CHAPTER 6

SYNTHESIS OF VHDL CODE

Synthesizing VHDL code is the process of realizing the VHDL description using the primitive logic cells from the target device's library. In Chapters 4 and 5, we discussed how to derive a conceptual diagram from VHDL statements. The conceptual diagram can be considered as the first step in realizing the code. The diagram is refined further during synthesis. The synthesis process involves complex algorithms and a large amount of data, and computers are needed to facilitate the process. Although today's synthesis software appears to be sophisticated and capable, there are fundamental limitations. Understanding the capability and limitation of synthesis software will help us better utilize this tool and derive more efficient designs. This chapter explains the realization of VHDL operators and data types, provides an in-depth overview on the synthesis process, and discusses the timing issue involved in synthesis.

6.1 FUNDAMENTAL LIMITATIONS OF EDA SOFTWARE

Developing a large digital circuit is a complicated process and involves many difficult tasks. We have to deal with complex algorithms and procedures and handle a large amount of data. Computers are used to facilitate the process. As computers become more powerful, we may ask if it is possible to develop a suite of software and completely automate the synthesis process. The ideal scenario is that human designers would only need to develop a high-level behavioral description and EDA software would perform the synthesis and placement and routing and automatically derive the optimal circuit implementation. The is unfortunately not possible. The limitation comes from the theoretical study of computational algorithms.

Although this book does not cover EDA algorithms, it will be helpful to know the capability and limitation of EDA software tools so that they can be used effectively.

For the purposes of discussion, we can separate an EDA software tool into a core and a shell. The *core* is the algorithms that perform the transformation or optimization, and the *shell* wraps the algorithm, including data conversion, memory and file management and user interface. Although the shell is important, the core algorithms ultimately determine the quality and efficiency of the software tool. The problems encountered in EDA are not unique. In fact, they are formulated and transformed into optimization problems in other fields, especially in the study of graph theory. This section provides a layperson's overview of computability and computation complexity, which helps us understand the fundamental limitation of EDA software.

6.1.1 Computability

Computability concerns whether a problem can be solved by a computer algorithm. If an algorithm exists, the problem is *computable* (or *decidable*). Otherwise, the problem is *uncomputable* (or *undecidable*). An example of an uncomputable problem is the "halting problem." Some programs, such as a compiler, take another program as input and check certain properties (e.g., syntax) of that program. The halting problem asks whether we can develop a program that takes any program and its input and determines whether computation of that program will eventually halt (e.g., no infinite loop). It can be proven mathematically that no such program can be developed, and thus the halting problem is uncomputable. Informally speaking, any attempt to examine the "meaning" of a program is uncomputable.

Equivalence checking discussed in Section 1.5.3 essentially compares whether two programs perform the same function, which goes further than the halting problem. Therefore, equivalence checking is uncomputable; i.e., it is not possible to develop an EDA tool that determines the equivalence of *any* two descriptions. However, it is possible to use some clever techniques to determine the equivalence of some descriptions, which are coded following certain guidelines. Thus, while equivalence checking cannot guarantee to work all of the time, it can be useful some of the time.

6.1.2 Computation complexity

If a problem is computable, an algorithm can be derived to solve the problem. The *computation complexity* concerns the efficiency of an algorithm. The computation complexity can be further divided into *time complexity*, which is a measure of the time needed to complete the computation, and *space complexity*, which is a measure of hardware resources, such as memory, needed to complete the computation. Since most statements on time complexity can be applied to space complexity as well, in the remaining section we focus on time complexity.

Big-O notation The computation time of an algorithm depends on the size of the input as well as on the type of processor, programming language, compiler and even personal coding style. It is difficult to determine the exact time needed to complete execution of an algorithm. To characterize an algorithm, we normally focus on the impact of input size and try to filter out the effect of the "interferences" on measurement. Instead of determining the exact function for computation time, we usually consider only the *order* of this function.

The order is defined as follows. Given two functions, $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ (pronounced as $f(n)$ is big-O of $g(n)$ or $f(n)$ is of order $g(n)$) if two constants,

Table 6.1 Scaling of some commonly used big-O functions

Input size	Big-O function					
n	n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	2 μs	1 μs	2 μs	4 μs	8 μs	4 μs
4	4 μs	2 μs	8 μs	16 μs	64 μs	16 μs
8	8 μs	3 μs	24 μs	64 μs	512 μs	256 μs
16	16 μs	4 μs	64 μs	256 μs	4 ms	66 ms
32	32 μs	5 μs	160 μs	1 ms	33 ms	71 min
48	48 μs	5.5 μs	268 μs	2 ms	111 ms	9 years
64	64 μs	6 μs	384 μs	4 ms	262 ms	600,000 years

n_0 and c, can be found to satisfy

$$f(n) < cg(n) \text{ for any } n, n > n_0$$

The $g(n)$ function is normally a simple function, such as n, $n \log_2 n$, n^2, n^3 or 2^n. For example, all the following functions are $O(n^2)$:

- $0.1n^2$
- $n^2 + 5n + 9$
- $500n^2 + 1000000$

The purpose of big-O notation is twofold. First, it drops the less important, secondary terms since the highest-order term becomes the dominant factor as n becomes large. Second, it concentrates on the rate of change and ignores the constant coefficient in a function. After removing the constant coefficients and lower-order terms, we eliminate the effect of coding style, instruction set and hardware speed, and can concentrate on the effectiveness of an algorithm. Big-O notation is essentially a scaling factor or growth rate, indicating the resources needed as input size increases.

Commonly encountered orders are $O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \log_2 n)$, $O(n^2)$, $O(n^3)$ and $O(2^n)$. $O(n)$ indicates the linear growth rate, in which the required computation resources increase in proportion to the input size. $O(1)$ means that the required computation resources are constant and do not depend on input size. $O(\log_2 n)$ indicates the logarithmic growth rate, which changes rather slowly. For a problem with $O(1)$ or $O(\log_2 n)$, the input size has very little impact on the resources. $O(n^2)$ and $O(n^3)$ have faster growth rates and the required computation resources become more significant as the input size increases. All of the orders discussed so far are considered as being of *polynomial order* since they have the form of $O(n^k)$, where k is a constant. On the other hand, $O(2^n)$ indicates the exponential growth rate and the computation time increases geometrically. Note that an increment of 1 in input size doubles the computation time. $O(2^n)$ grows faster than does any polynomial order.

An example using these functions is shown in Table 6.1, which lists the required computation times of algorithms of varying computation complexity. For comparison, we assume that it takes 2 μs for an $O(n)$ algorithm to perform a computation of input size 2. The table shows the required times as the input size increases from 2 to 64 under different big-O functions.

One example of $O(2^n)$ complexity is the exhaustive testing of a combinational circuit. One way to test a combinational circuit is to apply all possible input combinations exhaustively and examine their output responses. For a circuit with n inputs, there are

2^n possible input combinations. If we assume that the testing equipment can check 1 million patterns per second, exhaustively testing a 64-bit circuit takes about 600,000 years (i.e., $\frac{2^{64}}{10^6*60*60*24*365}$) to complete. Thus, although simple and straightforward, this method is not practical in reality.

Intractable and tractable problems In most problems, if a polynomial order ($O(n^k)$) algorithm can be found, the exponent k is normally very small (say, 1, 2, or 3). Even though the growth rate is much worse than the linear rate, we can tolerate applying the algorithm to problems with nontrivial input sizes. We call these problems *tractable*. On the other hand, computation theory has shown that a polynomial-order solution cannot be found or is "unlikely" to be found for some problems. The only existing solutions are the algorithms with nonpolynomial order, such as $O(2^n)$. We call these problems *intractable*. As we have seen in Table 6.1, the computation time for the $O(2^n)$ algorithm simply grows too fast and the algorithm is not practical even for a moderate-sized n. Improvement in hardware speed will not change the situation significantly.

The situation is not completely hopeless for an intractable problem. An intractable problem usually means that it takes $O(2^n)$ computation time to find the *optimal* answer for *any* given input. It is frequently possible to find a polynomial-order algorithm, based on some smart tactics and *heuristics* (an educated guess), that permits us to obtain a valid, *suboptimal* answer or the optimal solution for *some* input patterns.

Synthesis as an intractable problem The focus of this book is on describing a design in textual HDL code and then using synthesis software to realize the circuit. From the computation complexity point of view, the synthesis consists of several intractable problems, and thus no polynomial-time algorithm exists. We can treat the synthesis process as a searching procedure. For a given specification, there are possibly $O(2^n)$ valid circuit configurations. Finding the optimal configuration corresponds to a global search, exhaustively checking and comparing all $O(2^n)$ possible configurations. Real synthesis software must limit the search space. It normally performs the search on a local basis and applies some smart tactics and heuristics to guide the direction of the search. The starting point of the search corresponds to the configuration described in our HDL code. Since the search is local, the initial starting point plays a key role. A good initial description will put the starting point in a good location, and an efficient configuration can be obtained accordingly. On the other hand, if the initial description is poor, the good configurations will be far away. Since synthesis software doesn't perform a global search, it is unlikely that software can obtain an efficient configuration.

6.1.3 Limitations of EDA software

Like synthesis, other design tasks contain intractable or even undecidable computation problems. This is the inherent, theoretical limitation of EDA software and cannot be overcome by fast hardware, smart software code or human talents. Heuristics and tricks of software algorithms can sometimes find good solutions for certain types of inputs. There is no guarantee that the solutions are optimal or that the algorithm will work for all types of inputs. Therefore, it is impossible to use EDA software to completely automate the design process. This limitation is real and here to stay. The quality and efficiency of a design still rely on a human designer's experience, insight, ingenuity and imagination, which, to some degree, can be considered as the ultimate heuristics that cannot be coded into software.

6.2 REALIZATION OF VHDL OPERATORS

When we develop VHDL code for synthesis, language constructs in the code are eventually mapped to hardware. In the previous chapters, we illustrated the realization (i.e., the conceptual diagram) of basic concurrent and sequential statements. VHDL operators are used as building components in these diagrams. In a conventional programming language, we don't pay too much attention to the operators since most operations, including integer arithmetic operations, logical operations and shift operations, take the same amount of resources: one instruction cycle of the CPU. This is totally different in synthesis. Hardware complexities and operation speed of VHDL operators vary significantly and are processed differently during synthesis. To derive an efficient design, we have to be aware of the implications of VHDL operators on hardware implementation.

Only a subset of VHDL operators can be synthesized automatically. The subset normally includes the logical operators, relational operators as well as addition and subtraction operators. Some software may also include more complicated operators, such as shift or multiplication operators. Software can rarely automatically synthesize division (`/`), **mod**, **rem** and exponential (`**`) operators or any operators associated with floating-point data-type operands. The following subsections provide an overview of the realization of VHDL operators.

6.2.1 Realization of logical operators

Logical operators can be mapped directly to logic gates, and their synthesis is straightforward. The **and**, **nand**, **or** and **nor** operators have similar area and delay characteristics. The **xor** and **xnor** operators are slightly more involved and their implementation requires more silicon area and experiences a larger propagation delay.

In VHDL, a logical operation can be applied over operands with multiple bits. For example, let `a` and `b` be 8-bit signals with a data type of `std_logic_vector(7 downto 0)`. The expression `a xor b` means that the **xor** operation is applied to eight individual bits in parallel. Since each bit of the input operates independently, the area of the circuit grows linearly with the number of input bits (i.e., on the order of $O(n)$), and the propagation delay is a constant (i.e., on the order of $O(1)$).

6.2.2 Realization of relational operators

There are six relational operators in VHDL: `=`, `/=`, `<`, `<=`, `>` and `=>`. According to their hardware implementation, these operators can be divided into the equality group, which includes the `=` and `/=` operators, and the greater–less group, which includes the other four operators.

In the equality group, operators can easily be implemented by a tree-like structure. For this implementation, the circuit area grows linearly with the number of input bits (i.e., $O(n)$), and the delay grows at a relatively slow $O(\log_2 n)$ rate. In the greater–less group, the operation exhibits a strong data dependency of input bits. For example, to determine the "greater than" relationship, we first have to compare the most significant bits of two operands and, if they are equal, the next lower bits and so on. This leads to larger area and propagation delay. Because of the circuit complexity, these operators can be implemented in a variety of ways, each with a different area–delay characteristic. In the minimal-area implementation, both area and delay grow linearly (i.e., $O(n)$) with the number of input

bits. There are several different ways to improve the performance (i.e., reduce the delay), all at the expense of extra hardware.

6.2.3 Realization of addition operators

The addition operator (+) is the most basic arithmetic operator. Several other operators, including subtraction (-), negation (- with one operand) and absolute value (**abs**), can easily be derived from the addition operator.

The addition operation has an even stronger data dependency of individual bits since the least significant bit of input may affect the most significant bit of the result. It is normally the most complex operator that can be synthesized automatically. Since the adder is the basis of other arithmetic operations, its implementation has been studied extensively and a wide range of circuits that exhibit different area–delay characteristics has been developed. The minimal-area circuit, sometimes known as a *serial* or *ripple* adder, can easily be implemented by cascading a series of 1-bit full adders. In this implementation, both area and delay grow linearly (i.e., $O(n)$).

6.2.4 Synthesis support for other operators

Synthesis support for other more complicated operators is sporadic. It depends on individual synthesis software, the width of the input operands as well as the targeted device technology. Some high-end synthesis software can automatically derive multiplication operator (*) and shift operators (**sll**, **srl**, **sla**, **sra**, **rol** and **ror** of VHDL, and `shift_left`, `shift_right`, `rotate_left` and `rotate_right` of the IEEE `numeric_std` library). Because of the hardware complexity, we must be extremely careful if these operators are used in a VHDL code. Synthesis software rarely supports division-related operators (/, **mod** and **rem**) or the exponential operator (**) or any operators associated with floating-point data-type operands.

Since the emphasis in this book is on portable description, we will not use these operators in our VHDL codes. Examples in Chapters 8 and 15 show how to design and derive VHDL code for some of these operators.

6.2.5 Realization of an operator with constant operands

The operands of VHDL operators can sometimes be a constant expression, which does not depend on the value of any input signal. Such constant operands have a significant implication in the synthesis process.

Operator with all constant operands If all the operands of an expression are constants, we can evaluate the expression in advance and replace it with a constant value. However, it is good practice to use constant symbols and constant expressions in VHDL code. They make the code more descriptive. For example, consider the following code segment:

```
constant OFFSET: integer := 8;
signal boundary: unsigned(8 downto 0);
signal overflow: std_logic;
. . .
overflow <= '1' when boundary > (2**OFFSET-1) else
            '0';
```

The operands of operators ** and - are constants, and the 2**OFFSET-1 expression can be replaced by a constant, 255. Although we can use 255 in VHDL code, it is less clear about how the value is obtained. In a large, complex VHDL program that involves many constant values, keeping track of the meaning of all constants becomes difficult. It is advisable to use constant symbols and constant expressions.

During synthesis, software can easily detect constant expressions and replace them with constants during preprocessing (in the elaboration phase of VHDL code). Since no physical hardware will be inferred from constant expressions, we can use them freely in VHDL code.

Operator with partial constant operands Most VHDL operators have two operands. Sometimes one of the operands is a constant, as in count+1. Instead of using a full-fledged operator implementation, synthesis software can "propagate" and "embed" the constant value into the circuit implementation. From a synthesis point of view, a constant operand actually decreases the number of inputs of the circuit by half and thus can significantly reduce the circuit complexity. For example, if a and b are two 8-bit signals and op is a VHDL operator, implementing the a op b expression requires a combination circuit with 16 inputs. On the other hand, if one operand is a constant, say "0001001", implementing the a op "00010001" expression only requires a combination circuit with eight inputs.

The following three examples further depict the difference between a full-fledged circuit and the simplified implementation. The first example is of a rotation operator. Assume that x and y are 8-bit signals and consider the following rotation operation:

```
y <= rotate_right(x, 3);
```

Since the shifting amount is a constant of 3, no actual shifting circuit is needed. This operation can be implemented by properly connecting the input signals to the output signals, which requires no logic at all. It is the same as

```
y <= x(2 downto 0) & x(7 downto 3);
```

The second example is of an equality operator. Let us consider a 4-bit equality comparator with inputs of $x_3x_2x_1x_0$ and $y_3y_2y_1y_0$. The logic expression of this operation is

$$(x_3 \oplus y_3)' \cdot (x_2 \oplus y_2)' \cdot (x_1 \oplus y_1)' \cdot (x_0 \oplus y_0)'$$

If one operand is a constant, say, $y_3y_2y_1y_0 = 0000$, the expression can be simplified to

$$x_3' \cdot x_2' \cdot x_1' \cdot x_0'$$

The comparator is reduced to a 4-input nor gate. Thus, there is a significant difference between a full-fledged comparator and a reduced comparator.

The last example is of an addition operator. A frequently used operation in VHDL is incrementing: adding 1 to a signal, as in count+1. A minimal-area implementation of the addition operator is done by cascading 1-bit full adders. On the other hand, a minimal-area incrementor can be implemented by half adders, whose size is about one half that of full adders. Thus, the circuit area of an incrementor is only about one half that of a regular addition operator.

6.2.6 An example implementation

It will be helpful to have a comprehensive table that lists the areas and delays of synthesizable operators. However, because of the complexity of the synthesis process and device

Table 6.2 Circuit area and delay of some commonly used VHDL operators

Width	VHDL operator									
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	mux
	Area (gate count)									
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
	Delay (ns)									
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

technology, a small variation in VHDL code, synthesis algorithm, or device parameters will lead to different results. Table 6.2 shows one synthesis result for several representative operators of different input widths in a 0.55-micron CMOS standard-cell technology. The subscripts a and d indicates that the circuit is optimized for area and for delay respectively.

The unit of area is a gate count, which is the equivalent number of 2-input nand gates used to implement the circuit, and the unit of propagation delay is the nanosecond (ns). We need to be cautious about the data in the table. The data is valid only for a particular version of a particular software on a particular device technology and should not be overly interpreted or analyzed. However, this data does show a general trend and provide a rough idea about the relative complexity of different operators. The information for a 2-to-1 multiplexer, which is the basic component for routing, is also included in the table for reference.

There are several important observations to be made from the table. First, as we expect, the area and propagation delay vary significantly among the different operators. For example, the area of a 32-bit fast addition operator is more than 10 times larger than that of a 32-bit nand operator, and the propagation delay of the adder is more than 100 times longer than that of the nand operator.

The second observation is about the trade-off between area and delay. In digital system design, it is generally not possible to find an optimal implementation, which has both minimal area and minimal delay. We normally have to invest more resources (a larger area) for better performance (less delay). Except for the trivial implementation of logical operators, other operators have multiple implementations with different area–delay characteristics. Table 6.2 shows the area and delay characteristics of two implementations, in which one is optimized for a smaller area and the other is optimized for less delay.

The third observation is about scaling, the impact of increasing the size of the input of an operator (e.g., from 8 bits to 16 bits to 32 bits). The growth rates of area and delay are not always linear (i.e., $O(n)$). In general, the growth rate of delay is on the order of $O(1)$, $O(\log_2 n)$ or $O(n)$, while the growth rate of area is between the orders $O(n)$ and $O(n^2)$. Since the commercial synthesis software normally does not reveal its internal algorithms, the growth rate observation is true only for this particular software and device. Chapter 15 provides an in-depth discussion of the design of some operators.

oe	y
0	Z
1	a_in

Figure 6.1 Tri-state buffer.

6.3 REALIZATION OF VHDL DATA TYPES

6.3.1 Use of the `std_logic` data type

VHDL supports a rich set of data types. During synthesis, these data types must be mapped into binary representations so that they can be realized in a physical circuit. The VHDL standard itself does not define the mapping mechanism, and thus the mapping is left for synthesis software. To have better control of the final implementation, we limit our use of data types primarily to the `std_logic` data type and its derivatives, the `std_logic_vector`, `signed` and `unsigned` data types. The only exception is the user-defined enumeration data type, which is used for the description of a finite state machine and is discussed in Chapter 9.

Recall that there are nine possible values in the `std_logic` data type. Among them, `'0'` and `'1'` are interpreted as logic 0 and logic 1 and are used in regular synthesis. `'L'` and `'H'` are interpreted as weak 0 and weak 1, as in wired logic. Since modern device technologies no longer use this kind of circuitry, the two values should not be used. `'U'`, `'X'` and `'W'` are meaningful only in modeling and simulation, and they cannot be synthesized. The two remaining values, `'Z'` and `'-'`, which represent high impedance and "don't-care" respectively, have some impact on synthesis. Their use is discussed in the following subsections.

6.3.2 Use and realization of the `'Z'` value

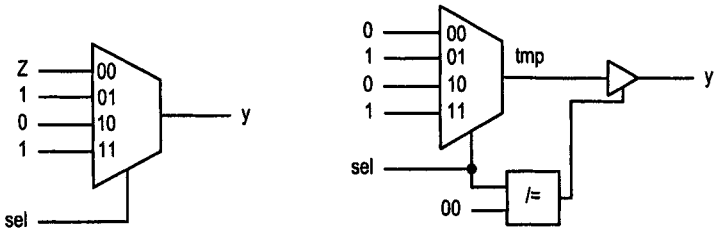
The `'Z'` value means high impedance or an open circuit. It is not a value in Boolean algebra but a special electrical property exhibited in a physical circuit. Only a special kind of component, known as a *tri-state buffer*, can have an output of this value. The symbol and function table of a tri-state buffer are shown in Figure 6.1. When the `oe` (for "output enable") signal is `'1'`, the buffer acts as a short circuit and the input is passed to output. On the other hand, when the `oe` signal is `'0'`, the y output appears to be an open circuit.

VHDL description of a tri-state buffer High impedance cannot be handled by regular logic and can exist only in the output of a tri-state buffer. The VHDL description of the tri-state buffer of Figure 6.1 is

```
y <= a_in when oe='1' else
     'Z';
```

We cannot use a value of `'Z'` as an input or manipulate it as a logic value. For example, the following statements cannot be realized and are meaningless in synthesis:

```
f <= 'Z' and a;
y <= data_a when in_bus='Z' else
     data_b;
```



(a) Confusing diagram (b) Better diagram

Figure 6.2 Use of 'Z' as an output value.

Since a tri-state buffer is not an ordinary logic value, it is a good idea to code it in a separate statement. For example, consider the following VHDL description:

```
with sel select
   y <= 'Z' when "00",
        '1' when "01"|"11",
        '0' when others;
```

Although the code is correct, direct transformation to a conceptual diagram, as shown in Figure 6.2(a), cannot be synthesized. To clarify the intended structure, the code should be modified as

```
with sel select
   tmp <= '1' when "01"|"11",
          '0' when others;
y <= tmp when sel/="00" else
     'Z';
```

Following the description, we can easily derive the intended block diagram, as shown in Figure 6.2(b).

The major application of a tri-state buffer is to implement a bidirectional I/O port to save the pin count and to form a bus.

VHDL description of a bidirectional I/O port As a silicon device packs more circuitry into a chip, the number of I/O signals increases accordingly. A bidirectional I/O pin can be used as either an input or an output and thus makes more efficient use of an I/O pin. Most FPGA and memory devices utilize bidirectional I/O pins.

The schematic of a simple circuit with bidirectional I/O port, `bi`, is shown in Figure 6.3. The `dir` signal controls the direction of the I/O port. When it is '0', the port is used as an input port. The tri-state buffer is in a high-impedance state, and thus the `sig_out` signal is blocked. The external signal connected to the `bi` port is routed to the `sig_in` signal. When the `dir` signal is '1', the port is used as an output port and the `sig_out` signal is connected to an external circuit. Note that the `sig_out` signal is implicitly routed back to the `sig_in` signal when the `dir` signal is '1'. If this causes a problem, we can add an additional tri-state buffer to break the return path, as shown in Figure 6.4. Since the control signals of tri-state buffers are connected to a complementary enable signal, only one tri-state buffer is enabled at a time.

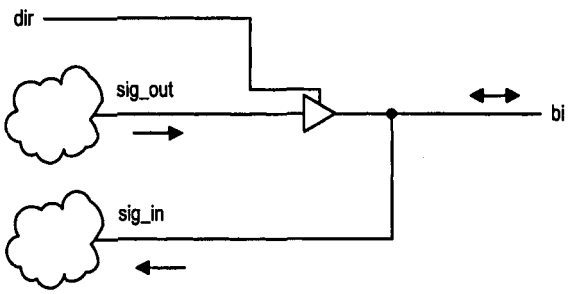


Figure 6.3 Single-buffer bidirectional I/O port.

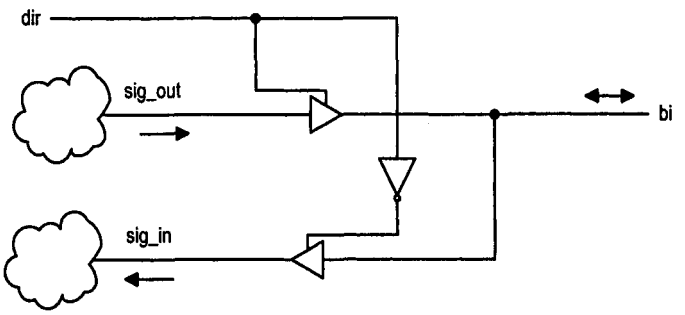


Figure 6.4 Dual-buffer bidirectional I/O port.

The VHDL description for a bidirectional port is straightforward. We first specify the mode as **inout** in port declaration and then describe the tri-state buffer accordingly. The VHDL segment for the single-buffer diagram of Figure 6.3 is

```
entity bi_demo is
   port(
      bi: inout std_logic;
      . . .

begin
   sig_out <= output_expression;
   . . .
   some_signal <= expression_with_sig_in;
   . . .
   bi <= sig_out when dir='1' else 'Z';
   sig_in <= bi;
   . . .
```

To accommodate the dual-buffer configuration of Figure 6.4, we just need to modify the last statement to reflect the change:

```
sig_in <= bi when dir='0' else 'Z';
```

Tri-state buffer–based bus Another application of the tri-state buffer is to form a bus. The diagram of a simple tri-state buffer–based bus (or simply tri-state bus) is shown in Figure 6.5, in which four sources are connected to the bus. The signal `src_select` specifies

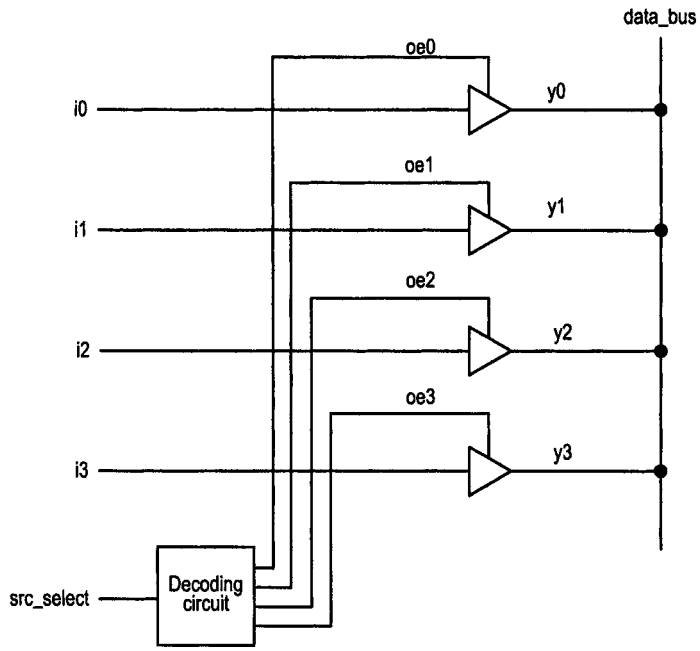


Figure 6.5 Tri-state bus.

which input source is to be placed on the bus. It is connected to a decoding circuit that generates four *non-overlapping* control signals, `oe(0)`, `oe(1)`, `oe(2)` and `oe(3)`. Only one can be activated at a time, and the input connected to the activated buffer is placed on the bus. The VHDL code for this circuit is

```
-- binary decoder
with src_select select
   oe <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when others; -- "11"
-- tri-state buffers
y0 <= i0 when oe(0)='1' else 'Z';
y1 <= i1 when oe(1)='1' else 'Z';
y2 <= i2 when oe(2)='1' else 'Z';
y3 <= i3 when oe(3)='1' else 'Z';
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

Despite its simple appearance, the internal tri-state buses presents a serious problem in the development flow. Since the theoretical models of most EDA algorithms are based on Boolean algebra, which is defined according to two logic values, the software tools cannot handle the high-impedance state. The tri-state bus thus imposes a problem in optimization, timing analysis, verification and testing. Furthermore, internal tri-state bus is technology dependent, and thus the design is less portable.

Table 6.3 Function tables of a 3-to-2 priority encoder

Input req	Output code
1 0 0	10
1 0 1	10
1 1 0	10
1 1 1	10
0 1 0	01
0 1 1	01
0 0 1	00
0 0 0	00

Input req	Output code
1 – –	10
0 1 –	01
0 0 1	00
0 0 0	00

Table 6.4 Don't-care used as an output value

input $a\ b$	output f
0 0	0
0 1	1
1 0	1
1 1	–

A tri-state bus essentially performs multiplexing. For example, the previous design can be replaced by a 4-to-1 multiplexer:

```
with src_select select
   data_bus <= i0 when "00",
               i1 when "01",
               i2 when "10",
               i3 when others;   -- "11"
```

This scheme is more robust and portable and thus is the preferred choice. The major application of the tri-state bus is to construct the external back-plan bus of a printed circuit board. An add-on card can easily be added to or removed from the bus without affecting subsystems residing on other cards.

6.3.3 Use of the '-' value

Don't-care is not a valid logic value in Boolean algebra but is used to facilitate the design process. Don't-care can be used as an input value to make a function table clear and compact. For example, the original function table of a 3-input priority encoder is shown on the left of Table 6.3. When `req(2)` is '1', the output should be "10" regardless of the values of other requests. Instead of using four rows, we can use `1--` to indicate the condition. The revised table, as on the right of Table 6.3, is more compact and more descriptive.

When used as an output value, don't-care indicates that the exact value is not important. This happens when some of the input combinations are not used. During the synthesis process, we can assign a value that helps to reduce the circuit complexity. A simple example is shown in Table 6.4, in which the output value for the input pattern "11" is don't-care. If don't-care is assigned to '0' during synthesis, f becomes $a' \cdot b + a \cdot b'$. On the other hand, when it is assigned to '1', f can be simplified to $a + b$, which requires much less hardware.

According to the definition of the std_logic data type, the '-' value is designated as "don't-care." However, VHDL treats '-' as an independent symbolic value of the std_logic data type rather than "0 or 1." This definition is somewhat different from our conventional use and may lead to unexpected behaviors and subtle mistakes. The following paragraphs discuss the use of this value.

Use of '-' as an input value Let us first examine the issues related to using '-' as an input value. Consider the priority function of Table 6.3. We may be tempted to code the circuit as follows:

```
y <= "10" when req="1--" else
     "01" when req="01-" else
     "00" when req="001" else
     "00";
```

The code is syntactically correct. However, in a physical circuit, an input signal can only assume a value of '0' or '1' but never '-', and thus the req="1--" and req="01-" expressions will always be false. If the value of the req signal is "111", none of the Boolean expression is true and "00" will be assigned to y accordingly. To correct the problem, we have to eliminate the comparison of '-' in Boolean expressions:

```
y <= "10" when req(2)='1' else
     "01" when req(2 downto 1)="01" else
     "00" when req(2 downto 0)="001" else
     "00";
```

The code is just for demonstration purposes and is not very efficient. Better code for priority encoding circuit was illustrated in Section 4.3.1.

In the IEEE numeric_std package, there is a function, std_match(), which performs don't-care comparisons according to the traditional interpretation. The function compares two inputs of std_logic_vector data type and interprets '-' as a don't-care in a conventional sense. The previous code can be written as

```
. . .
use ieee.numeric_std.all;
. . .
y <= "10" when std_match(req,"1--") else
     "01" when std_match(req,"01-") else
     "00" when std_match(req,"001") else
     "00";
```

Our discussion of '-' is also applied to the choice expression in a selected signal assignment statement and case statement. For example, the following code seems to be the direct implementation of the compact function table of Table 6.3:

```
with req select
   y <= "10" when "1--",
        "01" when "01-",
        "00" when "001",
        "00" when others;
```

The code is syntactically correct. Again, since a physical input signal can never assume a value of '-', the choices "1--" and "01-" will never occur. If the value of the req signal is "111", there is no match and "00" will be assigned to y. There is no easy fix in this case. We must explicitly specify choice expressions in terms of '0' and '1', as in the original left function table of Table 6.3. The correct VHDL code is

```
with req select
   y <= "10" when "100"|"101"|"110"|"111",
        "00" when "010"|"011",
        "00" when others;
```

Use of '-' as an output value Don't-care can also be used as an output value and assigned to a signal. For example, the function table of Table 6.3 can easily be translated to VHDL code:

```
sel <= a & b;
with sel select
   y <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '-' when others;
```

The code is syntactically correct. According to the VHDL definition, '-', not "0 or 1," will be assigned to y if `sel` is "11". Since a real '-' does not exist in a physical implementation, this symbol cannot be synthesized. During synthesis, some software flags an error, and others treat it as a conventional don't-care and perform optimization accordingly.

6.4 VHDL SYNTHESIS FLOW

Synthesizing VHDL code is the process of realizing a VHDL description using the primitive logic cells from the target device's library. It is a complex process. To make it manageable, we normally divide VHDL synthesis into steps, including *high-level synthesis*, *RT-level synthesis*, gate-level synthesis (commonly known as *logic synthesis*) and cell-level synthesis (commonly known as *technology mapping*). High-level synthesis transforms an algorithm into an architecture consisting of a *data path* and *control path*. It is substantially different from the other three steps and is done by specialized software tools. It is reviewed in Section 12.7.

RT-level synthesis, logic synthesis and technology mapping generate structural netlists utilizing generic RT-level components, generic gate-level components and device-dependent cells respectively. The detailed flow is shown in Figure 6.6. Basically, the entire circuit is transformed and optimized level by level, from an RT-level netlist to a gate-level netlist and then to a cell-level netlist, as shown in the left column of the flowchart. Some RT-level components, such as adder and comparator, can be quite complex. They are normally handled by a *module generator*, as shown in the right column of the flowchart. Our current discussion is limited to the synthesis flow of combinational circuits. It can easily be expanded to include sequential circuits, which are discussed in Chapter 8.

6.4.1 RT-level synthesis

RT-level synthesis transforms a behavioral VHDL description into a circuit constructed by components from a generic RT-level library. The term *generic* implies that the components are common to all technologies and thus the library is not technology dependent. The components can be classified into three categories: functional units, routing units and storage units. *Functional units* are used to implement the logic, relational and arithmetic operators encountered in VHDL code. *Routing units* are various multiplexers used to construct the routing structure of a VHDL description, as discussed in Chapters 4 and 5.

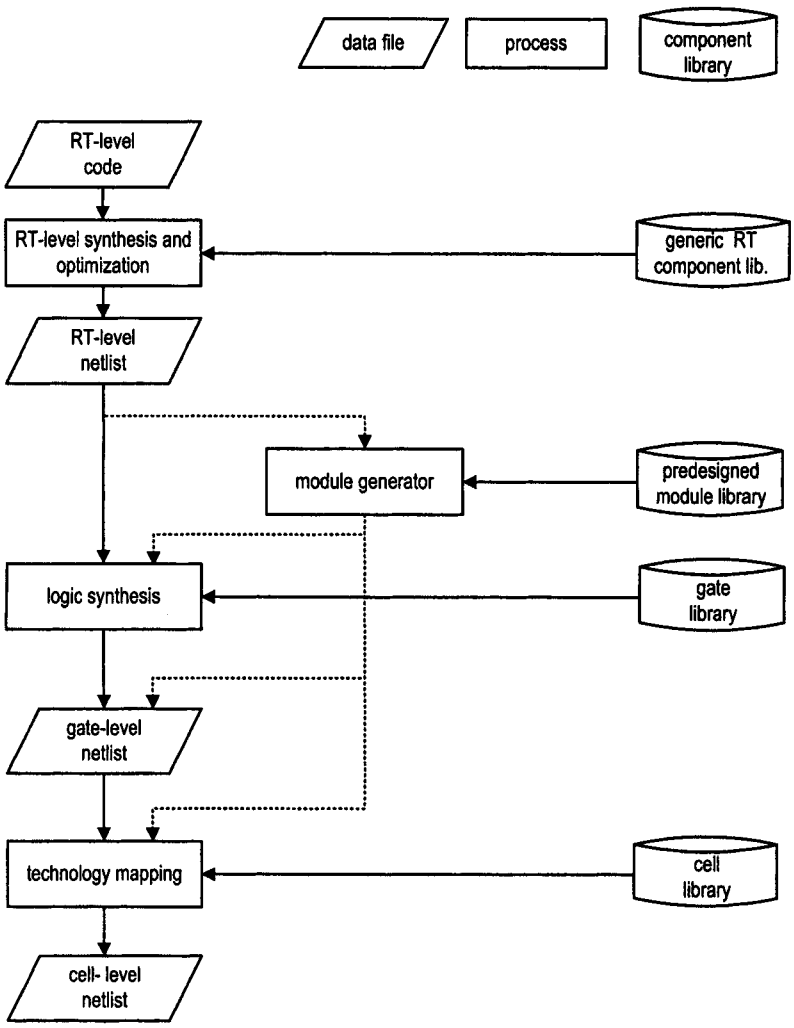


Figure 6.6 Synthesis flow.

Storage units are registers and latches, which are used only in sequential circuits and are discussed in Chapter 8.

RT-level synthesis includes the derivation and optimization of an RT-level netlist. During the process, VHDL statements are converted into corresponding structural implementation, somewhat similar to the derivation of conceptual diagrams discussed in Chapters 4 and 5. Some optimization techniques, such as operator sharing, common code elimination and constant propagation, can be applied to reduce circuit complexity or to enhance performance. Unlike gate- and cell-level synthesis, optimization at the RT level is performed in an ad hoc way and its scope is very limited. Good design can drastically alter the RT-level structure and help software to derive a more effective implementation.

6.4.2 Module generator

After the RT-level synthesis, the initial description is converted to a netlist of generic RT-level components. These components have to be transformed into lower-level implementation for further processing. Some RT-level components, such as logical operators and multiplexers, are simple and can be mapped directly into gate-level implementation. They are known as *random logic* since they show less regularity and can be optimized later in logic synthesis. The other components are quite complex and need special software, known as a *module generator*, to derive the gate-level implementation. These components include adder, subtractor, incrementor, decrementor, comparator and, if supported, shifter and multiplier as well. They usually show some kind of repetitive structure and sometimes are known as *regular logic*. Regular logic is usually designed in advance. A module generator can produce modules in different levels of detail:

- Gate-level behavioral description.
- Presynthesized gate-level netlist.
- Presynthesized cell-level netlist.

A gate-level behavioral description can be thought of as VHDL code that uses only simple signal assignment and logical operators, which can easily be mapped to a gate-level netlist. The description is general and independent of underlying device technology. The description will be flattened and combined with the random logic to form a single gate-level netlist. The merged netlist will be synthesized together later in logic synthesis. Chapter 15 discusses the generation of some frequently used components.

Because of the regular and repetitive nature of these components, it is possible to further explore their properties and manually derive and synthesize the netlist at the gate level or even at the cell level. Manual design can explore this regularity and derive a more efficient implementation. The resulting circuit is more efficient than a circuit obtained from logic synthesis. When a presynthesized gate- or cell-level netlist is used, it will not be flattened and merged with the random logic. The random logic will be independently processed through logic synthesis and even technology mapping. The netlist of random logic and the netlists of regular components will be merged after these processes. The right column in the synthesis flow of Figure 6.6 shows the various possibilities for module generation.

There are two advantages to the non-flattened approach. First, it can utilize predesigned, highly optimized modules. Second, since these modules are extracted from the original circuit, the remaining part is smaller and thus is easier to process and optimize. On the other hand, the non-flattened modules may isolate the random logic and thus reduce the chance for further optimization. For example, the adder of Figure 6.7 separates the random logic circuits into two parts and forces them to be processed independently. It may introduce

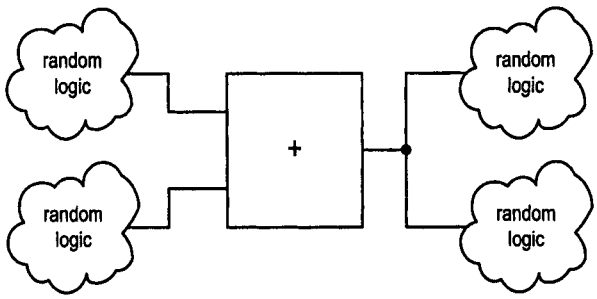
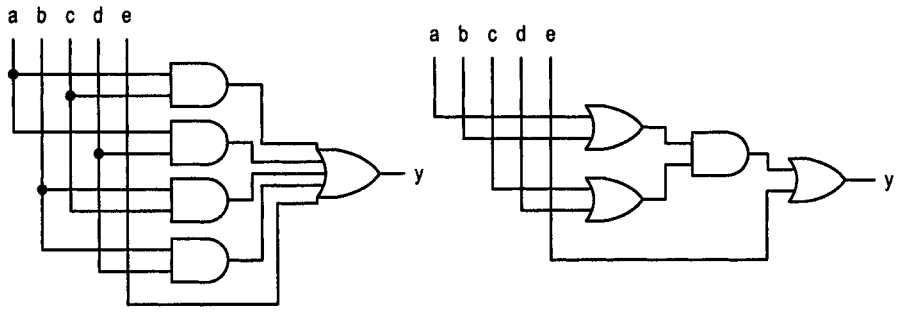


Figure 6.7 Random logic with a regular component.



(a) Two-level implementation (b) Multilevel implementation

Figure 6.8 Two-level versus multilevel implementation.

more optimization opportunities if we flatten the adder, merge it with the four random logic circuits, and then process and optimize them together. There is no clear-cut rule as to which approach is more effective. Some synthesis software allows users to specify the desired option.

6.4.3 Logic synthesis

Logic synthesis is the process of generating a structural view using an optimal number of generic primitive gate-level components, such as a not gate, and gate, nand gate, or gate and nor gate. Again, the term *generic* means that the components are not tied to a particular device technology and there is no detailed information about the components' size or propagation delay. At this level, a circuit can be expressed by a Boolean function, and these generic components are essentially the operators of Boolean algebra. Logic synthesis can be divided into *two-level synthesis* and *multilevel synthesis*.

The most commonly used two-level form is the sum-of-products form, in which the first level of logic corresponds to and gates and the second level to or gates. An example is shown in Figure 6.8(a). Other two-level forms can easily be derived from the sum-of-products form. Two-level synthesis is to derive an optimal sum-of-products form for a Boolean function. The goal of optimization is to reduce the number of product terms (i.e., the number of and gates) and the number of input literals (i.e., the total fan-ins of and gates). The well-known Karnaugh map technique is a method to manually obtain the optimal two-

level implementation for a circuit with up to four or five inputs. A more realistic circuit may contain dozens or even several hundred inputs and cannot be optimized manually. Obtaining the optimal two-level circuit is actually an intractable problem and thus is not practical. However, this process is well understood, and many efficient algorithms to obtain good, suboptimal circuits have been developed.

Because of the large number of fan-ins for the and and or gates, the two-level sum-of-products form can only be implemented by using a special ASIC structure, known as *programmable logic array* (*PLA*), and, with some modification, by using *programmable array logic* (*PAL*)-based CPLD devices. However, the two-level form is a formal way of expressing Boolean functions and is frequently used as a basis for processing and manipulating logic expressions. Two-level synthesis can reduce the information needed to represent a function and theoretically can serve as a staring point of multilevel processing.

Multilevel representation, as its name indicates, expresses a Boolean function by using multiple levels of gates. Its form is far less stringent than that of the two-level form and provides several degrees of freedom, leading to better efficiency and more flexibility. The implementation may be exploited by optimizing area, by optimizing delay, or even by obtaining an optimal area–delay trade-off point. An example of multilevel implementation of the previous two-level implementation is shown in Figure 6.8(b). It reduces both the number of gates and the number of fan-ins. Modern device technologies are based on small cells whose fan-in is limited to a small number. Thus, multilevel synthesis is more appropriate.

Processing and optimizing a multilevel logic are more difficult. Optimization is normally based on heuristic methods, which exploit various Boolean or algebraic transformations or search and replace circuit patterns according to a rule database. Because of the flexibility of multilevel representation, synthesis results vary significantly, and a minor modification in initial description may lead to a totally different implementation.

6.4.4 Technology mapping

Logic synthesis generates an optimized netlist that utilizes generic components. *Technology mapping* is the process of transforming the netlist using components from the target device's library. These components are commonly referred to as *cells*, and the technology library is normally provided by a semiconductor vendor who manufactured (as in FPGA technology) or will manufacture (as in ASIC technology) the device. Whereas a generic component is defined by its function, a cell is further characterized by a set of physical parameters, such as area, delay, and input and output capacitance load. In the case of ASIC technology, each cell is associated with the physical layout or prediffused patterns.

Although technology mapping can be done by simple translation between generic components and logic cells, the resulting circuit is not very efficient since the translation does not exploit the functionalities, areas and delays of the cells. Obtaining optimal mapping is a very difficult process, which involves intractable problems. Again, heuristic and rule-based algorithms are used to find suboptimal solutions. The following subsections use two simple examples to illustrate the technology mapping process of a hypothetical standard-cell ASIC library and a 5-input look-up table (LUT)–based FPGA.

Standard-cell technology A library from standard-cell technology normally consists of several dozen to several hundred cells, including combinational, sequential and interface cells. Combinational cells consist of simple gates, such as and, or, nand, nor, xor etc., and sometimes slightly complex circuits, such as 1-bit full adder, 1-bit 2-to-1 multiplexer

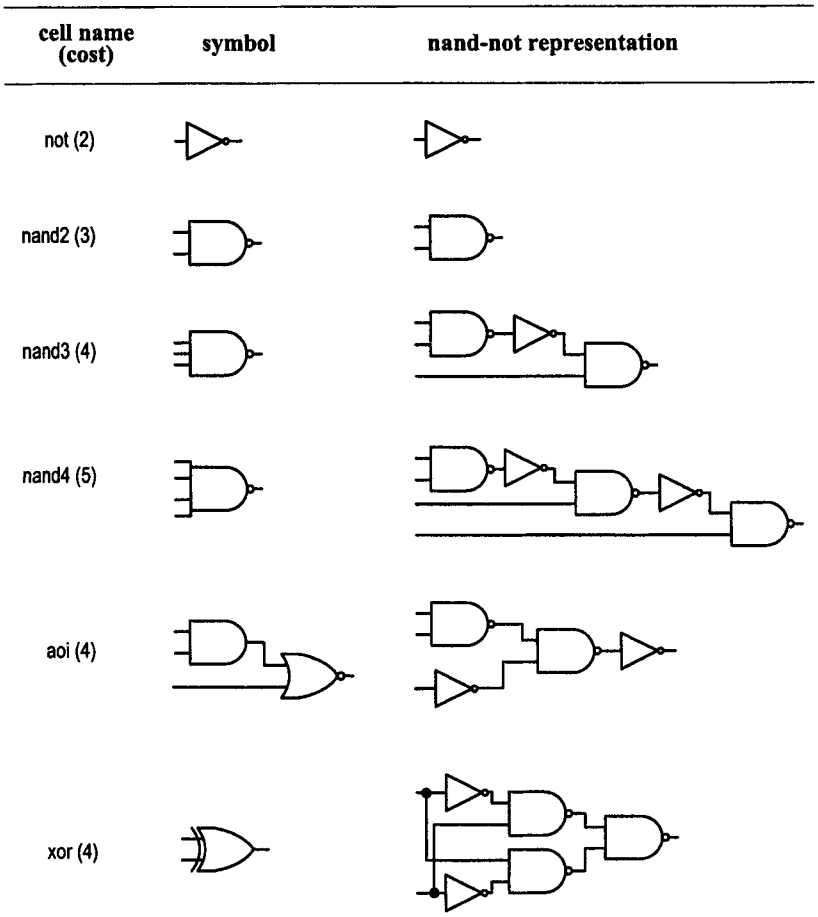
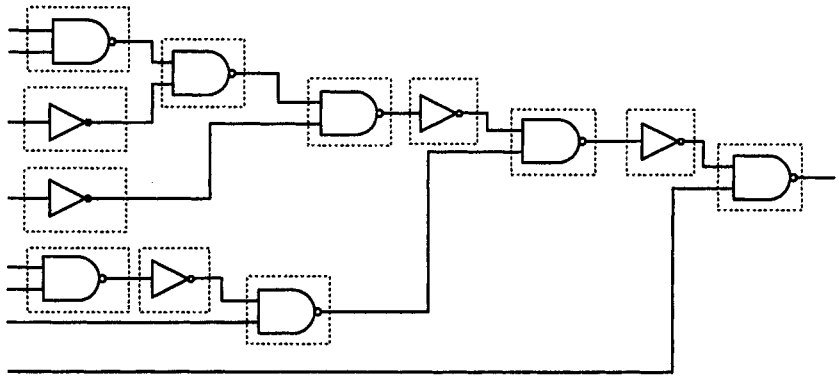


Figure 6.9 Simple hypothetical ASIC cell library.

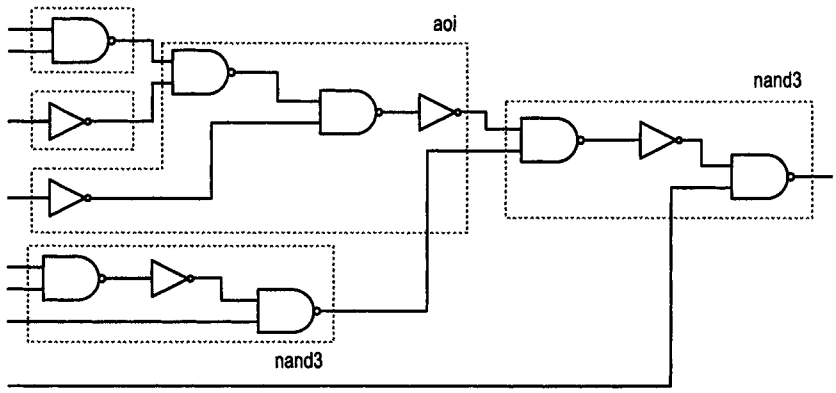
etc. A simple hypothetical technology library with seven cells is shown in Figure 6.9. The columns are the name of the cell, its relative area (cost), its symbol and its normal form. The *normal form*, which represents a cell using 2-input nand gates and inverters, is used to facilitate the mapping process.

The cells of a technology library are optimized and tuned for a particular technology. They are manually designed from scratch at the transistor level rather than being based on simple logic gates. For example, if the `aoi` cell is implemented using the simpler `nand2` and `not` cells, its area is 11, which is about four times the area of the `nand2` cell. However, if it is implemented directly at the transistor level, its area is 5, which is about twice the area of the `nand2` cell. This explains why there are many different primitive cells in a typical standard-cell library. Furthermore, since fine adjustments can be made at the transistor level, multiple cells of different area–delay trade-offs may exist for the same logic function.

The mapping can best be illustrated by the example shown in Figure 6.10. The initial mapping in Figure 6.10(a) is a trivial one-to-one gate-to-cell translation and its area is 31. The better one, in Figure 6.10(b), is optimized and its area is reduced to 17. Although

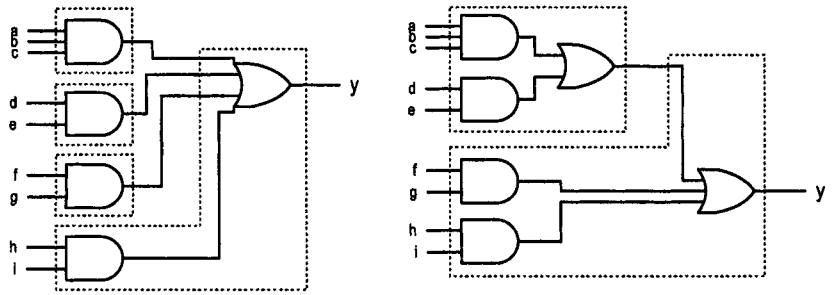


(a) Initial mapping



(b) Better mapping

Figure 6.10 Standard-cell technology mapping example.



(a) Initial mapping (b) Better mapping

Figure 6.11 LUT-based FPGA technology mapping example.

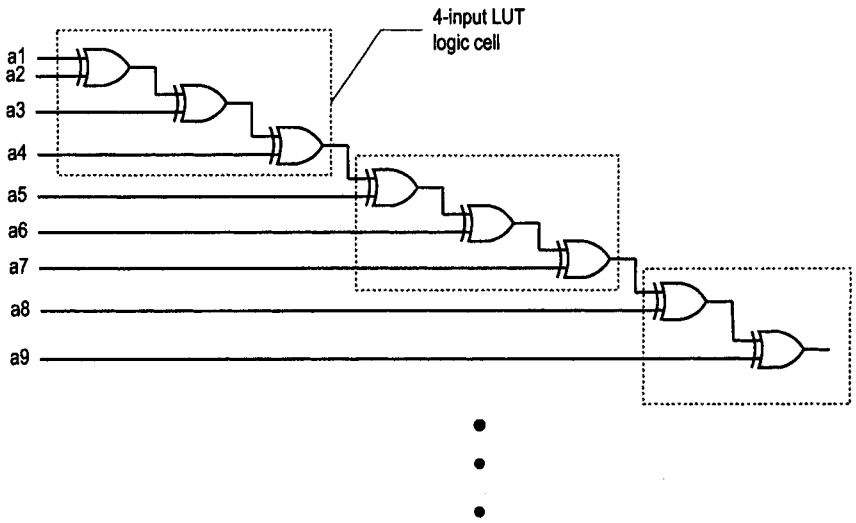
this is a simple example, it demonstrates the importance of good mapping as well as the complexity of the technology mapping process.

LUT-based FPGA technology Because an FPGA device is prefabricated in advance, its technology library normally consists of only a single cell. This cell can, however, be "programmed" or configured to perform different logic functions. The most commonly used construction is based on a small *look-up table* (*LUT*). We can program a LUT by specifying its contents, as in a truth table description of a logic function. If a LUT can accommodate 2^n rows (i.e., n inputs), it can be used to realize any combinational function with n or fewer inputs. A typical FPGA cell consists of a 4-, 5-, or 6-input LUT and a D-type flip-flop.

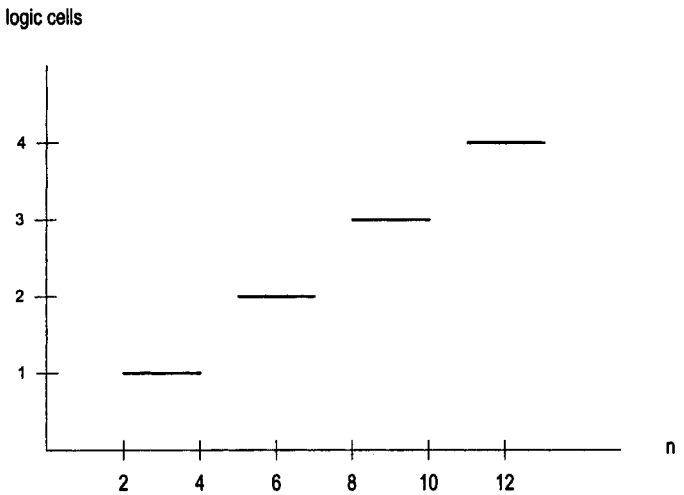
An example of technology mapping using 5-input LUT cells is shown in Figure 6.11. Since a LUT cell concerns only the number of inputs, the netlist does not need to be converted into normal form. The mapping in Figure 6.11(a) is a trivial one-to-one gate-to-cell translation, and it requires four LUT cells. The mapping in Figure 6.11(b) is more efficient and reduces the number to only two LUT cells.

Precaution with FPGA technology From technology mapping's point of view, one difference between ASIC and FPGA technologies is the size of the cells. The cell size of an ASIC device is very small, and thus any minor adjustment will be reflected in the implementation. For example, the previous standard-cell library has 2-, 3- and 4-input nand cells. If we can improve our design by eliminating one input of a product term in the logic expression, we can use a smaller nand cell and reduce the circuit area by a small amount.

On the other hand, the cell size of a FPGA device is relatively large. A 5-input LUT-based cell can implement any 1-, 2-, 3-, 4- or 5-input logic function, regardless of the complexity of the function. A wide range of functions can be implemented by this cell, and all of them are considered to have the same area under the FPGA technology. For example, both the $a \cdot b$ and $a \oplus b \oplus c \oplus d \oplus e$ expressions can be mapped into a single LUT cell. Although the internal utilizations of the cells are very different, the two expressions are considered to have the same area. This may cause an unexpected result when we synthesize a circuit using FPGA technology. This phenomenon will be further amplified if we take into consideration the built-in flip-flop within a logic cell. For example, we can construct a 1-bit counter and its area remains a single cell.



(a) 4-input LUT mapping of an odd-parity circuit



(b) Plot of number of logic cells versus input size

Figure 6.12 Discontinuity of LUT cell-based implementation.

The FPGA-based implementation may also exhibit a "discontinuity" phenomenon. For example, let us use a 4-input LUT logic cell to implement an odd-parity circuit, which has an expression of

$$a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n$$

A simple cascading chain implementation and mapping is shown in Figure 6.12(a). The number of logic cells needed for different input size (i.e., n) is plotted in Figure 6.12(b). It looks like a staircase and exhibits discontinuities (i.e., a sudden change) at certain points. For example, if we increase the input size from 6 to 7, there is no change in the number of

logic cells, and thus the area remains unchanged. But if we change the input size from 7 to 8, the number of logic cells increases from 2 to 3, and thus the area increases 50%.

For a larger, more complex circuit, we can expect that the cell utilization and discontinuity will average out and the result is more like that of an ASIC device. Nevertheless, occasional fluctuations and randomness are unavoidable, and targeting an FPGA device still introduces a new dimension of complication in synthesis. Although the discussion in the remainder of the book can be applied equally to both ASIC and FPGA devices, we target the design using ASIC devices for the area and performance data.

6.4.5 Effective use of synthesis software

Despite its fundamental limitation, synthesis software is still a powerful and necessary tool, which can automate many design tasks and perform certain tedious and repetitive computations. A good designer should understand the capabilities and limitation of software, and know what this tool can and cannot do as well as when to compromise.

VHDL description of logical operators In general, synthesis software is very effective in performing logic synthesis and technology mapping for a small to moderate-sized circuit whose complexity is around 5000 to 50,000 equivalent gates. Although optimization involves intractable problems, these problems have been studied thoroughly and many good heuristics and searching procedures have been developed. Furthermore, although a circuit is processed at the gate or cell level, even a very simple design consists of hundreds or thousands of components. It is not practical to manipulate the design manually at this level.

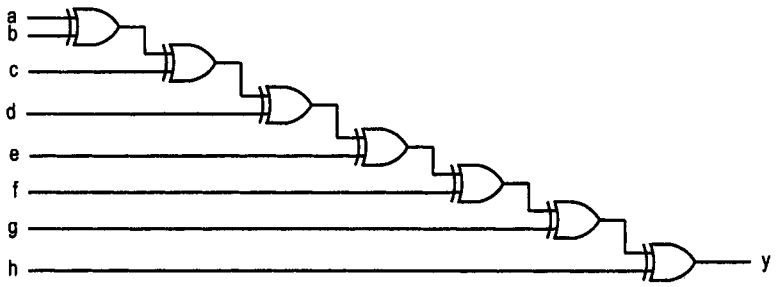
VHDL logical operators can be mapped directly to gate-level components. Their implementations are simple and straightforward. Since synthesis is very effective at this level, we need not worry about the sharing and optimization of logical operators in a VHDL description.

VHDL description of arithmetic and relational operators Optimization at the RT level involves complex arithmetic and relational operators and routing structure. It is not well developed and is frequently done on an ad hoc basis. Human intervention is required, and we have to specify explicitly the desired design in a VHDL description. Simple modifications on code frequently can improve circuit efficiency significantly.

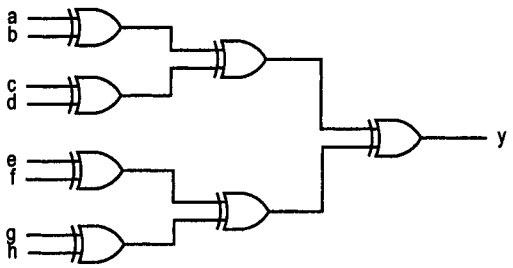
There is no comprehensive procedure or algorithm to detect sharing and to perform optimization for arithmetic and relational operators. It frequently depends on the designer's insight and knowledge of a circuit. VHDL is a good vehicle to explore design at this level. Sections 7.2 and 7.3 provide a comprehensive array of examples for this topic.

VHDL description of layout and routing structure *Routing structure* indicates how "data" propagate through various parts of the system, from input ports to output ports. Although a VHDL program cannot explicitly specify the placement of components or the layout of a design, it implicitly describes the routing structure and, to some degree, the shape of the implementation. Recall that each VHDL statement can be considered as a circuit part, and a VHDL program implicitly connects these parts. Although all parts of a combinational circuit operate concurrently, some outputs of these parts are not valid initially. The valid value can be thought of as data that propagates from one part to another and eventually to the circuit output. The data flow forms a routing structure, which, in turn, implicitly determines the shape or layout of the physical circuit.

Regardless of the shape of the initial VHDL description, the placement and routing process will eventually implement the circuit on a two-dimensional silicon chip. If the



(a) Cascading-chain structure



(b) Tree structure

Figure 6.13 Routing structures of an odd-parity circuit.

shape of the initial description resembles the shape of the chip, the description can help the placement and routing process and make the final implementation smaller and faster. Two routing structures of a simple example of an odd-parity circuit are shown in Figure 6.13. The one in Figure 6.13(a) is a cascading-chain structure described by the statement

```
y<=(((((((a xor b) xor c) xor d) xor e) xor f) xor g) xor h);
```

and the one in Figure 6.13(b) has a tree structure described by the statement

```
y<=((a xor b) xor (c xor d)) xor ((e xor f) xor (g xor h));
```

Both structures use the same number of xor gates, but the propagation delay is much smaller in the tree structure.

Although synthesis software can recognize a few specific patterns and rearrange the routing structure on a local basis, it cannot make any major global change. Good VHDL coding can outline the basic "skeleton" of the implementation and provide a framework for synthesis. It has a greater impact than the local optimization performed by synthesis software. The coding technique is discussed in detail in Section 7.4 .

6.5 TIMING CONSIDERATIONS

A digital circuit cannot respond instantaneously, and the output is actually a function of time. The most important time-domain characteristic is the *propagation delay*, which is the time required for the circuit to generate a valid, stabilized output value after an input change. It is one of the major design criteria for a circuit.

Another time-domain phenomenon, known as a *hazard*, is the possible occurrence of unwanted fluctuations of an output signal before it is stabilized. Although a hazard is a

transient response, it may cause circuit malfunction in a poorly conceived design. The following subsections examine the propagation delay and hazard in more detail and discuss several timing issues that have an impact on synthesis.

6.5.1 Propagation delay

It takes a digital circuit a certain amount of time to reach a valid stable output response after an input change. In digital design, we treat this time as the delay required to propagate a signal from the input port to the output port, and call it *propagation delay* or simply *delay*. A digital system normally has multiple input and output ports, and each input–output path may exhibit a different delay. We consider the worst-case scenario and use the largest input–output delay as the system's propagation delay.

The propagation delay reflects how fast a system can operate and is usually considered as the *performance* or the *speed* of the system. Combined with the circuit size (area), they are the two most important design criteria of a digital system.

To compute the delay of a system, we first determine the delays of individual components and identify all possible paths between input and output ports. We then calculate the delay of each path by summing up the individual component delays of the path and eventually determine the system delay.

The system delay calculation clearly depends on the information of its underlying components. The best estimation can be obtained at the cell level since the netlist is final, and the accurate physical and electrical characteristics of cells are provided. The least accurate estimation is at the RT level since the components must be further transformed and optimized.

Propagation delay at the cell level To determine the exact time-domain behavior of a cell, we have to examine and analyze it at the transistor level, which is modeled by transistors, resistors and capacitors. The delay is due mainly to parasitic capacitance, which occurs at two overlapping layers and thus exists everywhere. When a transistor changes state, these capacitors have to be charged or discharged and thus introduce a delay. Analyzing a cell at this level is extremely complex and can be done only at a small scale. The analysis provides basic data for cell-level modeling.

To manage the complexity, timing analysis at the cell level has to rely on a much simpler model. One commonly used approach is a simplified linear model, in which all parasitic capacitance is lumped as a single capacitor and only the first-order effect is considered. In this model, the delay of a cell is expressed as

$$delay = d_{intrinsic} + r * C_{load}$$

The first term in the expression, $d_{intrinsic}$, is associated with the internal circuit of the cell. It models the time required for transistors to change state (i.e., switch on or off). The second term is associated with the external circuits driven by the cell. The parameter C_{load} is the total capacitive load driven by this cell, which includes the input capacitance of cells connected to the output of current cell and the parasitic capacitance of the interconnect wires. An example is shown in Figure 6.14. The load is the summation of the input capacitance of three cells driven by the and gate (C_{g1}, C_{g2} and C_{g3}) and parasitic capacitances of three interconnect wires (C_{w1}, C_{w2} and C_{w3}).

The r parameter represents the driver capability of the cell and can loosely be considered as the output impedance of the cell. When r is small, the cell can allow more current (i.e., larger driver capability) and thus can charge or discharge the capacitance load in a shorter

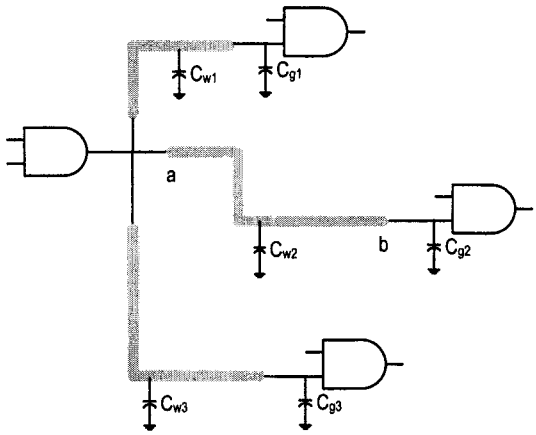


Figure 6.14 Delay estimation at the cell level.

period, leading to a smaller delay. At the transistor level, we can reduce the delay by using a larger transistor to increase the driver capability.

Impact of wiring on cell-level delay estimation The accuracy of cell-level delay estimation depends on several factors. The first factor is the accuracy of the parameters used in delay calculation. We can obtain fairly accurate values for $d_{intrinsic}$, r, and input capacitance from the manufacturer's data sheet. After technology mapping, fan-out of each cell can be obtained from the netlist, and thus the total input capacitance load can easily be determined. The wire capacitance, on the other hand, depends on the actual length and location of each wire. Since this information is not available at the synthesis stage, software sometimes uses a statistical model to provide a rough estimation. Accurate information can only be extracted after place and routing is performed. This is one reason that the system has to be simulated and verified again after the placement and routing process.

The second factor is the accuracy of the model. The linear cell-level model is only an approximation and ignores higher-order effects. In some circumstances, these effects become more dominant, and more sophisticated models have to be used. For example, a more complex distributed RC model can be used to obtain better estimation than a simple lumped circuit. Some models for a wire between points a and b are shown in Figure 6.15(b) - (d).

When the transistor geometry is relatively large, the wire capacitance and higher-order effects do not contribute much to the overall delay and can safely be ignored. Accurate timing information can be obtained in the synthesis stage. However, as the transistor becomes smaller and submicron technology becomes available, the wiring delay gradually becomes the dominant part and the high-order effects have more impact. This makes the design process harder since we need to do placement and routing to obtain accurate timing information.

In addition to the inherent errors of approximation, the fabrication process and operation environment (such as temperature) affect the delay characteristics as well. In general, there is no way that we can control the exact delay of a cell. A device manufacturer can only guarantee the boundary of operation, normally in terms of the *maximal* propagation delay. While VHDL incorporates the timing aspect in the language, it is primarily for modeling purposes. For example, we can specify an and gate with a 2-ns delay as:

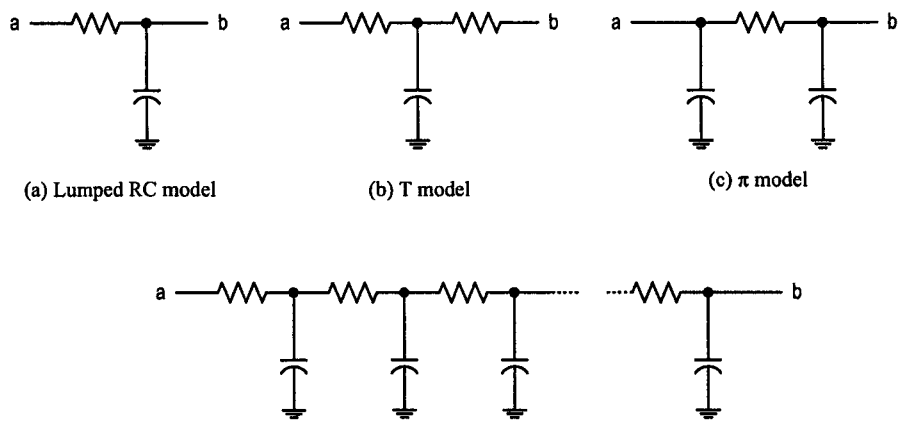


Figure 6.15 Wiring models.

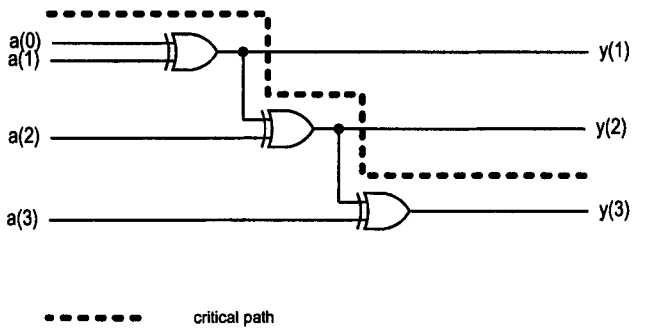


Figure 6.16 Topological critical path.

```
f <= b and c after 2 ns;
```

During synthesis, the timing part will be completely ignored since there is no technology that can produce a gate with an exact 2-ns delay.

System delay Once cell delays are known, we can calculate the delay of a path by adding the individual cell delays along the path. A digital system typically has many paths between input and output ports, and their delays are different. Since the system has to accommodate the worst-case scenario, the system delay is defined as the longest delay. The corresponding path is considered as the longest path and is known as the *critical path.*

A simple method of determining the critical path is to treat the netlist as a graph, extract all possible paths and then determine the longest path accordingly. An example is shown in Figure 6.16. Since the topology of the system alone determines the critical path, it is also known as the *topologically critical path.*

Using the topologically critical path to determine the system delay may occasionally overestimate the actual value because of a *false path,* a path along which no signal transition can propagate. An example of a false path is shown in Figure 6.17. The topologically critical

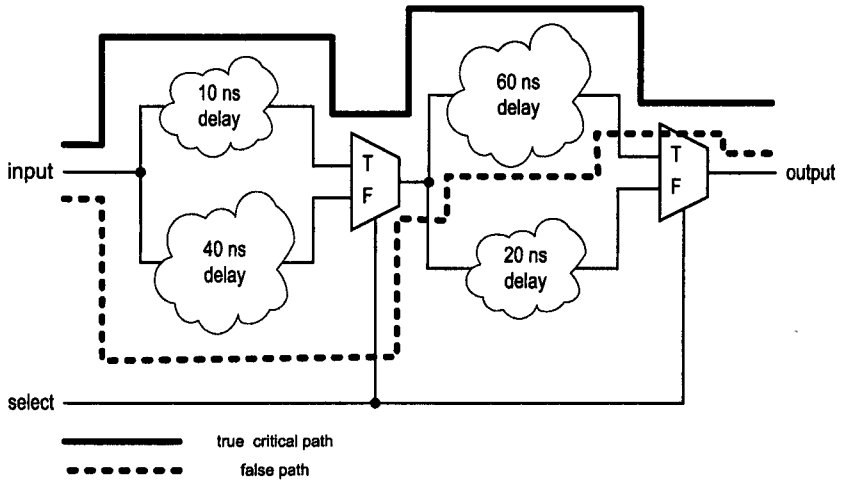


Figure 6.17 False path.

path is the route that includes the circuit with 40- and 60-ns delays. However, in realty, the input signal can propagate through either the top part (when the `select` signal is '1') or the bottom part (when the `select` signal is '0') but never the topologically critical path. Since no signal actually passes through the false paths, they should be excluded from system delay calculation. To determine the true critical path is much harder since the analysis involves not only the topology but also the internal logic operations.

Because of the large number of cells in a system, cell-level timing analysis is always done by software. This feature is normally integrated into the synthesis software. Most software uses the topological critical path to determine the system delay. Some software allows users manually to exclude potential false paths.

Delay estimation at the RT level We can apply the same principle to analyze and calculate the propagation delay at the RT level. The accuracy of the calculation depends on the components used in the RT-level diagram. If an RT-level diagram consists primarily of simple logical operators and is mainly random logic, the circuit is subjected to a significant amount of transformation and optimization during logic synthesis and technology mapping. Since the final circuit may not resemble the original RT-level diagram, the RT-level delay calculation will not faithfully reflect delay in the synthesized circuit.

On the other hand, if an RT-level diagram consists of many complex operators and function blocks, these components become the dominating part of a delay calculation. Furthermore, since these components are presdesigned and optimized, their delay characteristics will not change significantly during synthesis. Thus, the delay calculation will be much more accurate for this type of circuit. Calculating RT-level delay allows us to identify the critical path and thus better understand the performance of the circuit, and eventually helps to derive an efficient design and VHDL code with the desired area–delay characteristics. RT-level delay estimation is shown in many design examples in the subsequent chapters.

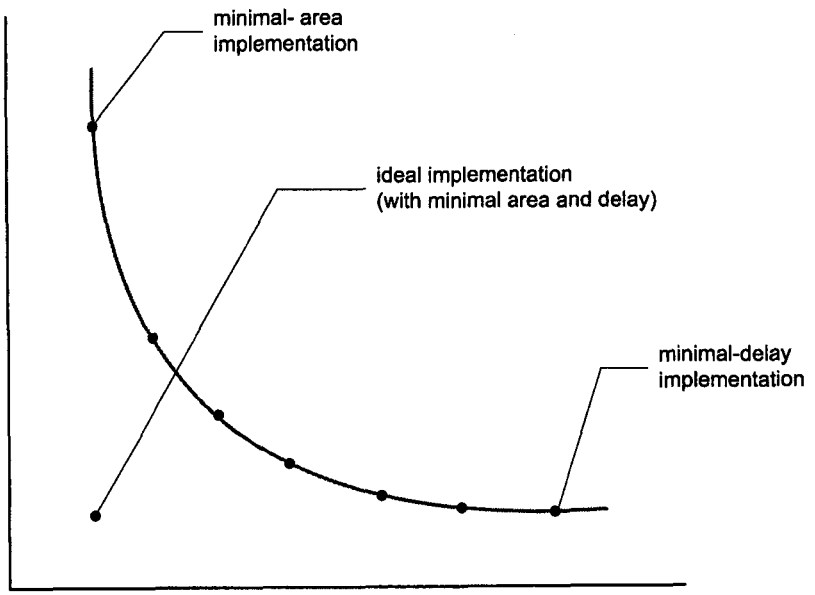
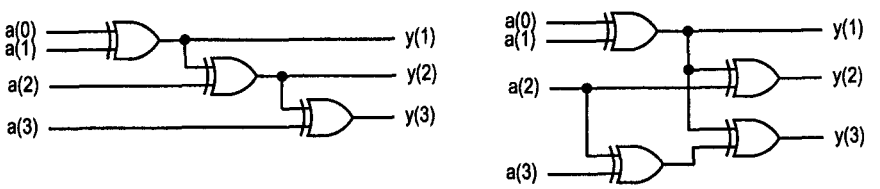


Figure 6.18 Area–delay trade-off curve.



(a) Optimized for area

(b) Optimized for delay

Figure 6.19 Delay constraint implementation.

6.5.2 Synthesis with timing constraints

The circuit area and system delay are two major design criteria. In most applications, we cannot find a design or an implementation that is optimized for both criteria. A faster circuit normally is more complex and needs more silicon real estate, and a smaller circuit normally has to sacrifice some performance. For the same application, there frequently exist multiple implementations that exhibit different area–delay characteristics. A typical area–delay curve is shown in Figure 6.18, in which each point is a possible implementation. Of course, the trade-off can be achieved only in a limited range. We cannot reduce the area or increase the performance indefinitely.

Multilevel logic synthesis is quite flexible, and it is possible to add additional gates to achieve shorter delay. An example is shown in Figure 6.19. The circuit performs three xor operations. The diagram in Figure 6.19(a) is the initial design, which is optimized for area. The critical path is from `a(0)` or `a(1)` to `y(3)`, and the system delay is three times the delay of an xor gate. The diagram in Figure 6.19(b) is the revised circuit. It shortens the

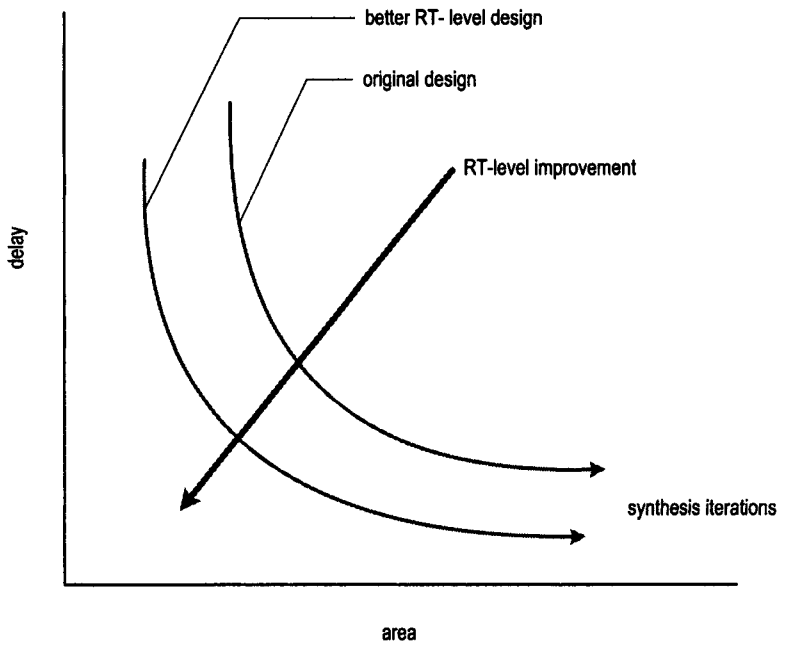


Figure 6.20 Synthesis iterations and the impact of RT-level change.

critical path by adding an extra xor gate, and the system delay is reduced to twice the delay of an xor gate.

The synthesis procedure discussed in Section 6.4 focuses on minimizing the circuit area. A combinational system is normally part of a larger system. To meet a certain performance goal, we sometimes have to add a specific timing constraint for synthesis. As we discussed earlier, it is impossible to synthesize a circuit with an exact propagation delay. Instead, the timing constraint is specified in terms of maximal allowable propagation delay. Since the system delay depends only on the delay of the critical path, it is not wise to blindly optimize all paths. Synthesis with a timing constraint utilizes an iterative procedure. First, the minimal-area implementation is obtained from regular synthesis. The implementation will be analyzed to determine the critical path and the system delay. If the delay exceeds the constraint, extra gates will be provided to speed up the critical path. The revised implementation will be analyzed again for the critical path (which is the second longest path in the original implementation) and checked to see whether the new system delay is within the constraint. The process may repeat several times until a satisfactory implementation is found. The iteration process in an area–delay space is shown in Figure 6.20.

The previous iteration procedure is done at the gate or cell level and thus is too tedious for human designers. However, it is possible to apply the procedure in at the RT level. A block diagram shows the basic routing structure and the locations of complex RT-level modules. Since the delays of the complex modules constitute the major portion of the system delay, we can identify the paths that contain these modules, estimate the rough delays of these paths, and determine the critical path accordingly. This kind of analysis helps us to explore various architectural alternatives and eventually to derive a more efficient design. Our understanding of the system and insight can lead to "global" optimization, and

it is normally much more effective than gate- or cell-level optimization done by synthesis software. The impact of an innovative RT-level architectural change on the area–delay space is shown in Figure 6.20.

6.5.3 Timing hazards

The propagation delay of a system is the time required to generate a valid, steady-state output value. *Timing hazards* are the fluctuations occurring during the transient period. In a digital system, many paths may lead to the same output port. Since each path's delay is different, signals may propagate to the output port at different times. Before the output port produces a steady-state value, it may fluctuate several times. The fluctuations are one or more short undesired pulses, known as *glitches*. We say that a circuit has timing hazards if it can produce glitches. The following subsections discuss the two types of hazards and how to deal with them.

Static hazards A *static hazard* is the condition that a circuit's output produces a glitch when it should remain at a steady value. It is further divided into static-1 hazard and static-0 hazard. A *static-1 hazard* occurs when a circuit's output produces a '0' glitch. An example is shown in Figure 6.21. The Karnaugh map of a function and its implementation are shown in Figure 6.21(a). The corresponding Boolean function is

$$sh = a \cdot b' + b \cdot c$$

Assume that a and c are '1', and that b changes from '1' to '0'. Regular analysis, which is based on Boolean algebra and deals with steady-state value, predicts that the output should be '1' all the time. However, if we consider transient behavior, there are two converging paths with different delays. Assume that the delay of inverter is T_{not} and the delay of the and gate and or gate is T_{and} and the wire delays are 0. The timing diagram and the sequence of events are shown in Figure 6.21(b). An unwanted '0' glitch of width T_{not} occurs at the output because the signal in the bottom path propagates faster than that in the top path.

Similarly, a *static-0 hazard* is the condition that a circuit's output produces a '1' glitch when Boolean algebra analysis predicts that the output should be a steady '0'.

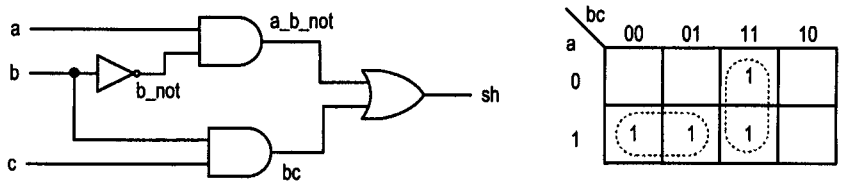
Dynamic hazards A *dynamic hazard* is the condition that a circuit's output produces a glitch when it changes from '1' to '0' or '0' to '1'. An example of a circuit with a dynamic hazard is shown in Figure 6.22(a). Assume that a, c and d are '1' and that b changes from '1' to '0'. The timing diagram in Figure 6.22(b) shows that there is a '1' glitch when the dh output changes from '0' to '1'. The glitch is due to the different propagation delays of the converging paths.

Dealing with hazards There are some techniques to eliminate hazards caused by a single input change. For example, we can add a redundant product term to eliminate the previous static hazard:

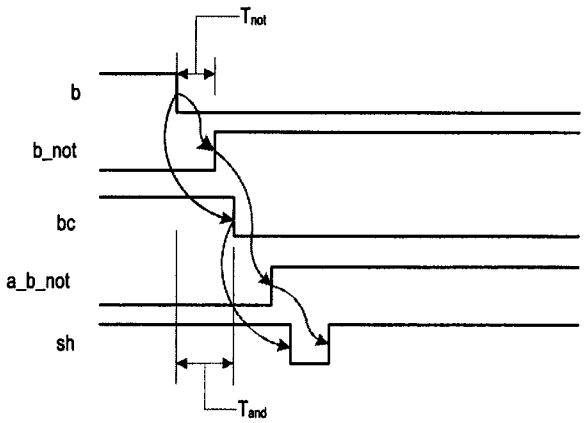
$$sh = a \cdot b' + b \cdot c + a \cdot c$$

The revised Karnaugh map and circuit are shown in Figure 6.21(c). Although deriving a hazard-free circuit is possible, this approach is problematic if the design is later processed by synthesis software. The problems are discussed in detail in the next section.

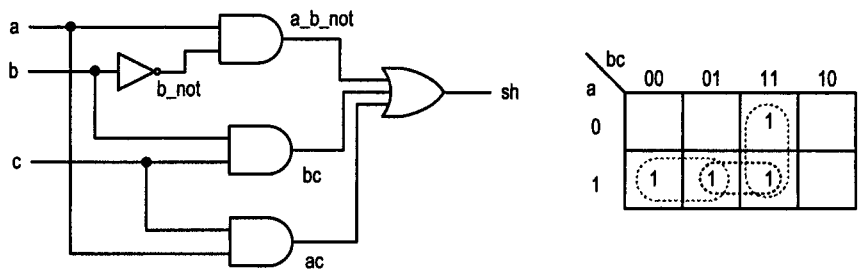
In a real-world application, the hazard situation will become even more complicated because of the possibility of multiple input signal transitions. If the inputs of a combinational



(a) Karnaugh map and schematic

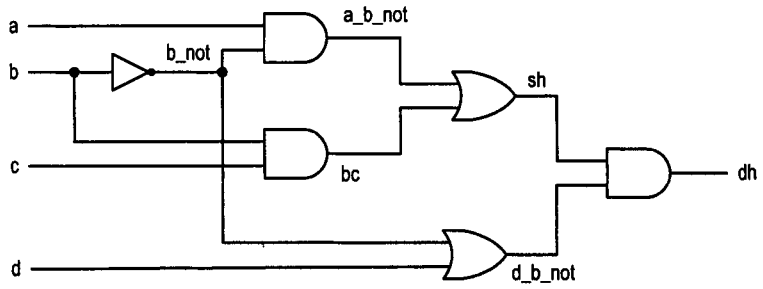


(b) Timing diagram

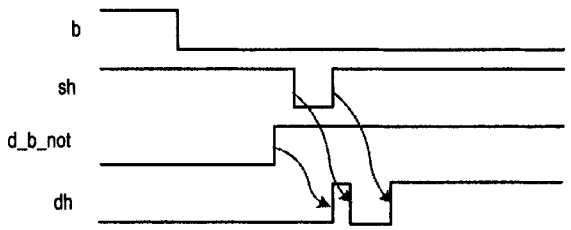


(c) Revised Karnaugh map and schematic to eliminate hazards

Figure 6.21 Static hazards example.



(a) Schematic



(b) Timing diagram

Figure 6.22 Dynamic hazards example.

circuit are connected to the outputs of an edge-triggered register, the register's outputs may change almost simultaneously at the transition edge of the clock signal. For example, when a 4-bit counter circulates from "1111" to "0000", four input bits change almost simultaneously. Multiple changes will activate several paths at the same time and frequently lead to glitches in an output signal. Unless we utilize a specialized counter, which is normally not practical, it is impossible to eliminate hazards.

Since there is no easy way to eliminate hazards, we have to live with them. In a combinational circuit, the most effective way to handle hazards is to ignore the output during the transient period. Recall that the propagation delay is the time for an input signal to propagate through the longest path in a system. If there is a glitch, it will occur within this period of time. After that, the output will always be a valid, steady-state value. As long as we know when to examine the output, the existence of glitches does not matter. This "wait until the output is stabilized" idea is one of the motivations behind the *synchronous design methodology*, in which a clock signal "samples" input signals at the proper time and stores the values in a register. The synchronous design methodology is elaborated in Chapter 8.

6.5.4 Delay-sensitive design and its dangers

In a digital system, most theoretical studies and design methodologies are based on steady-state analysis. Boolean algebra, the theoretical foundation of digital logic, conveys no time-domain information. When we use Boolean algebra to describe a digital circuit, we actually implicitly describe its steady-state behavior. Modeling and analyzing the transient behavior can be very hard, and most of the time we choose not to deal with it directly. Instead, we determine when the transient period ends and ignore the responses within the period. This approach is embedded in the concept of system delay, which specifies the time needed to reach the steady state in the worst-case scenario. Most design methodologies

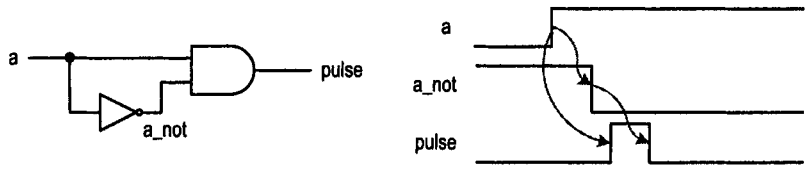


Figure 6.23 Delay-sensitive edge detection circuit.

and synthesis algorithms, such as time-constrained optimization, are based on system delay rather than the exact transient behavior.

In a few circumstances, we need to consider the transient behavior to understand a circuit's function and operation. We use the term *delay-sensitive design* to describe this type of circuit.

One example is the hazard elimination circuit in Figure 6.21. If we examine only the steady-state behavior, Boolean algebra shows that the $a \cdot c$ term does not serve any useful purpose and that the $a \cdot b' + b \cdot c + a \cdot c$ and $a \cdot b' + b \cdot c$ expressions are equivalent. The circuit is meaningful only if the transient behavior is considered.

One old, commonly used delay-sensitive design trick is to use cascading gates to generate a delay. An example is shown in Figure 6.23. The purpose of this circuit is to generate a short pulse when the input a switches from '0' to '1'. The inverter introduces a small delay and causes a monetary '1' pulse, as shown on the timing diagram. If we use steady-state analysis, the $a \cdot a'$ expression can be reduced to '0', and the circuit becomes a wire connected to ground. Again, this circuit makes sense only if we consider its transient behavior.

Although a delay-sensitive design can be useful in a few special situations, we should avoid using VHDL description and synthesis software to construct this kind of circuit. Transformation and optimization algorithms used in synthesis software are based on the model of steady-state value and propagation delay, and cannot interpret or process transient-related information.

Deriving VHDL code for a delay-sensitive circuit is not very difficult. For example, we can revise the VHDL code from

```
sh <= (a and (not b)) or (b and c);
```

to

```
sh <= (a and (not b)) or (b and c) or (a and c);
```

to describe the hazard-free circuit in Figure 6.21, and can use the statement

```
pulse <= a and (not a);
```

to describe the pulse generation circuit in Figure 6.23. However, it is unlikely that the desired effect can be preserved during the synthesis process. The potential complications are as follows:

- During logic synthesis, the logic expressions will be rearranged and optimized. Redundant product terms, if they exist, will be removed during the optimization process. It is unlikely that the original expression can be preserved.
- If we assume that the logic expression remains unchanged after logic synthesis, the netlist may be converted to other cells during technology mapping. Again, the original logic expression will be altered.

- If we assume that the original logic expression survives after technology mapping, wire delays will be changed after the placement and routing process. The change will alter the delay of the path and may invalidate the previous analysis.
- If we assume that the circuit is synthesized according to the specification, the design may hinder other steps in the verification and testing process. For example, the redundant product term used in the logic expression will complicate the test vector generation or even make the circuit untestable.

In summary, VHDL-based synthesis is not feasible for delay-sensitive design. If this kind of circuit is really needed, as in an asynchronous sequential circuit, we should construct the circuit manually using cells from the target device library. We may even need to manually perform the placement and routing to ensure that wire delay is within a tolerable range. Since our focus is on RT-level HDL synthesis, we will not discuss this approach in the remainder of the book.

6.6 SYNTHESIS GUIDELINES

- Be aware of the theoretical limitation of synthesis software.
- Be aware of the hardware complexity of different VHDL operators.
- Isolate tri-state buffers from other logic and code them in a separate segment.
- Unless there is a compelling reason, use a multiplexer instead of an internal tri-state bus.
- Avoid using the `'-'` value of the `std_logic` data type as an input value.
- In RT-level description, there is no effective way to eliminate glitches from a combinational circuit. We should deal with the glitches rather than attempting to derive a glitch-free combinational circuit.
- Do not use delay-sensitive design in RT-level description.

6.7 BIBLIOGRAPHIC NOTES

Synthesis is a complicated process and involves many difficult computation problems. The texts, *Synthesis and Optimization of Digital Circuits* by G. De Micheli, and *Logic Synthesis* by S. Devadas et al., provide comprehensive coverage of the theoretical foundations and relevant algorithms.

Because most software vendors do not allow users to publish benchmark information, there is very little documentation on the "behavior" of synthesis tools. The article, *Visualizing the Behavior of Logic Synthesis Algorithms* of *SNUG* (Synopsys Users Group Conference) 1998, by H. A. Landman, presents an interesting study of the relationship between the circuit area and timing constraints.

Problems

6.1 Determine the order (big-O) of the following functions:

(a) 1.5

(b) $2^n + 10^3 n^2$