ON THE DESIGN OF A COMPUTER-AIDED DESIGN SYSTEM FOR DIGITAL CIRCUITS

By

Tao Shinn Chen

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1988

ABSTRACT

ON THE DESIGN OF A COMPUTER-AIDED DESIGN SYSTEM
FOR
DIGITAL CIRCUITS

By

Tao Shinn Chen

A methodology for automating the design processes for digital circuits is investigated. This research focuses on establishing both the unification and consistency for the overall design environment. The study at the unification issue focuses on the circuit representation in all aspects in order to provide a tool-independent and component-oriented design database. And the objective of the consistency issue is to minimize redundant or repetitive tasks during the design processes.

Several tools are developed to illustrate this methodology. There are a graphic tool with interactive visualization for design capture, a logic-timing simulator at the MOS transistor-switch level, and a verification tool for circuit layout. The embedded theory for the methodology and tools is established and proven. For the graphic tool, the major endeavor focuses on developing efficient data structures for any circuit schematic with unlimited drawing size and only having fastest graphic operations executed to obtain the shortest computer response time. Next, a totally new method is used for switch-level simulation. This approach not only has linear-time complexity but also obeys the law of excluded middle. Moreover, the bidirectional problem for simulating MOS transistors at pure logic level is solved. The performance analysis shows that this tool can simulate a 15,000-transistor circuit with the speed of less than 25 seconds per clock on a typical workstation (SUN-3). Finally, a rule-based verification approach provides a static way to validate circuit structures from the layout level to a system level without specifying the inputs. The fundamental work which verifies a transistor network by its Boolean functions is developed. This method ensures full correctness of the circuit topology because it

takes all components into account.

Since the system was implemented in C and Prolog languages and the machine dependent codes are separated as much as possible, it is easily portable to other machines which have the graphic display capability. Future work should be directed toward increasing the rate of automation with consideration of human factor, extending the work into the field of computer-aided engineering, and incorporating a hardware description language, such as VHDL, for circuit algorithmic and behavioral development.

Acknowledgments

Thank my advisor, Dr. P. David Fisher, for his detailed review and very constructive criticism of my dissertation. My wife, Rueyeh, also deserves the appreciation for her endless support. And, for all the persons in the Electronic Research and Development Laboratory, especially Brian Musson, I acknowledge their assistance and excellent performance for maintaining the computer systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

The computer-aided design (CAD) for digital circuits involves three major objectives, i.e., design capture, design verification, and layout implementation. Since Mead and Conway [36] simplified the complexity of the design rule for integrated circuit layout, circuit implementation with customized chips spread rapidly. And, the current technology of integrated circuit design continues to evolve resulting in chips with greater functional complexity and speed. This thesis concerns the methodology for designing a CAD system which takes full advantage of current computer technology in order to enhance the custom integrated circuit design environment.

1.1 Problem Statement

A CAD system, which assists human to complete design tasks, must integrate many different tools in order to handle all aspects of a design. As a result, many systems are tool oriented. The translation of design data among different tools not only requires designers to handle different sets of tools but also may introduce unconscious errors in the design. Moreover, inconsistency of tools also may generate repetitive and/or redundant work. To cope with this, new methodologies are required in order to speed up the overall design process while minimizing the number of design errors.

Another crucial issue is design verification. It is well known that the complementary metal-oxide-semiconductor (CMOS) technology has brought the digital circuit design down to the transistor level. The traditional design method which focuses on the gate level in general doesn't meet the requirements for designing high-performance circuits. Although the logic simulation technology at the gate level has almost achieved linear-time complexity [26] [32], it can not apply directly to MOS transistors due to the bidirectional feature of these devices. As a result, many researchers use approximate circuit theory to deal with this problem [12] [24] [25] [34] [41]. However, the performance of

their simulators can not reach the linear-time complexity and is generally circuit-topology dependent. Moreover, their simulators are not suitable for CMOS logic design because the corresponding layout has not yet been implemented. To solve this problem, a new simulation technique, which is based on the MOS-transistor-logic model [10], needs to be developed. Such a simulator generates a new class of switch-level simulation which is higher than the current switch-level simulation and lower than the gate level simulation. Therefore, it is very suitable for analyzing digital transistor networks, such as pass transistor logic [43], before the layouts are implemented.

It is also well known that simulation may not discover some errors due to the complexity of digital circuits. To verify a layout from its circuit topology ensures the correctness of its function. However, one of the major difficulties of this approach is the variety of possible circuit structures that must be addressed [13] [14]. To deal with it, formal verification is required, which can take advantage of the circuit hierarchy in order to reduce the circuit complexity.

1.2 Approaches

The design process for digital circuits involves a set of diversified tasks which span design input to device fabrication and testing. Some portions of the design process are changing rapidly due to advanced fabrication technology. However, some other portions, such as the digital theory, remain almost the same. This is very similar to compiler techniques, i.e., the theory in the front end is fairly stable, but the back-end techniques vary from one processor to another. This same phenomenon guides the methodology of designing a CAD system. Another phenomenon in VLSI circuits is that the circuits are notoriously "unforgiving" with respect to design errors. Any defect in the design of a chip usually makes it useless. This implies that the verification tools are more and more important since debugging is usually impossible after a design is fabricated. Based on these facts, this research primarily focuses on the front end of the design process.

The front end is separated from the whole design process down to the transistor level. The fundamental elements in the front end are MOS transistors, i.e., they are modeled as switches or attenuators with some delay values. In other words, the goal of this research is to provide an enhanced environment to implement digital circuits from design input above the transistor level to design verification at the (ideal) transistor level. This environment supports necessary facilities to verify a design before it is translated into the physical layout level. Therefore, this CAD system is intended for use in the areas of full custom design, cell-based design, and random logic design. To achieve the objectives, three tasks are involved, i.e., develop a method for design capture, a technique for tool integration, and a design-verification methodology.

The initial task in implementing digital circuits is to capture the circuit connectivity. This can be realized by drawing the circuit schematics. The schematic-entry interface is a graphic tool using pop-up menus, icons and multiple-window management for design capture. The complexity of circuit schematics is unified by two objects, i.e., the connecting wires and the circuit components. Wires are modeled as continuous line segments while components are modeled as a set of elementary drawing commands. By interpreting the drawing commands, any kind of graphic symbol can be used to represent a component without program modification. This tool can be used not only as a circuit-schematic editor but also as a general-purpose graphic editor for some kinds of network analyses. This approach creates an environment which makes the drawings at any structure-description level possible, as long as it is higher than the layout level. Other techniques, including circuit expansion from the design level down to the (ideal) transistor level, hierarchical design methodology and documentation environment, are developed and integrated into the CAD system.

The second task is to develop a technique for tool integration. The objectives of this task are to combine individual tools together, to update inefficient tools, and to accommodate new tools. This work implies that tools should be as independent as possible on

their interesting data format and location. As part of the work required here, a unified database was developed to store circuit components. This database can grow in two directions, i.e., not only can new components be added but also new attributes of old components can be created without any modification to existing tools. Therefore, this database serves as the kernel for the CAD system. In order to create a team-design environment and take advantage of existing information provided by the UNIX file system, this database is realized as several directories, and components are files in these directories. A designer can use his/her own local database and share his/her work with other designers through the access of a group and/or global database.

The design verification task focuses on the ideal transistor level. Since this is the lowest level for logic timing simulation, the greatest accuracy above the switch level can be obtained by modeling MOS transistors with different delay values. A new algorithm for switch-level simulation with or without unit-delay constraints is developed. Here, digital circuits are modeled as time-invariant, linear, discrete-time dynamical systems, much like systems are modeled in control theory. And, MOS transistors are modeled as bidirectional switches. Since simulation is often done prior to the circuit layout, at which time the load and routing capacitances are not known, the delay of each transistor is obtained from the fanout information. A simulator based on the new algorithm is implemented, and it is proved that it can simulate any digital CMOS circuit. The computational complexity is linear with respect to the number of transistors in the target circuit. Clearly, it is more versatile than the current event-driven simulators, such as MOSSIM II [24], since they can not simulate oscillating circuits and the simulation speed is circuit-topology dependent.

Since simulation only depends on the input data, some errors may not be covered during simulation. A tool for symbolic verification is also developed. This rule-based symbolic verifier recognizes the CMOS gate structures in a circuit and generates the verified Boolean equations for the gates.

The research and development work reported here results in a schematic-entry CAD system for CMOS digital circuits with an open architecture. Currently, this CAD system is at the transistor-logic level for design verification and at any structural description level for design capture. However, it is easily extended down to the layout level by accommodating some tools for layout generation and verification, and up to the functional-description level by adopting some hardware description language for design capture, e.g. VHDL [16] [17].

1.3 Outline of the Thesis

This thesis contains seven chapters. Chapter 1 is the introduction which provides the problem statement, general approach, and overview of the accomplishments of this research. Chapter 2 explains the design methodology of the CAD system. The characteristics of typical digital circuit design methodology is considered first in this chapter. Then, the consistent working environment which supports the design process is described. At the end, the representation method for digital circuits is given which guides the development of this system. Chapter 3 concerns the structure of the embedded component database. The database operations and design "flattening" down to a specific component level are presented. Based on the unified database, several tools are developed. They are discussed in the rest of chapters.

Chapter 4 introduces the idea of design and implementation of a universal graphic tool for design capture. The data structures and necessary tasks are described. Chapter 5 is dedicated to the switch-level simulation. The simulation theory and the method which overcomes the bidirectional feature of MOS transistors in order to support the transistor-logic models are provided. Chapter 6 describes the techniques for structure verification. The outline of a hierarchical verification system and the work at the MOS transistor level are depicted. Chapter 7 provides summary and conclusion. The extensibility of this system is also discussed in this chapter.

Chapter 2 Design Methodology

The complexity of digital circuit design has increased rapidly due to the progress of integrated circuit (IC) technology. The systems of yesterday are the boards of today, while the boards of yesterday are the ICs of today. Design is a creative activity. However, it involves so many details in order to turn a concept into a VLSI circuit or system. The goals of computer-aided design (CAD) are to minimize all redundant or repetitive work and let designers concentrate on the creative aspects of a design. Two major issues of designing a CAD system are the working environment during the design process and the design representation. A friendly user interface is usually used to describe the environment of a system. However, to deal with a complete design task, it is only an essential condition.

Design representation is another crucial issue. For a specific tool, design representation is much easier since only one aspect of the design needs to be of concern and modeled. For example, a simple NAND gate may be modeled as a graphic symbol in a schematic editor, a set of statements in some hardware description language, a layout in a layout editor, or several transistors, capacitors, and resistors in circuit simulators. Individual tools, no matter how powerful, are rarely useful unless well integrated into a system [6]. A complete CAD system integrates many tools to achieve the design task. Therefore, it must have the capability to model as many aspects as possible. In other words, a design representation method must be developed in order to provide a uniform user interface for the development and use of CAD tools in an open system [7].

This chapter concerns the design methodology of building such a CAD system for digital circuits in the digital logic-design field. At the beginning, we characterize the digital circuits. Then, the working environment and design representation issues are discussed.

2.1 The Characteristics of Digital Circuit Design

A general design of digital circuits has three dimensions. The creation of a circuit occupies two dimensions since the components in the circuit must be placed and connected. The other dimension transforms a "dumb" circuit into a "smart" circuit. The micro-code design for a processor is an example. Although the third dimension may not be covered in a circuit design, the progress of VLSI technology shrinks the area of pure two-dimension designs. However, since the theory used in the third dimension is much like that in the programming world, we restrict the characteristic analysis of digital circuit design in the electronic world.

Rubin [1] gave a description of characteristics of digital circuit design. Based on his characterization, digital circuit design has four characteristics, i.e., hierarchy, different views, connectivity, and "flat" geometry in circuit layout. However, if we emphasize the "digital" portion, other characteristics can be obtained.

We characterize the endeavor of digital circuit design as follows:

(1) Simple primitive components -- A digital circuit, no matter how complicated it is, usually contains very few primitive components. For examples, a digital CMOS circuit only contains PMOS and NMOS enhancement transistors and a TTL circuit only contains NAND gates in the digital world. Most efforts in designing a digital circuit focus on the selection of components and making the connection of the selected components to compliance with the design specifications.

(2) Hierarchy -- The hierarchical approach is the natural way to design a digital circuit. According to the design specifications, a circuit is decomposed into many functional blocks. Each block is further divided into lower level blocks, etc. By hiding low-level details, one can view a design as a component tree. Recrusively, the whole structure of a digital circuit is a complex tree with the primitive components at the bottom of hierarchy, i.e, the leaves of the tree. A design which restricts all the components in the same level is said a "flat" design [6]. Due to the complexity of today's digital circuits, it is almost

impossible to design circuits this way.

(3) Connectivity -- The unified view of a digital circuit is a collection of components. Defining the relationship among the components is the major activity in circuit design. However, after the connectivity is established, the design endeavor concentrates on minimizing any parasitic phenomenon which is introduced by the connecting materials. Therefore, a design process can be further separated into two phases. The first phase, called logic design, deals with logic (ideal) components and the second phase, called implementation, is to reduce harmful parasitic effects which are introduced by connecting wires and real components.

Based on this methodology, a layout tool which mixes logic design and implementation together is not a satisfactory solution for digital circuit design because these tools would require the designer to pay attention to many details simultaneously in the implementation while doing logic design. Hence, one can say such a tool is an implementation tool rather than a design tool. However, this does not mean a layout tool is not necessary. On the contrary, it is an essential tool for integrated circuit design. The topic of argument is that it is merely a "tool", not a complete design "system".

(4) Documentation -- A chip or circuit board is almost useless if there is no manual along with it. All chips look the same regardless their actual size or shape. Digital circuits are usually embedded in some system to perform the desired function. Chips or circuit boards are generally intermediate products. Their complexity requires documents to carry necessary information for down stream work. Hence, a design is not completed until its document is finished. An apparently redundant work is to draw the whole design again in preparing the document after a design is almost finished. Although a circuit schematic can be printed or plotted alone, many manuals require that it appears along with the text. A complete CAD system should take care of this requirement and minimize any redundant work.

Briefly, our design methodology recognizes that digital circuit design is an endeavor of defining connections among the components. The design process benefits from hierarchical representation. Hierarchy divides design efforts into many levels. Therefore, at a giving level, the design complexity is greatly reduced. Most tools can take advantage of hierarchical design to speed up their execution. Moreover, the separation of the design process into two phases frees designers to concentrate on the creative aspects of circuit design. The logic design phase let designers focus on the creation of a circuit. Then, the created logic circuit is modified in the implementation phase to get rid of any fatal parasitics.

Notice that the implementation phase in the design process does not mean to produce final chips or circuit boards. For integrated circuits, the results from the implementation phase is the mask data. And, for circuit boards, it means the generation of information needed for board layout.

2.2 Working Environments

Several papers can be found which discuss the design methodology of a complete CAD system. Dunn describes the VLSI design methodology used at IBM [8]. This system called DAV ranges from graphic schematic entry for design capture to test generation for validating real chips. McCalla, et al. describes a VLSI design system called ChipBuster which is used in Hewlett-Packard [6]. His description also provides a general view of CAD system designs. Burling describes the product design and introduction support systems called SysCAD which is used in AT&T [9]. SysCAD contains many subsystems in order to provide a complete solution for circuit design. Harrison, et al. introduces the Berkeley design environment by discussing its fundamentals, i.e., a data manager called Oct and a graphic editor called VEM [7].

One common feature among these CAD systems is that each of them has a design database serving as the kernel. The database provides different aspects of a circuit to different tools and acts as a library for design sharing and for concurrent access by

designers. The other feature is that the graphic-entry approach is used by each for design capture.

From their discussions, we conclude that the working environment for digital circuit design can be classified into five portions. There are design capture, simulation and verification, team work, documentation, and implementation. The environment for design capture provides tools for creating the topology of a circuit and defining the circuit hierarchy. The simulation and verification environment supports tools for validating a design and may also provide data for circuit testings. The team work environment is a network and database facility which supports the necessary mechanism for sharing design achievement. The documentation environment provides tools for recording design results. And, finally, the implementation environment supports tools for generating detail specifications for constructing a circuit.

Nowadays, many CAD systems emphasize on the implementation environment and, more or less, ignore the other environments. Therefore, designers must repeat some tasks which have already done previously. For example, a hardware description language can be used to create a circuit design. However, after a design is created and validated by some simulation actions, the circuit schematic may still need to be created for documentation. Or, when using a graphic tool to create a circuit topology, one may need to generate a input file which specifies the topology for simulations and he/she still needs doing the circuit layout separately. For all of these repetitive tasks, design endeavor is distributed and shrunk.

In our methodology, we focus on the logic design phase with one objective being to provide a consistent working environment. Figure 2-1 is an abstract illustration, where A, B, etc., represent different working environment. To minimize the repetitive tasks, consistency among design environments is essential. And, the creativity of a designer is enhanced by providing a pure logic design environment.

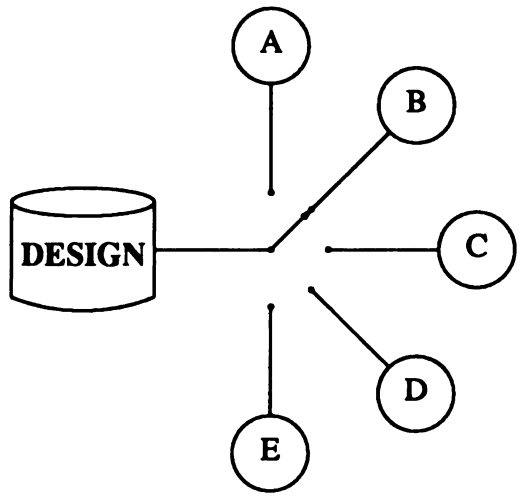


Figure 2-1 A consistent working environment.

2.2.1 Design Capture Environment

Two approaches are currently used for design capture. One is text entry and the other is graphic entry [10]. Since a general digital circuit design has three dimensions, each approach has advantages and disadvantages. The text entry approach, such as a hardware description language, is very suitable for designing circuits at the behavioral level. Many highly structured circuits, such as ROMs and PLAs, are easily described in text approach. However, text approach has one deficiency; it is awkward to describe the connectivity of a design. On the other hand, graphic entry overcomes this since a circuit schematic is much easier for people to understand the relations among components. Moreover, if a text description of the circuit is also required, this text-level description can easily be automatically provided once the circuit has been captured schematically. We assert that schematic entry for design capture is a valuable tool for a complete logic design system, even if some hardware description language is supported.

Hierarchy and connectivity guide the design of a schematic editor. According to the hierarchy feature of a circuit design, the objects at any given structural level are components and connecting wires. A schematic editor provides an interface for making all the connections among the components and a mechanism for "packing" a design at one

level and turning it into a component at a higher level. Based on these unified facts, one can implement a schematic editor for design capture at any structural level in the hierarchy. such as transistors, to the system level.

We designed and implemented a schematic editor based on the above considerations. Chapter 4 describes our approach. This tool takes advantage of evolving workstation capabilities, e.g., pop-up menus, icons, and multiple-window management for design capture. It ranges from the MOS transistor level to any higher structural level.

2.2.2 Simulation and Verification Environment

Design verification usually occupies a large portion of design endeavor [8]. Since the hierarchy of a design is generally established in the design capture environment, simulation at the current design level can be achieved by modelling the components at the current level [11]. Another approach is to "flatten" the whole design and simulate at a level which contains only the same components or the primitive components. Gate-level, switch-level, and circuit-level simulators use the second approach. Ruehli and Ditlow gave a good overview of simulation and verification technique for VLSI circuits [12]. Clearly, simulation at the lowest design level may have the greatest accuracy since all the details are involved.

To deal with the simulation issue, we developed a theory for logic timing simulation. Chapter 5 presents this simulation theory. The approach models a digital circuit as a network which consists of different kinds of basic build elements, where the basic building element need not be a primitive component. However, the I/O function and the delay time of a basic build element must be specified or calculated before simulation.

The computational complexity of this simulation theory was proven to be linear with respect to the number of basic build elements in the circuit being simulated. The disadvantage of this approach is that it can not handle the bidirectional characteristics of MOS transistors. However, an algorithm called the strength determination algorithm

(SDA) was developed, which overcomes the bidirectional problem. The simulation theory is still valid after SDA determines the signal flows among the transistor network of a design. The computational complexity of SDA was also proven to be linear with respect to the number of transistors. SDA is also described in Chapter 5.

The result obtained from the simulation is a logic timing diragram. The resolution of the timing diagram is the minimun delay time of the basic building elements. Several examples can be found in Chapter 5.

Another approach to validate a design is to verify its structure [13] [14]. A symbolic approach can verify a design without specifying the input data. Since the structure of a circuit is established in the logic design phase, verification with the structure seems very attractive after the implementation phase is nearly finished. For example, the layout of a integrated circuit design can be verified by its circuit structure, which was established in the logic design phase. This is because the simulation approach may not cover all of the possible input patterns and the structure verification method does not have this drawback.

A tool which generates a set of corresponding Boolean equations from a CMOS circuit layout was also developed. It validates a layout at the gate level, and the generated Boolean equations can be used as input for verification at a higher level. The embedded theory for this tool is presented in Chapter 6.

2.2.3 Team Work Environment

Team work can be classified as being vertical and horizontal [18]. A vertical team work implies that a task is divided into up-stream work and down-stream work. For example, "a chip which is designed by A and then fabricated by B" is a vertical team work. And, horizontal team work implies everybody is working at the same stage. Clearly, in the logic design phase, the horizontal attribute of team work is a very important feature.

The major consideration in the horizontal team work environment is the desire to minimize the problems which are introduced by the duplication of a component during the design process. It ensures that every designer can obtain the original component. For example, a designer may finish an ALU design and copy it to other designers. Later, this designer finds that there is an error in the ALU component. The price to make all the copies being the same may be very high. This is especially true for many geometric lay-out tools.

In this methodology, the team work environment is established by the structure of the design database. The database stores all the components in a design. Each designer has his or her own local database. A group which consists of several designers owns a group database. And, finally, a global database can be accessed by all the designers. This approach ensures that every designer can obtain the original component. Hence, it reduces the duplication problem.

The UNIX file structure helps us to design such a team work environment. This environment is built into the schematic editor. When a designer needs a component which does not belong to him or her, a search is automatically made from the group databases to the global database in order to find the component. Therefore, everyone can obtain the original and newest version of a component.

2.2.4 Documentation Environment

Documents should be treated as part of the design result. Although many word-processing tools are available and friendly to use, redrawing the schematic of a circuit design is an obviously repetitive and redundant task. Inconsistency may happen between circuit design and art work. Moreover, the requirement of mixing text and pictures should be considered.

We take advantage of the word-processing facility in UNIX systems to achieve these documentation objectives in circuit design. This facility is a set of programs called

TROFF [2]. A schematic can be automatically translated into a series of drawing commands in TROFF. These drawing commands are text but can be interpreted and printed out as same as the original schematic. The drawing commands can be mixed among other text and the final printout has the quality which is comparable to that of real text books.

This approach for documentation has another benefit. Design documents can be mailed through computer networks worldwide. Nowadays, electronic mail facilities are more and more popular and much cheaper, even faster, than other transmission or delivery methods. However, only text is usually allowed to be sent. Since we translate a schematic into text before printing, this restriction is of no effect to our approach.

2.3 Circuit Representation

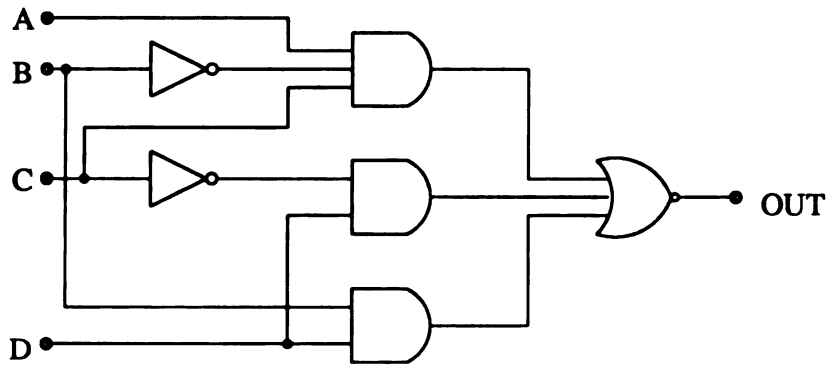
A well-defined circuit representation method can fully support the development of CAD systems since it provides a common interface to different tools and users [6] [7] [8] [9] [15]. The representation method should take circuit hierarchy and conectivity into account. We adopt the format used in predicate calculus since it meets the requirements of circuit representation.

2.3.1 Format

The circuit representation methods with predicate calculus format can be classified into three categories [3]. There are functional, extensional, and definitional methods.

(1) Functional Method -- Figure 2-2 shows an example using this method. An output is represented as a function of several inputs. There are two disadvantages to this technique. One is that only combinational circuits can be represented at the gate level, the other is every output needs a separate expression. However, a MOS transistor network is difficult to describe with this method since the transistors are bidirectional in the nature. And, feedback connections are also hard to be represented with this approach.

(2) Extensional Method -- This method represents circuits as modules and connection statements. An example is shown in Figure 2-3. The first argument in the modules is



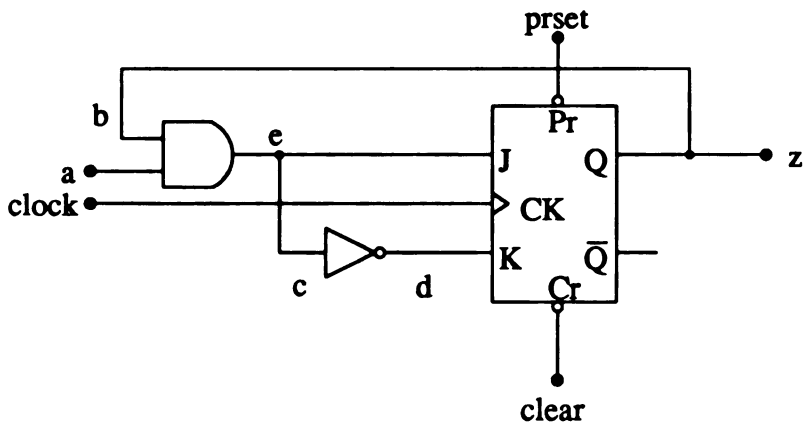
nor(and(A,not(B),C),and(not(C),D),and(B,D))

Figure 2-2 The functional method.

the function of that module, the second argument is the input variables, the last argument is the output variables. A connect statement shows a connection between the first argument and the second argument. This method can be at a higher level than the functional method and it can accommodate arbitrary types of circuits. But, one disadvantage is that modules are not represented by a single term with no systax relationship among them. In other words, wire names are multi-defined. Once again, like the functional method, this approach can not represent a MOS transistor network since it inherently defines the input and output attributes in module statements.

(3) Definitional Method -- This method is illustrated by an example in Figure 2-4. Components are described by Horn clauses [4] whose head is the circuit to be defined, and whose body is a composition of either already defined or primitive components. In other words, high-level components are much like rules and primitive components are facts in the Prolog environment [4]. The sequence of components in the body is arbitrary, but the sequence of the arguments in the head is fixed.

This kind of hierarchical representation makes modular decomposition a very easy task. Notice that the internal connections in a circuit are named by variables which do not appear in the head of the clause. These features make the definitional method more attractive since it masks low-level details. Thus, both circuit hierarchy and connectivity




```
component(and,[a,b],[e]).
component(not,[c],[d]).
component(jkmsff,[J,K,CK,Pr,Cr],[Q,Q̄]).
connect(and(b),z,jkmsff(Q)).
connect(not(c),and(e),jkmsff(J)).
connect(clock,jkmsff(CK)).
connect(not(d),jkmsff(K)).
connect(clear,jkmsff(Cr)).
connect(preset,jkmsff(Pr)).
```

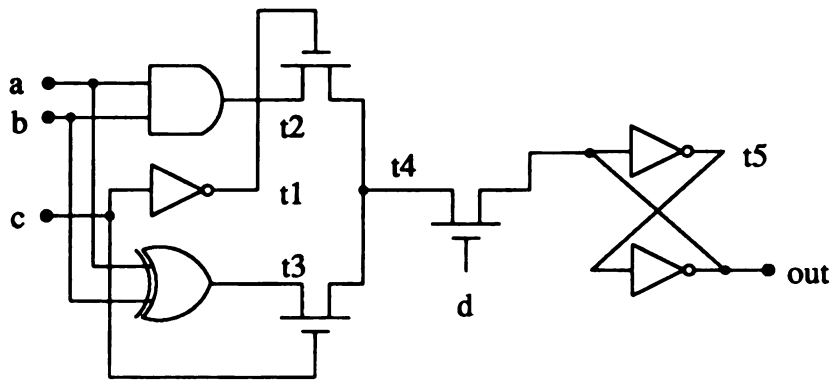
Figure 2-3 The extensional method.

are realized by this definitional method. Moreover, a MOS transistor network can be represented because this method does not restrict input and output relationships within a circuit.

The definitional method is chosen to represent digital circuits in our CAD system. Based on this representation, a generic component model is developed. The design database adopts the model and becomes the kernel of this CAD system.

2.3.2 Generic Component Model

In many cases, a design tool only models one or two aspects of a component. A complete CAD system has many tools to achieve design automation. The supported database in a CAD system should have the capability of providing a specific tool with all necessary data. Therefore, a component model must provide all the aspects in order to satify the needs of different tools. Figure 2-5 shows the abstract idea of a component




```
demo(a,b,c,d,out) :- not(out,t5),not(t5,out),and(a,b,t2),
                     not(c,t1),xor(a,b,t3),nt(t2,t4,t1),
                     nt(t3,t4,c),nt(out,t4,d).
```

Figure 2-4 The definitional method.

model. Consistency and hierarchy are the requirements to develop such a generic model.

The definitional method uses the component name to identify a component. Hierarchy implies that all the information which concerns the component must be found through the component name. Connectivity implies that the component communicates with other components through its input and output nodes which are the arguments in the component clause. The sequence of the component arguments is important since a specific node can be identified by knowing its position.

We model a component as a file with well-defined syntax. A component file is divided into arbitrary sections. Each section is dedicated to one aspect of the component. Sections are related through the sequence of the input and output nodes. For example, a schematic edtior may find the graphic symbol of a component from one of the sections in the component file and obtains the I/O node positions from a section which defines the I/O attributes.

There is a very special section called the structure section which defines the structure of a component. In this section, the definitional method is adopted. A component

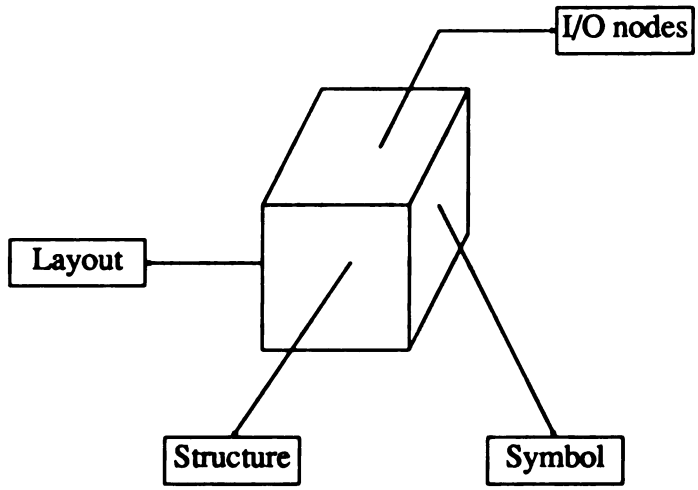


Figure 2-5 The model of a component.

structure is defined by other low-level components. The names of the low-level components are the file names which are used to find those components. Only the primitive components are defined by themselves. Hence, a hierarchical component tree is inherently formed by the structure section. Notice that the component hierarchy matches with the design hierarchy. This feature makes the add of a hardware description language, such as VHDL [16], into the CAD system much easier since the hierarchical programmability can be made consistent with the design hierarchy [5] [17].

A detailed description of our component model will be disscussed in Chapter 3 since the model is part of the design database.

Chapter 3 Unified Design Database

A typical design process usually is iterative, tentative, and evolutionary. To complete such a complex process, which may involve many design tools, a well-defined database with flexibility and extensibility is very essential for accommodating the different tools. However, the techniques and systems developed are usually to support business data processing. Database support for engineering remains to a large extent an open issue in CAD research because the representing entity for engineering database is much more complex.

Many endeavors have focused on this issue. Staley and Anderson not only presented a thorough specification for CAD databases but also gave a good survey in this field [19]. Their specification provides a clear goal for designing an engineering database.

A database to support this CAD system is described in this chapter. We adopt the relational model [20] as the fundamental database structure. Since there is only one type of representing entity, this database is called STOCK. Digital components are the entities in STOCK. The STOCK structure and its operations are discussed in this chapter.

3.1 Database Model

The relational data model in database theory is used to realize STOCK. However, owing to different requirements, the data model which is applicable to business database management must be modified in order to build such a design database.

The circuit representation method which was discussed in the previous chapter guides the STOCK design. The method defines a circuit as a component network. And, the circuit being defined can be used as another component for other circuits. Based on this unified point of view, there is only one type of entities in the design database, namely, the component type. Our design methodology treats any digital circuit as a component. Specifically, STOCK only contains one type of data, i.e., circuit components.

The architecture of STOCK is described by using the entity-relationship diagram, see Figure 3-1. The rectangle with bold edges represents a typical component entity, ellipses are the component attributes, and relationships among different components are represented by diamonds. Currently, each component has at most four attributes, i.e., the component name, the component structure, the graphic symbol, and the I/O nodes of the component. The name is used to identify the component in STOCK. The structure attribute describes the interconnection of the component. A component in STOCK is usually defined by other components except for the primitive components which are defined by themselves. Hence, the "*contains of*" relationship is embedded in the structure attribute.

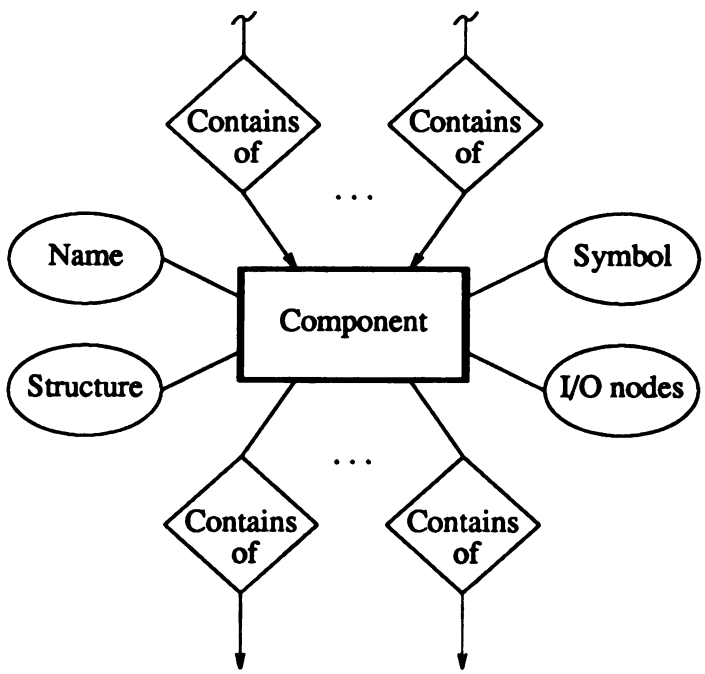


Figure 3-1 The entity-relationship diagram of STOCK.

STOCK is realized as several file directories. The directories in STOCK essentially have the same structure. There is a special file called "elsewhere" which is used to assist the automatic searching process. If a component can not be found in the current directory, this file will be opened and the search will go to the directories which are states in "elsewhere". This method distributes STOCK among directories which may even reside

on different machines. Clearly, the database maintenance problem is also reduced and localized. In other words, STOCK is a distributed database with unidirectional links. A typical structure of STOCK is illustrates in Figure 3-2. In the figure, each designer owns a local stock, several designers which form a group have a group stock, and all the designers have the right to access the global stock. More sophisticated structure can also be built by modifying the directory links.

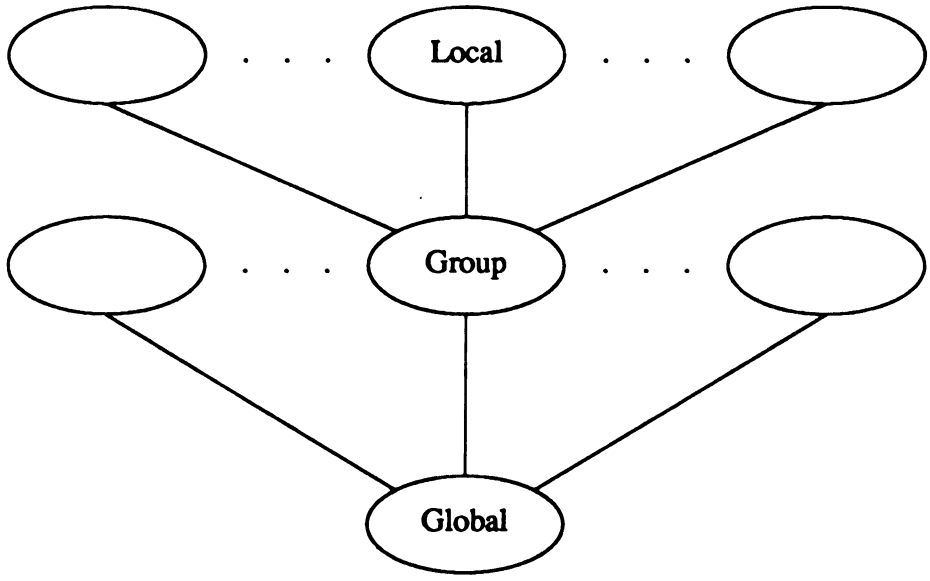


Figure 3-2 A typical structure of STOCK.

In STOCK, a component is realized as a file and each attribute occupies a space called a "section". We use the term "section" rather than the traditional term "record" because attributes have different formats and lengths. From the entity-relationship diagram in Figure 3-1, there are four attributes which must be modeled. Since a component is a file, the name attribute is handled by the operating system. Therefore, a component file currently has three sections to store the rest attributes. Figure 3-3 shows the structure of a typical component file which is divided into several sections. Notice that the sequence of sections is arbitrary. The section head which contains the section name is the key for finding the selected section.

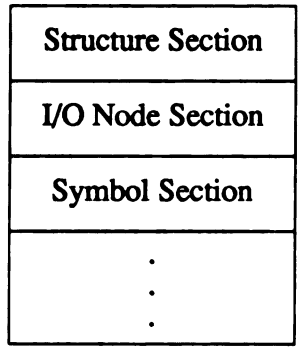


Figure 3-3 The structure of a component file.

The structure section of a component file defines the circuit topology which is the internal structure of the component. This definition is hierarchical, i.e., other components at the lower level, not necessarily the lowest level, are used to describe the component. In other words, a component is defined by other components, except for those at the lowest level. The components at the lowest level are defined by themselves. Figure 3-4 is an example of a component which is a J-K master/slave flip flop. The structure of this component is defined by several logic gates. These gates are also components in STOCK and they can be described by themselves or MOS transistors. It depends on whether those gates are primitive or not. Clearly, this approach utilizes the concept of hypertext [42].

```
nand3(%4,%0,%1,@0).
nand3(%1,%2,%5,@1).
nand3(%6,@0,@3,@2).
nand3(@2,@1,%3,@3).
nand2(@2,@6,@4).
nand2(@3,@6,@5).
nand2(@4,%4,%5).
nand2(@5,%5,%4).
not(%1,@6).
```

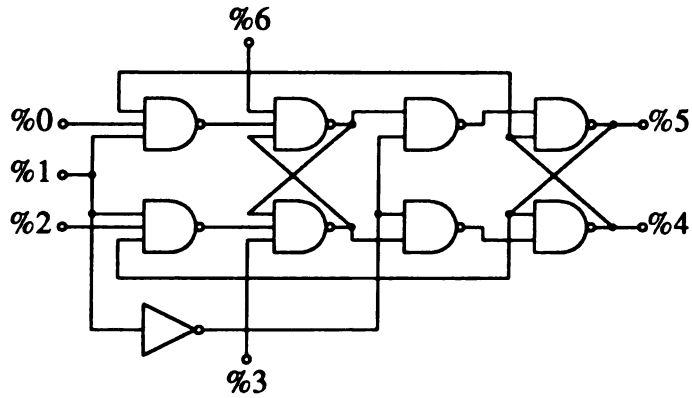


Figure 3-4 The structure section of a J-k master/slave flip flop.

The I/O nodes of a component occupy a section which defines the sequence, the location, and other attributes of input and output nodes. This section takes care of the

connectivity problem of a circuit. Therefore, we can say that the I/O nodes are "hot spots" of a component. A circuit schematic is a network of components which are connected by wires at their I/O terminals. Or, a layout tool must know those hot spots before the placement and routing can proceed.

The symbol section is used to portray a graphical representation for a component. Some drawing commands, such as drawing a line, a rectangular, a circle, and printing text, are defined. A component's symbol can be obtained by executing these commands. Figure 3-5 compares the drawing commands to the symbol of a J-K flip flop. It also shows the I/O node section of this component.

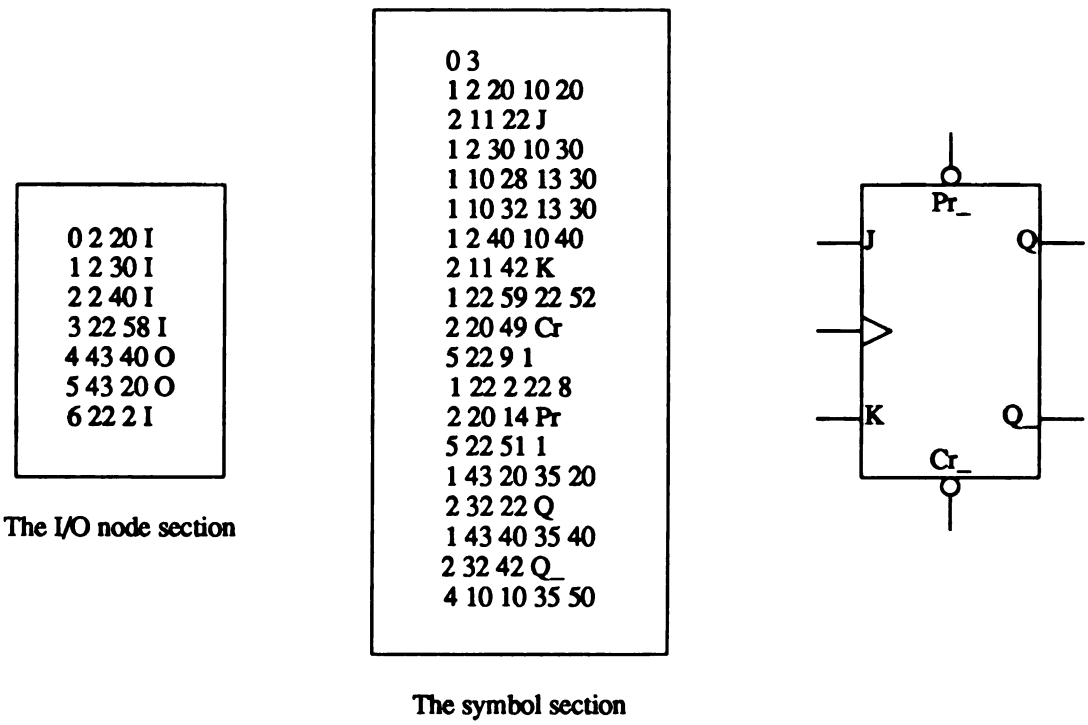


Figure 3-5 The I/O node and symbol sections of a J-k flip flop.

At this moment, the I/O node section only has a high relationship with the symbol section. Actually, the graphic locations of I/O nodes are determined by the parameters in the drawing commands. The reason we duplicate the information is to enhance the importance of node locations. By explicitly marking the locations, we eliminate a lot of restrictions in the drawing commands. In other words, the component's symbol can be

anything and it can be easily modified by changing the drawing commands. Moreover, this arrangement also makes the wiring process in schematic design capture simpler and more accurate since we need not extract the node locations from the symbol of a component.

3.2 Database Operations

According to the above database model, STOCK is a relational model with 4-tuples. Let R (name, structure, I/O nodes, symbol) be the relation of STOCK, the operations can be expressed with the relational algebra [20].

The union operation is achieved by adding a new link into the file "elsewhere". Specifically, let R_1 and R_2 be two relations, $R = R_1 \cup R_2$ will be obtained after the link of R_2 is added into the file "elsewhere" in R_1.

Since most tools are only interested in some aspects of a component, the most frequent operation in STOCK is the combination of selection (σ) and projection (π). For instance, during the schematic design capture session, $\pi_{3,4}(\sigma_1(R))$ is executed in order to obtain the graphic symbol and I/O node locations for a component. And, during the design flattening process, $\pi_2(\sigma_1(R))$ is the operation to discover all the low-level components.

In order to integrate new tools, new attributes should be able to be added easily. The natural join operation is the way to add new attributes. For example, let S (name,layout) be the relation which represents component layouts. Then, the natural join of R and S produces a 5-tuple relation, i.e., (name, structure, I/O nodes, symbol, layout). Clearly, the major difficulity in developing a new attribute is chosing its format , not the database operations themselves. However, several standard formats for layout attributes, such as the CIF and GDS II [23], are very useful if we want to extend the system into VLSI layout design.

3.2.1 Projection

One of the major differences between a business database and an engineering database is the attributes. For a business database, attributes usually contain a small amount of information, such as a string or a number. However, for an engineering database, an attribute may be used to model an aspect of a design, such as the layout or graphic symbol. Therefore, the amount of data is both greater and more complicated. The design of an attribute format is independent on the database operations. Tools which handle some specific attributes must have the capability to understand the format of attributes. Only the selection and projection operations are needed to provide the necessary attributes to a specific tool. In other words, when a tool queries a component, the database manager projects the necessary attributes from STOCK.

The current attributes in a component file are discussed in this section. Each attribute starts with a "#" character at the beginning of a line. The attribute name follows after the "#" character. Two names are supported, i.e., a numeric name and an ordinary name. Comments can also be added at a component file anywhere as long as a "." character begins a comment line. The following shows a typical component file :

```
.3-bit synchronous counter
#1 structure
nand3(%4,%0,%1,@0).
nand3(%1,%2,%5,@1).
nand3(%6,@0,@3,@2).
nand3(@2,@1,%3,@3).
nand2(@2,@6,@4).
nand2(@3,@6,@5).
nand2(@4,%4,%5).
nand2(@5,%5,%4).
not(%1,@6).
#2 nodes 45 60
0 2 20 I
1 2 30 I
2 2 40 I
3 22 58 I
4 43 40 O
5 43 20 O
6 22 2 I
#3 shape
```

```
0 3
1 2 20 10 20
2 11 22 J
. wire clock
1 2 30 10 30
1 10 28 13 30
1 10 32 13 30
1 2 40 10 40
2 11 42 K
1 22 59 22 52
2 20 49 Cr
5 22 9 1
1 22 2 22 8
2 20 14 Pr
5 22 51 1
1 43 20 35 20
2 32 22 Q
. wire Q_
1 43 40 35 40
2 32 42 Q
1 32 38 34 38
. the body
4 10 10 35 50
#4 end
```

The structure section defines the internal structure of a component. Each line in this section represents a component at the lower levels, not necessarily at the lowest level, with a function-like format. The "function" name is the lower-level component name which can also be found in STOCK. Arguments are used to define I/O terminals of the lower-level component. Those arguments starting with "%" indicate they are the I/O nodes of the component file. On the other hand, internal nodes are represented by the arguments starting with "@". The number in the arguments defines the sequence of nodes. This mechanism is simple, but powerful, and it is hierarchical in the nature. The naming technique in arguments provides an easy way to flatten a schematic into a transistor network. The lower-level component name indicate where to find the component in STOCK during the flattening process.

The I/O node section is currently closely related to the shape section currently. Each line in this section represents an I/O node; the first field in the sequence number; the second and third fields define the location of a node; and, finally, the rest of the field is reserved to describe other properties of a node for future extension.

The shape section defines drawing commands to make up a component symbol. Each line in this section is a drawing command. The first field is the command code. The rest fields are necessary parameters to execute the command. We summarize the available drawing commands as follows:

COMMANDS	FORMAT and DESCRIPTION
dSCALE :	0 scale_factor define the scale factor for a shape. This must be at the first line.
dLINE :	1 x0 y0 x1 y1 draw a line from (x0,y0) to (x1,y1).
dLABEL :	2 x y label write the label at (x,y).
dARC :	3 h k x0 y0 x1 y1 draw an arc at (h,k) from (x0,y0) to (x1,y1) clockwise.
dRECT :	4 x0 y0 x1 y1 draw a rectangular. (x0,y0) is the left-upper corner. (x1,y1) is the right-lower corner.
dCIRCLE :	5 h k r draw a circle at (h,k) with radius r.

Although STOCK only provides four attributes at this moment, its extensibility has already been considered. New attributes can be added without affecting the current attributes by using the natural join operation. Hence, other tools which operate on the current attributes do not need to be modified. The structure section provides all interconnection information about a component. Therefore, a lower-level component can be easily replaced or modified by changing its name or modifying its contents. (For example, this is very necessary to build a self-testable circuit [21].)

3.2.2 Design Flattening

Usually, a circuit design needs to be "flattened" before some simulation actions can be taken. This section shows how to flatten a design hierarchy from STOCK in order to facilitate the switch-level simulator in validating the circuit at the transistor level.

The whole process is divided into three steps which are shown as follows:

(1) from a schematic to the corresponding circuit network at the design level;

(2) from the circuit network to the corresponding transistor network; and, finally,

(3) from the transistor network to the corresponding network with numeric node names and a reference table.

Steps (1) and (3) are trivial; so, we skip them. Only Step (2) will be discussed. The input to Step (2) is a temporary file coming from Step (1) with the following format. Here, we use an example which is a 3-bit counter to explain the input and output relationship in design flattening.

A circuit netlist at the design level

```
nor2(jkmsff_0_4,jkmsff_1_4,nor2_3_2).
jkmsff(nor2_3_2,ck,nor2_3_2,cr,jkmsff_2_4,jkmsff_2_5,vdd).
jkmsff(jkmsff_0_5,ck,jkmsff_0_5,cr,jkmsff_1_4,jkmsff_1_5,vdd).
jkmsff(vdd,ck,vdd,cr,jkmsff_0_4,jkmsff_0_5,vdd).
```

Step (2) takes the above input and generates the following output which only contains CMOS transistors.

The corresponding circuit at the transistor level

```
nt(gnd,t_1_000_6,ck).
nt(gnd,t_2_002_0,vdd).
nt(gnd,t_2_019_0,ck).
pt(vdd,t_1_000_0,ck).
pt(vdd,t_1_000_1,vdd).
nt(t_2_001_1,t_1_000_1,q3).
pt(vdd,t_1_000_1,q3).
nt(t_2_002_1,t_1_000_2,t_1_000_3).
pt(vdd,t_1_000_2,t_1_000_0).
pt(vdd,t_1_000_2,t_1_000_3).
nt(t_2_003_1,t_1_000_3,cr).
...
```

The procedure at Step (2) can obtain the transistor-level description no matter what level the initial circuit description takes. Actually, this procedure has three parts. The first part, called *discover*, translates a circuit network into the next lower-level representation. The second part, called *examtr*, examines a network to determine whether it is only composed of transistors or not. The last part, called *manager*, executes *discover* several times

until the result passes the check provided by *examtr*. The following is the procedure *discover*.

```
Procedure discover
Input : A circuit network at any level
Output : The corresponding circuit network at the next low level
begin
 open Input file for READ;
 open Output file for WRITE;
 readline called component from Input file;
 while (not end_of_file(Input))
  begin
   separate component into component_name and arguments;
   open component_name file from STOCK for READ;
   readline called compt_low_level from component_name file;
   while (not end_of_file(component_name))
    begin
     separate compt_low_level into compt_name_low_level and
      arguments_low_level;
     change_name for arguments_low_level according to arguments;
     change_internal_name for compt_low_level;
     writeline called compt_low_level to Output file;
     readline called compt_low_level from component_name file;
    end;
   close component_name file;
   readline called component from Input file;
  end;
 close Input file;
 close Output file;
end.
```

The above algorithm assumes that a component file in the stock only contains the structure section to facilitate the description. *component_name* gives us the place to find the component. *Change_name* and *change_internal_name* recognize the symbols "%" and "@". They replace every I/O node by the corresponding name at the higher level. Internal nodes are renamed with new names which are the combinations of the iteration times of *discover*, the line number in the *component* file, and the sequence of the nodes. This method guarantees that any internal node is unique after flattening. The followings is the algorithm for the second part.

```
Procedure examtr
Input : A circuit network at any level
Output : Yes/No
begin
 open Input file for READ;
 readline called component from Input file;
```

```
 while (not end_of_file(Input))
  begin
   separate component into component_name and arguments;
   If (component_name not equal to (pt or nt)) return(No);
   readline called component from Input file;
  end;
 return(Yes);
end.
```

In the above algorithm, *pt* and *nt* are PMOS and NMOS transistors. Currently, they are the primitive components in STOCK.

```
procedure manager
Input : A circuit network at any level
Output : The corresponding transistor network
begin
 copy Input file to temp1 file;
 create temp2 file;
 repeat forever
  begin
   set input of discover to temp1;
   set output of discover to temp2;
   execute discover;
   set input of examtr to temp2;
   execute examtr;
   if (return(examtr) equal to Yes) break1;
   set input of discover to temp2;
   set output of discover to temp1;
   execute examtr;
   if (return(examtr) equal to Yes) break2;
  end;
 break1:
  begin
   copy temp2 file to Output file;
   return;
  end;
 break2:
  begin
   copy temp1 file to Output file;
   return;
  end;
end.
```

Procedure *manager* executes *discover* and then it uses *examtr* to decide the execution flow of the flattening procedure. The corresponding transistor network is generated after the execution of the above algorithms.

Chapter 4 Graphical Approach To Design Capture

This chapter discusses the approach adopted to build a universal schematic editor (USE) with friendly graphical interface. This tool is developed based on the important features of a circuit design, i.e., connectivity and hierarchy.

USE allows the designer to express a digital circuit at the same level that he/she thinks. The hierarchy information of a circuit is carried in STOCK. The output from USE is a high-level circuit description (netlist) which states the connectivity of a circuit design. Based on the generated netlist, a new component can be created. And down-stream work, such as implementation and/or verification, can be carried out. Moreover, USE is technology independent. It merely takes care of the component hierarchy and connectivity. The implementation or verification issues of a design are left for other tools. Hence, USE can easily cooperate with different kinds of implementation or verification tools through the use of a component database.

4.1 Overview

Before USE is discussed, we need to define the netlist of a circuit. A netlist is the representation of a component network. It carries all the information at the design level. In other words, the design endeavor during the creation phase is to generate such a net-list. Then, in the simulation or verification phase, the design effort is to modify the generated netlist in order to compliance with design specifications. Therefore, a design capture tool should possess a way to generate and modify a netlist with interactive visualization of a design.

Definition 4-1 (Netlist) : A netlist is used to represent a circuit design with the following format:

$C_1(t_{11}, t_{12}, t_{13}, ...)$.
$C_2(t_{21}, t_{22}, t_{23}, ...)$.
$C_3(t_{31}, t_{32}, t_{33}, ...)$.

...

Where $C_i, i>0$, are the names of the components, and $t_{ij}, j>0$, are the I/O nodes of C_i. All lower-level information which concerns of the components can be found in STOCK. The connection relationship among different nodes is represented by an identical name.

Based on this definition, a design process can be expressed as the work to create and modify such a netlist. To facilitate the work with interactive visualization, the graphics approach to design capture is preferred at structural level [23]. Figure 4-1 shows the editing process in USE. In order to create a human readable circuit schematic and obtain the netlist, a typical procedure is as follows:

(1) The designer acquires the necessary components from the stock. Then, he/she puts the components at the desirable places on the screen.

(2) The designer connects all the components to form a circuit. The appearance of wires are specified by the designer to increase the clearity for future modification.

(3) The designer marks some important nodes by giving them names. The timing of these nodes then can be observed later.

(4) A circuit schematic can be saved and loaded during the editing process.

(5) A design can be modified by adding or deleting components and drawing or erasing wires. It also can be packed and becomes a new component.

There are only two kinds of objects in USE, i.e., wires and components. However, since a schematic can become a component, USE can be used to create a circuit design at any structural level from the primitive level, e.g., transistors, resistors, and capacitors. The data strutures which can represent any shape of wires and components is described in the next section. Based on the structures, tasks for wiring and component editing are analyzed and developed.

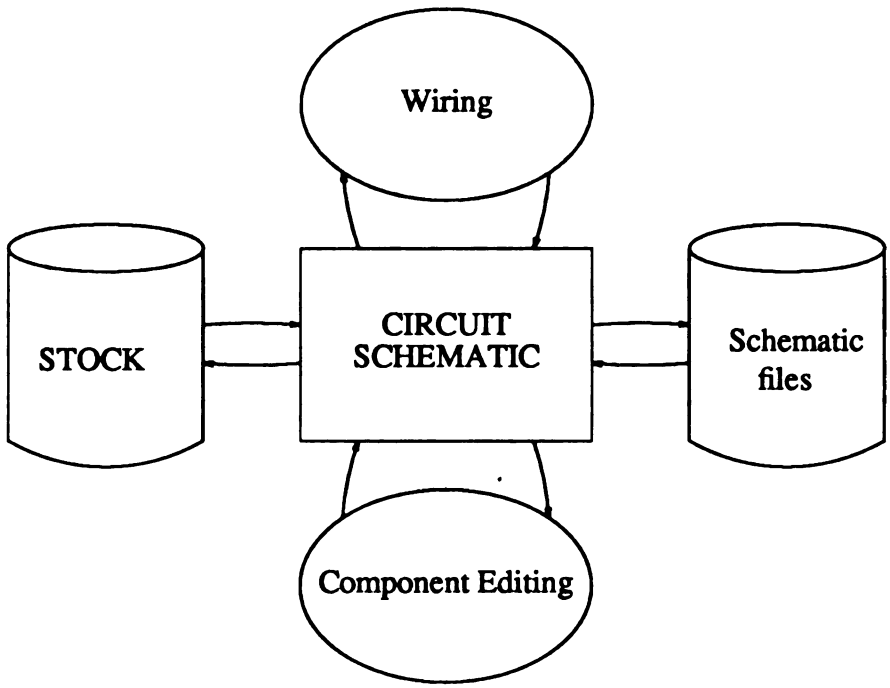


Figure 4-1 The editing process of a circuit schematic.

4.2 Data Structures

A circuit schematic is a two-dimensional diagram of variable size. The components can have all kinds of shapes. And wires have different lengths and can extend to everywhere just like a spider web. Figure 4-2 shows a typical circuit schematic. Since the data size is varying from one schematic to another, run-time memory allocation is a must and linked lists to represent components and wires are necessary. Based on the data structures discussed in this section, USE can easily and quickly draw any kind of circuit schematics.

Circuit Components

From a component point of view, USE is an interpreter which translates a set of primitive drawing commands into a graphic symbol on the screen. The drawing commands are defined in STOCK as a component attribute. USE also needs to know the I/O node locations in order to connect them. This information also can be found in STOCK.

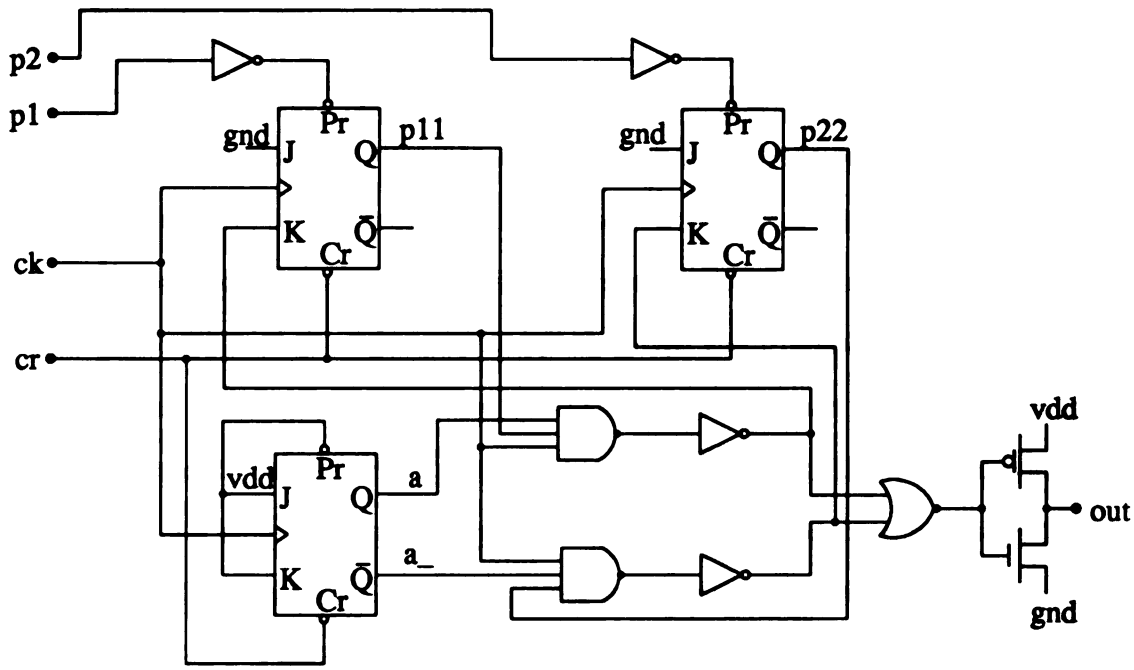


Figure 4-2 A typical circuit schematic.

In order to speed up the graphic operations, such as zoom, pan, and redraw, the drawing commands for all the same components are interpreted at most only once. The images of different components are saved and the fastest graphic operation, i.e., mapping, is used. In other words, the major graphic operations in USE are line drawing and memory mapping. Clearly, the speed is very high even on a small personal computer.

The components form a linked list with each element representing one component. Each element stores all the information of this particular component, including the location, the range, the label, the name, the drawing commands, and its I/O node information. The I/O nodes of a component form a linked list which is part of the component data structure. The images of the components also form a linked list. The component name is the key to match an image with a selected component. This approach which let many components share an image can reduce the run-time memory size significantly. Since two components in STOCK may have the same circuit structure but different graphic symbol, this approach does not put any limitation on drawings.

Wires

A "wire" is the object used to connect components. Those connections should be qualified from the designer's point of view. In the logic design level, wires carry only the information of connections. Other attributes, such as the dimensions and the materials, are not considered. In other words, we model a wire as an object which provides the connectivity among components to the designer. In order to create a designer-acceptable schematic, two types of wires are necessary, i.e., T-type and J-type wires. Any shape of a wire, no matter how complex it is, can be represented with these two types. A T-type wire is used to establish the first connection between two nodes. Then, J-type wires are used to connect a wire, which is a T-type or J-type, to other nodes. In other words, the terminals of a T-type wire are some I/O nodes of components. And, for a J-type wire, one terminal is an I/O node and the other terminal is on a T-type or J-type wire. Figure 4-3 gives an example, where node 1 to node 5 are connected together by a wire bundle. A "wire bundle" is the name we used to describe a wire with arbitrary shape. After the designer makes the connections, there is no need to distinguish which is the T-type wire and which are the J-type wires. Of course, from the designer's point of view, the distinction is not necessary at all as long as a wire bundle has been formed. A reasonable data structure is developed to maintain the connection relationship among nodes. Some obvious maintenance tasks are creating and deleting a wire bundle and disjoining a node from a wire bundle. The wire model with two different types has the advantage that it makes the internal data structure unified and simplifies the algorithm development.

A wire, either T-type or J-type, is formed by one or more line segments. A corner is the point where two line segments join with an arbitrary-degree angle. Certainly, 0, 45, and 90 degrees are commonly used in drawing a circuit schematic. A wire bundle also has a name used to identify it in order to search the corresponding data structure and then the maintenance tasks can be performed.

Based on the above description, a wire is specified by its name, type, terminals, and

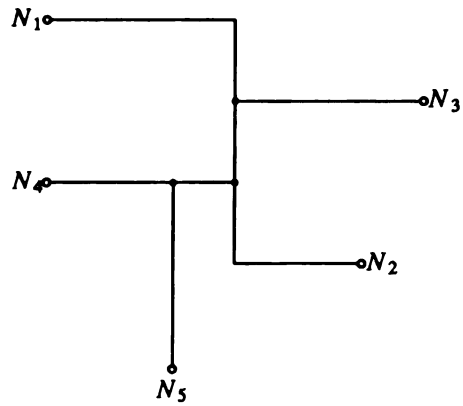


Figure 4-3 A typical wire bundle.

the corner locations. The joined degree of a corner is implicitly carried by the corner's location. A complex linked list is developed to represent wires. T-type and J-type wires are so similiar that the same data structure is used. Figure 4-4 and 4-5 illustrate our approach. In this figure, N represents a node, C represents a corner, and J represents a joint on the wire. Figure 4-4 says the wire bundle has a T-type wire and three J-type wires. The T-type wire, which has two corners, connects node N_1 and N_2. One J-type wire connects node N_3 and joint J_1, another J-type wire connects node N_4 and joint J_2, and the last J-type wire connects node N_5 and joint J_3. Hence, node N_1, N_2, N_3, N_4, and N_5 are connected together by this wire bundle. Joint J_1, J_2, and J_3 must be on the wire in other to show the conductivity of these nodes. The wire bundle which is represented in Figure 4-4 is shown in Figure 4-3.

Moreover, a hash table is used to increase the searching speed among all the wires. This table provides a link to some position of the wire list. The wire name decides the hash function. Hence, a wire is easy to find by its name.

4.3 Tasks

The design caputure process in USE is divided into three major tasks. There are component editing, wiring, and graphic issues. The first task deals with component creation, deletion, name assignment, etc. Wiring task deals with the creation and modification

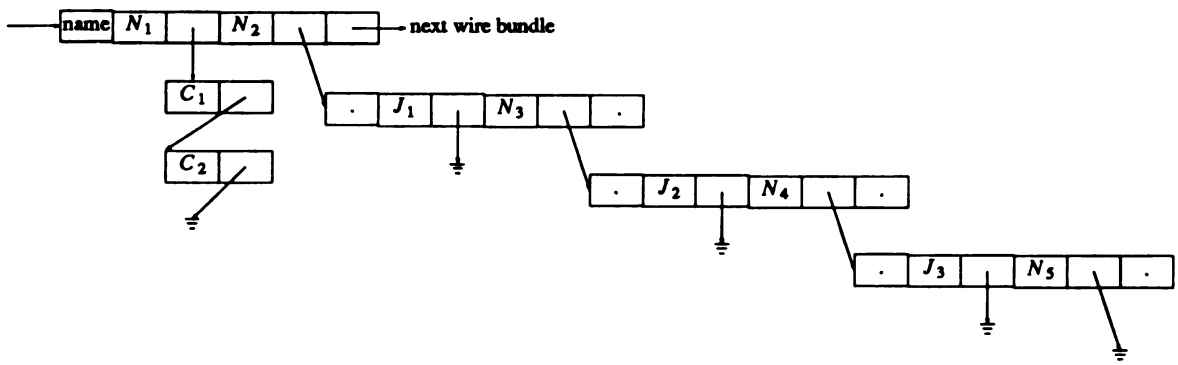


Figure 4-4 The data structure of a wire bundle.

of wire bundles. And, the last task makes USE friendly to use with vividly high-speed graphic operations.

Most of the algorithms which implement the commands in USE are trivial if the data structure is known. Therefore, only those with more significant meaning are discussed.

Node Disconnection

Modification of a schematic is the major activity during a circuit design. The kernel portion of this modification is to disconnect a selected node from its wire bundle. For example, to delete a component, all of its I/O nodes must be disconnected from the wires which connect to the component. And, this action must not affect the connectivity of other components. Since a wire bundle may be very complex in a large circuit schematic, the method to disconnect a node from a wire bundle needs to be carefully analyzed.

From the definition of the wire model, a connected node can only be at three positions, i.e., the start point of a T-type wire, the end point of a T-type wire, and the end point of a J-type wire. Let us call the start point of a T-type wire t(x0,y0), the end point of a T-type wire t(xn,yn), and the end point of a J-type wire j(xn,yn). Therefore, we can call the start point of a J-type wire j(x0,y0). Also, it is clear that the disjoined position must be at the beginning of some J-type wire. We call it dj(x0,y0) which is the nearest

joint to the disconnecting node along the wire bundle. After these definitions, we find that the wire from the node which needs to be disconnected to the disjoined position only has four possibilities :

(1) Part of a T-type wire from t(x0,y0) to dj(x0,y0);

(2) Part of a T-type wire from t(xn,yn) to dj(x0,y0);

(3) Part of a J-type wire from j(xn,yn) to dj(x0,y0);

(4) A complete J-type wire from j(xn,yn) to j(x0,y0).

Condition (4) is the simplest. We only need to take care of a J-type wire which has a complete data structure. For other conditions, this function not only needs to delete the unwanted piece of the wire but also needs to merge the J-type wire which starts from dj(x0,y0) into the rest part of the wire. Let us call the wire which contains the unwanted piece *wire1* and the wire starting from dj(x0,y0) *wire2*. Then, we can rewrite the above conditions into more executable forms:

(1) *Wire2* is reversed and merged into the rest of *wire1*. So, dj(x0,y0) becomes a corner and j(xn,yn) of *wire2* becomes the starting point of *wire1*.

(2) *Wire2* is merged into the rest of *wire1*. So, dj(x0,y0) becomes a corner and j(xn,yn) of *wire2* becomes the end point of *wire1*.

(3) Same as condition (2).

(4) *Wire2* is null. Hence, the whole *wire1* needs to be deleted.

The reason why *wire2* in (1) needs to be reversed is to meet the definition of wire structures. With the above analysis, no matter how complex a wire bundle is, any node can be disconnected efficiently.

Joint Verification

To create a J-type wire, the joint which is the starting point of the wire must be specified along a wire bundle. This specification must be verified to ensure the conduction property of a wire bundle. Hence, the connectivity of components is obtained. For

example, when the designer needs to connect a J-type wire, the first thing he/she needs to do is choose a node which has already been connected to some other nodes by a wire bundle. Then, he/she can make a connection from any place on the wire bundle to the desirable node. A J-type wire represents such a connection. In other words, the starting point of a J-type wire may be anywhere as long as this point is on the correct wire bundle. This operation needs to be confirmed in order to create a readable schematic and correct netlist. Since the wire model is a composition of many segments of lines, this checking is hierarchical. We check a wire bundle by checking each wire that belongs to the wire bundle and examine a wire by checking every segment which makes up the wire. To check a point whether it is on a segment, we use the following approach:

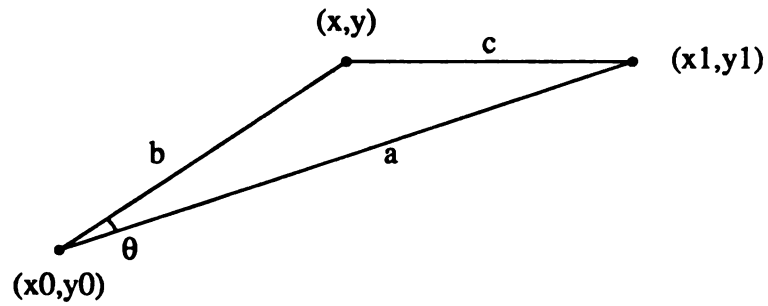


Figure 4-5 Joint Verification.

In Figure 4-5, a is the length of the segment; b is the distance between (x0,y0) and (x,y); c is the distance between (x1,y1) and (x,y). And, the angle θ can be obtained by the equation:

$$\theta = \cos^{-1}\left(\frac{a^2+b^2-c^2}{2ab}\right)$$

Then, the distance from the point to the segment is

$$h = b\sin\theta$$

After obtaining h, we can decide whether (x,y) is on the segment or not.

It is worth noting that this method is independent on the slope of the segment. Slope can not be used to implement this verification because the slope of a segment ranges from negative infinity to positive infinity. Hence, comparison with slopes can not give a satisfied solution. On the contrary, the method is universal for any kind of line segments. Actually, the above description is not completed. We also need to decide whether (x,y) is in the range of the segment, i.e., x is between x0 and x1, and y is between y0 and y1, before the above calculation can be performed.

Projecting function

USE does not put any limitation of the circuit size. Idealy, any size of circuit schematic can be created. In other words, the canvas to draw a schematic has no size limit. However, the only limitation is the capability of integer representation in the computer. For a 32-bit machine, the canvas size is $(2^{32}-1)\times(2^{32}-1)$ which is large enough to store any kind of circuit diagrams. Since the canvas is infinite, a projecting function must exist to map part of the canvas onto the graphic window.

Actually, the canvas does not exist. The data structure which represents a schematic is generated by projecting the coordinates on the graphic window onto the corresponding coordinates on the "virtual canvas". Therefore, all graphic operations operate according to the projecting function. The zoom operation is usually achieved by changing the zoom factor and the pan operation is achieved by changing the displacement. However, since there is no center point in an un-completed schematic, the zoom operation can move the schematic far away from the graphic window. In order to zoom a schematic without affecting its position on the graphic window, a displacement is needed before changing the zoom scale.

Let (R_x, R_y) be the coordinate on the virtual canvas and (W_x, W_y) be the corresponding coordinate on the graphic window. The projecting function f is expressed as follows:

$$R_x = \frac{(W_x - D_{2_x})}{Z} - D_{1_x};\quad R_y = \frac{(W_y - D_{2_y})}{Z} - D_{1_y}.$$

Where Z is the zoom factor. (D_{1_x}, D_{1_y}) and (D_{2_x}, D_{2_y}) are two displacement values, one is before and the other is after the zoom operation.

The reverse projecting function f^{-1} is easy to derive from f. f^{-1} is as follows:

$$W_x = (R_x + D_{1_x})Z + D_{2_x};\quad W_y = (R_y + D_{1_y})Z + D_{2_y}.$$

The projecting function provides a powerful mechanism which maps the internal data structure onto the graphic window which has a limited size and vice versa. As a result, the virtual canvas is almost infinite and schematics with arbitrary sizes can be created.

Chapter 5 Logic-Timing Simulation at Transistor-Switch Level

A switch-level logic-timing simulator with linear-time computational complexity called SWSIM is described in this chapter. MOS transistors are modeled as bidirectional switch-attenuators with ideal capacitors associated with the gate terminals. The simulating circuit is represented as a composite graph which consists of a set of node connected by transistor edges. During simulation, each transistor has a state (ON, 1/2ON, or OFF), and each node has a logic value and strength. The strength is used to establish the signal-flow direction during the simulation run. We developed a linear-time algorithm to evaluate the node strength. This technique ensures that the simulation of bidirectional transistors can be easily handled at the logic level.

5.1 Overview

Recent years, CMOS technology has boosted the development of Application-Specific Integrated Circuits (ASICs). Digital circuit design at the gate level has no longer met the requirements of designing an ASIC chip. The cost of fabricating ASIC chips is dominated by the silicon area occupied by the chips. It has been shown that such chips can have better performance and use less silicon area if they are implemented at the MOS transistor level. A typical example is the cache comparator design at MC68030 [35]. The designers implemented a series of exclusive-OR (XOR) gates for address comparison in an elegant way. This circuit was designed at the transistor level. If those gates were designed at the gate level, it would have cost much more area and had a lower operating speed.

Since a chip is designed at the transistor level, logic simulation at the same level becomes very essential to obtain accurate logic behavior of the chip. Hence, switch-level simulation for MOS circuits plays an important role in the field of digital ASIC design.

However, the bidirectional characteristic of MOS transistors deters the application of logic simulation theory at the transistor level. As a result, some researchers, such as Bryant [24], Schaefer [34], and Lengauer and Nä her [25], applied a circuit simulation technique which is used for obtaining the analog behavior of a circuit into a discrete set of data to obtain the digital behavior at the logic level. In Bryant's model, a wire is modeled as an input node or a storage node with different sizes to represent the effect of their relative capacitances in charge sharing. Transistors are modeled with different strengths and three states, i.e., open, closed, and indeterminate states. The different sizes of nodes can confuse a circuit designer since no capacitors appear in the circuit diagram. Thus, the simulation may be wrong if the simulator generates error node sizes. This may happen if a design has not been translated into a layout. In other words, this model forces designers to take care of the analog world while doing pure digital design at the transistor level. Inevitably, his model is quite different from that in the designer's mind. Moreover, the transistor states in his model are not sufficient even though transistors can have different strengths. For example, using an NMOS transistor in the closed state to pass VDD, the result is degraded by the threshold voltage, but PMOS transistors do not decrease the voltage. Therefore, in a closed state, both PMOS and NMOS transistors should produce different results when they pass VDD even though they have the same strength. This example also implies that the node states in Bryant's model, which represent low, high, and invalid voltages, are insufficient to characterize a digital circuit.

Besides the above problems, the simulator based on his model, called MOSSIM II, can not handle inputs with random timing and can not simulate self-oscillating circuits. (MOSSIM II was designed primarily for simulating clocked systems.) Even for some circuits, such as the XOR gate in Figure 5-1., it can not be simulated correctly. (But, his newest version of the simulator can handle this gate.)

In Schaefer's model, a transistor is modeled as a resistor and a non-input node is modeled as a capacitor. Apparently, Schaefer simplified the transistor model in the ana-

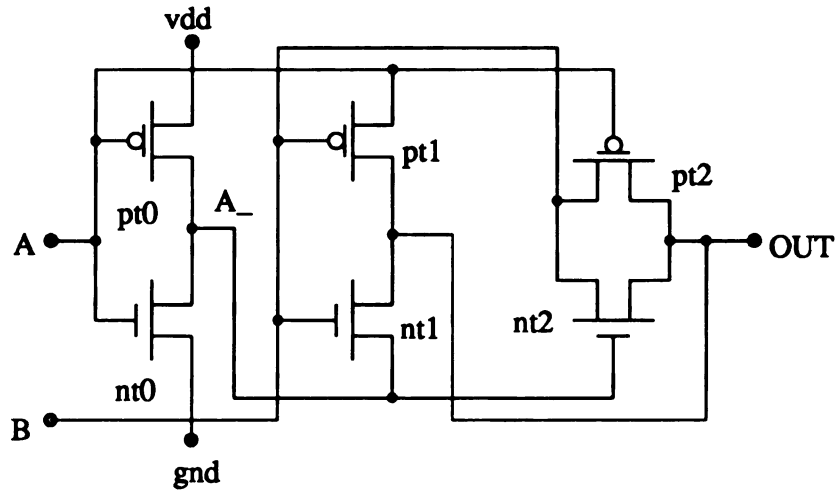


Figure 5-1 A two-input XOR gate.

log world in order to obtain faster simulation speed in circuit simulation. Therefore, the performance of his simulator is not comparable to the logic simulators even though the results are the same.

Since Bryant and Schaefer both applied circuit simulation techniques for switch-level simulation, their simulators are circuit-topology dependent and do not have linear-time complexity. This makes the use of switch-level simulations less attractive for large circuits.

However, SWSIM overcomes the difficulty of modeling bidirectional MOS transistors in the logic level. It is a pure logic simulator at the transistor level. Moreover, to prevent the NP-complete problem which occurs in the ternary logic simulation [26], SWSIM does not use any "valid but unknown" logic value. Hence, SWSIM obeys the law of Excluded Middle. This is because a prediction is made during initialization to get rid of any unknown logic value. For example, a latch in a circuit whose value can not be decided by the circuit input, such as a reset, may have either 1 or 0, but not unknown, after initialization. This prediction method correctly represents the power-on sequence of a circuit. As a result, SWSIM achieves the following goals:

(1) The computational complexity is linear with respect to the number of transistors for each simulation step.

(2) The signal-flow determination algorithm makes the simulation of bidirectional MOS transistors possible at the logic level.

(3) There is no restriction for input timing and circuit topology. Hence, any CMOS logic family and self-oscillating circuits can be simulated correctly.

(4) The transistor models predict sufficiently well the digital logic behavior of MOS transistors.

(5) The law of Excluded Middle is obeyed. No NP-complete problem occurs.

(6) The simulation speed is very fast. Hence, it is suitable for simulating an entire chip with more than 10,000 transistors.

In this chapter we first describe the circuit model and digital behavior of transistors. Next, the theory of signal-flow determination in digital CMOS circuits is stated. Then, the simulation theory is presented. Finally, the performance analysis and some key examples are given.

5.2 Circuit Model

This section describes the simulation domain of SWSIM. In this domain, only the digital behavior of CMOS circuits is concerned and modeled. We want to provide a well-defined area for circuit designers. In this area, we prove that SWSIM can simulate circuits well.

Definition 5-2-1 (Simulation domain) : The simulation domain of SWSIM is digital CMOS circuits which are composed of only MOS enhancement-mode transistors. Each NMOS (PMOS) transistor virtually has the same behavior.

Clearly, there is no circuit-topology restriction in Definition 5-2-1. Next, we define the meaning of logic values in SWSIM.

Definition 5-2-2 (Logic value) : Logic values are used to represent voltages in real circuits. Logic value **1** represents voltages from $VDD-V_T$ to VDD, where V_T is the threshold voltage of MOS transistors. And, logic value **0** represents voltages from *GND* to V_T. Finally, the logic value **x** is used to represent the high-impedance condition or voltage value between $VDD-V_T$ and V_T.

Based on this definition, we say that logic **1** is a 'perfect 1' or 'poor 1', and logic **0** is a 'perfect 0' or 'poor 0'. (A 'poor 1' corresponds to a voltage which approaches $VDD-V_T$, and a 'poor 0' corresponds to a voltage approaching V_T [10].) Let **r0**, **r1**, **p0**, and **p1** represent 'perfect 0', 'perfect 1', 'poor 0', and 'poor 1', respectively. Hence, logic **0** and **1** are external logic values which will be viewed in the output timing waveform, and **r0**, **r1**, **p0**, **p1** are internal logic values which SWSIM uses. **X** is used in both ways.

One interesting property of CMOS circuits is that many circuits utilize the gate capacitor for proper operation. We model these gate terminals as storage nodes which retain their values in the absence of inputs. This technique provides an ideal model for dynamic storage.

Definition 5-2-3 (Gate capacitor effect) : The gate capacitor of MOS transistors is an ideal capacitor, i.e., there is no leaking current during simulation time. As a result, a logic **1** or logic **0** can be stored at the gate of a transistor during a time period if and only if this gate is isolated during the period.

According to this definition, we introduce four other internal logic values ,i.e., **x0**, **x1**, **xp0**, **xp1**. The leading character **x** implies they are high impedance. Therefore, **x0** means the node is high impedance and stores an **r0**; **xp1** means the node is high impedance and stores a **p1**; etc. These high-impedance states are reported in the timing waveform as the logic value **x** which appears between **1** and **0**. Altogether, SWSIM has nine internal logic values and three external logic values, as shown in Table 5-1. This distinction is essential for generating easily readable timing waveforms without losing any serious information. Now, we define the MOS transistor models based on these logic

values.

Voltage level	Internal logic	External logic
VDD	**r1**	**1**
VDD-V_T	**p1**	
High-impedance or voltage between thresholds	**x, x0, xp0, x1, xp1**	**x**
V_T	**p0**	**0**
GND	**r0**	

Table 5-1 The logic representation of voltage values.

Definition 5-2-4 (P-switch) : A P-switch is used to model a PMOS enhancement-mode transistor. A P-switch is a perfect switch to pass logic **1** and a poor switch to pass logic **0**. The gate terminal is a storage node. The switch is ON if the value of the gate terminal is **r0** or **x0** , and it is 1/2ON if the gate terminal has a **p0** or **xp0**. If the gate terminal has a value **r1**, **p1**, **x1**, **xp1**, or **x**, then the P-switch is OFF.

Definition 5-2-5 (N-switch) : An N-switch is used to model an NMOS enhancement-mode transistor. An N-switch is a perfect switch to pass logic **0** and a poor switch to pass logic **1**. The gate terminal is a storage node. The switch is ON if the value of gate terminal is **r1** or **x1** , and it is 1/2ON if the gate terminal has a **p1** or **xp1**. If the gate terminal has a value **r0**, **p0**, **x0**, **xp0**, or **x**, then the N-switch is OFF.

When an NMOS (PMOS) is 1/2ON, it can pass an **r0** or **p0** (**r1** or **p1**), but not **r1** or **p1** (**r0** or **p0**). This situation models transistors as attenuators which degrade the signal voltage with an amount of V_T. Table 5-2 (5-3) states the NMOS (PMOS) transistor model based on the above definitions.

Definition 5-2-6 (Short-circuit effect) : If a node has **r1** and **r0** at the same time instant, **r0** is assigned to this node. In general, the priority of assignment is **r0** > **r1** > **p0** > **p1** > **x**. And **xp0**, **xp1**, **x0**, and **x1** are treated as **x**.

Transistor State	ON	1/2ON	OFF
Input value	Value passed		
r0	r0	r0	x
r1	p1	x	x
p0	p0	p0	x
p1	p1	x	x
x	x	x	x
x0	x	x	x
x1	x	x	x
xp0	x	x	x
xp1	x	x	x

Table 5-2 The NMOS enhancement-mode transistor model.

Transistor state	ON	1/2ON	OFF
Input value	Value passed		
r1	r1	r1	x
r0	p0	x	x
p1	p1	p1	x
p0	p0	x	x
x	x	x	x
x0	x	x	x
x1	x	x	x
xp0	x	x	x
xp1	x	x	x

Table 5-3 The PMOS enhancement-mode transistor model.

This definition represents the short-circuit effect. Whenever there is a conducting path (resistance = 0) from GND to a node, the node value is always **0**. This is true in the circuit theory. However, many logic simulators do not use this fact. Instead, they use a third value to represent this situation and claim the node has unknown value. Thus, an NP-complete problem occurs [26]. Since SWSIM deals with MOS transistors, many realistic properties of the elements are considered and adopted to avoid problems which can not happen is the real circuits. SWSIM ensures that a circuit which is built according to the definitions can be simulated correctly.

5.3 Signal-Flow Determination

To achieve this linear-time logic simulation at the transistor-switch level, determining the signal flow through transistors is an essential task. Some timing analysis programs, such as Crystal [27] and TV [28], require flow analysis first. Clocksin and Leeser [29] presented a method for automatically determination of signal flow. Their method only provides statistical analysis of the signal flow without considering the inputs. Thus, they label many transistors bidirectional even though they are not. In other words, their method only ensures the labeling of unidirectional transistors. Any transistor in which the signal flow can not be decided are labeled bidirectional. We introduce a method of dynamically determining the signal flow at each simulation step based on the current circuit condition. Thus, the signal flow of a given transistor may have different directions at different simulation steps. Moreover, the determination must be fast, otherwise the simulation performance will be seriously degraded.

To accomplish this, we give every node in a circuit a new attribute called strength. It can be thought of as a driving force and the difference of the forces between a transistor channel decides the signal-flow direction. We use a technique which is very similar to the depth-first search in the graph theory [30] for evaluating the strength of the nodes. If the source and drain node of a transistor have the same strength, the signal-flow direction is decided by the node values. Hence, we say the signal-flow direction of a transistor is a function of four arguments, i.e., the strength and the values of the non-gate nodes. Before defining the transistor graph, we define the strength in all input nodes. Here, we treat the power lines, VDD and GND, as input nodes for unifying descriptions.

Definition 5-3-1 (Input-node strength) : For any input node at a time instant, the strength of this node is fixed and defined as the node value a if $a \in$ {**r0, r1, p0, p1**}. Otherwise, the strength of the input node is **x**.

Since strength is defined according to the input-node values, the strength of a node also agrees with the short-circuit effect. This is to say if a node has been assigned two

different kinds of strength, the stronger strength will dominate the node.

Definition 5-3-2 (Transistor graph): A CMOS transistor circuit is a composite graph $G(V, E)$, where V is the set of all nodes and E is the set of all transistor channels between the non-gate terminals. The gate terminal of a transistor is not only a node but also a label for this transistor edge.

As an example, Figure 5-2 is the graph of the circuit in Figure 5-1. In this graph, every transistor is represented as an edge and the edge label. Hence, this transistor network forms a composite graph which is represented by three sub-graphs as shown in Figure 5-2.

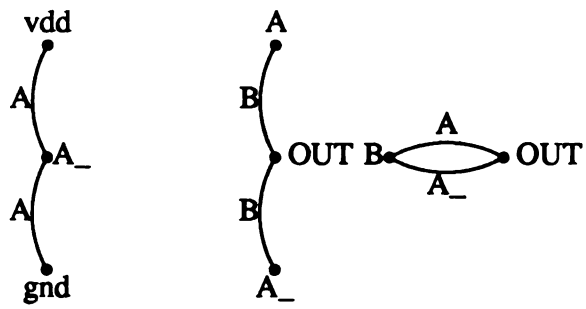


Figure 5-2 The graph representation of the circuit in Figure 5-1.

Clearly, given a graph $G(V, E)$, we can divide it into two graphs based on the transistor type. So, we have the following lemma:

Lemma 5-3-1 (P-graph and N-graph) : A transistor graph $G(V, E)$ is the union of a P-graph, $G_p(V_p, E_p)$, and an N-graph, $G_n(V_n, E_n)$, where $E_p(E_n)$ is the set of all PMOS(NMOS) transistors.

Proof : The proof follows directly since the primitives are only PMOS and NMOS transistors. □

It is the first level in the hierarchy to distinguish a general transistor graph $G(V, E)$. Clearly, extracting $G_p(G_n)$ from G is the same as separating $E_p(E_n)$ from E since $E_n \cap E_p = \varnothing$.

During simulation, each node has a strength at each time instant, and a transistor state is decided by the value in its gate node. According to Tables 5-2 and 5-3, a transistor graph can be further divided into 5 intrinsic graphs. This is the second level of the hierarchy. For example, an intrinsic graph $G_{r0}(V_{r0},E_{r0})$ is a graph where V_{r0} is the node set with strength **r0** and E_{r0} is a set of transistors which can merely pass **r0**. Clearly, E_{ro} (E_{r1}) can not contain a PMOS (NMOS) transistor.

Proposition 5-3-1 (Intrinsic graph) : A transistor graph $G(V,E)$ is divided into five intrinsic graphs. They are $G_{r0}(V_{r0},E_{r0})$, $G_{r1}(V_{r1},E_{r1})$, $G_{p0}(V_{p0},E_{p0})$, $G_{p1}(V_{p1},E_{p1})$, and $G_x(V_x,E_x)$. Let $a \in$ {**r0, r1, p0, p1, x**}; then, $G_a(V_a,E_a)$ is an intrinsic graph for a. Every node belonging to V_a has a strength a and every transistor in E_a has a state which can pass a.

Proof : We can find an intrinsic graph $G_a(V_a,E_a)$ by searching from all the input nodes with strength a. If a transistor can pass a, we collect it in E_a and put its non-gate terminals in V_a. If a transistor decreases the strength by the threshold value, we put the node which has the weak strength in the input-node list for other searches. So, G_a is formed after the search is done. And, since **p0** (**p1**) strength can be generated from **r0** (**r1**) through a PMOS (NMOS) transistor but a **p0** (**p1**) can not generate a **r0** (**r1**) strength during a search. It is essential that the process starts by finding G_{r0}, then G_{r1}. In other words, the search should start from the input nodes with stronger strength, since it may generate poor strength for some nodes. □

Figure 5-3 shows the hierarchy of a general transistor graph. Note that the intrinsic graphs may overlap each other. And a node may belong to more than one node sets. However, the strongest strength wins the competition for the final value.

The search for finding the intrinsic graphs implies that the strength of all nodes can be found after the short-circuit effect is applied to decide the final strength of the nodes. Hence, the short-circuit effect decides not only the node value during simulation but also

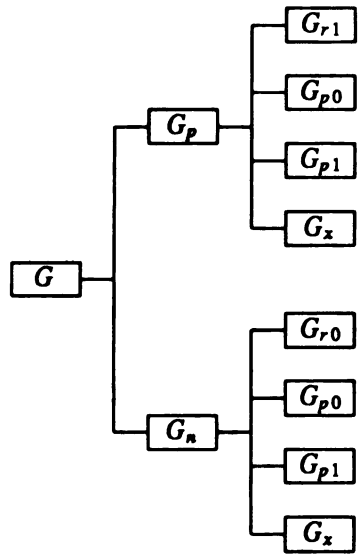


Figure 5-3 The graph hierarchy of a general transistor graph.

the node strength during signal-flow determination. It is a very important feature in SWSIM.

Proposition 5-3-2 (Strength competition) : Given a non-input node N and two different intrinsic graphs, $G_a(V_a,E_a)$ and $G_b(V_b,E_b)$, let $N \in V_a \cap V_b$. If $a > b$, then b has no effect on the strength of N.

Proof : Clearly, this property coincides with the short-circuit effect described earlier. If node N only appears in V_a and V_b, the strength is a because a has a stronger strength. However, if node N is also contained in other intrinsic graphs, the current strength a still needs to compete with others. □

Proposition 5-3-1 and 5-3-2 give a method of finding the strength of all nodes in a general transistor graph. Note that the strength of the input nodes can not be changed in any condition. Based on the strength information, the signal-flow direction of a transistor is decided as follows:

Definition 5-3-3 (Signal-flow determination) : The signal-flow direction of a non-OFF transistor is decided by four parameters. They are the strengths and values of the non-gate nodes. Let the transistor have two non-gate terminals, T_1 and T_2, $S()$ be the strength, and $V()$ be the value of its argument which is a node. Table 5-4 lists the determination.

Condition	direction
$S(T_1) > S(T_2)$	$T_1 \rightarrow T_2$
$S(T_1) < S(T_2)$	$T_1 \leftarrow T_2$
$S(T_1)=S(T_2)=V(T_1)=V(T_2)$	$T_1 \leftarrow\rightarrow T_2$
$S(T_1)=S(T_2)=V(T_1)\neq V(T_2)$	$T_1 \rightarrow T_2$
$S(T_1)=S(T_2)=V(T_2)\neq V(T_1)$	$T_1 \leftarrow T_2$

Table 5-4 Signal-flow determination of a non-OFF transistor.
(Conditions at higher columns have higher priority.)

The above definition is obtained based on circuit theory. For other conditions which did not list in Table 5-4, the values at T_1 and T_2 become **x** at the next simulation step since the previous values are not supported by their strength.

5.3.1 Strength Determination Algorithm

In the above section, we translated the problem of signal-flow determination into a strength evaluation problem. The strength evaluation problem can be solved by finding those intrinsic graphs and applying the short circuit effect. Here, we present an algorithm describing the process and prove that it is of linear-time complexity with respect to the number of transistors in the graph.

This algorithm called the Strength-Determination Algorithm (SDA) is divided into five parts. It finds all the intrinsic graphs and solves the competition of node strength at the same time. The input to this algorithm is a general transistor graph and all nodes have their values at some time instant. The output is the strength information of all nodes in the graph.

It is reasonable that we assume the input values are members of {**r0, r1, x**} before we start to describe this algorithm. The SDA is as follows :

Strength-Determination Algorithm

(1) Set the strength of all nodes **x**.
(2) Let T be an input node with value **r0**. Execute the procedure *decide_r0_strength(T)*.
(3) Repeat (2) until all input nodes with value **r0** have been used.
(4) Let T be an input node with value **r1**. Execute the procedure *decide_r1_strength(T)*.
(5) Repeat (4) until all input nodes with value **r1** have been used.
(6) Exit.

Since we restrict the input values, the strength **p0** (**p1**) is only generated by PMOS (NMOS) transistors if they pass **r0** (**r1**). Therefore, *decide_r0_strength()* will invoke *decide_p0_strength()* if the condition mentioned above happens. The following are the procedures which really decide the node strength.

decide_r0_strength(T_1)

(1) Let T_2 be a non-input node and connected to T_1 with a transistor edge Tr. If $Tr \in G_n$, its state is ON or 1/2ON, and $S(T_2) <$ **r0**, then let $S(T_2) =$ **r0** and execute *decide_r0_strength(T_2)*.
(2) If $Tr \in G_p$, its state is ON, and $S(T_2) <$ **p0**, then let $S(T_2) =$ **p0** and execute *decide_p0_strength(T_2)*.
(3) Repeat from (1) to (2) until all transistors with one non-gate terminal connected to T_1 have been visited.

decide_p0_strength(T_1)

(1) Let T_2 be a non-input node and connected to T_1 with a transistor edge Tr. If $Tr \in G_n$, its state is ON or 1/2ON, and $S(T_2) <$ **p0**, then let $S(T_2) =$ **p0** and execute *decide_p0_strength(T_2)*.
(2) If $Tr \in G_p$, its state is ON, and $S(T_2) <$ **p0**, then let $S(T_2) =$ **p0** and execute *decide_p0_strength(T_2)*.
(3) Repeat from (1) to (2) until all transistors with one non-gate terminal connected to T_1 have been visited.

decide_r1_strength(T_1)

(1) Let T_2 be a non-input node and connected to T_1 with a transistor edge Tr. If $Tr \in G_p$, its state is ON or 1/2ON, and $S(T_2) <$ **r1**, then let $S(T_2) =$ **r1** and execute *decide_r1_strength(T_2)*.

(2) If $Tr \in G_n$, its state is ON, and $S(T_2) <$ **p1**, then let $S(T_2) =$ **p1** and execute *decide_p1_strength*(T_2).

(3) Repeat from (1) to (2) until all transistors with one non-gate terminal connected to T_1 have been visited.

decide_p1_strength(T_1)

(1) Let T_2 be a non-input node and connected to T_1 with a transistor edge Tr. If $Tr \in G_p$, its state is ON or 1/2ON, and $S(T_2) <$ **p1**, then let $S(T_2) =$ **p1** and execute *decide_p1_strength*(T_2).

(2) If $Tr \in G_n$, its state is ON, and $S(T_2) <$ **p1**, then let $S(T_2) =$ **p1** and execute *decide_p1_strength*(T_2).

(3) Repeat from (1) to (2) until all transistors with one non-gate terminal connected to T_1 have been visited.

The above procedures are very similiar to each other with the principal differences among them being the transistor types and states. Clearly, they all have the same computational complexity. After the strength of all nodes is obtained, the signal-flow direction is determined by applying Definition 5-3-3. Then the simulation theory, which will be discussed in the next section, can be applied to the transistor network. Next, we prove the time complexity of the above procedures is linear.

Proposition 5-3-3 (Time complexity) : Given a transistor graph $G(V,E)$, the computational complexity of the strength-determination algorithm is $O(|E|)$.

Proof : For a transistor $Tr \in E$, let T_1 and T_2 be the non-gate nodes of Tr. Also, let the number of visits to Tr be vs. If Tr is in the OFF state, $vs = 0$. If $Tr \in G_p$ and it is ON (1/2ON), we have $vs \leq 3(2)$. Because Tr can pass **r1, p0, p1** when it is ON. Therefore, the maxinum number for visiting Tr is 3. If $Tr \in G_n$, we still have $vs \leq 3$ since Tr can pass **r0, p0, p1** when it is ON.

Thus, it follows that the upper bound of the number of visits to Tr is three. This is true for all the transistors in $G(V,E)$. So, the number of visiting the transistors by the algorithm is bounded by $3|E|$. Hence, the time complexity is $O(|E|)$. □

After the signal-flow directions of a transistor network are decided, the theory for

logic simulation can be applied to evaluate the circuit. Note that signal-flow determination must be done before running each simulation step. Fortunately, the speed of the singal-flow determination is linear and so fast that the influence to the overall performance is small.

5.4 Simulation Theory

Here, we present the simulation theory used in SWSIM. Based on the theory, circuits can be simulated with linear-time complexity. However, this simulation theory can not handle a primitive component which has bidirectional characteristics. In other words, this theory can not be applied to an element where its input and output nodes are not uniquely defined. This is not a problem in SWSIM since the signal-flow directions are decided before each simulation step runs.

Generic model for digital circuit

This generic model can be used for simulating digital circuits at any logic level. It describes a circuit as a network of basic building elements. There are no restrictions on the structure of the network. The BBEs in a network can have arbitrary connections in their input/output terminals. For example, two BBEs in a network can have common outputs, feedback connections, and/or cascade connections. However, the input and output nodes of a BBE must be known before running each simulation step.

Definition 5-4-1 (Basic Building Element): The basic building elements (BBEs) are the most primitive components modeled in a digital network. The outputs of a BBE are solely decided by its inputs, i.e., BBEs are combinational circuit building blocks. Let a BBE have n inputs and m outputs whose inputs are $i_1, i_2, ..., i_n$ and outputs are $o_1, o_2, ..., o_m$ (see Figure 5-4). We have

$$v'_{t+t_d}(o_j) = f_j(v_t(i_1), v_t(i_2), ..., v_t(i_n)), \quad 1 \le j \le m$$

where the notation is as follows: $v_t(i_1)$ is the value of node i_1 at time t; $v_t(i_2)$ is the value

of node i_2 at time t, etc.; $v'_{t+t_d}(o_j)$ is the value of node o_j at time $t+t_d$ which is driven by this BBE. The functions f_j, $1 \le j \le m$, describe the behavior of the BBE. f_j can be a formula in two-valued Boolean algebra, three-valued ternary algebra, or a general mathematical function in a high-level representations.

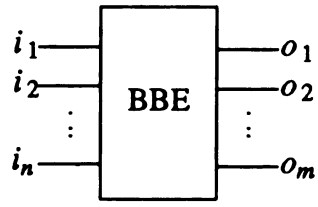


Figure 5-4 The general diagram of a basic building element.

Definition 5-4-1 assumes that all kinds of BBEs have the same delay time, t_d. The outputs in a BBE at the next time step are governed by the inputs in the present time step. And, for later use, we define a procedure f_{BBE} to calculate all f_j, $1 \le j \le m$, at time t.

$$f_{BBE}: \text{ for } (j=1 ; j \le m ; j++) \quad v'_{t+t_d}(o_j) = f_j(v_t(i_1), v_t(i_2), \ldots, v_t(i_n)) ;$$

The output of f_{BBE} is a set of all output node-value pairs.

Definition 5-4-2 (Uniform System): If a circuit can be decomposed into a number of BBEs which conform to Definition 5-4-1, then the circuit is called a uniform system regardless of the circuit's topology.

This definition says that a uniform system consists of BBEs only. A uniform system may have different types of BBEs. The structure of a uniform system is less important. Hence, a uniform system can have arbitrary connections among the BBEs.

Definition 5-4-3 (Node): The input and output terminals of a BBE are called nodes. Nodes are the connecting wires in a uniform system. A node only has a unique value for an instant in time.

The above definition states the attribute of a node. Nodes may have other attributes. Since the interconnection of a uniform system is not concerned, other attributes of a node, such as input and feedback, are unimportant.

Definition 5-4-4 (State): The state $S(U, t)$ of a uniform system U at a time instant t is a set of all node-value pairs in U.

$$S(U, t) = \{<d_1, v_t(d_1)>, <d_2, v_t(d_2)>, ..., <d_k, v_t(d_k)>\},$$

where U has a total of k nodes. $<d_j, v_t(d_j)>$, $1 \leq j \leq k$, are all 2-tuples whose first element is a node name and the second element is the node value at time t.

According to the above definition, the state of a uniform system, $S(U, t)$, completely describes the state of U at time t.

Definition 5-4-5 (Excitation): An excitation $Ex(U, t)$ of a uniform system U at time t is a set of all input node-value pairs in the system U, i.e., an $Ex(U, t)$ gives all input values to the system U at time t. Formally,

$$Ex(U, t) = \{<i_1, v_t(i_1)>, <i_2, v_t(i_2)>, ..., <i_p, v_t(i_p)>\},$$

where the system U has p input nodes.

Clearly, the excitation of a system can be changed each time when a new simulation process runs. $Ex(U, t)$ defines the values of the input nodes at time t and those input values are at least fixed during the time interval $[t, t+1)$. For example, the power supply to a uniform system is treated as one of the elements in the excitation set. The value of the power node is a constant through the whole simulation run.

Lemma 5-4-1 : $S(U, t)$ contains $Ex(U, t)$, where $S(U, t)$ is the state of a uniform system U at time t and $Ex(U, t)$ is an excitation to the system U at time t.

Proof : $S(U, t)$ is the set of all node-value pairs and $Ex(U, t)$ is the set of all input node-value pairs in U at time t. Since input nodes are part of all nodes in a system, we have that $S(U, t) \supset Ex(U, t)$. Notice that $S(U, t)$–$Ex(U, t)$ is not obtained from $Ex(U, t)$. □

Now, a potential problem arises. How does one decide the value of a common output node from several outputs of connected BBEs? A common output node is the node where outputs of more than one BBEs join. This feature is called bus connection. Here,

we define a competition function to evaluate the real value of a common output node. In real circuits, a competition function may vary to suit different technologies. For example, TTL gates with wired logic may use an AND operator to define the competition function when TTL circuits are modeled. On the contrary, wired-OR logic is used for ECL gates to implement the competition function. However, a general definition is as follows:

Definition 5-4-6 (Competition function): If a node d is a common output node of several BBEs, say q, in a uniform system, then the node value $v_t(d)$ is decided by a function called the competition function *Comp*. Formally,

$$v_t(d) = Comp(v_1', v_2', ..., v_q'),$$

where v_1' is the output value from one of the connected BBEs, v_2' from another connected BBEs, etc. There is no limitation to the number of arguments in a competition function. These arguments compete together to decide the actual value of the common output node d at time t. Of course, no competition function is needed, if a system has no common output nodes. This is sometimes true in high-level circuit descriptions. For convenience, a competition function is sometimes denoted as *Comp* (...), since the number of its arguments may vary.

The above definitions depict the generic model of a digital circuit. Obviously, this model is easy to adjust for different levels of circuit description. Speed and accuracy in the simulation task depend on the resolution of a specific model.

Simulation theory for the generic model

As with other simulation models, this simulation performs a set of steps. Each step is one unit delay time t_d. There are no changes in a uniform system during the time interval $(t, t+t_d)$. The simulation process drives the states of a uniform system U from the initial state $S(U, 0)$ to the next state $S(U, 1)$, then $S(U, 2)$, $S(U, 3)$, ..., etc. From those states, values of all nodes are obtained. They provide the transient response of the system. The law of transition in a uniform system is presented here. We prove that all uni-

form systems follow the transition law. Also, the computational complexity of the simulation theory is given.

Suppose we already know the state of a uniform system U at time t. This means we can use that information to calculate f_{BBE} for all the BBEs in U. Then, we can apply the competition function to all the common output nodes in order to obtain the real values of those nodes. Hence, all node values except the input nodes in U at time $t+t_d$ are obtained. Based on the above description, we define a procedure $f_U(S(U,t))$ to evaluate the node values at time $t+t_d$. Let a uniform system U have b BBEs (i.e., $BBE_1, BBE_2, ..., BBE_b$) and c common output nodes (i.e., $d_1, d_2, ..., d_c$). The procedure is as follows:

$f_U(S(U,t))$: { for (j=1; $j \le b$; j++) evaluate f_{BBE_j} ;

for (j=1; $j \le c$; j++) $v_{t+t_d}(d_j) = Comp(...)$; }

f_U is called the behavior of U. It depends on the relationship among the embedded BBEs and the technology used to implement the system. The output of $f_U(S(U,t))$ is defined as a set of all node-value pairs except the input nodes in U at time $t+t_d$. Also, a valid initial state can be obtained by assuming t_d=0 and evaluate $f_U(S(U,0))$.

Definition 5-4-7 (Transition Law): The law of transition

$$S(U, t+t_d) = f_u(S(U,t)) \cup Ex(U, t+t_d),$$

where $S(U, t+t_d)$ is the state of U at time $t+t_d$; $f_U(S(U,t))$ is the behavior of U; $S(U,t)$ is the current argument of f_U; and $Ex(U, t+t_d)$ is an excitation of U at time $t+t_d$.

The transition law says that a new state can be derived from an existing state and a new excitation. A state at time $t+t_d$ can not be obtained unless an excitation at the same time is also known. The transition law is the main principle in the simulation task. It gives the method to predict the feature of a uniform system at $t+t_d$ from the current time t. Actually, the transition law is very similar to the state equation of a linear, time-invariant, discrete-time dynamical system in the control theory [31].

Theorem 5-4-1: Every BBE behaves according to the transition law.

Proof: For a BBE with n input nodes, $i_1, i_2, ..., i_n$, and m output nodes, $o_1, o_2, ..., o_m$, let $Ex(BBE, t)$ and $Ex(BBE, t+t_d)$ are excitations of the BBE at time t and $t+t_d$, respectively. At time t, the state of the BBE is

$$S(BBE, t) = Ex(BBE, t) \cup \{<o_1, v_t(o_1)>, <o_2, v_t(o_2)>, ..., <o_m, v_t(o_m)>\},$$

where

$$Ex(BBE, t) = \{<i_1, v_t(i_1)>, <i_2, v_t(i_2)>, ..., <i_n, v_t(i_n)>\}.$$

Let f_{BBE} represent the behavior of the BBE, we use $S(BBE, t)$ as the argument to evaluate f_{BBE}. Formally,

$$f_{BBE}(S(BBE, t)) = \{<o_1, v_{t+t_d}(o_1)>, <o_2, v_{t+t_d}(o_2)>, ..., <o_m, v_{t+t_d}(o_m)>\}$$

Hence, we obtain

$$S(BBE, t+t_d) = f_{BBE}(S(BBE, t)) \cup Ex(BBE, t+t_d).$$ □

Theorem 5-4-2: Let system W be composed of two systems U and V such that the connection between U and V is arbitrary. Moreover, the following are true:

(a) U and V follow the transition law;

(b) all of the BBEs in U and V have the same delay time;

(c) a competition function exists in W.

Then W also follows the transition law.

Proof: $S(W, t)$ can be obtained from $S(U, t)$, $S(V, t)$, and the competition function of W. Since U and V follow the transition law, we have

$$S(U, t+t_d) = f_U(S(U, t)) \cup Ex(U, t+t_d)$$
$$S(V, t+t_d) = f_V(S(V, t)) \cup Ex(V, t+t_d)$$

Then we can apply the competition function of W among the common output nodes in these two sets $S(U, t+t_d)$ and $S(V, t+t_d)$. Hence, all node values except the input nodes of W are obtained. This procedure is exactly the same as evaluating $f_W(S(W, t))$. Hence, we have

$$S(W, t+t_d) = f_W(S(W, t)) \cup Ex(W, t+t_d),$$

where $Ex(W, t+t_d)$ is an excitation of W at $t+t_d$.

Clearly, $Ex(W, t+t_d)$ is a subset of $Ex(U, t+t_d) \cup Ex(V, t+t_d)$. □

Theorem 5-4-3: A uniform system always follows the transition law if a competition function exists.

Proof : Partition a uniform system into individual BBEs, then hierarchically apply Theorem 5-4-1 and 5-4-2. We have

$$S(U, t+t_d) = f_U(S(U, t)) \cup Ex(U, t+t_d),$$

which is always true at all levels. □

This theorem establishes the principle for simulating a uniform system which is governed by the transition law.

Theorem 5-4-4: The computational complexity in simulating a uniform system for one step is linear with respect to the number of BBEs in the system.

Proof: For one simulation step, let the time needed to calculate the behavior of a BBE in a uniform system is T_{BBE}, i.e., we spend T_{BBE} to evaluate f_{BBE} for one BBE. Let the time needed to compute the competition function for a common output node is T_{Comp}. Although T_{Comp} may vary for the number of arguments in the function, the worst case can be used to estimate T_{Comp}, i.e., there is always an upper bound for T_{Comp} in terms of the maximum number of connected BBEs in a common output node. Therefore, for a uniform system U with b BBEs and c common output nodes, $f_U(S(U, t))$ can be evaluated in time:

$$bT_{BBE} + cT_{Comp} \leq (b+c)Max.(T_{BBE}, T_{Comp})$$

Clearly, c is decided by the connecting topology of the BBEs and is bounded by the maximun number of input/output nodes of a BBE and b. Hence, the computational complexity is O(b).

□

In the above proof, we neglected the time needed to obtain an excitation. This is because it is a very small fraction of the total simulation time, since the number of inputs in a system is usually much smaller than the number of BBEs.

5.4.1 Multiple-Delay Model

In the previous section, we established the simulation theory for uniform systems. However, using uniform systems to model digital circuits does not complete the task. One obvious problem is the following. Due to the various load of the components in a digital circuit, the delays usually are very different among the basic building elements. Although the unit-delay approach is good to verify a circuit topology, it leaves too much work in the lower level design. Circuit designers still need to decide the size of all BEEs based on the output loads or fan-out numbers. In some worse case, buffers may be needed in order to drive the next stage.

The above problems occurring in the unit-delay simulation can be solved if the building blocks of a digital circuit can be modeled as multiple-delay elements. Thus, a functional block at high level description, such as an arithmetic logic unit, can be modeled as a building element with its own delay value. And since the same elements in a circuit with different delays can be modeled, the simulation results are more accurate than those for unit-delay simulation.

Based on the above discussion, we present a multiple-delay model for the building elements in digital circuits. Then, we will prove that the transition law is still valid in such circuits.

Definition 5-4-8 (Building Element): The building elements (BE) are the components in a digital circuit with the following features. A BE may have more than one input and output. Each output has its own delay value. Let a BE have n inputs and m outputs whose inputs are $i_1, i_2, ..., i_n$ and outputs are $o_1, o_2, ..., o_n$. And, the delay of an out-

puts o_j is T_{dj}, where $1 \le j \le m$. Then we say the value of o_j at time $t+T_{dj}$ is solely decided by the inputs at time t, i.e.,

$$v'_{t+T_{dj}}(o_j) = f_j(v_t(i_1), v_t(i_2), \ldots, v_t(i_n)), \qquad 1 \le j \le m$$

Where the notation follows Definition 5-4-1 except T_{dj}. Now, T_{dj} is the delay time which only associates with the output o_j.

Based on the above definition, we realize that a BE is a multiple-delay component and can be used to model any digital component above the transistor level. The output values are decided not only by the input value but also by the delays which may vary among these outputs. By evaluating the functions f_j, $1 \le j \le m$, we obtain all the output values of a BE. These values are at different discrete time instants because each output has its own delay. However, according to the transition law, the simulator needs to obtain all of the node values at the next time step before it can move on to the next step. Therefore, we need to define a mechanism for each output o_j which can not only record $v'_{t+T_{dj}}(o_j)$ but also provide $v'_{t+1}(o_j)$ to the simulator. We called the mechanism *Delay Ring*.

A delay ring is a number of storage cells which record some simulation results during a time interval. These cells form a ring, as shown in Fig. 3. Let an output o_j of a building element in a digital circuit has delay time d, then the delay ring which associates to o_j has d cells. Each delay ring has two pointers, one called Producer and the other called Consumer. The producer is used for writing the value $v'_{t+d}(o_j)$ into the ring. The consumer is used for reading the value $v'_{t+1}(o_j)$ from the ring. Before the simulation goes to the next step, these two pointers needs to move to the next cells respectively. Clearly, a delay ring is a data structure to hold the simulation results of a output node from the next time step to the time step $t+d$, where t is the current time and d is the delay of the output node. And the two pointers are dedicated to ring operations.

Let $D_{op}(o_j)$ be the delay ring operating function of o_j. We describe $D_{op}(o_j)$ as the following procedures:

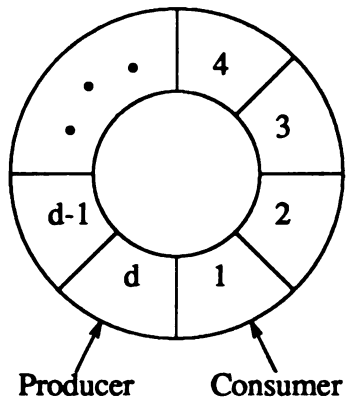


Figure 5-5 The structure of a delay ring.

(1) Evaluate $v'_{t+d}(o_j)$ by the input/output definition function f_j.

(2) Store $v'_{t+d}(o_j)$ at the location pointed to by the producer of the delay ring.

(3) Read $v'_{t+1}(o_j)$ from the location pointed to by the consumer of the delay ring.

(4) Move the two pointers one step counterclockwise.

(5) Return the value read, i.e., $v'_{t+1}(o_j)$.

Briefly, $D_{op}(o_j)$ stores the new simulation result, which will be used after the delay time, and returns the value for next simulation step.

Now, we can define the evaluating function for a given building element called f_{BE}. The f_{BE} is defined as the following executing sequence:

$$f_{BE} : \text{for}(j=1; j \le m; j++)\ v'_{t+1}(o_j) = D_{op}(o_j);$$

The output of evaluating f_{BE} is a set of all output node-value pairs at the next time step.

The delay ring operating function of a BE makes the BE acting as a basic building element. Hence, the law of transition can be applied to a network which is composed of components with different delay values. The simulation time step is the smallest delay among all the components in the network. As a result, circuits which contain elements from different levels in the circuit-description hierarchy can be simulated.

5.4.2 Tuning for CMOS Digital Circuits

Now, we tune the generic model to satisfying the CMOS circuit technology. First, we model PMOS and NMOS transistors as unit-delay BBEs.

Proposition 5-4-1 (Transistor model) : The PMOS and NMOS transistors are BBEs in a CMOS digital circuit.

Proof : Since we represent a CMOS circuit as a transistor graph, the strength-determination algorithm can be applied to evaluate the node strength. And, according to Definition 5-3-3, the signal-flow direction of a transistor can be decided. Hence, the transistor model shown at Table 5-2 (5-3) can be used to calculate f_{BBE}, where *BBE* is a transistor. □

Since a transistor can be modeled as a BBE, a CMOS digital circuit is a uniform system. Therefore, the transition law can be applied for simulation. However, a competition function specified for CMOS circuits still needs to be determined.

Proposition 5-4-2 (Competition function for CMOS circuits) : The CMOS competition function is dominated by the short-circuit effect.

Proof : The proof for this proposition is trivial since it follows the circuit theory. □

In order to obey the law of Excluded Middle, SWSIM uses a prediction method during initialization. At the begining, all nodes have the value of unknown. The unknown value at the gate node of a PMOS (NMOS) transistor turns it ON (1/2ON). Such a PMOS (NMOS) transistor is used to pass an **r1** (**p0**). Then, the initialization proceeds until a reasonable state is reached. A reasonable state of a circuit is a state which all the node values agree with the circuit behavior. This state is used as the state at the time instant 0 for later simulation. According to the competition function, an **r1** (**p0**) can be replaced by an **r0** (**r1**). Therefore, if the predictions are wrong for some nodes, they will be corrected

in the next initialization steps. As a result, no latches in a circuit can have any value other than logic 0 or logic 1. The simulation sequence of transistors determines the real initial value of a latch if it can not be evaluated by the circuit input.

Clearly, the initialization algorithm is very similiar to the strength-determination algorithm. Hence, it is also of linear-time complexity for each initialization step.

In summary, the generic model presented in this section captures the unification properties of digital circuits at any logic level. The time complexity of simulation was proven to be linear with respect to the circuit size. The drawback of this model at the transistor level is overcome by the signal-flow determination algorithm. At the end of this section, we tuned the generic model for application to digital CMOS circuits. Based on this model, SWSIM was implemented.

5.5 Performance Analysis

Here, we use one type of the circuits in the quick simulator benchmark [32] to analyze the performance of SWSIM. For the linear feedback shift registers (LFSR) at the first level in this benchmark, we choose N=10 and M=7, where N is the total number of stages and M is the feedback stage. Hence, we have 428 transistors to form a building block at the second level. (Refer to Greer's paper [32] for the circuit structure.)

Table 5-5 shows the simulation data. The circuit names also represent the hierarchical structures of those LFSRs. For example, R-7-4 is an LFSR which has 7 stages at the second level and 4 stages at the third level. The first level structure is an LFSR with 10 D-type flip-flops and the feedback comes from the 7th flip-flop. Clearly, this is the structure of R-1. The second field in the table is the total number of transistors in these circuits. Each circuit was simulated 5 times. The simulation period was 5 clocks with each clock = 5000 steps, where a step is defined as the unit delay time. The results are listed from the third field to the seventh filed in the table. Finally, the last field is the average time per clock. The time unit is in seconds.

Circuit name	#Tr	1	2	3	4	5	seconds/clock
R-1	428	6.8	6.6	6.7	6.5	6.6	1.33
R-2	856	9.3	9.3	9.0	8.8	8.7	1.80
R-3	1284	13.2	12.9	12.8	12.7	12.7	2.57
R-4	1712	16.4	16.2	16.1	15.9	16.1	3.23
R-5	2140	19.7	20.0	19.7	19.8	19.9	3.96
R-6	2568	22.9	23.3	23.3	22.9	23.3	4.63
R-7	2996	27.2	27.0	26.9	27.3	26.7	5.40
R-7-2	5992	48.5	48.3	48.4	48.5	47.9	9.66
R-7-3	8988	71.7	71.7	71.8	71.9	71.8	14.36
R-7-4	11984	96.2	95.8	96.3	96.2	96.0	19.22
R-7-5	14980	120.8	120.8	120.6	120.5	120.4	24.12

Table 5-5 Performance analysis of SWSIM on a SUN-3 workstation.
(Each run took 5 clocks with 1 clock = 5000 steps.)

Figure 5-6 shows the graph obtained from Table 5-5. SWSIM is demonstrated to be of linear-time complexity by this graph. Moreover, SWSIM can simulate many CMOS circuits which can not be simulated properly in other simulators. The examples in the next section demonstrate some of them.

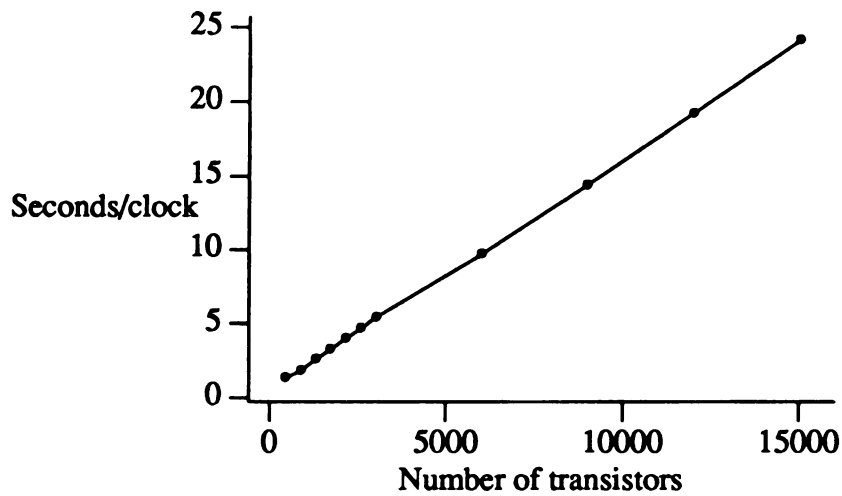


Figure 5-6 The performance of SWSIM on a SUN-3 workstation.

5.6 Key Examples

This section shows some representative circuit examples simulated in SWSIM.

Both the circuit diagrams and their timing results are listed.

Delay Demonstration

Figure 5-7 is a NOT gate with its input *A* connected to several pass transistors. When the *Control* signal goes high, the effect of *In* will be seen at the *Out* node after 5 unit time. However, when *Control* goes low, those pass transistors are turned off. Node *A* becomes high-impedance after one unit delay time. The capacitor associating with *A* still let *Out* high. The timing diagram is shown in Figure 5-8. This example shows the delay calculations in SWSIM.

Another example, as shown in Figure 5-9, demonstrates the effect of Schmitt trigger feedback. Without the feedback transistors, the delay time should be double. Figure 5-10 is the timing diagram. Some switch-level simulators mentioned in the Trimberger's book [33] can not deal with this circuit.

Clocked CMOS Logic

This example shows a 2-phase static D flip-flop. Figure 5-11 illustrates a circuit which is built from several transmission gates and inverters. The structure of transmission gates and inverters is omitted since they are well known. The timing diagram for the specific inputs is given in Figure 5-12. One may find this circuit in the Weste and Eshraghian's book [10]. Although the timings of C and $\overline{C}$ looks unsynchronized, their "effect" is synchronized. This is because the gates connected to $\overline{C}$ store the previous value.

Pass Transistor Logic

The circuit in Figure 5-1 is an XOR gate made by some pass transistors. Figure 5-13 shows its timing diagram.

Dynamic CMOS Logic

We use a 4-bit barrel shifter as an example. Figure 5-14 shows the circuit. Figure 5-15 is the timing diagram for all possible inputs. L_i are the inputs, S_i are the control signals, and R_i are the output nodes where $0 \leq i \leq 3$.

Self-oscillating Circuits

Here, we use a cascade of three inverters to demonstrate the simulation of self-oscillating circuits. Since each inverter is composed of a PMOS transistor and a NMOS transistor, the delay of each inverter is a unit time. Figure 5-16 is the circuit and Figure 5-17 is the timing diagram.

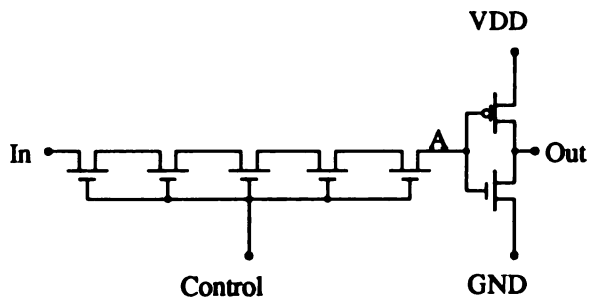


Figure 5-7 Delay demonstration using pass transistors.

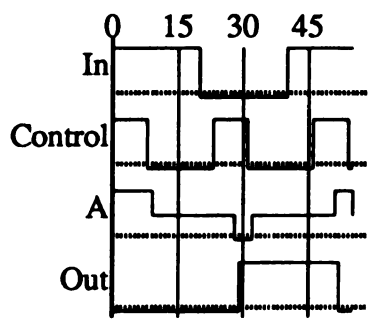


Figure 5-8 A timing diagram of the circuit in Figure 5-7.

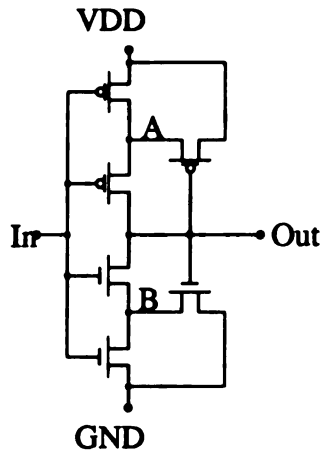


Figure 5-9 An inverter with Schmitt trigger feedback.

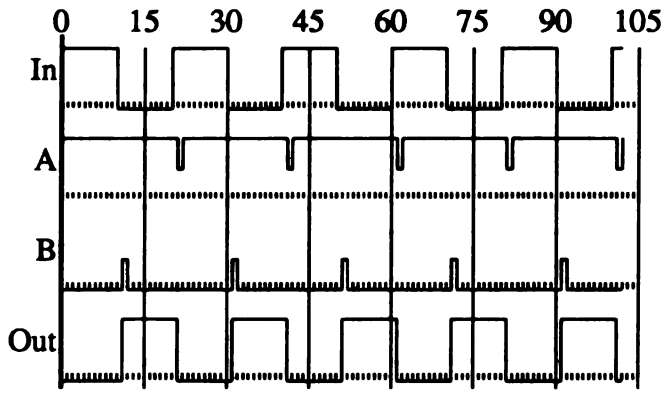


Figure 5-10 A timing diagram for the circuit in Figure 5-9.

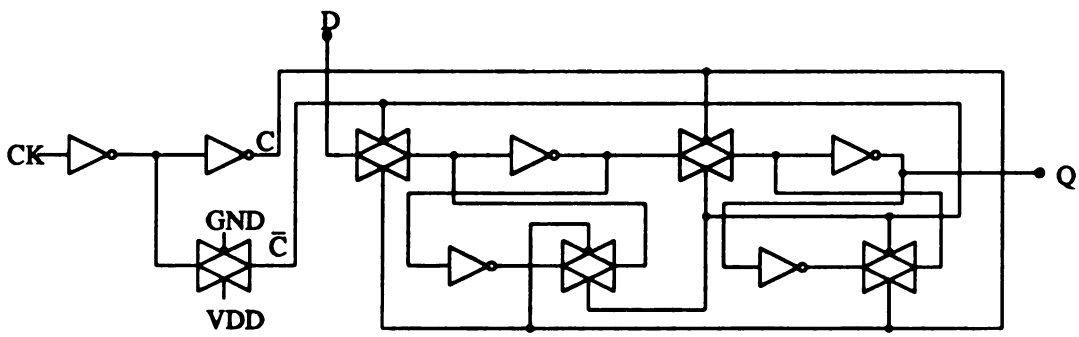


Figure 5-11 A dynamic D-type flip-flop.

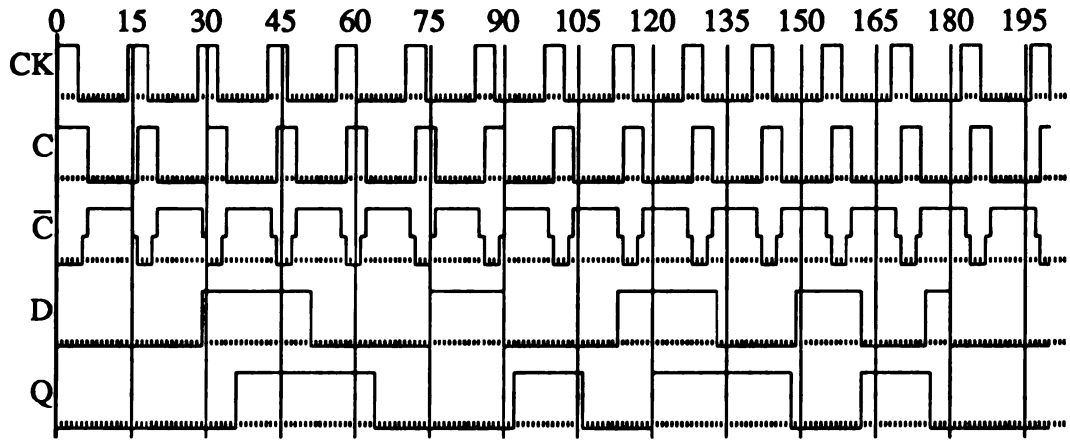


Figure 5-12 A timing diagram for the circuit in Figure 5-11.

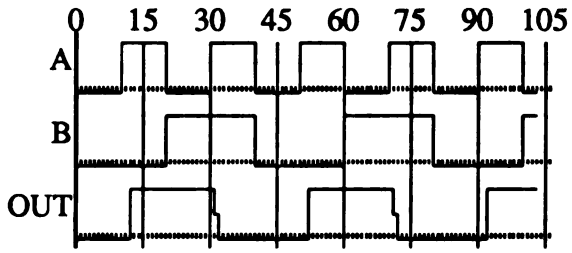


Figure 5-13 A timing diagram for the circuit in Figure 5-1.

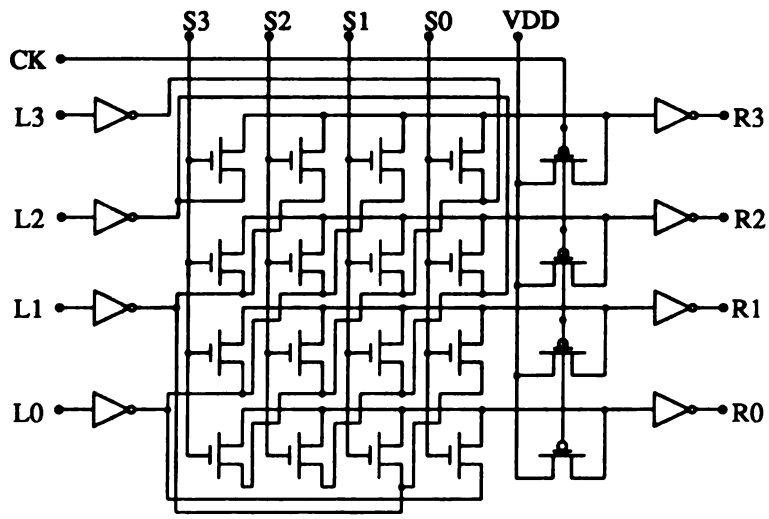


Figure 5-14 A 4-bit barrel shifter.

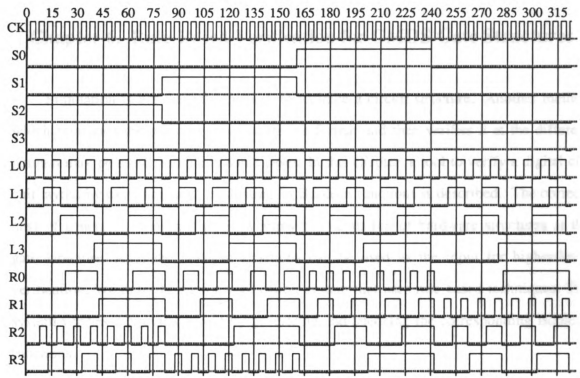


Figure 5-15 A timing diagram for the circuit in Figure 5-14.

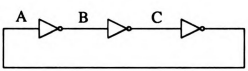


Figure 5-16 A self-oscillating circuit.

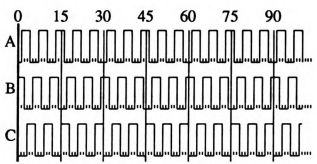


Figure 5-17 A timing diragram for the circuit in Figure 5-16.

Chapter 6. Rule-Based Verification for CMOS Gate Structures

Simulation is not the only method, to validate a circuit structure. Another method which translates the structure into a different format and then verifies it at the different domain may be more suitable in some cases, In this chapter, a tool to verify a digital circuit layout by extracting and evaluating its Boolean functions is described. The correctness of generated Boolean functions imply the validation of hardware structures at the gate structure. And, the Boolean functions can be used as the input for higher-level verification. We describe such a verification system that uses rule-based techniques. The system verifies a circuit's layout by generating and checking the corresponding Boolean functions.

6.1 Overview

Circuit verification of a VLSI chip layout is one crucial step in the custom-oriented design process. It is the designer's responsibility to ensure the validation of the circuit structures. VLSI circuits are notoriously "unforgiving" since any "unconscious" error in the physical layout can make the prototype unworkable. Unfortunately , a VLSI prototype is almost uncorrectable nowadays. This is the feature which makes verification play an important role in the VLSI design process.

In general, circuit verification can be cataloged into two fields, i.e., static and dynamic. Furthermore, static circuit verification can be divided into two hierarchical levels. The first level is the verification of geometric dimensions of physical layout, known as design rule checking [36]. Usually, design rule checking is implemented in graphic layout editing systems, such as in the MAGIC VLSI layout tool [37]. It can check the physical layout during the interactive layout-editing period and make sure that there are no violations against the fabricating resolution in the geometric dimensions. The second level of static verification is used to guarantee the correctness of higher level circuit

structures. For example, the layout of a NAND gate must not only obey the design rule checker in the layout system but also must be functionally correct.

The work done in this research falls into this latter category. But, instead of using binary patterns to verify MOS digital circuits, we use Boolean expressions to do functional verification at the gate level. We select this approach because two disadvantages exist in the current approach: First, if there are many input lines, functional verification requires the generation of a comparable number of output values. And the same problem exists for circuit testing. Specifically, for N inputs, 2^N testing patterns are needed. And, second, there is no way of extracting information regarding the structure of the target circuit. Of course, hardware verification at the level that is higher than the gate level also can be achieved if the hardware structures are specified or standardized.

The last kind of circuit verification is dynamic, which means that the verification involves another dimension, i.e., timing. Timing verification can decide the speed of a circuit, solve run-time bugs, etc. Since Boolean algebra has no timing relations among variables, dynamic circuit verification is beyond the scope in the chapter.

The purpose of this tool is to verify a circuit layout at the transistor level. The method we use here can also be used to synthesize a combinational circuit from its Boolean equation. Given a Boolean equation, better circuit performance is usually expected if we implement the equation at the transistor level. After a circuit layout is created by following our synthesis method, the best way to verify it is using a symbolic verification technique. In comparing our approach to that of others [39] [40], ours is more friendly and easier to use.

We adopt Prolog [4] to implement our system based on the following rationale: First, the topic has a well-defined domain. Second, circuit structures may be represented simply in Prolog. And, finally, Prolog provides a powerful capability for symbolic processing.

This verification system provides the following information about the circuit: First, for each gate in the target circuit, the system generates a Boolean expression to describe the gate. And, second, the interconnection of all gates in the target circuit are checked. Two basic, but important features are short-circuit checking and functional-completeness checking. Short-circuit checking can prevent an inadvertent conducting path from *vdd* to *gnd* through a transistor network. Functional-completeness checking ensures that there is only one unique output value for each input pattern. Actually, all necessary logic information, except the timing information, can be obtained by manipulating the results. Hence, hardware structures can be verified statically.

Figure 6-1 illustrates the verification system. A VLSI layout of a circuit is extracted and the corresponding Boolean expression with AND, OR, and NOT logic primitives is generated for each gate in the circuit. Two basic checks are performed in this phase to verify the gate structures. The circuit domain is restricted here to CMOS complementary logic [10]. Some complex structures, such as a PLA, can also be transformed into Boolean expressions. After phase I, a set of Boolean expressions is obtained which describes the target circuit. It carries all necessary logic information needed for high-level circuit structure verification. A good example of the verification is the implementation of a logic-level simulator with the following primitives: AND, OR and NOT.

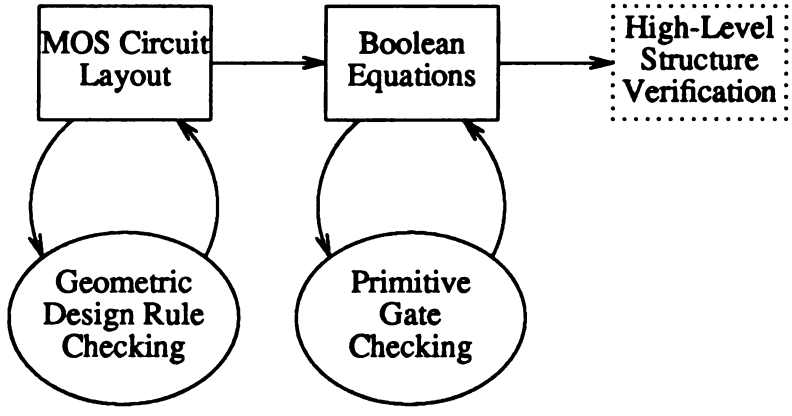


Figure 6-1 The rule-based approach for digital circuit verification.

Figure 6-2 shows the concept of a hierarchical verification system. At the lowest level, a geometric design rule checker is used to guarantee the correctness of physical dimensions of a circuit layout. At the next level, logic information is extracted from the layout, and the correctness of each gate is verified. Next, high-level hardware structures can be checked, e.g., flip-flops and registers. And, finally, the whole circuit can be verified based on those recognized structures. This tool focuses on the gate-level verification.

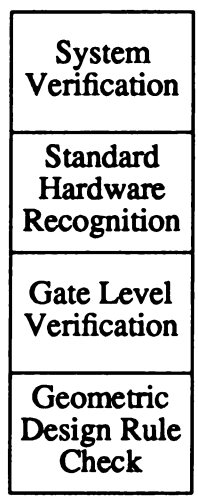


Figure 6-2 The hierarchical verification system.

6.2 The Knowledge Domain

In this section, models are described for digital MOS circuits, including connecting wires, circuit components, and logic gates. These models serve as the basis for formally describing a circuit's layout in the rule-based verification system.

(1) Connecting Wires

In the real world, each connecting wire in a circuit has its own resistor and capacitor values, and the dimension of a wire may change the behavior of a high-speed circuit. But, when we are only concerned about a circuit's static behavior, a connecting wire may be regarded as a variable in a Boolean expression, and each wire may be assigned a unique

name in a specific circuit.

(2) Circuit Components

Only transistors are considered to be active components in our circuit representation. They are the logic primitives at the lowest level. Boolean models for MOS transistors are established in the next section. In the abstract physical layout of a MOS circuit, a transistor is made by covering a diffusion layer with a polysilicon layer. This geometric information must be mapped into models for standard circuit components, such as transistors, resistors, and capacitors, before verification takes place. Since Boolean algebra has no direct relationship to resistors and capacitors in the MOS digital circuits, we assume all circuit components, except transistors, can be ignored. This means that only the transistor types and connections are what we need at the lowest level.

(3) Gate Structures

There are many kinds of gate structures in the MOS circuits. CMOS complementary logic, NMOS logic, dynamic CMOS logic and pass-transistor logic are typical examples [10]. A VLSI circuit designer may put more than one kind of gate structures into a design. This makes verification more complex. As a starting point, CMOS complementary logic is chosen as the circuit structure domain in our system. We choose this because this MOS implementation technology contains all of the information required to fully describe a circuit's interconnection topology. Consequently, it is not too difficult to recognize any other standard gate structures from the represented interconnection. The knowledge of generating a Boolean expression, which corresponds to a specified CMOS complementary gate, is described in the next section.

Using this approach, some higher level circuit representations can also be recognized and extracted to do circuit layout verification. A typical example is a simple latch composed of two feedback-connected NAND gates. Since there are a variety of high-level structures, we leave this for future extensions of this work. This feature will lead

the verification system into a specific tool which has expertise to verify some kinds of circuit components with well-defined complex structures.

6.3 The Boolean Model

The following rules give the Boolean model of CMOS complementary gates. Models for other types of gate structures can be developed in the same manner. This model can not only be used to verify a layout but also can be used to implement a Boolean equation at the transistor level.

Rule 1 (Transistor Definition): A transistor, $Tr(s,d,g)$, is defined as a path from s to d and the path is controlled by g, where s is the source terminal, d is drain, and g is gate. Tr is a variable and can take on the symbolic values nt or pt which define the transistor type as being N-diffusion or P-diffusion, respectively.

Rule 2 (Function Node Definition): A function node is defined as an output node of a gate. For two different transistors, $nt(s_1, d_1, g_1)$ and $pt(s_2, d_2, g_2)$, if d_1 equals d_2 or g_2 then d_1 is a function node. Or, if g_1 equals d_2 or g_2 then g_1 is a function node.

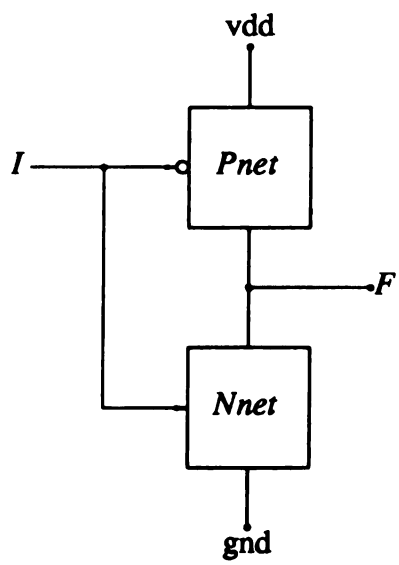


Figure 6-3 The topology of a CMOS gate structure.

Rule 3 (Transistor Network Definition): A *Pnet* (P-transistor network) is defined as a network between *vdd* and a function node. An *Nnet* (N-transistor network) is a

network between *gnd* and a function node. Thus, A *Pnet* and an *Nnet* with the same function node form a gate (see Figure 6-3). A gate has a "unique" Boolean equation which is obtained from its *Pnet* and *Nnet*. Let f be the equation and all inputs to the gate is a vector I. Then, from the *Pnet*, we have

$$f = B_{Pnet}(\bar{I})$$

And, from the *Nnet*, we have

$$\bar{f} = B_{Nnet}(I)$$

Where B denotes a Boolean equation derived from its subscript and its arguments are the items in the equation.

Rule 4 (NOT ($\bar{}$) Operator) : For a $pt(s,d,g)$, if it is a necessary path from *vdd* to a designate function node F then $\bar{g}$ will show up in the Boolean expression of F. In other words, a P-transistor conducts when $\bar{g}$ is present. And an N-transistor, $nt(s,d,g)$, conducts when g is present.

Rule 5 (AND (.) Operator) : If two transistors, $Tr_1(S_1,D_1,G_1)$ and $Tr_2(S_2,D_2,G_2)$ have the relationship Tr_1=Tr_2 and S_1=D_1 then the gates G_1 and G_2 are connected by an AND operator. Therefore, we define a conducting path in a transistor network (*Pnet* or *Nnet*) as a path from *vdd* (or *gnd*) to a function node F. Hence, a conducting path of F is a minterm in the Boolean expression of F. If there are more than one of the same gates existed in a path, they are reduced. Only one gate per item is allowed in a given path. This is the first minimization work.

Rule 6 (OR (+) Operator) : A transistor network for a function node F may have more than one conducting path. Each path is combined together by the OR operator. A path may be a subset of other paths. This implies that some minimization work can be done in this rule. Only the paths which are subsets of other paths need to be taken into account. By applying Rule 4, 5 and 6, the system can derive a Boolean expression with NOT, AND and OR primitives from a transistor network.

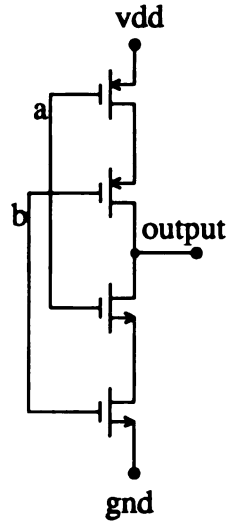


Figure 6-4 An un-complete gate.

Rule 7 (Completeness Checking) : For a given function node F, logic 1 is obtained from the Pnet and logic 0 from the Nnet. For all possible input patterns to the gate, a logic value of F should be obtained. In other words, F should be completely defined in terms of mapping every input pattern (see Figure 6-4 and Figure 6-5). To check a gate is complete, we first apply De Morgan's rules to the equation $B_{Nnet}(I)$ and then use maxterm decomposition rules to convert the equation from product-of-sum form into sum-of-product form. During the decomposition, we minimize those intermediate equations by adopting the following rules:

$$a + \bar{a} = 1$$
$$a \cdot \bar{a} = 0$$
$$a + 1 = 1$$
$$a \cdot 1 = a$$
$$a + a = a$$
$$a \cdot a = a$$
$$\bar{\bar{a}} = a$$
$$a + 0 = a$$
$$a \cdot 0 = 0$$

The formula used to minimize a Boolean expression

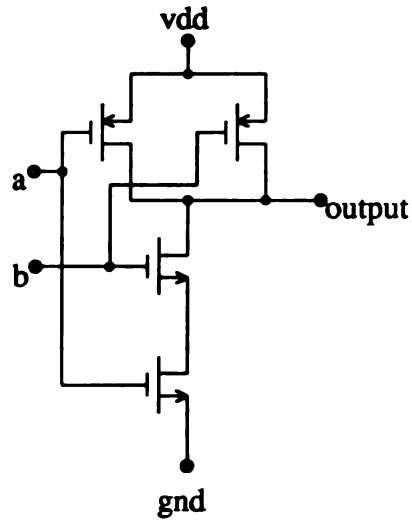


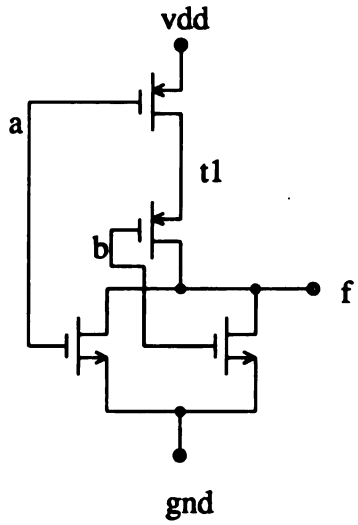
Figure 6-5 A complete gate (NAND).

Rule 8 (Short-Circuit Checking) : A path which conducts from *vdd* through a function node to *gnd* is not allowed. But, if it happens, then a short circuit exists in the gate which owns the path. Specifically, if one of the minterms which is obtained from the Pnet of a gate is the same as a minterm in the Nnet then there is a short circuit in that gate. This kind of error may be discovered by the completeness checking but is more time-consuming.

6.4 Implementation in Prolog

Prolog is a language with very complex building functions, such as unification of variables, different kinds of tree manipulations, and database (facts and rules) management [38]. Circuits are represented in Prolog as a set of facts. Each fact represents an elementary circuit component. Figure 6-6 shows a NOR gate representation. Clearly, facts in a circuit representation give all the structural information needed to generate its Boolean equations. Rule 1 in the last section provides the definition of transistors' facts. Hence, we can define our program as a mapping mechanism which maps a CMOS digital circuit from transistor structures into a set of syntactic Boolean equations.

Next, the function node definition becomes a rule in Prolog. Based on this rule, all



The representation of the NOR gate:

pt(vdd,t1,a).
pt(t1,f,b).
nt(gnd,f,a).
nt(gnd,f,b).

The corresponding Boolean equations:

$$f = \bar{a}+\bar{b},\ \bar{f} = a \cdot b;$$

Figure 6-6 A NOR gate representation.

gates in the target circuit can easily be separated. Then the program recursively processes each function node to find its corresponding Boolean equation. A similar strategy can be applied to NMOS circuits because there is always a load transistor in order to form a gate.

After finding a function node, the system defines the Pnet and Nnet which form the gate. Thus, the problem is reduced to a single-gate problem. In CMOS theory, a Pnet is used to produce a logic 1 and an Nnet to produce a logic 0. Therefore, a Boolean equation, which generates positive logic, can be obtained form the Pnet. And a Boolean equation which generates negative logic can be obtained from the Nnet. So, the target gate is represented by two complementary Boolean equations. For each input to the gate, either the Pnet or Nnet should generate a high-impedance output, but not both. This attribute is

examined by short-circuit checking and functional-completeness checking. The first type of checking takes the minterms which exist in the complementary logic equations to compare with in order to prevent a short path. Functional-completeness checking applies De Morgan's rules, maxterm decomposition and reducing rules mentioned above to the equations. Then, the results are compared with their corresponding complementary equations. Even today, it is still crucial to compare with two Boolean equations in an acceptable execution time. However, the method we have adopted in completeness checking can perform well, since the equations are generated by the hardware structures. Here, we show an example to illustrate our minimization technique. The Boolean equation we want to minimize is

$$z = a\cdot\bar{b}\cdot c + \bar{a}\cdot b\cdot c + a\cdot b\cdot c + \bar{a}\cdot\bar{b}\cdot c$$

After we apply De Morgan's rules to $\bar{z}$, we have

$$\bar{z} = (\bar{a}+b+\bar{c})(a+\bar{b}+\bar{c})(\bar{a}+\bar{b}+\bar{c})(a+b+\bar{c})$$

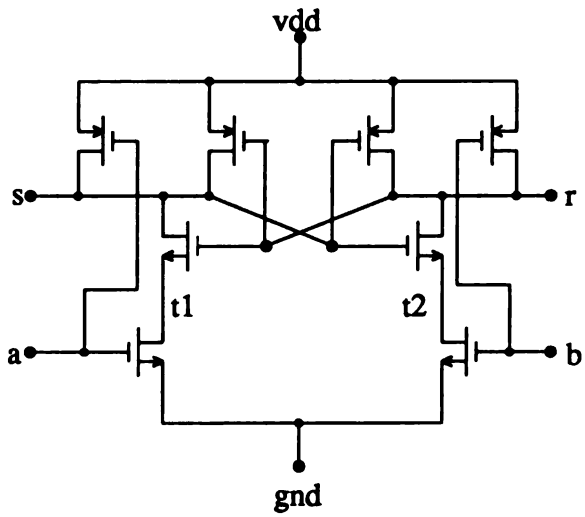
Then, we apply maxterm decomposition rules to the above equation. Step by step, we get the result:

$$\begin{aligned}\bar{z} &= (\bar{a}+b+\bar{c})(a+\bar{b}+\bar{c})(\bar{a}\cdot b+a\cdot\bar{b}+\bar{c}) \\ &= (\bar{a}+b+\bar{c})(a\cdot\bar{b}+\bar{c}) \\ &= \bar{a}\cdot\bar{c}+b\cdot\bar{c}+\bar{c} \\ &= \bar{c}\end{aligned}$$

The performance of the system is dominated by two factors: the sequence of the transistor facts and maxterm decomposition in completeness checking. Other factors have linear execution time. In Prolog, the sequence of facts determines the time needed to retrieve the necessary information. In the worst case, Prolog needs to spend the maximum time to obtain a fact in its database. To reduce the influence of this factor, those transistors' facts which form a gate should be placed as close together as possible. Next, unfortunately, maxterm decomposition has exponential executing time with respect to the number of maxterms in a given equation. However, the performance is still superior than Karnaugh-map method. For example, in Figure 6-8, the equation of output z has 8 max-

terms after applying De Morgan's theorems. But, z has 16 possibilities to reduce it by using the Karnaugh-map method, since z has 4 input variables.

Figure 6-6 is a NOR gate example. Figure 6-7 shows the results from a latch. The latch is made by two NAND gates which was illustrated in Figure 6-5. A more complex example is given in Figure 6-8 which is a 4-bit parity generator.



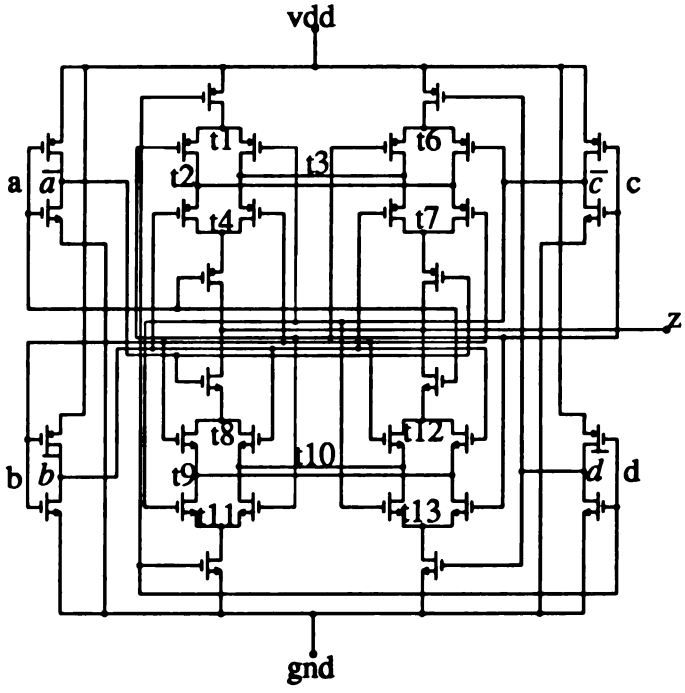
The representation of this latch:

pt(vdd,s,a).
pt(vdd,s,r).
pt(vdd,r,b).
pt(vdd,r,s).
nt(t1,s,a).
nt(gnd,t1,r).
nt(t2,r,b).
nt(gnd,t2,s).

The result:

$$r = \bar{b}+\bar{s},\ \bar{r} = b \cdot s;$$
$$s = \bar{a}+\bar{r},\ \bar{s} = a \cdot r;$$

Figure 6-7 A latch example.



The representation of a 4-bit parity generator:

pt(vdd,t1,d).	pt(t1,t2,c).	pt(t1,t3,$\bar{c}$).
pt(t2,t4,$\bar{b}$).	pt(t3,t4,b).	pt(t4,z,a).
pt(vdd,t6,$\bar{d}$).	pt(t6,t2,$\bar{c}$).	pt(t6,t3,c).
pt(t2,t7,b).	pt(t3,t7,$\bar{b}$).	pt(t7,z,$\bar{a}$).
nt(gnd,t11,d).	nt(t11,t9,$\bar{c}$).	nt(t11,t10,c).
nt(t9,t8,b).	nt(t10,t8,$\bar{b}$).	nt(t8,z,$\bar{a}$).
nt(gnd,t13,$\bar{d}$).	nt(t13,t10,$\bar{c}$).	nt(t13,t9,c).
nt(t9,t12,$\bar{b}$).	nt(t10,t12,b).	nt(t12,z,a).
pt(vdd,$\bar{a}$,a).	pt(vdd,$\bar{b}$,b).	pt(vdd,$\bar{c}$,c).
pt(vdd,$\bar{d}$,d).	nt(gnd,$\bar{a}$,a).	nt(gnd,$\bar{b}$,b).
nt(gnd,$\bar{c}$,c).	nt(gnd,$\bar{d}$,d).	

The results:

$$z = a\cdot b\cdot c\cdot\bar{d}+a\cdot b\cdot d\cdot\bar{c}+a\cdot c\cdot d\cdot\bar{b}+a\cdot\bar{b}\cdot\bar{c}\cdot\bar{d}+b\cdot c\cdot d\cdot\bar{a}+b\cdot\bar{a}\cdot\bar{c}\cdot\bar{d}+c\cdot\bar{a}\cdot\bar{b}\cdot\bar{d}+d\cdot\bar{a}\cdot\bar{b}\cdot\bar{c}$$

$$\bar{z} = a\cdot b\cdot c\cdot d+a\cdot b\cdot\bar{c}\cdot\bar{d}+a\cdot c\cdot\bar{b}\cdot\bar{d}+a\cdot d\cdot\bar{b}\cdot\bar{c}+b\cdot c\cdot\bar{a}\cdot\bar{d}+b\cdot d\cdot\bar{a}\cdot\bar{c}+c\cdot d\cdot\bar{a}\cdot\bar{b}+\bar{a}\cdot\bar{b}\cdot\bar{c}\cdot\bar{d}$$

$$\bar{a} = \bar{a},\ \bar{\bar{a}} = a;$$
$$\bar{b} = \bar{b},\ \bar{\bar{b}} = b;$$
$$\bar{c} = \bar{c},\ \bar{\bar{c}} = c;$$
$$\bar{d} = \bar{d},\ \bar{\bar{d}} = d;$$

Figure 6-8 A 4-bit parity generator.

Chapter 7 Conclusion

The goal of this research is to establish a methodology for building a database-centered CAD system [19] for digital circuits. The component-oriented design database minimizes the data size and complexity by taking the hierarchical nature of digital circuits into account. New tools can be added by performing the natural-join operation in order to put new attributes into the current database.

Three essential tools are designed and implemented, i.e., a schematic editor, a switch-level simulator, and a transistor-to-gate-level verifier. By applying these tools to some key circuits, the performance of this CAD system can be demonstrated.

7.1 Summary

One of the major features of this research is that we adopt a component-oriented instead of tool-oriented architecture to design this CAD system. Hierarchy and connectivity of digital circuits are the most important principles which guide the development of the design methodology. To present the design methodology, we first analyze the characteristics of digital circuit design from the unified point of view. Based on the analysis, a typical design process is divided into two phases, i.e., the logic design phase and implementation phase. During the logic design phase, designers deal with logic (ideal or well-done) components. The effort in this phase focuses on establishing the relationship of components, building the hierarchy of a circuit, and validating the design. After the logic circuit has been constructed, designers turn into the implementation phase to reduce harmful parasitic effects which are introduced by connecting wires and real components. In other words, this approach is intended to free circuit designers to concentrate the creative aspects of design activities and simplifies the effort for a single tool development.

The working environment to complete a design is also investigated. The consistency is the major concern in order to provide an environment with minimum redundant and repetitive tasks. We recognize that the whole design process is composed of many individual tasks from design specification, design capture, design verification, documentation, and implementation to design realization. Some of the tasks may repeat or can be eliminated. A tool-oriented system can not reduce the amount of tasks but a component-oriented system can since the results from different tasks are integrated and well-organized.

To fully support a component-oriented system, the circuit representation method is crucial. It must have the capability to represent any kind of circuits. In other words, it must be broad and still simple enough in order for many different tools. A representation method which has the format used in predicate calculus is adopted. This unified method, called the definitional method, not only takes good care of circuit hierarchy and connectivity but also can well represent bidirectional components such as MOS transistors.

Based on the design methodology, the design database and the essential tools to support our approach are summaried as follows :

The design database of this system is called STOCK. Nowadays, database support for engineering remains a relatively open issue in CAD research because the representing entity is much more complex than in other applications, e.g., business. (Therefore, many CAD systems are tool-oriented.) However, from the hierarchical point of view, any digital circuit, no matter how complex or simple it is, is merely a circuit component at some structural level. STOCK is designed to contain only one type of entities, i.e., circuit components. The complexity of representing a large circuit is minimized by its hierarchy. In STOCK, a component is usually defined by other components except for the primitive components which are defined by themselves. Different aspects of components can be added without modifying old tools or other unrelated components. Hence, new tools can be easily added to the system. STOCK is realized as several file directories. Each direc-

tory is a single component database with a link to other database. Those unidirectional links make STOCK a distributed database which can support horizontal team work. The database management issues are also distributed and localized. This exactly matches the nature of real component stocks.

We assert that the circuit schematic is an essential part of the overall documentation, even for a system which supports some hardware description language. Consequently, a universal schematic editor, called USE, was designed and implemented. This general-purpose schematic editor was developed from the elementary definition of a circuit netlist. One of the major design endeavors was spent on developing efficient and unified data structures for circuit schematics with un-limited drawing size. The other was focused on restricting the type of graphic operations in order to only use the fastest, i.e., memory mapping and line drawing. Hence, the computer response time is very short for each graphic interactive period, such as zoom or pan. USE can be used to construct any mixed-level circuit from the transistor level to a system level. It supports hierarchy for design capture, i.e., a schematic can become a component in STOCK. Therefore, USE allows designers to express a digital circuit at the same level that he/she thinks.

Verification is the major activity during the design process. However, because of the progress in IC technology, digital circuit design at the gate level no longer meets fully the requirements in designing the integrated circuits. It has been shown that integrated circuits can have better performance and use less silicon area if they are designed at the MOS transistor level [35]. A logic-timing switch-level simulator, called SWSIM, was designed and implemented. SWSIM was designed for CMOS digital circuits. Transistors are modeled as both switches and attenuators with an ideal capacitor associated with each gate terminal. As a result, transistors have three states, i.e., ON, 1/2ON, and OFF. And, node voltages are represented by nine logic values. In other words, the MOS-transistor logic models which was shown in Weste and Eshraghians' book [10] are exactly modeled. To solve the bidirectional problem, a method for determinating the signal-flow

directions was developed. The simulation theory for uniform systems was developed as well. The generic model of a uniform system can be used to model almost any kind of digital circuits. However, the input and output nodes of a basic building element must be specified before the model can be applied. The strength-determination algorithm solved this problem. Hence, the simulation theory can be applied to CMOS digital circuits, which have bidirectional components.

SWSIM has linear computational complexity with the speed comparable to the gate-level simulators. A performance analysis is presented and some key examples are given in Chapter 5. The performance analysis shows the speed is less than 25 seconds/clock for a 15,000-transistor circuit on a SUN-3 workstation. Moreover, the law of excluded middle is always obeyed. This prevents the NP-complete problem [26] which has occurred in gate-level simulation. There is also no restriction for input timing and circuit topology. Theoretically, any CMOS logic family can be simulated correctly.

To validate a circuit structure without specifying the inputs can overcome the drawbacks of traditional simulation approach for some cases. A hierarchical verification system from the layout level to system level was proposed, and the fundamental work up to the gate level was implemented. This rule-based approach to verify the transistor structure of a circuit takes advantage of the Prolog language. Through the use of Prolog's internal database, a circuit can be represented as facts and high-level structure can be represented as rules. The reasoning method which represents the circuit knowledge at a giving level validates the circuit structure statically. This method ensures full correctness of the circuit structure since it takes all components into account. In summary, a component-oriented CAD system with three tools was developed and evaluated.

Nowadays, tool-oriented systems need a set of translation programs to be the interfaces among different tools. This *ad hoc* approach increases complexity and may easily introduce inconsistency all over the design process. Some redundant tasks are inevitable in order to present the same information with different formats. However, a component-

oriented system overcomes this drawback. Most tools in such a system read the input from the component database and, then, put the results back into the database. Since different tools consider different aspects of a circuit component, the format of the component database must be very flexible in order to accommodate many tools. Hence, inefficiency may exist through the database operations. This is the bottleneck of a component-oriented system. To prevent this, our representation method takes full advantage of the circuit hierarchy in order to minimize the component size and complexity.

Currently, the whole system is implemented in C and Prolog with about 14,000 statements. By using the schematic editor, any CMOS digital circuit from the transistor level to a system level can be created. The circuit, then, can be expanded into the transistor level and simulated with the switch-level simulator SWSIM. Or, the Boolean functions of the circuit can be generated and verified with the rule-based verification tool.

7.2 Future Research and Development

Based on the current work, future research and development should be directed toward increasing the rate of automation, extending this work into the field of computer-aided-engineering (CAE), and incorporating a hardware description language, e.g., VHDL [15] [16] [17], for circuit algorithmic and behavioral development. In other words, tool integration in order to handle the whole process from design capture to implementation is the major task in the future. Certainly, to refine existing tools and implement new tools are also very important.

To extend the system into the CAE field, we consider three kinds of implementation. There are the integrated circuit (IC) layout, the printed circuit board (PCB) layout, and the whole system integration. Since a chip, which is at the highest level of IC layout, is still a component in the design database, the CAE techniques for PCB layout can be integrated into this system after the necessary IC layout tools are integrated. According to the same reason, several PCBs which form a complete system is also a component in the database. Therefore, tools for integrating a whole system are needed and should be developed using the design methodology described herein.

LIST OF REFERENCES

LIST OF REFERENCES

[1] S. M. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, chapter 1, 1987.

[2] Sun Microsystems, Inc., "Using Nroff and Troff on the SUN Workstation", Feb, 1986.

[3] G. Gabodi, P. Camurati, and P. Prinetto, "Experiences in Prolog-based DFT rule checking", IEEE International Conference on Computer-Aided Design, pp. 909-914, 1986.

[4] W. F. Clcoksin and C. S. Mellish, *Programming in Prolog*, 2nd edition, Springer-Verlag, Berlin, Heidelberg, 1984.

[5] S. M. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, chapter 8, 1987.

[6] B. McCalla, B. Infante, D. Brzezinski, and J. Beyers, "ChipBuster VLSI Design System", IEEE International Conference on Computer-Aided Design, pp. 20-23, 1986.

[7] D. S. Harrison, P. Moore, and R. L. Spickelmier, "Data Management and Graphics Editing in the Berkeley Design Environment", IEEE International Conference on Computer-Aided Design, pp. 24-27, 1986.

[8] L. N. Dunn, "IBM's Engineering Design System Support for VLSI Design and Verification", IEEE Design and Test of Computers, vol. 1, pp. 30-40, Feb. 1984.

[9] W. A. Burling, B. J. Bartels Lax, L. A. O'Neill, and T. P. Pennino, "Production Design and Introduction Support Systems", AT&T Technical Journal, vol. 66, issue 5, pp. 21-38, Sep/Oct. 1987.

[10] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A system Perspective*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[11] M. H. Doshi, R. B. Sullivan, and D. M. Schuler, "THEMIS Logic Simulator - A Mix Mode, Multi-Level, Hierarchical, Interactive Digital Circuit Simulator", 21st Design Automation Conference, pp. 24-31, 1984.

[12] A. E. Ruehli and G. S. Ditlow, "Circuit Analysis, Logic Simulation, and Design Verification for VLSI", IEEE Proceedings, vol. 71, pp. 34-48, Jan. 1983.

[13] R. E. Bryant, "Symbolic Verification of MOS Circuits", Proceedings of Chapel Hill Conference on VLSI, pp. 419-438, 1985.

[14] C. E. Wu, L. M. Ni, and A. S. Wojcik, "Functional Recognition of Static CMOS Circuits", IEEE International Conference on Computer-Aided Design, pp. 306-313, 1987.

[15] G. M. Nurie, "LOGAL+ -- A Hardware Description Language for Hierarchical Design and Multilevel Simulation", IEEE International Conference on Circuits and Computers, pp. 600-603, 1982.

[16] J. D. Nash and L. F. Saunders, "VHDL Critique", IEEE Design & Test on Computers, pp. 54-65, Apr. 1986.

[17] R. Waxman, "Hardware Design Languages for Computer Design and Test", IEEE Computer, pp. 90-97, Apr. 1986.

[18] C. W. Rosenthal and J. M. Dishman, "Computer-Aided Engineering and Design for Interconnection Technology", AT&T Technical Journal, vol. 64, issue 4, pp. 57-69, Jul/Aug. 1987.

[19] S. M. Staley and D. C. Anderson, "Functional Specification for CAD Database", Computer Aided Design, vol. 18, no. 3, pp. 132-138, Apr. 1986.

[20] J. D. Ullman, *Principles of Database Systems*, 2nd edition, Pitman Publishing Limited, London, chapter 1 and 5, 1982.

[21] T. W. Williams, "VLSI Testing", IEEE Computer, pp. 126-136, Oct. 1984.

[22] S. M. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, Appendix B and C, 1987.

[23] W. Andrews, "Silicon Compilers still struggling toward widespread acceptance", Computer Design, pp. 37-43, Feb. 1988.

[24] R. E. Bryant, "A switch-level model and simulator for MOS digital systems", *IEEE Trans. Comput.* , vol. C-33, pp. 160-177, Feb. 1984.

[25] T. Lengauer and S. Näher,"An analysis of ternary simulation as a tool for race detection in digital MOS circuits", *INTEGRATION, the VLSI journal* , vol. 4, pp. 309-330, 1986.

[26] H. P. Chang and J. A. Abraham,"The complexity of accurate logic simulation", *Proceedings of IEEE International Conference on Computer-Aided Design* , pp. 404-407, 1987.

[27] J. K. Ousterhout,"Crystal: A timing analyzer for NMOS VLSI circuits", *Proceedings of the 3rd Caltech VLSI Conference* , pp. 57-69, 1983.

[28] N. P. Jouppi,"TV: An NMOS timing analyzer", *Proceedings of the 3rd Caltech VLSI Conference* , pp. 71-76, 1983.

[29] W. F. Clocksin and M. E. Leeser,"Automatic determination of signal flow through MOS transistor networks", *INTEGRATION, the VLSI journal* , vol. 4, pp.53-63, 1986.

[30] S. Even, *Graph Algorithms* , Computer Science Press, Rockville, Maryland, Chapter 3, 1979.

[31] C.-T. Chen, *Linear System Theory and Design* , CBS College Publishing, New York, Chapter 3, Section 7, 1984.

[32] D. L. Greer, "The quick simulator benchmark", *VLSI systems design* , pp. 40-49, Nov. 1987.

[33] S. M. Trimberger, *An Introduction to CAD for VLSI* , Kluwer Academic Publishers, Norwell, Massachuestts, Chapter 9, 1987.

[34] T. J. Schaefer,"A transistor-level logic-with-timing simulator for MOS circuits", *Proceedings of the 22nd Design Automation Conference* , pp. 762-765, 1985.

[35] T. Hamilton, P. Harrod, and S. Dehgan,"MC68030 process and circuit technology", *Proceedings of IEEE International Conference on Computer Design* , pp. 590-593, 1987.

[36] C. Mead & L. Conway, *Introduction to VLSI System*, Addison-Wesley, Reading, Massachusetts, 1980.

[37] J. K. Ousterhout, et al., "MAGIC: A VLSI Layout System", 21st Design Automation Conference, pp. 152-159, 1984.

[38] A. Colmerauer, "Theoretical Model of Prolog II", *Logic Programming and Its Applications*, chapter 1, Ablex, Norwood, New Jersey, 1986.

[39] R. E. Bryant,"Symbolic Verification of MOS Circuits", Proc. Chapel Hill Conference on VLSI, pp.419-438, 1985.

[40] C. E. Wu, et al.,"A Rule-Based Circuit Representation for Automated CMOS Design and Verification",Proc. 24th Design Automation Conference, pp. 786-792, 1987.

[41] C. Svensson and R. Tjärnström, "Switch-Level Simulation and the Pass Transistor Exor Gate", IEEE Trans. Computer-Aided Design, vol. 7, pp. 994-997, 1988.

[42] J. Conklin, "Hypertext : An Introduction and Survey", IEEE Computer, pp. 17-41, Sep. 1987.

[43] C. Pedron and A. Stauffer, "Analysis and Synthesis of Combinational Pass Transistor Circuits", IEEE Trans. CAD, vol. 7, no. 7, pp. 775-786, July 1988.