

3. Design Representation

3.1 Levels of Abstraction

In this chapter we will examine design representations, also called *design models*. From the onset it is important to distinguish between design models and design specifications. The specifications describe the design in terms of its results, while the models describe the design's procedure. The model is used to simulate the design in a given domain and at a given level and to assert whether or not it conforms to specifications. It is also used in mapping the design to another level or domain. Use of the terms *domain* and *level* refer to the taxonomy illustrated with modifications in Fig. 3.1 [Gajski 1983, Walker 1985, Michel 1992].

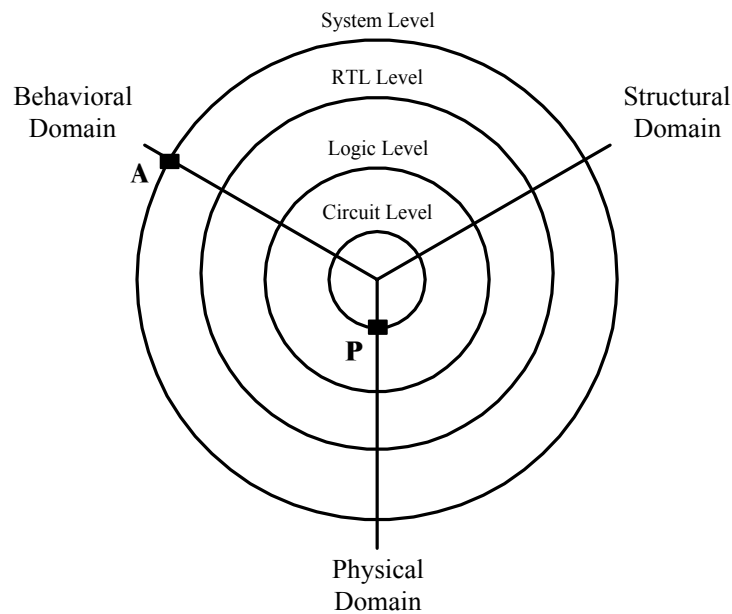


Figure 3.1 Levels of Abstraction

For each of the domains – *behavioral*, *structural*, and *physical* – we distinguish several levels: system, RTL, logic, and circuit. The behavioral domain gives a functional representation, while the structural domain describes the architectural blocks; the physical domain is the actual chip. For the logic level, the three domains are illustrated in Fig. 3.2, and Fig. 3.3 shows the levels in the structural domain. Table 3.1 shows the different levels of the design for each domain. The same level of a domain may be described in different formats: equations, tabular, programming language, or hardware description languages. In this chapter we examine these formats and illustrate

their use in the testing field. Familiarity with these different representations is important in order to understand their effect on the efficiency of CAD tools, which we discuss in Chapters 4 and 14.

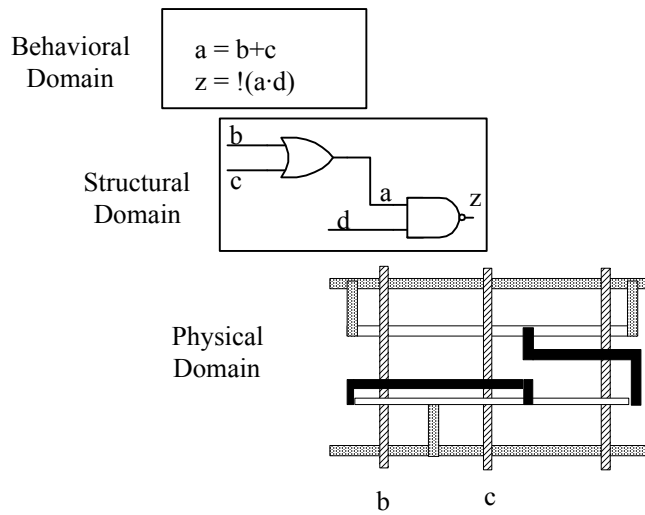


Figure 3.2 Domains of Circuit Representation on the Logic Level

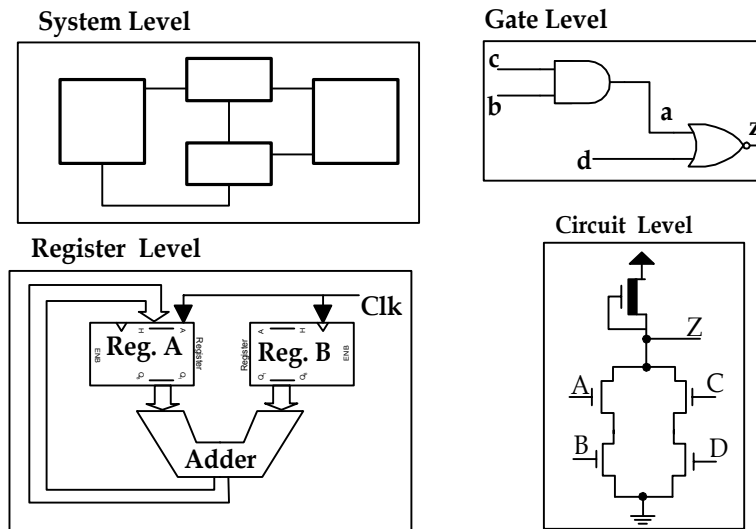


Figure 3.3 Levels of Abstractions in the Structural Domain

Table 3.1 Domains and Level of Design

Levels	D o m a i n		
	Behavioral	Structural	Physical
System	System specifications	Blocks	Chip
RTL	RTL specifications	Registers	Macro cells
Logic	Boolean functions	Logic gates	Standard cells
Circuit	Differential equations	Transistors	Masks

3.2 Mathematical Equations

Mathematical formulation is used to describe designs on the logic level as well as on the circuit level. Examples are combinational switching functions, finite state machines, and transistor-level circuits.

3.2.1 Switching Functions

The combinational functions that we customarily use in logic design are a special case of Boolean functions. They are described more accurately as switching functions. Any such function $f(x)$ has a range B and a domain B^n where $B = \{1,0\}$ and $f(x) : B^n \rightarrow B$.

Definitions:

- For any element $c \in B$, the constant function is $f(x_1, \dots, x_n) = c$, where $x_i \in B^n$.
- For any $x_i \in B^n$, the projection function is $f(x_1, \dots, x_n) = x_i$.
- The set of variables $\{x_1, x_2, \dots, x_n\}$ is called the support of the function.
- If g and h are n -variable functions, then the functions $g + h$, $g \cdot h$, and g' are defined by

$$(g + h)(x_1, \dots, x_n) = g(x_1, \dots, x_n) + h(x_1, \dots, x_n)$$

$$g \cdot h(x_1, \dots, x_n) = g(x_1, \dots, x_n) \cdot h(x_1, \dots, x_n)$$

$$\mathbf{g}'(x_1, \dots, x_n) = \mathbf{g}(x_1, \dots, x_n)'$$

There is only a finite set of distinct functions of the n -variable, $|B|^{2^n}$.

However, the same function may be expressed in a different form – for example, $f(x_1, x_2, x_3) = x_1x_2' + x_2 = x_1 + x_2$, by virtue of switching algebra postulates [Kohavi 1978, McCluskey 1986].

Definition - The Shannon expansion of any switching function is given by

$$f(x_1, \dots, x_n) = x_i' f(x_1, \dots, x_i = 0, \dots, x_n) + x_i f(x_1, \dots, x_i = 1, \dots, x_n) \quad (3.1)$$

or

$$f(x_1, \dots, x_n) = (x_i' + f(x_1, \dots, x_i = 0, \dots, x_n))(x_i + f(x_1, \dots, x_i = 1, \dots, x_n)) \quad (3.2)$$

The cofactors of x_i are the residues of the functions for $x_i = 0$ and $x_i = 1$. The residue of the function for $x_i = 1$ is the value of this function when x_i is substituted by 1. Successive applications of the Shannon expansion would eventually result in expressing the function in terms of products of x_i . These products are called the *minterms*. The function is thus expressed as a sum of minterms. For example, the function

$$f(x_1, x_2, x_3) = x_1x_2' + x_3 \quad (3.3)$$

may be expressed after three successive applications of Shannon expansion, in terms of x_3 , x_2 , and x_1 , into the sum of minterms

$$\begin{aligned} f(x_1, x_2, x_3) &= (x_1x_2')x_3 + (x_1x_2')x_3' + x_3 \\ f(x_1, x_2, x_3) &= x_1x_2'x_3 + x_1x_2'x_3' + x_2'(x_3) + x_2(x_3) \\ f(x_1, x_2, x_3) &= x_1x_2'x_3 + x_1x_2'x_3' + x_1'(x_2'x_3) + x_1(x_2'x_3) + x_1'(x_2x_3) + x_1(x_2x_3) \end{aligned}$$

and using $x + x = x$, we can eliminate the fourth term, which is identical to the first one, and obtain

$$f(x_1, x_2, x_3) = x_1x_2'x_3 + x_1x_2'x_3' + x_1x_2x_3 + x_1'x_2x_3 + x_1'x_2'x_3 \quad (3.4)$$

$$f(x_1, x_2, x_3) = \sum m(1,3,4,5,7) \quad (3.5)$$

3.2.2 Boolean Difference

Switching functions are used extensively in logic synthesis and, in particular, in minimization [Hachtel 1996]. They are used to develop test pattern generation as described below. To observe a stuck-at fault at any of the outputs of the circuit, we must have the response of the good circuit, R , different from the faulty circuit, R_f . In terms of Boolean algebra, this can be expressed as

$$R \oplus R_f = 1 \quad (3.6)$$

If the faulty node is a primary input, x_i , then for a function $f(x_1, \dots, x_n)$, R and R_f are the residues of the function with respect to x_i . Equation (3.6) can then be expressed as

$$f(x_1, \dots, x_i = 0, \dots, x_n) \oplus f(x_1, \dots, x_i = 1, \dots, x_n) = 1. \quad (3.7)$$

This last expression known as the *Boolean difference with respect to x_i* is denoted by $df(x)/dx_i$, where x is the support of the function. For simplicity, we express it as $df(x)/dx_i = f_i(0) \oplus f_i(1)$, where f_i is the residue of f with respect to x_i . The Boolean difference is then equated to 1 to determine the values of the variables that will allow the observability of the fault on node x_i at the primary output. In addition, for a SA1 or SA0 fault, we need $x_i = 0$ or $x_i = 1$. The test patterns to detect SA0 and SA1 on x_i are, respectively, given by

$$x_i df(x)/dx_i = 1 \text{ and } x_i' df(x)/dx_i = 1 \quad (3.8)$$

As an example, let us consider the function

$$f(x) = g(x) + x_3, \text{ where } g(x) = x_1 x_2 \quad (3.9)$$

which is illustrated by the gate-level description in Fig. 3.4. To determine the test patterns for the stuck-at faults on the primary input, x_2 , we first form the Boolean difference and equate it to 1. Thus $df(x)/dx_2 = x_3 \oplus (x_1 + x_3) = x_3' x_1 = 1$. This implies that $x_1 = 1$ and $x_3 = 0$. For the SA1 and SA0 faults on x_2 , the patterns are then $x_1 x_2 x_3 = (100)$ and (110) , respectively. We repeat this calculation for stuck-at faults on x_3 . First, we calculate the Boolean difference: $df(x)/dx_3 = g(x) \oplus 1 = x_1 x_2 \oplus 1 = (x_1 x_2)'$. Then we equate its products with x_3 and x_3' to 1. The patterns to detect the faults $x_3/0$ and $x_3/1$ are then $x_3 (x_1 x_2)' = 1$ and $x_3' (x_1 x_2)' = 1$. For the first fault, we must then have $x_3 x_1' + x_3 x_2' = 1$. This results in three patterns: $x_1 x_2 x_3 = (001, 011, \text{ or } 101)$, and for the other fault, we have $x_1 x_2 x_3 = (000, 010, \text{ or } 100)$.

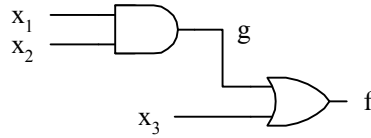


Figure 3.4 Example for Test Pattern Generation using Boolean Algebra

The fault may also be on an internal node of the circuit. In this case we include this node among the variables in expressing the function. For instance, for the internal node $g(x)$ of our example, we can express the function as $f(x_3, g)$ and the test pattern that detects $g/0$ by

$$g(x) \frac{df(x, g(x))}{dg(x)} = g(x) [f(x, g=0) \oplus f(x, g=1)] = 1 \quad (3.10)$$

From Eq. (3.9) we have $f(x, g=0) = x_3$ and $f(x, g=1) = 1$. Thus the expression in Eq. (3.10) becomes $g(x) \{x_3 \oplus 1\} = g(x) x_3 = 1$. Substituting for $g(x)$, we get $x_1 x_2 x_3 = 110$, which is the same pattern that detects $x_2/0$. This was expected from our knowledge of fault equivalence that was explained in Section 2.6.

The Boolean difference has several properties that can be used in dealing with more complex functions [Fujiwara 1986, Miczo 1986]. However, this approach is not computationally efficient for test pattern generation.

3.2.3 Finite State Machines

A finite state machine (FSM) is formally expressed as a six-tuplet $(I, S, \delta, S_0, O, \lambda)$, where

- I is the input alphabet, that is, a finite nonempty set of inputs.
- S is the finite and nonempty set of states.
- $\delta: S \times I \rightarrow S$ is the next state function.
- $S_0 \subseteq S$ is the set of initial states.
- O is the output alphabet.
- $\lambda: S \times I \rightarrow O$ is the output function for a Mealy machine [Mealy 1955].
- $\lambda: S \rightarrow O$ is the output function for a Moore machine [Moore 1956].

An example of an FSM, M , is given by the sets

$$I = \{0,1\}$$

$$S = \{A, B, C, D\}, O = \{0, 1\}$$

and the next-state function

$$\begin{aligned} \delta(A, 0) = C, \delta(A, 1) = B, \delta(B, 0) = C, \delta(B, 1) = B \\ \delta(C, 0) = D, \delta(C, 1) = C, \delta(D, 0) = A, \delta(D, 1) = C \end{aligned} \quad (3.11)$$

$$\begin{aligned} \lambda(A, 0) = 1, \lambda(B, 0) = 0, \lambda(C, 0) = 1, \lambda(D, 0) = 1 \\ \lambda(A, 1) = 0, \lambda(B, 1) = 1, \lambda(C, 1) = 1, \lambda(D, 1) = 0 \end{aligned} \quad (3.12)$$

When applied to the machine that is initially in state A , an input string (10110), will yield an output string (00111) and leaves the machine in the final state D .

Time t	0	1	2	3	4	5
Input I	1	0	0	1	0	
State $\delta(S, I)$	A	B	C	C	C	D
Output $\lambda(S, I)$	-	0	0	1	1	1

In this example, all states and outputs are fully specified. M is said to be a completely specified machine. Often, not all next states or outputs are known for each input combination. In such cases, the FSM is *incompletely specified*.

The mathematical representation of an FSM is cumbersome and other means are used more typically. A graph representation, illustrated in Section 3.5, is another useful way of expressing an FSM. This diagrammatic representation enhances the visualization of state transitions. However, when the FSM is large, that is, when the cardinality of the set S is too large, the mathematical equations and the graph representation become cumbersome. It is more practical then to represent the FSM in a tabular format, as explained in Section 3.3.

3.2.4 Transistor-Level Representation

On the circuit level, however, equations are still the most useful way for simulation and timing analysis. Before the early 1970s, circuits were analyzed almost exclusively by hand [Hodges 1983]. The first important automation software was SPICE (Simulation program, IC emphasis), developed at the University of California, Berkeley [Nagel 1975, Quarles 1994]. The simulation makes use of mathematical models of the transistor. Usually, the model is given as a set of equations that relates the voltages and currents through the network. The model equations are

complex since the transistor exhibits nonlinear resistive and capacitive characteristics. For a bipolar transistor, the Eber – Moll and the charge-based Gummel – Poon models have been merged to represent the currently used model, which is shown in Fig. 3.5a [Getreu 1976]. For a MOS transistor, the models used are also those developed for a SPICE simulator. There are four different models for the various levels of this simulator. They represent the effects of charge storage associated with thin oxide. Level 1 uses the Shichman – Hodges model shown in Fig.3.5b which relies on the current square-law expression in terms of the gate voltage. The other levels utilize more complex models to reflect the effect of submicron and *deep* submicron technologies. Submicron refers to feature size, $2\lambda < 1 \mu\text{m}$, and deep submicron is usually for $2\lambda < 0.5 \mu\text{m}$. For example, level 3 takes into account second-order effects such as short channel threshold voltage, subthreshold conduction, scattering-limits velocity saturation, and charge-controlled capacitances. There are several references that detail these models, among them [Antognetti 1988]. Recently developed LEVEL 4, the Berkeley Short-channel IGFET model (BSIM) model, is based on different parameters that are extracted from experimental data. Currently, it is the most used in industry to model the behavior of submicron MOS transistors [Jeng 1983, Sheu 1988, Cheng 1996].

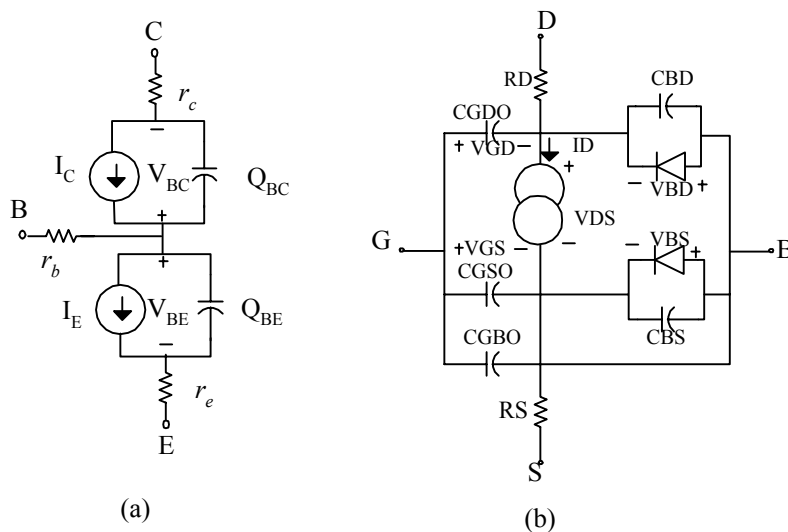


Figure 3.5 Spice Model for MOSFETs

The circuit description for SPICE simulation, usually called the *deck*, consists of the topology of the circuit and specifications of the transistor model, the parameters, and the technology. An example for a typical CMOS inverter is shown below.

- * CMOS inverter
- * The circuit

```

Mn 3 2 0 0 CMOSN W=1.8U, L=0.5U
Mp 3 2 1 1 CMOSP W=5.4U, L=0.5U
CL 3 0 2pF
*Voltages used
VDD 1 0 DC 5V
Vin 2 0 0 3.3 0.2
Vin 2 0 PULSE (0V 3.3V 0n 0n 0n 50n 100n)
* Transistor Models used
.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=9.6000E-09 J=0.200000U
+ TPG=1 VTO=0.6566 DELTA=6.9100E-01 LD=4.7290E-08 KP=1.9647E-04
+ UO=546.2 THETA=2.6840E-01 RSH=3.5120E+01 GAMMA=0.5976
+ NSUB=1.3920E+17 NFS=5.9090E+11 VMAX=2.0080E+05 ETA=3.7180E-02
+ KAPPA=2.8980E-02 CGDO=3.0515E-10 CGSO=3.0515E-10 CGBO=4 0239E-10
+ CJ=5.62E-04 MJ=0.559 CJSW=5.00E-11 MJSW=0.521 PB=0.99
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 4.1080E-07
.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=9.6000E-09 J=0.200000U
+ TPG=-1 VTO=-0.9213 DELTA=2.8750E-01 LD=3.5070E-08 KP=4.8740E-05
+ UO=135.5 THETA=1.8070E-01 RSH=1.1000E-01 GAMMA=0.4673
+ NSUB=8.5120E+16 NFS=6.5000E+11 VMAX=2.5420E+05 ETA=2.4500E-02
+ KAPPA=7.9580E+00 CGDO=2.3922E-10 CGSO=2.3922E-10 CGBO=3.7579E-10
+ CJ=9.35E-04 MJ=0.468 CJSW=2.89E-10 MJSW=0.505 PB=0.99
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 3.6220E-07
.TRAN 5N 1000N
.END

```

The first three lines describe the topology of the circuit with input node 2 and output node 3. Node 0 is connected to ground and node 1 is connected to V_{dd} respectively. In this example, Mn and Mp are the n and p -channel transistors that employ the user-defined models CMOSN and CMOSP, respectively. Their lengths and widths are indicated for the smallest devices in 0.5- μm technology. These models are described in the latter part of the SPICE deck. Only some parameters are specified; the others take default values. All such parameters are technology specific.

A set of differential equations for the voltage, current, and delays is developed and solved numerically. Many other derivatives of this program have been used in industry: HSPICE [Avanti 1997] and PSPICE [Microdim 1994]. SPICE-like simulators are now the standard in circuit-level simulation. One of the major problems with these programs is that under some conditions, the solution does not converge. To overcome this problem, lookup tables are used in the iterative solution process.

3.3 Tabular Format

The tabular form is one of the most popular representations of digital design because of its suitability for use with computer programs, an essential computer-aided design (CAD) tool for present designs. Behavioral or functional

models on the logic level are often represented in a tabular form. Most readers are familiar with truth tables and state tables for defining combinational logic functions and finite state machines from logic design courses.

3.3.1 Truth Tables

A truth table of a combinational logical function, $f(x_1, x_2, \dots, x_n)$, consists of 2^n rows. Each row includes a minterm (a zero cube) and the corresponding value of the function, 0 or 1. Three examples of simple functions are given in Table 3.2.

Table 3.2 Combinational Functions Defined in Tabular Form.

Minterm	$f_1(x_1, x_2, x_3)$	$f_2(x_1, x_2, x_3)$	Minterm	$f_3(x_1, x_2, x_3)$
000	1	1	000	1
001	1	1	0x1	1
010	0	0	010	0
011	0	d	1x0	1
100	1	d	101	0
101	0	0	111	1
110	1	1		
111	1	1		(b)

(a)

The first function, $f(x_1, x_2, x_3) = \sum m(0,1,4,6,7)$, is fully specified. For each minterm the function has one logic value, 0 or 1. The second function is specified incompletely; for minterms 3 and 4, the value of the function is indicated by d , which stands for "don't care." The designer may assign it a value of 1 or 0 for the benefit of simplifying the design. The third function is expressed in terms of 0-cubes as well as 1-cubes, where x implies that the function value is valid when $x = 0$ and 1.

3.3.2 State Tables

State tables easily represent finite state machines. A state table may be in one of the two formats illustrated in Table 3.3 for a machine, M , defined in Section 3.2.3. The first column in Table 3.3a contains the present state (PS) and the remaining columns list, for the various input combinations, the next state (NS) and the output. An alternative form is

shown in Table 3.3*b*. Here the first column represents the input combinations; the second through fourth columns represent the present state, the next state, and the output, respectively.

Table 3.3 State Table for FSM *M*

			<i>I</i>	PS	NS	Output
			0	<i>A</i>	<i>C</i>	1
			1	<i>A</i>	<i>B</i>	0
Present	Next State		0	<i>B</i>	<i>C</i>	0
State	<i>I</i> =0	<i>I</i> =1	1	<i>B</i>	<i>B</i>	1
<i>A</i>	<i>C</i> ,1	<i>B</i> ,0	0	<i>C</i>	<i>D</i>	1
<i>B</i>	<i>C</i> ,0	<i>B</i> ,1	1	<i>C</i>	<i>C</i>	1
<i>C</i>	<i>D</i> ,1	<i>C</i> ,1	0	<i>D</i>	<i>A</i>	1
<i>D</i>	<i>A</i> ,1	<i>C</i> ,0	1	<i>D</i>	<i>C</i>	0
(a)			(b)			

This tabular description of an FSM is useful in synthesis tools as well as in some testing practices, for example, finding the *checking sequence* of the machine [Hachtel 1996, Hennie 1964]. This sequence is to an FSM what exhaustive testing is to combinational circuits. This method is described in Chapter 6.

3.4 Graphical Representation

Graphical representation of a circuit is a convenient notation for designers. We are all familiar with schematic capture on the logic and circuit levels, which are illustrated in Fig. 3.3. As the designs become very large, on the order of 1 million gates, it becomes difficult to display such designs graphically as a whole. However, it is still convenient to deal with one part of the design at a time. All CAD tools continue to display schematic capture even for those designs that have been entered in HDL. This is done for the benefit of the designers to help in visual perception of the design.

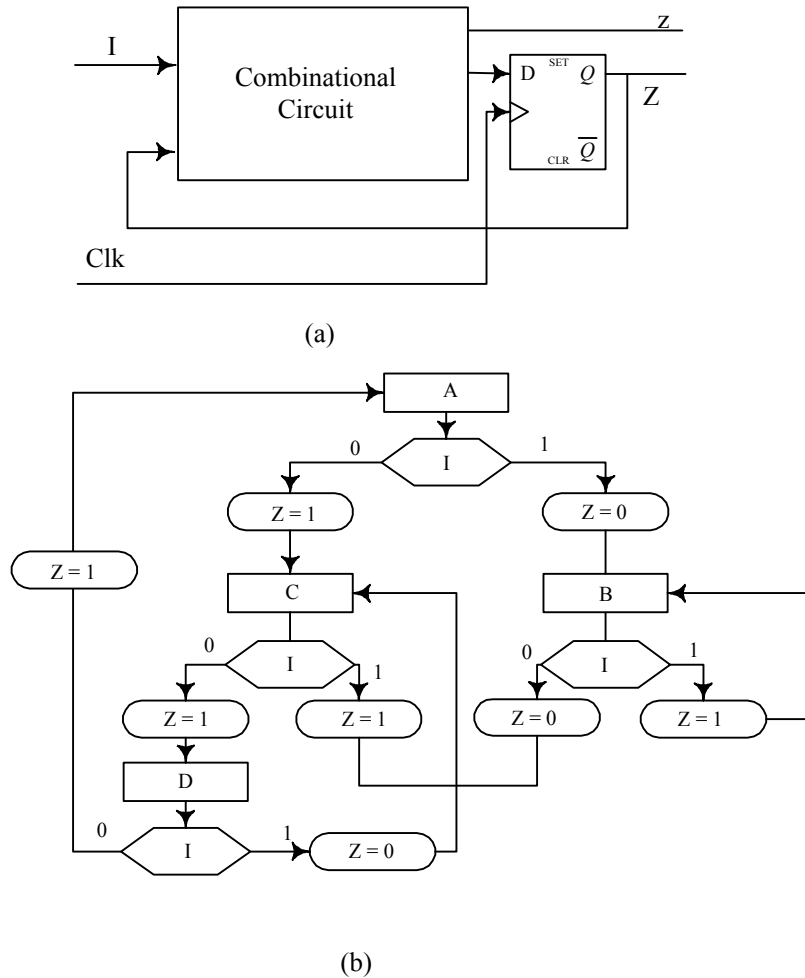


Figure 3.6 Finite State Machine Representations: (a) Huffmann Model, (b) Flowchart Model

A generic model for an FSM is shown in Fig. 3.6a. The combinational part represents the next state forming logic and the output forming logic. The one flip-flop shown represents all flip-flops in the circuit. Here we assume that only D flip-flops are used. This assumption is very realistic and does not limit the applicability of the scheme to other flip-flop types. An FSM may also be represented as a *flowchart*. This is used to represent a counter or any arbitrary control unit of a processor. This is suitable for an algorithmic representation of the FSM. An *algorithmic state machine* (ASM) gives a functional (behavioral) description of the FSM. Figure 3.6b shows the flowchart for the FSM M defined in Section 3.2.3. This representation can become very cumbersome for larger machines. In these cases, using hardware description language would be much more manageable, as we demonstrate later in the chapter.

3.5 Graphs

In this section we introduce the basic concepts of graph theory, which are used whenever needed throughout the book. A graph $G(V,E,W)$ consists of a set of *vertices* V , a set of *edges* E , and a set of *weights* W . The vertices are also called *nodes* and we use these two words interchangeably. An edge $e(u,v) \in E$, where $u \in V$ and $v \in V$ is a relation between vertices u and v . It may also be denoted as (u,v) . If the relation is from u onto v , but not vice versa, the graph is called a *directed graph* (digraph). Figure. 3.7a provides an example of directed graph on $V = \{a,b,c,d,e,f\}$ with $E = \{(a,a), (a,b), (b,c), (d,a), (b,f)\}$. The arrow indicates the direction of the edge. The edge (a,a) is a loop. The vertex a is said to have a self-loop. Nodes related by an edge are called *adjacent*. Node e is not related to any of the other nodes; it is an isolated node. The number of edges emanating from a vertex is called the *outdegree* of the vertex, and the number of edges incoming to the vertex is its *indegree*. Vertex b has an indegree of 1 and an outdegree of 2 since it points to both nodes, c and f .

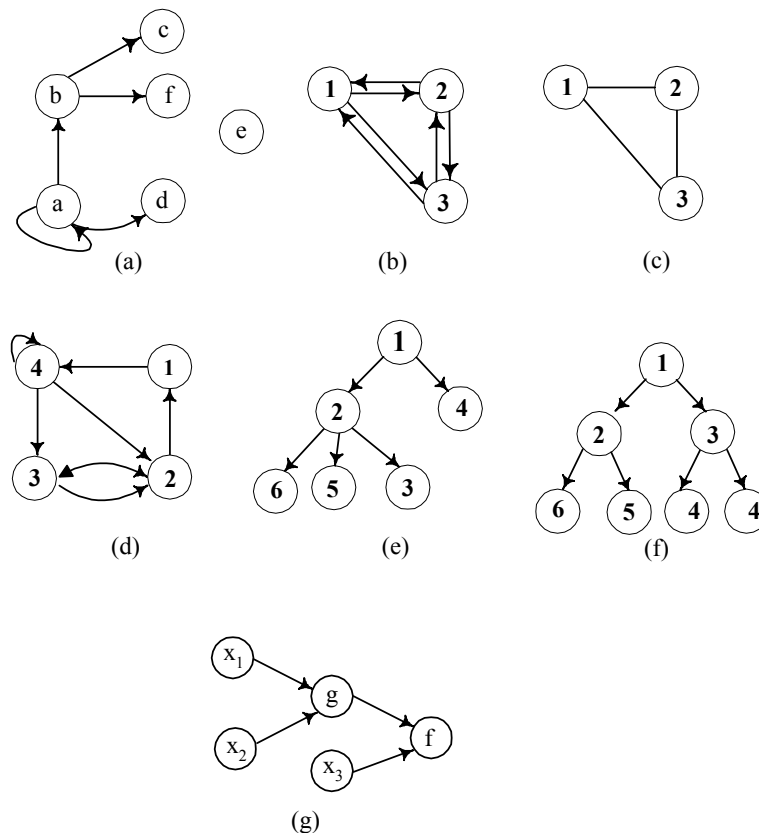


Figure 3.7 Examples of Graphs: (a) Directed Graph (b) Undirected Graph (c) Multi-directed Directed Graph, (d) Directed Graph, (e) A Tree, (f) Binary Tree, (g) Another Directed Graph.

The relation between the vertices may be symmetric. In such a case, we have $e(u,v) \equiv e(v,u)$, and one edge is sufficient to represent the relation in an *undirected graph*. This is illustrated by the graphs in Fig. 3.7b and its simplified equivalent in Fig. 3.7c. In general, the graph is assumed undirected unless it is stated otherwise.

A *path* is a sequence of edges: $(v, u), (u, w), (w, r)$. A graph is said to be *connected* if there is a path from any vertex to any other vertex. The graph in Fig. 3.7a is not connected, whereas that in Fig. 3.7b is connected. A *cycle* is a closed path. The path (a, a) of the graph in Fig. 3.7a is a cycle of length 1. The graph in Fig. 3.7d has cycles of length 1, 2, 3 and 4. A graph is said to be *complete* if it is such that every node has an edge to all other nodes.

A *tree*, $T(V, E, \text{ and } W)$, is an acyclic graph. An example is shown in Fig. 3.7e. It has a root node with indegree 0 and terminal nodes with outdegree 0. Any terminal node is called a *leaf*. A special type of tree, a *binary tree*, is shown in Fig. 3.7f. All of its internal nodes are such that each has an *indegree* of 1 and an *outdegree* of at most 2.

Given a graph, $G(V, E)$, a subgraph of G , $S(U, F)$ is such that $U \subset V$ and $F \subset E$. A subgraph that is complete is called a *clique*. Node e of the graph in Fig. 3.7a can be a subgraph and a clique, although it consists of only one node.

In digital design and testing, graphs are used extensively. The design at any level of representation is modeled as a graph, and operations on the design are developed for its graph model. Operations on graph models of the design include system partitioning, synthesis on logic and behavioral levels, synthesis for testability, test pattern generation, and physical design – placement and routing, back-annotation, simulation, and so on. The schematic capture discussed in the preceding section is easily represented by a graph, as illustrated in Fig. 3.7g for the circuit in Fig. 3.4. The vertices of the diagram are the primary inputs and the results of the operations of the various gates. The edges are the interconnections between the gates. In using this type of a graph for delay calculations, a weight is associated with each edge. To place the gate on a floor of a chip, the weight assigned may be the lengths of the wires connecting the gates. Another important graph representation is that of finite state machines. In this case the graph is called a state transition graph (STG). The vertices of the graph are the states and the edges are the transitions between states. The STG of machine M defined in Section 2.2.3 is shown in Fig. 3.7d. It is an isomorphism of the state transition table given in Table 3.3.

1. The BDD for the OR function can be deduced from that of the AND simply by complementing the labels of the edges and the values of the two leaves, 0 and 1. Another way of developing this diagram is to realize that:

If $x_1 = 0$ then $f = 0$,

else $f = x_2$

if $x_2 = 0$ then $f = 0$,

else $f = 1$.

This explains why in some publications, instead of labeling the edges 0 and 1, they may be labeled T (then) and E (else).

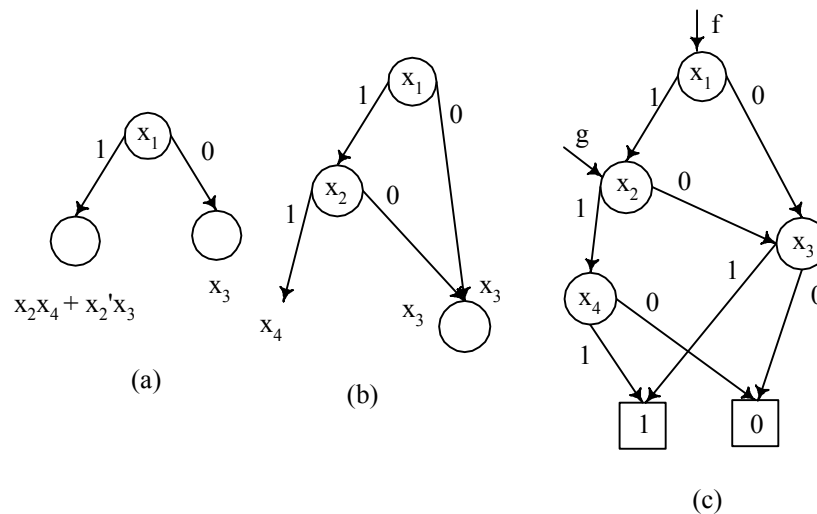


Figure 3.9 Building a BDD for the function

Notice that this function is symmetrical in both variables and the same diagram would have been obtained if we started with either variable. However, in general, the diagram is not unique and it is sensitive to the order in which the variables are considered. Next we consider a more complex function: $f(x_1, x_2, x_3, x_4) = x_1x_2x_4 + x_1'x_3 + x_2'x_3$. We first use the following order of variables: $x_1 \leq x_2 \leq x_3 \leq x_4$. The residues of the function for x_1 and x_1' are $f_{x_1} = x_2x_4 + x_2'x_3$ and $f_{x_1'} = x_3 + x_2'x_3 = x_3$, as illustrated in Fig. 3.9a. Next, we form the residues of $f_{x_1'}$ with respect to x_2 . They are $f_{x_1'x_2} = x_3$ and $f_{x_1'x_2'} = x_4$, respectively. They are added to the graph as shown in Fig. 3.9b. Finally, the residues for x_3 and x_4 are given to complete the diagram. Again, since the function is symmetrical in x_1 and x_2 , we would have obtained the same diagram had we started with x_2 instead of x_1 . The diagram becomes more complex when we start

with one of the other variables. We will follow the order $x_4 \leq x_3 \leq x_2 \leq x_1$. The residues for x_4 are $f_{x_4'} = x_1'x_3 + x_2'x_3$ and $f_{x_4} = x_1x_2 + x_1'x_3 + x_2'x_3 = x_1x_2 + x_3$. For each branch we take the residue with respect to x_3 as illustrated in Fig. 3.10. It is left for the exercises to find the BDD when x_3 is considered before x_4 .

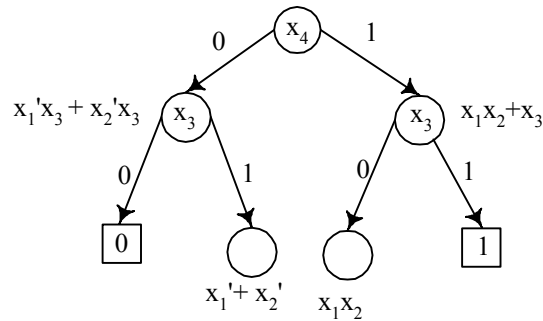


Figure 3.10

For more complex functions, the graph is formed with more than one node for some variables. It then may be reduced by deleting redundant nodes. The development of BDDs by hand is a tedious task. However, there are well-documented algorithms that build such graphs [Hachtel 1996]. The complexity of such an algorithm is of the order $O(2^N/N)$. The graph may include the representation of another function. For example, the function $g(x_2, x_3, x_4) = x_2x_4 + x_3$ is represented by a subgraph in Fig. 3.9b. BDDs can be used to determine various properties of the functions they represent. If we start from the root and trace through a path that leads to 1 (or 0), we find a set of variable values for which the function is 1 (or 0) independent of the other variables. For the diagram in Fig. 3.9, $f = 1$ for $x_1 = 0$ and $x_3 = 1$. This implies that $x_1'x_3$ is a prime implicant of f . Hence finding all paths to 1 would yield the products of the function.

Test pattern generation is another application where BDDs are useful. We will illustrate this using an example. Consider the function $f(x_1, x_2, x_3) = g + x_3 = x_1x_2 + x_3$ that we used in Section 3.2.2 to illustrate the application of Boolean difference in test pattern generation. In that method we assume that we have generated a test pattern for a SA0 on the primary input x_1 . Of course, we will have $x_1 = 1$. The response to this pattern of the good circuit may be 0 or 1, while the faulty circuit response will then be 1 or 0. Thus to find this test pattern, we can trace on the circuit's BDD the paths to 0 and to 1 starting from the root. Any variable other than x_1 , if it appears in both paths, must have the same value, either 1 or 0. For this example, starting from the root we can go through $x_1 = 1$ and $x_2 = 1$ to 1. Also, we trace the path through $x_1 = 0$ to 0 through $x_3 = 0$. The corresponding values of the variables

constitute the test pattern, $x_1x_2x_3 = 110$. The paths leading to 0 and 1 are shown in Fig. 3.11a in solid lines. For $x_2/0$, we start from node x_2 , but we need to recall that this node is reached when $x_1 = 1$. The paths leading to 0 and 1 are shown in Fig. 3.11b. For a stuck-at fault on an internal node, $g = x_1x_2$, we will rearrange the BDD as shown in Fig. 3.11c. The diagram represents both g and f . We need the paths for $g = 0$ and $g = 1$; however, this dictates that x_3 must be equal to 0.

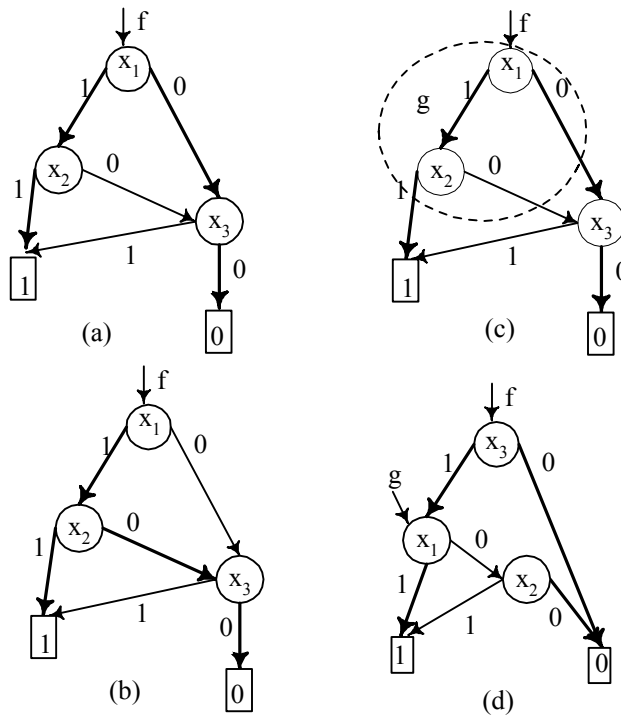


Figure 3.11 Test Pattern Generation using BDDs

3.7 Netlists

A netlist describes a design as a collection of connected modules. The modules may be logic gates or transistors. An example of a netlist is the SPICE deck listed in Section 3.2.4. A netlist consists of a set of records that represents the modules and their connectivities. A record usually gives the name of the module, its function, and the input and outputs to the module. The function is well defined in a library of components. The library may consist of electrical components such as transistors, resistance, and capacitance. It may be a list of logic primitives such as NAND, NOR, XOR, and so on. However, the components could also be macros that represent more complex components. The component may be a core that is defined by its I/O and functionality in a core-based system. Similarly, the components on a system level are RAMs, ALU, and so on. For the logic level they are logic gates (NAND, OR) and on the circuit level (NMOS, PMOS, resistance). Almost every CAD tool has its own representation of a netlist.

The logic-level representations shown in Fig. 3.3 for the structural domain can each be represented as a netlist.

```

AND-NOR (z,b,c,d)
g1      AND2 (a,b,c)
g2      NOR2 (z,a,d)
wire    a

```

"AND-NOR" is the circuit's name. It is associated with a list of primary inputs and outputs b , c , d , and z . Each gate has a name, a logic function, and input and output leads. For example, g_1 is a two-input AND and its inputs are b and c . The output of this gate is an internal node and it is thus declared as "wire" in the last entry of the netlist.

An important netlist representation has been developed for the purpose of facilitating the interchange of designs between different incompatible systems. One representation is the electronic description interchange format (EDIF) that is extremely verbose. Presently, EDIF is a standard language.

3.8 Hardware Description Languages

Hardware description languages (HDLs) are becoming the standard in designing digital systems. In other words, the issue is not whether or not to use HDLs, but which language is more advantageous to use. Although use of an HDL may be at the gate level, designers now prefer to express their designs on the behavioral or register transfer level (RTL) and defer the detail of the implementation to CAD tools. Logic synthesis tools are available to create circuits that readily become mapped into silicon. HDLs have several advantages for HDLs over schematics. Two of the main advantages are efficient management of complexity and shortening the design cycle. As a result of the continuous decrease in the minimum feature size of transistors, device density and design complexity have both increased steadily. Densities of several million transistors are now a reality, and to manage such complexity, it is only natural to design on a hierarchical basis.

In addition to their complexity, the life cycle of modern digital systems is becoming shorter than its design cycle. Thus, increasing the efficiency of the designer, reducing costs, and time-to-market are essential attributes that are needed for keeping competitive in the electronics field. The first HDLs were developed at universities in the mid-1970s. AHPL [Hill 1981] and ISP [Barbacci 1977] are just two examples of many others. Other languages have been used as proprietary languages in industry. However, because of the open system approach in the 1980s, there

was a need to develop a standard for HDLs that could be used on any platform. At present, there are several hardware description languages, the most popular being Hardware C, VHDL, and Verilog HDL.

We concentrate on the two most widely used languages: VHDL and Verilog HDL. Either language mentioned above has the ability of representing

- (1) a hierarchical description of the design and
- (2) the design on three main levels: behavioral, RTL, and gatelevel. These two aspects make the two languages efficient in synthesis. Although designs obtained by synthesis are considered “correct by construction,” it is very important to verify them by simulation. In addition, they may be used for simulation at all levels of abstraction. Unlike software programming language, writing correct design with HDL requires correct semantics as well as syntax. A software program may yield the correct results, although it is not very efficient. However, the “yield” of an HDL description is a design that has to be optimal. As with any programming language, an HDL consists of an alphabet to describe variables and other data structures and syntactical constructs. Some of the constructs are global for the entire design and others are for the description of the design flow.

3.8.1 Verilog Language

Verilog HDL was developed by a CAD tools company, Gateway Design Automation, to use with its main product, the Verilog compiler and simulator. The company was later acquired by Cadence, which made the language public domain. A nonprofit organization, Open Verilog International (OVI), was then founded to support the language throughout its standardization. Verilog HDL, which we will refer to in the remainder of this book simply as Verilog, is presently IEEE Standard 1364-1995. This language resembles C programming language. A design can be described in various levels: switch level, gate level, register transfer level, architectural level and algorithmic level. These various modeling levels may also be used in a mixed style, as illustrated in Fig. 3.12.

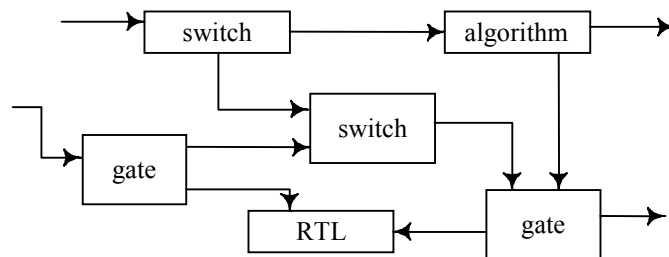


Figure 3.12 Verilog HDL Mixed Mode Design

With increased complexity of digital circuits, it is more important to concentrate on behavioral level design as described below for a full adder, FA_Behav [Bhasker 1998]:

```
module FA_Behav (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout,
  reg Sum, Cout;
  reg R1, R2, R3;
  always
  @(A or B or Cin) begin
    Sum = (A ^ B) ^ Cin;
    R1 = A & Cin;
    R2 = B & Cin;
    R3 = A & B;
    Cout = (R1 | R2) | R3;
  end
endmodule
```

The module defines both the interface (input and output) and the internal structure. In the example, A, B, and Cin are the inputs and Sum and Cout are the outputs. A number of elementary functions are built into the language. They represent basic logic functions such as OR ($|$), AND ($&$), and so on. In addition, *user defined primitive* (UDP) may be defined.

The structure of a circuit is described by making *instances* of modules and primitives within a higher-level module and connecting the instances together using *nets*. A *net* represents an electrical connection, a wire or a bus. A list of *port connections* is used to connect nets to the ports of a module or primitive instance, where a port represents a *pin*. Registers may also be connected to the input ports of an instance. Nets and registers have values formed from the logic values 0, 1, x (uninitialized), and Z (high impedance). In addition to logic values, nets also have a strength value. Strengths are used extensively in switchlevel models and to resolve situations where a net has more than one driver.

The function of a circuit is described using *initial* and *always* constructs. Statements inside an initial or always are in many ways similar to the statements in a software programming language. They are executed at times dictated by timing controls such as delays, and simulation event controls. Statements execute in sequence in a Begin – End block, or in parallel in a For – Join block. Before simulation, the source code is usually compiled. A compiler or an interpreter builds the data files necessary for simulation or synthesis. The simulation is not necessarily deterministic in the sense that the results may not be the same for different simulators because the order of events on an event queue is not defined by the standard. Usually, a test bench, a Verilog module that invokes the design and

applies signals on its inputs and checks the output response versus the expected results is used to run the simulation.

The test bench for the adder module presented above is given in Section 5.2.1.

3.8.2 VHDL Language

VHDL (Very high speed integrated circuit hardware description language) was started in Europe in 1981 and later adopted by the US Department of Defense (DOD) as a design language. In 1983, the DOD awarded a contract to a team of three companies – IBM, Texas Instruments, and Intermetrics – to develop a language. Version 7.2 was developed and released in 1985. The language was approved in 1987 as IEEE Standard 1076 -1987. Nevertheless, VHDL has different dialects. It is an Ada-like language and is highly structured. It describes digital circuit representation in two of the main domains mentioned earlier in the chapter, functional and logic. The language stops short of specifying the physical characteristics of circuit layout.

The language has two principal global constructs: *entity* and *architecture*. The entity declaration specifies the name of the design, its interface to other components, and the input and output ports. Hierarchy is defined by means of components, which are analogous to chip sockets. A component is instantiated within architecture to represent a copy of lower-level hierarchical blocks. The structure of a circuit is described by making instances of components within architecture and connecting the instances together using signal. A *signal* represents an electrical connection, a wire, or a bus. A port map is used to connect signals to the ports of a component instantiation, where a port represents a pin.

References

- Akers, S. B. (1978), Binary decision diagrams, *IEEE Trans. on Comput.*, Vol. C-27, No. 6, pp. 509 – 516.
- Antognetti, P. and G. Masobrio (Eds.) (1988), *Semiconductor Device Modeling with SPICE*, McGraw-Hill, New York.
- Avanti (1997), HSPICE, Avanti Corporation, Milpitas CA.
- Barbacci, M. R. et al. (1977), *The Symbolic Manipulation of Computer Descriptions: ISPS Primer and reference manual*, Research Report, Department of Computer Science, Carnegie – Mellon University., Pittsburgh, PA. Also, *IEEE Trans. on Comput.*, Vol. C-30, No. 1, pp. 24 – 40.
- Bhasker, J. (1998), *A Verilog HDL Primer*, Star Galaxy Press, Allentown, PA.
- Bryant, R. E. (1986), Graph based algorithms for Boolean function manipulation, *IEEE Trans. on Comput.*, Vol. C-35, No. 8, pp. 677 – 691.
- Cheng, Y. et al. (1996), *BSIM3v3 Manual*, Department of Electrical Engineering and Computers Science, University of California, Berkeley, CA.

- Fujiwara, H. (1986), *Logic Testing and Design for Testability*, MIT Press, Cambridge, MA.
- Gajski, D. D. and R. K. Kuhn (1983), Introduction: New VLSI Tools, *IEEE Comput.*, Vol. 6, No.12, pp. 11– 14.
- Getreu, I. (1976), *Modelling the Bipolar transistor*, Teknatrix, Inc., Beaverton, OR.
- Hachtel, G. D. and F. Somenzi (1996), *Logic Synthesis and Verification Algorithms*, Kluwer Academic, Norwell, MA.
- Hennie, F. C. (1974), Fault detection experiments for sequential circuits, *Proc. 5th Symposium on Switching Theory and Logical Design*, pp. 95 – 110.
- Hill, F. and G. Peterson (1981), *Introduction to Switching Theory and Logical Design*, Wiley, New York.
- Hodges, D. A. and H. G. Jackson (1983), *Analysis and Design of Digital Integrated Circuits*, McGraw-Hill, New York.
- Jeng, M. C. et al. (1983), *Theory, Algorithms, and User's Guide for BSIM and SCALP*, Electronic Research Laboratory Memorandum, UCB/ERL M87/35, University of California, Berkeley, CA.
- Kohavi, Z. (1978), *Switching and Finite Automata Theory*, McGraw Hill, New York.
- Lee, C.Y. (1959), Binary Decision Diagrams, *Bell System Technical Journal*, Vol. 38, No. 7, pp. 985 – 999.
- McCluskey, E. J. (1986), *Principles of Logic Design*, Prentice Hall, Upper Saddle River, NJ.
- Mealy, G. H. (1955), A method for synthesizing sequential circuits, *Bell Sys. Tech. J.*, Vol. 34, No. 9, pp. 1045 – 1079.
- Michel, P., U. Lauther, and P. Duzy (Eds.), (1992), *The Synthesis approach to Digital System Design*, Kluwer Academic, Norwell, Hingham, MA.
- Microsim (1994), *PSPICE The Design Center*, MicroSim Corporation, Los Angeles, CA.
- Miczko, A. (1986), *Digital Logic Testing and Simulation*, Harper & Row, New York.
- Moore, E. F. (1956), Gedanken experiments on sequential machines, in *Automata Studies*, C.E. Shannon and J. McCarthy (Eds.), Princeton University Press, Princeton, NJ.
- Nagel, L. W., (1975) *SPICE2: A Computer Program to Simulate Semiconductor Circuit*, Memo ERL-M520, University of California, Berkeley, CA.
- Quarles, T. et al. (1994), *SPICE 3 Version 3F5 User's Manual*, University of California, Berkeley, CA
- Sheu, B.J. et al. (1988), BSIM, Berkeley short-channel IGFET model, *IEEE J. of Solid-State Circuits*, Vol. SC-22, pp. 558 – 566.
- Walker, R. A. and D. E. Thomas (1985), A model of design representation and synthesis, *Proc. 22nd Design Automation Conference*, pp. 453 – 459.

Problems

- 3.1 Use Boolean difference to develop a test set for stuck-at faults on the primary inputs of the circuit in Fig. P3.1. Then, with the help of a fault simulator, determine the fault coverage for all stuck-at faults for this test.
- 3.2 Develop the BDD for the functions (a) $f(a,b,c,d,e,g) = ab + cd + eg$ and (b) $f(a,b,c,d) = a \oplus b \oplus c \oplus d$.
- 3.3 Use the BDDs that you developed in Problem 3.2 to determine the test patterns that detect stuck-at faults on inputs a and b .
- 3.4 Develop the state table and the state diagram for a synchronous sequential circuit with one input line and one output line that recognizes the input string $x = 1111$. The circuit is also required to recognize overlapping sequences, as can be seen in the output string that results from the following input string: $x = 1100111111101$. The output string is 00000001111000.
- 3.5 Draw the flowchart for the FSM of Problem 3.4.
- 3.6 For the circuit in Fig. P3.6, develop the BDD (use only one leaf for 1 and one leaf for 0). Then find all test patterns to detect stuck-at faults on the primary outputs.
- 3.7 Follow the example used in Section 3.7 and develop the netlist for the circuit shown in Fig. 2.6.
- 3.8 Write a Verilog structural code for the circuit used in Problem 3.7.

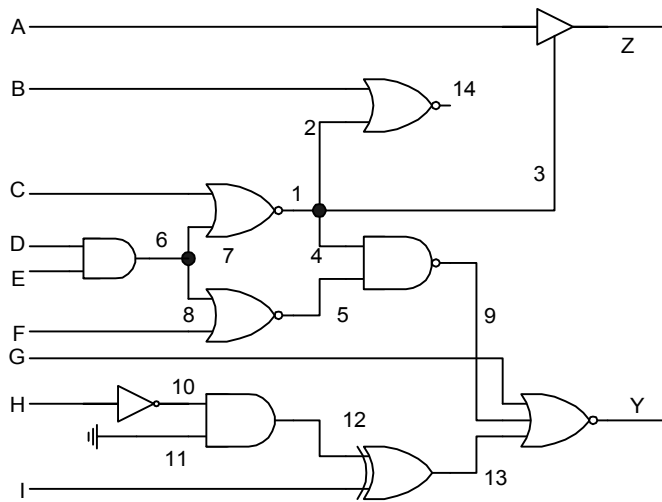


Figure P3.1

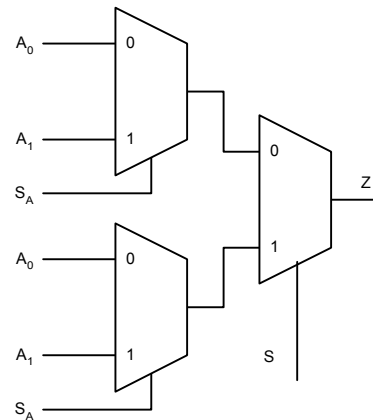


Figure P3.6