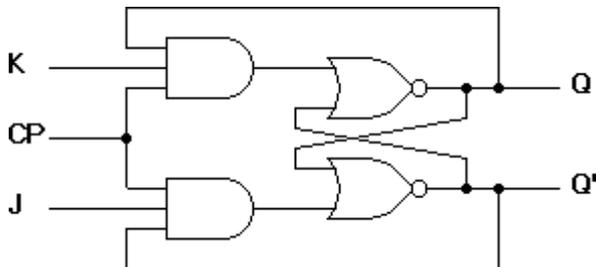
(a) Logic diagram

Q J K	Q(t+1)
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	0
1 1 0	1
1 1 1	0

(c) Transition table

Figure 6. Clocked JK flip-flop

T Flip-Flop

The T flip-flop is a single input version of the JK flip-flop. As shown in <u>Figure 7</u>, the T flip-flop is obtained from the JK type if both inputs are tied together. The output of the T flip-flop "toggles" with each clock pulse.

(a) Logic diagram

Digital System Design   *17EE35*
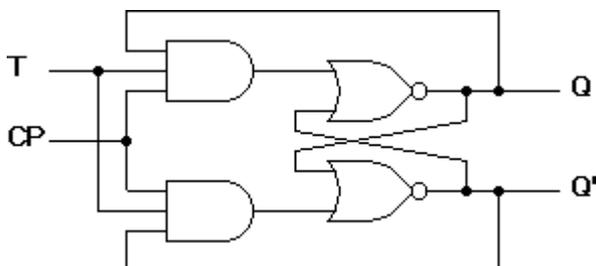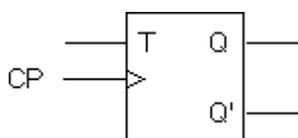
(b) Graphical symbol

Q T	Q(t+1)
0 0	0
0 1	1
1 0	1
1 1	0

(c) Transition table

Triggering of Flip-flops

The state of a flip-flop is changed by a momentary change in the input signal. This change is called a trigger and the transition it causes is said to trigger the flip-flop. The basic circuits of Figure 2 and Figure 3 require an input trigger defined by a change in signal level. This level must be returned to its initial level before a second trigger is applied. Clocked flip-flops are triggered by pulses.

The feedback path between the combinational circuit and memory elements in Figure 1 can produce instability if the outputs of the memory elements (flip-flops) are changing while the outputs of the combinational circuit that go to the flip-flop inputs are being sampled by the clock pulse. A way to solve the feedback timing problem is to make the flip-flop sensitive to the pulse transition rather than the pulse duration.

The clock pulse goes through two signal transitions: from 0 to 1 and the return from 1 to 0. As shown in Figure 8 the positive transition is defined as the positive edge and the negative transition as the negative edge.
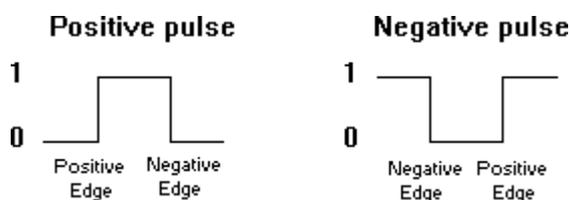
Figure 8. Definition of clock pulse transition

The clocked flip-flops already introduced are triggered during the positive edge of the pulse, and the state transition starts as soon as the pulse reaches the logic-1 level. If the other inputs change while the clock is still 1, a new output state may occur. If the flip-flop is made to respond to the positive (or negative) edge transition only, instead of the entire pulse duration, then the multiple-transition problem can be eliminated.

Master-Slave Flip-Flop

A master-slave flip-flop is constructed from two seperate flip-flops. One circuit serves as a master and the other as a slave. The logic diagram of an SR flip-flop is shown in Figure 9. The master flip-flop is enabled on the positive edge of the clock pulse CP and the slave flip-flop is disabled by the inverter. The information at the external R and S inputs is transmitted to the master flip-flop. When the pulse returns to 0, the master flip-flop is disabled and the slave flip-flop is enabled. The slave flip-flop then goes to the same state as the master flip-flop.
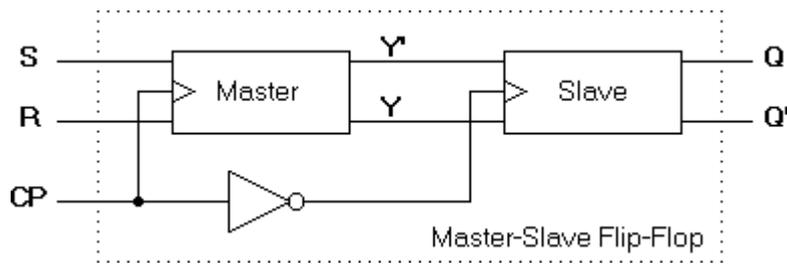
Figure 9. Logic diagram of a master-slave flip-flop
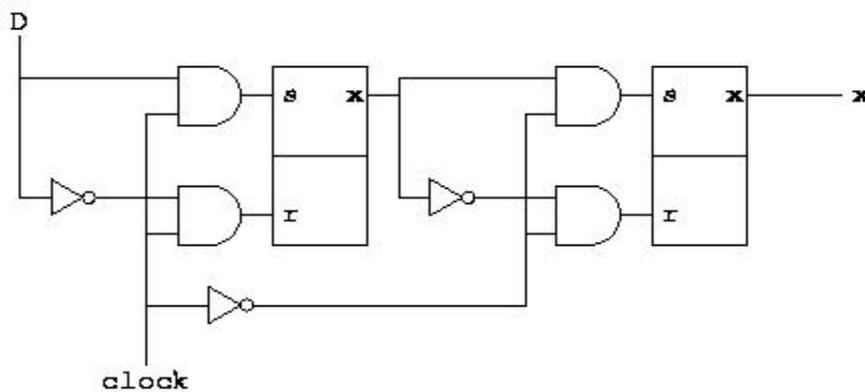
Master slave RS flip flop

The timing relationship is shown in Figure 10 and is assumed that the flip-flop is in the clear state prior to the occurrence of the clock pulse. The output state of the master-slave flip-flop occurs on the negative transition of the clock pulse. Some master-slave flip-flops change output state on the positive transition of the clock pulse by having an additional inverter between the CP terminal and the input of the master.
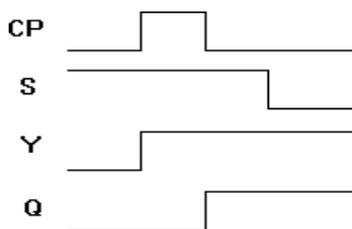
Figure 10. Timing relationship in a master slave flip-flop

Edge Triggered Flip-Flop

Another type of flip-flop that synchronizes the state changes during a clock pulse transition is the edge-triggered flip-flop. When the clock pulse input exceeds a specific threshold level, the inputs are locked out and the flip-flop is not affected by further changes in the inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the positive edge of the clock pulse (positive-edge-triggered), and others on the negative edge of the pulse (negative-edge-triggered). The logic diagram of a D-type positive-edge-triggered flip-flop is shown in Figure 11.
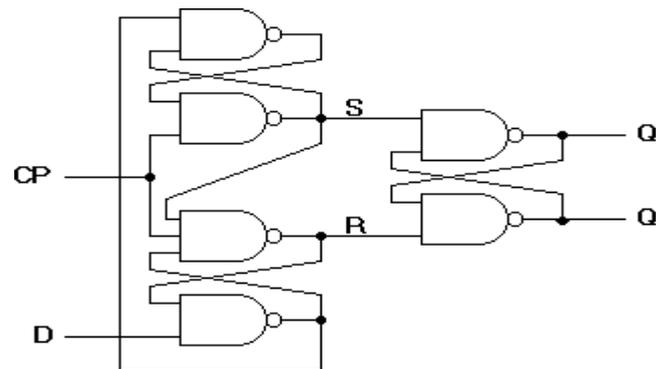
Figure 11. D-type positive-edge triggered flip-flop

When using different types of flip-flops in the same circuit, one must ensure that all flip-flop outputs make their transitions at the same time, ie., during either the negative edge or the positive edge of the clock pulse.

Direct Inputs

Flip-flops in IC packages sometimes provide special inputs for setting or clearing the flip-flop asynchronously. They are usually called preset and clear. They affect the flip-flop without the need for a clock pulse. These inputs are useful for bringing flip-flops to an intial state before their clocked operation. For example, after power is turned on in a digital system, the states of the flip-flops are indeterminate. Activating the clear input clears all the flip-flops to an initial state of 0. The graphic symbol of a JK flip-flop with an active-low clear is shown in Figure 12.
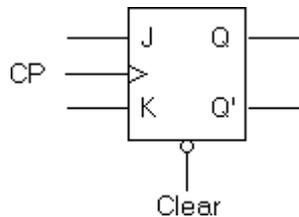
(a) Graphic Symbol

Inputs				Outputs	
Clear	Clock	J	K	Q	Q'
0	x	x	x	0	1
1	$0 \to 1$	0	0	No change	
1	$0 \to 1$	0	1	0	1
1	$0 \to 1$	1	0	1	0
1	$0 \to 1$	1	1	Toggle	

(b) Transition table

Figure 12. JK flip-flop with direct clear

Synchronous and asynchronous sequential circuit

asynchronous system is a system whose outputs depend upon the order in which its input variables change and can be affected at any instant of time.

Gate-type asynchronous systems are basically combinational circuits with feedback paths. Because of the feedback among logic gates, the system may, at times, become unstable. Consequently they are not often used.

Synchronous type of system uses storage elements called flip-flops that are employed to change their binary value only at discrete instants of time. Synchronous sequential circuits use logic gates and flip-flop storage devices. Sequential circuits have a clock signal as one of their inputs. All state transitions in such circuits occur only when the clock value is either 0 or 1 or happen at the rising or falling edges of the clock depending on the type of memory elements used in the circuit. Synchronization is achieved by a timing device called a clock pulse generator. Clock pulses are distributed throughout the system in such a way that the flip-flops are affected only with the arrival of the synchronization pulse. Synchronous sequential circuits that use clock pulses in the inputs are called clocked-sequential circuits. They are stable and their timing can easily be broken down into independent discrete steps, each of which is considered separately.

A clock signal is a periodic square wave that indefinitely switches from 0 to 1 and from 1 to 0 at fixed intervals. Clock cycle time or clock period: the time interval between two consecutive rising or falling edges of the clock.

Moore and Mealy model of sequential circuit

Mealy and Moore models are the basic models of state machines. A state machine which uses only Entry Actions, so that its output depends on the state, is called a Moore model. A state machine which uses only Input Actions, so that the output depends on the state and also on inputs, is called a Mealy model. The models selected will influence a design but there are no general indications as to which model is better. Choice of a model depends on the application, execution means (for instance, hardware systems are usually best realised as Moore models) and personal preferences of a designer or programmer. In practise, mixed models are often used with several action types

Design of Sequential Circuits

The design of a synchronous sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which a logic diagram can be obtained. In contrast to a combinational logic, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalence representation, such as a state diagram.

A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding the combinational structure which, together with the flip-flops, produces a circuit that fulfils the required specifications. The number of flip-flops is determined from the number of states needed in the circuit.

The recommended steps for the design of sequential circuits are set out below:
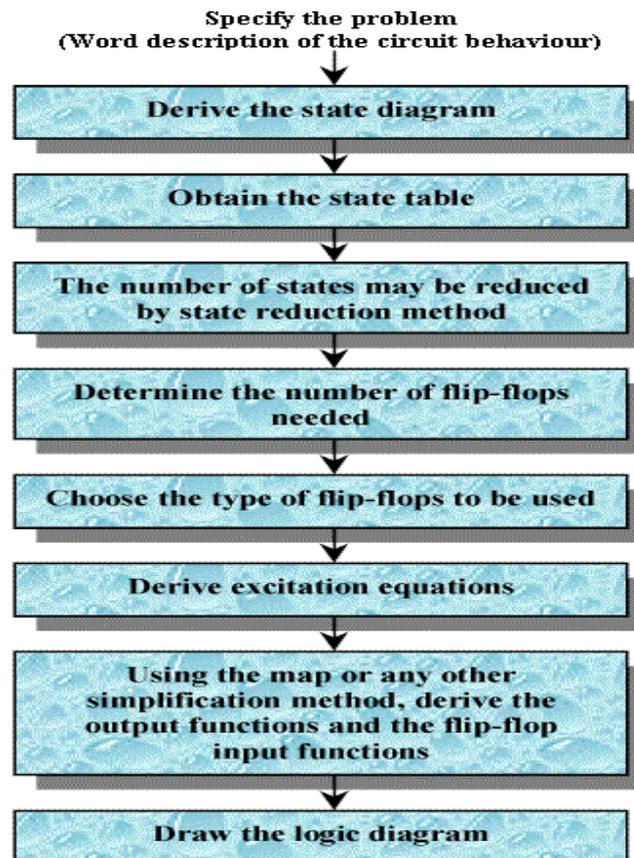
Analysis of a sequential circuit

We have examined a general model for sequential circuits. In this model the effect of all previous inputs on the outputs is represented by a state of the circuit. Thus, the output of the circuit at any time depends upon its current state and the input. These also determine the next state of the circuit. The relationship that exists among the inputs, outputs, present states and next states can be specified by either the **state table** or the **state diagram**.

State Table

The state table representation of a sequential circuit consists of three sections labelled *present state*, *next state* and *output*. The present state designates the state of flip-flops before the occurrence of a clock pulse. The next state shows the states of flip-flops after the clock pulse, and the output section lists the value of the output variables during the present state.

State Diagram

In addition to graphical symbols, tables or equations, flip-flops can also be represented graphically by a state diagram. In this diagram, a state is represented by a circle, and the transition between states is indicated by directed lines (or arcs) connecting the circles. An example of a state diagram is shown in Figure 3 below.
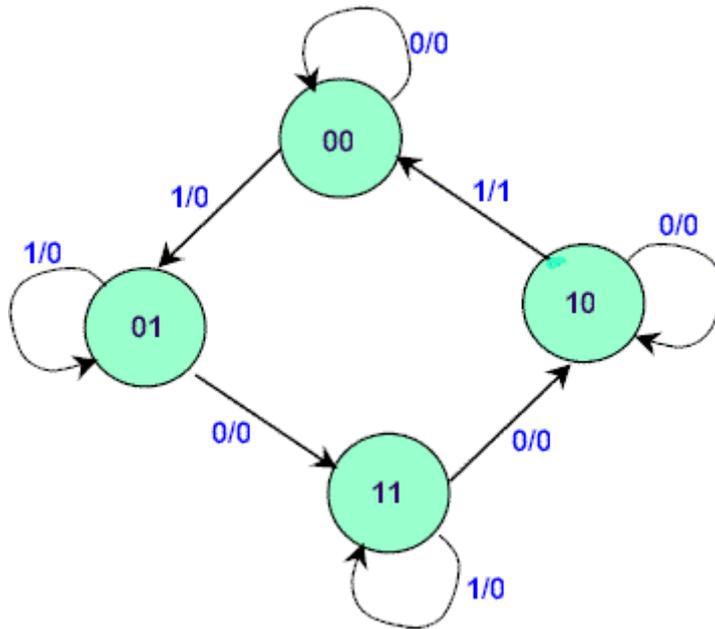
Figure 3. State Diagram

The binary number inside each circle identifies the state the circle represents. The directed lines are labelled with two binary numbers separated by a slash (/). The input value that causes the state transition is labelled first. The number after the slash symbol / gives the value of the output. For example, the directed line from state 00 to 01 is labelled 1/0, meaning that, if the sequential circuit is in a present state and the input is 1, then the next state is 01 and the output is 0. If it is in a present state 00 and the input is 0, it will remain in that state. A directed line connecting a circle with itself indicates that no change of state occurs. The state diagram provides exactly the same information as the state table and is obtained directly from the state table.

Example: Consider a sequential circuit shown in Figure 4. It has one input x, one output Z and two state variables Q_1Q_2 (thus having four possible present states 00, 01, 10, 11).
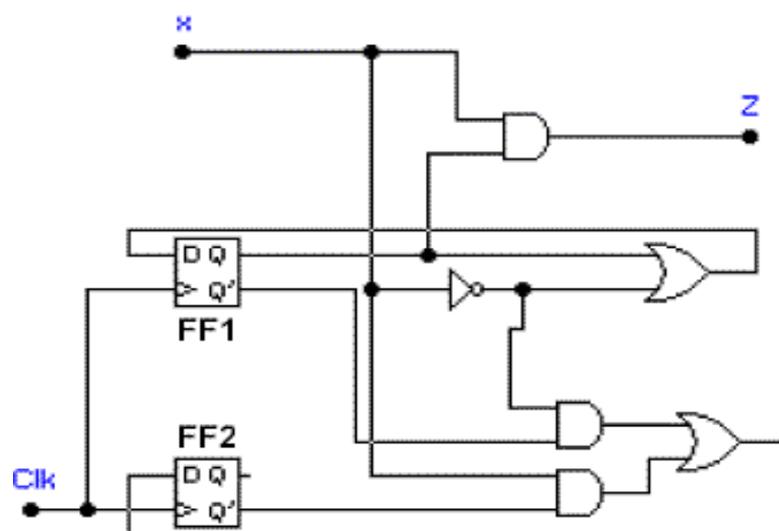

Figure 4. A Sequential Circuit

The behaviour of the circuit is determined by the following Boolean expressions:

Digital System Design   *17EE35*

$$Z = xQ_1$$
$$D_1 = x' + Q_1$$
$$D_2 = xQ_2' + x'*Q_1'$$

These equations can be used to form the state table. Suppose the present state (i.e. Q_1Q_2) = 00 and input x = 0. Under these conditions, we get Z = 0, D_1 = 1, and D_2 = 1. Thus the next state of the circuit D_1D_2 = 11, and this will be the present state after the clock pulse has been applied. The output of the circuit corresponding to the present state Q_1Q_2 = 00 and x = 1 is Z = 0. This data is entered into the state table as shown in Table 2.

Present State	Next State		Output (Z)	
Q1 Q2	X = 0 Q_1 Q_0	X = 1 Q_1 Q_0	x = 0	x = 1
0 0	1 1	0 1	0	0
0 1	1 1	0 0	0	0
1 0	1 0	1 1	0	1
1 1	1 0	1 0	0	1

Table 2. State table for the sequential circuit in Figure 4.

The state diagram for the sequential circuit in Figure 4 is shown in Figure 5.
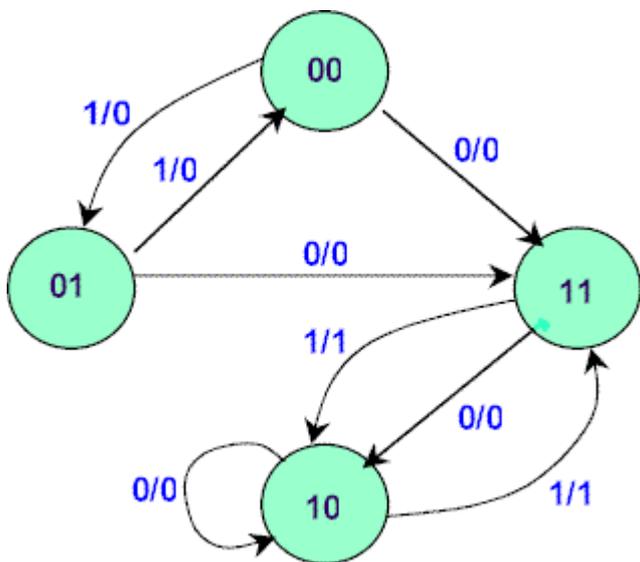
Figure 5. State Diagram of circuit in Figure 4.

State Diagrams of Various Flip-flops

Table 3 shows the state diagrams of the four types of flip-flops.

NAME	STATE DIAGRAM

Digital System Design   *17EE35*

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

|	
SR	
JK	
D	
T	
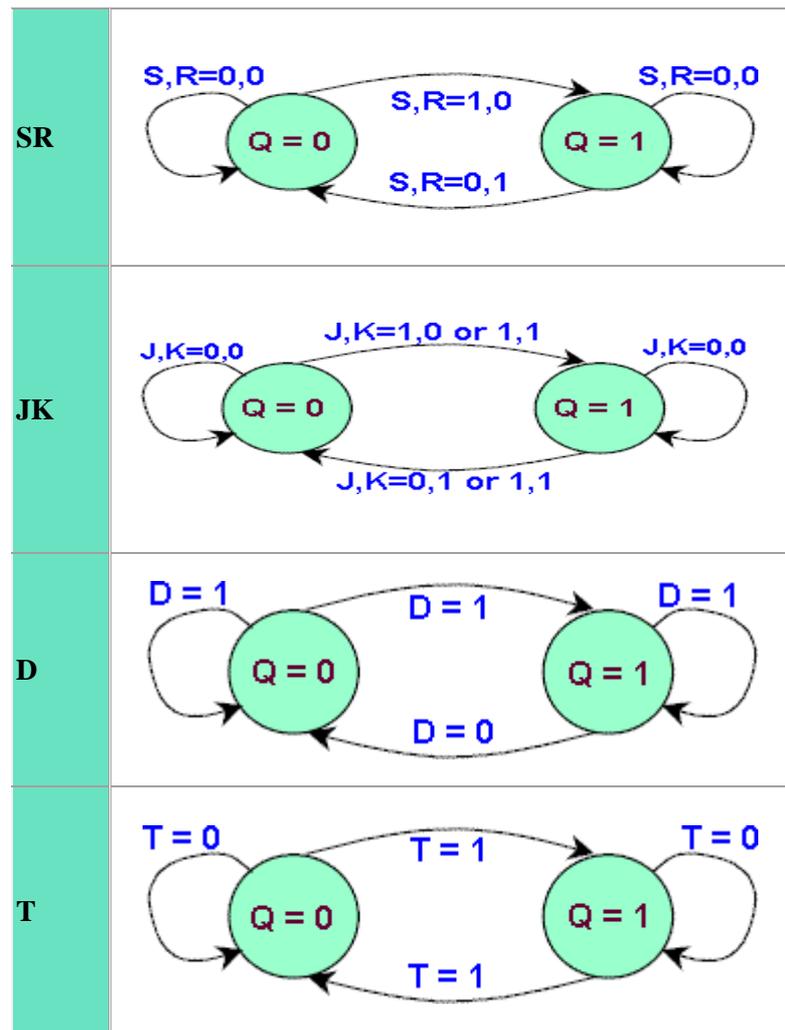
Table 3. State diagrams of the four types of flip-flops.

You can see from the table that all four flip-flops have the same number of states and transitions. Each flip-flop is in the set state when Q=1 and in the reset state when Q=0. Also, each flip-flop can move from one state to another, or it can re-enter the same state. The only difference between the four types lies in the values of input signals that cause these transitions.

A state diagram is a very convenient way to visualise the operation of a flip-flop or even of large sequential components.

Example 1.1

Derive the state table and state diagram for the sequential circuit shown in Figure 7.
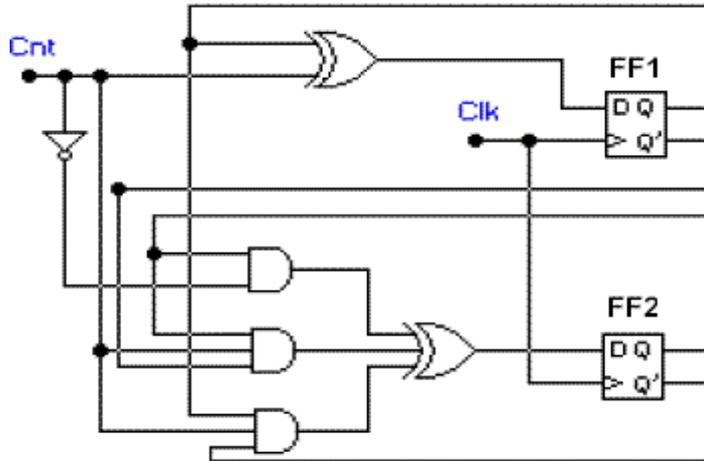
Figure 7. Logic schematic of a sequential circuit.

 SOLUTION:

STEP 1: First we derive the Boolean expressions for the inputs of each flip-flops in the schematic, in terms of external input Cnt and the flip-flop outputs Q_1 and Q_0. Since there are two D flip-flops in this example, we derive two expressions for D1 and D0:

$$D_0 = Cnt \oplus Q_0 = Cnt'Q_0 + CntQ_0'$$
$$D_1 = Cnt'Q_1 + CntQ_1'Q_0 + CntQ_1Q_0'$$

These Boolean expressions are called **excitation equations** since they represent the inputs to the flip-flops of the sequential circuit in the next clock cycle.

STEP 2: Derive the next-state equations by converting these excitation equations into flip-flop characteristic equations. In the case of D flip-flops, Q(next) = D. Therefore the next state equal the excitation equations.

$$Q_0(next) = D_0 = Cnt'Q_0 + CntQ_0'$$
$$Q_1(next) = D_1 = Cnt'Q_1 + CntQ_1'Q_0 + CntQ_1Q_0'$$

STEP 3:   Now convert these next-state equations into tabular form called the next-state table.

Present State	Next State	
Q_1 Q_0	Cnt = 0 Q_1 Q_0	Cnt = 1 Q_1 Q_0
0 0	0 0	0 1
0 1	0 1	1 0
1 0	1 0	1 1
1 1	1 1	0 0

Each row is corresponding to a state of the sequential circuit and each column represents one set of input values. Since we have two flip-flops, the number of possible states is four - that is, Q_1Q_0 can be equal to 00, 01, 10, or 11. These are present states as shown in the table.

Digital System Design   *17EE35*

For the next state part of the table, each entry defines the value of the sequential circuit in the next clock cycle after the rising edge of the **Clk**. Since this value depends on the present state and the value of the input signals, the next state table will contain one column for each assignment of binary values to the input signals. In this example, since there is only one input signal, **Cnt**, the next-state table shown has only two columns, corresponding to **Cnt = 0 and Cnt =1**.

Note that each entry in the next-state table indicates the values of the flip-flops in the next state if their value in the present state is in the row header and the input values in the column header.

Each of these next-state values has been computed from the next-state equations in STEP 2.

STEP 4:   The state diagram is generated directly from the next-state table, shown in Figure 8.
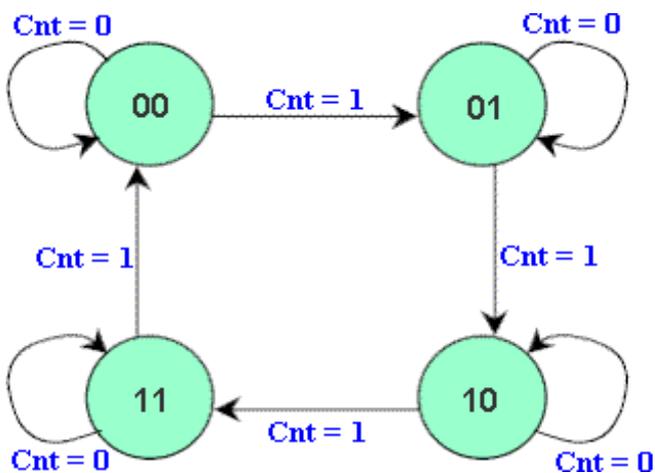

Figure 8. State diagram

Each arc is labelled with the values of the input signals that cause the transition from the present state (the source of the arc) to the next state (the destination of the arc).

Example 1.2

Derive the next state, the output table and the state diagram for the sequential circuit shown in Figure 10.
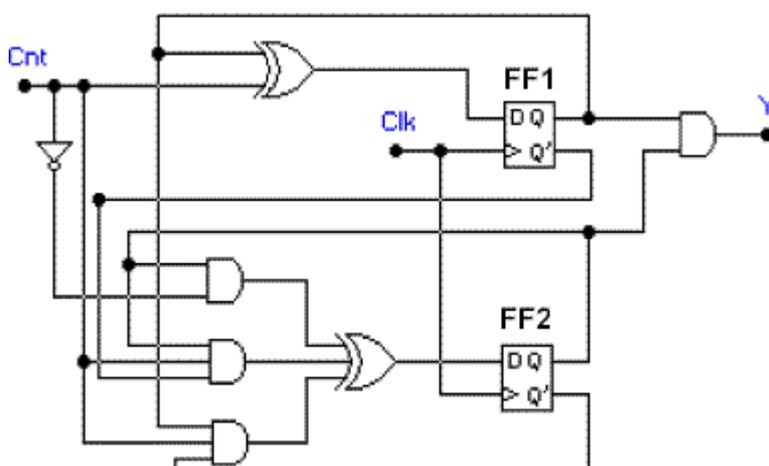

Figure 10. Logic schematic of a sequential circuit.

 SOLUTION:

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

The input combinational logic in Figure 10 is the same as in example1.1 so the excitation and the next-state equations will be the same as in Example 1.1.

Excitation equations:

$$D_0 = Cnt \oplus Q_0 = Cnt'Q_0 + CntQ_0' \qquad\qquad D_1 = Cnt'Q_1 + CntQ1'Q_0 + CntQ_1Q_0'$$

Next-state equations: $Q_0(next) = D_0 = Cnt'Q_0 + CntQ_0' \qquad Q_1(next) = D_0 = Cnt'Q_1 + CntQ_1'Q_0 + CntQ_1Q_0'$

In addition, however, we have computed the output equation.

Output equation: $\qquad Y = Q_1Q_0$

As this equation shows, the output Y will equal to 1 when the counter is in state Q1Q0 = 11, and it will stay 1 as long as the counter stays in that state.

Next-state and output table:

Present State Q1 Q0	Next State Cnt = 0 $Q_1 Q_0$	Cnt = 1 $Q_1 Q_0$	Output Z
0 0	0 0	0 1	0
0 1	0 1	1 0	0
1 0	1 0	1 1	0
1 1	1 1	0 0	1

State diagram:

Fig   0.

Sta

An   ising the cost of the final circuit. The two most obvious cost red   the number of gates.

Th   elated to the complexity of the resulting circuit. It is therefore des   ent in all aspects. The process of eliminating the equivalent or red   **state reduction**.
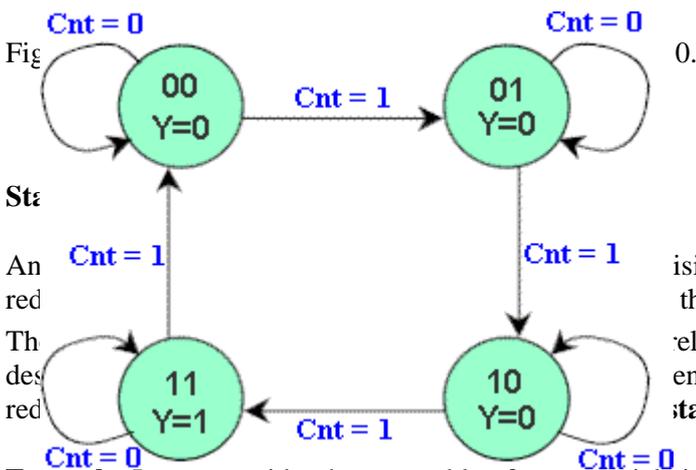
Example: Let us consider the state table of a sequential circuit shown in Table 6.

Present State	Next State x = 0	x = 1	Output x = 0	x = 1

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

A	B	C	1	0
B	F	D	0	0
C	D	E	1	1
D	F	E	0	1
E	A	D	0	0
F	B	C	1	0

Table 6. State table

It can be seen from the table that the present state A and F both have the same next states, B (when x=0) and C (when x=1). They also produce the same output 1 (when x=0) and 0 (when x=1). Therefore states A and F are equivalent. Thus one of the states, A or F can be removed from the state table. For example, if we remove row F from the table and replace all F's by A's in the columns, the state table is modified as shown in Table 7.

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
A	B	C	1	0
B	A	D	0	0
C	D	E	1	1
D	A	E	0	1
E	A	D	0	0

Table 7. State F removed

It is apparent that states B and E are equivalent. Removing E and replacing E's by B's results in the reduce table shown in Table 8.

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
A	B	C	1	0
B	A	D	0	0
C	D	B	1	1
D	A	B	0	1

Table 8. Reduced state table

The removal of equivalent states has reduced the number of states in the circuit from six to four. Two states are considered to be **equivalent** if and only if for every input sequence the circuit produces the same output sequence irrespective of which one of the two states is the starting state.

Example 1.3

We wish to design a synchronous sequential circuit whose state diagram is shown in Figure 13. The type of flip-flop to be use is J-K.
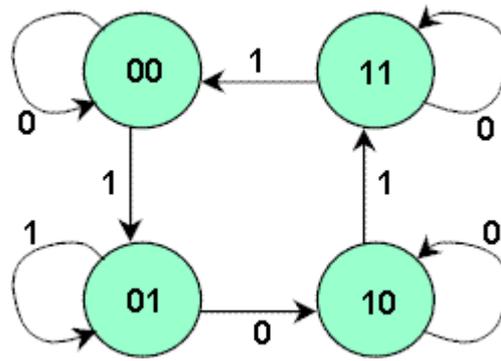
Figure 13. State diagram

From the state diagram, we can generate the state table shown in Table 9. Note that there is no output section for this circuit. Two flip-flops are needed to represent the four states and are designated Q0Q1. The input variable is labelled x.

Present State	Next State	
Q0 Q1	X = 0 $Q_0 Q_1$	X = 1 $Q_0 Q_1$
0 0	0 0	0 1
0 1	1 0	0 1
1 0	1 0	1 1
1 1	1 1	0 0

Table 9. State table.

We shall now derive the excitation table and the combinational structure. The table is now arranged in a different form shown in Table 11, where the present state and input variables are arranged in the form of a truth table. Remember, the excitable for the JK flip-flop was derive in table 1

Table 10. Excitation table for JK flip-flop

Output Transitions	Flip-flop inputs
Q →Q(next)	J K
0 → 0	0 X
0 →1	1 X
1 → 0	X 1
1 →1	X 0

Table 11. Excitation table of the circuit

Present State	Input	Next State	Flip-flop Inputs	
Q0 Q1	x	Q0 Q1	J0 K0	J1 K1

Digital System Design   *17EE35*

0 0	0	0 0	0	X	0	X
0 0	1	0 1	0	X	1	X
0 1	0	1 0	1	X	X	1
0 1	1	0 1	0	X	X	0
1 0	0	1 0	X	0	0	X
1 0	1	1 1	X	0	1	X
1 1	0	1 1	X	0	X	0
1 1	1	0 0	X	1	X	1

In the first row of Table 11, we have a transition for flip-flop Q0 from 0 in the present state to 0 in the next state. In Table 10 we find that a transition of states from 0 to 0 requires that input J = 0 and input K = X. So 0 and X are copied in the first row under J0 and K0 respectively. Since the first row also shows a transition for the flip-flop Q1 from 0 in the present state to 0 in the next state, 0 and X are copied in the first row under J1 and K1. This process is continued for each row of the table and for each flip-flop, with the input conditions as specified in Table 10.

The flip-flop input functions are derived:

$J0 = Q1*x'$ $K0 = Q1*x$
$J1 = x$ $K1 = Q0'*x' + Q0*x = Q0 \odot x$

Note: the symbol $\odot$ is exclusive-NOR.

The logic diagram is drawn in Figure 15.

Figure 15. Logic diagram of the sequential circuit.

Example 1.4 Design a sequential circuit whose state tables are specified in Table 12, using D flip-flops.

Table 12. State table of a sequential circuit.

Digital System Design   *17EE35*

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design

Present State	Next State		Output	
Q0 Q1	$X = 0$ Q_0 Q_1	$X = 1$ Q_0 Q_1	x = 0	x = 1
0 0	0 0	0 1	0	0
0 1	0 0	1 0	0	0
1 0	1 1	1 0	0	0
1 1	0 0	0 1	0	1

Table 13. Excitation table for a D flip-flop.

Output Transitions Q→Q(next)	Flip-flop inputs D
0 → 0	0
0 → 1	1
1 → 0	0
1 → 1	1

Next step is to derive the excitation table for the design circuit, which is shown in Table 14. The output of the circuit is labelled Z.

Present State Q0 Q1	Next State Q0 Q1		Input x	Flip-flop Inputs D0 D1		Output Z
0 0	0 0		0	0	0	0
0 0	0 1		1	0	1	0
0 1	0 0		0	0	0	0
0 1	1 0		1	1	0	0
1 0	1 1		0	1	1	0
1 0	1 0		1	1	0	0
1 1	0 0		0	0	0	0
1 1	0 1		1	0	1	1

Table 14. Excitation table

Now plot the flip-flop inputs and output functions on the Karnaugh map to derive the Boolean expressions, which is shown in Figure 16.

Digital System Design   *17EE35*
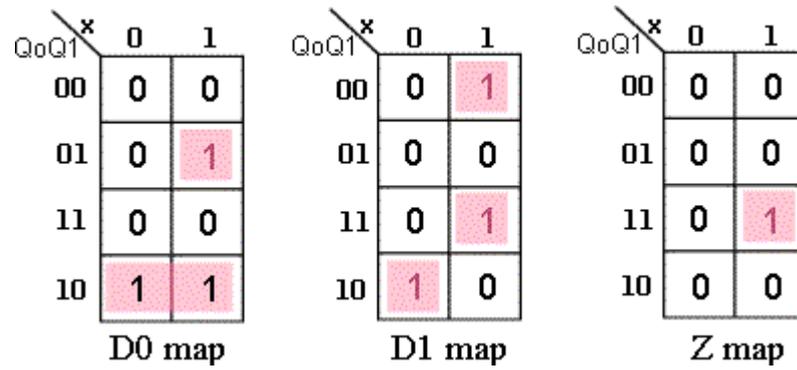
Figure 16. Karnaugh maps

The simplified Boolean expressions are:

$$D_0 = Q_0 * Q_1' + Q_0' * Q_1 * x$$
$$D_1 = Q_0' * Q_1' * x + Q_0 * Q_1 * x + Q_0 * Q_1' * x'$$
$$Z = Q_0 * Q_1 * x$$

Finally, draw the logic diagram.
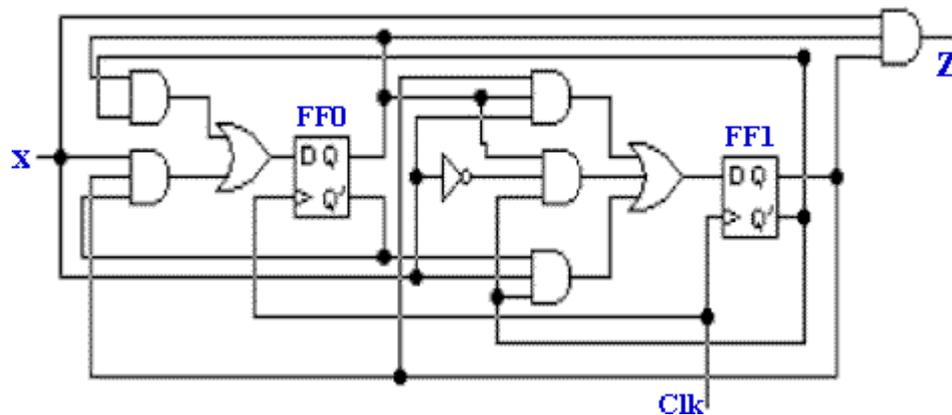
Figure 17. Logic diagram of the sequential circuit.

Register

A *register* is a sequential circuit with *n + 1* (not counting the clock) inputs and *n* output. To each of the outputs corresponds an input. The first *n* inputs will be called x_0 trough x_{n-1} and the last input will be called ld (for load). The *n* outputs will be called y_0 trough y_{n-1}.

When the ld input is 0, the outputs are uneffected by any clock transition. When the ld input is 1, the x inputs are stored in the register at the next clock transition, making the y outputs into copies of the x inputs before the clock transition.

We can explain this behavior more formally with a state table. As an example, let us take a register with *n = 4*. The left side of the state table contains 9 columns, labeled x0, x1, x2, x3, ld, y0, y1, y2, and y3. This means that the state table has 512 rows. We will therefore abbreviate it. Here it is:

```
 ld x3 x2 x1 x0 y3 y2 y1 y0 | y3′ y2′ y1′ y0′
 -------------------------------------------
  0 -- -- -- -- c3 c2 c1 c0 | c3  c2  c1  c0

  1 c3 c2  c1   c0 -- -- -- -- | c3  c2  c1  c0
```

As you can see, when ld is 0 (the top half of the table), the right side of the table is a copy of the values of the old outputs, independently of the inputs. When ld is 1, the right side of the table is instead a copy of the values of the inputs, independently of the old values of the outputs.

Registers play an important role in computers. Some of them are visible to the programmer, and are used to hold variable values for later use. Some of them are hidden to the programmer, and are used to hold values that are internal to the central processing unit, but nevertheless important.

Shift registers

Shift registers are a type of sequential logic circuit, mainly for storage of digital data. They are a group of flip-flops connected in a chain so that the output from one flip-flop becomes the input of the next flip-flop. Most of the registers possess no characteristic internal sequence of states. All the flip-flops are driven by a common clock, and all are set or reset simultaneously.

In this section, the basic types of shift registers are studied, such as Serial In - Serial Out, Serial In - Parallel Out, Parallel In - Serial Out, Parallel In - Parallel Out, and bidirectional shift registers. A special form of counter - the shift register counter, is also introduced.

Serial In - Serial Out Shift Registers

A basic four-bit shift register can be constructed using four D flip-flops, as shown below. The operation of the circuit is as follows. The register is first cleared, forcing all four outputs to zero. The input data is then applied sequentially to the D input of the first flip-flop on the left (FF0). During each clock pulse, one bit is transmitted from left to right. Assume a data word to be 1001. The least significant bit of the data has to be shifted through the register from FF0 to FF3.
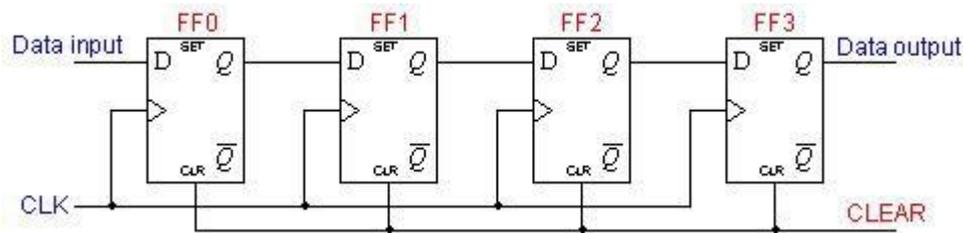
In order to get the data out of the register, they must be shifted out serially. This can be done destructively or non-destructively. For destructive readout, the original data is lost and at the end of the read cycle, all flip-flops are reset to zero.

To avoid the loss of data, an arrangement for a non-destructive reading can be done by adding two AND gates, an OR gate and an inverter to the system. The construction of this circuit is shown below.
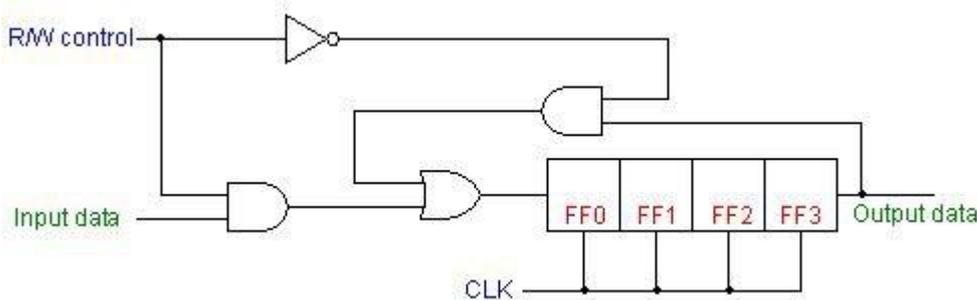

The data is loaded to the register when the control line is HIGH (ie WRITE). The data can be shifted out of the register when the control line is LOW (ie READ)

Serial In - Parallel Out Shift Registers

For this kind of register, data bits are entered serially in the same manner as discussed in the last section. The difference is the way in which the data bits are taken out of the register. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously. A construction of a four-bit serial in - parallel out register is shown below.
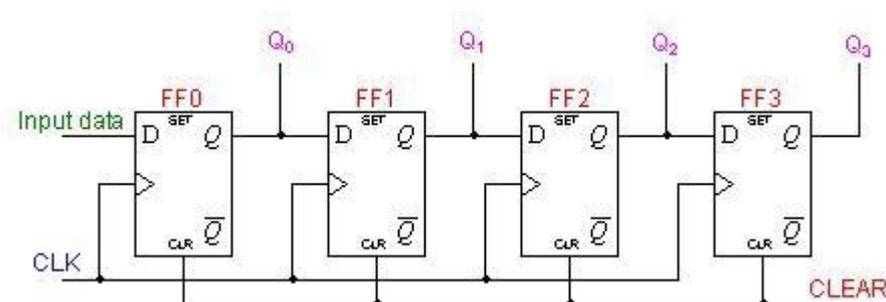
A four-bit parallel in - serial out shift register is shown below. The circuit uses D flip-flops and NAND gates for entering data (ie writing) to the register.
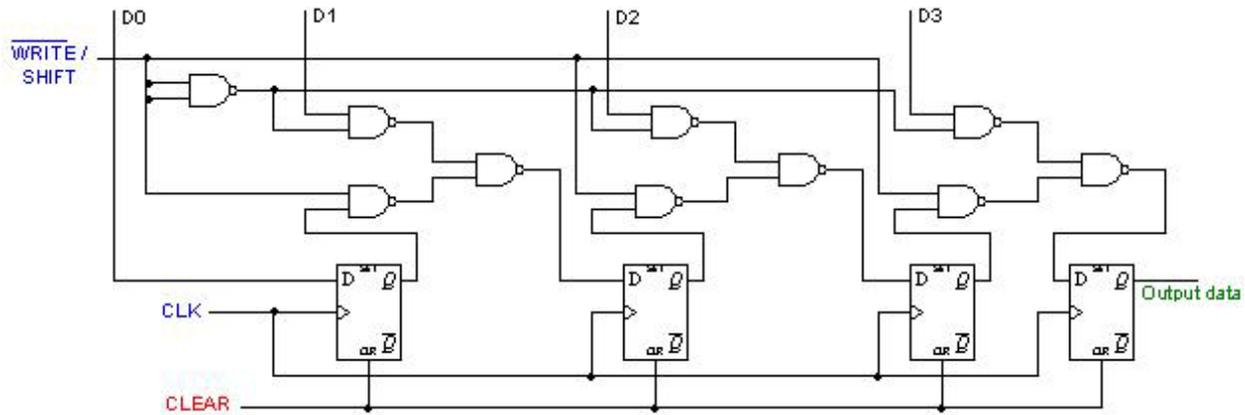
D0, D1, D2 and D3 are the parallel inputs, where D0 is the most significant bit and D3 is the least significant bit. To write data in, the mode control line is taken to LOW and the data is clocked in. The data can be shifted when the mode control line is HIGH as SHIFT is active high

Parallel In - Parallel Out Shift Registers

For parallel in - parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. The following circuit is a four-bit parallel in - parallel out shift register constructed by D flip-flops.
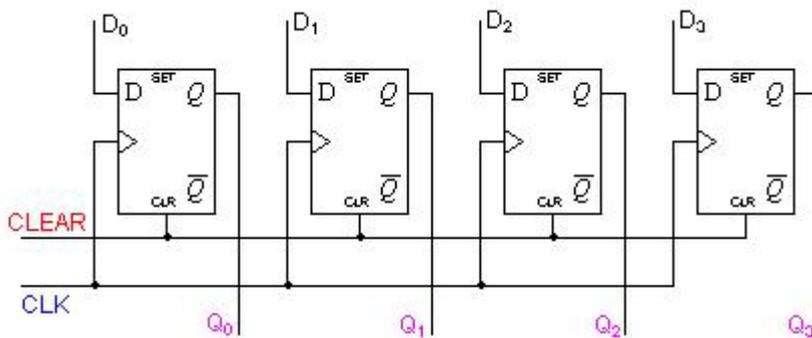
The D's are the parallel inputs and the Q's are the parallel outputs. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously**.**

Bidirectional Shift Registers

The registers discussed so far involved only right shift operations. Each right shift operation has the effect of successively dividing the binary number by two. If the operation is reversed (left shift), this has the effect of multiplying the number by two. With suitable gating arrangement a serial shift register can perform both operations.

A *bidirectional*, or *reversible*, shift register is one in which the data can be shift either left or right. A four-bit bidirectional shift register using D flip-flops is shown below.

S J P N Trust's
Hirasugar Institute of Technology, Nidasoshi.
Inculcating Values, Promoting Prosperity
Approved by AICTE, Recognized by Govt. of Karnataka and Affiliated to VTU Belagavi.

Electrical
Academic
Notes
Digital System Design
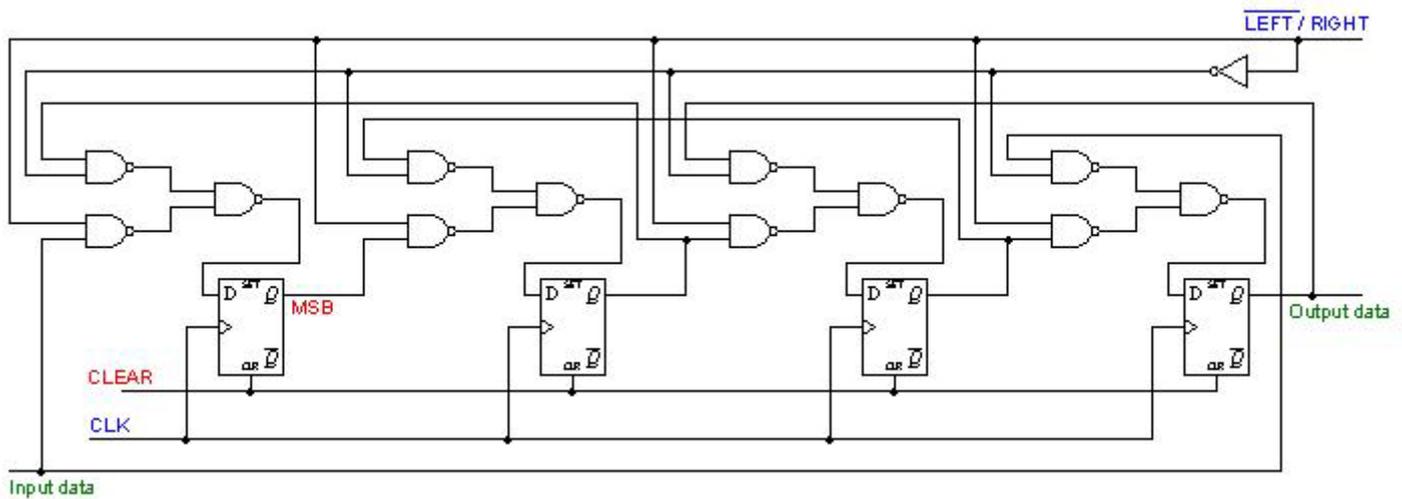
Here a set of NAND gates are configured as OR gates to select data inputs from the right or left adjacent bistables, as selected by the LEFT/RIGHT control line.

Shift Register Counters

Two of the most common types of shift register counters are introduced here: the Ring counter and the Johnson counter. They are basically shift registers with the serial outputs connected back to the serial inputs in order to produce particular sequences. These registers are classified as counters because they exhibit a specified sequence of states.

Ring Counters

A ring counter is basically a circulating shift register in which the output of the most significant stage is fed back to the input of the least significant stage. The following is a 4-bit ring counter constructed from D flip-flops. The output of each stage is shifted into the next stage on the positive edge of a clock pulse. If the CLEAR signal is high, all the flip-flops except the first one FF0 are reset to 0. FF0 is preset to 1 instead.
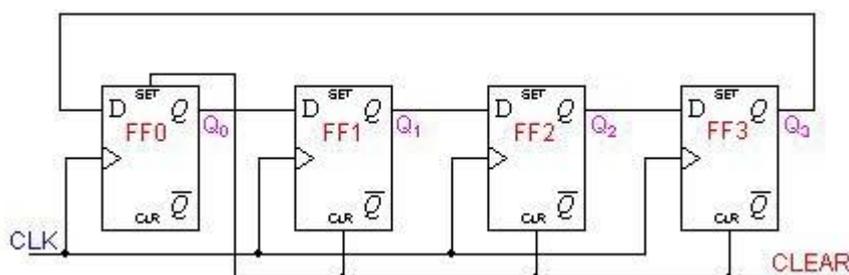

Since the count sequence has 4 distinct states, the counter can be considered as a mod-4 counter. Only 4 of the maximum 16 states are used, making ring counters very inefficient in terms of state usage. But the major advantage of a ring counter over a binary counter is that it is self-decoding. No extra decoding circuit is needed to determine what state the counter is in.

Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

Johnson Counters

Johnson counters are a variation of standard ring counters, with the inverted output of the last stage fed back to the input of the first stage. They are also known as twisted ring counters. An n-stage Johnson counter yields a count sequence of length $2n$, so it may be considered to be a mod-$2n$ counter. The circuit above shows a 4-bit Johnson counter. The state sequence for the counter is given in the table

Digital System Design   17EE35
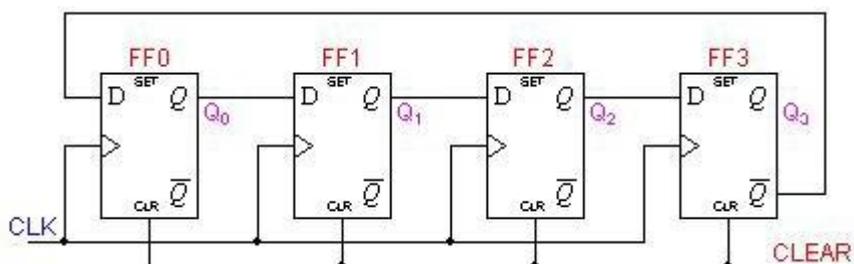
Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0
6	1	1	0	0
7	1	0	0	0

Again, the apparent disadvantage of this counter is that the maximum available states are not fully utilized. Only eight of the sixteen states are being used**.**

Counters

A sequential circuit that goes through a prescribed sequence of states upon the application of input pulses is called a **counter**. The input pulses, called **count pulses**, may be clock pulses. In a counter, the sequence of states may follow a binary count or any other sequence of states. Counters are found in almost all equipment containing digital logic. They are used for counting the number of occurrences of an even and are useful for generating timing sequences to control operations in a digital system.

A *counter* is a sequential circuit with *0* inputs and *n* outputs. Thus, the value after the clock transition depends only on old values of the outputs. For a counter, the values of the outputs are interpreted as a sequence of *binary digits* (see the section on binary arithmetic).

We shall call the outputs o0, o1, ..., on-1. The value of the outputs for the counter after a clock transition is a binary number which is *one plus* the binary number of the outputs before the clock transition.

We can explain this behavior more formally with a state table. As an example, let us take a counter with $n = 4$. The left side of the state table contains 4 columns, labeled o0, o1, o2, and o3. This means that the state table has 16 rows. Here it is in full:

```
o3 o2 o1 o0 | o3′ o2′ o1′ o0′
----------------------------
 0 0 0 0 | 0  0  0  1
 0 0 0 1 | 0  0  1  0
 0 0 1 0 | 0  0  1  1
 0 0 1 1 | 0  1  0  0
 0 1 0 0 | 0  1  0  1
 0 1 0 1 | 0  1  1  0
 0 1 1 0 | 0  1  1  1
 0 1 1 1 | 1  0  0  0
 1 0 0 0 | 1  0  0  1
 1 0 0 1 | 1  0  1  0
 1 0 1 0 | 1  0  1  1
 1 0 1 1 | 1  1  0  0
 1 1 0 0 | 1  1  0  1
 1 1 0 1 | 1  1  1  0
 1 1 1 0 | 1  1  1  1
 1 1 1 1 | 0  0  0  0
```
As you can see, the right hand side of the table is always one plus the value of the left hand side of the table, except for

the last line, where the value is 0 for all the outputs. We say that the counter *wraps around*.

Counters (with some variations) play an important role in computers. Some of them are visible to the programmer, such as the program counter (PC). Some of them are hidden to the programmer, and are used to hold values that are internal to the central processing unit, but nevertheless important.

Important variations include:

- The ability to count up or down according to the value of an additional input
- The ability to count or not according the the value of an additional input
- The ability to clear the contents of the counter if some additional input is 1
- The ability to act as a register as well, so that a predetermined value is loaded when some additional input is 1
- The ability to count using a different representation of numbers from the normal (such as Gray-codes, 7-segment codes, etc)
- The ability to count with different increments that 1

Design of Counters

Example 1.5 A counter is first described by a state diagram, which is shows the sequence of states through which the counter advances when it is clocked. Figure 18 shows a state diagram of a 3-bit binary counter.
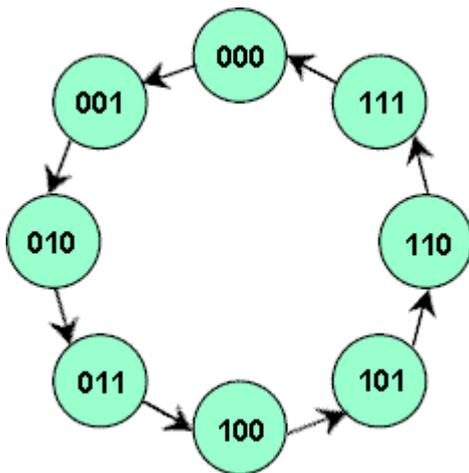

Figure 18. State diagram of a 3-bit binary counter.

The circuit has no inputs other than the clock pulse and no outputs other than its internal state (outputs are taken off each flip-flop in the counter). The next state of the counter depends entirely on its present state, and the state transition occurs every time the clock pulse occurs. Figure 19 shows the sequences of count after each clock pulse.

Digital System Design   *17EE35*
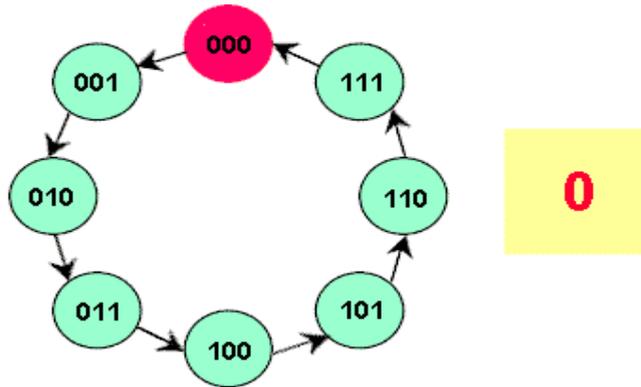
Clk

Once the sequential circuit is defined by the state diagram, the next step is to obtain the next-state table, which is derived from the state diagram in Figure 18 and is shown in Table 15.

Table 15. State table

Present State Q2 Q1 Q0	Next State Q2 Q1 Q0
0 0 0	0 0 1
0 0 1	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 0 0	1 0 1
1 0 1	1 1 0
1 1 0	1 1 1
1 1 1	0 0 0

Since there are eight states, the number of flip-flops required would be three. Now we want to implement the counter design using JK flip-flops.

Next step is to develop an excitation table from the state table, which is shown in Table 16.

Table 16. Excitation table

Output State Transitions		Flip-flop inputs		
Present State Q2 Q1 Q0	Next State Q2 Q1 Q0	J2 K2	J1 K1	J0 K0
0 0 0	0 0 1	0 X	0 X	1 X
0 0 1	0 1 0	0 X	1 X	X 1
0 1 0	0 1 1	0 X	X 0	1 X
0 1 1	1 0 0	1 X	X 1	X 1
1 0 0	1 0 1	X 0	0 X	1 X
1 0 1	1 1 0	X 0	1 X	X 1
1 1 0	1 1 1	X 0	X 0	1 X

1 1 1	0 0 0	X 1	X 1	X 1

Now transfer the JK states of the flip-flop inputs from the excitation table to Karnaugh maps to derive a simplified Boolean expression for each flip-flop input. This is shown in Figure 20.
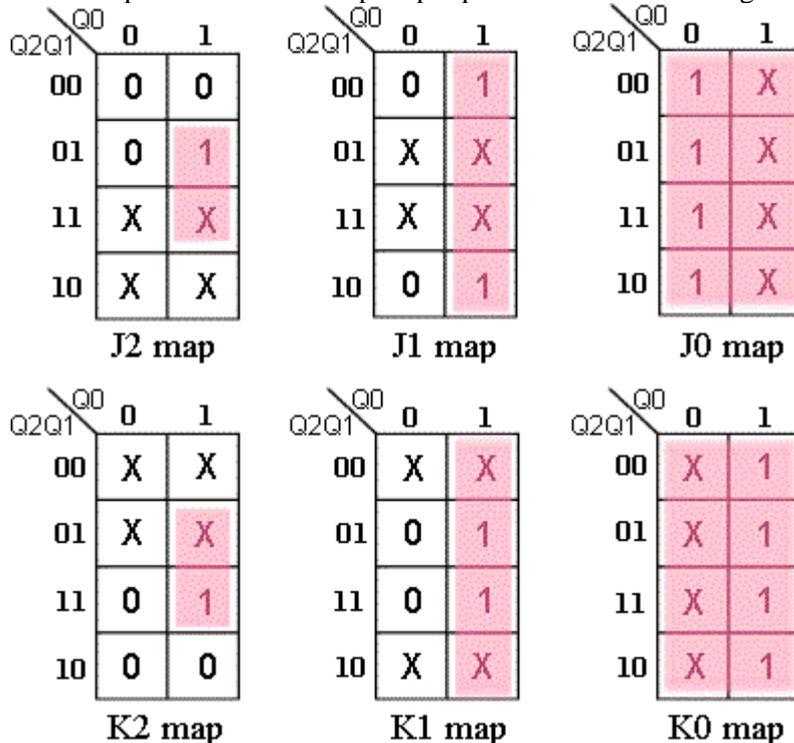

Figure 20. Karnaugh maps

The 1s in the Karnaugh maps of Figure 20 are grouped with "don't cares" and the following expressions for the J and K inputs of each flip-flop are obtained:

J0 = K0 = 1

J1 = K1 = Q0

J2 = K2 = Q1*Q0

The final step is to implement the combinational logic from the equations and connect the flip-flops to form the sequential circuit. The complete logic of a 3-bit binary counter is shown in Figure 21.
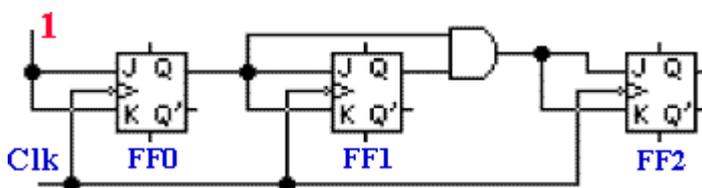
Figure 21. Logic diagram of a 3-bit binary counter

Example 1.6   Design a counter specified by the state diagram in Example 1.5 using T flip-flops. The state diagram is shown here again in Figure 22.

Digital System Design   17EE35
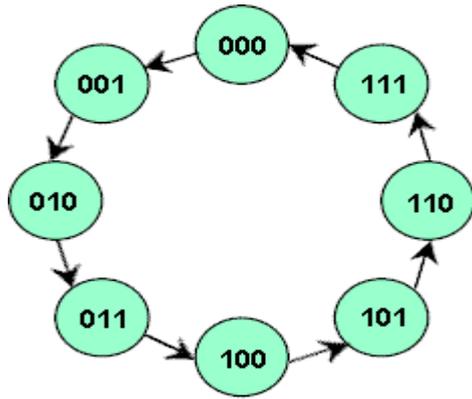
Figure 22. State diagram of a 3-bit binary counter.

The state table will be the same as in Example 1.5.

Now derive the excitation table from the state table, which is shown in Table 17.

Table 17. Excitation table.

Output State Transitions		Flip-flop inputs T2 T1 T0
Present State Q2 Q1 Q0	Next State Q2 Q1 Q0	
0 0 0	0 0 1	0 0 1
0 0 1	0 1 0	0 1 1
0 1 0	0 1 1	0 0 1
0 1 1	1 0 0	1 1 1
1 0 0	1 0 1	0 0 1
1 0 1	1 1 0	0 1 1
1 1 0	1 1 1	0 0 1
1 1 1	0 0 0	1 1 1

Next step is to transfer the flip-flop input functions to Karnaugh maps to derive a simplified Boolean expressions, which is shown in Figure 23.
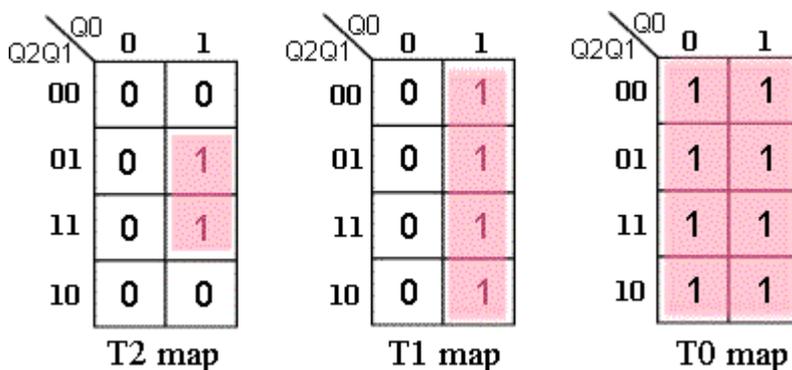

Figure 23. Karnaugh maps

The following expressions are obtained:

T0 = 1;   T1 = Q0;   T2 = Q1*Q0

Finally, draw the logic diagram of the circuit from the expressions obtained. The complete logic diagram of the counter is shown in Figure 24.
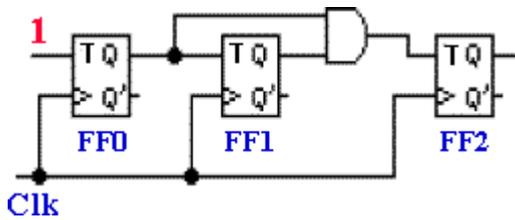
Figure 24. Logic diagram of 3-bit binary counter.

Exercises

Analysis of Sequential Circuits.

1. Derive a) excitation equations, b) next state equations, c) a state/output table, and d) a state diagram for the circuit shown in Figure 1.1. Draw the timing diagram of the circuit.
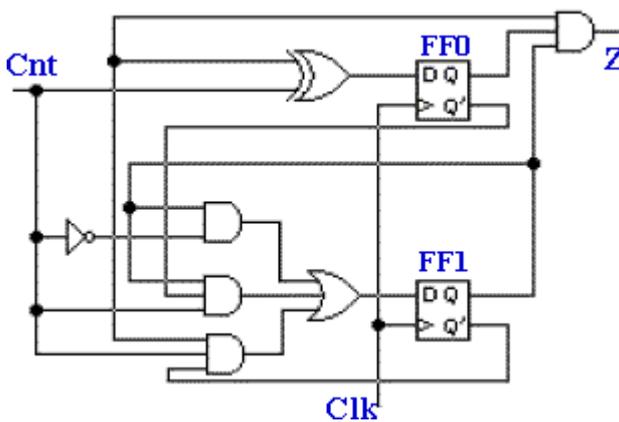
Figure 1.1

2. Derive a) excitation equations, b) next state equations, c) a state/output table, and d) a state diagram for the circuit shown in Figure 1.2.
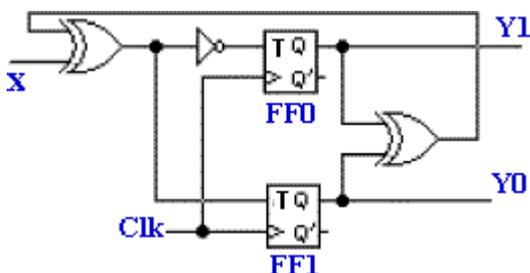
Figure 1.2

3. Derive a) excitation equations, b) next state equations, c) a state/output table, and d) a state diagram for the circuit shown in Figure 1.3.

Digital System Design   *17EE35*
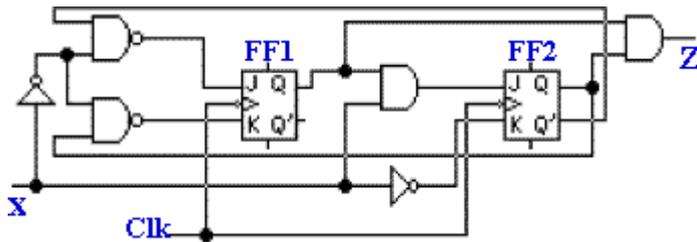
Figure 1.3

4. Derive the state output and state diagran for the sequential circuit shown in Figure 1.4.
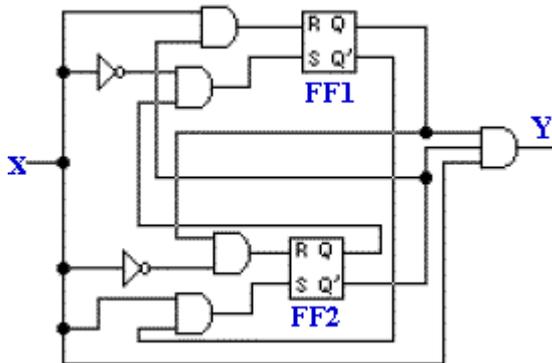
Figure 1.4

5. A sequential circuit uses two D flip-flops as memory elements. The behaviour of the circuit is described by the following equations:

$$D_1 = Q_1 + x'*Q_2$$
$$D_2 = x*Q_1' + x'*Q_2$$
$$Z = x'*Q_1*Q_2 + x*Q_1'*Q_2'$$

Derive the state table and draw the state diagram of the circuit.

6. Design a sequential circuit specified by Table 6.1, using JK flip-flops.

Table 6.1

| Present State | Next State | | Output | |
Q_0 Q_1	x = 0	x = 1	x = 0	x = 1
0 0	0 0	0 1	0	0
0 1	0 0	1 0	0	0
1 0	1 1	1 0	0	0
1 1	0 0	0 1	0	1

7. Design the sequential circuit in question 6, using T flip-flops.

8. Design a mod-5 counter which has the following binary sequence: 0, 1, 2, 3, 4. Use JK flip-flops.

9. Design a counter that has the following repeated binary sequence: 0, 1, 2, 3, 4, 5, 6, 7. Use RS flip-flops.

10. Design a counter with the following binary sequence: 1, 2, 5, 7 and repeat. Use JK flip-flops.

*Digital System Design*   *17EE35*

11. Design a counter with the following repeated binary sequence: 0, 4, 2, 1, 6. Use T flip-flops.

12. Design a counter that counts in the sequence 0, 1, 3, 6, 10, 15, using four a) D, b) SR, c) JK and d) T flip-flops.

13. The content of a 5-bit shift register serial in parallel out with rotation capability is initially 11001. The register is shifted four times to the right. What are the content and the output of the register after each shift?

Tri-state logic

Both combinatorial circuits and sequential circuits are fairly easy to understand, since we pretty much only have to understand logic gates and how they are interconnected.

With _tri-state logic circuits_, this is no longer true. As their names indicate, they manipulate signals that can be in one of _three_ states, as opposed to only 0 or 1. While this may sound confusing at first, the idea is relatively simple.

Consider a fairly common case in which there are a number of source circuits S_1, S_2, etc in different parts of a chip (i.e., they are not real close together). At different times, exactly one of these circuit will generate some binary value that is to be distributed to some set of destination circuits D_1, D_2, etc also in different parts of the chip. At any point in time, exactly one source circuit can generate a value, and the value is always to be distributed to all the destination circuits. Obviously, we have to have some signals that _select_ which source circuit is to generate information. Assume for the moment that we have signals s_1, s_2, etc for exactly that purpose. One solution to this problem is indicated in this figure:
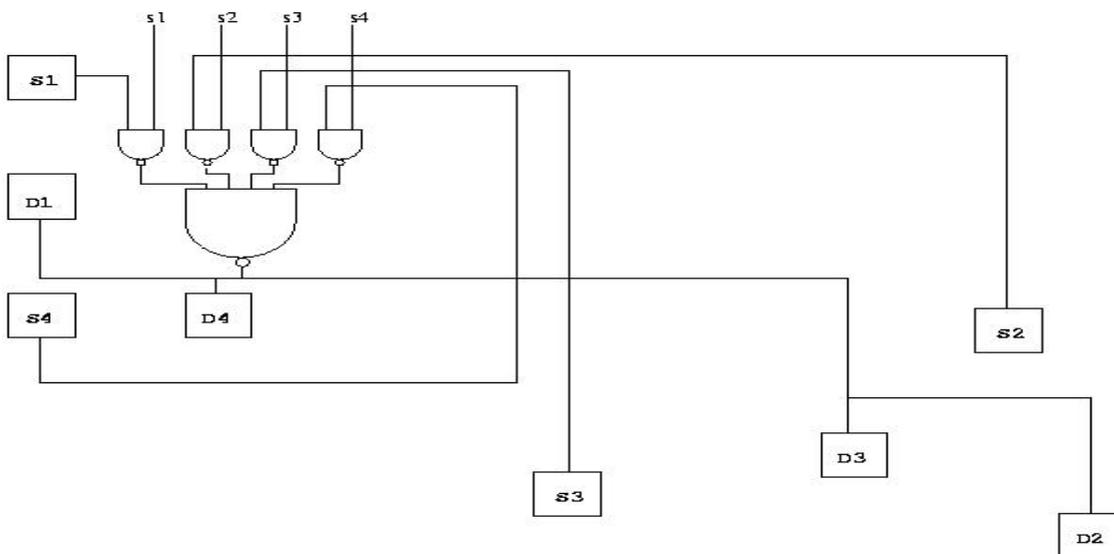

As you can see, this solution requires that all outputs are routed to a central place. Often such solutions are impractical or costly. Since only one of the sources is "active" at one point, we ought to be able to use a solution like this:

Digital System Design   _17EE35_
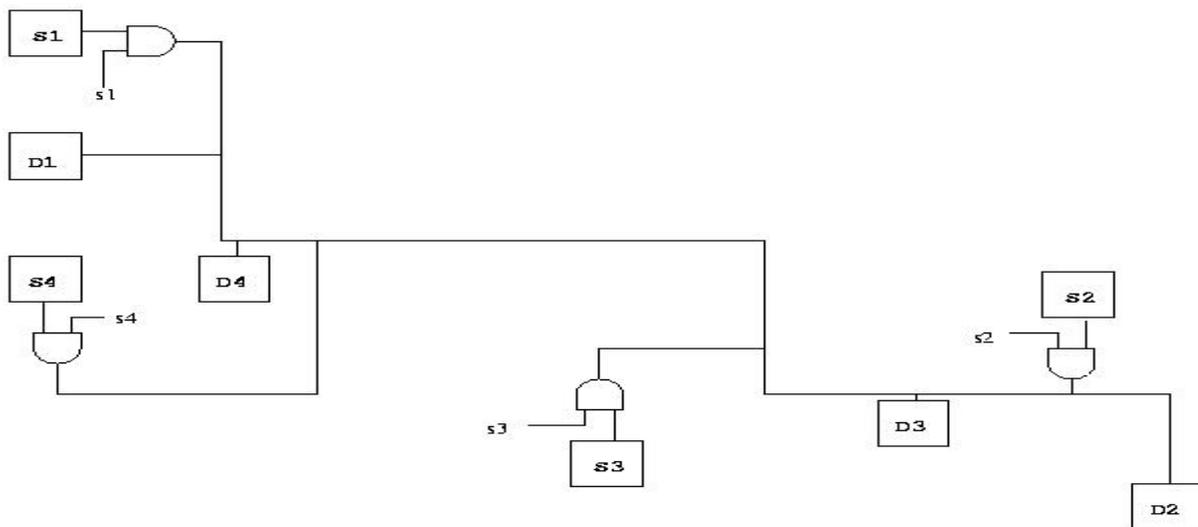
However, connecting two or more outputs together is likely to destroy the circuits. The solution to our problem is to use *tri-state* logic.

Return to the transistors

A *tri-state* circuit (combinatorial or sequential) is like an ordinary circuit, except that it has an additional input that we shall call *enable*. When the enable input is 1, the circuit behaves exactly like the corresponding normal (not tri-state) circuit. When the enable input is 0, *the outputs are completely disconnected from the rest of the circuit*. It is as if there we had taken an ordinary circuit and added a *switch* on every output, such that the switch is open when enable is 0 and closed if enable is 1 like this:
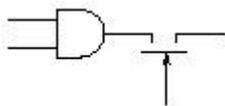

which is pretty close to the truth. The switch is just another transistor that can be added at a very small cost.

Any circuit can exist in a tri-state version. However, as a special case, we can convert any ordinary circuit to a tri-state circuit, by using a special tri-state combinatorial circuit that simply copies its inputs to the outputs, but that also has an *enable* input. We call such a circuit a *bus driver* for reasons that will become evident when we discuss buses. A bus driver with one input is drawn like this:
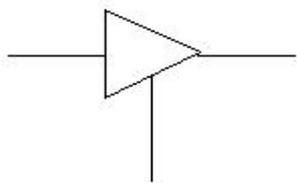
Here is a version for bidirectional signals:

Digital System Design   *17EE35*
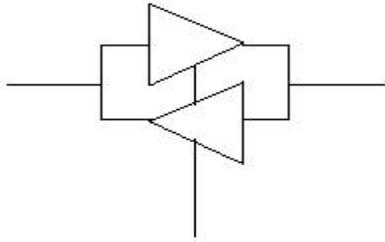
Memories

A memory is neither a sequential circuit (since we require sequential circuits to be clocked, and memories are not clocked), nor a combinatorial circuit, since its output values depend on past values.

In general, a memory has *m* inputs that are called the *address inputs* that are used to select exactly one out of 2^m *words*, each one consisting of *n* bits.

Furthermore, it has *n* connectors that are *bidirectional* that are called the *data lines*. These data lines are used both as *inputs* in order to store information in a word selected by the address inputs, and as *outputs* in order to recall a previously stored value. Such a solution reduces the number of required connectors by a factor two.

Finally, it has an input called *enable* (see the section on tri-state logic for an explanation) that controls whether the data lines have defined states or not, and an input called *r/w* that determines the direction of the data lines.

A memory with an arbitrary value of *m* and an arbitrary value of *n* can be built from memories with smaller values of these parameters. To show how this can be done, we first show how a one-bit memory (one with *m = 0* and *n = 1*) can be built. Here is the circuit:
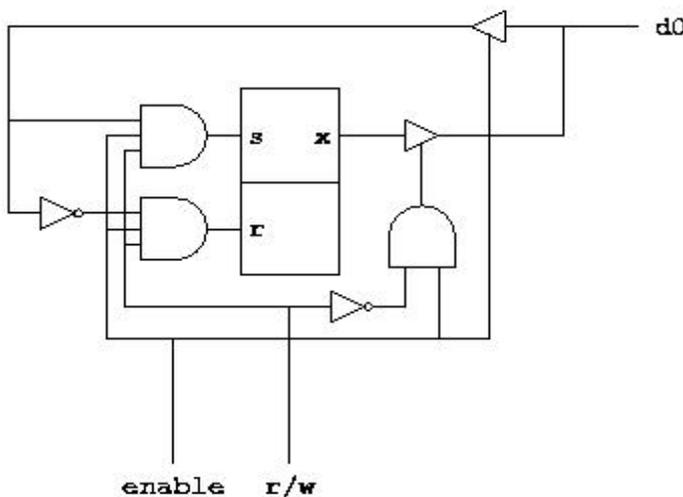

The central part of the circuit is an SR-latch that holds one bit of information. When enable is 0, the output d_0 is isolated both from the inputs to and the output from the SR-latch. Information is passed from d_0 to the inputs of the latch when enable is 1 and r/w is 1 (indicating *write*). Information is passed from the output x to d_0 when enable is 1 and r/w is 0 (indicating *read*).

Now that we know how to make a one-bit memory, we must figure out how to make larger memories. First, suppose we have *n* memories of 2^m *words*, each one consisting of a single bit. We can easily convert these to a single memory with 2^m *words*, each one consisting of a *n* bits. Here is how we do it:

Digital System Design   *17EE35*
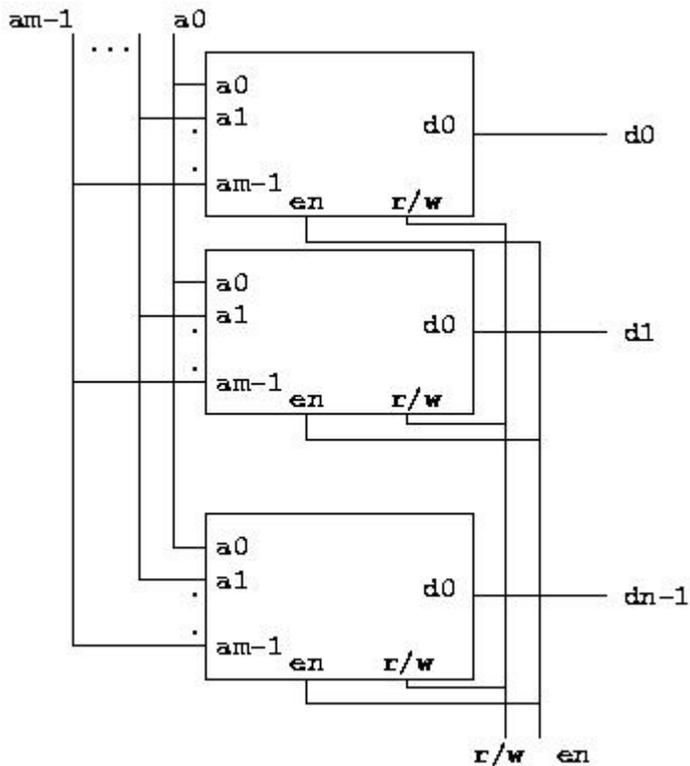
We have simply connected all the address inputs together, all the enables together, and all the read/writes together. Each one-but memory supplies one of the bits of the *n*-bit word in the final circuit.

Next, we have to figure out how to make a memory with more words. To show that, we assume that we have two memories each with *m* address inputs and *n* data lines. We show how we can connect them so as to obtain a single memory with *m + 1* address inputs and *n* data lines. Here is the circuit:
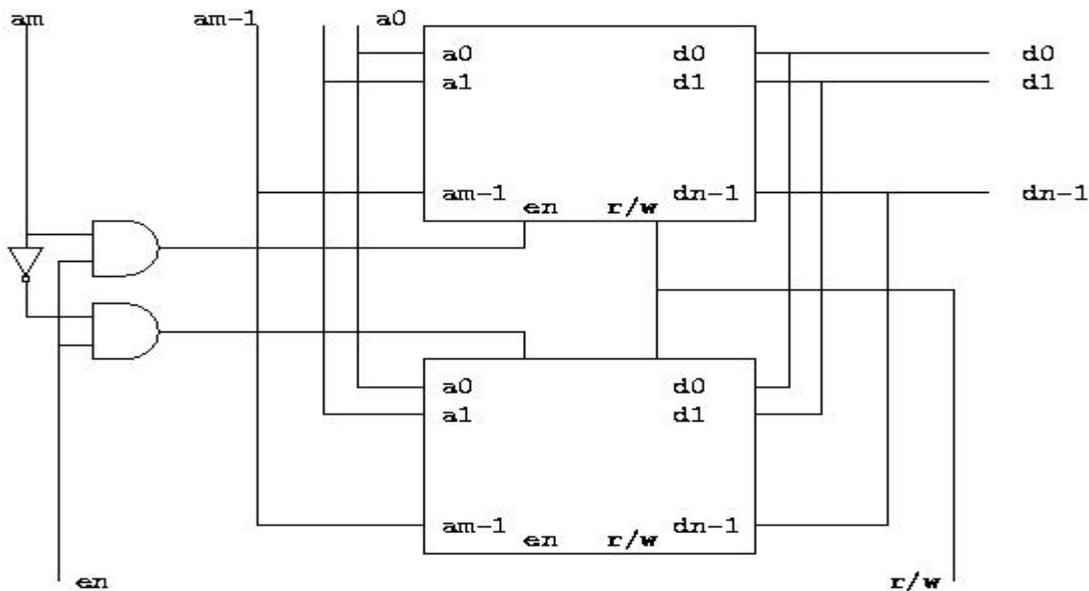
As you can see, the additional address line is combined with the enable input to select one of the two smaller memories. Only one of them will be connected to the data lines at a time (because of the way tri-state logic works).

Digital System Design   *17EE35*

Read-only memories

A *read-only memory* (or *ROM* for short), is like an ordinary memory, except that it does not have the capability of writing. Its contents is fixed at the factory.

Since the contents cannot be altered, we don't have a r/w signal. Except for the enable signal, a ROM is thus like an ordinary combinatorial circuit with *m* inputs and *n* outputs.

ROMs are usually *programmable*. They are often sold with a contents of all 0s or all 1s. The user can then stick it in a special machine and fill it with the desired contents, i.e. the ROM can be *programmed*. In that case, we sometimes call it a PROM (programmable ROM).

Some varieties of PROMS can be erased and re-programmed. The way they are erased is typically with ultra-violet light. When the PROM can be erased, we sometimes call it EPROM (erasable PROM).

A programmable logic device (PLD)

A programmable logic device or PLD is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed

Using a ROM as a PLD

Before PLDs were invented, read-only memory (ROM) chips were used to create arbitrary combinational logic functions of a number of inputs. Consider a ROM with m inputs (the address lines) and n outputs (the data lines). When used as a memory, the ROM contains 2m words of n bits each. Now imagine that the inputs are driven not by an m-bit address, but by m independent logic signals. Theoretically, there are 2m possible Boolean functions of these m signals, but the structure of the ROM allows just 2n of these functions to be produced at the output pins. The ROM therefore becomes equivalent to n separate logic circuits, each of which generates a chosen function of the m inputs.

The advantage of using a ROM in this way is that any conceivable function of the m inputs can be made to appear at any of the n outputs, making this the most general-purpose combinatorial logic device available. Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialised hardware or software. However, there are several disadvantages:

•   They are usually much slower than dedicated logic circuits,

•   they cannot necessarily provide safe "covers" for asynchronous logic transitions so the PROM's outputs may glitch as the inputs switch,

•   They consume more power, and

•   Because only a small fraction of their capacity is used in any one application, they often make an inefficient use of space.

Since most ROMs do not have input or output registers, they cannot be used stand-alone for sequential logic. An external TTL register was often used for sequential designs such as state machines.

Digital System Design   *17EE35*

References

Alan Clements, *Principles of computer hardware*. second edition oxford science publications

M. M. Mano, *Digital Design*, Prentice Hall

http://www.ied.edu.hk/has/phys/de/de-ba.htm

http://www.eelab.usyd.edu.au/digital_tutorial/

http://cwx.prenhall.com/bookbind/pubbooks/mano2/chapter5/deluxe.html

http://www.eelab.usyd.edu.au/digital_tutorial/part3/

http://wearcam.org/ece385/lectureflipflops/flipflops/

http://users.ece.gatech.edu/~leehs/ECE2030/reading/mixed-logic.pdf