CHAPTER 10 DESIGN AUTOMATION AND VERIFICATION

10.0 INTRODUCTION

Design automation and design verification are the keys to effective use of large-scale integrated circuit technology today. When circuits consisted of only a few transistors or gates, layout and checking of circuits by hand were reasonable. As circuit complexity increased to thousands and tens of thousands of transistors, manual tools were no longer sufficient for design, causing computer-based design aids to become prominent. With present integrated circuits containing hundreds of thousands of transistors, heavy dependence on design automation and design verification is necessary to design these circuits.

This chapter describes the nature and use of basic design automation and design verification tools as applied to the design of integrated circuits. *Design automation tools* are defined here as those computer-based tools that assist through automation of procedures that would otherwise be performed manually, if at all. Simulation of proposed design functionality and synthesis of integrated circuit logic and layout are just two examples. *Design verification tools*, on the other hand, are those computer-based tools used to verify that circuit design or layout meets certain prescribed objectives. A geometrical design rule checker for examining layout characteristics is an example, and a logic simulator with a specific set of input vectors and corresponding desired output vectors is another. Note that simulation can be classified in either category according to its purpose. Both design automation tools and design verification tools are included in the more general class known as CAD (computer-aided design) tools.

Both design automation and design verification tools require computer-readable descriptions of the underlying circuit function and structure to operate. These computer-based descriptions vary from simple geometrical specification languages such as CIF[1] (Caltech Intermediate Form) to high-level functional description languages such as VHDL[2] (a hardware design language). Initially the focus of this chapter is on a description of design tools related to or based on geometrical layout. A simplified geometrical specification language will be examined. Required functionality provided by tools that input and display integrated circuit layout will also be described. Then, tools that check layout geometries and extract circuit net list information will be detailed.

Design tools for higher-level design description and verification are described next. Circuit, switch, and logic simulation for digital circuits are introduced and compared. Timing analysis is examined as a way to verify the temporal operation of digital circuits. Hardware design languages such as VHDL and EDIF[3] (Electronic Design Interchange Format) are introduced with simple examples provided to clarify important concepts.

The descriptions of design verification and design automation tools provided here use MOS examples primarily. The concepts are directly applicable to bipolar designs, although some changes in specific tool capability may be required by different technologies. The chapter concludes with an introduction to automated methods of generating layout from high-level descriptions of digital circuits via silicon compilers.

10.1 INTEGRATED CIRCUIT LAYOUT

Historically, integrated circuit design and integrated circuit layout functions were performed by separate groups. The circuit design task resulted in mixed logic and transistor-level circuit diagrams describing the intended circuits. A circuit description like that of Fig. 10.1-1 was given to layout artists, who were experts at converting circuit diagrams to geometrical layouts such as the one shown in Fig. 10.1-2. For early commercial products, the layout drawings were transferred to rubylith masks by hand. Later, layouts were drawn on vellum—a tough, semitransparent drafting material—to withstand the many design modifications

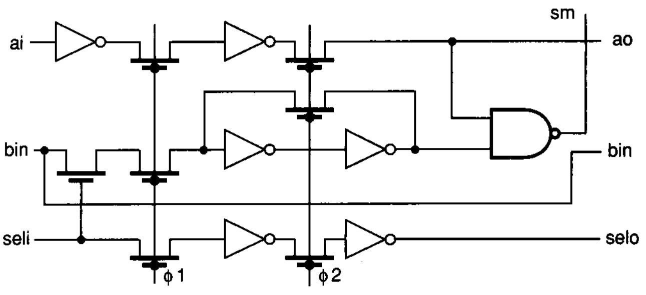


FIGURE 10.1-1
Partial circuit diagram for bit-serial multiplier of Fig. 10.1-2.

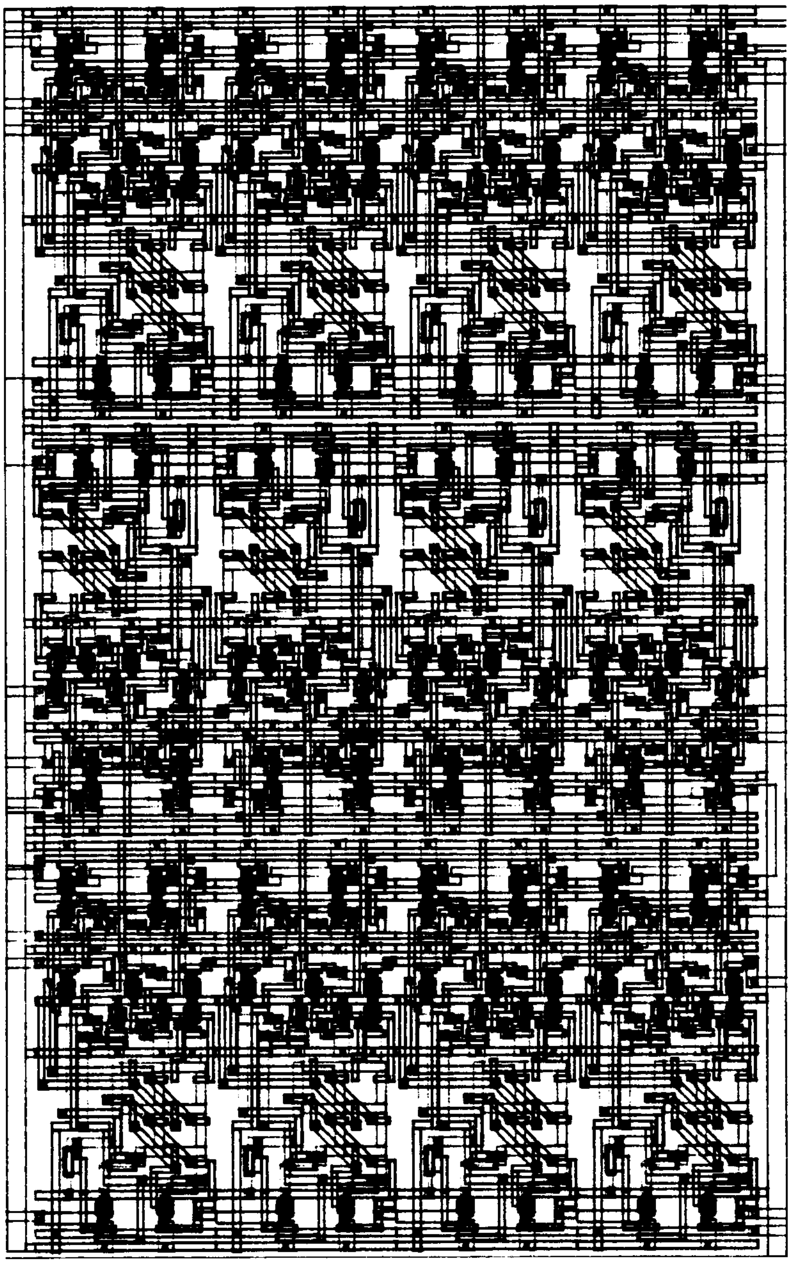


FIGURE 10.1-2
Layout for bit-serial multiplier based on circuit of Fig. 10.1-1.

that are inherent in the normal design process. The layouts from the large vellum plots were digitized to computer-readable form to allow automated checks and to provide input to the mask-making process. Although this method worked for many years, including the early days of microprocessors, the large number of devices required in modern integrated circuits causes fully manual layout to be too time-consuming and prone to error. However, even today, critical sections of the newest microprocessors are still handcrafted to pack the circuit into the smallest possible area.

Many modern methods of integrated circuit layout include both synthesis of control logic and handcrafting of critical building blocks that will be repeated. These layout pieces are entered into a computer at an early stage to allow mechanized help with replicating, checking, and plotting the complete integrated circuit layout. Design layouts may be entered via tools that help convert graphic layout information to computer-readable form. An early tool, shown in Fig. 10.1-3, is called a *digitizer* and was used to enter layout coordinates directly into a computer from a layout plot. Sometimes layout is converted directly to text input in the form of a geometrical specification language. Most often, geometrical layout information is entered through a color graphics workstation to specify the desired integrated circuit layout.

10.1.1 Geometrical Specification Languages

Geometrical specification languages for integrated circuits allow computer-readable definition of the geometries for the mask layers required to fabricate an integrated circuit. These specification languages contain primitive structures such as wires and boxes to specify geometrical shapes and layout levels. Organizational constructs are also provided to allow placement and repetition of the geometrical structures. A geometrical specification language is much like a computer programming language, with the geometrical shape primitives corresponding to instructions and the organizational constructs corresponding to procedures with parameter values.

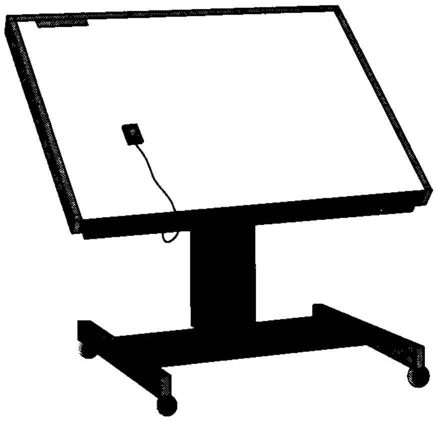


FIGURE 10.1-3
Digitizer board.

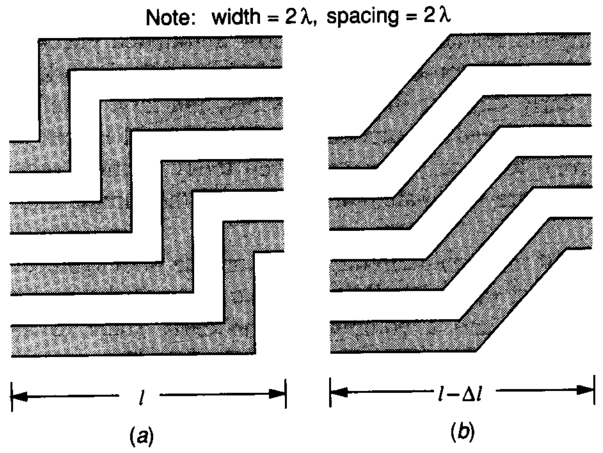


FIGURE 10.1-4
Layout Styles: (*a*) Manhattan, (*b*) Diagonal.

A simplified geometrical specification language for Manhattan style designs for a general MOS process is used here to illustrate relevant concepts. A *Manhattan design style* is one that supports only horizontal and vertical geometries. The name arises because Manhattan style layouts resemble an aerial view of the street layout of New York's Manhattan borough. This style precludes diagonal structures, such as interconnection jogs, that are sometimes used within circuit layouts to minimize area. Figure 10.1-4 shows layout styles with and without diagonal structures. The potential area savings with diagonal structures must be weighed against the increased complexity of programs used to verify the final design. Many commercial integrated circuit manufacturers allow diagonal layout structures but limit these to 45° angles from horizontal and vertical structures.

The simplified geometrical specification language defined here provides only two primitive statements. The two primitives are *boxes* and *levels*, while the organizational constructs include *macros* and *calls*. A macro is like a high-level language (HLL) procedure, and a call is like an HLL procedure call. Table 10.1-1 provides the syntax for these primitives and organizational constructs.

All parameter values are integers. Lengths are in terms of λ, a measure related to the characteristic resolution of the process and the layout design rule set. Macro numbers, layout levels, and orientations are limited to positive integers. A minimum set of layers for a typical NMOS n-well CMOS process is defined in Table 10.1-2. Appendices 2A and 2B define corresponding layers for a double polysilicon NMOS and a p-well CMOS process, respectively.

TABLE 10.1-1
Simplified geometrical specification language

B x y dx dy	Box structure with length dx, width dy, and lower left-hand corner placed at x, y
L n	Layout level for the box definitions that follow
M n	Start of macro number n
E	End of a macro
C n x y m	Call for macro number n with translation x, y and orientation m
Q	End of layout file

TABLE 10.1-2
MOS layer definitions

Layer	CMOS	NMOS
1	n-diffusion	n-diffusion
2	p-diffusion	Ion implant
3	Polysilicon	Polysilicon
4	Metal	Metal
5	Contact	Contact
8	n-well	—
9	Overglass	Overglass

The orientation represents possible rotations of the geometrical figure after translation. The relative order of translation and rotation is important (see Prob. 10.3). Here, rotation is performed first with translation following. The possible orientations are defined in Table 10.1-3 and demonstrated with the block letter P in Fig. 10.1-5.

This simple geometrical specification language will suffice to specify any MOS Manhattan integrated circuit layout if the necessary layout levels are defined. The description is based on alphanumeric characters and is easily displayed, edited, or transferred between computer systems.

TABLE 10.1-3
Rotations of geometries

Orientation	Description
1	No rotation
2	Rotate 90° CCW
3	Rotate 180° CCW
4	Rotate 270° CCW
5	Mirror about y-axis
6	Rotate 90° CCW and mirror about y-axis
7	Rotate 180° CCW and mirror about y-axis
8	Rotate 270° CCW and mirror about y-axis

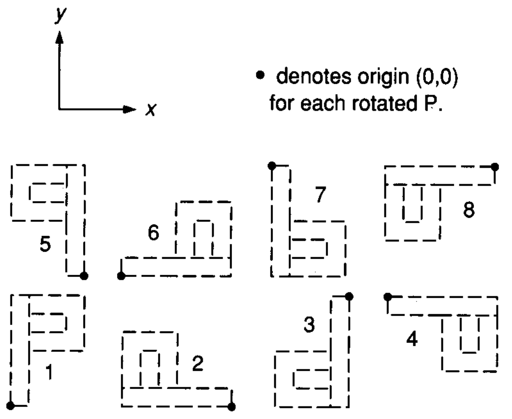
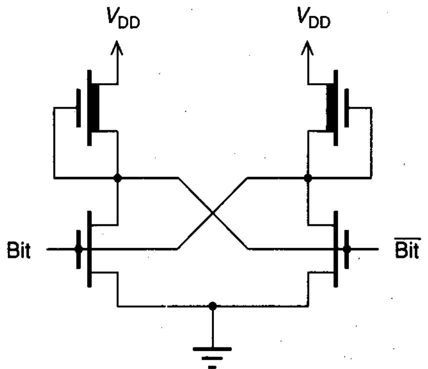


FIGURE 10.1-5
Cell orientations.

```
M 5
L 1
B 11 0 4 5
B 5 3 6 2
B 1 3 4 7
B 1 10 2 3
B 1 13 14 2
L 2
B 4 1 8 6
L 3
B 6 1 4 8
B 8 6 4 6
B 10 12 2 7
L 4
B 0 0 16 2
B 1 6 11 4
B 0 13 16 2
L 5
B 12 0 2 1
B 2 7 2 2
B 9 7 2 2
B 7 14 2 1
E
M 8
C 5 0 0 1
C 5 16 30 3
E
C 8 0 0 1
Q
```

(*a*)



(*b*)

FIGURE 10.1-6
Static memory cell definition: (*a*) Geometrical specification file, (*b*) Circuit diagram.

An example of the geometrical specification file for a static memory cell composed of two inverters tied back to back is shown in Fig. 10.1-6 along with the corresponding circuit diagram. A single inverter consisting of an enhancement pulldown transistor and a depletion pullup transistor is defined by macro 5. This inverter is placed twice, once in a rotated and translated position, to create the static memory cell defined as macro 8. Macro 8 is placed once to create the layout plot shown in Fig. 10.1-7.

10.1.2 Layout Styles

In spite of high labor costs, handcrafted layout is still used within the semiconductor industry because of the necessity to minimize the area required by high-volume integrated circuits. Even automated layout methods such as silicon compilation and standard cell synthesis use handcrafted layout to optimize the primitive cells that are combined through automated techniques. Frequently the basic form for the integrated circuit is sketched and optimized on paper prior to entry into

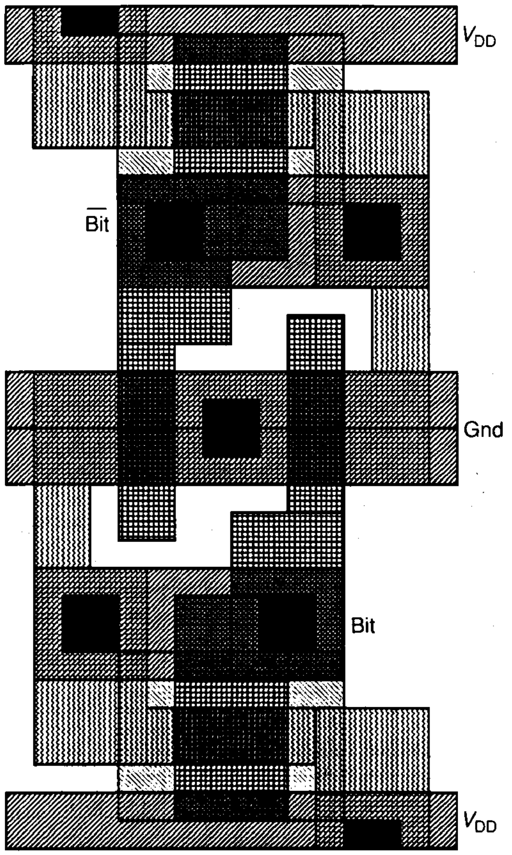


FIGURE 10.1-7
Static memory cell layout.

a computer. The resulting geometrical layouts are digitized, sometimes through use of a symbolic layout language but primarily with the help of an interactive CRT graphics editor.

Handcrafted layouts can be entered directly into a computer in geometrical form through use of an interactive CRT graphics editor. A mouse or joystick is used in conjunction with a cursor to size and position geometrical objects such as boxes on a high-resolution CRT display. A corresponding data file is kept in computer memory to describe the displayed geometries. With an operator's command, this data file may be converted to a geometrical specification language description or can be saved for further use. Several advantages of this graphical editor accrue from bypassing the need to input numerical data to a computer with a text editor or digitizer and from the ease with which geometries can be changed or duplicated.

The graphics editor called Magic,[4] currently popular with universities, uses the painting idiom to create geometrical objects on a color CRT display. The user chooses a color (layout level) from a palette on the screen and paints areas on the screen by specifying two opposite corners of a rectangular field.

The chosen color fills the area. The final result is the same as for other layout methods: a geometrical specification file is created in computer memory, saved, and ultimately transferred to the mask shop.

Graphical editors in both industrial and university environments typically maintain their own unique memory and disk representations of layout geometries. For reasons of efficiency (fast editing response and minimum memory requirements), these representations are often highly optimized binary data structures. In the university environment, the geometrical specification language CIF was defined as a common interchange format among universities and between universities and the MOSIS fabrication service. In industry, EDIF was defined as the interchange format. Most industrial CAD tools provide conversions between their internal format and EDIF. In addition, most industrial CAD tools convert their internal format to a special binary format for submittal to the mask shop. The Berkeley Oct tool set[5] provides conversion from its internal format (Oct) to and from both CIF and EDIF.

In summary, both specification of layout geometries and designer entry of layout geometries are described in this section. Many different geometrical specification languages have been defined and used. A very simple one was defined here for demonstration purposes only. In the university environment, CIF is the predominant interchange format, and EDIF is the interchange standard in industry.

10.2 SYMBOLIC CIRCUIT REPRESENTATION

Descriptions of integrated circuit layouts can take many forms. The geometrical specification language of the previous section provides a primitive textual description of a circuit. Other, more symbolic, forms of representation are often used by designers to specify layouts. A hierarchy of these, including a parameterized layout representation, parameterized module generation, a graphical symbolic representation, and logic equations, is described here.

10.2.1 Parameterized Layout Representation

A symbolic layout language[1] (SLL) allows a textual description of circuit layout in a form that is more easily generated and understood by humans than the geometrical specification language of the previous section. In the past, an SLL was used to represent design layouts that were drawn by hand on graph paper and then digitized. Two main characteristics differentiate an SLL from the geometrical specification language described previously. First, the SLL uses descriptive identifiers for the parameters necessary to specify a geometrical layout. Examples are BOX, POLY, and DX for the geometrical shape, the layer, and the width in the x direction, respectively. This provides a readable description of geometries that specify a layout. Thus, the SLL description is easily entered into a computer using the designer's favorite text editor. Second, symbolic entries are allowed in addition to the numerical data of the geometrical specification language. For example, the x and y position of a geometry might be specified by the variables XPOINT and YPOINT. This allows the final placement of the geometry to depend on the placement of other cells. At some point in the design process, the SLL must be converted to a geometrical specification language for use by other CAD tools and for transmittal to the mask shop. XPOINT and YPOINT must be assigned numerical values to specify the location of the geometry before this conversion takes place.

In addition to the use of symbolic parameters in an SLL, programming constructs such as loops and conditionals can provide additional capability in the specification of a cell's layout. The use of an SLL to describe layout is much like the use of assembly language to describe the machine language (binary) program for a computer. An assembly language program uses mnemonics for the instructions and symbols for variables to simplify and expedite the process of programming a digital computer. Both forms describe the same end object; the binary representation provides the most concise description, while the assembly language is a preferable working medium for programmers.

An SLL description for the layout of the CMOS inverter of Fig. 7.5-5 is given in Fig. 10.2-1. Note the verbose nature of this description compared to the geometrical specification file of Fig. 10.1-6. The description of Fig. 10.2-2 demonstrates the use of variables to allow the inverter cell of Fig. 7.5-5 to be stretched in either the VERT (vertical) or HORZ (horizontal) directions. Also, a REPEAT statement is included to allow the cell to be repeated NR times. RX and RY are the repeat distances along the x and y axes, respectively. If the variables VERT and HORZ are each set to a value of 0 and NR is set to 4, the inverter cascade of Fig. 10.2-3 is produced. The two variables VERT and HORZ can be used to stretch the inverter cell to match the pitch of adjacent cells by

```
CELLNAME CMOSINV;
BOX NDIF X=3 Y=0 DX=4 DY=4;
BOX NDIF X=3 Y=4 DX=2 DY=4;
BOX NDIF X=3 Y=8 DX=4 DY=4;
BOX PDIF X=3 Y=20 DX=4 DY=4;
BOX PDIF X=3 Y=24 DX=5 DY=4;
BOX PDIF X=3 Y=28 DX=4 DY=4;
BOX POLY X=0 Y=5 DX=7 DY=2;
BOX POLY X=0 Y=7 DX=2 DY=18;
BOX POLY X=0 Y=25 DX=10 DY=2;
BOX POLY X=4 Y=14 DX=8 DY=4;
BOX MET1 X=0 Y=0 DX=12 DY=4;
BOX MET1 X=0 Y=28 DX=12 DY=4;
BOX MET1 X=3 Y=8 DX=4 DY=16;
BOX MET1 X=7 Y=14 DX=1 DY=4;
BOX CONT X=4 Y=1 DX=2 DY=2;
BOX CONT X=4 Y=9 DX=2 DY=2;
BOX CONT X=5 Y=15 DX=2 DY=2;
BOX CONT X=4 Y=29 DX=2 DY=2;
BOX CONT X=4 Y=21 DX=2 DY=2;
BOX NWEL X=0 Y=18 DX=12 DY=16;
END CMOSINV;
```

Figure 10.2-1
Symbolic layout language description of CMOS inverter of Fig. 7.5-5

```
CELLNAME CMOSINV;
BOX NDIF X=3 Y=0 DX=4 DY=4;
BOX NDIF X=3 Y=4 DX=2 DY=4;
BOX NDIF X=3 Y=8 DX=4 DY=4;
BOX PDIF X=3 Y=20 DX=4 DY=4;
BOX PDIF X=3 Y=24 DX=5 DY=4;
BOX PDIF X=3 Y=28 DX=4 DY=4+VERT;
BOX POLY X=0 Y=5 DX=7 DY=2;
BOX POLY X=0 Y=7 DX=2 DY=18;
BOX POLY X=0 Y=25 DX=10 DY=2;
BOX POLY X=4 Y=14 DX=8+HORZ DY=4;
BOX MET1 X=0 Y=0 DX=12+HORZ DY=4;
BOX MET1 X=0 Y=28+VERT DX=12+HORZ DY=4;
BOX MET1 X=3 Y=8 DX=4 DY=16;
BOX MET1 X=7 Y=14 DX=1 DY=4;
BOX CONT X=4 Y=1 DX=2 DY=2;
BOX CONT X=4 Y=9 DX=2 DY=2;
BOX CONT X=5 Y=15 DX=2 DY=2;
BOX CONT X=4 Y=29+VERT DX=2 DY=2;
BOX CONT X=4 Y=21 DX=2 DY=2;
BOX NWEL X=0 Y=18 DX=12 DY=16+VERT;
END CMOSINV;
CELLNAME FOURINV;
REPEAT CMOSINV NR=4 RX=12+HORZ RY=0;
END FOURINV;
```

FIGURE 10.2-2
Parameterized symbolic layout language description for inverter cascade of Fig. 10.2-3

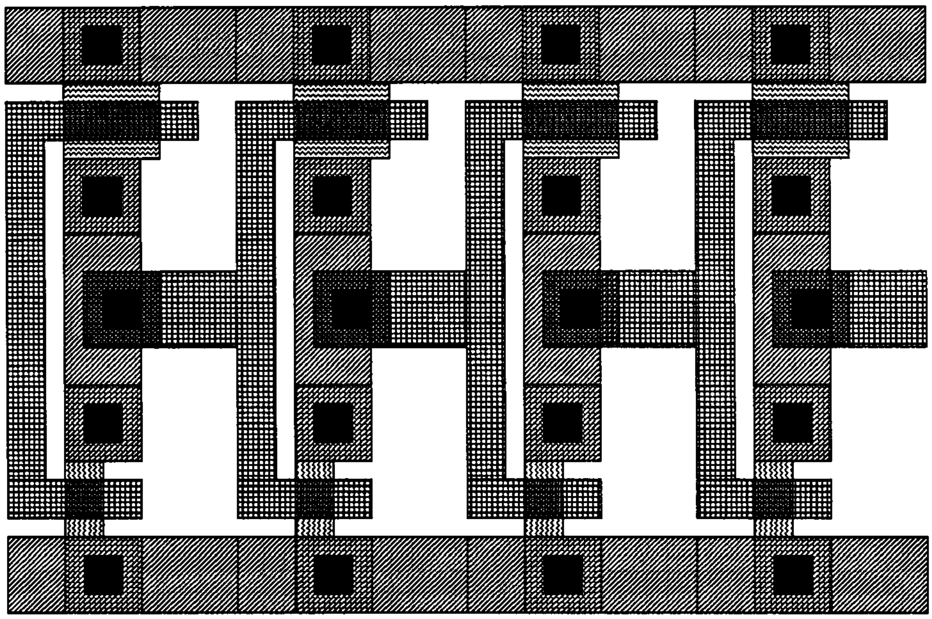


FIGURE 10.2-3
Inverter cascade created from parameterized symbolic layout language description.

specifying positive values for one or both of VERT and HORZ. The use of a programmatic description of layout greatly expands the capabilities of a layout designer in specifying the geometrical structure of a circuit.

10.2.2 Parameterized Module Generation

A recent advance in the area of symbolic layout descriptions is the use of parameterized module generators. A parameterized module generator is a software procedure that can generate many different cell layouts depending on values that are specified when the generator program is executed. Parameterized module generators have been written for RAMs, ROMs, PLAs, multipliers, adders, and data paths, for example. Many of these generators use input parameters to specify the width or number of bits in the generated layout.

As an example, three separate designs might require an 8-bit adder for the first design, a 16-bit adder for the second design, and a 32-bit adder for the third design. Typical design style would use an interactive graphics editor to create each of these adders separately. If a parameterized generator for the adder module could be defined, however, a single module generator could be used to produce an N-bit adder where N is a parameter that can be set to 8, 16, 32, or some other integer value. Then each of the three adders could be created from the same parameterized description. A parameterized module generator is particularly well suited to modern integrated circuit design styles, which commonly utilize regular structures such as rows of cells and arrays of cells.

Parameterized module generators use many of the powerful constructs of high-level programming languages to describe layout structure, position subcells, and fit the overall layout of a larger cell together. Parameterized variables are used with their values bound to a specific value when a module is generated. Conditional statements allow creation of specialized edge cells and programming of memory and PLA contents. For example, a parameterized module generator for an array of cells might include conditional statements such that if both the x and y indices were equal to 0, then an upper-left corner cell would be generated. If the x and y indices were each between the smallest and largest values, a center cell would be generated, and so forth.

The use of high-level programming language techniques also provides a disadvantage for many parameterized module generators. That is, the layout cannot be visualized until the generation program has been compiled and linked to instantiate the layout for a module. These potentially time-consuming steps may hinder the use of interactive layout in designing a module generator for a new cell. To circumvent this problem, there is ongoing research on ways to provide interactive graphical feedback as the geometrical structure of a cell is defined.[6]

With such a tool, a silicon layout specialist can create the parameterized modules that are required in a design. Then a circuit or logic designer can use these blocks by specifying parameters appropriate to the design task. Recently, parameterized module generators were used to specify the layout of a commercial RISC processor (see Sec. 10.11). An interesting, but unsolved problem, is to prove that the output of a parameterized module generator is correct over the valid range of parameters for the module generator.

10.2.3 Graphical Symbolic Layout

The parameterized layout representation described previously provides little insight into the geometrical relationships between circuit elements. This important insight can be provided by another symbolic form for integrated circuit description, called graphical symbolic layout. An early form of graphical symbolic layout is called Sticks.[7] Sticks and related symbolic methods provide an abbreviated, graphical description that combines circuit connectivity with layout topology information. In the Sticks symbology, circuit connections are shown with colored (or weighted) lines representing layout levels, while transistors are formed by the intersection of the lines representing polysilicon and diffusion. The entire layout diagram is composed of simple line symbols that show both connectivity and topology but not actual or relative size for geometrical constructions.

The combination of connectivity and topological information is important in the generation of integrated circuit layouts, as is shown with the aid of the circuit diagram for the quasi-static memory cell of Fig. 10.2-4*a*. This circuit diagram shows a forward path from the first inverter to the second inverter and a clocked feedback path from the second inverter to the input of the first inverter. The circuit diagram does not indicate topological requirements to realize this path.

The geometrical layout of the memory cell of Fig. 10.2-4*a* requires decisions on changes of layout levels to prevent unwanted transistors and connections. The Sticks diagram of Fig. 10.2-4*b* retains all the circuit connectivity information

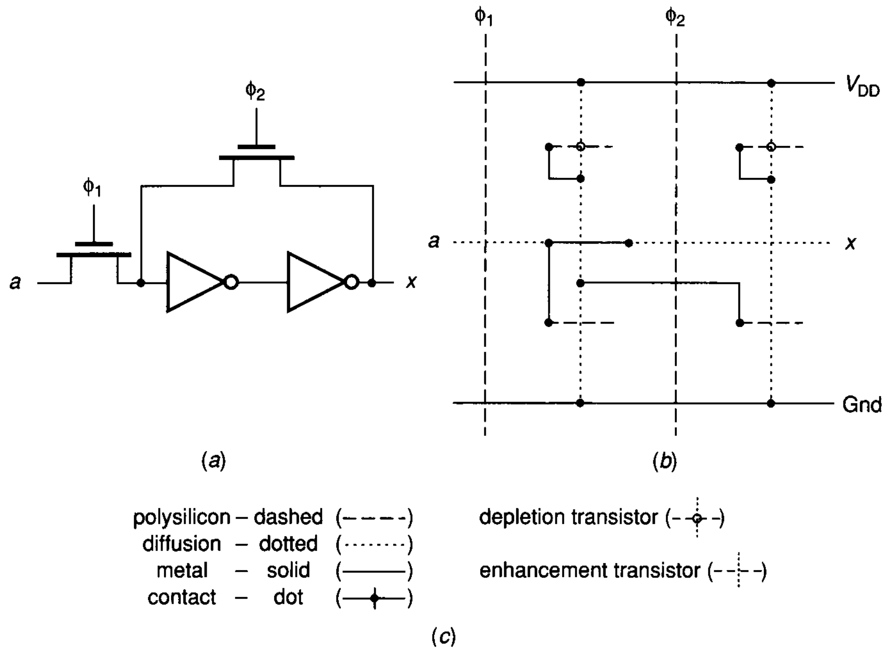


FIGURE 10.2-4
Quasi-static memory cell: (*a*) Logic diagram, (*b*) Symbolic diagram for NMOS layout, (*c*) Layer legend.

for the memory cell and also symbolically specifies the topology of the final integrated circuit layout. In particular, it shows that the feedforward path must be changed from the diffusion layer to the metal layer to cross the polysilicon ϕ_2 clock line without creating an unwanted transistor. Also, the Sticks diagram shows that the feedback transistor is conveniently formed by allowing the polysilicon ϕ_2 line to cross the diffusion feedback path. The diagram shows that power and ground are provided in metal and that the input and output signals are both in the diffusion level. The utility of Sticks and other graphical symbolic layout methods is derived from the simple abstracted notation for layout topology and circuit description.

Once a graphical symbolic layout for a circuit is generated, it is often simple for a designer to convert to a full layout form. The layout task has been simplified to the process of fattening connection lines and compacting the layout, especially if required transistor length-to-width ratios have been noted on the graphical symbolic layout. In fact, this process is simple enough to be automated.[8] If the graphical symbolic layout description has been entered into a computer, perhaps through an interactive graphics terminal, a symbolic compiler program can convert the symbolic layout to a full layout by expanding the line symbols according to a technology specification and then compacting the resulting layout.

As with most automated layout aids, a symbolic compiler usually trades reduced designer efforts for increased silicon area. An increase in the area for a layout generated with a program is not uncommon when compared to a handcrafted layout. As a result, high-volume integrated circuits such as microprocessors and memory continue to utilize handcrafted layout of replicated cells as a major design component. This does not, however, minimize the value of the symbolic representation to the designer. Capturing layout topology in symbolic form early in the layout design prevents later problems such as isolation of a circuit from direct metal connection to power buses.

10.2.4 Logic Equation Symbology

If the function of a digital integrated circuit can be captured by a set of Boolean logic equations, these equations suffice to generate an integrated circuit layout. Thus, logic equations represent a fourth symbolic means to describe a combinational logic circuit. One frequently used means to convert logic equations into layout topology is with a PLA generator, as described in Chapter 9. Two other methods for generating geometrical layouts from logic equations are discussed next: the Weinberger array[9] and SLAP[10] (a methodology for silicon layout).

A Weinberger array uses a regular structure of NOR gates to implement combinational logic in an integrated circuit form. This array structure was introduced in Chapter 9. Figure 10.2-5 shows a Weinberger array used to implement the full adder carry function described by

$$K = AB + AC + BC \tag{10.2-1}$$

Since the final structure is regular, it is not difficult to construct a computer program to generate the array layout using logic equations as program input. By

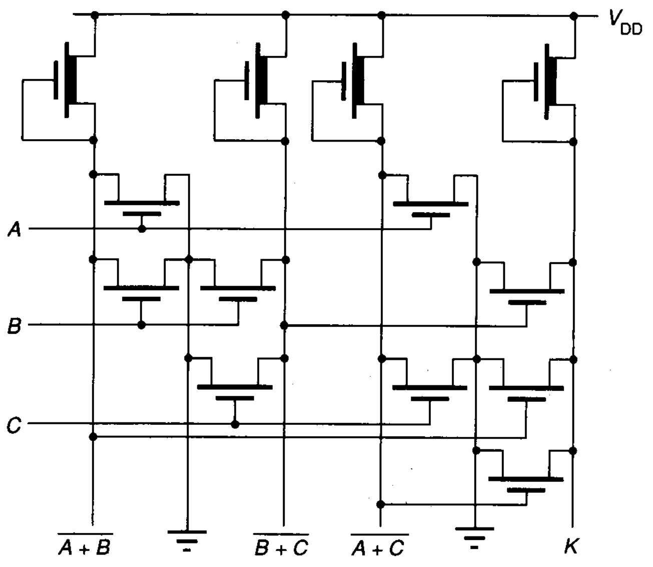


FIGURE 10.2-5
Weinberger array for full-adder carry.

use of DeMorgan's theorem, any combinational logic function can be realized using only NOR gates. In fact, the Weinberger array requires at most a series path of three NOR gates between an input and an output to realize a combinational logic function. Remember that a single-input NOR gate is an inverter. Thus, a first NOR gate may be required to provide the complement of an input while the final two levels use the NOR-NOR logic form to realize the logic function in product-of-sums form.

The use of NOR gates for a Weinberger array allows a constant size for the pullup devices even though the number of inputs and their corresponding pulldown devices may differ for each gate. Careful design allows adjacent gates to share a single ground path, as shown in the layout of Fig. 10.2-6. This array structure can be easily expanded by adding input variables at the bottom and NOR gates to the right without changing the existing structure.

A comparison of the Weinberger array with the PLA yields an interesting result. Even though the logic of a PLA is realized entirely with NOR gates, the AND-OR logic form corresponding to a sum-of-products description is normally used. The AND-OR logic form can be realized with NOR gates only by inverting both the inputs and outputs. This requires a series string of four or five NOR gates between the PLA inputs and outputs, thus causing more delay for a PLA implementation of logic than for a Weinberger array implementation which requires only three levels of logic.

In contrast to the PLA and the Weinberger array, both with predefined array structures, a third method called SLAP has been proposed to compile logic equations into layout form. SLAP first converts logic equations into a directed graph with a graph level for each level of the logic equations. If double-rail inputs

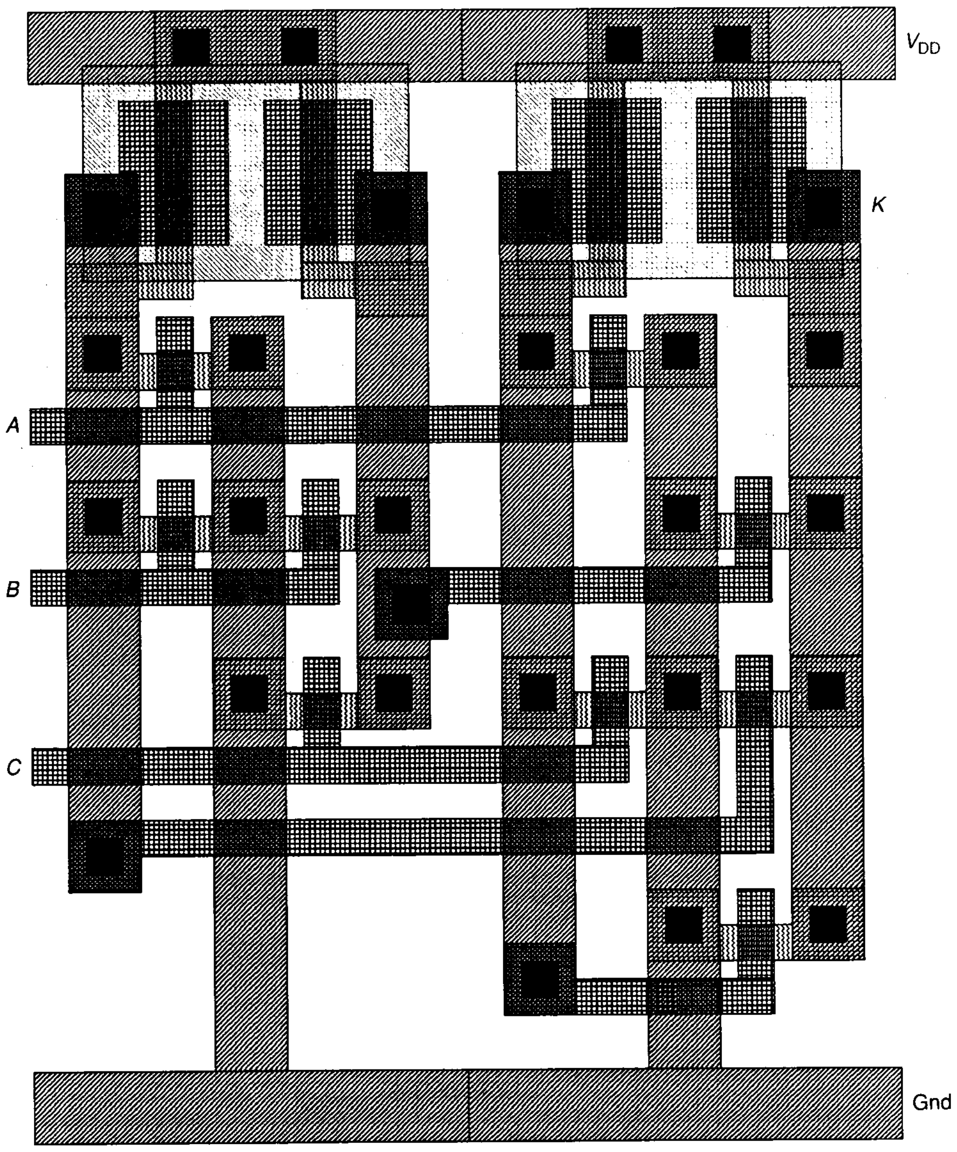


FIGURE 10.2-6
Layout for Weinberger array.

are available, at least two levels of gates are required to implement a general logic function. The SLAP methodology, however, allows realization of intermediate outputs that may then be used as inputs for other logic functions. A graph with an arbitrary number of levels may be required, depending on the particular representation for the logic. Figure 10.2-7 shows the directed graph for the logic functions of the following equations.

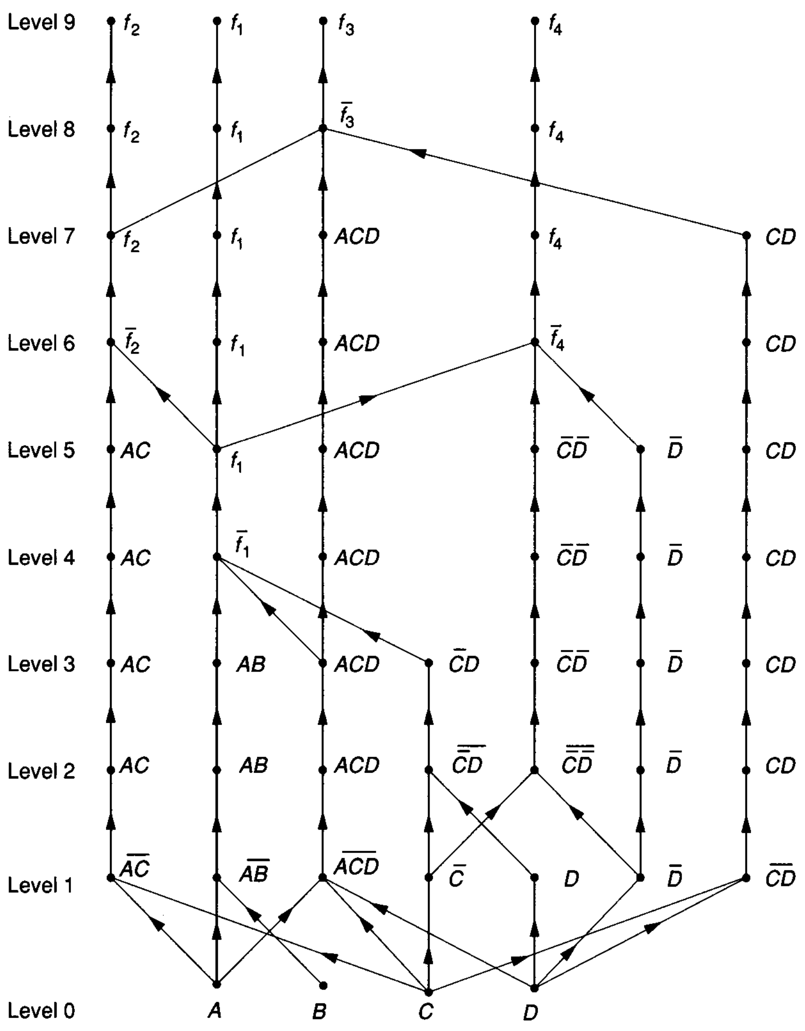


FIGURE 10.2-7
Directed graph for Eqs. 10.2-2 through 10.2-5.

$$f_1 = AB + \bar{C}D + ACD \qquad (10.2\text{-}2)$$

$$f_2 = AC + f1 \qquad (10.2\text{-}3)$$

$$f_3 = ACD + CD + f_2 \qquad (10.2\text{-}4)$$

$$f_4 = \bar{C}\bar{D} + \bar{D} + f_1 \qquad (10.2\text{-}5)$$

This directed graph is formed by placing logic gates with external inputs at the first level, secondary logic gates at the next level, and so on. Heuristics are then used to improve the organization by reducing the number of required levels, if possible, and to reduce the resulting layout area required. The layout density achieved with this method is about the same as that accomplished with gate array structures. An important characteristic of the SLAP methodology is that general logic structures can be compiled directly into a geometrical layout, whereas the PLA format forces a two-level logic realization.

In this section, four methods of generating layout from symbolic representations were introduced. Of the first two, parameterized layout representation and parameterized module generation, the second is growing in popularity for layout of today's designs. Graphical symbolic layout also enjoys success as a technique for layout of random logic. Synthesis of layout directly from the fourth symbolic form, logic equations, is fast becoming a widely used technique for generating integrated circuit layout.

10.3 COMPUTER CHECK PLOTS

Generation of a layout plot from a geometrical specification file for an integrated circuit is often desirable. In the past, large-scale plots, some almost big enough to cover one end of a basketball court, were generated so that visual checking of circuit layout could be performed. Most of these visual checks can now be performed directly from a computer-based geometrical specification file without manual intervention. A computer program can verify fixed rules for the millions of geometrical figures used to describe VLSI circuits without tiring and without error—a task that is essentially impossible for humans. However, human capability to critique overall structure or to detect inconsistencies in an otherwise regular design is difficult to duplicate with computer-based checks. As a result, hardcopy plots of integrated circuit designs are still used for finding errors, for promotional literature, and for many other purposes. Such plots are called *computer check plots*.

Computer check plots for integrated circuit designs are created in both soft- and hardcopy form on CRTs, printers, and plotters using color or black on white representations for the layout artifacts. Check plot devices range from monochrome CRTs, with only 24×80 character resolution for the entire display, to laser printers with 300 dots per inch or higher resolution. To compare the maximum usable display capability over this range of resolution, an example using a static memory cell is examined next.

The static memory cell of Fig. 10.1-7 has dimensions of $16\ \lambda \times 30\ \lambda$ for an area of $480\ \lambda^2$. A monochrome alphanumeric CRT using character graphics with 24 lines by 80 columns can display an area of $1920\ \lambda^2$, although the effective area is somewhat less because of the 1:3 aspect ratio of the CRT display resolution. All details of the static memory cell are visible in the CRT display, as shown in the hardcopy plot of Fig. 10.3-1, but the cell's relation to other cells is lost. As a second example, a dot matrix drawing normally requires a resolution of at least 5 dots per λ to define the smallest details of a circuit. For a printer with a resolution of 100 dots per inch, the static memory cell requires a plot that is about 0.8×1.5 in. to show the details of the circuit. Figure 10.3-2 provides a plot of this size for the memory cell of Fig. 10.1-7. If the memory cell were part of a 1K-bit memory (32 cells $\times$ 32 cells), a high-resolution plot of the entire memory array would require 25.6×48 in. Of course, the general form of the memory area could be discerned with a much smaller plot. Figure 10.3-3 shows a plot at one-tenth this scale (2.56×4.8 in.) for the 32-by-32 cell array.

```
................................................................................
................................................................................
................................................................................
................................................................................
..........##...........####...........##........................................
..........##.|||####|||####........|||##........................................
..........##.|||#55#|||####........|||#5..........              ................
..........##.|||#55#...####........|||#5..........   Legend     ................
..........##.|||####.==####===####.|||##..........              ................
..........##.||.####.==####===#55#.||.##..........  # metal     ................
..........##====####...####.==#55#====##..........              ................
..........##====####...####.==####====##..........  = polysilicon ................
..........##====####==.####...####====##..........              ................
..........##====#55#==.####...####====##..........  | diffusion ................
..........##.||.#55#===####==.####.||.##..........              ................
..........##|||.####===####==.####|||.##..........  5 contact   ................
..........5#|||........####...#55#|||.##..........              ................
..........5#|||........####|||#55#|||.##..........  . blank     ................
..........##|||........####|||####|||.##........................................
..........##...........####...........##........................................
................................................................................
................................................................................
................................................................................
```

FIGURE 10.3-1
Hardcopy plot of SRAM cell as displayed on 24 line by 80 character CRT (λ = one character width).

A typical graphics CRT display with a 19 in. diagonal screen (15 in. horizontal × 11 in. vertical) might have a resolution of 760 by 480 dots. This is roughly 50 dots per display inch. Based on the analysis above, the details of a 152 λ by 96 λ circuit could be displayed in its entirety on the screen. This would correspond to about a five-by-six array of the memory cells described above. Figure 10.3-4 shows a hardcopy plot of the memory cells that could be seen on the CRT display. Of course, an entire chip can be displayed if the layout is scaled so that the finer details of the chip are lost. Figure 10.3-5 shows the entire layout for a 220 × 230 mil image-processing chip composed of sixteen, 12-bit serial multipliers with associated circuitry and input/output pads.

Color displays and plots are always a higher-cost feature than black and white; where color is available, each integrated circuit layer is represented using

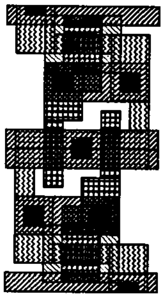


FIGURE 10.3-2
Minimum size plot for Fig. 10.1-7 with 100 dots/inch resolution.

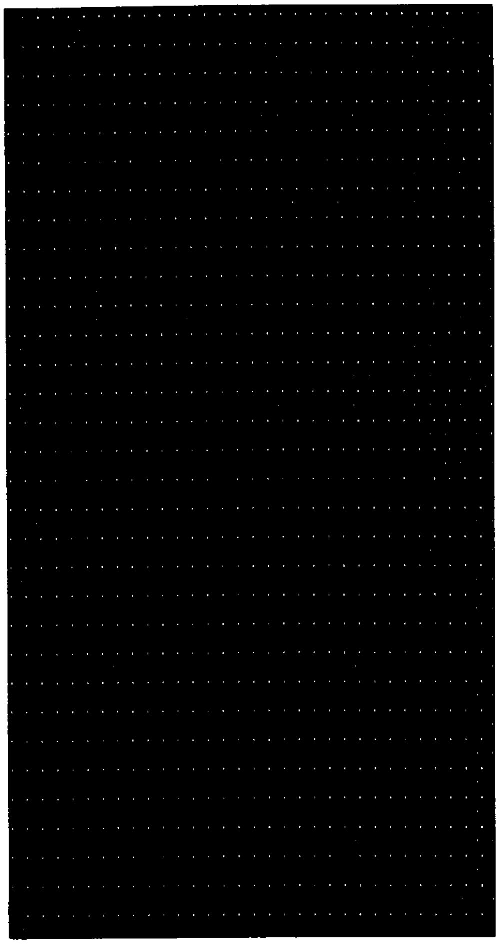


FIGURE 10.3-3
Plot of 32 × 32 memory cell array.

a different color. Aside from their aesthetic appeal, color renditions of circuits show higher information content per unit area, allowing display of larger circuits in a given area. For a color display, only 2 to 3 dots per λ of resolution are necessary to delineate circuit details. Additionally, individual color levels can be used to show labels, flag geometrical design rule violations, or highlight specific features of a circuit. Most modern graphics workstations provide color displays.

When black on white plots are generated, two primary methods are used to distinguish individual layers. Line drawings, with each layer represented by a different style of line (solid, dotted, dashed, dot-dash, etc.) are producible on almost any printer with dot graphics capability (see Fig. 10.3-6). Filled drawings with different layers shown by characteristic area fill pattern (fine dots, heavy dots, diagonal lines, vertical lines, etc.) are popular, even at the expense of increased computer time to generate the plots, greater wear on the printer mechanism, and longer time to print the plots. Laser printers provide good resolution

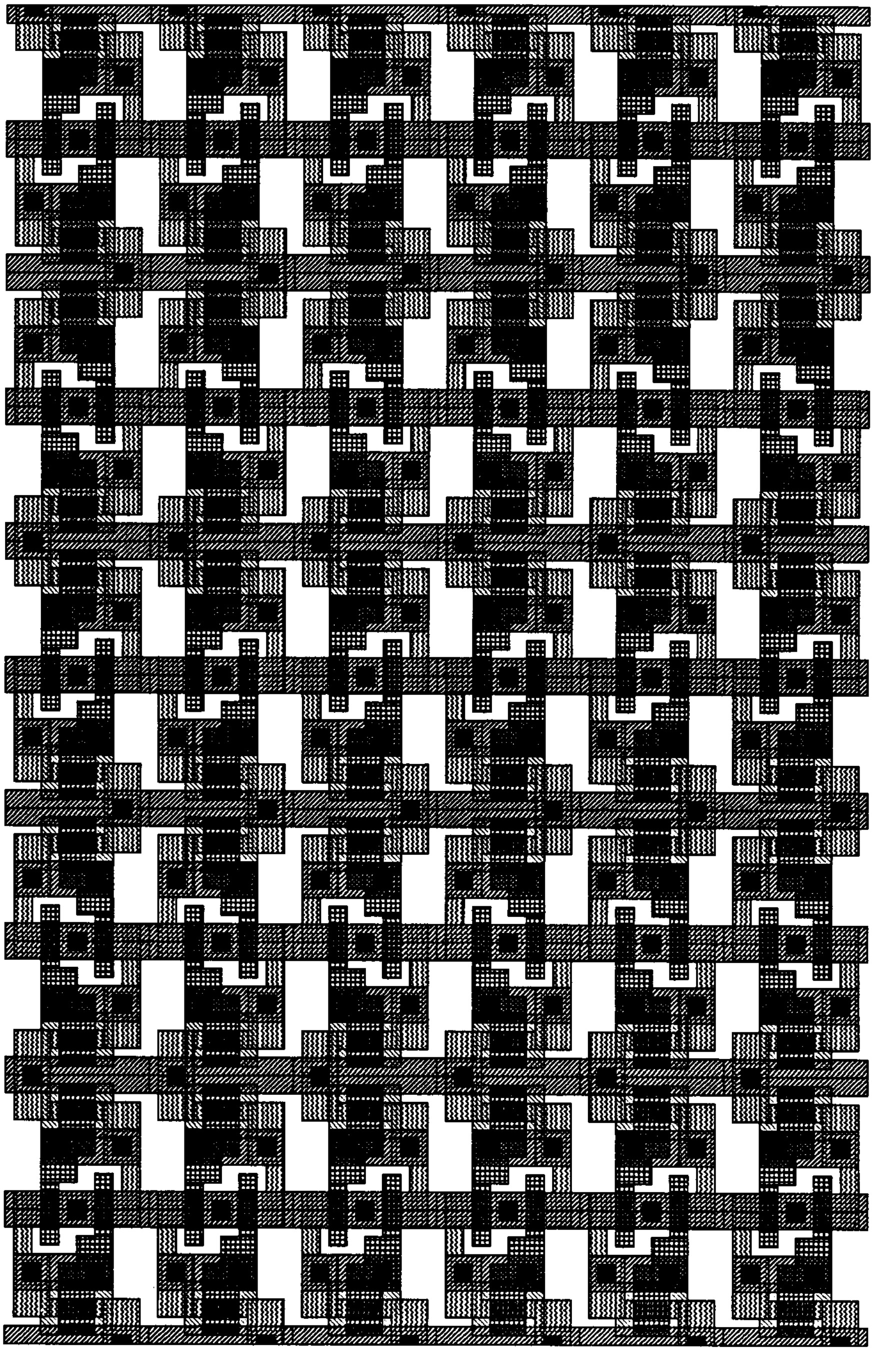


FIGURE 10.3-4
Hardcopy plot of memory cells visible on typical graphics CRT display.

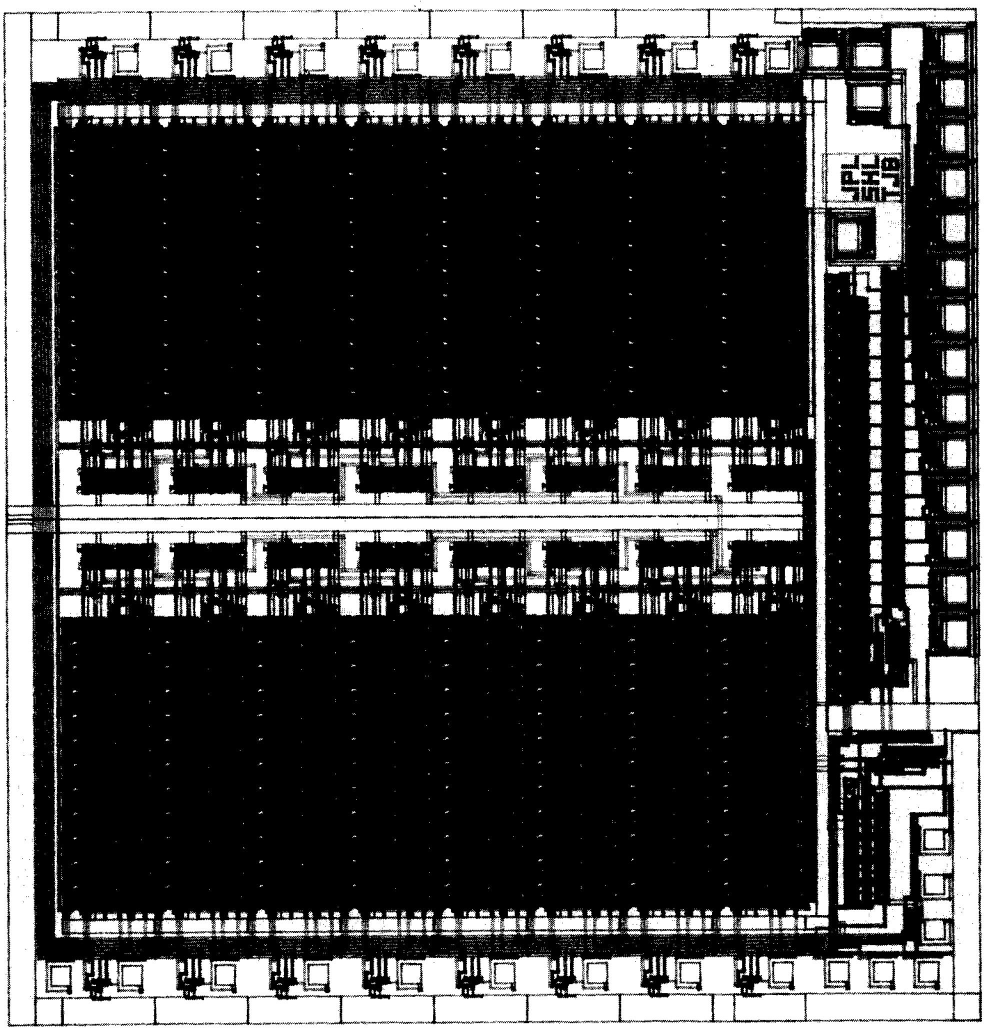


FIGURE 10.3-5
Image-processing chip (220 mil × 230 mil).

(300 dots per inch) and are frequently used for area fill check plots. A primary advantage of the filled drawing of Fig. 10.1-7, compared with the line drawing of Fig. 10.3-6, is that the concept of area for integrated circuit layers is quickly conveyed to the viewer by the filled drawing. This concept is important to the designer since the fabrication process operates on contiguous areas rather than the individual boxes used to describe them.

In this section, a short summary of integrated circuit display media and their corresponding resolution requirements was presented. It is important to have high-resolution display and hardcopy capability for integrated circuit layout design.

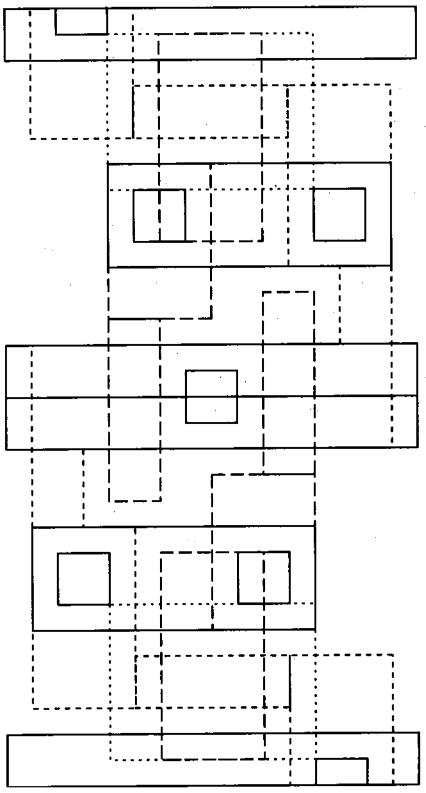


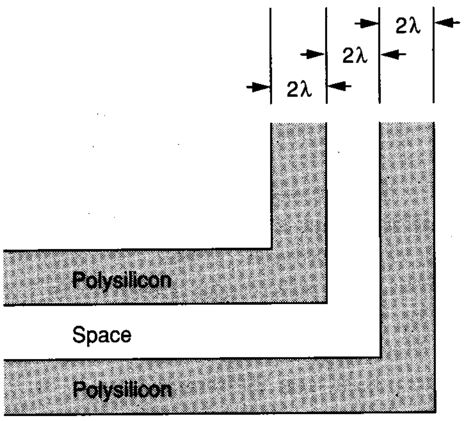
FIGURE 10.3-6
Line check plot of layout of Fig. 10.1-7.

10.4 DESIGN RULE CHECKS

Integrated circuits are created from several layers whose geometrical structures are defined by photolithographic masks. At any given time, a minimum resolution exists for the structures that can be fabricated on silicon because of lithographic and processing constraints. Any attempt to define structures that require higher resolution or accidental specification of a higher resolution through carelessness may lead to nonfunctional circuits. Also, violation of certain geometrical relationships among layers may cause failures because of processing constraints. For each process, a set of guidelines called *design rules* is specified to encapsulate geometrical fabrication constraints. The design rules for the CMOS process described in Table 2B of Appendix 2 are used as the basis for the following discussion. However, most of the rules are determined by general lithographic and processing constraints so that similar rules apply to other processes as well.

10.4.1 Geometrical Design Rules

A conceptual explanation of geometrical design rules is provided in this section. Design rules were introduced in Sec. 2.3 of this text. Geometrical design rules for a single integrated circuit layer are simple; they involve only spacings and widths. Figure 10.4-1 demonstrates a 2 λ spacing between polysilicon conductors

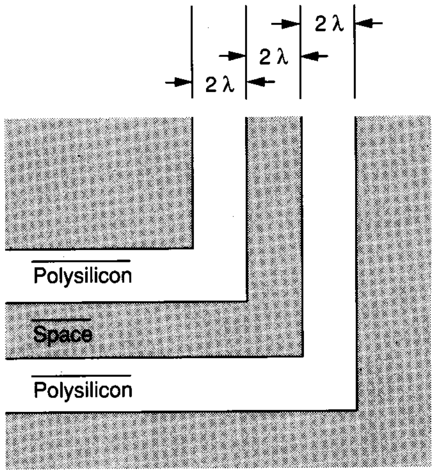


177155

FIGURE 10.4-1
Minimum width polysilicon conductors.

that are each 2λ wide. It is worth noting that if a mask layer is complemented, all widths become spacings. This is shown in Fig. 10.4-2, where the complemented polysilicon conductor widths from Fig. 10.4-1 appear as spacings. Therefore, if width is considered in terms of the complement of the layer definition, all single-layer rules can be treated as simple spacing rules. This means that the same computer algorithm can be used to check for both width and spacing errors.

An interesting conceptual understanding of design rules was provided by Lyon.[11] His explanation is based on the scalable parameter λ, which is said to describe the minimum resolution of the fabrication process. In practice, fabrication processes are usually characterized by their minimum transistor length. The parameter λ is normally specified as half the minimum transistor length. Thus, a 2 μ process has a minimum gate length (and width) of 2 μ, and λ would be set to 1 μ. Thus, λ is not directly a measure of process resolution, but rather is proportional to the minimum device length. With this in mind, the following two meta rules (a meta rule is a rule about rules) were proposed by Lyon to generalize geometrical design rules in terms of λ.



165177

FIGURE 10.4-2
Complemented layout, where spacings become widths.

1. A 1 λ error should not be fatal, although the intended performance of the integrated circuit may be degraded.
2. A 2 λ error may be fatal and almost certainly will degrade the performance of the integrated circuit.

Consider the minimum width of 2 λ for the polysilicon conductor shown in Fig. 10.4-3*a*. If the width of the actual polysilicon that is fabricated on the chip is 1 λ less than this minimum, as in Fig. 10.4-3*b*, the polysilicon will still conduct, although its resistance will double. If the fabricated polysilicon conductor is 1 λ wider than the minimum, as in Fig. 10.4-3*c*, the resistance is lowered, but the polysilicon still functions as a conductor. Thus, a change in width of 1 λ does not cause an obviously fatal problem for the polysilicon interconnection.

Now consider a 2 λ deviation from the design width of 2 λ. If a minimum width polysilicon conductor is narrowed by 2 λ, as in Fig. 10.4-4*a*, there is no conductor left—certainly a fatal error unless the connection was redundant. If the width is increased by 2 λ as in Fig. 10.4-4*b* and the minimum polysilicon spacing is 2 λ, there is a chance that the polysilicon conductor will contact an adjacent polysilicon conductor, causing a short circuit—also a fatal error.

Other design rules involve more than one level and are harder to remember and to verify. As an example of a two-level rule, consider that a transistor is created by the area common to polysilicon and diffusion. This transistor area must satisfy the 2 λ minimum length rule, so the smallest transistor size is 2 λ by 2 λ. The diffusion areas for the source and drain of a transistor also must satisfy a 2 λ minimum length. This rule is sometimes confusing from a layout viewpoint since the source, the drain, and the transistor gate area appear as one contiguous diffusion area. Thus, a source area 1 λ long combined with a transistor area 2 λ long and a drain area 2 λ long, shown in Fig. 10.4-5*a*, appears as a diffusion area 5 λ long and does not seem to violate the 2 λ diffusion length rule. However, Fig. 10.4-5*b* shows that a source only 1 λ long could disappear as a result of a 1 λ alignment error between polysilicon and diffusion—thus a 2 λ rule must be specified for the transistor source/diffusion dimensions. Typical design rule sets for several processes, including NMOS and CMOS, are provided in Appendix 2.

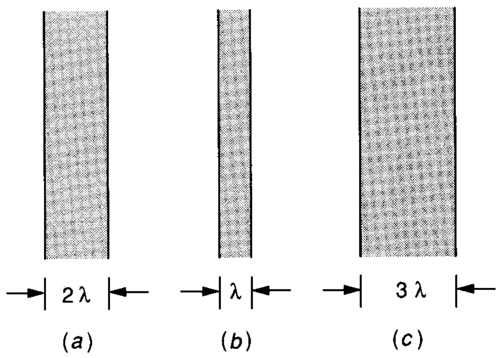


FIGURE 10.4-3
DRC degradation meta rule.

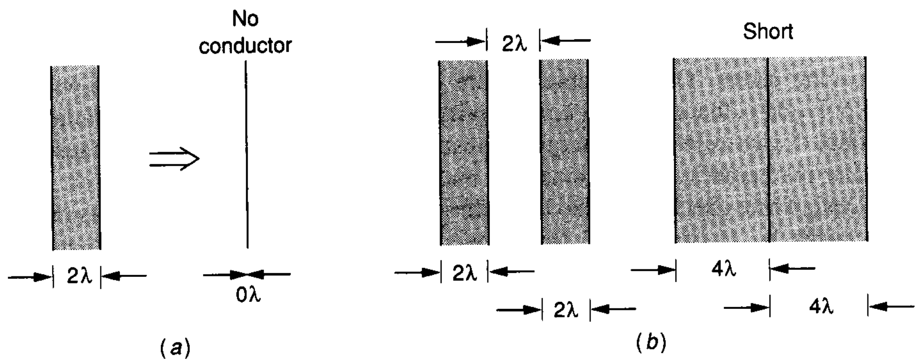


FIGURE 10.4-4
DRC fatality meta rule.

10.4.2 Computer Design Rule Checks

If a designer creates or changes a geometrical specification file manually, a *design rule check* (DRC) is required. Because of the large number of geometries and the wide variation in number and style of geometrical design rules in today's circuits, computer-based DRCs are necessary. Two different styles of DRC programs are in wide use. These can be categorized as polygonal checks and raster scan checks. Both styles will be described briefly.

Polygonal design rule checks are widely used within the semiconductor industry. The geometrical specification file is expanded to produce polygons defining all connected areas for the layer(s) of interest. Note that the layer of interest may be a composite area such as active transistor area or perhaps depletion transistor area. Or it may be a difference area such as the ion implantation overhang created by subtracting the depletion transistor area from the ion implan-

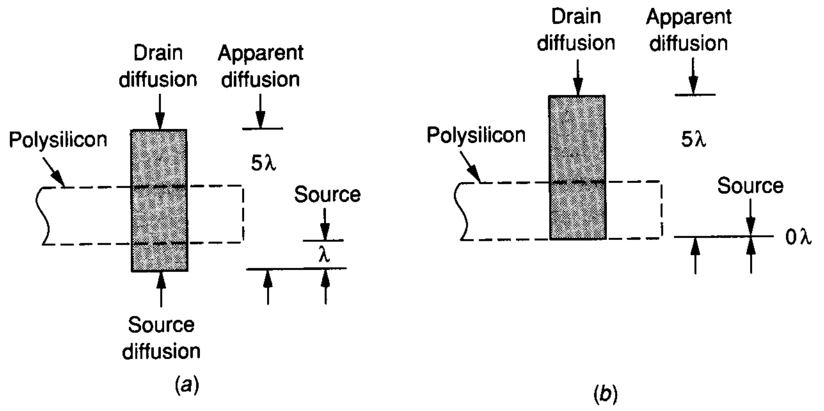


FIGURE 10.4-5
Transistor source width.

tation area. These special areas can be defined by logical operations on primitive layers. Once the polygonal definitions are formed, they can be analyzed for width and spacing errors. One valuable feature of encircling a connected area with a single polygon is that electrical connectivity information is immediately available. Polygonal design rule checks require substantial computing resources because of the many mathematical operations that must be performed during the check.

Design rule checks can also be performed in a relatively simple way as raster scan checks by passing small filters over a rasterized image of the integrated circuit. To allow this, an entire geometrical specification file is instantiated (expanded into the geometries and layers that represent the layout) within a two-dimensional array where the dimensions represent the x and y coordinates of a point and the contents are binary variables to indicate the presence or absence of each layout level. The resolution of the x and y coordinates limits the precision of the design rule checks. Filters such as a 4×4 array,[12] a "plus" symbol, or a circled "plus" symbol[13] have been used to scan the instantiated layout to check for design rule violations. These methods are conceptually simple and computationally clean, but lack the accuracy and connectivity information of the polygonal methods.

10.4.3 Design Rule Checker Output

To demonstrate the results from a raster scan DRC program, several errors were placed in a geometrical specification file. The layout for this file is shown in Fig. 10.4-6. The resulting output from the DRC program is shown in Fig. 10.4-7. The DRC program outputs a heading that gives the name of the file, the date and time, the bounding box coordinates for the checked area, and the macro number. Below the heading, a list of all vertical and horizontal errors is provided. This particular sample contains three vertical and four horizontal errors. Each violation is shown by a one-line entry containing the identification of the violated rule, the x and y coordinates of the violation, the violation or error distance, and the length over which the violation occurred. The resolution of the layout of Fig. 10.4-6 and the DRC results of Fig. 10.4-7 is 0.5 λ.

Definitions of the seven rule violations from Fig. 10.4-7 are given in Table 10.4-1. In each case these errors involve a spacing violation. For example, Rule 6.2 is a metal spacing error. A glance at the upper left corner of Fig. 10.4-6 shows a T formed by a long horizontal metal section and a short vertical metal section separated from the horizontal metal (top of the T) by about 1 λ. From Rule 6.2, the spacing must be at least 3 λ unless the two metal sections should be joined, in which case the spacing would be zero. As an exercise, the reader should find the location of each of the errors listed in Fig. 10.4-7.

Once the cause of an error is determined, corrective action must be initiated. Since the DRC output gives the exact x and y coordinates of the violation, it is usually relatively simple to use an interactive graphics CRT to display the error. Actually correcting the error may not be so simple. If the layout is loosely packed, correction in place by adjusting a single geometrical figure can possibly be done. For some layouts, however, an error will occur in a space-critical area,

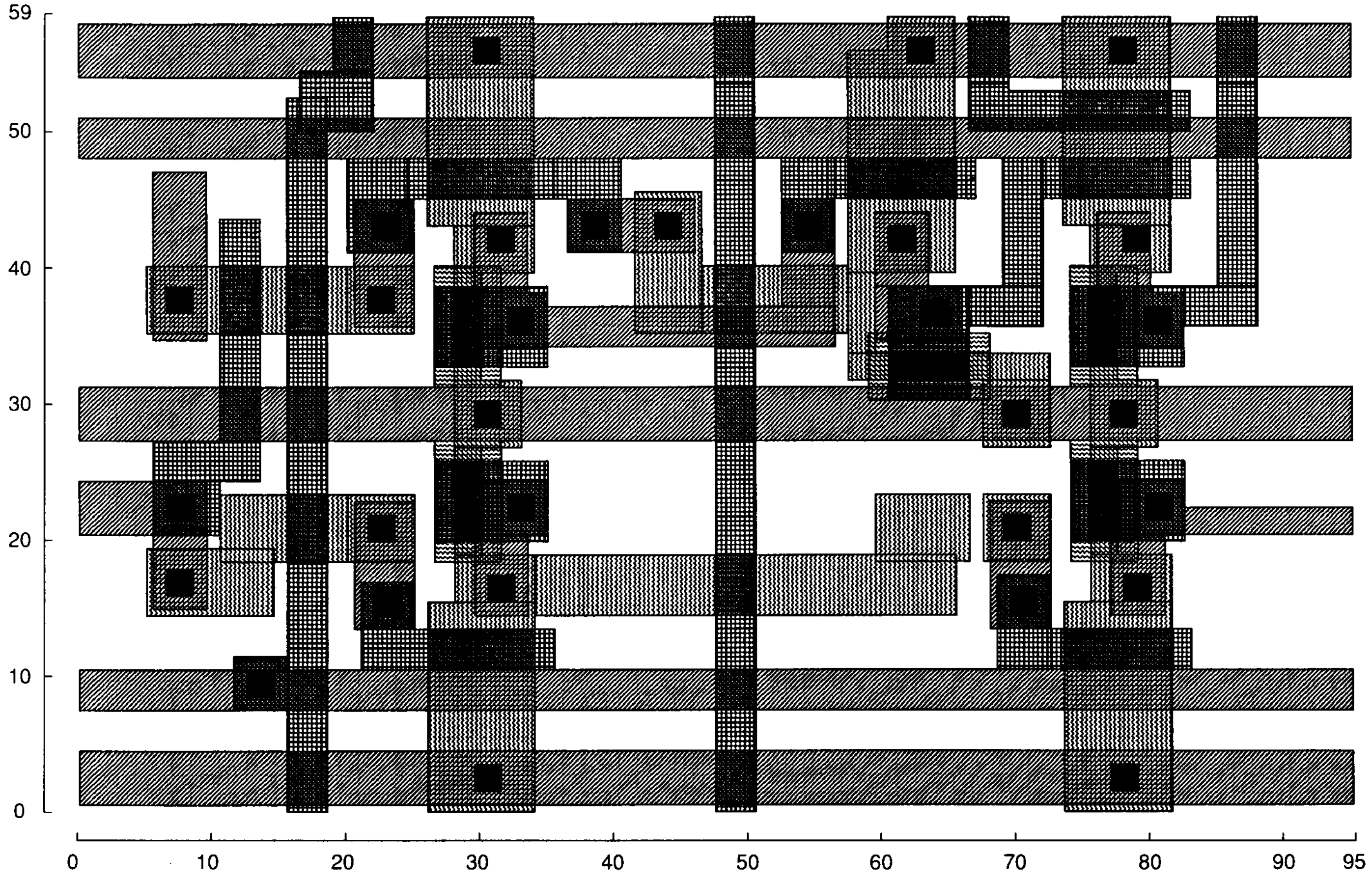


FIGURE 10.4-6
Sample layout for DRC.

```
LDRC version 3.115

Design rule check of file: BGA.TAM
Date  9-MAR-89  Time 21:08:05
X min =   0.0  X max =    95.0
Y min =   0.0  Y max =    59.0

Macro name is BGMLT
Macro number is  99

Vertical errors

Rule     X loc     Y loc     Error     X len
---------------------------------------------
6.2       5.5      47.5      1.0        4.0
5.3      22.0      17.0      0.5        2.0
6.1      82.5      20.5      1.0       12.5

Vertical error count:  3

Horizontal errors

Rule     X loc     Y loc     Error     X len
---------------------------------------------
4.3      10.5      20.5      0.0       3.0
4.2      18.5      41.5      0.5       7.0
1.2      66.5      18.5      1.0       5.0
5.6      80.0      41.5      1.0       2.0

Horizontal error count:    4

Total number of Design rule violations:    7

Design-Rule Checker Execution:

 CPU Time    0: 0:26.06
 Page Faults       354
```

FIGURE 10.4-7
DRC output for Fig. 10.4-6.

requiring changes of a large number of geometries. For this reason, it is crucial to generate a correct layout through automatic means or, in the case of a handcrafted design, to check the layout frequently for geometrical design-rule errors as it is generated. With care, errors are caught early before correction causes difficult problems.

TABLE 10.4-1
Design rule error definitions

Rule	Length	Definition
1.2	$3\,\lambda$	Diffusion spacing
4.2	$2\,\lambda$	Polysilicon spacing
4.3	λ	Polysilicon-to-diffusion spacing
5.3	λ	Polysilicon larger than contact
5.6	λ	Metal larger than contact
6.1	$3\,\lambda$	Metal width
6.2	$3\,\lambda$	Metal spacing

The DRC program used here was run in the batch mode on a computer after the layout was complete. Many CAD systems allow DRCs as geometries are entered through an interactive graphics CRT using an incremental DRC program. Either the designer is prevented from placing geometries that would violate design rules, or a pending violation is flagged immediately by an error message. This minimizes the need for major changes after the layout is almost complete.

DRCs are one of the more time-consuming, yet important, design verification steps. Both polygonal and raster scan DRCs are possible. A good DRC program provides output that accurately identifies the type and location of each error. A good interface between the DRC program and an interactive graphics editor is important for displaying and correcting DRC errors.

10.5 CIRCUIT EXTRACTION

After the design and layout process is complete, MOS circuits are characterized by a machine-readable specification prior to the mask-making step. This specification is usually a geometrical specification file as described earlier. This file contains all the information about the geometries, levels, and placements for the circuit to be fabricated. Because geometrical specification files contain large quantities of detailed information about the integrated circuit, it is difficult for a designer to determine whether this information accurately describes the circuit that was intended. Fortunately, computerized methods exist to extract the circuit information from the geometrical specification file. The process of extracting the circuit information from the geometrical description is called *circuit extraction*.

A circuit extraction program expands the geometrical specification file of the integrated circuit into a layer-by-layer description of the geometries and their placements. This description is then scanned to locate all transistors and interconnections for the circuit. A result of the circuit extraction program is a net list. A *net list* is a set of statements that specifies the elements of a circuit (for example, transistors or gates) and their interconnection. Individual transistors are described along with the nodes to which they connect. This information allows creation of a circuit diagram based on the actual geometrical specification file. Most importantly, the extracted circuit can be compared with the original circuit specified by the designer so that differences are annotated. A difference usually indicates an error that must be corrected. This comparison is called an LVS (layout versus schematic) design verification step.

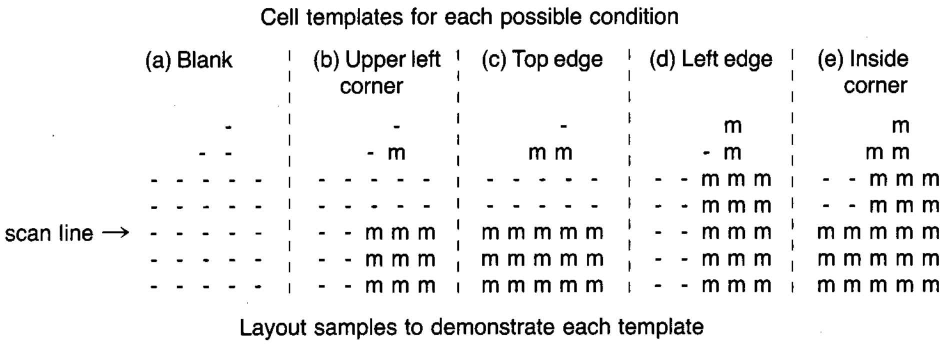
In addition to providing the details of circuit interconnections, circuit extraction is useful for calculating layout areas and perimeters for each integrated circuit layer at each node of the circuit. These layout areas and perimeters can be used to accurately calculate the parasitic capacitances and resistances that load the active devices. Prior to the layout and extraction step, most circuit parasitics can only be estimated by the designer. With accurate capacitance and resistance values from circuit extraction, a design can be accurately simulated to ensure correct operation. Thus, circuit extraction is an essential design verification tool for accurate characterization of modern integrated circuits.

10.5.1 A Simple Circuit Extraction Algorithm

One simple method of circuit extraction consists of two main steps. First, the geometrical specification file is instantiated as a set of coordinates and levels within a computer memory. This is essentially the same operation that was required for the raster scan DRC described in the previous section. This method requires a large computer memory to store the integrated circuit levels at a resolution matching the smallest features of the integrated circuit. For example, a 5 mm by 5 mm die using a process with a λ of 1 μ could require over 25 million individual memory locations to store the instantiated layout, where each memory location corresponds to a 1 λ by 1 λ cell of layout. For a CMOS process, roughly 14 possible layout levels must be remembered for each location. This results in a storage requirement of more than 350 megabits. One useful approach to minimize the memory requirements is to instantiate the design file in overlapping strips. All required circuit information is extracted from each strip before the next strip is instantiated.

The second main step in circuit extraction is the extraction of transistor and connectivity information from the instantiated layout. This is a straightforward, but time-consuming task. The instantiated layout is scanned using a format typical of that used to display television images. The scanning order described here is left-to-right and top-to-bottom, with all integrated circuit levels scanned in parallel. Information on the extent of each level is obtained, and relations between levels that form transistors and contact cuts are derived.

A simple algorithm to determine connectivity at each level can be described as follows. This algorithm requires the program to look at the current cell, the cell to the left, and the cell above. Figure 10.5-1 shows conditions of interest where a "–" indicates no level present and an "m" indicates the presence of a level (e.g., metal). If the current cell does not contain a level, action is not required. This condition is shown in Fig. 10.5-1*a* by a template (upper part) and a 5 λ by 5 λ layout sample (lower part). If the current cell contains a level, four possible cases are of interest; these are shown in Figs. 10.5-1*b* through 10.5-1*e*.


Cell templates for each possible condition
(a) Blank
(b) Upper left corner
(c) Top edge
(d) Left edge
(e) Inside corner
scan line →
Layout samples to demonstrate each template

```
            (a) Blank  | (b) Upper left | (c) Top edge | (d) Left edge | (e) Inside
                       |     corner     |              |               |   corner
                    -  |      -         |      -       |      m        |     m
                  - -  |    - m         |    m m       |    - m        |   m m
            - - - - -  |  - - - - -     |  - - - - -   |  - - m m m    | - - m m m
            - - - - -  |  - - - - -     |  - - - - -   |  - - m m m    | - - m m m
scan line → - - - - -  |  - - m m m     |  m m m m m   |  - - m m m    | m m m m m
            - - - - -  |  - - m m m     |  m m m m m   |  - - m m m    | m m m m m
            - - - - -  |  - - m m m     |  m m m m m   |  - - m m m    | m m m m m
```

FIGURE 10.5-1
Connectivity extraction.

If the current cell contains a level, say metal, then four cases must be examined. First, if neither the cell to the left nor the cell above contains metal, then an *upper left corner* has been encountered as in Fig. 10.5-1*b*, and a new node number must be assigned to this location. As a second case, if the cell to the left contains metal but the cell above does not, as in Fig. 10.5-1*c*, then the extractor is moving along a *top edge*, and the current node is assigned the same node number as the cell to the left. As a third case, if the cell above contains metal but the cell to the left does not, as is shown in Fig. 10.5-1*d*, a *left edge* has been found, and this node is assigned the same node number as the cell above. As a final case, if both the cell above and the cell to the left contain metal, either an *internal point* or an *inside corner* has been found. If the node numbers for these cells are different, they should be merged. The inside corner template and sample layout section are shown in Fig. 10.5-1*e*.

The procedure just described produces a list of nodes for each level and a list of nodes that should be merged. Other information is also kept: for example, a count of the number of times each node is encountered (the area), a count of the number of nodes along an edge (the perimeter), and the location of the first occurrence of each node. In addition, relationships between levels such as contact cuts result in a second node merge list. This node merge list must be kept separate from the homogeneous node merge list since the contact cuts represent nodes of different materials that are connected. Electrically they represent the same circuit node, but for capacitance and resistance calculations their individual identity, area, and perimeter must be maintained.

Other interactions between levels must also be considered. Wherever polysilicon and diffusion are coincident, an additional level (transistor) must be created. This artificial level is processed in a manner similar to the other levels to generate individual transistors and maintain their areas for capacitance and drive strength calculations.

10.5.2 Circuit Extractor Output

As a minimum, the output from a circuit extraction program should contain a complete list of transistors showing the type of transistor (p-channel, n-channel, depletion, etc.) and the nodes to which the transistor is connected. The circuit of Fig. 10.5-2 was extracted to show typical output. A sample of such output, called a net list, is shown in Fig. 10.5-3.

The extracted output of Fig. 10.5-3 lists an arbitrary transistor number; the drain (DS1), source (DS2), and gate (G) connections; the type of transistor (enhancement or depletion); the shape (*ok* means rectangular); the length and width of the transistor; and the x and y coordinates of the upper left corner of the transistor. All dimensions are based on the parameter λ. The resolution of Fig. 10.5-2 and its extracted output listings is 0.5 λ. With this information, transistor size can be verified, individual transistors can be located, and the V_{DD} connection for the depletion transistors (the normal case) can be verified. The net list provides sufficient information to allow reconstruction of a transistor-level circuit diagram

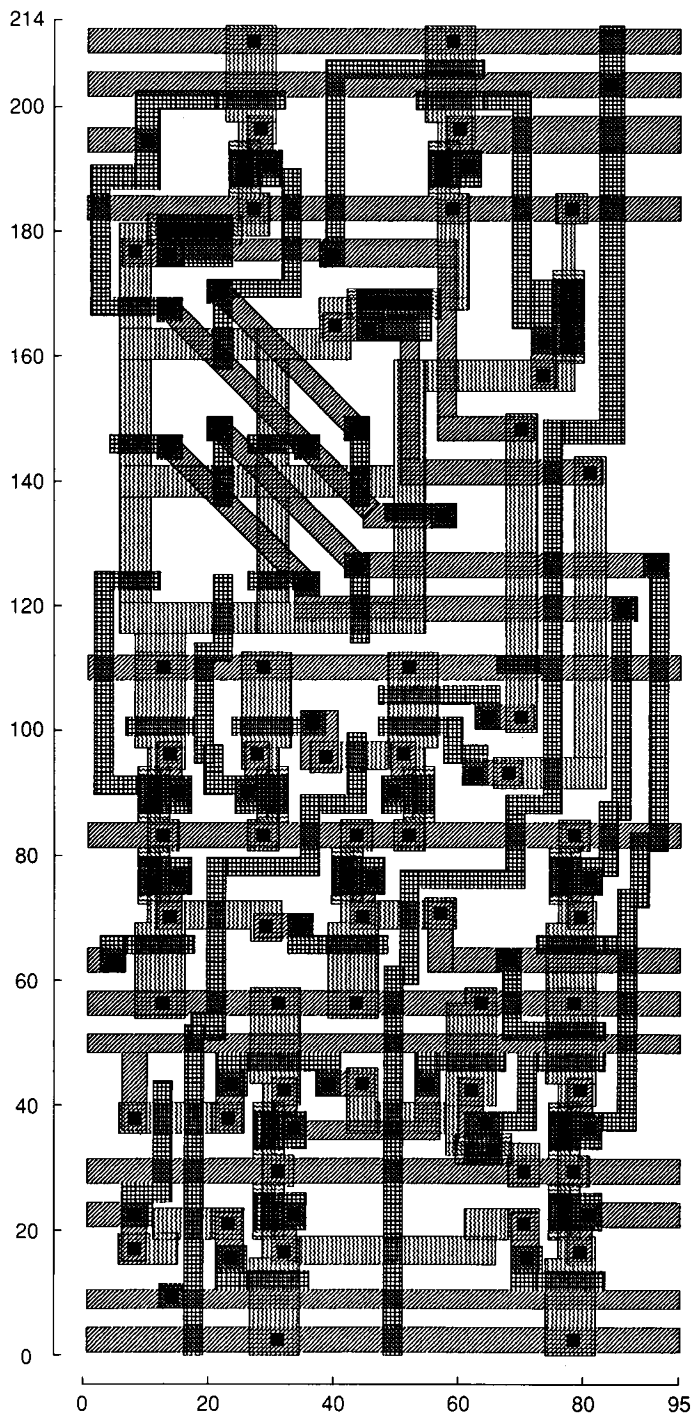


FIGURE 10.5-2
Sample layout for circuit extraction.

```
LEXTRACT version 3.337

Date  4-MAR-89  Time  19:54:46

X min =   0.0  X max =    95.0
Y min =   0.0  Y max =   214.0

Macro name is BGMLT
Macro number is  99

Final merge node list

Num    DS1    DS2    G    Type   Shape   Length   Width   X-loc   Y-loc
--------------------------------------------------------------------------
 1     GND     42    3    enhN   ok        3.0     8.0     54.0   208.5
 2     GND      6    4    enhN   ok        3.0     8.0     22.0   203.5
 3      42      7    5    enhN   ok        3.0     8.0     54.0   203.5
 4       6    VDD    6    depN   ok        6.0     2.0     24.0   194.0
 5       7    VDD    7    depN   ok        6.0     2.0     56.0   194.0
 6       3    VDD    3    depN   ok       12.0     2.0     11.0   182.0
 7     VDD      5    5    depN   ok       12.0     2.0     76.0   173.0
 8       3     51    4    enhN   ok        3.0     5.0      5.0   170.0
 9       9    VDD    9    depN   ok       12.0     2.0     43.0   170.0
10      51      9    6    enhN   ok        3.0     5.0     20.0   165.0
11      51     55   11    enhN   ok        3.0     5.0      5.0   148.0
12       9     12    4    enhN   ok        3.0     5.0     27.0   148.0
13      55     12   10    enhN   ok        3.0     5.0     20.0   143.0
14      12      5    6    enhN   ok        3.0     5.0     42.0   143.0
15       5     13    4    enhN   ok        3.0     5.0     49.0   137.0
16      55    GND   14    enhN   ok        3.0     5.0      5.0   126.0
17      12     17   11    enhN   ok        3.0     5.0     27.0   126.0
18     GND     17   15    enhN   ok        3.0     5.0     20.0   121.0
19      17     13   10    enhN   ok        3.0     5.0     42.0   121.0
20       3     18    2    enhN   ok        3.0     5.0     67.0   112.5
```

FIGURE 10.5-3
Partial net list generated from Fig. 10.5-2 by circuit extractor (VDD and GND labels entered by user).

(not shown) for the integrated circuit. The extracted circuit diagram can be compared with the intended circuit diagram for omissions or errors.

Additional information based on the circuit extraction should be provided. For example, for each integrated circuit layout level, a complete list of nodes with their corresponding areas and perimeters can be provided. If the capacitance per unit area is known for each level, the circuit extraction program can provide an accurate estimate of the capacitance at each node. Figure 10.5-4 provides a partial circuit extractor output for the layout of Fig. 10.5-2 showing the details of the integrated circuit layers that form the nodes of a circuit. For each extracted geometry, this output lists the area, top edge length, left edge length, x and y coordinates of the upper left corner of the geometry, the new merged node number, the old node number assigned to the geometry during extraction, the layout level, and the node name (if any).

The output of Fig. 10.5-4 shows that node 1* is composed of a diffusion geometry (level 1) with area of 84 square units and perimeter of 37 units, a metal

```
LEXTRACT version 3.337

Date  4-MAR-89  Time  19:54:46

X min =  0.0  X max =   95.0
Y min =  0.0  Y max =  214.0

Macro name is BGMLT
Macro number is  99

Final merge node list

Area     Top    Left   X-loc   Y-loc   New   Old  Lev  Name
------------------------------------------------------------
 84.0    8.0   10.5    22.0   214.0    1*     1    1
  4.0    2.0    2.0    25.5   212.5           5    5   GND
380.0   95.0    4.0     0.0   213.5           4    4
  4.0    2.0    2.0    57.5   212.5           6    5   GND
 44.0    8.0    5.5    54.0   214.0           2    1
254.0    4.0   63.5    82.0   214.0    2*     3    3   Phi-2
  4.0    2.0    2.0    83.0   205.5          11    5
154.0   13.0   38.0    73.0   150.5          87    3
380.0   95.0    4.0     0.0   206.5           9    4
 90.0   10.5   22.5    65.5   112.5         135    3
 54.0    9.0   12.0    67.0    90.0         193    3
 97.5   20.0   15.5    50.0    78.0         222    3
195.0    5.5   62.5    47.5    62.5         260    3
 27.0    5.0    6.0    24.0   188.0    8*    33    1
  4.0    2.0    2.0    25.5   185.5          38    5   VDD
 12.0    6.0    2.0    23.0   182.0          46    1
380.0   95.0    4.0     0.0   186.5          37    4
  4.0    2.0    2.0    57.5   185.5          39    5   VDD
 87.0    5.0   18.0    56.0   188.0          34    1
 12.0    6.0    2.0    55.0   170.0          68    1
  4.0    2.0    2.0    76.5   185.5          40    5
 43.0    5.0   14.0    75.0   187.0          36    1
193.0   15.0   20.5     5.0   123.0   16*   122    1
  4.0    2.0    2.0    11.0   111.5         137    5   GND
380.0   95.0    4.0     0.0   112.5         134    4
  4.0    2.0    2.0    27.0   111.5         139    5   GND
 84.0    8.0   10.5    24.5   113.0         132    1
335.0   95.0   15.5     0.0    51.5   30*   273    4
  4.0    2.0    2.0     6.5    39.0         308    5   B-in
 27.5    5.5    5.0     5.0    40.5         299    1
```

FIGURE 10.5-4
Partial layer detail generated by circuit extractor for Fig. 10.5-2.

geometry (level 4) with area of 380 square units, another diffusion geometry (level 1) of 44 square units area and 27 units perimeter, and two contacts (level 5) with area 4 square units each. The x and y coordinates of the upper left corner of each geometry are given, allowing location of the geometry on a display or plot. With the area and perimeter sizes determined, calculation of interconnection capacitances is relatively easy using the values from Table 10.5-1. Example 10.5-1 demonstrates this calculation.

TABLE 10.5-1
Typical capacitance values (from Table 2B)

Layer	Capacitance
Metal	$0.025\ \mathrm{fF}/\mu^2$
Polysilicon	$0.045\ \mathrm{fF}/\mu^2$
Gate	$0.7\ \mathrm{fF}/\mu^2$
Diffusion (bottom)	$0.33\ \mathrm{fF}/\mu^2$
Diffusion (sidewall)	$0.9\ \mathrm{fF}/\mu$

Example 10.5-1 Calculation of nodal interconnect capacitance. For a typical MOS process, parasitic capacitance values to ground are given in Table 10.5-1. Determine the total capacitance for node 1* of the circuit extraction output given in Fig. 10.5-4. The units of extracted dimensions are μ.

Solution: The total capacitance to ground at node 1* is the sum of the capacitance of the layers that compose the node (the contact capacitances are neglected). The capacitance can be calculated as follows.

$$C_{\text{total}} = C_{\text{diff}} + C_{\text{sidewall}} + C_{\text{poly}} + C_{\text{metal}}$$

$$C = (84 + 44)0.33 + (37 + 27)0.9 + (0)0.045 + (380)0.025\ \text{fF}$$

$$C = 42.24 + 57.6 + 9.5\ \text{fF}$$

$$C = 109.34\ \text{fF}$$

If the geometrical specification language allows names to be assigned to nodes, the names can be associated with their respective nodes by the circuit extraction program. The ability to name nodes adds to the complexity of the circuit extraction program since the name information must be kept after the geometric layout is instantiated. This adds substantially to the active computer memory required during a circuit extraction.

A node list with associated names is particularly valuable when checking for open circuits and short circuits. For example, if all power and ground nodes are named (V_{DD} or GND) and an individual node is associated with both the names V_{DD} and GND, a short circuit between power and ground is indicated. This is *not* desirable! Conversely, if the name GND is associated with two disjoint nodes, an open circuit may be indicated for the GND node. Of course, these same name tests can be applied to signal nodes and names, and this can be automated to report potential problems. Figure 10.5-4 shows circuit extractor output for a circuit with named nodes.

The nodes of Fig. 10.5-4 are named GND, Phi-2, VDD, and B-in. The fact that two separate nodes (1* and 16*) are named GND is cause for suspicion. This may indicate a discontinuity in the ground connection or, as in this case, it may be the result of extracting a partial layout. It is very important to provide node names early in a design and carry these names through the layout and simulation steps.

10.5.3 Interface to Other Programs

The output from a circuit extraction program can provide valuable input to circuit and logic simulation programs. Without circuit extraction results, circuit and logic simulations are based on manual input of the intended circuit connections and estimated circuit parameters. If certain process characteristics such as layer capacitance and transistor conductance are provided, a computer program can combine the circuit extraction output with process characteristics to create an input file for circuit simulation and logic simulation. Automatic generation of the input files eliminates human error in providing these data and allows accurate specification of capacitance values and transistor sizes.

Many modern integrated circuits are designed with a high-level circuit description provided in the form of a hardware design language (HDL). If this high-level description allows specification of circuit connections, a particularly important check on circuit integrity can be performed as a result of circuit extraction. The top-down circuit description from the HDL can be compared directly with the bottom-up circuit description from the extracted circuit. This check is valuable because it allows comparison between the designer's intent and the actual computer specification used to generate the fabrication masks.

Circuit extraction is a valuable design verification tool. With the aid of an LVS program, the extracted circuit can be compared to the intended circuit. Circuit extractor derived capacitances and resistances are extremely valuable for accurate circuit simulation. The use of named nodes in the geometrical specification file and subsequent extraction of these nodes allows open, short, and circuit continuity tests.

10.6 DIGITAL CIRCUIT SIMULATION

Accurate circuit simulation is essential for the design of analog circuits such as filters, comparators, and operational amplifiers. The need for circuit simulation extends to the design of semiconductor memory chips even though their data are stored in binary or digital form. For example, extremely sensitive sense amplifiers are required within DRAM circuits to respond to the small change in voltage caused by selecting a storage cell. SRAM circuits often use differential sensing circuits to increase the speed of the data access operation. Both of these memory types require accurate circuit simulation for proper design. Circuits whose external operation is totally digital may require accurate circuit simulation to model critical signal-delay paths. Circuit simulations of high accuracy are almost universally performed with a version of the SPICE circuit simulator described in Chapter 4.

Because of the large number of transistors in digital circuits such as microprocessors, peripheral controllers, and digital signal processors, it is not computationally feasible to perform a circuit simulation for the entire circuit. Since the execution time of circuit simulation programs increases at a rate that is only slightly less than the square of the number of nodes under consideration, verification of the operation of large circuits must be accomplished by other means. Many times a simulation at the logic or switch level (described in the next section) can provide sufficient verification of a digital circuit's functionality. Sometimes even logic simulation programs are too slow to model an entire processor's behavior. Special-purpose hardware simulators are required in these cases.[14]

An intermediate class of circuit simulators is being investigated to provide accurate circuit simulation without the computational penalty of a full circuit simulator.[15,16] These new simulators usually depend on one of two characteristic features of digital circuits. First, most digital circuits are loosely coupled. This means that disjoint parts of the circuit may be relatively independent of one another. There are methods that take advantage of this structure by partitioning the network to simplify the equations that must be solved during a simulation. Second, only a small part, perhaps 25%, of a digital circuit is active during each clock cycle. If a circuit simulator can take advantage of those quiescent portions of the circuit, then only a small part of the circuit will result in simulation calculations at any given time. In either of these two cases, accurate digital circuit simulation can proceed at a relatively rapid rate compared to standard circuit simulation. Nonetheless, digital circuits of any size are rarely simulated in their entirety with circuit-level simulators. Rather, switch-level or logic-level simulators are preferred. Such simulators are described in the next section.

10.7 LOGIC AND SWITCH SIMULATION

Digital integrated circuits are designed to operate with binary representations for data. The basic presumption is that only two logic states are important for each signal line. Thus, knowledge of a precise voltage versus time characteristic for each node in the circuit is not necessary to design or analyze digital circuits. For many purposes, this simplifies both the circuits and their analysis compared to analog circuits. Nevertheless, computer simulation and verification of a circuit's functionality are necessary. Even though a digital circuit is designed based on logic gates, the logic gates are fabricated from the basic transistors and conductors allowed by the integrated circuit process. Therefore, it is often the case that the electrical operation of a simple logic circuit must be characterized by using a circuit simulator such as SPICE.

Though circuit simulation of digital circuits is frequently used, such circuit simulation has several drawbacks. As described in the previous section, the large number of logic gates in most digital integrated circuits precludes circuit simulation of the entire system because of the extended computer time required. Also, standard circuit simulators provide more detail about circuit voltages than is required to analyze a logic circuit. In an effort to reduce computer simulation time and to provide appropriate data to characterize the operation of digital circuits, *logic simulators* were developed.

10.7.1 Logic-level Simulation

Logic simulators allow specification of the operation of a circuit block in terms of its behavior. For example, a simple logic gate is described by its behavior, such as AND, OR, or NOT. More complex digital blocks such as full adders and multiplexers are each described by their corresponding behavior rather than their circuit structure. The circuit inputs are specified as binary values that change at discrete time intervals. Logic simulator outputs are provided as binary values as well. Pure logic simulation does not model time delays through logic blocks. Only the logical behavior of the simulated system is considered, although the concept of sequence wherein one action precedes another is important. Timed logic simulation considers the delays of logic gates and blocks in determining when outputs will change. Because a logic simulator models the circuit in terms of an abstracted (less detailed) representation, larger circuits can be simulated in a much shorter length of time than with circuit simulation. Consider the following example.

Example 10.7-1 Comparison of circuit and logic simulation. In terms of the number of circuit elements, nodes, and calculations, compare circuit simulation and logic simulation requirements for a full adder built from a classical CMOS circuit and from CMOS gates.

Solution

Circuit simulation. The two-level logic circuit for a classical CMOS full adder requires 56 transistors and 33 nodes. This circuit is shown in Fig. 10.7-1. In addition, continuous input waveforms that generate the eight possible logic input conditions of three inputs must be provided. Each of these conditions must be stable for a length of time sufficient to allow the sum and carry outputs to stabilize. This requires about 100 to 200 time steps for each input condition. As a rough estimate, a minimum of 800 to 1600 time-step calculations would be required to characterize the full-adder operation.

Logic simulation. A classical two-level logic circuit for a full adder requires three inverters, three 2-input NAND gates, five 3-input NAND gates, and a 4-input NAND gate, for a total of 12 logic gates and 15 nodes. The logic gate implementation is given in Fig. 10.7-2. Eight possible input combinations exist for the full adder. Each of these combinations generates a digital value for the sum and carry outputs. Correctness of the sum and carry outputs is easily verified by these eight calculations.

Thus, circuit simulation requires approximately 1600 time-step calculations involving 56 transistors and 33 nodes at each calculation. Logic simulation, on the other hand, requires only 8 calculations, involving 12 logic gates and 15 nodes for each calculation. Clearly, if simulation of the logical operation of the full adder is the goal, logic simulation is simpler and faster. If accurate signal propagation time or waveform characteristics are required, then circuit simulation is necessary.

Commercial logic simulators model digital logic in terms of four or more states. As a minimum, these states include 1, 0, X, and Z. The logic values 1 and 0 model the high and low logic states. The value X is used to model an unknown condition. For example, the value of an internal logic node may be unknown when simulation is started. The value Z is used to model high-impedance (undriven) nodes. A tri-state bus with all driving circuits turned off is an example of this condition. Additional states may be defined to model the relative driving strength of logic outputs. Of course, as the number of allowable states increases, the simulator complexity and run time increase correspondingly.

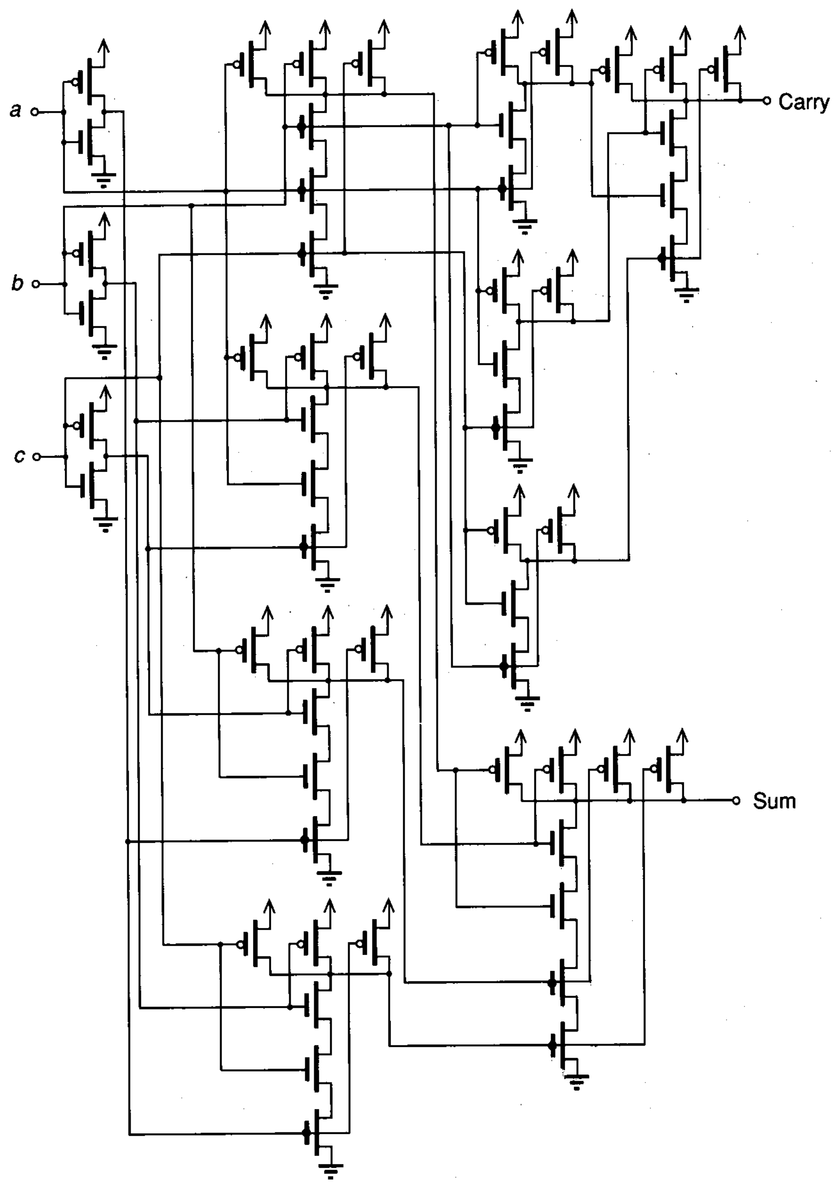


FIGURE 10.7-1
Classical CMOS full-adder circuit.

Many logic simulators provide a variety of digital blocks for use in modeling a digital system. Besides the simple logic gates and more complex logic blocks mentioned previously, models for large digital blocks such as ROMs, RAMs, PLAs, ALUs, and even FSMs are often provided. Simulation capability is normally provided for both synchronous and asynchronous sequential circuits in addition to simple combinational logic.

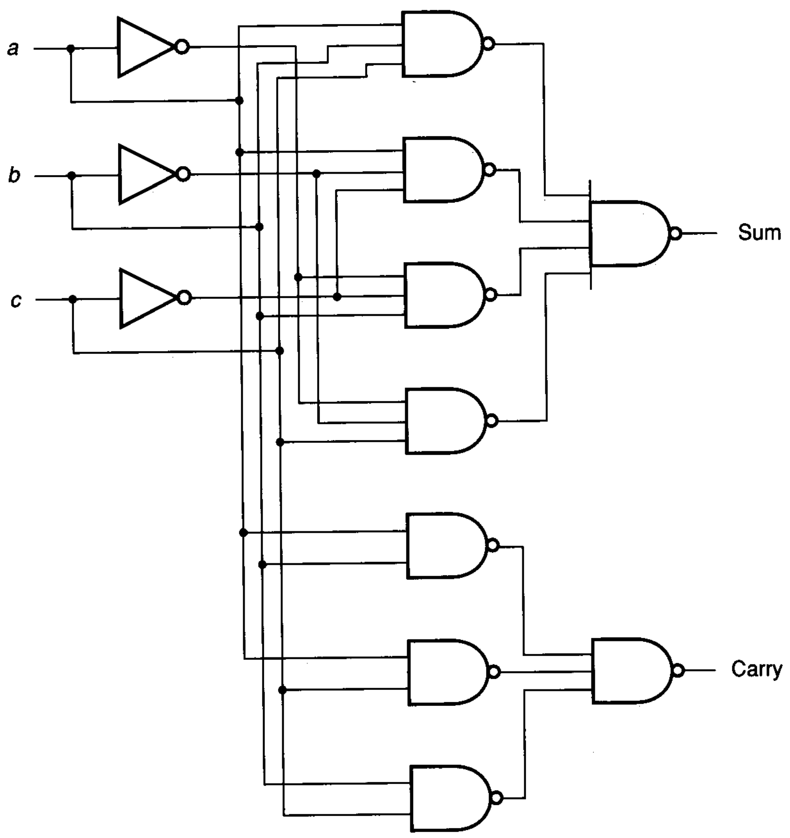


FIGURE 10.7-2
NAND-NAND full-adder logic diagram.

Most logic simulators today are *event driven*. That is, calculations are required only in response to external or internal events. External events include changes in the state of inputs to the circuit. An internal event occurs when the output of a logic function changes in response to changes in its inputs. For example, when the input to an inverter changes, the corresponding change in the inverter output is considered an event. The use of event-driven rather than fixed time-step simulation algorithms reduces the time required for simulation of a circuit.

The capability of logic simulation is often measured in terms of *events per second* or *evaluations per second*. Whenever the inputs to a logic block change, an *evaluation* must occur to determine the correct output for the logic block. Thus, an evaluation is the application of a circuit's inputs to its behavior in order to determine its outputs. An average factor of 2.5 evaluations per event is typical for digital circuits. The performance of logic simulators depends on many factors including the number of logic states, the cleverness of the algorithms chosen for simulation, and the execution speed of the computer on which the simulator is run. An execution rate of several thousand events per second is common for today's logic simulators on typical computer workstations.

10.7.2 Switch-level Simulation

MOS integrated circuits present special problems for standard logic simulators because of bidirectional pass transistors, transmission gates and charge storage. Pass transistors are used frequently because of their desirable power dissipation and interconnection characteristics. Pass transistors are difficult to simulate as simple logic gates with a standard logic simulator. It might seem that the pass transistor of Fig. 10.7-3*a* could be simulated by using the AND gate of Fig. 10.7-3*b*. The following discussion shows why this is impractical.

A simple analysis of the operation of the circuits of Fig. 10.7-3 shows that the two circuits are not equivalent. Assume initially that both inputs and the output are low for both circuits. If a logic 1 is placed on a single input, the output remains low for both circuits. If a logic 1 is placed on both inputs, the output goes high for both circuits. If a logic 0 is placed at the *i* input of the two circuits, the output goes to a 0 for both circuits. Thus far, the operation of the two circuits seems identical. However, assume that all inputs and outputs are initially high. Further, consider that the source diffusion of the output of the pass transistor provides parasitic capacitance to ground. If the *c* input to both circuits is moved to a logic 0, the AND gate output goes to a logic 0 while the pass transistor output remains high because of the charge storage at its output. Clearly, the operation of the pass transistor cannot be accurately modeled in this fashion. Either a more complex logic circuit is required, or the logic simulator must be modified to account for drive strengths and charge storage. Examples of drive strength are *driven*, *resistive pullup*, and *undriven*. The output of the pass transistor just considered is undriven when its gate terminal is at 0 V.

Because selector circuits constructed from pass transistors and transmission gates are widely used within MOS circuitry, a logic simulator for MOS must allow specification of individual transistors and their connections in addition to simple logic gates. When a logic simulator can describe transistors in addition to standard Boolean logical primitives, it is called a *switch-level simulator*.

A typical switch-level simulator operates on circuits described by nodes, transistors, and logic gate primitives. Nodes are equipotential points to which one or more terminals of one or more transistors or logic primitives are connected. Each node has an associated name, logic state, capacitance (to ground), list of events, and perhaps other information. Each transistor has a type (n-channel, p-channel, or depletion), effective resistance (width and length are required), and node connection for its terminals. Macros are often allowed to describe circuits composed of several transistors; for example, logic gates may be constructed from nodes and transistors. These logic gates are then used as primitives.

A byte-wide MOS binary adder circuit will be used as an example to show the operation of a switch-level simulator.[17] The circuit for a full-adder stage is

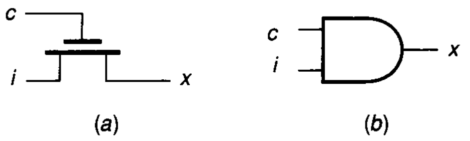


FIGURE 10.7-3
(*a*) Pass transistor logic, (*b*) AND-gate logic.

given in Fig. 10.7-4, and the corresponding input net list for the switch-level simulator is provided in Fig. 10.7-5. This net list describes the circuit of Fig. 10.7-4 in terms of four primitive elements: invert, trans, nor, and pulldown. The net list starts with a definition of a single-bit adder macro and its five inputs {*a b cif phil phi2*} and two outputs {*sum cof*}. Additionally, seven local signals {*af bf ci p pf k phi2f*} that are internal to the full-adder macro are specified. Each primitive element is then instantiated with its connections to other circuit nodes defined by arguments. The formats for these four procedure calls are: (invert out in), (trans gate source drain), (nor out in0 in1 in2), and (pulldown drain gate).

Next, eight single-bit full adders are combined to define a byte-wide binary adder, as shown in Fig. 10.7-6. The external nodes of the byte-wide full adder are first defined. The **a**, **b**, **cof**, and **sum** nodes represent 8-bit vectors that are expanded by the repeat statement. Signals *phil* and *phi2* are the nonoverlapping two-phase clock inputs. The connect statement joins the *cifi* carry-input scalar to the first carry-in bit, *cof.0*. The repeat statement next creates eight copies of the full-adder circuit.

The results of a sample switch-level simulation run for the byte-wide adder are explained next. The input vector **a** was set to 11111111, while the input vector **b** was set to 00000000. This condition provides the longest carry propagation path for the full adder. The initial carry-in bit *cifi* is set to the low-true condition. A nonoverlapping two-phase clock is defined with each phase high for 90 ns and a 10 ns separation between phases. The results from a simulation for a complete

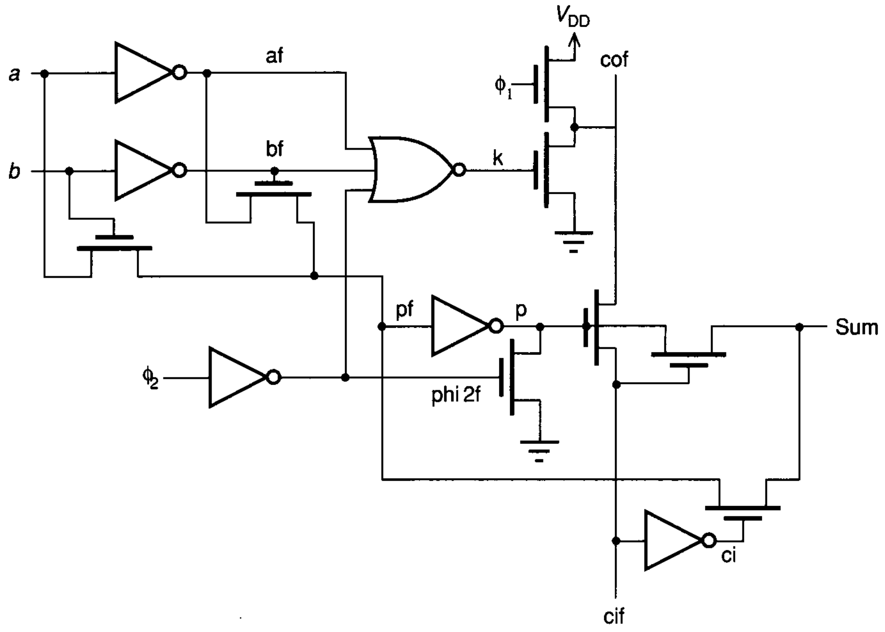


FIGURE 10.7-4
Single-bit slice of clocked full-adder circuit.

```
;Begin Full-Adder Macro
(macro adder (a b cif phi1 phi2 sum cof)
  (local af bf ci p pf k phi2f)
  (invert bf b)
  (invert af a)
  (trans b pf a)
  (trans bf pf af)
  (invert (p 2 16) pf)
  (invert (ci 2 16) cif)
  (trans cif sum p)
  (trans ci sum pf)
  (invert phi2f phi2)
  (nor k af bf phi2f)
  (pulldown cof k)
  (pulldown p phi2f)
  (trans phi1 cof vdd)
  (trans p cof cif)
)
;End Full-Adder Macro
```

FIGURE 10.7-5
Input net list for logic simulator describing circuit of Fig. 10.7-4.

```
;Instantiate Byte-Wide Adder
(node a b cifi phi1 phi2 sum cof)
(connect cifi cof.0)
(repeat i 1 8
  (adder a.i b.i cof.(1 - i) phi1 phi2 sum.i cof.i)
)
;End of Byte-Wide Adder
```

FIGURE 10.7-6
Input net list for a byte-wide binary adder.

cycle (200 ns) of the two-phase clocks are given for the first and last sum (*sum.1*, *sum.8*) and carry-out (*cof.1*, *cof.8*) bits only. Only changes in logic value of these bits are provided; that is, only simulator events for these bits are included. A typical event produces a statement with the format: name = value @ time.

```
φ1 cycle: (t = 0 ns to 90 ns), precharge
   cof.8 = 1 @ 2.4
   cof.1 = 1 @ 2.6
  sum.1 = 0 @ 2.8
  sum.8 = 0 @ 3.2

φ2 cycle: (t = 100 ns to 190 ns), evaluate
   cof.1 = 0 @ 103.2
   cof.1 = 1 @ 104.4
  sum.8 = 1 @ 104.9
   cof.1 = 0 @ 109
  sum.8 = 0 @ 129.8
   cof.8 = 0 @ 130
```

Note that all carry bits are precharged to a 1 during each ϕ_1 cycle. According to the simulation results shown, the *cof.8* bit changed to 1 at 2.4 ns and the *cof.1* bit changed to 1 at 2.6 ns after ϕ_1 was set high. As can be determined from the circuit connections of Fig. 10.7-4, the sum bits should be set to 0 during each ϕ_1 precharge cycle. The *sum.1* bit went to 0 at 2.8 ns and the *sum.8* bit went to 0 at 3.2 ns after ϕ_1 was set high.

During the ϕ_2 evaluate cycle, the carry and sum bits are set according to the sum of the two addends **a** = 11111111 and **b** = 00000000 and the carry in *cifi* = 0 (indicates a carry in). During the evaluate cycle *cof.1* changed to 0 at 3.2 ns, to 1 at 4.4 ns, and then back to 0 at 9.0 ns after ϕ_2 was set high. The most significant carry bit, *cof.8*, was set to 0 some 30 ns after ϕ_2 was set high. Also, *sum.8* was set to 1 at 4.9 ns and then to 0 at 29.8 ns after ϕ_2 was set high. For the input vectors given, each full-adder stage should have set its sum bit to 0 to indicate a sum of 0 and its carry bit to 0 to indicate a carry out of 1 (the carry bits use negative logic). The final results from simulating the first clock cycle are as expected. Note that the final event (*cof.8* set to 0) occurred 30 ns after the ϕ_2 clock was set high.

Prior to the second clock cycle, the carry-in bit is set to a false condition (*cifi* = 1). The following simulation results are for the second clock cycle (200 ns $\leq t <$ 400 ns).

ϕ_1 cycle: (t = 200 ns to 290 ns), precharge
    cof.8 = 1 @ 200.2
    cof.1 = 1 @ 200.4

ϕ_2 cycle: (t = 300 ns to 390 ns), evaluate
    sum.8 = 1 @ 304.9
    sum.1 = 1 @ 304.9

During the second ϕ_1 cycle, the carry bits change as they are each precharged to 1. The sum bits do not change during ϕ_1 since they were already each left set to 0 after the previous ϕ_2 cycle. During the second ϕ_2 cycle, the sum and carry bits should be changed to indicate the sum of the two addends **a** = 11111111 and **b** = 00000000 and the carry in *cifi* = 1. Thus, all sum bits should be set to 1 and all carry out bits should be set to 1 indicating no carry out. The simulation results show that the sum bits are each correctly set to 1 during the second ϕ_2 cycle. The carry bits do not change since they were each set to 1 during the precharge cycle.

For a third clock cycle (400 ns $\leq t <$ 600 ns), the carry in bit is set to 0 again (*cifi* = 0) and the results of the first clock cycle are repeated. These results are as follows.

ϕ_1 cycle: (t = 400 ns to 490 ns), precharge
    sum.1 = 0 @ 402.8
    sum.8 = 0 @ 403.2

```
φ2 cycle: (t = 500 ns to 590 ns), evaluate
   cof.1 = 0 @ 503.2
   cof.1 = 1 @ 504.4
   sum.8 = 1 @ 504.9
   cof.1 = 0 @ 509
   sum.8 = 0 @ 529.8
   cof.8 = 0 @ 530
```

The previous results for three clock cycles demonstrate the operation of a switch-level simulator. Both the timing of the byte-wide adder and the correct logical operation of the adder are observed for the input conditions provided. Other switch-level simulators have different input and output formats and different capabilities, but all operate assuming discretized values for the circuit variables, and all produce results much faster than complete circuit simulation.

10.7.3 Hardware Logic Simulation

Even with the increased speed of logic simulators as compared with circuit simulators, full simulation of large digital circuits via general-purpose computers is not practical. An alternate approach is in use by several companies. Special-purpose hardware that executes many simulation steps in parallel has been developed to speed the simulation process. One of the early, large parallel simulators was the YSE (Yorktown Simulation Engine)[18] developed by IBM. This hardware consists of hundreds of identical processing units that each simulate part of the target circuit. By spreading the calculations over a large number of processors, even large-mainframe computers can be simulated in detail. Of course, such special-purpose hardware is expensive to build and to operate. Even so, several companies now offer hardware accelerators to enhance the speed of logic simulation.

In the future, methods of machine verification other than total logic simulation must be found. Logic simulation time increases exponentially with the number of logic components to be simulated. Thus, faster computers are necessary to simulate next-generation computers that contain more logic components. But how can the next-generation computers be built if the simulation capability of present-generation computers is inadequate?

Two current approaches to this problem are verification proofs and hierarchical simulation. For relatively simple hardware, it has been possible to verify correct logical operation by mathematical proofs. Unfortunately, the utility of this method diminishes quickly as the size and complexity of the hardware increase. The second method, hierarchical simulation, attempts to model the target machine at various levels of abstraction. Small blocks of hardware are verified by logic simulation. These blocks are then interconnected and simulated together without the internal detail of each block. Neither of these methods has been entirely successful, and both are now active areas of research and development.

10.8 TIMING ANALYSIS

For most digital circuits, a very important parameter is the maximum rate at which the circuit can correctly process data. For microprocessors, the processing speed is usually given in MIPS (millions of instructions per second); for scientific calculations, the rate of execution is given in FLOPS (floating-point operations per second); and for logical inference machines, the characteristic measure is LIPS (logical inferences per second). The execution rate of each of these machines is limited by parasitics and governed by its input clock. A primary goal in the design of a digital computing machine is to operate with the fastest possible input clock.

Each digital integrated circuit has a maximum rate of operation. This rate of operation is limited by the output drive capability of its logic elements and by the capacitance and resistance of the loads they must drive. In a FSM (finite-state machine), the clocking rate is limited primarily by the longest path through its combinational logic section. For integrated circuits composed of large blocks of circuitry, the maximum clocking rate may be limited by signal lines that must carry information between the blocks. The designer's task, then, is to find those paths in an integrated circuit design that cause the maximum delay and then to modify the circuitry to minimize that delay.

Finding the longest delay paths, called *critical paths*, for an integrated circuit is not a simple task. Until recently, the most common technique for finding critical delays was for the designer to perform detailed circuit simulation on the paths that were suspected of contributing long delay times. Of course, using circuit simulation for this task was not foolproof. Many times an unsuspected path that was not considered for simulation would limit the maximum clock speed. More recently, computer programs have been designed specifically to seek out delay paths directly from the circuit definition without requiring simulation. This type of computer analysis is called *timing analysis*.

10.8.1 Timing Analysis Methodology

Timing analysis differs from circuit and logic simulation in that all possible signal paths are considered. Circuit simulation and logic simulation both require the specification of input signals to control the simulation. Thus, only delay paths that are exercised by the particular set of inputs are tested. For many digital circuits, it is computationally impossible to provide sufficient input conditions to test the circuit fully. Timing analysis tools work by tracing *signal paths* instead of simulating the circuit for specific inputs. Specifically, timing analysis uses *state-independent* path tracing. Each time a logic gate is encountered, the gate is assumed to pass the signal regardless of the state of the other inputs to the gate. A signal path is terminated only when an output is reached or a clocked storage element is found. With this method, all possible delay paths are tested.

An example of timing analysis signal propagation through two logic gates is shown in Fig. 10.8-1. The signal path starts at input x and reaches the NAND gate. Inputs a and b for the NAND gate are assumed high to allow continuation of the signal path. When the signal reaches the NOR gate, input c is assumed low

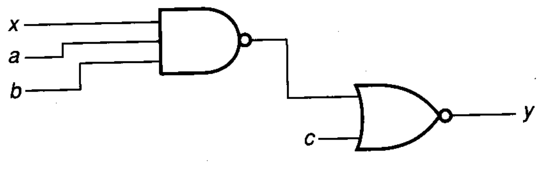


FIGURE 10.8-1
State-independent path trace.

to allow continued propagation of the signal. Finally, the signal reaches an output y, where it terminates. The delay for this signal path includes the contributions of the NAND gate, the NOR gate, and the series interconnections. The delay paths from x to y, b to y, a to y and c to y would all be found by a timing analysis of this circuit.

A second example shows a deficiency of timing analysis. From Fig. 10.8-2, signal paths from a to b and from a to c are expected. However, state-independent path tracing will also find a signal path from b to c and vice versa. Although the path from b to c is a real path, it will not normally be exercised within this circuit because node n is actively driven by the inverter. Analysis of additional paths that will not be exercised during operation of a circuit can degrade the performance of a timing analysis program. Circuit-level timing analyzers allow direction setting for pass transistors and transmission gates to circumvent this problem. Unfortunately, unless this is carefully done, some critical signal paths may be eliminated from consideration.

10.8.2 Timing Analysis Tools

To provide further insight into the capabilities of circuit-level timing analysis programs, two such programs will be described here. The first of these, called TV,[19] attempts to set directions for circuit elements by using rules. These rules, by setting some transistor directions, minimize the number of false paths that are found. The second tool, Crystal,[20] provides a wide range of capability, including improved delay models and coverage for circuits built from CMOS technology.

TV timing analyzer for NMOS designs, operates from extracted circuit parasitics and considers only stable, rising, and falling signal values. Program execution time is minimized by a *static analysis* that sets signal flow direction and clock qualification where possible. Otherwise, signal flow direction is determined from a set of direction-finding rules. Some of the rules are independent of design style. For example, the *constant-propagation rule* says that any transistor source

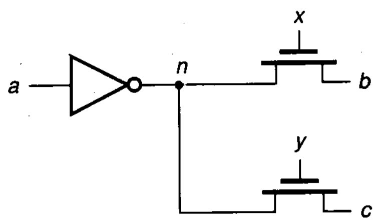


FIGURE 10.8-2
Problem paths for timing analysis.

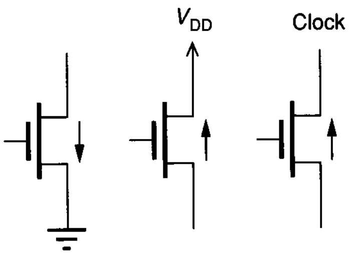


FIGURE 10.8-3
Constant propagation rule to set directions.

or drain connected to power, ground, or a clock must be a sink of signal flow, while the other terminal must be a source. Figure 10.8-3 demonstrates this rule, which by itself sets the directions for more than half the transistors in a typical circuit. Another rule, demonstrated in Fig. 10.8-4, is based on Kirchoff's current law. This rule, the *node current rule*, states that if all but one of the transistors to a node have a known direction, and the known transistors all sink or all source signal flow, then the unknown transistor must transmit flow in the opposite direction relative to the node.

Other signal-flow rules depend on technology or design style. For example, in an NMOS technology design, the *k-ratio rule* for inverters can be used to set direction. This rule is based on a standard device sizing ratio k as discussed in Chapter 7 for ratio logic. By finding the minimum resistance to ground through each unset (direction not specified) transistor connected to a pullup, a transistor can be considered as a pulldown (signal flow toward the pullup) or a pass transistor (signal flow away from the pullup), depending on the resistance ratio. The reasoning is that resistances to ground that satisfy the device sizing ratio k with respect to the pullup path must be part of the pulldown circuit for a logic gate. Transistors that cannot satisfy ratio rules can be safely classified as pass transistors and their direction set accordingly. Other rules cover pass transistors connected to a common node and having a common gate signal, and analogous structures where the direction of a boundary transistor can be determined, thereby allowing arrayed versions of the structure to have their directions set accordingly.

Signal path analysis is started from the clock or other input nodes. Paths are investigated in a breadth-first manner in accordance with the transistor directions that were set by the static analysis. Delays for paths are calculated based on the capacitance of the interconnections and the resistance of driving and series pass

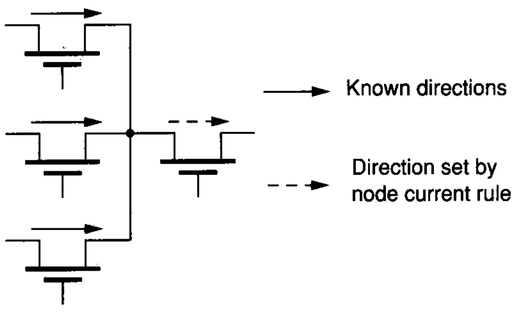


FIGURE 10.8-4
Node current rule.

transistors. Transistors are assigned a rising and a falling resistance from tables based on their use in the circuit. Signal direction changes are propagated so that a rising input signal to an inverter produces a falling output signal and vice versa. Pass transistors continue the direction of the input signal. Because path delays are calculated from a linearized model, the delays may differ from actual circuit values by 30% or more.

Output of the TV program includes a user-selectable number of the worst-case paths. Equivalent paths, such as parallel paths in a data bus, are condensed in the output list so that only the last path in the list is reported. Other useful information such as slack time for paths, excessive power used to drive a noncritical path, and nodes with unusually high capacitance are reported. The TV timing analyzer was successfully used in the analysis of the MIPS series of microprocessor chips[21] developed at Stanford University.

Another timing analysis tool, Crystal, was developed to analyze the RISC computer chips[22] developed at the University of California at Berkeley. This tool has found widespread use throughout the VLSI design community, particularly within universities. The timing analysis is based on a circuit description that is extracted directly from a geometrical specification file. This description includes transistor sizes and types, interconnection capacitance, and a rough calculation of interconnection resistance. A simple delay model is used for each stage to provide quick calculation of signal propagation delays along a path.

The Crystal timing analyzer was developed for MOS circuits with multiple nonoverlapping clocks. The program attempts to determine how long each clock phase must be to allow all signals to propagate to their destinations. The analysis is state-independent, so all possible paths are checked. The user must specify a minimum of information to begin the analysis. For two-phase clocking schemes, only two signals must be specified. One of the clock phases is specified as a rising edge or a falling edge to trigger the analysis. The other clock phase is specified as a stable low value. The reason for this can be seen from the shift register circuit of Fig. 10.8-5. Here a signal path trace is started from the ϕ_1 clock. Without a specified value for the ϕ_2 clock, the signal path would continue through all the stages shown. If the ϕ_2 clock is set to a stable low condition, then the signal path will terminate correctly after the first stage. The path delay

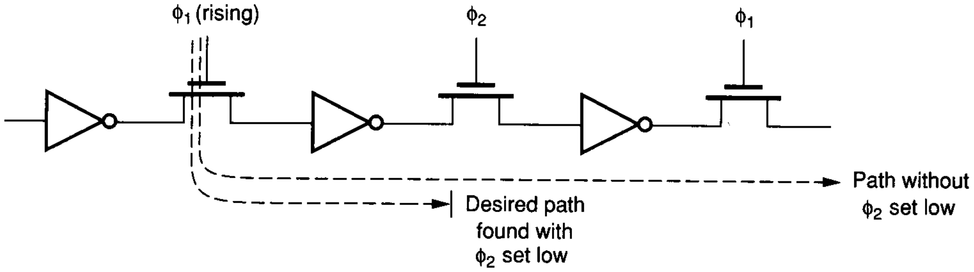

ϕ_1 (rising)
ϕ_2
ϕ_1
Path without ϕ_2 set low
Desired path found with ϕ_2 set low

FIGURE 10.8-5
Clocked path analysis.

will consist of the time for the first inverter to discharge (charge) the input of the second inverter through the pass transistor gated by ϕ_1 plus the time for the second inverter to charge (discharge) the interconnection capacitance up to the input of the pass transistor gated by ϕ_2.

Each path trace for a signal is started from a rising or falling input specified by the user. As the signal path proceeds through inverters or logic gates, the appropriate rising or falling direction is determined to correctly model asymmetric stage delays. The path-trace analysis is done with a depth-first search algorithm. Thus, a signal path is followed until it reaches a circuit output or is stopped by a static signal specified by the user (like the ϕ_2 condition examined in the previous paragraph). Delay information from previous paths is maintained at each node so that the signal path can be aborted on later path traces through the same node if the cumulative delay is less than the stored value.

As with all state-independent timing analysis methods, the possibility of reporting false paths exists. A simple example is given in Fig. 10.8-6, where a signal path is gated by a signal and its complement. From a logical viewpoint, there is not a signal path from node *a* to node *c* because one of the AND gates will be disabled by x or $\bar{x}$. Since timing analysis is state-independent, this logical constraint is not recognized, and the path from node *a* through node *b* to node *c* will be considered and its delay calculated. A 1-of-*n* selector circuit is a classic example of this condition. In normal operation, only one path through the selector circuit will be enabled at any time, but state-independent timing analysis finds all *n* paths. In most timing analyzers the capability exists to set signals to a stable value to disable paths; however, this capability must be used carefully to avoid accidentally disabling critical delay paths.

To facilitate fast operation, Crystal uses a simple delay model consisting of an equivalent resistance for the drive transistor and a resistance and capacitance for the interconnections and load devices. The transistor drive model is table-driven with the equivalent resistance selected based on input signal slope and capacitive load value. This is not as accurate as a circuit-level simulation but is much faster. Once critical delay paths are found, they can be investigated with a circuit simulator if more accurate results are required.

In summary, timing analysis is an important tool for integrated circuit design. By using state-independent path tracing, it performs a function that is difficult, if not impossible, to perform with timed logic simulators. The execution time for timing analysis programs is determined by the size of the circuit

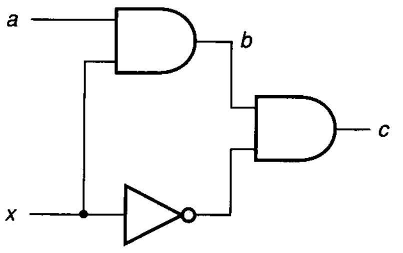


FIGURE 10.8-6
Logically impossible path.

being analyzed. While timing analysis is used to find and correct critical delay paths, correct functional operation can be verified with a logic simulator. Thus, logic simulation and timing analysis function as partners to ensure that a digital integrated circuit is functionally correct and that it operates at the proper speed.

10.9 REGISTER-TRANSFER-LEVEL SIMULATION

Specifications for the operation of digital integrated circuits are often given in terms of high-level operations on information. These high-level operations describe transformations on data as it moves from one storage device or register to another such device. For this reason, descriptions of this type are known as *register-transfer-level* descriptions.

Register-transfer-level descriptions provide a useful level of abstraction for the description and simulation of digital systems. The logic simulators described previously require too much detail about the exact logical structure of an integrated circuit for early design simulation. Also, because of the detailed specification of the logical structure of the circuit, complete logic simulation of an entire circuit such as a microprocessor requires impractically large computer resources. Alternatively, at the highest level, a natural language description of the function of a digital system is often ambiguous and vague. A concise natural language description of a next-generation computer might be, "build a new computer that is like computer XYZ, but is twice as fast and uses less power." To fill this gap between natural language descriptions and logical definitions, high-level description and simulation languages have been developed. Of these, register-transfer-level simulation languages allow specification and simulation of operations on data words, in addition to single-bit operations.

The operation of a digital system can be defined precisely through the use of a register-transfer-level description. In fact, one such language, ISPS (Instruction Set Processor Specification), was developed to allow unambiguous description and specification of computer operation.[23] The ISPS language allows data bits to be grouped into words. Logic and arithmetic operations are allowed on both bit-level and word-level entities as they are moved between storage registers. Operations common to most programming languages, such as conditional statements, if-then-else constructs, case statements, and procedures, are allowed. Thus, a register-transfer language is a special programming language tailored to describing the operation of digital systems.

10.9.1 Simple RTL

For demonstration purposes, a primitive register-transfer language (RTL) will be defined and used to describe the execution of one instruction from an early 8-bit microprocessor. This primitive RTL is defined in Table 10.9-1. The first operation required is the *transfer operation*—the contents of one group of bits (register) are placed into another storage device. Second, the common *arithmetic operations* of add and subtract are provided. Third, a simple *conditional capability* to alter control flow is added.

TABLE 10.9-1
Primitve RTL Definition

Operation	Description
A ← B	transfer
C ← A + B	addition
D ← A − B	subtraction
PC ← B if A = 0	conditional

The operation to be described using this RTL is the extended load of the A accumulator of the Motorola 6802 microprocessor. This microprocessor has a 16-bit address bus and a separate 8-bit data bus. The A accumulator is an 8-bit register. Execution of this instruction requires four memory cycles: fetch the 8-bit instruction, obtain the high byte of the operand address, obtain the low byte of the operand address, and obtain the data from the operand address. The approximate register-transfer-level steps are given in Table 10.9-2 and are explained next.

The first step moves contents of the program counter (PC) to the address bus (AB) in preparation for fetching the instruction byte. While the processor is waiting for the memory to respond, the program counter is incremented. After a delay time, the DBI (data bus input) is moved to the instruction register (IR). This ends the first memory cycle. As the instruction is being decoded, the incremented PC is moved to the address bus in preparation to fetch the next byte. The PC is incremented again, and the contents of the DBI are moved to the internal data bus (DB) and on to a temporary register (TMP) to complete the second cycle. To begin the third memory cycle, the previously incremented PC is moved to the AB and the PC value is incremented. The DBI contents are moved to DB where they are held in preparation for the cycle that outputs the data address (this is slightly oversimplified). The fourth and final cycle moves the data from DB to the low-order bits of the address bus (ABL) and the contents of TMP to the high-order bits of the address bus (ABH). At this point, the extended 16-bit address of

TABLE 10.9-2
Microprocessor Instruction Execution

Cycle	Operation	Explanation
1	AB ← PC	pc to address bus
	PC ← PC + 1	incr pc
	IR ← DBI	data to ir
2	AB ← PC	pc to address bus
	PC ← PC + 1	incr pc
	TMP ← DB ← DBI	data to tmp
3	AB ← PC	pc to address bus
	PC ← PC + 1	incr pc
	DB ← DBI	data to dynamic store
4	ABL ← DB	data adr to address bus
	ABH ← TMP	data adr to address bus
	ACCA ← DB ← DBI	data to accumulator

the data is present on the address bus. The memory responds with the requested data, and this data is moved from DBI to DB and into accumulator A (ACCA) to complete execution of the instruction. These RTL statements describe at a high level the execution of a simple microprocessor instruction.

10.9.2 ISPS Specification and Simulation

The Instruction Set Processor Specification (ISPS) language was developed for the certification, architectural evaluation, simulation, fault analysis, and design automation of instruction set processors. The language provides a behavioral rather than a structural description. There are no part numbers, pin assignments, layouts, or technologies defined. Of course, some structural information such as register lengths, data path widths, and connections of components are necessarily a part of the simulation. The operation of each part of a processor is specified algorithmically by its behavior.

The ISPS notation includes an interface and entities. First, the carriers (memory) elements are defined. This usually includes an array of memory locations with a specified bit width and number of words. Second, the procedures necessary for the execution of the processor statements are defined. This usually includes instruction decoding, effective address calculation, arithmetic and logical operation definitions, and memory load/store functions. ISPS provides a typical set of program operators, including assignment, if, case, and repeat. Additionally, provisions are made for concurrent or sequential processing. It is possible to specify the bit length of words. Aliases are available for variables, and bit fields of variables can be addressed directly by other variables. Normal number representations include binary, hex, decimal, and octal. An example will be presented to demonstrate briefly some of the capabilities of the ISPS language.

The Motorola 68000 microprocessor will be used as the example to describe typical ISPS capabilities. Figure 10.9-1 shows the definition of the memory and processor state. The memory is defined here as 1 K 16-bit words with the name M and the alias Memory. The processor state includes definition of the program counter (PC) and extended program counter (PCA), the register array (REG), the instruction register (IR), and other required processor state holders. In each case, the number of registers and the width in bits are defined. Multiple references to some resources are specified. For example, an array of sixteen 32-bit registers (REG) is defined. Then the data registers (D) are specified as the first eight registers, and the address registers (A) are specified as the second eight registers of the register array.

Partial instruction execution for the 68000 microprocessor is defined in Fig. 10.9-2. In the figure, calculation of a displacement for the indexed address mode is demonstrated, and the effective address calculation is partially defined. Note the use of the Begin/End statements to define a block of operations and the use of ":=" as the assignment operator. A decode statement provides a multiway branch depending on the value of one or more bits. For example, in the displacement calculation, bit 11 of the memory addressed by the PC defines whether the instruction is a word index (bit 11 = 0) or a long index (bit 11 = 1).

```
M68000 :=
   BEGIN

   **Memory.State**
   M\Memory[0:1K]<15:0>,

   **Processor.State**
   PCA\Program.Counter.with.A0<23:0>,
        PC\Program.Counter<22:0>           :=PCA<23:1>,
   REG\Registers[0:15]<31:0>,
        D\Data.Registers[0:7]<31:0>        :=REG[0:7]<31:0>,
        A\Adr.Registers[0:7]<31:0>         :=REG[8:15]<31:0>,
   IR\Instruction.Register<15:0>,
        OP\OP.Code<1:0>                    :=IR<15:14>,
        SIZE\OP.Size<1:0>                  :=IR<13:12>,
        DREG\Destination.Reg<2:0>          :=IR<11:9>,
        DM\Destination.Mode<2:0>           :=IR<8:6>,
        SM\Source.Mode<2:0>                :=IR<5:3>,
        SREG\Source.Reg<2:0>               :=IR<2:0>,
   T\Temporary.Reg<31:0>,
   PCT\Temp.PC<23:0>,
   EA\Effective.Address<23:0>,
        EAE\EA.without.A0<22:0>            :=EA<23:1>,
        BYTE\HiLo.Byte< >                  :=EA<0>
```

FIGURE 10.9-1
ISPS description of M68000 microprocessor state.

The effective address calculation of Fig. 10.9-2 demonstrates use of the decode statement with a 3-bit field. This field is used to specify indirect, postincrement, predecrement, displacement, indexed, and assorted (not shown) address modes.

A complete ISPS description of a state-of-the-art microprocessor is several pages in length. Such a description is invaluable for two reasons. First, the description provides an unambiguous specification of the operation of the microprocessor (note that the description could be unambiguous and still be incorrect). Second, the description can be simulated to verify desired operation or to explore architectural characteristics of design choices early in the design cycle.

10.9.3 RTL Simulation with LISP

A less formal but very powerful means to simulate high-level behavior for a digital system is through special-purpose programs in a general-purpose programming language such as LISP or C. In fact, LISP is particularly well suited to this task because of its interactive nature and its symbolic representation capability. The behavior of each element of the digital system can be represented as a separate function. In the case of a simulation for a computer architecture, each instruction can be represented by a LISP function. These functions can be executed and changed interactively to examine or to verify operation of the instruction set. An example will be used to demonstrate this.

```
**Instruction.Execution**
DIS\Displacement<23.0> :=                    ! CALCULATES OFFSET CAUSE BY
     Begin                                    ! INDEX AND DISPLACEMENT IN
     DIS <=M[PC]<7:0>Next                     ! THE INDEX MODE.
     Decode M[PC]<11>=>
        Begin
     '0\Word.Index  := EX<=REG[M[PC]<15:12>]<15:0>,
     '1\Long.Index  := EA=REG[M[PC]<15:12>]
      EndNext
     EA=EA + DIS
     End,
CEA\Effective:Adr(MO<2:0>,R<2:0>)<>:=         ! SPECIFIES EFFECTIVE ADDRESS
     Begin                                    ! IN MEMORY FOR ANY MEMORY
     Decode MO=>                              ! RELATED OPERATION
        Begin
     '010\Indirect      :=EA=A[R]
     '011\Post.Inc      :=
          Begin
          EA=A[R]Next
          Decode SIZE =>
             Begin
          '01\Byte      :=A[R]=A[R]+1,
          '11\Word      :=A[R]=A[R]+2,
          '10\Long      :=A[R]=A[R]+4,
             End
          End,
     '100\Pre.Dec       :=
          Begin
          Decode SIZE =>
             Begin
          '01\Byte      :=A[R]=A[R]-1,
          '11\Word      :=A[R]=A[R]-2,
          '10\Long      :=A[R]=A[R]-4,
             EndNext
          EA=A[R]
          End,
      '101\Displacement:=(EA<=M[PC]NextEA=EA+A[R]Next PC=PC+1),
      '110\Index        :=(DIS()Next EA=EA+A[R] Next PC=PC+1),
      End,
End,
```

FIGURE 10.9-2

Partial ISPS description of M68000 instruction execution.

A partial LISP definition of a RISC processor is given in Fig. 10.9-3. A subset of the arithmetic, logical, and load functions is presented. Other functions, especially PC modification instructions, must be included to allow full execution of a RISC program. Each instruction is represented by a separate function with arguments that are derived from the bit fields of the instruction.

The add instruction of Fig. 10.9-3 will be examined to clarify the instruction definitions provided by the LISP functions in the figure. This instruction is a triadic instruction on this RISC processor. That is, the instruction requires three arguments: two sources and a destination. On many computers, because of instruction word width limitations, the add instruction is dyadic, requiring the destination and one source to be specified by the same bit field. The arguments

```
(defun add (rs s2 dest)
  (setq rd (+ rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun sub (rs s2 dest)
  (setq rd (- rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun and (rs s2 dest)
  (setq rd (and rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun or (rs s2 dest)
  (setq rd (or rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun xor (rs s2 dest)
  (setq rd (xor rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun sll (rs s2 dest)
  (setq rd (shiftl rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun sra (rs s2 dest)
  (setq rd (shiftra rs s2))
  (store (reg (eard dest)) rd)
  (setq pc (add1 pc)))

(defun ldl (rs s2 dest)
  (setq rd (plus rs s2))
  (store (mem (dest)))
  (setq pc (add1 pc)))
```

FIGURE 10.9-3
Partial LISP definition of RISC processor.

to the triadic add instruction presented here include *rs* as one source, *s2* as the second source, and *dest* as the destination for the add. The first line of the function defines the operation and the required arguments as "defun add (*rs s2 dest*)."

The operation of the function body for the add instruction of Fig. 10.9-3 can be explained as follows. The second line of the function definition sets a temporary variable *rd* to the sum of the contents of *rs* and *s2*. The third line invokes two functions (definitions not shown in the example) to store the results of the add in a register array. The *eard* function calculates the effective address within the register array for the store. The *eard* function must include the effects of the overlapped register storage mechanism usually employed within a RISC processor. The *store* function places the contents of the previously calculated *rd* into the proper slot within the register array. The final line of the add function increments the program counter *pc* by one.

A top-level program is required to accept a test instruction stream, decode the instruction into the appropriate bit fields, and then call the instruction primitive definitions of Fig. 10.9-3 with the arguments set appropriately. The operation of the program can be observed by including print statements at appropriate places, by tracing the execution of the program, or by examining the program's side effects on the register array, pc, other processor state holders, and memory contents.

Because of the ease with which variations in instruction definition can be tested, an interactive simulation through a LISP program is a powerful tool for system development. The interactive nature of LISP provides an excellent means to correct errors and to test new ideas quickly. There is no need to wait for compile and load steps between each change in the model. As a final comment, it should be noted that the example presented for the RISC processor did not include any effects of word length or arithmetic overflow. Additional statements are necessary to include these effects.

In this section the concepts of high-level definition and simulation of digital systems were introduced. A primitive RTL was used to define the execution of a simple microprocessor instruction. Then the ISPS language was presented as one example of an RTL language that was designed to specify and evaluate instruction processor architectures. Finally, an example was presented that used LISP as a powerful, but informal, method of simulating and evaluating digital system architectures.

10.10 HARDWARE DESIGN LANGUAGES

Machine-readable descriptions of integrated circuit designs have become an important factor in designing VLSI circuits. These descriptions are often defined in terms of design languages that, like computer languages, have specific syntax and semantics. Such design languages have been used to describe circuits from the geometrical level up through the architectural level. As new designs become increasingly dependent on CAD tools, machine-readable descriptions become extremely important. Two hardware design languages have evolved as ANSI (American National Standards Institute) standards within the last few years. One of these, EDIF (Electronic Design Interchange Format), is intended to describe designs from the layout level through the logic level. Another such language, VHDL (VHSIC Hardware Description Language), is used to characterize both the function and structure of designs from logical primitives through architectural descriptions. The basics of these two languages will be introduced here along with simple examples of each.

10.10.1 EDIF Design Description

As integrated circuit designs increased in complexity and the use of computers became prominent within the semiconductor industry, the need for a common interchange format for integrated circuit design information arose. With such a standard, silicon foundries could accept design descriptions from many sources, CAD vendors could create widely applicable programs to process designs, and designers would benefit from wider availability of CAD tools and silicon processing. The EDIF (Electronic Design Interchange Format) standard was created by interested companies to fulfill this need.

Key elements in the design of the EDIF language were broad applicability and easy extensibility. To meet these goals, EDIF was designed with a syntax that is similar to LISP with all data represented as symbolic expressions. Primitive data such as strings, signals, ports, layers, numbers, and identifiers are the *atoms* of EDIF. These atoms are formed into more complex structures as *lists*; many times, the first element of a list is a keyword that gives a particular meaning to the subsequent elements of the list. This syntax is easily parsed, and the keywords—not the syntax—provide the semantics of the language. Thus, it is desirable to design EDIF parsers that respond to the particular set of keywords for their intended function. Unrecognized keywords may be ignored successfully, allowing upward compatibility with new extensions of the language.

EDIF is intended neither as a programming language nor a database language, but rather as an efficient interchange format for integrated circuit designs. The LISP-like structure is relatively compact and yet maintains a textlike property that allows it to be read and written directly by humans. An EDIF description may contain mask descriptions, technology information, net lists, test instructions, documentation, and other user-defined information. The structure is hierarchical in that larger design descriptions can be built from component descriptions and libraries of standard elements.

The basic organizational entity for describing designs within EDIF is the *cell.* A cell may contain different representations or *views* of a design. For example, one view might contain mask layout information while another view may contain behavioral-level modeling information. A view may be one of several types such as *physical, document, behavior, topology,* or *stranger.* Each view will contain a specific type of information about the cell. For example, the physical view may contain geometric figures for circuit schematics or mask artwork, but it will not contain behavioral information. The topology view might contain net list descriptions, schematic diagrams, or symbolic layout. The document view could contain a textual description of a design, figures describing the design, or specifications for the behavior of the design. The stranger view is provided for data that does not meet the conventions of the other view types.

Each view of a cell may specify its *interface* to the external world. This interface includes a list of external ports and their characteristics. The interface description does not specify how the cell performs its function internally but rather defines how the cell will relate to its environment. A second part of the cell definition is its *contents*. The contents provide the detailed implementation for each view. This could include instances of other cells or could be the actual definition of mask geometry for the cell layout. A net list view and a mask layout view for a full adder are described here as examples of EDIF contents.

10.10.2 EDIF Net List View of Full Adder

The net list view is available in EDIF to describe collections of cells and their interconnections. Cell instances have interface sections that describe their ports. Within the EDIF net list view, the *joined* construct is used to show the interconnection of cells and interface ports. A sample EDIF file segment that describes the net list for the full adder of Fig. 10.10-1 is given in Fig. 10.10-2. This net list view starts with an interface description that declares the three input ports and two output ports of the full adder. This is followed by the contents section, which declares local signals, instantiates component cells of the full adder, and then joins appropriate signals to realize the full-adder circuit. The component cells are from a p-well CMOS library of cells. The reader should verify that the EDIF net list of Fig. 10.10-2 accurately describes the full-adder circuit of Fig. 10.10-1.

10.10.3 EDIF Mask Layout View of Full Adder

EDIF allows hierarchical descriptions of mask layout information. Public domain formats such as CIF, as well as company proprietary formats for artwork descriptions, can be described within EDIF. As an example, a partial layout of the cell-

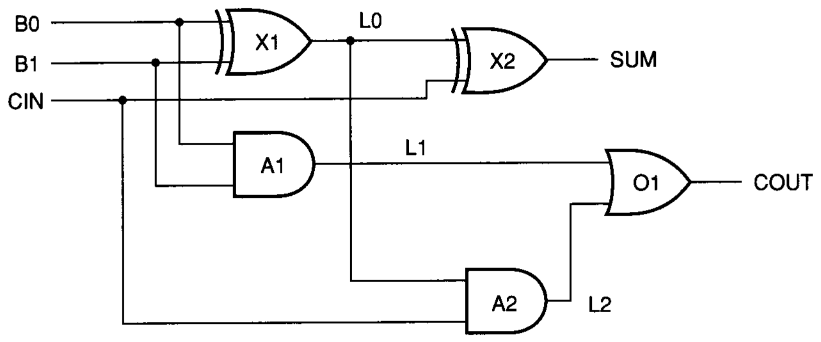


FIGURE 10.10-1
Full-adder circuit.

```
(cell FullAdder
  (view Topology Netlist
    (interface
      (declare input port (B0 B1 CIN))
      (declare output port (SUM COUT))
     )
     (contents
       (declare local signal (L0 L1 L2))
       (instance pwellcmoslib:xor X1)
       (instance pwellcmoslib:xor X2)
       (instance pwellcmoslib:and A1)
       (instance pwellcmoslib:and A2)
       (instance pwellcmoslib:or O1)
       (joined B0 X1:a A1:a)
       (joined B1 X1:b A1:b)
       (joined CIN X2:b A2:b)
       (joined L0 X1:c X2:a A2:a)
       (joined L1 A1:c O1:a)
       (joined L2 A2:c O1:b)
       (joined SUM X2:c)
       (joined COUT O1:c)
     )
  )
)
```

FIGURE 10.10-2
EDIF description of net list for full adder.

based full adder of Fig. 10.10-1 is described in CIF as shown in Fig. 10.10-3*a*. To simplify the figure, only the interconnection layout for the input signals B0, B1, and CIN is provided by the CIF description. The description presumes that the CIF layout descriptions for the five component cells of the full adder have been instantiated. Definitions for the CIF statements used in this example are provided in Fig. 10.10-3*b*. This CIF example allows a comparison with the corresponding EDIF description for the interconnection layout of the cell-based full adder of Fig. 10.10-1, as provided in Fig. 10.10-4. The EDIF description contains definitions of the cell name (CONNE), celltype (GENERIC), view (physical), viewtype (MASKLAYOUT), and the figures (rectangle) that form the interconnections among the full-adder cells. Each rectangle is described by the endpoints of one of the diagonal lines that pass through a corner of and bisect the rectangle. The EDIF keywords used here should be self-explanatory. For additional detail on EDIF layout descriptions, see the EDIF standard[3].

Since its introduction and later adoption as a standard, the EDIF language has become widely accepted within the semiconductor industry for the interchange of design information. It is now supported as an interchange mechanism by most CAD vendors. Thus, for example, results from design entry or computer-based analysis on a workstation can be moved to a different workstation or to a mainframe computer for further processing. A full description of the EDIF standard is provided by ANSI/EIA standard EDIF 2 0 0.[3]

```
DS 2;
9 CONNE;
L MET1;
B 80 80 40 700;
B 200 80 100 840;
B 1200 80 600 980;
L COND;
B 40 40 40 700;
B 40 40 160 840;
B 40 40 720 980;
B 40 40 1160 980;
L POLY;
B 120 40 140 220;
B 40 460 60 430;
B 40 740 60 1110;
B 80 80 40 700;
B 120 40 140 1460;
B 40 40 180 380;
B 40 440 140 580;
B 40 440 140 1100;
B 80 80 160 840;
B 40 40 180 1300;
B 120 40 820 220;
B 40 740 740 570;
B 80 80 720 980;
B 80 80 1160 980;
B 40 300 1140 1170;
B 40 40 1180 1300;
DF;

C 2 T 0 0;
E
```

(*a*)

```
DS 2          ; define symbol number 2
9 ABCDE       ; label (cell name)
L MET1        ; layer definition (metal)
B DX DY X Y   ; rectangle, length DX, width DY,
              ;              location X,Y
DF            ; end of symbol definition
C N T X Y     ; call symbol N, translate by X,Y
E             ; end of CIF definition
```

(*b*)

Figure 10.10-3
CIF layout example, (*a*) CIF layout file for input connections to cell-based full adder of Fig. 10.10-1, (*b*) Definition of CIF statements used in part *a*.

```
(cell CONNE
  (cellType GENERIC)
  (view physical
    (viewType MASKLAYOUT)
    (interface)
    (contents
      (figure Met1 (rectangle (pt 0 660) (pt 80 740))
                   (rectangle (pt 0 800) (pt 200 880))
                   (rectangle (pt 0 940) (pt 1200 1020)))
      (figure Cont (rectangle (pt 20 680) (pt 60 720))
                   (rectangle (pt 140 820) (pt 180 860))
                   (rectangle (pt 700 960) (pt 740 1000))
                   (rectangle (pt 1140 960) (pt 1180 1000)))
      (figure Poly (rectangle (pt 80 200) (pt 200 240))
                   (rectangle (pt 40 200) (pt 80 660))
                   (rectangle (pt 40 740) (pt 80 1480))
                   (rectangle (pt 0 660) (pt 80 740))
                   (rectangle (pt 80 1440) (pt 200 1480))
                   (rectangle (pt 160 360) (pt 200 400))
                   (rectangle (pt 120 360) (pt 160 800))
                   (rectangle (pt 120 880) (pt 160 1320))
                   (rectangle (pt 120 800) (pt 200 880))
                   (rectangle (pt 160 1280) (pt 200 1320))
                   (rectangle (pt 760 200) (pt 880 240))
                   (rectangle (pt 720 200) (pt 760 940))
                   (rectangle (pt 680 940) (pt 760 1020))
                   (rectangle (pt 1120 940) (pt 1200 1020))
                   (rectangle (pt 1120 1020) (pt 1160 1320))
                   (rectangle (pt 1160 1280) (pt 1200 1320)))
    )
  )
)
```

Figure 10.10-4
EDIF physical layout file corresponding to CIF file of Fig. 10.10-3.

10.10.4 VHDL Design Description

VHDL was developed for the design, description, and simulation of VHSIC components. VHSIC is the acronym for the Very High Speed Integrated Circuits program of the U.S. Department of Defense. Thus, the language was originally developed to describe hardware designs for military purposes. Because the need for a standard hardware description language is industrywide, the VHDL language was adopted by the IEEE and formalized as an industry standard.

VHDL is concerned primarily with description of the functional operation and/or the logical organization of designs.[24] This description is accomplished by first specifying the inputs and outputs of a system or device. Then either its *behavior* (outputs as functions of inputs) or its *structure* (in terms of interconnected subcomponents) is specified. The primary abstraction in VHDL is called a *design entity*. A design entity has two parts: the *interface description* and one or more *body descriptions*.

An interface description must perform several functions. It must define the logical interface to the outside world. It must specify the input and output ports and their characteristics. Additionally, operating conditions and characteristics may be included. To accomplish this, the interface description provides a *port declaration* for each input and output of the design entity. Each port declaration includes a port *name* and an associated *mode* and *type*. The mode specifies direction as *in, out, inout, buffer*, or *linkage*. The type qualifies the data that flows through a port. Standard types include *BIT, INTEGER, REAL, CHARACTER*, and *BIT_VECTOR*. Additionally, user-defined types are acceptable.

As a simple example with well-defined interface characteristics, the interface description for the full adder of Fig. 10.10-1 is given in Fig. 10.10-5. The full adder has three binary inputs, B0, B1, and CIN, and two binary outputs, SUM and COUT. The interface description may be thought of as the "black box" view of the design entity.

The body description of VHDL defines the internal operation or organization of the hardware, providing an "open box" view of the design entity. The internal operation is often termed a *behavioral* description, while the organization is called a *structural* description. These descriptions can occur at one of several levels, such as a logical definition, a register-transfer definition, or an algorithmic definition. The body description contains a header that provides a name for the description and identifies the associated interface description. The block...end block section contains all the descriptive information about the internal operation and organization of the hardware.

```
entity FULL_ADDER is
  port (B0,B1: in  BIT;   -- one-bit addend
        CIN:   in  BIT;   -- carry input
        SUM:   out BIT;   -- single-bit sum
        COUT:  out BIT);  -- carry output
end FULL_ADDER;
```

FIGURE 10.10-5
VHDL interface description for full adder.

```
architecture GATE_IMPLEMENTATION of FULL_ADDER is
  -- component declarations
  component AND_GATE port (X,Y: in BIT; Z: out BIT); end component;
  component XOR_GATE port (X,Y: in BIT; Z: out BIT); end component;
  component  OR_GATE port (X,Y: in BIT; Z: out BIT); end component;
  -- local signal declarations
  signal L0, L1, L2: BIT;
begin
  -- component instantiations
  X1: XOR_GATE port (B0, B1, L0);
  X2: XOR_GATE port (L0, CIN, SUM);
  A2: AND_GATE port (CIN, L0, L2);
  A1: AND_GATE port (B0, B1, L1);
  O1:  OR_GATE port (L1, L2, COUT);
end GATE_IMPLEMENTATION;
```

FIGURE 10.10-6
VHDL gate-level description for full adder.

The full-adder example of Fig. 10.10-1 will be used to demonstrate three different body descriptions. A gate-level implementation of a full adder is defined in Fig. 10.10-6. GATE_IMPLEMENTATION describes a common network of simple logic gates that realizes the full-adder function. This definition for the full adder uses AND, XOR, and OR gate components that must be defined elsewhere. The *component declarations* include an interface description for each of the logic gates. Following the component declarations, a *signal declaration* specifies signals that are used internally in the full-adder implementation. (Remember that the interface description of the full adder specifies signals that appear externally.) Finally, a *procedure block* describes the interconnection of the previously declared components that realizes the full-adder function. GATE_IMPLEMENTATION is a structural definition; that is, information is given about how to interconnect the components that compose the full adder. Without further knowledge of the behavior of components used in the definition, insufficient information is provided for simulation of the full adder.

The full adder can also be defined through a register-transfer-level description. The RTL_IMPLEMENTATION of Fig. 10.10-7 provides such a description. RTL_IMPLEMENTATION is a behavioral-level description. The structure of the implementation is left undefined; only the logical relationship

```
architecture RTL_IMPLEMENTATION of FULL_ADDER is
  signal L0, L1, L2: BIT;
begin
  L0    <= B0 xor B1;
  SUM   <= L0 xor CIN;
  L1    <= B0 and B1;
  L2    <= L0 and CIN;
  COUT  <= L1 or L2;
end RTL_IMPLEMENTATION;
```

FIGURE 10.10-7
VHDL RTL description of full adder.

of the signals is given. The description is given in terms of external signals defined in the interface description and three internal signals defined within the RTL description. The procedure block defines the relationship among the external and internal signals in terms of standard logic functions. Assuming that standard logical operations are known by the VHDL simulator, the behavior of the full adder could be simulated. Although this description does not specify structure for the full adder, an implied structure is provided in this case because there is a well-known mapping from the logic operations to standard hardware components.

As a final description of the operation of the full adder, an algorithmic declaration is given as ALG_IMPLEMENTATION, shown in Fig. 10.10-8. The ALG_IMPLEMENTATION declaration of the full adder is another behavioral description. This definition bears little relationship to the underlying physical implementation. Instead, a procedure is given to calculate the outputs of the interface description based on the inputs from the same description. This declaration is sufficient to simulate the operation of the full adder but provides little indication about its structure. This type of description is most useful for high-level definition and simulation of hardware operation. A high-level description can be provided early in the design to allow use of simulation to verify expected system behavior. Typically, an algorithmic description can have many different physical realizations.

A complex hardware system is normally described by a hierarchy of VHDL design entities. Initially, subcomponents of the design are defined by component declarations that are similar to the interface descriptions given earlier for the full adder. These components are interconnected to form more complex structures as defined within body descriptions. These complex structures may, in turn, be used as components in still more complex definitions. Ultimately, the definitions

```
architecture ALG_IMPLEMENTATION of FULL_ADDER is
begin
  process (B0, B1, CIN)
    variable S: BIT_VECTOR (1 to 3) := B0 & B1 & CIN;
    variable Num: INTEGER range 0 to 3 := 0;
  begin
    for I in 1 to 3 loop
      if S(I) = '1' then
        Num := Num + 1;
      end if;
    end loop;
    case Num is
      when 0 => SUM <= '0'; COUT <= '0';
      when 1 => SUM <= '1'; COUT <= '0';
      when 2 => SUM <= '0'; COUT <= '1';
      when 3 => SUM <= '1'; COUT <= '1';
    end case;
  end process;
end ALG_IMPLEMENTATION;
```

FIGURE 10.10-8
VHDL Algorithmic description of full adder.

of lower-level components, such as logic gates, are bound to VHDL library definitions of primitive components. Then a particular instance of the component is created along with its interconnections to other components, as defined within the VHDL block statements. Thus, a VHDL description can be created for an arbitrarily complex digital system design.

In this section, the two primary hardware design languages, EDIF and VHDL, were introduced. Both have become ANSI standards, EDIF in 1987 and VHDL in 1988. A full adder was used to provide simple examples of some of the capabilities of each standard. Both EDIF and VHDL are in the process of becoming widely accepted and supported by manufacturers and CAD vendors. EDIF functions primarily to allow simplified interchange of circuit and layout information between companies and within the same company. VHDL provides high-level definition and simulation of complex digital systems. It can serve to support analysis of design alternatives and to function as a common definition of digital system operation in the presence of multiple vendors.

10.11 ALGORITHMIC LAYOUT GENERATION

Algorithmic generation of integrated circuit layout is often perceived as a solution to the VLSI complexity problem. The basis of this well-known problem is that integrated circuit design cost is increasing for complex chips while the product life cycle is decreasing for these same chips. Design cost increases because of the design time and computer resources that must be expended to complete a state-of-the-art chip or system. Product life cycle is decreasing for these same designs because of rapid advances in technology and fierce competition to get the next-generation product to the market first.

Three approaches have been suggested to address this problem.[25] The first approach is to enhance the productivity of the human designer with faster computer workstations and improved design analysis tools. To date, this approach has been the most evident, and its description comprises the bulk of the topics in this chapter. A second approach is to capture the knowledge of a human designer with an expert system. This involves a knowledge base of concepts, rules, and strategies. These are processed by an inference engine that produces design fragments and design refinements to aid the design process. This approach is a subject of active research. A third approach is to algorithmically generate or synthesize designs from high-level descriptions or from parameterized definitions. Each variant of this approach tends to concentrate on a particular target architecture. For example, the PLA generators discussed earlier accept Boolean equations and generate layout in a well-defined form. More complex algorithmic generators are often termed *silicon compilers*. This section describes two pioneering efforts in this area and follows with a description of a state-of-the-art microprocessor chip set that was designed with heavy dependence on a commercial silicon compiler.

10.11.1 Bristle Blocks Silicon Compiler

The Bristle Blocks silicon compiler was first described in 1979.[26] The goal of the Bristle Blocks system was to produce a layout mask set from a single-page, high-level description of an integrated circuit. Many designs have their high-level structure and function frozen early in the design cycle, before the effects of such decisions are well known. If, on the other hand, a designer could use a few building blocks, organize them, and then obtain complete mask layouts and simulations early in the design cycle, then experimental configurations could be tried with a minimum of effort.

The Bristle Blocks system attempted to build designs based on a philosophy that includes structured design, hierarchical design, and multiple design representations. The structured design methodology encourages the use of regular computing structures. The design philosophy is hierarchical in that a chip is divided into sections that are subdivided to exploit hierarchical DRCs and simulations. Finally, the blocks are described via multiple design representations of increasing abstraction including layout, sticks, transistors, logic, text, simulation, and ultimately block as shown in Fig. 10.11-1. Note the general agreement between these levels of abstraction and those given in Fig. 7.1-1 of Chapter 7.

The fundamental unit in the Bristle Blocks system is the cell. Each cell can contain geometrical primitives and references to other cells. A cell can be compared to an HLL (high-level language) subroutine that contains some primitive operations and contains some references to other subroutines. A cell has the capability of containing each of the seven representations just presented. Each cell contains only local information. External connections are specified by their location and type. The location indicates where along the cell boundary the connection should occur, and the type specifies the kind of connection—for example, external output pad. The Bristle Blocks methodology gets its name from the connection points, which are like bristles along the edges of the cells. A primary directive of this method is that local information is kept local to the cell, while global information such as the location and routing to an external pad is kept separately.

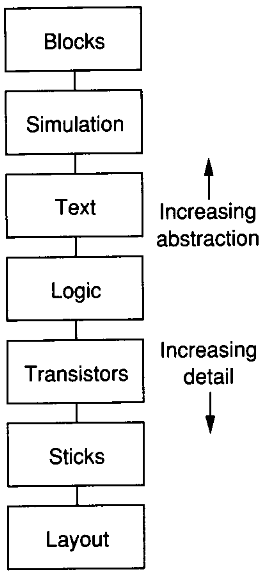


FIGURE 10.11-1
Bristle Blocks design abstraction hierarchy.

Information specifying the various representations of cells is kept in cell libraries and is accessed as needed. Each low-level cell must have been designed before it can be used in the Bristle Blocks system. Each such cell is defined by specifying the actual layout of the cell. It is felt that design of low-level cells does not take much time because of their small size. Also, the design is relatively error-free, and designer ingenuity is most beneficial at this design level.

The format of chip design using Bristle Blocks consists of physical, logical, and temporal information. The physical format is composed of a central core of operational logic and an instruction decoder, with these elements surrounded by interface pads as in Fig. 10.11-2. The instruction decoder and pads are automatically generated based on the requirements of the core section. The logical format consists of core execution units that are interconnected by two buses. In general, the order of placement of the core units is irrelevant to the operation of the system. The appropriate control functions are generated from microcode words that are provided from an external source and applied to the decoder inputs. The temporal format is a nonoverlapping two-phase clock. One clock phase controls the transfer of data between execution units via the buses. The other clock phase controls execution within the core execution units. During the execution clock phase, the buses are precharged to a high state.

The operation of the Bristle Blocks compiler requires three passes: a core pass, a control pass, and a pad pass. The first pass constructs the core execution units from user input and library cell definitions. The control pass adds the instruction decoder to generate signals required by control connection points in

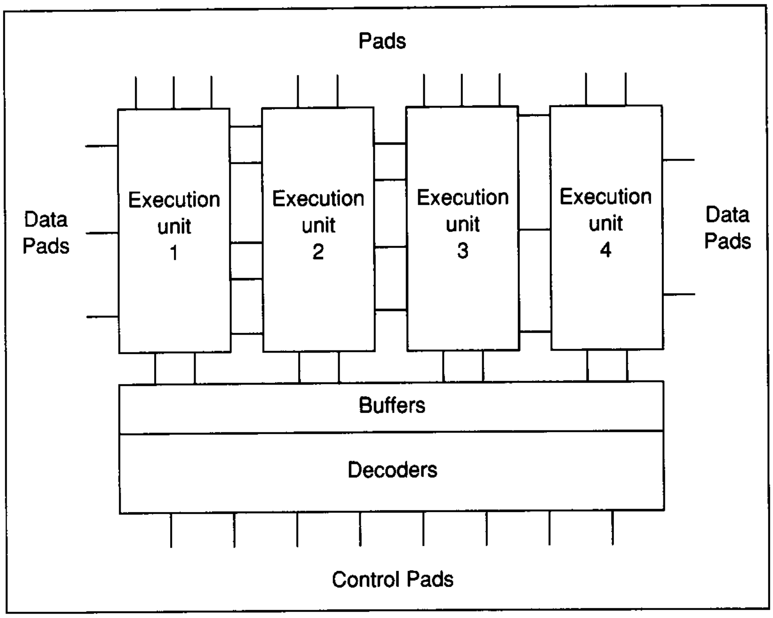


FIGURE 10.11-2
Physical format for Bristle Blocks compiler layout.

the core section. The pad pass adds pads to the perimeter of the chip and routes connections to the pads. User input to the compiler consists of three types of information. First, the microcode width and field decomposition of the control word is specified. Then the data word width and the buses that run through the core of the chip are defined. Finally, the execution units of the chip's core are defined along with any parameter values required to expand the units.

During the core layout phase, the various core cells must be interconnected. To minimize intercell routing of wires, it is advantageous for the cells to maintain a common pitch for interface connections. This requires a common width for all cells, so all cells must be designed to match the width of the widest cell. If a wider cell is added in the future, then all other cells would have to be redesigned to match the new constraint. A solution to this dilemma is to provide stretchable cells. This idea is a major contribution of this methodology. Each core cell is designed with places to stretch so that the cell width is constrained only by a minimum dimension. During the first pass, all core cells are scanned to determine the cell that constrains the minimum width. Then all other cells are stretched to match this width.

Other layout details are fixed during the first phase as well. For example, power requirements may indicate widening of the power buses. Each individual core cell is designed under interface constraints necessary to allow it to mesh with any other core cell without causing design or electrical rule violations. Finally, a bus start and stop capability along with precharge circuits are added to each bus.

The control phase generates control signal buffers to drive the control lines required by the core execution units. Then the appropriate instruction decoder is added to provide the control signals. The final stage of pad layout collects all pad connection points, sorts the points into clockwise order, and then routes connections to the pads.

The Bristle Blocks system generates data path chips based on microprogram control from an external source. Chip area for layout was reported to be within about 10% of hand layout using the same structured design methodology. Although attempts were made to generalize the structure implied by the Bristle Blocks methodology, other architectures are sufficiently different so as to require separate classes of Bristle Blocks compilers. Several commercial vendors have used the Bristle Blocks methodology as a basis for their products.

10.11.2 MacPitts Silicon Compiler

A flexible register-transfer-type language called MacPitts was described in 1982 to address the generation of microprogram-sequenced data path designs.[27] Designs described in this high-level language are compiled into a technology-independent intermediate form. The intermediate form is then compiled into a CIF geometrical layout description, which can be submitted to a silicon foundry for fabrication. The latter compilation is accomplished by limiting the possible degrees of freedom in mask layout and restricting the layout to a fixed target architecture. The target architecture consists of two distinct sections: a data path and a control unit. This architecture is shown in Fig. 10.11-3.

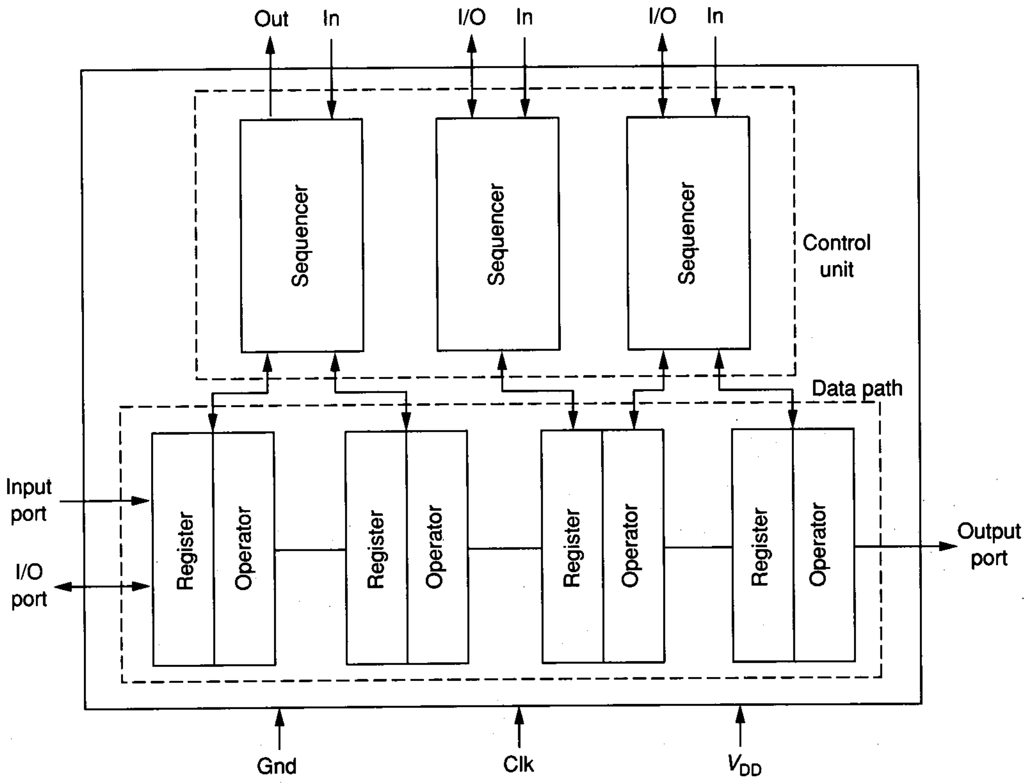


FIGURE 10.11-3
MacPitts data path/control architecture.

The data path consists of registers of width specified by the MacPitts source program. Operators for testing and modifying the data stored in the registers are also created. Data is communicated to the external world through parallel buses of wires called ports. A particular port can be an input port, a tri-state port, or an I/O port. The operations performed by the data path are specified by the control unit. In general, the control unit generates signals that cause the data path to perform certain operations. The data path returns signals that can be used to alter the control sequence. In addition, the control unit communicates to the external world through single-wire signals that may be input, output, tri-state, or I/O lines.

The data path is unconventional because it contains more than just a register array and an ALU, as is common in many microprocessors. Rather, the data path may contain many functional units interspersed among the registers. As many functional units as are needed to compute a set of parallel operations may be included between global buses. The functional units are interconnected by dedicated local buses as required by the function they perform. A given unit may take its input from several possible sources, so a multiplexer is often included to select the particular input for an operation. The output of the units is either a full word used by the data path or possibly a test result that is used directly by the control unit. A unit like an adder can generate both a word (sum) for the data path and a test signal (overflow) for the control unit. The number and type of register/operator units provided in the data path differ from system to system as specified by the MacPitts source language.

The control unit is implemented as a simplified variation of a finite-state machine. A typical FSM consists of combinatorial logic and a state register; the combinatorial logic computes the output signals and the next-state information. If the program flow is sequential, this general form of FSM is less efficient than simply using a counter to present the next state. The MacPitts compiler generates a FSM consisting of a counter and a state stack to allow subroutine calls. The logic portion of the control unit is implemented by a Weinberger array layout style consisting of interconnected NOR gates. This regular form for logic allows multilevel realizations of logic within the control unit for increased efficiency compared with a PLA-style implementation.

The MacPitts silicon compiler is an example of the use of algorithmic-level design specifications and an automation of the refinement process used to create a layout description. Standard design practice cycles between a synthesis step to create a design and an analysis step to demonstrate that the design meets prescribed objectives. Usually, the analysis step requires location and removal of flaws that are injected during the design synthesis. If the MacPitts compilation correctly generates layout corresponding to the high-level description, the design task is reduced to one of properly specifying that high-level description. An additional potential advantage of this design method accrues because of the technology independence of the intermediate-level representation generated from the high-level MacPitts source language. Because a technology-dependent synthesizer is used to create the layout from the intermediate-level representation, only this portion of the synthesis system needs to be replaced to generate the same design in a different technology.

10.11.3 Commercial Silicon Compilers

Following the early efforts described in this section, several commercial ventures were started to develop silicon compiler technology. New companies were formed to capitalize on the potential of this methodology, and existing CAD vendors developed efforts in the synthesis and silicon compilation areas. For the most part, silicon compilation has been applied only in isolated cases without great commercial success. However, a possible exception that may demonstrate the maturing of silicon compiler technology is described next.

A recently announced product, the Motorola 88000 RISC processor, was developed largely with silicon compiler technology.[28] Skilled IC designers completed the design of the 164,000-transistor CPU chip in only 20 calendar months, a productivity increase reported to be a factor of 10 to 20. A second team built the companion 750,000-transistor cache chip in only 11 months. The individual leaf cells of these products were laid out manually, but parameterized module generators speeded the design once the leaf cells were complete.

The design was started in a top-down manner with executable behavioral-level specifications. Then designers began to implement the logic and layout design of selected blocks. These low-level blocks were used to simulate the timing requirements for the chip, with architectural changes made based on the simulation results. Reusable, parameterized module generators were created for blocks such as adders, subtracters, multipliers, register files, and decoders. Module generators were also written for high-speed static RAM, tag memory, and translation buffers on a memory management chip. Through the use of parameterized modules, designers were able to make architectural revisions late in the design. Since the module generators are reusable, further versions of this chip set should be relatively easy to create. Also, because of the technology-independent description, the design should be easier to port to another process or technology.

The result reported here is an important step in the application of silicon compilers to commercial chip development. It may be noted that the apparent success in this case is a result of automating the assembly process of handcrafted leaf cells. Silicon compilation has also been extended to analog design for circuits such as CMOS op amps.[29] It will be interesting to watch the development of silicon compilers with broader applicability and with true synthesis capabilities.

10.12 SUMMARY

The use of computers has become essential to the design of VLSI circuits because of the complexity of such circuits. Computers are used to create, store, verify, modify, and interchange design information. The application areas of computer-based tools are broad and extend over the range of design hierarchies shown in Fig. 7.1-1. In fact, one expert in the area has classified computer-based tools according to their level of hierarchy and date of widespread use. This evolution commenced with the 1970s, when computers were used to aid in the design and checking of integrated circuit layout. The early 1980s saw an influx of computer-based tools for circuit and logic design, including schematic capture tools. Then the late 1980s saw the introduction of computer-based tools that work at the RTL level of design. These include synthesis tools that automatically create lower levels of the design hierarchy from previously designed cells. In the early 1990s, tools that work at the system level will likely become prominent. Synthesis and analysis tools, both based on high-level block diagrams and behavioral descriptions of a design, are examples of this capability.

As each new generation of CAD tools becomes prominent, new tool ideas and new companies are formed. Eventually, the market settles on a few concepts and firms that represent the most useful innovations with the best evolutionary ties to existing design tools. At each stage of this development, the world of VLSI design opens to a broader cadre of designers who require less knowledge of the underlying technology to make productive use of VLSI. For example, the number of logic designers is much greater than the number of integrated circuit layout specialists. In the 1980s, when computer tools based on logic descriptions became widely available, a far greater number of designers could use VLSI technology. The number of system designers and programmers who could use VLSI based on RTL or algorithmic descriptions is, in turn, much larger than the number of skilled logic designers. Thus, it has been the trend that more and more designers have access to the capabilities of VLSI technology as time progresses. Computer-based tools are the primary driving force for this trend.

The material in this chapter represents an introduction to many of the computer-based tools that are used in design automation and design verification. The section headings indicate coverage of integrated circuit layout, symbolic circuit representation, computer check plots, design rule checks, circuit extraction, digital circuit simulation, switch and logic simulation, timing analysis, RTL simulation, hardware design languages, and algorithmic layout generation. Other important areas of integrated circuit CAD that are not introduced in this chapter include process simulation, schematic capture, place and route (discussed briefly in conjunction with gate arrays in Chapter 9), mixed-mode simulation (combined analog and digital simulation—a growing number of integrated circuits contain both analog and digital sections), testability and fault analysis, and logic synthesis. Each of these areas provides its own important contributions to the design of VLSI circuits.

The intent of this chapter has been to cover many of the CAD tools and methods that blend with the material presented in the first nine chapters and to introduce some tools that are just now coming into prominence, such as hardware design languages and algorithmic layout generation. Two primary sources of information regarding new CAD tools in any of the areas mentioned above are (1) the Design Automation Conference (DAC) held each summer and sponsored by the Association of Computing Machinery (ACM) and the IEEE Computer Society, and (2) the International Conference on Computer-Aided Design (ICCAD) held each fall and sponsored by the IEEE.

REFERENCES

1. C. A. Mead and L. S. Conway: *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.
2. 1076-1987 VHDL Language Reference Manual, IEEE Catalog No. SH11957, 1987.
3. EDIF, Electronic Design Interchange Format, Version 2 0 0, Electronic Industries Association, ANSI/EIA-548-1988.
4. J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor: "Magic: A VLSI Layout System," *Proc. 21st Design Automation Conf.*, pp. 152–159, June 1984.
5. D. S. Harrison, Peter Moore, R. L. Spickelmier, and A. R. Newton: "Data Mangement and Graphics Editing in the Berkeley Design Environment," *Int. Conf. on Computer-Aided Design*, pp. 24–27, 1986.
6. P. Six, L. Claesen, J. Rabaey, and H. De Man: "An Intelligent Module Generator Environment," *Proc. 23rd Design Automation Conf.*, pp. 730–735, June, 1986.
7. J. D. Williams: "STICKS: A Graphical Compiler for High-Level LSI Design," *AFIPS Conf. Proc.*, vol. 47, pp. 289–295, June 1978.
8. R. Zinszner, Hugo De Man, K. Croes: "Technology Independent Symbolic Layout Tools," *Int. Conf. on Computer Aided Design*, pp. 12–13, September 1983.
9. A. Weinberger: "Large Scale Integration of MOS Complex Logic: A Layout Method," *IEEE J. Solid State Electron.*, vol. SC-2, no. 4, pp. 182–190, December 1967.
10. S. P. Reiss and J. E. Savage: "SLAP—A Methodology for Silicon Layout," *Proc. Int. Conf. on Circuits and Computers*, ICCC 82, pp. 281–284, September 1982.

11. Richard F. Lyon: "Simplified Design Rules for VLSI Layouts," *LAMBDA*, vol. II, no. 1, 1981.
12. C. Baker and C. Terman: "Tools for Verifying Integrated Circuit Designs," LAMBDA, vol. I, no. 4, pp. 22–30, 1980.
13. A. E. Harwood: *A VLSI Design Rule Check Program Generator*, Master's Thesis, Texas A&M University, December 1985.
14. T. Blank: "A Survey of Hardware Accelerators Used on Computer-aided Design," *IEEE Design & Test*, vol. 1, no. 3, pp. 21–39, August 1984.
15. R. A. Saleh, J. E. Kleckner, and A. R. Newton: "Iterated Timing Analysis in SPLICE1," *IEEE Int. Conf. on Computer-Aided Design*, pp. 139–140, September 1983.
16. L. M. Vidigal, S. R. Nassif, and S. W. Director: "CINNAMON: Coupled Integration and Nodal Analysis of MOS Networks," *Proc. 23rd Design Automation Conf.*, pp. 179–185, June 1986.
17. C. J. Terman: "User's Guide to NET, PRESIM, and RNL/NL," MIT Laboratory for Computer Science, pp. 1–48, September 1982.
18. Gregory F. Pfister: "The Yorktown Simulation Engine: Introduction," *Proc. 19th Design Automation Conf.*, pp. 51–73, 1982.
19. N. P. Jouppi: "TV: An nMOS Timing Analyzer," *Proc. Third Caltech Conf. on VLSI*, pp. 71–85, 1983.
20. J. K. Ousterhout: "Crystal: A Timing Analyzer for nMOS VLSI Circuits," *Proc. Third Caltech Conf. on VLSI*, pp. 57–69, 1983.
21. J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross: "Design of a High Performance VLSI Processor," *Proc. Third Caltech Conf. on VLSI*, pp. 33-54, 1983.
22. M. G. H. Katevenis: *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, Mass., 1984.
23. Mario R. Barbacci: "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications," *IEEE Trans. Comput.*, vol. c-30, no. 1, pp. 25–40, January 1981.
24. James R. Armstrong: *Chip-Level Modeling with VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
25. D. D. Gajski: "Silicon Compilers and Expert Systems for VLSI," *Proc. 21st Design Automation Conf.*, pp. 86–87, June 1984.
26. D. Johannsen: "Bristle Blocks: A Silicon Compiler," *Proc. Caltech Conf. on VLSI*, pp. 303–313, January 1979.
27. J. M. Siskind, J. R. Southard, and K. W. Crouch: "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," *Proc. MIT Conference on Advanced Research in VLSI*, pp. 28–39, January 1982.
28. R. Goering: "Silicon Compilation Boosts Productivity in 88000 Design," *Computer Design*, p. 28, May 1, 1988.
29. L. R. Carley and R. A. Rutenbar, "How to Automate Analog IC Designs," IEEE Spectrum, pp. 26–30, August 1988.

PROBLEMS

Section 10.1

10.1. Using engineering paper or the equivalent, plot the layout described by the following statements, based on the definitions of Table 10.1-1 and Table 10.1-2.

```
L 1           L 4
B 0 13 4 4    B 0 0 15 4
B 0 15 2 3    B 0 18 15 4
B 0 13 8 2    L 5
B 8 0 4 15    B 9 1 2 2
L 3           B 4 10 2 2
B 0 5 14 2    B 9 12 2 2
B 3 9 4 8     B 1 19 2 2
```

10.2. By hand, digitize the Manhattan layout shown in Fig. 10.1-4*a*. Assume that the lines are metal that ends at the figure edges and the width and spacing are 2 units each.

10.3. The layout for the block letter L is described by the following macro, based on the definitions of Table 10.1-1 and Table 10.1-2.

```
M 4
L 1
B 0 0 4 1
B 0 1 1 5
E
C 4 10 10 2
```

Show the layout resulting from the C statement above (*a*) if the rotation precedes the translation and (*b*) if the order of translation and rotation is reversed. Is the first order sufficient to create any desired layout?

Section 10.2

10.4. Show how to modify the description of Fig. 10.1-6 so that the parameter VERT can be used to modify the vertical dimension and the parameter HORZ can be used to modify the horizontal dimension.

10.5. Create a Sticks diagram for the circuit of Fig. 10.2-5.

10.6. Show the circuit diagram of a Weinberger array for an exclusive-OR gate with inputs *a* and *b* and output *c*.

10.7. Show a digraph for the logic specified by the following equations.

$$X = AB + \bar{C}D$$

$$Y = BC + X$$

$$Z = AB + \bar{A}Y + X$$

Section 10.3

10.8. Assume that a good layout density metric is 200 λ^2 per transistor. How many transistors can reasonably be displayed on a 24-line by 80-character A/N CRT display?

10.9. If a resolution of 5 dots per λ is sufficient to display the details of a layout and the layout requires 200 λ^2 per transistor, how many transistors can reasonably be displayed on a laser printer with a resolution of 300 dots per inch and a page size of 8 by 10 inches?

10.10. A Macintosh personal computer display has a resolution of 512 dots by 342 dots. Using a metric of 5 dots per λ for a readable display and 250 λ^2 per transistor, how many transistors can be displayed on the Macintosh screen?

Section 10.4

10.11. Identify all the design rule errors listed in Fig. 10.4-7 on a copy of the check plot of Fig. 10.4-6.

10.12. If a window template formed from a "plus" symbol is passed in raster scan fashion over a design to check for spacing and width violations, some errors are missed. Show an example of such an error.

10.13. Simple design rules are on the order of 1 to 3 λ for spacings and widths. What are the horizontal and vertical λ dimensions required for a "plus" symbol used in a raster scan DRC to check for Manhattan design rule violations?

Section 10.5

10.14. Based on the capacitance values in Table 10.5-1, calculate the capacitance for node 2* in Fig. 10.5-4 if layer 3 is polysilicon, layer 5 is a contact, layer 4 is aluminum, and the extracted dimensions are in microns.

10.15. Using the raster scan algorithm described in this chapter, how many different node numbers will be assigned in scanning the block letter H represented as a 5×7 dot matrix? At what point in the scan (left to right and top to bottom) will the list of nodes that must be merged be complete? (Give the x,y coordinates of the point.)

10.16. Some circuit extraction algorithms estimate connection resistance from the extracted area and perimeter values assuming rectangular shapes. Derive an algorithm based on area A and perimeter P to estimate resistance R in terms of resistance per square (sheet resistance). Estimate the resistance for an area of 10 square units and a perimeter of 22 units, assuming the terminals are on opposite sides. Is there more than one possible answer?

Section 10.6

10.17. If the time to simulate a circuit goes up as the 1.75 power of the number of nodes, and a 100-node circuit requires 30 seconds of computer time, approximately how much time would be required to simulate a circuit with 100,000 nodes?

Section 10.7

10.18. Provide a logic diagram for the circuit defined by the following net list description. The syntax is (function output input-1...input-n).

```
(invert sb s)
(nor x a s)
(nor y b sb)
(nor f x y)
```

10.19. Based on the switch-level results for the byte-wide adder presented in Sec. 10.7, estimate the maximum clock frequency for the circuit, and explain what limits this clock frequency.

10.20. Provide logic-level and transistor-level net list descriptions for the quasi-static memory cells of Fig. 10.2-4 *a* and *b*. The function (pullup a) can be used to describe a depletion pullup transistor attached to node a.

Section 10.8

10.21. For a direct realization of the following logic equations, identify all signal paths. Label the paths by using the logical names for signals. The path B,BC,X,Y is an example of one path. Assuming unit delays for the logic gates, find the longest and shortest paths.

$$X = AD + BC$$

$$Y = AC + X + BD$$

$$Z = BY + ACX$$

10.22. Assume a string of *n* ripple-carry full-adders where the carry out *cout(n-1)* of full-adder (*n*-1) is sent to the carry in *cin(n)* of full adder *n*. If a timing analyzer is used on this string of full-adders, what would you expect to find for the longest path?

10.23. For a circuit with four input ports, three output ports, and one bidirectional port, how many signal paths are possible? How does the number of paths increase as the number of ports increases?

10.24. For the circuit of Fig. 10.2-5, (if possible) set the signal directions of each transistor using the rules developed in this chapter.

10.25. For the circuit of Fig. 10.7-4, set the signal directions of all possible transistors using the rules developed in this chapter.

10.26. If the *a* input of a two-input exclusive-OR gate is rising, what can you tell about the output signal in terms of the *b* input?

Section 10.9

10.27. Describe a 4 × 4–bit shift-and-rotate multiplication using the simple RTL defined in the chapter. You may want to add shift and logical operators.

10.28. Based on the description in Fig. 10.9-1, identify and total the unique bits of processor state defined for the 68000 processor.

10.29. Using the definition of the effective address operation of Fig. 10.9-2, indicate the operations performed to compute the effective address for a word-length postincrement instruction. How does the word-length predecrement instruction differ?

10.30. Using the partial LISP definition of a RISC processor in Fig. 10.9-4 as an example, write a LISP function for the NOT operation.

Section 10.10

10.31. Based on the EDIF description given in Fig. 10.10-2 for the full-adder of Fig. 10.10-1, give an EDIF description of the NAND-NAND full-adder circuit of Fig. 10.7-2.

10.32. Using the EDIF physical layout description of Fig. 10.10-4 as an example, convert the static memory cell definition of Fig. 10.1-6*a* to an equivalent EDIF description.

10.33. Give a VHDL interface description and structural body description for the NAND-NAND full-adder circuit of Fig. 10.7-2.

10.34. Provide a VHDL interface description and RTL body description for the NAND-NAND full-adder circuit of Fig. 10.7-2.