1. INTRODUCTION

Testing and Diagnosis

Testing of a system is an experiment in which the system is exercised and its resulting response is analyzed to ascertain whether it behaved correctly. If incorrect behavior is detected, a second goal of a testing experiment may be to *diagnose*, or locate, the cause of the misbehavior. Diagnosis assumes knowledge of the internal structure of the system under test. These concepts of testing and diagnosis have a broad applicability; consider, for example, medical tests and diagnoses, test-driving a car, or debugging a computer program.

Testing at Different Levels of Abstraction

The subject of this book is *testing and diagnosis of digital systems.* "Digital system" denotes a complex digital circuit. The *complexity* of a circuit is related to the *level of abstraction* required to describe its operation in a meaningful way. The level of abstraction can be roughly characterized by the type of information processed by the circuit (Figure 1.1). Although a digital circuit can be viewed as processing analog quantities such as voltage and current, the lowest level of abstraction we will deal with is the *logic level.* The information processed at this level is represented by discrete *logic values.* The classical representation uses *binary logic values* (0 and 1). More accurate models, however, often require more than two logic values. A further distinction can be made at this level between combinational and sequential circuits. Unlike a combinational circuit, whose output logic values depend only on its present input values, a sequential circuit can also remember past values, and hence it processes *sequences of logic values.*

Control	Data	Level of abstraction
Logic values (or sequences of logic values)		Logic level
Logic values	Words	Register level
Instructions	Words	Instruction set level
Programs	Data structures	Processor level
Messages		System level

Figure 1.1 Levels of abstraction in information processing by a digital system

We start to regard a circuit as a system when considering its operation in terms of processing logic values becomes meaningless and/or unmanageable. Usually, we view a system as consisting of a *data* part interacting with a *control* part. While the control function is still defined in terms of logic values, the information processed by the data part consists of *words*, where a word is a group (*vector*) of logic values. As data words are stored in registers, this level is referred to as the *register level.* At the next level of abstraction, the *instruction set level*, the control information is also organized as words, referred to as *instructions*. A system whose operation is directed by a set of instructions is called an *instruction set processor*. At a still higher level of abstraction, the *processor level*, we can regard a digital system as processing sequences of instructions, or *programs*, that operate on blocks of data, referred to as *data structures*. A different view of a system (not necessarily a higher level of abstraction) is to consider it composed of independent subsystems, or units, which communicate via blocks of words called *messages*; this level of abstraction is usually referred to as the *system level*.

In general, the stimuli and the response defining a testing experiment correspond to the type of information processed by the system under test. Thus testing is a generic term that covers widely different activities and environments, such as

- one or more subsystems testing another by sending and receiving messages;
- a processor testing itself by executing a diagnostic program;
- automatic test equipment (ATE) checking a circuit by applying and observing binary patterns.

In this book we will not be concerned with *parametric tests*, which deal with electrical characteristics of the circuits, such as threshold and bias voltages, leakage currents, and so on.

Errors and Faults

An instance of an incorrect operation of the system being tested (or UUT for *unit under test*) is referred to as an *(observed) error*. Again, the concept of error has different meanings at different levels. For example, an error observed at the diagnostic program level may appear as an incorrect result of an arithmetic operation, while for ATE an error usually means an incorrect binary value.

The causes of the observed errors may be *design errors*, *fabrication errors*, *fabrication defects*, and *physical failures*. Examples of design errors are

- incomplete or inconsistent specifications;
- incorrect mappings between different levels of design;
- violations of design rules.

Errors occurring during fabrication include

- wrong components;
- incorrect wiring;
- shorts caused by improper soldering.

Fabrication defects are not directly attributable to a human error; rather, they result from an imperfect manufacturing process. For example, shorts and opens are common defects in manufacturing MOS Large-Scale Integrated (LSI) circuits. Other fabrication defects include improper doping profiles, mask alignment errors, and poor encapsulation. Accurate location of fabrication defects is important in improving the manufacturing yield.

Physical failures occur during the lifetime of a system due to component wear-out and/or environmental factors. For example, aluminum connectors inside an IC package thin out with time and may break because of electron migration or corrosion. Environmental factors, such as temperature, humidity, and vibrations, accelerate the aging of components. Cosmic radiation and α-particles may induce failures in chips containing high-density random-access memories (RAMs). Some physical failures, referred to as "infancy failures," appear early after fabrication.

Fabrication errors, fabrication defects, and physical failures are collectively referred to as *physical faults*. According to their stability in time, physical faults can be classified as

- *permanent*, i.e., always being present after their occurrence;
- *intermittent*, i.e., existing only during some intervals;
- *transient*, i.e., a one-time occurrence caused by a temporary change in some environmental factor.

In general, physical faults do not allow a direct mathematical treatment of testing and diagnosis. The solution is to deal with *logical faults*, which are a convenient representation of the effect of the physical faults on the operation of the system. A fault is *detected* by observing an error caused by it. The basic assumptions regarding the nature of logical faults are referred to as a *fault model*. The most widely used fault model is that of a single line (wire) being permanently "stuck" at a logic value. Fault modeling is the subject of Chapter 4.

Modeling and Simulation

As design errors precede the fabrication of the system, *design verification testing* can be performed by a testing experiment that uses a *model* of the designed system. In this context, "model" means a digital computer representation of the system in terms of data structures and/or programs. The model can be exercised by stimulating it with a representation of the input signals. This process is referred to as *logic simulation* (also called design verification simulation or true-value simulation). Logic simulation determines the evolution in time of the signals in the model in response to an applied input sequence. We will address the areas of modeling and logic simulation in Chapters 2 and 3.

Test Evaluation

An important problem in testing is *test evaluation*, which refers to determining the effectiveness, or quality, of a test. Test evaluation is usually done in the context of a fault model, and the quality of a test is measured by the ratio between the number of faults it detects and the total number of faults in the assumed fault universe; this ratio is referred to as the *fault coverage*. Test evaluation (or test grading) is carried out via a simulated testing experiment called *fault simulation*, which computes the response of the circuit in the presence of faults to the test being evaluated. A fault is detected when the response it produces differs from the expected response of the fault-free circuit. Fault simulation is discussed in Chapter 5.

Types of Testing

Testing methods can be classified according to many criteria. Figure 1.2 summarizes the most important attributes of the testing methods and the associated terminology.

Criterion	Attribute of testing method	Terminology
When is testing performed?	• Concurrently with the normal system operation • As a separate activity	On-line testing Concurrent testing Off-line testing
Where is the source of the stimuli?	• Within the system itself • Applied by an external device (tester)	Self-testing External testing
What do we test for?	• Design errors • Fabrication errors • Fabrication defects • Infancy physical failures • Physical failures	Design verification testing Acceptance testing Burn-in Quality-assurance testing Field testing Maintenance testing
What is the physical object being tested?	• IC • Board • System	Component-level testing Board-level testing System-level testing
How are the stimuli and/or the expected response produced?	• Retrieved from storage • Generated during testing	Stored-pattern testing Algorithmic testing Comparison testing
How are the stimuli applied?	• In a fixed (predetermined) order • Depending on the results obtained so far	 Adaptive testing

Figure 1.2 Types of testing

Testing by *diagnostic programs* is performed off-line, at-speed, and at the system level. The stimuli originate within the system itself, which works in a self-testing mode. In systems whose control logic is microprogrammed, the diagnostic programs can also be microprograms (microdiagnostics). Some parts of the system, referred to as *hardcore*, should be fault-free to allow the program to run. The stimuli are generated by software or

Criterion	Attribute of testing method	Terminology
How fast are the stimuli applied?	• Much slower than the normal operation speed	DC (static) testing
	• At the normal operation speed	AC testing At-speed testing
What are the observed results?	• The entire output patterns	
	• Some function of the output patterns	Compact testing
What lines are accessible for testing?	• Only the I/O lines	Edge-pin testing
	• I/O and internal lines	Guided-probe testing Bed-of-nails testing Electron-beam testing In-circuit testing In-circuit emulation
Who checks the results?	• The system itself	Self-testing Self-checking
	• An external device (tester)	External testing

Figure 1.2 (Continued)

firmware and can be adaptively applied. Diagnostic programs are usually run for field or maintenance testing.

In-circuit emulation is a testing method that eliminates the need for hardcore in running diagnostic programs. This method is used in testing microprocessor (μP)-based boards and systems, and it is based on removing the μP on the board during testing and accessing the μP connections with the rest of the UUT from an external tester. The tester can emulate the function of the removed μP (usually by using a μP of the same type). This configuration allows running of diagnostic programs using the tester's μP and memory.

In *on-line testing*, the stimuli and the response of the system are not known in advance, because the stimuli are provided by the patterns received during the normal mode of operation. The object of interest in on-line testing consists not of the response itself, but of some properties of the response, properties that should remain invariant throughout the fault-free operation. For example, only one output of a fault-free decoder should have logic value 1. The operation code (opcode) of an instruction word in an instruction set processor is restricted to a set of "legal" opcodes. In general, however, such easily definable properties do not exist or are difficult to check. The general approach to on-line testing is based on *reliable design techniques* that create invariant properties that are easy to check during the system's operation. A typical example is the use of an additional parity bit for every byte of memory. The parity bit is set to create an easy-to-check invariant property, namely it makes every extended byte (i.e., the original byte plus the parity bit) have the same parity (i.e., the number of 1 bits, taken modulo 2). The parity bit is *redundant*, in the sense that it does not carry any information useful for the normal operation of the system. This type of *information redundancy* is characteristic for systems using *error-detecting and error-correcting codes*. Another type of reliable design based on redundancy is *modular redundancy*, which is based on replicating a module several times. The replicated modules (they must have the same function, possibly with different implementations) work with the same set of inputs, and the invariant property is that all of them must produce the same response. Self-checking systems have subcircuits called *checkers*, dedicated to testing invariant properties. Self-checking design techniques are the subject of Chapter 13.

Guided-probe testing is a technique used in board-level testing. If errors are detected during the initial edge-pin testing (this phase is often referred to as a GO/NO GO test), the tester decides which internal line should be monitored and instructs the operator to place a probe on the selected line. Then the test is reapplied. The principle is to trace back the propagation of error(s) along path(s) through the circuit. After each application of the test, the tester checks the results obtained at the monitored line and determines whether the site of a fault has been reached and/or the backtrace should continue. Rather than monitoring one line at a time, some testers can monitor a group of lines, usually the pins of an IC.

Guided-probe testing is a sequential diagnosis procedure, in which a subset of the internal accessible lines is monitored at each step. Some testers use a fixture called *bed-of-nails* that allows monitoring of all the accessible internal lines in a single step.

The goal of *in-circuit testing* is to check components already mounted on a board. An external tester uses an IC clip to apply patterns directly to the inputs of one IC and to observe its outputs. The tester must be capable of electronically isolating the IC under test from its board environment; for example, it may have to overdrive the input pattern supplied by other components.

Algorithmic testing refers to the generation of the input patterns during testing. Counters and feedback shift registers are typical examples of hardware used to generate the input stimuli. *Algorithmic pattern generation* is a capability of some testers to produce combinations of several fixed patterns. The desired combination is determined by a control program written in a tester-oriented language.

The expected response can be generated during testing either from a known good copy of the UUT — the so-called *gold unit* — or by using a real-time emulation of the UUT. This type of testing is called *comparison testing*, which is somehow a misnomer, as the comparison with the expected response is inherent in many other testing methods.

Methods based on checking some function *f(R)* derived from the response *R* of the UUT, rather than *R* itself, are said to perform *compact testing*, and *f(R)* is said to be a *compressed* representation, or *signature*, of *R*. For example, one can count the number of 1 values (or the number of 0 to 1 and 1 to 0 transitions) obtained at a circuit output and compare it with the expected 1-count (or transition count) of the fault-free circuit. Such a compact testing procedure simplifies the testing process, since instead of bit-by-bit comparisons between the UUT response and the expected output, one needs only one comparison between signatures. Also the tester's memory requirements are significantly reduced, because there is no longer need to store the entire expected response. Compression techniques (to be analyzed in Chapter 10) are mainly used in self-testing circuits, where the computation of *f(R)* is implemented by special hardware added to the circuit. Self-testing circuits also have additional hardware to generate the stimuli. Design techniques for circuits with Built-In Self-Test (BIST) features are discussed in Chapter 11.

Diagnosis and Repair

If the UUT found to behave incorrectly is to be repaired, the cause of the observed error must be diagnosed. In a broad sense, the terms diagnosis and repair apply both to physical faults and to design errors (for the latter, "repair" means "redesign"). However, while physical faults can be effectively represented by logical faults, we lack a similar mapping for the universe of design errors. Therefore, in discussing diagnosis and repair we will restrict ourselves to physical (and logical) faults.

Two types of approaches are available for fault diagnosis. The first approach is a *cause-effect analysis*, which enumerates all the possible faults (causes) existing in a fault model and determines, before the testing experiment, all their corresponding responses (effects) to a given applied test. This process, which relies on fault simulation, builds a data base called a *fault dictionary*. The diagnosis is a dictionary look-up process, in which we try to match the actual response of the UUT with one of the precomputed responses. If the match is successful, the fault dictionary indicates the possible faults (or the faulty components) in the UUT.

Other diagnosis techniques, such as guided-probe testing, use an *effect-cause analysis* approach. An effect-cause analysis processes the actual response of the UUT (the effect) and tries to determine directly only the faults (cause) that could produce that response. Logic-level diagnosis techniques are treated in Chapter 12 and system-level diagnosis is the subject of Chapter 15.

Test Generation

Test generation (TG) is the process of determining the stimuli necessary to test a digital system. TG depends primarily on the testing method employed. On-line testing methods do not require TG. Little TG effort is needed when the input patterns are provided by a feedback shift register working as a pseudorandom sequence generator. In contrast, TG for design verification testing and the development of diagnostic programs involve a large effort that, unfortunately, is still mainly a manual activity. *Automatic TG* (ATG) refers to TG algorithms that, given a model of a system, can generate tests for it. ATG has been developed mainly for edge-pin stored-pattern testing.

TG can be *fault oriented* or *function oriented*. In fault-oriented TG, one tries to generate tests that will detect (and possibly locate) specific faults. In function-oriented TG, one tries to generate a test that, if it passes, shows that the system performs its specified function. TG techniques are covered in several chapters (6, 7, 8, and 12).

Design for Testability

The *cost of testing* a system has become a major component in the cost of designing, manufacturing, and maintaining a system. The cost of testing reflects many factors such as TG cost, testing time, ATE cost, etc. It is somehow ironic that a $10 μP may need a tester thousands times more expensive.

Design for testability (DFT) techniques have been increasingly used in recent years. Their goal is to reduce the cost of testing by introducing testability criteria early in the design stage. Testability considerations have become so important that they may even dictate the overall structure of a design. DFT techniques for external testing are discussed in Chapter 9.