UNITED NATIONS EDUCATIONAL SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL ATOMIC ENERGY AGENCY
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY. CABLE: CENTRATOM TRIESTE

SMR/943 - 3

ICTP-UNU-MICROPROCESSOR LABORATORY:
FOURTH COURSE ON
BASIC VLSI DESIGN TECHNIQUES
18 November - 13 December 1996

LOGIC DESIGN

Magali ESTRADA
Secc. de Electronica del Edo. Solido
Depto. Ingenieria Electica
Centro de Ivestigacione y de Estudios Avanzados del IPN
Av. IPN No. 2508
Sn. Pedro Zacatenco
07300 Mexico City
MEXICO

LOGICAL DESIGN

Dr. Magali Estrada del Cueto

Fourth Course on Basic VLSI Design Techniques
Laboratory of Microprocessors,
ICTP, Trieste, Italy

November 18 to December 13, 1996.

LOGICAL DESIGN

1.- GENERAL CONCEPTS

1.1.- Design methodology

The design methods of complex digital systems, have been object of intensive development through the last decade. At present, they are based on a well-define set of techniques, and described by such concepts as: **high-level, top-down and structured.**

Top-down design: is used to express that the design starts with the specification of the complete object of design in a compact form (global representation of the system).

This global representation at the top level is subsequently divided into sub-units. Each sub-unit and their interconnections, must be described in

more detail, and once again subdivided into new sub-units. This process will continue until the system is completely specified to its very finest details.

This structured or hierarchical, (by levels), top-down conception of a system, provides an organization that permits its implementation or description by means of well-established programming techniques, and the automation of the designing process, including synthesis, analysis, feedback and testing of the results. This feedback permits to incorporate bottom-up technical information, for the adjustment of critical parameters, and so, turn a top-down design into a high-level design.

Resuming, a **high-level design** incorporates both the **top-down plus the bottom-up designs**, and provides design capture, high-level and gate-level simulation, synthesis, verification of gate-level logic, generation of test vectors, among others.

1.1.1.- High level design: basic concepts and characteristics.

The high-level design methodology requires:

- **abstraction** - to conceive a global and then enter into its details;
- **formalism** - to have the rules and procedures for working at each level;
- **concepts** - so that everybody can understand the other;

It consists of a hierarchical structure of levels. The lower 4 levels are well defined and recognized in all publications on this subject. The upper levels are still in debate, and their quantity, as well as their denominations and definitions vary from one author to another. Without loosing in generality, we will consider for our analysis, that the design process consists of 6 levels of abstraction, shown in Table 1.1.

Level	Structural primitives	Behavioral representation
System	CPU, memories, ports, bus interface, etc.	performance specifications
Chip	processors, memories, ports, etc.	I/O response, algorithm
Registers	registers, counters, ALU, flip-flop, combinational logic, etc.	truth tables, state tables, data flow
Gates	gates, flip-flops, multiplexers, adders, counters, decoders, etc.	Boolean equations
Circuits	transistors, capacitors, etc.	differential equations
silicon	topology	none

Table 1.1. Levels of detail commonly used in design.

Each level of abstraction can be described by means of:

a) - **A structural domain** : when a component is described in terms of an interconnection of more primitive components.

b) - **A behavioral domain**: when a component is described by defining its input/output response, using a given procedure.

One can represent a design at any of these levels. Fig. 1.1.a shows an example of structural primitives representation at each level of abstraction. The lower levels are closer to the physical implementation, and it is intended for the designers to have little to do with them. Their influence on the design is included in the models for simulations for a given technology and firm.

For many years, the gate and register levels were the main levels for ASIC designers. At the gate level, AND, OR, and inverters are the elementary blocks used. Their interconnection give rise to more complex combinational and sequential functional blocks like XORs, COINCIDENCE, flip-flops, counters, adders, etc. Within this level of abstraction, we can create more complex blocks which we describe structurally as the interconnection of more primitive blocks, but no matter how complex we make these blocks, while we stand at this level, we will describe their behavior by means of the Boolean algebra. In this case, the creation of more complex blocks give rise to a hierarchy by complexity and a degree of nesting inside the level, which should not be confused with the above definition of hierarchical or structured levels into which we divided the design process.

At the register level, the main blocks, also called functional blocks, are composed of registers, counters, multiplexers, etc., which can also be used at the gate level, but their behavioral representation is expressed at this level by means of truth tables, and state tables, and not by the interconnection of gates. **The gate and register levels are tightly related to the HW implementation**. These behavioral representations are also termed **data flow**, since they reflect the way data is actually distributed in real implementation. These flow description can be implemented by means of a HW description language (HDL).

For the chip level, the structural primitives, (microprocessor, memories, ports, controllers, etc.) are still more complex blocks, which behavior is described by single model entities instead of by the interconnection of more simple primitives. The models describe the procedure they execute. For example, an I/O port will not be described as the interconnection of registers and counters, but as the algorithm the device executes. This level is also called Register Transfer Level (RTL). It tells not only the "what" of the design, but also in a good deal the "how", still having the advantage of being **technology-independent**.

In the system level, again the functional blocks can be some of the ones found at the chip layer, or others more complex. The main difference lays again in the form the behavioral representation is made. In this case the system is described through its performance. For example, the behavioral content at this level describes the MIP ratio of a processor, or the bandwidth in bits per second of a data path. This level says only what a design is supposed to do functionally. The expression of the behavior of components at this level are also called behavioral description. Since the behavioral synthesis is still emerging, not all the high-level CAD systems include it. This means that in many cases the transformation of the behavioral description into the RTL has to be done manually. (Be careful not to confuse the behavioral representation in Table 1.1, with the term "behavioral description" related to the system level.)

The expression of the behavior of components at each level may be represented in a pictorial or textual form. Block diagrams, timing diagrams, state diagrams and truth tables are considered pictorial representations, while equations, natural languages, or specialized languages are considered textual. For the last years, the introduction of a hardware description languages for high level programming, specially constructed for describing and modeling hardware, has become the key for incrementing the productivity of designers, as well as the certitude that the system will function correctly. One of these languages is called **VHDL [Very High Speed (VHSIC) Hardware Description Language].**

Fig. 1.1.c shows the different ways of presenting the behavioral representation of a circuit which detects two consecutive ones or two consecutive zeros, referred to as TWO_CON, (see Fig. 1.1.b), in a pictorial way.

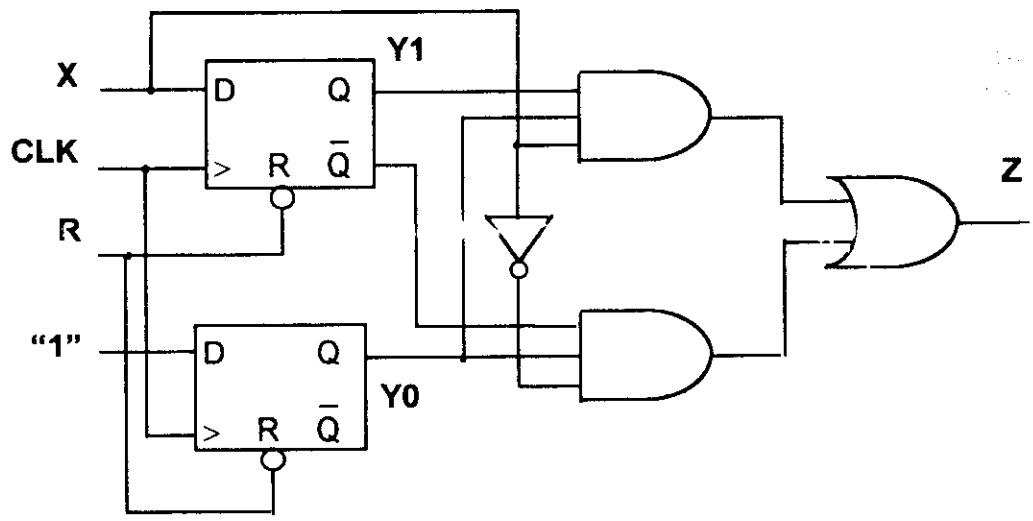
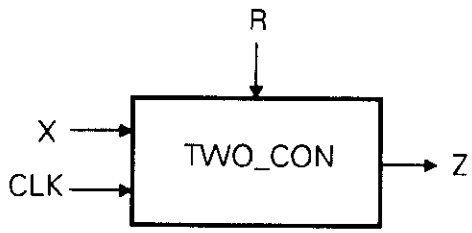
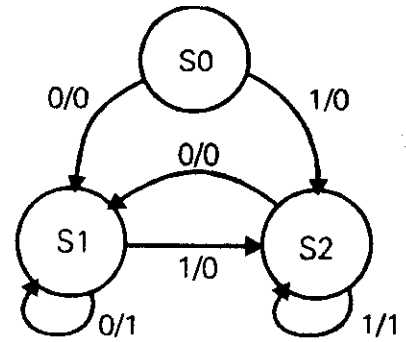


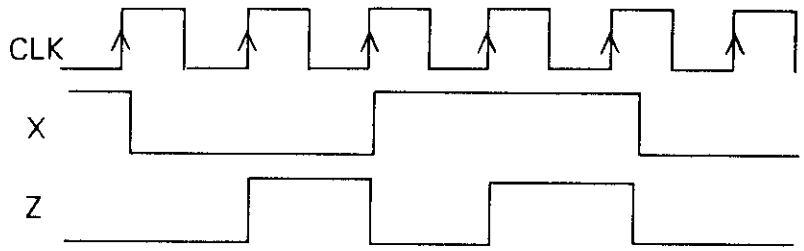
Fig. 1.1.b. Schematic of circuit TWO_CON



Block Diagram



State Diagram



Timing Diagram

	X			code
state	0	1	state	y1y0
S0	S1/0	S2/0	S0	00
1	S1/1	S2/0	S1	01
S2	S1/0	S2/1	S2	11

State Table **State Assigment**

code	X		code	X		code	X	
y1y0	0	1	y1y0	0	1	y1y0	0	1
00	0	1	0u	1	1	0u	0	0
01	0	1	01	1	1	01	1	0
11	0	1	11	1	1	11	0	1
10	-	-	10	-	-	10	-	-

Y1 **Y0** **Z**

Truth tables

1.1.c Pictorial representations of logic circuits

Resuming, behavioral descriptions in HDL can be represented in the following ways:

- **Algorithmic**: a behavioral description in which the procedure defining the I/O response is NOT meant to imply any particular physical implementation. An algorithmic description models the behavior of a device, to check that it is performing the correct function, without worrying about how it is to be built. Here we are at the chip level.
- **Data-flow:** a behavioral description in which the data dependencies in the description match those in a real implementation. Here we are at the register level.

Fig. 1.1.d shows the behavioral representation using VHDL for a data flow description, while Fig. 1.1.e presents the algorithmic representation for the TWO_CON circuit of Fig. 1.1.b.

```
entity TWO_CONSECUTIVE is
    port(CLK,R,X: in BIT;Z: out BIT);
end TWO_CONSECUTIVE;

architecture DATAFLOW of TWO_CONSECUTIVE is
    signal Y1,Y0: BIT;
begin
    STATE: block((CLK = '1'and not CLK'STABLE) or R = '0')
    begin
Y1 <= guarded '0' when R = '0' else X;
Y0 <= guarded '0' when R = '0' else '1';
    end block STATE;
    Z <= Y0 and ((not Y1 and not X) or (Y1 and X));
end DATAFLOW
```

Fig 1.1.d Dataflow representation of TWO_CON circuit.

```
architecture ALGORITHMIC of TWO_CONSECUTIVE is
        type STATE is (S0,S1,S2);
        signal Q: STATE := S0;
begin
        process(R,X,CLK,Q)
        begin
                if (R'EVENT and R = '0') then                   --reset event
                        q  <= S0;
                elsif (CLK'EVENT and CLK = '1') then            --clok event
                        if X = '0' then
                                Q <= S1;
                        else
                                Q <= S2;
                        end if;
                end if;
                if  Q'EVENT or X'EVENT then                     --output function
                        if (Q=S1 and X='0') or (Q=S2 and X='1') then
                                Z <= '1';
                        else
                                Z <= '0';
                        end if;
                end if;
        end process;
end ALGORITHMIC;
```

Fig. 1.1.e Algorithmic description of TWO_CON circuit

1.1.2.- Design Process: different approaches and main steps

Design: will be defined as a series of transformations from one representation of a system to another, until a representation exists, that can be fabricated.

The design process involves synthesis.

Synthesis: is the process of transforming one representation in the design abstraction hierarchy, to another representation. Each step in the design process can be referred to as a synthesis step.

The synthesis process will end with the physical realization of a logic, starting from its description; while **analysis** is the inverse process of obtaining the behavioral description of a given circuit from its physical representation.

Each synthesis step will involve the transformation from level i to level j, with $i \leq j$. If i = j, the synthesis will involve the transformation of a behavioral domain to structural, at the same level.

A design process consists of the following transformations (synthesis steps):

1. -Transformation from a language representation, to an algorithmic representation (language synthesis);
2. - Translation from an algorithmic representation to a data flow or a gate level representation (algorithmic synthesis);
3. - Translation from data flow representation to a structural logic gate representation (logic synthesis); at the same time the behavioral domain transforms into a structural domain;
4. - Translation from a logic gate representation to layout representation (layout synthesis); Note that the circuit level has been skipped.

Getting into the details, a high-level design starts with the same specifications as any traditional design, and is based on:

- function to execute
- quality (fault coverage)

- cost (die size/area)
- timing constraints (global; i.e. 20 MHz)

In Tables 1.2 to 1.5 we show the steps followed in different design approaches, starting from the case (Table 1.2) where only synthesis from gate level to topology is realized, and ending with a high-level HDL-based design. In each case, we show in black letters, the new steps incorporated from one to another approach. All of them include gate level simulation.

Table 1.2. Schematic design without synthesis

1. Architecture is specified and partitioned into functional blocks.
2. Schematic entry is used for design capture on a computer aided engineering (CAE) system, and further partitioning takes place.
3. Function, performance, and area of design blocks are monitored simultaneously through gate-level simulation and analysis tools.
4. As blocks are completed, they are integrated together and the interface between them is verified.
5. Complete design validation takes place (gate level simulation).
6. Design goes to place and route stage.
7. Final gate level verification is performed.

Approach 2 (Table 1.3) already incorporates synthesis at register and gate levels, and so, the possibility of optimization by parameters as time and number of elements. It also provides analysis.

Table 1.3. Schematic design with synthesis and optimization

1. Architecture is specified and partitioned into functional blocks.
2. Schematic entry is used for design capture on a computer aided engineering (CAE) system, and further partitioning takes place.
3. Function, performance, and area of design blocks are monitored simultaneously through gate-level simulation and analysis tools.
4. **Design blocks are transferred into the synthesis tool and optimized, quickly producing multiple implementations of a design by trading off speed and area.**
5. As blocks are completed, they are integrated together and the interface between them is verified.
6. **Complete design can be optimized using synthesis.**
7. **Synthesis results can be analyzed either in the synthesis environment or in the host CAE environment.**
8. **If you are satisfied with results, you can replace the original block with the optimized version.**

9. Complete design validation takes place (gate level simulation).
10. Design goes to place and route stage.
11. Final gate level verification is performed.

The next approach, (Table 1.4), already includes other types of entries as netlist, which is the description of the gate interconnections; Boolean equations; states tables, and other HDL options.

Table 1.4. Design with partial schematic and HDL entries, synthesis and optimization

1. Architecture is specified and partitioned into functional blocks.
2. **Part** of the design is captured using schematic entry.
3. **The rest of the design is captured using other methods, all of which can be used to create a netlist. Theses methods can include: existing netlist/schematics (from CAE vendors), Boolean equations, state tables, VHDL, and Verilog. Synthesis can be used to automatically create a schematic. The first types of hardware modules typically chosen to be captured in HDL are state machines and control logic.**
4. Function, performance, and area of design blocks are monitored simultaneously through gate-level simulation and analysis tools.
5. Design blocks are transferred into the synthesis tool and optimized, quickly producing multiple implementations of a design by trading off speed and area.
6. As blocks are completed, they are integrated together and the interface between them is verified.
7. Complete design can be optimized using synthesis.
8. Synthesis results can be analyzed either in the synthesis environment or in the host CAE environment.
9. If you are satisfied with results, you can replace the original block with the optimized version.
10. Complete design validation takes place (gate level simulation).
11. Design goes to place and route stage.
12. Final gate level verification is performed.

The approach in Table 1.5 will be the final objective to achieve. It can include also, test synthesis. It is usually accepted by designers when, over time, they gain confidence in the synthesis tools and high-level methodology. The pass from previous design methodologies to full high-level is not only related with the capabilities of the tools the designers have or have the possibility to acquire. It is also a way of thinking. The designer must be aware of its advantages, and have confidence in the results obtained.

Table 1.5. Full HDL-based design with synthesis

1. Architecture is specified and partitioned into functional blocks.
2. **Entire design is entered via HDL.**
3. **Design is further partitioned in HDL.**
4. **Design is validated with an HDL simulator.**
5. **Design is translated into gates.**
6. **Gate level blocks are optimized via synthesis**
7. Complete design validation takes place (gate level simulation).
8. **Complete design is optimized using synthesis and analysis.**
9. Design goes to place and route stage.
10. Final gate level verification is performed.

Test synthesis tools automate design for test, so you can build tests into your design automatically, as the design is being created. This can cover both functional testing and automatic test pattern generation (ATPG) to create fault coverage test patterns. Testing is a real complex procedure and the aspects concerning this field are usually included in a thematic called **Design for Testability.** Unfortunately most design systems still today, do not fully incorporate these facilities.

Even if the design cycle can be carried out practically automatically in all the stages, nevertheless a design engineer must know in some degree what is done at each step in order to "help" the designing tools, and at one of the levels, he must specify what the object of design must do.

In order to be able to do this, he must have the necessary knowledge of logic design. The next paragraphs will be dedicated to review some general concepts and rules of logic design, as well as some methods that give better results in preparing the information that must be entered via HDL in the design tool.

1.2.- Logic constants and variables, logical operators and logic primary elements.

The state of a logical statement, is described by one of two possible values or conditions: it can be true or false. The logic constants define the condition or state of a given logical statement. If we want to describe that the radio is working, we can use the variable RA for representing the phrase "the radio is working", and a constant TRUE to specify that it is working, or FALSE, to specify that it is not working.

For example:

RA = TRUE
or RA = FALSE

If we want that if both, the radio AND the television are working, some action is to be executed, we can verify if this condition is TRUE, in order to execute a predetermined action.

In a similar way, if we want to do something when either the radio OR the television is working, we can verify if the given condition is TRUE. The function of the element "NOT", is to negate a statement. For example, Mary arrived is converted to Mary did NOT arrive.

The different conditions to verify can be implemented starting from these three logic primary elements: NOT, AND, OR.

In addition, two more functions will be included in our set of basic building blocks; they will represent the concepts of "different" (EXCLUSIVE OR, EOR, XOR), and "same" (COINCIDENCE or EQUIVALENCE). The first is true only if its inputs have different logical values, while the second is true only when all inputs are the same (all are true or are false).

Each of this actions is represented by means of an OPERATOR. There are several ways of representing these operators. We will use the following notation:

NOT RA = $\overline{RA}$
B AND C = B * C
A OR B = A + B
EXCLUSIVE OR (XOR): $A \oplus B = \overline{A} * B + A * \overline{B}$
COINCIDENCE: $A \odot B = \overline{A} * \overline{B} + A * B$

In order to implement these logical operations in HW, there is a logic primary element corresponding to each of these operators.

1.3 Truth tables and logic equations

They are three forms of representing the expected logical values of a function for different conditions of the input variables: the TRUTH TABLES (or EXCITATION TABLES), LOGICAL VECTORS and the LOGICAL EQUATIONS.

The truth tables (TT), describe the expected values of a function in a tabular form, for different conditions at the input. They are said to " canonical",

if the rows are represented in binary notation (number of rows = 2^n), starting from all zeros, and incrementing each by one, for example:

ROW	A	B	W
0	F	F	F
1	F	T	F
2	T	F	T
3	T	T	F

where A and B are the input variables and W the function described.

The logical vectors (LV), indicate the variables of the function, and at the right in parentheses are indicated the values of the function for each row, separated by commas. For example:

$$W(A,B) = (0,0,1,0)$$

is the logical vector of the above truth table.

The logical equation (LE) describes through the Boolean algebra, how function W depends on A and B, in this case:

$$W = A * \overline{B}$$

2.- ELEMENTS OF BOOLEAN ALGEBRA

2.1.-Properties of operators:

COMMUTE	A + B = B + A
ASSOCIATE	A + (B + C) = (A + B) + C
DISTRIBUTE	A *(B + C) = (A*B)+(A*C)
	A+(B*C) = (A+B)*(A+C)

The operations are to be executed in the following order:

First NOT, then XOR and COINCIDENCE, then AND, and then OR. Parentheses override the normal hierarchy.

The expressions can be written as:

1) SUM OF PRODUCTS

$Y = A*B + B*C$
$X = A + B$

2) PRODUCTS OF SUM

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

2.2.- FUNDAMENTAL RELATIONS

For sums of products	For products of sums
$A + F = A$	$\overline{\overline{A}} = A$
$A + T = T$	$A*T = A$
$A + A = A$	$A*F = F$
$A + \overline{A} = T$	$A*A = A$
$\overline{A + B} = \overline{A} * \overline{B}$	$A * \overline{A} = F$
$\overline{A + B + C} = \overline{A} * \overline{B} * \overline{C}$	$\overline{A * B} = \overline{A} + \overline{B}$
	$\overline{A * B * C} = \overline{A} + \overline{B} + \overline{C}$

The last two expressions result in the principle of duality, or the Morgan's Law. These relations are very important from the designing point of view, as they permit the conversion of AND elements, to ORs, and vice versa.

Other identities used in simplifications are:

For sums of products	For products of sums
$A*(A+B) = A$	$A+A*B = A$
$A * (\overline{A} + B) = A * B$	$A + (\overline{A} * B) = A + B$
	$A * B + \overline{A} * B = B$

2.3.- Rules for manipulation

There are several simple rules to obtain the logical equations for a function represented in form of a TRUTH TABLE (TT). For example:

a) To derive a logical equation in the form of SUM OF PRODUCTS, for a function represented in a canonical TT, write the OR of the MINTERMS for which the function is TRUE.

A MINTERM is a canonical product term with all variables on it. For n variables, there are 2^n possible MINTERMS m_i, where i is the row for which it is TRUE.

b) To derive a SUM OF PRODUCTS form for the complement of a function, write the OR of the MINTERMS for which the function is FALSE.

c) To derive a PRODUCT OF SUMS of a function from a canonical TT, write the AND (product) of the opposite of each MAXTERM for which the function is FALSE.

A MAXTERM is the term that contains one occurrence of every variable. If all the terms of a PRODUCT-OF SUMS are MAXTERMS, the product-of-sums is canonical.

d) To derive a product-of-sums of the complement of a function from a canonical TT, write the AND of the opposite of each MAXTERM for which the function is TRUE.

Similar rules can be used to obtain TT from logical equations.

a) For the SUM-OF-PRODUCTS, a MINTERM will yield one row with an output TRUE. A product term with fewer variables yields more TRUE rows, since it is true for any value of the missing variable.

For example, the equation $Y = J*\overline{K} + \overline{J}*K*L + J*K*\overline{L} + K*L$ will produce the following canonical TT:

ROW	J	K	L	Y
0	0	0	0	F
1	0	0	1	F
2	0	1	0	F
3	0	1	1	T Due to terms 2 and 4
4	1	0	0	T Due to term 1
5	1	0	1	T Due to term 1
6	1	1	0	T Due to term 3
7	1	1	1	T Due to term 4

b) For the PRODUCT-OF SUM, each sum will assure a false expression whenever all its variables are the opposite of the form in the term.

In the example below, the second term is not canonical, so it will yield two FALSE rows:

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

ROW	A	B	C	G
0	0	0	0	T
1	0	0	1	T
2	0	1	0	T
3	0	1	1	T
4	1	0	0	F Due to terms 1 and 2
5	1	0	1	F Due to term 2
6	1	1	0	T
7	1	1	1	F Due to term 3

Another procedure will be to condense a TT when both values of the logic constant at one input produce the same output.

2.4.- Karnaugh maps

It is another form of representing a truth table, specially useful for simplifying Boolean equations. They are easy to use, when dealing with 4 or less variables. They are also useful if you have to prevent the presence of glitches.

How to convert a truth table into its Karnaugh map:

1.- Each square in the K-map corresponds to a row of a truth table; each combination of variables identifies a square in the map.

2.- The first and second variables (if more than two) are the labels for the horizontal squares of the first row. The second variable (if only two) or the third an fourth (if more than 2), are the labels for the vertical squares for the first column.

3.- Each other square contains the value of the function Y corresponding to the appropriate truth table row, as specified by the labels on the edges of the K-map.

4.- The order of the labels are so, that when moving from square to square across a row (or column), the value of only one variable changes at a time.

For example:

A	B	C	D	Y
0	0	0	0	**Y0**
0	0	0	1	**Y1**
0	0	1	0	**Y2**
0	0	1	1	**Y3**
0	1	0	0	**Y4**
0	1	0	1	**Y5**
0	1	1	0	**Y6**
0	1	1	1	**Y7**
1	0	0	0	**Y8**
1	0	0	1	**Y9**
1	0	1	0	**Y10**
1	0	1	1	**Y11**
1	1	0	0	**Y12**
1	1	0	1	**Y13**
1	1	1	0	**Y14**
1	1	1	1	**Y15**

AB / **CD**	**00**	**01**	**11**	**10**
00	Y0	Y4	Y12	Y10
01	Y1	Y5	Y13	Y11
11	Y3	Y7	Y15	Y9
10	Y2	Y6	Y14	Y8

To write a logic equation from a K-map, each isolated 1, produces one of the product minterms of the sum of products. If there are adjacent ones, simplification is possible.

How to simplify with K-maps:

- Draw circles among adjacent ones. You can have one, two, four, eight,... ones.
- Circling two ones, causes two canonical terms to collapse in one term; the variable that changes, when passing from one to the adjacent "one", drops out.

- Circling four ones, causes four terms into one, eliminating the two variables that change.
- Circling eighth ones, causes eighth terms into one, eliminating the three variables that change.

Some examples are shown in Fig. 2.1.

3.- REALIZING LOGIC IN HARDWARE

3.1. HW representation of logic constants; logic conventions.

In order to use the logical relationships mentioned above, it is necessary to find a physical way of implementing the fundamental logic constants TRUE and FALSE.

The simplest way to represent these constants is by means of a switch that can be in one of two states, closed or open. In digital electronic circuits, these logic constants are usually represented through the parameter "VOLTAGE". By assigning one or two predetermined levels, you can consider that if the voltage is greater than the higher level, one of the logic states is represented. If it is less than the lower level, the other logic constant can be represented.

For example, the well known TTL (Transistor-transistor logic) integrated circuits of the 74LS family, produces two voltage levels: < 0.5V for the low level (L or logic 0) , and > 2,7 V for the high level (H or logic 1).

The CMOS integrated circuits of the family 74HC, for a 5 V power supply, produce the levels: < 0.9 V for the low level and > than 3.15 V for the high level.

Any of these levels can be associated to the TRUE or FALSE. The digital integrated circuits mentioned above, as well as some others, are called standard digital circuits, since they are used for general purposes, to build complex electronic board. They are designed on the basis of more elemental circuits, named "primitive gates", that perform the basic logical functions NOT, AND, OR described in chapter 2.

These primitive gates, are also used as elemental blocks for designing the Application Specific Integrated Circuits, (ASIC), so we will analyze in more detail how to work with them. They are part of the vendors library, as well as other more complex gates and functional blocks, which can be prepared on their basis.

The two logic levels can be represented in two ways by the voltage levels:

When TRUE is always associated to the High level, the logic is called positive. If TRUE is associated to the LOW level, the logic is called negative. When one of these relationships is used consistently throughout a complete design, it is said that the design uses the positive-logic convention, or the

negative-logic convention. If both the positive and negative convention are used at different parts of the same design, it is said that the mixed-logic convention is used.

3.2.- Mixed logic: its representation and theory; analysis and synthesis.

First of all we will show how these elemental blocks (gates), are represented during the design and documentation. Although at present, some different notations can be used for the same purpose, we selected the mixed logic representation, since it has some interesting features that will be presented in the text below. This notation fulfills the following requirements:

1. It represents the Boolean expressions in AND, OR and NOT form, which is the natural way we develop our logic.
2. The correspondence between a logical value (TRUE or FALSE) and its voltage implementation (H or L) is quite evident everywhere in the circuit diagram.
3. The notation clearly identifies each physical device in the circuit.

The mixed logic notation was first published by [Kintner, P.M., Computer Design, August 1971, pp 55), although the technique is somewhat older.

The symbols in Fig. 3.1 are used for the basic operations NOT, AND, OR, EXCLUSIVE, COINCIDENCE.

Inputs to the symbols are connected to the left, and outputs, to the right.

For example the notation in Fig. 3.2 a and b will represent the logic equations Z = X * PDQ and XYZ = A + B.

It must be noticed that each graphic symbol implies a physical device that performs a logic operation.

When T = L, we represent it by a small circle on the corresponding terminal of the logic symbol. The absence of a small circle means that T = H at that point. It is important to remark that in the mixed logic notation, the circles DO NOT CHANGE the logic operation.

Since we know the truth table (given by the symbol's shape) and the voltage representation of the truth on each input and output (by the presence or absence of circles), we can immediately write down the voltage table for any symbol. Then referring to a data book for integrated circuits or library parts, we can identify the device.

If a signal has T = L, we will append a terminal .L to the logic variable's name.

If a signal has T = H, we will append a terminal .H to the logic variable's name.

In Fig. 3.3, there are 4 possible choices for the representation of the voltage at the input and output of this functional gate representing an identity. In this case it is an evident result. It is interesting to notice that in case T= H at the input and T = L at the output, the identity function is realized by means of a voltage inverter, represented by the triangle with a single circuit at the input or output.

In the case of Fig. 3.4 we must consider that the logical NOT implies that the input variable MUST BE INVERTED. Care must be taken to notice that, since the symbol for the logical NOT includes a circle at the output, when the voltage representation at the output is H, a complementary circle must be included.

Similarly, if the True at the input is low, a circle at the input must be incorporated. Also, since the operation converts a T in F, if we define that T= H at the output, the implementation of the voltage relationship that, if the input is T the output is F, is indicated only by a voltage inverter; if the T value at the output is L, then the T at the input is H and the F at the output is H also, so the logic inverter is implemented by a wire and no voltage inverter is required.

This interesting feature of the mixed logic representation implies that, while positive or negative logicians have only one way to implement a logical NOT (through the logical NOT), the mixed logician has 4, two using a piece of hardware, and two just using a wire that connects input with output.

Since a logical NOT can be generated without a device, the mixed logic notation requires for a symbol to indicate logical inversion. This is made by means of a slash.

Something similar is obtained for the logical AND, Fig. 3.5, and for the logical OR, Fig. 3.6. In the last two, the number of possible symbols that represent the same truth table is 8.

It will be important to determine to which physical devices they are associated. For example the first and second representations of the AND in Fig. 3.5 can be implemented by means of a AND, and NAND physical device respectively, while the last two by means of the NOR and OR physical devices. This is a very important feature of the mixed logic representation.

In general a Boolean function of two input variables may have 16 different output functions, some of which we have already dealt with. Its TT is the following:

A	B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
F	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T
F	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
T	F	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T
T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T

The outputs execute the following functions:

Z0 = F
Z1 = A AND B
Z2 = NOT (A IMPLIES B)
Z3 = A
Z4 = NOT (B IMPLIES A)
Z5 = B
Z6 = A XOR B
Z7 = A OR B
Z8 = NOT(A OR B)
Z9 = A COINCIDENCE B
Z10 = NOT B
Z11 = B IMPLIES A (If B = T output = A)
Z12 = NOT A
Z13 = A IMPLIES B
Z14 = NOT(A AND B)
Z15 = T

Knowing this generalize truth table for two input functions, it is possible to determine new possibilities for the different physical devices to implement mixed-logical functions. For example, in the case of the physical NAND gates, they can implement the mixed logic representations shown in Fig. 3.7. Something similar occurs with the rest of the physical devices. It is this characteristic of the mixed logic convention that provides its main advantage of presenting the logic in a way that allows the reader to retrieve the designer's original expression.

For analyzing a logical expression from a circuit you must:

1. - ignore circles and inverters, since they perform no logic by themselves);
2. - interpret the slash as logical NOT, and the others AND, OR, XOR and COINCIDENCE symbols as the logical operations thev implement, and derive the logic expression from the diagram. For the example, the resulting logic equation from analyzing the circuit in Fig. 3.8 is:

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

As you see, the process of synthesis, that is the creation of a physical realization of a logic, starting from its description; and the analysis, consisting in obtaining the logical expression that describe a given circuit from its physical representation, are quite adequately implemented using the mixed logic notation.

For comparison, let us analyze a similar process when using positive-logic convention. In this case the T = H and the F = L everywhere. The symbols are fixed to a truth table, that means that each symbol corresponds only to one truth table. In Fig. 3.9 are shown the examples for the NAND, inverted-input NOR, inverted-input AND, NOR; AND, NOR, inverted-input NAND, and OR, implementing the 4 of the 8 different truth tables for the AND, and the OR in the mixed logic. In this logic, the small circles do represent the logical inversion operation.

How can a mixed logician read a positive logic circuit?

Transform graphically, the positive-logic convention into mixed logic. To do this, append .H to the positive logic inputs and output. Replace the negated input or output by non-negated mixed-logic forms. When the circles do not match at the ends of a line, insert a slash to emphasize the implied logical NOT. Where a gate is surrounded by slashes, you may simplify the solution by altering the AND, and OR gate symbol to its mixed-logic OR and AND counterpart. This is an application of De Morgan's law, and on the diagram the result is an inversion of circles and a change of the logic symbol to its dual. The circle inversions require rectification of the slashes on the gate input and output lines, leading to a simpler circuit.

3.3. Common building blocks

Logic theory shows that all digital operations may be reduced to elementary logic functions, but in this case, a complex digital system had to be treated as a huge collection of AND, OR and NOT gates, very difficult to understand. A SW analogy would be comparable to programming only in binary machine language. The concept of structured design allows the use of commonly used blocks, inserted in the available library as basic elements, but which are constructed on the bases of the already studied elementary gates. These blocks are at least one level of abstraction higher than the elementary gates.

Among the common operations to be performed when designing, we have the followings:

a) movement of data from one part of the system to another;
b) selection of the given data from several possible;
c) routing data from a source to one or several destinations;
d) transformation of the data from one representation to another;
e) comparing data arithmetically with another data;
f) manipulation of the data, arithmetically or logically, for example, the sum of two binary numbers;

The building of these blocks, allows us to suppress much irrelevant detail and design at a higher level.

3.3.1 Combinational blocks

Combinational blocks are those which outputs depend only on the present value of the inputs. The followings are examples of combinational blocks, present in almost every library for the design of ASICs.

1. - **Multiplexer**; permits to select one of several possible input signals, which is transferred to its output. A Boolean equation for describing this circuits will be:

$$Y = G*(A*\bar{S} + B*S)$$

If we need to select an output among more than two inputs, the number of select inputs should be greater than one. Each select line can manage 2 inputs, so if we have s select lines, we can manage 2^s inputs. The multiplexers can differ in the voltage representation of the true at the inputs and at the output, in the number of inputs, in having enable input or not, etc. When it is necessary to select, look up or address one of a small number of items, the multiplexer is a good solution. Several multiplexers can be addressed by a common signal, forming a multibit lookup.

2. - The **demultiplexer**: it sends data from a single source, to one of several destinations. It is a data distributor or a data router. Below we have the logic equations and the truth table corresponding to a 4 output demultiplexer.

$$Y0 = \bar{B}*\bar{A}*G$$
$$Y1 = \bar{B}*A*G$$
$$Y2 = B*\bar{A}*G$$
$$Y3 = B*A*G$$

Demultiplexer logic

G	B	A	Y0	Y1	Y2	Y3
F	X	X	F	F	F	F
T	F	F	T	F	F	F
T	F	T	F	T	F	F
T	T	F	F	F	T	F
T	T	T	F	F	F	T

3. - The **decoder**: associates an encoded representation of a set of items at the input, to one of the output signals, in other words it identifies a particular code, for example a BCD-to-decimal decoder. Another important use is for decoding the operation codes of a processor.

For example the TT for the BCD to decimal decoding is the following:

Output valid	Input values D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

The operation codes of the following TT can be decoded by means of a 4-to-10 lines decoder, or a 3-TO-8 lines with enable. Each operation code (input logic variables) is translated into the output variables (instructions).

Code	C	B	A	INSTRUCTION
0	0	0	0	AND
1	0	0	1	$TAD = \overline{C} \cdot \overline{B} \cdot A$
2	0	1	0	ISZ
3	0	1	1	DSA
4	1	0	0	$JMS = C \cdot \overline{B} \cdot \overline{A}$
5	1	0	1	JUMP
6	1	1	0	DOT
7	1	1	1	OP

4. - The **encoder**: forms an encoded representation of a set of inputs. An N-bit code is generated depending upon which of the inputs is excited. The Boolean notation and truth table for encoding the decimal numbers 0 to 9 are shown below.

Inputs										Outputs			
W9	**W8**	**W7**	**W6**	**W5**	**W4**	**W3**	**W2**	**W1**	**W0**	**Y3**	**Y2**	**Y1**	**Y0**
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

5. - **Code converter:**

One example of this circuit is he BCD to seven-segment code converter. It is very easily done, using a 16 word by 7 bits ROM with 4 address inputs consisting of the BAD code. The truth table of this code converter is the following:

0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10-15	-	-	-	-	-	-	-	-	-	-	-

For implementing this code converter you can use combinational logic, but you can also use a **ROM**. In this case each row of the TT will correspond to an address of the ROM. The table will consist of a 16 word memory, each word of 7 bits and a 4-bit address.

In general a ROM can be used to implement an arbitrary logic function. Since the ROM is a canonical structure, when used to synthesize logic functions, only a part of it is really used.

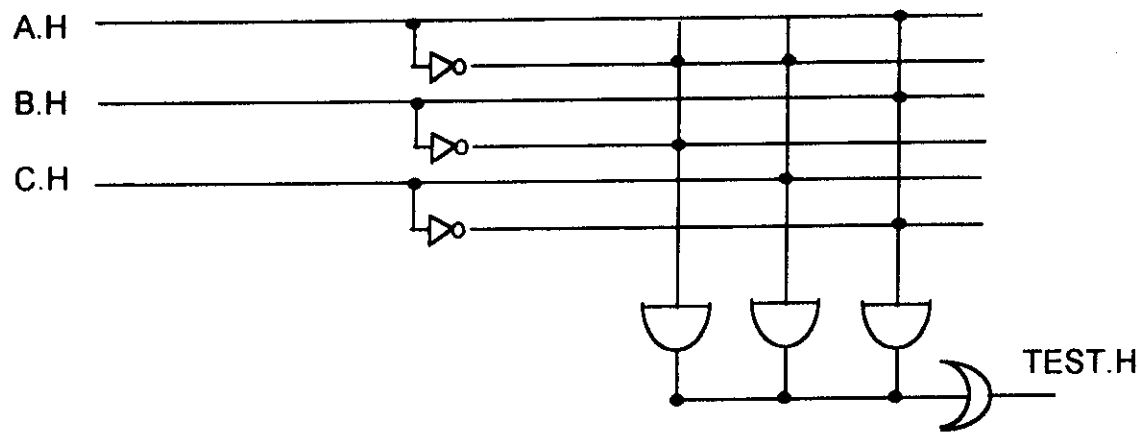
There are other structures that can be also used.

a) **PLA** stands for Programmable Logic Array. It expresses the logic function as a sum-of-products. The matrix is formed on the basis of ANDs and ORs and are used to generate complex logical functions. For example a PLA of 3 inputs will accomplish on each row the function

$$A * \overline{A} * B * \overline{B} * C * \overline{C}$$

Programming is made by connecting only the necessary elements.

The function $TEST = \overline{A} * \overline{B} + \overline{A} * C + A * B * \overline{C}$ is accomplished by the connections shown below:



b) **PLE** (Programmable Logic Element) is a PROM that provides all possible product terms of its input. To generate a logic function, program a "1" if the canonical term contributes, and a "0" if not.

c) **PAL** (programmable Array Logic) allows the designer to specify the nature of the product term. The way in which the product may be form into sums is fixed in the chip.

Since the actual tendency is toward accomplishing as most regularity in the design as possible, these features for implementing logic are widely used.

6.- **Comparator**: verifies the coincidence of two patterns of n input bits. The function is described by the logical equation:

A·EQB = (A0 ⊙ B0)*(A1 ⊙ B1)*.....*(An ⊙ Bn)

The implementation may be as shown in Fig. 3.10.

The comparator may also verify that a digit is greater than or smaller than another.

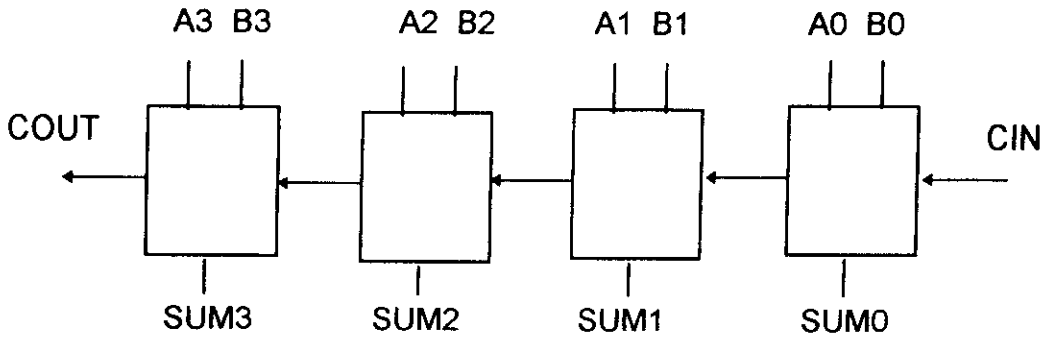
7.- **Universal Logic Circuit**: We already saw that a 2-input Boolean function can have 16 different outputs. If we consider a block with 4 input control signals to select one of these functions we are dealing with the universal logic circuit.

8.- **Full adder**: is described by the following Boolean expression for the sum and carry bits:

$$SUM = \overline{CIN} * \overline{A} * B + \overline{CIN} * A * \overline{B} + CIN * \overline{A} * \overline{B} + CIN * A * B = A + B + CIN$$

$$COUT = \overline{CIN} * A * B + CIN * \overline{A} * B + CIN * A * \overline{B} + CIN * A * B = A * B + CIN * (A + B)$$

A 4-bit full adder is shown below. Although this simple block performs the required operations, there are several different approaches that include speeding techniques.



9.- **Arithmetic Logic Unit** (ALU) combines the universal logic circuit with a general set of binary arithmetic operations.

The basic arithmetic operations include:

addition:	A PLUS B
subtraction:	A PLUS (MINUS B), where MINUS A is realized by the negation (complement) of B.
incrementing:	A PLUS 1
decrementing:	A MINUS 1
	A PLUS A
	B MINUS A
	MINUS A
	MINUS B

and others not exceeding in total 16 operations, which means that 4 control inputs are enough for their selection.

Since we already saw that the Universal Logic Unit requires also 4 control inputs, adding one more to determine when we are dealing with logical operations and when with arithmetic will be enough to perform the 16 logic functions of two variables plus the 16 arithmetic functions. In addition our block will have 2 4-bit inputs, 1 4-bit output, CIN, COUT.

Propagation delay effects.

The propagation delay of the signals, when they pass across the gates, must always be taken into account. They give rise to spurious outputs, called hazards or glitches, which must be overcome, usually waiting for a given amount of time.

For example, if we introduce the time effects when we analyze the circuit in Fig. 3.11, we can see in the waveforms corresponding to the input and output signals, the presence of a spurious pulse that lasts one gate delay. For combinational circuits, if we wait sufficient time, (more than the expected delay time), these spurious outputs will disappear and the outputs of the gates will assume the values predicted by the Boolean algebra.

The use of Karnaugh maps provides another tool for cleaning the circuits of these spurious pulses. It can be shown that a function having two adjacent ones that do not share a common circle may have a hazard. Building circuits for which the adjacent ones are all included in common circles overrides these problems, but as the circuit becomes more complex, the solutions also become complex, so the first method that consists in waiting for a fixed time after the inputs change, so that the hazards die out, is the basis of the synchronous (clocked) design, used in current design methodologies.

3.3.2 Sequential blocks:

These blocks operate in synchronism with a train of pulses. The value of the outputs after the setting time, depends not only on the external inputs, but also on the original value of the outputs. This becomes possible because of feedback connections of the output to some inputs.

In the case of sequential circuits, the presence of these time effects in feedback connections, give rise to memory effects. The value stored in the "memory" can be controlled through the external inputs. The most common sequential building blocks are: latches, flip-flops, and registers. As in the case of combinational logic, they are conformed using the elementary gates.

3.3.2.1. Latch

In Fig. 3.12 is shown the diagram for a latch. Its behaviour is described for the following two cases.

Case A: HOLD=F, Y=DATA

Case B: HOLD = T. Any occurrence of DATA = T will be capture, and the output will thereafter remain true until HOLD becomes false. We consider 3 subcases:

Case B1: Data is false throughout the period when HOLD is true. Then Y is false.

Case B2: Data is true when HOLD is true. When HOLD becomes true, the latch captures the true value of DATA and stores it as long as HOLD remains true. After HOLD becomes false, case A applies.

Case B3: Data is false when HOLD becomes true. At the beginning, Y is false. The first occurrence of a true signal on the DATA line will cause Y to become true; the output will remain true until HOLD becomes false.

RS flip-flop

It is a bistable device. Its behavior is described by saying that the circuit is in a stable state when gate 1 outputs L and gate 2 outputs H. Once the circuit assumes this state, it will remain on it as long as there are no changes in R and S inputs. There is another stable state during which gate 1 outputs H and gate 2 outputs L. By convention the set state corresponds to Q = H and the reset to Q = L. This flip-flop is called asynchronous because there is no master clocking signal governing the activity of the flip-flop, therefore it is sensitive to noise and glitches. For that reason it is recommended not to use RS flip-flops as a general design tool.

The excitation table for this block is shown below, and its gate implementation is shown in Fig. 3.13a. The excitation table for sequential circuits, is equivalent to the truth table for combinational circuits.

S	R	Q(t)	X(t)	Q(t+δ)	X(t+δ)	Function
L	L	q	x	q	x	HOLD
L	H	q	x	L	H	RESET
H	L	q	x	H	L	SET
H	H	q	x			DISALLOWED

Clocked RS flip-flop

The same RS flip-flop but with a clock signal that enables the inputs R and S is shown in Fig 3.13b. The flip-flop may change its outputs at the time the clock is true (level-driven) or during the transition of the clock signal (edge-driven) from L to H (positive edge) or from H to L (negative edge). To avoid hazards it is only recommended to use edge-driven flip-flops.

JK flip-flop

A typical excitation table for a JK flip-flop is the following:

Clock	J	K	Q(n)	Q(n+1)	
F	X	X	q	q	
T	X	X	q	q	
↑	F	F	q	q	HOLD
↑	F	T	q	F	RESET
↑	T	F	q	T	SET
↑	T	T	q	q	TOGGLE (Complement)

The JK flip-flop can be used:

1) when we must set, clear or toggle a signal to form a specific value for later use;
2) to transfer the data stored in Q(n) to Q(n+1) ;
3) for entering data, for example entering D into the flip- flop on a clock edge by having J = D , independent of the value of K.

D Flip-flop

The excitation table of the D- flip-flop is:

D	Q(n+1)
0	0
1	1

a) The active clock edge can be positive (L → H) or negative(H → L).
b) It may include asynchronous set and clear inputs, usually active low .

The D-flip-flop can be used:

1) To delay the value of the signal at its input by one clock time;
2) As a synchronizer of an input signal;
3) For data storage, where the data is loaded every cycle.

When you want to change the stored information only at a given time, you should use the enabled D flip-flop of Fig. 3.13b. You should never gate with a clock ANDed to a control signal. This is not a good solution.

Registers:

A register is an ordered set of flip-flops. It is normally a temporary storage. They are usually prepared using D flip-flops with enable.

Counters:

A typical circuit for a 4-bit(modulo 16) synchronous counter is shown in Fig. 3.14. The same for a ripple or asynchronous counter is shown is Fig. 3.15.

The synchronous counters have all their outputs changing at the same time, t_p after the clock edge. The asynchronous counters on the contrary change their outputs in a staggered fashion. The change in an output must ripple through all the lower-bits before it can serve as a clock for a high-order bit. Its configuration is simpler, but they are recommended only if you do not require any temporal relation of Q(n) to any lower bits. They are usually used as frequency dividers.

The counters have also a clear, that can be asynchronous or synchronous, and some other control inputs (enable, set, count up or down, etc.). It is recommended not to use the asynchronous clear to implement any logic except clear of the counter during power-up or general reset.

Shift-register

They perform an orderly lateral movement of data from one position to an adjacent one, every time a clock arises. They can have the following configurations:

- parallel-in, parallel-out;
- serial-in, serial-out;
- serial-in parallel-out;
- parallel-in, serial-out;

They include the following functions:

a) data loading;
b) shifting of bits one position right or left, while accepting one bit more at the input and discarding one at the output;
c) storing the data, while shifting is not done;

d) possibility of examining the output without changing their content.

A typical excitation table is shown below:

Clock	S1	S2	Result	Selected mux position	Required mux input
↑	0	0	Hold present data	0	Q(i)
↑	0	1	Shift right	1	Q(i+1)
↑	1	0	Shift left	2	Q(i-1)
↑	1	1	Load new data	3	DATA

Other blocks present in ASICs libraries

- **Processors**: The architecture of a typical processor can be represented as shown in Fig. 3.16. each of the elements in it has already been mentioned. This general blocks for processing units are available in different vendor's library, to be incorporated into ASIC designs. These circuits require a previous study of its characteristics, (architecture, set of instructions, and timing) in order to make a correct use of them in your design.

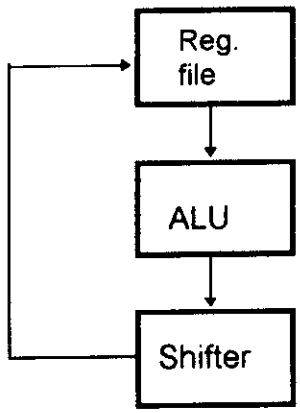


Fig. 3.16 Architecture of a processor

- RAMs

A memory that requires the same time to access each data bit is called a Random Access Memory (RAM). They can be static, when the storage cell closely resembles a D flip-flop, and the data is stored without losses; or dynamic, when the data is stored in form of

a charged ("1") or uncharged ("0") microscopic capacitor, that as all capacitors, has a discharging time.

Therefore, to maintain stable the stored information, a refreshing periodical pulse is necessary to maintain the capacitor in its programmed state (charged or uncharged). This RAMs contain also internal decoders for the selection of the row and column of the cell to be written or read, and other internal logic for selecting the mode of operation (Read, write, chip select) and for input and output of the data. Timing requirements vary from one memory to another, and differs also whether you are working with a static or a dynamic RAM. In any case you must carefully study the time requirements and adjust your design to fulfill it.

- **Devices with three state outputs**:

  An identity block or a logic NOT is said to have a tree state output if it provides in addition to the usual H and L levels, a high impedance mode called Z, in which the output appears as if it were disconnected from its destinations. They require an enabling tree-state control input. This structures are specially used in the Output Pads that connect the ASIC with the external world.

Metastability

Digital devices are in practice, analog devices that behave digitally only when stringent rules of operation are obeyed. In addition to establishing proper voltage levels at the inputs, to assure proper operation of a sequential device, you must adhere to the set up times, hold times, and other timing specifications. When this requirements are met, the devices will function proper, and the changes at the inputs and outputs will occur cleanly, showing the proper voltage levels. In this sequential circuits, except during the period of transition, the circuit will remain in one of its stable states. As long as no more than one input is changing at a time, the sequential circuit will perform well. If the voltage level of more than one input is allowed to change at nearly the same time there is a timing requirement that must be fulfill. If it is violated, the circuit may fall into a metastable state during which the outputs may hold improper or nondigital values for an unspecified duration, which are indicated during simulation as errors, indefinite states or violations. Metastability can be disastrous. In synchronous design, we try to avoid metastability by never changing the inputs in the vicinity of the clock.

4.- DESIGN METHODS

We will describe them with an example that consists of a 1K-by-eight-bit memory board connected to a hypothetical bus. The interface block provides

timing and control for the memory chips and also implements the bus handshaking logic. There are six signals that interconnects to the external world.

4.1.- Main steps:

1. - Separation of the control algorithm from the architecture to be controlled by this algorithm. The design will be partitioned into a control algorithm and the architecture controlled by this algorithm. Fig. 4.1.

2. - Details of the control algorithm at an abstract level, independent of the HW. A complete flowchart can be done without becoming engaged to an specific HW. This process may go through several iteration. Afterwards the control algorithm will suggest the HW for the architecture, so the algorithm should guide you to the HW solution. The description of Fig. 4.1, textually, will sound something like this:

A problem requires that a word be written into a memory.

The memory will require the following inputs:

- a memory address MA on n lines to tell where to write the data, a word of DATA of m;
- lines for input or output, a line R/W to tell whether to read or write;
- a GO signal to start the read or write operation.

The only status returned by the memory will be memory cycle complete CC.

The numbers n and m depend on the characteristics of the memory selected, for instance a 1K-by 8 bits memory would have n = 10 ($1024 = 2^{10}$) and m = 8 (length of the word).

In a similar form we can describe the algorithm that initiates a memory write operation, without knowing exactly how we will translate that algorithm into hardware. The algorithm will look something like Fig. 4.2. The purpose of the first step STW is to issue a GO signal to the memory, along with the necessary data and commands to initiate a writing operation. The next step WAITS until the memory has finished the writing.

A very good recommendation is to realize as much as possible of the design, under these general consideration. The decisions at this level are more easily to alter than once you have entered the HW frontier.

As we go down in the design, we must precise the knowledge of the architecture of the system, seeing it as a set of high-level building blocks. The following rules should be carefully observed:

1. It is very important that the selection of the elements should be made on the basis of what specific building blocks the developing control

algorithm requires, and not by looking what we have in the libraries or in the data book.

2. A good architecture must be as simpler, clear, and easy to control as possible.
3. The precision obtained with the above recommendations should lead to a better conception of the algorithm and its relation with the architecture. After we complement and optimize the algorithm, we can reconsider the architecture and our choice of building blocks. Simplifications or speed up by using different architectural components can be achieved. When this iterative process ends we should have:

   a) The architecture: as a detailed set of blocks and data paths, the specifications for these blocks, a statement of the command signals these components require, and the status signals they produce. The architecture does not include any logic to generate these commands, since the generator of commands is assigned to the control algorithm.

   b) The algorithm: as a set of command signals to make the architecture perform the original problem. At this level no hardware is yet define to implement the algorithm.

4. No one designs a system strictly from the top-down. A knowledge of low-level components and techniques always influences the design, even at the highest levels. The best op-down hardware designers have an intimate knowledge of hardware, and this knowledge tempers and guides the high-level design decision. Good designers use their knowledge of low-level technology to avoid unproductive approaches. One can dip for a while into lower levels, but invariably returns to the top. The high-level design methodology provides the discipline that keeps the designer thinking for the most possible time at the productive level.

4.2.- Notations for expressing an abstract algorithm

For synchronous circuits, among the pictorial descriptions, we have the Algorithmic State Machine (ASM), having much similarity to a typical flow chart in computer programming. Synchronous circuits have state times determined by only one master clock, usually a periodic square-wave voltage. The transition from one stage to another and other actions of the circuit are triggered by either the positive (transition from L to H)or the negative (transition from H to L) edge of the clock pulse. The time in H is equal to the time in L. Each active transition of the clock causes a change of state from the present state to the next. A state is represented usually as a rectangle with its symbolic name enclosed in a small circle at the upper left hand corner. In Fig. 4.2 we represented the ASM for Fig. 4.1. It has one unconditional output (OUT1), two states (STW and WAIT); one condition for transition given by the variable CC. The states can be

represented by one variable, for example A. In Fig. 4.3 is represented another ASM with four outputs (OUT1 to OUT4), the second and fourth being a conditional outputs. There are 4 states, P, R, Q, S ; there are four conditions for transition given by the variables X,Y,Z,W. The 4 states can be represented by two variables, in this case A and B.

4.3.- Synthesis from an ASM chart

4.3.1. Traditional method

The traditional technique for state generation is to produce an encoded representation of the present state and compute the code for the next state. If the code has n bits, it can describe up to 2^n states. On the ASM chart, the binary representation of each state is written on the right-hand side of the state rectangle. The test diamonds or the conditional output do not require a label, since they are part of a state. The values of the state variables are the address that points to the present state at each moment. The next address is computed by means of a combinational circuit and stored in flip-flops. Jk flip-flops result in less combinational logic than D flip-flops, but require more input lines. In the example we will use D flip-flops which provide more clarity The combinational logic must compute the value of the next address. In the example of Fig. 4.2, the logic must implement the following state transition table:

Present state		Next state
A	CC	A(D)
0	x	1
1	0	1
1	1	0

It is also necessary to take into account that there might appear possible patterns of the flip-flop's outputs that are not used by the algorithm. Nevertheless it is completely necessary to prevent these situations and force the algorithm to return to the main loop (go back to state 00), if by an unexpected situation they are reached. This rule must be strictly complimented in any state generator.

From de logic table above, the equation for the state flip-flop's input A(D) is:

= A + A*CC

The equations for the output is:

OUT1 = WAIT.CC

If the equations are complex, K-maps can be used to simplify them.

The HW for the ASM is shown in Fig. 4.4. Unfortunately the traditional method results in no obvious correspondence between HW and the algorithm, so our goal of clarity in design is not fulfilled.

4.3.2.- Multiplexer controller method.

The main advantage of this method is that it produces a design that has a direct correspondence with the algorithm that generated it.

The main difference with respect to the traditional method is that instead of using gates to compute the next state code, it uses a table look up, implemented with multiplexers each of which provides the input to its respective state flip-flop. At the same time the assembly of multiplexers yields the code for the next state. The multiplexers must have at least the same number of inputs that states the ASM. The present-state address code is the ordered output of the state flip-flops which is fed into the select inputs of each multiplexer to select the appropriate input for the present state of the system. The design must provide that for each present state, the mux inputs provide the 1 or 0 necessary to produce the next-state code. The following table shows how to produce the next state standing in each state.

State transition data for the ASM in Fig. 4.2

Present State		Next State		Condition for Transition
Number	Name	Name	A	
0	STW	WAIT	1	T
1	WAIT	WAIT	1	$\overline{CC}$
1	WAIT	STW	0	CC

The resulting state generator is shown in Fig 4.5.

The synthesis of the ASM shown in Fig. 4.3 will look as in Fig. 4.7.

Resuming the rules for synthezising by means of the multiplexer method we have:

1. - Create a state transition data;
2. - Use an encoded representation of the present state and compute the code for the next state:

   n bits describe up to 2^n states;

3. - Write the equations for the flip-flop 's inputs;

4. - Write the equations for the outputs;

5. - Use a D- flip-flop each next state;
6. - Use a multiplexer at the input of each state flip-flop, to produce the new input to the flip-flop. The multiplexer must have n selection lines and 2^n inputs;
7. - Check that all unused states return the machine to the start position.

With the increase of the number of states above 16, the number of inputs to the multiplexers can become too large. An alternative solution in this cases is the One-hot method.

4.3.3.- One-hot method.

In this method one D flip-flop is used to generate each state. There is no encoding of the states. Only one of the state flip-flops can be true during each state time, so combinational logic at the flip-flops' inputs must provide the one true input at each state. This property of only one true flip-flop at a time is called the one-hot, and has the advantage of being easy to implement in design synthesizers.

This method requires a specific initial condition, (initialization), that provides that only one flip-flop, the one representing the starting state is true and the rest are false.

In Fig. 4.7. is shown the synthesis from the ASM chart of Fig. 4.2, implemented by the one-hot method, and in Fig. 4.8a the one corresponding to the ASM of Fig. 4.3. In Fig. 4.8.b the synthesis has being made in order to use only physical NANDs for its implementation.

Resuming the rules for synthesizing by means of the **one-hot method** we have:

1. - Create a state transition table;

2. - Write the equations for the flip-flop 's inputs;
3. - Write the equations for the outputs;

4. - Use a D- flip-flop each state;
5. - Arrange so that only one flip-flop is true at each state time.

4.3.4.-The ROM-base method

In this method, the look-up table is implemented by means of a ROM, PROM or EPROM. It has the advantage of being very regularly, but as the ASM becomes more complex, the size of the ROM may become too large. The looks-up table in this case is shown in the following table, and the HW implementation in Fig. 4.9.

State generator and outputs for the ASM of Fig. 4.3, using a ROM

Address						Ouput					
B	A	X	Y	W	Z	B(D)	A(D)	OUT1	OUT2	OUT3	OUT4
0	0	0	0	X	X	1	0	1	0	0	0
0	0	0	1	X	X	0	1	1	0	0	0
0	0	1	X	X	X	0	0	1	0	0	0
0	1	X	X	0	X	0	0	0	1	1	0
0	1	X	X	1	X	1	0	0	1	0	0
1	0	X	X	X	X	0	0	0	0	0	0
1	1	X	X	X	0	1	1	0	0	0	0
1	1	0	X	X	1	0	0	0	0	0	1
1	1	1	X	X	1	1	0	0	0	0	0

4.4.- Other general aspects to take into consideration; clock skew

The combination logic required to generate signals B(D) and A(D) in Fig. 4.8 will generate the correct new values only some time after the new values of signals A and B (transmitted with CLKA and CLKB) enter the combinational logic block. If both CLKA and CLKB arrive at the same time, no problem occurs, provided we wait for this settling time. Now suppose that CLKA arrives, and CLKB is delayed in such a way that it arrives before the settling time has elapsed. Since B(D) may have momentary wrong values CLKB edge can record a wrong value. If on the contrary the settling time has elapsed, the new value of B(D) based on the correct new value of A and the unchanged value of B will yield a new pair of values A(D) and B(D), which are incorrect. When CLKB arrives, it will store anyway an incorrect B. The clock skew arises from gates connected in the clock path or from different wire length between the clock source and the clock inputs. To avoid this problem:

1- Try not to insert gates in the clock lines. For example do not use mixed flip-flops with positive and negative active edges, so you do not have to introduce inverters.

2- Distribute the clock lines radially from the clock source, rather than connecting them along one chain. Try to use lines with more or less the same length.

3- If buffers are required to manage power , try to use the same kind of buffers for all the lines.

Manual designer usually had to solve these problems by themselves. The automated design tools, take them into account and during synthesis at the different levels, they provide programmed solutions for the different cases. Of course they can not take into account all possible cases, so again we can remark that an **experienced designer with a good idea of the problems arising during the physical implementation, will always produce better designs.**

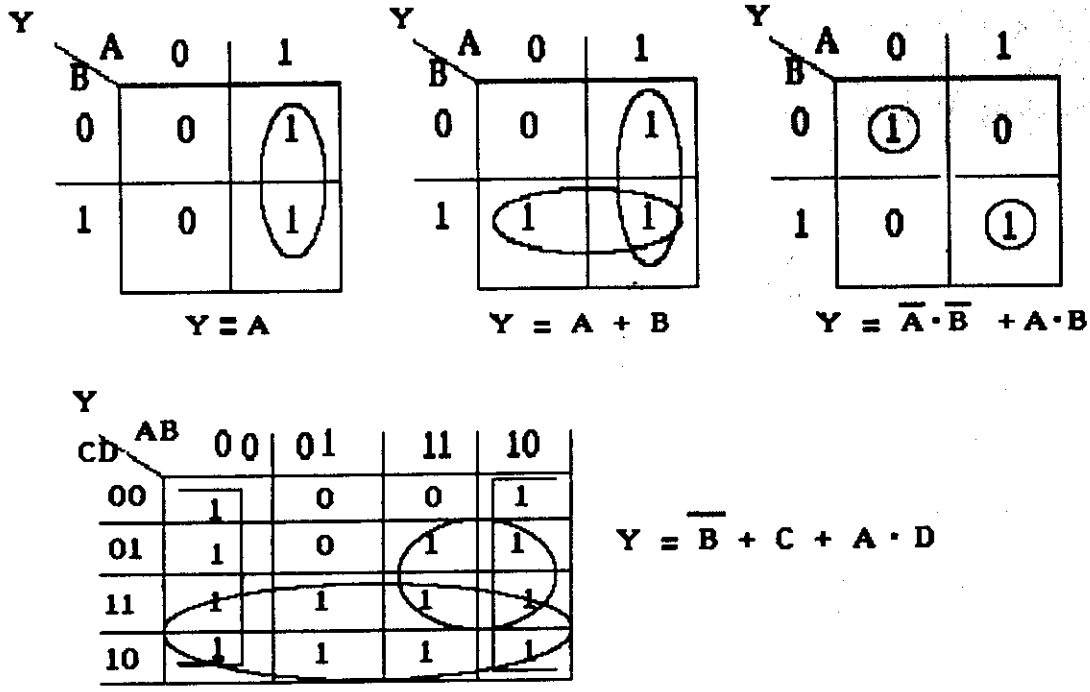


Fig. 2.1 Simplifing with K-maps

TT for logical identity operation

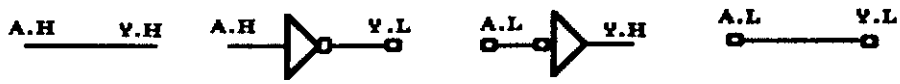
A	Y
F	F
T	T

A.H	Y.H
L	L
H	H

A.H	Y.L
L	H
H	L

A.L	Y.H
H	L
L	H

A.L	Y.L
H	H
L	L



Piece of wire | Voltage inverter | Voltage Inverter | Piece of wire

Fig. 3.3 Realization of the logical identity for four choices of voltage.

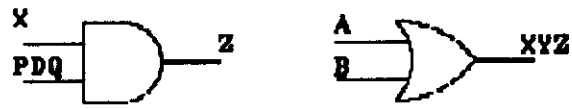


Fig. 3.2

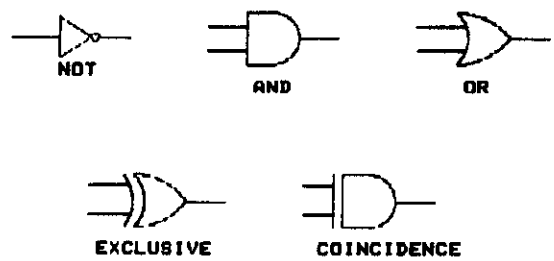


Fig. 3.1 Symbols for the elementary logical operations.

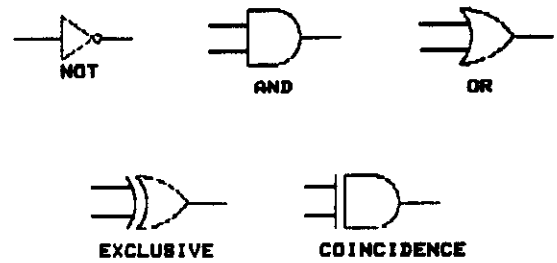
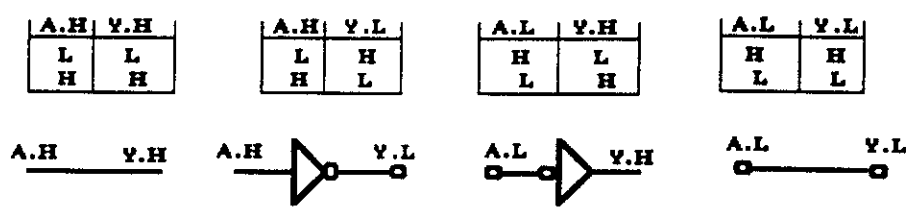


Fig. 3.1 Symbols for the elementary logical operations.

Fig. 3.2

TT for logical identity operation

A	Y
F	F
T	T



Piece of wire | Voltage inverter | Voltage Inverter | Piece of wire

Fig. 3.3 Realization of the logical identity for four choices of voltage.

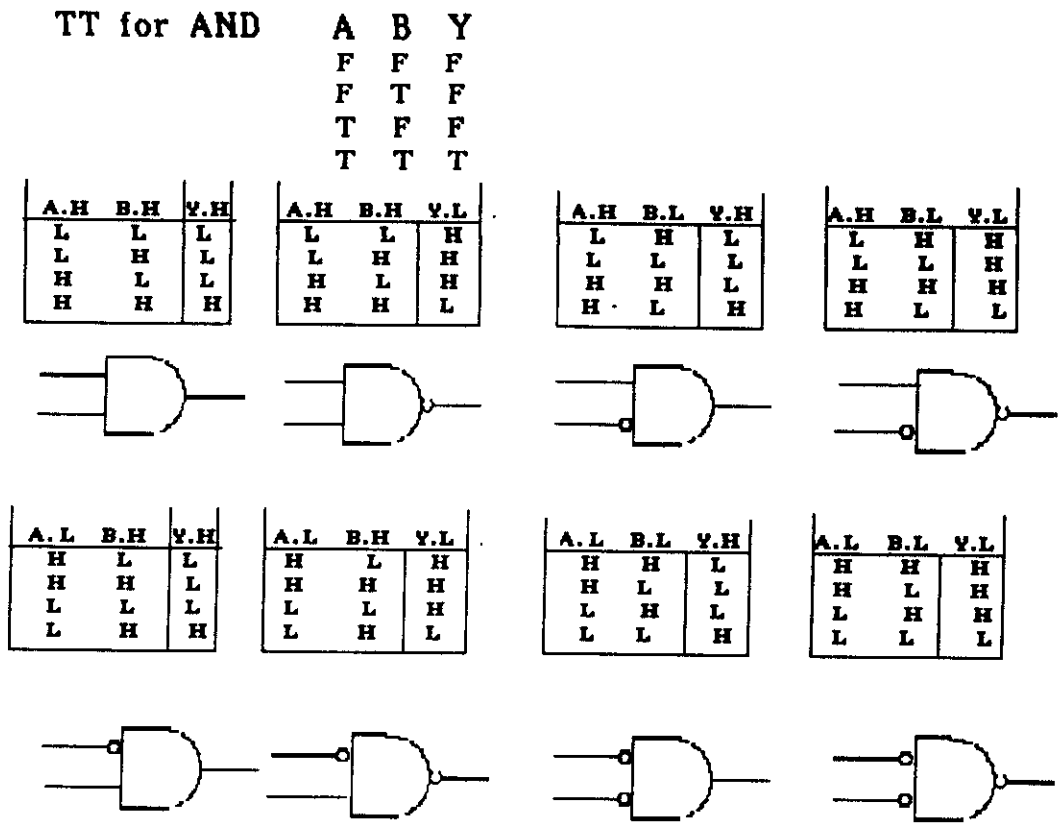


Fig 3.5. Realizations of logical AND for eight choices of voltage.

TT for logical NOT

A	Y
F	T
T	F

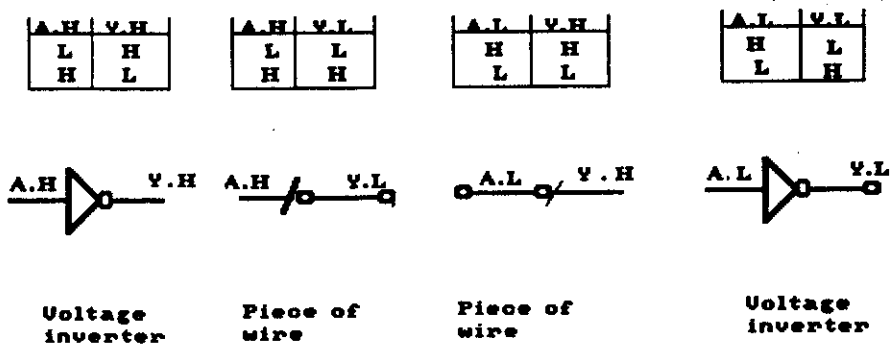


Fig. 3.4 Realization of the logical NOT for four choices of voltage.

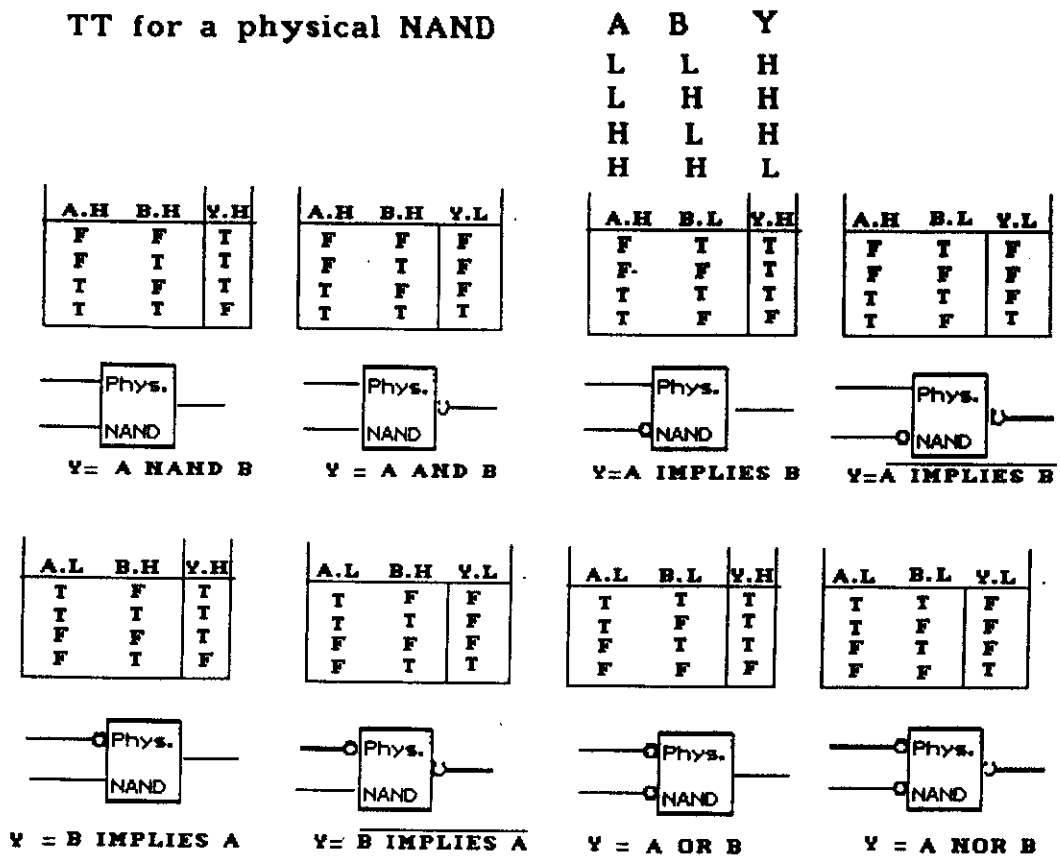


Fig. 3.7 LOGIC PERFORMED BY A PHYSICAL NAND

TT for logic OR

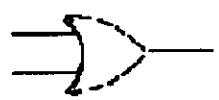
A	B	Y
F	F	F
F	T	T
T	F	T
T	T	T

A.H	B.H	Y.H
L	L	L
L	H	H
H	L	H
H	H	H

A.H	B.H	Y.L
L	L	H
L	H	L
H	L	L
H	H	L

A.H	B.L	Y.H
L	H	L
L	L	H
H	H	H
H	L	H

A.H	B.L	Y.L
L	H	H
L	L	L
H	H	L
H	L	L



A.L	B.H	Y.H
H	L	L
H	H	H
L	L	H
L	H	H

A.L	B.H	Y.L
H	L	H
H	H	L
L	L	L
L	H	L

A. L	B.L	Y.H
H	H	L
H	L	H
L	H	H
L	L	H

A. L	B.L	Y.L
H	H	H
H	L	L
L	H	L
L	L	L

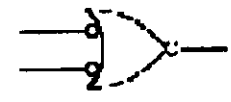


Fig. 3.6. Realizations of logical OR for eight choices of voltage.

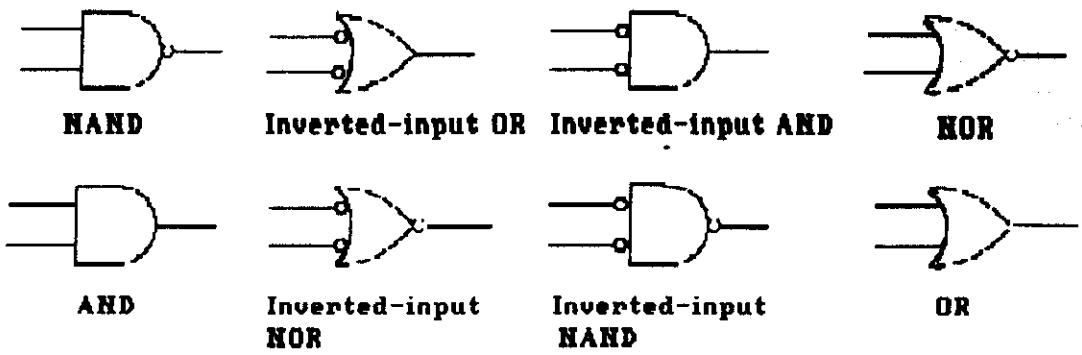
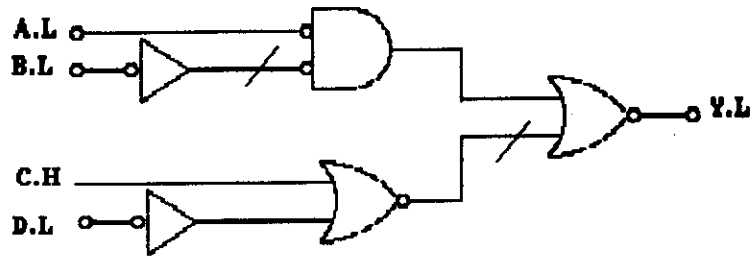


Fig. 3.9 Symbols for the positive-logic convention.



The logical equation corresponding to this circuit is

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

Fig 3.8 Analysis of a circuit.

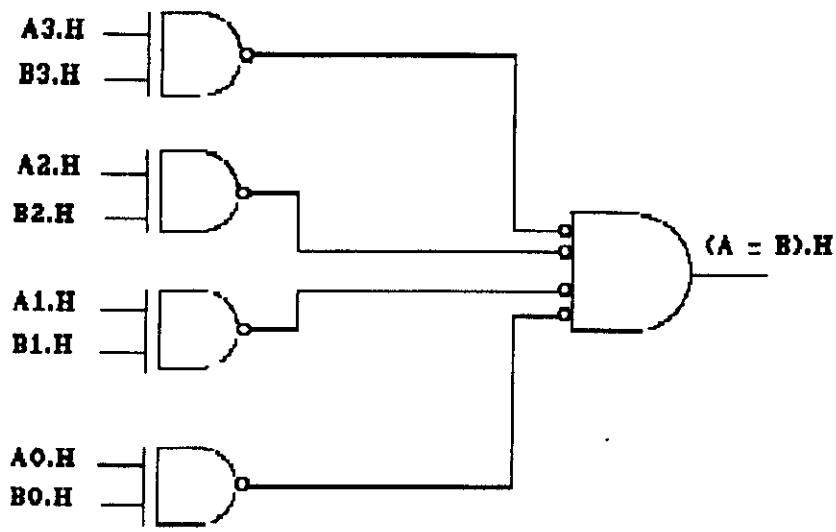


Fig.3.10 Circuit for comparing two 4-bit signals.

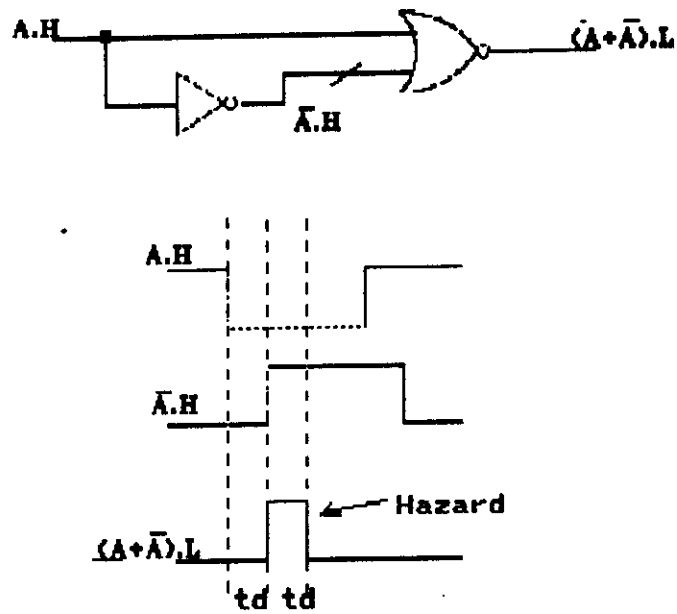


Fig. 3.11. Appearance of a hazard.

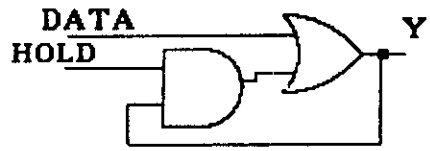


Fig 3.12 Schematic circuit for a latch.

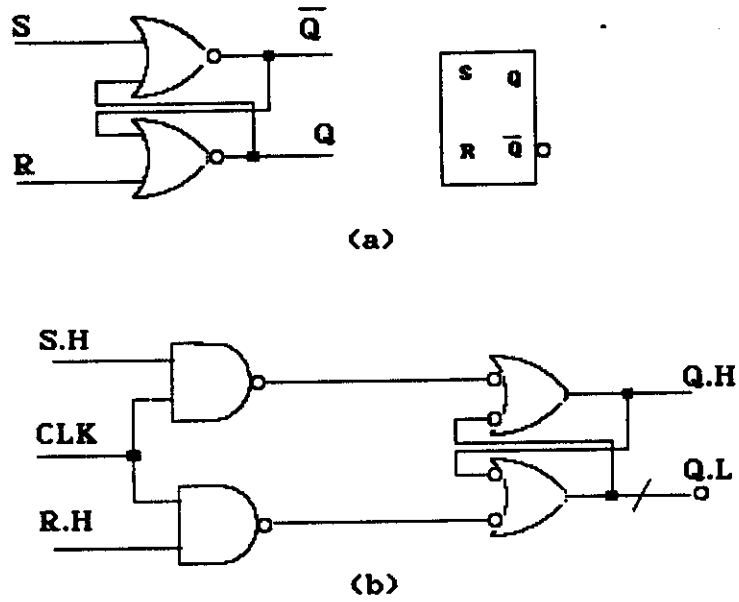


Fig. 3.13. a) RS flip-flop; b) Clocked RS flip-flop

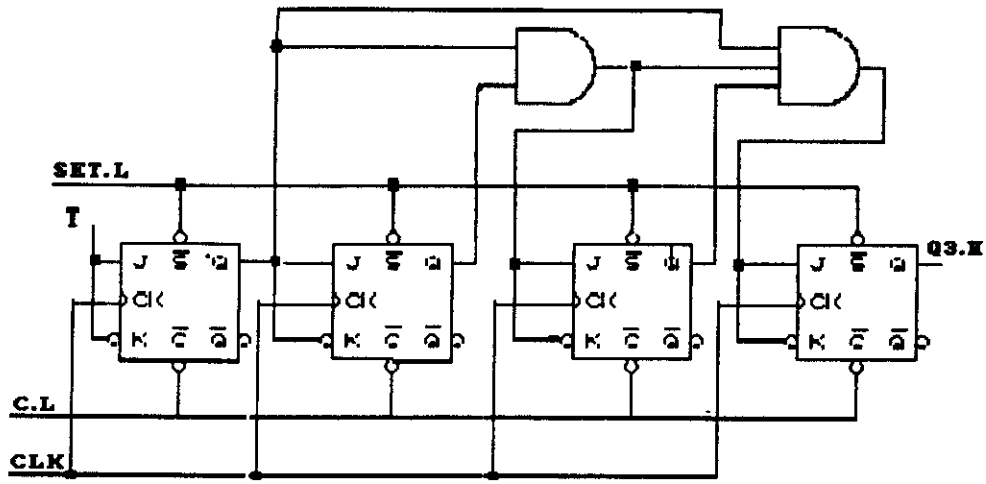


Fig. 3.1 4 Synchronous counter

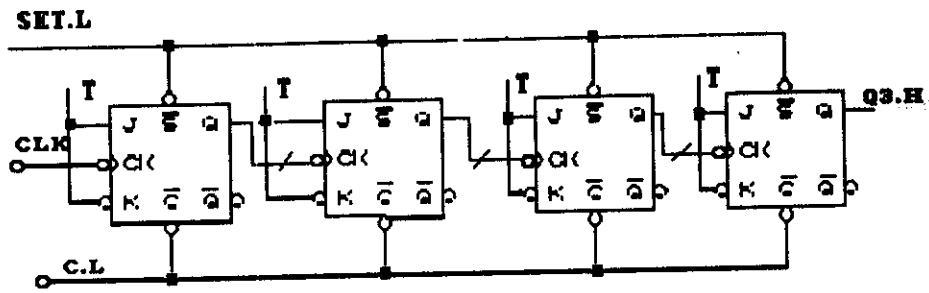


Fig. 3.15 Asynchronous counter

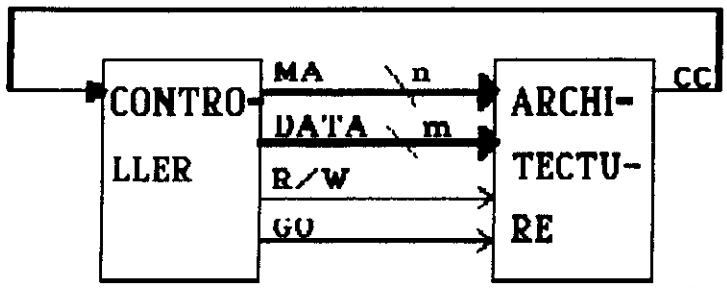


Fig. 4.1 Step one: separation of control algorithm from architecture to be controlled by it.

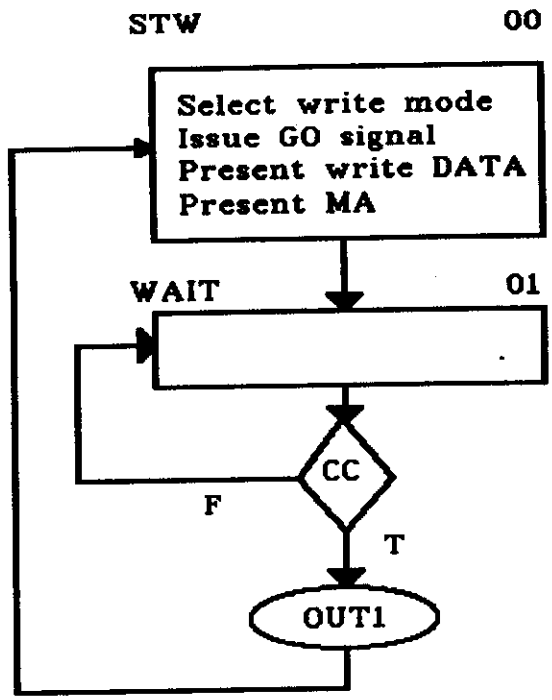


Fig. 4.2 Description of the control algorithm.

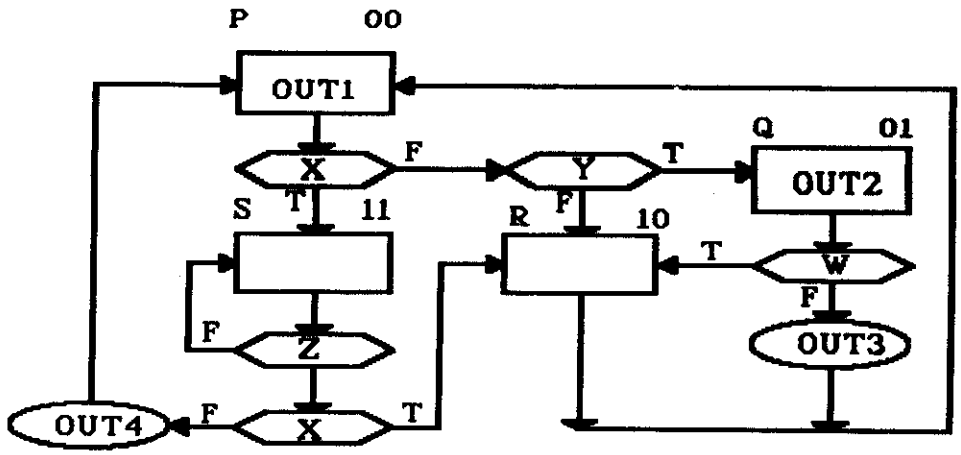
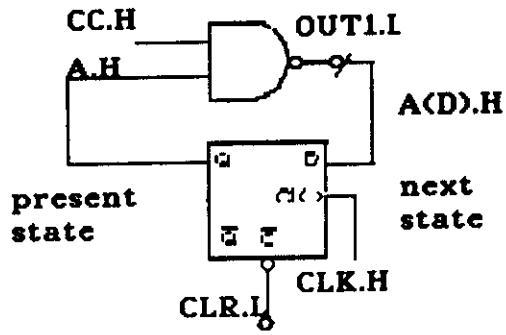


Fig. 4.3 A more complex ASM.



$A(D) = A \cdot CC$

Fig 4.4 HW for ASM in Fig. 4.2 obtained by the traditional method.

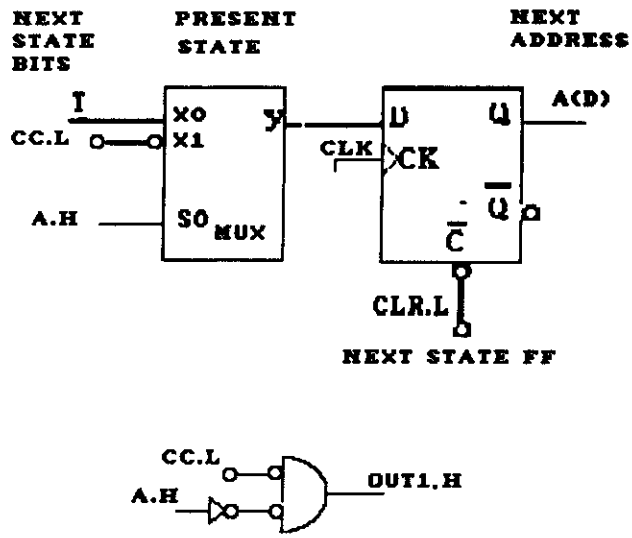


Fig. 4.5 Implementation of the ASM of Fig. 4.2, using the multiplexer-controlled method.

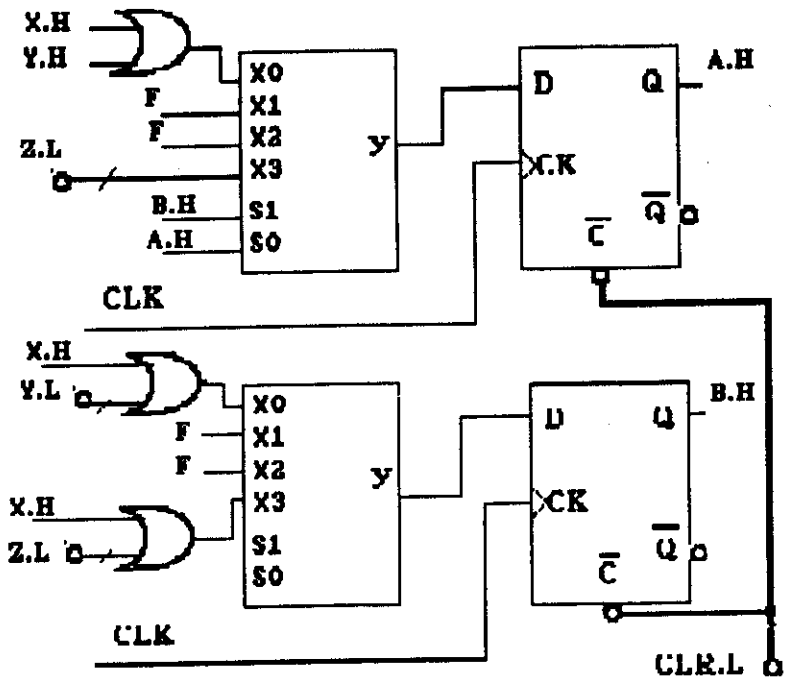


Fig. 4.6 Implementation of the ASM of Fig. 4.3 using the multiplexer method.

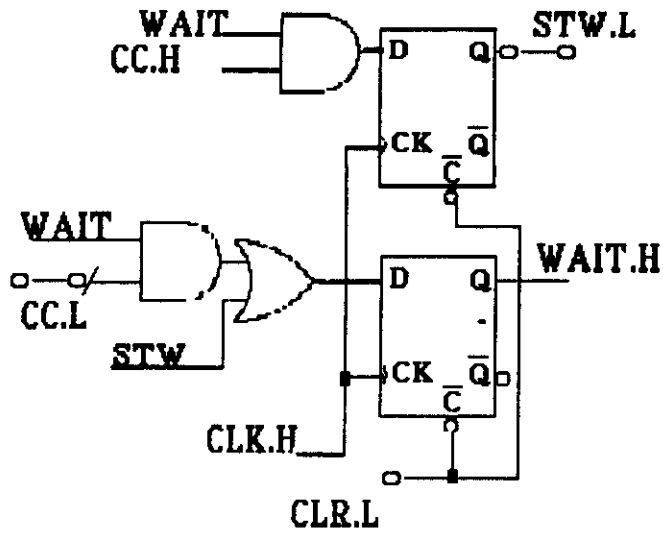


Fig. 4.7 Implementation of the ASM in Fig. 4.2 by the one-hot method.

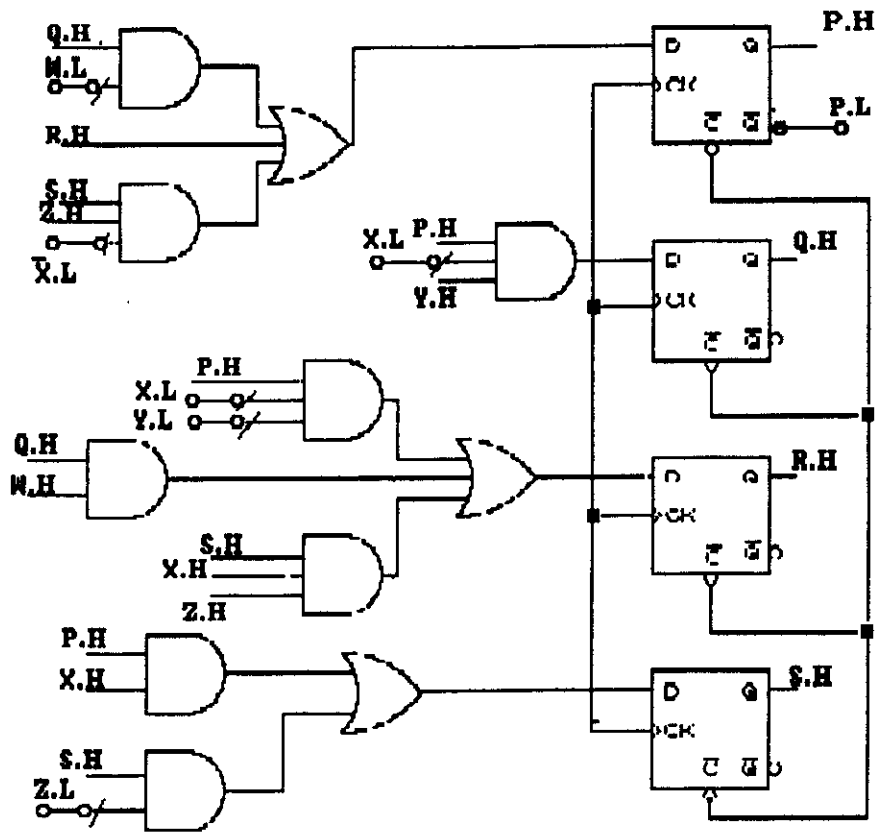


Fig. 4.8 a. A one-hot controller for the ASM of Fig. 4.3.

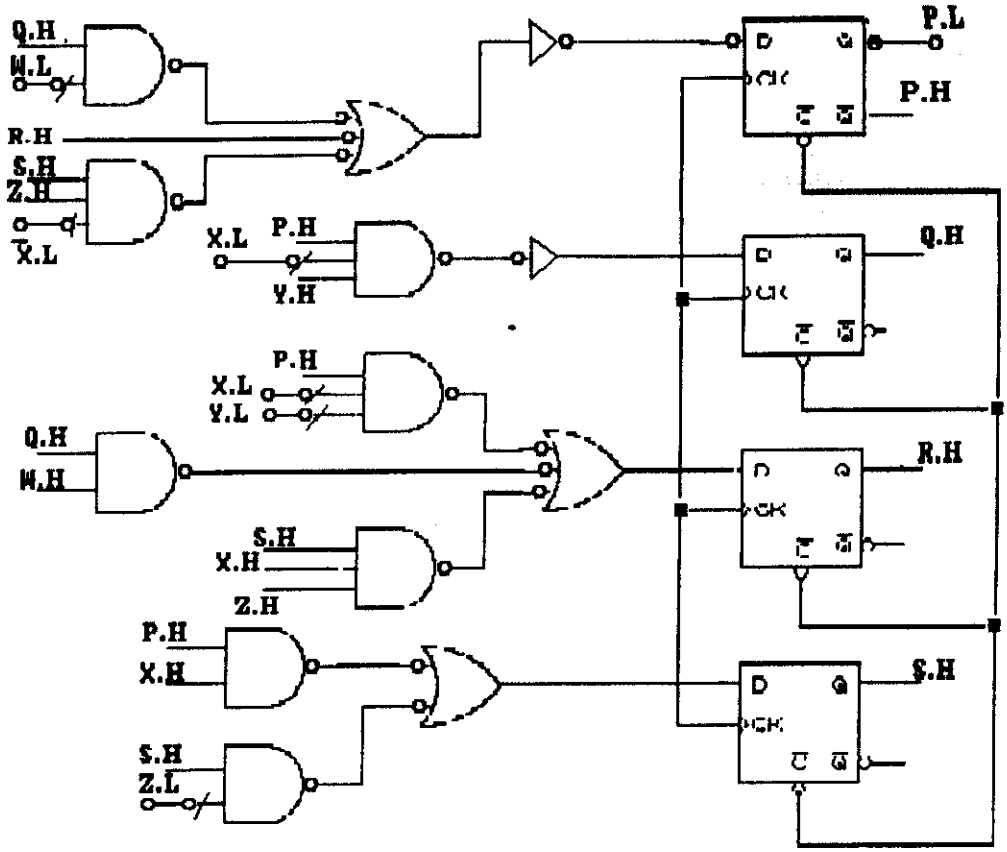


Fig. 4.8 b. Same as Fig. 4.8a, using NANDs.

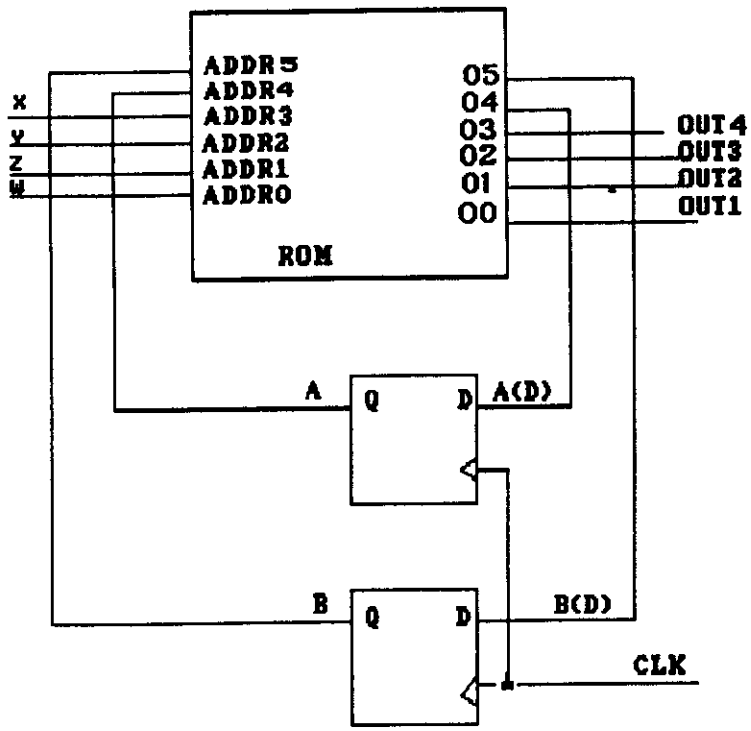


Fig. 4.9 Implementation of the ASM in Fig. 4.3 using a ROM.