supports the test fixture. The light on top indicates a functioning or stopped machine and is designed to be visible across a production floor where many machines might be operating. A screen at the top right provides status information to the operator. The unit has wheels for easy movement, but also has firm footings, which are lowered when the machine is in use.

Handlers add a constant time to the test process, typically around 1 second. Thus, load boards and handlers are often constructed to deal with two or four chips at once to reduce the cost of testing. Because a load board must be designed to fit to a given handler, select the handler before starting design of the load board.

15.3 Logic Verification Principles

Figure 15.6(a) shows a combinational circuit with N inputs. To test this circuit exhaustively, a sequence of 2^N inputs (or test vectors) must be applied and observed to fully exercise the circuit. This combinational circuit is converted to a sequential circuit with addition of M registers, as shown in Figure 15.6(b). The state of the circuit is determined by the inputs and the previous state. A minimum of 2^{N+M} test vectors must be applied to exhaustively test the circuit. As observed by [Williams83] more than two decades ago,

> *With LSI, this may be a network with $N = 25$ and $M = 50$, or 2^{75} patterns, which is approximately 3.8×10^{22}. Assuming one had the patterns and applied them at an application rate of 1 μs per pattern, the test time would be over a billion years (10^9).*

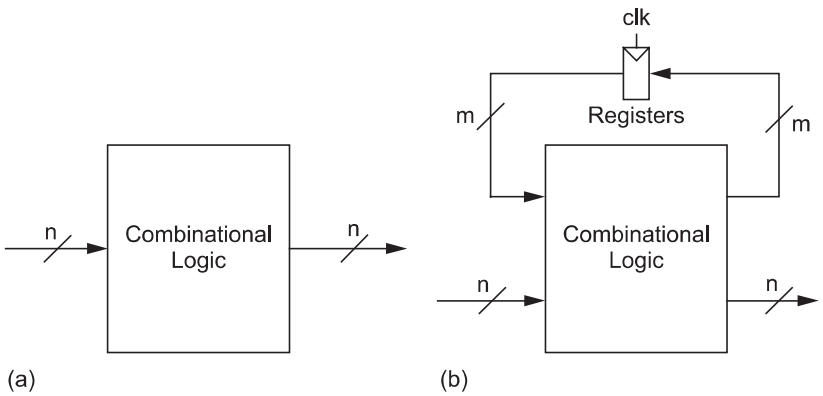

FIGURE 15.6 The combinational explosion in test vectors

Clearly, exhaustive testing is infeasible for most systems. Fortunately, the number of potentially nonfunctional nodes on a chip is much smaller than the number of states. The verification engineer must cleverly devise test vectors that detect any (or nearly any) defective node without requiring so many patterns.

15.3.1 Test Vectors

Test vectors are a set of patterns applied to inputs and a set of expected outputs. Both logic verification and manufacturing test require a good set of test vectors. The set should be

large enough to catch all the logic errors and manufacturing defects, yet small enough to keep test time (and cost) reasonable.

Directed and *random* vectors are the most common types. Directed vectors are selected by an engineer who is knowledgeable about the system. Their purpose is to cover the corner cases where the system might be most likely to malfunction. For example, in a 32-bit datapath, likely corner cases include the following:

```
0x00000000  All zeros
0xFFFFFFFF  All ones
0x00000001  One in the lsb
0x80000000  One in the msb
0x55555555  Alternating 0's and 1's
0xAAAAAAAA  Alternating 1's and 0's
0x7A39D281  A random value
```

The circuit could be tested by applying all combinations of these directed vectors to the various inputs. Directed vectors are an efficient way to catch the most obvious design errors and a good logic designer will always run a set of directed tests on a new piece of RTL to ensure a minimum level of quality.

Applying a large number of random or semirandom vectors is a surprisingly good way to detect more subtle errors. The effectiveness of the set of vectors is measured by the fault coverage, which is discussed in Section 15.5.6. Automatic test pattern generation tools are good at producing high fault coverage for manufacturing test and are discussed in Section 15.5.7.

15.3.2 Testbenches and Harnesses

A verification *test bench* or *harness* is a piece of HDL code that is placed as a wrapper around a core piece of HDL to apply and check test vectors. In the simplest test bench, input vectors are applied to the module under test and at each cycle, the outputs are examined to determine whether they comply with a predefined expected data set. The expected outputs can be derived from the golden model and saved as a file or the value can be computed on the fly.

Simulators usually provide settable break points and single or multiple stepping abilities to allow the designer to step through a test sequence while debugging discrepancies.

15.3.3 Regression Testing

High-level language scripts are frequently used when running large testbenches, especially for *regression testing*. Regression testing involves performing a suite of simulations to automatically verify that no functionality has inadvertently changed in a module or set of modules. During a design, it is common practice to run a regression script every night after design activities have concluded to check that bug fixes or feature enhancements have not broken completed modules.

Example 15.3

Figure 14.11 showed a possible software radio architecture that used a combination of an IQ conversion block and a multiplier-based multiprocessor. The following regression testing might be done:

```
Test IQ Conversion
   Test Upconverter
      Test NCO
         Test Read and Write of All Registers
         Test Phase Incrementer
         Test Phase Adder
         Test Sine ROM (Read Contents)
         Test Overall NCO at a set of frequencies
      Test Multiplier
   Test Downconverter
      Test NCO
         ...
      Test Multiplier
         ...
      Test Low Pass Filter
         ...
Test Microprocessor Memory Core
   Test Microprocessor
      Test ALU
      Test Instruction Decode
      Test Program Counter
      Test Register File Read/Write
      Exhaustive Instruction Test
   Test Memory Read/Write
Test Interprocessor Bus IO
Test IQ Conversion to Processor pathways
Test Overall Software Radio Functionality
```

Note the way in which the correctness of modules is slowly built up by verifying lower-level models first. The low-level tests are gradually built up in complexity until the complete functionality can be verified. At low levels, it is easier to exhaustively verify that logic is correct. For instance, we can verify that the sine ROM is in fact generating a sine wave for one frequency. We then use this knowledge to postulate that it generates correct sine waves for all input frequencies when we verify at the levels above the NCO. At the chip level, we assume that IQ conversion is correct for all combinations of signal frequency and local oscillator frequency even though we may only check a small subset. If we started at the top level and ran a simulation for a few frequencies, we could never have confidence that the lower levels were correct. In addition, if there is a problem, trying to locate the problem by debugging at the top level is futile. Running regression tests from the bottom up is designed to overcome this verification nightmare.

15.3.4 Version Control

Combined with regression testing is the use of versioning, that is, the orderly management of different design iterations. Unix/Linux tools such as CVS or Subversion are useful for this.

Example 15.4

In the software radio example, the regression testing halts at the ALU test in the example given above. Working late, the design leader, Vanessa Eagleeye, examines the CVS

history and discovers that Fred Codechanger has made an edit to the ALU design to try a new adder during the day. She is able to revert the code to what was previously working and then rerun the regression test and have a peaceful night's sleep. Fred corrects his mistake the next day and is advised to remember to run the regression verification step before submitting such hurried edits.

15.3.5 Bug Tracking

Another important tool to use during verification (and in fact the whole design cycle) is a bug-tracking system. Bug-tracking systems such as the Unix/Linux based GNATS allow the management of a wide variety of bugs. In these systems, each bug is entered and the location, nature, and severity of the bug noted. The bug discoverer is noted, along with the perceived person responsible for fixing the bug.

Example 15.5

After Example 15.4, Vanessa enters a bug report describing the bug. She cites Fred as the person responsible and the level as severe. The next day, Fred fixes the problem and changes the bug status to fixed. The bug report is kept in the system, but does not appear in any listing of outstanding bugs. It is kept to track the re-introduction of bugs, as this might give managers an idea of a problem area in the design management.

Tracking the number of bugs can give you an idea of the rate at which a design is converging toward a finished state. If the trend is downward, the design is converging. On the other hand, an upward trend tends to indicate a design early in its verification cycle.

15.4 Silicon Debug Principles

The area of basic digital debugging was introduced in Section 15.1.2. A major challenge in silicon debugging is when the chip operates incorrectly, but you cannot ascertain the cause by making measurements at the chip pins or scan chain outputs (see Section 15.6.2).

There are a number of techniques for directly accessing the silicon. First, specific signals can be brought to the top of the chip as *probe points*. These are small squares (5–10 μm on a side) of top-level metal that connect to key points in the circuit that the designer has had the foresight to include before debug. The overglass cut mask should specify a hole in the passivation over the probe pads so the metal can be reliably contacted. Typical of these kinds of test points might be internal bias points in linear circuits or perhaps key points in a high-speed signal chain (be careful not to excessively load the circuit to be probed). The exposed squares can be probed with a picoprobe (fine-tipped probe) in a fixture under a microscope. During design, the load of the picoprobe has to be taken into account by providing buffers if necessary. The Model 35 probe from GGB Industries shown in Figure 15.7 has a capacitance of 50 fF, input resistance of 1.25 MΩ, and frequency response from DC to 26 GHz. It can probe down to a 10×10 μm window.

The die can also be probed electrically or optically if mechanical contact is not feasible. An *electron beam* (ebeam) probe uses a scanning electron microscope to produce a