TEST

ASICs are tested at two stages during manufacture using production tests . First, the silicon die are tested after fabrication is complete at wafer test or wafer sort . Each wafer is tested, one die at a time, using an array of probes on a probe card that descend onto the bonding pads of a single die. The production tester applies signals generated by a test program and measures the ASIC test response . A test program often generates hundreds of thousands of different test vectors applied at a frequency of several megahertz over several hundred milliseconds. Chips that fail are automatically marked with an ink spot. Production testers are large machines that take up their own room and are very expensive (typically well over $1 million). Either the customer, or the ASIC manufacturer, or both, develops the test program.

A diamond saw separates the die, and the good die are bonded to a lead carrier and packaged. A second, final test is carried out on the packaged ASIC (usually with the same test vectors used at wafer sort) before the ASIC is shipped to the customer. The customer may apply a goods-inward test to incoming ASICs if the customer has the resources and the product volume is large enough. Normally, though, parts are directly assembled onto a bare printed-circuit board (PCB or board) and then the board is tested. If the board test shows that an ASIC is bad at this point, it is difficult to replace a surface-mounted component soldered on the board, for example. If there are several board failures due to a particular ASIC, the board manufacturer typically ships the defective chips back to the ASIC vendor. ASIC vendors have sophisticated failure analysis departments that take packaged ASICs apart and can often determine the failure mechanism. If the ASIC production tests are adequate, failures are often due to the soldering process, electrostatic damage during handling, or other problems that can occur between the part being shipped and board test. If the problem is traced to defective ASIC fabrication, this indicates that the test program may be inadequate. As we shall see, failure and diagnosis at the board level is very expensive. Finally, ASICs may be tested and replaced (usually by swapping boards) either by a customer who buys the final product or by servicing-this is field repair . Such system-level diagnosis and repair is even more expensive.

Programmable ASICs (including FPGAs) are a special case. Each programmable ASIC is tested to the point that the manufacturer can guarantee with a high degree of confidence that if your design works, and if you program the FPGA correctly, then your ASIC will work. Production testing is easier for some programmable ASIC architectures than others. In a reprogrammable technology the manufacturer can test the programming features. This cannot be done for a one-time programmable antifuse technology, for example. A programmable ASIC is still tested in a similar fashion to any other ASIC and you are still paying for test development and design. Programmable ASICs also have similar test, defect, and manufacturing problems to other members of the ASIC family. Finally, once a programmable ASIC is soldered to a board and part of a system, it looks just like any other ASIC. As you will see in the next section, considering board-level and system-level testing is a very important part of ASIC design.

14.1 The Importance of Test

14.2 Boundary-Scan Test

14.3 Faults

14.4 Fault Simulation

14.5 Automatic Test-Pattern Generation

14.6 Scan Test

14.7 Built-in Self-test

14.8 A Simple Test Example

14.9 The Viterbi Decoder Example

14.10 Summary

14.11 Problems

14.12 Bibliography

14.13 References

14.1 The Importance of Test

One measure of product quality is the defect level . If the ABC Company sells 100,000 copies of a product and 10 of these are defective, then we say the defect level is 0.1 percent or 100 ppm. The average quality level (AQL) is equal to one minus the defect level (ABC's AQL is thus 99.9 percent).

Suppose the semiconductor division of ABC makes an ASIC, the bASIC, for the PC division. The PC division buys 100,000 bASICs, tested by the semiconductor division, at $10 each. The PC division includes one surface-mounted bASIC on each PC motherboard it assembles for the aPC computer division. The aPC division tests the finished motherboards. Rejected boards due to defective bASICs incur an average $200 board repair cost. The board repair cost as a function of the ASIC defect level is shown in Table 14.1 . A defect level of 5 percent in bASICs costs $1 million dollars in board repair costs (the same as the total ASIC part cost). Things are even worse at the system level, however.

TABLE 14.1 Defect levels in printed-circuit boards (PCB). 1

ASIC defect level	Defective ASICs	Total PCB repair cost
5%	5000	$1million
1%	1000	$200,000
0.1%	100	$20,000
0.01%	10	$2,000

Suppose the ABC Company sells its aPC computers for $5,000, with a profit of $500 on each. Unfortunately the aPC division also has a defect level. Suppose that 10 percent of the motherboards that contain defective bASICs that passed the chip test also manage to pass the board tests (10 percent may seem high, but chips that have hard-to-test faults at the chip level may be very hard to find at the board level-catching 90 percent of these rogue chips would be considered good). The system-level repair cost as a function of the bASIC defect level is shown in Table 14.2 . In this example a 5 percent defect level in a $10 bASIC part now results in a $5 million cost at the system level. From Table 14.2 we can see it would be worth spending $4 million (i.e., $5 million - $1 million) to reduce the bASIC defect density from 5 percent to 1 percent.

TABLE 14.2 Defect levels in systems. 2

ASIC defect level	Defective ASICs	Defective boards	Total repair cost at system level
5%	5000	500	$5 million
1%	1000	100	$1 million
0.1%	100	10	$100 ,000
0.01%	10	1	$10,000

1. Assumptions: The number of parts shipped is 100,000; part price is $10; total part cost is $1 million; the cost of a fault in an assembled PCB is $200.

2. Assumptions: The number of systems shipped is 100,000; system cost is $5,000; total cost of systems shipped is $500 million; the cost of repairing or replacing a system due to failure is $10,000; profit on 100,000 systems is $50 million.

14.2 Boundary-Scan Test

It is possible to test ICs in dual-in-line packages (DIPs) with 0.1 inch (2.5 mm) lead spacing on low-density boards using a bed-of-nails tester with probes that contact test points underneath the board. Mechanical testing becomes difficult with board trace widths and separations below 0.1 mm or 100 mm, package-pin separations of 0.3 mm or less, packages with 200 or more pins, surface-mount packages on both sides of the board, and multilayer boards [Scheiber, 1995].

In 1985 a group of European manufacturers formed the Joint European Test Action Group (JETAG) to

study board testing. With the addition of North American companies, JETAG became the Joint Test Action Group (JTAG) in 1986. The JTAG 2.0 test standard formed the basis of the IEEE Standard 1149.1 Test Port and Boundary-Scan Architecture [IEEE 1149.1b, 1994], approved in February 1990 and also approved as a standard by the American National Standards Institute (ANSI) in August 1990 [Bleeker, v. d. Eijnden, and de Jong, 1993; Maunder and Tulloss, 1990; Parker, 1992]. The IEEE standard is still often referred to as JTAG, although there are important differences between the last JTAG specification (version 2.0) and the IEEE 1149.1 standard.

Boundary-scan test (BST) is a method for testing boards using a four-wire interface (five wires with an optional master reset signal). A good analogy would be the RS-232 interface for PCs. The BST standard interface was designed to test boards, but it is also useful to test ASICs. The BST interface provides a standard means of communicating with test circuits on-board an ASIC. We do need to include extra circuits on an ASIC in order to use BST. This is an example of increasing the cost and complexity (as well as potentially reducing the performance) of an ASIC to reduce the cost of testing the ASIC and the system.

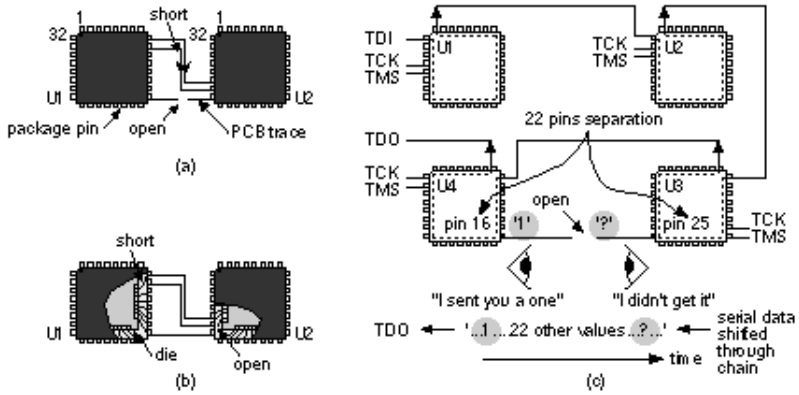


FIGURE 14.1 IEEE 1149.1 boundary scan. (a) Boundary scan is intended to check for shorts or opens between ICs mounted on a board. (b) Shorts and opens may also occur inside the IC package. (c) The boundary-scan architecture is a long chain of shift registers allowing data to be sent over all the connections between the ICs on a board.

Figure 14.1 (a) illustrates failures that may occur on a PCB due to shorts or opens in the copper traces on the board. Less frequently, failures in the ASIC package may also arise from shorts and opens in the wire bonds between the die and the package frame (Figure 14.1 b). Failures in an ASIC package that occur during ASIC fabrication are caught by the ASIC production test, but stress during automated handling and board assembly may cause package failures. Figure 14.1 (c) shows how a group of ASICs are linked together in boundary-scan testing. To detect the failures shown in Figure 14.1 (a) or (b) manufacturers use boundary scan to test every connection between ASICs on a board. During boundary scan, test data is loaded into each ASIC and then driven onto the board traces. Each ASIC monitors its inputs, captures the data received, and then shifts the captured data out. Any defects in the board or ASIC connections will show up as a discrepancy between expected and actual measured continuity data.

In order to include BST on an ASIC, we add a special logic cell to each ASIC I/O pad. These cells are joined together to form a chain and create a boundary-scan shift register that extends around each ASIC. The input to a boundary-scan shift register is the test-data input (TDI). The output of a boundary-scan

shift register is the test-data output (TDO). These boundary-scan shift registers are then linked in a serial fashion with the boundary-scan shift registers on other ASICs to form one long boundary-scan shift register. The boundary-scan shift register in each ASIC is one of several test-data registers (TDR) that may be included in each ASIC. All the TDRs in an ASIC are connected directly between the TDI and TDO ports. A special register that decodes instructions provides a way to select a particular TDR and control operation of the boundary-scan test process.

Controlling all of the operations involved in selecting registers, loading data, performing a test, and shifting out results are the test clock (TCK) and test-mode select (TMS). The boundary-scan standard specifies a four-wire test interface using the four signals: TDI, TDO, TCK, and TMS. These four dedicated signals, the test-access port (TAP), are connected to the TAP controller inside each ASIC. The TAP controller is a state machine clocked on the rising edge of TCK, and with state transitions controlled by the TMS signal. The test-reset input signal (TRST* , nTRST , or TRST -always an active-low signal) is an optional (fifth) dedicated interface pin to reset the TAP controller.

Normally the boundary-scan shift-register cells at each ASIC I/O pad are transparent, allowing signals to pass between the I/O pad and the core logic. When an ASIC is put into boundary-scan test mode, we first tell the TAP controller which TDR to select. The TAP controller then tells each boundary-scan shift register in the appropriate TDR either to capture input data, to shift data to the neighboring cell, or to output data.

There are many acronyms in the IEEE 1149.1 standard (referred to as " dot one "); Table 14.3 provides a list of the most common terms.

TABLE 14.3 Boundary-scan terminology.

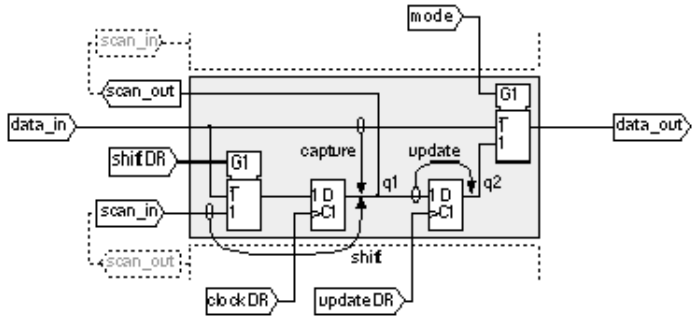
Acronym	Meaning	Explanation
BR	Bypass register	A TDR, directly connects TDI and TDO, bypassing BSR
BSC	Boundary-scan cell	Each I/O pad has a BSC to monitor signals
BSR	Boundary-scan register	A TDR, a shift register formed from a chain of BSCs
BST	Boundary-scan test	Not to be confused with BIST (built-in self-test)
IDCODE	Device-identification register	Optional TDR, contains manufacturer and part number
IR	Instruction register	Holds a BST instruction, provides control signals
JTAG	Joint Test Action Group	The organization that developed boundary scan
TAP	Test-access port	Four- (or five-)wire test interface to an ASIC
TCK	Test clock	A TAP wire, the clock that controls BST operation
TDI	Test-data input	A TAP wire, the input to the IR and TDRs
TDO	Test-data output	A TAP wire, the output from the IR and TDRs
TDR	Test-data register	Group of BST registers: IDCODE, BR, BSR
TMS	Test-mode select	A TAP wire, together with TCK controls the BST state
TRST* or nTRST	Test-reset input signal	Optional TAP wire, resets the TAP controller

TRST* or nTRST Test-reset input signal (active-low)

14.2.1 BST Cells

Figure 14.2 shows a data-register cell (DR cell) that may be used to implement any of the TDRs. The most common DR cell is a boundary-scan cell (BS cell , or BSC), or boundary-register cell (this last name is not abbreviated to BR cell, since this term is reserved for another type of cell) [IEEE 1149.1b-1994, p. 10-18, Fig. 10-16].

A BSC contains two sequential elements. The capture flip-flop or capture register is part of a shift register formed by series connection of BSCs. The update flip-flop , or update latch , is normally drawn as an edge-triggered D flip-flop, though it may be a transparent latch. The inputs to a BSC are: scan in (serial in or SI); data in (parallel in or PI); and a control signal, mode (also called test / normal). The BSC outputs are: scan out (serial out or SO); data out (parallel out or PO). The BSC in Figure 14.2 is reversible and can be used for both chip inputs and outputs. Thus data_in may be connected to a pad and data_out to the core logic or vice versa.



entity DR_cell **is port** (mode, data_in, shiftDR, scan_in, clockDR, updateDR: BIT;

data_out, scan_out: **out** BIT); **end** DR_cell;

architecture behave **of** DR_cell **is signal** q1, q2 : BIT; **begin**

CAP : **process** (clockDR) **begin if** clockDR = ’1’ **then**

if shiftDR = ’0’ **then** q1 <= data_in; **else** q1 <= scan_in; **end if** ; **end if** ;

end process ;

UPD : **process** (updateDR) **begin if** updateDR = ’1’ **then** q2 <= q1; **end if** ; **end process** ;

data_out <= data_in **when** mode = ’0’ **else** q2; scan_out <= q1;

end behave;

FIGURE 14.2 A DR (data register) cell. The most common use of this cell is as a boundary-scan cell (BSC).

The IEEE 1149.1 standard shows the sequential logic in a BSC controlled by the gated clocks: clockDR (whose positive edge occurs at the positive edge of TCK) and updateDR (whose positive edge occurs at the negative edge of TCK). The IEEE 1149.1 schematics illustrate the standard but do not define how circuits should be implemented. The function of the circuit in Figure 14.2 (and its model) follows the IEEE 1149.1 standard and many other published schematics, but this is not necessarily the best, or even a safe, implementation. For example, as drawn here, signals clockDR and updateDR are gated clocks-normally to be avoided if possible. The update sequential element is shown as an edge-triggered D flip-flop but may be implemented using a latch.

Figure 14.3 [IEEE 1149.1b-1994, Chapter 9] shows a bypass-register cell (BR cell). The BR inputs and outputs, scan in (serial in, SI) and scan out (serial out, SO), have the same names as the DR cell ports, but DR cells and BR cells are not directly connected.

```
entity BR_cell is port (

 clockDR,shiftDR,scan_in : BIT; scan_out : out BIT );

end BR_cell;

architecture behave of BR_cell is

signal t1 : BIT; begin t1 <= shiftDR and scan_in;

process (clockDR) begin

if (clockDR = '1') then scan_out <= t1; end if ;

end process ;

end behave;
```

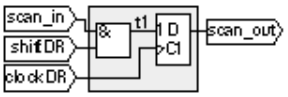
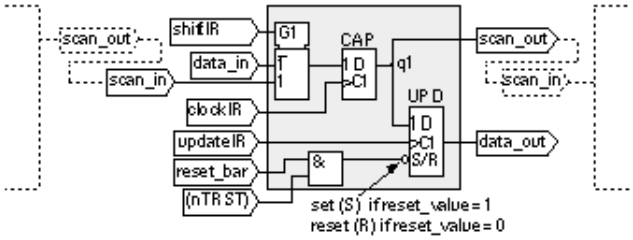


FIGURE 14.3 A BR (bypass register) cell.

Figure 14.4 shows an instruction-register cell (IR cell) [IEEE 1149.1b-1994, Chapter 6]. The IR cell inputs are: scan_in , data_in ; as well as clock, shift, and update signals (with names and functions similar to those of the corresponding signals in the BSC). The reset signals are nTRST and reset_bar (active-low signals often use an asterisk, reset* for example, but this is not a legal VHDL name). The two LSBs of data_in must permanently be set to '01' (this helps in checking the integrity of the scan chain during testing). The remaining data_in bits are status bits under the control of the designer. The update sequential element (sometimes called the shadow register) in each IR cell may be set or reset (depending on reset_value). The IR cell outputs are: data_out (the instruction bit passed to the

instruction decoder) and scan_out (the data passed to the next IR cell in the IR).



entity IR_cell **is port** (

shiftIR, data_in, scan_in, clockIR, updateIR, reset_bar, nTRST, reset_value : BIT;

data_out, scan_out : **out** BIT); **end** IR_cell;

architecture behave **of** IR_cell **is signal** q1, SR : BIT; **begin**

scan_out <= q1; SR <= reset_bar **and** nTRST;

CAP: **process** (clockIR) **begin**

if (clockIR = ’1’) **then**

if (shiftIR = ’0’) **then** q1 <= data_in; **else** q1 <= scan_in; **end if** ;

end if ;

end process ;

UPD: **process** (updateIR, SR) **begin**

if (SR = ’0’) **then** data_out <= reset_value;

elsif ((updateIR = ’1’) **and** updateIR’EVENT) **then** data_out <= q1;

end if ;

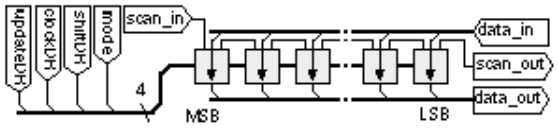
end process ;

end behave;

FIGURE 14.4 An IR (instruction register) cell.

14.2.2 BST Registers

Figure 14.5 shows a boundary-scan register (BSR), which consists of a series connection, or chain, of BSCs. The BSR surrounds the ASIC core logic and is connected to the I/O pad cells. The BSR monitors (and optionally controls) the inputs and outputs of an ASIC. The direction of information flow is shown by an arrow on each of the BSCs in Figure 14.5 . The control signal, mode , is decoded from the IR. Signal mode is drawn as common to all cells for the BSR in Figure 14.5 , but that is not always the case.



entity BSR **is**

generic (width : INTEGER := 3);

port (shiftDR, clockDR, updateDR, mode, scan_in : BIT;

scan_out : **out** BIT;

data_in : BIT_VECTOR(width-1 **downto** 0);

data_out : **out** BIT_VECTOR(width-1 **downto** 0));

end BSR;

architecture structure **of** BSR **is**

component DR_cell **port** (

mode, data_in, shiftDR, scan_in, clockDR, updateDR : BIT;

data_out, scan_out : **out** BIT);

end component ;

for all : DR_cell **use entity** WORK.DR_cell(behave);

signal int_scan : BIT_VECTOR (data_in'RANGE);

begin

BSR : **for** i **in** data_in'LOW **to** data_in'HIGH **generate**

RIGHT : **if** (i = 0) **generate**

BSR_LSB : DR_cell **port map** (mode, data_in(i), shiftDR,

int_scan(i), clockDR, updateDR, data_out(i), scan_out);

end generate ;

MIDDLE : **if** ((i > 0) **and** (i < data_in'HIGH)) **generate**

BSR_i : DR_cell **port map** (mode, data_in(i), shiftDR,

int_scan(i), clockDR, updateDR, data_out(i), int_scan(i-1));

end generate ;

LFET : **if** (i = data_in'HIGH) **generate**

BSR_MSB : DR_cell **port map** (mode, data_in(i), shiftDR,

scan_in, clockDR, updateDR, data_out(i), int_scan(i-1));

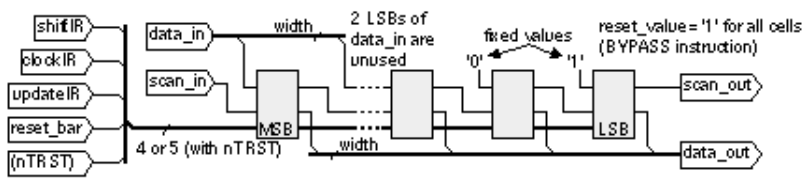
end generate ;

end generate ;

end structure;

FIGURE 14.5 A BSR (boundary-scan register). An example of the component data-register (DR) cells (used as boundary-scan cells) is shown in Figure 14.2 .

Figure 14.6 shows an instruction register (IR), which consists of at least two IR cells connected in series. The IEEE 1149.1 standard specifies that the IR cell is reset to '00...01' (the optional IDCODE instruction). If there is no IDCODE TDR, then the IDCODE instruction defaults to the BYPASS instruction.



entity IR **is generic** (width : INTEGER := 4); **port** (

shiftIR, clockIR, updateIR, reset_bar, nTRST, scan_in : BIT; scan_out : **out** BIT;

```
data_in : BIT_VECTOR (width-1 downto 0) ;
data_out : out BIT_VECTOR (width-1 downto 0) );
end IR;

architecture structure of IR is
component IR_cell port (shiftIR, data_in, scan_in, clockIR,
updateIR, reset_bar, nTRST, reset_value : BIT ; data_out, scan_out : out BIT );
end component ;
for all : IR_cell use entity WORK.IR_cell(behave);
signal int_scan : BIT_VECTOR (data_in'RANGE);
signal Vdd : BIT := '1'; signal GND : BIT := '0';
begin
IRGEN : for i in data_in'LOW to data_in'HIGH generate
FIRST : if (i = 0) generate
IR_LSB: IR_cell port map (shiftIR, Vdd, int_scan(i),
clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), scan_out);
end generate ;
SECOND : if ((i = 1) and (data_in'HIGH > 1)) generate
IR1 : IR_cell port map (shiftIR, GND, int_scan(i),
clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));
end generate ;
MIDDLE : if ((i < data_in'HIGH) and (i > 1)) generate
IRi : IR_cell port map (shiftIR, data_in(i), int_scan(i),
clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));
```

end generate ;

LAST : **if** (i = data_in'HIGH) **generate**

IR_MSB : IR_cell **port map** (shiftIR, data_in(i), scan_in,

clockIR, updateIR, reset_bar, nTRST, Vdd, data_out(i), int_scan(i-1));

end generate ; **end generate** ;

end structure;

FIGURE 14.6 An IR (instruction register).

14.2.3 Instruction Decoder

Table 14.4 shows an instruction decoder . This model is capable of decoding the following minimum set of boundary-scan instructions:

1. EXTEST , external test. Drives a known value onto each output pin to test connections between ASICs.
2. SAMPLE/PRELOAD (often abbreviated to SAMPLE). Performs two functions: first sampling the present input value from input pad during capture; and then preloading the BSC update register output during update (in preparation for an EXTEST instruction, for example).
3. IDCODE . An optional instruction that allows the device-identification register (IDCODE) to be shifted out. The IDCODE TDR is an optional register that allows the tester to query the ASIC for the manufacturer's name, part number, and other data that is shifted out on TDO. IDCODE defaults to the BYPASS instruction if there is no IDCODE TDR.
4. BYPASS . Selects the single-cell bypass register (instead of the BSR) and allows data to be quickly shifted between ASICs.

The IEEE 1149.1 standard predefines additional optional instructions and also defines the implementation of custom instructions that may use additional TDRs.

TABLE 14.4 An IR (instruction register) decoder.

entity IR_decoder **is generic** (width : INTEGER := 4); **port** (

shiftDR, clockDR, updateDR : BIT; IR_PO : BIT_VECTOR (width-1 **downto** 0) ;

test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR, updateBSR : **out** BIT);

end IR_decoder;

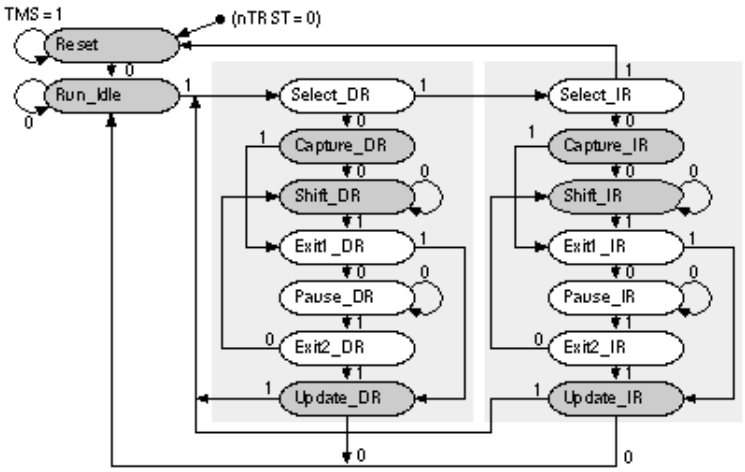
architecture behave **of** IR_decoder **is**

```
type INSTRUCTION is (EXTEST, SAMPLE_PRELOAD, IDCODE, BYPASS);

signal I : INSTRUCTION;

begin process (IR_PO) begin case BIT_VECTOR'( IR_PO(1), IR_PO(0) ) is

when "00" => I <= EXTEST; when "01" => I <= SAMPLE_PRELOAD;

when "10" => I <= IDCODE; when "11" => I <= BYPASS;

end case ; end process ;

test_mode <= '1' when I = EXTEST else '0';

selectBR  <= '1' when (I = BYPASS or I = IDCODE) else '0';

shiftBR <= shiftDR;

clockBR <= clockDR when (I = BYPASS or I = IDCODE) else '1';

shiftBSR <= shiftDR;

clockBSR <= clockDR when (I = EXTEST or I = SAMPLE_PRELOAD) else '1';

updateBSR <= updateDR when (I = EXTEST or I = SAMPLE_PRELOAD) else '0';

end behave;
```

14.2.4 TAP Controller

Figure 14.7 shows the TAP controller finite-state machine. The 16-state diagram contains some symmetry: states with suffix '_DR' operate on the data registers and those with suffix '_IR' apply to the instruction register. All transitions between states are determined by the TMS (test mode select) signal and occur at the rising edge of TCK , the boundary-scan clock. An optional active-low reset signal, nTRST or TRST* , resets the state machine to the initial state, Reset . If the dedicated nTRST is not used, there must be a power-on reset signal (POR)-not an existing system reset signal.

The outputs of the TAP controller are not shown in Figure 14.7 , but are derived from each TAP controller state. The TAP controller operates rather like a four-button digital watch that cycles through several states (alarm, stopwatch, 12 hr / 24 hr, countdown timer, and so on) as you press the buttons. Only the shaded states in Figure 14.7 affect the ASIC core logic; the other states are intermediate steps. The pause states let the controller jog in place while the tester reloads its memory with a new set of test vectors, for example.



use work.TAP. **all** ; **entity** TAP_sm_states **is**

port (TMS, TCK, nTRST : **in** BIT; S : **out** TAP_STATE); **end** TAP_sm_states;

architecture behave **of** TAP_sm_states **is**

type STATE_ARRAY **is array** (TAP_STATE, 0 to 1) **of** TAP_STATE;

constant T : STATE_ARRAY := ((Run_Idle, Reset),

(Run_Idle, Select_DR), (Capture_DR, Select_IR), (Shift_DR, Exit1_DR),

(Shift_DR, Exit1_DR), (Pause_DR, Update_DR), (Pause_DR, Exit2_DR),

(Shift_DR, Update_DR), (Run_Idle, Select_DR), (Capture_IR, Reset),

(Shift_IR, Exit1_IR), (Shift_IR, Exit1_IR), (Pause_IR, Update_IR),

(Pause_IR, Exit2_IR), (Shift_IR, Update_IR), (Run_idle, Select_DR));

begin process (TCK, nTRST) **variable** S_i: TAP_STATE; **begin**

if (nTRST = ’0’) **then** S_i := Reset;

elsif (TCK = ’1’ **and** TCK’EVENT) **then** -- transition on +VE clock edge

if (TMS = ’1’) **then** S_i := T(S_i, 1); **else** S_i := T(S_i, 0); **end if** ;

end if ; S <= S_i; -- update signal with already updated internal variable

end process ;

end behave;

FIGURE 14.7 The TAP (test-access port) controller state machine.

Table 14.5 shows the output control signals generated by the TAP state machine. I have taken the unusual step of writing separate entities for the state machine and its outputs. Normally this is bad practice because it makes it difficult for synthesis tools to extract and optimize the logic, for example. This separation of functions reflects the fact that the operation of the TAP controller state machine is precisely defined by the IEEE 1149.1 standard-independent of the implementation of the register cells and number of instructions supported. The model in Table 14.5 contains the following combinational, registered, and gated output signals and will change with different implementations:

- reset_bar . Resets the IR to IDCODE (or BYPASS in absence of IDCODE TDR).
- selectIR . Connects a register, the IR or a TDR, to TDO .
- enableTDO . Enables the three-state buffer that drives TDO . This allows data to be shifted out of the ASIC on TDO , either from the IR or from the DR, in states shift_IR or shift_DR respectively.
- shiftIR . Selects the serial input to the capture flip-flop in the IR cells.
- clockIR . Causes data at the input of the IR to be captured or the contents of the IR to be shifted toward TDO (depending on shiftIR) on the negative edge of TCK following the entry to the states shift_IR or capture_IR . This is a dirty signal.
- updateIR . Clocks the update sequential element on the positive edge of TCK at the same time as the exit from state update_IR . This is a dirty signal.
- shiftDR , clockDR , and updateDR . Same functions as corresponding IR signals applied to the TDRs. These signals may be gated to the appropriate TDR by the instruction decoder.

The signals reset_bar , enableTDO , shiftIR , and shiftDR are registered or clocked by TCK (on the positive edge of TCK). We say these signals are clean (as opposed to being dirty gated clocks).

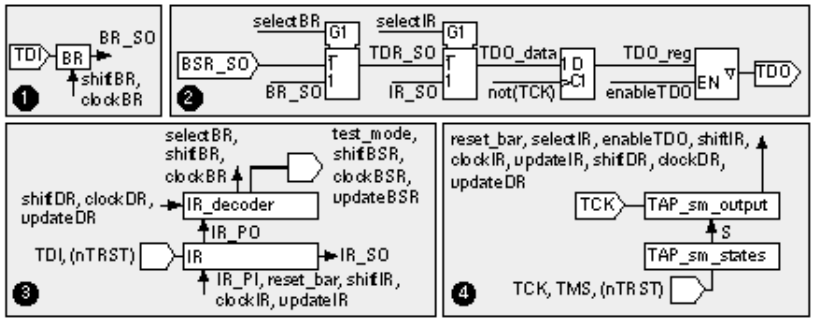
TABLE 14.5 The TAP (test-access port) control. 1

State / Output	Reset	Run_Idle	Select_DR	Capture_DR	Shift_DR	Exit1_DR	Pause_DR	Exit2_DR	Updat
reset_bar	0R								
selectIR	1	1							
enableTDO					1R				
shiftIR									
clockIR									
updateIR									
shiftDR					1R				
clockDR				0G	0G				
updateDR									1G

```
use work.TAP. all ; entity TAP_sm_output is
port (TCK : in BIT; S : in TAP_STATE; reset_bar, selectIR, enableTDO, shiftIR,
clockIR, updateIR, shiftDR, clockDR, updateDR : out BIT);
end TAP_sm_output;
architecture behave_1 of TAP_sm_output is begin -- registered outputs
process (TCK) begin if ( (TCK = '0') and TCK'EVENT ) then
if S = Reset then reset_bar <= '0'; else reset_bar <= '1'; end if ;
if S = Shift_IR or S = Shift_DR then enableTDO <= '1'; else enableTDO <= '0'; end if ;
if S = Shift_IR then ShiftIR <= '1'; else shiftIR <= '0'; end if ;
if S = Shift_DR then ShiftDR <= '1'; else shiftDR <= '0'; end if ;
end if ;
end process ;
process (TCK) begin -- dirty outputs gated with not(TCK)
if (TCK = '0' and (S = Capture_IR or S = Shift_IR))
then clockIR <= '0'; else clockIR <= '1'; end if ;
if (TCK = '0' and (S = Capture_DR or S = Shift_DR))
then clockDR <= '0'; else clockDR <= '1'; end if ;
if TCK = '0' and S=Update_IR then updateIR <= '1'; else updateIR <= '0'; end if ;
if TCK = '0' and S=Update_DR then updateDR <= '1'; else updateDR <= '0'; end if ;
end process ;
selectIR <= '1' when (S = Reset or S = Run_Idle or S = Capture_IR or S = Shift_IR
or S = Exit1_IR or S = Pause_IR or S = Exit2_IR or S = Update_IR) else '0';
end behave_1;
```

14.2.5 Boundary-Scan Controller

Figure 14.8 shows a boundary-scan controller. It contains the following four parts:



library IEEE; **use** IEEE.std_logic_1164. **all** ; use work.TAP. **all** ;

entity Control **is generic** (width : INTEGER := 2); **port** (TMS, TCK, TDI, nTRST : BIT;

TDO: **out** STD_LOGIC; BSR_SO : BIT; BSR_PO : BIT_VECTOR (width-1 **downto** 0);

shiftBSR, clockBSR, updateBSR, test_mode : **out** BIT); **end** Control;

architecture mixed **of** Control **is use** work.BST_components. **all** ;

signal reset_bar, selectIR, enableTDO, shiftIR, clockIR, updateIR, shiftDR,

clockDR, updateDR, IR_SO, BR_SO, TDO_reg, TDO_data, TDR_SO, selectBR,

clockBR, shiftBR : BIT;

signal IR_PI, IR_PO : BIT_VECTOR (1 **downto** 0); **signal** S : TAP_STATE;

begin

IR_PI <= "01";

TDO <= TO_STDULOGIC(TDO_reg) **when** enableTDO = ’1’ **else** ’Z’;

R1 : **process** (TCK) **begin if** (TCK=’0’) **then** TDO_reg <= TDO_data; **end if** ; **end process** ;

TDO_data <= IR_SO **when** selectIR = ’1’ **else** TDR_SO;

TDR_SO <= BR_SO **when** selectBR = ’1’ **else** BSR_SO;

TC1 : TAP_sm_states **port map** (TMS, TCK, nTRST, S);

TC2 : TAP_sm_output **port map** (TCK, S, reset_bar, selectIR, enableTDO,

shiftIR, clockIR, updateIR, shiftDR, clockDR, updateDR);

IR1 : IR **generic map** (width => 2) **port map** (shiftIR, clockIR, updateIR,

reset_bar, nTRST, TDI, IR_SO, IR_PI, IR_PO);

DEC1 : IR_decoder **generic map** (width => 2) **port map** (shiftDR, clockDR, updateDR,

IR_PO, test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR, updateBSR);

BR1 : BR_cell **port map** (clockBR, shiftBR, TDI, BR_SO);

end mixed;

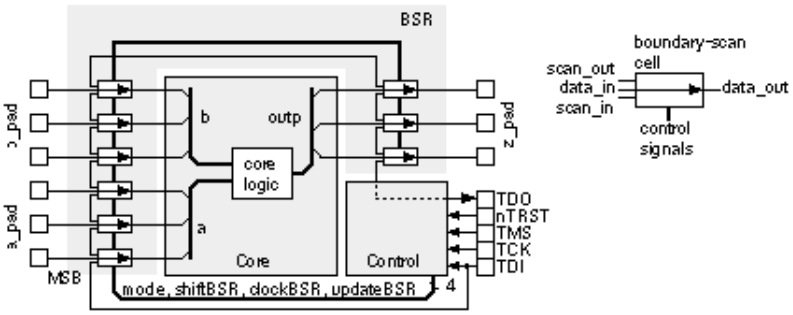
FIGURE 14.8 A boundary-scan controller.

1. Bypass register.
2. TDO output circuit. The data to be shifted out of the ASIC on TDO is selected from the serial outputs of bypass register (BR_SO), instruction register (IR_SO), or boundary-scan register (BSR_SO). Notice the registered output means that data appears on TDO at the negative edge of TCK . This prevents race conditions between ASICs.
3. Instruction register and instruction decoder.
4. TAP controller.

The BSR (and other optional TDRs) are connected to the ASIC core logic outside the BST controller.

14.2.6 A Simple Boundary-Scan Example

Figure 14.9 shows an example of a simple ASIC (our comparator/MUX example) containing boundary scan. The following two packages define the TAP states and the components (these are not essential to understanding what follows, but are included so that the code presented here forms a complete BST model):



entity Core **is port** (a, b : BIT_VECTOR (2 **downto** 0);

outp : **out** BIT_VECTOR (2 **downto** 0)); **end** Core;

architecture behave **of** Core **is begin** outp <= a **when** a < b **else** b;

end behave;

library IEEE; **use** IEEE.std_logic_1164. **all** ;

entity BST_ASIC **is port** (TMS, TCK, TDI, nTRST : BIT; TDO : **out** STD_LOGIC;

a_PAD, b_PAD : BIT_VECTOR (2 **downto** 0); z_PAD : **out** BIT_VECTOR (2 **downto** 0));

end BST_ASIC;

architecture structure **of** BST_ASIC **is use** work.BST_components. **all** ;

component Core **port** (a, b: BIT_VECTOR (2 **downto** 0);

outp: **out** BIT_VECTOR (2 **downto** 0)); **end component** ;

for all : Core **use entity** work.Core(behave);

constant BSR_width : INTEGER := 9;

signal BSR_SO, test_mode, shiftBSR, clockBSR, updateBSR : BIT;

signal BSR_PI, BSR_PO : BIT_VECTOR (BSR_width-1 **downto** 0);

signal a, b, z : BIT_VECTOR (2 **downto** 0);

begin BSR_PI <= a_PAD & b_PAD & z ;

a <= BSR_PO(8 **downto** 6); b <= BSR_PO(5 **downto** 3); z_pad <= BSR_PO(2 **downto** 0);

```
CORE1 : Core port map (a, b, z);
C1 : Control generic map (width => BSR_width) port map (TMS, TCK, TDI, nTRST,
TDO, BSR_SO, BSR_PO, shiftBSR, clockBSR, updateBSR, test_mode);
BSR1 : BSR generic map (width => BSR_width) port map (shiftBSR, clockBSR,
updateBSR, test_mode, TDI, BSR_SO, BSR_PI, BSR_PO);
end structure;
```

FIGURE 14.9 A boundary-scan example.

```
package TAP is
type TAP_STATE is (reset, run_idle, select_DR, capture_DR,
shift_DR, exit1_DR, pause_DR, exit2_DR, update_DR, select_IR,
capture_IR, shift_IR, exit1_IR, pause_IR, exit2_IR, update_IR);
end TAP;
use work.TAP. all ; library IEEE; use IEEE.std_logic_1164. all ;
package BST_Components is
component DR_cell port (
mode, data_in, shiftDR, scan_in, clockDR, updateDR: BIT;
data_out, scan_out : out BIT );
end component ;
component IR_cell port (
shiftIR, data_in, scan_in, clockIR, updateIR, reset_bar,
nTRST, reset_value : BIT; data_out, scan_out : out BIT);
end component ;
component BR_cell port (
```

```
clockDR,shiftDR,scan_in : BIT; scan_out: out BIT );
end component ;
component BSR
generic (width : INTEGER := 5); port (
shiftDR, clockDR, updateDR, mode, scan_in : BIT;
scan_out : out BIT;
data_in : BIT_VECTOR(width-1 downto 0);
data_out : out BIT_VECTOR(width-1 downto 0));
end component ;
component IR generic (width : INTEGER := 4); port (
shiftIR, clockIR, updateIR, reset_bar, nTRST,
scan_in : BIT; scan_out : out BIT;
data_in : BIT_VECTOR (width-1 downto 0) ;
data_out : out BIT_VECTOR (width-1 downto 0) );
end component ;
component IR_decoder generic (width : INTEGER := 4); port (
shiftDR, clockDR, updateDR : BIT;
IR_PO : BIT_VECTOR (width-1 downto 0);
test_mode, selectBR, shiftBR, clockBR, shiftBSR, clockBSR,
updateBSR: out BIT );
end component ;
component TAP_sm_states port (
TMS, TCK, nTRST : in BIT; S : out TAP_STATE); end component ;
component TAP_sm_output port (
```

TCK: BIT; S : TAP_STATE; reset_bar, selectIR,

enableTDO, shiftIR, clockIR, updateIR, shiftDR, clockDR,

updateDR : **out** BIT);

end component ;

component Control **generic** (width : INTEGER := 2); **port** (

TMS, TCK, TDI, nTRST : BIT; TDO : **out** STD_LOGIC;

BSR_SO : BIT; BSR_PO : BIT_VECTOR (width-1 **downto** 0);

shiftBSR, clockBSR, updateBSR, test_mode : **out** BIT);

end component ;

component BST_ASIC **port** (

TMS, TCK, TDI : BIT; TDO : **out** STD_LOGIC;

a_PAD, b_PAD : BIT_VECTOR (2 **downto** 0);

z_PAD : out BIT_VECTOR (2 **downto** 0));

end component ;

end ;

The following testbench, Test_BST , performs these functions:

1. Resets the TAP controller at t = 10 ns using nTRST .
2. Continuously clocks the BST clock, TCK , at a frequency of 10 MHz. Rising edges of TCK occur at 100 ns, 200 ns, and so on.
3. Drives a series of values onto the TAP inputs TDI and TMS . The sequence shifts in instruction code? ’01’ (SAMPLE/PRELOAD),?followed by ’00’ (EXTEST).

library IEEE; **use** IEEE.std_logic_1164. **all** ;

library STD; **use** STD.TEXTIO. **all** ;

entity Test_BST **is end** ;

architecture behave **of** Test_BST **is**

component BST_ASIC **port** (TMS, TCK, TDI, nTRST: BIT;

```
TDO : out STD_LOGIC; a_PAD, b_PAD : BIT_VECTOR (2 downto 0);
z_PAD : out BIT_VECTOR (2 downto 0));
end component ;
for all : BST_ASIC use entity work.BST_ASIC(behave);
signal TMS, TCK, TDI, nTRST : BIT; signal TDO : STD_LOGIC;
signal TDI_TMS : BIT_VECTOR (1 downto 0);
signal a_PAD, b_PAD, z_PAD : BIT_VECTOR (2 downto 0);
begin
TDI <= TDI_TMS(1) ; TMS <= TDI_TMS(0) ;
ASIC1 : BST_ASIC port map
(TMS, TCK, TDI, nTRST, TDO, a_PAD, b_PAD, z_PAD);
nTRST_DRIVE : process begin
nTRST <= '1', '0' after 10 ns, '1' after 20 ns; wait ;
PAD_DRIVE : process begin
a_PAD <= ('0', '1', '0'); b_PAD <= ('0', '1', '1'); wait ;
end process ;
end process ;
TCK_DRIVE : process begin -- rising edge at 100 ns
TCK <= '0' after 50 ns, '1' after 100 ns; wait for 100 ns;
if (now > 3000 ns) then wait ; end if ;
end process ;
BST_DRIVE : process begin TDI_TMS <=
-- State after +VE edge:
('0', '1') after 0 ns, -- Reset
```

('0', '0') **after** 101 ns, -- Run_Idle

('0', '1') **after** 201 ns, -- Select_DR

('0', '1') **after** 301 ns, -- Select_IR

('0', '0') **after** 401 ns, -- Capture_IR

('0', '0') **after** 501 ns, -- Shift_IR

('1', '0') **after** 601 ns, -- Shift_IR

('0', '1') **after** 701 ns, -- Exit1_IR

('0', '1') **after** 801 ns, -- Update_IR, 01 = SAMPLE/PRELOAD

('0', '1') **after** 901 ns, -- Select_DR

('0', '0') **after** 1001 ns, -- Capture_DR

('0', '0') **after** 1101 ns, -- Shift_DR

-- shift 111111101 into BSR, TDI(time) = 101111111 starting now

('1', '0') **after** 1201 ns, -- Shift_DR

('0', '0') **after** 1301 ns, -- Shift_DR

('1', '0') **after** 1401 ns, -- Shift_DR -- shift 4 more 1's

('1', '0') **after** 1901 ns, -- Shift_DR -- in-between

('1', '1') **after** 2001 ns, -- Exit1_DR

('0', '1') **after** 2101 ns, -- Update_DR

('0', '1') **after** 2201 ns, -- Select_DR

('0', '1') **after** 2301 ns, -- Select_IR

('0', '0') **after** 2401 ns, -- Capture_IR

('0', '0') **after** 2501 ns, -- Shift_IR

('0', '0') **after** 2601 ns, -- Shift_IR

('0', '1') **after** 2701 ns, -- Exit1_IR

('0', '1') **after** 2801 ns, -- Update_IR, 00=EXTEST

('0', '0') **after** 2901 ns; -- Run_Idle

wait ;

end process ;

process (TDO, a_pad, b_pad, z_pad) **variable** L : LINE; **begin**

write (L, now, RIGHT, 10); write (L, STRING'(" TDO="));

if TDO = 'Z' **then** write (L, STRING'("Z")) ;

else write (L, TO_BIT(TDO)); **end if** ;

write (L, STRING'(" PADS=")); write (L, a_pad & b_pad & z_pad);

writeline (output, L);

end process ;

end behave;

Here is the output from this testbench:

0 ns TDO=0 PADS=000000000

0 ns TDO=Z PADS=010011000

0 ns TDO=Z PADS=010011010

650 ns TDO=1 PADS=010011010

750 ns TDO=0 PADS=010011010

850 ns TDO=Z PADS=010011010

1250 ns TDO=0 PADS=010011010

1350 ns TDO=1 PADS=010011010

1450 ns TDO=0 PADS=010011010

1550 ns TDO=1 PADS=010011010

1750 ns TDO=0 PADS=010011010

1950 ns TDO=1 PADS=010011010

2050 ns TDO=0 PADS=010011010

2150 ns TDO=Z PADS=010011010

2650 ns TDO=1 PADS=010011010

2750 ns TDO=0 PADS=010011010

2850 ns TDO=Z PADS=010011010

2950 ns TDO=Z PADS=010011101

This trace shows the following activities:

- All changes to TDO and at the pads occur at the negative edge of TCK .
- The core logic output is z_pad = ’010’ and appears at the I/O pads at t = 0 ns. This is the smaller of the two inputs, a_pad = ’010’ and b_pad = ’011’ , and the correct output when the pads are connected to the core logic.
- At t = 650 ns the IDCODE instruction ’01’ is shifted out on TDO (with ’1’ appearing first). If we had multiple ASICs in the boundary-scan chain, this would show us that the chain was intact.
- At t = 850 ns the TDO output is floated (to ’Z’) as we exit the shift_IR state.
- At t = 1200 ns the TAP controller begins shifting the serial data input from TDI (’111111101’) into the BSR.
- At t = 1250 ns the BSR data starts shifting out. This is data that was captured during the SAMPLE/PRELOAD instruction from the device input pins, a_pad and b_pad , as well as the driver of the output pins, z_pad . The data appears as the pattern ’010011010’ . This pattern consists of a_pad = ’010’ , b_pad = ’011’ , followed by z_pad = ’010’ (notice that TDO does not change at t = 1650 ns or 1850 ns).
- At t = 2150 ns, TDO is floated after we exit the shift_DR state.
- At t = 2650 ns the IDCODE instruction ’01’ (loaded into the IR as we passed through capture_IR the second time) is again shifted out as we shift the EXTEST instruction from TDI into the IR.
- At t = 2650 ns the TDO output is floated after we exit the shift_IR state.
- At t = 2950 ns the output, z_pad , is driven with ’101’ . The inputs a_pad and b_pad remain unchanged since they are driven from outside the chip. The change on z_pad occurs on the negative edge of TCK because the IR is loaded with the instruction EXTEST on the negative edge of TCK . When this instruction is decoded, the signal mode changes (this signal controls the MUX at the output of the BSCs).

Figure 14.10 shows a signal trace from the MTI simulator for the last four negative edges of TCK . Notice that we shifted in the test pattern on TDI in the order ’101111111’ . The output z_pad (3 bits wide) is last in the BSR (nearest TDO) and thus is driven with the first 3 bits of this pattern, ’101’ . Forcing ’101’ onto the ASIC output pins would allow us to check that this pattern is correctly received at inputs of other connected ASICs through the bonding wires and board traces. In a later test cycle we can force ’010’ onto z_pad to check that both logic levels can be transmitted and received. We may also capture other signals (which are similarly being forced onto the outputs of neighboring ASICs) at the inputs.

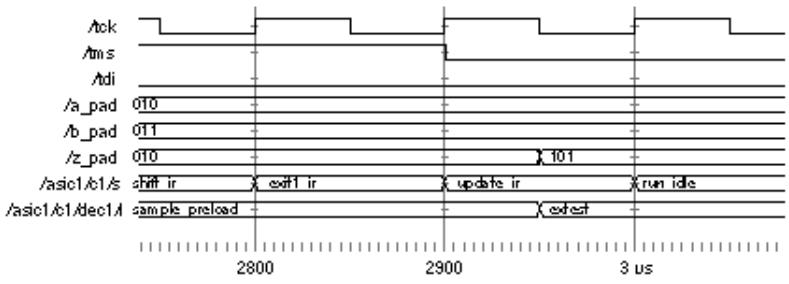


FIGURE 14.10 Results from the MTI simulator for the boundary-scan testbench.

14.2.7 BSDL

The boundary-scan description language (BSDL) is an extension of IEEE 1149.1 but without any overlap. BSDL uses a subset of VHDL. The BSDL for an ASIC is part of an imaginary data sheet; it is not intended for simulation and does not include models for any boundary-scan components. BSDL is a standard way to describe the features and behavior of an ASIC that includes IEEE 1149.1 boundary scan and a standard way to pass information to test-generation software. Using BSDL, test software can also check that the BST features are correct. As an example, test software can use the BSDL to check that the ASIC uses the correct boundary-scan cells for the instructions that claim to be supported. BSDL cannot prove that an implementation works, however.

The following example BSDL description corresponds to our halfgate ASIC example with BST (this code was generated automatically by the Compass tools):

entity asic_p **is**

generic (PHYSICAL_PIN_MAP : STRING := "DUMMY_PACKAGE");

port (

pad_a: **in** BIT_VECTOR (0 **to** 0);

pad_z: **buffer** BIT_VECTOR (0 **to** 0);

TCK: **in** BIT;

TDI: **in** BIT;

TDO: **out** BIT;

TMS: **in** BIT;

TRST: **in** BIT);

use STD_1149_1_1994. **all** ;

attribute PIN_MAP **of** asic_p : **entity is** PHYSICAL_PIN_MAP;

-- CUSTOMIZE package pin mapping.

constant DUMMY_PACKAGE : PIN_MAP_STRING :=

"pad_a:(1)," &

"pad_z:(2)," &

"TCK:3," &

"TDI:4," &

"TDO:5," &

"TMS:6," &

"TRST:7";

attribute TAP_SCAN_IN **of** TDI : **signal is** TRUE;

attribute TAP_SCAN_MODE **of** TMS : **signal is** TRUE;

attribute TAP_SCAN_OUT **of** TDO : **signal** is TRUE;

attribute TAP_SCAN_RESET **of** TRST : **signal** is TRUE;

-- CUSTOMIZE TCK max freq and safe stop state.

attribute TAP_SCAN_CLOCK of TCK : **signal is** (20.0e6, BOTH);

attribute INSTRUCTION_LENGTH **of** asic_p : **entity is** 3;

attribute INSTRUCTION_OPCODE **of** asic_p : **entity is**

"IDCODE (001)," &

"STCTEST (101)," &

"INTEST (100)," &

"BYPASS (111)," &

"SAMPLE (010)," &

"EXTEST (000)";

attribute INSTRUCTION_CAPTURE **of** asic_p : **entity is** "001";

-- attribute INSTRUCTION_DISABLE of asic_p : entity is " "

-- attribute INSTRUCTION_GUARD of asic_p : entity is " "

-- attribute INSTRUCTION_PRIVATE of asic_p : entity is " "

attribute IDCODE_REGISTER **of** asic_p : **entity is**

"0000" & -- 4-bit version

"0000000000000000" & -- 16-bit part number

"00000101011" & -- 11-bit manufacturer

"1"; -- mandatory LSB

-- attribute USERCODE_REGISTER of asic_p : entity is " "

attribute REGISTER_ACCESS **of** asic_p : **entity is**

"BOUNDARY (STCTEST)";

attribute BOUNDARY_CELLS **of** asic_p : **entity is**

"BC_1, BC_2";

attribute BOUNDARY_LENGTH **of** asic_p : **entity is** 2;

attribute BOUNDARY_REGISTER **of** asic_p : **entity is**

-- num cell port function safe [ccell disval rslt]

" 1 (BC_2, pad_a(0), input, X)," &

" 0 (BC_1, pad_z(0), output2, X)";

-- " 98 (BC_1, OE, input, X), " &

-- " 98 (BC_1, *, control, 0), " &

-- " 99 (BC_1, myport(0), output3, X, 98, 0, Z);

end asic_p;

The functions of the lines of this BSDL description are as follows:

- Line 2 refers to the ASIC package. We can have the same part (with identical pad numbers on the silicon die) in different ASIC packages. We include the name of the ASIC package in line 2 and the pin mapping between bonding pads and ASIC package pins in lines 14 - 21 .
- Lines 3 - 10 describe the signal names of inputs and outputs, the TAP pins, and the optional fifth TAP reset signal. The BST signals do not have to be given the names used in the standard: TCK, TDI, and so on.
- Line 11 refers to the VHDL package, STD_1149_1_1994 . This is a small VHDL package (just over 100 lines) that contains definitions of the constants, types, and attributes used in a BSDL description. It does not contain any models for simulation.
- Lines 22 - 25 attach signal names to the required TAP pins and the optional fifth TAP reset signal.
- Lines 26 - 27 refer to the maximum test clock frequency in hertz, and whether the clock may be stopped in both states or just the low state (just the high state is not valid).
- Line 28 describes a 3-bit IR (in the comparator/MUX example we used a 2-bit IR). Length must be greater than or equal to 2.
- Lines 29 - 35 describe the three required instruction opcodes and mnemonics (BYPASS, SAMPLE, EXTEST) and three optional instructions: IDCODE, STCTEST (which is a scan test mode), and INTEST (which supports internal testing in the same fashion as EXTEST supports external testing). EXTEST must be all ones; BYPASS must be all zeros. A mnemonic may have more than one opcode (and opcodes may be specified using 'x'). Other instructions that may appear here include CLAMP and HIGHZ , both optional instructions that were added to 1149.1 (see Supplement A, 1149.1a). String concatenation is used in BSDL to avoid line-break problems.
- Lines 37 - 39 include instruction attributes INSTRUCTION_DISABLE (for HIGHZ), INSTRUCTION_GUARD (for CLAMP), as well as INSTRUCTION_PRIVATE (for user-defined instructions) that are not used in this example.
- Lines 40 - 44 describe the IDCODE TDR. The 11-bit manufacturer number is determined from codes assigned by JEDEC Publication 106-A.
- Line 45 describes the USERCODE TDR in a similar fashion to IDCODE, but is not used here.
- Lines 46 - 47 describe the TDRs for user-defined instructions. In this case the existing BOUNDARY TDR is inserted between TDI and TDO during STCTEST . User-defined instructions listed here may use the other existing IDCODE and BYPASS TDRs or define new TDRs.
- Lines 48 - 49 list the boundary-scan cells used in the ASIC. These may be any of the following cells defined in the 1149.1 standard and defined in the VHDL package, STD_1149_1_1994 : BC_1 (Figs. 10-18, 10-29, 10-31c, 10-31d, and 10-33c), BC_2 (Figs. 10-14, 10-30, 10-32c, 10-32d, 10-35c), BC_3 (Fig. 10-15), BC_4 (Figs. 10-16, 10-17), BC_5 (Fig. 10-41c), BC_6 (Fig. 10-34d). The figure numbers in parentheses here refer to the IEEE 1149.1 standard [IEEE 1149.1b-1994]. Alternatively the cells may be user-defined (and must then be declared in a package).
- Line 50 must be an integer greater than zero and match the number defined by the following register description.
- Lines 51 - 54 are an array of records, numbered by cell, with seven fields: four required and three that only appear for certain cells. Field 1 specifies the scan cell name as defined in the STD_1149_1_1994 or user-defined package. Field 2 is the port name, with a subscript if the type is BIT_VECTOR . An '*' denotes no connection. Field 3 is one of the following cell functions (with figure or page numbers from the IEEE standard [IEEE 1149.1b-1994]): input (Fig. 10-18), clock (Fig. 10-17), output2 (two-state output, Fig. 10-29), output3 (three-state, Fig. 10-31d),

internal (p. 33, 1149.1b), control (Fig. 10-31c), controlr (Fig. 10-33c), bidir_in (a reversible cell acting as an input, Fig. 10-34d), bidir_out (a reversible cell acting as an output, Fig. 10-34d). Field 4, safe , contains the safe value to be loaded into the update flip-flop when otherwise unspecified, with 'X' as a don't care value.

- Lines 55 - 57 illustrate the use of the optional three fields. Field 5, ccell or control cell, refers to the cell number (98 in this example) of the cell that controls an output or bidirectional cell. The control cell number 98 is a merged cell in this example with an input cell, input signal name OE , also labeled as cell number 98. The ASIC input OE (for output enable) thus directly controls (enables) the ASIC three-state output, myport(0) .

The boundary-scan standard may seem like a complicated way to test the connections outside an ASIC. However, the IEEE 1149.1 standard also gives us a method to communicate with test circuits inside an ASIC. Next, we turn our attention from problems at the board level to problems that may occur within the ASIC.

---

1. Outputs: G = gated with -TCK, R = registered on falling edge of TCK. Only active levels are shown in the table.

14.3 Faults

Fabrication of an ASIC is a complicated process requiring hundreds of processing steps. Problems may introduce a defect that in turn may introduce a fault (Sabnis [1990] describes defect mechanisms). Any problem during fabrication may prevent a transistor from working and may break or join interconnections. Two common types of defects occur in metallization [Rao, 1993]: either underetching the metal (a problem between long, closely spaced lines), which results in a bridge or short circuit (shorts) between adjacent lines, or overetching the metal and causing breaks or open circuits (opens). Defects may also arise after chip fabrication is complete-while testing the wafer, cutting the die from the wafer, or mounting the die in a package. Wafer probing, wafer saw, die attach, wire bonding, and the intermediate handling steps each have their own defect and failure mechanisms. Many different materials are involved in the packaging process that have different mechanical, electrical, and thermal properties, and these differences can cause defects due to corrosion, stress, adhesion failure, cracking, and peeling. Yield loss also occurs from human error-using the wrong mask, incorrectly setting the implant dose-as well as from physical sources: contaminated chemicals, dirty etch sinks, or a troublesome process step. It is possible to repeat or rework some of the reversible steps (a lithography step, for example-but not etching) if there are problems. However, reliance on rework indicates a poorly controlled process.

14.3.1 Reliability

It is possible for defects to be nonfatal but to cause failures early in the life of a product. We call this infant mortality . Most products follow the same kinds of trend for failures as a function of life. Failure rates decrease rapidly to a low value that remains steady until the end of life when failure rates increase again; this is called a bathtub curve . The end of a product lifetime is determined by various wearout mechanisms (usually these are controlled by an exponential energy process). Some of the most important wearout mechanisms in ASICs are hot-electron wearout, electromigration, and the failure of

antifuses in FPGAs.

We can catch some of the products that are susceptible to early failure using burn-in . Many failure mechanisms have a failure rate proportional to exp (- E_a /kT). This is the Arrhenius equation , where E_a is a known activation energy (k is Boltzmann's constant, 8.62 ¥ 10^{-5} eVK^{-1} , and T the absolute temperature). Operating an ASIC at an elevated temperature accelerates this type of failure mechanism. Depending on the physics of the failure mechanism, additional stresses, such as elevated current or voltage, may also accelerate failures. The longer and harsher the burn-in conditions, the more likely we are to find problems, but the more costly the process and the more costly the parts.

We can measure the overall reliability of any product using the mean time between failures (MTBF) for a repairable product or mean time to failure (MTTF) for a fatal failure. We also use failures in time (FITs) where 1 FIT equals a single failure in 10^9 hours. We can sum the FITs for all the components in a product to determine an overall measure for the product reliability. Suppose we have a system with the following components:

- Microprocessor (standard part) 5 FITs
- 100 TTL parts, 50 parts at 10 FITs, 50 parts at 15 FITs
- 100 RAM chips, 6 FITs

The overall failure rate for this system is 5 + 50 ¥ 10 + 50 ¥ 15 + 100 ¥ 6 = 1855 FITs. Suppose we could reduce the component count using ASICs to the following:

- Microprocessor (custom) 7 FITs
- 9 ASICs, 10 FITs
- 5 SIMMs, 15 FITs

The failure rate is now 10 + 9 ¥ 10 + 5 ¥ 15 = 175 FITs, or about an order of magnitude lower. This is the rationale behind the Sun SparcStation 1 design described in Section 1.3 , " Case Study ."

14.3.2 Fault Models

Table 14.6 shows some of the causes of faults. The first column shows the fault level -whether the fault occurs in the logic gates on the chip or in the package. The second column describes the physical fault . There are too many of these and we need a way to reduce and simplify their effects-by using a fault model.

There are several types of fault model . First, we simplify things by mapping from a physical fault to a logical fault . Next, we distinguish between those logical faults that degrade the ASIC performance and those faults that are fatal and stop the ASIC from working at all. There are three kinds of logical faults in Table 14.6 : a degradation fault, an open-circuit fault, and a short-circuit fault.

TABLE 14.6 Mapping physical faults to logical faults.

		Logical fault		
Fault level	Physical fault	Degradation fault	Open-circuit fault	Short-circuit fault

Chip				
	Leakage or short between package leads	*		*
	Broken, misaligned, or poor wire bonding		*	
	Surface contamination, moisture	*		
	Metal migration, stress, peeling		*	*
	Metallization (open or short)		*	*
Gate				
	Contact opens		*	
	Gate to S/D junction short	*		*
	Field-oxide parasitic device	*		*
	Gate-oxide imperfection, spiking	*		*
	Mask misalignment	*		*

A degradation fault may be a parametric fault or delay fault (timing fault). A parametric fault might lead to an incorrect switching threshold in a TTL/CMOS level converter at an input, for example. We can test for parametric faults using a production tester. A delay fault might lead to a critical path being slower than specification. Delay faults are much harder to test in production. An open-circuit fault results from physical faults such as a bad contact, a piece of metal that is missing or overetched, or a break in a polysilicon line. These physical faults all result in failure to transmit a logic level from one part of a circuit to another-an open circuit. A short-circuit fault results from such physical faults as: underetching of metal; spiking, pinholes or shorts across the gate oxide; and diffusion shorts. These faults result in a circuit being accidentally connected-a short circuit. Most short-circuit faults occur in interconnect; often we call these bridging faults (BF). A BF usually results from metal coverage problems that lead to shorts. You may see reference to feedback bridging faults and nonfeedback bridging faults , a useful distinction when trying to predict the results of faults on logic operation. Bridging faults are a frequent problem in CMOS ICs.

14.3.3 Physical Faults

Figure 14.11 shows the following examples of physical faults in a logic cell:

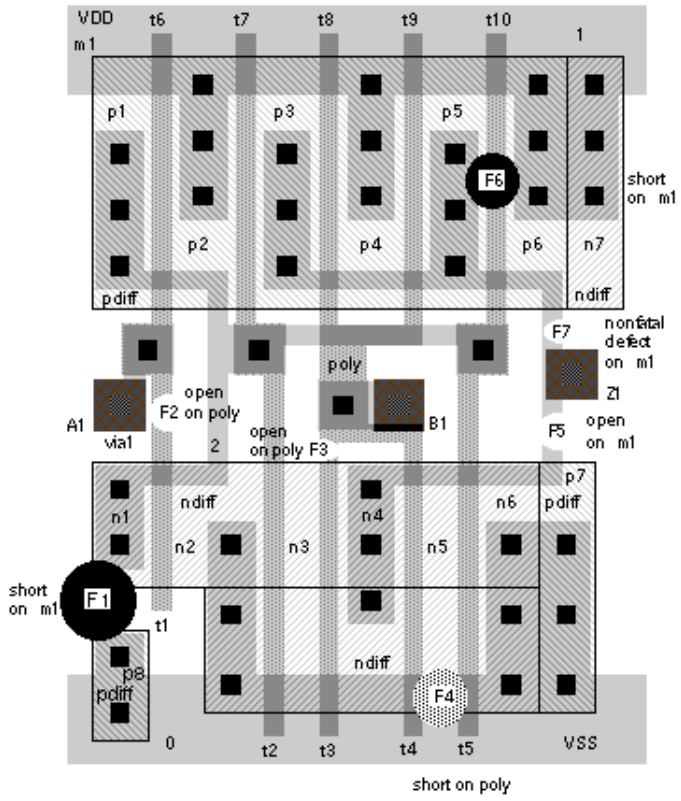


FIGURE 14.11 Defects and physical faults. Many types of defects occur during fabrication. Defects can be of any size and on any layer. Only a few small sample defects are shown here using a typical standard cell as an example. Defect density for a modern CMOS process is of the order of 1 cm $^{-2}$ or less across a whole wafer. The logic cell shown here is approximately 64 ¥ 32 l ^{2} , or 250 m m ^{2} for a l = 0.25 m m process. We would thus have to examine approximately 1 cm $^{-2}$ /250 m m ^{2} or 400,000 such logic cells to find a single defect.

- F1 is a short between m1 lines and connects node n1 to VSS.
- F2 is an open on the poly layer and disconnects the gate of transistor t1 from the rest of the circuit.
- F3 is an open on the poly layer and disconnects the gate of transistor t3 from the rest of the circuit.
- F4 is a short on the poly layer and connects the gate of transistor t4 to the gate of transistor t5.
- F5 is an open on m1 and disconnects node n4 from the output Z1.
- F6 is a short on m1 and connects nodes p5 and p6.
- F7 is a nonfatal defect that causes necking on m1.

Once we have reduced the large number of physical faults to fewer logical faults, we need a model to predict their effect. The most common model is the stuck-at fault model .

14.3.4 Stuck-at Fault Model

The single stuck-at fault (SSF) model assumes that there is just one fault in the logic we are testing. We use a single stuck-at fault model because a multiple stuck-at fault model that could handle several faults in the logic at the same time is too complicated to implement. We hope that any multiple faults are

caught by single stuck-at fault tests [Agarwal and Fung, 1981; Hughes and McCluskey, 1986]. In practice this seems to be true.

There are other fault models. For example, we can assume that faults are located in the transistors using a stuck-on fault and stuck-open fault (or stuck-off fault). Fault models such as these are more realistic in that they more closely model the actual physical faults. However, in practice the simple SSF model has been found to work-and work well. We shall concentrate on the SSF model.

In the SSF model we further assume that the effect of the physical fault (whatever it may be) is to create only two kinds of logical fault. The two types of logical faults or stuck-at faults are: a stuck-at-1 fault (abbreviated to SA1 or s@1) and a stuck-at-0 fault (SA0 or s@0). We say that we place faults (inject faults , seed faults , or apply faults) on a node (or net), on an input of a circuit, or on an output of a circuit. The location at which we place the fault is the fault origin .

A net fault forces all the logic cell inputs that the net drives to a logic ’1’ or ’0’ . An input fault attached to a logic cell input forces the logic cell input to a ’1’ or ’0’ , but does not affect other logic cell inputs on the same net. An output fault attached to the output of a logic cell can have different strengths. If an output fault is a supply-strength fault (or rail-strength fault) the logic-cell output node and every other node on that net is forced to a ’1’ or ’0’ -as if all these nodes were connected to one of the supply rails. An alternative assigns the same strength to the output fault as the drive strength of the logic cell. This allows contention between outputs on a net driving the same node. There is no standard method of handling output-fault strength , and no standard for using types of stuck-at faults. Usually we do not inject net faults; instead we inject only input faults and output faults. Some people use the term node fault -but in different ways to mean either a net fault, input fault, or output fault.

We usually inject stuck-at faults to the inputs and outputs, the pins, of logic cells (AND gates, OR gates, flip-flops, and so on). We do not inject faults to the internal nodes of a flip-flop, for example. We call this a pin-fault model and say the fault level is at the structural level , gate level, or cell level. We could apply faults to the internal logic of a logic cell (such as a flip-flop) and (the fault level would then be at the transistor level or switch level. We do not use transistor-level or switch-level fault models because there is often no need. From experience, but not from any theoretical reason, it turns out that using a fault model that applies faults at the logic-cell level is sufficient to catch the bad chips in a production test.

When a fault changes the circuit behavior, the change is called the fault effect . Fault effects travel through the circuit to other logic cells causing other fault effects. This phenomenon is fault propagation . If the fault level is at the structural level, the phenomenon is structural fault propagation . If we have one or more large functional blocks in a design, we want to apply faults to the functional blocks only at the inputs and outputs of the blocks. We do not want to place (or cannot place) faults inside the blocks, but we do want faults to propagate through the blocks. This is behavioral fault propagation .

Designers adjust the fault level to the appropriate level at which they think there may be faults. Suppose we are performing a fault simulation on a board and we have already tested the chips. Then we might set the fault level to the chip level, placing faults only at the chip pins. For ASICs we use the logic-cell level. You have to be careful, though, if you mix behavioral level and structural level models in a mixed-level fault simulation . You need to be sure that the behavioral models propagates faults correctly. In particular, if the behavioral model responds to faults on its inputs by propagating too many unknown ’X’ values to its outputs, this will decrease the fault coverage, because the model is hiding the

logic beyond it.

14.3.5 Logical Faults

Figure 14.12 and the following list show how the defects and physical faults of Figure 14.11 translate to logical faults (not all physical faults translate to logical faults-most do not):

- F1 translates to node n1 being stuck at 0, equivalent to A1 being stuck at 1.
- F2 will probably result in node n1 remaining high, equivalent to A1 being stuck at 0.
- F3 will affect half of the n -channel pull-down stack and may result in a degradation fault, depending on what happens to the floating gate of T3. The cell will still work, but the fall time at the output will approximately double. A fault such as this in the middle of a chain of logic is extremely hard to detect.
- F4 is a bridging fault whose effect depends on the relative strength of the transistors driving this node. The fault effect is not well modeled by a stuck-at fault model.
- F5 completely disables half of the n -channel pulldown stack and will result in a degradation fault.
- F6 shorts the output node to VDD and is equivalent to Z1 stuck at 1.
- Fault F7 could result in infant mortality. If this line did break due to electromigration the cell could no longer pull Z1 up to VDD. This would translate to a Z1 stuck at 0. This fault would probably be fatal and stop the ASIC working.

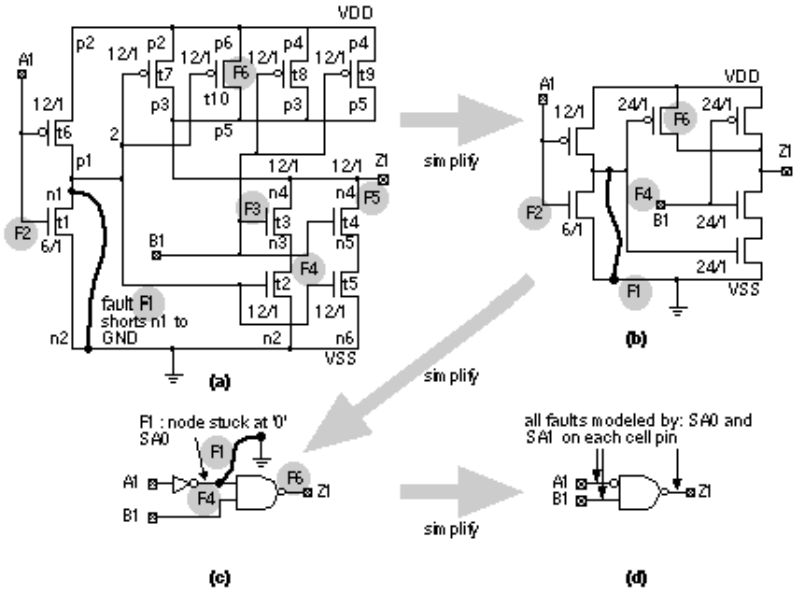


FIGURE 14.12 Fault models. (a) Physical faults at the layout level (problems during fabrication) shown in Figure 14.11 translate to electrical problems on the detailed circuit schematic. The location and effect of fault F1 is shown. The locations of the other fault examples from Figure 14.11 (F2-F6) are shown, but not their effect. (b) We can translate some of these faults to the simplified transistor schematic. (c) Only a few of the physical faults still remain in a gate-level fault model of the logic cell. (d) Finally at the functional-level fault model of a logic cell, we abandon the connection between physical and logical faults and model all faults by stuck-at faults. This is a very poor model of the physical reality, but it works well in practice.

14.3.6 IDDQ Test

When they receive a prototype ASIC, experienced designers measure the resistance between VDD and GND pins. Providing there is not a short between VDD and GND, they connect the power supplies and measure the power-supply current. From experience they know that a supply current of more than a few milliamperes indicates a bad chip. This is exactly what we want in production test: Find the bad chips quickly, get them off the tester, and save expensive tester time. An IDDQ (IDD stands for the supply current, and Q stands for quiescent) test is one of the first production tests applied to a chip on the tester, after the chip logic has been initialized [Gulati and Hawkins, 1993; Rajsuman, 1994]. High supply current can result from bridging faults that we described in Section 14.3.2 . For example, the bridging fault F4 in Figure 14.11 and Figure 14.12 would cause excessive IDDQ if node n1 and input B1 are being driven to opposite values.

14.3.7 Fault Collapsing

Figure 14.13 (a) shows a test for a stuck-at-1 output of a two-input NAND gate. Figure 14.13 (b) shows tests for other stuck-at faults. We assume that the NAND gate still works correctly in the bad circuit (also called the faulty circuit or faulty machine) even if we have an input fault. The input fault on a logic cell is presumed to arise either from a fault from a preceding logic cell or a fault on the connection to the input.

Stuck-at faults attached to different points in a circuit may produce identical fault effects. Using fault collapsing we can group these equivalent faults (or indistinguishable faults) into a fault-equivalence class . To save time we need only consider one fault, called the prime fault or representative fault , from a fault-equivalence class. For example, Figure 14.13 (a) and (b) show that a stuck-at-0 input and a stuck-at-1 output are equivalent faults for a two-input NAND gate. We only need to check for one fault, Z1 (output stuck at 1), to catch any of the equivalent faults.

Suppose that any of the tests that detect a fault B also detects fault A, but only some of the tests for fault A also detect fault B. W say A is a dominant fault , or that fault A dominates fault B (this the definition of fault dominance that we shall use, some texts say fault B dominates fault A in this situation). Clearly to reduce the number of tests using dominant fault collapsing we will pick the test for fault B. For example, Figure 14.13 (c) shows that the output stuck at 0 dominates either input stuck at 1 for a two-input NAND. By testing for fault A1, we automatically detect the fault Z1. Confusion over dominance arises because of the difference between focusing on faults (Figure 14.13 d) or test vectors (Figure 14.13 e).

Figure 14.13 (f) shows the six stuck-at faults for a two-input NAND gate. We can place SA1 or SA0 on each of the two input pins (four faults in total) and SA1 or SA0 on the output pins. Using fault equivalence (Figure 14.13 g) we can collapse six faults to four: SA1 on each input, and SA1 or SA0 on the output. Using fault dominance (Figure 14.13 h) we can collapse six faults to three. There is no way to tell the difference between equivalent faults, but if we use dominant fault collapsing we may lose information about the fault location.

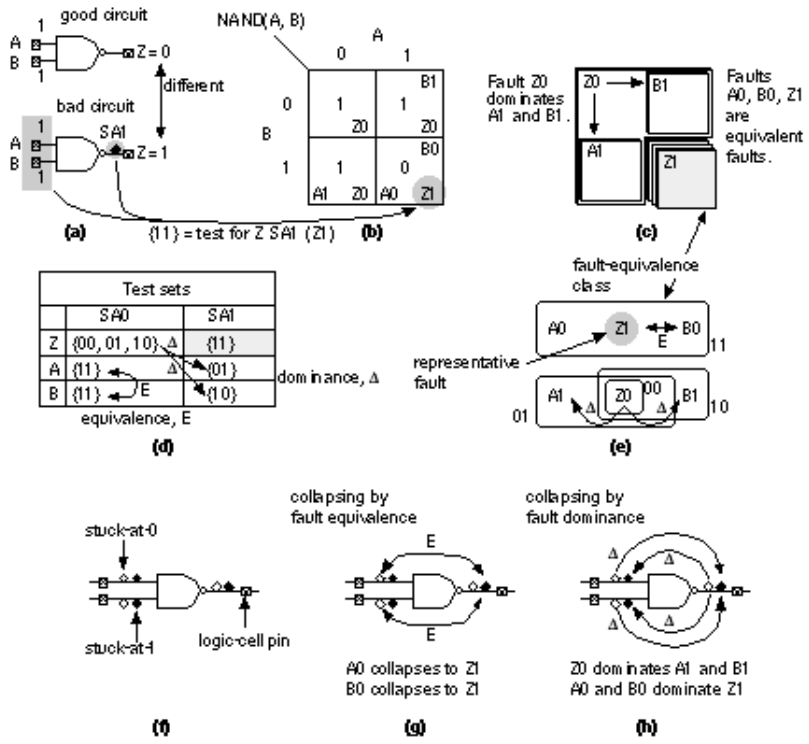


FIGURE 14.13 Fault dominance and fault equivalence. (a) We can test for fault Z0 (Z stuck at 0) by applying a test vector that makes the bad (faulty) circuit produce a different output than the good circuit. (b) Some test vectors provide tests for more than one fault. (c) A test for A stuck at 1 (A1) will also test for Z stuck at 0; Z0 dominates A1. The fault effects of faults: A0, B0 and Z1 are the same. These faults are equivalent. (d) There are six sets of input vectors that test for the six stuck-at faults. (e) We only need to choose a subset of all test vectors that test for all faults. (f) The six stuck-at faults for a two-input NAND logic cell. (g) Using fault equivalence we can collapse six faults to four. (h) Using fault dominance we can collapse six faults to three.

14.3.8 Fault-Collapsing Example

Figure 14.14 shows an example of fault collapsing. Using the properties of logic cells to reduce the number of faults that we need to consider is called gate collapsing . We can also use node collapsing by examining the effect of faults on the same node. Consider two inverters in series. An output fault on the first inverter collapses with the node fault on the net connecting the inverters. We can collapse the node fault in turn with the input fault of the second inverter. The details of fault collapsing depends on whether the simulator uses net or pin faults, the fanin and fanout of nodes, and the output fault-strength model used.

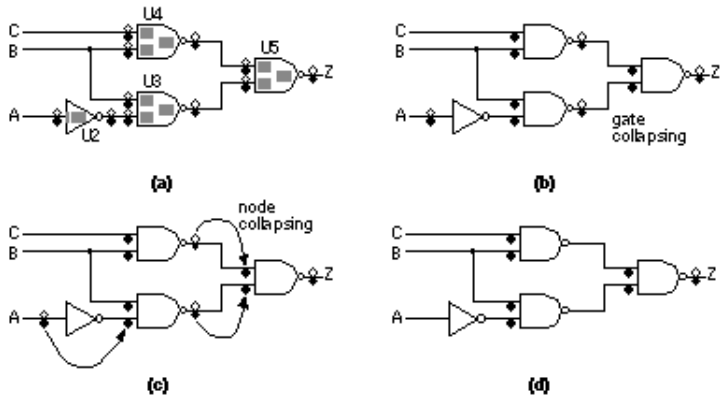


FIGURE 14.14 Fault collapsing for A'B + BC. (a) A pin-fault model. Each pin has stuck-at-0 and stuck-at-1 faults. (b) Using fault equivalence the pin faults at the input pins and output pins of logic cells are collapsed. This is gate collapsing. (c) We can reduce the number of faults we need to consider further by collapsing equivalent faults on nodes and between logic cells. This is node collapsing. (d) The final circuit has eight stuck-at faults (reduced from the 22 original faults). If we wished to use fault dominance we could also eliminate the stuck-at-0 fault on Z. Notice that in a pin-fault model we cannot collapse the faults U4.A1.SA1 and U3.A2.SA1 even though they are on the same net.

14.4 Fault Simulation

We use fault simulation after we have completed logic simulation to see what happens in a design when we deliberately introduce faults. In a production test we only have access to the package pins-the primary inputs (PIs) and primary outputs (POs). To test an ASIC we must devise a series of sets of input patterns that will detect any faults. A stimulus is the application of one such set of inputs (a test vector) to the PIs of an ASIC. A typical ASIC may have several hundred PIs and therefore each test vector is several hundred bits long. A test program consists of a set of test vectors. Typical ASIC test programs require tens of thousands and sometimes hundreds of thousands of test vectors.

The test-cycle time is the period of time the tester requires to apply the stimulus, sense the POs, and check that the actual output is equal to the expected output. Suppose the test cycle time is 100 ns (corresponding to a test frequency of 10 MHz), in which case we might sense (or strobe) the POs at 90 ns after the beginning of each test cycle. Using fault simulation we mimic the behavior of the production test. The fault simulator deliberately introduces all possible faults into our ASIC, one at a time, to see if the test program will find them. For the moment we dodge the problem of how to create the thousands of test vectors required in a typical test program and focus on fault simulation.

As each fault is inserted, the fault simulator runs our test program. If the fault simulation shows that the POs of the faulty circuit are different than the PIs of the good circuit at any strobe time, then we have a detected fault ; otherwise we have an undetected fault . The list of fault origins is collected in a file and as the faults are inserted and simulated, the results are recorded and the faults are marked according to the result. At the end of fault simulation we can find the fault coverage ,

fault coverage = detected faults / detectable faults. (14.1)

Detected faults and detectable faults will be defined in Section 14.4.5 , after the description of fault simulation. For now assume that we wish to achieve close to 100 percent fault coverage. How does fault coverage relate to the ASIC defect level?

Table 14.7 shows the results of a typical experiment to measure the relationship between single stuck-at fault coverage and AQL. Table 14.7 completes a circle with test and repair costs in Table 14.1 and defect levels in Table 14.2 . These experimental results are the only justification (but a good one) for our assumptions in adopting the SSF model. We are not quite sure why this model works so well, but, being engineers, as long as it continues to work we do not worry too much.

TABLE 14.7 Average quality level as a function of single stuck-at fault coverage.

Fault coverage	Average defect level	Average quality level (AQL)
50%	7%	93%
90%	3%	97%
95%	1%	99%
99%	0.1%	99.9%
99.9%	0.01%	99.99%

There are several algorithms for fault simulation: serial fault simulation, parallel fault simulation, and concurrent fault simulation. Next, we shall discuss each of these types of fault simulation in turn.

14.4.1 Serial Fault Simulation

Serial fault simulation is the simplest fault-simulation algorithm. We simulate two copies of the circuit, the first copy is a good circuit. We then pick a fault and insert it into the faulty circuit. In test terminology, the circuits are called machines , so the two copies are a good machine and a faulty machine . We shall continue to use the term circuit here to show the similarity between logic and fault simulation (the simulators are often the same program used in different modes). We then repeat the process, simulating one faulty circuit at a time. Serial simulation is slow and is impractical for large ASICs.

14.4.2 Parallel Fault Simulation

Parallel fault simulation takes advantage of multiple bits of the words in computer memory. In the simplest case we need only one bit to represent either a ’1’ or ’0’ for each node in the circuit. In a computer that uses a 32-bit word memory we can simulate a set of 32 copies of the circuit at the same time. One copy is the good circuit, and we insert different faults into the other copies. When we need to perform a logic operation, to model an AND gate for example, we can perform the operation across all bits in the word simultaneously. In this case, using one bit per node on a 32-bit machine, we would expect parallel fault simulation to be about 32 times faster than serial simulation. The number of bits per node that we need in order to simulate each circuit depends on the number of states in the logic system we are using. Thus, if we use a four-state system with ’1’ , ’0’ , ’X’ (unknown), and ’Z’ (high-impedance) states, we need two bits per node.

Parallel fault simulation is not quite as fast as our simple prediction because we have to simulate all the

circuits in parallel until the last fault in the current set is detected. If we use serial simulation we can stop as soon as a fault is detected and then start another fault simulation. Parallel fault simulation is faster than serial fault simulation but not as fast as concurrent fault simulation. It is also difficult to include behavioral models using parallel fault simulation.

14.4.3 Concurrent Fault Simulation

Concurrent fault simulation is the most widely used fault-simulation algorithm and takes advantage of the fact that a fault does not affect the whole circuit. Thus we do not need to simulate the whole circuit for each new fault. In concurrent simulation we first completely simulate the good circuit. We then inject a fault and resimulate a copy of only that part of the circuit that behaves differently (this is the diverged circuit). For example, if the fault is in an inverter that is at a primary output, only the inverter needs to be simulated-we can remove everything preceding the inverter.

Keeping track of exactly which parts of the circuit need to be diverged for each new fault is complicated, but the savings in memory and processing that result allow hundreds of faults to be simulated concurrently. Concurrent simulation is split into several chunks, you can usually control how many faults (usually around 100) are simulated in each chunk or pass . Each pass thus consists of a series of test cycles. Every circuit has a unique fault-activity signature that governs the divergence that occurs with different test vectors. Thus every circuit has a different optimum setting for faults per pass . Too few faults per pass will not use resources efficiently. Too many faults per pass will overflow the memory.

14.4.4 Nondeterministic Fault Simulation

Serial, parallel, and concurrent fault-simulation algorithms are forms of deterministic fault simulation . In each of these algorithms we use a set of test vectors to simulate a circuit and discover which faults we can detect. If the fault coverage is inadequate, we modify the test vectors and repeat the fault simulation. This is a very time-consuming process.

As an alternative we give up trying to simulate every possible fault and instead, using probabilistic fault simulation , we simulate a subset or sample of the faults and extrapolate fault coverage from the sample.

In statistical fault simulation we perform a fault-free simulation and use the results to predict fault coverage. This is done by computing measures of observability and controllability at every node.

We know that a node is not stuck if we can make the node toggle-that is, change from a ’0’ to ’1’ or vice versa. A toggle test checks which nodes toggle as a result of applying test vectors and gives a statistical estimate of vector quality , a measure of faults detected per test vector. There is a strong correlation between high-quality test vectors, the vectors that will detect most faults, and the test vectors that have the highest toggle coverage . Testing for nodes toggling simply requires a single logic simulation that is much faster than complete fault simulation.

We can obtain a considerable improvement in fault simulation speed by putting the high-quality test vectors at the beginning of the simulation. The sooner we can detect faults and eliminate them from having to be considered in each simulation, the faster the simulation will progress. We take the same approach when running a production test and initially order the test vectors by their contribution to fault

coverage. This assumes that all faults are equally likely. Test engineers can then modify the test program if they discover vectors late in the test program that are efficient in detecting faulty chips.

14.4.5 Fault-Simulation Results

The output of a fault simulator separates faults into several fault categories . If we can detect a fault at a location, it is a testable fault . A testable fault must be placed on a controllable net , so that we can change the logic level at that location from ’0’ to ’1’ and from ’1’ to ’0’ . A testable fault must also be on an observable net , so that we can see the effect of the fault at a PO. This means that uncontrollable nets and unobservable nets result in faults we cannot detect. We call these faults untested faults , untestable faults , or impossible faults .

If a PO of the good circuit is the opposite to that of the faulty circuit, we have a detected fault (sometimes called a hard-detected fault or a definitely detected fault). If the POs of the good circuit and faulty circuit are identical, we have an undetected fault . If a PO of the good circuit is a ’1’ or a ’0’ but the corresponding PO of the faulty circuit is an ’X’ (unknown, either ’0’ or ’1’), we have a possibly detected fault (also called a possible-detected fault , potential fault , or potentially detected fault).

If the PO of the good circuit changes between a ’1’ and a ’0’ while the faulty circuit remains at ’X’ , then we have a soft-detected fault . Soft-detected faults are a subset of possibly detected faults. Some simulators keep track of these soft-detected faults separately. Soft-detected faults are likely to be detected on a real tester if this sequence occurs often. Most fault simulators allow you to set a fault-drop threshold so that the simulator will remove faults from further consideration after soft-detecting or possibly detecting them a specified number of times. This is called fault dropping (or fault discarding). The more often a fault is possibly detected, the more likely it is to be detected on a real tester.

A redundant fault is a fault that makes no difference to the circuit operation. A combinational circuit with no such faults is irredundant . There are close links between logic-synthesis algorithms and redundancy. Logic-synthesis algorithms can produce combinational logic that is irredundant and 100 % testable for single stuck-at faults by removing redundant logic as part of logic minimization.

If a fault causes a circuit to oscillate, it is an oscillatory fault . Oscillation can occur within feedback loops in combinational circuits with zero-delay models. A fault that affects a larger than normal portion of the circuit is a hyperactive fault . Fault simulators have settings to prevent such faults from using excessive amounts of computation time. It is very annoying to run a fault simulation for several days only to discover that the entire time was taken up by simulating a single fault in a RS flip-flop or on the clock net, for example. Figure 14.15 shows some examples of fault categories.

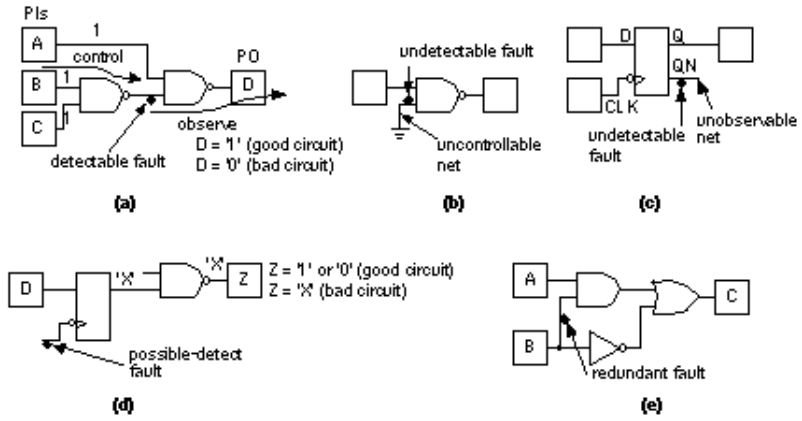


FIGURE 14.15 Fault categories. (a) A detectable fault requires the ability to control and observe the fault origin. (b) A net that is fixed in value is uncontrollable and therefore will produce one undetected fault. (c) Any net that is unconnected is unobservable and will produce undetected faults. (d) A net that produces an unknown ’X’ in the faulty circuit and a ’1’ or a ’0’ in the good circuit may be detected (depending on whether the ’X’ is in fact a ’0’ or ’1’), but we cannot say for sure. At some point this type of fault is likely to produce a discrepancy between good and bad circuits and will eventually be detected. (e) A redundant fault does not affect the operation of the good circuit. In this case the AND gate is redundant since AB + B’ = A + B’.

14.4.6 Fault-Simulator Logic Systems

In addition to the way the fault simulator counts faults in various fault categories, the number of detected faults during fault simulation also depends on the logic system used by the fault simulator. As an example, Cadence’s VeriFault concurrent fault simulator uses a logic system with the six logic values: ’0’ , ’1’ , ’Z’ , ’L’ , ’H’ , ’X’ . Table 14.8 shows the results of comparing the faulty and the good circuit simulations.

From Table 14.8 we can deduce that, in this logic system:

- Fault detection is possible only if the good circuit and the bad circuit both produce either a ’1’ or a ’0’ .
- If the good circuit produces a ’Z’ at a three-state output, no faults can be detected (not even a fault on the three-state output).
- If the good circuit produces anything other than a ’1’ or ’0’ , no faults can be detected.

A fault simulator assigns faults to each of the categories we have described. We define the fault coverage as:

fault coverage = detected faults / detectable faults. (14.2)

The number of detectable faults excludes any undetectable fault categories (untestable or redundant faults). Thus,

detectable faults = faults - undetectable faults, (14.3)

undetectable faults = untested faults + redundant faults. (14.4)

The fault simulator may also produce an analysis of fault grading . This is a graph, histogram, or tabular listing showing the cumulative fault coverage as a function of the number of test vectors. This information is useful to remove dead test cycles , which contain vectors that do not add to fault coverage. If you reinitialize the circuit at regular intervals, you can remove vectors up to an initialization without altering the function of any vectors after the initialization. The list of faults that the simulator inserted is the fault list. In addition to the fault list, a fault dictionary lists the faults with their corresponding primary outputs (the faulty output vector). The set of input vectors and faulty output vectors that uniquely identify a fault is the fault signature . This information can be useful to test engineers, allowing them to work backward from production test results and pinpoint the cause of a problem if several ASICs fail on the tester for the same reasons.

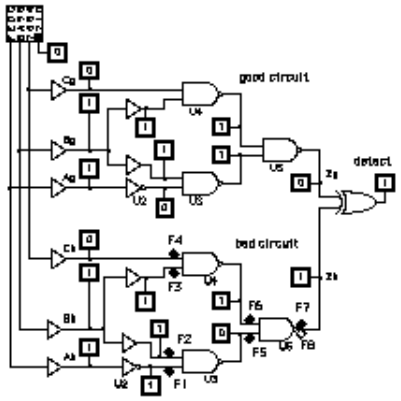
TABLE 14.8 The VeriFault concurrent fault simulator logic system. 1

		Faulty circuit					
		0	1	Z	L	H	X
Good circuit	0	U	D	P	P	P	P
	1	D	U	P	P	P	P
	Z	U	U	U	U	U	U
	L	U	U	U	U	U	U
	H	U	U	U	U	U	U
	X	U	U	U	U	U	U

14.4.7 Hardware Acceleration

Simulation engines or hardware accelerators use computer architectures that are tuned to fault-simulation algorithms. These special computers allow you to add multiple simulation boards in one chassis. Since each board is essentially a workstation produced in relatively low volume and there are between 2 and 10 boards in one accelerator, these machines are between one and two orders of magnitude more expensive than a workstation. There are two ways to use multiple boards for fault simulation. One method runs good circuits on each board in parallel with the same stimulus and generates faulty circuits concurrently with other boards. The acceleration factor is less than the number of boards because of overhead. This method is usually faster than distributing a good circuit across multiple boards. Some fault simulators allow you to use multiple circuits across multiple machines on a network in distributed fault simulation .

Fault	Type	[1] Vectors (hex)	Good output	Bad output
F1	SA1	3	0	1
F2	SA1	0, 4	0, 0	1, 1



F2	SA1	0, 4	0, 0	1, 1
F3	SA1	4, 5	0, 0	1, 1
F4	SA1	3	0	1
F5	SA1	2	1	0
F6	SA1	7	1	0
F7	SA1	0, 1, 3, 4, 5	0, 0, 0, 0, 0	1, 1, 1, 1, 1
F8	SA0	2, 6, 7	1, 1, 1	0, 0, 0

[1] Test vector format:

3 = 011, so that CBA = 011: C = ’0’, B = ’1’, A = ’1’

FIGURE 14.16 Fault simulation of A’B + BC. The simulation results for fault F1 (U2 output stuck at 1) with test vector value hex 3 (shown in bold in the table) are shown on the LogicWorks schematic. Notice that the output of U2 is 0 in the good circuit and stuck at 1 in the bad circuit.

14.4.8 A Fault-Simulation Example

Figure 14.16 illustrates fault simulation using the circuit of Figure 14.14 . We have used all possible inputs as a test vector set in the following order: {000, 001, 010, 011, 100, 101, 110, 111} . There are eight collapsed SSFs in this circuit, F1-F8. Since the good circuit is irredundant, we have 100 percent fault coverage. The following fault-simulation results were derived from a logic simulator rather than a fault simulator, but are presented in the same format as output from an automated test system.

Total number of faults: 22

Number of faults in collapsed fault list: 8

Test Vector Faults detected Coverage/% Cumulative/%

---------- --------------- ---------- ------------

000 F2, F7 25.0 25.0

001 F7 12.5 25.0

010 F5, F8 25.0 62.5

011 F1, F4, F7 37.5 75.0

100 F2, F3, F7 37.5 87.5

101 F3, F7 25.0 87.5

110 F8 12.5 100.0

111 F6, F8 25.0 100.0

Total number of vectors : 8

Noncollapsed Collapsed

Fault counts:

Detected 16 8

Untested 0 0

------ ------

Detectable 16 8

Redundant 0 0

Tied 0 0

FAULT COVERAGE 100.00 % 100.00 %

Fault simulation tells us that we need to apply seven test vectors in order to achieve full fault coverage. The highest-quality test vectors are {011} and {100} . For example, test vector {011} detects three faults (F1, F4, and F7) out of eight. This means if we were to reduce the test set to just {011} the fault coverage would be 3/8, or 37 percent. Proceeding in this fashion we reorder the test vectors in terms of their contribution to cumulative test coverage as follows: {011, 100, 010, 111, 000, 001, 101, 110} . This is a hard problem for large numbers of test vectors because of the interdependencies between the faults detected by the different vectors. Repeating the fault simulation gives the following fault grading:

Test Vector Faults detected Coverage/% Cumulative/%

---------- --------------- ---------- ------------

011 F1, F4, F7 37.5 37.5

100 F2, F3, F7 37.5 62.5

010 F5, F8 25.0 87.5

111 F6, F8 25.0 100.0

000 F2, F7 25.0 100.0

001 F7 12.5 100.0

101 F3, F7 25.0 100.0

110 F8 12.5 100.0

Now, instead of using seven test vectors, we need only apply the first four vectors from this set to achieve 100 percent fault coverage, cutting the expensive production test time nearly in half. Reducing the number of test vectors in this fashion is called test-vector compression or test-vector compaction .

The fault signatures for faults F1-F8 for the last test sequence, {011, 100, 010, 111, 000, 001, 101, 110} , are as follows:

fail good bad

-- ---- ---- -------- --------

F1 10000000 00110001 10110001

F2 01001000 00110001 01111001

F3 01000010 00110001 01110011

F4 10000000 00110001 10110001

F5 00100000 00110001 00010001

F6 00010000 00110001 00100001

F7 11001110 00110001 11111111

F8 00110001 00110001 00000000

The first pattern for each fault indicates which test vectors will fail on the tester (we say a test vector fails when it successfully detects a faulty circuit during a production test). Thus, for fault F1, pattern ’10000000’ indicates that only the first test vector will fail if fault F1 is present. The second and third patterns for each fault are the POs of the good and bad circuits for each test vector. Since we only have one PO in our simple example, these patterns do not help further distinguish between faults. Notice, that as far as an external view is concerned, faults F1 and F4 have identical fault signatures and are therefore indistinguishable. Faults F1 and F4 are said to be structurally equivalent . In general, we cannot detect structural equivalence by looking at the circuit. If we apply only the first four test vectors, then faults F2 and F3 also have identical fault signatures. Fault signatures are only useful in diagnosing fault locations if we have one, or a very few faults.

Not all fault simulators give all the information we have described. Most fault simulators drop hard-detected faults from consideration once they are detected to increase the speed of simulation. With dropped hard-detected faults we cannot independently grade each vector and we cannot construct a fault dictionary. This is the reason we used a logic simulator to generate the preceding results.

14.4.9 Fault Simulation in an ASIC Design Flow

At the beginning of this section we dodged the issue of test-vector generation. It is possible to automatically generate test vectors and test programs (with certain restrictions), and we shall discuss these methods in Section 14.5 . A by-product of some of these automated systems is a measure of fault coverage. However, fault simulation is still used for the following reasons:

- Test-generation software is expensive, and many designers still create test programs manually and then grade the test vectors using fault simulation.
- Automatic test programs are not yet at the stage where fault simulation can be completely omitted in an ASIC design flow. Usually we need fault simulation to add some vectors to test logic not covered automatically, to check that test logic has been inserted correctly, or to understand and correct fault coverage problems.
- It is far too expensive to use a production tester to debug a production test. One use of a fault simulator is to perform this function off line.
- The reuse and automatic generation of large cells is essential to decrease the complexity of large ASIC designs. Megacells and embedded blocks (an embedded microcontroller, for example) are normally provided with canned test vectors that have already been fault simulated and fault graded. The megacell has to be isolated during test to apply these vectors and measure the response. Cell compilers for RAM, ROM, multipliers, and other regular structures may also generate test vectors. Fault simulation is one way to check that the various embedded blocks and their vectors have been correctly glued together with the rest of the ASIC to produce a complete set of test vectors and a test program.
- Production testers are very expensive. There is a trend away from the use of test vectors to include more of the test function on an ASIC. Some internal test logic structures generate test vectors in a random or pseudorandom fashion. For these structures there is no known way to generate the fault coverage. For these types of test structures we will need some type of fault simulation to measure fault coverage and estimate defect levels.

---

1. L = 0 or Z; H = 1 or Z; Z = high impedance; X = unknown; D = detected; P = potentially detected; U = undetected.

14.5 Automatic Test-Pattern Generation

In this section we shall describe a widely used algorithm, PODEM, for automatic test-pattern generation (ATPG) or automatic test-vector generation (ATVG). Before we can explain the PODEM algorithm we need to develop a shorthand notation and explain some terms and definitions using a simpler ATPG algorithm.

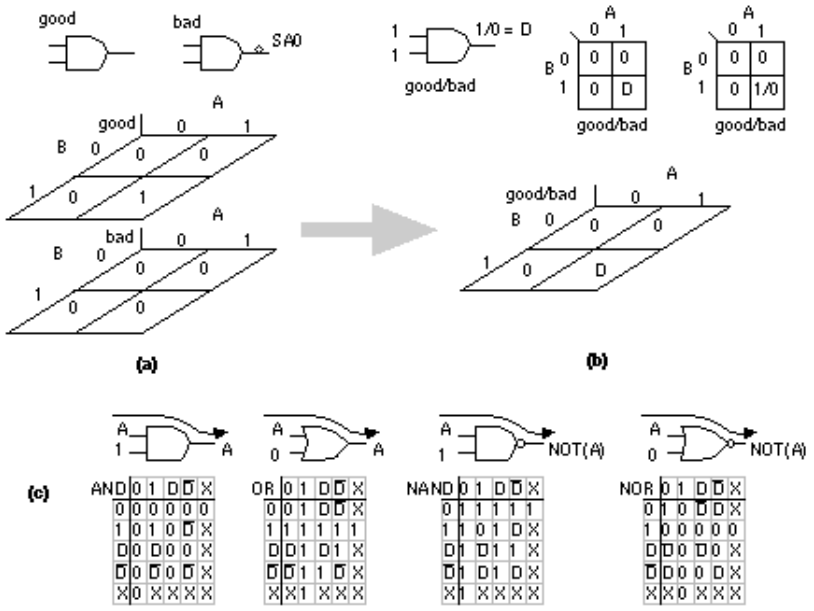


FIGURE 14.17 The D-calculus. (a) We need a way to represent the behavior of the good circuit and the bad circuit at the same time. (b) The composite logic value D (for detect) represents a logic ’1’ in the good circuit and a logic ’0’ in the bad circuit. We can also write this as D = 1/0. (c) The logic behavior of simple logic cells using the D-calculus. Composite logic values can propagate through simple logic gates if the other inputs are set to their enabling values.

14.5.1 The D-Calculus

Figure 14.17 (a) and (b) shows a shorthand notation, the D-calculus , for tracing faults. The D-calculus was developed by Roth [1966] together with an ATPG algorithm, the D-algorithm . The symbol D (for detect) indicates the value of a node is a logic ’0’ in the good circuit and a logic ’1’ in the bad circuit. We can also write this as D = 0/1. In general we write g/b, a composite logic value , to indicate a node value in the good circuit is g and b in the bad circuit (by convention we always write the good circuit value first and the faulty circuit value second). The complement of D is D = 1/0 (D is rarely written as D’ since D is a logic value just like ’1’ and ’0’). Notice that D does not mean not detected, but simply that we see a ’0’ in the good circuit and a ’1’ in the bad circuit. We can apply Boolean algebra to the composite logic values D and D as shown in Figure 14.17 (c). The composite values 1/1 and 0/0 are equivalent to ’1’ and ’0’ respectively. We use the unknown logic value ’X’ to represent a logic value that is one of ’0’, ’1’, D, or D , but we do not know or care which.

If we wish to propagate a signal from one or more inputs of a logic cell to the logic cell output, we set the remaining inputs of that logic cell to what we call the enabling value . The enabling value is ’1’ for AND and NAND gates and ’0’ for OR and NOR gates. Figure 14.17 (c) illustrates the use of enabling values. In contrast, setting at least one input of a logic gate to the controlling value , the opposite of the enabling value for that gate, forces or justifies the output node of that logic gate to a fixed value. The controlling value of ’0’ for an AND gate justifies the output to ’0’ and for a NAND gate justifies the output to ’1’. The controlling values of ’1’ justifies the output of an OR gate to ’1’ and justifies the output of a NOR gate to ’0’. To find controlling and enabling values for more complex logic cells, such as AOI and OAI logic cells, we can use their simpler AND, OR, NAND, and NOR gate representations.

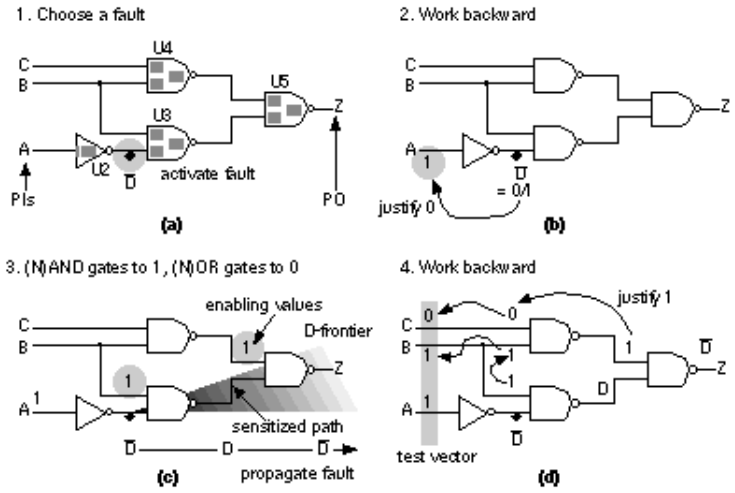


FIGURE 14.18 A basic ATPG (automatic test-pattern generation) algorithm for A'B + BC. (a) We activate a fault, U2.ZN stuck at 1, by setting the pin or node to '0', the opposite value of the fault. (b) We work backward from the fault origin to the PIs (primary inputs) by recursively justifying signals at the output of logic cells. (c) We then work forward from the fault origin to a PO (primary output), setting inputs to gates on a sensitized path to their enabling values. We propagate the fault until the D-frontier reaches a PO. (d) We then work backward from the PO to the PIs recursively justifying outputs to generate the sensitized path. This simple algorithm always works, providing signals do not branch out and then rejoin again.

14.5.2 A Basic ATPG Algorithm

A basic algorithm to generate test vectors automatically is shown in Figure 14.18 . We detect a fault by first activating (or exciting the fault). To do this we must drive the faulty node to the opposite value of the fault. Figure 14.18 (a) shows a stuck-at-1 fault at the output pin, ZN, of the inverter U2 (we call this fault U2.ZN.SA1). To create a test for U2.ZN.SA1 we have to find the values of the PIs that will justify node U2.ZN to '0' . We work backward from node U2.ZN justifying each logic gate output until we reach a PI. In this case we only have to justify U2.ZN to '0' , and this is easily done by setting the PI A = '0'. Next we work forward from the fault origin and sensitize a path to a PO (there is only one PO in this example). This propagates the fault effect to the PO so that it may be observed . To propagate the fault effect to the PO Z, we set U3.A2 = '1' and then U5.A2 = '1'.

We can visualize fault propagation by supposing that we set all nodes in a circuit to unknown, 'X'. Then, as we successively propagate the fault effect toward the POs, we can imagine a wave of D's and D 's, called the D-frontier , that propagates from the fault origin toward the POs. As a value of D or D reaches the inputs of a logic cell whose other inputs are 'X', we add that logic cell to the D-frontier. Then we find values for the other inputs to propagate the D-frontier through the logic cell to continue the process.

This basic algorithm of justifying and then propagating a fault works when we can justify nodes without interference from other nodes. This algorithm breaks down when we have reconvergent fanout .

Figure 14.19 (a) shows another example of justifying and propagating a fault in a circuit with reconvergent fanout. For direct comparison Figure 14.19 (b) shows an irredundant circuit, similar to part (a), except the fault signal, B stuck at 1, branches and then reconverges at the inputs to gate U5. The reconvergent fanout in this new circuit breaks our basic algorithm. We now have two sensitized paths that propagate the fault effect to U5. These paths combine to produce a constant ’1’ at Z, the PO. We have a multipath sensitization problem.

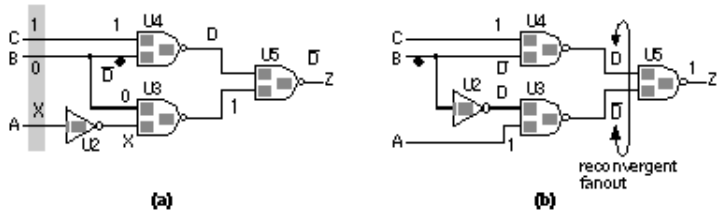


FIGURE 14.19 Reconvergent fanout. (a) Signal B branches and then reconverges at logic gate U5, but the fault U4.A1 stuck at 1 can still be excited and a path sensitized using the basic algorithm of Figure 14.18 . (b) Fault B stuck at 1 branches and then reconverges at gate U5. When we enable the inputs to both gates U3 and U4 we create two sensitized paths that prevent the fault from propagating to the PO (primary output). We can solve this problem by changing A to ’0’, but this breaks the rules of the algorithm illustrated in Figure 14.18 . The PODEM algorithm solves this problem.

14.5.3 The PODEM Algorithm

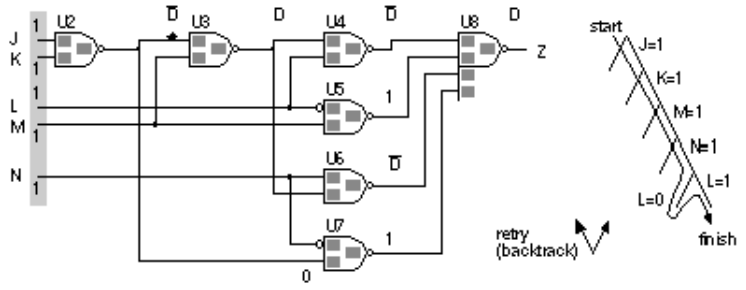
The path-oriented decision making (PODEM) algorithm solves the problem of reconvergent fanout and allows multipath sensitization [Goel, 1981]. The method is similar to the basic algorithm we have already described except PODEM will retry a step, reversing an incorrect decision. There are four basic steps that we label: objective , backtrace , implication , and D-frontier . These steps are as follows:

1. Pick an objective to set a node to a value. Start with the fault origin as an objective and all other nodes set to ’X’.
2. Backtrace to a PI and set it to a value that will help meet the objective.
3. Simulate the network to calculate the effect of fixing the value of the PI (this step is called implication). If there is no possibility of sensitizing a path to a PO, then retry by reversing the value of the PI that was set in step 2 and simulate again.
4. Update the D-frontier and return to step 1. Stop if the D-frontier reaches a PO.

Figure 14.20 shows an example that uses the following iterations of the four steps in the PODEM algorithm:

1. We start with activation of the fault as our objective, U3.A2 = ’0’. We backtrace to J. We set J = ’1’. Since K is still ’X’, implication gives us no further information. We have no D-frontier to update.
2. The objective is unchanged, but this time we backtrace to K. We set K = ’1’. Implication gives us U2.ZN = ’1’ (since now J = ’1’ and K = ’1’) and therefore U7.ZN = ’1’. We still have no D-frontier to update.

3. We set U3.A1 = ’1’ as our objective in order to propagate the fault through U3. We backtrace to M. We set M = ’1’. Implication gives us U2.ZN = ’1’ and U3.ZN = D. We update the D-frontier to reflect that U4.A2 = D and U6.A1 = D, so the D-frontier is U4 and U6.
4. We pick U6.A2 = ’1’ as an objective in order to propagate the fault through U6. We backtrace to N. We set N = ’1’. Implication gives us U6.ZN = D . We update the D-frontier to reflect that U4.A2 = D and U8.A1 = D , so the D-frontier is U4 and U8.
5. We pick U8.A1 = ’1’ as an objective in order to propagate the fault through U8. We backtrace to L. We set L = ’0’. Implication gives us U5.ZN = ’0’ and therefore U8.ZN = ’0’ (this node is Z, the PO). There is then no possible sensitized path to the PO Z. We must have made an incorrect decision, we retry and set L = ’1’. Implication now gives us U8.ZN = D and we have propagated the D-frontier to a PO.



Iteration	Objective	Backtrace [1]	Implication	D-frontier
1	U3.A2 = 0	J = 1		
2	U3.A2 = 0	K = 1	U7.ZN = 1	
3	U3.A1 = 1	M = 1	U3.ZN = D	U4, U6
4	U6.A2 = 1	N = 1	U6.ZN = D	U4, U8
5a	U8.A1 = 1	L = 0	U8.ZN = 1	U4, U8
5b	Retry	L = 1	U8.ZN = D	A

[1] Backtrace is not the same as retry or backtrack.

FIGURE 14.20 The PODEM (path-oriented decision making) algorithm.

We can see that the PODEM algorithm proceeds in two phases. In the first phase, iterations 1 and 2 in Figure 14.20 , the objective is fixed in order to activate the fault. In the second phase, iterations 3-5, the objective changes in order to propagate the fault. In step 3 of the PODEM algorithm there must be at least one path containing unknown values between the gates of the D-frontier and a PO in order to be able to complete a sensitized path to a PO. This is called the X-path check .

You may wonder why there has been no explanation of the backtrace mechanism or how to decide a value for a PI in step 2 of the PODEM algorithm. The decision tree shown in Figure 14.20 shows that it does not matter. PODEM conducts an implicit binary search over all the PIs. If we make an incorrect decision and assign the wrong value to a PI at some step, we will simply need to retry that step. Texts,

programs, and articles use the term backtrace as we have described it, but then most use the term backtrack to describe what we have called a retry, which can be confusing. I also did not explain how to choose the objective in step 1 of the PODEM algorithm. The initial objective is to activate the fault. Subsequently we select a logic gate from the D-frontier and set one of its inputs to the enabling value in an attempt to propagate the fault.

We can use intelligent procedures, based on controllability and observability , to guide PODEM and reduce the number of incorrect decisions. PODEM is a development of the D-algorithm, and there are several other ATPG algorithms that are developments of PODEM. One of these is FAN (fanout-oriented test generation) that removes the need to backtrace all the way to a PI, reducing the search time [Fujiwara and Shimono, 1983; Schulz, Trischler, and Sarfert, 1988]. Algorithms based on the D-algorithm, PODEM, and FAN are the basis of many commercial ATPG systems.

14.5.4 Controllability and Observability

In order for an ATPG system to provide a test for a fault on a node it must be possible to both control and observe the behavior of the node. There are both theoretical and practical issues involved in making sure that a design does not contain buried circuits that are impossible to observe and control. A software program that measures the controllability (with three l' s) and observability of nodes in a circuit is useful in conjunction with ATPG software.

There are several different measures for controllability and observability [Butler and Mercer, 1992]. We shall describe one of the first such systems called SCOAP (Sandia Controllability/Observability Analysis Program) [Goldstein, 1979]. These measures are also used by ATPG algorithms.

Combinational controllability is defined separately from sequential controllability . We also separate zero-controllability and one-controllability . For example, the combinational zero-controllability for a two-input AND gate, Y = AND (X_1 , X_2), is recursively defined in terms of the input controllability values as follows:

$$CC0\,(Y) = \min\{\, CC0\,(X_1),\, CC0\,(X_2)\,\} + 1\,. \quad (14.5)$$

We choose the minimum value of the two-input controllability values to reflect the fact that we can justify the output of an AND gate to '0' by setting any input to the control value of '0'. We then add one to this value to reflect the fact that we have passed through an additional level of logic. Incrementing the controllability measures for each level of logic represents a measure of the logic distance between two nodes.

We define the combinational one-controllability for a two-input AND gate as

$$CC1\,(Y) = CC1(X_1) + CC1\,(X_2) + 1\,. \quad (14.6)$$

This equation reflects the fact that we need to set all inputs of an AND gate to the enabling value of '1' to justify a '1' at the output. Figure 14.21 (a) illustrates these definitions.

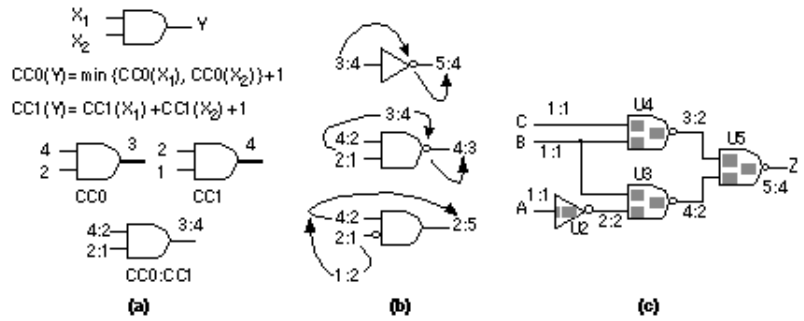


FIGURE 14.21 Controllability measures. (a) Definition of combinational zero-controllability, CC0, and combinational one-controllability, CC1, for a two-input AND gate. (b) Examples of controllability calculations for simple gates, showing intermediate steps. (c) Controllability in a combinational circuit.

An inverter, Y = NOT (X), reverses the controllability values:

$$CC1\,(Y) = CC0\,(X) + 1 \text{ and } CC0\,(Y) = CC1\,(X) + 1\,. \quad (14.7)$$

Since we can construct all other logic cells from combinations of two-input AND gates and inverters we can use Eqs. 14.5 - 14.7 to derive their controllability equations. When we do this we only increment the controllability by one for each primitive gate. Thus for a three-input NAND with an inverting input, $Y = NAND\,(X_1, X_2, NOT\,(X_3))$:

$$CC0\,(Y) = CC1\,(X_1) + CC1\,(X_2) + CC0\,(X_3) + 1\,,$$
$$CC1\,(Y) = \min\{\, CC0\,(X_1), CC0\,(X_2), CC1\,(X_3)\,\} + 1\,. \quad (14.8)$$

For a two-input NOR, $Y = NOR\,(X_1, X_2) = NOT\,(AND\,(NOT\,(X_1), NOT\,(X_2))$:

$$CC1\,(Y) = \min\{\, CC1\,(X_1), CC1\,(X_2)\,\} + 1\,,$$
$$CC0\,(Y) = CC0\,(X_1) + CC0\,(X_2) + 1\,. \quad (14.9)$$

Figure 14.21 (b) shows examples of controllability calculations. A bubble on a logic gate at the input or output swaps the values of CC1 and CC0. Figure 14.21 (c) shows how controllability values for a combinational circuit are calculated by working forward from each PI that is defined to have a controllability of one.

We define observability in terms of the controllability measures. The combinational observability , $OC\,(X_1)$, of input X_1 of a two-input AND gate can be expressed in terms of the controllability of the other input $CC1\,(X_2)$ and the combinational observability of the output, OC (Y):

$$OC\,(X_1) = CC1\,(X_2) + OC\,(Y) + 1\,. \quad (14.10)$$

If a node X_1 branches (has fanout) to nodes X_2 and X_3 we choose the most observable of the branches:

$$OC(X_1) = \min \{ O(X_2) + O(X_3) \} . \quad (14.11)$$

Figure 14.22 (a) and (b) show the definitions of observability. Figure 14.22 (c) illustrates calculation of observability at a three-input NAND; notice we sum the CC1 values for the other inputs (since the enabling value for a NAND gate is one, the same as for an AND gate). Figure 14.22 (d) shows the calculation of observability working back from the PO which, by definition, has an observability of zero.

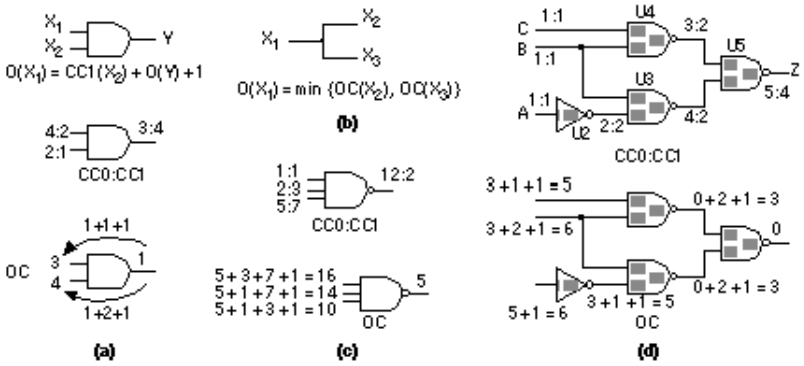


FIGURE 14.22 Observability measures. (a) The combinational observability, $OC(X_1)$, of an input, X_1, to a two-input AND gate defined in terms of the controllability of the other input and the observability of the output. (b) The observability of a fanout node is equal to the observability of the most observable branch. (c) Example of an observability calculation at a three-input NAND gate. (d) The observability of a combinational network can be calculated from the controllability measures, CC0:CC1. The observability of a PO (primary output) is defined to be zero.

Sequential controllability and observability can be measured using similar equations to the combinational measures except that in the sequential measures (SC1, SC0, and OS) we measure logic distance in terms of the layers of sequential logic, not the layers of combinational logic.

14.6 Scan Test

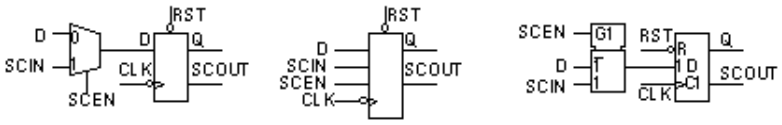
Sequential logic poses a very difficult ATPG problem. Consider the example of a 32-bit counter with a final carry. If the designer included a reset, we have to clock the counter 2^{32} (approximately 4×10^9) times to check the carry logic. Using a 1 MHz tester clock this requires 4×10^3 seconds, 1 hour, or (at approximately $0.25 per second) $1,000 of tester time. Consider a 16-bit state machine implemented using a one-hot state register with 16 D flip-flops. If the designer did not include a reset we have a very complicated initialization problem. A sequential ATPG algorithm must consider over 2000 states when constructing sequential test vectors. In an ad hoc approach to testing we could construct special reset circuits or create manual test vectors to deal with these special situations, one at a time, as they arise.

Instead we can take a structured test approach (also called design for test , though this term covers a wider field).

We can automatically generate test vectors for combinational logic, but ATPG is much harder for sequential logic. Therefore the most common sequential structured test approach converts sequential logic to combinational logic. In full-scan design we replace every sequential element with a scan flip-flop. The result is an internal form of boundary scan and, if we wish, we can use the IEEE 1149.1 TAP to access (and the boundary-scan controller to control) an internal-scan chain.

Table 14.9 shows a VHDL model and schematic symbols for a scan flip-flop. There is an area and performance penalty to pay for scan design. The scan MUX adds the delay of a 2:1 MUX to the setup time of the flip-flop; this will directly subtract from the critical path delay. The 2:1 MUX and any separate driver for the scan output also adds approximately 10 percent to the area of the flip-flop (depending on the features present in the original flip-flop). The scan chain must also be routed, and this complicates physical design and adds to the interconnect area. In ASIC design the benefits of eliminating complex sequential ATPG and the addition of observability and controllability usually outweigh these disadvantages.

TABLE 14.9 Scan flip-flop.




```
library IEEE; use IEEE.STD_LOGIC_1164. all ;

entity DFFSCAN is

generic (reset_value : STD_LOGIC := '0');

port ( Q : out STD_LOGIC ; D, CLK, RST : in STD_LOGIC;

SCOUT : out STD_LOGIC; SCIN, SCEN : in STD_LOGIC );

end DFFSCAN;

architecture behave of DFFSCAN is

signal RST_IN, CLK_IN , SCEN_IN , SCIN_IN, D_IN : STD_LOGIC ;

begin

RST_IN <= to_X01(RST); CLK_IN <= to_X01(CLK);

SCEN_IN <= to_X01(SCEN); SCIN_IN <= to_X01(SCIN); D_IN <= to_X01(D);
```

```
DFSCAN : process (CLK_IN, RST_IN) begin

if RST_IN = '0' then Q <= reset_value; SCOUT <= reset_value;

elsif RST_IN = '1' and rising_edge (CLK_IN) then

if SCEN_IN = '1' then Q <= SCIN_IN; SCOUT <= SCIN_IN;

end if ;

elsif SCEN_IN = '0' then Q <= D_IN; SCOUT <= D_IN;

else Q <= 'X' ; SCOUT <= 'X';

end if ;

elsif RST_IN = 'X' or CLK_IN = 'X' or SCEN_IN = 'X' then Q <= 'X'; SCOUT <= 'X';

end if ;

end process DFSCAN;

end behave;
```

The highly structured nature of full scan allows test software (usually called a test compiler) to perform automatic scan insertion . Using scan design we turn the output of each flip-flop into a pseudoprimary input and the input to each flip-flop into a pseudoprimary output . ATPG software can then generate test vectors for the combinational logic between scan flip-flops.

There are other approaches to scan design. In partial scan we replace a subset of the sequential elements with scan flip-flops. We can choose this subset using heuristic procedures to allow the remaining sequential logic to be tested using sequential ATPG techniques. In destructive scan we remove the values at the outputs of the flip-flops during the scan process (this is the usual form of scan design). In nondestructive scan we keep the flip-flop outputs intact so that we can shift out the scan chain and then resume where we left off. Level-sensitive scan design (LSSD) is a form of scan design developed at IBM that uses separate clock phases to drive scan elements.

14.7 Built-in Self-test

The trend to include more test logic on an ASIC has already been mentioned. Built-in self-test (BIST) is a set of structured-test techniques for combinational and sequential logic, memories, multipliers, and other embedded logic blocks. In each case the principle is to generate test vectors, apply them to the circuit under test (CUT) or device under test (DUT), and then check the response.

14.7.1 LFSR

Figure 14.23 shows a linear feedback shift register (LFSR). The exclusive-OR gates and shift register act to produce a pseudorandom binary sequence (PRBS) at each of the flip-flop outputs. By correctly choosing the points at which we take the feedback from an n -bit shift register (see Section 14.7.5), we can produce a PRBS of length $2^n - 1$, a maximal-length sequence that includes all possible patterns (or vectors) of n bits, excluding the all-zeros pattern.

FIGURE 14.23 A linear feedback shift register (LFSR). A 3-bit maximal-length LFSR produces a repeating string of seven pseudorandom binary numbers: 7, 3, 1, 4, 2, 5, 6.

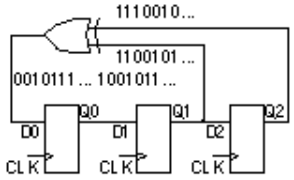


Table 14.10 shows the maximal-length sequence, with length $2^3 - 1 = 7$, for the 3-bit LFSR shown in Figure 14.23 . Notice that the first (clock tick 1) and last rows (clock tick 8) are identical. Rows following the seventh row repeat rows 1-7, so that the length of this 3-bit LFSR sequence is $7 = 2^3 - 1$, the maximal length. The shaded regions show how bits are shifted from one clock cycle to the next. We assume the register is initialized to the all-ones state, but any initial state will work and produce the same PRBS, as long as the initial state is not all zeros (in which case the LFSR will stay stuck at all zeros).

TABLE 14.10 LFSR example of Figure 14.23 .

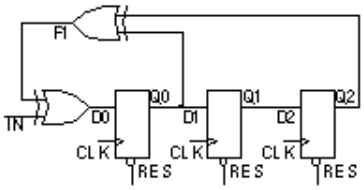
Clock tick, t =	$Q0_{t+1} = Q1_t \,?\, Q2_t$	$Q1_{t+1} = Q0_t$	$Q2_{t+1} = Q1_t$	Q0Q1Q2
1	1	1	1	7
2	0	1	1	3
3	0	0	1	1
4	1	0	0	4
5	0	1	0	2
6	1	0	1	5
7	1	1	0	6
8	1	1	1	7

14.7.2 Signature Analysis

Figure 14.24 shows the LFSR of Figure 14.23 with an additional XOR gate used in the first stage of the shift register. If we apply a binary input sequence to IN , the shift register will perform data compaction (or compression) on the input sequence. At the end of the input sequence the shift-register contents, Q0Q1Q2 , will form a pattern that we call a signature . If the input sequence and the serial-input signature register (SISR) are long enough, it is unlikely (though possible) that two different input sequences will produce the same signature. If the input sequence comes from logic that we wish to test, a fault in the logic will cause the input sequence to change. This causes the signature to change from a

known good value and we shall then know that the circuit under test is bad. This technique, called signature analysis , was developed by Hewlett-Packard to test equipment in the field in the late 1970s.

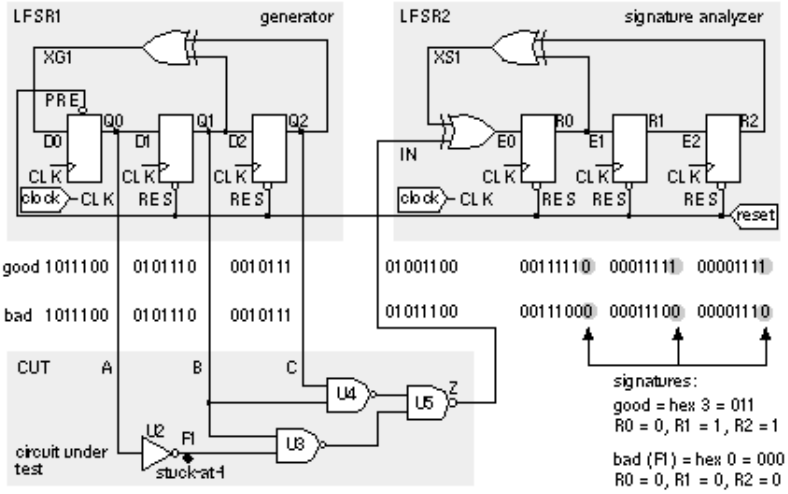
FIGURE 14.24 A 3-bit serial-input signature register (SISR) using an LFSR (linear feedback shift register). The LFSR is initialized to Q1Q2Q3 = ’000’ using the common RES (reset) signal. The signature, Q1Q2Q3, is formed from shift-and-add operations on the sequence of input bits (IN).



14.7.3 A Simple BIST Example

We can combine the PRBS generator of Figure 14.23 together with the signature register of Figure 14.24 to form the simple BIST structure shown in Figure 14.25 (a). LFSR1 generates a maximal-length ($2^3 - 1 = 7$ cycles) PRBS. LFSR2 computes the signature (’011’ for the good circuit) of the CUT. LFSR1 is initialized to ’100’ (Q0 = 1, Q1 = 0, Q2 = 0) and LFSR2 is initialized to ’000’. The schematic in Figure 14.25 (a) shows the bit sequences in the circuit, both for a good circuit and for a bad circuit with a stuck-at-1 fault, F1. Figure 14.25 (b) shows how the bit sequences are calculated in the good circuit. The signature is formed as R0R1R2 seven clock edges (on the eighth clock cycle) after the active-low reset is taken high. Figure 14.26 shows the waveforms in the good and bad circuit. The bad circuit signature, ’000’, differs from the good circuit and the signature can either be compared with the known good signature on-chip or the signature may be shifted out and compared off-chip (both approaches are used in practice).

(a)



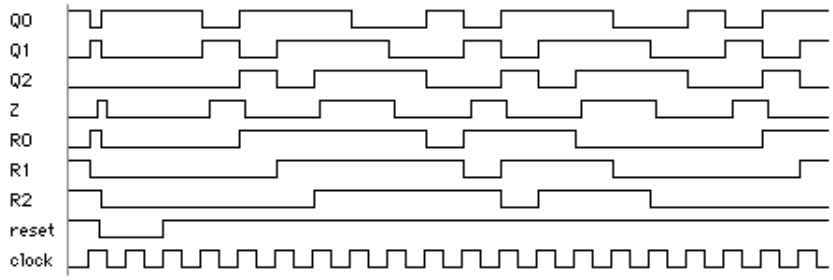
(b)

$Q0_{t+1}$ = Z = $R0_{t+1}$ =

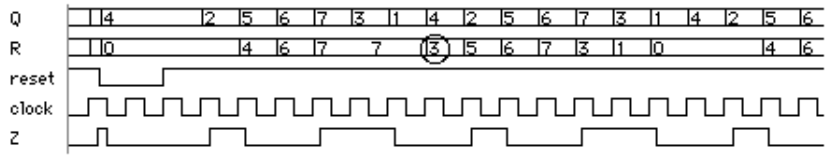
$Q1_t$? $Q2_t$	$Q1_{t+1} = Q0_t$	$Q2_{t+1} = Q1_t$	Q0'.Q1 + Q1.Q2	Z_t ? $R0_t$? $R2_t$	$R1_{t+1} = R0_t$	$R2_{t+1} = R1_t$
1	0	0	0	0	0	0
0	1	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	1	1	0
1	1	1	1	1	1	1
0	1	1	1	1	1	1
0	0	1	0	1	1	1
1	0	0	0	0	1	1

FIGURE 14.25 BIST example. (a) A simple BIST structure showing bit sequences for both good and bad circuits. (b) Bit sequence calculations for the good circuit. The signature appears on the eighth clock cycle (after seven positive clock edges) and is R0 = '0', R1 = '1', R2 = '1'; with R2 as the MSB this is '011' or hex 3.

(a)



(b)



(c)

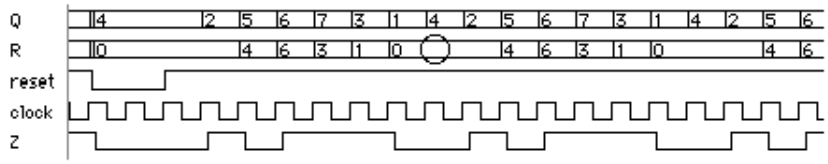


FIGURE 14.26 The waveforms of the BIST circuit of Figure 14.25 . (a) The good-circuit response.

The waveforms Q1 and Q2, as well as R1 and R2, are delayed by one clock cycle as they move through each stage of the shift registers. (b) The same good-circuit response with the register outputs Q0-Q2 and R0-R2 grouped and their values displayed in hexadecimal (Q0 and R0 are the MSBs). The signature hex 3 or '011' (R0 = 0, R1 = 1, R2 = 1) in R appears seven positive clock edges after the reset signal is taken high. This is one clock cycle after the generator completes its first sequence (hex pattern 4, 2, 5, 6, 7, 3, 1). (b) The response of the bad circuit with fault F1 and fault signature hex 0 (circled).

14.7.4 Aliasing

In Figure 14.26 the good and bad circuits produced different signatures. There is a small probability that the signature of a bad circuit will be the same as a good circuit. This problem is known as aliasing or error masking . For the example in Figure 14.25 , the bit stream input to the signature analysis register is 7 bits long. There are 2^7 or 128 possible 7-bit-long bit-stream patterns. We assume that each of these 128 bit-stream patterns is equally likely to produce any of the eight (all-zeros is an allowed pattern in a signature register) possible 3-bit signatures. It turns out that this is a good assumption. Thus there are 128 / 8 or 16 bit-streams that produce the good signature, one of these belongs to the good circuit, the remaining 15 cause aliasing. Since there are a total of 128 - 1 = 127 bit-streams due to bad circuits, the fraction of bad-circuit bit-streams that cause aliasing is 15 / 127, or 0.118. If all bad circuit bit-streams are equally likely (and this is a poor assumption) then 0.118 is also the probability of aliasing.

In general, if the length of the test sequence is L and the length of the signature register is R the probability p of aliasing (not detecting an error) is

$$p = \frac{2^{L-R} - 1}{2^{L} - 1} \qquad (14.12)$$

Thus, for the example in Figure 14.25 , L = 7 and R = 3, and the probability of aliasing is $p = (2^{(7-3)} - 1) / (2^7 - 1) = 15 / 127 = 0.118$, as we have just calculated. This is a very high probability of error and we would not use such a short test sequence and such a short signature register in practice.

For L >> R the error probability is

$$p \approx 2^{-R} \qquad (14.13)$$

For example, if R = 16, $p \approx 0.0000152$ corresponding to an error coverage (1 - p) of approximately 99.9984 percent. Unfortunately, these equations for error coverage are rather meaningless since there is no easy way to relate the error coverage to fault coverage. The problem lies in our assumption that all bad-circuit bit-streams are equally likely, and this is not true in practice (for example, bit-stream outputs of all ones or all zeros are more likely to occur as a result of faults). Nevertheless signature analysis with high error-coverage rates is found to produce high fault coverage.

14.7.5 LFSR Theory

The operation of LFSRs is related to the mathematics of polynomials and Galois-field theory. The properties and behavior of these polynomials are well known and they are also used extensively in coding theory. Every LFSR has a characteristic polynomial that describes its behavior. The characteristic polynomials that cause an LFSR to generate a maximum-length PRBS are called primitive polynomials. Consider the primitive polynomial

$$P(x) = 1 ? x^1 ? x^3 \quad , (14.14)$$

where a ? b represents the exclusive-OR of a and b . The order of this polynomial is three, and the corresponding LFSR will generate a PRBS of length $2^3 - 1 = 7$. For a primitive polynomial of order n , the length of the PRBS is 2 n - 1. Figure 14.27 shows the nonzero coefficients of some primitive polynomials [Golomb et al., 1982].

n	s	Octal	Binary
1	0, 1	3	11
2	0, 1, 2	7	111
3	0, 1, 3	13	1011
4	0, 1, 4	3	10011
5	0, 2, 5	45	100101
6	0, 1, 6	103	1000011
7	0, 1, 7	211	10001001
8	0, 1, 5, 6, 8	435	100011101
9	0, 4, 9	1021	1000010001
10	0, 3, 10	2011	10000001001

For n = 3 and s = 0, 1, 3: c 0 = 1, $c_1 = 1$, $c_2 = 0$, $c_3 = 1$

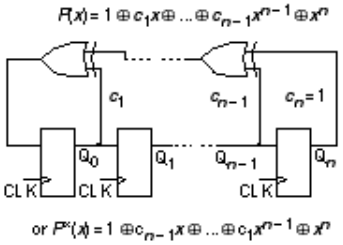


FIGURE 14.27 Primitive polynomial coefficients for LFSRs (linear feedback shift registers) that generate a maximal-length PRBS (pseudorandom binary sequence). A schematic for a type 1 LFSR is shown.

Any primitive polynomial can be written as

$$P(x) = c_0 ? c_1 x^1 ? \ldots c_n x^n \quad , (14.15)$$

where c_0 and c_n are always one. Thus for example, from Figure 14.27 for n = 3, we see s = 0, 1, 3; and thus the nonzero coefficients are c_0 , c_1 , and c_3 . This corresponds to the primitive polynomial P(x) = $1 ? x^1 ? x^3$. There is no easy way to determine the coefficients of primitive polynomials, especially for large n . There are many primitive polynomials for each n , but Figure 14.27 lists the one with the fewest nonzero coefficients.

The schematic in Figure 14.27 shows how the feedback taps on a LFSR correspond to the nonzero coefficients of the primitive polynomial. If the i th coefficient c_i is 1, then we include a feedback connection and an XOR gate in that position. If c_i is zero, there is no feedback connection and no XOR

gate in that position.

The reciprocal of a primitive polynomial, P*(x) , is also primitive, where

$$P^*(x) = x^n P^*(x^{-1}) . \quad (14.16)$$

For example, by taking the reciprocal of the primitive polynomial $P(x) = 1 ? x^1 ? x^3$ from Eq. 14.17 , we can form

$$P^*(x) = 1 ? x^3 ? x^4 , \quad (14.17)$$

which is also a primitive polynomial.

This means that there are two possible LFSR implementations for every P(x) . Or, looked at another way, for every LFSR implementation, the characteristic polynomial can be written in terms of two primitive polynomials, P(x) and P*(x) , that are reciprocals of each other.

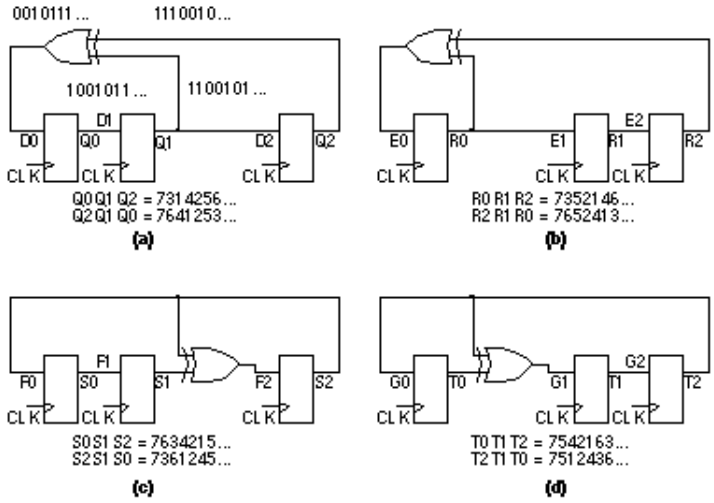


FIGURE 14.28 For every primitive polynomial there are four linear feedback shift registers (LFSRs). There are two types of LFSR; one type uses external XOR gates (type 1) and the other type uses internal XOR gates (type 2). For each type the feedback taps can be constructed either from the polynomial P(x) or from its reciprocal, P*(x). The LFSRs in this figure correspond to $P(x) = 1 ? x ? x^3$ and $P^*(x)= 1 ? x^2 ? x^3$. Each LFSR produces a different pseudorandom sequence, as shown. The binary values of the LFSR seen as a register, with the bit labeled as zero being the MSB, are shown in hexadecimal. The sequences shown are for each register initialized to '111', hex 7. (a) Type 1, P*(x). (b) Type 1, P(x). (c) Type 2, P(x). (d) Type 1, P*(x).

We may also implement an LFSR by using XOR gates in series with each flip-flop output rather than external to the shift register. The external-XOR LFSR is called a type 1 LFSR and the internal-XOR LFSR is called a type 2 LFSR (this is a nomenclature that most follow). Figure 14.28 shows the four different LFSRs that may be constructed for each primitive polynomial, P(x) .

There are differences between the four different LFSRs for each polynomial. Each gives a different output sequence. The outputs for the type 1 LFSRs, taken from the Q outputs of each flip-flop, are identical, but delayed by one clock cycle from the previous output. This is a problem when we use the parallel output from an LFSR to test logic because of the strong correlation between the test signals. The type 2 LFSRs do not have this problem. The type 2 LFSRs also are capable of higher-frequency operation since there are fewer series XOR gates in the signal path than in the corresponding type 1 LFSR. For these reasons, the type 2 LFSRs are usually used in BIST structures. The type 1 LFSR does have the advantage that it can be more easily constructed using register structures that already exist on an ASIC.

Table 14.11 shows primitive polynomial coefficients for higher values of n than Figure 14.27 . Test length grows quickly with the size of the LFSR. For example, a 32-bit generator will produce a sequence with 2^{32} = 4,294,967,296 ª 4.3 ¥ 10^{9} bits. With a 100 MHz clock (with 10 ns cycle time), the test time of 43 seconds would be impractical.

TABLE 14.11 Nonzero coefficients of primitive polynomials for LFSRs (linear feedback shift registers) that generate a maximal-length PRBS (pseudorandom binary sequence).

n	s	n	s	n	s	n	s
1	0, 1	11	0, 2, 11	21	0, 2, 21	31	0, 3, 31
2	0, 1, 2	12	0, 3, 4, 7, 12	22	0, 1, 22	32	0, 1, 27, 28, 32
3	0, 1, 3	13	0, 1, 3, 4, 13	23	0, 5, 23	40	0, 2, 19, 21, 40
4	0, 1, 4	14	0, 1, 11, 12, 14	24	0, 1, 3, 4, 24	50	0, 1, 26, 27, 50
5	0, 2, 5	15	0, 1, 15	25	0, 3, 25	60	0, 1, 60
6	0, 1, 6	16	0, 2, 3, 5, 16	26	0, 1, 7, 26	70	0, 1, 15, 16, 70
7	0, 1, 7	17	0, 3, 17	27	0, 1, 7, 27	80	0, 1, 37, 38, 80
8	0, 1, 5, 6, 8	18	0, 7, 18	28	0, 3, 28	90	0, 1, 18, 19, 90
9	0, 4, 9	19	0, 1, 5, 6, 19	29	0, 2, 29	100	0, 37, 100
10	0, 3, 10	20	0, 3, 20	30	0, 1, 15, 16, 30	256	0, 1, 3, 16, 256

There is confusion over naming, labeling, and drawing of LFSRs in texts and test programs. Looking at the schematic in Figure 14.27 , we can draw the LFSR with signals flowing from left to right or vice versa (two ways), we can name the leftmost flip-flop output Q_0 or Q_n (two more ways), and we can name the coefficient that goes with Q_0 either c_0 or c_{n-1} (two more ways). There are thus at least 2^3 ¥ 4 different ways to draw an LFSR for a given polynomial. Four of these are distinct. You can connect the LFSR feedback in the reverse order and the LFSR will still work-you will, however, get a different sequence. Usually this does not matter.

14.7.6 LFSR Example

We can use a cell compiler to produce LFSR and signature register BIST structures. For example, we might complete a property sheet as follows:

property name value property name value

---------- ----- ---------- -----

LFSR_is_bilbo false LFSR_configuration generator

LFSR_length 3 LFSR_init_hex_value 4

LFSR_scan false LFSR_mux_data false

LFSR_mux_output false LFSR_xor_hex_function max_length

LFSR_zero_state false LFSR_signature_inputs 1

The Verilog structural netlist for the compiled type 2 LFSR generator is shown in Table 14.12 . According to our notation and the primitive polynomials in Figure 14.27 , the corresponding primitive polynomial is $P^*(x) = 1 ? x^2 ? x^3$. The LFSR has both serial and parallel outputs (taken from the inverted flip-flop outputs with inverting buffers, cell names in02d1). The clock and reset inputs are buffered (with noninverting buffers, cell names ni01d1) since these inputs would normally have to drive a load of more than 3 bits. Looking in the cell data book we find that the flip-flop corresponding to the MSB, instance FF0 with cell name dfptnb , has an active-low set input SDN . The remaining flip-flops, cell name dfctnb , have active-low clears, CDN . This gives us the initial value '100'.

Table 14.13 shows the serial-input signature register compiled using the reciprocal polynomial. Again the compiler has included buffers. All the flip-flops, cell names dfctnb , have active-low clear so that the initial content of the register is '000'.

TABLE 14.12 Compiled LFSR generator, using $P^*(x) = 1 ? x^2 ? x^3$.

module lfsr_generator (OUT, SERIAL_OUT, INITN, CP);

output [2:0] OUT; **output** SERIAL_OUT; **input** INITN, CP;

dfptnb FF2 (.D(FF0_Q), .CP(u4_Z), .SDN(u2_Z), .Q(FF2_Q), .QN(FF2_QN));

dfctnb FF1 (.D(XOR0_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF1_Q), .QN(FF1_QN));

dfctnb FF0 (.D(FF1_Q), .CP(u4_Z), .CDN(u2_Z), .Q(FF0_Q), .QN(FF0_QN));

ni01d1 u2 (.I(u3_Z), .Z(u2_Z)); ni01d1 u3 (.I(INITN), .Z(u3_Z));

ni01d1 u4 (.I(u5_Z), .Z(u4_Z)); ni01d1 u5 (.I(CP), .Z(u5_Z));

xo02d1 XOR0 (.A1(FF2_Q), .A2(FF0_Q), .Z(XOR0_Z));

in02d1 INV2X0 (.I(FF0_QN), .ZN(OUT[0]));

in02d1 INV2X1 (.I(FF1_QN), .ZN(OUT[1]));

in02d1 INV2X2 (.I(FF2_QN), .ZN(OUT[2]));

in02d1 INV2X3 (.I(FF0_QN), .ZN(SERIAL_OUT));

endmodule

TABLE 14.13 Compiled serial-input signature register, using $P(x) = 1 ? x ? x^3$.

module lfsr_signature (OUT, SERIAL_OUT, INITN, CP, IN);

output [2:0] OUT; **output** SERIAL_OUT; **input** INITN, CP; **input** [0:0] IN;

dfctnb FF2 (.D(XOR1_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF2_Q), .QN(FF2_QN));

dfctnb FF1 (.D(FF2_Q), .CP(u4_Z), .CDN(u2_Z), .Q(FF1_Q), .QN(FF1_QN));

dfctnb FF0 (.D(XOR0_Z), .CP(u4_Z), .CDN(u2_Z), .Q(FF0_Q), .QN(FF0_QN));

ni01d1 u2 (.I(u3_Z), .Z(u2_Z)); ni01d1 u3 (.I(INITN), .Z(u3_Z));

ni01d1 u4 (.I(u5_Z), .Z(u4_Z)); ni01d1 u5 (.I(CP), .Z(u5_Z));

xo02d1 XOR1 (.A1(IN[0]), .A2(FF0_Q), .Z(XOR1_Z));

xo02d1 XOR0 (.A1(FF1_Q), .A2(FF0_Q), .Z(XOR0_Z));

in02d1 INV2X1 (.I(FF1_QN), .ZN(OUT[1]));

in02d1 INV2X2 (.I(FF2_QN), .ZN(OUT[2]));

in02d1 INV2X3 (.I(FF0_QN), .ZN(SERIAL_OUT));

in02d1 INV2X0 (.I(FF0_QN), .ZN(OUT[0]));

endmodule

14.7.7 MISR

A serial-input signature register can only be used to test logic with a single output. We can extend the idea of a serial-input signature register to the multiple-input signature register (MISR) shown in Figure 14.29 . There are several ways to connect the inputs to both types (type 1 and type 2) of LFSRs to form an MISR. Since the XOR operation is linear and associative, so that (A ? B) ? C = A ? (B ? C), as long as the result of the additions are the same then the different representations are equivalent. If we have an n -bit long MISR we can accommodate up to n inputs to form the signature. If we use $m < n$ inputs we do not need the extra XOR gates in the last n - m positions of the MISR.

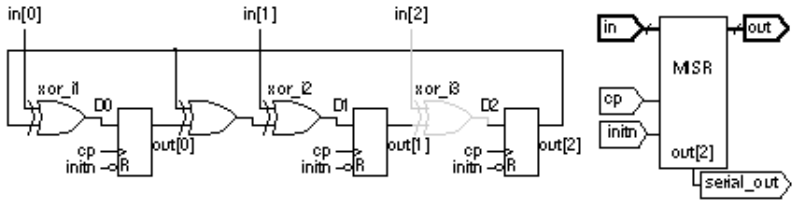


FIGURE 14.29 Multiple-input signature register (MISR). This MISR is formed from the type 2 LFSR (with $P^*(x) = 1 ? x^2 ? x^3$) shown in Figure 14.28 (d) by adding XOR gates xor_i1, xor_i2, and xor_i3. This 3-bit MISR can form a signature from logic with three outputs. If we only need to test two outputs then we do not need XOR gate, xor_i3, corresponding to input in[2].

There are several types of BIST architecture based on the MISR. By including extra logic we can reconfigure an MISR to be an LFSR or a signature register; this is called a built-in logic block observer (BILBO). By including the logic that we wish to test in the feedback path of an MISR, we can construct circular BIST structures. One of these is known as the circular self-test path (CSTP).

We can test compiled blocks including RAM, ROM, and datapath elements using an LFSR generator and a MISR. To generate all 2 n address values for a RAM or ROM we can modify the LFSR feedback path to force entrance and exit from the all-zeros state. This is known as a complete LFSR . The pattern generator does not have to be an LFSR or exhaustive.

For example, if we were to apply an exhaustive test to a 4-bit by 4-bit multiplier this would require 2^8 or 256 vectors. An 8-bit by 8-bit multiplier requires 65,536 vectors and, if it were possible to test a 32-bit by 32-bit multiplier exhaustively, it would require $1.8 ¥ 10^{19}$ vectors. Table 14.14 shows two sets of nonexhaustive test patterns, {SA} and {SAE}, if A and B are both 4 bits wide. The test sequences {SA} and {SAE} consist of nested sequences of walking 1's and walking 0's (S1 and S1B), walking pairs (S2 and S2B), and triplets (S3, S3B). The sequences are extended for larger inputs, so that, for example, {S2} is a sequence of seven vectors for an 8-bit input and so on. Intermediate sequences {SX} and {SXB} are concatenated from S1, S2, and S3; and from S1B, S2B, and S3B respectively. These sequences are chosen to exercise as many of the add-and-carry functions within the multiplier as possible.

TABLE 14.14 Multiplier test patterns. 1

Sequence {SX}	Sequence {SXB}	Sequence {SA}	Sequence {SAE}
			{ { AB = {S1, SX} }
S1= {1000 0100 0010 0001}	S1B = {0111 1011 1101 0111}		{ AB = {S1B, SXB} }
S2 = {1100 0110 0011}	S2B = {0011 1001 1100}	{	{ AB = {S2, SX} }
S3 = {1110 0111}	S3B = {0001 1000}	A B= {S1, SX}	{ AB = {S2B, SXB} }
		}	{ A B= {S3, SX} }

SX = { {S1} {S2} {S3}}	SXB = { {S1B} {S2B} {S3B} }		{ AB = {S3B, SXB} } }
Total = 3(X - 1) = 9, X = 4	Total = 3(X - 1) = 9, X = 4	Total = 4 ¥ 9 = 3A(B - 1) = 36	Total = 3(2A - 1)(3B - 2) = 3 ¥ 7 ¥ 10 = 210

The sequence length of {SA} is 3A (B - 1), and 3(2A - 1)(3B - 2) for {SAE}, where A and B are the sizes of the multiplier inputs. For example, {SA} is 168 vectors for A = B = 8 and 2976 vectors for A = B = 32; {SAE} is 990 vectors (A = B = 8) and 17,766 vectors (A = B = 32). From fault simulation, the stuck-at fault coverage is 93 percent for sequence {SA} and 97 percent for sequence {SAE}.

Figure 14.30 shows an MISR with a scan chain. We can now include the BIST logic as part of a boundary-scan chain, this approach is called scanBIST .

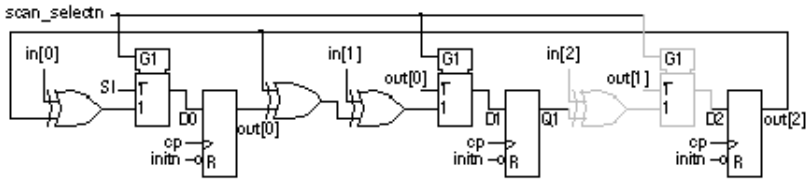


FIGURE 14.30 Multiple-input signature register (MISR) with scan generated from the MISR of Figure 14.29 .

---

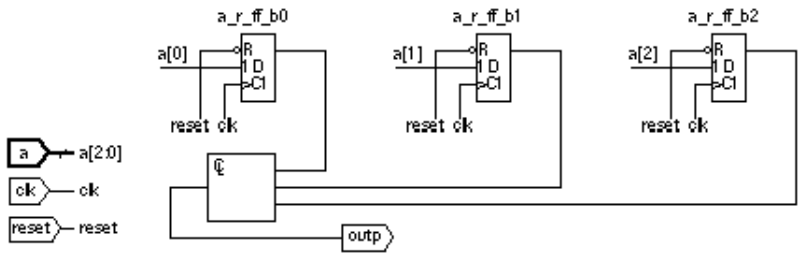
1. {AB = {S1, SB} } means for each value of A in the sequence {S1} set B equal to all the values in {SB}.

14.8 A Simple Test Example

As an example, we will describe automatic test generation using boundary scan together with internal scan. We shall use the function Z = A'B + BC for the core logic and register the three inputs using three flip-flops. We shall test the resulting sequential logic using a scan chain. The simple logic will allow us to see how the test vectors are generated.

14.8.1 Test-Logic Insertion

Figure 14.31 shows a structural Verilog model of the logic core. The three flip-flops (cell name dfctnb) implement the input register. The combinational logic implements the function, outp = a_r[0]'.a_r[1] + a_r[1].a_r[2]. This is the same function as Figure 14.14 and Figure 14.16 .




```
module core_p (outp, reset, a, clk);

output outp; input reset, clk; input [2:0] a; wire [2:0] a_r;

dfctnb a_r_ff_b0 (.D(a[0]), .CP(clk), .CDN(reset), .Q(a_r[0]), .QN(\a_r_ff_b0.QN ));

dfctnb a_r_ff_b1 (.D(a[1]), .CP(clk), .CDN(reset), .Q(a_r[1]), .QN(\a_r_ff_b1.QN ));

dfctnb a_r_ff_b2 (.D(a[2]), .CP(clk), .CDN(reset), .Q(a_r[2]), .QN(\a_r_ff_b2.QN ));

in01d0 u2 (.I(a_r[0]), .ZN(u2_ZN));

nd02d0 u3 (.A1(u2_ZN), .A2(a_r[1]), .ZN(u3_ZN));

nd02d0 u4 (.A1(a_r[1]), .A2(a_r[2]), .ZN(u4_ZN));

nd02d0 u5 (.A1(u3_ZN), .A2(u4_ZN), .ZN(outp));

endmodule
```

FIGURE 14.31 Core of the Threegates ASIC.

Table 14.15 shows the structural Verilog for the top-level logic of the Threegates ASIC including the I/O pads. There are nine pad cells. Three instances (up1_b0 , up1_b1 , and up1_b2) are the data-input pads, and one instance, up2_1 , is the output pad. These were vectorized pads (even for the output that had a range of 1), so the synthesizer has added suffixes ('_1' and so on) to the pad instance names. Two pads are for power, one each for ground and the positive supply, instances up11 and up12 . One pad, instance up3_1 , is for the reset signal. There are two pad cells for the clock. Instance up4_1 is the clock input pad attached to the package pin and instance up6 is the clock input buffer.

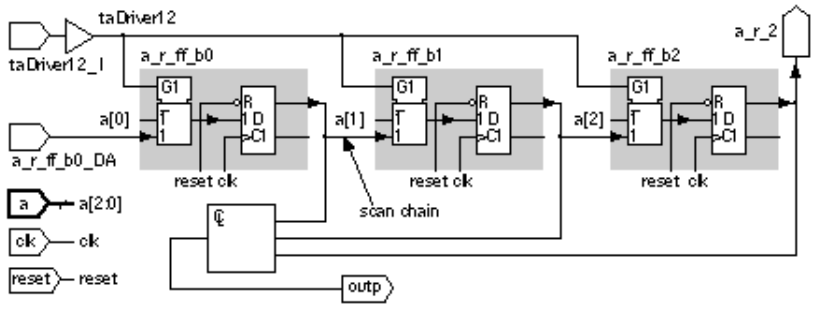
The next step is to insert the boundary-scan logic and the internal-scan logic. Some synthesis tools can create test logic as they synthesize, but for most tools we need to perform test-logic insertion as a separate step. Normally we complete a parameter sheet specifying the type of test logic (boundary scan with internal scan in this case), as well as the ordering of the scan chain. In our example, we shall include all of the sequential cells in the boundary-scan register and order the boundary-scan cells using the pad numbers (in the original behavioral input). Figure 14.32 shows the modified core logic. The test software has changed all the flip-flops (cell names dfctnb) to scan flip-flops (with the same instance names, but the cell names are changed to mfctnb). The test software also adds a noninverting buffer to

drive the scan-select signal to all the scan flip-flops.

The test software also adds logic to the top level. We do not need a detailed understanding of the automatically generated logic, but later in the design flow we will need to understand what has been done. Figure 14.33 shows a high-level view of the Threegates ASIC before and after test-logic insertion.

TABLE 14.15 The top level of the Threegates ASIC before test-logic insertion.

```
module asic_p (pad_outp, pad_a, pad_reset, pad_clk);

output [0:0] pad_outp; input [2:0] pad_a; input [0:0] pad_reset, pad_clk;

wire [0:0] reset_sv, clk_sv, outp_sv; wire [2:0] a_sv; supply1 VDD; supply0 VSS;

core_p uc1 (.outp(outp_sv[0]), .reset(reset_sv[0]), .a(a_sv[2:0]), .clk(clk_bit));

pc3o07 up2_1 (.PAD(pad_outp[0]), .I(outp_sv[0]));

pc3c01 up6 (.CCLK(clk_sv[0]), .CP(clk_bit));

pc3d01r up3_1 (.PAD(pad_reset[0]), .CIN(reset_sv[0]));

pc3d01r up4_1 (.PAD(pad_clk[0]), .CIN(clk_sv[0]));

pc3d01r up1_b0 (.PAD(pad_a[0]), .CIN(a_sv[0]));

pc3d01r up1_b1 (.PAD(pad_a[1]), .CIN(a_sv[1]));

pc3d01r up1_b2 (.PAD(pad_a[2]), .CIN(a_sv[2]));

pv0f up11 (.VSS(VSS));

pvdf up12 (.VDD(VDD));

endmodule
```




```
module core_p_ta (a_r_2, outp, a_r_ff_b0_DA, taDriver12_I, a, clk, reset);
```

output a_r_2, outp; **input** a_r_ff_b0_DA, taDriver12_I;

input [2:0] a; **input** clk, reset; **wire** [1:0] a_r; **supply1** VDD; **supply0** VSS;

ni01d5 taDriver12 (.I(taDriver12_I), .Z(taDriver12_Z));

mfctnb a_r_ff_b0 (.DA(a_r_ff_b0_DA), .DB(a[0]), .SA(taDriver12_Z), .CP(clk),

.CDN(reset), .Q(a_r[0]), .QN(\a_r_ff_b0.QN));

mfctnb a_r_ff_b1 (.DA(a_r[0]), .DB(a[1]), .SA(taDriver12_Z), .CP(clk), .CDN(reset),

.Q(a_r[1]), .QN(\a_r_ff_b1.QN));

mfctnb a_r_ff_b2 (.DA(a_r[1]), .DB(a[2]), .SA(taDriver12_Z), .CP(clk), .CDN(reset),

.Q(a_r_2), .QN(\a_r_ff_b2.QN));

in01d0 u2 (.I(a_r[0]), .ZN(u2_ZN));

nd02d0 u3 (.A1(u2_ZN), .A2(a_r[1]), .ZN(u3_ZN));

nd02d0 u4 (.A1(a_r[1]), .A2(a_r_2), .ZN(u4_ZN));

nd02d0 u5 (.A1(u3_ZN), .A2(u4_ZN), .ZN(outp));

endmodule

FIGURE 14.32 The core of the Threegates ASIC after test-logic insertion.

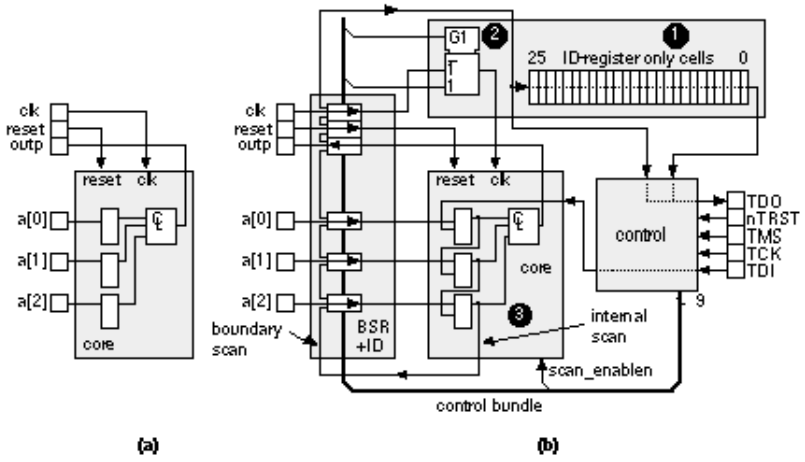


FIGURE 14.33 The Threegates ASIC. (a) Before test-logic insertion. (b) After test-logic insertion.

14.8.2 How the Test Software Works

The structural Verilog for the Threegates ASIC is lengthy, so Figure 14.34 shows only the essential parts. The following main blocks are labeled in Figure 14.34 :

1. This block is the logic core shown in Figure 14.32 . The Verilog module header shows the "local" and "formal" port names. Arrows indicate whether each signal is an input or an output.
2. This is the main body of logic added by the test software. It includes the boundary-scan controller and clock control.
3. This block groups together the buffers that the test software has added at the top level to drive the control signals throughout the boundary-scan logic.
4. This block is the first boundary-scan cell in the BSR. There are six boundary-scan cells: three input cells for the data inputs, one output cell for the data output, one input cell for the reset, and one input cell for the clock. Only the first (the boundary-scan input cell for a[0]) and the last boundary-scan cells are shown. The others are similar.
5. This is the last boundary-scan cell in the BSR, the output cell for the clock.
6. This is the clock pad (with input connected to the ASIC package pin). The cell itself is unchanged by the test software, but the connections have been altered.
7. This is the clock-buffer cell that has not been changed.
8. The test software adds five I/O pads for the TAP. Four are input pad cells for TCK, TMS, TDO, and TRST. One is a three-state output pad cell for TDO.
9. The power pad cells remain unchanged.
10. The remaining I/O pad cells for the three data inputs, the data output, and reset remain unchanged, but the connections to the core logic are broken and the boundary-scan cells inserted.

The numbers in Figure 14.34 link the signals in each block to the following explanations:

1. The control signals for the input BSCs are C_0, C_1, C_2, and C_4 and these are all buffered, together with the test clock TCK . The single output BSC also requires the control signal C_3 and this is driven from the BST controller.
2. The clock enters the ASIC package through the clock pad as .PAD(clk[0]) and exits the clock pad cell as .CIN(up_4_1_CIN1) . The test software routes this to the data input of the last boundary-scan cell as .PI(up_4_1_CIN1) and the clock exits as .PO(up_4_1_cin) . The clock then passes through the clock buffer, as before.
3. The serial input of the first boundary-scan cell comes from the controller as .bst_control_BST_SI(test_logic_bst_control_BST_SI) .
4. The serial output of the last boundary-scan cell goes to the controller as .bst_control_BST(up4_1_bst_SO) .
5. The beginning of the BSR is the first scan flip-flop in the core, which is connected to the TDI input as .a_r_ff_b0_DA(ta_TDI_CIN) .
6. The end of the scan chain leaves the core as .a_r_2(uc1_a_r_2) and enters the controller as .bst_control_scan_SO(uc1_a_r_2) .
7. The scan-enable signal .bst_control_C_9(test_logic_bst_control_C_9) is generated by the boundary-scan controller, and connects to the core as .taDriver12_I(test_logic_bst_control_C_9) .

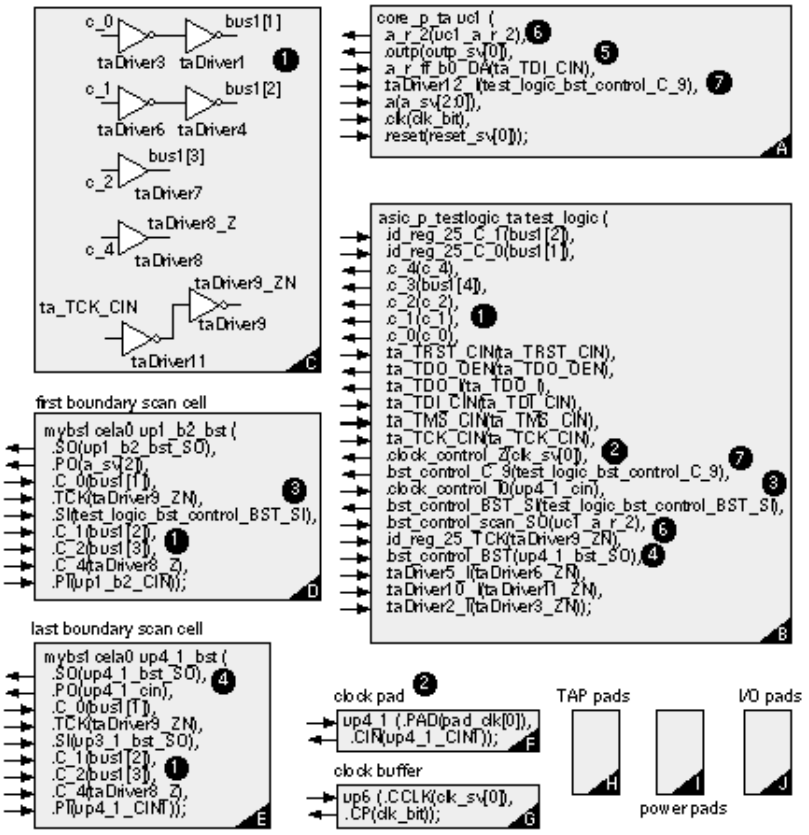


FIGURE 14.34 The top level of the Threegates ASIC after test-logic insertion.

The added test logic is shown in Figure 14.35 . The blocks are as follows:

1. This is the module declaration for the test logic in the rest of the diagram, it corresponds to block B in Table 14.34 .
2. This block contains buffers and clock control logic.
3. This is the boundary-scan controller.
4. This is the first of 26 IDR cells. In this implementation the IDCODE register is combined with the BSR. Since there are only six BSR cells we need (32 - 6) or 26 IDR cells to complete the 32-bit IDR.
5. This is the last IDR cell.

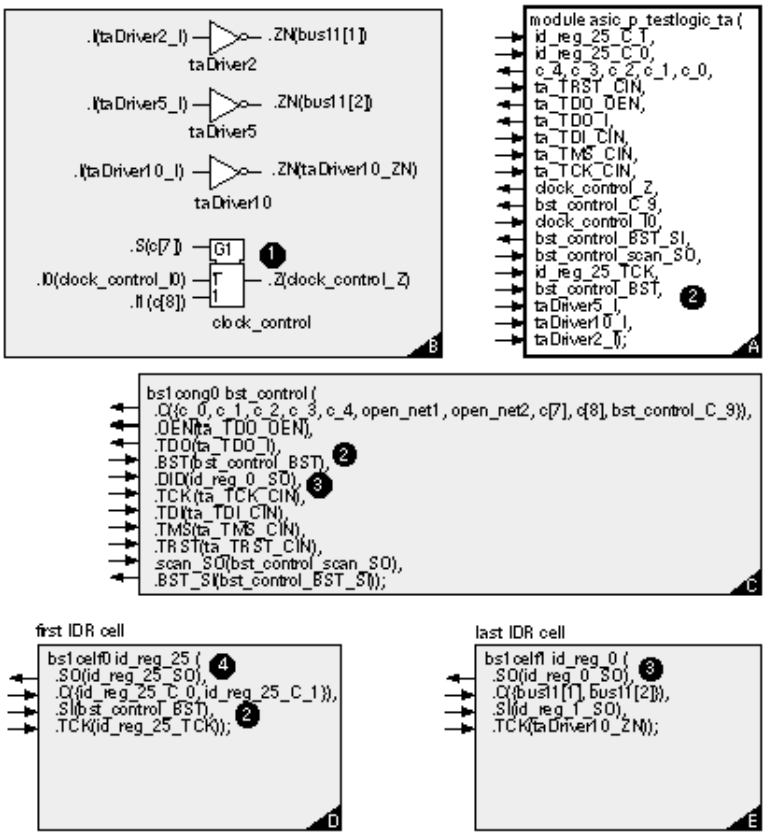


FIGURE 14.35 Test logic inserted in the Threegate ASIC.

TABLE 14.16 The TAP (test-access port) control. 1

TAP state	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8 2
Reset	x	x	xxxx0xx	xxxx0xx	xxxx0xx	xxxx0xx	xxxx1xx	xxxx0xx	xxxx0x
Run_Idle	00x0xxx	11x1xxx	0	1001011	0001011	0000010	1	0000001	000000
Select_DR	00x0xxx	11x1xxx	0	1001011	0001011	0000010	1	0000001	000000
Capture_DR	00x01xx	00x00xx	0	1001011	0001011	0000010	1	0000001	000000
Shift_DR	11x11xx	11x11xx	0	1001011	0001011	0000010	1111101	0000001	000000
Exit1_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	000000
Pause_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	000000
Exit2_DR	00x00xx	11x11xx	0	1001011	0001011	0000010	1	0000001	000000
Update_DR	00x0xxx	11x1xxx	110100	1001011	0001011	0	1111101	0000001	000000
Select_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	000000
Capture_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	000000
Shift_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	000000
Exit1_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	000000
Pause_IR	x	x	0	1001011	0001011	00000x0	11111x1	0000001	000000

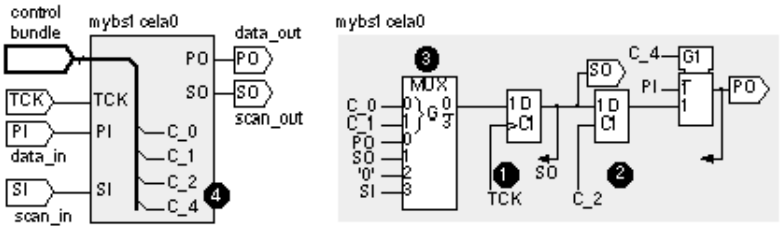
Exit2_IR	x	x	0	1001011 0001011 00000x0 11111x1 0000001 0000000
Update_IR	x	x	0	1001011 0001011 00000x0 1111101 0000001 0000000

The numbers in Figure 14.35 refer to the following explanations:

1. The system clock (CLK, not the test clock TCK) from the top level (after passing through the boundary-scan cell) is fed through a MUX so that CLK may be controlled during scan.
2. The signal bst_control_BST is the end (output) of the boundary-scan cells and the start (input) to the ID register only cells.
3. The signal id_reg_0_SO is the end (output) of the ID register.
4. The signal bst_control_BST_SI is the start of the boundary-scan chain.

The job of the boundary-scan controller is to produce the control signals (C_1 through C_9) for each of the 16 TAP controller states (reset through update_IR) for each different instruction. In this BST implementation there are seven instructions: the required EXTEST , SAMPLE , and BYPASS ; IDCODE ; INTEST (which is the equivalent of EXTEST , but for internal test); RUNBIST (which allows on-chip test structures to operate); and SCANM (which controls the internal-scan chains). The boundary-scan controller outputs are shown in Table 14.16 .

There are two very important differences between this controller and the one described in Table 14.5 . The first, and most obvious, is that the control signals now depend on the instruction. This is primarily because INTEST requires the control signal at the output of the BSCs to be in different states for the input and output cells. The second difference is that the logic for the boundary-scan cell control signals is now purely combinational-we have removed the gated clocks. For example, Figure 14.36 shows the input boundary-scan cell. The clock for the shift flip-flop is now TCK and not a gated clock as it was in Table 14.5 . We can do this because the output of the flip-flop, SO , the scan output, is added as input to the MUX that feeds the flip-flop data input. Thus, when we wish to hold the state of the flip-flop, the control signals select SO to be connected from the output to the input. This is called a polarity-hold flip-flop . Unfortunately, we have little choice but to gate the system clock if we make the scan chain part of the BSR. We cannot have one clock for part of the BSR and another for the rest. The costly alternative is to change every scan flip-flop to a scanned polarity-hold flip-flop.




```
module mybs1cela0 (SO, PO, C_0, TCK, SI, C_1, C_2, C_4, PI);

output SO, PO; input C_0, C_1, C_2, C_4, TCK, SI, PI;

in01d1 inv_0 (.I(C_0), .ZN(iv0_ZN));
```

```
in01d1 inv_1 (.I(C_1), .ZN(iv1_ZN));

oa03d1 oai221_1 (.A1(C_0), .A2(SO), .B1(iv0_ZN), .B2(SI), .C(C_1), .ZN(oa1_ZN));

nd02d1 nand2_1 (.A1(na2_ZN), .A2(oa1_ZN), .ZN(na1_ZN));

nd03d1 nand3_1 (.A1(PO), .A2(iv0_ZN), .A3(iv1_ZN), .ZN(na2_ZN));

mx21d1 mux21_1 (.I0(PI), .I1(upo), .S(C_4), .Z(PO));

dfntnb dff_1 (.D(na1_ZN), .CP(TCK), .Q(SO), .QN(\so.QN ));

lantnb latch_1 (.E(C_2), .D(SO), .Q(upo), .QN(\upo.QN ));

endmodule
```

FIGURE 14.36 Input boundary-scan cell (BSC) for the Threegates ASIC. Compare this to the generic data-register (DR) cell (used as a BSC) shown in Figure 14.2 .

14.8.3 ATVG and Fault Simulation

Table 14.17 shows the results of running the Compass ATVG software on the Threegates ASIC. We might ask: Why so many faults? and why is the fault coverage so poor? First we look at the details of the test software output. We notice the following:

- Line 2 . The backtrace limit is 30. We do not have any deep complex combinational logic so that this should not cause a problem.
- Lines 4 - 6 . An uncollapsed fault count of 184 indicates the test software has inserted faults on approximately 100 nodes, or at most 50 gates assuming a fanout of 1, less gates with any realistic fanout. Clearly this is less than all of the test logic that we have inserted.

TABLE 14.17 ATVG (automatic test-vector generation) report for the Threegates ASIC.

```
CREATE: Output vector database cell defaulted to [svf]asic_p_ta

CREATE: Backtrack limit defaulted to 30

CREATE: Minimal compression effort: 10 (default)

Fault list generation/collapsing

Total number of faults: 184

Number of faults in collapsed fault list: 80

Vector generation
```

#

VECTORS FAULTS FAULT COVER

processed

#

5 184 60.54%

#

Total number of backtracks: 0

Highest backtrack : 0

Total number of vectors : 5

#

STAR RESULTS summary

Noncollapsed Collapsed

Fault counts:

Aborted 0 0

Detected 89 43

Untested 58 20

------ ------

Total of detectable 147 63

#

Redundant 6 2

Tied 31 15

#

FAULT COVERAGE 60.54 % 68.25 %

#

Fault coverage = nb of detected faults / nb of detectable faults

Vector/fault list database [svf]asic_p_ta created.

To discover why the fault coverage is 68.25 percent we must examine each of the fault categories. First, Table 14.18 shows the undetected faults.

TABLE 14.18 Untested faults (not observable) for the Threegates ASIC.

Faults	Explanation
TADRIVER4.ZN sa0	Internal driver for BST control bundle (seven more faults like this).
TA_TRST.1.CIN sa0	BST reset TRST is active-low and tied high during test.
TDI.O sa0 sa1	TDI is BST serial input.
UP1_B0.1.CIN sa0 sa1	Data input pad (two more faults like this one).
UP3_1.1.CIN sa0	System reset is active-low and tied high during test.
UP4_1.1.CIN sa0 sa1	System clock input pad.

Total number: 20

The ATVG program is generating tests for the core using internal scan. We cannot test the BST logic itself, for example. During the production test we shall test the BST logic first, separately from the core-this is often called a flush test . Thus we can ignore any faults from the BST logic for the purposes of internal-scan testing.

Next we find two redundant faults: TA_TDO.1.I sa0 and sa1 . Since TDO is three-stated during the test, it makes no difference to the function of the logic if this node is tied high or low-hence these faults are redundant. Again we should ensure these faults will be caught during the flush test. Finally, Table 14.19 shows the tied faults.

TABLE 14.19 Tied faults.

Fault(s)	Explanation
TADRIVER1.ZN sa0	Internal BST buffer (seven more faults like this one).
TA_TMS.1.CIN sa0	TMS input tied low.
TA_TRST.1.CIN sa1	TRST input tied high.
TEST_LOGIC.BST_CONTROL.U1.ZN sa1	Internal BST logic.
UP1_B0_BST.U1.A2 sa0	

UP3_1.1.CIN sa1	Input pad (two more faults like this). Reset input pad tied high.

Total number: 15

Now that we can explain all of the undetectable faults, we examine the detected faults. Table 14.20 shows only the detected faults in the core logic. Faults F1-F8 in the first part of Table 14.20 correspond to the faults in Figure 14.16 . The fault list in the second part of Table 14.20 shows each fault in the core and whether it was detected (D) or collapsed and detected as an equivalent fault (CD). There are no undetected faults (U) in the logic core.

TABLE 14.20 Detected core-logic faults in the Threegates ASIC.

Fault(s)	Explanation
UC1.U2.ZN sa1	F1
UC1.U3.A2 sa1	F2
UC1.U3.ZN sa1	F5
UC1.U4.A1 sa1	F3
UC1.U4.ZN sa1	F6
UC1.U5.ZN sa0	F8
UC1.U5.ZN sa1	F7
UC1.A_R_FF_B2.Q.O sa1	F4

Fault list

UC1.A_R_FF_B0.Q: (O) CD CD	SA0 and SA1 collapsed to U3.A1
UC1.A_R_FF_B1.Q: (O) D D	SA0 and SA1 detected.
UC1.A_R_FF_B2.Q: (O) CD D	SA0 collapsed to U2. SA1 is F4.
UC1.U2: (I) CD CD (ZN) CD D	I.SA1/0 collapsed to O.SA1/0. O. SA1 is F1.
UC1.U3: (A1) CD CD (A2) CD D (ZN) CD D	A1.SA1 collapsed to U2.ZN.SA1.

UC1.U4: (A1) CD D (A2) CD CD (ZN) CD D A2.SA1 collapsed to A_R_FF_B2.Q.SA1.

UC1.U5: (A1) CD CD (A2) CD CD (ZN) D D A1.SA1 collapsed to U3.ZN.SA1

14.8.4 Test Vectors

Next we generate the test vectors for the Threegates ASIC. There are three types of vectors in scan testing. Serial vectors are the bit patterns that we shift into the scan chain. We have three flip-flops in the scan chain plus six boundary-scan cells, so each serial vector is 9 bits long. There are serial input vectors that we apply as a stimulus and serial output vectors that we expect as a response. Parallel vectors are applied to the pads before we shift the serial vectors into the scan chain. We have nine input pads (three core data, one core clock, one core reset, and four input TAP pads- TMS , TCK , TRST , and TDI) and two outputs (one core data output and TDO). Each parallel vector is thus 11 bits long and contains 9 bits of stimulus and 2 bits of response. A test program consists of applying the stimulus bits from one parallel vector to the nine input pins for one test cycle. In the next nine test cycles we shift a 9-bit stimulus from a serial vector into the scan chain (and receive a 9-bit response, the result of the previous tests, from the scan chain). We can generate the serial and parallel vectors separately, or we can merge the vectors to give a set of broadside vectors . Each broadside vector corresponds to one test cycle and can be used for simulation. Some testers require broadside vectors; others can generate them from the serial and parallel vectors.

TABLE 14.21 Serial test vectors

Serial-input scan data

#1 1 1 1 0 1 0 1 1 0

#2 1 0 1 1 0 1 0 0 1

#3 1 1 0 1 1 1 0 1 0

#4 0 0 0 1 0 0 0 0 0

#5 0 1 0 0 1 1 1 0 1

^UC1.A_R_FF_B0.Q ^UP1_B2_BST.SO.Q ^UP2_1_BST.SO.Q

^UC1.A_R_FF_B1.Q ^UP1_B1_BST.SO.Q ^UP3_1_BST.SO.Q

^UC1.A_R_FF_B2.Q ^UP1_B0_BST.SO.Q ^UP4_1_BST.SO.Q

Fault	Fault number	Vector number	Core input
UC1.U2.ZN sa1	F1	3	011
UC1.U3.A2 sa1	F2	4	000
UC1.U3.ZN sa1	F5	5	010
UC1.U4.A1 sa1	F3	2	101

UC1.U4.ZN sa1	F6	1	111
UC1.U5.ZN sa0	F8	1	111
UC1.U5.ZN sa1	F7	2	101
UC1.A_R_FF_B2.Q.O sa1	F4	2	101

Table 14.21 shows the serial test vectors for the Threegates ASIC. The third serial test vector is ’110111010’ . This test vector is shifted into the BSR, so that the first three bits in this vector end up in the first three bits of the BSR. The first three bits of the BSR, nearest TDI , are the scan flip-flops, the other six bits are boundary-scan cells). Since UC1.A_R_FF_B0.Q is a_r[0] and so on, the third test vector will set a_r = 011 where a_r[2] = 0. This is the vector we require to test the function a_r[0]’.a_r[1] + a_r[1].a_r[2] for fault UC1.U2.ZN sa1 in the Threegates ASIC. From Figure 14.31 we see that this is a stuck-at-1 fault on the output of the inverter whose input is connected to a_r[0] . This fault corresponds to fault F1 in the circuit of Figure 14.16 . The fault simulation we performed earlier told us the vector ABC = 011 is a test for fault F1 for the function A’B + BC.

14.8.5 Production Tester Vector Formats

The final step in test-program generation is to format the test vectors for the production tester. As an example the following shows the Sentry tester file format for testing a D flip-flop. For an average ASIC there would be thousands of vectors in this file.

```
# Pin declaration: pin names are separated by semi-colons (all pins

# on a bus must be listed and separated by commas)

pre_; clr_; d; clk; q; q_;

# Pin declarations are separated from test vectors by $

$

# The first number on each line is the time since start in ns,

# followed by space or a tab.

# The symbols following the time are the test vectors

# (in the same order as the pin declaration)

# an "=" means don’t do anything

# an "s" means sense the pin at the beginning of this time point

# (before the input changes at this time point have any effect)
```

#

pcdcqq

rlal _

ertk

__a

00 1010== # clear the flip-flop

10 1110ss # d=1, clock=0

20 1111ss # d=1, clock=1

30 1110ss # d=1, clock=0

40 1100ss # d=0, clock=0

50 1101ss # d=0, clock=1

60 1100ss # d=0, clock=0

70 ====ss

14.8.6 Test Flow

Normally we leave test-vector generation and the production-test program generation until the very last step in ASIC design after physical design is complete. All of the steps have been described before the discussion of physical design, because it is still important to consider test very early in the design flow. Next, as an example of considering test as part of logical design, we shall return to our Viterbi decoder example.

TABLE 14.22 Timing effects of test-logic insertion for the Viterbi decoder.

Timing of critical paths before test-logic insertion

Slack(ns) Num Paths

-3.3826 1 *

-1.7536 18 *******

-.1245 4 **

1.5045 1 *

3.1336 0 *

4.7626 0 *

6.3916 134 **

8.0207 6 ***

9.6497 3 **

11.2787 0 *

12.9078 24 ********
instance name

inPin --> outPin incr arrival trs rampDel cap cell

(ns) (ns) (ns) (pf)

v_1.u100.u1.subout6.Q_ff_b0

CP --> QN 1.73 1.73 R .20 .10 dfctnb

...

v_1.u100.u2.metric0.Q_ff_b4

setup: D --> CP .16 21.75 F .00 .00 dfctnh

After test-logic insertion
-4.0034 1 *

-1.9835 18 *****

.0365 4 **

2.0565 1 *

4.0764 0 *

6.0964 138 ******************************

```
# 8.1164 2 *

# 10.1363 3 **

# 12.1563 24 ******

# 14.1763 0 *

# 16.1963 187 ****************************************
# v_1.u100.u1.subout7.Q_ff_b1

# CP --> Q 1.40 1.40 R .28 .13 mfctnb

...

# v_1.u100.u2.metric0.Q_ff_b4

# setup: DB --> CP .39 21.98 F .00 .00 mfctnh
```

---

1. Outputs are specified for each instruction as 0123456, where: 0 = EXTEST, 1 = SAMPLE, 2 = BYPASS, 3 = INTEST, 4 = IDCODE, 5 = RUNBIST, 6 = SCANM.

2. T denotes gated clock TCK.

14.9 The Viterbi Decoder Example

Table 14.22 shows the timing analysis for the Viterbi decoder before and after test insertion. The Compass test software inserts internal scan and boundary scan exactly as in the Threegates example. The timing analysis is in the form of histograms showing the distributions of the timing delays for all paths. In this analysis we set an aggressive constraint of 20 ns (50 MHz) for the clock. The critical path before test insertion is 21.75 ns (the slack is thus negative at -1.75 ns). The path starts at u1.subout6.Q_ff_b0 and ends at u2.metric0.Q_ff_b4 , both flip-flops inside the flattened block, v_1.u100 , that we created during synthesis in an attempt to improve speed. The first flip-flop in the path is a dfctnb ; the last flip-flop is a dfctnh . The suffix ’b’ denotes 1X drive and suffix ’h’ denotes 2X drive.

After test insertion the critical path is 21.98 ns. The end point is identical, but the start point is now subout7.Q_ff_b1 . This is not too surprising. What is happening is that there are a set of paths of nearly equal length. Changing the flip-flops to their scan versions (mfctnb and mfctnh) increases the delay slightly. The exact delay depends on the capacitive load at the output, the path (clock-to-Q, clock-to-QN, or setup), and the input signal rise time.

Adding test logic has not increased the critical path delay substantially. Almost as important is that the distribution of delays has not changed substantially. Also very important is the fact that the distributions show that there are only approximately 20 paths with delays close to the critical path delay. This means that we should be able to constrain these paths during physical design and achieve a performance after routing that is close to our preroute predictions.

TABLE 14.23 Fault coverage for the Viterbi decoder.

Fault list generation/collapsing

Total number of faults: 8846

Number of faults in collapsed fault list: 3869

Vector generation

#

VECTORS FAULTS FAULT COVER

processed

#

20 7515 82.92%

40 8087 89.39%

60 8313 91.74%

80 8632 95.29%

87 8846 96.06%

Total number of backtracks: 3000

Highest backtrack : 30

Total number of vectors : 87

STAR RESULTS summary

Noncollapsed Collapsed

Fault counts:

Aborted 178 85

```
# Detected 8427 3680

# Untested 168 60

# ------ ------

# Total of detectable 8773 3825

#

# Redundant 10 6

# Tied 63 38

#

# FAULT COVERAGE 96.06 % 96.21 %
```

Next we check the logic for fault coverage. Table 14.23 shows that the ATPG software has inserted nearly 9000 faults, which is reasonable for the size of our design. Fault coverage is 96 percent. Most of the untested and tied faults arise from the BST logic exactly as we have already described in the Threegates example. If we had not completed this small test case first, we might not have noticed this. The aborted faults are almost all within the large flattened block, v_1.u100