2. MODELING

About This Chapter

Modeling plays a central role in the design, fabrication, and testing of a digital system. The way we represent a system has important consequences for the way we simulate it to verify its correctness, the way we model faults and simulate it in the presence of faults, and the way we generate tests for it.

First we introduce the basic concepts in modeling, namely behavioral versus functional versus structural models and external versus internal models. Then we discuss modeling techniques. This subject is closely intertwined with some of the problems dealt with in Chapter 3. This chapter will focus mainly on the description and representation of models, while Chapter 3 will emphasize algorithms for their interpretation. One should bear in mind, however, that such a clear separation is not always possible.

In the review of the basic modeling techniques, we first discuss methods that describe a circuit or system as one "box"; circuits are modeled at the logic level and systems at the register level. Then we present methods of representing an interconnection of such boxes (structural models).

2.1 Basic Concepts

Behavioral, Functional, and Structural Modeling

At any level of abstraction, a digital system or circuit can be viewed as a *black box*, processing the information carried by its inputs to produce its output (Figure 2.1). The I/O mapping realized by the box defines the **behavior** of the system. Depending on the level of abstraction, the behavior can be specified as a mapping of logic values, or of data words, etc. (see Figure 1.1). This transformation occurs over time. In describing the behavior of a system, it is usually convenient to separate the value domain from the time domain. The **logic function** is the I/O mapping that deals only with the value transformation and ignores the I/O timing relations. A **functional model** of a system is a representation of its logic function. A **behavioral model** consists of a functional model coupled with a representation of the associated timing relations. Several advantages result from the separation between logic function and timing. For example, circuits that realize the same function but differ in their timing can share the same functional model. Also the function and the timing can be dealt with separately for design verification and test generation. This distinction between function and behavior is not always made in the literature, where these terms are often used interchangeably.

A **structural model** describes a box as a collection of interconnected smaller boxes called **components** or **elements**. A structural model is often *hierarchical* such that a component is in turn modeled as an interconnection of lower-level components. The bottom-level boxes are called **primitive elements**, and their functional (or behavioral) model is assumed to be known. The function of a component is shown by its **type**. A *block diagram* of a computer system is a structural model in which the types of the

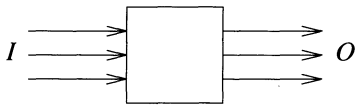


Figure 2.1 Black box view of a system

components are CPUs, RAMs, I/O devices, etc. A *schematic diagram* of a circuit is a structural model using components whose types might be AND, OR, 2-to-4 DECODER, SN7474, etc. The type of the components may be denoted graphically by special shapes, such as the shapes symbolizing the basic gates.

A structural model always carries (implicitly or explicitly) information regarding the function of its components. Also, only small circuits can be described in a strictly functional (black-box) manner. Many models characterized as "functional" in the literature also convey, to various extents, some structural information. *In practice structural and functional modeling are always intermixed.*

External and Internal Models

An **external model** of a system is the model viewed by the user, while an **internal model** consists of the data structures and/or programs that represent the system inside a computer. An external model can be graphic (e.g., a schematic diagram) or text based. A text-based model is a description in a formal language, referred to as a *Hardware Description Language* (HDL). HDLs used to describe structural models are called *connectivity languages.* HDLs used at the register and instruction set levels are generally referred to as *Register Transfer Languages* (RTLs). Like a conventional programming language, an RTL has *declarations* and *statements.* Usually, declarations convey structural information, stating the existence of hardware entities such as registers, memories, and busses, while statements convey functional information by describing the way data words are transferred and transformed among the declared entities.

Object-oriented programming languages are increasingly used for modeling digital systems [Breuer *et al.* 1988, Wolf 1989].

2.2 Functional Modeling at the Logic Level

2.2.1 Truth Tables and Primitive Cubes

The simplest way to represent a combinational circuit is by its **truth table**, such as the one shown in Figure 2.2(a). Assuming binary input values, a circuit realizing a function $Z(x_1, x_2, ..., x_n)$ of n variables requires a table with 2^n entries. The data structure representing a truth table is usually an array V of dimension 2^n. We arrange the input combinations in their increasing binary order. Then $V(0) = Z(0,0,...,0)$, $V(1) = Z(0,0,...,1)$, ..., $V(2^n-1) = Z(1,1,...,1)$. A typical procedure to determine the value of Z, given a set of values for $x_1, x_2, ..., x_n$, works as follows:

1. Concatenate the input values in proper order to form one binary word.
2. Let i be the integer value of this word.
3. The value of Z is given by $V(i)$.

x_1	x_2	x_3	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

x_1	x_2	x_3	Z
x	1	0	0
1	1	x	0
x	0	x	1
0	x	1	1

(b)

Figure 2.2 Models for a combinational function $Z(x_1, x_2, x_3)$ (a) truth table (b) primitive cubes

For a circuit with m outputs, an entry in the V array is an m-bit vector defining the output values. Truth table models are often used for Read-Only Memories (ROMs), as the procedure outlined above corresponds exactly to the ROM addressing mechanism.

Let us examine the first two lines of the truth table in Figure 2.2(a). Note that $Z = 1$ independent of the value of x_3. Then we can compress these two lines into one whose content is $00x \mid 1$, where x denotes an unspecified or "don't care" value. (For visual clarity we use a vertical bar to separate the input values from the output values). This type of compressed representation is conveniently described in *cubical notation.*

A **cube** associated with an ordered set of signals $(a_1\ a_2\ a_3\ \cdots)$ is a vector $(v_1\ v_2\ v_3\ \cdots)$ of corresponding signal values. A *cube of a function* $Z(x_1, x_2, x_3)$ has the form $(v_1\ v_2\ v_3 \mid v_Z)$, where $v_Z = Z(v_1, v_2, v_3)$. Thus a cube of Z can represent an entry in its truth table. An implicant g of Z is represented by a cube constructed as follows:

1. Set $v_i = 1(0)$ if x_i $(\overline{x_i})$ appears in g.
2. Set $v_i = x$ if neither x_i nor $\overline{x_i}$ appears in g.
3. Set $v_Z = 1$.

For example, the cube $00x \mid 1$ represents the implicant $\bar{x}_1\bar{x}_2$.

If the cube q can be obtained from the cube p by replacing one or more x values in p by 0 or 1, we say that p **covers** q. The cube $00x \mid 1$ covers the cubes $000 \mid 1$ and $001 \mid 1$.

A cube representing a prime implicant of Z or $\bar{Z}$ is called a **primitive cube**. Figure 2.2(b) shows the primitive cubes of the function Z given by the truth table in Figure 2.2(a).

To use the primitive-cube representation of Z to determine its value for a given input combination $(v_1\ v_2\ \cdots\ v_n)$, where the v_i values are binary, we search the primitive cubes of Z until we find one whose input part covers v. For example, $(v_1\ v_2\ v_3) = 010$ matches (in the first three positions) the primitive cube $x10|0$; thus Z(010) = 0. Formally, this matching operation is implemented by the *intersection operator*, defined by the table in Figure 2.3. The intersection of the values 0 and 1 produces an *inconsistency*, symbolized by $\varnothing$. In all other cases the intersection is said to be *consistent*, and values whose intersection is consistent are said to be *compatible*.

$\cap$	0	1	x
0	0	$\varnothing$	0
1	$\varnothing$	1	1
x	0	1	x

Figure 2.3 Intersection operator

The intersection operator extends to cubes by forming the pairwise intersections between corresponding values. The intersection of two cubes is consistent iff* all the corresponding values are compatible. To determine the value of Z for a given input combination $(v_1\ v_2\ \cdots\ v_n)$, we apply the following procedure:

1. Form the cube $(v_1\ v_2\ \cdots\ v_n|\ x)$.
2. Intersect this cube with the primitive cubes of Z until a consistent intersection is obtained.
3. The value of Z is obtained in the rightmost position.

Primitive cubes provide a compact representation of a function. But this does not necessarily mean that an internal model for primitive cubes consumes less memory than the corresponding truth-table representation, because in an internal model for a truth table we do not explicitly list the input combinations.

Keeping the primitive cubes with $Z = 0$ and the ones with $Z = 1$ in separate groups, as shown in Figure 2.2(b), aids in solving a problem often encountered in test generation. The problem is to find values for a subset of the inputs of Z to produce a specified value for Z. Clearly, the solutions are given by one of the two sets of primitive cubes of Z. For

* if and only if

example, from Figure 2.2(b), we know that to set $Z = 1$ we need either $x_2 = 0$ or $x_1x_3 = 01$.

2.2.2 State Tables and Flow Tables

A finite-state sequential function can be modeled as a *sequential machine* that receives inputs from a finite set of possible inputs and produces outputs from a finite-set of possible outputs. It has a finite number of *internal states* (or simply, states). A finite state sequential machine can be represented by a *state table* that has a row corresponding to every internal state of the machine and a column corresponding to every possible input. The entry in row q_i and column I_m represents the next state and the output produced if I_m is applied when the machine is in state q_i. This entry will be denoted by $N(q_i,I_m)$, $Z(q_i,I_m)$. N and Z are called the *next state* and *output function* of the machine. An example of a state table is shown in Figure 2.4.

		x = 0	x = 1
q	1	2,1	3,0
	2	2,1	4,0
	3	1,0	4,0
	4	3,1	3,0

$N(q,x),Z(q,x)$

Figure 2.4 Example of state table (q = current state, x = current input)

In representing a sequential function in this way there is an inherent assumption of *synchronization* that is not explicitly represented by the state table. The inputs are synchronized in time with some timing sequence, $t(1)$, $t(2)$, ..., $t(n)$. At every time $t(i)$ the input is sampled, the next state is entered and the next output is produced. A circuit realizing such a sequential function is a *synchronous sequential circuit.*

A synchronous sequential circuit is usually considered to have a *canonical structure* of the form shown in Figure 2.5. The combinational part is fed by the primary inputs x and by the state variables y. Every state in a state table corresponds to a different combination of values of the state variables. The concept of synchronization is explicitly implemented by using an additional input, called a *clock* line. Events at times $t(1)$, $t(2)$, ..., are initiated by pulses on the clock line. The state of the circuit is stored in bistable memory elements called *clocked flip-flops* (F/Fs). A clocked F/F can change state only when it receives a clock pulse. Figure 2.6 shows the circuit symbols and the state tables of three commonly used F/Fs (C is the clock).

More general, a synchronous sequential circuit may have several clock lines, and the clock pulses may propagate through some combinational logic on their way to F/Fs.

Sequential circuits can also be designed without clocks. Such circuits are called *asynchronous.* The behavior of an asynchronous sequential circuit can be defined by a *flow table.* In a flow table, a state transition may involve a sequence of state changes,

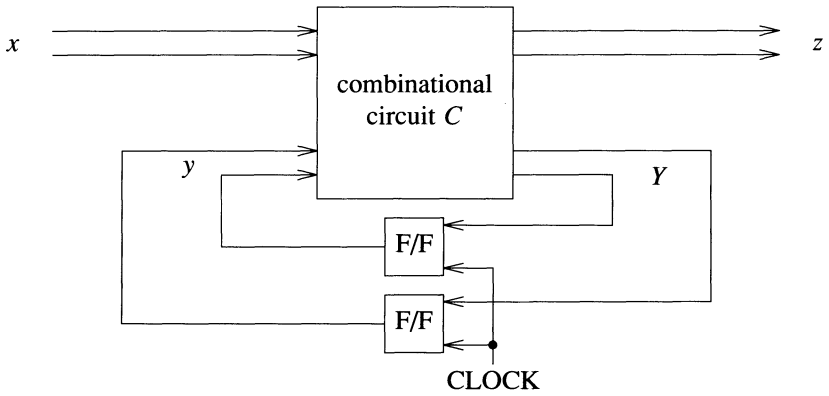


Figure 2.5 Canonical structure of a synchronous sequential circuit

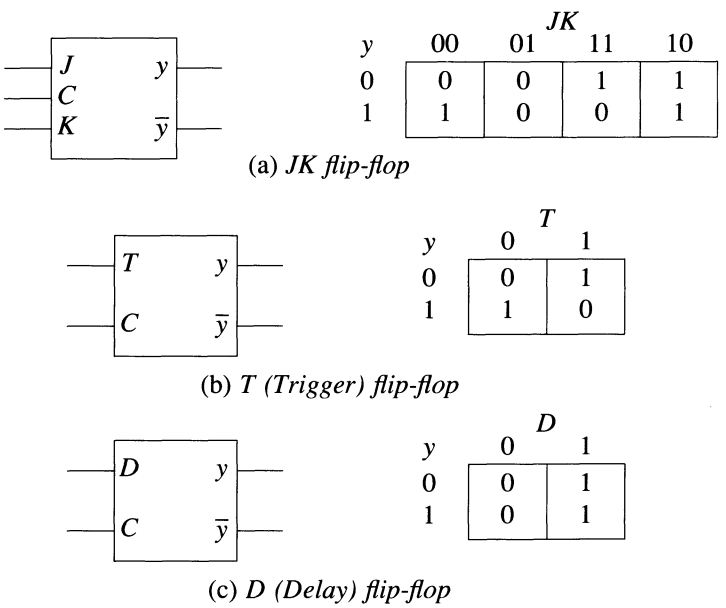


Figure 2.6 Three types of flip-flops

caused by a single input change to I_j, until a stable configuration is reached, denoted by the condition $N(q_i, I_j) = q_i$. Such stable configurations are shown in boldface in the flow table. Figure 2.7 shows a flow table for an asynchronous machine, and Figure 2.8 shows the canonical structure of an asynchronous sequential circuit.

	x_1x_2 00	01	11	10
1	**1**,0	5,1	2,0	**1**,0
2	1,0	**2**,0	**2**,0	5,1
3	**3**,1	2,0	4,0	**3**,0
4	3,1	5,1	**4**,0	**4**,0
5	3,1	**5**,1	4,0	**5**,1

Figure 2.7 A flow table

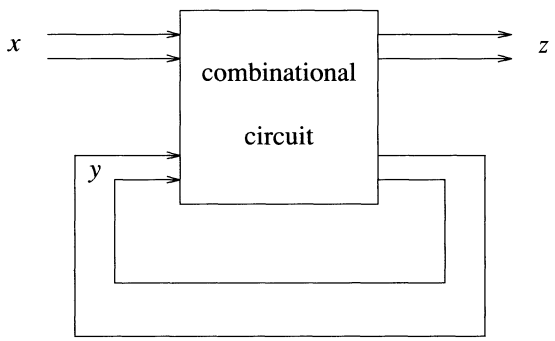


Figure 2.8 Canonical structure of an asynchronous sequential circuit

For test generation, a synchronous sequential circuit S can be modeled by a pseudocombinational iterative array as shown in Figure 2.9. This model is equivalent to the one given in Figure 2.5 in the following sense. Each cell $C(i)$ of the array is identical to the combinational circuit C of Figure 2.5. If an input sequence $x(0)\,x(1)...x(k)$ is applied to S in initial state $y(0)$, and generates the output sequence $z(0)\,z(1)...z(k)$ and state sequence $y(1)\,y(2)...y(k+1)$, then the iterative array will generate the output $z(i)$ from cell i, in response to the input $x(i)$ to cell i $(1 \leq i \leq k)$. Note that the first cell also receives the values corresponding to $y(0)$ as inputs. In this transformation the clocked F/Fs are modeled as combinational elements, referred to as pseudo-F/Fs. For a JK F/F, the inputs of the combinational model are the present state q and the excitation inputs J and K, and the outputs are the next state q^+ and the device outputs y and $\bar{y}$. The present state q of the F/Fs in cell i must be equal to the q^+ output of the F/Fs in cell $i-1$. The combinational element model corresponding to a JK F/F is defined by the truth table in Figure 2.10(a). Note that here $q^+ = y$. Figure 2.10(b) shows the general F/F model.

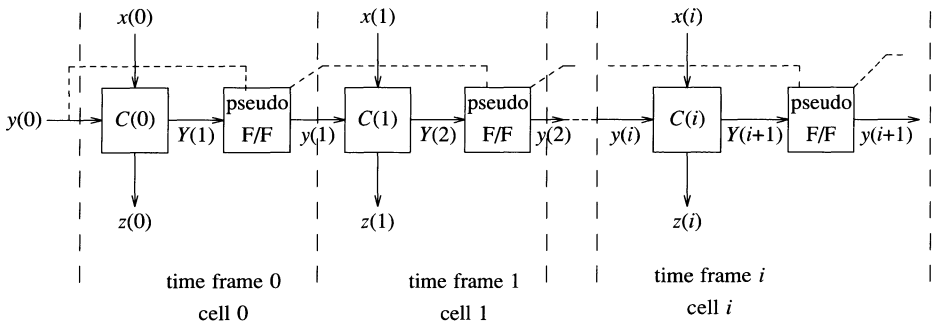
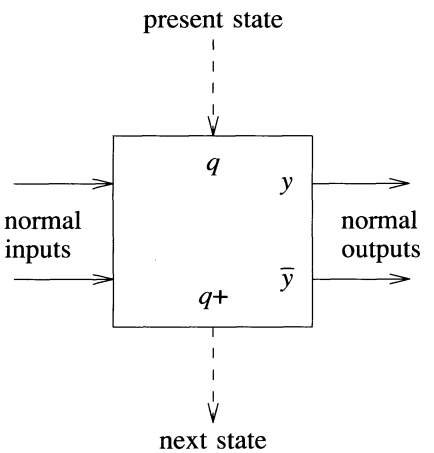


Figure 2.9 Combinational iterative array model of a synchronous sequential circuit

q	J	K	q^+	y	$\bar{y}$
0	0	0	0	0	1
0	0	1	0	0	1
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	1	0	1	1	0
1	1	1	0	0	1

(a)



(b)

Figure 2.10 (a) Truth table of a *JK* pseudo-F/F (b) General model of a pseudo-F/F

This modeling technique maps the time domain response of the sequential circuit into a space domain response of the iterative array. Note that the model of the combinational part need not be actually replicated. This transformation allows test generation methods developed for combinational circuits to be extended to synchronous sequential circuits. A similar technique exists for asynchronous sequential circuits.

2.2.3 Binary Decision Diagrams

A *binary decision diagram* [Lee 1959, Akers 1978] is a graph model of the function of a circuit. A simple graph traversal procedure determines the value of the output by sequentially examining values of its inputs. Figure 2.11 gives the diagram of $f = \bar{a}b\bar{c} + ac$. The traversal starts at the top. At every node, we decide to follow the left or the right branch, depending on the value (0 or 1) of the corresponding input variable. The value of the function is determined by the value encountered at the exit branch. For the diagram in Figure 2.11, let us compute f for abc=001. At node a we take the left branch, then at node b we also take the left branch and exit with value 0 (the reader may verify that when a=0 and b=0, f does not depend on the value of c). If at an exit branch we encounter a variable rather than a value, then the value of the function is the value of that variable. This occurs in our example for a=1; here f=c.

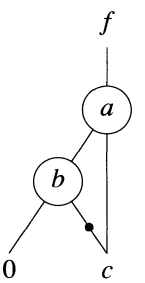


Figure 2.11 Binary decision diagram of $f = \bar{a}b\bar{c} + ac$

When one dot is encountered on a branch during the traversal of a diagram, then the final result is complemented. In our example, for a=0 and b=1, we obtain f=$\bar{c}$. If more than one dot is encountered, the final value is complemented if the number of dots is odd.

Binary decision diagrams are also applicable for modeling sequential functions. Figure 2.12 illustrates such a diagram for a *JK* F/F with asynchronous set (S) and reset (R) inputs. Here q represents the previous state of the F/F. The diagram can be entered to determine the value of the output y or $\bar{y}$. For example, when computing y for S=0, R=0, C=1, and q=1, we exit with the value of K inverted (because of the dot), i.e., y=$\bar{K}$. The outputs of the F/F are undefined (x) for the "illegal" condition S = 1 and R = 1.

The following example [Akers 1978] illustrates the construction of a binary decision diagram from a truth table.

Example 2.1: Consider the truth table of the function $f = \bar{a}b\bar{c} + ac$, given in Figure 2.13(a). This can be easily mapped into the binary decision diagram of Figure 2.13(b), which is a complete binary tree where every path corresponds to one of the eight rows of the truth table. This diagram can be simplified as follows. Because both branches from the leftmost node c result in the same value 0, we remove this

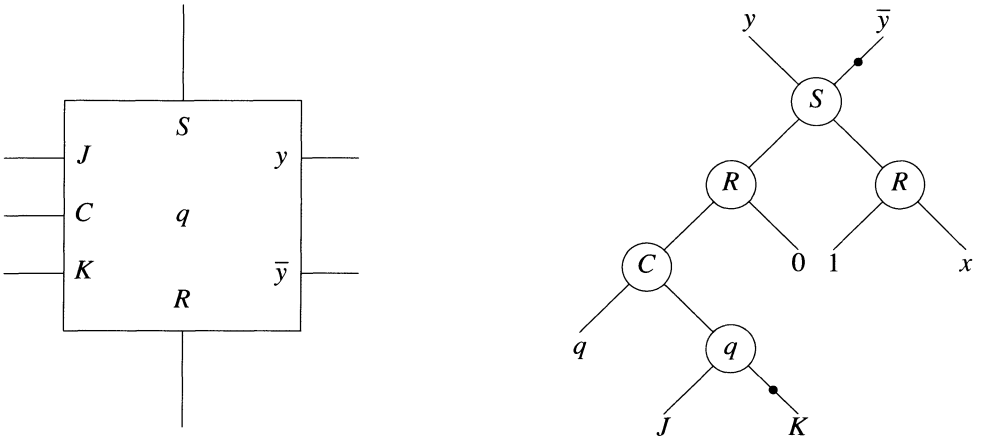


Figure 2.12 Binary decision diagram for a *JK* F/F

node and replace it by an exit branch with value 0. Because the left and right branches from the other c nodes lead to 0 and 1 (or 1 and 0) values, we remove these nodes and replace them with exit branches labeled with c (or $\overline{c}$). Figure 2.13(c) shows the resulting diagram. Here we can remove the rightmost b node (Figure 2.13(d)). Finally, we merge branches leading to c and $\overline{c}$ and introduce a dot to account for inversion (Figure 2.13(e)). □

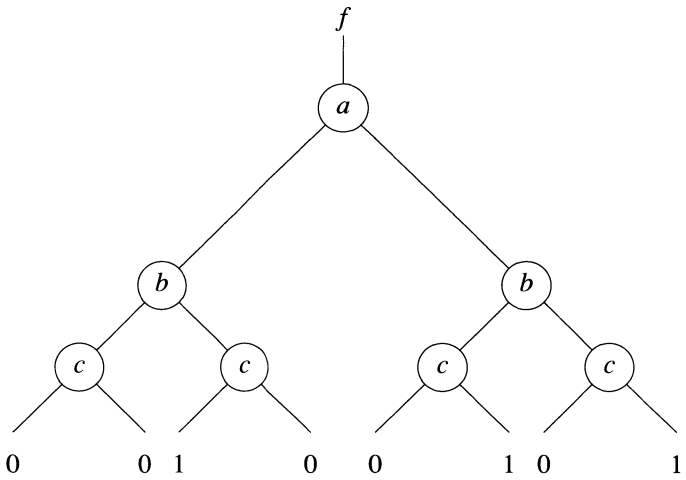
2.2.4 Programs as Functional Models

A common feature of the modeling techniques presented in the previous sections is that the model consists of a data structure (truth table, state table, or binary decision diagram) that is interpreted by a model-independent program. A different approach is to model the function of a circuit directly by a program. This type of code-based modeling is always employed for the primitive elements used in a structural model. In general, models based on data structures are easier to develop, while code-based models can be more efficient because they avoid one level of interpretation. (Modeling options will be discussed in more detail in a later section).

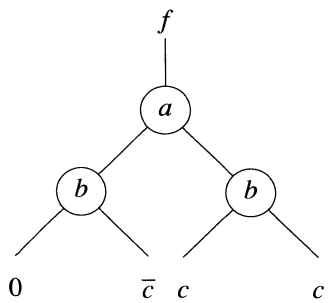
In some applications, a program providing a functional model of a circuit is automatically generated from a structural model. If only binary values are considered, one can directly map the logic gates in the structural model into the corresponding logic operators available in the target programming language. For example, the following assembly code can be used as a functional model of the circuit shown in Figure 2.14 (assume that the variables *A,B,...,Z* store the binary values of the signals *A,B,...,Z*):

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

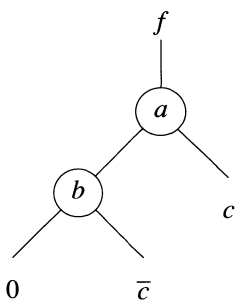
(a)



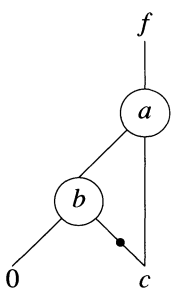
(b)



(c)



(d)



(e)

Figure 2.13 Constructing a binary decision diagram

```
LDA A    /* load accumulator with value of A */
AND B    /* compute A.B */
AND C    /* compute A.B.C */
STA E    /* store partial result */
LDA D    /* load accumulator with value of D */
INV      /* compute D̄ */
OR E     /* compute A.B.C + D̄ */
STA Z    /* store result */
```

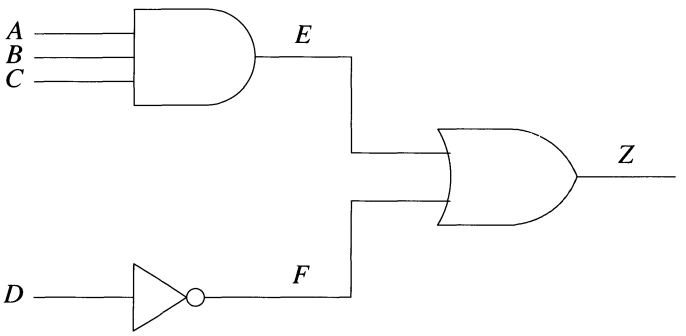


Figure 2.14

Using the C programming language, the same circuit can be modeled by the following:

$$E = A \,\&\, B \,\&\, C$$
$$F = \sim D$$
$$Z = E \mid F$$

As the resulting code is compiled into machine code, this type of model is also referred to as a *compiled-code* model.

2.3 Functional Modeling at the Register Level

2.3.1 Basic RTL Constructs

RTLs provide models for systems at the register and the instruction set levels. In this section we will discuss only the main concepts of RTL models; the reader may refer to [Dietmeyer and Duley 1975, Barbacci 1975, Shahdat *et al.* 1985] for more details. Data (and control) words are stored in registers and memories organized as arrays of registers. Declaring the existence of registers inside the modeled box defines a *skeleton structure* of the box. For example:

register *IR* [0→7]

defines an 8-bit register *IR*, and

memory *ABC* [0→255; 0→15]

denotes a 256-word memory *ABC* with 16-bit/word.

The data paths are implicitly defined by describing the processing and the transfer of data words among registers. RTL models are characterized as functional, because they emphasize functional description while providing only summary structural information.

The processing and the transfer of data words are described by reference to *primitive operators*. For example, if *A*, *B*, and *C* are registers, then the statement

$$C = A + B$$

denotes the addition of the values of *A* and *B*, followed by the transfer of the result into C, by using the primitive operators "+" for addition and "=" for transfer. This notation does not specify the hardware that implements the addition, but only implies its existence.

A reference to a primitive operator defines an *operation*. The control of data transformations is described by *conditional operations*, in which the execution of operations is made dependent on the truth value of a control condition. For example,

if *X* **then** $C = A + B$

means that $C = A + B$ should occur when the control signal *X* is 1. More complex control conditions use Boolean expressions and relational operators; for example:

if (*CLOCK* **and** (*AREG* < *BREG*)) **then** *AREG* = *BREG*

states that the transfer *AREG* = *BREG* occurs when *CLOCK* = 1 *and* the value contained in *AREG* is smaller than that in *BREG*.

Other forms of control are represented by constructs similar to the ones encountered in programming languages, such as 2-way decisions (**if...then...else**) and multiway decisions. The statement below selects the operation to be performed depending on the value contained in the bits 0 through 3 of the register *IR*.

test (*IR*[0→3])
  case 0: operation$_0$
  case 1: operation$_1$
  .
  .
  .
  .
  case 15: operation$_{15}$
testend

This construct implies the existence of a hardware decoder.

RTLs provide compact constructs to describe hardware addressing mechanisms. For example, for the previously defined memory *ABC*,

ABC[3]

denotes the word at the address 3, and, assuming that *BASEREG* and *PC* are registers of proper dimensions, then

ABC[*BASEREG* + *PC*]

denotes the word at the address obtained by adding the current values of the two registers.

Combinational functions can be directly described by their equations using Boolean operators, as in

$$Z = (A \textbf{ and } B) \textbf{ or } C$$

When Z, A, B, and C are n-bit vectors, the above statement implies n copies of a circuit that performs the Boolean operations between corresponding bits.

Other primitive operators often used in RTLs are shift and count. For example, to shift *AREG* right by two positions and to fill the left positions with a value 0, one may specify

shift_right (*AREG*, 2, 0)

and incrementing *PC* may be denoted by

incr (*PC*)

Some RTLs allow references to past values of variables. If time is measured in "time units," the value of X two time units ago would be denoted by $X(-2)$ [Chappell *et al.* 1976]. An action caused by a positive edge (0 to 1 transition) of X can be activated by

if ($X(-1)$=0 **and** X=1) **then** ...

where X is equivalent to $X(0)$ and represents the current value of X. This concept introduces an additional level of abstraction, because it implies the existence of some memory that stores the "history" of X (up to a certain depth).

Finite-state machines can be modeled in two ways. The *direct approach* is to have a *state register* composed of the state variables of the system. State transitions are implemented by changing the value of the state register. A more *abstract approach* is to partition the operation of the system into disjoint blocks representing the states and identified by *state names*. Here state transitions are described by using a **go to** operator, as illustrated below:

```
state  S1, S2, S3

S1:  if X then
        begin
          P = Q + R
          go to S2
        end
     else
          P = Q − R
          go to S3
S2: ...
```

Being in state *S1* is an implicit condition for executing the operations specified in the *S1* block. Only one state is active (current) at any time. This more abstract model allows a finite state machine to be described before the state assignment (the mapping between state names and state variables) has been done.

2.3.2 Timing Modeling in RTLs

According to their treatment of the concept of time, RTLs are divided into two categories, namely *procedural languages* and *nonprocedural languages*. A procedural RTL is similar to a conventional programming language where statements are sequentially executed such that the result of a statement is immediately available for the following statements. Thus, in

$$A = B$$
$$C = A$$

the value transferred into C is the new value of A (i.e., the contents of B). Many procedural RTLs directly use (or provide extensions to) conventional programming languages, such as Pascal [Hill and vanCleemput 1979] or C [Frey 1984].

By contrast, the statements of a nonprocedural RTL are (conceptually) executed in parallel, so in the above example the old value of A (i.e., the one before the transfer $A = B$) is loaded into C. Thus, in a nonprocedural RTL, the statements

$$A = B$$
$$B = A$$

accomplish an exchange between the contents of A and B.

Procedural RTLs are usually employed to describe a system at the instruction set level of abstraction. An implicit cycle-based timing model is often used at this level. Reflecting the instruction cycle of the modeled processor, during which an instruction is fetched, decoded, and executed, a cycle-based timing model assures that the state of the model accurately reflects the state of the processor at the end of a cycle.

More detailed behavioral models can be obtained by specifying the delay associated with an operation, thus defining when the result of an operation becomes available. For example:

$$C = A + B, \textbf{ delay } = 100$$

shows that C gets its new value 100 time units after the initiation of the transfer.

Another way of defining delays in an RTL is to specify delays for variables, rather than for operations. In this way, if a declaration

delay C 100

has been issued, then this applies to every transfer into C.

Some RTLs allow a mixture of procedural and nonprocedural interpretations. Thus, an RTL that is mostly procedural may have special constructs to denote concurrent

operations or may accept delay specifications. An RTL that is mostly nonprocedural may allow for a special type of variable with the property that new values of these variables become immediately available for subsequent statements.

2.3.3 Internal RTL Models

An RTL model can be internally represented by data structures or by compiled code. A model based on data structures can be interpreted for different applications by model-independent programs. A compiled-code model is usually applicable only for simulation.

Compiled-code models are usually generated from procedural RTLs. The code generation is a 2-step process. First, the RTL description is translated to a high-level language, such as C or Pascal. Then the produced high-level code is compiled into executable machine code. This process is especially easy for an RTL based on a conventional programming language.

Internal RTL models based on data structures are examined in [Hemming and Szygenda 1975].

2.4 Structural Models

2.4.1 External Representation

A typical structural model of a system in a connectivity language specifies the I/O lines of the system, its components, and the I/O signals of each component. For example, the following model conveys the same information as the schematic diagram in Figure 2.15 (using the language described in [Chang *et al.* 1974]).

```
CIRCUIT XOR
INPUTS = A,B
OUTPUTS = Z

NOT   D, A
NOT   C, B
AND   E, (A,C)
AND   F, (B,D)
 OR   Z, (E,F)
```

The definition of a gate includes its type and the specification of its I/O terminals in the form *output, input_list*. The interconnections are implicitly described by the signal names attached to these terminals. For example, the primary input *A* is connected to the inverter *D* and to the AND gate *E*. A signal line is also referred to as a *net*.

Gates are components characterized by single output and functionally equivalent inputs. In general, components may have multiple outputs, both the inputs and the outputs may be functionally different, and the component itself may have a distinct name. In such a case, the I/O terminals of a component are distinguished by predefined names. For example, to use a *JK* F/F in the circuit shown in Figure 2.16, the model would include a statement such as:

U1 JKFF Q=ERR, QB=NOTERR, J=ERRDET, K=A, C=CLOCK1

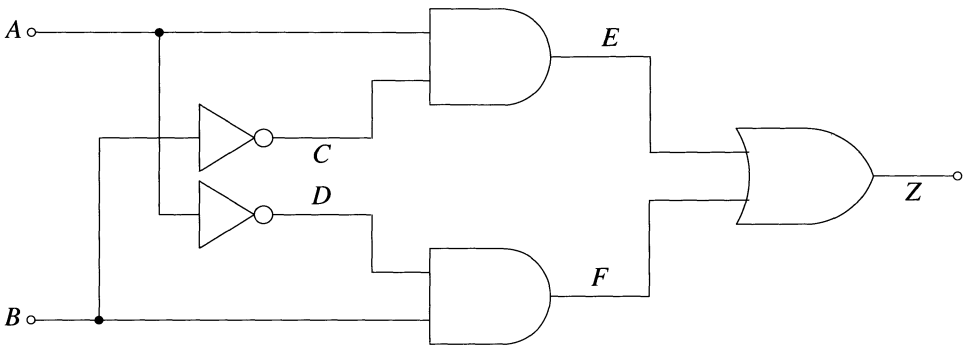


Figure 2.15 Schematic diagram of an XOR circuit

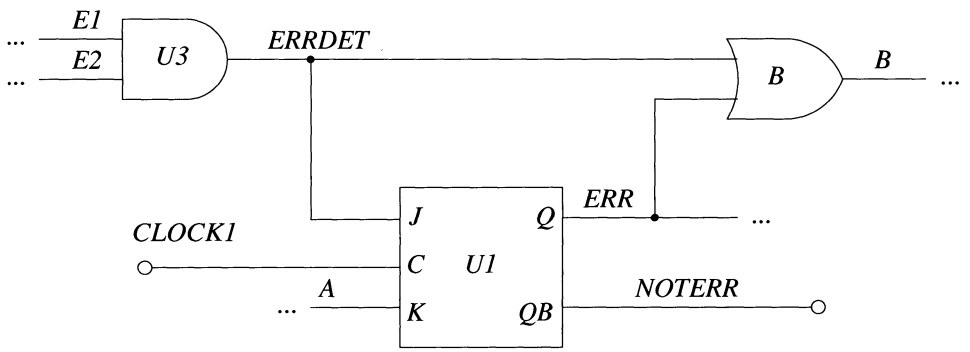


Figure 2.16

which defines the name of the component, its type, and the name of the nets connected to its terminals.

The compiler of an external model consisting of such statements must understand references to types (such as *JKFF*) and to terminal names (such as *Q*, *J*, *K*). For this, the model can refer to primitive components of the modeling system (*system primitive components*) or components previously defined in a *library of components* (also called *part library* or *catalog*).

The use of a library of components implies a bottom-up hierarchical approach to describing the external model. First the components are modeled and the library is built. The model of a component may be structural or an RTL model. Once defined, the library components can be regarded as *user primitive components*, since they become the building blocks of the system. The library components are said to be

generic, since they define new types. The components used in a circuit are *instances* of generic components.

Usually, structural models are built using a *macro approach* based on the macro processing capability of the connectivity language. This approach is similar to those used in programming languages. Basically, a macro capability is a text-processing technique, by means of which a certain text, called the macro body, is assigned a name. A reference to the macro name results in its replacement by the macro body; this process is called *expansion*. Certain items in the macro body, defined as macro parameters, may differ from one expansion to another, being replaced by actual parameters specified at the expansion time. When used for modeling, the macro name is the name of the generic component, the macro parameters are the names of its terminals, the actual parameters are the names of the signals connected to the terminals, and the macro body is the text describing its structure. The nesting of macros allows for a powerful hierarchical modeling capability for the external model.

Timing information in a structural model is usually defined by specifying the delays associated with every instance of a primitive component, as exemplified by

AND *X*, (*A*,*B*) **delay** 10

When most of the components of the same type have the same delay, this delay is associated with the type by a declaration such as

delay AND 10

and this will be the default delay value of any AND that does not have an explicit delay specification.

Chapter 3 will provide more details about timing models.

2.4.2 Structural Properties

A structural model of a system can be represented by a graph. This is useful since many concepts and algorithms of the graph theory can be applied. Initially we will consider only circuits in which the information is processed by components and is unidirectionally transmitted along signal lines. In cases when every net connects only two components, a natural representation of a structural model is a *directed graph* in which the components and nets are mapped, respectively, into nodes and edges of the graph (Figure 2.17). The primary inputs and outputs are mapped into nodes of special types. In general, however, a net may propagate a signal from one source to more than one destination. Such a signal is said to have **fanout**. A circuit in which no signal has fanout is said to be **fanout-free**. The graph model of a fanout-free combinational circuit is always a tree. A general way to represent a circuit with fanout is by a *bipartite directed graph*, in which both the components and the nets are mapped into nodes such that any edge connects a node representing a component to a node representing a net, as illustrated in Figure 2.18. (The nodes marked with *X* correspond to nets.) This model also allows multioutput components.

Reconvergent fanout refers to different paths from the same signal reconverging at the same component. This is the case of the two paths from *ERRDET* to *B* in Figure 2.18.

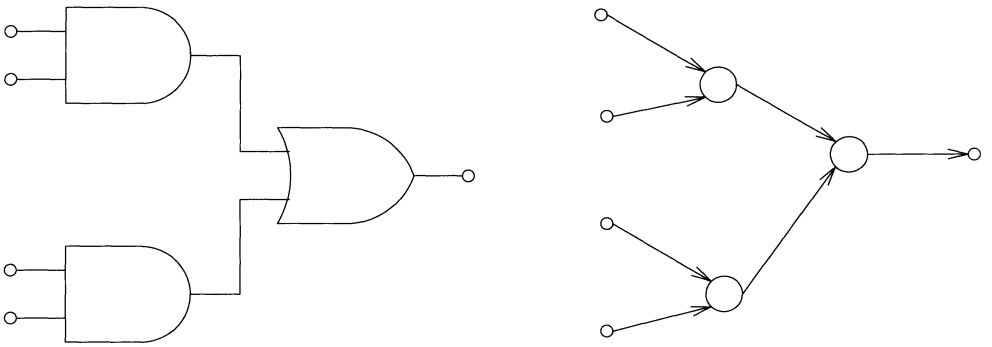


Figure 2.17 Fanout-free circuit and its graph (tree) representation

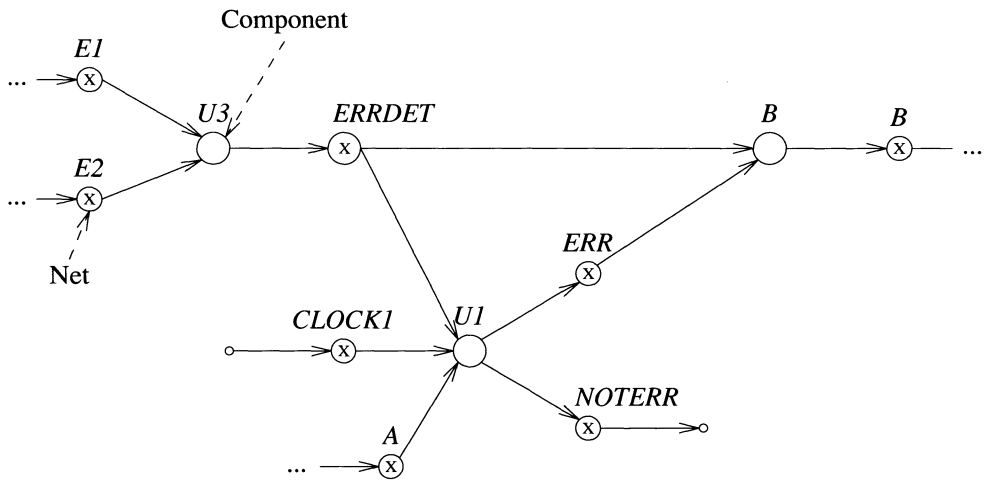


Figure 2.18 Bipartite graph model for the circuit of Figure 2.16

The next chapters will show that the presence of reconvergent fanout in a circuit greatly complicates the problems of test generation and diagnosis.

In circuits composed only of AND, OR, NAND, NOR, and NOT gates, we can define the **inversion parity** of a path as being the number, taken modulo 2, of the inverting gates (NAND, NOR, and NOT) along that path.

In circuits in which every component has only one output it is redundant to use separate nodes to represent nets that connect only two components. In such cases, the graph model used introduces additional nodes, called *fanout nodes*, only to represent

nets with fanout. This is illustrated in Figure 2.19. A net connecting one component (the source) with k other components (destinations or loads) is represented by one fanout node with one edge (the *stem*) from the node corresponding to the source and with k edges (*fanout branches*) going to the nodes corresponding to the destinations.

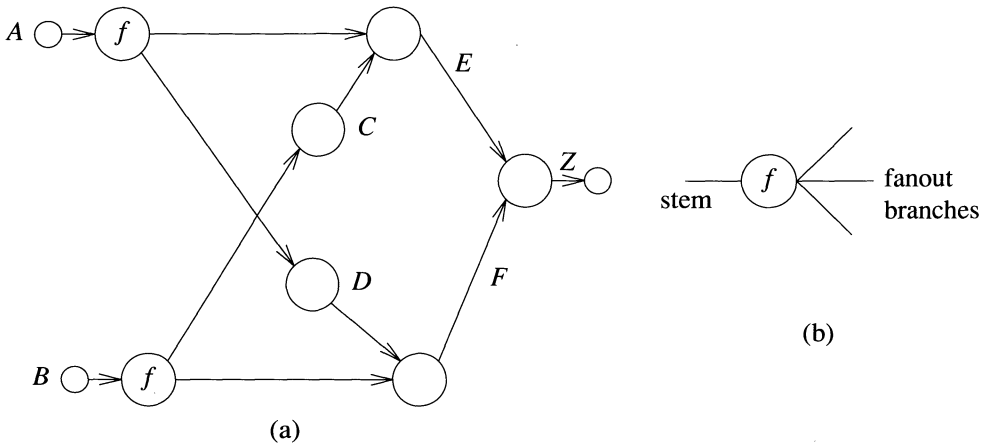


Figure 2.19 (a) Graph model for the circuit of Figure 2.15 (b) A fanout node

The (logic) **level** of an element in a combinational circuit is a measure of its "distance" from the primary inputs. The level of the primary inputs is defined to be 0. The level of an element i, $l(i)$, whose inputs come from elements $k_1, k_2, ..., k_p$, is given by

$$l(i) = 1 + \max_j \; l(k_j) \tag{2.1}$$

The following procedure computes levels via a breadth-first traversal of the circuit:

1. For every primary input i set $l(i) = 0$.
2. For every element i not yet assigned a level, such that all the elements feeding i have been assigned a level, compute $l(i)$ by equation (2.1).

The concept of level is extended to sequential circuits by considering the feedback lines to be pseudo primary inputs and hence at level 0. For example, for the circuit of Figure 2.20, the following levels are determined:

level 1: E, D, J
level 2: F, K
level 3: G
level 4: W

Instead of explicit levels, we can use an implicit leveling of the circuit by ordering the elements such that for any two elements numbered i_1 and i_2, $i_1 < i_2$ if $l(i_1) \leq l(i_2)$. These numbers can then be used as internal names of the elements (see next section).

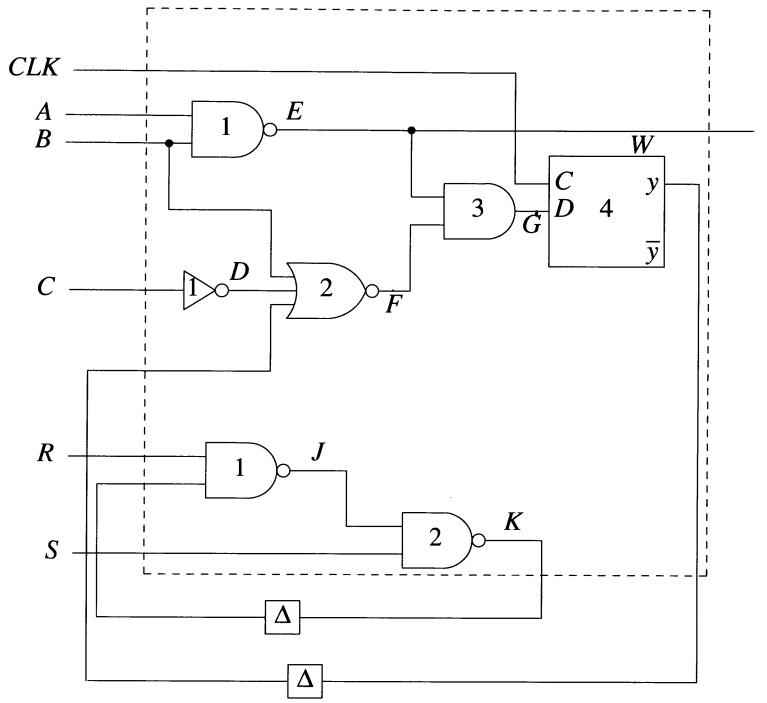


Figure 2.20 Logic levels in a circuit

2.4.3 Internal Representation

Figure 2.21 illustrates a portion of typical data structures used to represent a structural model of the circuit shown in Figure 2.16. The data structure consists of two sets of "parallel" tables, the ELEMENT TABLE and the SIGNAL TABLE, and also of a FANIN TABLE and a FANOUT TABLE. An element is internally identified by its position (index) in the ELEMENT TABLE, such that all the information associated with the element i can be retrieved by accessing the i-th entry in different columns of that table. Similarly, a signal is identified by its index in the SIGNAL TABLE. For the element i, the ELEMENT TABLE contains the following data:

1. NAME(i) is the external name of i.
2. TYPE(i) defines the type of i. (PI denotes primary input; PO means primary output.)
3. NOUT(i) is the number of output signals of i.
4. OUT(i) is the position (index) in the SIGNAL TABLE of the first output signal of i. The remaining NOUT(i)–1 output signals are found in the next NOUT(i)–1 positions.

ELEMENT TABLE

	NAME	TYPE	NOUT	OUT	NFI	FIN
1	*U1*	JKFF	2	1	3	1
2	*B*	OR	1	8	2	4
3	*U3*	AND	1	3	2	6
4	*CLOCK1*	PI	1	4	0	—
5	*NOTERR*	PO	0	—	1	8

SIGNAL TABLE

	NAME	SOURCE	NFO	FOUT
1	*ERR*	1	2	1
2	*NOTERR*	1	1	3
3	*ERRDET*	3	2	4
4	*CLOCK1*	4	1	6
5	*E1*	...	1	7
6	*E2*	...	1	8
7	*A*	...	1	9
8	*B*	2	...	...

FANIN TABLE

	INPUTS
1	3
2	4
3	7
4	1
5	3
6	5
7	6
8	2

FANOUT TABLE

	FANOUTS
1	2
2	...
3	5
4	1
5	2
6	1
7	3
8	3
9	1

Figure 2.21 Typical data structures for the circuit of Figure 2.16

5. NFI(i) is the fanin count (the number of inputs) of i.
6. FIN(i) is the position in the FANIN TABLE of the first input signal of i. The remaining NFI(i)−1 input signals are found in the next NFI(i)−1 positions.

For a signal j, the SIGNAL TABLE contains the following data:

1. NAME(j) is the external name of j.
2. SOURCE(j) is the index of the element where j originates.
3. NFO(j) is the fanout count of j (the number of elements fed by j).
4. FOUT(j) is the position in the FANOUT TABLE of the first element fed by j. The remaining NFO(j)−1 elements are found in the next NFO(j)−1 positions.

If every element has only one output, the data structure can be simplified by merging the ELEMENT and SIGNAL tables.

The described data structure is somewhat redundant, since the FANOUT TABLE can be constructed from the FANIN TABLE and vice versa, but it is versatile as it allows the graph representing the circuit to be easily traversed in both directions.

This data structure is extended for different applications by adding columns for items such as delays, physical design information, signal values, or internal states.

Although the macro approach allows hierarchical modeling for a structural external model, it does not carry this hierarchy to the internal model. This is because every reference to a defined macro (library component) results in its expansion, such that the final model interconnects only primitive elements.

An alternative approach, referred to as the *subsystem approach*, allows a hierarchical internal structural model. We will consider only two levels of hierarchy. The components of the top level are called subsystems, and the internal model describing their interconnections is similar to that described in Figure 2.21. The ELEMENT TABLE has an additional column pointing to a set of tables describing the internal structure of the subsystem as an interconnection of primitive elements.

Thus in the subsystem approach all the instances of the same type of a subsystem point to the same data structures representing the subsystem. This is helpful for systems containing many replicas of the same type of subsystem, such as bit-slice architectures. By contrast, the use of a macro approach would result in replicating the same structure as many times as necessary. Hence the subsystem approach can realize important memory savings for systems exhibiting repetitive structure.

On the negative side, the subsystem approach is more complicated to implement and many applications based on following connectivity incur a slight overhead by alternating between the two levels of hierarchy. This is why more than two levels appear to be impractical.

2.4.4 Wired Logic and Bidirectionality

The modeling techniques described so far rely on two basic assumptions:

- The information is processed by components and transmitted by signals.
- The information flow is unidirectional, that is, from input(s) to output(s) within a component and from source to destination(s) along a signal line.

These assumptions, however, are not generally valid. For example, in many technologies it is possible to directly connect several outputs such that the connecting net introduces a new logic function. This mechanism is referred to as *wired logic*, to denote that the logic function is generated by a wire rather than a component. An AND function realized in this way is called a wired-AND. Figure 2.22(a) shows a schematic representation of a wired-AND. A simple modeling technique for wired logic is to insert a "dummy" gate to perform the function of the wire, as illustrated in Figure 2.22(b). Although its output values are logically correct, the unidirectionality implied by the dummy gate may lead to incorrect values for its inputs. In the model in Figure 2.22(b), A and B may have different values, say A=0 and B=1. This, however, cannot occur in the real circuit (where A and B are tied together) and will cause further errors if B has additional fanout. A correct modeling technique should always force the value of C back onto A and B. One way to achieve this is to make A and B appear as special fanouts of C, as suggested by the dashed arrows in Figure 2.22(b). Then for every wired signal X we should maintain two values:

- the *driven value*, which is the value computed by the element driving X;
- the *forced value*, which is the value resulting from the wired logic.

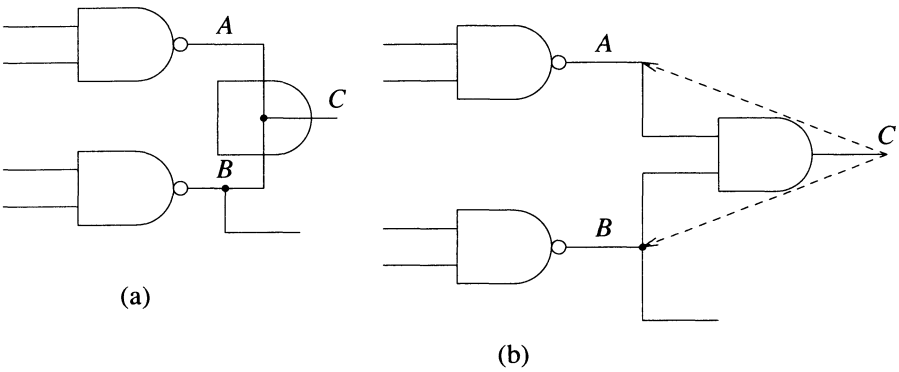


Figure 2.22 Wired-AND (a) schematic representation (b) modeling by a "dummy" gate

Note that the arguments for the wired logic function (which produces the forced value) are the driven values of the wired signals. Modeling of a tristate bus is similar and will be discussed in Chapter 3.

The insertion of dummy components for wired logic is a "work-around" introduced to maintain the basic modeling assumptions. The insertion of dummy components for modeling complicates the correspondence between the model and the real circuit and thus adversely affects fault modeling and diagnosis. A direct modeling technique that allows logic functions to be performed by bidirectional wires is therefore preferable. This is especially important in MOS circuits where wired logic is commonly employed. MOS circuits deviate even more from the basic modeling assumptions. The MOS transistor is a bidirectional component that acts like a voltage-controlled switch. The capacitance of a wire can be used to store information as charge (dynamic memory). The behavior of a MOS circuit may also be influenced by the relative resistances of transistors (ratioed logic). Although several work-around techniques [Wadsack 1978, Flake *et al.* 1980, Sherwood 1981, Levendel *et al.* 1981, McDermott 1982] have been devised to extend a conventional modeling framework to accommodate MOS technology, these methods offer only limited accuracy in their ability to model MOS circuits. Accurate solutions are based on *switch-level models* (see [Hayes 1987] for a review), which use transistors as primitive components.

2.5 Level of Modeling

The level of modeling refers to the types of primitive components used in a model. A model that (after expansion) contains only gates is called a *gate-level model.* Similarly, there is a *transistor-level model.* The lowest-level primitive components in a modeling system are those that cannot be structurally decomposed into simpler

components; usually at the lowest level we have either gates or transistors. Any other primitive component is said to be a *functional* (or a *high-level*) primitive component. Hence, in a modeling system that uses transistors as the lowest-level primitive components, a gate can be considered a functional primitive component.

Nonprimitive components whose function is described using RTL primitive operators are also characterized as functional. In the literature, a component modeled at the functional level is also called a *module*, or a *functional block.*

Let us review what alternatives one may have in modeling a system. First, one can have a specialized program to represent the system. This type of approach is used, for example, to provide a model of a processor to support its software development. In this way the entire system is viewed as a primitive (top-level) component. Second, one may develop an RTL model of the system. Third, one can structurally model the system as an interconnection of lower-level components. The same three alternatives exist for modeling every component (except those at the lowest level): primitive functional model, RTL model, or further structural decomposition.

Note that a primitive functional model of a component is usually provided by the developer of the modeling system, while an RTL model of a component is provided by the user. In general, the effort involved in developing primitive functional models for complex components is larger than that involved in developing the corresponding RTL models. In addition, an RTL model is more adequate for fault simulation and test generation than is a procedure invisible to the user. This is why primitive functional models are usually provided only for components of low to moderate complexity that can be used as building blocks of many other different components. The ability to define and interconnect user-specified functional blocks is very important since low-level models may not be available, either because they are not provided by the IC manufacturers or because they have not yet been designed.

Some simulation systems allow *physical models* to replace functional models of existing ICs [Widdoes and Stump 1988]. A physical model uses a hardware device as a generic model for all its instances in a circuit. The use of physical models eliminates the effort involved in developing software models for complex devices.

Figure 2.23 illustrates different levels of modeling used in describing a 2-bit counter. The highest level is an RTL model. The next lower level is a structural model composed of F/Fs and gates. The *D* F/F can be a functional primitive, or a functional block described by an RTL model, or it can be expanded to its gate-level model. Finally, a gate can be regarded as a primitive component, or it can be expanded to its transistor-level model.

Different levels of modeling are needed during the design process. Following a top-down approach, a design starts as a high-level model representing the specification and progresses through several stages of refinement toward its final implementation. The transition from specification (*what* is to be done) toward implementation (*how* it is done) corresponds to a transition from a higher to a lower level of modeling.

A high-level model provides a more abstract view of the system than a corresponding low-level model (gate or transistor-level), but it involves a loss of detail and accuracy, especially in the area of timing modeling (this subject will be further discussed in the next chapter). At the same time, the size and amount of detail of a low-level model

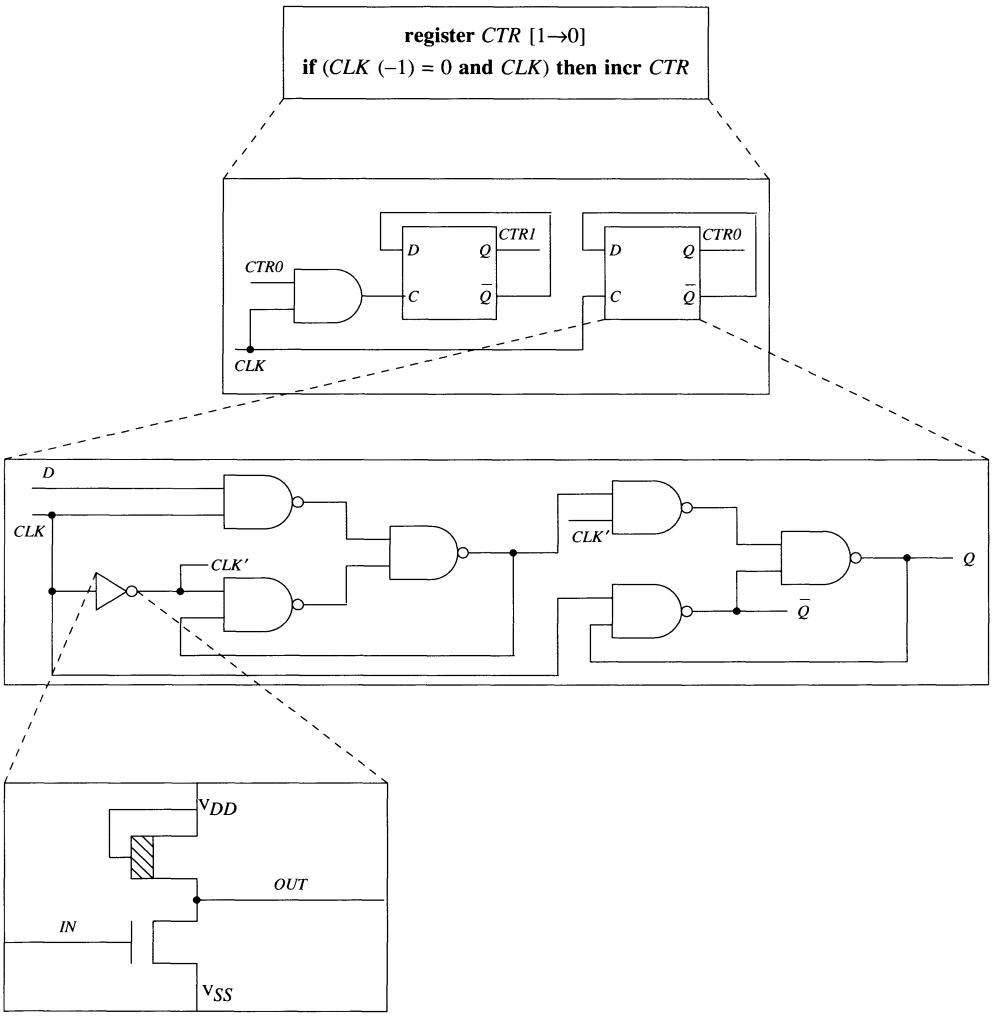


Figure 2.23 Levels of modeling for a 2-bit counter

require large computing resources, in terms of both memory capacity and processing time, for any application using that model. The need for large computing resources has become critical because of the increasing complexity of VLSI circuits.

We can observe that the level of modeling controls the trade-off between accuracy and complexity in a model. An efficient solution of this problem is offered by *hierarchical mixed-level modeling* different, which allows different levels of modeling to be mixed in the same description of a system. Mixed-level modeling is usually coupled with *multimodeling*, which consists of having more than one model for the same component and being able to switch from one model to another.

With mixed-level modeling, different parts of the system can be modeled at different levels. It is also possible to mix analog electrical models with digital ones. Thus, parts in different stages of development can be combined allowing members of design teams to proceed at different rates. Also, mixed-level modeling allows one to focus on low-level details only in the area of interest, while maintaining high-level models for the rest of the system for the sake of efficiency.

REFERENCES

[Akers 1978] S. B. Akers, "Binary Decision Diagrams," *IEEE Trans. on Computers*, Vol. C-27, No. 6, pp. 509-516, June, 1978.

[Barbacci 1975] M. R. Barbacci, "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," *IEEE Trans. on Computers*, Vol. C-24, No. 2, pp. 137-150, February, 1975.

[Breuer *et al.* 1988] M. A. Breuer, W. Cheng, R. Gupta, I. Hardonag, E. Horowitz, and S. Y. Lin, "Cbase 1.0: A CAD Database for VLSI Circuits Using Object Oriented Technology," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 392-395, November, 1988.

[Bryant 1986] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677-691, August, 1986.

[Chang *et el.* 1974] H. Y. Chang, G. W. Smith, and R. B. Walford, "LAMP: System Description," *Bell System Technical Journal*, Vol. 53, No. 8, pp. 1431-1449, October, 1974.

[Chappell *et al.* 1976] S. G. Chappell, P. R. Menon, J. F. Pellegrin, and A. M. Schowe, "Functional Simulation in the LAMP System," *Proc. 13th Design Automation Conf.*, pp. 42-47, June, 1976.

[Dietmeyer and Duley 1975] D. L. Dietmeyer and J. R. Duley, "Register Transfer Languages and Their Translation," in "*Digital System Design Automation: Languages, Simulation, and Data Base*" (M. A. Breuer, ed.), Computer Science Press, Woodland Hills, California, 1975.

[Flake *et al.* 1980] P. L. Flake, P. R. Moorby, and G. Musgrave, "Logic Simulation of Bi-directional Tri-state Gates," *Proc. 1980 IEEE Conf. on Circuits and Computers*, pp. 594-600, October, 1980.

[Frey 1984] E. J. Frey, "*ESIM*: A Functional Level Simulation Tool," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 48-50, November, 1984.

[Hayes 1987] J. P. Hayes, "An Introduction to Switch-Level Modeling," *IEEE Design & Test of Computers*, Vol. 4, No. 4, pp. 18-25, August, 1987.

[Hemming and Szygenda 1975] C. W. Hemming, Jr., and S. A. Szygenda, "Register Transfer Language Simulation," in "*Digital System Automation: Languages, Simulation, and Data Base*" (M. A. Breuer, ed.), Computer Science Press, Woodland Hills, California, 1975.

[Hill and vanCleemput 1979] D. Hill and V. vanCleemput, "SABLE: A Tool for Generating Structured, Multi-level Simulations," *Proc. 16th Design Automation Conf.*, pp. 272-279, June, 1979.

[Johnson *et al.* 1980] W. Johnson, J. Crowley, M. Steger, and E. Woosley, "Mixed-Level Simulation from a Hierarchical Language," *Journal of Digital Systems*, Vol. 4, No. 3, pp. 305-335, Fall, 1980.

[Lee 1959] C. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell System Technical Journal*, Vol. 38, No. 6, pp. 985-999, July, 1959.

[Levendel *et al.* 1981] Y. H. Levendel, P. R. Menon, and C. E. Miller, "Accurate Logic Simulation for TTL Totempole and MOS Gates and Tristate Devices," *Bell System Technical Journal*, Vol. 60, No. 7, pp. 1271-1287, September, 1981.

[McDermott 1982] R. M. McDermott, "Transmission Gate Modeling in an Existing Three-Value Simulator," *Proc. 19th Design Automation Conf.*, pp. 678-681, June, 1982.

[Shahdad *et al.* 1985] M. Shahdad, R. Lipsett, E. Marschner, K. Sheehan, H. Cohen, R. Waxman, and D. Ackley, "VHSIC Hardware Description Language," *Computer*, Vol. 18, No. 2, pp. 94-103, February, 1985.

[Sherwood 1981] W. Sherwood, "A MOS Modeling Technique for 4-State True-Value Hierarchical Logic Simulation," *Proc. 18th Design Automation Conf.*, pp. 775-785, June, 1981.

[Wadsack 1978] R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, Vol. 57, No. 5, pp. 1449-1474, May-June, 1978.

[Widdoes and Stump 1988] L. C. Widdoes and H. Stump, "Hardware Modeling," *VLSI Systems Design*, Vol. 9, No. 7, pp. 30-38, July, 1988.

[Wolf 1989] W. H. Wolf, "How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 3, pp. 288-301, March, 1989.

PROBLEMS

2.1 Determine whether the following cubes can be cubes of a function $Z(x_1,x_2,x_3,x_4)$.

0	x	1	x	|	0
x	1	0	1	|	0
1	x	1	1	|	0
0	1	0	x	|	1
x	0	1	0	|	1

2.2 Construct a binary decision diagram for the exclusive-OR function of two variables.

2.3 Derive a binary decision diagram for the function given in Figure 2.2(a).

2.4 Write two RTL models for a positive edge-triggered *D* flip-flop (the output takes the value of *D* after a 0 to 1 transition of the clock). First assume an RTL that does not allow accessing past values of signals. Remove this restriction for the second model.