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

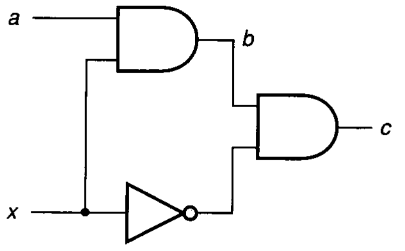

FIGURE 10.8-6
Logically impossible path.

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-

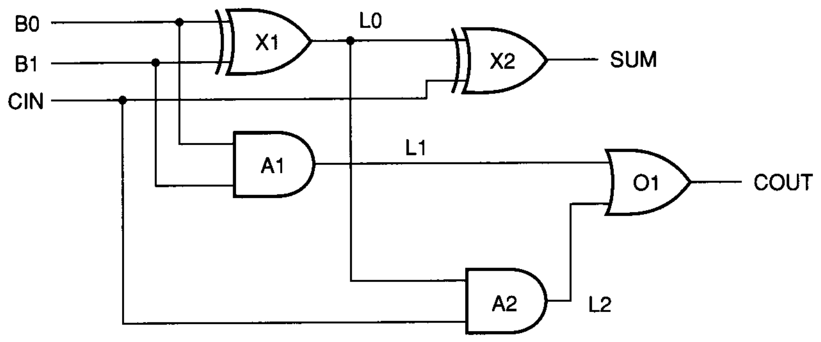

FIGURE 10.10-1
Full-adder circuit.

```
(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.

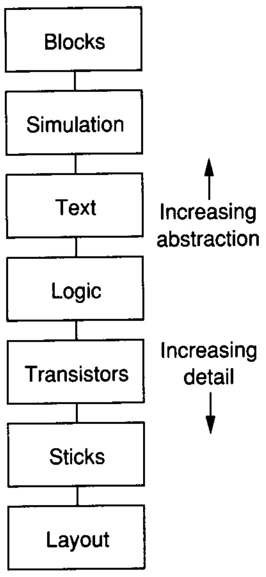

FIGURE 10.11-1
Bristle Blocks design abstraction hierarchy.

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

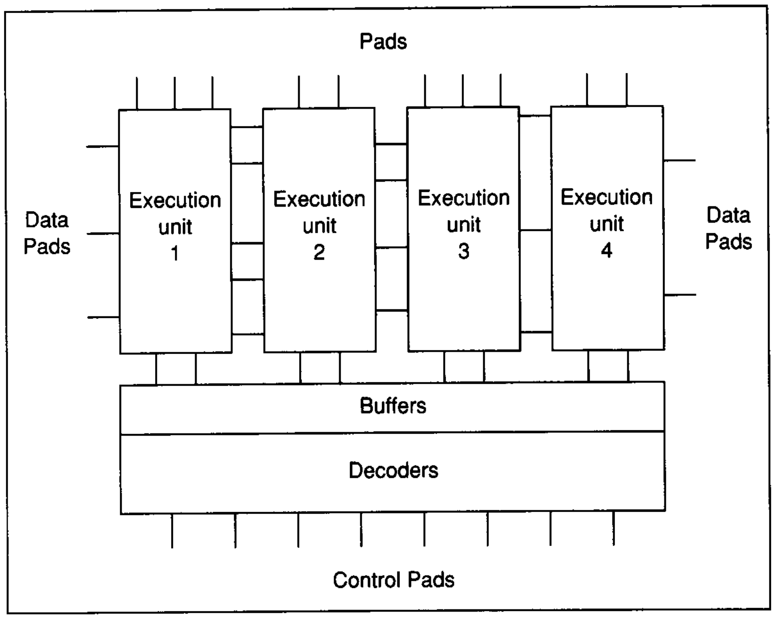

FIGURE 10.11-2
Physical format for Bristle Blocks compiler layout.

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.

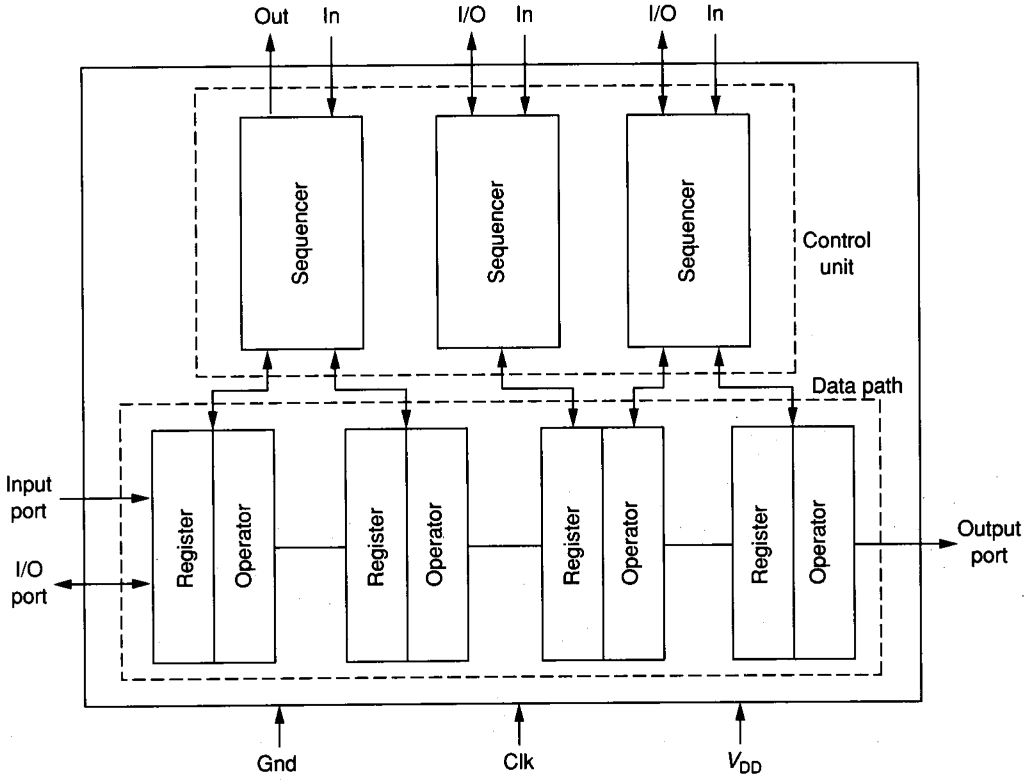

FIGURE 10.11-3
MacPitts data path/control architecture.

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.

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

$$Y = BC + X$$

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

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.