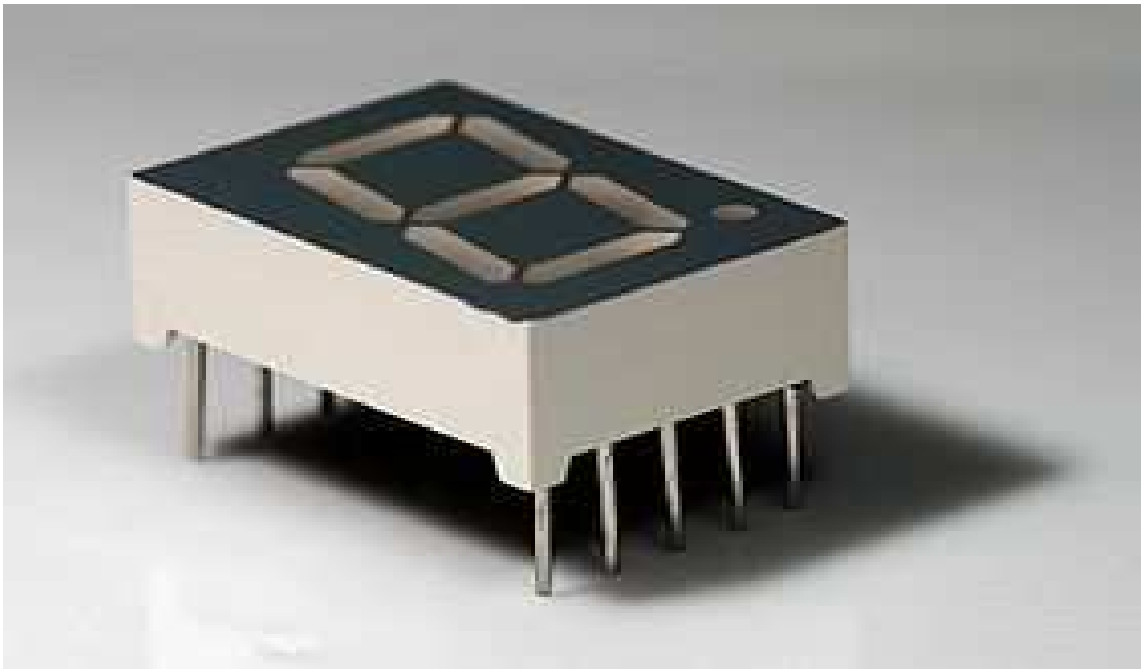
UNESCO-NIGERIA TECHNICAL &
VOCATIONAL EDUCATION
REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



Introduction to Digital Electronics

COURSE CODE: COM 112
THEORY BOOK

YEAR I, SEMESTER I

Version 1: December 2008

WEEK 1 Number Systems 4
Decimal System 4
Binary Number System 4
Octal Number System 5
Hexadecimal Number System 5
WEEK 2 1 Binary to Decimal & Decimal to Binary 7
1.1 Binary to Decimal Conversion 7
1.2 Decimal to Binary Conversion 7
Decimal Values and Binary Equivalents chart: 10
1.3 Binary to Octal & Octal to Binary 10
1.4 Binary to Octal Conversion 11
1.5 Octal to Binary Conversion 11
1.6 Binary to HEX & HEX to Binary 12
WEEK 3 Binary Coded Decimal 14
Excess-three Code: 14
Seven Segment Display Code: 15
WEEK 4 Boolean Postulates 18
Introduction 18
Laws of Boolean Algebra 18
WEEK 5 Basic Digital Logic 21
Basic Boolean Algebra Manipulation 22
WEEK 6 *The Karnaugh Map* 25
WEEK 7 *NAND and NOR implementation* 29
K-Map with Don't care states 31
WEEK 8 Simple Adders 33
Full Adders 34
NAND Gate Implementation of Half Adder 36
NAND Gate Implementation of Full Adder 37
Full Adder 37
WEEK 9 Terminologies used to characterize Integrated Circuits 42
WEEK 10 Integrated Circuits (Chips) 45
Pin numbers 45
Chip holders (DIL sockets) 45
Categories of Integrated Circuits Based on Packing Density 46
Logic IC Series 46
Packages in Digital ICs 46
Identification of Integrated Circuits 47
WEEK 11 Transistor-Transistor Logic (TTL) Technology 49
Comparison with other logic families 50
Applications of TTL 50
Diode–transistor logic 50
Operation 50
Speed disadvantage 51

Emitter-coupled logic 51
History 51
Explanation 51
Characteristics 52
Usage 52
WEEK 12 *Introducing bistable (Flip-Flops)* 53
Basic Flip-Flop circuit 53
Block representation of a Flip-Flop 55
Clocked S-R Flip-Flop 55
The J-K Flip-Flop 56
WEEK 13 Digital Counters 58
Synchronous Counters 59
Why do we need counters? 60
How are counters made? 60
The difference between asynchronous and synchronous counters. 61
WEEK 14 Design of Counters 63
WEEK 15 SHIFT REGISTERS 67
Serial and Parallel Transfers and Conversion 67
Scaling 69
Shift Register Operations 70
Serial-to-Parallel Conversion 72

Week 1

Objective:
1. To understand number systems

Number Systems

Decimal System

The Decimal system is what you use everyday when you count/ Its name is derived from the Latin word Decem, which means ten. This makes sense since the system uses ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. These digits are what we call the *symbols* of the decimal system.

Since we have ten symbols, we can count from 0 to 9. Note that 0, even though it often means 'nothing', is a symbol that counts! After all, you need a numeric way to say 'nothing'. When you want to count past what your simple symbols will allow, you combine multiple digits. The table below shows this concept, which is demonstrated by adding one for every step:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29

The table has 10 numbers across, which is the same number of symbols as the decimal system. As you look at row 2, you notice that we added symbol 1 to the 0, making 10. In row 3, the one is replaced by a 2, giving 20. The further you go down the table, the higher the numbers get.

Binary Number System

Binary is a number system used by digital devices like computers, cd players, etc. Binary is Base 2 unlike our counting system decimal which is Base 10 (denary). In other words, Binary has only 2 different numerals (0 and 1), unlike Decimal which has 10 numerals (0,1,2,3,4,5,6,7,8 and 9). Here is an example of a binary number: 10011100

As you can see it is simply a bunch of zeroes and ones, there are 8 numerals in all which make this an 8 bit binary number, bit is short for Binary Digit, and each numeral is classed as a bit.

The bit on the far right (in this case a zero) is known as the Least significant bit (LSB), and the bit on the far left (in this case a 1) is known as the Most significant bit (MSB)

When writing binary numbers you will need to signify that the number is binary (base 2), for example take the value 101, as it is written it would be hard to work out whether it is a binary or decimal (denary) value, to get around this problem it is common to denote the base to which the number belongs by writing the base value with the number, for example:

1012 is a binary number and 10110 is a decimal (denary) value.

Octal Number System

Although this was once a popular number base, especially in the Digital Equipment Corporation PDP/8 and other old computer systems, it is rarely used today. The Octal system is based on the binary system with a 3-bit boundary. The Octal Number System:

Uses base 8
Includes only the digits 0 through 7 (any other digit would make the number an invalid octal number)

The weighted values for each position are as follows:

8^5	8^4	8^3	8^2	8^1	8^0
32768	**4096**	**512**	**64**	**8**	**1**

Hexadecimal Number System

Binary is an effective number system for computers because it is easy to implement with digital electronics. It is inefficient for humans to use binary, however, because it requires so many digits to represent a number. The number 76, for example, takes only two digits to write in decimal, yet takes seven digits to write in binary (1001100). To overcome this limitation, the *hexadecimal number system* was developed. Hexadecimal is more compact than binary but is still based on the digital nature of computers.

Hexadecimal works in the same way as binary and decimal, but it uses sixteen digits instead of two or ten. Since the western alphabet contains only ten digits, hexadecimal

uses the letters A-F to represent the digits ten through fifteen. Here are the digits used in hexadecimal and their equivalents in binary and decimal:

Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Week 2

Objective:

1. Understand Number Systems
2. Conversion from Binary to Decimal & Decimal to Binary
3. Conversion from Binary to Octal & Octal to Binary
4. Conversion from Binary to Hexadecimal & Hexadecimal to Binary

1 Binary to Decimal & Decimal to Binary

1.1 Binary to Decimal Conversion

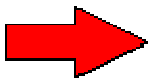
Binary Evaluate	2^4	2^3	2^2	2^1	2^0		Value	Decimal Number
Decimal Value	16	8	4	2	1			
					0	>	0	0
					1	>	1	1
			1	1	0	>	4+2+0	6
		1	0	1	0	>	8+0+2+0	10
	1	0	1	1	0	>	16+0+4+2+0	22
	1	1	0	0	1	>	16+8+0+0+1	25
	1	1	1	1	1	>	16+8+4+2+1	31

1.2 Decimal to Binary Conversion

To convert a decimal number to binary, first subtract the largest possible power of two, and keep subtracting the next largest possible power form the remainder, marking 1st in each column where this is possible and 0s where it is not.

Example 1 - (Convert Decimal 44 to Binary)

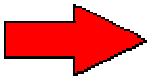
```
  44
- 32
----
  12
-  8
----
   4
-  4
----
   0
----
```



32	16	8	4	2	1
1	0	1	1	0	0

Example 2 - (Convert Decimal 15 to Binary)

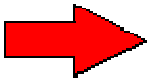
```
  15
-  8
----
   7
-  4
----
   3
-  2
----
   1
----
```



32	16	8	4	2	1
0	0	1	1	1	1

Example 3 - (Convert Decimal 62 to Binary)

```
  62
- 32
----
  30
- 16
----
  14
----
```



32	16	8	4	2	1
1	1	1	1	1	0

Decimal Values and Binary Equivalents chart:

DECIMAL	BINARY
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
16	10000
32	100000
64	1000000
100	1100100
256	100000000
512	1000000000
1000	1111110100
1024	10000000000

1.3 Binary to Octal & Octal to Binary

The following table show Octal numbers and their Binary Equivalent

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

1.4 Binary to Octal Conversion

It is easy to convert from an integer binary number to octal. This is accomplished by:

1. Break the binary number into 3-bit sections from the LSB to the MSB.
2. Convert the 3-bit binary number to its octal equivalent.

For example, the binary value 1010111110110010 will be written:

001	010	111	110	110	010
1	2	7	6	6	2

1.5 Octal to Binary Conversion

It is also easy to convert from an integer octal number to binary. This is accomplished by:

1. Convert the decimal number to its 3-bit binary equivalent.
2. Combine the 3-bit sections by removing the spaces.

For example, the octal value 127662 will be written:

1	2	7	6	6	2
001	010	111	110	110	010

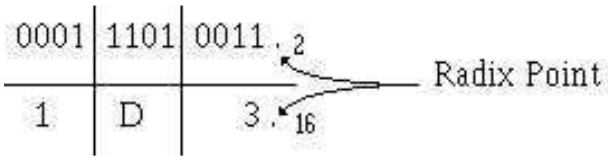
This yields the binary number 001010111110110010 or 00 1010 1111 1011 0010 in more readable format.

1.6 Binary to HEX & HEX to Binary

HEX		BINARY
0	=	0000
1	=	0001
2	=	0010
3	=	0011
4	=	0100
5	=	0101
6	=	0110
7	=	0111
8	=	1000
9	=	1001
A	=	1010
B	=	1011
C	=	1100
D	=	1101
E	=	1110
F	=	1111

Using this relationship, you can easily convert binary numbers to hex. Starting at the radix point and moving either right or left, break the number into groups of four. The grouping of binary into four bit groups is called binary-coded hexadecimal (BCH).

Convert 111010011_2 to hex:



Add 0s to the left of the MSD of the whole portion of the number and to the right of the LSD of the fractional part to form a group of four. Convert $.111_2$ to hex:

Convert $.111_2$ to hex:

$$\frac{.1110_2}{.E_{16}}$$

In this case, if a 0 had not been added, the conversion would have been $.7_{16}$, which is incorrect.

Convert the following binary numbers to hex:

Q1. 10_2

Q2. 1011_2

Q3. 101111_2

Q4. 0011_2

Q5. 110011

Week 3

Objective:

1. Binary Coded Decimal
2. Excess-Three Code
3. Seven Segment Display Code

Binary Coded Decimal

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. Other bit combinations are sometimes used for sign or other indications.

Excess-three Code:

A number code in which the decimal digit *n* is represented by the four-bit binary equivalent of *n* + 3.

Excess-3 binary coded decimal (XS-3), also called **biased** representation or **Excess-N**, is a numeral system that uses a pre-specified number *N* as a biasing value. It is a way to represent values with a balanced number of positive and negative numbers. In XS-3, numbers are represented as decimal digits, and each digit is represented by four bits as the BCD value plus 3 (the "excess" amount):

- The smallest binary number represents the smallest value. (i.e. 0 - Excess Value)
- The greatest binary number represents the largest value. (i.e. 2^N - Excess Value - 1)

Decimal	Binary	Decimal	Binary	Decimal	Binary	Decimal	Binary
-3	0000	**1**	0100	**5**	1000	**9**	1100
-2	0001	**2**	0101	**6**	1001	**10**	1101
-1	0010	**3**	0110	**7**	1010	**11**	1110

0	0011	**4**	0111	**8**	1011	**12**	1111

To encode a number such as 127, then, one simply encodes each of the decimal digits as above, giving (0100, 0101, 1010).

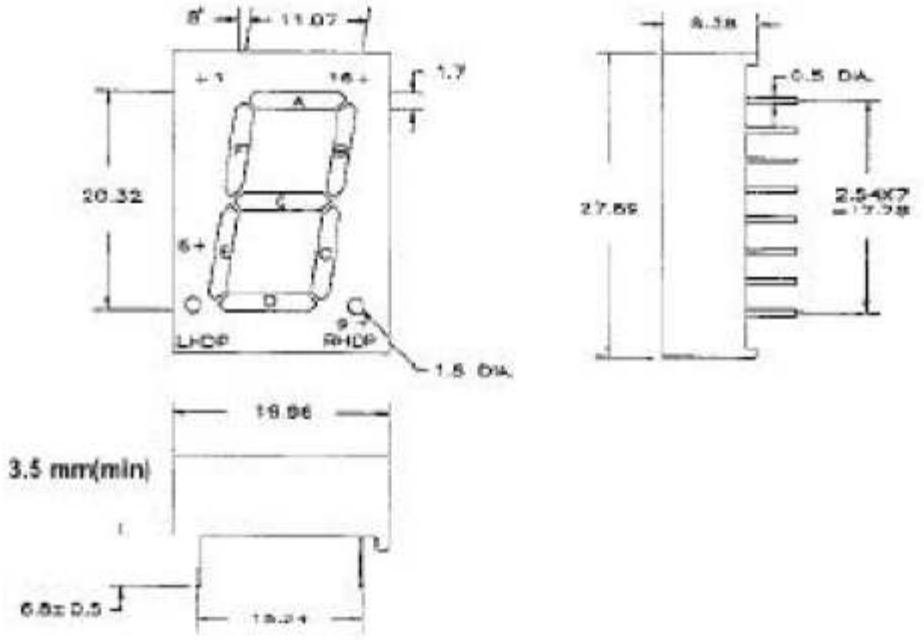
The primary advantage of XS-3 coding over BCD coding is that a decimal number can be nines' complemented (for subtraction) as easily as a binary number can be ones' complemented; just invert all bits.

Adding Excess-3 works on a different algorithm than BCD coding or regular binary numbers. When you add two XS-3 numbers together, the result is not an XS-3 number. For instance, when you add 1 and 0 in XS-3 the answer seems to be 4 instead of 1. In order to correct this problem, when you are finished adding each digit, you have to subtract 3 (binary 11) if the digit is less than decimal 10 and add three if the number is greater than or equal to decimal 10.

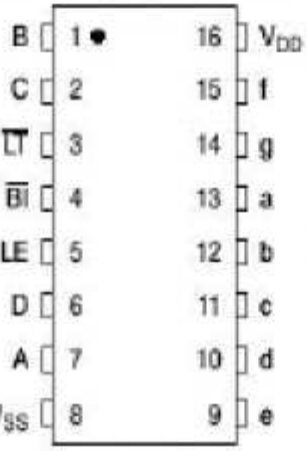
Seven Segment Display Code:

Binary numbers are necessary, but very hard to read or interpret. A seven-segment (LED) display is used to display binary to decimal information.

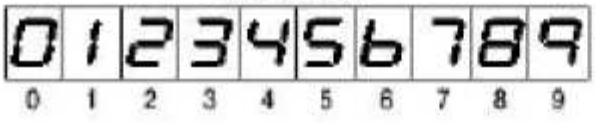
A seven-segment display may have 7, 8, or 9 leads on the chip. Usually leads 8 and 9 are decimal points. The figure below is a typical component and pin layout for a seven segment display.



PIN ASSIGNMENT

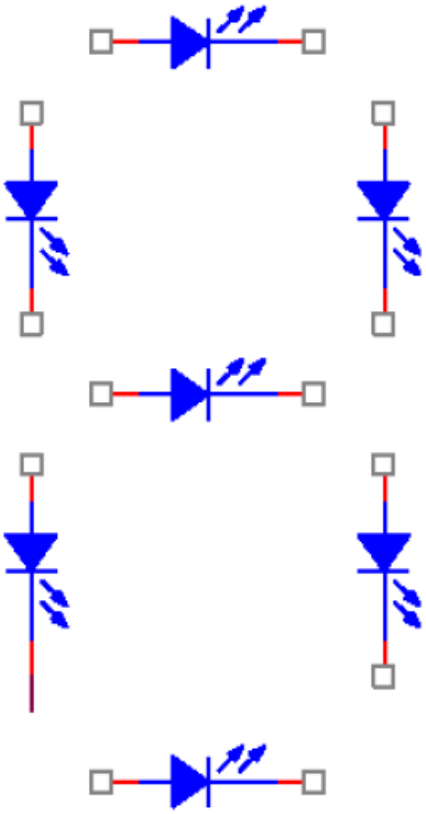


DISPLAY



7 SEGMENT DISPLAY

The light emitting diodes in a seven-segment display are arranged in the figure below.



DIODE PLACEMENT IN A SEVEN SEGMENT DISPLAY

The image below is your typical seven segment display with each of the segments labeled with the letters A through G. To display digits on these displays you turn on some of the LEDs. For example, when you illuminate segments B and C for example your eye perceives it as looking like the number "1." Light up A, B, and C and you will see what looks like a "7."

Week 4

Objective:

1. Know the fundamentals of Boolean Algebra
2. Understand Boolean Postulates
3. Minimize Logical expressions algebraically.

Boolean Postulates

Introduction

The most obvious way to simplify Boolean expressions is to manipulate them in the same way as normal algebraic expressions are manipulated. With regards to logic relations in digital forms, a set of rules for symbolic manipulation is needed in order to solve for the unknowns.

A set of rules formulated by the English mathematician *George Boole* describe certain propositions whose outcome would be either *true or false*. With regard to digital logic, these rules are used to describe circuits whose state can be either, *1 (true) or 0 (false)*. In order to fully understand this, the relation between the AND gate, OR gate and NOT gate operations should be appreciated. A number of rules can be derived from these relations as Table 1 demonstrates.

- **P1: X = 0 or X = 1**
- **P2: 0 . 0 = 0**
- **P3: 1 + 1 = 1**
- **P4: 0 + 0 = 0**
- **P5: 1 . 1 = 1**
- **P6: 1 . 0 = 0 . 1 = 0**
- **P7: 1 + 0 = 0 + 1 = 1**

Table 1: Boolean Postulates

Laws of Boolean Algebra

Table 2 shows the basic Boolean laws. Note that every law has two expressions, (a) and (b). This is known as *duality*. These are obtained by changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa.
It has become conventional to drop the **.** (AND symbol) i.e. A.B is written as AB.

Commutative Law

(a) $A + B = B + A$

(b) $A\,B = B\,A$

Associate Law

(a) $(A + B) + C = A + (B + C)$
(b) $(A\ B)\ C = A\ (B\ C)$

Distributive Law

(a) $A\ (B + C) = A\ B + A\ C$
(b) $A + (B\ C) = (A + B)\ (A + C)$

Identity Law

(a) $A + A = A$
(b) $A\ A = A$

(a) $A\bar{B} + AB = A$
(b) $(A+\bar{B})(A+B) = A$

Redundance Law

(a) $A + A\ B = A$
(b) $A\ (A + B) = A$

(a) $0 + A = A$
(b) $0\ A = 0$

(a) $1 + A = 1$
(b) $1\ A = A$

(a) $\bar{A} + A = 1$
(b) $\bar{A}\ A = 0$

(a) $A + \bar{A}\ B = A + B$
(b) $A\ (\bar{A} + B) = A\ B$

De Morgan's Theorem

De Morgan's theorem are rules in formal logic relating pairs of dual logical operators in a systematic manner expressed in terms of negation. The relationship so induced is called De Morgan duality.

not (P and Q) = (not P) or (not Q)
not (P or Q) = (not P) and (not Q)

De Morgan's laws are based on the equivalent truth-values of each pair of statements.

(a) $\overline{(A+B)} = \overline{A}\ \overline{B}$

(b) $\overline{(A\,B)} = \overline{A} + \overline{B}$

Minimize Logic Expressions Algebraically

Using the laws given above, complicated expressions can be simplified.

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

$Z = AA + A\overline{B}C + A\overline{B} + \overline{B}\overline{B}C + A\overline{C} + \overline{B}C\overline{C}$

$Z = A(1 + \overline{B}C + \overline{B} + \overline{C}) + \overline{B}C + \overline{B}C\overline{C}$ from laws T8b and Tb

$Z = A + \overline{B}C$ from laws T8a, T8b and T9b

Week 5

Objective:

1. Know the fundamentals of Boolean Algebra.
2. Understand Boolean Algebra Manipulation.
3. Using Boolean Postulates to minimize Logic equations.

Basic Digital Logic

There are three major functions in Digital Electronics. These functions are used to make more complicated circuits, so an understanding of how these building blocks work is key to understanding how circuits work.

The "AND" function requires that multiple inputs are all true for the output to be true. For example, if you turn your car's ignition key, and step on the gas, then your car will start. Simply turning the key or stepping on the gas isn't enough, both must be done to get the correct output. Likewise, all the inputs into an AND gate must be true for the output to be true.

The "OR" function requires any input to be true for the output to be true. For example, you can enter your home through either the back door or front door. Once you enter either one, you are inside your home. Likewise, at least one of the inputs into an OR gate must be true for the output to be true. If more then one input is true, the output is still true, since the minimum requirement is one.

The "INVERTER" function (also known as the "NOT") simply changes the condition. If it was true it becomes false, and if it was false it becomes true. For example, it is never day and night at the same time. If it is day, it is not night. Likewise, an INVERTER gate will logically change the input. For the output to be true, the input must be false.

In digital electronics, a false condition is 0 volts (called VSS), while a true condition is the applied voltage (called VCC or VDD). Since the applied voltage can range from under 3 volts to 5 volts, the true condition is normally simply called a logical 1, and the false condition is called a logical 0.

Using this information, it is possible to create what is called a "truth table." A truth table lists each possible input combination, and the resulting output for each combination. While the AND and the OR functions can each have two or more inputs, the truth table given here will assume two inputs.

```
   AND        OR       INV
#1 #2 O    #1 #2 O    I O
-------    -------    ---
```

```
0  0  0    0  0  0    0 1
0  1  0    0  1  1    1 0
1  0  0    1  0  1
1  1  1    1  1  1
```

To read this table, read across. For example, look at the third line down. If input #1 is a logical 1 while input #2 is a logical 0, the output of an AND gate is a logical 0. On the other hand, the same inputs into an OR gate will generate a logical 1 output. Remember that for an AND gate all inputs must be true (input #1 AND input #2) to get a true output. However, for an OR gate only one must be true (input #1 OR input #2) to get a true output.

Basic Boolean Algebra Manipulation

Boolean Algebra equations can be manipulated by following a few basic rules.

Manipulation Rules

$A + B = B + A$

$A * B = B * A$

$(A + B) + C = A + (B + C)$

$(A * B) * C = A * (B * C)$

$A * (B + C) = (A * B) + (A * C)$

$A + (B * C) = (A + B) * (A + C)$

Equivalence Rules

$=$

$A = A$ (double negative)

$A + A = A$

$A * A = A$

$A * \overline{A} = 0$

$A + \overline{A} = 1$

Rules with Logical Constants

$0 + A = A$

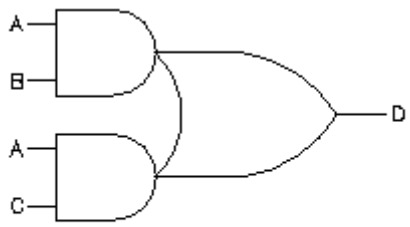
$1 + A = 1$

$0 * A = 0$

$1 * A = A$

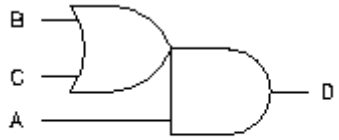
Many of these look identical to Matrix Operations in Linear Algebra. At any rate, this permits a circuit designer to create a circuit as it comes to their mind, then manipulate the formula to generate an equivalent circuit that does the same thing but requires less space.

This can be illustrated using the 5th manipulation rule.



D = (A * B) + (A * C)

Using the rule, generating an equivalent circuit that does the exact same thing, but be less complicated, can be done with reasonable ease.



D = A * (B + C)

In the case of CMOS, the right hand side of the formula can also be manipulated, just always remember to invert. The manipulation occurs under the invert bar.

$$D = \overline{(A * B) + (A * C)}$$

is the same as...

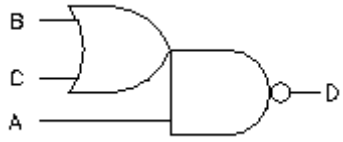
$$D = \overline{A * (B + C)}$$

The manipulation is done the exact same way. Once there is a simplified formula, using the rules with logical constants permit the placement of values directly into the formula to see what the answer is. For example, using the above non inverted formula, C is a logical 1.

```
D = A * (B + C)
D = A * (B + 1)
D = A * (1)            [1 + A = 1]
D = A                  [1 * A = A]
```

If C is known to be a logical 1, anything OR logical 1 is always a logical 1. Since the minimum requirement is one input, once a single input is true (in this case C), the other inputs don't alter the result. On the other hand, the AND gate requires all inputs. With B+C true, the only other requirement is A. As the formula gave, D will be whatever A is.

Many Boolean Algebra problems can be solved using more then one formula, just like most Algebra problems. For example...



$D = \overline{A * (B + C)}$ [given formula]

$D = \overline{A} + \overline{(B + C)}$ [DeMorgan $\overline{(A * B)} = \overline{A} + \overline{B}$]

$D = \overline{A} + (\overline{B} * \overline{C})$ [DeMorgan $\overline{(A + B)} = \overline{A} * \overline{B}$]

$D = (\overline{A} + \overline{B}) * (\overline{A} + \overline{C})$
[A + (B * C) = (A + B) * (A + C)]

$D = \overline{(A * B)} * \overline{(A * C)}$ [DeMorgan $\overline{(A * B)} = \overline{A} + \overline{B}$]

$D = \overline{(A * B) + (A * C)}$ [DeMorgan $\overline{(A + B)} = \overline{A} * \overline{B}$]

Manipulation rule number 5 could have been used to go from the first step to the last one in a single move. However, using DeMorgan's Theorem, the problem turns into something that manipulation rule number 6 can then be used on instead. DeMorgan's Theorem changes the logic of the formula.

Week 6

Objective:

1. Know the fundamentals of Boolean algebra
2. Understand Karnaugh Map

The Karnaugh Map

The Karnaugh map provides a simple and straight-forward graphic method of minimising boolean expressions. It groups together expressions with common factors, thus eliminating unwanted variables. With the Karnaugh map, Boolean expressions having up to four and even six variables can be simplified.

The Karnaugh map is a rectangular map of the value of the expression for all possible input values, it comprises a box for every line in the truth table. But unlike a truth table, in which the input values typically follow a standard binary sequence (00, 01, 10, 11), the Karnaugh map's input values must be ordered such that the values for adjacent columns vary by only a single bit, for example, 00, 01, 11, and 10. This ordering is known as a gray code, and it is a key factor in the way in which Karnaugh maps work.

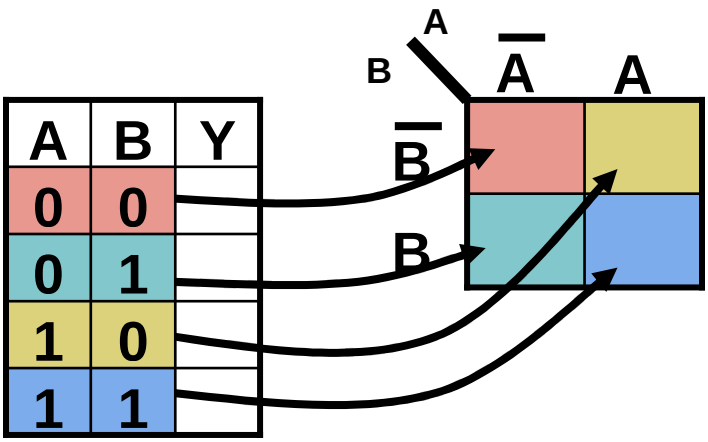
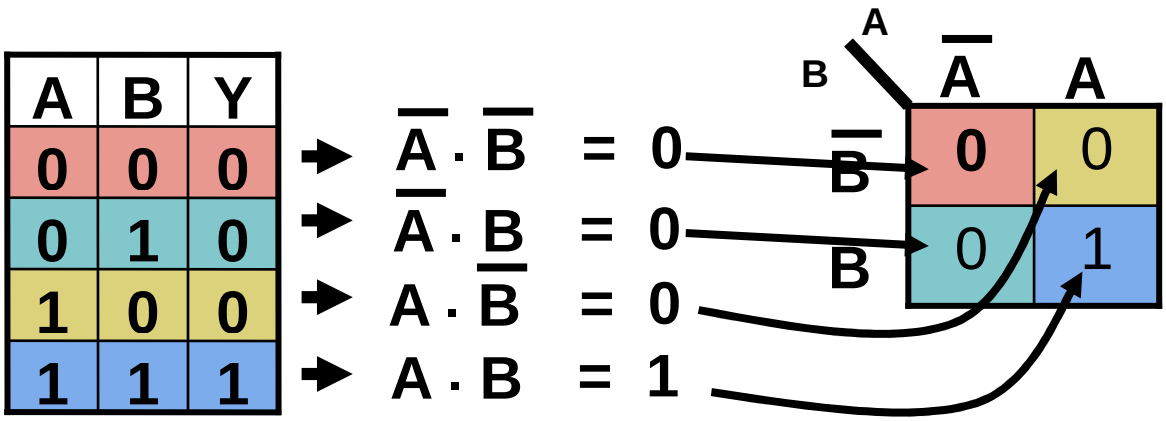


Figure illustrates the concept of a Karnaugh map for 2 Inputs.

Example of Karnaugh map for 2 input AND gate



An Example of Karnaugh Map for AND gate

Lets take an example of 4 inputs (A, B, C and D) as in Figure 2.

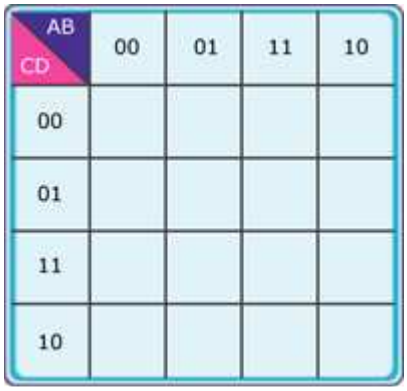


Figure 2: The basic Karnaugh map

Two things are noteworthy about this map. First, we've arranged the 16 possible values of the four inputs as a 4x4 array, with two bits encoding each row or column.

The second and key feature is the way we number the rows and columns. They aren't in binary sequence, as you might think. As you can see, they have the sequence 00, 01, 11, 10. Some of you may recognize this as a Gray code.

Why this particular sequence? Because the codes associated with any two adjacent rows or columns represent a change in only one variable. In a true binary counting code, sometimes several digits can change in a single step; for example, the next step after 0x1111 is 0x10000. Five output signals must change values simultaneously. In digital circuits, this can cause glitches if one gate delay is a bit faster or slower than another. The Gray code avoids the problem. It's commonly used in optical encoders.

Suppose the output value for two adjacent cells is the same. Since only one input variable is changed between the two cells, this tells us that the output doesn't depend on that input. It's a "don't care" for that output.

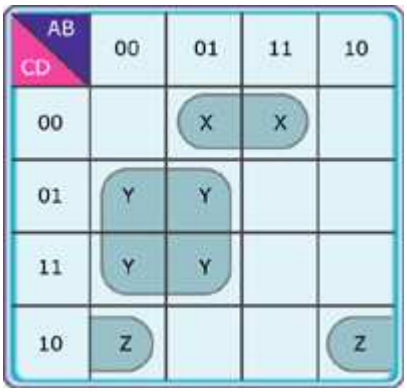


Figure 3: Looking for patterns

Look at Figure 3. Group X is true for inputs *ABCD* = 0100 and 1100. That means that it doesn't depend on *A*, and we can write:

$$X = B\overline{C}\overline{D} \quad (20)$$

Similarly, Group Y doesn't depend on *B* or *C*. Its value is:

$$Y = \overline{A}D \quad (21)$$

Note that the groupings don't necessarily have to be in contiguous rows or columns. In Group Z, the group wraps around the edges of the map.

If we can group cells by twos, we eliminate one input. By fours, two inputs, and so on. If the cell associated with a given output is isolated, it depends on all four inputs, and no minimization is possible.

The Karnaugh map gives us a wonderful, graphical picture that lets us group the cells in a near-optimal fashion. In doing so, we minimize the representation. Neat, eh?

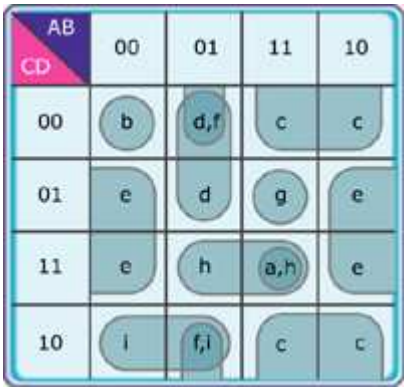


Figure 4: Equation 2, mapped

Now let's see how Equation 2 plots onto the Karnaugh map, as shown in Figure 4. To make it easier to see, I'll assign a different lowercase letter to each term of the equation:

$a = ABC\overline{D}$
$b = \overline{A}\overline{B}\overline{C}\overline{D}$
$c = A\overline{D}$
$d = \overline{A}B\overline{C}$
$e = \overline{B}D$
$f = \overline{A}B\overline{C}$
$g = AB\overline{C}D$
$h = BCD$
$i = \overline{A}C\overline{D}$ (22)

In this case, the individual groups don't matter much. All that counts is the largest group we can identify, which is of course the entire array of 16 cells. The output *X* is true for all values of the inputs, so all four are don't-care variables.

As you can see, using a Karnaugh map lets us see the results and, usually, the optimal grouping at a glance. It might seem that drawing a bunch of graphs is a tedious way to minimize logic, but after a little practice, you get where you can draw these simple arrays fast and see the best groupings quickly. You have to admit, it's a better approach than slogging through Equations 2 through 19.

Week 7

Objective:

1. Know the fundamentals of Boolean algebra
2. Implement K-Maps with don't care states

NAND and NOR implementation

Function F can be implemented using NAND gates only, i.e.:

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

Complement twice:

$$F = (\overline{\overline{F}})$$

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

F now uses NAND gates only

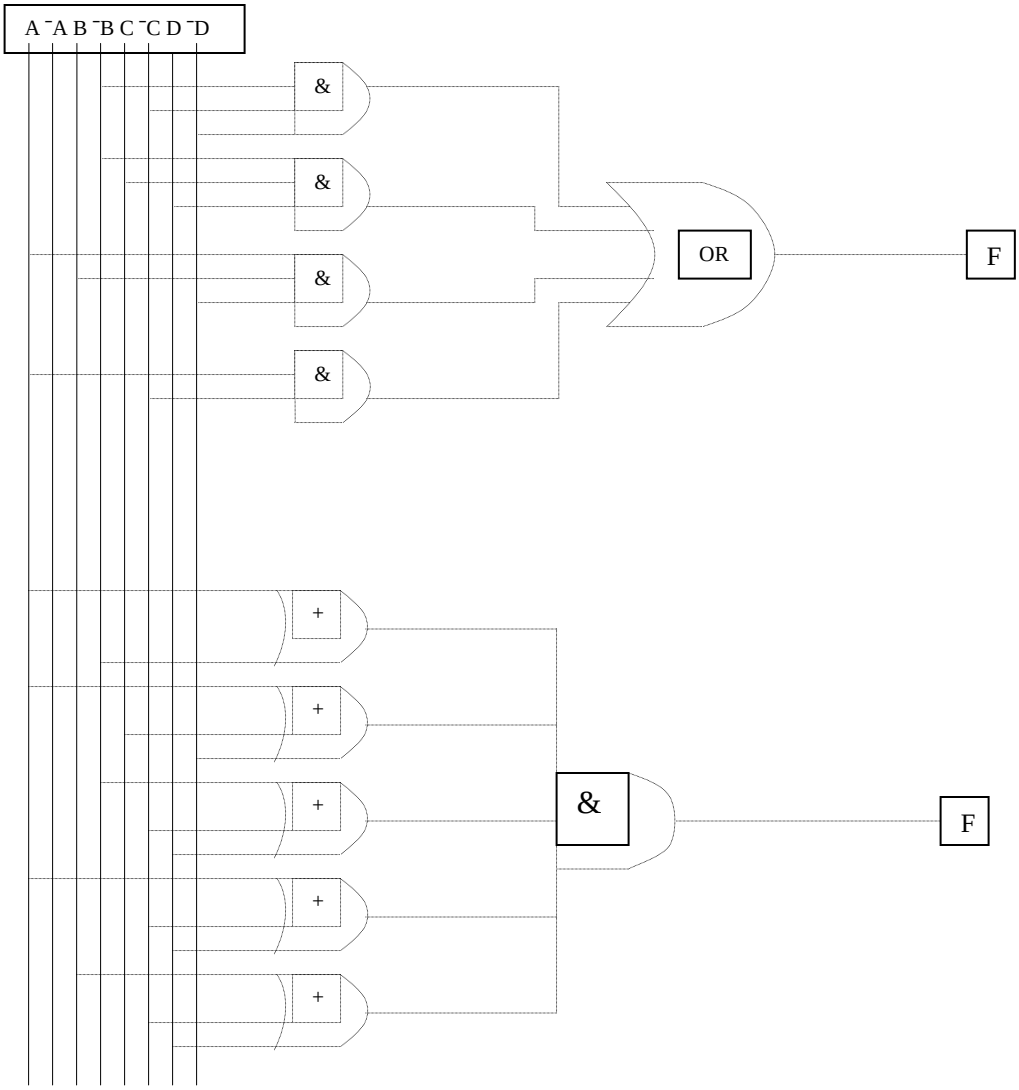
Similarly, F can be implemented using NOR gates only

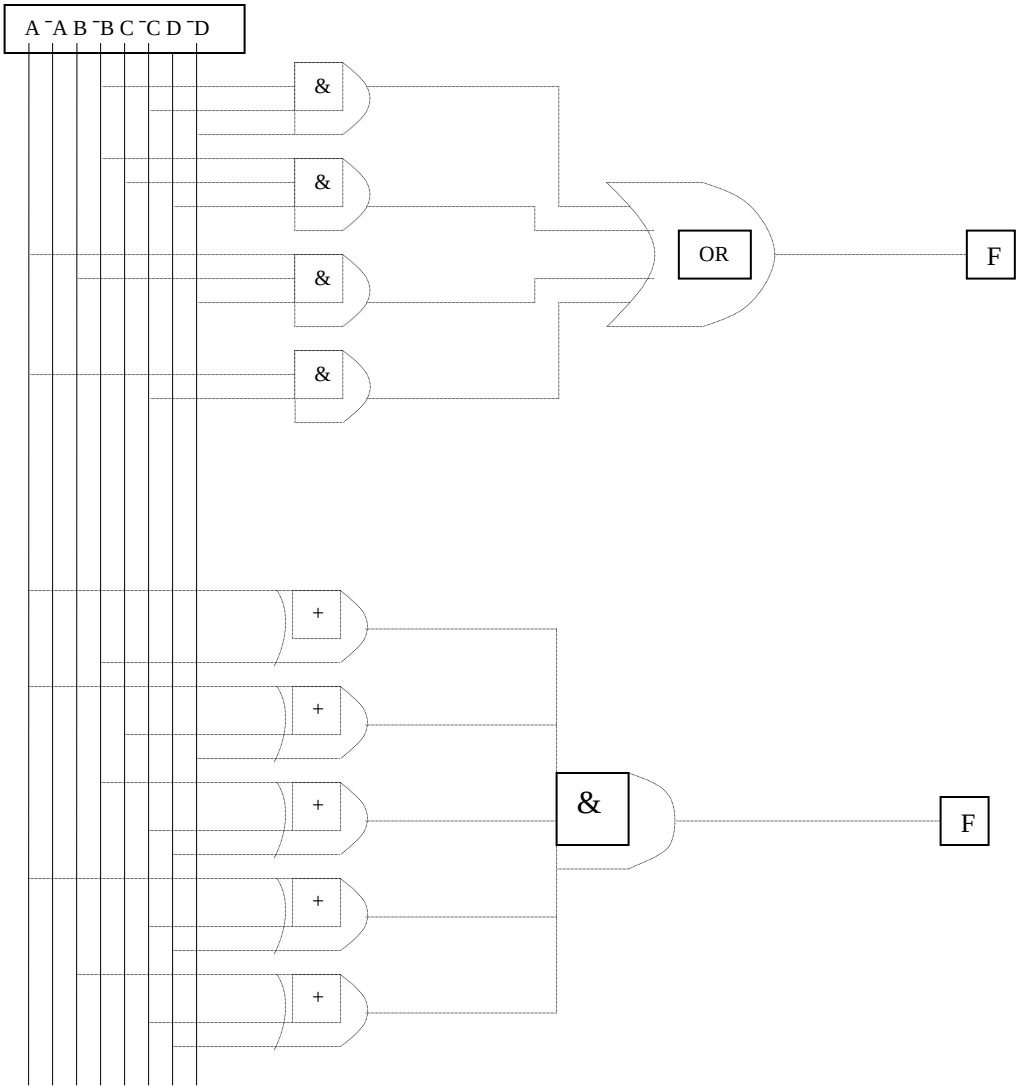
$$F = \overline{\overline{A} \cdot B + \overline{A} \cdot \overline{C} \cdot D + B \cdot C \cdot D + \overline{A} \cdot C \cdot \overline{D} + \overline{B} \cdot C \cdot \overline{D}}$$

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

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

Now a circuit diagram using NOR gates only can be drawn


A ¯A B ¯B C ¯C D ¯D
&
&
&
&
OR
F
+
+
+
+
+
&
F



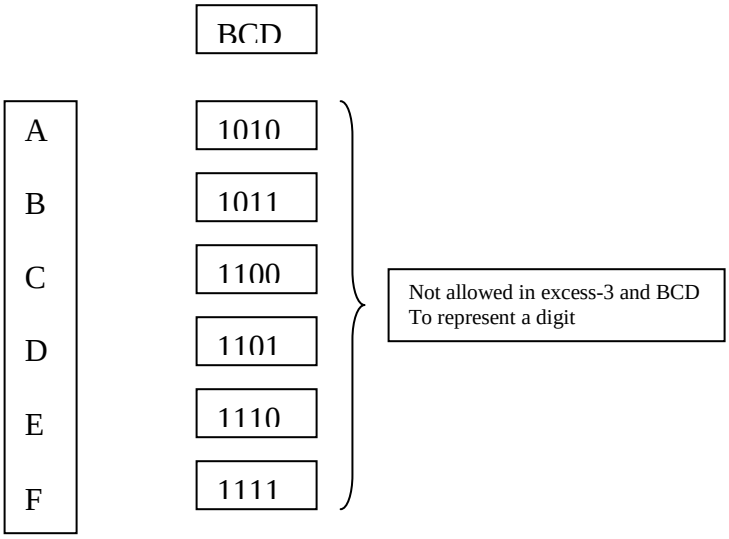
K-Map with Don't care states

K-Map is constructed from a truth table, where all the combinations are given. Assuming sum of minterms selection, a '1' is inserted in the K-Map whenever a certain combination results in obtaining a '1'.

In certain circumstances, a few combinations may not happen or if it does one may not be so concerned about its occurance and its subsequent result. These combinations are called DON'T CARE states and they are represented an a K-Map by a 'X'.

Eg:		BCD ----------→	Excess-3
	^	0000	0011
	|||	0001	0100
Viewed	|||	0010	0101
	|||	0011	0110
	|||	0100	0111
	|||	...	...

Don't care states are



Week 8

Objective:

1. Know the implementation of the addition operation in the computer
2. Design Half Adder
3. Design Full Adder

Simple Adders

In order to design a circuit capable of binary addition one would have to look at all of the logical combinations. You might do that by looking at the following four sums:

```
  0     0     1     1
+ 0   + 1   + 0   + 1
  0     1     1    10
```

That looks fine until you get to 1 + 1. In that case, you have a **carry bit** to worry about. If you don't care about carrying (because this is, after all, a 1-bit addition problem), then you can see that you can solve this problem with an XOR gate. But if you do care, then you might rewrite your equations to always include **2 bits of output**, like this:

```
  0     0     1     1
+ 0   + 1   + 0   + 1
 00    01    01    10
```

From these equations you can form the logic table:

1-bit Adder with Carry-Out

A	B	Q	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

By looking at this table you can see that you can implement Q with an XOR gate and CO (carry-out) with an AND gate.

What if you want to add two 8-bit bytes together? This becomes slightly harder. The easiest solution is to modularize the problem into reusable components and then replicate components. In this case, we need to create only one component: a full binary adder.

The difference between a full adder and the previous adder we looked at is that a full adder accepts an A and a B input plus a **carry-in** (CI) input. Once we have a full adder, then we can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

In the next section, we'll look at how a full adder is implemented into a circuit.

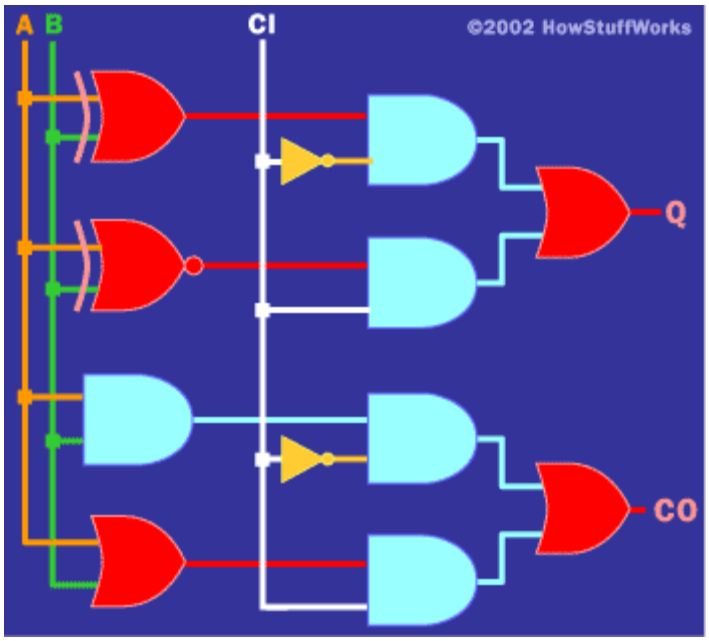
Full Adders

The logic table for a full adder is slightly more complicated than the tables we have used before, because now we have **3 input bits**. It looks like this:

One-bit Full Adder with Carry-In and Carry-Out

CI	A	B	Q	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

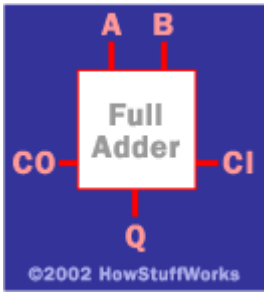
There are many different ways that you might implement this table. If you look at the Q bit, you can see that the top 4 bits are behaving like an XOR gate with respect to A and B, while the bottom 4 bits are behaving like an XNOR gate with respect to A and B. Similarly, the top 4 bits of CO are behaving like an AND gate with respect to A and B, and the bottom 4 bits behave like an OR gate. Taking those facts, the following circuit implements a full adder:



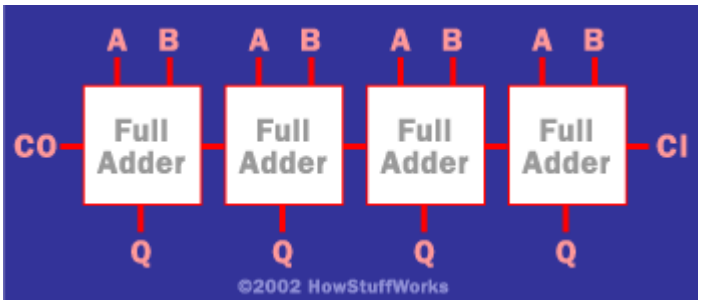
This definitely is not the most efficient way to implement a full adder, but it is extremely easy to understand and trace through the logic using this method.

Exercise: Implement the above logic using fewer gates

Now we have a piece of functionality called a "full adder." What a computer engineer then does is "black-box" it so that he or she can stop worrying about the details of the component. A **black box** for a full adder would look like this:



With that black box, it is now easy to draw a **4-bit full adder**:



In this diagram the carry-out from each bit feeds directly into the carry-in of the next bit over. A 0 is hard-wired into the initial carry-in bit. If you input two 4-bit numbers on the A and B lines, you will get the 4-bit sum out on the Q lines, plus 1 additional bit for the final carry-out. You can see that this chain can extend as far as you like, through 8, 16 or 32 bits if desired.

The 4-bit adder we just created is called a **ripple-carry** adder. It gets that name because the carry bits "ripple" from one adder to the next. This implementation has the advantage of simplicity but the disadvantage of speed problems. In a real circuit, gates take time to switch states (the time is on the order of nanoseconds, but in high-speed computers nanoseconds matter). So 32-bit or 64-bit ripple-carry adders might take 100 to 200 nanoseconds to settle into their final sum because of carry ripple. For this reason, engineers have created more advanced adders called **carry-lookahead** adders. The number of gates required to implement carry-lookahead is large, but the settling time for the adder is much better.

NAND Gate Implementation of Half Adder

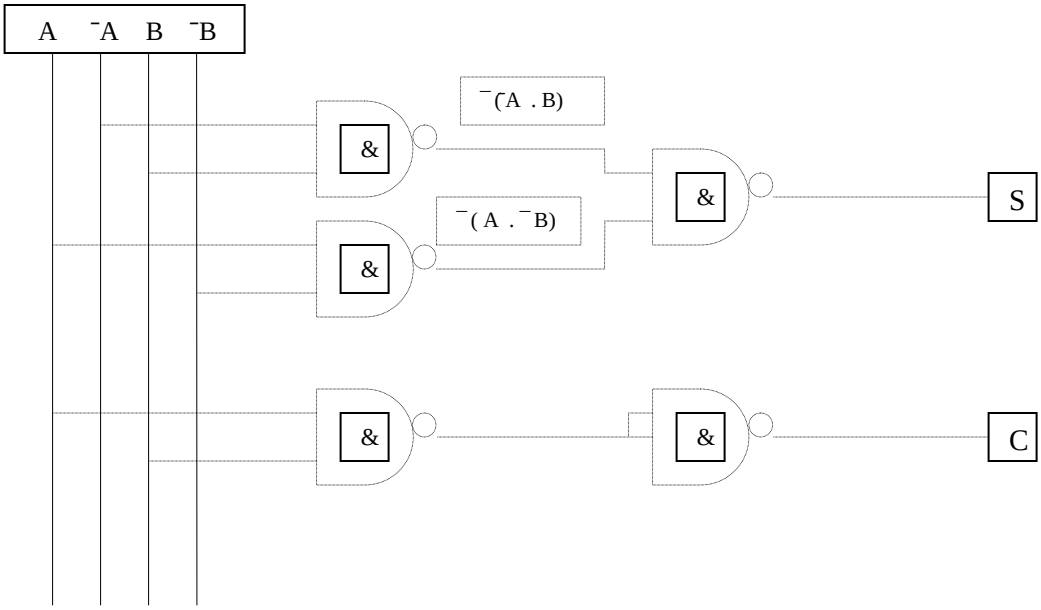
Half adder:
A Combinational Circuit that performs the addition of two bits is called a Half adder.

Full adder:
A Combinational Circuit that performs the addition of three bits (two significant bits and a previous carry) is a full adder.

NAND Implementation:

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

$C = A \cdot B$



Half adder

NAND Gate Implementation of Full Adder

Full Adder

x	y	Z	C	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

Not defined

$S = \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot z$

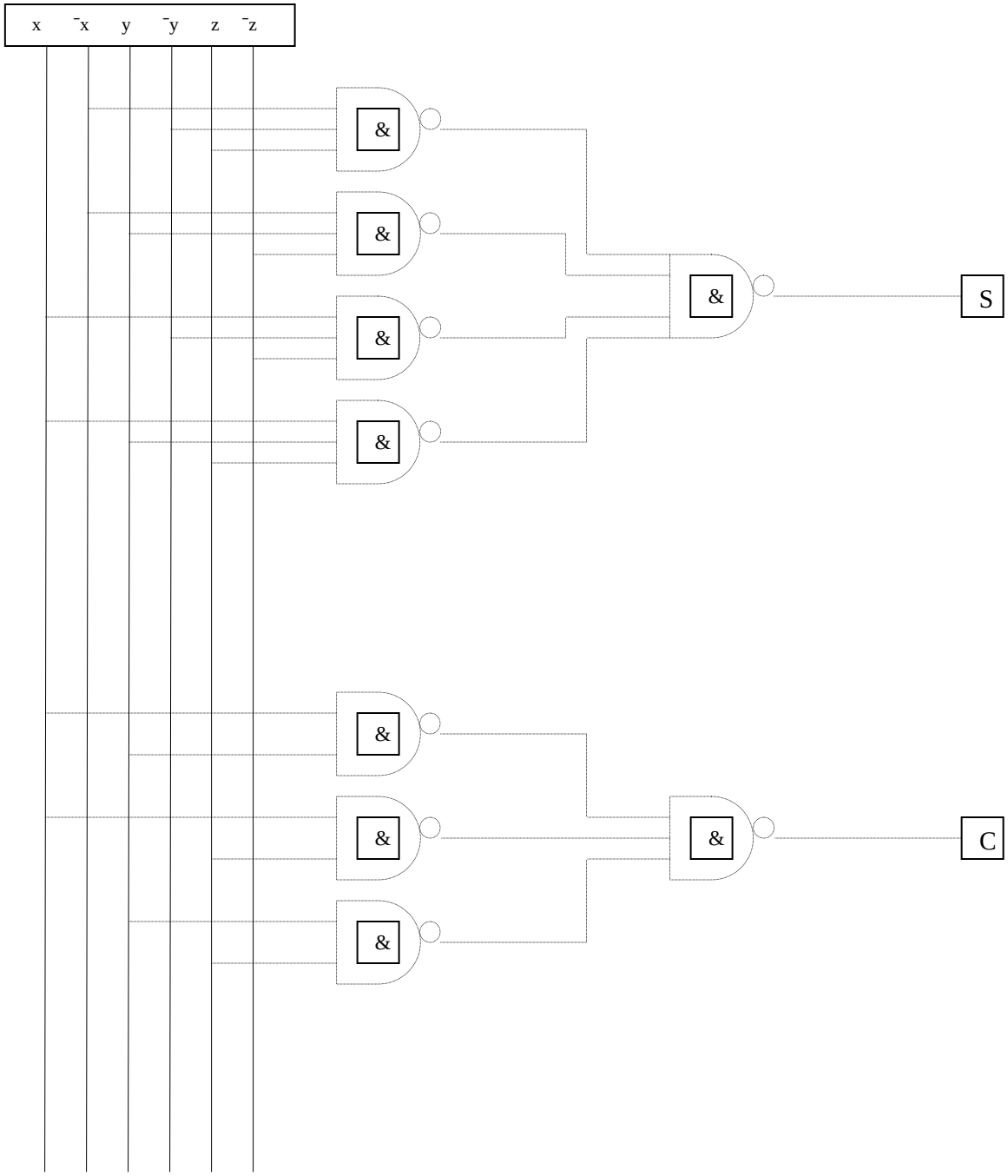
$C = \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z$

xy\z	0	1
00		1
01	1	
11		1
10	1	

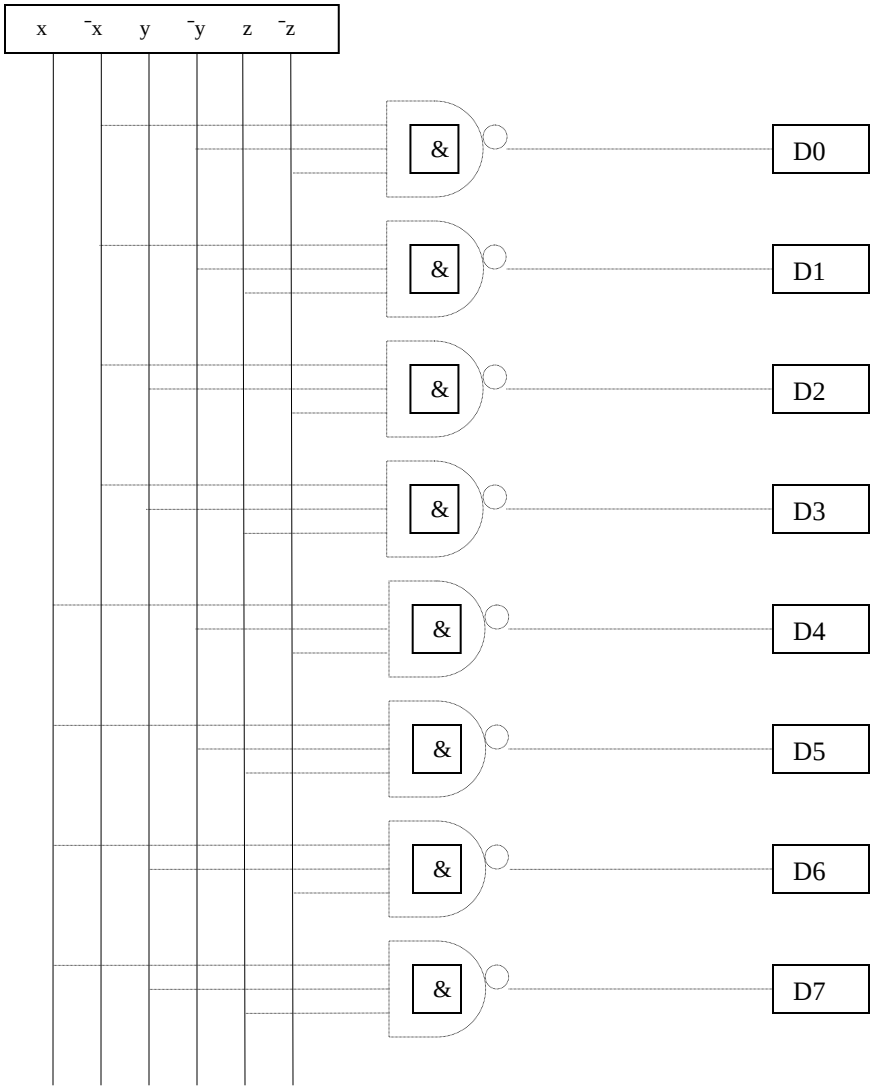
S = No Simplification

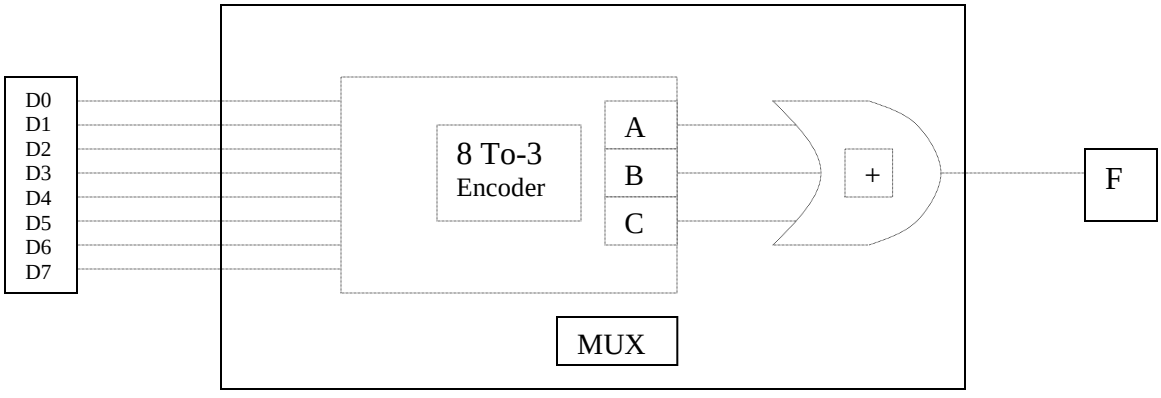
xy\z	0	1
00		
01		1
11	1	1
10		1

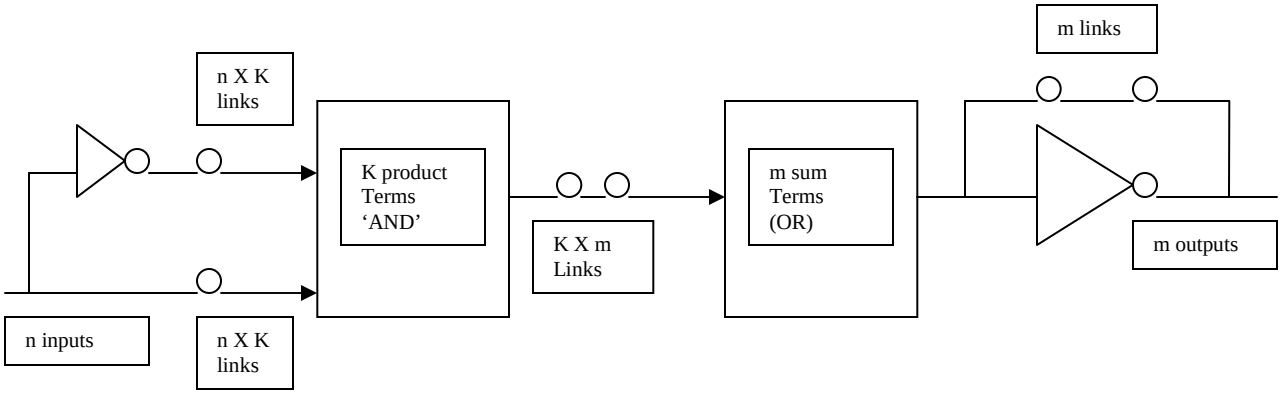
C = x . y + x . z + y . z



NAND Implementation of Full Adder (F A)


x
¯x
y
¯y
z
¯z
&
&
&
&
&
&
&
&
D0
D1
D2
D3
D4
D5
D6
D7


D0
D1
D2
D3
D4
D5
D6
D7
8 To-3
Encoder
A
B
C
+
F
MUX


m links
n X K
links
K product
Terms
'AND'
m sum
Terms
(OR)
K X m
Links
m outputs
n inputs
n X K
links

Week 9

Objective:

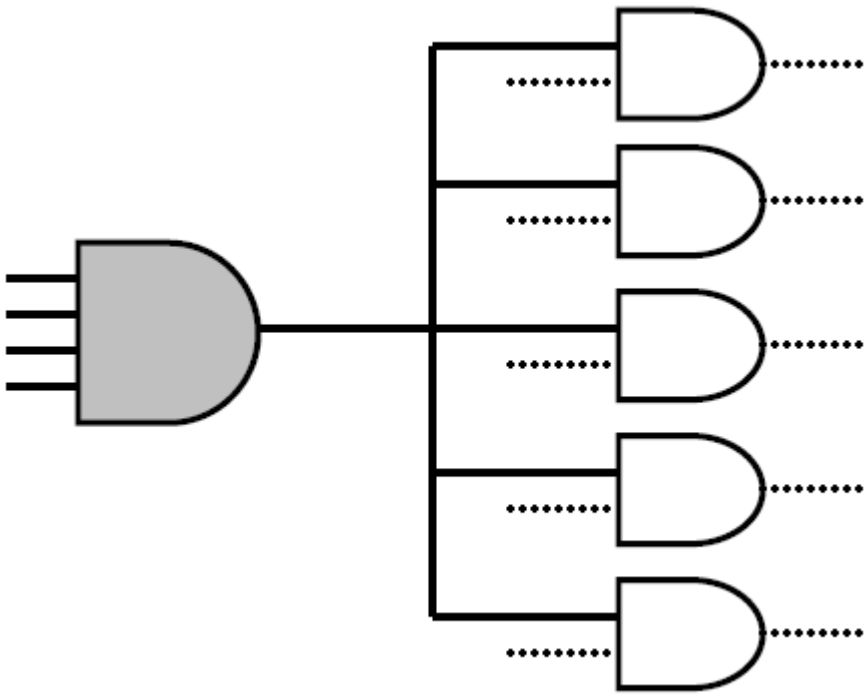
1. Understand Small -Scale Integrated Circuit
2. Understand various terminologies used to characterize integrated circuits

Terminologies used to characterize Integrated Circuits

Fan in: Is the number of inputs of a digital gate.

Fan out: Is the number of gates' inputs connected to the output of a gate (the amount of loading). Sometimes the other types of loads (wires, pads, etc.) are expressed as fan out equivalent.

Example: For the following gate the Fan in is **4** and the Fan out is **5**.



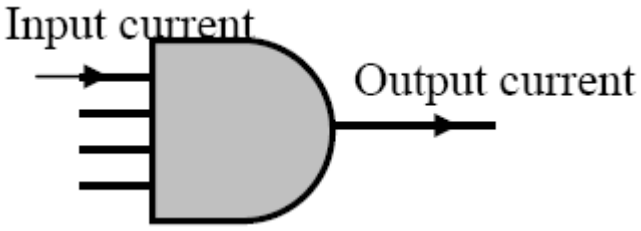
Input & Output Currents:

IIL is the input current to a gate when the input has a logic value of 0 (low input current)

IIH is the input current to a gate when the input has a logic value of 1 (high input current)

IOL is the output current of a gate when the output has a logic value of 0 (low output current)

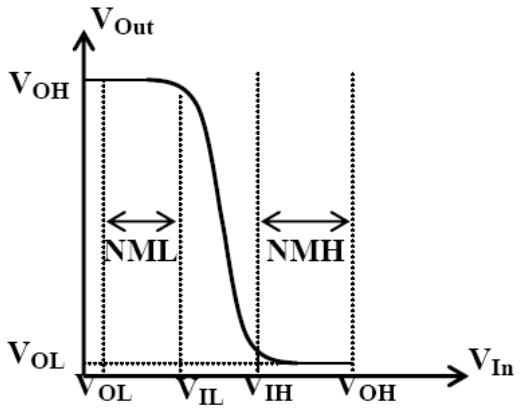
IOH is the output current of a gate when the output has a logic value of 1 (high output current)



Noise Margins: These circuit parameters specify the circuit's ability to withstand noise.

The low noise margin (NML or NM0) specifies by how much an input voltage representing logic 0 can change before an error occurs due to it being interpreted as 1.

The high noise margin (NMH or NM1) specifies by how much an input voltage representing logic 1 can change before an error occurs due to it being interpreted as 0.
A typical voltage characteristic (output voltage versus input voltage) of a digital gate is shown below.

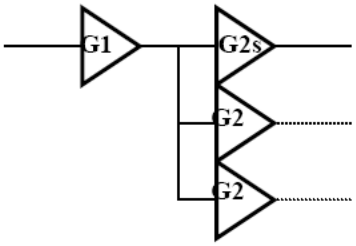


V_{OH} is the high output level
V_{OL} is the low output level
V_{IL} is the maximum low (logic 0) input voltage
V_{IH} is the minimum high (logic 1) input voltage

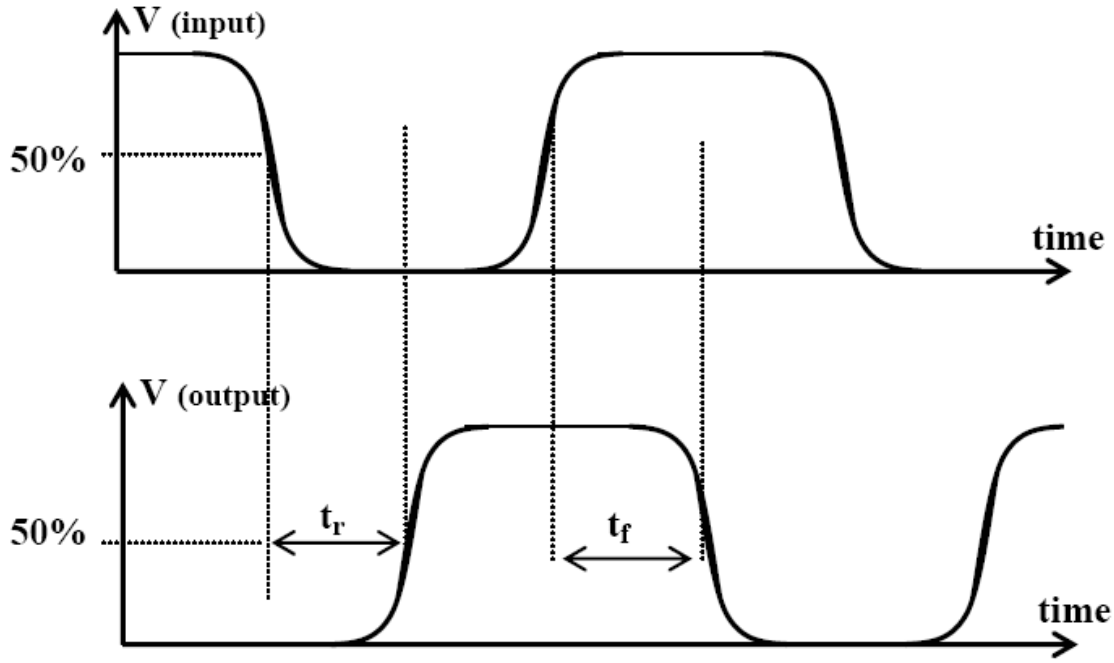
NML = V_{IL} - V_{OL}

NMH = V_{OH} – V_{IH}

For these gates to operate properly,
V_{OL} of G1 should be < V_{IL} of G2 &
V_{OH} of G1 should be > V_{IH} of G2 &
I_{OH} of G1 should be > 3 x I_{IH} of G2 &
I_{OL} of G1 should be > 3 x I_{IL} of G2



Propagation Delay: The figure below shows the input and output waveforms of a digital gate where all the delay parameters are defined on the figure.



Average propagation delay $T_D = (t_r + t_f)/2$

Power Dissipation: In general, the amount of power dissipated by a logic gate has two components a static one and a dynamic one.

a. Static Power (PSt) is due to DC current flow between the two supplies (VDD and ground) and = IDC x VDD

b. Dynamic Power is due to the charging and discharging of capacitances at the outputs of the gates. These capacitances are made of wiring capacitances, capacitances of the output transistors of the gate itself, and input capacitances of the gates connected to the output of the gate (the Fan out). The average value of the dynamic power PD = f x C x VDD2 where f is the switching frequency in hertz and C is the output (load) capacitance.

Total power PT = PSt + PD .

Week 10

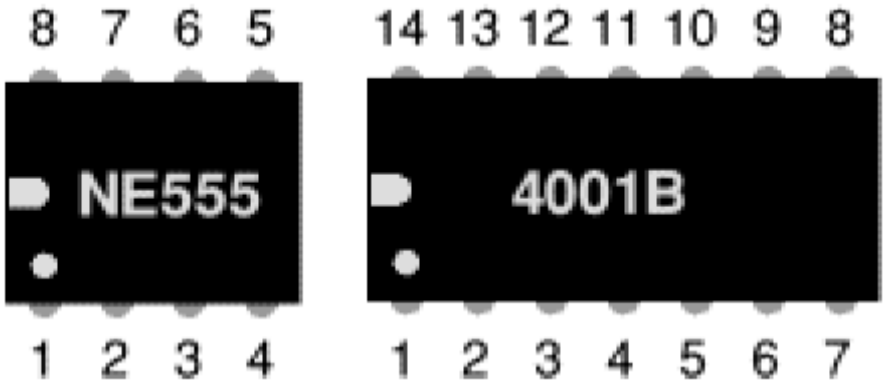
Objective:
1. Understand Small -Scale Integrated Circuit.
2. Explain Pin connections of ICs.

Integrated Circuits (Chips)

Integrated Circuits are usually called ICs or chips. They are complex circuits which have been etched onto tiny chips of semiconductor (silicon). The chip is packaged in a plastic holder with pins spaced on a 0.1" (2.54mm) grid which will fit the holes on stripboard and breadboards. Very fine wires inside the package link the chip to the pins.

Pin numbers

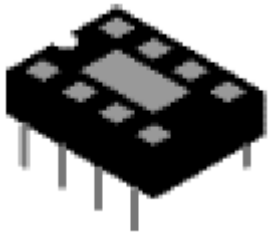
The pins are numbered anti-clockwise around the IC (chip) starting near the notch or dot. The diagram shows the numbering for 8-pin and 14-pin ICs, but the principle is the same for all sizes.



Chip holders (DIL sockets)

ICs (chips) are easily damaged by heat when soldering and their short pins cannot be protected with a heat sink. Instead we use a chip holder, strictly called a DIL socket (DIL = Dual In-Line), which can be safely soldered onto the circuit board. The chip is pushed into the holder when all soldering is complete.

Chip holders are only needed when soldering so they are not used on breadboards.
Commercially produced circuit boards often have chips soldered directly to the board without a chip holder, usually this is done by a machine which is able to work very quickly. Please don't attempt to do this yourself because you are likely to destroy the chip and it will be difficult to remove without damage by de-soldering.



Now-a-days, most of the logic or digital systems are available in the market as digital IC building blocks in various logic families. In fact, it is comparatively more convenient and cheaper to build logic circuits and systems using ICs which are more reliable compared to discrete components gates. In this chapter we shall be discussing the ICs based on packing density, IC series and their handling procedures.

Categories of Integrated Circuits Based on Packing Density

Ssi (Small scale integration) means integration levels typically below 12 equivalent gates per IC package.

MSI (Medium scale integration) means integration typically between 12 and 100 equivalent gates per IC package.

LSI (Large scale integration) implies integration typically above IDO equivalent gates per IC package.

VLSI (Very large scale integration) means integration levels with extra high number of gates. For example, a RAM may have more than 4000 gates in a single chip.

Logic IC Series

Commonly used Logic IC Families are

a. Standard TTL (Type 74/54)
b. CMOS (Type 4000 B)
c. Low power Schottky TTL (Type 74LS/54LS)
d. Schottky TTL (Type 74S/54S)
e. ECL (Type 10,000).

Packages in Digital ICs

Digital ICs come in four major packaged forms. These forms are shown in Fig. 1.3.1.
Dual-in-Line Package (DIP)" Most TTL and MOS devices in SSI, MSI and LSI are packaged in 14, 16, 24 or 40 pin DIPs.

Mini Dual-in-Line Package (Mini DIP) Mini DIPs are usually 8 pin packages.

Flat Pack Flat packages are commonly used in applications where light weight is essential requirement. Many military and space applications use flat packs. The number of pins on a flat pack varies from device to device. TO-5, TO-S Metal Can The number of pins on a TO-5 or TO-8 can vary from 2 to 12.

All the above styles of packaging have different systems of numbering pins. For knowing about how the pins of a particular package are numbered, the manufacturer's data sheet on package type and pin numbers must be consulted.

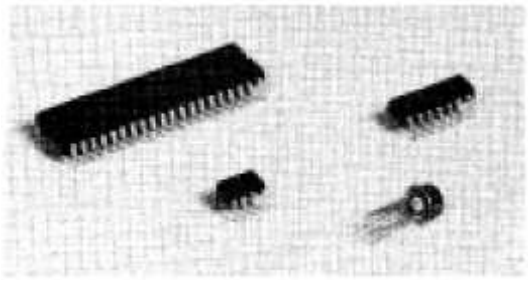


Fig. 1 Typical packaging systems in digital integrated circuits.

Identification of Integrated Circuits

Usually the digital integrated circuits come in a dual-in-line (DIP) package. Sometimes, the device in a DIP package may be an analog component –an operational amplifier or tapped resistors and therefore, it is essential to understand as to how to identify a particular IC. In a schematic diagram, the ICs are represented in one of the two methods.

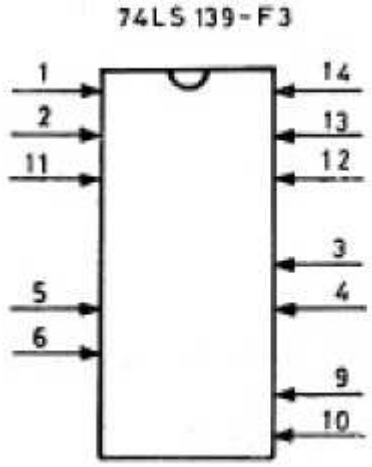
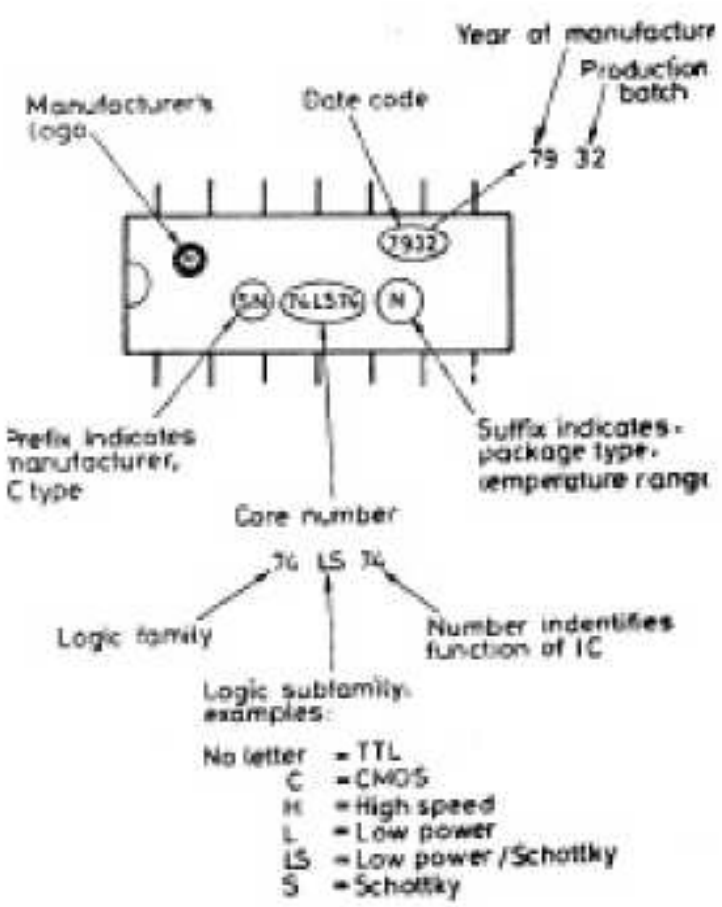


Fig. 1.3.2 Representation of schemes for digital ICs

a. IC is represented by a rectangle (Fig. 1.3.2) with pin numbers shown along with each pin. The identification number of the IC is listed on the schematic.

b. Representation of the IC in terms of its simple logic elements. For example, IC 74LS08 is Quad 2-input and gate and when it is represented in a schematic, it is listed as 74 LS 08.

An IC can be identified from the informa-tion-given on the IC itself. The numbering system, though has been standardized, has some variations from manufacturer to manufacturer. Usually an IC has the following markings on its surface (Fig. 1.3.3).



Core Number identifies the logic family and its functions. In 74 LS 51, the first two numbers indicate that the IC is a member of the 7400 series IC family. Last letters give the function of the IC. Letters inserted in the centre of the core number show the logic sub-family. Since TTL is the most common series,

Week 11

Objective:

1. Understand TTL Technology
2. Understand DTL Technology
3. Understand ECL Technology

Transistor-Transistor Logic (TTL) Technology

Transistor–transistor logic (**TTL**) is a class of digital circuits built from bipolar junction transistors (BJT), and resistors. It is called *transistor–transistor logic* because both the logic gating function (e.g., AND) and the amplifying function are performed by transistors (contrast this with RTL and DTL).

TTL is notable for being a widespread integrated circuit (IC) family used in many applications such as computers, industrial controls, test equipment and instrumentation, consumer electronics, synthesizers, etc. The designation *TTL* is sometimes used to mean TTL-compatible logic levels, even when not associated directly with TTL integrated circuits, for example as a label on the inputs and outputs of electronic instruments.

TTL contrasts with the preceding resistor–transistor logic (RTL) and diode–transistor logic (DTL) generations by using transistors not only to amplify the output, but also to isolate the inputs. The p-n junction of a diode has considerable capacitance, so changing the logic level of an input connected to a diode, as in DTL, requires considerable time and energy.

TTL is particularly well suited to integrated circuits because the inputs of a gate may all be integrated into a single base region to form a multiple-emitter transistor. Such a highly customized part might increase the cost of a circuit where each transistor is in a separate package. However, by combining several small on-chip components into one larger device, it reduces the cost of implementation on an IC.

As with all bipolar logic, a small amount of current must be drawn from a TTL input to ensure proper logic levels. The total current drawn must be within the capacities of the preceding stage, which limits the number of nodes that can be connected **(the fanout).**

All standardized common TTL circuits operate with a 5-volt power supply. A TTL input signal is defined as "low" when between 0 V and 0.8 V with respect to the ground terminal, and "high" when between 2.2 V and 5 V (precise logic levels vary slightly between sub-types). Standardization of the TTL levels was so ubiquitous that complex circuit boards often contained TTL chips made by many manufacturers, selected for availability and cost and not just compatibility. Within usefully broad limits, logic gates could be treated as ideal Boolean devices without concern for electrical limitations.

Comparison with other logic families

TTL devices consume substantially more power than an equivalent CMOS device at rest, but power consumption does not increase with clock speed as rapidly as for CMOS devices. Compared to contemporary ECL circuits, TTL uses less power and has easier design rules, but is substantially slower. Designers could combine ECL and TTL devices in the same system to achieve best overall performance and economy, but level-shifting devices were required between the two logic families. TTL was less sensitive to damage from electrostatic discharge than early CMOS devices.

Due to the output structure of TTL devices, the output impedance is asymmetrical between the high and low state, making them unsuitable for driving transmission lines. This is usually solved by buffering the outputs with special line driver devices where signals need to be sent through cables. ECL, by virtue of its symmetric low-impedance output structure, does not have this drawback.

Applications of TTL

Before the advent of VLSI devices, TTL integrated circuits were a standard method of construction for the processors of mini-computer and mainframe processors; such as the DEC VAX and Data General Eclipse, and for equipment such as machine tool numerical controls, printers, and video display terminals. As microprocessors became more functional, TTL devices became important for "glue logic" applications, such as fast bus drivers on a motherboard, which tie together the function blocks realized in VLSI elements.

Diode–transistor logic

Diode–Transistor Logic (**DTL**) is a class of digital circuits built from bipolar junction transistors (BJT), diodes and resistors; it is the direct ancestor of transistor–transistor logic. It is called *diode–transistor logic* because the logic gating function (e.g., AND) is performed by a diode network and the amplifying function is performed by a transistor .

Operation

With the simplified circuit shown in the picture the voltage at the base will be near 0.7 volts even when one input is held at ground level, which results in unstable or invalid operation. Two diodes in series with R3 are commonly used to lower the base voltage and prevent any base current when one or more inputs are at low logic level. The IBM 1401 used DTL circuits almost identical to this simplified circuit, but solved the base bias level problem mentioned above by alternating NPN and PNP based gates operating on different power supply voltages instead of adding extra diodes.

Speed disadvantage

A major advantage over the earlier resistor–transistor logic is the increased fan-in. However, the propagation delay is still relatively large. When the transistor goes into saturation from all inputs being high, charge is stored in the base region. When it comes out of saturation (one input goes low) this charge has to be removed and will dominate the propagation time. One way to speed it up is to connect a resistor to a negative voltage at the base of the transistor which aids the removal of the minority carriers from the base.

The above problem is solved in TTL by replacing the diodes of the DTL circuit with a multiple-emitter transistor, which also slightly reduces the required area per gate in an integrated circuit implementation.

Emitter-coupled logic

In electronics, **emitter-coupled logic**, or **ECL**, is a logic family in which current is steered through bipolar transistors to implement logic functions. ECL is sometimes called current-mode logic or current-switch emitter-follower (CSEF) logic.

The chief characteristic of ECL is that the transistors are never in the saturation region and can thus change states at very high speed. Its major disadvantage is that the circuit continuously draws current, which means it requires a lot of power.

History

ECL was invented in August 1956 at IBM by Hannon S. Yourke. Originally called *current steering logic*, it was used in the Stretch, IBM 7090, and IBM 7094 computers.

While ECL circuits in the mid-1960s through the 1990s consisted of a differential amplifier input stage to perform logic, followed by an emitter follower to drive outputs and shift the output voltages so they will be compatible with the inputs, Yourke's current switch, also known as ECL, consisted only of differential amplifiers. To provide compatible input and output levels, two complementary versions were used, an NPN version and a PNP version. The NPN output could drive PNP inputs, and vice-versa. "The disadvantages are that more different power supply voltages are needed, and both pnp and npn transistors are required." Motorola introduced their first digital monolithic integrated circuit line, MECL I, in 1962.

Explanation

TTL and related families use transistors as digital switches where transistors are either cut off or saturated, depending on the state of the circuit. ECL gates use differential amplifier configurations at the input stage. A bias configuration supplies a constant voltage at the midrange of the low and high logic levels to the differential amplifier, so

that the appropriate logical function of the input voltages will control the amplifier and the base of the output transistor (this output transistor is used in common collector configuration). The propagation time for this arrangement can be less than a nanosecond, making it for many years the fastest logic family.

Characteristics

Other noteworthy characteristics of the ECL family include the fact that the large current requirement is approximately constant, and does not depend significantly on the state of the circuit. This means that ECL circuits generate relatively little power noise, unlike many other logic types which typically draw far more current when switching than quiescent, for which power noise can become problematic. In an ALU - where a lot of switching occurs - ECL can draw lower mean current than CMOS.

Usage

The drawbacks associated with ECL have meant that it has been used mainly when high performance is a vital requirement. Other families (particularly advanced CMOS variants) have replaced ECL in many applications, even mainframe computers. However, some experts predict increasing use of ECL in the future, particularly in conjunction with more widespread adoption of advanced semiconductors such as gallium arsenide, which has always been seen as the semiconductor of the future, but cannot be produced as cheaply or cleanly as silicon.

Older high-end mainframe computers, such as the Enterprise System/9000 members of IBM's ESA/390 computer family, used ECL; current IBM mainframes use CMOS.

The equivalent of emitter-coupled logic made out of FETs is called source-coupled FET logic (SCFL).

Week 12

Objective:

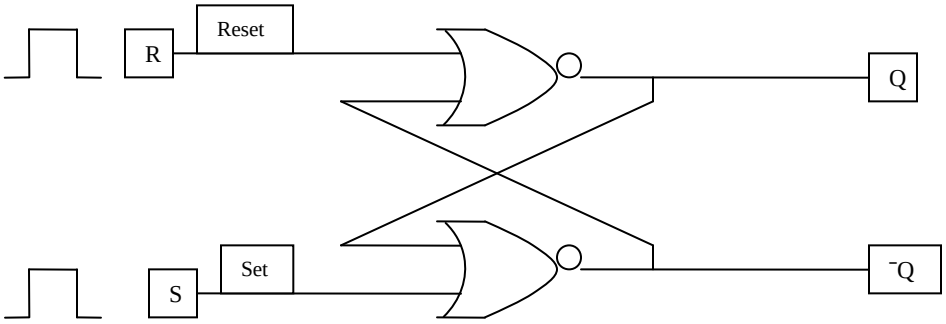
1. Understand the concept and methodology of sequential circuit design.
2. The design and operations of various bi-stables.

Introducing bistable (Flip-Flops)

A Flip-Flop is a sequential circuit which is capable of retaining a unit of information such as ‘0’, or ‘1’.

Basic Flip-Flop circuit

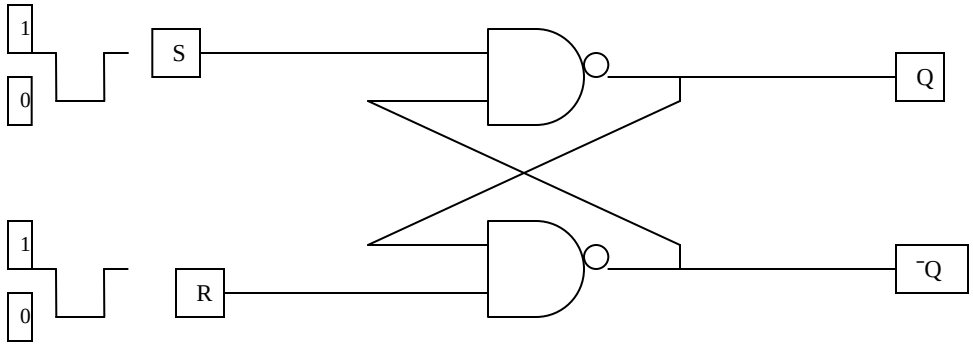
(a) Using NOR gates:



Truth Table

S	R	Q	¯Q
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
1	1	0	0

(b) Using NAND gates



S	R	Q	¯Q
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
1	1	1	0
0	0	1	1

Let's consider a general truth table

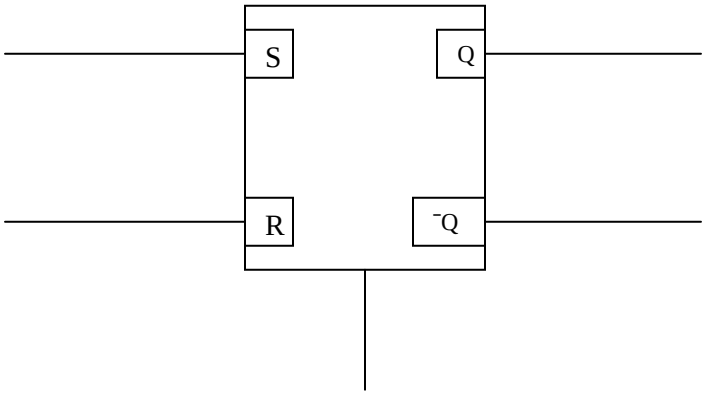
Qt = means present state at t=0
Qt-1 = means previous state at t=-0
-0 means a time very close to present

S	R	Qt-1	Qt	¯Qt
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	x	X	x
1	1	x	X	x

x = indeterminate

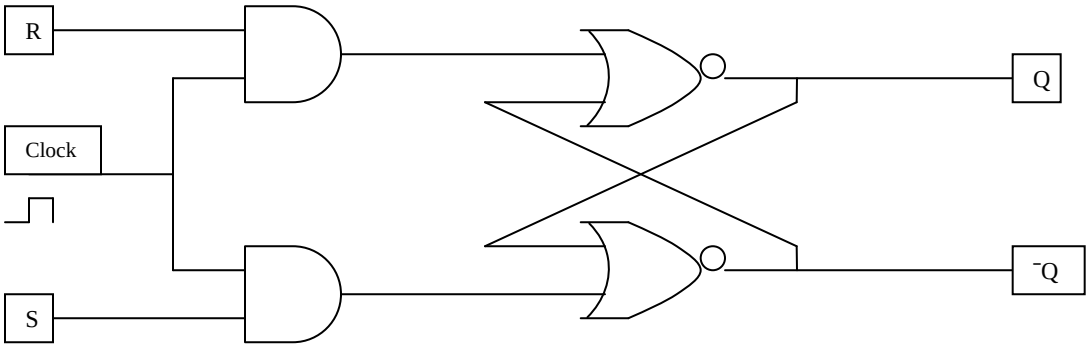
SET --------→ make Q = 1
Reset --------→ make Q =

Block representation of a Flip-Flop



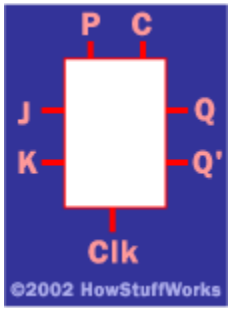
Clocked S-R Flip-Flop

To convert the basic Flip-Flop from **Asynchronous** to **Synchronous** a clock pulse must be incorporated



The J-K Flip-Flop

A very common form of flip-flop is the **J-K flip-flop**. It is unclear, historically, where the name "J-K" came from, but it is generally represented in a black box like this:



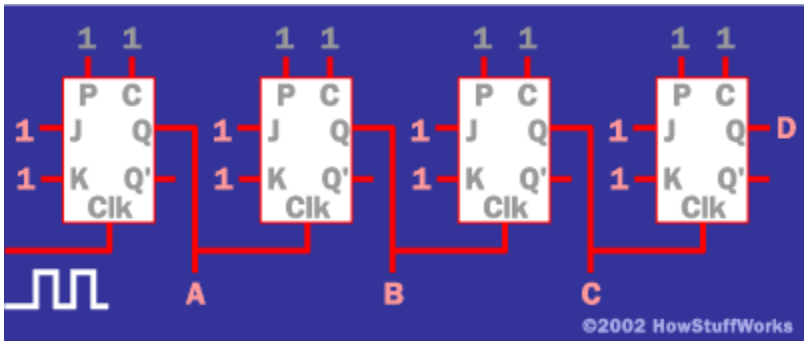
In this diagram, **P** stands for "Preset," **C** stands for "Clear" and **Clk** stands for "Clock." The logic table looks like this:

P	C	Clk	J	K	Q	Q'
1	1	1-to-0	1	0	1	0
1	1	1-to-0	0	1	0	1
1	1	1-to-0	1	1		Toggles
1	0	X	X	X	0	1
0	1	X	X	X	1	0

Here is what the table is saying: First, Preset and Clear override J, K and Clk completely. So if Preset goes to 0, then Q goes to 1; and if Clear goes to 0, then Q goes to 0 no matter what J, K and Clk are doing. However, if both Preset and Clear are 1, then J, K and Clk can operate. The **1-to-0** notation means that when the clock changes from a 1 to a 0, the value of J and K are remembered if they are opposites. At the **low-going edge** of the clock (the transition from 1 to 0), J and K are stored. However, if both J and K happen to be 1 at the low-going edge, then Q simply **toggles**. That is, Q changes from its current state to the opposite state.

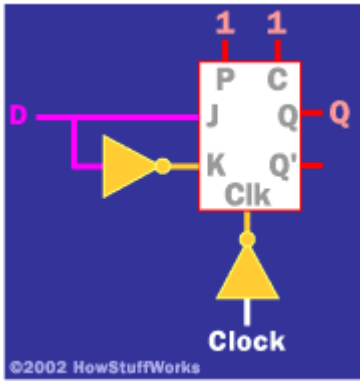
The concept of "edge triggering" is very useful. The fact that J-K flip-flop only "latches" the J-K inputs on a transition from 1 to 0 makes it much more useful as a memory device. J-K flip-flops are also extremely useful in **counters** (which are used extensively when

creating a digital clock). Here is an example of a 4-bit counter using J-K flip-flops:



The outputs for this circuit are A, B, C and D, and they represent a 4-bit binary number. Into the clock input of the left-most flip-flop comes a signal changing from 1 to 0 and back to 1 repeatedly (an **oscillating signal**). The counter will count the low-going edges it sees in this signal. That is, every time the incoming signal changes from 1 to 0, the 4-bit number represented by A, B, C and D will increment by 1. So the count will go from 0 to 15 and then cycle back to 0. You can add as many bits as you like to this counter and count anything you like. For example, if you put a magnetic switch on a door, the counter will count the number of times the door is opened and closed. If you put an optical sensor on a road, the counter could count the number of cars that drive by.

Another use of a J-K flip-flop is to create an **edge-triggered latch**, as shown here:



In this arrangement, the value on D is "latched" when the clock edge goes from low to high. **Latches** are extremely important in the design of things like central processing units (CPUs) and peripherals in computers.

Week 13

Objective:

1. Understand Digital Counters
2. Understand Ripple (Asynchronous) Counter
3. Understand Synchronous Counters
4. Introduce Shift Registers

Digital Counters

A **digital counter**, or simply counter, is a semiconductor device that is used for counting the number of times that a digital event has occurred. The counter's output is indexed by one LSB every time the counter is clocked.

A simple implementation of a 4-bit counter is shown in Figure 1, which consists of 4 stages of cascaded J-K flip-flops. This is a binary counter, since the output is in binary system format, i.e., only two digits are used to represent the count, i.e., '1' and '0'. With only 4 bits, it can only count up to '1111', or decimal number 15.

As one can see from Figure 1, the J and K inputs of all the flip-flops are tied to '1', so that they will toggle between states every time they are clocked. Also, the output of each flip-flop in the counter is used to clock the next flip-flop. As a result, the succeeding flip-flop toggles between '1' and '0' at only half the frequency as the flip-flop before it.

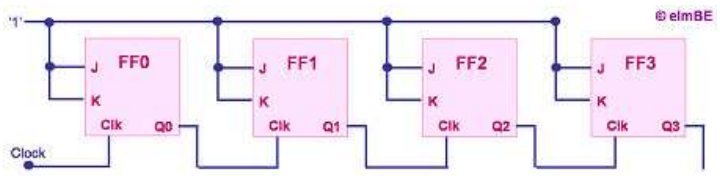


Figure 1. A Simple Ripple Counter Consisting of J-K Flip-flops

Thus, in Figure 1's 4-bit example, the last flip-flop will only toggle after the first flip-flop has already toggled 8 times. This type of binary counter is known as a 'serial', 'ripple', or 'asynchronous' counter. The name 'asynchronous' comes from the fact that this counter's flip-flops are not being clocked at the same time.

A 4-bit counter, which has 16 unique states that it can count through, is also called a modulo-16 counter, or mod-16 counter. By definition, a modulo-k or base-k counter is one that returns to its initial state after k cycles of the input waveform. A counter that has N flip-flops is a modulo 2^N counter.

An asynchronous counter has a serious drawback - its speed is limited by the cumulative propagation times of the cascaded flip-flops. A counter that has N flip-flops, each of which has a

propagation time t, must therefore wait for a duration equal to N x t before it can undergo another transition clocking.

A better counter, therefore, is one whose flip-flops are clocked at the same time. Such a counter is known as a synchronous counter. A simple 4-bit synchronous counter is shown in Figure 2.

Not all counters with N flip-flops are designed to go through all its 2^N possible states of count. In fact, digital counters can be used to output decimal numbers by using logic gates to force them to reset when the output becomes equal to decimal 10. Counters used in this manner are said to be in binary-coded decimal (BCD).

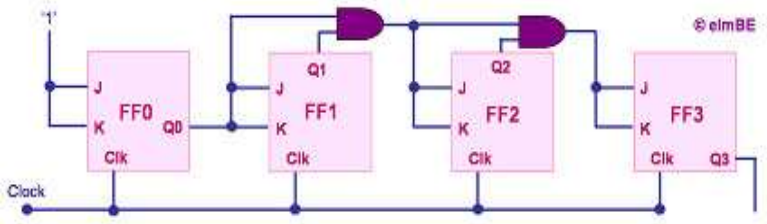
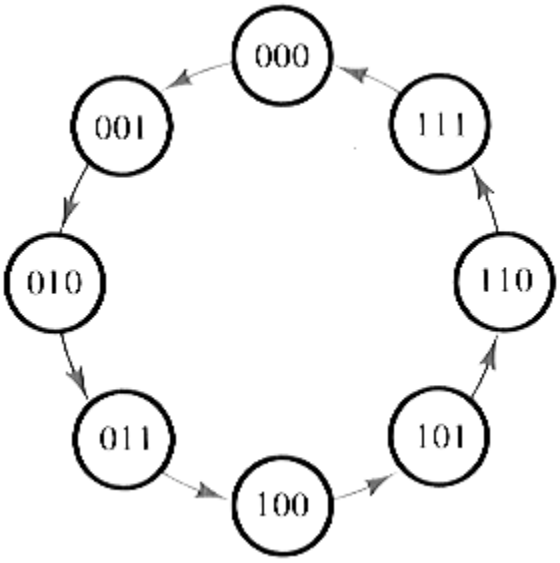


Figure 2. A Simple Synchronous Counter Consisting of J-K Flip-flops and AND gates

Synchronous Counters

Basically, any 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 or they may originate from an external source and may occur at prescribed intervals of time or at random. The sequence of states in a counter may follow a binary count or any other sequence.



Why do we need counters?

In a digital circuit, counters are used to do 3 main functions: timing, sequencing and counting. A timing problem might require that a high-frequency pulse train, such as the output of a 10-MHz crystal oscillator, be divided to produce a pulse train of a much lower frequency, say 1 Hz. This application is required in a precision digital clock, where it is not possible to build a crystal oscillator whose natural frequency is 1 Hz.

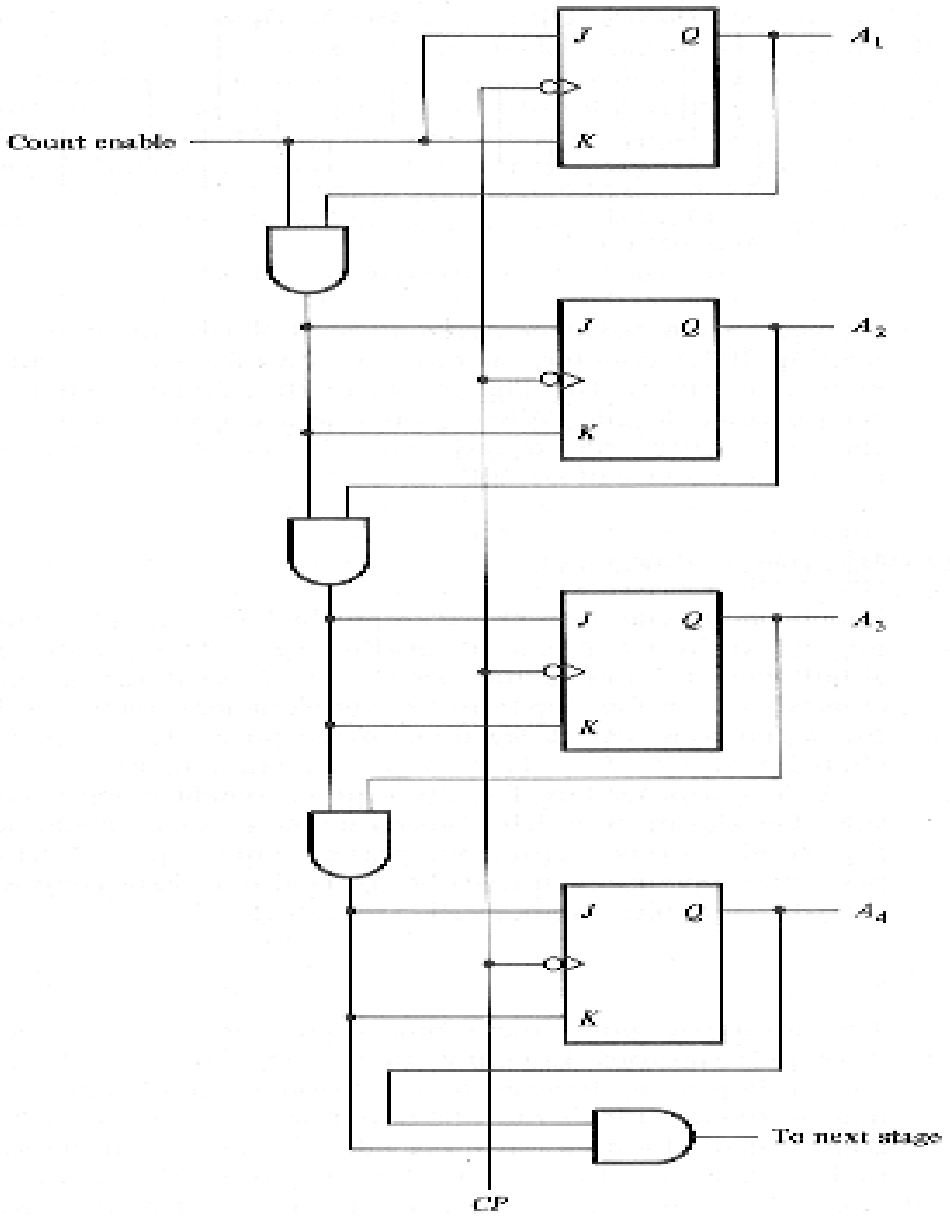
Measuring the flow of auto traffic on roadway is an application in which an event (the passage of a vehicle) must increment a tally. This can be done automatically with an electronic counter triggered by a photocell or road sensor. In this way, the total number of vehicles passing a certain point can be counted.

How are counters made?

Counters are generally made up of flip-flops and logic gates. Like flip-flops, counters can retain an output state after the input condition which brought about that state has been removed. Consequently, digital counters are classified as sequential circuits. While a flip-flop can occupy one of only two possible states, a counter can have many more than two states. In the case of a counter, the value of a state is expressed as a multi-digit binary number, whose `1's and `0's are usually derived from the outputs of internal flip-flops that make up the counter. The number of states a counter may have is limited only by the amount of electronic hardware that is available. The main types of flip-flops used are J-K flip-flops or T flip-flops, which are J-K flip-flops with both J and K inputs tied together. Before that, here's a quick reminder of how a J-K flip-flop works:

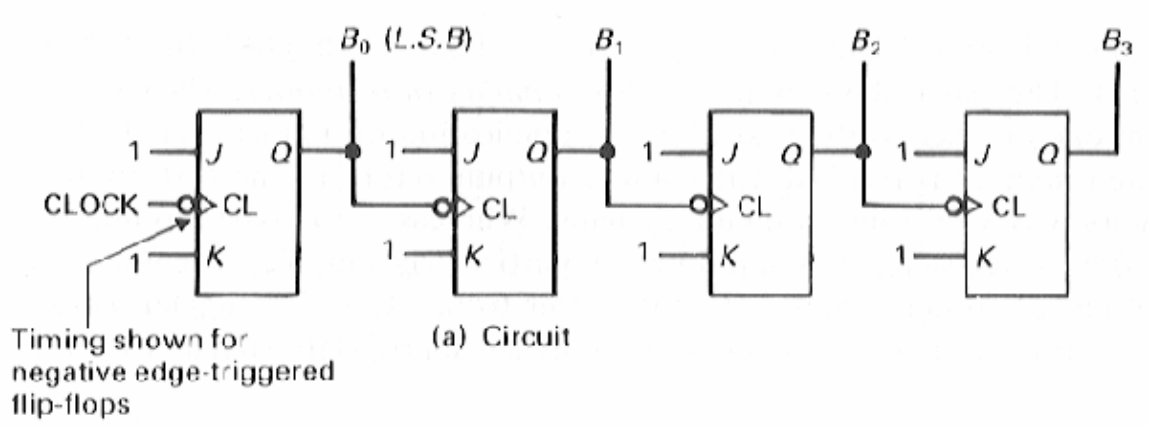
J input	K input	Output, Q
0	0	Q
0	1	0
1	0	1
1	1	not Q

T flip-flops are used because set/reset ([1,0] [0,1]) functions are seldom used. Only the "do nothing" and toggle ([0,0] [1,1]) functions are used. Logic gates are used to decide when to toggle which outputs. Below is an example of a synchronous binary counter, implemented using J-K flip-flops and AND gates.



The difference between asynchronous and synchronous counters.

In an asynchronous counter, an external event is used to directly SET or CLEAR a flip-flop when it occurs. In a synchronous counter however, the external event is used to produce a pulse that is synchronised with the internal clock. An example of an asynchronous counter is a ripple counter. Each flip-flop in the ripple counter is clocked by the output from the previous flip-flop. Only the first flip-flop is clocked by an external clock. Below is an example of a 4-bit ripple counter:



Shift registers, like counters, are a form of *sequential logic*. Sequential logic, unlike combinational logic is not only affected by the present inputs, but also, by the prior history. In other words, sequential logic remembers past events.

Shift registers produce a discrete delay of a digital signal or waveform. A waveform synchronized to a *clock*, a repeating square wave, is delayed by **"n"** discrete clock times, where **"n"** is the number of shift register stages. Thus, a four stage shift register delays "data in" by four clocks to "data out". The stages in a shift register are *delay stages*, typically type **"D"** Flip-Flops or type **"JK"** Flip-flops.

Formerly, very long (several hundred stages) shift registers served as digital memory. This obsolete application is reminiscent of the acoustic mercury delay lines used as early computer memory.

Serial data transmission, over a distance of meters to kilometers, uses shift registers to convert parallel data to serial form. Serial data communications replaces many slow parallel data wires with a single serial high speed circuit.

Serial data over shorter distances of tens of centimeters, uses shift registers to get data into and out of microprocessors. Numerous peripherals, including analog to digital converters, digital to analog converters, display drivers, and memory, use shift registers to reduce the amount of wiring in circuit boards.

Some specialized counter circuits actually use shift registers to generate repeating waveforms. Longer shift registers, with the help of feedback generate patterns so long that they look like random noise, *pseudo-noise*.

Basic shift registers are classified by structure according to the following types:

- Serial-in/serial-out
- Parallel-in/serial-out
- Serial-in/parallel-out
- Universal parallel-in/parallel-out
- Ring counter

Week 14

Objective:

Understand the basic design of counters

Design of Counters

This example is taken from T. L. Floyd, *Digital Fundamentals*, Fourth Edition, Macmillan Publishing, 1990, p.395.

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.

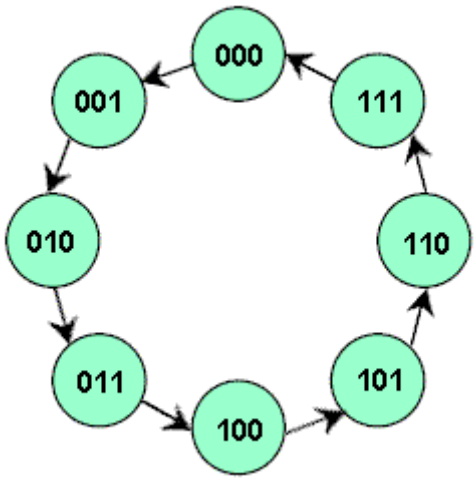
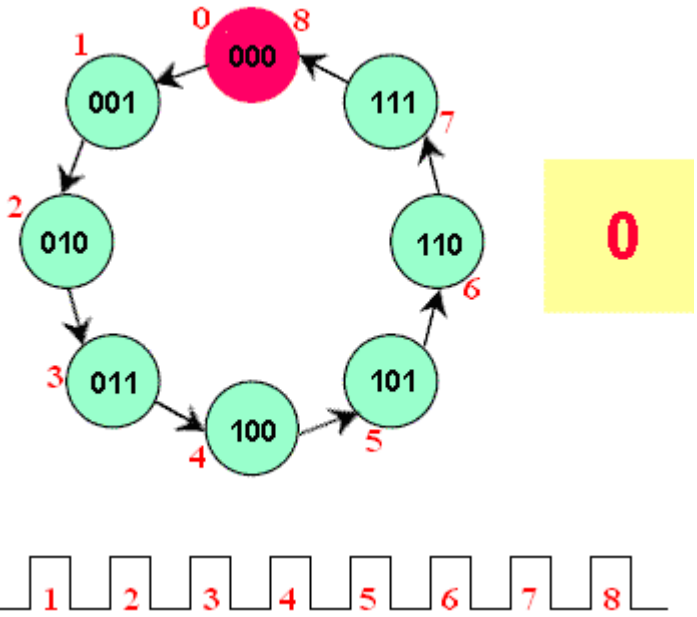


Figure 18. State diagram of a 3-bit binary counter.

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.



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 1.

Table 1. 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.

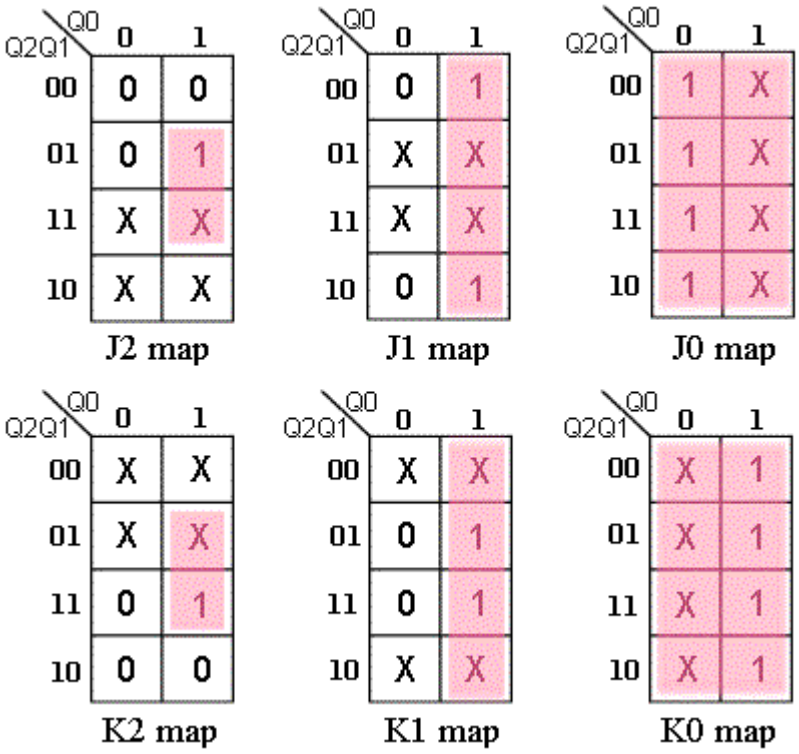


Figure 20. Karnaugh maps

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.

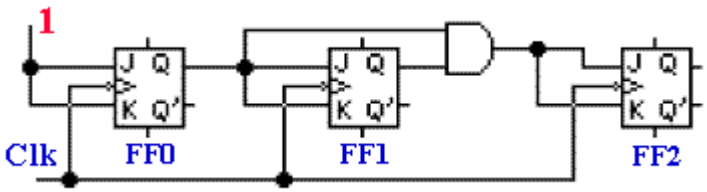


Figure 21. Logic diagram of a 3-bit binary counter

Week 15

Objective:

1. Understand shift register operation
2. Understand the different methods of data transfer with shift registers
3. Scaling

SHIFT REGISTERS

A shift register is a register in which the contents may be shifted one or more places to the left or right. This type of register is capable of performing a variety of functions. It may be used for serial-to-parallel conversion and for scaling binary numbers.

Before we get into the operation of the shift register, let's discuss serial-to-parallel conversion, parallel-to-serial conversion, and scaling.

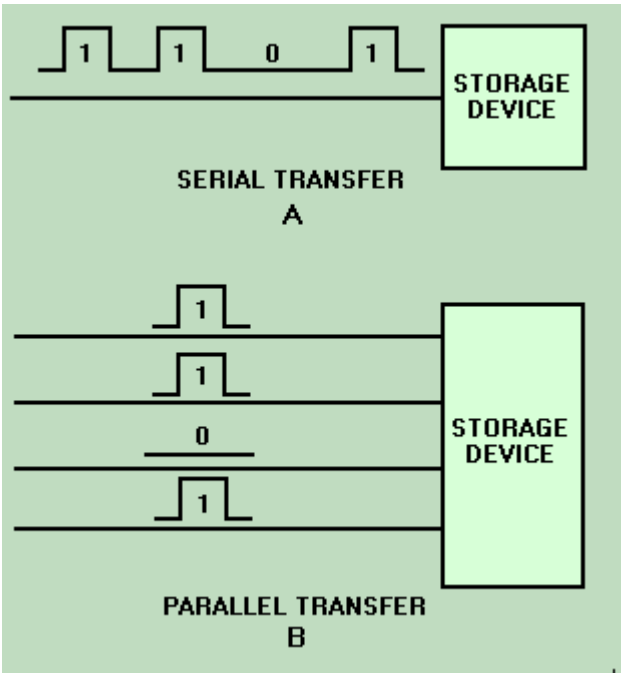
Serial and Parallel Transfers and Conversion

Serial and parallel are terms used to describe the method in which data or information is moved from one place to another. SERIAL TRANSFER means that the data is moved along a single line one bit at a time. A control pulse is required to move each bit. PARALLEL TRANSFER means that each bit of data is moved on its own line and that all bits transfer simultaneously as they did in the parallel register. A single control pulse is required to move all bits.

Figure 1 shows how both of these transfers occur. In each case, the four-bit word 1101 is being transferred to a storage device. In view A, the data moves along a single line. Each bit of the data will be stored by an individual control pulse. In view B, each bit has a separate input line. One control pulse will cause the entire word to be stored.

Figure 1. - Data transfer methods: A. Serial transfer; B.

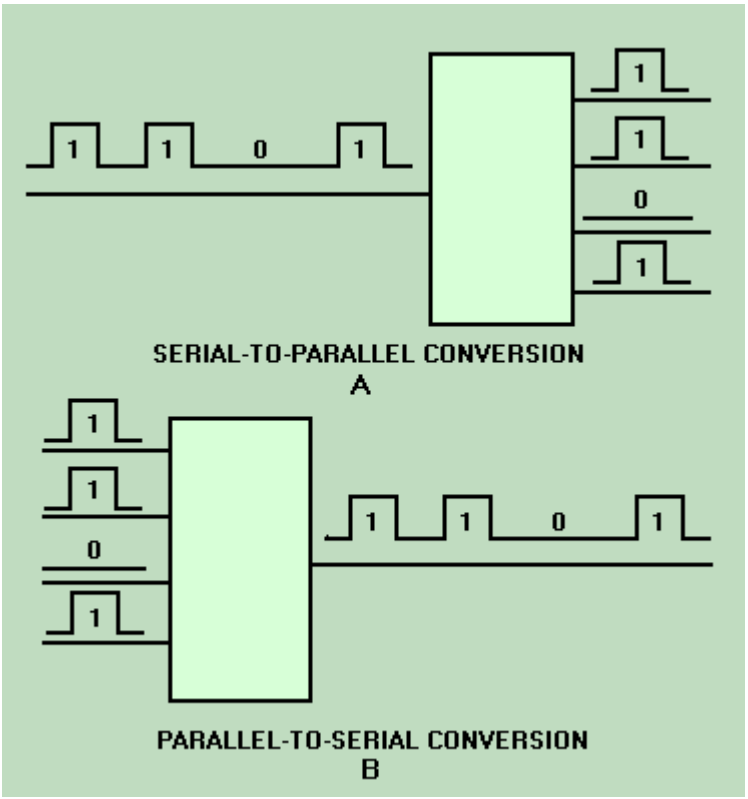
Parallel transfer.



Serial-to-parallel conversion or parallel-to-serial conversion describes the manner in which data is stored in a storage device and the manner in which that data is removed from the storage device.

Serial-to-parallel conversion means that data is transferred into the storage device or register in serial fashion and removed in parallel fashion, as in figure 3-30, view A. Parallel-to-serial conversion means the data is transferred into the storage device in parallel and removed as serial data, as shown in view B.

Figure 2. - Data conversion methods: A. Serial-to-parallel; B. Parallel-to-serial.



Serial transfer takes time. The longer the word length, the longer the transfer will take. Although parallel transfer is much faster, it requires more circuitry to transfer the data.

Scaling

SCALING means to change the magnitude of a number. Shifting binary numbers to the left increases their value, and shifting to the right decreases their value. The increase or decrease in value is based on powers of 2.

A shift of one place to the left increases the value by a power of 2, which in effect is multiplying the number by 2. To demonstrate this, let's assume that the following block diagram is a 5-bit shift register containing the binary number 01100.

0 1 1 0 0

Shifting the entire number one place to the left will put the register in the following condition:

1 1 0 0 0

The binary number 01100 has a decimal equivalent of 12. If we convert 11000_2 to decimal, we find it has a value of 24_{10}. By shifting the number one place to the left, we have multiplied it by 2. A shift of two places to the left would be the equivalent of multiplying the number by 2^2, or 4; three places by 2^3, or 8; and so forth.

Shifting a binary number to the right decreases the value of the number by a power of 2 for each place. Let's look at the same 5-bit register containing 01100_2 and shift the number to the right.

0 1 1 0 0

A shift of one place to the right will result in the register being in the following condition:

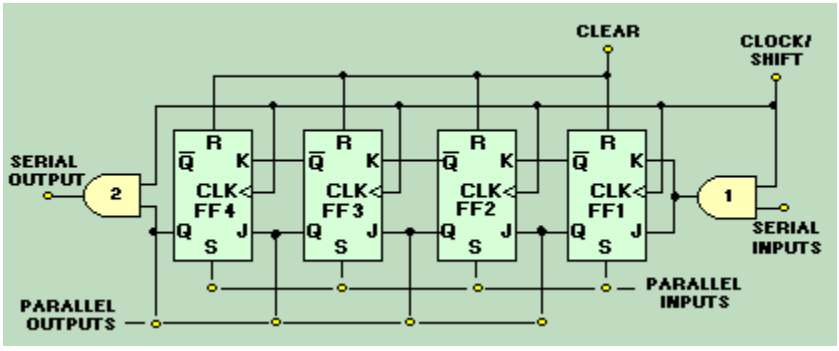
0 0 1 1 0

By comparing decimal equivalents you can see that we have decreased the value from 12_{10} to 6_{10}. We have effectively divided the number by 2. A shift of two places to the right is the equivalent of dividing the number by 2^2, or 4; three places by 2^3; or 8; and so forth.

Shift Register Operations

Figure 3-31 shows a typical 4-bit shift register. This particular register is capable of left shifts only. There are provisions for serial and parallel input and serial and parallel output. Additional circuitry would be required to make right shifts possible.

Figure 3. - Shift register.



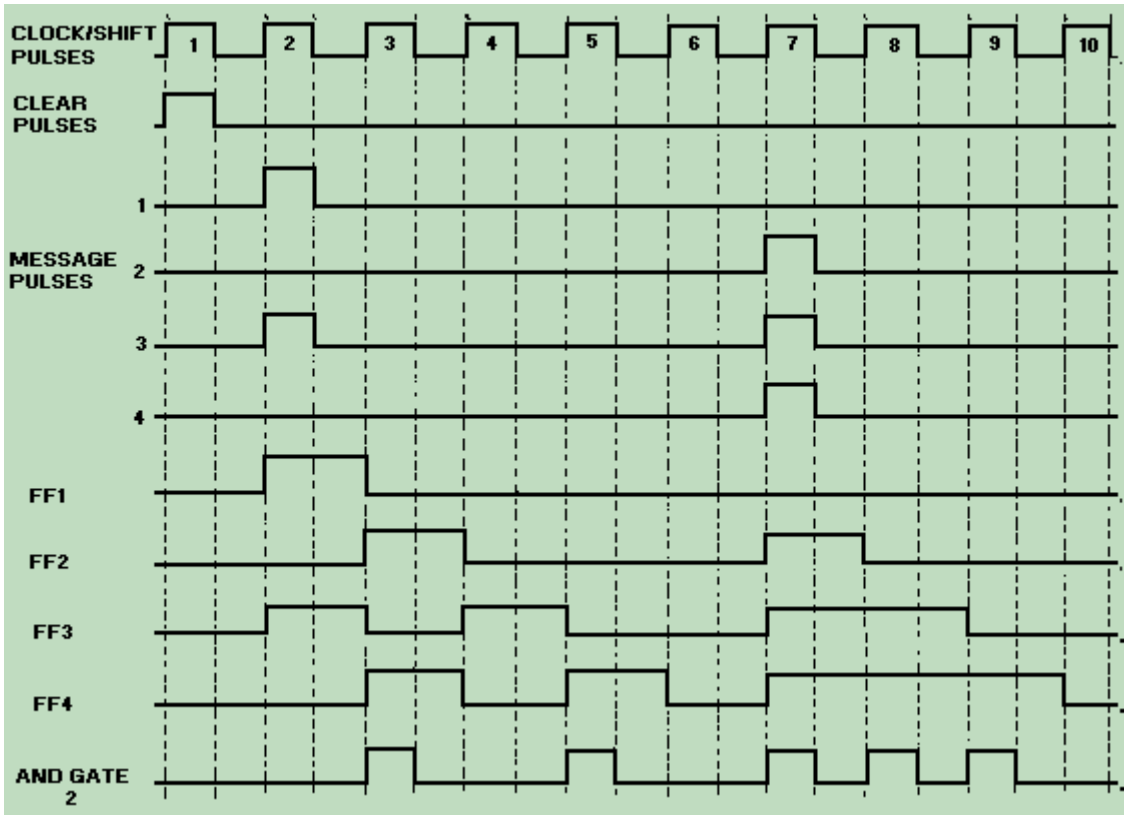
Before any operation takes place, a CLEAR pulse is applied to the RESET terminal of each FF to ensure that the Q output is LOW.

The simplest modes of operation for this register are the parallel inputs and outputs. Parallel data is applied to the SET inputs of the FFs and results in either a 1 or 0 output, depending on the input. The outputs of the FFs may be sampled for parallel output. In this mode, the register functions just like the parallel register covered earlier in this section.

Parallel-to-Serial Conversion

Now let's look at parallel-to-serial conversion. We will use the 4-bit shift register in figure 3 and the timing sequence in figure 4 to aid you in understanding the operations.

Figure 4. - Parallel-to-serial conversion timing diagram.



At CP1, a CLEAR pulse is applied to all the FFs, resetting the register to a count of 0. The number 0101_2 is applied to the parallel inputs at CP2, causing FF1 and FF3 to set. At this point, the J inputs of FF2 and FF4 are HIGH. AND gate 2 has a LOW output since the FF4 output is LOW. This LOW output represents the first digit of the number 0101_2 to be output in serial form. At the same time we have HIGHs on the K inputs of FF1 and FF3. (Notice the NOT symbol on FF1 at input K. With no serial input to AND gate 1, the output is LOW; therefore, the K input to FF1 is held HIGH). With these conditions CP3 causes FF1 and FF3 to reset and FF2 and FF4 to set. The HIGH output of FF4, along with CP3, causes AND gate 2 to output a HIGH. This represents the second digit of the number 0101_2.

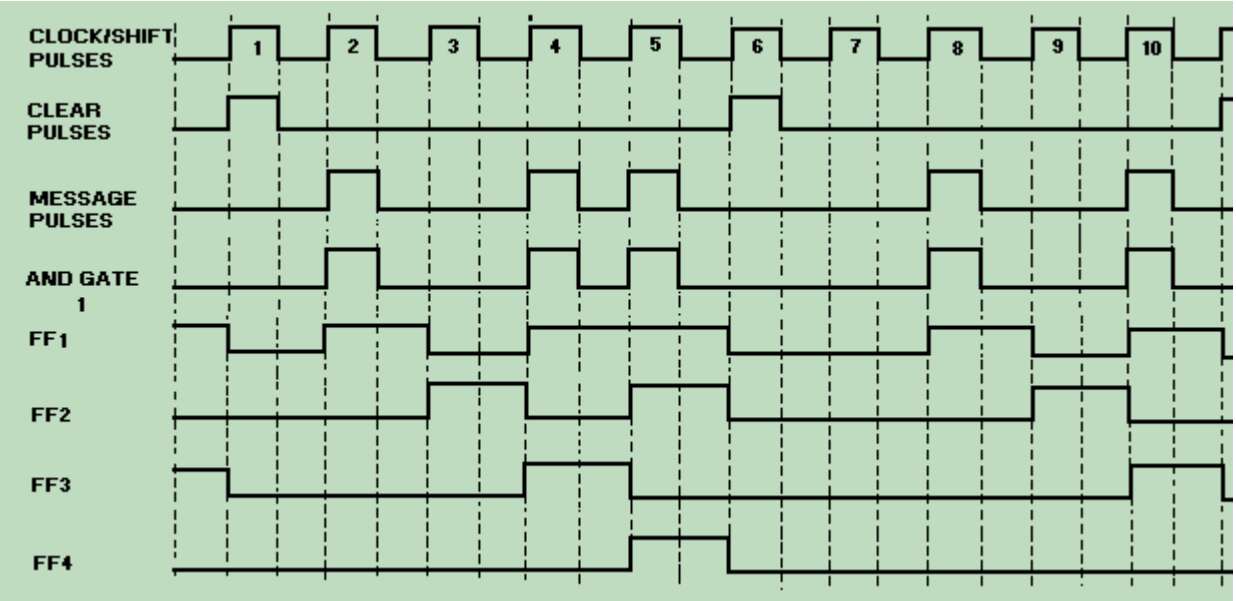
At CP4, FF2 and FF4 reset, and FF3 sets. FF1 remains reset because of the HIGH at the K input. The output of AND gate 2 goes LOW because the output of FF4 is LOW and the third digit of the number is output on the serial line. CP5 causes FF4 to set and FF3 to reset. CP5 and the HIGH from FF4 cause AND gate 2 to output the last digit of the number on the serial line. It took a total of four CLK pulses to input the number in parallel and output it in serial. CP6 causes FF4 to reset and effectively clears the register for the next parallel input. Between CP7 and CP10, the number 1110_2 is input as parallel data and output as serial data.

Serial-to-Parallel Conversion

Serial input is accomplished much in the same manner as serial output. Instead of shifting the data out one bit at a time, we shift the data in one bit at a time.

To understand this conversion, you should again use figure 3 and also the timing diagram shown in figure 5. In this example we will convert the number 1011_2 from serial data to parallel data.

Figure 5. - Serial-to-parallel conversion timing diagram.



A CLEAR pulse resets all the FFs at CP1. At CP2, the most significant bit of the data is input to AND gate 1. This HIGH along with the clock pulse causes AND gate 1 to output a HIGH. The HIGH from the AND gate and the clock pulse applied to FF1 cause the FF to set. FFs 2, 3, and 4 are held reset. At this point, the MSD of the data has been shifted into the register.

The next bit of data is a 0. The output of AND gate 1 is LOW. Because of the inverter on the K input of FF1, the FF senses a HIGH at that input and resets. At the same time this is occurring, the HIGH on the J input of FF2 (from FF1) and the CLK cause FF2 to set. The two MSDs, 1 and 0, are now in the register.

CP4 causes FF3 to set and FF2 to reset. FF1 is set by the CLK pulse and the third bit of the number. The register now contains 0101_2, as a result of shifting the first three bits of data.

The remaining bit is shifted into the register by CP5. FF1 remains set, FF2 sets, FF3 resets, and FF4 sets. At this point, the serial transfer is complete. The binary word can be

sampled on the parallel output lines. Once the parallel data is transferred, a CLEAR pulse resets the FFs (CP6), and the register is ready to input the next word.