LECTURE NOTES ON

DIGITAL SYSTEM DESIGN

II B.Tech III semester

(Autonomous R18) (2019-20)

Dr. V Vijay, Associate Professor
Dr. P Munaswamy, Professor
Dr. Lalit Kumar Kaul, Professor

ELECTRONICS AND COMMUNICATIONENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)

Dundigal
Hyderabad 500043

DIGITAL SYSTEM DESIGN

LECTURE NOTES

UNIT I

LOGIC SIMPLIFICATION AND COMBINATIONAL LOGIC DESIGN

1.1 LOGICAL STATEMENTS AND ARGUMENTS

Logic is reasoning. A logical statement is one which may be either true or false. There is no scope for ambiguity in a logical statement. The statement ‘I am wearing a sweater’ will either be true or false. There cannot be any ambiguity in the truth or falsehood of the statement. On the other hand the statement ‘When I feel cold I may wear a sweater ’ is a statement which has no true or false answer and is quite ambiguous. This statement could well be broken down into several components as a) I shall feel cold if the temperature falls below 15 degrees Celsius or if I have fever b) I shall wear a sweater. The statement in a) has three independent statements i) ‘I shall feel cold’ will be true if ii) The temperature falls below 15 degrees Celsius or iii) ‘I have fever’ We may tabulate the statements made above as shown in TABLE 1.1 (a). From this table, we observe all possible answers that the two component statements and the resulting statement can have. We allot symbols to the statements or represent them with switches as follows. If the statements are true we will say the switch is ON and if it is false we may say the switch is off.

‘A’ The temperature is below 15 degrees Celsius

‘B’ I have fever

‘X’ I shall wear a sweater

In TABLE 1.1 (b) the same argument depicted in TABLE 1.1 (a) is shown by representing TRUE with ON and FALSE with OFF.

In the table 1.1 (b) we may consider ‘A’, ‘B’, and ‘X’ as variables which can have one of two values ‘1’ for ON and ‘0’ for OFF. We may further represent table 9.1 (b) as table 9.1 (c) replacing ‘ON’ with ‘1’ and ‘ OFF’ with ‘0’.

George Boole symbolised logic i.e., provided symbols to represent logical statements and defined the operators between logical statements to build logical arguments. In the above argument ‘A’, ‘B’ and ‘X’ are Boolean variable and ‘OR’ is the operation between ‘A’ and ‘B’.   X is the resulting statement. We can represent the truth table shown in table 1.1 (c) shown above in the form of a Boolean Expression as (A+B =X) the ‘+’ sign between ‘A’ and ‘B’ represents a logical OR operation in Boolean Algebra and not addition.

1.2 LOGIC GATES AND OPERATIONS:- AND, OR & NOT

Logic gates are diagrammatic representations of the Logic Operators like ‘OR’, ‘AND’ and ‘NOT’. The logic gates have one or more inputs and only one output. There are three basic logic gates which perform the three basic logic functions ‘OR’, ‘AND’ and ‘NOT’. The FIG. 1.1 shows the logic diagram for each of these and also their algebraic symbols. In section 1.1 the truth table for each of these operations has already been shown.

We may show electrical connections of switches to depict the OR and AND operations as in FIG. 1.2(a) and FIG. 1.2 (b). It may be observed that in OR operation the bulb will light if ‘A’ OR ‘B’ switch is closed or both are closed. In AND operation the bulb will light only if switches ‘A’ and ‘B’ are both closed. FIG. 1.2 (c) the bulb will light only if switch ‘A’ is off indicating the ‘NOT’ function.

1.3 BOOLEAN ALGEBRA

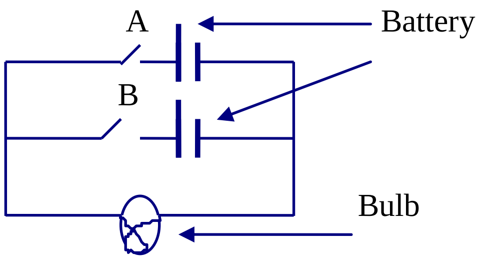
Boolean algebra provides a symbolic representation of logical arguments which are made up of logical statements. A Boolean expression will have Boolean variables and operators.

From the discussion presented in section 9.1 above we may summarise the following in connection with Boolean Algebra:

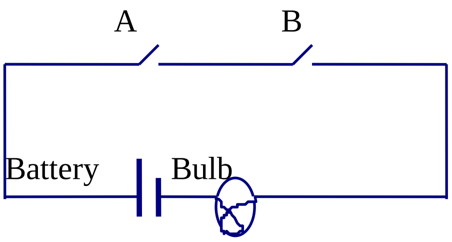
a) A Boolean variable can have one of two values ‘1’ or ‘0’.

b) The only operators permitted in Boolean algebra are ‘OR’ , ‘AND’ and ‘NOT’

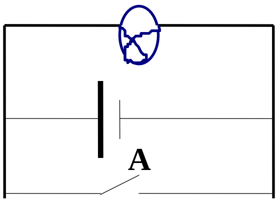
a) OR operation



b) AND Operation



c) NOT Operation



1.3.1 Some Basic Postulates of Boolean Algebra

We may enumerate some basic postulates of Boolean Algebra mainly to highlight its differences with conventional algebra and also to familiarise with symbolic logic.

a) A+ 0 = A

b) A+1 = 1, In fact 1 OR anything will be 1 as , as long as one of the inputs for the
OR operation is true the output will be true.

c) A. 0 = 0 , 0 AND anything will be 0 as all the inputs for the AND operation must
be true for the output to be true.

d) A.1 = A

e) A+ A'= 1, As if A is 0 , A' will be 1 and if A is 1 ,A' will be 0. 1 OR anything is 1
A+A' , is *Tautology* as its value is always true.

f) A. A' = 0, As either a or A' will be 0 as 0 and anything is 0.
A.A' is *Fallacy* as its value is always false.

g) A+A= A ↕ *Idempotent Law* Proof: A+A= (A+A).1=(A+A)(A+A')=A+A.A'=A

h) A. A = A Proof: A.A=A.A+0=A.A+A.A'=A(A+A')= A.1=A

i) A.B=B.A ↕ *Commutative Law* Proof : By making the truth table and showing LHS=RHS

j) A+B=B+A

k) (A+B)+C=A+(B+C) ↕ *Associative Law* Proof: By truth table

l) (A.B).C=A.(B.C)

m) A+ AB = A , ↕ Proof: A+AB= A(1+ B) = A. 1= A

Absorption Law

n) A.(A+B)=A

o) A.(B+C)=AB+AC ↕ *Distributive Law*

p) A+BC=(A+B).(A+C)

Proof: (A+B).(A+C)=AA+AC+AB+BC
=A(1+A+C+B)+BC
=A+BC

q) A’’ = A *Involution*

r) A+ A’B = A+B ↕ A(B+B’)+A’B = AB+AB’+ A’B+AB since AB+AB=AB
= A(B+B’)+B(A+A’) =A+B

1.3.2 De’Morgan’s Theorems

De’Morgan’s Theorems’ are useful in changing the forms of Boolean expressions . The two theorems are:

i) The complement of the sum is equivalent to the product of individual complements.

(A+B)’ = A’ . B’

Proof: The truth table for the expression is as shown in Table 1.4 below:

From table 1.4 it may be observed that for all possible input combinations of A and B the LHS (Left Hand Side) is the same as RHS (Right Hand Side). The table thus shows the proof of the theorem.

ii) The complement of the product is equal to the sum of individual complements.

(A.B)’ = A’ + B’

Proof: The truth table for the above expression is shown in Table 1.5

From Table 1.5 it is observed that the LHS and RHS of the expression have the same value for all possible input combinations of ‘A’ and ‘B’. The theorem is thus proved.

1.3.3 Proof of De’ Morgan’s theorem by induction

I) To prove (A+B)’ = A’.B’

Now as $\underset{\mathbf{X}}{\underleftrightarrow{(A+B)}}.\ \underset{\mathbf{Y}}{\underleftrightarrow{A'.B'}}\mathbf{=0}$ and $\underset{\mathbf{X}}{\underleftrightarrow{(A+B)}} + \underset{\mathbf{Y}}{\underleftrightarrow{A'.B'}}\ \mathbf{=1}$

$$\mathbf{X' = Y}$$

$$\begin{aligned}
(A+B).\ A'.B' &= A'.B'.\ (A+B) && \text{*ANDing both sides by (A+B)}\\
&= A'.\ B'\ .\ A + A'.\ B'.\ B\\
&= A.A'.B + A'.B'.\ B\\
&= 0.\ B + A'.\ 0 && \text{* A . A' = 0}\\
&= 0 + 0\\
&= 0
\end{aligned}$$

$$\begin{aligned}
A + B + A'.B' &= (A+B+A').(A+B+B')\\
&= (1+ B).(A+1)\\
&= 1.1\\
&= 1
\end{aligned}$$

i.e., $(A+B)' = A'.B'$

II) To prove $(A.B)' = A' + B'$

Now as $\underleftrightarrow{A.B}\ \underleftrightarrow{(A' + B')} = 0$ and $\underleftrightarrow{A.B} + \underleftrightarrow{(A'+B')} = 1$

$$\mathbf{X\ .\ Y = 0 \qquad X + Y = 1}$$

$$\mathbf{X' = Y}$$

$$\begin{aligned}
A.B(A'+ B') &= A.B.A' + A.B.B'\\
&= A.A'.B + A.B.B'\\
&= 0.B + A.0\\
&= 0 + 0 = 0
\end{aligned}$$

$$\begin{aligned}
A.B + (A' + B') &= (A' + B' + A).\ (A' + B' + B)\\
&= (1+B').(A' +1)\\
&= 1.1 = 1
\end{aligned}$$

i.e., $(A.B)' = A' + B'$

TABLE 1.4 DE’ MORGAN’S THEOREM

A	B	LHS= (A+B)’	RHS= A’ . B’
0	0	(0+0)’= 0’ = 1	0’. 0’= 1.1 = 1
0	1	(0+1)’= 1’ = 0	0’.1’= 1.0 = 0
1	0	(1+0)’= 1’= 0	1’.0’= 0.1 = 0
1	1	(1+1)’= 1’= 0	1’.1’= 0.0 = 0

TABLE 1.5 DE’MORGAN’S THEOREM

A	B	LHS= (A .B)’	RHS= A’+. B’
0	0	(0.0)’= 0’ = 1	0’+ 0’= 1+1 = 1
0	1	(0.1)’= 0’ = 1	0’+0’= 1+1 = 1
1	0	(1. 0)’= 0’= 1	1’+0’= 0+1 = 1
1	1	(1+1)’= 1’= 0	1’+1’= 0+0 = 0

1.4 TRUTH TABLES

While writing truth tables the following points should be remembered:

a) A truth table with ‘n’ inputs should have 2^n rows.

b) All inputs are entered in the left most column of the truth table.

c) Outputs are entered in the right most column.

d) The order of inputs should be such that the decimal values obtained from the conversion of bits should be in ascending order row wise.

1.5 PRINCIPLE OF DUALITY

According to this principle, *For every valid expression in Boolean Algebra there exists an equally valid dual expression.* The dual expression can be obtained by following the following three steps:

i) Complement each 0 and 1 (Change the 0’s to 1’s and 1’s to 0’s)

ii) Replace each OR (+) sign by AND (.) and each AND (.) sign by OR (+)

iii) Leave NOTs unchanged.

The dual of i) X.1 = X is X + 1= X
ii) X + (YZ) is X . (Y+ Z)
iii) X.(Y+Z) is X+ (Y.Z)

1.6 SIMPLIFICATION OF BOOLEAN (LOGICAL) EXPRESSIONS

We have already seen a number of identities and also the De’ Morgans theorems. These can be used in reducing Boolean expressions. The main purpose of reducing Boolean expressions is to implement the logic with the use of minimum hardware. The minimisation of hardware would involve reducing the the number of inputs to a logic gate and also the number of gates without altering the truth table of the expressions. In other words simplifying the logical argument without beating around the bush.

The following examples demonstrate the utility of the identities and De’ Morgan’s theorems in reducing Boolean expressions:

Reduce the following

i) (xyz+x(yz)’+ xy’z)’

= (xyz)’. (x(yz)’)’ . (xy’z)’ * Removing the outermost complement using
= (x’+y’+z’)(x’+(yz)’’)(x’+y’’+z’) De’Morgans theorems
= (x’+y’+z’)(x’+yz)(x’+y+z’)
= (x’+x’yz+x’y’+0+x’z’+0)(x’+y+z’)
= x’(1+ yz+ y’+z’)(x’+y+z’)
= x’(1)(x’+y+z’) * 1 OR anything will be 1
= x’+x’y+x’z’
= x’(1+y+z’)
= x’

ii) ((x+y+z)’ + (xyz)’)’
= (x+y+z)’’ . (xyz)’’
= (x+y+z).xyz
= xyz+xyz+xyz
= xyz

iii) (x (yz)’ ((xy)’z(xyz)’+xyz’+x(yz)’))’

$$= \overline{x\,\overline{yz}\,(\overline{xy}\,z\,.\,\overline{xyz} + xy\,\overline{z} + x\,\overline{yz})}$$

$= x \;\; yz + xy \;\; z \,.\, xyz \;\; + xy\, z + x \;\; yz$

$= \bar{x} + \overline{\overline{yz}} + \overline{\overline{xy}\, z} + \overline{\overline{xyz}} \,.\, \overline{xy\,\bar{z}} \,.\, \overline{x\,\overline{yz}}$

$= \overline{x} + yz + (xy + \bar{z} + xyz.(\bar{x}+\bar{y} + z).\overline{(x} + yz)$

$= \bar{x} + yz + xy + \bar{z} \;\; = \bar{x} + y + \bar{z}$ *Using A’ + AB= A’+B

1.7 CANONICAL FORMS OF BOOLEAN EXPRESSIONS

Canonical forms are standard forms. Let us consider the Boolean expression in three variables whose truth table is shown in table 1.6

From truth table 1.6 we have the Boolean expression :

A’B’C’ + AB’C + ABC’ +ABC = 1 I

In this expression we see that there are four terms and each term contains all the variables which occur in the expression. The expression is true for any of the terms which occur in the expression. Each one of the product terms which occur in the expression having all the variables of the expression is called a min term and the expression itself is in min terms canonical form.

From the same truth table 1.6 we can also write the following expression :

A’B’C+A’BC’+A’BC+AB’C’ = 0 II

Normally the Boolean expression is written for a true value. If we commplement both sides of expresion II we have:

(A’B’C+A’BC’+A’BC+AB’C’)’ = 0’

or (A+B+C’)(A+B’+C)(A+B’+C’)(A’+B+C) = 1 *By using De’Morgan’s theorem III

Scrutinising expression III we see that each sum term in the product of sums has all the three variables of the expression. Each one of these terms is called a max term and the expression III is an expression in max terms canonical form or Product Of Sums (POS) canonical form .

1.8 THE THREE VARIABLE KARNAUGH MAP

From the truth table we can make a Karnaugh Map, these maps are very useful in reducing Boolean expressions as we shall see shortly. While making a Karnaugh map it must be ensured that in adjacent cells of the map only one variable should change. The Karnaugh map for the truth table 1.6 is shown in FIG. 1.3

FIG 1.3 KARNAUGH MAP FOR TRUTH TABLE 9.6

The expression is A’B’C’ + AB’C + ABC’ +ABC = 1

	C’	C
A’B’	1 (0)	0 (1)
A’B	0 (2)	0 (3)
AB	1 (6)	1 (7)
AB’	0 (4)	1 (5)

In FIG 1.3 the truth values in each cell are reproduced from the truth table 1.6 and within parenthesis the cell numbers are marked for ready reference. We may also observe that only one variable varies in adjacent cells. The Karnaugh maps may be folded from left to right and also from top to bottom, thus cells (0) and (1) are adjacent and cells (0) and (4) are also adjacent. Cells (1) and (5) are adjacent but cells (0) and (5) are not adjacent. Let us take cell (0) in this the variables are A’B’C’ and in cell (5) the variable are AB’C thus A and C change in these two cells so they are not adjacent.

Let us now consider the two adjacent cells (6) and (7) . Considering these two cells alone we get the expression :

ABC’+ABC = 1 i.e., AB(C+C’) = 1 , i.e., AB = 1 . The variable C which changes in these two cells gets eliminated. We can thus presume that if we have a pair of 1’s in adjacent cells we can eliminate one variable. FIG. 1.4 shows the combinations which can be made to reduce the Boolean expression I.

We have an isolated 1 in the cell A’B’C’ and two pairs in cells ABC’, ABC and ABC, AB’C as shown enclosed in boxes.

The reduced expression thus becomes A’B’C’+AB+AC=1..

From the Karnaugh map 1.4 we also get expression II, i.e., **A’B’C+A’BC’+A’BC+AB’C’ = 0 .** We may consider the adjacent cell 0’s. We see that we have 0’s in adjacent cells (1),(3) and (2)(3) and there is an isolated 0 in cell 4. Eliminating the changing variable in adjacent cells we have **A’C+A’B+AB’C’=0.** Complementing both sides of this expression we get the expression **(A+C’)(A+B’)(A’+B+C) = 1.** Thus from the same Karnaugh map we can get two reduced expressions which are true, one in a **Sum of Products (SOP) form and another in a Product of Sums(POS) form.**

The expression

A’B’C’ + AB’C + ABC’ +ABC = 1 as f(A,B,C) = $\sum$ **(0, 5, 6,7)** i.e. Sum of Products of cell numbers 0, 5, 6 and 7

and the expression

(A+B+C’)(A+B’+C)(A+B’+C’)(A’+B+C) = 1 as f(A, B, C) = $\prod$ **(1, 2, 3, 4)** i.e., the Product of Sums of cells 1, 2, 3 and 4..

1.8 THE FOUR VARIABLE KARNAUGH MAP

The Karnaugh maps in theory can be drawn for Boolean expressions with any number of variables. We shall however restrict or discussions up-to four variables. In section 1.7 we had restricted ourselves only to forming pairs with adjacent cells. We shall soon see that we can combine adjacent cells making groups of 2^n where n is an integer. Thus we can combine adjacent cells 2^2 = 4 cells or **quads** 2^3 = 8 cells called **Octets** etc. A quad will eliminate two variables, an Octet will eliminate three variables as we shall soon see.

Let us consider the Boolean expression:

The expression in sum of products canonical form is:

A’B’C’D’+A’B’C’D+A’B’CD+A'BC'D'|+A’BCD+AB’C’D’+AB'C'D+AB’CD+ ABC’D’ = 1 I

Table 1.7 A shows the truth table for the expression alongwith the cell numbers and min terms represented by each cell. FIG.1.5 shows the Karnaugh map for the expression with cell numbers given in parenthesis. FIG. 1.5A shows that two quads

‘a’ and ‘b’ can be formed by combining adjacent cells with a truth value of ‘1’ and a pair ‘c’ can also be formed with truth value ‘1’ in adjacent cells. Let us now consider the two quads ‘a’ and ‘b’ and the pair ‘c’ individually.

From quad ‘a’ we get:

A’B’C’D’+A’BC’D’+ABC’D’+AB’C’D’ = 1

OR A’C’D’(B’+B)+ AC’D’(B+B’)= 1
OR A’C’D’+AC’D’= 1
OR C’D’(A’+A)= 1
OR C’D’ = 1 i

Note that in this quad the variables A and B change and hence can be eliminated.

The truth table for the expression is shown in table 1.7

TABLE 1.7 TRUTH TABLE FOR F (A, B, C, D) = $\sum$ (0,1, 3, 4, 7, 8, 9, 11, 12)

A	B	C	D	F	CELL NO.	MIN TERM
0	0	0	0	1	0	A’B’C’D’
0	0	0	1	1	1	A’B’C’D
0	0	1	0	0	2	A’B’CD’
0	0	1	1	1	3	A’B’CD
0	1	0	0	1	4	A’BC’D’
0	1	0	1	0	5	A’BC’D
0	1	1	0	0	6	A’BCD’
0	1	1	1	1	7	A’BCD
1	0	0	0	1	8	AB’C’D’
1	0	0	1	1	9	AB’C’D
1	0	1	0	0	10	AB’CD’
1	0	1	1	1	11	AB’CD
1	1	0	0	1	12	ABC’D’
1	1	0	1	0	13	ABC’D
1	1	1	0	0	14	ABCD’
1	1	1	1	0	15	ABCD

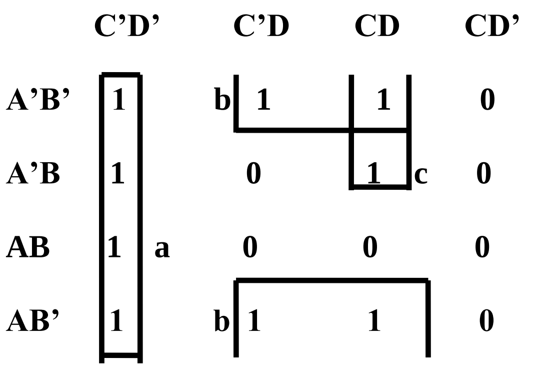
FIG 1.5 KARNAUGH MAP

	C’D’	C’D	CD	CD’
A’B’	**1 (0)**	**1(1)**	**1 (3)**	**0(2)**
A’B	**1(4)**	**0(5)**	**1(7)**	**0(6)**
AB	**1(12)**	**0(13)**	**0(15)**	**0(14)**
AB’	**1(8)**	**1(9)**	**1(11)**	**0(10)**

*****The cell numbers are marked in parenthesis

FIG. 1.5 A

Reproducing the Karnaugh map without cell nos.
and combining adjacent cells



	C’D’	C’D	CD	CD’
A’B’	**1**	**1**	**1**	**0**
A’B	**1**	**0**	**1**	**0**
AB	**1**	**0**	**0**	**0**
AB’	**1**	**1**	**1**	**0**

From quad ‘b’ (Obtained by folding or wrapping around the Karnaugh map) we get:

A’B’C’ D + A’B’CD + AB’ C’D+ AB’ CD = 1
OR A’B’D(C’+C) + AB’D(C’+C) = 1
OR A’B’D + AB’D = 1
OR B’D(A’+A)= 1
OR B’D = 1 ii

Note that in this quad the variables A and C change and can be eliminated

From the pair ‘c’ we get:

A’B’CD + A’BCD = 1
OR A’CD(B’+B)= 1
OR A’CD = 1 iii

Note that in this pair only the variable B changes and can be eliminated.

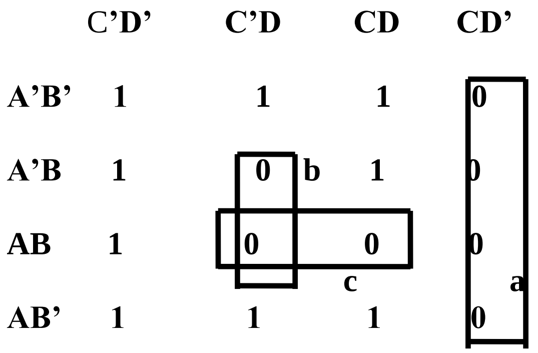
From i, ii and iii above we get:

C’D’ + B’D + A’CD = 1 ....(X)

The original expression (A’B’C’D’+A’B’C’D+A’B’CD+A'BC'D'+A’BCD+AB’C’D’+AB'C'D+AB’CD+ABC’D’ = 1) reduces to the expression (C’D’ + B’D + A’CD = 1) with the help of the Karnaugh map.

The Karnaugh map of FIG. 1.5A is reproduced in FIG. 1.5B. We shall now reduce the expression to obtain a simplified expression in product of sums form.

FIG.1. 5B SIMPLIFICATION FOR POS FORM



	C’D’	C’D	CD	CD’
A’B’	**1**	**1**	**1**	**0**
A’B	**1**	**0**	**1**	**0**
AB	**1**	**0**	**0**	**0**
AB’	**1**	**1**	**1**	**0**

Combining the 0’s we get the following simplified expression:

CD’ (FROM QUAD ‘A’) + BC’D (FROM PAIR ‘b’) + ABD (from pair ‘c’) = 0

complementing both sides we have:

$$\overline{CD' + BC'D + ABD} = \overline{0}$$

OR (CD’)’. (BC’D)’. (ABD)’ = 1

OR (C’ + D). (B’+C+D’). (A’+B’+D’) = 1 ....(Y)

Note : You may verify that if all the cells in a Karnaugh map contain a 1 then the sum of all min terms will be 1 and the Boolean expression will be true for all possible input combinations. Similarly if all cells are 0 then the output is false for possible input combinations.

LOGIC GATES

We have seen the basic logic gates, namely, OR AND and NOT in section 1.2 in chapter I

With the help of the basic logic gates shown in the FIG. 1.1 some combinational gates as shown in FIG. 2.1 are built which are very commonly used. These gates and their truth tables are shown in FIG. 2.1. The NOT gate following the OR gate results in a NOR gate. The NOT gate following the AND gate results in a NAND gate. The combination of basic gates required to form the EXCLUSIVE OR (EX-OR) gate is also shown in FIG. 2.1. The EX-OR gate is very useful and is also used for constructing the adders as will be shown in the next section. The EX-OR gate gives a true output if the inputs are not similar. For similar inputs the EX-OR gate gives a false output. In the truth tables shown in FIG. 2.1 '1' represents true and '0' represents false . The EXCLOSIVE NOR (EX-NOR) gate gives a true output for similar inputs. The NAND gates and NOR gates are called universal building blocks as any logic can be implemented by using either NAND gates alone or NOR gates alone, as will be shown later in this chapter.

2.2 USING LOGIC GATES TO ADD BITS

Adding two bits

We can use a combination of logic gates to add two bits. Let us say that the two bits to be added are 'A' and 'B'. Each of these bits may have a value of '0' or '1' . Taking all possible combinations of 'A' and 'B' their sum in binary will be as shown in TABLE 2.1 below: The logic diagram for the implementation of the half adder is shown in FIG. 2.2 (a)

Adding three bits

The logic diagram for adding three bits (full adder) is shown in FIG 2.2 (b) and the truth table for this logic diagram is shown in TABLE 2.2. We may observe from the logic diagram of the full adder shown in FIG. 2.2 (b) that if two half adders are cascaded and the carry outputs of the two half adders input to an OR gate for generating the carry we can add one more bit. Similarly if a number of half adders are cascaded we can add more and more bits. In an Integrated Circuit (IC) a number of half adders could be cascaded to enable addition of a number of bits.

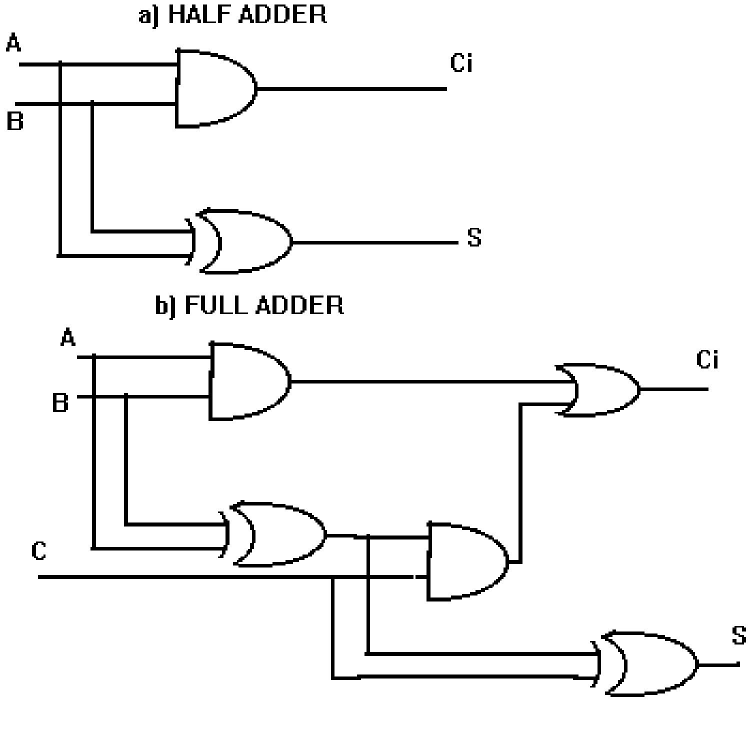
You may verify from the truth table of the full adder that the Boolean expressions for the carry 'Ci' and the sum 'S' are as follows:

Ci = A’BC+AB’C+ABC’+ABC

S = A’B’C+A’BC’+AB’C’+ABC

We can use logic circuits for adding the complement of negative numbers to perform subtraction. Since multiplication is repeated addition and division is repeated subtraction, all arithmetic operations can be performed by using logic gates.

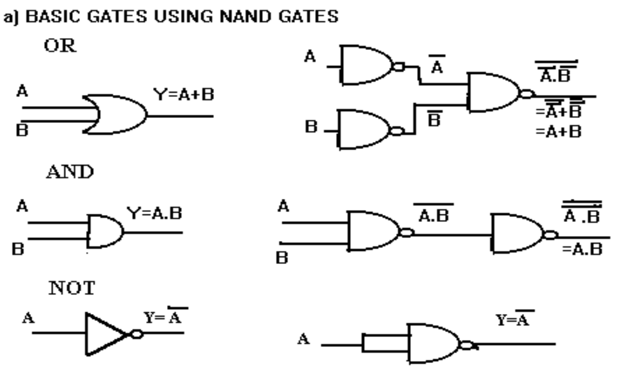
FIG.9.4 LOGIC DIAGRAMS FOR ADDERS



a) HALF ADDER

b) FULL ADDER

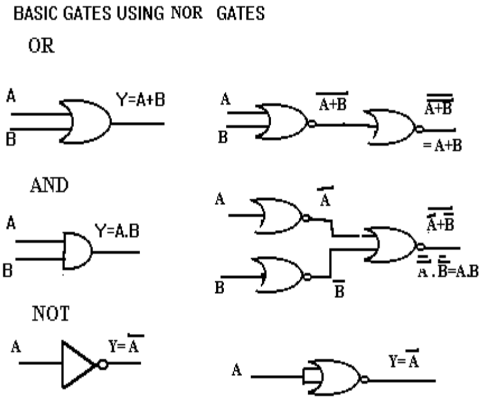
FIG. 9.5 UNIVERSAL BUILDING BLOCKS



a) BASIC GATES USING NAND GATES

OR
A
B
Y=A+B
A
B
$\overline{A}$
$\overline{B}$
$\overline{\overline{A}.\overline{B}}$
$=\overline{\overline{A}}+\overline{\overline{B}}$
=A+B
AND
A
B
Y=A.B
A
B
$\overline{A.B}$
$\overline{\overline{A.B}}$
=A.B
NOT
A
$Y=\overline{A}$
A
$Y=\overline{A}$

FIG. 9.6 UNIVERSAL BUILDING BLOCKS


BASIC GATES USING NOR GATES
OR
A
B
Y=A+B
A
B
$\overline{A+B}$
$\overline{\overline{A+B}}$
= A+B
AND
A
B
Y=A.B
A
$\overline{A}$
$\overline{\overline{A}+\overline{B}}$
$\overline{\overline{A}}.\overline{\overline{B}}$=A.B
B
$\overline{B}$
NOT
A
Y=$\overline{A}$
A
Y=$\overline{A}$

1.3 UNIVERSAL BUILDING BLOCKS

In section 2.1 above it was mentioned that with the help of ‘NAND’ gates alone or ‘NOR’ gates alone any logic can be implemented. The basic gates are ‘OR’, ‘AND’ and ‘NOT’. It therefore follows that if these three basic gates can be implemented , any logic can be implemented. In FIG. 2.3A implementation of the basic gates by ‘NAND’ gates is shown. and in FIG. 2.3B implementation of the basic gates by ‘NOR’ gates is shown.

The NAND gates alone could be used to implement any logic as it is possible to implement the basic gates (OR, AND and NOT) using NAND gates. Similarly NOR gates alone could be used to implement any logic. Production of only one type of gates would be more convenient than production of an assortment of gates. The NAND gates are universal building blocks.

1.4 USING LOGIC GATES TO IMPLEMENT LOGICAL ARGUMENTS

We have seen in section 2.2 how logic gates could be used to perform arithmetic functions. The logic gates could also be used to implement logical arguments. Let us say I have two friends, A and B. B plays badminton, A plays tennis and all of us play cricket. I have a holiday and I invite both my friends. Four situations could arise as follows:

i) If non of my friends come I shall read a book.

ii) If only B accepts my invitation I shall play badminton with him.

iii) If only A accepts the invitation I shall play tennis with him.

iv) If both come we shall all play cricket.

FIG. 2.4 shows how this argument can be implemented using Logic gates.

In FIG. 2.4 there are two inputs ‘A’ and ‘B’. These inputs are fed two AND gates either in complemented or uncomplemented form . If the inputs are uncomplemented they are represented by ‘1’ (True) and if they are in uncomplemented form they are represented by ‘0’(false). One of the four outputs of the AND gates will be true depending whether the inputs are true or false. If ‘A’ and ‘B’ are both absent, then ‘A’ and ‘B’ will both be ‘0’ and their complements ‘1’, so the top most AND gate will give a True output and all other AND gates will give a False output. Under these conditions I shall read a book. Proceeding similarly for other values of ‘A’ and ‘B’ as shown in the truth table one of the output paths will be selected.

Two inputs enable us to select from 4 output paths. Three inputs will enable selection from among 8 output paths. for selection from among ‘2n’ output paths we will require n inputs.

UNIT II

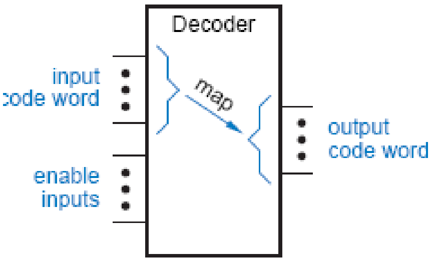
MSI DEVICES

Combinational Logic Design

Decoders

A *decoder* is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code, and there is a one-to one mapping from input code words into output code words. In a *one-to-one mapping*, each input code word produces a different output code word. The general structure of a decoder circuit is shown in Figure. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single, “disabled,” output code word. The most commonly used input code is an *n*-bit binary code, where an *n*-bit word represents one of 2*n* different coded values, normally the integers from 0 through 2*n*□1. Sometimes an *n*-bit binary code is truncated to represent fewer than 2*n* values. For example, in the BCD code, the 4-bit combinations 0000 through 1001 represent the decimal digits 0–9, and combinations 1010 through 1111 are not used. The most commonly used output code is a 1-out-of-*m* code, which contains *m* bits, where one bit is asserted at any time. Thus, in a 1-out-of-4 code with active-high outputs, the code words are 0001, 0010, 0100, and 1000. With active-low outputs, the code words are 1110, 1101, 1011, and 0111.

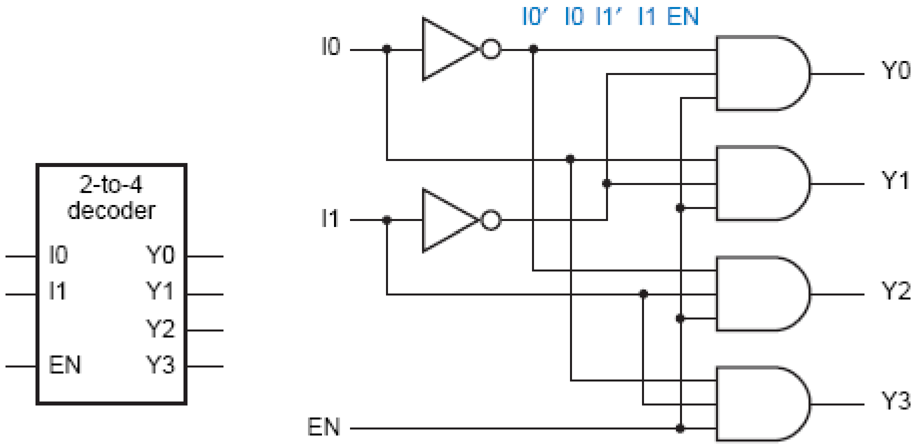
DECODER



Binary Decoders

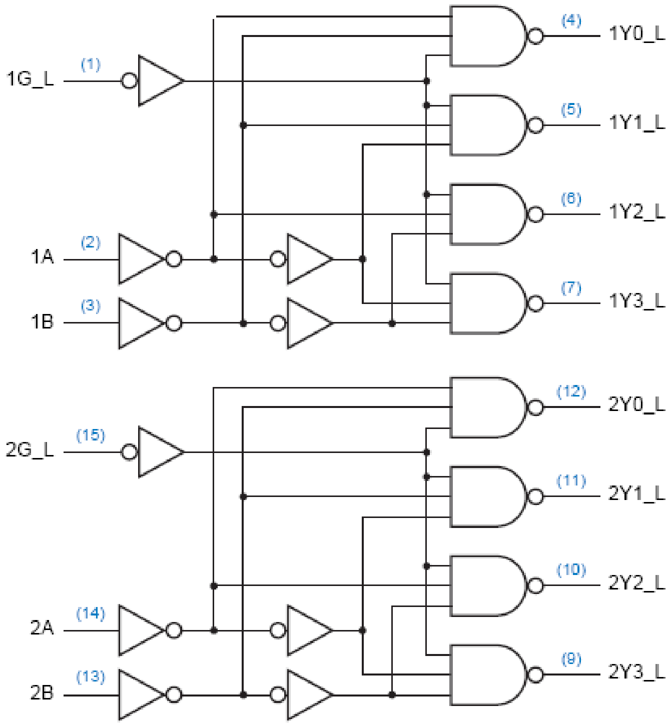
The most common decoder circuit is an *n*-to-2*n* decoder or *binary decoder*. Such a decoder has an *n*-bit binary input code and a 1-out-of-2*n* output code. A binary decoder is used when you need to activate exactly one of 2*n* outputs based on an *n*-bit input value.For example, Y*i* equal to 1 if and onlyif the input code word is the binary representation of *i* and the *enable input* EN is 1. If EN is 0, then all of the outputs are 0. A gate-level circuit for the 2-to-4 decoder is shown in Figure. Each AND gate *decodes* one combination of the input code word I1,I0. The binary decoder's truth table introduces a "don't-care" notation for input combinations. If one or more input values do not affect the output values for some combination of the remaining inputs, they are marked with an "x" for that input combination. This convention can greatly reduce the number of rows in the truth table, as well as make the functions of the inputs more clear.

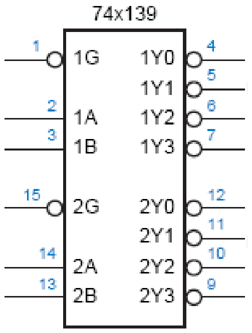
2-4 DECODER

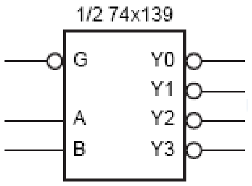


The 74x139 Dual 2-to-4 Decoder

Two independent and identical 2-to-4 decoders are contained in a single MSI part, the *74x139*. The gate-level circuit diagram for this IC is shown in Figure. Notice that the outputs and the enable input of the '139 are active-low. Most MSI decoders were originally designed with active-low outputs, since TTL inverting gates are generally faster than non inverting ones. Also notice that the '139 has extra inverters on its select inputs. Without these inverters, each select input would present three AC or DC loads instead of one.





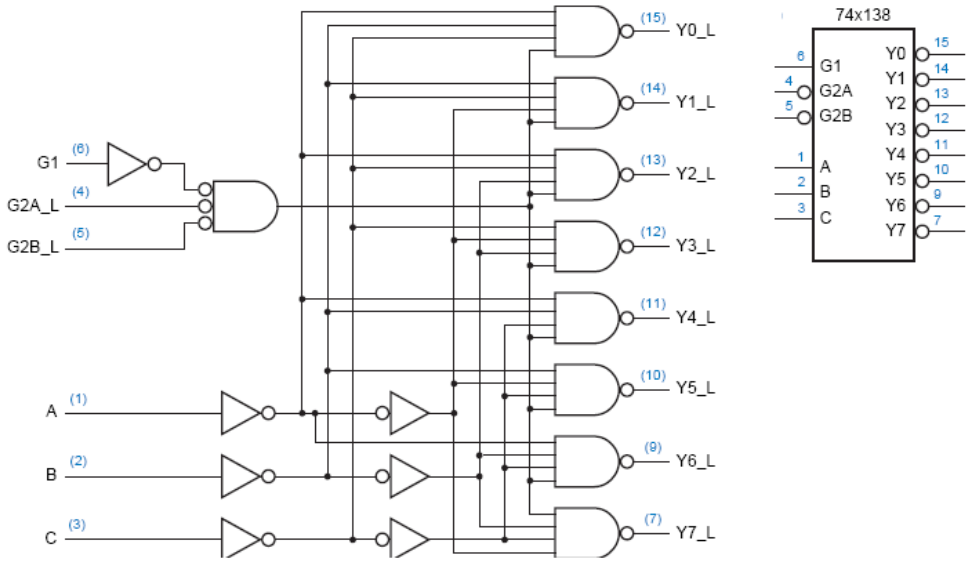


A logic symbol for the 74x139 is shown in Figure. Notice that all of the signal names inside the symbol outline are active-high (no "_L"), and that inversion bubbles indicate active-low inputs and outputs. Often a schematic may use a generic symbol for just one decoder, one-half of a '139, as shown in (c). In this case, the assignment of the generic function to one half or the other of a particular '139 package can be deferred until the schematic is completed.

The 74x138 3-to-8 Decoder

Like the 74x139, the 74x138 has active-low outputs, and it has three enable inputs (G1, /G2A, /G2B), all of which must be asserted for the selected output to be asserted. The logic function of the '138 is straightforward—an output is asserted if and only if the decoder is enabled and the output is selected. Thus, we can easily write logic equations for an internal output signal such as Y5 in terms of the internal input signals: However, because of the inversion bubbles, we have the following relations between internal and external signals: Therefore, if we're interested, we can write the following equation for the external output signal Y5_L in terms of external input signals:

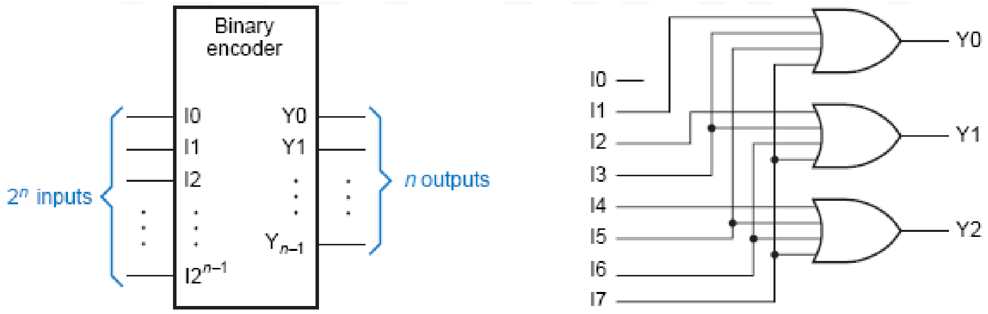
On the surface, this equation doesn't resemble what you might expect for a decoder, since it is a logical sum rather than a product. However, if you practice bubble-to-bubble logic design, you don’t have to worry about this; you just give the output signal an active-low name and remember that it’s active low when you connect it to other inputs.



Encoders

A decoder’s output code normally has more bits than its input code. If the device’s output code has *fewer* bits than the input code, the device is usually called an *encoder*. For example, consider a device with eight input bits representing an unsigned binary number, and two output bits indicating whether the number is prime or divisible by 7. We might call such a device a lucky/prime encoder. Probably the simplest encoder to build is a 2*n*-to-*n* or *binary encoder*. As shown in Figure, it has just the opposite function as a binary *de*coder— its input code is the 1-out-of-2*n* code and its output code is *n*-bit binary.

Binary Encoder



Priority Encoders

The 1-out-of-2*n* coded outputs of an *n*-bit binary decoder are generally used to control a set of 2*n* devices, where at most one device is supposed to be active at any time.

Conversely, consider a system with 2*n inputs*, each of which indicates a request for service, as in Figure. This structure is often found in microprocessor input/output subsystems, where the inputs might be interrupt requests. In this situation, it may seem natural to use a binary encoder of the type shown in Figure to observe the inputs and indicate which one is requesting service at any time. However, this encoder works properly only if the inputs are guaranteed to be asserted at most one at a time. If multiple requests can be made simultaneously, the encoder gives undesirable results. For example, suppose that inputs I2 and I4 of the 8-to-3 encoder are both 1; then the output is 110, the binary encoding of 6. Either 2 or 4, not 6, would be a useful output in the preceding example, but how can the encoding device decide which? The solution is to assign *priority* to the input lines, so that when multiple requests are asserted, the encoding device produces the number of the highest-priority requestor. Such a device is called a *priority encoder*.

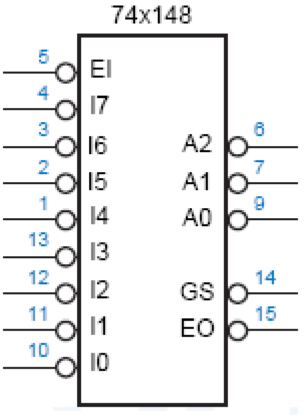
The 74x148 Priority Encoder

The *74x148* is a commercially available, MSI 8-input priority encoder. Its logic symbol is shown in Figure and its schematic is shown in Figure. The main difference between this IC and the “generic” priority encoder of Figure 5-47 is that its inputs and outputs are active low. Also, it has an enable input, EI_L, that must be asserted for any of its outputs to be asserted. The complete truth table is given in Table 5-22. Instead of an IDLE output, the ’148 has a GS_L output that is asserted when the device is enabled and one or more of the request inputs is asserted. The manufacturer calls this “Group Select,” but it’s easier to remember as “Got Something.” The EO_L signal is an enable *output* designed to be connected to the EI_L input of another ’148 that handles lower-priority requests. /EO is asserted if EI_L is asserted but no request input is asserted; thus, a lower-priority ’148 may be enabled. Figure 5-50 shows how four 74x148s can be connected in this way to accept 32 request inputs and produce a 5-bit output, RA4–RA0, indicating the highest-priority requestor. Since the A2–A0 outputs of at most one ’148 will be enabled at any time, the outputs of the individual ’148s can be ORed to produce RA2–RA0. Likewise, the individual GS_L outputs can be combined in a 4-to-2 encoder to produce RA4 and RA3. The RGS output is asserted if any GS output is asserted.

Truth table for a 74x148 8-input priority encoder.

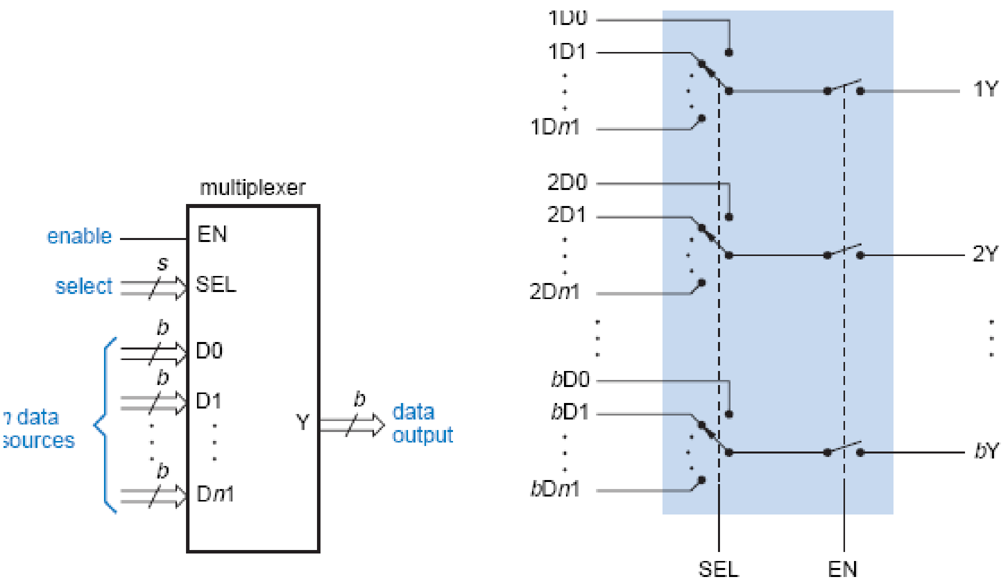
Inputs									**Outputs**				
/EI	/I0	/I1	/I2	/I3	/I4	/I5	/I6	/I7	/A2	/A1	/A0	/GS	/EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1

0 x x x x x x x 0 0 0 0 0 1
0 x x x x x x 0 1 0 0 1 0 1
0 x x x x x 0 1 1 0 1 0 0 1
0 x x x x 0 1 1 1 0 1 1 0 1
0 x x x 0 1 1 1 1 1 0 0 0 1
0 x x 0 1 1 1 1 1 1 0 1 0 1
0 x 0 1 1 1 1 1 1 1 1 0 0 1
0 0 1 1 1 1 1 1 1 1 1 1 0 1
0 1 1 1 1 1 1 1 1 1 1 1 1 0



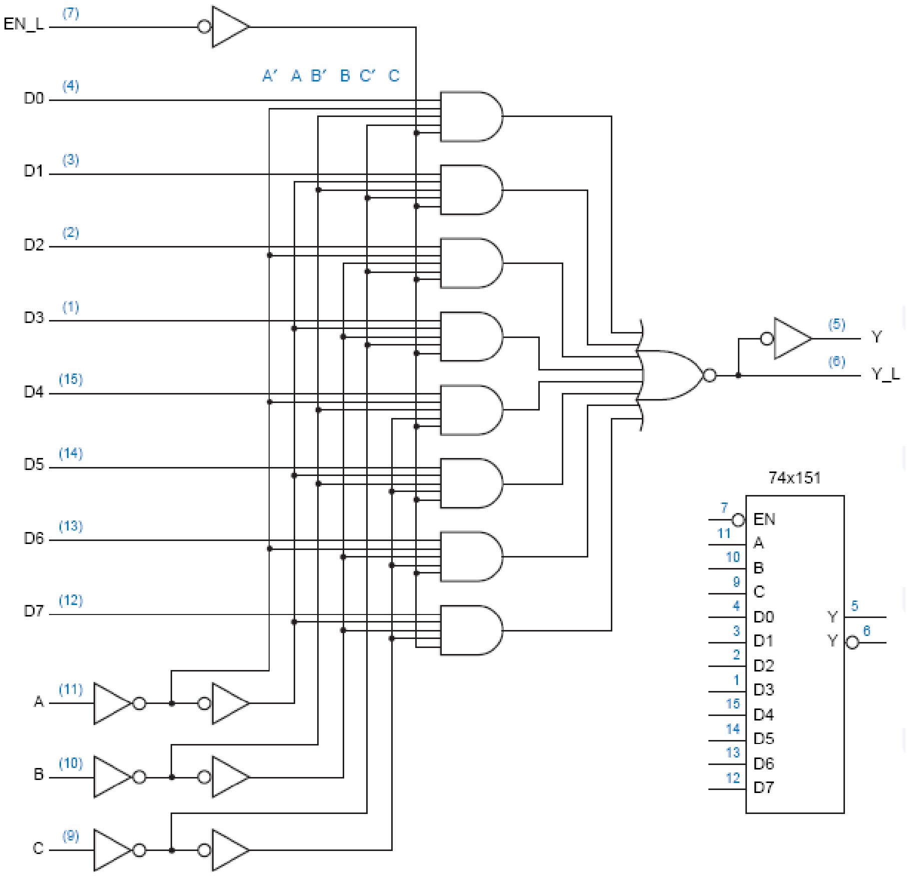
Multiplexers

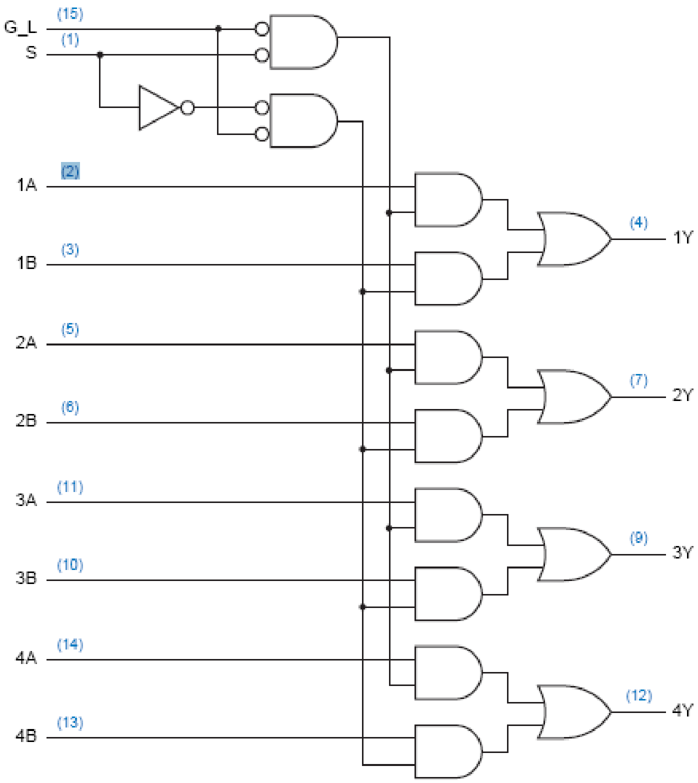
A *multiplexer* is a digital switch—it connects data from one of *n* sources to its output. Figure shows the inputs and outputs of an *n*-input, *b*-bit multiplexer. There are *n* sources of data, each of which is *b* bits wide, and there are *b* output bits. In typical commercially available multiplexers, *n* *□1, 2, 4, 8, or 16, and *b* *□1, 2, or 4.

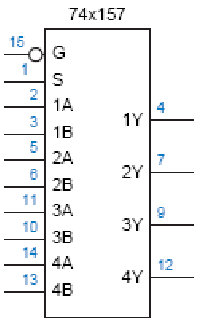


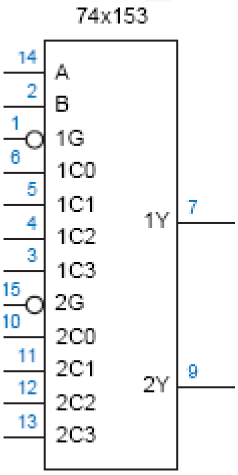
Standard MSI Multiplexers

The sizes of commercially available MSI multiplexers are limited by the number of pins available in an inexpensive IC package. Commonly used muxes come in 16-pin packages. At one extreme is the *74x151*, shown in Figure, which selects among eight 1-bit inputs. The select inputs are named C, B, and A, where C is most significant numerically. The enable input EN_L is active low; both active-high (Y) and active-low (Y_L) versions of the output are provided.





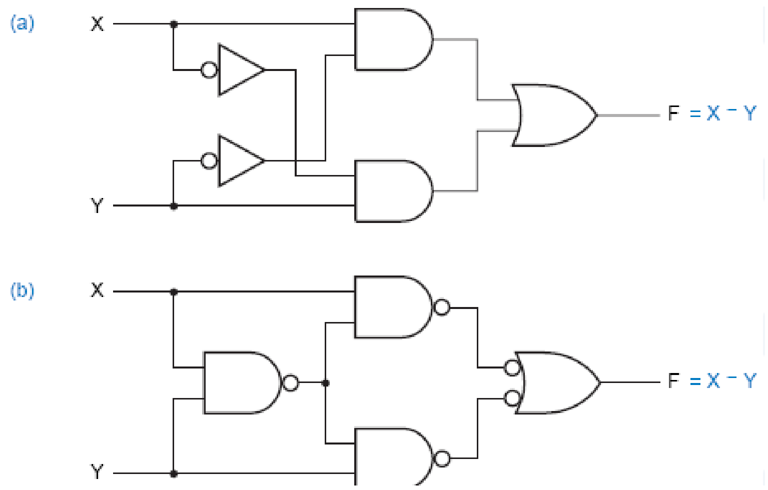


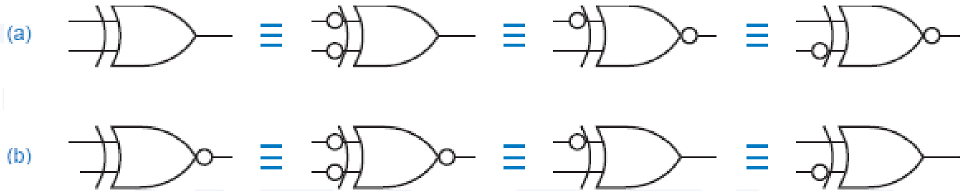


EXCLUSIVE OR Gates and Parity Circuits

EXCLUSIVE OR and EXCLUSIVE NOR Gates

An Exclusive OR (XOR) gate is a 2-input gate whose output is 1 if exactly one of its inputs is 1. Stated another way, an XOR gate produces a 1 output if its inputs are different. An Exclusive NOR (XNOR) or Equivalence gate is just the opposite—it produces a 1 output if its inputs are the same.





The logic symbols for XOR and XNOR functions are shown in Figure. There are four equivalent symbols for each function. All of these alternatives are a consequence of a simple rule: Any two signals (inputs or output) of an XOR or XNOR gate may be complemented without changing the resulting logic function. In bubble-to-bubble logic design, we choose the symbol that is most expressive of the logic function being performed.

Parity Circuits

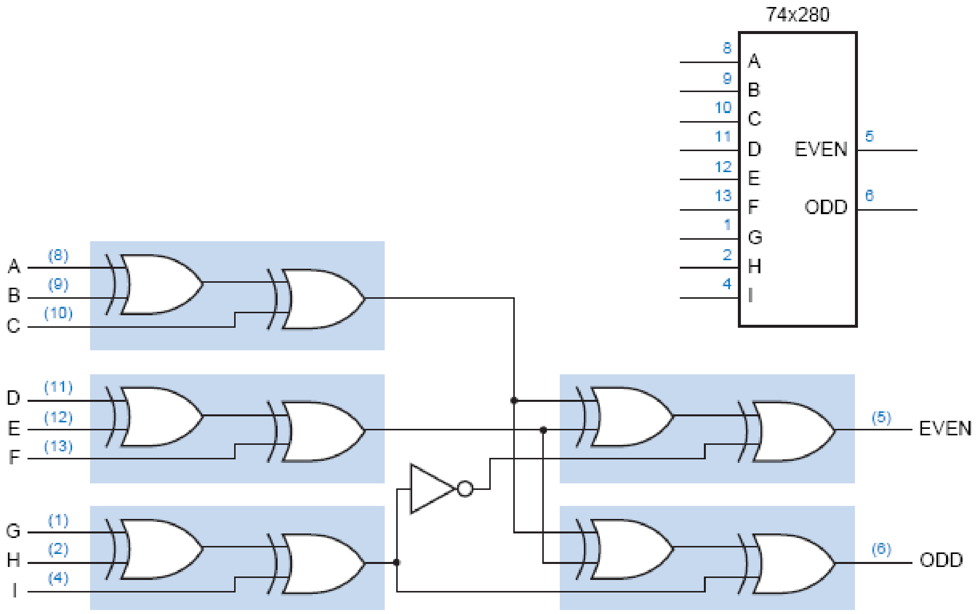
As shown in Figure *n* XOR gates may be cascaded to form a circuit with *n* □□1 inputs and a single output. This is called an *odd-parity circuit*, because its output is 1 if an odd number of its inputs are 1. The circuit in (b) is also an odd parity circuit, but it’s faster because its gates are arranged in a tree-like structure. If the output of either circuit is inverted, we get an *even-parity circuit*, whose output is 1 if an even number of its inputs are 1.

The 74x280 9-Bit Parity Generator

Rather than build a multibit parity circuit with discrete XOR gates, it is more economical to put all of the XORs in a single MSI package with just the primary inputs and outputs available at the external pins. The *74x280* 9-bit parity generator, shown in Figure, is such a device. It has nine inputs and two outputs that indicate whether an even or odd number of inputs are 1.

Parity-Checking Applications

Error-detecting codes use an extra bit, called a parity bit, to detect errors in the transmission and storage of data. In an even parity code, the parity bit is chosen so that the total number of 1 bits in a code word is even. Parity circuits like the 74x280 are used both to generate the correct value of the parity bit when a code word is stored or transmitted, and to check the parity bit when a code word is retrieved or received.



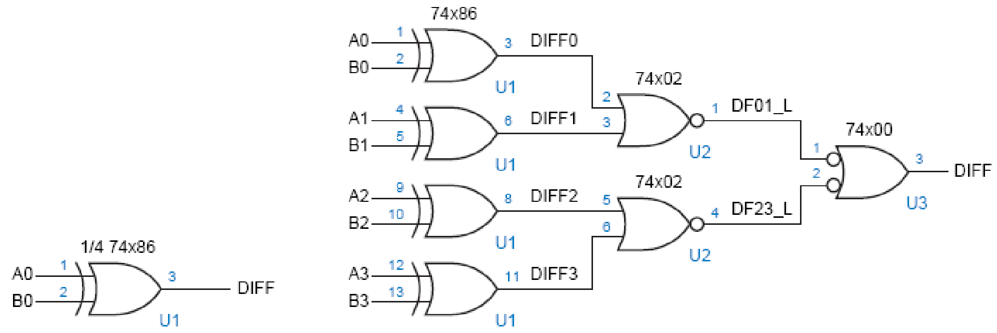
Comparators

Comparing two binary words for equality is a commonly used operation in computer systems and device interfaces. A circuit that compares two binary words and indicates whether they are equal is called a *comparator*. Some comparators interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between the words. These devices are often called *magnitude comparators*.

Comparator Structure

EXCLUSIVE OR and EXCLUSIVE NOR gates may be viewed as 1-bit comparators. Figure shows an interpretation of the 74x86 XOR gate as a 1-bit comparator. The active-high output, DIFF, is asserted if the inputs are different. The outputs of four

XOR gates are ORed to create a 4-bit comparator in (b). The DIFF output is asserted if any of the input-bit pairs are different. Given enough XOR gates and wide enough OR gates, comparators with any number of input bits can be built.

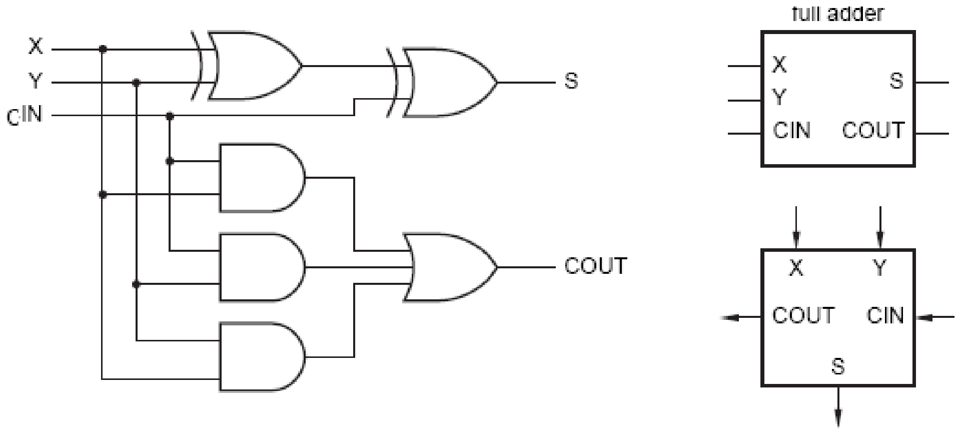


Adders, Subtractors, and ALUs

Addition is the most commonly performed arithmetic operation in digital systems. An *adder* combines two arithmetic operands using the addition rules. The addition rules and therefore the same adders are used for both unsigned and two's-complement numbers. An adder can perform subtraction as the addition of the minuend and the complemented (negated) subtrahend, but you can also build *subtractor* circuits that perform subtraction directly.

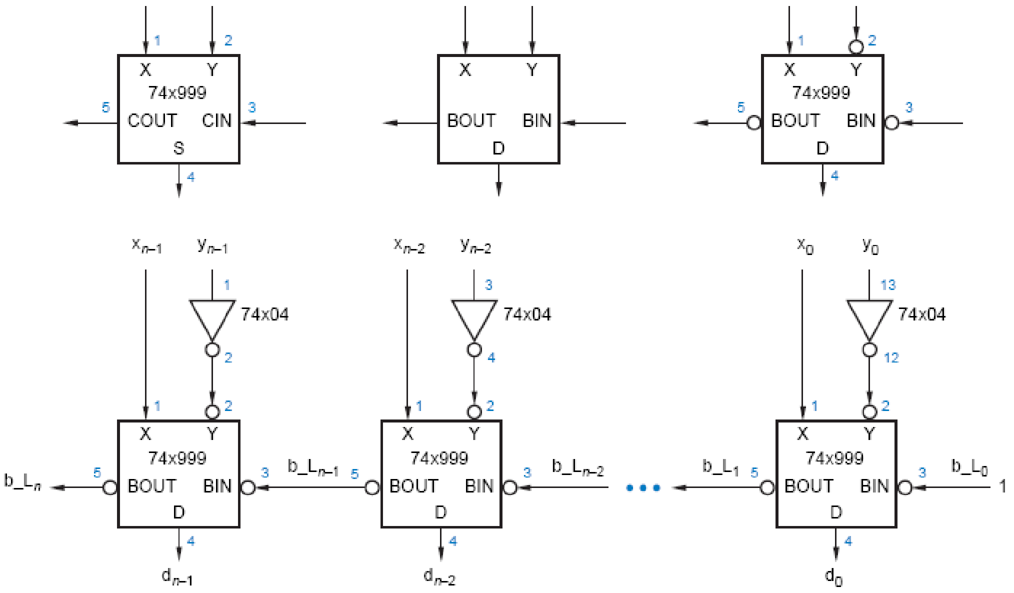
Half Adders and Full Adders

The simplest adder, called a *half adder*, adds two 1-bit operands X and Y, producing a 2-bit sum. The sum can range from 0 to 2, which requires two bits to express. The low-order bit of the sum may be named HS (half sum), and the high-order bit may be named CO (carry out). To add operands with more than one bit, we must provide for carries between bit positions. The building block for this operation is called a *full adder*. Besides the addend-bit inputs X and Y, a full adder has a carry-bit input, CIN.



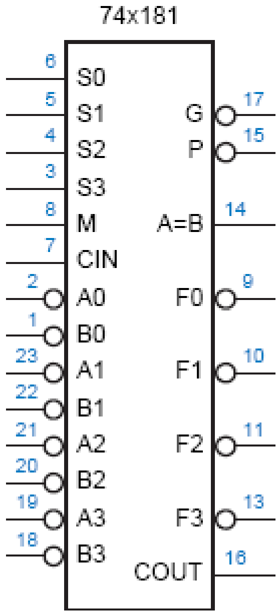
Subtractors

A binary subtraction operation analogous to binary addition.. A *full subtractor* handles one bit of the binary subtraction algorithm, having input bits X (minuend), Y (subtrahend), and BIN (borrow in), and output bits D (difference) and BOUT (borrow out).



MSI Arithmetic and Logic Units

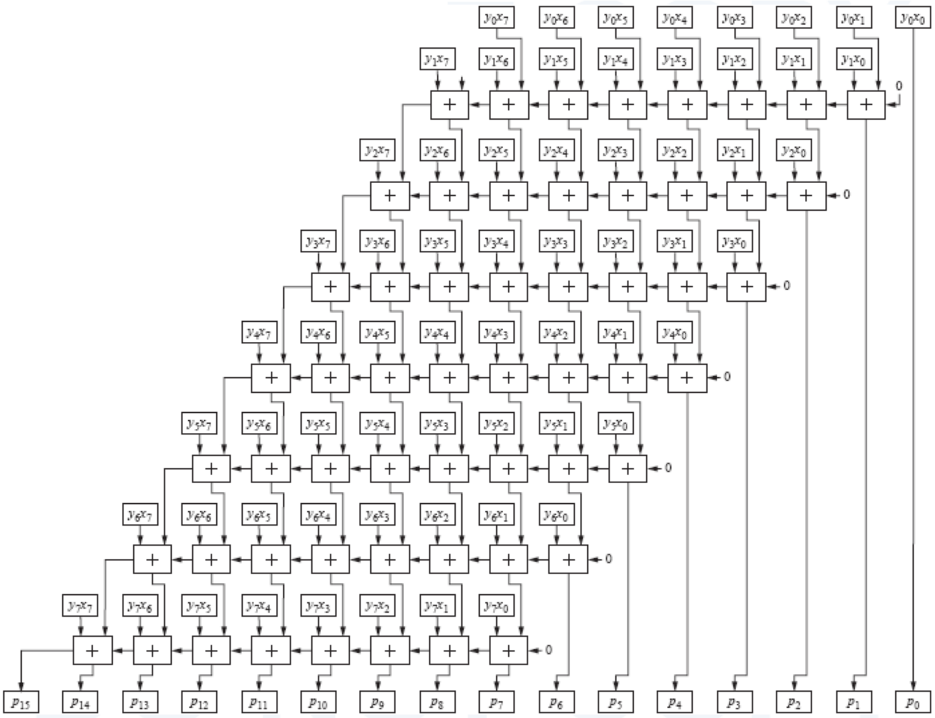
An *arithmetic and logic unit (ALU)* is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of *b*-bit operands. The operation to be performed is specified by a set of function-select inputs. Typical MSI ALUs have 4-bit operands and three to five function select inputs, allowing up to 32 different functions to be performed. Figure is a logic symbol for the *74x181* 4-bit ALU. Note that the identifiers A, B, and F in the table refer to the 4-bit words A3–A0, B3–B0, and F3–F0; and the symbols . □and □□refer to logical AND and OR operations. The 181's M input selects between arithmetic and logical operations. When M =□1, logical operations are selected, and each output Fi is a function only of the corresponding data inputs, Ai and Bi. No carries propagate between stages, and the CIN input is ignored. The S3–S0 inputs select a particular logical operation; any of the 16 different combinational logic functions on two variables may be selected.



Combinational Multipliers

Combinational Multiplier Structures

Although the shift-and-add algorithm emulates the way that we do paper-and-pencil multiplication of decimal numbers, there is nothing inherently “sequential” or “time dependent” about multiplication. That is, given two *n*-bit input words *X* and *Y*, it is possible to write a truth table that expresses the 2*n*-bit product *P*= □*XY* as a *combinational* function of *X* and *Y*. A *combinational multiplier* is a logic circuit with such a truth table. Most approaches to combinational multiplication are based on the paperand- pencil shift-and-add algorithm. Figure illustrates the basic idea for an 8*8 multiplier for two unsigned integers, multiplicand *X* = *x*7*x*6*x*5*x*4*x*3*x*2*x*1*x*0 and multiplier *Y* = *y*7*y*6*y*5*y*4*y*3*y*2*y*1*y*0. We call each row a *product component*, a shifted multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit. Each small box represents one product-component bit *yixj*, the logical AND of multiplier bit *yi* and multiplicand bit *xj*. The product *P* = *p*15*p*14 . ..*p*2*p*1*p*0 has 16 bits and is obtained by adding together all the product components. Figure shows one way to add up the product components. Here, the product-component bits have been spread out to make space, and each “+” box is a full adder equivalent to Figure. The carries in each row of full adders are connected to make an 8-bit ripple adder. Thus, the first ripple adder combines the first two product components to product the first partial product. Subsequent adders combine each partial product with the next product component.



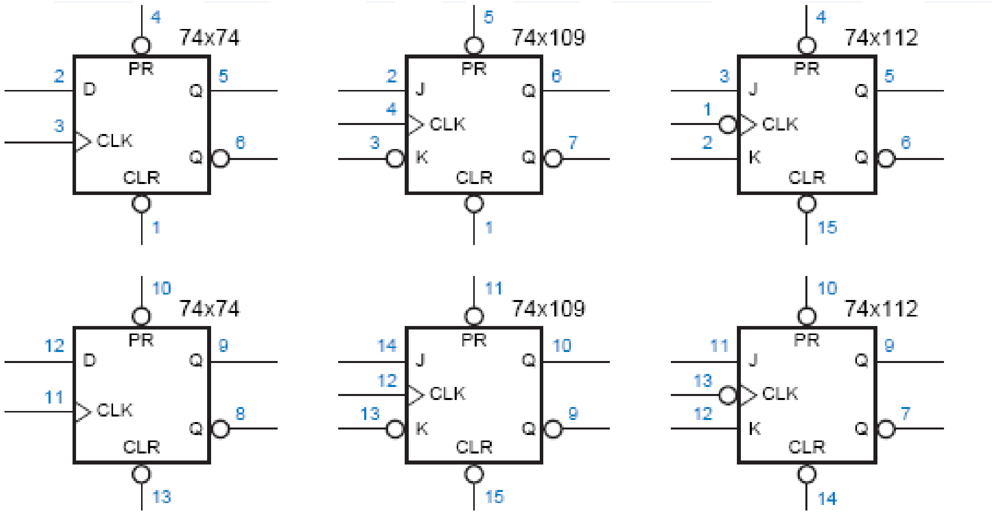
y_0x_7 y_0x_6 y_0x_5 y_0x_4 y_0x_3 y_0x_2 y_0x_1 y_0x_0
y_1x_7 y_1x_6 y_1x_5 y_1x_4 y_1x_3 y_1x_2 y_1x_1 y_1x_0
y_2x_7 y_2x_6 y_2x_5 y_2x_4 y_2x_3 y_2x_2 y_2x_1 y_2x_0
y_3x_7 y_3x_6 y_3x_5 y_3x_4 y_3x_3 y_3x_2 y_3x_1 y_3x_0
y_4x_7 y_4x_6 y_4x_5 y_4x_4 y_4x_3 y_4x_2 y_4x_1 y_4x_0
y_5x_7 y_5x_6 y_5x_5 y_5x_4 y_5x_3 y_5x_2 y_5x_1 y_5x_0
y_6x_7 y_6x_6 y_6x_5 y_6x_4 y_6x_3 y_6x_2 y_6x_1 y_6x_0
y_7x_7 y_7x_6 y_7x_5 y_7x_4 y_7x_3 y_7x_2 y_7x_1 y_7x_0
+
0
p_{15} p_{14} p_{13} p_{12} p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

UNIT III
Sequential Logic Design

Latches and Flip-Flops

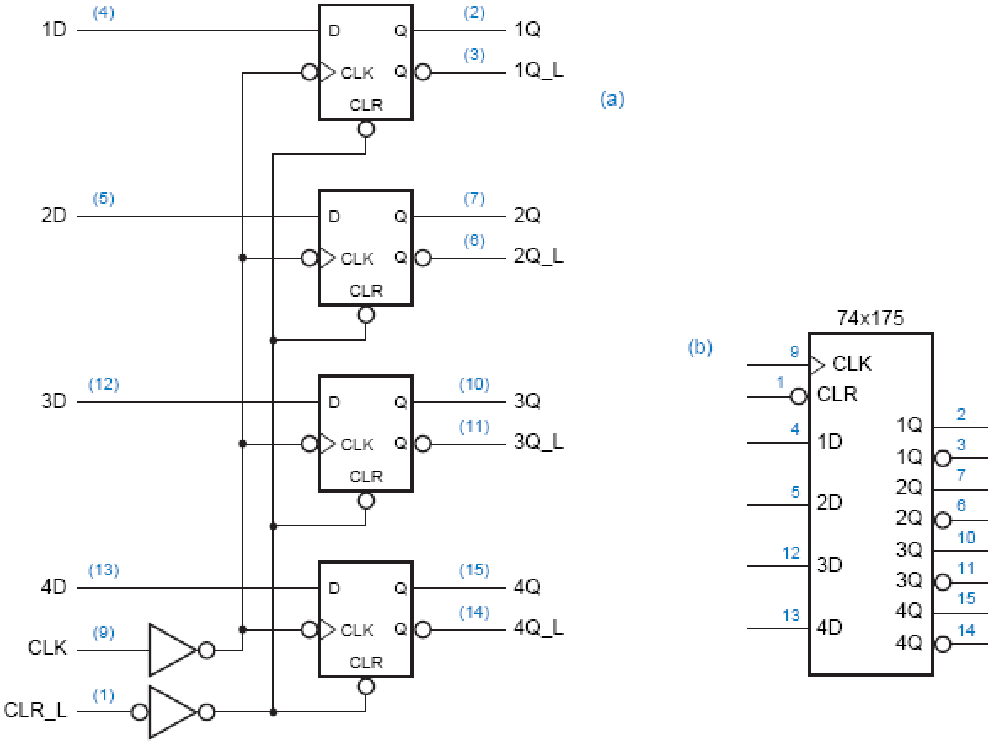
SSI Latches and Flip-Flops

Several different types of discrete latches and flip-flops are available as SSI parts. These devices are sometimes used in the design of state machines and “unstructured” sequential circuits that don’t fall into the categories of shift registers, counters, and other sequential MSI functions presented later in this chapter. However, SSI latches and flip-flops have been eliminated to a large extent in modern designs as their functions are embedded in PLDs and FPGAs. Nevertheless, a handful of these discrete building blocks still appear in many digital systems, so it’s important to be familiar with them. Figure shows the pinouts for several SSI sequential devices. The only latch in the figure is the *74x375,* which contains four D latches. Because of pin limitations, the latches are arranged in pairs with a common C control line for each pair. Among the devices in Figure, the most important is the *74x74*, which contains two independent positive-edge-triggered D flip-flops with preset and clear inputs. Besides the 74x74's use in "random" sequential circuits, fast versions of the part, such as the *74F74* and *74ACT74*, find application in synchronizers for asynchronous input signals. The *74x109* is a positive-edge-triggered J-K flip-flop with an active-low K input (named K or K_L). We Another J-K flip-flop is the *74x112*, which has an active-low clock input.



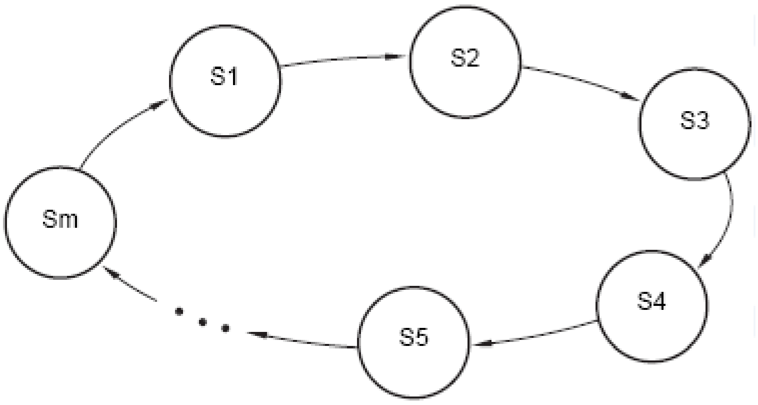
Multibit Registers and Latches

A collection of two or more D flip-flops with a common clock input is called a *register*. Registers are often used to store a collection of related bits, such as a byte of data in a computer. However, a single register can also be used to store unrelated bits of data or control information; the only real constraint is that all of the bits are stored using the same clock signal. Figure shows the logic diagram and logic symbol for a commonly used MSI register, the *74x175*. The 74x175 contains four edge-triggered D flip-flops with a common clock and asynchronous clear inputs. It provides both active high and active-low outputs at the external pins of the device. The individual flip-flops in a '175 are negative-edge triggered, as indicated by the inversion bubbles on their CLK inputs. However, the circuit also contains an inverter that makes the flip-flops positive-edge triggered with respect to the device's external CLK input pin. The common, active-low, clear signal (CLR_L) is connected to the asynchronous clear inputs of all four flip-flops. Both CLK and CLR_L are buffered before fanning out to the four flip-flops, so that a device driving one of these inputs sees only one unit load instead of four. This is especially important if a common clock or clear signal must drive many such registers.



Counters

The name *counter* is generally used for any clocked sequential circuit whose state diagram contains a single cycle, as in Figure. The *modulus* of a counter is the number of states in the cycle. A counter with *m* states is called a *modulo-m counter* or, sometimes, a *divide-by-m counter*. A counter with a non power- of-2 modulus has extra states that are not used in normal operation.



Ripple Counters

An *n*-bit binary counter can be constructed with just *n* flip-flops and no other components, for any value of *n*. Figure shows such a counter for *n*=4. Recall that a T flip-flop changes state (toggles) on every rising edge of its clock input. Thus, each bit of the counter toggles if and only if the immediately preceding bit changes from 1 to 0. This corresponds to a normal binary counting sequence—when a particular bit changes from 1 to 0, it generates a carry to the next most significant bit. The counter is called a *ripple counter* because the carry information ripples from the less significant bits to the more significant bits, one bit at a time.

Synchronous Counters

Although a ripple counter requires fewer components than any other type of binary counter, it does so at a price—it is slower than any other type of binary counter. In the worst case, when the most significant bit must change, the output is not valid until time *n=t/*TQ after the rising edge of CLK, where TQ is the propagation delay from input to output of a T flip-flop. A *synchronous counter* connects its entire flip-flop clock inputs to the same common CLK signal, so that all of the flip-flop outputs change at the same time, after only *t* TQ ns of delay. As shown in Figure this requires the use of T flip-flops with enable inputs; the output toggles on the rising edge of T if and only if EN is asserted. Combinational logic on the EN inputs determines which, if any, flip-flops toggle on each rising edge of T.

MSI Counters and Applications

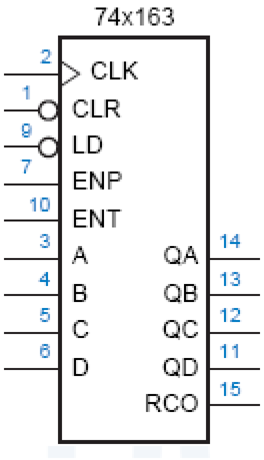
The most popular MSI counter is the *74x163*, a synchronous 4-bit binary counter with active-low load and clear inputs, with the traditional logic symbol shown in Figure. Its function is summarized by the state table in Table and its internal logic diagram is shown in Figure. The ’163 uses D flip-flops rather than T flip-flops internally to facilitate the load and clear functions. Each D input is driven by a 2-input multiplexer consisting of an OR gate and two AND gates. The multiplexer output is 0 if the CLR_L input is asserted. Otherwise, the top AND gate passes the data input (A, B, C, or D) to the output if LD_L is asserted. If neither CLR_L nor LD_L is asserted, the bottom AND gate passes the output of an XNOR gate to the multiplexer output.

State table for a 74x163 4-bit binary counter.

Inputs Current State Next State

CLR_L	**LD_L**	**ENT**	**ENP**	**QD**	**QC**	**QB**	**QA**	**QD□**	**QC□**	**QB□**	**QA□**
0	x	x	x	x	x	x	x	0	0	0	0
1	0	x	x	x	x	x	x	D	C	B	A
1	1	0	x	x	x	x	x	QD	QC	QB	QA
1	1	x	0	x	x	x	x	QD	QC	QB	QA
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	1	0

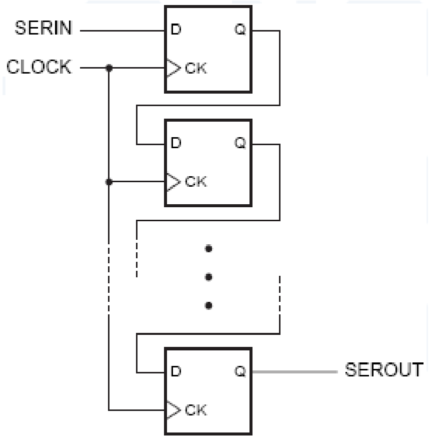
1 1 1 1 0 0 1 0 0 0 1 1
1 1 1 1 0 0 1 1 0 1 0 0
1 1 1 1 0 1 0 0 0 1 0 1
1 1 1 1 0 1 0 1 0 1 1 0
1 1 1 1 0 1 1 0 0 1 1 1
1 1 1 1 0 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0 1 0 0 1
1 1 1 1 1 0 0 1 1 0 1 0
1 1 1 1 1 0 1 0 1 0 1 1
1 1 1 1 1 0 1 1 1 1 0 0
1 1 1 1 1 1 0 0 1 1 0 1
1 1 1 1 1 1 0 1 1 1 1 0
1 1 1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0

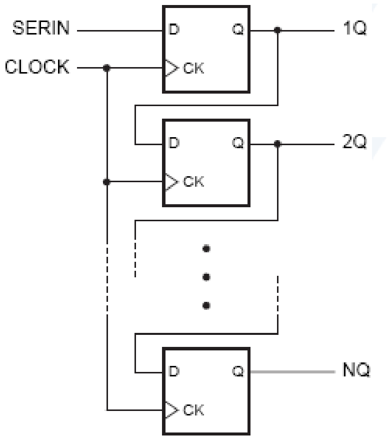


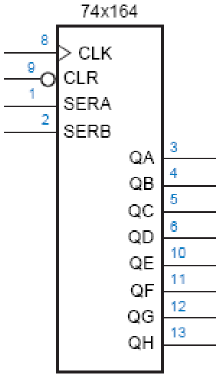
Shift Registers

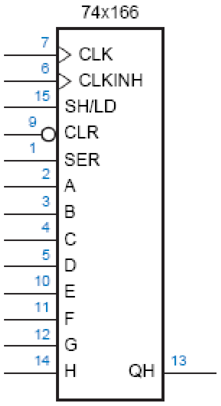
Shift-Register Structure

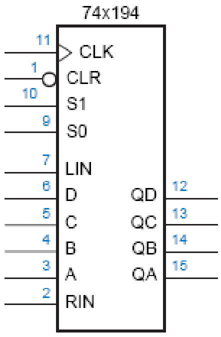
A *shift register* is an *n*-bit register with a provision for shifting its stored data by one bit position at each tick of the clock. Figure shows the structure of a serial-in, serial-out shift register. The *serial input,* SERIN, specifies a new bit to be shifted into one end at each clock tick. This bit appears at the *serial output,* SEROUT, after *n* clock ticks, and is lost one tick later. Thus, an *n*-bit serial-in, serial-out shift register can be used to delay a signal by *n* clock ticks. A *serial-in, parallel-out shift register,* shown in Figure, has outputs for all of its stored bits, making them available to other circuits. Such a shift register can be used to perform *serial-to-parallel conversion,* as explained later in this section. Conversely, it is possible to build a *parallel-in, serial-out shift register.* Figure shows the general structure of such a device. At each clock tick, the register either loads new data from inputs 1D–ND, or it shifts its current contents, depending on the value of the LOAD/SHIFT control input (which could be named LOAD or SHIFT_L). Internally, the device uses a 2-input multiplexer on each flip-flop's D input to select between the two cases. A parallel-in, serial-out shift register can be used to perform *parallel-to-serial conversion*, as explained later in this section. By providing outputs for all of the stored bits in a parallel-in shift register, we obtain the p*arallel-in, parallel-out shift register* shown in Figure, Such a device is general enough to be used in any of the applications of the previous shift registers.











Impediments to Synchronous Design

Although the synchronous approach is the most straightforward and reliable method of digital system design, a few nasty realities can get in the way. We'll discuss them in this section.

Clock Skew

Synchronous systems using edge-triggered flip-flops work properly only if all flip-flops see the triggering clock edge at the same time. Figure shows what can happen otherwise. Here, two flip-flops are theoretically clocked by the same signal, but the clock signal seen by FF2 is delayed by a significant amount relative to FF1's clock. This difference between arrival times of the clock at different devices is called *clock skew*. We've named the delayed clock in Figure "CLOCKD." If FF1's propagation delay from CLOCK to Q1 is short, and if the physical connection of Q1 to FF2 is short, then the change in Q1 caused by a CLOCK edge may actually reach FF2 *before* the corresponding CLOCKD edge. In this case, FF2 may go to an incorrect next state determined by the *next* state of FF1 instead of the current state, as shown in (b). If the change in Q1 arrives at FF2 only slightly early relative to CLOCKD, then FF2’s hold-time specification may be violated, in which case FF2 may become metastable and produce an unpredictable output..

Read-Only Memory (ROM)

One major type of memory that is used in PCs is called *read-only memory*, or *ROM* for short. ROM is a type of memory that normally can only be read, as opposed to RAM which can be both read and written. There are two main reasons that read-only memory is used for certain functions within the PC:

- **Permanence:** The values stored in ROM are always there, whether the power is on or not. A ROM can be removed from the PC, stored for an indefinite period of time, and then replaced, and the data it contains will still be there. For this reason, it is called *non-volatile storage*. A hard disk is also non-volatile, for the same reason, but regular RAM is not.
- **Security:** The fact that ROM cannot easily be modified provides a measure of security against accidental (or malicious) changes to its contents. You are not going to find viruses infecting true ROMs, for example; it's just not possible. (It's technically possible with *erasable* EPROMs, though in practice never seen.)

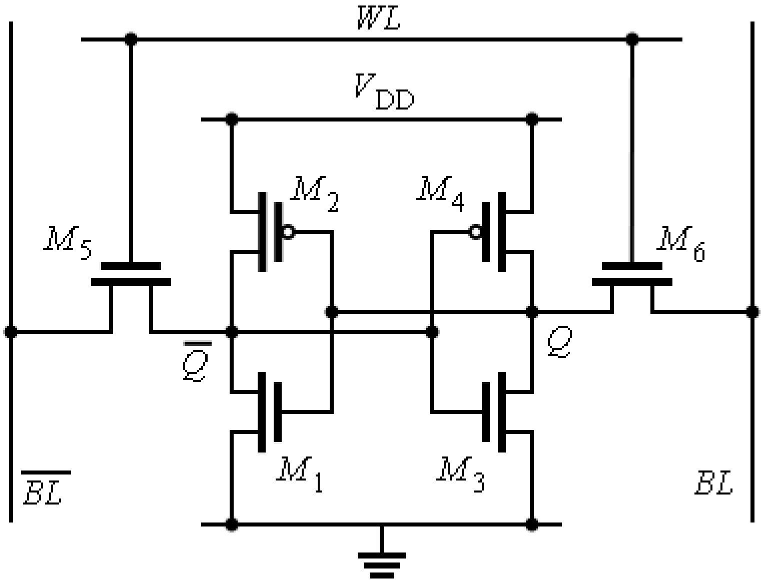
Read-only memory is most commonly used to store system-level programs that we want to have available to the PC at all times. The most common example is the system BIOS program, which is stored in a ROM called (amazingly enough) the *system BIOS ROM*. Having this in a permanent ROM means it is available when the power is turned on so that the PC can use it to boot up the system. Remember that when you first turn on the PC the system memory is empty, so there has to be *something* for the PC to use when it starts up. See this section for a description of the system BIOS ROM; see here for a description of the system boot sequence.

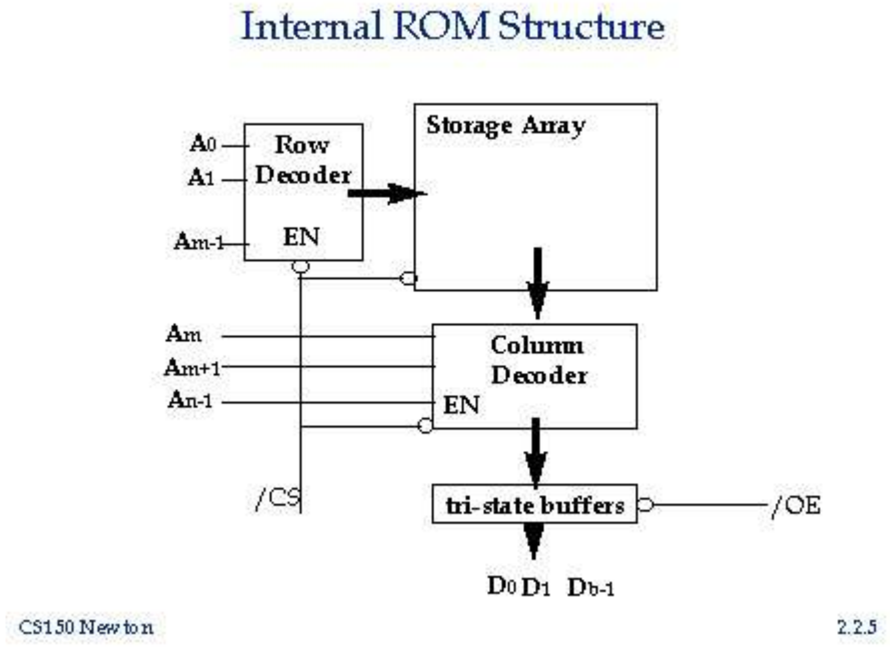
While the whole point of a ROM is supposed to be that the contents cannot be changed, there are times when being able to change the contents of a ROM can be very useful. There are several ROM variants that can be changed under certain circumstances; these can be thought of as "*mostly* read-only memory". The following are the different types of ROMs with a description of their relative modifiability:

- **ROM:** A regular ROM is constructed from hard-wired logic, encoded in the silicon itself, much the way that a processor is. It is designed to perform a specific function and cannot be changed. This is inflexible and so regular ROMs are only used generally for programs that are static (not changing often) and mass-produced. This product is analogous to a commercial software CD-ROM that you purchase in a store.

- **Programmable ROM (PROM):** This is a type of ROM that can be programmed using special equipment; it can be written to, but only once. This is useful for companies that make their own ROMs from software they write, because when they change their code they can create new PROMs without requiring expensive equipment. This is similar to the way a CD-ROM recorder works by letting you "burn" programs onto blanks once and then letting you read from them many times. In fact, programming a PROM is also called *burning*, just like burning a CD-R, and it is comparable in terms of its flexibility.
- **Erasable Programmable ROM (EPROM):** An *EPROM* is a ROM that can be erased and reprogrammed. A little glass window is installed in the top of the ROM package, through which you can actually see the chip that holds the memory. Ultraviolet light of a specific frequency can be shined through this window for a specified period of time, which will erase the EPROM and allow it to be reprogrammed again. Obviously this is much more useful than a regular PROM, but it does require the erasing light. Continuing the "CD" analogy, this technology is analogous to a reusable CD-RW.
- **Electrically Erasable Programmable ROM (EEPROM):** The next level of erasability is the *EEPROM*, which can be erased under software control. This is the most flexible type of ROM, and is now commonly used for holding BIOS programs. When you hear reference to a "flash BIOS" or doing a BIOS upgrade by "flashing", this refers to reprogramming the BIOS EEPROM with a special software program. Here we are blurring the line a bit between what "read-only" really means, but remember that this rewriting is done maybe once a year or so, compared to real read-write memory (RAM) where rewriting is done often many times per second!

ROM CIRCUIT DIAGRAM




Internal ROM Structure

The memory matrix is the main part of the ROM. For each wordline, a connection is made to those bitlines (vertical) that should be activated for the corresponding memory word. In bipolar technology, simple diodes can be used to make the connections of wordlines to bitlines. However, usually transistors are used to amplify the wordline signal.

In MOS technologies, N-type transistors are preferred, because they are faster than P-type transistors of the same size. The transistor gate is connected to the wordline, drain to the bitline, and the transistor source to Vss (ground). This results in the wired-AND structure shown in the applet. The pullup resistors on each bitline drive the bitline to a weak high voltage (logical H), when no transistors are active. If a transistor is used at the connection of a wordline and bitline, the transistor will conduct whenever the wordline is active, driving the bitline to a low voltage (logical 0).

As N-type transistors are used in the memory matrix, an active bitline is driven to zero voltage, while an inactive bitline remains at (weak) high. Therefore, an additional stage of amplifiers and output buffers is required to generate a strong output signal. The applet uses a single stage of inverters.

All together, the ROM shown in the applet implements the following logical function:

addr	A3	A3	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
15	1	1	1	1	1	1	1	1	1	1	1	1
14	1	1	1	0	0	0	0	0	0	0	0	0
13	1	1	0	1	1	0	0	1	0	0	0	0
12	1	1	0	0	0	0	0	0	0	0	1	1
11	1	0	1	1	0	0	0	0	1	1	1	1
10	1	0	1	0	1	1	1	1	0	0	0	0
9	1	0	0	1	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0
7	0	1	1	1	0	0	0	0	0	0	0	1
6	0	1	1	0	0	0	0	0	0	0	1	0
5	0	1	0	1	0	0	0	0	0	1	0	0
4	0	1	0	0	0	0	0	0	1	0	0	0
3	0	0	1	1	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	1	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0

Due to electrical reasons it is difficult to connect more that a few hundred transistors to one bitline or wordline. Therefore, larger ROMs use a slightly different architecture, with a set of smaller memory blocks instead of one single large memory block. Each memory block uses approximately the same number of wordlines and bitlines (the applet has 16 wordlines and 8 bitlines). For example, a good size for one memory block might consist of 256 wordlines and 256 bitlines. This means that a 64Kx8 bit ROM would be organized internally as a set of (65536x8) / (256x256) = 8 memory blocks. The top 3 address bits select the memory block to use, the next 8 bits select the wordline to use, and the remaining 5 address bits select which part of the memory block output of 256 bits is to be sent to the 8 data outputs.

Each bit in an SRAM is stored on four transistors that form two cross-coupled inverters. This storage cell has two stable states which are used to denote **0** and **1**. Two additional *access* transistors serve to control the access to a storage cell during read and write operations. A typical SRAM uses six MOSFETs to store each memory bit. In addition to such 6T SRAM, other kinds of SRAM chips use 8T, 10T, or more transistors per bit.This is sometimes used to implement more than one (read and/or write) port, which may be useful in certain types of video memory and register files implemented with multi-ported SRAM circuitry.Generally, the fewer transistors needed per cell, the smaller each cell can be. Since the cost of processing a silicon wafer is relatively fixed, using smaller cells and so packing more bits on one wafer reduces the cost per bit of memory.

Memory cells that use fewer than 6 transistors are possible — but such 3T or 1T cells are DRAM, not SRAM (even the so-called 1T-SRAM).Access to the cell is enabled by the word line (WL in figure) which controls the two *access* transistors M_5 and M_6 which, in turn, control whether the cell should be connected to the bit lines: BL and BL. They are used to transfer data for both read and write operations. Although it is not strictly necessary to have two bit lines, both the signal and its inverse are typically provided in order to improve noise margins.

During read accesses, the bit lines are actively driven high and low by the inverters in the SRAM cell. This improves SRAM bandwidth compared to DRAMs—in a DRAM, the bit line is connected to storage capacitors and charge sharing causes the bitline to swing upwards or downwards. The symmetric structure of SRAMs also allows for differential signaling, which makes small voltage swings more easily detectable. Another difference with DRAM that contributes to making SRAM faster is that commercial chips accept all address bits at a time. By comparison, commodity DRAMs have the address multiplexed in two halves, i.e. higher bits followed by lower bits, over the same package pins in order to keep their size and cost down.The size of an SRAM with m address lines and n data lines is 2^m words, or $2^m \times n$ bits.

SRAM operation

An SRAM cell has three different states. It can be in: *standby* (the circuit is idle), *reading* (the data has been requested) and *writing* (updating the contents). The SRAM to operate in read mode and write mode should have "readability" and "write stability" respectively. The three different states work as follows:

- **Standby**

If the word line is not asserted, the *access* transistors M_5 and M_6 disconnect the cell from the bit lines. The two cross-coupled inverters formed by M_1 – M_4 will continue to reinforce each other as long as they are connected to the supply.

- **Reading**

Assume that the content of the memory is a **1**, stored at Q. The read cycle is started by precharging both the bit lines to a logical **1**, then asserting the word line WL, enabling both the *access* transistors. The second step occurs when the values stored in Q and Q are transferred to the bit lines by leaving BL at its precharged value and discharging BL through M_1 and M_5 to a logical **0** (i. e. eventually discharging through the transistor M_1 as it is turned on because the Q is logically set to **1**). On the BL side, the transistors M_4 and M_6 pull the bit line toward V_{DD}, a logical **1** (i. e. eventually being charged by the transistor M_4 as it is turned on because Q is logically set to **0**). If the content of the memory was a **0**, the opposite would happen and BL would be pulled toward **1** and BL toward **0**. Then these BL and BL will have a small difference of delta between them and then these lines reach a sense amplifier, which will sense which line has higher voltage and thus will tell whether there was **1** stored or **0**. The higher the sensitivity of sense amplifier, the faster the speed of read operation is.

- **Writing**

The start of a write cycle begins by applying the value to be written to the bit lines. If we wish to write a **0**, we would apply a **0** to the bit lines, i.e. setting BL to **1** and BL to **0**. This is similar to applying a reset pulse to an SR-latch, which causes the flip flop to change state. A **1** is written by inverting the values of the bit lines. WL is then asserted and the value that is to be stored is latched in. Note that the reason this works is that the bit line input-drivers are designed to be much stronger than the relatively weak transistors in the cell itself, so that they can easily override the previous state of the cross-coupled inverters. Careful sizing of the transistors in an SRAM cell is needed to ensure proper operation.

Bus behavior

RAM with an access time of 70 ns will output valid data within 70 ns from the time that the address lines are valid. But the data will remain for a hold time as well (5-10 ns). Rise and fall times also influence valid timeslots with approximately ~5 ns. By reading the lower part of an address range bits in sequence (page cycle) one can read with significantly shorter access time (30 ns). [7]

Applications and uses

Characteristics

SRAM is more expensive, but faster and significantly less power hungry (especially idle) than DRAM. It is therefore used where either bandwidth or low power, or both, are principal considerations. SRAM is also easier to control (interface to) and generally more truly *random access* than modern types of DRAM. Due to a more complex internal structure, SRAM is less dense than DRAM and is therefore not used for high-capacity, low-cost applications such as the main memory in personal computers.

Clock rate and power

The power consumption of SRAM varies widely depending on how frequently it is accessed; it **can** be as power-hungry as dynamic RAM, when used at high frequencies, and some ICs can consume many watts at full bandwidth. On the other hand, static RAM used at a somewhat slower pace, such as in applications with moderately clocked microprocessors, draws very little power and can have a nearly negligible power consumption when sitting idle — in the region of a few micro-watts.

Static RAM exists primarily as:

- general purpose products
  - with *asynchronous* interface, such as the 28 pin 32Kx8 chips (usually named XXC256), and similar products up to 16 Mbit per chip

    - with *synchronous* interface, usually used for caches and other applications requiring burst transfers, up to 18 Mbit (256Kx72) per chip
- integrated on chip
    - as RAM or cache memory in micro-controllers (usually from around 32 bytes up to 128 kilobytes)
    - as the primary caches in powerful microprocessors, such as the x86 family, and many others (from 8 kB, up to several megabytes)
    - to store the registers and parts of the state-machines used in some microprocessors (see register file)
    - on application specific ICs, or ASICs (usually in the order of kilobytes)
    - in FPGAs and CPLDs

Embedded use

Many categories of industrial and scientific subsystems, automotive electronics, and similar, contain static RAM. Some amount (kilobytes or less) is also embedded in practically all modern appliances, toys, etc. that implement an electronic user interface. Several megabytes may be used in complex products such as digital cameras, cell phones, synthesizers, etc.

SRAM in its dual-ported form is sometimes used for realtime digital signal processing circuits

In computers

SRAM is also used in personal computers, workstations, routers and peripheral equipment: internal CPU caches and external burst mode SRAM caches, hard disk buffers, router buffers, etc. LCD screens and printers also normally employ static RAM to hold the image displayed (or to be printed). Small SRAM buffers are also found in CDROM and CDRW drives; usually 256 kB or more are used to buffer track data, which is transferred in blocks instead of as single values. The same applies to cable modems and similar equipment connected to computers.

Types of SRAM

Non-volatile SRAM

Non-volatile SRAMs have standard SRAM functionality, but they save the data when the power supply is lost, ensuring preservation of critical information. nvSRAMs are used in a wide range of situations—networking, aerospace, and medical, among many others[9] —where the preservation of data is critical and where batteries are impractical.

Asynchronous SRAM

Asynchronous SRAM are available from 4 Kb to 64 Mb. The fast access time of SRAM makes asynchronous SRAM appropriate as main memory for small cache-less embedded processors used in everything from industrial electronics and measurement systems to hard disks and networking equipment, among many other applications. They are used in various applications like switches and routers, IP-Phones, IC-Testers, DSLAM Cards, to Automotive Electronics.

By transistor type

- Bipolar junction transistor (used in TTL and ECL) — very fast but consumes a lot of power
- MOSFET (used in CMOS) — low power and very common today

By function

- Asynchronous — independent of clock frequency; data in and data out are controlled by address transition
- Synchronous — all timings are initiated by the clock edge(s). Address, data in and other control signals are associated with the clock signals

By feature

- ZBT (ZBT stands for zero bus turnaround) — the turnaround is the number of clock cycles it takes to change access to the SRAM from **write** to **read** and vice versa. The turnaround for ZBT SRAMs or the latency between read and write cycle is zero.

- syncBurst (syncBurst SRAM or synchronous-burst SRAM) — features synchronous burst write access to the SRAM to increase write operation to the SRAM
- DDR SRAM — Synchronous, single read/write port, double data rate I/O
- Quad Data Rate SRAM — Synchronous, separate read & write ports, quadruple data rate I/O

By flip-flop type

- Binary SRAM
- Ternary SRAM

Dynamic random-access memory (DRAM)

It is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1. Since capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a *dynamic* memory as opposed to SRAM and other *static* memory.

The main memory (the "RAM") in personal computers is Dynamic RAM (DRAM). It is the RAM in laptop, notebook and workstation computers as well as some of the RAM of home game consoles (PlayStation3, Xbox 360 and Wii),

The advantage of DRAM is its structural simplicity: only one transistor and a capacitor are required per bit, compared to six transistors in SRAM. This allows DRAM to reach very high densities. Unlike flash memory, DRAM is volatile memory (cf. non-volatile memory), since it loses its data quickly when power is removed. The transistors and capacitors used are extremely small; billions can fit on a single memory chip.

DRAM is usually arranged in a square array of one capacitor and transistor per data bit storage cell. The illustrations to the right show a simple example with 4 by 4 cells (modern DRAM matrices are many thousands of cells in height and width).

The long horizontal lines connecting each row are known as word Lines. Each column of cells is actually composed of two bit lines, each one connected to every other storage cell in the column (the illustration to the right does not include this important detail). They are generally known as the + and − bit lines. A sense amplifier is essentially a pair of cross-connected inverters between the bit lines, with the first inverter connected from the + bit line to the − bit line, and the second from the − bit line to the + bit line. This is an example of positive feedback, and the arrangement is only stable after one bit line is high and one bit line is low.

To read a bit from a column, the following operations take place:

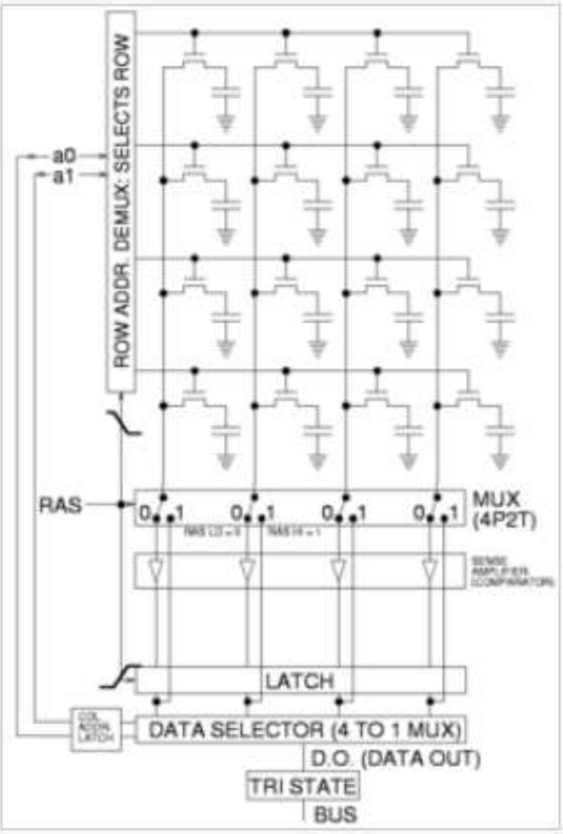
1. The sense amplifier is disconnected, then the bit lines are precharged to exactly equal voltages that are in-between high and low logic levels. The bit lines are physically symmetrical to keep the capacitance as equal and therefore the voltages as equal as possible.
2. The precharge circuit is switched off. Because the bit lines are relatively long, they have enough capacitance to maintain the pre-charged voltage for a brief time. This is an example of dynamic logic.
3. The desired row's word line is then driven high to connect a cell's storage capacitor to its bit line. This causes the transistor to conduct, transferring charge between the storage cell and the connected bit line. If the storage cell capacitor is discharged, it will greatly decrease the voltage on the bit-line as the precharge is transferred to the storage capacitor. If the storage cell is charged, the bit-line voltage decreases only slightly; this is because every effort is made to keep the capacitance of the storage cells high and the capacitance of the bit lines low.
4. The sense amplifier is switched on. The positive feedback takes over and amplifies the small voltage difference between bit-lines until one bit line is fully at the lowest voltage and the other is at the maximum high voltage. Once this has happened, the row is "open" (the desired cell data is available).

5. All columns are sensed in simultaneously and the result sampled into the data latch. A provided Column address then selects which latch bit to connect to the external circuit. Many reads can be performed quickly without delay sense for the open row, all data has already been sensed and latched.
6. While reading all columns proceeds (the normal and desirable method because it most quickly provides data), current is flowing back up the bit lines from the sense amplifiers to the storage cells. This reinforces (i.e. "refreshes") the charge in the storage cell by increasing the voltage in the storage capacitor if it was charged to begin with, or by keeping it discharged if it was empty. Note that due to the length of the bit lines creating a fairly long propagation delay for the charge to be transferred, this takes significant time beyond the end of sense amplification, and thus overlaps with one or more column reads.
7. When done with the reading all the columns in the current row, the word line is switched off to disconnect the cell storage capacitors (the row is "closed"), the sense amplifier is switched off, and the bit lines are precharged again.

To write to memory, the row is opened and a given column's sense amplifier is temporarily forced to the desired high or low voltage state, thus it drives the bit line to charge or discharge the cell storage capacitor to the desired value. Due to positive feedback, the amplifier will then hold it stable even after the forcing is removed. During a write to a particular cell, all the columns in a row are sensed simultaneously just as in reading, a single column's cell storage capacitor charge is changed, and then the entire row is written back in, as illustrated in the figure to the right.

Typically, manufacturers specify that each row must have its storage cell capacitors refreshed every 64 ms or less, as defined by the JEDEC (Foundation for developing Semiconductor Standards) standard. Refresh logic is provided in a DRAM controller which automates the periodic refresh, that is no software or other hardware has to perform it. This makes the controller's logic circuit more complicated, but this drawback is outweighed by the fact that DRAM is much cheaper per storage cell and because each storage cell is very simple, DRAM has much greater capacity per geographic area than SRAM.

Some systems refresh every row in a burst of activity involving all rows every 64 ms. Other systems refresh one row at a time staggered throughout the 64 ms interval. For example, a system with 2^{13} = 8192 rows would require a staggered refresh rate of one row every 7.8 µs which is 64 ms divided by 8192 rows. A few real-time systems refresh a portion of memory at a time determined by an external timer function that governs the operation of the rest of a system, such as the vertical blanking interval that occurs every 10–20 ms in video equipment. All methods require some sort of counter to keep track of which row is the next to be refreshed. Most DRAM chips include that counter. Older types require external refresh logic to hold the counter (under some conditions, most of the data in DRAM can be recovered even if the DRAM has not been refreshed for several minutes).


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
0 1
MUX (4P2T)
SENSE AMPLIFIER (COMPARATOR)
LATCH
COL ADDR LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

Principle of operation of DRAM read, for simple 4 by 4 array.

Synchronous DRAM

This is the basic form from which all others derive. An asynchronous DRAM chip has power connections, some number of address inputs (typically 12), and a few (typically one or four) bidirectional data lines. There are four active-low control signals:

- **/RAS**, the Row Address Strobe. The address inputs are captured on the falling edge of /RAS, and select a row to open. The row is held open as long as /RAS is low.
- **/CAS**, the Column Address Strobe. The address inputs are captured on the falling edge of /CAS, and select a column from the currently open row to read or write.
- **/WE**, Write Enable. This signal determines whether a given falling edge of /CAS is a read (if high) or write (if low). If low, the data inputs are also captured on the falling edge of /CAS.
- **/OE**, Output Enable. This is an additional signal that controls output to the data I/O pins. The data pins are driven by the DRAM chip if /RAS and /CAS are low, /WE is high, and /OE is low. In many applications, /OE can be permanently connected low (output always enabled), but it can be useful when connecting multiple memory chips in parallel.

This interface provides direct control of internal timing. When /RAS is driven low, a /CAS cycle must not be attempted until the sense amplifiers have sensed the memory state, and /RAS must not be returned high until the storage cells have been refreshed. When /RAS is driven high, it must be held high long enough for precharging to complete.

Although the RAM is asynchronous, the signals are typically generated by a clocked memory controller, which limits their timing to multiples of the controller's clock cycle.

UNIT IV
LOGIC FAMILIES AND SEMICONDUCTOR MEMORIES

Introduction to logic families

Logic Signals and Gates

Digital logic hides the pitfalls of the analog world by mapping the infinite set of real values for a physical quantity into two subsets corresponding to just two possible numbers or logic values—0 and 1. As a result, digital logic circuits can be analyzed and designed functionally, using switching algebra, tables, and other abstract means to describe the operation of well-behaved 0s and 1s in a circuit.

A logic value, 0 or 1, is often called a binary digit, or bit. If an application requires more than two discrete values, additional bits may be used, with a set of n bits representing 2n different values. Examples of the physical phenomena used to represent bits in some modern (and not-so-modern) digital technologies are given in Table 3-1. With most phenomena, there is an undefined region between the 0 and 1 states (e.g., voltage = 1.8 V, dim light, capacitor slightly charged, etc.). This undefined region is needed so that the 0 and 1 states can be unambiguously defined and reliably detected. Noise can more easily corrupt results if the boundaries separating the 0 and 1 states are too close. When discussing electronic logic circuits such as CMOS and TTL, digital designers often use the words “LOW” and “HIGH” in place of “0” and “1” to remind them that they are dealing with real circuits, not abstract quantities:

LOW A signal in the range of algebraically lower voltages, which is interpreted as a logic 0.

HIGH A signal in the range of algebraically higher voltages, which is interpreted as a logic 1.

Note that the assignments of 0 and 1 to LOW and HIGH are somewhat arbitrary. Assigning 0 to LOW and 1 to HIGH seems most natural, and is called positive logic. The opposite assignment, 1 to LOW and 0 to HIGH, is not often used, and is called negative logic. Because a wide range of physical values represent the same binary value, digital logic is highly immune to component and power supply variations and noise. Furthermore, buffer amplifier circuits can be used to regenerate “weak” values into “strong” ones, so that digital signals can be transmitted over arbitrary distances without loss of information. For example, a buffer amplifier for CMOS Digital logic hides the pitfalls of the analog world by mapping the infinite set of real values for a physical quantity into two subsets corresponding to just two possible numbers or logic values—0 and 1. As a result, digital logic circuits can be analyzed and designed functionally, using switching algebra, tables, and other abstract means to describe the operation of well-behaved 0s and 1s in a circuit.

A logic value, 0 or 1, is often called a binary digit, or bit. If an application requires more than two discrete values, additional bits may be used, with a set of n bits representing 2n different values. Examples of the physical phenomena used to represent bits in some modern (and not-so-modern) digital technologies are given in Table 3-1. With most phenomena, there is an undefined region between the 0 and 1 states (e.g., voltage = 1.8 V, dim light, capacitor slightly charged, etc.). This undefined region is needed so that the 0 and 1 states can be unambiguously defined and reliably detected. Noise can more easily corrupt results if the boundaries separating the 0 and 1 states are too close.

When discussing electronic logic circuits such as CMOS and TTL, digital designers often use the words “LOW” and “HIGH” in place of “0” and “1” to remind them that they are dealing with real circuits, not abstract quantities: LOW A signal in the range of algebraically lower voltages, which is interpreted as a logic 0.

HIGH A signal in the range of algebraically higher voltages, which is interpreted as a logic 1.

Note that the assignments of 0 and 1 to LOW and HIGH are somewhat arbitrary. Assigning 0 to LOW and 1 to HIGH seems most natural, and is called positive logic. The opposite assignment, 1 to LOW and 0 to HIGH, is not often used, and is called negative logic. Because a wide range of physical values represent the same binary value, digital logic is highly immune to component and power supply variations and noise. Furthermore, buffer amplifier circuits can be used to regenerate “weak” values into “strong” ones, so that digital signals can be transmitted over arbitrary distances without loss of information. For example, a buffer amplifier for CMOS logic converts any HIGH input voltage into an output very close to 5.0 V, and any

LOW input voltage into an output very close to 0.0 V.

A logic circuit can be represented with a minimum amount of detail simply as a “black box” with a certain number of inputs and outputs. For example, Figure 3-1 shows a logic circuit with three inputs and one output. However, this representation does not describe how the circuit responds to input signals.

From the point of view of electronic circuit design, it takes a lot of information

to describe the precise electrical behavior of a circuit. However, since the inputs of a digital logic circuit can be viewed as taking on only discrete 0 and 1 values, the circuit’s “logical” operation can be described with a table that ignores electrical behavior and lists only discrete 0 and 1 values.

CMOS LOGIC

The functional behavior of a CMOS logic circuit is fairly easy to understand, even if your knowledge of analog electronics is not particularly deep. The basic (and typically only) building blocks in CMOS logic circuits are MOS transistors, described shortly. Before introducing MOS transistors and CMOS logic circuits, we must talk about logic levels.

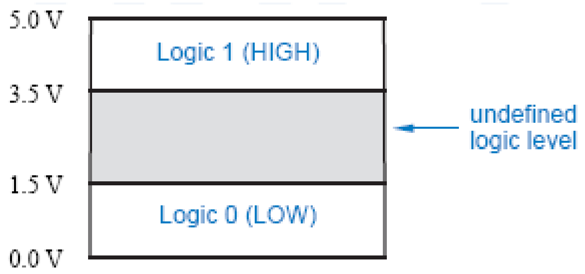
CMOS Logic Levels

Abstract logic elements process binary digits, 0 and 1. However, real logic circuits process electrical signals such as voltage levels. In any logic circuit, there is a range of voltages (or other circuit conditions) that is interpreted as a logic 0, and another, non overlapping range that is interpreted as a logic 1. A typical CMOS logic circuit operates from a 5-volt power supply. Such a circuit may interpret any voltage in the range 0–1.5 V as a logic 0, and in the range 3.5–5.0 V as a logic 1. Thus, the definitions of LOW and HIGH for 5-volt CMOS logic are as shown in Figure. Voltages in the intermediate range are not expected to occur except during signal transitions, and yield undefined logic values (i.e., a circuit may interpret them as either 0 or 1). CMOS circuits using other power supply voltages, such as 3.3 or 2.7 volts, partition the voltage range similarly.

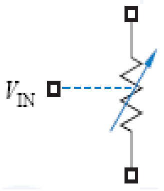
MOS Transistors

A MOS transistor can be modeled as a 3-terminal device that acts like a voltage controlled resistance. As suggested by Figure an input voltage applied to one terminal controls the resistance between the remaining two terminals. In digital logic applications, a MOS transistor is operated so its resistance is always either very high (and the transistor is "off") or very low (and the transistor is "on").

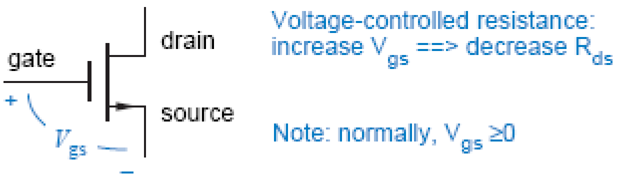
Logic Levels for Typical CMOS Circuits



MOS transistor as Voltage Controlled Resistor



There are two types of MOS transistors, *n*-channel and *p*-channel; the names refer to the type of semiconductor material used for the resistance-controlled terminals. The circuit symbol for an *n-channel MOS (NMOS) transistor* is shown in Figure. The terminals are called *gate*, *source*, and *drain*. (Note that the "gate" of a MOS transistor has nothing to do with a "logic gate.") As you might guess from the orientation of the circuit symbol, the drain is normally at a higher voltage than the source.



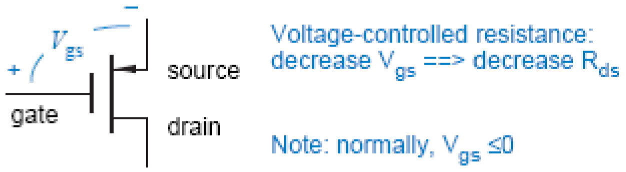
Voltage-controlled resistance: increase V_{gs} ==> decrease R_{ds}

Note: normally, $V_{gs} \geq 0$

Circuit Symbol for NMOS Transistor

The voltage from gate to source (*V*gs) in an NMOS transistor is normally zero or positive. If *V*gs = 0, then the resistance from drain to source (*R*ds) is very high, on the order of a megohm (106 ohms) or more. As we increase *V*gs (i.e., increase the voltage on the gate), *R*ds decreases to a very low value, 10 ohms or less in some devices. The circuit symbol for a *p-channel MOS (PMOS) transistor* is shown in Figure. Operation is analogous to that of an NMOS transistor, except that the source is normally at a higher voltage than the drain, and *V*gs is normally zero or negative. If *V*gs is zero, then the resistance from source to drain (*R*ds) is very high. As we algebraically decrease *V*gs (i.e., *decrease* the voltage on the gate), *R*ds decreases to a very low value. The gate of a MOS transistor has a very high impedance. That is, the gate is separated from the source and the drain by an insulating material with a very high resistance. However, the gate voltage creates an electric field that enhances or retards the flow of current between source and drain. This is the “field effect” in the “MOSFET” name.

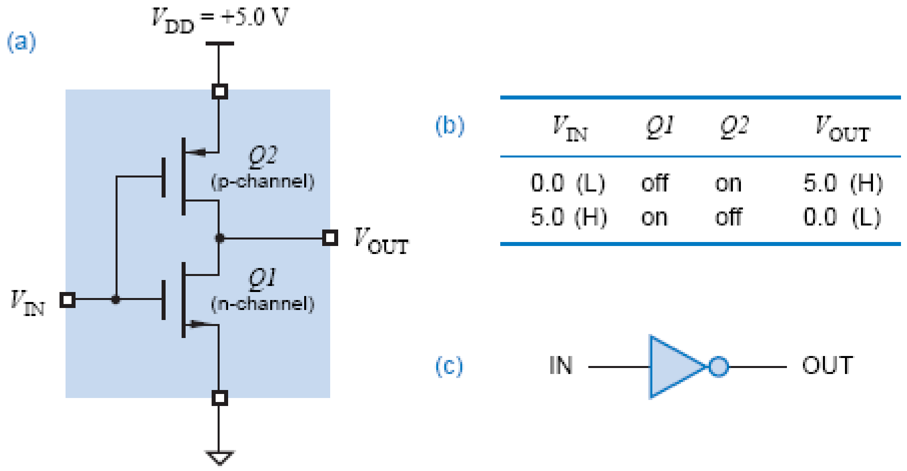
Circuit Symbol for NMOS Transistor


+
V_{gs}
−
gate
source
drain
Voltage-controlled resistance:
decrease V_{gs} ==> decrease R_{ds}
Note: normally, $V_{gs} \leq 0$

Basic CMOS Inverter Circuit

NMOS and PMOS transistors are used together in a complementary way to form *CMOS logic*. The simplest CMOS circuit, a logic inverter, requires only one of each type of transistor, connected as shown in Figure. The power supply voltage, *V*DD, typically may be in the range 2–6 V, and is most often set at 5.0 V for compatibility with TTL circuits. Ideally, the functional behavior of the CMOS inverter circuit can be characterized by just two cases tabulated in Figure *V*IN is 0.0 V. In this case, the bottom, *n*-channel transistor *Q1* is off, since its *V*gs is 0, but the top, *p*-channel transistor *Q2* is on, since its *V*gs is a large negative value (□5.0 V). Therefore, *Q2* presents only a small resistance between the power supply terminal (*V*DD, □5.0 V) and the output terminal (*V*OUT), and the output voltage is 5.0 V.

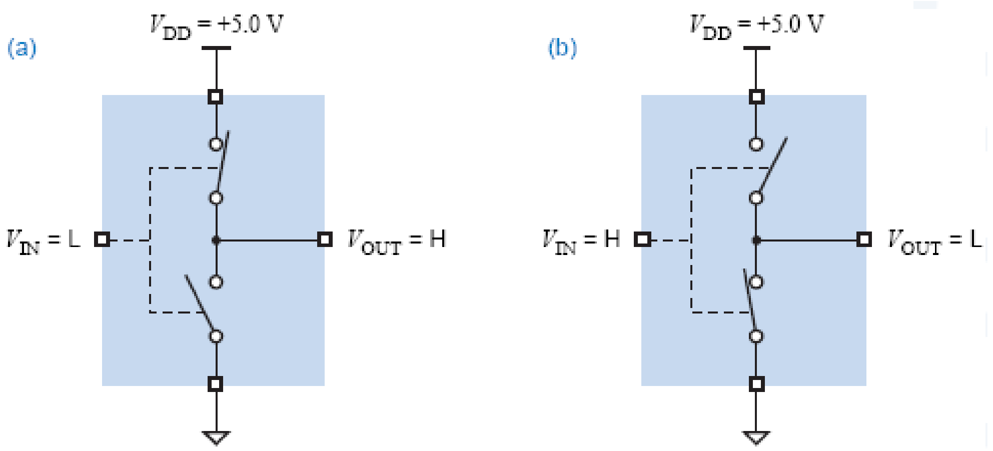
CMOS Inverter



V_{IN}	Q1	Q2	V_{OUT}
0.0 (L)	off	on	5.0 (H)
5.0 (H)	on	off	0.0 (L)

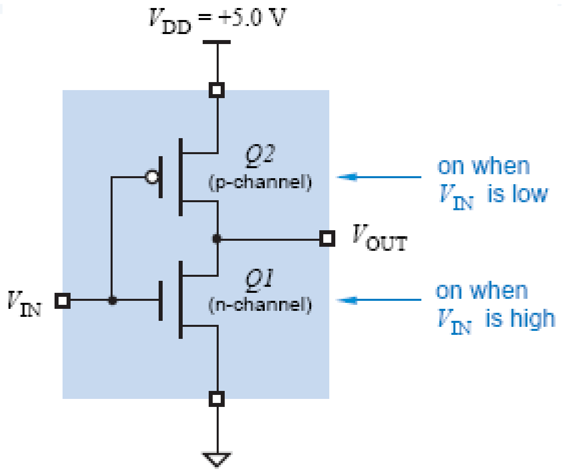
2. V_{IN} is 5.0 V. Here, *Q1* is on, since its V_{gs} is a large positive value (+5.0 V), but *Q2* is off, since its V_{gs} is 0. Thus, *Q1* presents a small resistance between the output terminal and ground, and the output voltage is 0 V. With the foregoing functional behavior, the circuit clearly behaves as a logical inverter, since a 0-volt input produces a 5-volt output, and vice versa Another way to visualize CMOS operation uses switches. As shown in Figure, the *n*-channel (bottom) transistor is modeled by a normally-open switch, and the *p*-channel (top) transistor by a normally-closed switch. Applying a HIGH voltage changes each switch to the opposite of its normal state, as shown in (b).

Switch model for CMOS inverter



The switch model gives rise to a way of drawing CMOS circuits that makes their logical behavior more readily apparent. As shown in Figure, different symbols are used for the *p*- and *n*-channel transistors to reflect their logical behavior. The *n*-channel transistor (*Q1*) is switched "on," and current flows between source and drain, when a HIGH voltage is applied to its gate; this seems natural enough. The *p*-channel transistor (*Q2*) has the opposite behavior. It is "on" when a LOW voltage is applied; the inversion bubble on its gate indicates this inverting behavior.

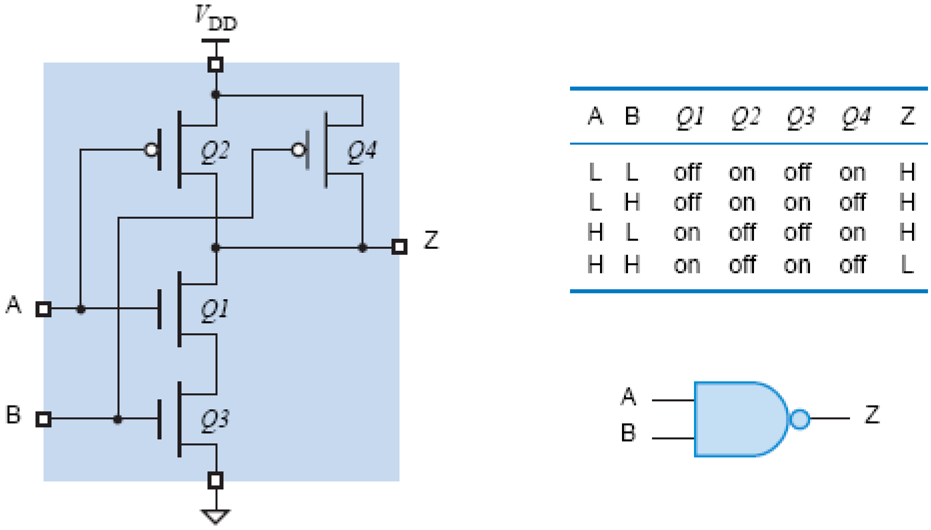
CMOS inverter Operation



CMOS NAND and NOR Gates

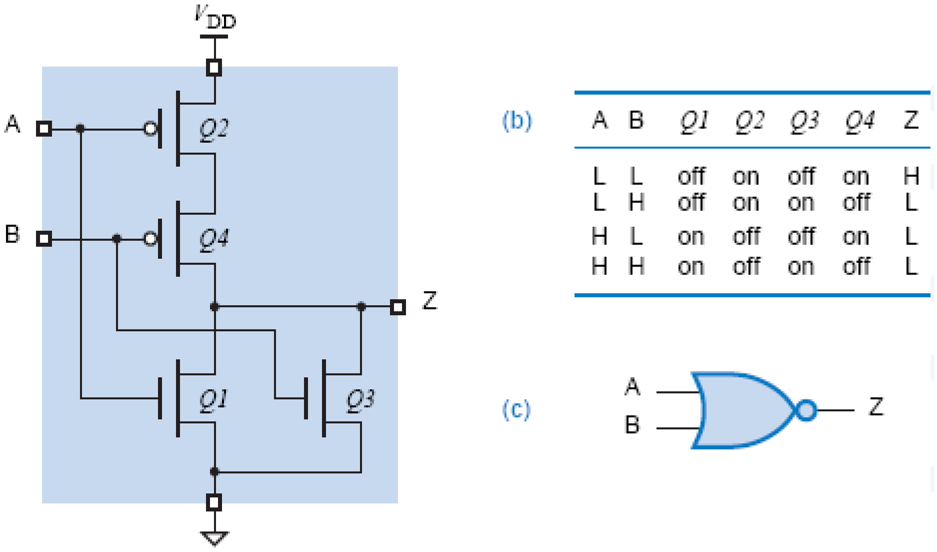
Both NAND and NOR gates can be constructed using CMOS. A *k*-input gate uses *k p*-channel and *k n*-channel transistors. Figure shows a 2-input CMOS NAND gate. If either input is LOW, the output Z has a low-impedance connection to *V*DD through the corresponding "on" *p*-channel transistor, and the path to ground is blocked by the corresponding "off" *n*-channel transistor. If both inputs are HIGH, the path to *V*DD is blocked, and Z has a low-impedance connection to ground. Figure shows the switch model for the NAND gate's operation. Figure shows a CMOS NOR gate. If both inputs are LOW, the output Z has a low-impedance connection to *V*DD through the "on" *p*-channel transistors, and the path to ground is blocked by the “off” *n*-channel transistors. If either input is HIGH, the path to *V*DD is blocked, and Z has a low-impedance connection to ground.

CMOS 2 input NAND gate



A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	H
H	L	on	off	off	on	H
H	H	on	off	on	off	L

CMOS 2 input NOR gate

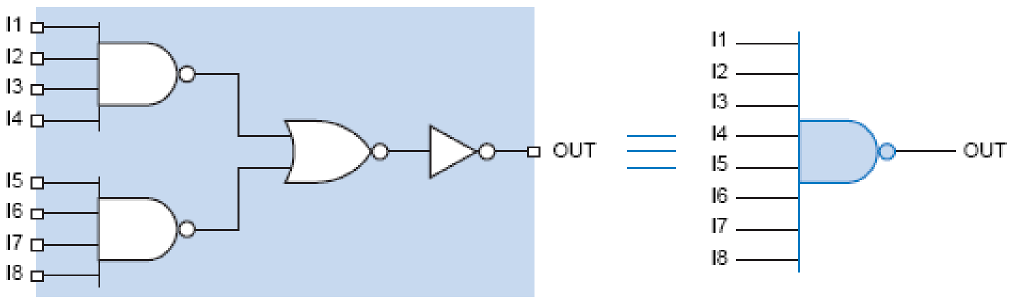


A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	L
H	L	on	off	off	on	L
H	H	on	off	on	off	L

Fan-In

The number of inputs that a gate can have in a particular logic family is called the logic family’s *fan-in*. CMOS gates with more than two inputs can be obtained by extending series-parallel designs on Figures in the obvious manner. For example, Figure shows a 3-input CMOS NAND gate. In principle, you could design a CMOS NAND or NOR gate with a very large number of inputs. In practice, however, the additive “on” resistance of series transistors limits the fan-in of CMOS gates, typically to 4 for NOR gates and 6 for NAND gates.

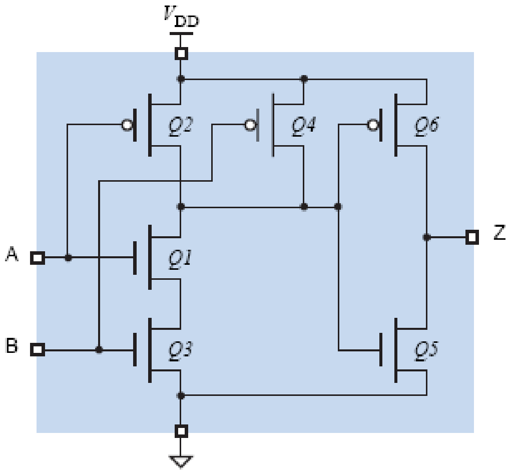
8 input CMOS NAND gate

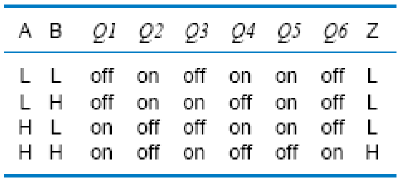


Non inverting Gates

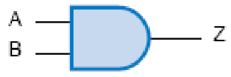
In CMOS, and in most other logic families, the simplest gates are inverters, and the next simplest are NAND gates and NOR gates. A logical inversion comes “for free,” and it typically is not possible to design a non inverting gate with a smaller number of transistors than an inverting one. CMOS non inverting buffers and AND and OR gates are obtained by connecting an inverter to the output of the corresponding inverting gate. Combining Figure with an inverter yields an OR gate. Thus, Figure shows a non inverting buffer and Figure shows an AND gate.

CMOS 2 input AND gate





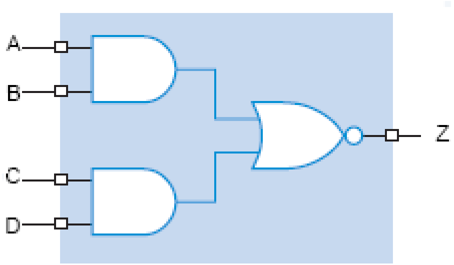
A	B	Q1	Q2	Q3	Q4	Q5	Q6	Z
L	L	off	on	off	on	on	off	L
L	H	off	on	on	off	on	off	L
H	L	on	off	off	on	on	off	L
H	H	on	off	on	off	off	on	H



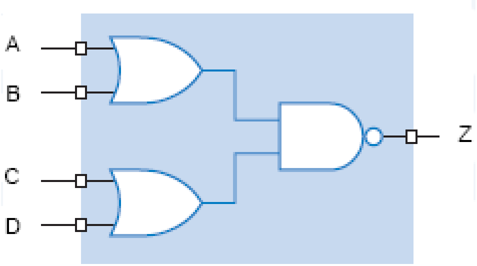
CMOS AND-OR-INVERT and OR-AND-INVERT Gates

CMOS circuits can perform two levels of logic with just a single “level” of transistors. For example, the circuit in Figure is a two-wide, two-input CMOS AND-OR-INVERT (AOI) *gate*. The function table for this circuit is shown in figure and a logic diagram for this function using AND and NOR gates is shown in Figure. Transistors can be added to or removed from this circuit to obtain an AOI function with a different number of ANDs or a different number of inputs per AND. The contents of each of the *Q1–Q8* columns in Figure depends only on the input signal connected to the corresponding transistor’s gate. The last column is constructed by examining each input combination and determining whether Z is connected to *V*DD or ground by “on” transistors for that input combination. Note that Z is never connected to *both V*DD and ground for any input combination; in such a case the output would be a non-logic value some- where between LOW and HIGH, and the output structure would consume excessive power due to the low-impedance connection between *V*DD and ground. A circuit can also be designed to perform an OR-AND-INVERT function. For example, Figure is a two-wide, two-input CMOS OR-AND-INVERT (OAI) *gate*. The function table for this circuit is shown in figure, the values in each column are determined just as we did for the CMOS AOI gate. A logic diagram for the OAI function using OR and NAND gates is shown in Figure. The speed and other electrical characteristics of a CMOS AOI or OAI gate are quite comparable to those of a single CMOS NAND or NOR gate. As a result, these gates are very appealing because they can perform two levels of logic (AND-OR or OR-AND) with just one level of delay. Most digital designers don't bother to use AOI gates in their discrete designs. However, CMOS VLSI devices often use these gates internally, since many HDL synthesis tools can automatically convert AND/OR logic into AOI gates when appropriate.

CMOS AOI GAte



CMOS AOI GAte



CMOS Steady-State Electrical Behavior

This section discusses the steady-state behavior of CMOS circuits, that is, the circuits' behavior when inputs and outputs are not changing. The next section discusses dynamic behavior, including speed and power dissipation.

Logic Levels and Noise Margins

The table in Figure defined the CMOS inverter's behavior only at two discrete input voltages; other input voltages may yield different output voltages. The complete input-output transfer characteristic can be described by a graph such as Figure . In this graph, the input voltage is varied from 0 to 5 V, as shown on the X axis; the Y axis plots the output voltage. If we believed the curve in Figure , we could define a CMOS LOW input level as any voltage under 2.4 V, and a HIGH input level as anything over 2.6 V. Only when the input is between 2.4 and 2.6 V does the inverter produce a non logic output voltage under this definition. Unfortunately, the typical transfer characteristic shown in Figure is just that—typical, but not guaranteed. It varies greatly under different conditions of power supply voltage, temperature, and output loading. The transfer characteristic may even vary depending on when the device was fabricated. For example, after months of trying to figure out why gates made on some days were good and on other days were bad, one manufacturer discovered that the bad gates were victims of airborne contamination by a particularly noxious perfume worn by one of its production-line workers! Sound engineering practice dictates that we use more conservative specifications for LOW and HIGH. The conservative specs for a typical CMOS logic family (HC-series) are depicted in Figure.

*V*OHmin The minimum output voltage in the HIGH state.

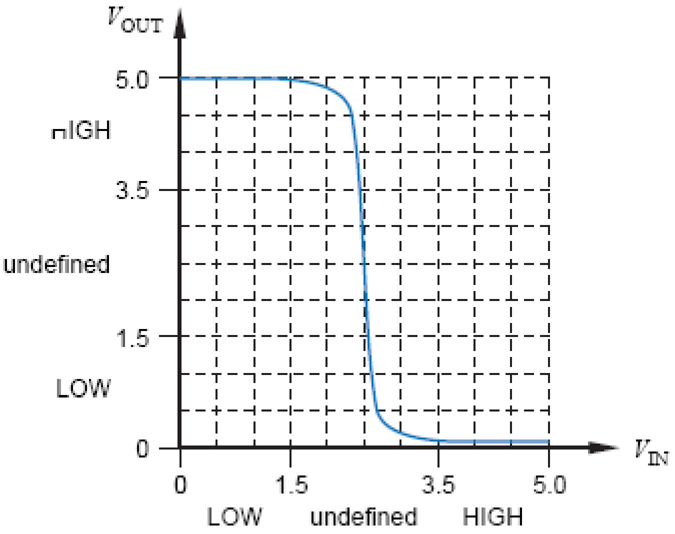
*V*IHmin The minimum input voltage guaranteed to be recognized as a HIGH.

*V*ILmax The maximum input voltage guaranteed to be recognized as a LOW.

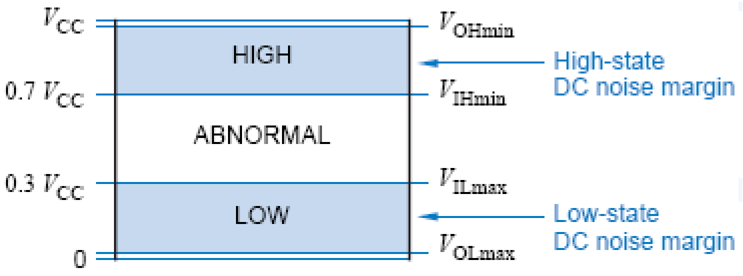
*V*OLmax The maximum output voltage in the LOW state.

The input voltages are determined mainly by switching thresholds of the two transistors, while the output voltages are determined mainly by the "on" resistance of the transistors.

Input oput characteristics of CMOS inverter



Logic levels and Noise Margin



The power-supply voltage *V*CC and ground are often called the *powersupply rails*. CMOS levels are typically a function of the power-supply rails:

*V*OHmin *V*CC □□0.1 V

*V*IHmin 70% of *V*CC

*V*ILmax 30% of *V*CC

*V*OLmax ground + 0.1 V

DC noise margin is a measure of how much noise it takes to corrupt a worst-case output voltage into a value that may not be recognized properly by an input. For HC-series CMOS in the LOW state, *V*ILmax (1.35 V) exceeds *V*OLmax (0.1 V) by 1.25 V so the LOW-state DC noise margin is 1.25 V. Likewise, there is DC noise margin of 1.25 V in the HIGH state. In general, CMOS outputs have excellent DC noise margins when driving other CMOS inputs. Regardless of the voltage applied to the input of a CMOS inverter, the input consumes very little current, only the leakage current of the two transistors' gates. The maximum amount of current that can flow is also specified by the device manufacturer:

*I*IH The maximum current that flows into the input in the LOW state.

*I*IL The maximum current that flows into the input in the HIGH state.

Effects of Loading

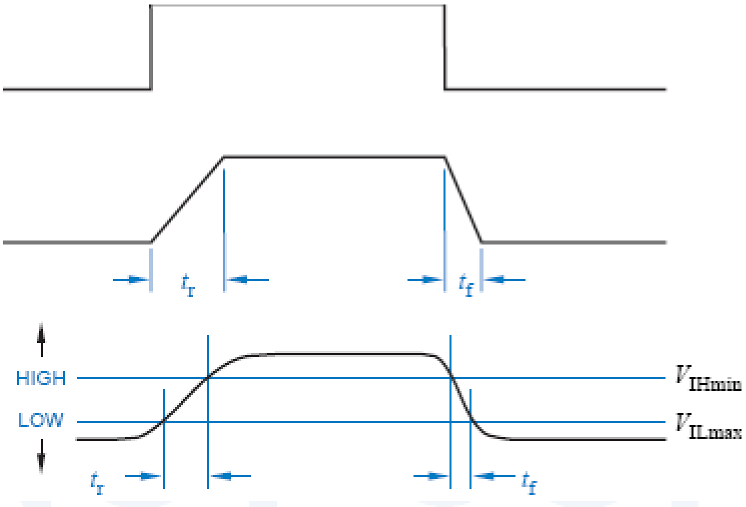
Loading an output beyond its rated fanout has several effects:

- In the LOW state, the output voltage (*V*OL) may increase beyond *V*OLmax.
- In the HIGH state, the output voltage (*V*OH) may fall below *V*OHmin.
- Propagation delay to the output may increase beyond specifications.
- Output rise and fall times may increase beyond their specifications.
- The operating temperature of the device may increase, thereby reducing the reliability of the device and eventually causing device failure.

CMOS Dynamic Electrical Behavior

Both the speed and the power consumption of a CMOS device depend to a large extent on AC or dynamic characteristics of the device and its load, that is, what happens when the output changes between states. As part of the internal design of CMOS ASICs, logic designers must carefully examine the effects of output loading and redesign where the load is too high. Even in board-level design, the effects of loading must be considered for clocks, buses, and other signals that have high fanout or long interconnections. Speed depends on two characteristics, transition time and propagation delay, discussed in the next two subsections. Power dissipation is discussed in the third subsection.

Transition Times



Transition Time

The amount of time that the output of a logic circuit takes to change from one state to another is called the *transition time*. Figure shows how we might like outputs to change state—in zero time. However, real outputs cannot change instantaneously, because they need time to charge the stray capacitance of the wires and other components that they drive. A more realistic view of a circuit's output is shown in figure An output takes a certain time, called the *rise time* (*t*r), to change from LOW to HIGH, and a possibly different time, called the *fall time (t*f), to change from HIGH to LOW. To avoid difficulties in defining the endpoints, rise and fall times are normally measured at the boundaries of the valid logic levels as indicated in the figure.

The rise and fall times indicate how long an output voltage takes to pass through the "undefined" region between LOW and HIGH. The initial part of a transition is not included in the rise- or fall-time number. Instead, the initial part of a transition contributes to the "propagation delay" number discussed in the next subsection. The rise and fall times of a CMOS output depend mainly on two factors, the "on" transistor resistance and the load capacitance. A large capacitance increases transition times; since this is undesirable, it is very rare for a logic designer to purposely connect a capacitor to a logic circuit's output. However, *stray capacitance* is present in every circuit; it comes from at least three sources: 1. Output circuits, including a gate's output transistors, internal wiring, and packaging, have some capacitance associated with them, on the order of picofarads (pF) in typical logic families, including CMOS. 2. The wiring that connects an output to other inputs has capacitance, about 1 pF per inch or more, depending on the wiring technology. 3. Input circuits, including transistors, internal wiring, and packaging, have capacitance, from 2 to 15 pF per input in typical logic families. Stray capacitance is sometimes called a *capacitive load* or an *AC load*. A CMOS output's rise and fall times can be analyzed using the equivalent circuit shown in Figure. As in the preceding section, the *p*-channel and *n*channel transistors are modeled by resistances *R*p and *R*n, respectively. In normal operation, one resistance is high and the other is low, depending on the output's state. The output's load is modeled by an *equivalent load circuit* with three components:

*R*L, *V*L These two components represent the DC load and determine the voltages and currents that are present when the output has settled into a stable HIGH or LOW state. The DC load doesn't have too much effect on transition times when the output changes states. *C*L This capacitance represents the AC load and determines the voltages and currents that are present while the output is changing, and how long it takes to change from one state to the other

Propagation Delay

Rise and fall times only partially describe the dynamic behavior of a logic element; we need additional parameters to relate output timing to input timing. A *signal path* is the electrical path from a particular input signal to a particular output signal of a logic element. The *propagation delay t*p of a signal path is the amount of time that it takes for a change in the input signal to produce a change in the output signal. A complex logic lement with multiple inputs and outputs may specify a different value of *t*p for each different signal path. Also, different values may be specified for a particular signal path, depending on the direction of the output change. Ignoring rise and fall times, Figure shows two different propagation delays for the input-to-output signal path of a CMOS inverter, depending on the direction of the output change:

*t*pHL The time between an input change and the corresponding output change

when the output is changing from HIGH to LOW.

*t*pLH The time between an input change and the corresponding output change

when the output is changing from LOW to HIGH.

Several factors lead to nonzero propagation delays. In a CMOS device, the rate at which transistors change state is influenced both by the semiconductor physics of the device and by the circuit environment, including input-signal transition rate, input capacitance, and output loading. Multistage devices such as non inverting gates or more complex logic functions may require several internal transistors to change state before the output can change state. And even when the output begins to change state, with nonzero rise and fall times it takes quite some time to cross the region between states, as we showed in the preceding subsection. All of these factors are included in propagation delay. To factor out the effect of rise and fall times, manufacturers usually specify propagation delays at the midpoints of input and output transitions, as shown in Figure. However, sometimes the delays are specified at the logic-level boundary points, especially if the device's operation may be adversely affected by slow rise and fall times.

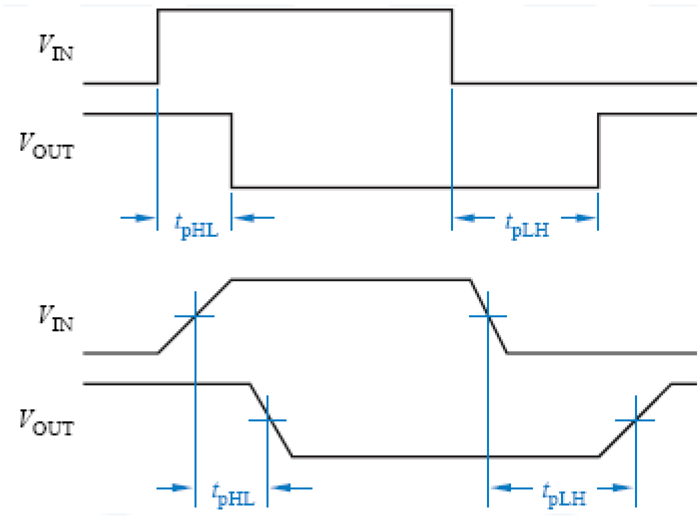
*t*pHL The time between an input change and the corresponding output change when the output is changing from HIGH to LOW.

*t*pLH The time between an input change and the corresponding output change when the output is changing from LOW to HIGH.

Several factors lead to nonzero propagation delays. In a CMOS device, the rate at which transistors change state is influenced both by the semiconductor physics of the device and by the circuit environment, including input-signal transition rate, input capacitance, and output loading. Multistage devices such as noninverting gates or more complex logic functions may require several internal transistors to change state before the output can change state. And even when the output begins to change state, with nonzero rise and fall times it takes quite some time to cross the region between states, as we showed in the preceding subsection. All of these factors are included in propagation delay.

To factor out the effect of rise and fall times, manufacturers usually specify propagation delays at the midpoints of input and output transitions, as shown in Figure. However, sometimes the delays are specified at the logic-level boundary points, especially if the device's operation may be adversely affected by slow rise and fall times.

Propagation Delay



Power Consumption

The power consumption of a CMOS circuit whose output is not changing is called *static power dissipation* or *quiescent power dissipation*. (The words *consumption* and *dissipation* are used pretty much interchangeably when discussing how much power a device uses.) Most CMOS circuits have very low static power dissipation. This is what makes them so attractive for laptop computers and other low-power applications—when computation pauses, very little power is consumed. A CMOS circuit consumes significant power only during transitions; this is called *dynamic power dissipation*. One source of dynamic power dissipation is the partial short-circuiting of the CMOS output structure. When the input voltage is not close to one of the power supply rails (0 V or *V*CC), both the *p*-channel and *n*-channel output transistors may be partially "on," creating a series resistance of 600 ohms or less. In this case, current flows through the transistors from *V*CC to ground. The amount of power consumed in this way depends on both the value of *V*CC and the rate at which output transitions occur, according to the formula pt=cpd*vcc*vcc*f.

The following variables are used in the formula:

*P*T The circuit's internal power dissipation due to output transitions.

*V*CC The power supply voltage. As all electrical engineers know, power dissipation across a resistive load (the partially-on transistors) is proportional to the *square* of the voltage.

f The *transition frequency* of the output signal. This specifies the number of power-consuming output transitions per second. (But note that frequency is defined as the number of transitions divided by 2.)

CPD The *power dissipation capacitance*. This constant, normally specified by the device manufacturer, completes the formula. *C*PD turns out to have units of capacitance, but does not represent an actual output capacitance. Rather, it embodies the dynamics of current flow through the changing output-transistor resistances during a single pair of output transitions, HIGH-to-LOW and LOW-to-HIGH. For example, *C*PD for HC-series CMOS gates is typically 20–24 pF, even though the actual output capacitance is much less.

The *P*T formula is valid only if input transitions are fast enough, leading to fast output transitions. If the input transitions are too slow, then the output transistors stay partially on for a longer time, and power consumption increases. Device manufacturers usually recommend a maximum input rise and fall time, below which the value specified for *C*PD is valid.

A second, and often more significant, source of CMOS power consumption is the capacitive load (*C*L) on the output. During a LOW-to-HIGH transition, current flows through a *p*-channel transistor to charge *C*L. Likewise, during a HIGH-to-LOW transition, current flows through an *n*-channel transistor to discharge *C*L. In each case, power is dissipated in the "on" resistance of the transistor. We'll use *P*L to denote the total amount of power dissipated by charging and discharging *C*L. The units of *P*L are power, or energy usage per unit time. During a transition, the voltage across the load capacitance *C*L changes by ±*V*CC. According to the definition of capacitance, the total amount of charge that must flow to make a voltage change of *V*CC across *C*L is . The total amount of energy used in one transition is charge times the average voltage change. The first little bit of charge makes a voltage change of *V*CC, while the last bit of charge makes a vanishingly small voltage change, hence the average change is *V*CC/2. The total energy per transition is therefore . If there are 2*f* transitions per second, the total power dissipated due to the capacitive load is pl=cl*vcc*vcc*f Based on this formula, dynamic power dissipation is often called *CV*2*f power*. In most applications of CMOS circuits, *CV*2*f* power is by far the major contributor to total power dissipation. Note that *CV*2*f* power is also consumed by bipolar logic circuits like TTL and ECL, but at low to moderate frequencies it is insignificant compared to the static (DC or quiescent) power dissipation of bipolar circuits.

CMOS Logic Families

The first commercially successful CMOS family was *4000-series CMOS*. Although 4000-series circuits offered the benefit of low power dissipation, they were fairly slow and were not easy to interface with the most popular logic family of the time, bipolar TTL. Thus, the 4000 series was supplanted in most applications by the more capable CMOS families discussed in this section. All of the CMOS devices that we discuss have part numbers of the form "74FAM*nn*," where "FAM" is an alphabetic family mnemonic and *nn* is a numeric function designator. Devices in different families with the same value of *nn* perform the same function. For example, the 74HC30, 74HCT30, 74AC30, 74ACT30, and 74AHC30 are all 8-input NAND gates. The prefix "74" is simply a number that was used by an early, popular supplier of TTL devices, Texas Instruments. The prefix "54" is used for identical parts that are specified for operation over a wider range of temperature and power-supply voltage, for use in military applications. Such parts are usually fabricated in the same way as their 74-series counterparts, except that they are tested, screened, and marked differently, a lot of extra paperwork is generated, and a higher price is charged, of course.

HC and HCT

The first two 74-series CMOS families are *HC (High-speed CMOS)* and *HCT (High-speed CMOS, TTL compatible).* Compared with the original 4000 family, HC and HCT both have higher speed and better current sinking and sourcing capability. The HCT family uses a power supply voltage *V*CC of 5 V and can be intermixed with TTL devices, which also use a 5-V supply. The HC family is optimized for use in systems that use CMOS logic exclusively, and can use any power supply voltage between 2 and 6 V. A higher voltage is used for higher speed, and a lower voltage for lower power dissipation. Lowering the supply voltage is especially effective, since most CMOS power dissipation is proportional to the square of the voltage (*CV*2*f* power). Even when used with a 5-V supply, HC devices are not quite compatible with TTL. In particular, HC circuits are designed to recognize CMOS input levels.

VHC and VHCT

Several new CMOS families were introduced in the 1980s and the 1990s. Two of the most recent and probably the most versatile are *VHC (Very High-Speed CMOS)* and *VHCT (Very High-Speed CMOS, TTL compatible)*. These families are about twice as fast as HC/HCT while maintaining backwards compatibility with their predecessors. Like HC and HCT, the VHC and VHCT families differ from each other only in the input levels that they recognize; their output characteristics are the same. Also like HC/HCT, VHC/VHCT outputs have *symmetric output drive*. That is, an output can sink or source equal amounts of current; the output is just as “strong” in both states. Other logic families, including the FCT and TTL families introduced later, have *asymmetric output drive;* they can sink much more current in the LOW state than they can source in the HIGH state.

HC, HCT, VHC, and VHCT Electrical Characteristics

Electrical characteristics of the HC, HCT, VHC, and VHCT families are summarized in this subsection. The specifications assume that the devices are used with a nominal 5-V power supply, although (derated) operation is possible with any supply voltage in the range 2–5.5 V (up to 6 V for HC/HCT). Specifications for a 74x00 two-input NAND gate and a 74x138 3-to-8 decoder in the HC, HCT, VHC, and VHCT families. The ’00 NAND gate is included as the smallest logic-design building block in each family, while the ’138 is a “medium-scale” part containing the equivalent of about 15 NAND gates.

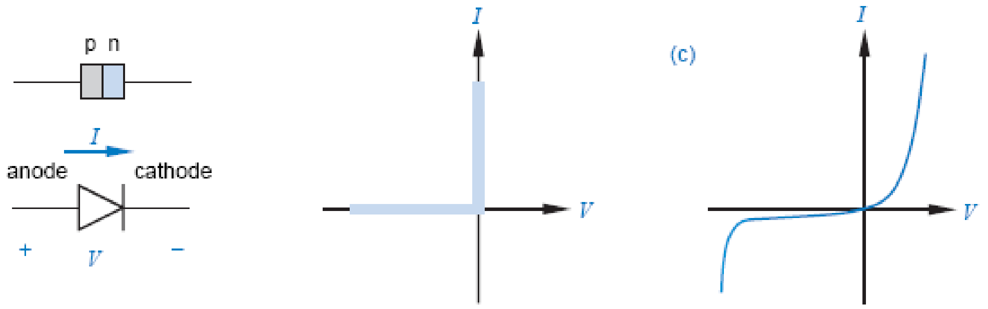
Bipolar Logic

Bipolar logic families use semiconductor diodes and bipolar junction transistors as the basic building blocks of logic circuits. The simplest bipolar logic elements use diodes and resistors to perform logic operations; this is called diode logic. Most TTL logic gates use diode logic internally and boost their output drive capability using transistor circuits. Some TTL gates use parallel configurations of transistors to perform logic functions.. This section covers the basic operation of bipolar logic circuits made from diodes and transistors, and the next section covers TTL circuits in detail. Although TTL is the most commonly used bipolar logic family, it has been largely supplanted by the CMOS families that we studied in previous sections. Also, an understanding of TTL may give you insight into the fortuitous similarity of logic levels that allowed the industry to migrate smoothly from TTL to 5-V CMOS logic, and now to lower-voltage, higher-performance 3.3-V CMOS logic, as described in Section 3.13. If you're not interested in all the gory details of TTL, you can skip to Section 3.11 for an overview of TTL families.

Diodes

A *semiconductor diode* is fabricated from two types of semiconductor material, called *p*-type and *n*-type, that are brought into contact with each other as shown in Figure . This is basically the same material that is used in *p*-channel and *n*-channel MOS transistors. The point of contact between the *p* and *n* materials is called a *pn junction.* (Actually, a diode is normally fabricated from a single monolithic crystal of semiconductor material in which the two halves are "doped" with different impurities to give them *p*-type and *n*-type properties.) The physical properties of a *pn* junction are such that positive current can easily flow from the *p*-type material to the *n*-type. Thus, if we build the circuit shown in Figure the *pn* junction acts almost like a short circuit. However, the physical properties also make it very difficult for positive current to flow in the opposite direction, from *n* to *p*. Thus, in the circuit of Figure , the *pn* junction behaves almost like an open circuit. This is called *diode action*. Although it's possible to build vacuum tubes and other devices that exhibit diode action, modern systems use *pn* junctions—semiconductor diodes—which we'll henceforth call simply *diodes.* Figure shows the schematic symbol for a diode. As we've shown, in normal operation significant amounts of current can flow only in the direction indicated by the two arrows, from *anode* to *cathode*. In effect, the diode acts like a short circuit as long as the voltage across the anode-to-cathode junction is nonnegative. If the anode-to-cathode voltage is negative, the diode acts like an open circuit and no current flows. The transfer characteristic of an ideal diode shown in Figure further illustrates this principle. If the anode-to-cathode voltage, *V*, is negative, the diode is said to be *reverse biased* and the current *I* through the diode is zero. If *V* is nonnegative, the diode is said to be *forward biased* and *I* can be an arbitrarily large positive value. In fact, *V* can never get larger than zero, because an ideal diode acts like a zero-resistance short circuit when forward biased.

Diode characteristics



Transistor-Transistor Logic

The most commonly used bipolar logic family is transistor-transistor logic. Actually, there are many different TTL families, with a range of speed, power consumption, and other characteristics. The circuit examples in this section are based on a representative TTL family, Low-power Schottky (LS or LS-TTL). TTL families use basically the same logic levels as the TTL-compatible CMOS families in previous sections. We'll use the following definitions of LOW and HIGH in our discussions of TTL circuit behavior: LOW 0–0.8 volts. HIGH 2.0–5.0 volts.

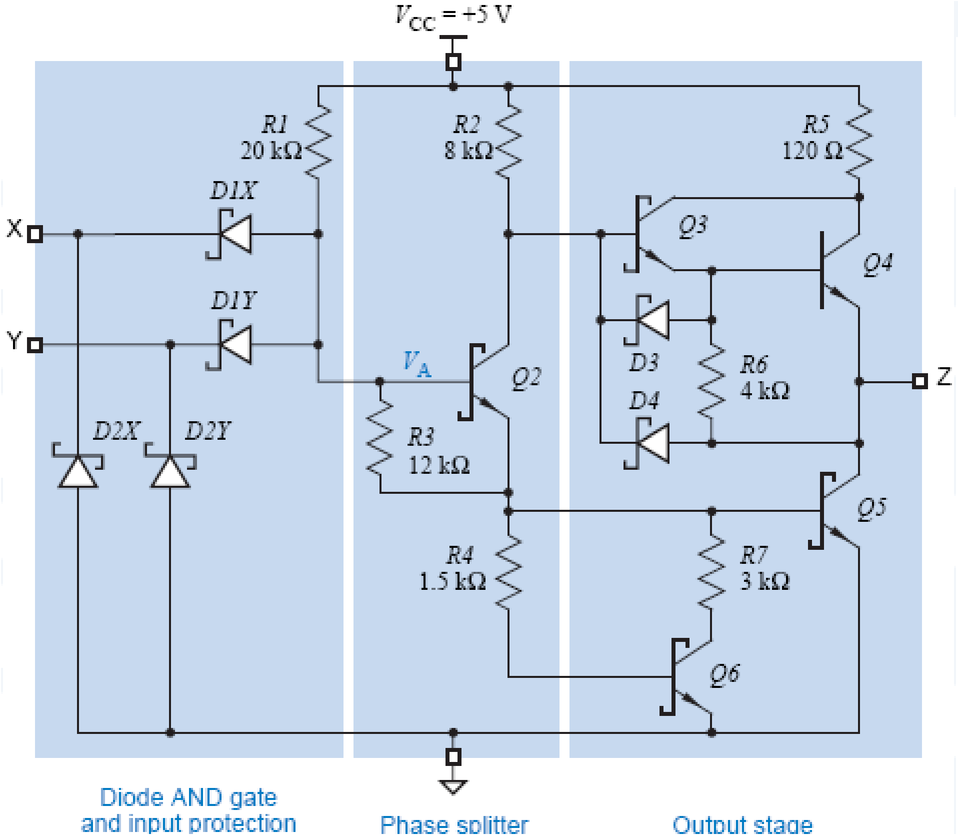
Basic TTL NAND **Gate**

The circuit diagram for a two-input LS-TTL NAND gate, part number 74LS00, is shown in Figure 3-75. The NAND function is obtained by combining a diode AND gate with an inverting buffer amplifier. The circuit's operation is best understood by dividing it into the three parts that are shown in the figure and discussed in the next three paragraphs:

- Diode AND gate and input protection.
- Phase splitter.
- Output stage.

Diodes *D1X* and *D1Y* and resistor *R1* in Figure 3-75 form a *diode* AND *gate. Clamp diodes D2X* and *D2Y* do nothing in normal operation, but limit undesirable negative excursions on the inputs to a single diode drop. Such negative excursions may occur on HIGH-to-LOW input transitions as a result of transmission-line effects. Transistor *Q2* and the surrounding resistors form a *phase splitter* that controls the output stage. Depending on whether the diode AND gate produces a "low" or a "high" voltage at *V*A, *Q2* is either cut off or turned on.

2 input TTL NAND gate



The *output stage* has two transistors, *Q4* and *Q5*, only one of which is on at any time. The TTL output stage is sometimes called a *totem-pole* or *push-pull output*. Similar to the *p*-channel and *n*-channel transistors in CMOS, *Q4* and *Q5* provide active pull-up and pull-down to the HIGH and LOW states, respectively. TTL NAND gates can be designed with any desired number of inputs simply by changing the number of diodes in the diode AND gate in the figure. Commercially available TTL NAND gates have as many as 13 inputs. A TTL inverter is designed as a 1-input NAND gate, omitting

diodes *D1Y* and *D2Y* in Figure Since the output transistors *Q4* and *Q5* are normally complementary—one ON and the other OFF—you might question the purpose of the 120 ohm resistor *R5* in the output stage. A value of 0 would give even better driving capability in the HIGH state. This is certainly true from a DC point of view. However, when the TTL output is changing from HIGH to LOW or vice versa, there is a short time when both transistors may be on. The purpose of *R5* is to limit the amount of current that flows from VCC to ground during this time. Even with a 120 ohm resistor in the TTL output stage, higher-than-normal currents called current spikes flow when TTL outputs are switched. These are similar to the current spikes that occur when high-speed CMOS outputs switch. So far we have shown the input signals to a TTL gate as ideal voltage sources. Figure shows the situation when a TTL input is driven LOW by the output of another TTL gate. Transistor *Q5A* in the driving gate is ON, and thereby provides a path to ground for the current flowing out of the diode *D1XB* in the driven gate. When current flows *into* a TTL output in the LOW state, as in this case, the output is said to be *sinking current*. Figure shows the same circuit with a HIGH output. In this case, *Q4A* in the driving gate is turned on enough to supply the small amount of leakage current flowing through reverse-biased diodes

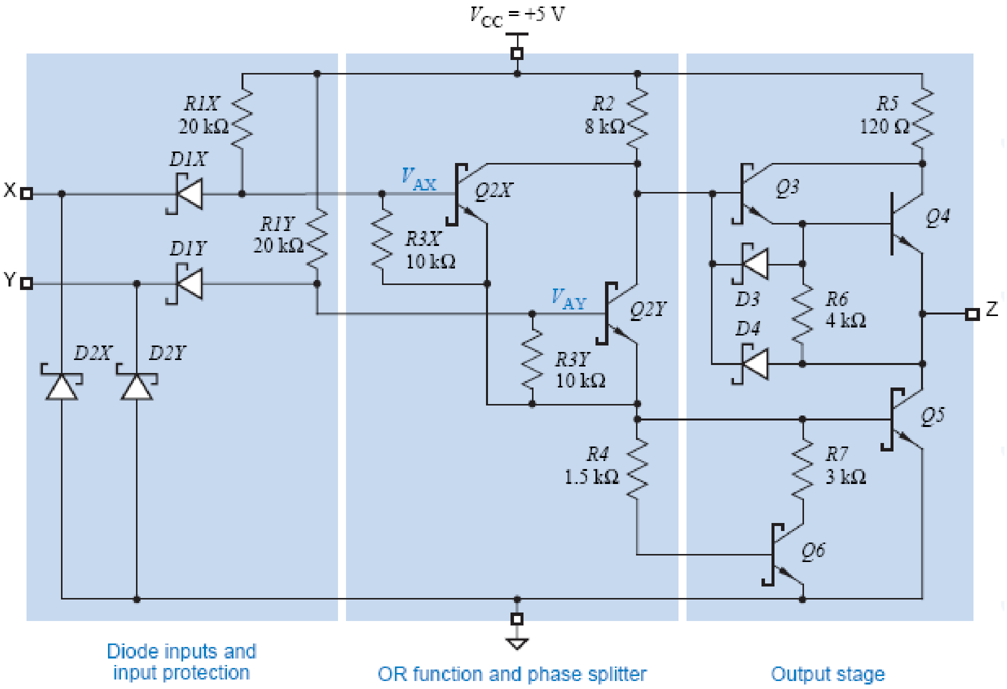
D1XB and *D2XB* in the driven gate. When current flows out of a TTL output in the HIGH state, the output is said to be *sourcing current*.

Additional TTL Gate Types

Although the NAND gate is the “workhorse” of the TTL family, other types of gates can be built with the same general circuit structure. The circuit diagram for an LS-TTL NOR gate is shown in Figure . If either input X or Y is HIGH, the corresponding phase-splitter transistor *Q2X* or *Q2Y* is turned on, which turns off *Q3* and *Q4* while turning on *Q5* and *Q6*, and the output is LOW. If both inputs are LOW, then both phase-splitter transistors are off, and the output is forced HIGH. This functional operation is summarized in Figure .The LS-TTL NOR gate’s input circuits, phase splitter, and output stage are almost identical to those of an LS-TTL NAND gate. The difference is that an LSTTL NAND gate uses diodes to perform the AND function, while an LS-TTL

NOR gate uses parallel transistors in the phase splitter to perform the OR function. The speed, input, and output characteristics of a TTL NOR gate are comparable to those of a TTL NAND. However, an *n*-input NOR gate uses more transistors and resistors and is thus more expensive in silicon area than an *n*input NAND. Also, internal leakage current limits the number of *Q2* transistors that can be placed in parallel, so NOR gates have poor fan-in. (The largest discrete TTL NOR gate has only 5 inputs, compared with a 13-input NAND.) As a result, NOR gates are less commonly used than NAND gates in TTL designs. The most “natural” TTL gates are inverting gates like NAND and NOR. Noninverting TTL gates include an extra inverting stage, typically between the input stage and the phase splitter. As a result, noninverting TTL gates are typically larger and slower than the inverting gates on which they are based. Like CMOS, TTL gates can be designed with three-state outputs. Such gates have an “output-enable” or “output-disable” input that allows the output to be placed in a high-impedance state where neither output transistor is turned on.

2 input TTL NOR Gate



TTL Families

TTL families have evolved over the years in response to the demands of digital designers for better performance. As a result, three TTL families have come and gone, and today's designers have five surviving families from which to choose. All of the TTL families are compatible in that they use the same power supply voltage and logic levels, but each family has its own advantages in terms of speed, power consumption, and cost.

Early TTL Families

The original TTL family of logic gates was introduced by Sylvania in 1963. It was popularized by Texas Instruments, whose "7400-series" part numbers for gates and other TTL components quickly became an industry standard. As in 7400-series CMOS, devices in a given TTL family have part numbers of the form 74FAM*nn*, where "FAM" is an alphabetic family mnemonic and *nn* is a numeric function designator. Devices in different families with the same value of *nn* perform the same function. In the original TTL family, "FAM" is null and the family is called *74-series TTL*. Resistor values in the original TTL circuit were changed to obtain two more TTL families with different performance characteristics. The *74H (Highspeed TTL)* family used lower resistor values to reduce propagation delay at the expense of increased power consumption. The *74L (Low-power TTL)* family used higher resistor values to reduce power consumption at the expense of propagation delay. The availability of three TTL families allowed digital designers in the

1970s to make a choice between high speed and low power consumption for their circuits. However, like many people in the 1970s, they wanted to "have it all, now." The development of Schottky transistors provided this opportunity, and made 74, 74H, and 74L TTL obsolete.

Schottky TTL Families

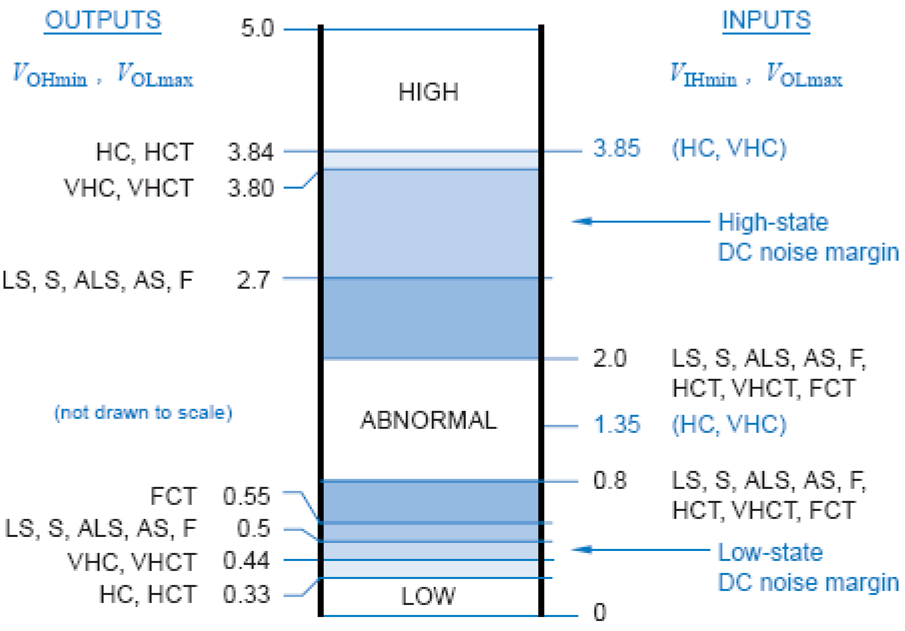
Historically, the first family to make use of Schottky transistors was *74S (Schottky* consumption. The *74ALS (Advanced Low-power Schottky TTL)* family offers both lower power and higher speeds than 74LS, and rivals 74LS in popularity for general-purpose requirements in new TTL designs. The *74F (Fast TTL)* family is positioned between 74AS and 74ALS in the speed/power tradeoff, and is probably the most popular choice for high-speed requirements in new TTL designs. *TTL)*. With Schottky transistors and low resistor values, this family has much higher speed, but higher power consumption, than the original 74-series TTL. Perhaps the most widely used and certainly the least expensive TTL family is *74LS (Low-power Schottky TTL)*, introduced shortly after 74S. By combining Schottky transistors with higher resistor values, 74LS TTL matches the speed of 74-series TTL but has about one-fifth of its power consumption. Thus, 74LS is a

preferred logic family for new TTL designs. Subsequent IC processing and circuit innovations gave rise to two more Schottky logic families. The *74AS (Advanced Schottky TTL)* family offers speeds approximately twice as fast as 74S with approximately the same power

CMOS/TTL Interfacing

A digital designer selects a “default” logic family to use in a system, based on general requirements of speed, power, cost, and so on. However, the designer may select devices from other families in some cases because of availability or other special requirements. (For example, not all 74LS part numbers are available in 74HCT, and vice versa.) Thus, it’s important for a designer to understand the implications of connecting TTL outputs to CMOS inputs, and vice versa. There are several factors to consider in TTL/CMOS interfacing, and the first is noise margin. The LOW-state DC noise margin depends on *V*OLmax of the driving output and *V*ILmax of the driven input, and equals *V*ILmax □□*V*OLmax. Similarly, the HIGH-state DC noise margin equals *V*OHmin □□*V*IHmin. Figure 3-84 shows the relevant numbers for TTL and CMOS families. In other words, TTL driving HC or AC doesn’t work, unless the TTL HIGH output happens to be higher and the CMOS HIGH input threshold happens to be lower by a total of 1.15 V compared to their worst-case specs. To drive CMOS inputs properly from TTL outputs, the CMOS devices should be HCT, VHCT. or FCT rather than HC or VHC. The next factor to consider is fanout. As with pure TTL, a designer must sum the input current requirements of devices driven by an output and compare with the output’s capabilities in both states. Fanout is not a problem when TTL drives CMOS, since CMOS inputs require almost no current in either state. On the other hand, TTL inputs, especially in the LOW state, require sub- stantial current, especially compared to HC and HCT output capabilities. For example, an HC or HCT output can drive 10 LS or only two S-TTL inputs. The last factor is capacitive loading. We’ve seen that load capacitance increases both the delay and the power dissipation of logic circuits. Increases in delay are especially noticeable with HC and HCT outputs, whose transition times increase about 1 ns for each 5 pF of load capacitance. The transistors in FCT outputs have very low “on” resistances, so their transition times increase only about 0.1 ns for each 5 pF of load capacitance. For a given load capacitance, power-supply voltage, and application, all of the CMOS families have similar dynamic power dissipation, since each variable in the *CV*2*f* equation is the same. On the other hand, TTL outputs have somewhat lower dynamic power dissipation, since the voltage swing between TTL HIGH and LOW levels is smaller.

O/p and i/p levels of CMOS TTL interfacing



Low-Voltage CMOS Logic and Interfacing

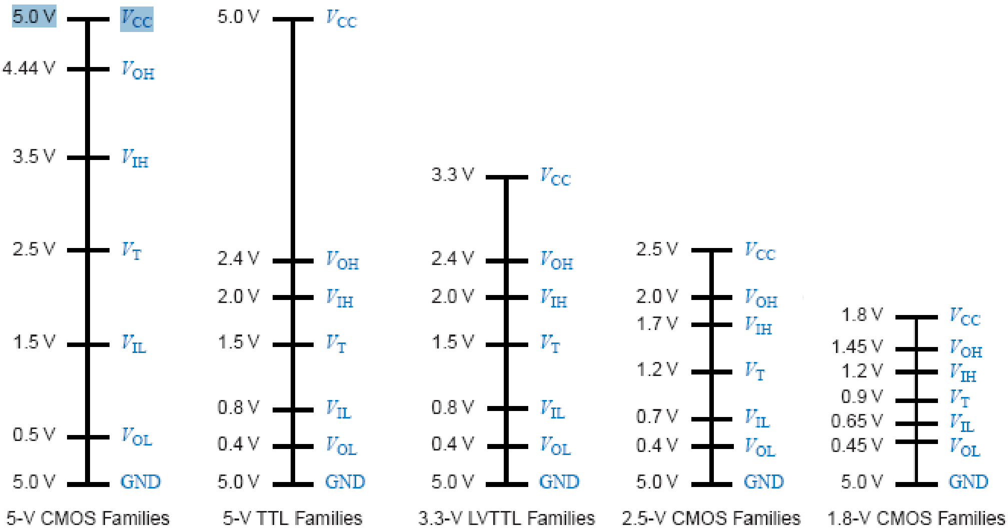
Two important factors have led the IC industry to move towards lower powersupply voltages in CMOS devices:

- In most applications, CMOS output voltages swing from rail to rail, so the *V* in the *CV*2*f* equation is the power-supply voltage. Cutting power-supply voltage reduces dynamic power dissipation more than proportionally.
- As the industry moves towards ever-smaller transistor geometries, the oxide insulation between a CMOS transistor's gate and its source and drain is getting ever thinner, and thus incapable of insulating voltage potentials as "high" as 5 V.

As a result, JEDEC, an IC industry standards group, selected 3.3V ± 0.3V, 2.5V ± 0.2V, and 1.8V± 0.15V as the next "standard" logic power-supply voltages. JEDEC standards specify the input and output logic voltage levels for devices operating with these power-supply voltages. The migration to lower voltages has occurred in stages, and will continue to do so. For discrete logic families, the trend has been to produce parts that operate and produce outputs at the lower voltage, but that can also tolerate inputs at the higher voltage. This approach has allowed 3.3-V CMOS families to operate with 5-V CMOS and TTL families, as we'll see in the next section Many ASICs and microprocessors have followed a similar approach, but another approach is often used as well. These devices are large enough that it can make sense to provide them with two power-supply voltages. A low voltage, such as 2.5 V, is supplied to operate the chip's internal gates, or *core logic*. A higher voltage, such as 3.3 V, is supplied to operate the external input and output circuits, or *pad ring*, for compatibility with older-generation devices in the system. Special buffer circuits are used internally to translate safely and quickly between the core-logic and the pad-ring logic voltages.

3.3-V LVTTL and LVCMOS Logic

The JEDEC standard for 3.3-V logic actually defines two sets of levels. *LVCMOS (low-voltage CMOS)* levels are used in pure CMOS applications where outputs have light DC loads (less than 100 □A), so *V*OL and *V*OH are maintained within 0.2 V of the power-supply rails. *LVTTL (low-voltage TTL)* levels, shown in figure, are used in applications where outputs have significant DC loads, so *V*OL can be as high as 0.4 V and *V*OH can be as low as 2.4 V. The positioning of TTL's logic levels at the low end of the 5-V range was really quite fortuitous.



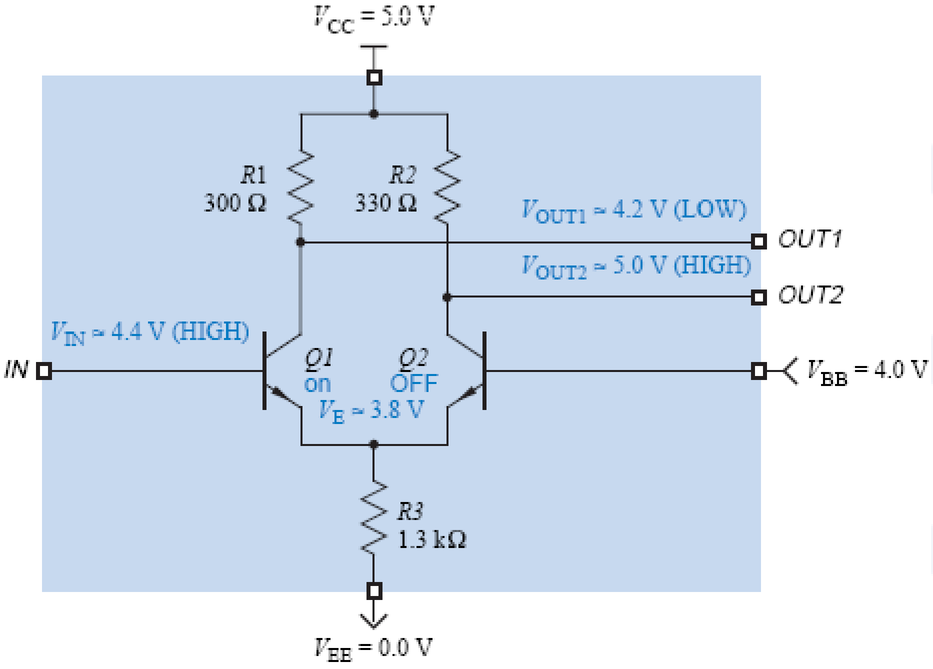
Emitter-Coupled Logic

The key to reducing propagation delay in a bipolar logic family is to prevent a gate's transistors from saturating. It is possible to prevent saturation by using a radically different circuit structure, called *current-mode logic (CML)* or *emitter-coupled logic (ECL)*. Unlike the other logic families in this chapter, CML does not produce a large voltage swing between the LOW and HIGH levels. Instead, it has a small voltage swing, less than a volt, and it internally switches current between two possible paths, depending on the output state. The first CML logic family was introduced by General Electric in 1961. The concept was soon refined by Motorola and others to produce the still popular 10K and 100K *emitter-coupled logic (ECL)* families. These families are extremely fast, offering propagation delays as short as 1 ns. The newest ECL family, ECLinPS (literally, ECL in picoseconds), offers maximum delays under 0.5 ns (500 ps), including the signal delay getting on and off of the IC package. Throughout the evolution of digital circuit technology, some type of ECL has always been the fastest technology for discrete, packaged logic components. Still, commercial ECL families aren't nearly as popular as CMOS and TTL, mainly because they consume much more power. In fact, high power consumption made the design of ECL supercomputers, such as the Cray-1 and Cray-2, as much of a challenge in cooling technology as in digital design. Also, ECL has a poor speed-power product, does not provide a high level of integration, has fast edge rates requiring design for transmission-line effects in most applications, and is not directly compatible with TTL and CMOS. Nevertheless, ECL still finds its place as a logic and interface technology in very high-speed communications gear, including fiber-optic transceiver interfaces for gigabit Ethernet and Asynchronous Transfer Mode (ATM) networks.

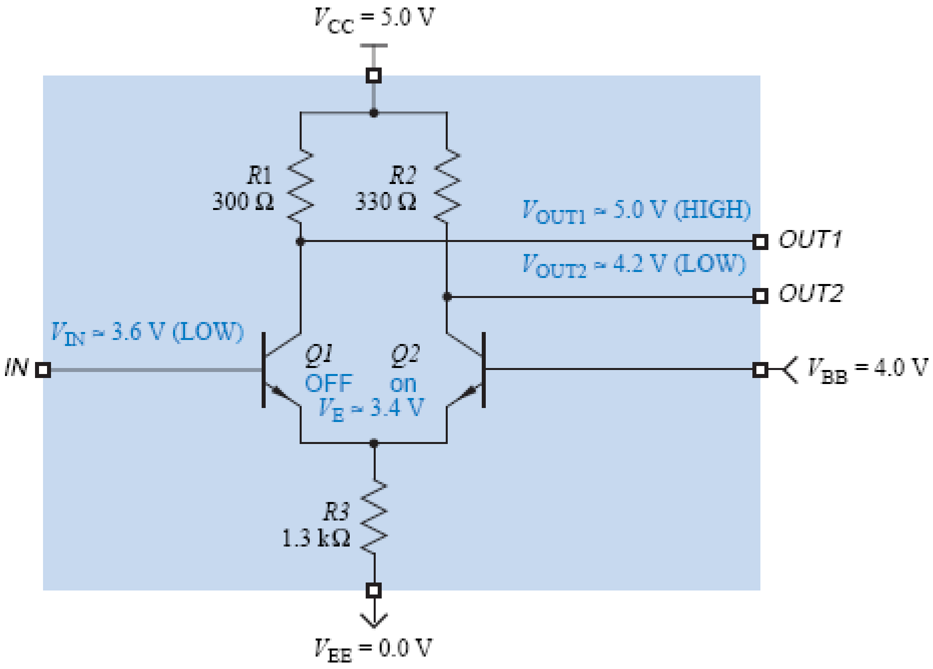
Basic CML Circuit

The basic idea of current-mode logic is illustrated by the inverter/buffer circuit in Figure 3-88. This circuit has both an inverting output (OUT1) and a noninverting output (OUT2). Two transistors are connected as a *differential amplifier* with a common emitter resistor. This circuit actually produces output LOW and HIGH levels that are 0.6 V higher (4.2 and 5.0 V), but this is corrected in real ECL circuits.

Basic CML inverter /buffer with input high



Basic CML inverter /buffer with input high



When *V*IN is HIGH, as shown in the figure, transistor *Q1* is on, but not saturated, and transistor *Q2* is OFF. This is true because of a careful choice of resistor values and voltage levels. Thus, *V*OUT2 is pulled to 5.0 V (HIGH) through *R2*, and it can be shown that the voltage drop across *R1* is about 0.8 V so that *V*OUT1 is about 4.2 V (LOW). When *V*IN is LOW, as shown in Figure, transistor *Q2* is on, but not saturated, and transistor *Q1* is OFF. Thus, *V*OUT1 is pulled to 5.0 V through *R1*, and it can be shown that *V*OUT2 is about 4.2 V. The outputs of this inverter are called *differential outputs* because they are always complementary, and it is possible to determine the output state by looking at the difference between the output voltages (*V*OUT1 □□*V*OUT2) rather than their absolute values. That is, the output is 1 if (*V*OUT1 □□*V*OUT2) > 0, and it is 0 if (*V*OUT1 □□*V*OUT2) > 0. It is possible to build input circuits with two wires per logical input that define the logical signal value in this way; these are called *differential inputs* Differential signals are used in most ECL “interfacing” and “clock distribution” applications because of their low skew and high noise immunity. They are “low skew” because the timing of a 0-to-1 or 1-to-0 transition does not depend critically on voltage thresholds, which may change with temperature or between devices. Instead, the timing depends only on when the voltages cross over relative to each other. Similarly, the “relative” definition of 0 and 1 provides outstanding noise immunity, since noise created by variations in the power supply or coupled from external sources tend to be *common-mode signals* that affect both differential signals similarly, leaving the difference value unchanged.

UNIT V
VLSI DESIGN FLOW

The VHDL Hardware Description Language

VHDL stands for **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. In the mid-1980’s the U.S. Department of Defense and the IEEE sponsored the development of this hardware description language with the goal to develop very high-speed integrated circuit. It has become now one of industry’s standard languages used to describe digital systems. The other widely used hardware description language is Verilog. Both are powerful languages that allow you to describe and simulate complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language) which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry. This tutorial deals with VHDL, as described by the IEEE standard 1076-1993.

5.1. LEVELS OF REPRESENTATION AND ABSTRACTION

A digital system can be represented at different levels of abstraction. This keeps the description and design of complex systems manageable. Figure 1 shows different levels of abstraction.

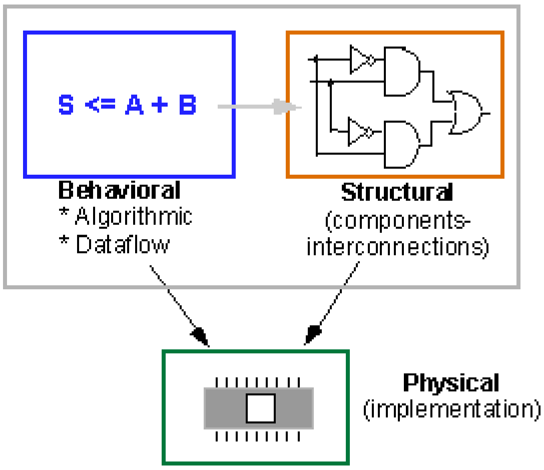


Figure 1: Levels of abstraction: Behavioral, Structural and Physical

The highest level of abstraction is the **behavioral** level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.

As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock. At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system. For the above example, the structural representation is shown in Figure 2 below.

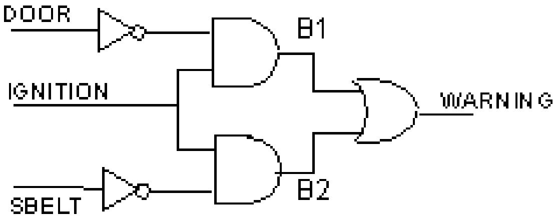


Figure 2: Structural representation of a "buzzer" circuit

VHDL allows one to describe a digital system at the structural or the behavioral level. The behavioral level can be further divided into two kinds of styles: **Data flow** and **Algorithmic**. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input. On the other hand, sequential statements are executed in the sequence that they are specified. VHDL allows both concurrent and sequential signal assignments that will determine the manner in which they are executed.

Program Structure

BASIC STRUCTURE OF A VHDL FILE

A digital system in VHDL consists of a design **entity** that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an ***entity declaration*** and an ***architecture body***. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure 3 below.

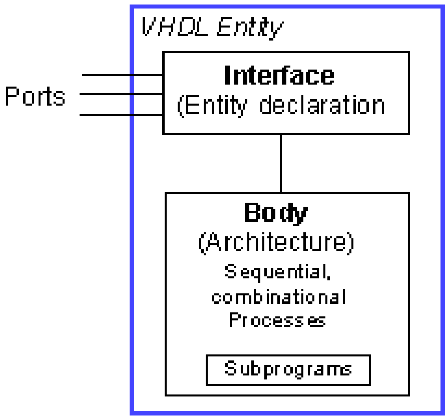


Figure 3: A VHDL entity consisting of an interface and a body

VHDL uses reserved **keyword**s that cannot be used as signal names or identifiers. Keywords and user-defined identifiers are **case insensitive**. Lines with comments start with two adjacent hyphens (--) and will be ignored by the compiler. VHDL also ignores line breaks and extra spaces. VHDL is **a strongly typed** language which implies that one has always to declare the type of every object that can have a value, such as signals, constants and variables.

5.2. ENTITY DECLARATION

The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

```
        entity NAME_OF_ENTITY is [(generic generic_declarations);]
                port (signal_names: mode type;
                        signal_names: mode type;
                                :
                        signal_names: mode type);
        end [NAME_OF_ENTITY];
```

An entity always starts with the keyword **entity**, followed by its name and the keyword **is**. Next are the port declarations using the keyword **port**. An entity declaration always ends with the keyword **end**, optionally [ ] followed by the name of the entity.

- The NAME_OF_ENTITY is a user-selected identifier
- *signal_names* consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.
- **mode**: is one of the reserved words to indicate the signal direction:

  - **in** – indicates that the signal is an input.
  - **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
  - **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity’s architecture.
  - **inout** – the signal can be an input or an output.
- *type*: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.
  - *bit* – can have the value 0 and 1
  - *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)
  - *std_logic, std_ulogic, std_logic_vector, std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
  - *boolean* – can have the value TRUE and FALSE.
  - *integer* – can have a range of integer values.
  - *real* – can have a range of real values.
  - *character* – any printing character.
  - *time* – to indicate time.

- **generic:** generic declarations are optional and determine the local constants used for timing and sizing (e.g. bus widths) the entity. A generic can have a default value. The syntax for a generic follows,

```
generic (
constant_name: type [:=value] ;
constant_name: type [:=value] ;
:
constant_name: type [:=value] );
```

For the example of Figure 2 above, the entity declaration looks as follows.

```
-- comments: example of the buzzer circuit of fig. 2
entity BUZZER is
   port (DOOR, IGNITION, SBELT: in std_logic;
                    WARNING: out std_logic);
end BUZZER;
```

The entity is called BUZZER and has three input ports, DOOR, IGNITION and SBELT and one output port, WARNING. Notice the use and placement of semicolons! The name BUZZER is an *identifier*. Inputs are denoted by the keyword **in**, and outputs by the keyword **out**. Since VHDL is a strongly typed language, each port has a defined *type*. In this case, we specified the std_logic *type*. This is the preferred type of digital signals. In contrast to the bit type that can only have the values ‘1’ and ‘0’, the std_logic and std_ulogic types can have nine values. This is important to describe a digital system accurately including the binary values 0 and 1, as well as the unknown value X, the uninitialized value U, “-” for don’t care, Z for high impedance, and several symbols to indicate the signal strength (e.g. L for weak 0, H for weak 1, W for weak unknown - see section on Enumerated Types). The std_logic type is defined in the std_logic_1164 package of the IEEE library. The type defines the set of values an object can have. This has the advantage that it helps with the creation of models and helps reduce errors. For instance, if one tries to assign an illegal value to an object, the compiler will flag the error.

A few other examples of entity declarations follow.

Four-to-one multiplexer of which each input is an 8-bit word.

```
entity mux4_to_1 is
        port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
            SEL: in std_logic_vector (1 downto 0);
                OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

An example of the entity declaration of a D flip-flop with set and reset inputs is

```
entity dff_sr is
        port (D,CLK,S,R: in std_logic;
        Q,Qnot: out std_logic);
end dff_sr;
```

a. Architecture body

The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

The architecture body looks as follows,

```
architecture architecture_name of NAME_OF_ENTITY is
-- Declarations
        -- components declarations
        -- signal declarations
        -- constant declarations
        -- function declarations
        -- procedure declarations
        -- type declarations
        :
begin
-- Statements

        :

end architecture_name;
```

Behavioral model

The architecture body for the example of Figure 2, described at the behavioral level, is given below,

```
        architecture behavioral of BUZZER is
        begin
                WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);
        end behavioral;
```

The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword **begin** and gives the Boolean expression of the function. We will see later that a behavioral model can be described in several other ways. The "<= " symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an **end** keyword followed by the architecture name.

A few other examples follow. The behavioral description of a two-input AND gate is shown below.

```
        entity AND2 is
                port (in1, in2: in std_logic;
                out1: out std_logic);
        end AND2;

        architecture behavioral_2 of AND2 is
        begin
                        out1 <= in1 and in2;
        end behavioral_2;
```

An example of a two-input XNOR gate is shown below.

```
        entity XNOR2 is
                port (A, B: in std_logic;
                Z: out std_logic);
        end XNOR2;

        architecture behavioral_xnor of XNOR2 is
                -- signal declaration (of internal signals X, Y)
                signal X, Y: std_logic;
        begin
                        X <= A and B;
                        Y <= (not A) and (not B);
                        Z <= X or Y;
        end behavioral_xnor;
```

The statements in the body of the architecture make use of logic operators. Logic operators that are allowed are: **and, or, nand, nor, xor, xnor** and **not**. In addition, other types of operators including relational, shift, arithmetic are allowed as well (see section on Operators). For more information on behavioral modeling see section on Behavioral Modeling.

Concurrency

It is worth pointing out that the signal assignments in the above examples are *concurrent* statements. This implies that the statements are executed when one or more of the signals on the right hand side change their value (i.e. an event occurs on one of the signals). For instance, when the input A changes, the internal signals X and Y change values that in turn causes the last statement to update the output Z. There may be a propagation delay associated with this change. Digital systems are basically data-driven and an event which occurs on one signal will lead to an event on another signal, etc. The execution of the statements is determined by the flow of signal values. As a result, the order in which these statements are given does not matter (i.e., moving the statement for the output Z ahead of that for X and Y does not change the outcome). This is in contrast to conventional, software programs that execute the statements in a sequential or procedural manner.

Structural description

The circuit of Figure 2 can also be described using a structural model that specifies what gates are used and how they are interconnected. The following example illustrates it.

```
architecture structural of BUZZER is
        -- Declarations
        component AND2
                port (in1, in2: in std_logic;
                   out1: out std_logic);
        end component;
        component OR2
                port (in1, in2: in std_logic;
                   out1: out std_logic);
        end component;
        component NOT1
                port (in1: in std_logic;
                   out1: out std_logic);
        end component;
        -- declaration of signals used to interconnect gates
        signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;
begin
        -- Component instantiations statements
        U0: NOT1 port map (DOOR, DOOR_NOT);
        U1: NOT1 port map (SBELT, SBELT_NOT);
        U2: AND2 port map (IGNITION, DOOR_NOT, B1);
```

```
                U3: AND2 port map (IGNITION, SBELT_NOT, B2);
                U4: OR2  port map (B1, B2, WARNING);

        end structural;
```

Following the header is the *declarative* part that gives the **components** (gates) that are going to be used in the description of the circuits. In our example, we use a two- input AND gate, two-input OR gate and an inverter. These gates have to be defined first, i.e. they will need an entity declaration and architecture body (as shown in the previous example). These can be stored in one of the *packages* one refers to in the header of the file (see Library and Packages below). The declarations for the components give the inputs (e.g. in1, in2) and the output (e.g. out1). Next, one has to define internal nets (**signal** names). In our example these signals are called DOOR_NOT, SBELT_NOT, B1, B2 (see Figure 2). Notice that one always has to declare the type of the signal.

The *statements* after the **begin** keyword gives the instantiations of the components and describes how these are interconnected. A component instantiation statement creates a new level of hierarchy. Each line starts with an *instance name* (e.g. U0) followed by a *colon* and a *component name* and the keyword **port map**. This keyword defines how the components are connected. In the example above, this is done through positional association: DOOR corresponds to the input, in1 of the NOT1 gate and DOOR_NOT to the output. Similarly, for the AND2 gate where the first two signals (IGNITION and DOOR_NOT) correspond to the inputs in1 and in2, respectively, and the signal B1 to the output out1. An alternative way is to use explicit association between the ports, as shown below.

label: component-name **port map** (*port1=>signal1, port2=> signal2,… port3=>signaln*);

```
U0: NOT1 port map (in1 => DOOR, out1 => DOOR_NOT);
U1: NOT1 port map (in1 => SBELT, out1 => SBELT_NOT);
U2: AND2 port map (in1 => IGNITION, in2 => DOOR_NOT, out1 => B1);
U3: AND2 port map (in1 => IGNITION, in2 => SBELT_NOT, B2);
U4: OR2  port map (in1 => B1, in2 => B2, out1 => WARNING);
```

Notice that the order in which these statements are written has no bearing on the execution since these statements are concurrent and therefore executed in parallel. Indeed, the schematic that is described by these statements is the same independent of the order of the statements.

Structural modeling of design lends itself to hierarchical design, in which one can define components of units that are used over and over again. Once these components are defined they can be used as blocks, cells or macros in a higher level entity. This can significantly reduce the complexity of large designs. Hierarchical design approaches are always preferred over flat designs. We will illustrate the use of a hierarchical design approach for a 4-bit adder, shown in Figure 4 below. Each full adder can be described by the Boolean expressions for the sum and carry out signals,

$$\text{sum} = (A \oplus B) \oplus C$$

$$\text{carry} = AB + C(A \oplus B)$$

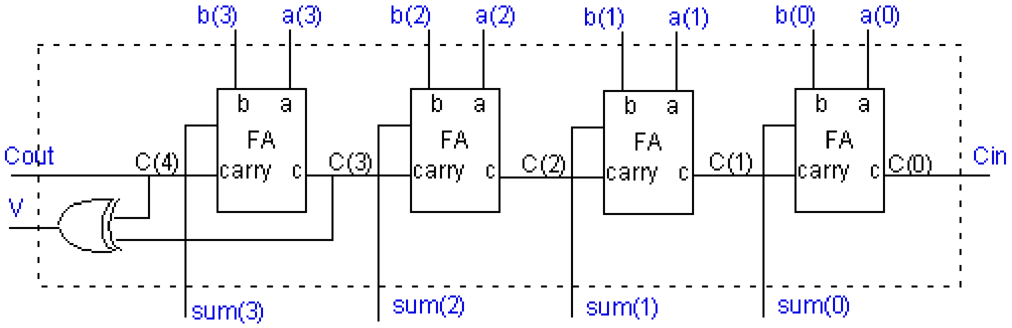


Figure 4: Schematic of a 4-bit adder consisting of full adder modules.

In the VHDL file, we have defined a component for the full adder first. We used several instantiations of the full adder to build the structure of the 4-bit adder. We have included the library and use clause as well as the entity declarations.

Four Bit Adder – Illustrating a hierarchical VHDL model

```
-- Example of a four bit adder
library  ieee;
use  ieee.std_logic_1164.all;
-- definition of a full adder
entity FULLADDER is
        port (a, b, c: in std_logic;
        sum, carry: out std_logic);
end FULLADDER;
architecture fulladder_behav of FULLADDER is
begin
        sum <= (a xor b) xor c ;
        carry <= (a and b) or (c and (a xor b));
end fulladder_behav;

-- 4-bit adder
library  ieee;
use  ieee.std_logic_1164.all;

entity FOURBITADD is
        port (a, b: in std_logic_vector(3 downto 0);
                Cin : in std_logic;
                sum: out std_logic_vector (3 downto 0);
                Cout, V: out std_logic);
end FOURBITADD;

architecture fouradder_structure of FOURBITADD is
        signal c: std_logic_vector (4 downto 0);
        component FULLADDER
```

```
                port(a, b, c: in std_logic;
                        sum, carry: out std_logic);
        end component;
begin
        FA0: FULLADDER
                port map (a(0), b(0), Cin, sum(0), c(1));
        FA1: FULLADDER
                port map (a(1), b(1), C(1), sum(1), c(2));
        FA2: FULLADDER
                port map (a(2), b(2), C(2), sum(2), c(3));
        FA3: FULLADDER
                port map (a(3), b(3), C(3), sum(3), c(4));
        V <= c(3) xor c(4);
        Cout <= c(4);
end fouradder_structure;
```

Notice that the same input names a and b for the ports of the full adder and the 4-bit adder were used. This does not pose a problem in VHDL since they refer to different levels. However, for readability, it may be easier to use different names. We needed to define the internal signals c(4:0) to indicate the nets that connect the output carry to the input carry of the next full adder. For the first input we used the input signal Cin. For the last carry we defined c(4) as an internal signal since the last carry is needed as the input to the xor gate. We could not use the output signal Cout since VHDL does not allow the use of outputs as internal signals! For this reason we had to define the internal carry c(4) and assign c(4) to the output carry signal Cout.

Types And Constants

DATA TYPES

Each data object has a type associated with it. The type defines the set of values that the object can have and the set of operations that are allowed on it. The notion of *type* is key to VHDL since it is a strongly typed language that requires each object to be of a certain type. In general one is not allowed to assign a value of one type to an object of another data type (e.g. assigning an integer to a bit type is not allowed). There are four classes of data types: scalar, composite, access and file types. The scalar types represent a single value and are ordered so that relational operations can be performed on them. The scalar type includes integer, real, and enumerated types of Boolean and Character. Examples of these will be given further on.

Data Types defined in the Standard Package

VHDL has several predefined types in the *standard* package as shown in the table below. To use this package one has to include the following clause:

```
library std, work;
use std.standard.all;
```

Types defined in the Package *Standard* of the *std* Library

Type	**Range of values**	**Example**
bit	‘0’, ‘1’	signal A: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE’
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. ‘#’)	variable VAL: character :=’$’;
file_open_kind*	read_mode, write_mode, append_mode	
file_open_status*	open_ok, status_error, name_error, mode_error	
integer	range is implementation dependent but includes at least –(2^{31} – 1) to +(2^{31} – 1)	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;
positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;
real*	floating point number in the range of –1.0 x 10^{38} to +1.0x 10^{38} (can be implementation dependent. *Not supported by the Foundation synthesis program.*	variable VAR3: real :=+64.2E12;
severity_level	note, warning, error, failure	
string	array of which each element is of the type character	variable VAR4: string(1 to 12):= “@$#ABC*()_%Z”;
time*	an integer number of which the range is implementation defined; units can be expressed in sec, ms, us, ns, ps, fs, min and hr. . *Not supported by the Foundation synthesis program*	variable DELAY: time :=5 ns;

* *Not supported by the Foundation synthesis program*

User-defined Types

One can introduce new types by using the type declaration, which names the type and specifies its value range. The syntax is

type identifier **is** type_definition;

Here are a few examples of type definitions,

Integer types

type small_int **is range** 0 **to** 1024**;**

type my_word_length **is range** 31 **downto** 0**;**

subtype data_word **is** my_word_length **range** 7 **downto** 0;

A subtype is a subset of a previously defined type. The last example above illustrates the use of subtypes. It defines a type called data_word that is a sybtype of my_word_length of which the range is restricted from 7 to 0. Another example of a subtype is,

```
        subtype int_small is  integer range -1024 to +1024;
```

Floating-point types

```
        type cmos_level is range 0.0 to 3.3;
        type pmos_level is range -5.0 to 0.0;
        type probability is range 0.0 to 1.0;
        subtype cmos_low_V is cmos_level range 0.0 to +1.8;
```

Note that floating point data types are not supported by the Xilinx Foundation synthesis program.

Physical types

The physical type definition includes a units identifier as follows,

```
        type conductance is range 0 to 2E-9
                units
                        mho;
                        mmho = 1E-3 mho;
                        umho = 1E-6 mho;
                        nmho = 1E-9 mho;
                        pmho = 1E-12 mho;
                end units conductance;
```

Here are some object declarations that use the above types,

```
        variable BUS_WIDTH: small_int :=24;
        signal DATA_BUS: my_word_length;
        variable VAR1: cmos_level range 0.0 to 2.5;
        constant LINE_COND: conductance:= 125 umho;
```

Notice that a space must be left before the unit name.

The physical data types are not supported by the Xilinx Foundation Express synthesis program.

In order to use our own types, we need either to include the type definition inside an architecture body or to declare the type in a package. The latter can be done as follows for a package called “my_types”.

```
package my_types is
        type small_int is range 0 to 1024;
        type my_word_length is range 31 downto 0;
        subtype data_word is my_word_length is range 7 downto 0;
        type cmos_level is range 0.0 to 3.3;
        type conductance is range 0 to 2E-9
                units
```

```
                    mho;
                    mmho = 1E-3 mho;
                    umho = 1E-6 mho;
                    nmho = 1E-9 mho;
                    pmho = 1E-12 mho;
            end units conductance;
end package my_types;
```

c. Enumerated Types

An enumerated type consists of lists of character literals or identifiers. The enumerated type can be very handy when writing models at an abstract level. The syntax for an enumerated type is,

```
        type type_name is (identifier list or character literal);
```

Here are some examples,

```
type my_3values is (‘0’, ‘1’, ‘Z’);
type PC_OPER  is (load, store, add, sub, div, mult, shiftl, shiftr);
type hex_digit  is (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, 8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’);
type state_type is (S0, S1, S2, S3);
```

Examples of objects that use the above types:

```
        signal SIG1: my_3values;
        variable ALU_OP: pc_oper;
        variable first_digit: hex_digit :=’0’;
        signal STATE: state_type :=S2;
```

If one does not initialize the signal, the default initialization is the leftmost element of the list.

Enumerated types have to be defined in the architecture body or inside a package as shown in the section above.

An example of an enumerated type that has been defined in the std_logic_1164 package is the std_ulogic type, defined as follows

```
        type STD_ULOGIC is (
                ‘U’,            -- uninitialized
                ‘X’,            -- forcing unknown
                ‘0’,            -- forcing 0
                ‘1’,            -- forcing 1
                ‘Z’,            -- high impedance
                ‘W’,            -- weak unknown
                ‘L’,            --  weak 0
                ‘H’.            -- weak 1
                ‘-‘);           -- don’t care
```

In order to use this type one has to include the clause before each entity declaration.

```
        library ieee; use ieee.std_logic_1164.all;
```

It is possible that multiple drivers are driving a signal. In that case there could be a conflict and the output signal would be undetermined. For instance, the outputs of an AND gate and NOT gate are connected together into the output net OUT1. In order to resolve the value of the output, one can call up a *resolution* function. These are usually a user-written function that will resolve the signal. If the signal is of the type std_ulogic and has multiple drivers, one needs to use a resolution function. The std_logic_1164 package has such a resolution function, called RESOLVED predefined. One can then use the following declaration for signal OUT1

```
signal OUT1: resolved: std_ulogic;
```

If there is contention, the RESOLVED function will be used to intermediate the conflict and determine the value of the signal. Alternatively, one can declare the signal directly as a std_logic type since the subtype std_logic has been defined in the std_logic_1164 package.

```
signal OUT1: std_logic;
```

d. Composite Types: Array and Record

Composite data objects consist of a collection of related data elements in the form of an *array* or *record*. Before we can use such objects one has to declare the composite type first.

Array Type

An array type is declared as follows:

```
type array_name is array (indexing scheme) of element_type;
type MY_WORD is array (15 downto 0) of std_logic;
type YOUR_WORD is array (0 to 15) of std_logic;
type VAR is array (0 to 7) of integer;
type STD_LOGIC_1D is array (std_ulogic) of std_logic;
```

In the first two examples above we have defined a one-dimensional array of elements of the type std_logic indexed from 15 down to 0, and 0 up to 15, respectively. The last example defines a one-dimensional array of the type std_logic elements that uses the type std_ulogic to define the index constraint. Thus this array looks as follows:

Index: ‘U’ ‘X’ ‘0’ ‘1’ ‘Z’ ‘W’ ‘L’ ‘H’ ‘-‘

Element:

We can now declare objects of these data types. Some examples are given

```
signal MEM_ADDR: MY_WORD;
signal DATA_WORD: YOUR_WORD :=  B“1101100101010110”;
constant SETTING: VAR := (2,4,6,8,10,12,14,16);
```

In the first example, the signal MEM_ADDR is an array of 16 bits, initialized to all ‘0’s. To access individual elements of an array we specify the index. For example, MEM_ACCR(15) accesses the left most bit of the array, while DATA_WORD(15) accesses the right most bit of the array with value ‘0’. To access a subrange, one specifies the index range, MEM_ADDR(15 **downto** 8) or DATA_WORD(0 **to** 7).

Multidimensional arrays can be declared as well by using a similar syntax as above,

```
type MY_MATRIX3X2 is array (1 to 3, 1 to 2) of natural;
type YOUR_MATRIX4X2 is array (1 to 4, 1 to 2) of integer;
type STD_LOGIC_2D is array (std_ulogic, std_ulogic) of std_logic;
```

```
        variable DATA_ARR: MY_MATRIX :=((0,2), (1,3), (4,6), (5,7));
```

The variable array DATA_ARR will then be initialized to,

```
                        0  2
                        1  3
                        4  6
                        5  7
```

To access an element one specifies the index, e.g. DATA_ARR(3,1) returns the value 4.

The last example defines a 9x9 array or table with an index the elements of the std_ulogic type.

Sometimes it is more convenient not to specify the dimension of the array when the array type is declared. This is called an unconstrained array type. The syntax for the array declaration is,

```
        type array_name is array (type range <>) of element_type;
```

Some examples are

```
        type MATRIX is array (integer range <>) of integer;
        type VECTOR_INT is array (natural range <>) of integer;
        type VECTOR2 is array (natural range <>, natural range <>) of std_logic;
```

The range is now specified when one declares the array object,

```
        variable MATRIX8: MATRIX (2 downto -8) := (3, 5, 1, 4, 7, 9, 12, 14, 20, 18);
        variable ARRAY3x2: VECTOR2 (1 to 4, 1 to 3)) := ((‘1’,’0’), (‘0’,’-‘), (1, ‘Z’));
```

Record Type

A second composite type is the *records* type. A record consists of multiple elements that may be of different types. The syntax for a record type is the following:

```
        type name is
                record
                        identifier :subtype_indication;
                                :
                        identifier :subtype_indication;
                end record;
```

As an example,

```
        type MY_MODULE  is
                record
                        RISE_TIME       :time;
                        FALL_TIME       : time;
                        SIZE            : integer range 0 to 200;
                        DATA            : bit_vector (15 downto 0);
                end record;
        signal A, B: MY_MODULE;
```

To access values or assign values to records, one can use one of the following methods:

```
A.RISE_TIME <= 5ns;
A.SIZE <= 120;
B <= A;
```

e. Type Conversions

Since VHDL is a strongly typed language one cannot assign a value of one data type to a signal of a different data type. In general, it is preferred to the same data types for the signals in a design, such as std_logic (instead of a mix of std_logic and bit types). Sometimes one cannot avoid using different types. To allow assigning data between objects of different types, one needs to convert one type to the other. Fortunately there are functions available in several packages in the ieee library, such as the std_logic_1164 and the std_logic_arith packages. As an example, the std_logic_1164 package allows the following conversions:

Conversions supported by std_logic_1164 package	
Conversion	**Function**
std_ulogic to bit	to_bit(*expression*)
std_logic_vector to bit_vector	to_bitvector(*expression*)
std_ulogic_vector to bit_vector	to_bitvector(*expression*)
bit to std_ulogic	To_StdULogic(*expression)*
bit_vector to std_logic_vector	To_StdLogicVector(*expression*)
bit_vector to std_ulogic_vector	To_StdUlogicVector(*expression*)
std_ulogic to std_logic_vector	To_StdLogicVector(*expression*)
std_logic to std_ulogic_vector	To_StdUlogicVector(*expression*)

The IEEE std_logic_unsigned and the IEEE std_logic_arith packages allow additional conversions such as from an integer to std_logic_vector and vice versa.

An example follows.

```
entity QUAD_NAND2 is
        port (A, B: in bit_vector(3 downto 0);
        out4: out std_logic_vector (3 downto 0));
end QUAD_NAND2;
architecture behavioral_2 of QUAD_NAND2 is
begin
                out4 <= to_StdLogicVector(A and B);
end behavioral_2;
```

The expression "A **and** B" which is of the type bit_vector has to be converted to the type std_logic_vector to be of the same type as the output signal out4.

The syntax of a type conversion is as follows:

```
type_name (expression);
```

In order for the conversion to be legal, the *expression* must return a type that can be converted into the type *type_name*. Here are the conditions that must be fulfilled for the conversion to be possible.

- Type conversions between integer types or between similar array types are possible
- Conversion between array types is possible if they have the same length and if they have identical element types or convertible element types.
- Enumerated types cannot be converted.

DATA OBJECTS: SIGNALS, VARIABLES AND CONSTANTS

A data object is created by an *object declaration* and has a *value* and *type* associated with it. An object can be a Constant, Variable, Signal or a File. Up to now we have seen signals that were used as input or output ports or internal nets. Signals can be considered wires in a schematic that can have a current value and future values, and that are a function of the signal assignment statements. On the other hand, Variables and Constants are used to model the behavior of a circuit and are used in processes, procedures and functions, similarly as they would be in a programming language. Following is a brief discussion of each class of objects.

Constant

A constant can have a single value of a given type and cannot be changed during the simulation. A constant is declared as follows,

constant *list_of_name_of_constant*: type [:= initial value] ;

where the initial value is optional. Constants can be declared at the start of an architecture and can then be used anywhere within the architecture. Constants declared within a process can only be used inside that specific process.

constant RISE_FALL_TME: time := 2 ns;

constant DELAY1: time := 4 ns;

constant RISE_TIME, FALL_TIME: time:= 1 ns;

constant DATA_BUS: integer:= 16;

Variable

A variable can have a single value, as with a constant, but a variable can be updated using a variable assignment statement. The variable is updated without any delay as soon as the statement is executed. Variables must be declared *inside* a process (and are local to the process). The variable declaration is as follows:

variable *list_of_variable_names*: type [:= initial value] ;

A few examples follow:

variable CNTR_BIT: bit :=0;

variable VAR1: boolean :=FALSE;

variable SUM: integer **range** 0 **to** 256 :=16;

variable STS_BIT: bit_vector (7 **downto** 0);

The variable SUM, in the example above, is an integer that has a range from 0 to 256 with initial value of 16 at the start of the simulation. The fourth example defines a bit vector or 8 elements: STS_BIT(7), STS_BIT(6),… STS_BIT(0).

A variable can be updated using a variable assignment statement such as

```
        Variable_name := expression;
```

As soon as the expression is executed, the variable is updated *without any delay*.

Signal

Signals are declared *outside* the process using the following statement:

signal *list_of_signal_names*: type [:= initial value] ;

```
        signal SUM, CARRY: std_logic;
        signal CLOCK: bit;
        signal TRIGGER: integer :=0;
        signal DATA_BUS: bit_vector (0 to 7);
        signal VALUE: integer range 0 to 100;
```

Signals are updated when their signal assignment statement is executed, *after a certain delay*, as illustrated below,

```
        SUM <= (A xor B) after 2 ns;
```

If no delay is specified, the signal will be updated after a *delta* delay. One can also specify multiple waveforms using multiple events as illustrated below,

```
        signal wavefrm : std_logic;
        wavefrm <= ‘0’, ‘1’ after 5ns, ‘0’ after 10ns, ‘1’ after 20 ns;
```

It is important to understand the difference between variables and signals, particularly how it relates to when their value changes. A variable changes instantaneously when the variable assignment is executed. On the other hand, a signal changes a delay after the assignment expression is evaluated. If no delay is specified, the signal will change after a *delta* delay. This has important consequences for the updated values of variables and signals. Lets compare the two files in which a process is used to calculate the signal RESULT [7].

Example of a process using Variables

```
architecture VAR of EXAMPLE is
        signal TRIGGER, RESULT: integer := 0;
begin
        process
                variable variable1: integer :=1;
                variable variable2: integer :=2;
                variable variable3: integer :=3;
        begin
                wait on TRIGGER;
                variable1 := variable2;
                variable2 := variable1 + variable3;
                variable3 := variable2;
                RESULT <= variable1 + variable2 + variable3;
        end process;
end VAR
```

Example of a process using Signals

```
architecture SIGN of EXAMPLE is
        signal TRIGGER, RESULT: integer := 0;
        signal signal1: integer :=1;
        signal signal2: integer :=2;
        signal signal3: integer :=3;
begin
        process
        begin
                wait on TRIGGER;
                signal1 <= signal2;
                signal2 <= signal1 + signal3;
                signal3 <= signal2;
                RESULT  <= signal1 + signal2 + signal3;
        end process;
end SIGN;
```

In the first case, the variables "variable1, variable2 and variable3" are computed sequentially and their values updated instantaneously after the TRIGGER signal arrives. Next, the RESULT, which is a signal, is computed using the new values of the variables and updated a time *delta* after TRIGGER arrives. This results in the following values (after a time TRIGGER): variable1 = 2, variable2 = 5 (=2+3), variable3= 5. Since RESULT is a signal it will be computed at the time TRIGGER and updated at the time TRIGGER + Delta. Its value will be RESULT=12.

On the other hand, in the second example, the signals will be computed at the time TRIGGER. All of these signals are computed at the same time, using the old values of signal1, 2 and 3. All the signals will be updated at Delta time after the TRIGGER has arrived. Thus the signals will have these values: signal1= 2, signal2= 4 (=1+3), signal3=2 and RESULT=6.

Functions And Procedures

libraries and packages

Library and Packages: library and use keywords

A library can be considered as a place where the compiler stores information about a design project. A VHDL package is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.

As mentioned earlier that **std_logic** is defined in the package **ieee.std_logic_1164** in the **ieee** library. In order to use the **std_logic** one needs to specify the **library** and **package**. This is done at the beginning of the VHDL file using the **library** and the **use** keywords as follows:

```
        library  ieee;
        use  ieee.std_logic_1164.all;
```

The .**all** extension indicates to use all of the **ieee.std_logic_1164** package.

The Xilinx Foundation Express comes with several packages.

ieee Library:

- **std_logic_1164 package**: defines the standard data types.
- **std_logic_arith** package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types.
- **std_logic_unsigned**
- **std_logic_misc** package: defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.

To use any of these one must include the library and use clause:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

In addition, the synopsis library has the attributes package:

```
library SYNOPSYS;
use SYNOPSYS.attributes.all;
```

One can add other libraries and packages.

Package declaration

The syntax to declare a package is as follows:

```
-- Package declaration
        package name_of_package is
                package declarations
        end package name_of_package;


--Package body declarations
        package body name_of_package is
                package body declarations
        end package body name_of_package;
```

For instance, the basic functions of the AND2, OR2, NAND2, NOR2, XOR2, etc. components need to be defined before one can use them. This can be done in a package, e.g. basic_func for each of these components, as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package basic_func is
        -- AND2 declaration
        component AND2
                generic (DELAY: time :=5ns);
                port (in1, in2: in std_logic; out1: out std_logic);
        end component;
        -- OR2 declaration
        component OR2
                generic (DELAY: time :=5ns);
                port (in1, in2: in std_logic; out1: out std_logic);
```

```
          end component;
end package basic_func;
```

Package body declarations

```
library ieee;
use ieee.std_logic_1164.all;
package body basic_func is
-- 2 input AND gate
          entity AND2 is
                    generic (DELAY: time);
                    port (in1, in2: in std_logic; out1: out std_logic);
          end AND2;
          architecture model_conc of AND2 is
                    begin
                              out1 <= in1 and in2 after DELAY;
          end model_conc;
-- 2 input OR gate
          entity OR2 is
                    generic (DELAY: time);
                    port (in1, in2: in std_logic; out1: out std_logic);
          end OR2;
          architecture model_conc2 of AND2 is
                    begin
                              out1 <= in1 or in2 after DELAY;
          end model_conc2;
end package body basic_func;
```

Notice that we included a delay of 5 ns. However, it should be noticed that delay specifications are ignored by the Foundation synthesis tool. We made use of the predefined type std_logic that is declared in the package std_logic_1164. We have included the **library** and **use** clause for this package. This package needs to be compiled and placed in a library. Lets call this library my_func. To use the components of this package one has to declare it using the *library* and *use* clause:

```
          library  ieee, my_func;
          use  ieee.std_logic_1164.all, my_func.basic_func.all;
```

One can concatenate a series of names separated by periods to select a package. The library and use statements are connected to the subsequent entity statement. The library and use statements have to be repeated for each entity declaration.

One has to include the library and use clause for each entity as shown for the example of the four-bit adder above.

Structural design elements

STRUCTURAL MODELING

Structural modeling was described briefly in the section Structural Modeling in "Basic Structure of a VHDL file". A structural way of modeling describes a circuit in terms of components and its interconnection. Each component is supposed to be defined earlier (e.g. in package) and can be described as structural, a behavioral or dataflow model. At the lowest hierarchy each component is described as a behavioral model, using the basic logic operators defined in VHDL. In general structural modeling is very good to describe complex digital systems, though a set of components in a *hierarchical* fashion.

A structural description can best be compared to a schematic block diagram that can be described by the components and the interconnections. VHDL provides a formal way to do this by

- Declare a list of components being used
- Declare signals which define the nets that interconnect components
- Label multiple instances of the same component so that each instance is uniquely defined.

The components and signals are declared within the architecture body,

```
architecture architecture_name of NAME_OF_ENTITY is
        -- Declarations
                component declarations
                signal declarations
        begin
        -- Statements
                component instantiation and connections
                :
        end architecture_name;
```

a. Component declaration

Before components can be instantiated they need to be declared in the architecture declaration section or in the package declaration. The component declaration consists of the component name and the interface (ports). The syntax is as follows:

```
component component_name [is]
        [port (port_signal_names: mode type;
                 port_signal_names: mode type;
                         :
                port_signal_names: mode type);]
end component [component_name];
```

The component name refers to either the name of an entity defined in a library or an entity explicitly defined in the VHDL file (see example of the four bit adder).

The list of interface ports gives the name, mode and type of each port, similarly as is done in the entity declaration.

A few examples of component declaration follow:

```
component OR2
        port (in1, in2: in std_logic;
                out1: out std_logic);
end component;

component PROC
        port (CLK, RST, RW, STP: in std_logic;
                ADDRBUS: out std_logic_vector (31 downto 0);
                DATA: inout integer range 0 to 1024);

component FULLADDER
        port(a, b, c: in std_logic;
               sum, carry: out std_logic);
end component;
```

As mentioned earlier, the component declaration has to be done either in the architecture body or in the package declaration. If the component is declared in a package, one does not have to declare it again in the architecture body as long as one uses the **library** and **use** clause.

b. Component Instantiation and interconnections

The component instantiation statement references a component that can be

- Previously defined at the current level of the hierarchy or
- Defined in a technology library (vendor's library).

The syntax for the components instantiation is as follows,

```
instance_name : component name
    port map (port1=>signal1, port2=> signal2,… port3=>signaln);
```

The instance name or label can be any legal identifier and is the name of this particular instance. The component name is the name of the component declared earlier using the component declaration statement. The port name is the name of the port and signal is the name of the signal to which the specific port is connected. The above port map associates the ports to the signals through named association. An alternative method is the positional association shown below,

```
    port map (signal1,  signal2,…signaln);
```

in which the first port in the component declaration corresponds to the first signal, the second port to the second signal, etc. The signal position must be in the same order as the declared component's ports. One can mix named and positional associations as long as one puts all positional associations before the named ones. The following examples illustrates this,

```
component NAND2
        port (in1, in2: in std_logic;
            out1: out std_logic);
end component;
signal int1, int2, int3: std_logic;
```

```
architecture struct of EXAMPLE is
        U1: NAND2 port map (A,B,int1);
        U2: NAND2 port map (in2=>C, in2=>D, out1=>int2);
        U3: NAND3 port map (in1=>int1, int2, Z);
        …..
```

Another example is the Buzzer circuit of Figure 2.

Data flow design elements

DATAFLOW MODELING – CONCURRENT STATEMENTS

Behavioral modeling can be done with *sequential* statements using the process construct or with concurrent statements. The first method was described in the previous section and is useful to describe complex digital systems. In this section, we will use *concurrent* statements to describe behavior. This method is usually called dataflow modeling. The dataflow modeling describes a circuit in terms of its *function* and the *flow of data* through the circuit. This is different from the *structural* modeling that describes a circuit in terms of the interconnection of components.

Concurrent signal assignments are event triggered and executed as soon as an event on one of the signals occurs. In the remainder of the section we will describe several concurrent constructs for use in dataflow modeling.

a. Simple Concurrent signal assignments.

We have discussed several concurrent examples earlier in the tutorial. In this section we will review the different types of concurrent signal assignments.

A simple concurrent signal assignment is given in the following examples,

```
        Sum <= (A xor B) xor Cin;
        Carry <= (A and B);
        Z <= (not X) or Y after 2 ns;
```

The syntax is as follows:

```
        Target_signal <= expression;
```

in which the value of the expression transferred to the target_signal. As soon as an event occurs on one of the signals, the expression will be evaluated. The type of the target_signal has to be the same as the type of the value of the expression.

Another example is given below of a 4-bit adder circuit. Notice that we specified the package: IEEE.std_logic_unsigned in order to be able to use the "+" (addition) operator.

Example of a Four bit Adder using concurrent/behavioral modeling

```
        library ieee;
        use IEEE.std_logic_1164.all;
        use IEEE.std_logic_unsigned.all;
        entity ADD4 is
          port (
            A: in STD_LOGIC_VECTOR (3 downto 0);
            B: in STD_LOGIC_VECTOR (3 downto 0);
            CIN: in STD_LOGIC;
            SUM: out STD_LOGIC_VECTOR (3 downto 0);
            COUT: out STD_LOGIC
          );
        end ADD4;
        architecture ADD4_concurnt of ADD4 is
        -- define internal SUM signal including the carry
        signal SUMINT: STD_LOGIC_VECTOR(4 downto 0);
        begin
          -- <<enter your statements here>>
            SUMINT <=  ('0' & A) + ('0' & B) + ("0000" & CIN);
          COUT <= SUMINT(4);
          SUM <= SUMINT(3 downto 0);
        end ADD4_concurnt;
```

b. Conditional Signal assignments

The syntax for the conditional signal assignment is as follows:

```
Target_signal <= expression when Boolean_condition else
                 expression when Boolean_condition else
                            :
                 expression;
```

The target signal will receive the value of the first expression whose Boolean condition is TRUE. If no condition is found to be TRUE, the target signal will receive the value of the final expression. If more than one condition is true, the value of the first condition that is TRUE will be assigned.

An example of a 4-to-1 multiplexer using conditional signal assignments is shown below.

```
entity MUX_4_1_Conc is
  port (S1, S0, A, B, C, D: in std_logic;
                  Z: out std_logic);
  end MUX_4_1_Conc;
architecture concurr_MUX41 of MUX_4_1_Conc is
begin
        Z <=    A when S1='0' and S0='0' else
                B when S1='0' and S0='1' else
                C when S1='1' and S0='0' else
                D;
end concurr_MUX41;
```

The conditional signal assignment will be re-evaluated as soon as any of the signals in the conditions or expression change. The when-else construct is useful to express logic function in the form of a truth table. An example of the same multiplexer as above is given below in a more compact form.

```
entity MUX_4_1_funcTab is
  port (A, B, C, D: in std_logic;
            SEL: in std_logic_vector (1 downto  0);
                 Z: out std_logic);
  end MUX_4_1_ funcTab;
architecture concurr_MUX41 of MUX_4_1_ funcTab is
begin
        Z <=    A when SEL = ”00” else
                B when SEL = ”01” else
                C when SEL = “10” else
                D;
end concurr_MUX41;
```

Notice that this construct is simpler than the If-then-else construct using the *process* statement or the case statement. An alternative way to define the multiplexer is the case construct inside a process statement, as discussed earlier.

c. Selected Signal assignments

The selected signal assignment is similar to the conditional one described above. The syntax is as follows,

```
with choice_expression select
        target_name <= expression when choices,
        target_name <= expression when choices,
                :
        target_name <= expression when choices;
```

The target is a signal that will receive the value of an expression whose choice includes the value of the choice_expression. The expression selected is the first with a matching choice. The choice can be a static expression (e.g. 5) or a range expression (e.g. 4 **to** 9). The following rules must be followed for the choices:

- No two choices can overlap
- All possible values of choice_expression must be covered by the set of choices, unless an **others** choice is present.

An example of a 4-to-1 multiplexer is given below.

```
entity MUX_4_1_Conc2 is
  port (A, B, C, D: in std_logic;
                SEL: in std_logic_vector(1 downto 0);
                Z: out std_logic);
  end MUX_4_1_Conc2;
architecture concurr_MUX41b of MUX_4_1_Conc2 is
begin
        with SEL select
                Z <=    A when “00”,
                                B when “01”,
                                C when “10”,
                                D when “11”;
        end concurr_MUX41b;
```

The equivalent process statement would make use of the case construct. Similarly to the when-else construct, the selected signal assignment is useful to express a function as a truth table, as illustrated above.

The choices can express a single value, a range or combined choices as shown below.

```
                target <= value1 when “000”,
                                value2 when “001” | “011”  | “101” ,
                                value3 when others;
```

In the above example, all eight choices are covered and only once. The others choice must the last one used.

Notice that the Xilinx Foundation Express does not allow a vector as choice_expression such as std_logic_vector’(A,B,C).

As an example, lets consider a full adder with inputs A, B and C and outputs sum and cout,

```
entity FullAdd_Conc is
  port (A, B, C: in std_logic;
                sum, cout: out std_logic);
  end FullAdd_Conc;
architecture FullAdd_Conc of FullAdd_Conc is
        --define internal signal: vector INS of the input signals
        signal INS: std_logic_vector (2 downto 0);
begin
        --define the components of vector INS of the input signals
        INS(2) <= A;
        INS(1) <= B;
        INS(0) <= C;
        with INS select
                (sum, cout) <=  std_logic_vector’(“00”) when “000”,
                                        std_logic_vector’(“10”) when “001”,
                                        std_logic_vector’(“10”) when “010”,
                                        std_logic_vector’(“01”) when “011”,
                                        std_logic_vector’(“10”) when “100”,
                                        std_logic_vector’(“01”) when “101”,
                                        std_logic_vector’(“01”) when “110”,
                                        std_logic_vector’(“11”) when “111”,
                                        std_logic_vector’(“11”) when others;
end FullAdd_Conc; ]
```

Notice: In the example above we had to define an internal vector INS(A,B,C) of the input signals to use as part of the **with-select-when** statement. This was done because the Xilinx Foundation does not support the construct std_logic_vector'(A,B,C).

Behavioral Design Elements

BEHAVIORAL MODELING: SEQUENTIAL STATEMENTS

As discussed earlier, VHDL provides means to represent digital circuits at different levels of representation of abstraction, such as the behavioral and structural modeling. In this section we will discuss **different constructs** for describing the behavior of components and circuits in terms **of sequential statements**. The basis for sequential modeling is the *process* construct. As you will see, the *process* construct allows us to model complex digital systems, in particular sequential circuits.

Process

A process statement is the main construct in behavioral modeling that allows you to use sequential statements to describe the behavior of a system over time. The syntax for a process statement is

```
[process_label:] process [ (sensitivity_list) ] [is]
        [ process_declarations]
begin
        list of sequential statements such as:
                signal assignments
                variable assignments
                case statement
                exit statement
                if statement
                loop statement
                next statement
                null statement
                procedure call
                wait statement
end process [process_label];
```

An example of a positive edge-triggered D flip-flop with asynchronous clear input follows.

```
library ieee;
use ieee.std_logic_1164.all;
entity DFF_CLEAR is
        port (CLK, CLEAR, D : in std_logic;
                Q : out std_logic);
end DFF_CLEAR;
architecture BEHAV_DFF of DFF_CLEAR is
begin
DFF_PROCESS: process (CLK, CLEAR)
```

```
        begin
            if (CLEAR = ‘1’) then
                Q <= ‘0’;
            elsif (CLK’event and CLK = ‘1’) then
                Q <= D;
            end if;
        end process;
    end BEHAV_DFF;
```

A process is declared within architecture and is a *concurrent* statement. However, the statements inside a process are executed *sequentially*. Like other concurrent statements, a process reads and writes signals and values of the interface (input and output) ports to communicate with the rest of the architecture. One can thus make assignments to signals that are defined externally (e.g. interface ports) to the process, such as the Q output of the flip-flop in the above example. The expression CLK’**event and** CLK = ‘1’ checks for a positive clock edge (clock event AND clock high).

The sensitivity list is a set of signals to which the process is sensitive. Any change in the value of the signals in the sensitivity list will cause immediate execution of the process. If the sensitivity list is not specified, one has to include a **wait** statement to make sure that the process will halt. Notice that one cannot include both a sensitivity list and a wait statement. Variables and constants that are used inside a process have to be defined in the *process_declarations* part before the keyword **begin**. The keyword **begin** signals the start of the computational part of the process. The statements are *sequentially* executed, similarly as a conventional software program. It should be noted that variable assignments inside a process are executed immediately and denoted by the “:=” operator. This is in contrast to signal assignments denoted by “<=” and which changes occur after a delay. As a result, changes made to variables will be available immediately to all subsequent statements within the same process. For an example that illustrates the difference between signal and variable assignments see the section on Data Types (difference between signals and variables).

The previous example of the D flip-flop illustrates how to describe a sequential circuit with the process statement. Although the process is mainly used to describe sequential circuits, one can also describe combinational circuits with the process construct. The following example illustrates this for a Full Adder, composed of two Half Adders. This example also illustrates how one process can generate signals that will trigger other processes when events on the signals in its sensitivity list occur [3]. We can write the Boolean expression of a Half Adder and Full Adder as follows:

$S\_ha = (A \oplus B)$ and $C\_ha = AB$

For the Full Adder:

$Sum = (A \oplus B) \oplus Cin = S\_ha \oplus Cin$

$Cout = (A \oplus B)Cin + AB = S\_ha.Cin + C\_ha$

Figure 5 illustrates how the Full Adder has been modeled.

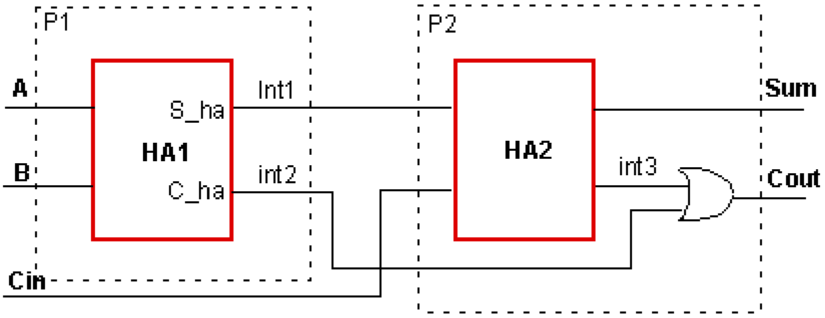


Figure 5: Full Adder composed of two Half Adders, modeled with two processes P1 and P2.

```
library ieee;
use ieee.std_logic_1164.all;
entity FULL_ADDER is
        port (A, B, Cin : in std_logic;
                Sum, Cout : out std_logic);
end FULL_ADDER;

architecture BEHAV_FA of FULL_ADDER is
signal int1, int2, int3: std_logic;
begin
-- Process P1 that defines the first half adder
P1: process (A, B)
        begin
                int1<= A xor B;
                int2<= A and B;
        end process;
-- Process P2 that defines the second half adder and the OR -- gate
P2: process (int1, int2, Cin)
        begin
                Sum <= int1 xor Cin;
                int3 <= int1 and Cin;
                Cout <= int2 or int3;
        end process;
end BEHAV_FA;
```

Of course, one could simplify the behavioral model significantly by using a single process.

b. If Statements

The if statement executes a sequence of statements whose sequence depends on one or more conditions. The syntax is as follows:

```
if condition then
        sequential statements
```

```
   [elsif condition then
            sequential statements ]
   [else
            sequential statements ]
end if;
```

Each condition is a Boolean expression. The if statement is performed by checking each condition in the order they are presented until a “true” is found. Nesting of if statements is allowed. An example of an if statement was given earlier for a D Flip-flop with asynchronous clear input. The if statement can be used to describe combinational circuits as well. The following example illustrates this for a 4-to-1 multiplexer with inputs A, B, C and D, and select signals S0 and S1. This statement must be inside a process construct. We will see that other constructs, such as the Conditional Signal Assignment (“When-else”) or “Select” construct may be more convenient for these type of combinational circuits.

```
entity MUX_4_1a is
  port (S1, S0, A, B, C, D: in std_logic;
                  Z: out std_logic);
  end MUX_4_1a;
architecture behav_MUX41a of MUX_4_1a is
begin
  P1: process (S1, S0, A, B, C, D)
  begin
        if (( not S1 and not S0 )=’1’) then
                        Z <= A;
                elsif (( not S1 and S0) = ‘1’) then
                        Z<=B;
                elsif ((S1 and not S0) =’1’) then
                        Z <=C;
                else
                        Z<=D;
        end if;
  end process P1;
end behav_MUX41a;
```

A slightly different way of modeling the same multiplexer is shown below,

```
if S1=’0’ and  S0=’0’ then
        Z <= A;
        elsif S1=’0’ and  S0=’1’ then
                Z <= B;
        elsif S1=’1’ and  S0=’0’ then
                Z <= C;
```

```
    elsif S1=’1’ and  S0=’1’ then
        Z <= D;
end if;
```

If statements are often used to implement state diagrams. For an example of a Mealy machine see Example Mealy Machine later on.

Case statements

The case statement executes one of several sequences of statements, based on the value of a single expression. The syntax is as follows,

```
case expression is
    when choices =>
        sequential statements
    when choices =>
        sequential statements
        -- branches are allowed
    [ when others => sequential statements ]
end case;
```

The expression must evaluate to an integer, an enumerated type of a one-dimensional array, such as a bit_vector. The case statement evaluates the expression and compares the value to each of the choices. The when clause corresponding to the matching choice will have its statements executed. The following rules must be adhered to:

- no two choices can overlap (i.e. each choice can be covered only once)
- if the “when others" choice is not present, all possible values of the expression must be covered by the set of choices.

An example of a case statement using an enumerated type follows. It gives an output D=1 when the signal GRADES has a value between 51 and 60, C=1 for grades between 61 and 70, the when others covers all the other grades and result in an F=1.

```
library ieee;
use ieee.std_logic_1164.all;
entity GRD_201 is
    port(VALUE: in integer range 0 to 100;
        A, B, C, D: out bit);
end GRD_201;
architecture behav_grd of GRD_201 is
```

```
begin
        process (VALUE)
         A <= ’0’;
         B <= ’0’;
         C <= ’0’;
         D <= ’0’;
         F <= ’0’;
         begin
                case VALUE is
                 when 51 to 60 =>
                        D <= ’1’;
                 when 61 to 70 | 71 to 75 =>
                        C <= ’1’;
                 when 76 to 85 =>
                        B <= ’1’;
                 when 86 to 100 =>
                        A <= ’1’;
                 when others  =>
                        F <= ‘1’;
                end case;
        end process;
end behav_grd;
```

We used the vertical bar (|) which is equivalent to the “or” operator, to illustrate how to express a range of values. This is a useful operator to indicate ranges that are not adjacent (e.g. 0 to 4 | 6 to 10).

Another example using the case construct is a 4-to-1 MUX.

```
entity MUX_4_1 is
  port ( SEL: in std_logic_vector(2 downto 1);
                A, B, C, D: in std_logic;
                Z: out std_logic);
  end MUX_4_1;
architecture behav_MUX41 of MUX_4_1 is
begin
  PR_MUX: process (SEL, A, B, C, D)
  begin
        case SEL is
                when “00” => Z <= A;
                when “01” => Z <= B;
                when “10” => Z <= C;
```

```
                        when "11" => Z <= D;
                        when others => Z <= 'X';
                end case;
          end process PR_MUX;
        end behav_MUX41;
```

The "when others" covers the cases when SEL="0X", "0Z", "XZ", "UX", etc. It should be noted that these combinational circuits can be expressed in other ways, using concurrent statements such as the "With – Select" construct. Since the case statement is a sequential statement, one can have nested case statements.

Loop statements

A loop statement is used to repeatedly execute a sequence of sequential statements. The syntax for a loop is as follows:

```
[ loop_label :]iteration_scheme loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
end loop [loop_label];
```

Labels are optional but are useful when writing nested loops. The next and exit statement are sequential statements that can only be used inside a loop.

- The **next** statement terminates the rest of the current loop iteration and execution will proceed to the next loop iteration.
- The **exit** statement skips the rest of the statements, terminating the loop entirely, and continues with the next statement after the exited loop.

There are three types of iteration schemes:

- basic **loop**
- **while … loop**
- **for … loop**

Basic Loop statement

This loop has no iteration scheme. It will be executed continuously until it encounters an exit or next statement.

```
[ loop_label :] loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
```

```
end loop [ loop_label];
```

The basic loop (as well as the while-loop) must have at least one **wait** statement. As an example, lets consider a 5-bit counter that counts from 0 to 31. When it reaches 31, it will start over from 0. A wait statement has been included so that the loop will execute every time the clock changes from ‘0’ to ‘1’.

Example of a basic **loop** to implement a counter that counts from 0 to 31

```
entity COUNT31 is
  port ( CLK: in std_logic;
                   COUNT: out integer);
  end COUNT31;
architecture behav_COUNT of COUNT31 is
begin
  P_COUNT: process
         variable intern_value: integer :=0;
  begin
         COUNT <= intern_value;
         loop
           wait until CLK='1';
           intern_value:=(intern_value + 1) mod 32;
           COUNT <= intern_value;
         end loop;
         end process P_COUNT;
end behav_COUNT;
```

We defined a variable intern_value inside the process because output ports cannot be read inside the process.

While-Loop statement

The while … loop evaluates a Boolean iteration condition. When the condition is TRUE, the loop repeats, otherwise the loop is skipped and the execution will halt. The syntax for the while…loop is as follows,

```
 [ loop_label :] while condition loop
         sequential statements
         [next  [label] [when condition];
         [exit  [label] [when condition];
end loop[ loop_label ];
```

The condition of the loop is tested before each iteration, including the first iteration. If it is false, the loop is terminated.

For-Loop statement

The for-loop uses an integer iteration scheme that determines the number of iterations. The syntax is as follows,

```
 [ loop_label :] for identifier in range loop
         sequential statements
```

[**next** [*label*] [**when** *condition*];

[**exit** [*label*] [**when** *condition*];

end loop[*loop_label*]**;**

- The *identifier (index)* is automatically declared by the loop itself, so one does not need to declare it separately. The value of the identifier can only be read inside the loop and is not available outside its loop. One cannot assign or change the value of the index. This is in contrast to the while-loop whose condition can involve variables that are modified inside the loop.
- The *range* must be a computable integer range in one of the following forms, in which integer_expression must evaluate to an integer:
    - *integer_expression* **to** *integer_expression*
    - *integer_expression* **downto** *integer_expression*

Next and Exit Statement

The next statement skips execution to the next iteration of a loop statement and proceeds with the next iteration. The syntax is

next [*label*] [**when** *condition*];

The **when** keyword is optional and will execute the next statement when its condition evaluates to the Boolean value TRUE.

The exit statement skips the rest of the statements, terminating the loop entirely, and continues with the next statement after the exited loop. The syntax is as follows:

exit [*label*] [**when** *condition*];

The **when** keyword is optional and will execute the next statement when its condition evaluates to the Boolean value TRUE.

Notice that the difference between the next and exit statement, is that the exit statement terminates the loop.

a. Wait statement

The wait statement will halt a process until an event occurs. There are several forms of the wait statement,

wait until *condition;*

wait for *time expression;*

wait on *signal;*

wait;

The Xilinx Foundation Express has implemented only the first form of the wait statement. The syntax is as follows,

wait until *signal* = *value;*

wait until *signal'event* **and** *signal* = *value;*

wait until not *signal'***stable and** signal = *value;*

The condition in the "**wait until**" statement must be TRUE for the process to resume. A few examples follow.

wait until CLK='1';

```
wait until CLK=’0’;
wait until CLK’event and CLK=’1’;
wait until not CLK’stable and CLK=’1’;
```

For the first example the process will wait until a positive-going clock edge occurs, while for the second example, the process will wait until a negative-going clock edge arrives. The last two examples are equivalent to the first one (positive-edge or 0-1 transitions). The hardware implementation for these three statements will be identical.

It should be noted that a process that contains a wait statement can not have a sensitivity list. If a process uses one or more wait statements, the Foundation Express synthesizer will use sequential logic. The results of the computations are stored in flip-flops.

Null statement

The null statement states that no action will occur. The syntax is as follows,

```
null;
```

It can be useful in a case statement where all choices must be covered, even if some of them can be ignored. As an example, consider a control signal CNTL in the range 0 to 31. When the value of CNTL is 3 or 15, the signals A and B will be xor-ed, otherwise nothing will occur.

```
entity EX_WAIT is
  port ( CNTL: in integer range 0 to 31;
                  A, B: in std_logic_vector(7 downto 0);
                  Z: out std_logic_vector(7 downto 0) );
  end EX_WAIT;
architecture arch_wait of EX_WAIT is
begin
  P_WAIT: process (CNTL)
  begin
          Z <=A;
          case CNTL is
                  when 3 | 15 =>
                          Z <= A xor B;
                  when others =>
                          null;
          end case;
          end process P_WAIT;
end arch_wait;
```

Example of a Mealy Machine

The sequence following detector recognizes the input bit sequence X: "1011". The machine will keep checking for the proper bit sequence and does not reset to the initial state after it recognizes the string. In case we are implementing a Mealy machine, the output is associated with the transitions as indicated on the following state diagram (Figure 6).

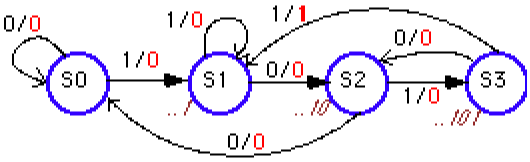


Figure 6: Sequence detector (1011), realized as a Mealy Machine.

The VHDL file is given below.

VHDL file for a sequence detector (1011) implemented as a Mealy Machine

```
library ieee;
use ieee.std_logic_1164.all;
entity myvhdl is
   port (CLK, RST, X: in STD_LOGIC;
            Z: out STD_LOGIC);
end;
architecture myvhdl_arch of myvhdl is
-- SYMBOLIC ENCODED state machine: Sreg0
type Sreg0_type is (S1, S2, S3, S4);
signal Sreg0: Sreg0_type;
begin
--concurrent signal assignments
Sreg0_machine: process (CLK)
begin
if CLK'event and CLK = '1' then
   if RST='1' then
      Sreg0 <= S1;
   else
   case Sreg0 is
      when S1 =>
         if X='0' then
            Sreg0 <= S1;
         elsif X='1' then
            Sreg0 <= S2;
         end if;
      when S2 =>
         if X='1' then
            Sreg0 <= S2;
         elsif X='0' then
            Sreg0 <= S3;
         end if;
      when S3 =>
         if X='1' then
            Sreg0 <= S4;
         elsif X='0' then
            Sreg0 <= S1;
         end if;
      when S4 =>
         if X='0' then
            Sreg0 <= S3;
         elsif X='1' then
            Sreg0 <= S2;
         end if;
      when others =>
         null;
   end case;
   end if;
end if;
end process;
-- signal assignment statements for combinatorial outputs
Z_assignment:
Z <= '0' when (Sreg0 = S1 and X='0') else
      '0' when (Sreg0 = S1 and X='1') else
      '0' when (Sreg0 = S2 and X='1') else
      '0' when (Sreg0 = S2 and X='0') else
      '0' when (Sreg0 = S3 and X='1') else
      '0' when (Sreg0 = S3 and X='0') else
      '0' when (Sreg0 = S4 and X='0') else
      '1' when (Sreg0 = S4 and X='1') else
      '1';
end myvhdl_arch;
```