Dewey, A. “Digital and Analog Electronic Design Automation”
The Electrical Engineering Handbook
Ed. Richard C. Dorf
Boca Raton: CRC Press LLC, 2000

34

Digital and Analog Electronic Design Automation

Allen Dewey
Duke University

34.1 Introduction

The field of **design automation** (DA) technology, also commonly called *computer-aided design* (CAD) or *computer-aided engineering* (CAE), involves developing computer programs to conduct portions of product design and manufacturing on behalf of the designer. Competitive pressures to produce more efficiently new generations of products having improved function and performance are motivating the growing importance of DA. The increasing complexities of microelectronic technology, shown in Fig. 34.1, illustrate the importance of relegating portions of product development to computer automation [Barbe, 1980].

Advances in microelectronic technology enable over 1 million devices to be manufactured on an **integrated circuit** that is smaller than a postage stamp; yet the ability to exploit this capability remains a challenge. Manual design techniques are unable to keep pace with product design cycle demands and are being replaced by automated design techniques [Saprio, 1986; Dillinger, 1988].

Figure 34.2 summarizes the historical development of DA technology. DA computer programs are often simply called *applications* or *tools*. DA efforts started in the early 1960s as academic research projects and captive industrial programs; these efforts focused on tools for physical and logical design. Follow-on developments extended logic simulation to more-detailed *circuit* and *device* simulation and more-abstract *functional* simulation. Starting in the mid to late 1970s, new areas of test and synthesis emerged and vendors started offering commercial DA products. Today, the electronic design automation (EDA) industry is an international business with a well-established and expanding technical base [Trimberger, 1990]. EDA will be examined by presenting an overview of the following areas:

- Design entry,
- Synthesis,
- Verification,
- Physical design, and
- Test.

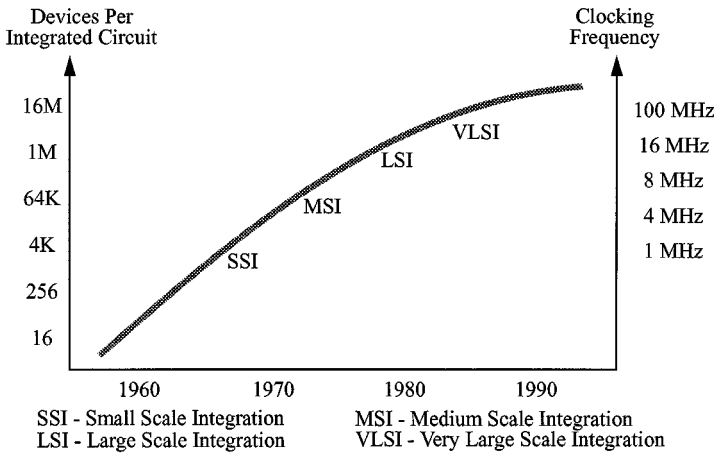

FIGURE 34.1 Microelectronic technology complexity.

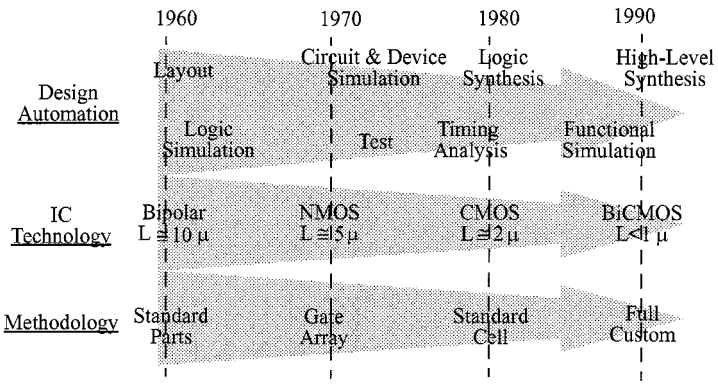

FIGURE 34.2 DA technology development.

34.2 Design Entry

Design entry, also called *design capture*, is the process of communicating with a DA system. In short, design entry is how an engineer "talks" to a DA application and/or system.

Any sort of communication is composed of two elements: language and mechanism. Language provides common semantics; mechanism provides a means by which to convey the common semantics. For example, people communicate via a language, such as English or German, and a mechanism, such as a telephone or electronic mail. For design, a digital system can be described in many ways, involving different perspectives or *abstractions*. An abstraction defines at a particular level of detail the behavior or semantics of a digital system, i.e., how the outputs respond to the inputs. Fig. 34.3 illustrates several popular levels of abstractions. Moving from the lower left to the upper right, the level of abstraction generally increases, meaning that physical models are the most detailed and specification models are the least detailed. The trend toward higher levels of design entry abstraction supports the need to address greater levels of complexity [Peterson, 1981].

The physical level of abstraction involves geometric information that defines electrical devices and their interconnection. Geometric information includes the shape of objects and how objects are placed relative to each other. For example, Fig. 34.4 shows the geometric shapes defining a simple complementary metal-oxide semiconductor (**CMOS**) inverter. The shapes denote different materials, such as aluminum and polysilicon, and connections, called *contacts* or **vias**.

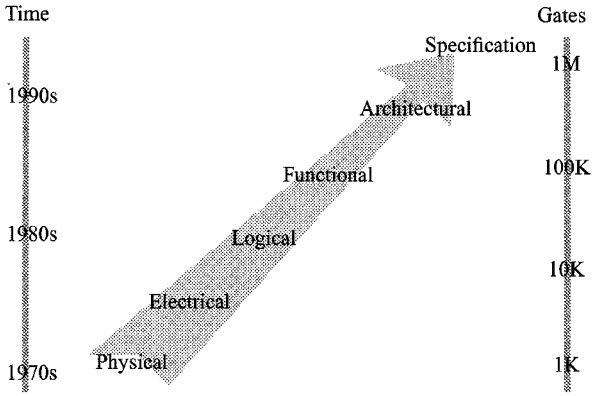

FIGURE 34.3 DA abstractions.

Design entry mechanisms for physical information involve textual and graphical techniques. With textual techniques, geometric shape and placement are described via an artwork description language, such as Caltech Intermediate Form (CIF) or Electronic Design Intermediate Form (EDIF). With graphical techniques, geometric shape and placement are described by rendering the objects on a display terminal.

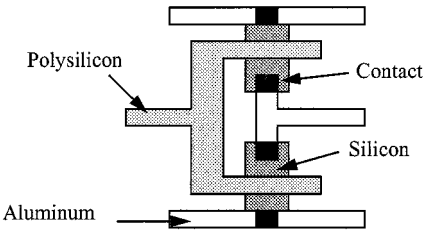

FIGURE 34.4 Physical abstraction.

The electrical level abstracts physical information into corresponding electrical devices, such as **capacitors**, **transistors**, and **resistors**. Electrical information includes device behavior in terms of terminal current and voltage relationships. Device behavior may also be defined in terms of manufacturing parameters. Fig. 34.5 shows the electrical symbols denoting a CMOS inverter.

The logical level abstracts electrical information into corresponding logical elements, such as **and** gates, **or** gates, and inverters. Logical information includes truth table and/or characteristic-switching algebra equations and active-level designations. Fig. 34.6 shows the logical symbol for a CMOS inverter. Notice how the amount of information decreases as the level of abstraction increases.

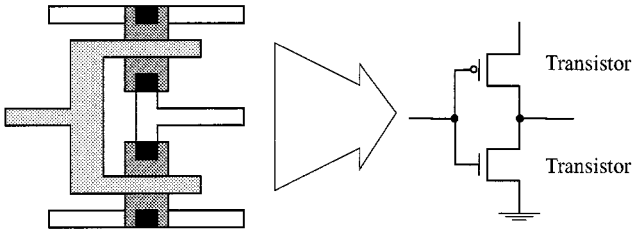

FIGURE 34.5 Electrical abstraction.

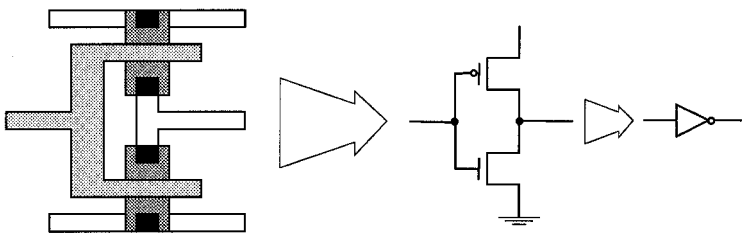

FIGURE 34.6 Logical abstraction.

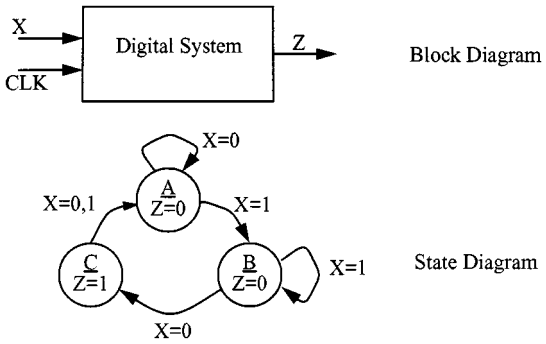

FIGURE 34.7 State diagram.

Design entry mechanisms for electrical and logical abstractions are collectively called *schematic capture* techniques. Schematic capture defines hierarchical structures, commonly called **netlists**, of components. A designer creates instances of components supplied from a library of predefined components and connects component pins or ports via wires [Douglas-Young, 1988; Pechet, 1991].

The functional level abstracts logical elements into corresponding computational units, such as registers, multiplexers, and arithmetic logic units (ALUs). The architectural level abstracts functional information into computational algorithms or paradigms. Examples of common computational paradigms are listed below:

- State diagrams,
- Petri nets,
- Control/data flow graphs,
- Function tables,
- Spreadsheets, and
- Binary decision diagrams.

These higher levels of abstraction support a more expressive, "higher-bandwidth" communication interface between engineers and DA programs. Engineers can focus their creative, cognitive skills on concept and behavior, rather than on the complexities of detailed implementation. Associated design entry mechanisms typically use hardware description languages with a combination of textual and graphic techniques [Birtwistle and Subrahmanyan, 1988].

Figure 34.7 shows an example of a simple state diagram. The state diagram defines three states, denoted by circles. State-to-state transitions are denoted by labeled arcs; state transitions depend on the present state and the input X. The output, Z, per state is given within each state. Since the output is dependent on only the present state, the digital system is classified as a Moore **finite state machine**. If the output is dependent on the present state and input, then the digital system is classified as a Mealy finite state machine.

A hardware description language model written in **VHDL** of the Moore finite state machine is given in Fig. 34.8. The VHDL model, called a *design entity*, uses a "**data flow**" description style to describe the state machine [Dewey, 1983, 1992, 1997]. The entity statement defines the interface, i.e., the ports. The ports include two input signals, X and CLK, and an output signal Z. The ports are of type BIT, which specifies that the signals may only carry the values 0 or 1. The architecture statement defines the input/output transform via two concurrent signal assignment statements. The internal signal STATE holds the finite state information and is driven by a guarded, conditional concurrent signal assignment statement that executes when the associated block expression

(CLK='1' **and not** CLK'STABLE)

is true, which is only on the rising edge of the signal CLK. STABLE is a predefined attribute of the signal CLK; CLK'STABLE is true if CLK has *not* changed value. Thus, if "**not** CLK'STABLE" is true, meaning that CLK has

```
-- entity statement
entity MOORE_MACHINE is
 port (X, CLK : in BIT;   Z : out BIT);
end MOORE_MACHINE;

-- architecture statement
architecture FSM of MOORE_MACHINE is
  type STATE_TYPE is (A, B, C);
  signal STATE : STATE_TYPE := A;
begin
 NEXT_STATE:
 block (CLK='1' and not CLK'STABLE)
 begin
    -- guarded conditional concurrent signal assignment statement
    STATE <= guarded B when (STATE=A and X='1') else
                          C when (STATE=B and X='0') else
                          A when (STATE=C) else
                          STATE;
 end block NEXT_STATE;
 -- unguarded selected concurrent signal assignment statement
 with STATE select
   Z <= '0' when A,
        '0' when B,
        '1' when C;
end FSM;
```

FIGURE 34.8 VHDL model.

just changed value, and "CLK='1'," then a rising transition has occurred on CLK. The output signal Z is driven by a nonguarded, selected concurrent signal assignment statement that executes any time STATE changes value.

34.3 Synthesis

Figure 34.9 shows that the **synthesis** task generally follows the design entry task. After describing the desired system via design entry, synthesis DA programs are invoked to assist generating the required detailed design.

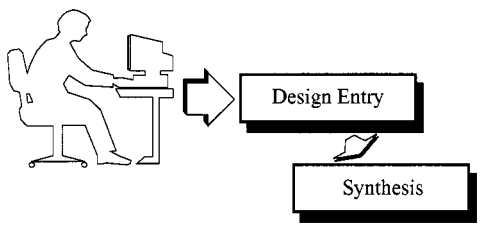

FIGURE 34.9 Design process — synthesis.

Synthesis translates or transforms a design from one level of abstraction to another, more-detailed level of abstraction. The more-detailed level of abstraction may be only an intermediate step in the entire design process, or it may be the final implementation. Synthesis programs that yield a final implementation are sometimes called **silicon compilers** because the programs generate sufficient detail to proceed directly to silicon fabrication [Ayres, 1983; Gajski, 1988].

Like design abstractions, synthesis techniques can be hierarchically categorized, as shown in Fig. 34.10. The higher levels of synthesis offer the advantage of less complexity, but also the disadvantage of less control over the final design.

Algorithmic synthesis, also called *behavioral* synthesis, addresses "multicycle" behavior, which means behavior that spans more than one *control step*. A control step equates to a clock cycle of a synchronous, sequential digital system, i.e., a state in a finite-state machine controller or a microprogram step in a microprogrammed controller.

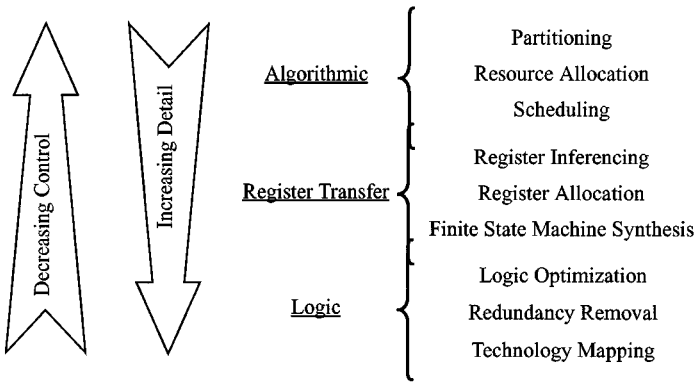

FIGURE 34.10 Taxonomy of synthesis techniques.

Algorithmic synthesis typically accepts sequential design descriptions that define an input/output transform, but provide little information about the parallelism of the final design [Camposano and Wolfe, 1991; Gajski et al., 1992].

Partitioning decomposes the design description into smaller behaviors. Partitioning is an example of a high-level transformation. High-level transformations include common software programming compiler optimizations, such as loop unrolling, subprogram in-line expansion, constant propagation, and common subexpression elimination.

Resource allocation associates behaviors with hardware computational units, and scheduling determines the order in which behaviors execute. Behaviors that are mutually exclusive can potentially share computational resources. Allocation is performed using a variety of graph clique covering or node coloring algorithms. Allocation and scheduling are interdependent, and different synthesis strategies perform allocation and scheduling different ways. Sometimes scheduling is performed first, followed by allocation; sometimes allocation is performed first, followed by scheduling; and sometimes allocation and scheduling are interleaved.

Scheduling assigns computational units to control steps, thereby determining which behaviors execute in which clock cycles. At one extreme, all computational units can be assigned to a single control step, exploiting maximum concurrency. At the other extreme, computational units can be assigned to individual control steps, exploiting maximum sequentiality. Several popular scheduling algorithms are listed below:

- As-soon-as-possible (ASAP),
- As-late-as-possible (ALAP),
- List scheduling,
- Force-directed scheduling, and
- Control step splitting/merging.

ASAP and ALAP scheduling algorithms order computational units based on data dependencies. List scheduling is based on ASAP and ALAP scheduling, but considers additional, more-global constraints, such as maximum number of control steps. Force-directed scheduling computes the probabilities of computational units being assigned to control steps and attempts to evenly distribute computation activity among all control steps. Control step splitting starts with all computational units assigned to one control step and generates a schedule by splitting the computational units into multiple control steps. Control step merging starts with all computational units assigned to individual control steps and generates a schedule by merging or combining units and steps [Paulin and Knight, 1989; Camposano and Wolfe, 1991].

Register transfer synthesis takes as input the results of algorithmic synthesis and addresses "per-cycle" behavior, which means the behavior during one clock cycle. Register transfer synthesis selects logic to realize the hardware computational units generated during algorithmic synthesis, such as realizing an addition operation with a carry–save adder or realizing addition and subtraction operations with an arithmetic logic unit.

Data that must be retained across multiple clock cycles are identified, and registers are allocated to hold the data. Finally, finite-state machine synthesis involves state minimization and state assignment. State minimization seeks to eliminate redundant or equivalent states, and state assignment assigns binary encodings for states to minimize combinational logic [Brayton et al., 1992; Sasao, 1993].

Logic synthesis optimizes the logic generated by register transfer synthesis and maps the optimized logic operations onto physical gates supported by the target fabrication technology. Technology mapping considers the foundry cell library and associated electrical restrictions, such as **fan-in/fan-out** limitations.

34.4 Verification

Figure 34.11 shows that the **verification** task generally follows the synthesis task. The verification task checks the correctness of the function and performance of a design to ensure that an intermediate or final design faithfully realizes the initial, desired specification. Three major types of verification are listed below:

- Timing analysis,
- Simulation, and
- Emulation.

Timing Analysis

Timing analysis checks that the overall design satisfies operating speed requirements and that individual signals within a design satisfy transition requirements. Common signal transition requirements, also called *timing hazards*, include *rise* and *fall times*, *propagation delays*, *clock periods*, *race conditions*, *glitch detection*, and *setup* and *hold times*. For instance, setup and hold times specify relationships between data and control signals to ensure that memory devices (level-sensitive latches or edge-sensitive flip-flops) correctly and reliably store desired data. The data signal carrying the information to be stored in the memory device must be stable for a period equal to the setup time prior to the control signal transition to ensure that the correct value is sensed by the memory device. Also, the data signal must be stable for a period equal to the hold time after the control signal transition to ensure that the memory device has enough time to store the sensed value.

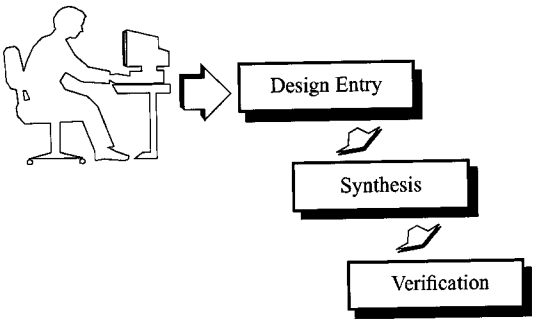

FIGURE 34.11 Design process — verification.

Another class of timing transition requirements, commonly called signal integrity checks, include *reflections*, *crosstalk*, **ground bounce**, and *electromagnetic interference*. Signal integrity checks are typically required for high-speed designs operating at clock frequencies above 75 MHz. At such high frequencies, the transmission line behavior of wires must be analyzed. A wire should be properly terminated, i.e., connected, to a port having an impedance matching the wire characteristic impedance to prevent signal reflections. Signal reflections are portions of an emanating signal that "bounce back" from the destination to the source. Signal reflections reduce the power of the emanating signal and can damage the source. Crosstalk refers to unwanted reactive coupling between physically adjacent signals, providing a connection between signals that are supposed to be electrically isolated. Ground bounce is another signal integrity problem. Since all conductive material has a finite impedance, a ground signal network does not in practice offer the exact same electrical potential throughout an entire design. These potential differences are usually negligible because the distributive impedance of the ground signal network is small compared with other finite-component impedances. However, when many signals switch value simultaneously, a substantial current can flow through the ground signal network. High intermittent currents yield proportionately high intermittent potential drops, i.e., ground bounces, which can

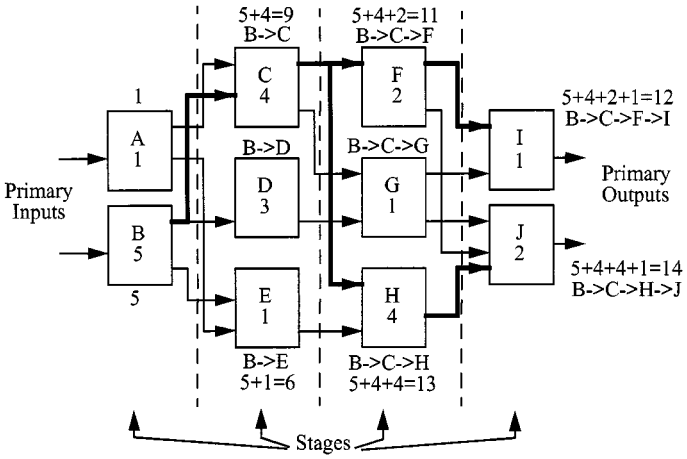

FIGURE 34.12 Block-oriented static timing analysis.

cause unwanted circuit behavior. Finally, electromagnetic interference refers to signal harmonics radiating from design components and interconnects. This harmonic radiation may interfere with other electronic equipment or may exceed applicable environmental safety regulatory limits [McHaney, 1991].

Timing analysis can be performed dynamically or statically. Dynamic timing analysis exercises the design via simulation or emulation for a period of time with a set of input stimuli and records the timing behavior. Static timing analysis does not exercise the design via simulation or emulation. Rather, static analysis records timing behavior based on the timing behavior, e.g., propagation delay, of the design components and their interconnection.

Static timing analysis techniques are primarily *block oriented* or *path oriented*. Block-oriented timing analysis generates design input (also called primary input) to design output (also called primary output), and propagation delays by analyzing the design "stage-by-stage" and by summing up the individual stage delays. All devices driven by primary inputs constitute stage 1, all devices driven by the outputs of stage 1 constitute stage 2, and so on. Starting with the first stage, all devices associated with a stage are annotated with worst-case delays. A worst-case delay is the propagation delay of the device plus the delay of the last input to arrive at the device, i.e., the signal path with the longest delay leading up to the device inputs. For example, the device labeled "H" in stage 3 in Fig. 34.12 is annotated with the worst-case delay of 13, representing the device propagation delay of 4 and the delay of the last input to arrive through devices "B" and "C" of 9 [McWilliams and Widdoes, 1978]. When the devices associated with the last stage, i.e., the devices driving the primary outputs, are processed, the accumulated worst-case delays record the longest delay from primary inputs to primary outputs, also call the critical paths. The critical path for each primary output is highlighted in Fig. 34.12.

Path-oriented timing analysis generates primary input to primary output propagation delays by traversing all possible signal paths one at a time. Thus, finding the critical path via path-oriented timing analysis is equivalent to finding the longest path through a directed acyclic graph, where devices are graph vertices and interconnections are graph edges [Sasiki et al., 1978].

To account for realistic variances in component timing due to manufacturing tolerances, aging, or environmental effects, timing analysis often provides stochastic or statistical checking capabilities. Statistical timing analysis uses random-number generators based on empirically observed probabilistic distributions to determine component timing behavior. Thus, statistical timing analysis describes design performance and the likelihood of the design performance.

Simulation

Simulation exercises a design over a period of time by applying a series of input stimuli and generating the associated output responses. The general event-driven, also called schedule-driven, simulation algorithm is diagrammed in Fig. 34.13. An event is a change in signal value. Simulation starts by initializing the design;

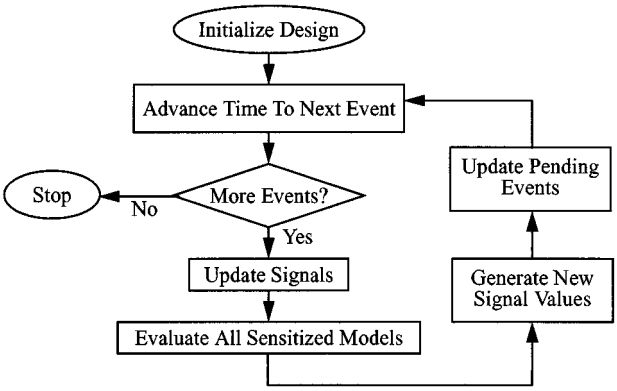

FIGURE 34.13 General event-driven simulation algorithm.

initial values are assigned to all signals. Initial values include starting values and pending values that constitute future events. Simulation time is advanced to the next pending event(s), signals are updated, and sensitized models are evaluated [Pooch, 1993]. The process of evaluating the sensitized models yields new, potentially different, values for signals, i.e., a new set of pending events. These new events are added to the list of pending events, time is advanced to the next pending event(s), and the simulation algorithm repeats. Each pass through the loop in Fig. 34.13 of evaluating sensitized models at a particular time step is called a **simulation cycle**. Simulation ends when the design yields no further activity, i.e., when there are no more pending events to process.

Logic simulation is computationally intensive for large, complex designs. As an example, consider simulating 1 s of a 200K-gate, 20-MHz processor design. By assuming that, on average, only 10% of the total 200K gates are active or sensitized on each processor clock cycle, Eq. 34.1 shows that simulating 1 s of actual processor time equates to 400 billion events.

$$\begin{aligned} 400 \text{ billion events} &= (20 \text{ million clock cycles})(200\text{K gates})(10\% \text{ activity}) \\ 140 \text{ h} &= (400 \text{ billion events})\left(\frac{50 \text{ instructions}}{\text{event}}\right)\left(\frac{50 \text{ million instructions}}{\text{s}}\right) \end{aligned} \tag{34.1}$$

Assuming that, on average, a simulation program executes 50 computer instructions per event on a computer capable of processing 50 million instructions per second (MIP), Eq. 34.1 also shows that processing 400 billion events requires 140 h or just short of 6 days. Fig. 34.14 shows how simulation computation generally scales with design complexity.

To address the growing computational demands of simulation, several simulation acceleration techniques have been introduced. Schedule-driven simulation, explained above, can be accelerated by removing layers of interpretation and running a simulation as a native executable image; such an approach is called complied, scheduled-driven simulation.

As an alternative to schedule-driven simulation, *cycle-driven* simulation avoids the overhead of event queue processing by evaluating all devices at regular intervals of time. Cycle-driven simulation is efficient when a design exhibits a high degree of concurrency, i.e., when a large percentage of the devices are active per simulation cycle. Based on the staging of devices, devices are *rank-ordered* to determine the order in which they are evaluated at each time step to ensure the correct causal behavior yielding the proper ordering of events. For functional verification, logic devices are often assigned zero-delay and memory devices are assigned unit-delay. Thus, any number of stages of logic devices may execute between system clock periods.

In another simulation acceleration technique, *message-driven* simulation, also called *parallel* or *distributed* simulation, device execution is divided among several processors and the device simulations communicate

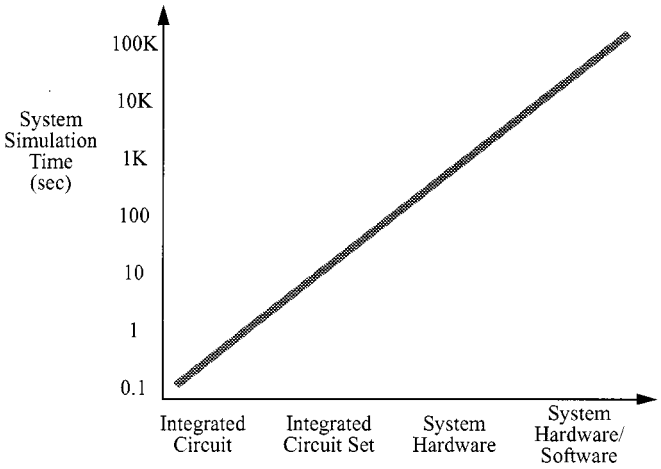

FIGURE 34.14 Simulation requirements.

event activity via messages. Messages are communicated using conservative or optimistic strategies. Optimistic message-passing strategies, such as *time warp* and *lazy cancellation,* make assumptions about future event activity to advance local device simulation. If the assumptions are correct, the processors operate more independently and better exploit parallel computation. However, if the assumptions are incorrect, then local device simulations may be forced to "roll back" to synchronize local device simulations [Bryant, 1979; Chandy and Misra, 1981].

Schedule-driven, cycle-driven, and message-driven simulations are software-based simulation acceleration techniques. Simulation can also be accelerated by relegating certain simulation activities to dedicated hardware. For example, *hardware modelers* can be attached to software simulators to accelerate the activity of device evaluation. As the name implies, hardware modeling uses actual hardware devices instead of software models to obtain stimulus/response information. Using actual hardware devices reduces the expense of generating and maintaining software models and provides an environment to support application software development. However, it is sometimes difficult for a slave hardware modeler to preserve accurate real-time device operating response characteristics within a master non-real-time software simulation environment. For example, some hardware devices may not be able to retain state information between invocations, so the hardware modeler must save the history of previous inputs and reapply them to bring the hardware device to the correct state to apply a new input.

Another technique for addressing the growing computational demands of simulation is via simulation engines. A simulation engine can be viewed as an extension of the simulation acceleration technique of hardware modeling. With a hardware modeler, the simulation algorithm executes in software and component evaluation executes in dedicated hardware. With a simulation engine, the simulation algorithm *and* component evaluation execute in dedicated hardware. Simulation engines are typically two to three orders of magnitude faster than software simulation [Takasaki et al., 1989].

Analog Simulation

Analog simulation involves time-domain analyses and frequency-domain analyses, which are generally conducted using some form of direct current (DC) simulation, diagrammed in Fig. 34.15. DC simulation determines the quiescent or steady-state operating point for a circuit, specifying **node voltages**, **branch** currents, input/output resistances, element sensitivities, and input/output gains [Chua and Lin, 1975; Nagel, 1975].

Several popular equation formulation schemes are summarized in Table 34.1. Equation formulation schemes generate a set of linear equations denoting relationships between circuit voltages and currents; these relationships are based on the physical principle of the conservation of energy expressed via **Kirchoff's current law** (KCL), **Kirchoff's voltage law** (KVL), and branch constitutive relationships (BCRs). A circuit having N nodes and B branches possesses $2B$ independent variables defining B branch voltages and B branch currents. These

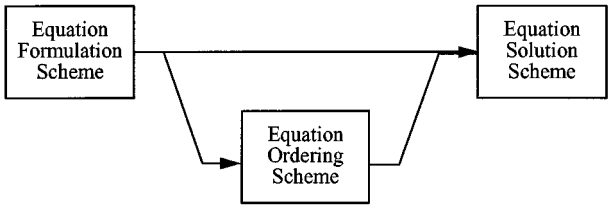

FIGURE 34.15 DC simulation.

TABLE 34.1 Common Circuit Equation Formulation Schemes

Equation Formulation Schemes	Desired Unknowns
Nodal analysis	Node voltages
Modified nodal analysis	Node voltages
	Dependent source currents
	Independent voltage source currents
Sparse tableau analysis	Node voltages
	Branch currents
	Branch voltages
Reduced tableau analysis	Node voltages
	Branch currents
Tree analysis	Tree branch voltages
Link analysis	Link branch currents

variables are governed by $2B$ linearly independent equations composed of $N-1$ KCL equations, $B-N+1$ KVL equations, and B BCR equations [Hachtel et al., 1971; Ho et al., 1975].

Equation-ordering schemes augment equation formulation schemes by reorganizing, modifying, and scaling the equations to improve the efficiency and/or accuracy of the subsequent equation solution scheme. More specifically, equation-ordering schemes seek to improve the "diagonal dominance" structure of the coefficient matrix by maximizing the number of "off-diagonal" zeros. Popular equation-ordering schemes include pivoting and row ordering (Markowitz) [Zlatev, 1980].

Finally, equation solution schemes determine the values for the independent variables that comply with the governing equations. There are basically two types of equation solution schemes: explicit and implicit. Explicit solution schemes, such as Gaussian elimination and/or LU factorization, determine independent variable values using closed-form, deterministic techniques. Implicit solution schemes, such as Gauss–Jacobi and Gauss–Seidel, determine independent variable values using iterative, nondeterministic techniques.

Emulation

Emulation, also called *computer-aided prototyping*, verifies a design by realizing the design in "preproduction" hardware and exercising the hardware. The term *preproduction* hardware means nonoptimized hardware providing the correct functional behavior, but not necessarily the correct performance. That is, emulation hardware may be slower, require more area, or dissipate more power than production hardware. At present, preproduction hardware commonly involves some form of **programmable logic devices** (PLDs), typically field-programmable **gate arrays** (FPGAs). PLDs provide generic combinational and sequential digital system logic that can be programmed to realize a wide variety of designs [Walters, 1991].

Emulation offers the advantage of providing prototype hardware early in the design cycle to check for errors or inconsistencies in initial functional specifications. Problems can be isolated and design modifications can be easily accommodated by reprogramming the logic devices. Emulation can support functional verification at computational rates much greater than conventional simulation. However, emulation does not generally support performance verification because, as explained above, prototype hardware typically does not operate at production clock rates.

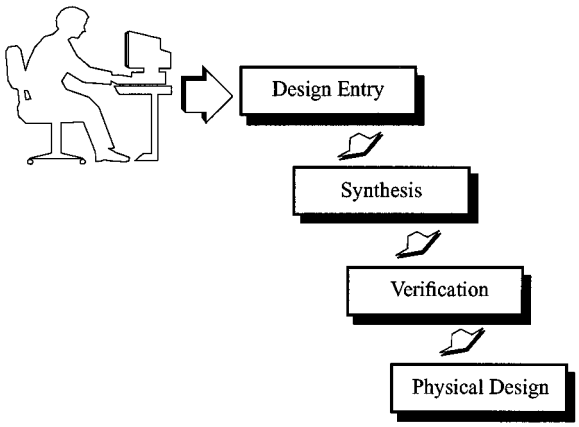

FIGURE 34.16 Design process — physical design.

34.5 Physical Design

Figure 34.16 shows that the physical design task generally follows the verification task. Having validated the function and performance of the detailed design during verification, physical design realizes the detailed design by translating logic into actual hardware. Physical design involves placement, routing, artwork generation, rules checking, and back annotation [Sait and Youseff, 1995].

Placement transforms a logical hierarchy into a physical hierarchy by defining how hardware elements are oriented and arranged relative to each other. Placement determines the overall size, i.e., area, a digital system will occupy. Two popular placement algorithms are *mincut* and *simulated annealing*. Mincut placement techniques group highly connected cells into clusters. Then, the clusters are sorted and arranged according to user-supplied priorities. Simulated annealing conducts a series of trial-and-error experiments by pseudorandomly moving cells and evaluating the resulting placements, again according to user-supplied priorities.

Routing defines the wires that establish the required port-to-port connections. Routing is often performed in two stages: global and local. Global routing assigns networks to major wiring regions, called tracks; local routing defines the actual wiring for each network within its assigned track. Two common classes of routing algorithms are *channel* and *maze*. Channel routing connects ports abutting the same track. Maze routing, also called switch-box routing, connects ports abutting different channels. Routing considers a variety of metrics, including timing skew, wire length, number of vias, and number of jogs (corners) [Spinks, 1985; Preas et al., 1988].

Rules checking verifies that the final layout of geometric shapes and their orientation complies with logical, electrical, and physical constraints. Logical rules verify that the implementation realizes the desired digital system. Electrical rules verify conformance to loading, noise margins, and fan-in/fan-out connectivity constraints. Finally, physical rules verify conformance to dimensional, spacing, and alignment constraints [Hollis, 1987].

34.6 Test

Figure 34.17 shows that **test** follows physical design. After physical design, the digital system is manufactured and test checks the resulting hardware for correct function and performance. Thus, the primary objective of test is to detect a faulty device by applying input test stimuli and observing expected results [Buckroyd, 1989; Weyerer and Goldemund, 1992].

The test task is difficult because designs are growing in complexity; more components provide more opportunity for manufacturing defects. Test is also challenged by new microelectronic fabrication processes. These new processes support higher levels of integration that provide fewer access points to probe internal electrical nodes and new failure modes that provide more opportunity for manufacturing defects.

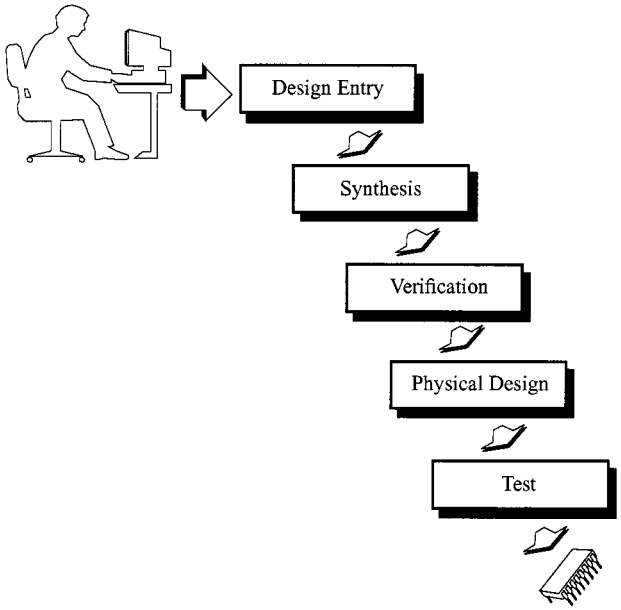

FIGURE 34.17 Design process — test.

Fault Modeling

What is a fault? A fault is a manufacturing or aging defect that causes a device to operate incorrectly or to fail. A sample listing of common integrated circuit physical faults are given below:

- Wiring faults,
- Dielectric faults,
- Threshold faults, and
- Soft faults.

Wiring faults are unwanted opens and shorts. Two wires or networks that should be electrically connected, but are not connected constitute an open. Two wires or networks that should not be electrically connected, but are connected constitute a short. Wiring faults can be caused by manufacturing defects, such as metallization and etching problems, or aging defects, such as corrosion and **electromigration**. Dielectric faults are electrical isolation defects that can be caused by **masking defects**, chemical impurities, material imperfections, or electrostatic discharge. Threshold faults occur when the turn-on and turn-off voltage potentials of electrical devices exceed allowed ranges. Soft faults occur when radiation exposure temporarily changes electrical charge distributions. Such changes can alter circuit voltage potentials, which can, in turn, change logical values, also called *dropping bits*. Radiation effects are called "soft" faults because the hardware is not permanently damaged [Zobrist, 1993].

To simplify the task of fault testing, the physical faults described above are translated into logical faults. Typically, a single logical fault covers several physical faults. A popular logical fault model is the *single stuck line* (SSL) fault model. The single stuck line fault model supports faults that denote wires permanently set to a logic 0, "stuck-at-0," or a logic 1, "stuck-at-1." Building on the single stuck line fault model, the *multiple stuck line* (MSL) fault model supports faults where multiple wires are stuck-at-0/stuck-at-1. Stuck fault models do not address all physical faults because not all physical faults result in signal lines permanently set to low or high voltages, i.e., stuck-at-0 or stuck-at-1 logic faults. Thus, other fault models have been developed to address specific failure mechanisms. For example, the *bridging* fault model addresses electrical shorts that cause unwanted coupling or spurious feedback loops.

Fault Testing

Once the physical faults that may cause device malfunction have been identified and categorized and how the physical faults relate to logical faults has been determined, the next task is to develop tests to detect these faults. When the tests are generated by a computer program, this activity is called *automatic test program generation* (ATPG). Examples of fault testing techniques are listed below:

- Stuck-at techniques,
- Scan techniques,
- Signature techniques,
- Coding techniques, and
- Electrical monitoring techniques.

Basic stuck-at fault testing techniques address combinational digital systems. Three of the most popular stuck-at fault testing techniques are the D algorithm, the Path-Oriented Decision Making (Podem) algorithm, and the Fan algorithm. These algorithms first identify a circuit fault, e.g., stuck-at-0 or stuck-at-1, and then try to generate an input stimulus that detects the fault and makes the fault visible at an output. Detecting a fault is called *fault sensitization* and making a fault visible is called *fault propagation*. To illustrate this process, consider the simple combinational design in Fig. 34.18 [Goel, 1981; Fujiwara and Shimono, 1983].

The combinational digital design is defective because a manufacturing defect has caused the output of the top **and** gate to be permanently tied to ground, i.e., stuck-at-0, using a positive logic convention. To sensitize the fault, the inputs A and B should both be set to 1, which should force the top **and** gate output to a 1 for a good circuit. To propagate the fault, the inputs C and D should both be set to 0, which should force the **xor** gate output to 1, again for a good circuit. Thus, if A = 1, B = 1, C = 0, and D = 0 in Fig. 34.18, then a good circuit would yield a 1, but the defective circuit yields a 0, which detects the stuck-at-0 fault at the top **and** gate output.

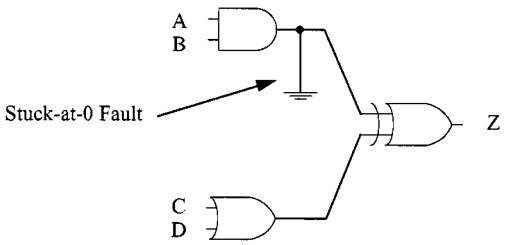

FIGURE 34.18 Combinational logic stuck-at fault testing.

Sequential ATP generation is more difficult than combinational ATPG because exercising or sensitizing a particular circuit path to detect the presence of a possible manufacturing fault may require a *sequence* of input test vectors. One technique for testing sequential digital systems is scan fault testing. Scan fault testing is an example of *design for testability* (DFT) because it modifies or constrains a design in a manner that facilitates fault testing. Scan techniques impose a logic design discipline that connects all state registers into one or more chains to form "scan rings," as shown in Fig. 34.19 [Eichelberger and Williams, 1977].

During normal device operation, the scan rings are disabled and the registers serve as conventional memory (state) storage elements. During test operation, the scan rings are enabled and stimulus test vectors are shifted into the memory elements to set the state of the digital system. The digital system is exercised for one clock cycle and then the results are shifted out of the scan ring to record the response.

A variation of scan DFT, called *boundary scan*, has been defined for testing integrated circuits on printed circuit boards (PCBs). Advancements in PCB manufacturing, such as fine-lead components, surface mount assembly, and **multichip modules**, have yielded high-density boards with fewer access points to probe individual pins. These PCBs are difficult to test. As the name implies, boundary scan imposes a design discipline for PCB components to enable the input/output pins of the components to be connected into scan chains. As an example, Fig. 34.20 shows a simple PCB containing two integrated circuits configured for boundary scan. Each integrated circuit contains scan registers between its input/output pins and its core logic to enable the PCB test bus to control and observe the behavior of individual integrated circuits [Parker, 1989].

Another DFT technique is signature analysis, also called *built-in self-test* (BIST). Signature testing techniques use additional logic, typically linear feedback shift registers, to generate automatically pseudorandom test vectors. The output responses are compressed into a single vector and compared with a known good vector. If the output response vector does not exactly match the known good vector, then the design is considered faulty.

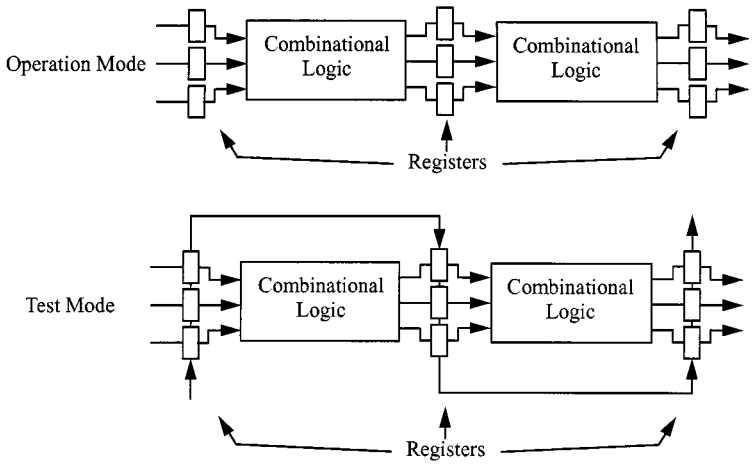

FIGURE 34.19 Scan-based DFT.

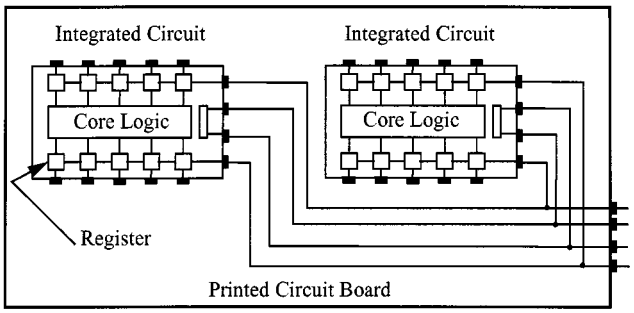

FIGURE 34.20 Boundary scan.

Matching the output response vector and a known good vector does not guarantee correct hardware; however, if enough pseudorandom test vectors are exercised, then the chances are acceptably small of obtaining a false positive result. Signature analysis is often used to test memories [Agrawal et al., 1993].

Coding test techniques encode signal information so that errors can be detected and possibly corrected. Although often implemented in software, coding techniques can also be implemented in hardware. For example, a simple coding technique called *parity checking* is often implemented in hardware. Parity checking adds an extra bit to multibit data. The parity bit is set such that the total number of logic 1s in the multibit data *and* parity bit is either an even number (even parity) or an odd number (odd parity). An error has occurred if an even-parity-encoded signal contains an odd number of logic 1s or if an odd-parity-encoded signal contains an even number of logic 1s. Coding techniques are used extensively to detect and correct transmission errors on system buses and networks, storage errors in system memory, and computational errors in processors [Peterson and Weldon, 1972].

Finally, electrical monitoring testing techniques, also called current/voltage testing, rely on the simple observation that an out-of-range current or voltage often indicates a defective or bad part. Possibly a short or open is present causing a particular input/output signal to have the wrong voltage or current. Current testing, or I_{ddq} *testing*, is particularly useful for digital systems using CMOS integrated circuit technology. Normally, CMOS circuits have very low static or quiescent currents. However, physical faults, such as **gate oxide** defects, can increase static current by several orders of magnitude. Such a substantial change in static current is straightforward to detect. The principal advantages of current testing are that the tests are simple and the fault models address detailed transistor-level defects. However, current testing requires that enough time be allotted between input stimuli to allow the circuit to reach a static state, which slows testing down and causes problems with circuits that cannot be tested at scaled clock rates.

34.7 Summary

DA technology offers the potential of serving as a powerful fulcrum in leveraging the skills of a designer against the growing demands of electronic system design and manufacturing. DA programs help to relieve the designer of the burden of tedious, repetitive tasks that can be labor-intensive and error prone.

DA technology can be broken down into several topical areas, such as design entry, synthesis, verification, physical design, and test. Each topical area has developed an extensive body of knowledge and experience.

Design entry defines a desired specification. Synthesis refines the initial design specification into a detailed design ready for implementation. Verification checks that the detailed design faithfully realizes the desired specification. Physical design defines the implementation, i.e., the actual hardware. Finally, test checks that the manufactured part is functionally and parametrically correct.

Defining Terms

Bipolar: Type of semiconductor transistor that involves both minority and majority carrier conduction mechanisms.

BiCMOS: Bipolar/complementary metal-oxide semiconductor. A logic family and form of microelectronic fabrication.

Branch: A circuit element between two nodes. Branch current is the current through the branch. Branch voltage is the potential difference between the nodes. The relationship between the branch current and voltage is defined by the branch constitutive relationship.

Capacitor: Two-terminal electronic device governed by the branch constitutive relationship, Charge = Capacitance $\times$ Voltage.

CMOS: Complementary metal-oxide semiconductor. A logic family and form of microelectronic fabrication.

Data Flow: Nonprocedural modeling style in which the textual order that statements are written has no bearing on the order in which they execute.

Design automation: Computer programs that assist engineers in performing digital system development.

Design entry: Area of DA addressing modeling analog and digital electronic systems. Design entry uses a hierarchy of models involving physical, electrical, logical, functional, and architectural abstractions.

Electromigration: Gradual erosion of metal due to excessive currents.

Fan-in/fan-out: Fan-in defines the maximum number of logic elements that may drive another logic element. Fan-out defines the maximum number of logic elements a logic element may drive.

Finite state machine: Sequential digital system. A finite state machine is classified as either Moore and Mealy.

Gate array: Application-specific integrated circuit implementation technique that realizes a digital system by programming the metal interconnect of a prefabricated array of gates.

Gate oxide: Dielectric insulating material between the gate and source/drain terminals of a MOS transistor.

Ground bounce: Transient condition when the potential of a ground network varies appreciably from its uniform static value.

Integrated circuit: Electronic circuit manufactured on a monolithic piece of semiconductor material, typically silicon.

Kirchoff's current law: The amount of current entering a circuit node equals the amount of current leaving a circuit node.

Kirchoff's voltage law: Any closed loop of circuit branch voltages sums to zero.

Masking defects: Defects in masking plate patterns used for integrated circuit lithography that result in errant material composition and/or placement.

Multichip modules: Multiple integrated circuits interconnected on a monolithic substrate.

Netlist: Collection of wires that are electrically connected to each other.

NMOS: N-type metal-oxide semiconductor. A logic family and form of microelectronic fabrication.

Node voltage: Potential of a circuit node relative to ground potential.

Programmable logic devices (PLDs): Generic logic devices that can be programmed to realize specific digital systems. PLDs include programmable logic arrays, programmable array logic, memories, and field-programmable gate arrays.

Resistor: Two-terminal electronic device governed by the branch constitutive relationship, Voltage = Resistance × Current.

Silicon compilation: Synthesis application that generates final physical design ready for silicon fabrication.

Simulation: Computer program that examines the dynamic semantics of a model of a digital system by applying a series of inputs and generating the corresponding outputs. Major types of simulation include schedule driven, cycle driven, and message driven.

Skew: Timing difference between two events that are supposed to occur simultaneously.

Standard cell: Application-specific integrated circuit implementation technique that realizes a digital system using a library of predefined (standard) logic cells.

Synthesis: Computer program that helps generate a digital/analog system design by transforming a high-level model of abstract behavior into a lower-level model of more-detailed behavior.

Test: Area of EDA that addresses detecting faulty hardware. Test involves stuck-at, scan, signature, coding, and monitoring techniques.

Timing analysis: Verifies timing behavior of electronic system including rise time, fall time, setup time, hold time, glitch detection, clock periods, race conditions, reflections, and cross talk.

Transistor: Electronic device that enables a small voltage and/or current to control a larger voltage and/or current. For analog systems, transistors serves as amplifiers. For digital systems, transistors serve as switches.

Verification: Area of EDA that addresses validating designs for correct function and expected performance. Verification involves timing analysis, simulation, emulation, and formal proofs.

VHDL: Hardware description language used as an international standard for communicating electronic systems information.

Via: Connection or contact between two materials that are otherwise electrically isolated.

Related Topics

23.2 Testing • 25.1 Integrated Circuit Technology • 25.3 Application-Specific Integrated Circuits

References

Design Automation

D. Barbe, Ed., *Very Large Scale Integration (VLSI) — Fundamentals and Applications*, New York: Springer-Verlag, 1980.

T. Dillinger, *VLSI Engineering*, Englewood Cliffs, N.J.: Prentice-Hall, 1988.

S. Sapiro, *Handbook of Design Automation*, Englewood Cliffs, Prentice-Hall, N.J.: 1986

S. Trimberger, *An Introduction to CAD for VLSI*, Calif.: Domancloud Publishers, 1990.

Design Entry

G. Birtwistle, and Subrahmanyan, P., *VLSI Specification, Verification, and Synthesis*, Boston: Kluwer Academic Publishers, 1988.

A. Dewey, "VHSIC hardware description language development program," *Proceedings Design Automation Conference*, June, 1983.

A. Dewey, "VHDL: towards a unified view of design," *IEEE Design and Test of Computers*, June, 1992.

A. Dewey, *Analysis and Design of Digital Systems with VHDL*, Boston: PWS Publishing, 1997.

J. Douglas-Young, *Complete Guide to Reading Schematic Diagrams*, Englewood Cliffs, N.J.: Prentice-Hall, 1988.

M. Pechet, Ed., *Handbook of Electrical Package Design*, New York: Marcel Dekker, 1981.

J. Peterson, *Petri Net Theory and Modeling of Systems*, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

Synthesis

R. Ayres, *VLSI: Silicon Compilation and the Art of Automatic Microchip Design*, Englewood Cliffs, N.J.: Prentice-Hall, 1983.

Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Boston: Kluwer Academic Publishers, 1992.

R. Camposano, and Wolfe, W., *High-Level VLSI Synthesis*, Boston: Kluwer Academic Publishers, 1991.
D. Gajski, Ed., *Silicon Compilation*, Boston: Addison-Wesley, 1988.
D. Gajski, et al., *High-Level Synthesis — Introduction to Chip and System Design*, Boston: Kluwer Academic Publishers, 1992.
P. Paulin, and Knight, J., "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Design and Test of Computers*, October, 1989.
T. Sasao, Ed., *Logic Synthesis and Optimization*, Boston: Kluwer Academic Publishers, 1993.

Verification

R. Bryant, "Simulation on distributed systems," *Proceedings International Conference on Distributed Systems,* 1979.
K. Chandy, and Misra, J., "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, April, 1981.
L. Chua, and Lin, P., *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Englewood Cliffs, N.J.: Prentice-Hall, 1975.
G. Hachtel, Brayton, R., and Gustavson, F., "The sparse tableau approach to network analysis and design," *IEEE Transactions on Circuit Theory*, CT-18, 1971.
W. Hahn, and Fischer, K., "High performance computing for digital design simulation", *VLSI85*, New York: Elsevier Science Publishers, 1985.
C. Ho, Ruehli, A., and Brennan, P., "The modified nodal analysis approach to network analysis," *IEEE Transactions on Circuits and Systems*, 1975.
R. McHaney, *Computer Simulation: A Practical Perspective*, New York: Academic Press, 1991.
T. McWilliams, and Widdoes, L., "SCALD — structured computer aided logic design," *Proceedings Design Automation Conference*, June, 1978.
L. Nagel, SPICE2: A Computer Program to Simulate Semiconductor Circuits, Electronic Research Laboratory, ERL-M520, Berkeley: University of California, 1975.
U. Pooch, *Discrete Event Simulation: A Practical Approach*, Boca Raton, Fla.: CRC Press, 1993.
T. Sasiki, et al., "Hierarchical design and verification for large digital systems," in *Proceedings Design Automation Conference*, June, 1978.
S. Takasaki, Hirose, F., and Yamada, A., "Logic simulation engines in Japan," *IEEE Design and Test of Computers*, October, 1989.
S. Walters, "Computer-aided prototyping for ASIC-based synthesis," *IEEE Design and Test of Computers*, June, 1991.
Z. Zlatev, "On some pivotal strategies in Gaussian elimination by sparse technique," *SIAM Journal of Numerical Analysis*, vol. 17, no. 1, 1980.

Physical Design

E. Hollis, *Design of VLSI Gate Array Integrated Circuits*, Englewood Cliffs, N.J.: Prentice-Hall, 1987.
B. Preas, B., Lorenzetti, M., and Ackland, B., Eds., *Physical Design Automation of VLSI Systems*, New York: Benjamin Cummings, 1988.
S. Sait and Youssef, H., *VLSI Physical Design Automation: Theory and Practice*, New York: McGraw-Hill, 1995.
B. Spinks, *Introduction to Integrated Circuit Layout*, Englewood Cliffs, N.J.: Prentice-Hall, 1985.

Test

V. Agrawal, Kime, C., and Saluja, K., "A tutorial on built-in self-test," *IEEE Design and Test of Computers*, June, 1993.
A. Buckroyd, *Computer Integrated Testing*, New York: Wiley, 1989.
E. Eichelberger, and Williams, T., "A logic design structure for LSI testability," *Proceedings Design Automation Conference*, June, 1977.
H. Fujiwar, and Shimono, T., "On the acceleration of test generation algorithms," *IEEE Transactions on Computers*, December, 1983.
P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, March, 1981.

K. Parker, "The impact of boundary scan on board test," *IEEE Design and Test of Computers*, August, 1989.
W. Peterson and Weldon, E., *Error-Correcting Codes*, Boston: The MIT Press, 1972.
M. Weyerer and Goldemund, G., *Testability of Electronic Circuits*, Englewood Cliffs, N.J.: Prentice-Hall, 1992.
G. Zobrist, Ed., *VLSI Fault Modeling and Testing Technologies*, New York: Ablex Publishing Company, 1993.