CHAPTER 1

INTRODUCTION TO DIGITAL SYSTEM DESIGN

Developing and producing a digital system is a complicated process and involves many tasks. The design and synthesis of a register transfer level circuit, which is the focus of this book, is only one of the tasks. In this chapter, we present an overview of device technologies, system representation, development flow and software tools. This helps us to better understand the role of the design and synthesis task in the overall development and production process.

1.1 INTRODUCTION

Digital hardware has experienced drastic expansion and improvement in the past 40 years. Since its introduction, the number of transistors in a single chip has grown exponentially, and a silicon chip now routinely contains hundreds of thousands or even hundreds of millions of transistors. In the past, the major applications of digital hardware were computational systems. However, as the chip became smaller, faster, cheaper and more capable, many electronic, control, communication and even mechanical systems have been "digitized" internally, using digital circuits to store, process and transmit information.

As applications become larger and more complex, the task of designing digital circuits becomes more difficult. The best way to handle the complexity is to view the circuit at a more abstract level and utilize software tools to derive the low-level implementation. This approach shields us from the tedious details and allows us to concentrate and explore high-level design alternatives. Although software tools can automate certain tasks, they are capable of performing only limited transformation and optimization. They cannot, and

RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. By Pong P. Chu

will not, do the design or convert a poor design to a good one. The ultimate efficiency still comes from human ingenuity and experience. The goal of this book is to show how to systematically develop an efficient, portable design description that is both abstract, yet detailed enough for effective software synthesis.

Developing and producing a digital circuit is a complicated process, and the design and synthesis are only two of the tasks. We should be aware of the "big picture" so that the design and synthesis can be efficiently integrated into the overall development and production process. The following sections provide an overview of device technologies, system representation, abstraction, development flow, and the use and limitations of software tools.

1.2 DEVICE TECHNOLOGIES

If we want to build a custom digital system, there are varieties of device technologies to choose, from off-the-shelf simple field-programmable components to full-custom devices that tailor the application down to the transistor level. There is no single best technology, and we have to consider the trade-offs among various factors, including chip area, speed, power and cost.

1.2.1 Fabrication of an IC

To better understand the differences between the device technologies, it is helpful to have a basic idea of the fabrication process of an integrated circuit (IC). An IC is made from layers of doped silicon, polysilicon, metal and silicon dioxide, built on top of one another, on a thin silicon wafer. Some of these layers form transistors, and others form planes of connection wires.

The basic step in IC fabrication is to construct a layer with a customized pattern, a process known as *lithography*. The pattern is defined by a *mask*. Today's IC device technology typically consists of 10 to 15 layers, and thus the lithography process has to be repeated 10 to 15 times during the fabrication of an IC, each time with a unique mask.

One important aspect of a device technology is the silicon area used by a circuit. It is expressed by the length of a smallest transistor that can be fabricated, usually measured in *microns* (a millionth of a meter). As the device fabrication process improved, the transistor size continued to shrink and now approaches a tenth of a micron.

1.2.2 Classification of device technologies

There is an array of device technologies that can be used to construct a custom digital circuit. One major characteristic of a technology is how the customization is done. In certain technologies, all the layers of a device are predetermined, and thus the device can be prefabricated and manufactured as a standard off-the-shelf part. The customization of a circuit can be performed "in the field," normally by downloading a connection pattern to the device's internal memory or by "burning the internal silicon fuses." On the other hand, some device technologies need one or more layers to be customized for a particular application. The customization involves the creation of tailored masks and fabrication of the patterned layers. This process is expensive and complex and can only be done in a fabrication plant (known as a *foundry* or a *fab*). Thus, whether a device needed to be fabricated in a fab is the most important characteristic of a technology. In this book, we use

the term *application-specific IC (ASIC)* to represent device technologies that require a fab to do customization.

With an understanding of the difference between ASIC and non-ASIC, we can divide the device technologies further into the following types:

- Full-custom ASIC
- Standard-cell ASIC
- Gate array ASIC
- Complex field-programmable logic device
- Simple field-programmable logic device
- Off-the-shelf small- and medium-scaled IC (SSI/MSI) components

Full-custom ASIC In *full-custom ASIC* technology, all aspects of a digital circuit are tailored for one particular application. We have complete control of the circuit and can even craft the layout of a transistor to meet special area or performance needs. The resulting circuit is fully optimized and has the best possible performance. Unfortunately, designing a circuit at the transistor level is extremely complex and involved, and is only feasible for a small circuit. It is not practical to use this approach to design a complete system, which now may contain tens and even hundreds of millions of transistors. The major application of full-custom ASIC technology is to design the basic logic components that can be used as building blocks of a larger system. Another application is to design special-purpose "bit-slice" typed circuits, such as a 1-bit memory or 1-bit adder. These circuits have a regular structure and are constructed through a cascade of identical slices. To obtain optimal performance, full-custom ASIC technology is frequently used to design a single slice. The slice is then replicated a number of times to form a complete circuit.

The layouts of a full-custom ASIC chip are tailored to a particular application. All layers are different and a mask is required for every layer. During fabrication, all layers have to be custom constructed, and nothing can be done in advance.

Standard-cell ASIC In *standard-cell ASIC* (also simply known as *standard-cell*) technology, a circuit is constructed by using a set of predefined logic components, known as *standard cells*. These cells are predesigned and their layouts are validated and tested. Standard-cell ASIC technology allows us to work at the gate level rather than at the transistor level and thus greatly simplifies the design process. The device manufacturer usually provides a library of standard cells as the basic building blocks. The library normally consists of basic logic gates, simple combinational components, such as an and-or-inverter, 2-to-1 multiplexer and 1-bit full adder, and basic memory elements, such as a D-type latch and D-type flip-flop. Some libraries may also contain more sophisticated function blocks, such as an adder, barrel shifter and random access memory (RAM).

In standard-cell technology, a circuit is made of cells. The types of cells and the interconnection depend on the individual application. Whereas the layout of a cell is predetermined, the layout of the complete circuit is unique for a particular application and nothing can be constructed in advance. Thus, fabrication of a standard-cell chip is identical to that of a full-custom ASIC chip, and all layers have to be custom constructed.

Gate array ASIC In *gate array* ASIC (also simply known as *gate array*) technology, a circuit is built from an array of predefined cells. Unlike standard-cell technology, a gate array chip consists of only one type of cell, known as a *base cell*. The base cell is fairly simple, resembling a logic gate. Base cells are prearranged and placed in fixed positions, aligned as a one- or two-dimensional array. Since the location and type are predetermined,

the base cells can be prefabricated. The customization of a circuit is done by specifying the interconnect between these cells. A gate array vendor also provides a library of predesigned components, known as *macro cells*, which are built from base cells. The macro cells have a predefined interconnect and provide the designer with more sophisticated logic blocks.

Compared to standard-cell technology, the fabrication of a gate array device is much simpler, due to its fixed array structure. Since the array is common to all applications, the cell (and transistors) can be fabricated in advance. During construction of a chip, only the masks of metal layers, which specify the interconnect, are unique for an application and therefore must be customized. This reduces the number of custom layers from 10 to 15 layers to 3 to 5 layers and simplifies the fabrication process significantly.

Complex field-programmable device We now examine several non-ASIC technologies. The most versatile non-ASIC technology is the complex field-programmable device. In this technology, a device consists of an array of generic logic cells and general interconnect structure. Although the logic cells and interconnect structure are prefabricated, both are *programmable*. The programmability is obtained by utilizing semiconductor "fuses" or "switches," which can be set as open- or short-circuit. The customization is done by configuring the device with a specific fuse pattern. This process can be accomplished by a simple, inexpensive device programmer, normally constructed as an add-on card or an adaptor cable of a PC. Since the customization is done "in the field" rather than "in a fab," this technology is known as *field programmable*. (In contrast, ASIC technologies are "programmed" via one or more tailored masks and thus are *mask programmable*.)

The basic structures of gate array ASICs and complex field-programmable devices are somewhat similar. However, the interconnect structure of field-programmable devices is predetermined and thus imposes more constraints on signal routing. To reduce the amount of connection, more functionality is built into the logic cells of a field-programmable device, making a logic cell much more complex than a base cell or a standard cell of ASIC. According to the complexity and structure of logic cells, complex field-programmable devices can be divided roughly into two broad categories: *complex programmable logic device* (*CPLD*) and *field programmable gate array* (*FPGA*).

The logic cell of a CPLD device is more sophisticated, normally consisting of a D-type flip-flop and a PAL-like unit with configurable product terms. The interconnect structure of a CPLD device tends to be more centralized, with few groups of concentrated routing lines. On the other hand, the logic cell of an FPGA device is usually smaller, typically including a D-type flip-flop and a small look-up table or a set of multiplexers. The interconnect structure between the cells tends to be distributed and more flexible. Because of its distributive nature, FPGA is better suited for large, high-capacity complex field-programmable devices.

Simple field-programmable device Simple field-programmable logic devices, as the name indicates, are programmable devices with simpler internal structure. Historically, these devices are generically called *programmable logic devices* (*PLDs*). We add the word *simple* to distinguish them from FPGA and CPLD devices. Simple field-programmable devices are normally constructed as a two-level array, with an *and plane* and an *or plane*. The interconnect of one or both planes can be programmed to perform a logic function expressed in sum-of-product format. The devices include *programmable read only memory* (*PROM*), in which the or plane can be programmed; *programmable array logic* (*PAL*), in which the and plane can be programmed; and *programmable logic array* (*PLA*), in which both planes can be programmed.

Unlike FPGA and CPLD devices, simple field-programmable logic devices do not have a general interconnect structure, and thus their functionality is severely limited. They are

gradually being phased out. ROM, PAL and PLA are now used as internal components of an ASIC or CPLD device rather than as an individual chip.

Off-the-shelf SSI/MSI components Before the emergence of field-programmable devices, the only alternative to ASIC was to utilize the prefabricated off-the-shelf SSI/MSI components. These components are small parts with fixed, limited functionality. One example is the 7400 series transistor transistor logic (TTL) family, which contains more than 100 parts, ranging from simple nand gates to a 4-bit arithmetic unit. A custom system can be designed by a bottom-up approach, building the circuit gradually from the small existing parts. A tailored printed circuit board is needed for each application. The major disadvantage of this approach is that the most resources (power, board area and manufacturing cost) are consumed by the "package" but not by the "silicon," which performs the actual computation. Furthermore, none of today's synthesis software can utilize off-the-shelf SSI/MSI components, and thus automation is virtually impossible. As the programmable devices become more capable and less expensive, designing a large custom circuit using SSI/MSI components is no longer a feasible option and should not be considered.

Summary We have reviewed six device technologies used to implement custom digital systems. Among them, off-the-shelf SSI/MSI components and simple programmable devices are gradually being phased out and full-custom ASIC is feasible only for a small, specialized circuit. Thus, for a large digital system, there are only three viable device technologies: standard-cell ASIC, gate array ASIC and CPLD/FPGA. In the following subsection, we examine the trade-offs among these technologies.

1.2.3 Comparison of technologies

Once deciding to develop custom hardware for an application, we need to choose from the three device technologies. The major criteria for selection are *area*, *speed*, *power* and *cost*. The first three involve the technical aspects of a circuit. Cost concerns the expenditure associated with the design and production of the circuit as well as the potential lost profits. Each technology has its strengths and weaknesses, and the "best" technology depends on the needs of a particular application.

Area Chip area (or size) corresponds to the required silicon real estate to implement a particular application. A smaller chip needs fewer resources, simplifies the testing and provides better yield. The chip size depends on the architecture of the circuit and the device technology. The same function can frequently be realized by different architectures, with different areas and speeds. For example, an addition circuit can be realized by a ripple adder (simple but slow), a parallel adder (complex but fast) or a carry-look-ahead adder (somewhere in-between). Once the architecture of a circuit is determined, the area depends on the device technology. In standard-cell technology, the cells and interconnects are customized to this particular application and no silicon is wasted in irrelevant functionality. Thus, the resulting chip is fully optimized and the area is minimal. In gate array technology, the circuit has to be constructed by predefined, prearranged base cells. Since functionality and the placement of the base cells are not tailored to a specific application, silicon use is not optimal. The area of the resulting circuit is normally larger than that of a standard-cell chip. In FPGA technology, a significant portion of the silicon is dedicated to achieving programmability, which introduces a large overhead. Furthermore, the functionalities of logic cells and the interconnect are fixed in advance and it is unlikely that an application

can be an exact match for the predetermined structure. A certain percentage of the capacity will be left unutilized. Because of the overhead and relatively low utilization, the area of the resulting FPGA chip is much larger than that of an ASIC chip.

Due to the drastic difference between the device fabrication process and the diversity of applications, it is difficulty to determine the exact silicon areas in three technologies. However, it is important to recognize that the difference between standard-cell and gate array technologies is much smaller than that of FPGA and ASIC. In general, a gate array chip may need 20% to 100% larger silicon area than that of a standard-cell chip, but an FPGA chip frequently requires two to five times the area of an ASIC chip.

Speed The speed of a digital circuit corresponds to the time required to perform a function, frequently represented by the worst-case propagation delay between input and output signals. A faster circuit is always desirable and is essential for computation-intensive applications. At the architecture level, faster operation can be achieved by using a more sophisticated design, which requires a larger area. However, if the identical architecture is used, a chip with a larger area is normally slower, due to its large parasitic capacitance. Since a standard-cell chip has tailored interconnect and utilizes a minimal amount of silicon area, it has the smallest propagation delay and best speed. On the other hand, an FPGA chip has the worst propagation delay. In addition to its large size, the programmable interconnect has a relatively large resistance and capacitance, which introduces even more delay. As with chip area, the speed difference between standard-cell and gate array technologies is much less significant than that between FPGA and ASIC.

Power Power concerns the energy consumed by a part. In certain applications, such as battery-operated handheld equipment, a low power circuit is of primary importance. At the architecture level, a system can be redesigned to reduce the use of power. If the identical architecture is used, a smaller chip, which consists of fewer transistors, usually consumes less power. Thus, a standard-cell chip consumes the least amount of power and an FPGA chip uses the most power.

Standard-cell technology is clearly the best choice from a technical perspective. A chip constructed using standard-cell ASIC is small and fast, and consumes less power. This should not come as a surprise since the chip is highly optimized and wastes no resources on unnecessary overhead. The price associated with customization is the complexity. Designing and fabricating a standard-cell chip is more involved and time consuming than for the other two technologies.

Cost The design of a custom digital circuit is seldom a goal in itself. It is an economic activity, and the cost is an important, if not the deciding, factor. We consider three major expenses: production cost, development cost, and time-to-market cost.

Production cost is the expense to produce a single unit. It includes two segments: non-recurring engineering (NRE) cost and part cost. *NRE cost* (C_{nre}) is the expense that occurs only once (and thus is not recurring) during the production process, regardless of the number of units sold. Thus, it is on a "per design" basis. *Part cost* (C_{per_part}), on the other hand, is on a "per unit" basis, covering the expense required for each individual unit, such as the expense of materials, assembly and manufacturing. Note that the NRE cost is shared by all the units and that the share of each part becomes smaller as the volume increases. The per unit production cost (C_{per_unit}) can be expressed as

$$C_{per\_unit} = C_{per\_part} + \frac{C_{nre}}{\text{units produced}}$$

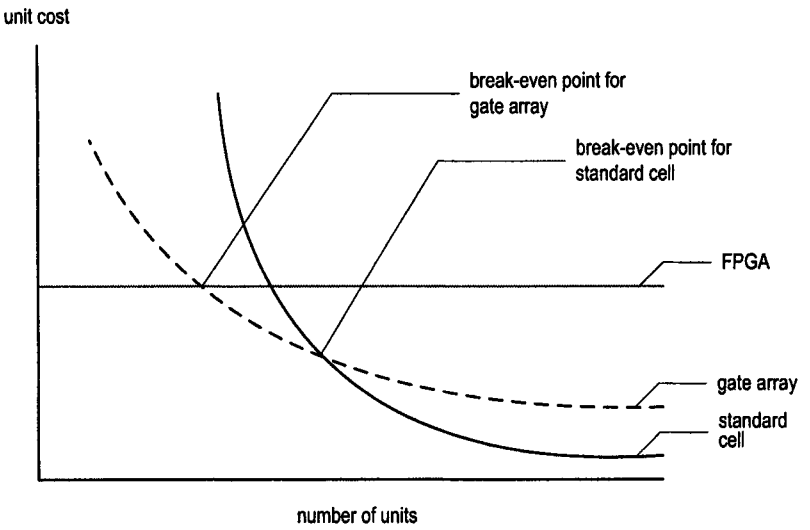


Figure 1.1 Comparison of per unit cost.

The NRE cost of a custom ASIC chip includes the creation of the tailored masks, the development of tests and the fabrication of initial sample chips. The charge is high and can range from several hundred thousand dollars to several million dollars or more. A major factor in the NRE cost is the number of custom masks needed. A standard-cell chip may need 15 or more tailored masks and thus is much more expensive than a gate array chip, which needs only three to five tailored metal layers. On the contrary, an FPGA-based design needs only an inexpensive device programmer to do customization. The NRE cost of creating a mask is negligible and can be considered as zero.

The part cost of an ASIC chip is smaller than that of an FPGA chip since the ASIC chip requires less silicon real estate and has better yield. By the same token, the part cost of a standard-cell chip is smaller than that of a gate array chip since the standard-cell chip is further optimized. If we consider both part cost and NRE cost, the per unit production cost depends on the volume of units, as shown by the previous equation. The volume versus per unit cost plots of three technologies is shown in Figure 1.1. As the volume increases, first the gate array and then the standard-cell technologies, become cost-effective. The intersections of the curves are the break-even points for the FPGA and gate array technologies, and for the gate array and standard-cell technologies.

The second major expense is the *development cost*. The process of transforming an idea to a custom circuit is by no means a simple task. The expense involved in this process is the development cost. It includes the compensation for engineering time as well as the expense of the computing facility and software tools. Although the synthesis procedure is somewhat similar for all device technologies, developing ASIC requires more effort, including physical design, placement and routing, verification and testing. Since the development process is more complex for ASIC, the development cost of an ASIC chip is much higher than that of an FPGA chip. Similarly, due to the high-level optimization, the development cost for a standard-cell chip is much higher than that for a gate array chip.

The third major expense is the *time-to-market cost*. It is actually not a cost, but the lost revenue. In many applications, such as PC peripherals, the life cycle of a product is

Table 1.1 Comparison of device technologies

	FPGA	Gate array	Standard cell
Tailored masks	0	3 to 5	15 or more
Area			best (smallest)
Speed			best (fastest)
Power			best (minimal)
NRE cost	best (smallest)		
Per part cost			best (smallest)
Development cost	best (easiest)		
Time to market	best (shortest)		
Per unit cost		depends on volume	

very short. Eighteen months, the time required to double the chip density, is sometimes considered as the life cycle of the product. Thus, it is very important to introduce the product in a timely manner, and a shipping delay can mean a significant loss in sales. The standard-cell technology requires the most lead time to validate, test and manufacture, ranging from a few months to a year. The gate array technology requires less lead time, from a few weeks to a few months. For FPGA technology, customization involves the programming of a prefabricated chip and can be done in a few minutes.

Summary The major characteristics of the three device technologies are summarized in Table 1.1. In general, the trade-off is between the optimal use of hardware resources (in terms of chip area, speed and power) and the ease of design (in terms of NRE cost, development cost and manufacturing lead time).

The choice of technology is not necessarily mutual exclusive. For example, ASIC and FPGA developments can be done in parallel to get the benefits of both technologies. The FPGA devices are used as prototypes and in initial shipments to cut the manufacturing lead time. When the ASIC devices become available later, they are used for volume production to reduce cost.

1.3 SYSTEM REPRESENTATION

A large digital system is quite complex. During the development and production process, each task may require a specific kind of information about the system, ranging from system specification to physical component layout. The same system is frequently described in different ways and is examined from different perspectives. We call these perspectives the *representations* or *views* of a system. There are three views:

- Behavioral view
- Structural view
- Physical view

A *behavioral view* describes the functionality (i.e., "behavior") of a system. It treats the system as a black box and ignores its internal implementation. The view focuses on the relationship between the input and output signals, defining the output response when a particular set of input values is applied. The description of a behavioral view is seldom unique. Normally, there are a wide variety of ways to specify the same input–output characteristics.

A *structural view* describes the internal implementation (i.e., structure) of a system. The description is done by explicitly specifying what components are used and how these components are connected. It is more or less the schematic or the diagram of a system. In computer software, we use the term *net* to represent a set of wires that are connected to the same node, and use the term *netlist*, which is a collection of nets, to represent the schematic.

A *physical view* describes the physical characteristics of the system and adds additional information to the structural view. It specifies the physical sizes of components, the physical locations of the components on a board or a silicon wafer, and the physical path of each connection line. An example of a physical view is the printed circuit board layout of a system.

Clearly, the physical view of a system provides the most detailed information. It is the final specification for the system fabrication. On the other hand, the behavioral view imposes fewest constraints and is the most abstract form of description.

1.4 LEVELS OF ABSTRACTION

As chip density reaches hundreds of millions of transistors, it is impossible for a human being, or even a computer, to process this amount of data directly. A key method of managing complexity is to describe a system in several levels of abstraction. An *abstraction* is a simplified model of the system, showing only the selected features and ignoring the associated details. The purpose of an abstraction is to reduce the amount of data to a manageable level so that only the critical information is presented. A high-level abstraction is focused and contains only the most vital data. On the other hand, a low-level abstraction is more detailed and takes account of previously ignored information. Although it is more complex, the low-level abstraction model is more accurate and is closer to the real circuit. In the development process, we normally start with a high-level abstraction and concentrate on the most vital characteristics. As the system is better understood, we then include more details and develop a lower-level abstraction.

Four levels of abstraction are considered in digital system development:

- Transistor level
- Gate level
- Register transfer (RT) level
- Processor level

The division of these levels is based primarily on the size of basic building blocks, which are the transistors, logic gates, function modules and processors respectively.

The level of abstraction and the view are two independent dimensions of a system, and each level has its own views. The levels of abstraction and views can be combined in a *Y-chart*, which is shown in Figure 1.2. In this chart, each axis represents a view and the levels of abstraction increase from the center to the outside.

The following subsections discuss the four levels of abstraction. In the discussion, we examine the five main characteristics at each level of abstraction:

- Basic building blocks
- Signal representation
- Time representation
- Behavioral representation
- Physical representation

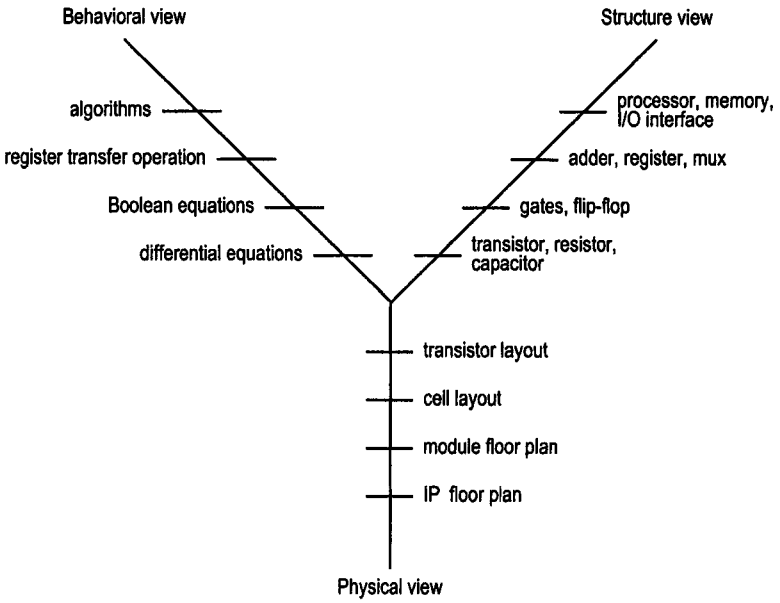


Figure 1.2 Y-chart.

Basic building blocks are the most commonly used parts at the level. These parts are the components used in the structure view. Behavioral and physical representations are the descriptions for the behavioral and physical views.

Signal and timing representations concern how to express a signal's value and how the value changes over time. While the physical signal remains the same, the interpretation of its value and timing is different at each abstraction level. As we expect, more detailed information will be provided at lower levels.

1.4.1 Transistor-level abstraction

The lowest level of abstraction is the transistor level. At this level, the basic building blocks are transistors, resistors, capacitors and so on. The behavior description is usually done by a set of differential equations or even by some type of current–voltage diagram. Analog system simulation software, such as SPICE, can be used to obtain the desired input–output characteristics.

At the transistor level, a digital circuit is treated as an analog system, in which signals are time-varying and can take on any value of a continuous range. For example, the output response of an inverter is plotted at the top of Figure 1.3.

The physical description of the transistor level comprises the detailed layout of components and their interconnections. It essentially defines the masks of various layers and is the final result of the design process.

1.4.2 Gate-level abstraction

The next level of abstraction is the gate level. Typical building blocks include simple logic gates, such as and, or, xor and 1-bit 2-to-1 multiplexer, and basic memory elements, such

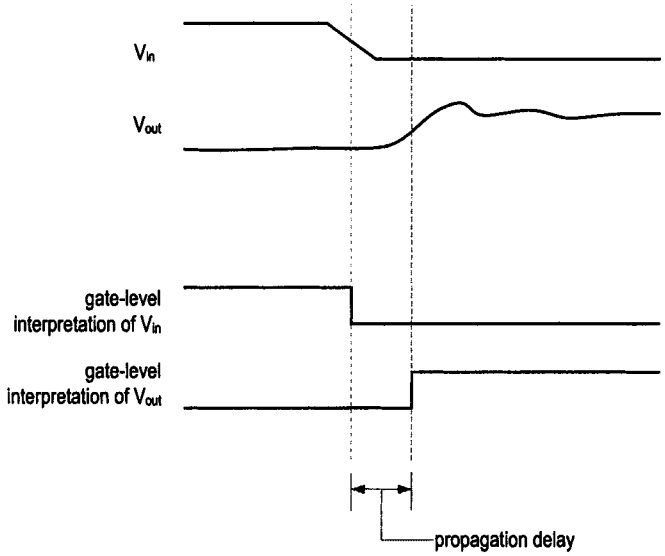


Figure 1.3 Timing characteristic of an inverter.

as latch and flip-flop. Instead of using continuous values, we consider only whether a signal's voltage is above or below a threshold, which is interpreted as logic 1 or logic 0 respectively. Since there are only two values, the input–output behavior is described by Boolean equations. The abstraction essentially converts a continuous system to a discrete system and discards the complex differential equations. Note that logic 0 and logic 1 are only our interpretation, depending on whether a signal's voltage level exceeds a predefined threshold, and the real signal is still the same continuous signal.

The timing information is also simplified at this level. A single discrete number, known as the *propagation delay*, which is defined as the time interval for a system to obtain a stable output response, is used to specify the timing of a gate. The plot at the bottom of Figure 1.3 shows a gate-level interpretation of the corresponding transistor-level signal.

The physical description at this level is the placement of the gates (or cells) and the routing of the interconnection wires.

So far, we use the term *area* or *size* to describe the silicon real estate used to construct a circuit. Alternatively, we can count the number of gates in this circuit (known as *gate count*) and make the measurement independent of the underlying device technology. The area of the two-input nand gate is used as the base unit since it is frequently the simplest physical logic circuit. Instead of using the physical area, we express the size or the complexity of a circuit in terms of the number of equivalent nand gates in that particular device technology.

1.4.3 Register-transfer-level (RT-level) abstraction

At the register-transfer (RT) level, the basic building blocks are modules constructed from simple gates. They include functional units, such as adders and comparators, storage components, such as registers, and data routing components, such as multiplexers. A reasonable name for this level would be *module-level* abstraction. However, the term *register transfer* is normally used in digital design and we follow the general convention.

Register transfer is a somewhat confusing term. It is used in two contexts. Originally, the term was used to describe a design methodology in which the system operation is specified by how the data are manipulated and transferred between storage registers. Since the main components used in the register transfer methodology are the intermediate-size modules, the term has been borrowed to describe module-level abstraction. As the title indicates, the coverage and discussion of this book focus on the RT level. We use the term *RT level* for module-level abstraction and *RT methodology* for the specific design methodology. RT methodology is discussed in Chapters 11 and 12.

The data representation at the RT level becomes more abstract. Signals are frequently grouped together and interpreted as a special kind of data type, such as an unsigned integer or system state. The behavioral description at this level uses general expressions to specify the functional operation and data routing, and uses an extended finite state machine (FSM) to describe a system designed using RT methodology.

A major feature of the RT-level description is the use of a common *clock signal* in the storage components. The clock signal functions as a sampling and synchronizing pulse, putting data into the storage component at a particular point, normally the rising or falling edge of the clock signal. In a properly designed system, the clock period is long enough so that all data signals are stabilized within the clock period. Since the data signals are sampled only at the clock edge, the difference in propagation delays and glitches have no impact on the system operation. This allows us to consider timing in terms of number of clock cycles rather than by keeping track of all the propagation delays.

The physical layout at this level is known as the *floor plan*. It is helpful for us to find the slowest path between the storage components and to determine the clock period.

1.4.4 Processor-level abstraction

Processor-level abstraction is the highest level of abstraction. The basic building blocks at this level, frequently known as *intellectual properties* (*IPs*), include processors, memory modules, bus interfaces and so on. The behavioral description of a system is more like a program coded in a conventional programming language, including computation steps and communication processes. The signals are grouped and interpreted as various data types. Time measurement is expressed in terms of a computation step, which is composed of a set of operations defined between two successive synchronization points. A collection of computations may run concurrently in parallel hardware and exchange data through a predefined communication or bus protocol. The physical layout of a processor-level system is also known as the floor plan. Of course, the components used in a floor plan are much larger than those of an RT-level system.

Table 1.2 summarizes the main characteristics at each level. It lists the typical building blocks, signal representation, time representation, representative behavioral description and representative physical description.

1.5 DEVELOPMENT TASKS AND EDA SOFTWARE

Developing a custom digital circuit is essentially a refining and validating process. A system is gradually transformed from an abstract high-level description to final mask layouts. Along with each refinement, the system's function should be validated to ensure that the final product works correctly and meets the specification and performance goals. The major design tasks of developing a digital system are:

Table 1.2 Characteristics of each abstraction level

	Typical blocks	Signal representation	Time representation	Behavioral description	Physical description
Transistor	transistor, resistor	voltage	continuous function	differential equation	transistor layout
Gate	and, or, xor, flip-flop	logic 0 or 1	propagation delay	Boolean equation	cell layout
RT	adder, mux, register	integer, system state	clock tick	extended FSM	RT-level floor plan
Processor	processor, memory	abstract data type	event sequence	algorithm in C	IP-level floor plan

- Synthesis
- Physical design
- Verification
- Testing

1.5.1 Synthesis

Synthesis is a refinement process that realizes a description with components from the lower abstraction level. The original description can be in either a behavioral view or a structural view, and the resulting description is a structural view (i.e., netlist) in the lower abstraction level. In the Y-chart, the process either moves the system from behavioral view to structural view or moves it from a high-level abstraction to a low-level abstraction. Thus, synthesis either derives a structural implementation from a behavioral description or realizes an upper level description using finer components. As the synthesis process progresses, more details are added. The final result is a gate-level structural representation using the primitive cells from the chosen device technology. To make the process manageable, synthesis is usually divided into several smaller steps, each performing a specific transformation. The major steps are:

- High-level synthesis
- RT-level synthesis
- Gate-level synthesis
- Technology mapping

High-level synthesis transforms an algorithm into an RT-level description, which is specified explicitly in terms of register transfer operations. Due to the complexity of transformation, it can only be applied to relatively simple algorithms in a narrowly defined application domain.

RT-level synthesis analyzes an RT-level behavioral description and derives the structural implementation using RT-level components. It may also perform a limited degree of optimization to reduce the number of components.

Gate-level synthesis is similar to RT-level synthesis except that gate-level components are used in structural implementation. After the initial circuit is derived, two-level or multilevel

optimization is used to minimize the size of the circuit or to meet the timing constraint. In general, generic components are used in gate-level synthesis, and thus the synthesis process is independent of device technology.

Each device technology includes a set of predesigned primitive gate-level components, which can be cells of a standard-cell library or a generic logic cell of an FPGA device. To implement the gate-level circuit in a particular device technology, the generic components have to map into the cells of the chosen technology. The transforming process is known as *technology mapping*. It is the last step in synthesis, and clearly the process is technology dependent.

The synthesis procedure is discussed in detail in Section 6.4.

1.5.2 Physical design

Physical design includes two major parts. The first part is the refinement process between the structural and physical views, which derives a layout for a netlist. The second part involves the analysis and tuning of a circuit's electrical characteristics. The main tasks in physical design include floor planning, placement and routing and circuit extraction.

Floor planning derives layouts at the processor and RT levels. It partitions the system into large function blocks and places these blocks in proper locations to reduce future routing congestion or to achieve certain timing objectives. Furthermore, floor planning may also provide a global plan for the power and clock distribution schemes. *Placement and routing* derives a layout at the gate level. The layout involves the detailed placement of cells and the routing of interconnecting wires.

After the placement and routing are complete, the exact length and location of each interconnect are known, and the associated parasitic capacitance and resistance can be calculated. This process is known as *circuit extraction*. The extracted data are used to construct a resistance and capacitance network, which in turn is used to compute the propagation delays.

In addition to the foregoing tasks, the physical design also includes design rule checking, derivation of the power grid, derivation of the clock distribution network, estimation of power use and assurance of signal integrity.

1.5.3 Verification

Verification is the process of checking whether a design meets the specification and performance goals. It concerns the correctness of the initial design as well as the correctness of refinement processes during synthesis and physical design. Verification has two aspects: *functionality* and *performance*. *Functional verification* checks whether a system generates the desired output response. Performance is represented as certain *timing constraints*. *Timing verification* checks whether the response is generated within the given time constraint. Verification is done in different phases of the design and at different levels of abstraction.

Functional verification The design of a custom system usually begins with a high-level behavioral description. When it is first created, the primary concern is whether the design functions according to the specifications. We need to check its operation and compare its responses to those desired. Once the functionality of the initial design is verified, we can start the refinement process and gradually convert it to a gate-level structural description. In general, if the initial design does not depend on the internal propagation delay (i.e., is not *delay-sensitive*), the functionality should be maintained through the refinement processes.

In the ideal situation, the design should be "correct by construction" and require no further functional verification. In reality, subtle errors may be introduced in a refinement process, and thus functional verification is still performed after each process to ensure that the new, refined description works correctly.

Timing verification Timing verification checks whether a system meets its performance goals, which are normally expressed in terms of maximal propagation delay or minimal clock frequency. At the processor or RT level, the propagation delay of an input–output path can be calculated by identifying the components in the path and summating the individual delays. However, since these components will be further refined and synthesized, the information is just a rough estimation.

At the gate level, the propagation delay of a path is affected by the delays of the components as well as the interconnection wires. The wiring delay depends on the locations and the lengths of wires. Although they can be estimated during synthesis, the exact values can be obtained only after the placement and routing process. As the size of a transistor continues to shrink, the effect of a wiring delay becomes more dominant. This makes timing verification more difficult since accurate delay information is not available during the synthesis process.

Timing issues and propagation delay are discussed in more detail in Section 6.5.1.

Methods of verification The most commonly used verification method is *simulation*, which is the process of constructing a model of a system, executing the model with input test patterns in a computer, and examining and analyzing the output responses. The model can be an actual or a hypothetical circuit that incorporates functionality and timing information. Simulation is a versatile process that be applied at any level of abstraction, and in behavioral as well as structural views. Utilizing simulation allows us to examine a system's operation in a computer and to detect errors without actually constructing the system.

Simulation essentially provides a sequence of snapshots of system operation, defined by a set of input stimuli. However, there is no guarantee that the selected stimuli can exercise every part of the system and verify the correctness of the entire design. Whereas simulation can do spot checks and detect major design mistakes, it cannot guarantee the absence of errors.

Another limitation of simulation comes from its computation complexity. Hardware operation is concurrent and parallel in nature, and it is time consuming to model its operation in a computer, which performs computational steps sequentially. It becomes a serious problem when we want to simulate low-level models, which may consist of hundreds of thousands or even millions of components.

In addition to simulation, several other methods are used for verification, including timing analysis, formal verification and hardware emulation. *Timing analysis* focuses only on the timing aspects of a circuit. It analyzes the structure of a circuit, determines all possible input–output paths, calculates the propagation delays of these paths and determines the relevant timing parameters, such as worst-case propagation delay and maximal clock frequency. Simulation can provide the relevant timing information for the selected test patterns. However, since these test patterns do not always exercise the critical paths, timing analysis is needed to verify that the system meets the timing specifications.

Formal verification applies formal mathematical techniques to analyze a circuit and determine its property. A popular method in formal verification is *equivalence checking*, which compares two representations of a system and determines whether the two representations perform the same function. It is frequently applied in synthesis to verify that the functionality of a synthesized circuit is identical to the original one. Unlike simulation,

formal verification is based on rigorous mathematical reasoning and can ensure that the synthesis is completely error-free.

Hardware emulation physically constructs a prototyping circuit that mimics operation of the system. A common application is to construct an FPGA circuit to emulate a complex ASIC design. Although the FPGA-based system is normally larger and slower than the ASIC system, it is much faster than simulation and it can be physically interfaced with other circuits and studied in detail.

1.5.4 Testing

The meanings of verification and testing are somewhat similar in a dictionary sense. However, they are two very different tasks in digital system development. *Verification* is the process of determining whether a design meets the specification and performance goals. It concerns the correctness of the initial design as well as the refinement processes. On the other hand, *testing* is the process of detecting the physical defects of a die or a package that occurred during manufacturing. When a device is being tested, we already know that the design is correct and the purpose of testing is simply to ensure that this particular part was properly fabricated.

At first glance, testing appears to be easy. All we need to do is simply to apply all possible input combinations and check the output responses. However, because of the large number of input combinations, this approach is not feasible. Instead, we have to utilize special algorithms to obtain a small set of test patterns. This process is known as *test pattern generation*.

For a small circuit, we can develop the testing procedure after completing the initial design and synthesis. However, as a digital circuit becomes larger and more complex, this approach becomes more difficult. Instead of as an afterthought, we have to consider the testing procedure in the initial design and frequently need to add auxiliary circuitry, such as a *scan chain* or *built-in-self-test circuit*, to facilitate the future requirements. This is known as *design-for-test*.

1.5.5 EDA software and its limitations

Developing a large digital circuit is a complicated process that involves complex algorithms and a large amount of data. Computer software is used to automate some tasks. This is known as *electronic design automation (EDA)*. As computers become more powerful, we may ask if it possible to develop a suite of software and automate the development process completely. The ideal scenario would be that human designers only need to develop a high-level behavioral description, and EDA software will perform the synthesis and placement and routing and derive the optimal circuit implementation automatically. The answer is, unfortunately, negative. This is due to the theoretical limitations that cannot be overcome by faster computers or smart software codes.

The synthesis software should be treated as a tool to perform transformation and local optimization. It cannot alter the original architecture or convert a poor design into a good one. The efficiency of the final circuit depends mainly on the initial description.

The limitations and effective use of the EAD software are elaborated in Section 6.1.

1.6 DEVELOPMENT FLOW

Developing a digital circuit is essentially a refining and validating process, gradually transforming an abstract high-level description into a detailed low-level structural description. While all developments follow the basic refinement–validation process, detailed flows depend on the size of the circuit and the target device technology.

The optimization algorithms used in synthesis software are complex. The needed computation time and memory space increase drastically as the circuit size grows. Thus, size is a limiting factor in many synthesis software tools. The software is most effective for an intermediate-sized circuit, which ranges between 2000 and 50,000 gates. For a larger system, we must first partition the circuit into smaller blocks and then process each block individually.

Another factor is the target device technology. The fabrication processes of FPGA and ASIC are very different. Whereas an FPGA chip is an off-the-shelf part that has been prefabricated and pretested, an ASIC design must go through a lengthy, complex fabrication process. Many extra steps are needed to ensure the correctness of the final physical circuit.

The following subsections show the typical development flow of three different types of designs and explain the extra steps needed as the complexity increases. Three types of designs are:

- Medium-sized design targeting FPGA
- Large design targeting FPGA
- Large design targeting ASIC

1.6.1 Flow of a medium-sized design targeting FPGA

The term *medium-sized* here means a design that requires no partition and does not need predesigned IP cores. It is a circuit with up to about 50,000 gates. Current synthesis software and placement-and-routing software can effectively process a circuit of this complexity. This size is not trivial. It corresponds to that of a moderately complex circuit, such as a simple processor or bus interface. The development flow is depicted in Figure 1.4. It is shown in three columns, representing a synthesis track, physical design track and verification track respectively.

The flow starts with the design file, which is normally an RT-level description of the circuit. It may be accompanied by a set of constraints that specify the timing requirements. A separate file, known as a *testbench*, provides a virtual experiment bench for simulation and verification. It incorporates the code to generate input stimuli and to monitor the output responses. Once these files are created, the circuit can be constructed and verified accordingly. The steps in an ideal flow are detailed below.

1. Develop the design file and testbench.
2. Use the design file as the circuit description, and perform a simulation to verify that the design functions as desired.
3. Perform a synthesis.
4. Use the output netlist file of the synthesizer as the circuit description, and perform a simulation and timing analysis to verify the correctness of the synthesis and to check preliminary timing.
5. Perform placement and routing.

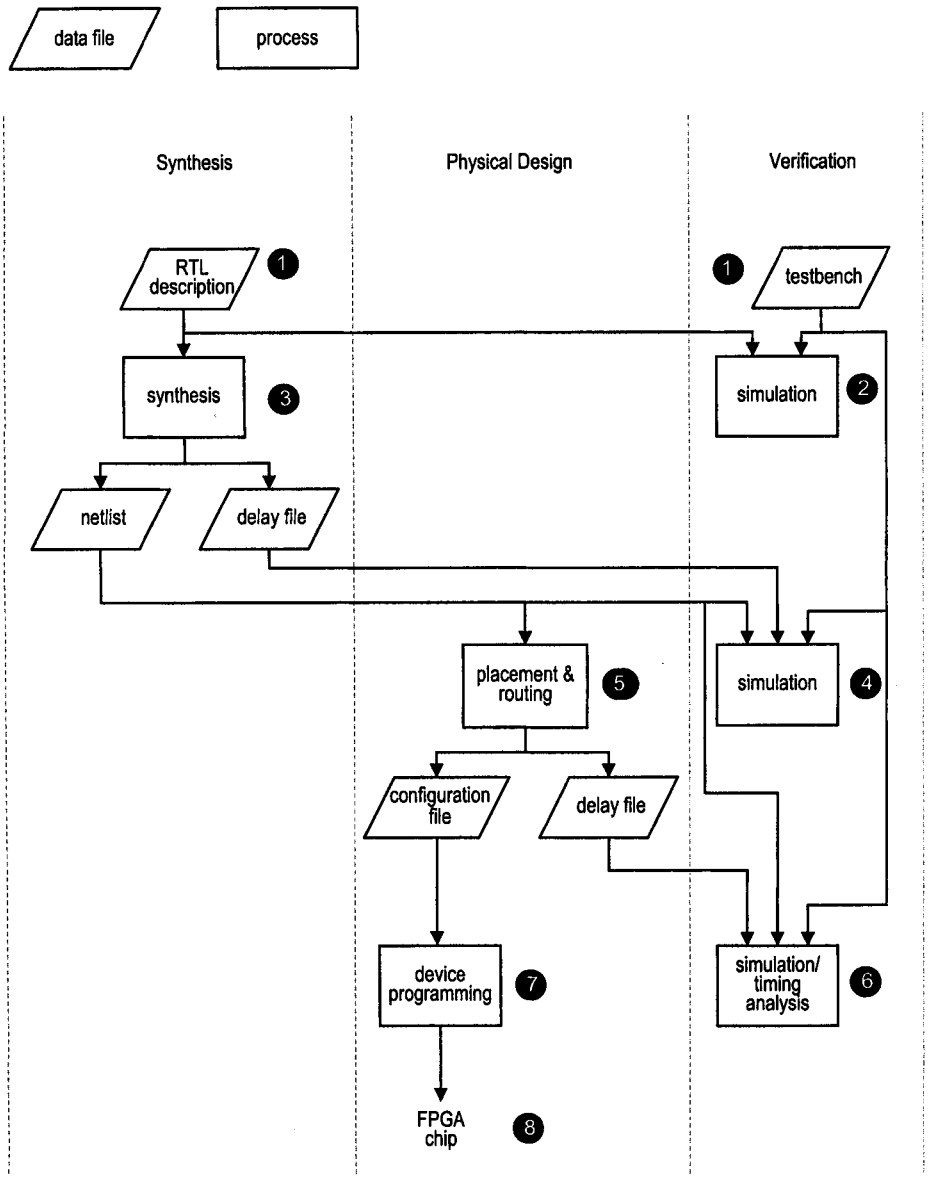


Figure 1.4 Development flow of a medium-sized design targeting FPGA.

6. Annotate the accurate timing information to the netlist, and perform a simulation and timing analysis to verify the correctness of the placement and routing and to check whether the circuit meets the timing constraints.
7. Generate the configuration file and program the device.
8. Verify operation of the physical part.

The flow described above represents an ideal process since it assumes that the initial design description follows the functional specification and meets the timing constraints. In reality, the development flow may consist of several iterations to correct the functional errors or timing problems. We may need to revise the original design file or to fine-tune parameters in synthesis and placement-and-routing software.

1.6.2 Flow of a large design targeting FPGA

A large, complex digital circuit may contain several hundred thousand or even a few million gates. Synthesis tools are not able to perform transformation and optimization effectively in this range. It is necessary to partition the circuit into smaller blocks and to process the blocks individually. The partition process also allows us to use previously designed subsystems or commercial IP cores.

To accommodate a larger design, additional processes must be added to the flow of Figure 1.4. The initial design description tends to be an abstract, high-level behavioral description of the circuit. In the synthesis track, a *partition process* is needed to divide the systems into blocks of adequate size and functionality. The output of the partition process can be considered as a netlist of large blocks. Some blocks may be already designed and verified subsystems, either from a previous project design or from a commercial IP vendor. The other blocks must be designed and synthesized individually as medium-sized circuits, following the development flow of the previous subsection.

In the verification track, an extra step is needed to verify the correctness of the partition results and to check the initial timing. Because of the large number of components, the gate-level netlist becomes very involved, and simulation consumes a significant amount of time. Formal verification techniques and cycle-based simulation are frequently used as an alternative to verify the functionality.

In the physical design track, a floor planning process may be needed. It performs initial placement for the processor-level blocks.

1.6.3 Flow of a large design targeting ASIC

Due to the complexity of ASIC fabrication, the development flow becomes more involved. The additional requirements are the inclusion of a testing track and the expansion of the physical design track.

The purpose of testing is to detect defects in the fabrication process. FPGA devices are tested by vendors before being shipped, and thus we don't need to worry about physical defects of the device. On the other hand, the testing is the integral part of the ASIC design and plays an important role. At the RT level, additional built-in-self-test circuits and special scanning control circuits are frequently added to aid the final testing. These circuits become an integral part of the design and have to be synthesized and verified. At the gate level, scan registers will be strategically inserted around circuit blocks or I/O boundaries. The scan circuit also needs to be synthesized and verified along with the regular design. Finally, test vectors have to be generated for combinational circuit blocks, and simulation has to be performed to ensure that the vectors provide proper fault coverage.

In FPGA-based flow, the physical design track involves only the floor planning and placement and routing, which is accomplished by configuring the FPGA device's programmable interconnect structure. The physical design process of an ASIC device is much more complicated since it involves development and verification of the masks. After placement and routing, several additional steps are needed, including design rule checking, physical verification and circuit extraction.

Due to the high NRE cost of an ASIC-based device, it is important that the circuit is simulated and checked thoroughly before fabrication. Thus, the verification track of ASIC-based design flow has to be more comprehensive and more exhaustive.

1.7 OVERVIEW OF THE BOOK

1.7.1 Scope

This book focuses primarily on the design and synthesis of RT-level circuits. A subset of the VHDL hardware description language is used to describe the design. The book is not intended to be a comprehensive ASIC or FPGA book. All other issues, such as device architecture, placement and routing, simulation and testing, are discussed only from the context of RT-level design.

After completing this book, readers should be able to develop and design efficient RT-level systems or subsystem blocks. A physical chip for a medium-sized FPGA design or a large, manually partitioned FPGA design can be obtained with a general synthesis and placement and routing software package. Additional knowledge and more specialized software tools are needed to cover the other tasks for an ASIC design.

1.7.2 Goal

The goal of the book is to learn how to *systematically develop efficient, portable RT-level designs that can easily be integrated into a larger system.* The goal includes three major parts:

- Design for efficiency
- "Design for large"
- Design for portability

Design for efficiency Availability of HDL and synthesis software relieves us from many tedious, repetitive implementation details and allows us to explore the design at a more abstract level. However, algorithms used in synthesis software can only do transformation and perform local search and optimization. They cannot, and will not, create a good design description or convert a poor design description to a good one. The quality of the circuit lies primarily in the initial description.

The book shows the relationship between VHDL constructs and the hardware components as well as the effective use of synthesis software tools, and introduces a disciplined way to develop the initial description that leads to efficient implementation.

"Design for large" We use the term *design for large* loosely to cover three aspects:

- Design of a large module
- Design to be incorporated in a larger system
- Design to facilitate the overall development process

The main purpose of most digital system books, including this one, is to illustrate basic concepts and procedures. For clarity, the design examples are normally explained by circuits with small input size. However, the design and description for a system with a small number of inputs (e.g., a 2-bit multiplier) and a system with a larger number of inputs (e.g., a 32-bit multiplier) can be very different. Although small-input-size examples are used in this book, the design approach and coding style are aimed at a large input size, and thus the design can easily be expanded to a larger, more practical system.

As the digital system becomes more complex, an RT-level description is likely to be a part of a larger system. Although a large processor-level system is not the focus of this book, the coding and development take this into consideration so that the RT-level design can easily be incorporated into a larger system when needed.

The discussion in Section 1.6 shows that the development of a large digital system involves many tasks. RT-level design and synthesis are not an isolated part. A poorly constructed RT-level circuit makes simulation, verification and testing processes unnecessarily difficult or even impossible, and sometimes may need to be revised at a later stage of the development process. While the book focuses on RT-level design and synthesis, it treats this task as an integral part of the development process and uses methodology that can facilitate and even simplify other tasks.

Design for portability Portability means that the same design description can be used in different applications. We can examine design for portability from three perspectives:

- Device independent
- Software independent
- Design reuse

Device independent means that the same design description can be synthesized to different device technologies. From time to time, the same design may need to migrate to a different technology. It can be from one FPGA vendor to another or from FPGA to ASIC for volume production. The design descriptions of this book carefully avoid any device-dependent feature so that the code can be used for multiple device technologies.

Software independent means that the design description can be accepted by most synthesis software. Since synthesis is a very complex process, software packages from different vendors have different capabilities, support different subsets of hardware description language and may have different interpretations on some subtle language constructs. We try to use the minimal common denominator of the synthesis software so that a design description can be accepted by most software tools and its function will be interpreted in a similar manner.

Design reuse means that the whole or part of the design description can be used again in a different application or project. We interpret the term *reuse* in a broad sense, from the copying of a few lines of code to a complete IP core. While developing an IP core is not the primary goal, we try to make the code modular and scalable when possible so that the same code can be reused in different applications with minimal or no revision.

1.8 BIBLIOGRAPHIC NOTES

This book includes a short bibliographic section at the end of each chapter. The purpose of the section is to provide several of the most relevant references for further exploration. A complete comprehensive bibliography is provided at the end of the book.

Developing a large digital system is a complex process. The text, *Methodology Manual for System-on-a-Chip Designs, 3rd edition* by M. Keating and P. Bricaud, provides an overview and guidelines for the process. The text, *The Design Warrior's Guide to FPGAs* by C. M. Maxfield, introduces relevant issues on FPGAs. Two texts, *FPGA-Based System Design* and *Modern VLSI Design: System-on-Chip Design, 3rd edition*, both by W. Wolf, provide more in-depth reviews of the FPGA and ASIC technologies.

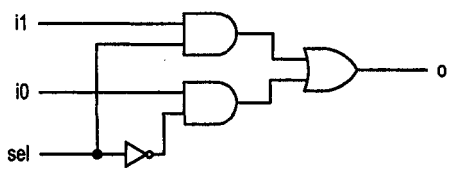
Problems

1.1 An engineer claims the following about the digital format: "In a digital system, logic 0 and logic 1 are represented by two voltage levels. Since there is a significant voltage difference between the two levels, noise will not affect the logic value, and thus digitized information is immune to noise." Is the statement correct? Explain.

1.2 Volume of sale (i.e., the number of parts sold) is a factor when determining which device technology is to be used. Assume that a system can be implemented by FPGA, gate array or standard-cell technology. The per part cost is \$15, \$3 and \$1 for FPGA, gate array and standard cell respectively. Gate array and standard-cell technologies also involve a one-time mask generation cost of \$20,000 and \$100,000 respectively.

- **(a)** Assume that the number of parts sold is N. Derive the equation of per unit cost for the three technologies.
- **(b)** Plot the three equations with N as the x-axis.
- **(c)** Determine the range of N for which FPGA technology has the minimal per unit cost.
- **(d)** Determine the range of N for which gate array technology has the minimal per unit cost.
- **(e)** Determine the range of N for which standard-cell technology has the minimal per unit cost.

1.3 What is the view (behavioral, structural or physical) of the following illustration?



1.4 What is abstraction? Why is it important for digital system design?

1.5 What is the difference between testing and verification?

1.6 In Figure 1.4, the synthesized circuit is simulated in steps 4 and 6. Is the simulation in step 6 necessary? Explain.